Examensarbete

HÖGSKOLAN
I SKÖVDE

JÄMFÖRELSE I PRESTANDA MELLAN MYSQL OCH MONGODB FÖR HANTERING AV DYKDATA

En jämförelse av prestanda vid interaktion av data mellan MySQL och MongoDB

PERFORMANCE COMPARISON BETWEEN MYSQL AND MONGODB FOR MANAGING OF DIVING DATA

A performance comparison for interacting with data using MySQL and MongoDB

Examensarbete inom huvudområdet Informationsteknologi
Grundnivå 30 högskolepoäng
Vårtermin 2017

Emil Andersson

Handledare: Mikael Berndtsson
Examinator: Henrik Gustavsson

Sammanfattning

Varje gång det sker ett dyk ska det skrivas en dyklogg som beskriver vad som gjorts och vad som setts. Dock brukar dessa loggar lagras fysiskt vilket leder till att datan försvinner med tiden. Man bör därför lagra loggarna i en databas för att bevara datan och förenkla interaktion. Ett problem med att lagra data i en databas är att datamängderna kan öka till stora mängder och databaserna då kan bli långsamma och nästintill oanvändbara. I studien utförs ett experiment som undersöker vilken databas som bör användas mellan MySQL och MongoDB och ger svar på hypotesen *”Genom att använda sig av MongoDB kan man få bättre svarstider än med MySQL, särskilt vid användning av större datamängder”.* Detta undersöks genom att jämföra svarstiden mellan respektive databas och resultatet visar att MongoDB presterar bättre än MySQL vid hantering av dykloggar. I framtiden vore det intressant att använda fler databaser samt andra tekniker för att hantera datan.

Nyckelord: Dykloggar, MongoDB, MySQL, Prestandajämförelse

Innehållsförteckning

1 Introduktion

Varje gång ett dyk sker skapas ny dykdata som på något sätt ska sparas. Denna data sparas främst i fysiska loggböcker men efter ett antal dyk blir dessa loggböcker fulla och man måste skaffa en ny bok om och om igen. Detta leder till mängder av loggböcker som man måste förvara någonstans vilket i sin tur leder till att böckerna ofta försvinner eller förstörs. När en loggbok försvinner, försvinner även den data som existerar i boken. Denna data har inte bara ett personligt värde hos dess ägare utan är även data som kan vara användbar vid forskning inom det marina området. Genom att använda dykdata kan man bland annat få fram information om ändringar inom marinlivet då man med datan kan se vart specifika arter befinner sig och när. Om man har data över flera år kan man då se hur arterna har förflyttat sig samt även göra analyser om hur populationen har ökat eller minskat.

Det vore därför optimalt att kunna lagra denna information på ett säkert sätt över flera år vilket kan ske via databaser. En applikation där användare kan lagra sina dykloggar digitalt istället för fysiskt skulle inte bara göra det möjligt att lagra datan under längre perioder utan det skulle även samla datan på ett effektivt sätt samt göra det möjligt för användare att enkelt leta igenom sina gamla dyk.

För att en sådan applikation ska fungera måste man lagra datan vilket sköts bäst med databaser, i regel används antingen en relationsdatabas (SQL) eller en icke-relationsdatabas (NoSQL). Relationsdatabaser har använts som standard under många år men ett problem med relationsdatabaser är att de inte är gjorde för att hantera större datamängder och får då problem med prestandan. Man har därför börjat använda sig mer utav icke-relationsdatabaser då dessa är bättre anpassade för att hantera stora datamängder (Mahmood, K., Risch, T. and Zhu, M., 2015). Ett exempel på en icke-relationsdatabas är MongoDB och lyfts fram av Mahmood m.fl. som ett bra alternativ.

Studien använder sig utav experiment för att jämföra de bägge databaserna. Experiment är bra vid prestandamätningar då de sker i en kontrollerad och sluten miljö samt är repeterbara för framtida studier (Wohlin et al. 2012). Experiment används i ett flertal studier, tillexempel använder Mahmod et al. (2015) experiment för att mäta hur en NoSQL-databas kan användas för skalbar logganalys och Abramova & Bernardino (2013) samt Gyrödi et al. (2015) använder experiment för att jämföra två databaser.

Det finns även ett kapitel som går igenom hur de olika databaserna och teknikerna som används har implementerats samt arbetets progression. Där finns även en pilotstudie som visar hur mätningarna kommer gå till i studien samt ger en första bild på vad man kan förvänta sig vid mätningarna.

2 Bakgrund

2.1 Lagring av data

För att hantera måste den lagras i någon form, en metod för att lagra data är att använda sig av en databas, detta gör det även möjligt att hämta och presentera datan på ett effektivt sätt. En databas fungerar genom att strukturera och organisera datan för att sedan använda referenser för att bland annat lagra, hämta, uppdatera eller radera data. Generellt finns det två typer av databaser, relationsdatabaser och icke-relationsdatabaser. Dessa brukar refereras som SQL-databaser respektive NoSQL-databaser. Relationsdatabaser må vara effektiva men det blir allt vanligare att använda sig av icke-relationsdatabaser, särskilt när man börjar arbeta med större mängder data (Mahmood m.fl. 2015).

MySQL är en traditionell relationsdatabas som använder sig av tabeller med nycklar och värden. Dessa nycklar används sedan för att referera till relaterad data.

```
CREATE TABLE Divers (
    ID INT NOT NULL PRIMARY KEY,
    diverID INT NOT NULL,
    firstName VARCHAR(20),
    lastName VARCHAR(20),
    noDives INT
)

INSERT INTO Divers(ID, diverID, firstName, lastName, noDives)
VALUES('18129s4a7', '13', 'Erik', 'Eriksson', '5');
```

Figur 1 Exempel på hur man lagrar data i MySQL

För att få en så optimal databas som möjligt krävs det att man följer vissa principer. Dessa principer innehåller specifika krav som man strävar att uppnå, en av dessa är CAP vilket enligt Brewer (2012) innebär:

- **C**onstistency; alla noder har samma data vid samma tillfälle.
- Availability; alla anrop ger en respons.
- Partition tolerance; även om en del av systemet faller kommer hela systemet hållas samman.

Dock säger Brewer (2012) att det är omöjligt att nå alla mål samtidigt utan man får istället prioritera de olika målen och oftast är det endast ett mål som uppnås till fullo, ett som uppnås delvis och det tredje uppnås inte alls.

Utöver CAP finns det två ytterliga principer som man bör känna till, ACID och BASE. ACID har tagits fram för att ytterligare optimera prestanda och är baserat på CAP, när de talar om relationsdatabaser nämner Abramova & Bernadino (2013) ACID och de krav som det innebär:

- **A**tomic; en transaktion är endast komplett när alla operation är färdiga, om inte alla operationer är färdiga sker en rollback.
- Consistent; en transaktion kan inte kollapsa en databas, detta sker ses operationen som illegal och en rollback sker.
- Isolated; en transaktion är självständig och kan inte påverka andra transaktioner.
- Durable; när en commit har skett kan transaktioner inte ångras. En commit är en operation som konfirmerar alla ändringar och gör dem permanenta.

Om man följer ACID bör man uppnå en robust och korrekt databas, dock är det svårt att uppnå dessa principer när man arbetar med stora mängder data (Abramova & Bernadino, 2013). Man brukar då istället använda sig av BASE som innebär:

- **B**asically **A**vailable; all data är distribuerad och tillgänglig, detta innebär att även när det finns ett problem i systemet kommer systemet fortsätta fungera.
- Soft state; det finns inga garantier gällande konsistens.
- Eventually consistent; systemet garanterar att även om datan inte är konsistent så kommer den bli det i slutändan.

Noterbart är att BASE fortfarande följer CAP och kan ses som en utökning av principer. Vilken princip man ska använda sig av beror på vilken typ av databas man ska använda, generellt brukar relationsdatabaser följa ACID medans NoSQL databaser följer BASE.

2.2 Icke-relationsdatabaser, NoSQL

Huvudanledningen till att NoSQL databaser togs fram var att mängden data i databaser ökade, särskilt till sidor så som sociala nätverk och Google (Abramova & Bernadino 2013). NoSQL databaser gör det enkelt att använda sig av horisontell scaling vilket innebär att man ökar prestanda genom att öka mängden lagringsenheter. NoSQL databaser är gjorda för att kunna hantera alla sorters fel då det är smartare att acceptera att fel kommer att hända istället för att se fel som speciella undantag (Abramova & Bernardino (2013). I grunden är det inga större skillnader mellan SQL och NoSQL databaser då de grundläggande operationerna fortfarande är desamma. De grundläggande operationerna hos databaser är; lägg till, ta bort, uppdatera och hämta data och kan utföras på alla databaser (Györödi, C., Györödi, R., Pecherle, G. Och Olah, A. 2015). Däremot är det skillnad när det kommer till interaktion med databaserna då SQL databaser använder sig av ett standardiserat SQL språk medan NoSQL databaser generellt använder sig utav UNIX kommandon för interaktion.

Det finns en mängd olika sorters NoSQL databaser som alla är baserad på samma principer men är unika på sitt egna sätt. Abromova och Bernardino (2013) säger att man kan dela upp NoSQL databaser i fyra kategorier;

- **Key-Value Store**; All data lagras i set en nyckel och värden, en nyckel är unik och tillgång till datan sker genom att relatera nycklarna till värden.
- **Column-family;** Data struktureras i kolumner, dessa kolumner kommer i tre olika former; *Column*, där data identifieras av nyckel och värde. *Super-Column,* en större kolumn som innehåller kolumner och *Column family,* en grupp strukturerad data som

liknar tabeller från relationsdatabaser, datan består av en mängd super-kolumner. Column-Family är den typ av NoSQL databas som mest efterliknar en traditionell relationsdatabas.

- **Graph database;** Databaser byggda för att lagra data som kan presenteras i grafer, ett exempel är sociala nätverk.
- **Document Store;** Dessa databaser kan ses som set av *Key-Value Stores* som sedan förvandlas till dokument. Varje dokument är identifierat av en unik nyckel och dokumenten kan grupperas tillsammans och datatillgång sker via nycklar eller specifika värden. Dokumenttyperna brukar existera i standardiserade former så som XML och JSON.

Under gruppen **Document Store** kan man hitta MongoDB.

2.3 MongoDB

MongoDB är som sagt en Document Store databas vilket innebär att datan sparas i dokument, i MongoDBs fall sparas datan i BSON-format, vilket är en binär form av JSON och fungerar på liknande sätt med som JSON då det bland annat går att använda arrayer i dokumenten men finns även utökade egenskaper så som ytterligare typer av numerisk data och stöd för att hantera binär data (Hows et al. 2013). Att datan sparas i BSON-format kommer även underlätta under arbetet då det kommer användas javascript för visualisering och det underlättar att datan är i ett strukturerat format som är enkelt att arbeta med. MongoDB har även stöd för sekundärindexering vilket kan komma till nytta.

Mahmood et al. (2015) rekommenderar användandet av MongoDB vid analys av stora mängder loggdata och anser att det är ett användbart alternativ till relationsdatabaser. Lutz et al. (2014) har använt MongoDB för att visualisera stora mängder data och styrker MongoDB som en bra databas både är agil och samtidigt lika pålitlig som en traditionell databas.

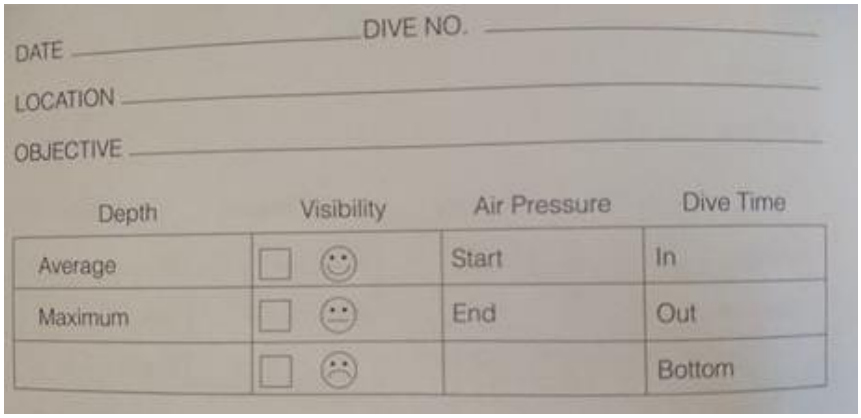
```
db.diverCollection.insert( {
                    _id: "18129s3a7",
                    DiverID: "13"    ,
                    name: {First:"Erik", Last:"Eriksson"},
                    noDives: "5"
                    } )
```

Figur 2 Exempel på insättning av data i MongoDB

2.4 Dataloggar

Exakt hur dykloggar struktureras kan variera mellan olika dykcentra och forum men vissa punkter finns alltid med i loggarna. Dessa punkter är: Datum, Plats, Djup, Lufttryck och Tid.

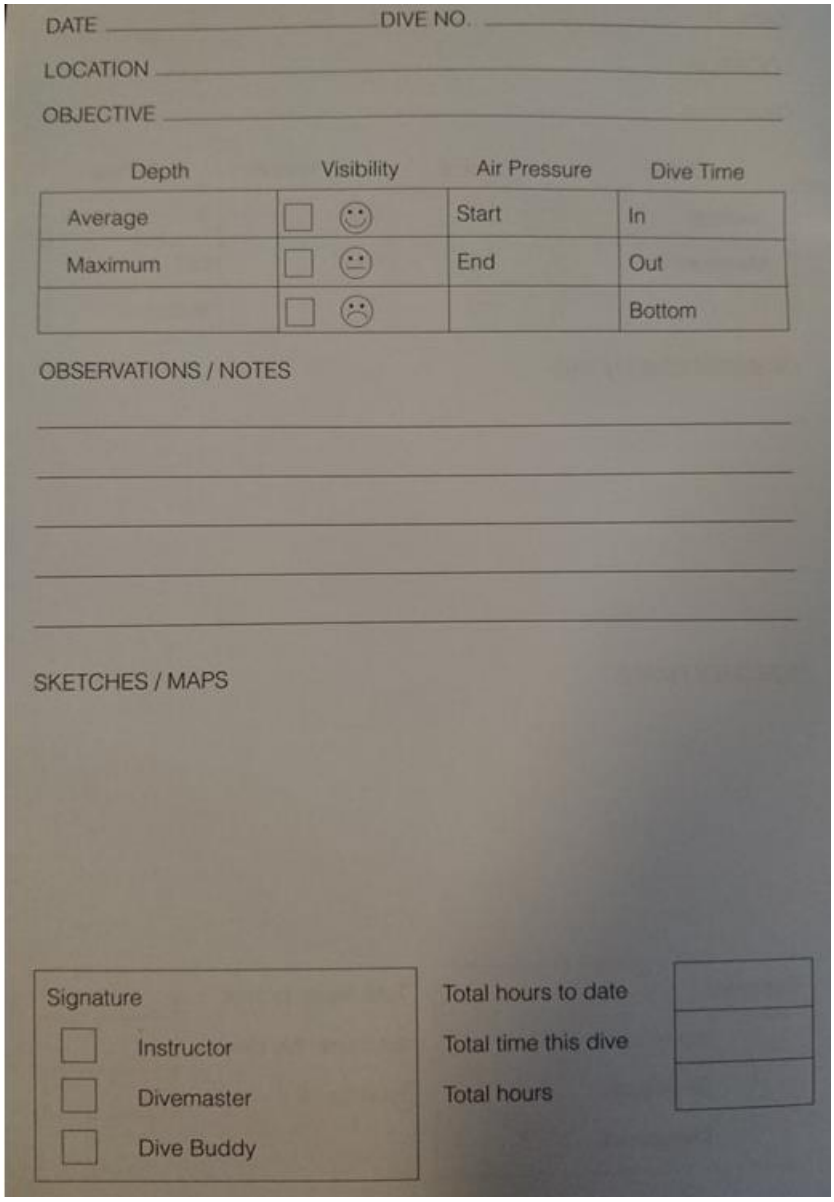
Strukturen i arbetet kommer baseras på en personlig loggbok köpt hos Lanta Diver i Koh Lanta, Thailand. Denna loggbok fungerar som en bra grund då Lanta Diver är certifierade av PADI som är den ledande organisationen vid utbildning av dykare.



Figur 3 Exempel på de viktiga delarna i en dyklogg

Figur 3 är en bild hämtad från en personlig loggbok, i figuren kan vi se att alla de viktiga punkterna är inkluderade. Datum och plats är vitala för arbetet då en plats och en tidpunkt är nödvändigt för att analysera hur datan ändras över tid och vart. Det finns även fält för vilket djup som dyket skedde på, i loggboken kan man lägga in både det genomsnittliga djupet samt den djupaste punkt man nådde. Då man i de flesta dyk tar sig till ett specifikt djup och sedan håller sig till det djupet under majoriteten av djupet är vi endast intresserade av dykets djupaste punkt. Lufttryck visar hur mycket syrgas man har inandats under dykets gång. Eftersom det finns olika storlek av tuber och de inte alltid är fyllda till max fyller man i ett start och ett slutvärde för att alltid få ett så korrekt värde som möjligt. Lufttryck är inte nödvändigt vid analys av t.ex migration men är ett intressant fält för dykare då man kan se ändringar i sin konsumtion över åren. Man skriver alltid ner hur mycket syrgas man använt efter ett dyk för att hålla koll på sina värden och undvika dykarsjuka. Detta är även fallet för dyktid, det finns inte mycket användning för det vid analyser men det är ett viktigt fält för dykare.

I figur 3 finns det även fält för dyknummer och sikt. Dyknummer används för att räkna hur många dyk man har gjort, detta nummer lämnas ofta tomt av erfarna dykare då de inte längre kan hålla koll på alla sina dyk mellan alla loggböcker. Sikt berättar hur väl man kunde se under dyket, detta fält kommer inte inkluderas då det inte är intressant för arbetet.



Figur 4 Exempel på en komplett dyklogg utan data

I figur 4 kan man utöver de viktiga delarna av en dyklogg även se; Observationer, där man fyller i de olika arter man har sett, Sketcher/Kartor, som används främst för att visualisera observationer, Signatur, där någon man dykt med skriver en signatur och även sammanlagd dyktid. Av dessa fält kommer endast observationer och sammanlagd dyktid finnas med i arbetet. Signatur kommer inte inkluderas då detta används främst vid utbildning av dykare där instruktören skriver under. Sketcher/Kartor kommer inte finnas med då detta inte kommer vara användbart i arbetet.

Observationer kommer vara kärnan för arbetet då det är här man skriver ner vad man har sett under dyket vilket kommer vara det som används främst inom sökningar och därför är extremt viktigt för arbetet.

Sammanlagd dyktid kommer inte att inkluderas då detta endast är trevlig information för en användare och inte intressant för arbetet.

3 Problemformulering

3.1 Problemet

Dykdata är inte bara trevlig information att ha själv utan kan även användas vid forskning inom det marina området då datan innehåller bland annat vilka djur som skådats under ett dyk. Om man kan samla datan vore det därför möjligt att t.ex använda datan för att spåra hur djur har migrerat över åren och hur detta hör samman med ändringar i marinbiologi. Det vore därför optimalt att lagra datan i en databas då detta gör det möjligt att samla data på ett effektivare sätt där datan struktureras och även gör det möjlig att söka i datan för att hitta det man söker. Det existerar redan ett fåtal dykapplikationer som gör detta möjligt men då dykdata snabbt kan uppnå stora datamängder kan detta leda till problem i databasen. Det vore därför intressant att hitta den optimala databasen för en dykapplikation för att bevara datan på så bra sätt som möjligt.

Det första problem man stöter på är datastruktur. Struktur på datan kan skilja mellan applikationer och olika databaser följer olika struktur, för att hitta så optimal databas som möjligt är det först steget då att jämföra databaser med olik struktur. Ett annat problem är mängden data, en erfaren dykare kan ha ett par hundra dyk och datamängden i en dykapplikation kan stiga till stora mängder vilket generellt brukar skapa problem hos relationsdatabaser. Detta kan bero på att datan i en relationsdatabas måste vara konsistent och förlitar sig starkt på indexering, en möjlig lösning på detta är att använda en NoSQL databas istället för en relationsdatabas då NoSQL databaser har betydligt färre krav gällande konsistens (Mahmood et al., 2015). Dock är relationsdatabaser ofta bättre när det kommer till frågeställningar och om en NoSQL databas inte konfigureras och indexeras på korrekt sätt är det möjligt att relationsdatabaser har bättre svarstid trots att de inte är byggda för större datamängder men förutsatt att NoSQL databasen görs på korrekt sätt ska de vara bättre lämpade för hantering av stora datamängder (Catell 2010).

Vilken databas man använder kan även det göra skillnad. Mahmood et al. (2015) anser att MongoDB är ett bra alternativ av NoSQL databas, detta styrks sedan av Györödi et al. (2015) som har jämför MySQL med MongoDB och kom till slutsatsen att MongoDB presterar bättre än MySQL gällande att sätta in, ta bort, uppdatera eller hämta data. Dessa tester utfördes med datamängder på tiotusen rader och om man ökar mängden till tiotusentals eller kanske tillochmed närmre hundratusen rader bör detta ge ännu tydligare resultat då stora datamängden ska vara till NoSQL databasers fördel.

MongoDB användes även av Lutz et. Al (2014) vid visualisering med motivationen att MongoDB är en agil databas med horisontell skalning som snabbt anpassar scheman efter ändringar.

För att studien ska ge ett så korrekt resultat som möjligt måste datan som används vara så lik faktisk dykdata som möjligt. För att uppnå detta handskrivna loggar från en personlig loggbok användas som grund för att sedan generera data som baseras på den struktur som existerar i boken.

3.2 Hypotes

MongoDB har presterat bra i de tester som utförts av Mahmood et al. (2015) och Györödi et al. (2015) och ger bra svarstider när det kommer till att hämta data (Lutz et al. 2014). Denna forskning tyder på att MongoDB bör ge snabbare svarstider för hantering av dykloggar. Dock får man inte glömma att MySQL har inbyggd indexering som kan fungera bättre en den indexering som finns i MongoDB. MySQL har även bättre stöd gällande avancerade operationer vilket kan förbättra svarstider då färre operationer krävs.

Den hypotes som ställs är: *”Genom att använda sig av MongoDB kan man få bättre svarstider än med MySQL, särskilt vid användning av större datamängder”.*

3.3 Metodbeskrivning

Arbetet kommer använda sig av teknikorienterade experiment då dessa fokuserar på den tekniska delen istället för den mänskliga faktorn (Wohlin et al. 2012). Anledningen att jag använder mig av teknikorienterade experiment över människoorienterade experiment är att jag kommer använda mig av tester för att se vilken typ av databas som ger bäst resultat och kommer därför inte använda mig av människors åsikter. Jag använder mig av experiment då dessa sker i en kontrollerad miljö vilket är bra vid testning då man vill att dessa ska ske så kontrollerat som möjligt (Wohlin et al. 2012).

Experiment är en av de vanligaste metoderna och har använts av bland annat Györödi et al. (2015) och Abramova & Bernardino (2013) som använder experiment för att jämföra databaser och även av Mahmood et al. (2015) och Swaminathan et al. (2016) som använder experiment föra att undersöka skalbarhet i NoSQL databaser. Utifrån detta kan man dra slutsatsen att experiment är effektivt när man ska utföra tester för att jämföra databaser.

För insamling av data kommer data från personliga loggböcker användas, dessa kommer att användas som referens för hur datan ska vara uppbyggd för att sedan skapa så korrekt data som möjligt för att öka datamängden.3.3.1 Alternativa metoder

Det finns en del forskningsmetoder utöver experiment. En alternativ forskningsmetod vore användarstudier där användare testar ens applikation och sedan ger input utifrån deras åsikter angående applikationen. Användarstudier är användbara då man utifrån dem får veta om man har tagit fram en produkt som folk vill använda samt vad man bör lägga till eller ändra i sin applikation för att öka intresset hos användare. En användarstudie kommer inte användas i detta arbete då svarstider ska jämföras så noggrant som möjligt och användare kan sällan ge detaljerade svar angående svarstider.

En annan alternativ metod vore att använda sig av fallstudier där man undersöker specifika delar av ett verkligt projekt under en viss tid (Wohlin et al. 2012). Problemet med fallstudier är att de endast är applicerbara för ett specifikt fall och det är svårt att repetera undersökningen. Däremot blir fallstudier mer realistiska än experiment då de är kopplade till ett verkligt projekt och inkluderar användare (Wohlin et al. 2012).

En nackdel med experiment är att man inte kan säga att resultaten gäller vid alla tillfällen utan endast vid detta specifika tillfälle (Berndtsson et al. 2008). Anledningen till detta är att det finns en mängd faktorer som kan påverka resultatet och även om exakt samma experiment

utförs vid ett senare tillfälle så kan detta ge olika resultat då saker så som näthastighet kan skilja. Man kan däremot stärka sin hypotes med sina resultat.

3.4 Forskningsetik

Det främsta etiska problemet med arbetet är datakällor då loggböcker är en form av personlig information. Används data från individer som inte är inkluderade i arbetet måste dessa informeras och ge tillåtelse att datan används. Detta arbete kommer därför främst använda genererad data som följer den struktur som påvisas av existerande självskrivna loggböcker. Ett problem med detta är att alla loggböcker inte följer exakt samma struktur vilket man inte får glömma under arbetet då resultaten hade kunnat påverkas om data av olika struktur hade använts.

För att göra experimentet repeterbart kommer all programkod publiceras. Data från självskrivna loggböcker kommer även dessa publiceras för att styrka datastrukturen, eventuell data från andra individer kommer inte att publiceras då denna ses som personlig information. All teknisk information kommer även detta publiceras, detta inkluderar saker så som hårdvara, näthastighet och mjukvara. Detta är viktigt för att göra arbetet repeterbart då dessa kan påverka testerna enormt. För att få så specifik information om hårdvaran som möjligt kommer testerna ske på en maskin där hårdvaran begränsas till specifika mått för att ge en tydlig bild på vad som använts. Inom mjukvara existerar bland annat webbläsare då testresultat kan variera beroende på vilken webbläsare som använts. Testerna bör därför utföras på olika webbläsare för att ge så utförliga resultat som möjligt. För optimala resultat bör man även utföra testerna med olika gränser på hårdvara och även på maskiner med olika operativsystem. Ett problem med detta är att maskiner med olika operativsystem även har olika typer av hårdvara och det kan vara svårt att utföra så rättvisa mätningar som möjligt.

4 Genomförande/Implementation/ Projektbeskrivning

Detta kapitel går igenom de viktiga komponenter som implementerats för att göra det möjligt att genomföra projektet för att sedan kunna svara på de frågeställningar som studien baserats på. För att kunna besvara hypotes och frågeställningar krävs två databaser varav en är en relationsdatabas, MySQL, och en NoSQL databas, MongoDB, för att lagra data. För att sedan interagera med dessa databaser krävs en applikation som ligger på en webbserver med stöd för PHP. Via applikationen kan man sedan välja databas och sedan hämta och presentera datan som söks.

4.1 Research / Förstudie

Då applikationen använder sig av flera olika programmeringsspråk och metoder har inspiration hämtats från flera olika källor beroende på vad som skulle utföras och vilken del applikationen som skulle utvecklas. Grundidén till arbete är baserat på en existerande applikation för loggböcker, Diviac, där man kan se diverse dyk samt lagra loggböcker. Funderingen kring den mest effektiva typen av databas ställdes efter att ha använt deras applikation.

Fundering kring NoSQL är baserad på boken Getting Started with NoSQL (Vaish, 2013) som går igenom vad som specificerar en NoSQL databas och hur de skiljer sig ifrån relationsdatabaser. Boken ger även en bra grund och fungerar som en bra introduktion till NoSQL då den förklarar saker tydligt och genomgående från en nybörjarpunkt. För installation av MongoDB används MongoDBs egna dokumentation (MongoDB docs, 2018) och ger även en bra introduktion till att förstå MongoDBs struktur samt hur man bygger upp en fungerande databas med kollektioner.

Boken The Definitive Guide to MongoDB (Hows et al., 2013) ger sedan stora mängder information om just MongoDB. Boken ger en bra grund för MongoDB men ger även mer avancerad information angående hur man hanterar och interagerar med data via MongoDB samt hur MongoDB och PHP fungerar tillsammans. Boken tar upp både grundläggande och mer avancerade frågeställningar och ger grundlig information om MongoDB. Tyvärr är boken något utdaterad då MongoDB nyligen genomgick stora förändringar och de metoder som används för att bland annat koppla MongoDB med PHP fungerar inte i dagsläget. En uppdaterad beskrivning för detta hittas på PHP.net (The PHP Group, 2018) som ger en tydlig introduktion till hur man gör det möjligt att arbeta med MongoDB via PHP samt hur man interagerar med datan. En viktig komponent för att skapa en koppling mellan MongoDB och PHP är verktyget Composer som finns hos getcomposer.org (Adermann et al., 2018). Composer ger tillgång till viktiga bibliotek som krävs för att bland annat skapa en koppling mellan MongoDB och PHP. För installation och introduktion till MySQL användes MySQLs egna dokumentation (MySQL docs, 2018).

Viktigt att tänka på när man hämtar data från både MySQL och MongoDB är det format datan hämtas i. MySQL data hämtas i form av strängar och MongoDB data hämtas i BSON-format vilket inte går att läsa i Javascript utan datan måste omvandlas till JSON-Format. Stackoverflow ger exempel på lösning vid bägge tillfällen. Paolo Bergantino (2008) ger svar på omvandling av MySQL data och Sammaye (2013) ger svar vid omvandling av BSON data.

4.2 Implementation och progression

4.2.1 Databaser

Det första som krävs är en fungerade server för MySQL respektive MongoDB. De guider som använts för att genomföra detta är databasernas individuella dokumentation och i MongoDBs. Det är viktigt att datan i de bägge databaserna följer en så identisk struktur som möjligt, detta då de ska innehålla samma data och datan ska användas till samma sak. Figur 5 visar ett exempel på hur data struktureras i MySQL (Se även Apppendix X).

```
create table diveLog(
    diveDate date,
    Location varchar (50),
    Depth int,
    diveTime int,
    Sightings varchar(50),
    pressureStart int,
    pressureEnd int,
    primary key(diveDate, Location)
)engine=innodb;
```

Figur 5 Exempel på datastruktur i MySQL

I början fanns planer att skapa en fullständig applikation med bland annat inloggning, detta skulle kunna vara användbart vid sökning i databaserna då man skulle kunna skapa mer komplexa frågor mot databasen, i slutändan beslutades det dock att endast ha med dyk och det som sågs under dyker, detta för att underlätta vid automatisering av sökning och datagenerering. Slutresultatet blev då att MySQL innehåller två tabeller, en för dyk och en för de djur som setts. Dessa två blir sedan sammankopplade via en gemensam id, i en komplett applikation skulle detta id representera en specifik användare. För att koppla samman dyk med djur i MongoDB används inbäddad data i MongoDB som tillåter att man lagrar en array direkt i en samling.

4.2.2 Hämta data

När man genomför en sökning måste man hämta data från databaserna. För att göra detta möjligt används Javascript (Appendix A) för att öppna kommunikation mellan applikationen och PHP och PHP (Appendix B och Appendix D) används sedan för att kommunicera med databaserna. För att hämta data skapas ett AJAX-anrop i Javascript, detta anrop kallar sedan på en funktion i PHP som sedan utför sin uppgift, till exempel att hämta data från en databas. Datan omvandlas sedan till JSON och skickas sedan tillbaka till Javascript-funktionen som presenterar datan på applikationen. Figur 6 visar ett exempel på hur ett anrop i AJAX ser ut.

```
$.ajax({
    type: "POST",
    url: activeURL,
    data: {searchData : $("#searchField").val()},
    dataType: "json",
    success: function(data) {
     var obj = data;
     var result = "<ul>"
     $.each(obj, function() {
       result = result + "<li>Date : " + this['diveDate'] + " , Location : " + this['Location'] +
"</li>";
     });
     result = result + "</ul>"
     $("#result").html(result);

     var diffTime = (new Date).getTime() - startTime;
     storeTime(diffTime);
    },
    error: function(exception){
     console.log("ERROR " + exception.responseText);
   }
  });
 });
```

Figur 6 Exempel på ett AJAX-anrop

I anropet deklarerar man bland annat vart anropet ska ta vägen och vilken data som ska skickas vidare. En viktig punkt i anropet är raden dataType: ”json”, detta är nödvändigt då datan som hämtas via anropet måste vara av datatypen json.

Figur 7 visar exempel på hur omvandlingen från MySQL data till JSON går till. För att omvandla från BSON till JSON krävs ett extra steg jämfört med MySQL omvandlingen, detta går att se i figur 8.

```
$result = $sql->fetchAll(PDO::FETCH_ASSOC);

echo json_encode($result);
```

Figur 7 Omvandling MySQL-data till JSON

```
$result = $db->find( [ 'Column' => 'value']);

echo json_encode(iterator_to_array($result));
```

Figur 8 Omvandling BSON till JSON

Det som sker i den ändring som gjorts för BSON är att man itererar genom all data via funktionen iterator to array. Detta krävs då MongoDB, som till skillnad från MySQL som returnerar datan i en ”rå” form, returnerar datan i ett färdigt format, BSON, och man måste därför iterera genom den returnerade datan för att komma åt den faktiska datan. Då det

krävs ett extra steg tar omvandlingen från BSON till JSON lite längre tid än omvandlingen från MySQL datan.

4.2.3 Generera data

Innan man genomför en sökning måste man ha data i sin databas. Från början var planen att skapa ett formulär i applikationen och sedan skicka data från formuläret till databasen. För att generera datan kunde man sedan använda TamperMonkey för att fylla formuläret med data. Detta ändrades dock till ett script direkt i Javascript då mätningarna endast mäter tiden från Ajax-anropet till dess att datan lagrats och ett formulär vore överflödigt för arbetet. Scriptet (**appendix**) som används för att fylla databasen med data hämtar värden från diverse funktioner, dessa värden lagras sedan i databasen. Värdena för djup, tid samt tryck går att hämta via simpla algoritmer som genererar ett värde inom de gränser som har satts. För datum används en funktion som hämtar ett slumpat datum mellan dagens datum och första januari 2000. Då plats kräver specifika värden skapas en array med förbestämda värden, vid genereringen hämtas sedan ett slumpmässigt värde ur arrayen. Vid generering av de djur som setts används återigen en array med förbestämda värden, det som skiljer är att medans det endast hämtas en plats hämtas det ett flertal djur, mängden djur som hämtas sker slumpmässigt mellan 4 och 8 djur.

När data för ett dyk har genererats ska det läggas in i databasen, detta sker genom att kalla på PHP kod via ett AJAX anrop. PHP koden hanterar sedan datan och lagrar den i databasen. Insättning av data skiljer mellan MySQL (**Appendix**) och MongoDB (**Appendix**) då deras datastruktur är väldigt olika. För MongoDB kan man direkt lägga in alla värden, inklusive en array med alla djur, i ett dokument utan problem. MySQL däremot har inget stöd för att lagra en array direkt in i databasen utan man måste istället lagra varje värde i arrayen enskilt. Detta betyder att när MongoDB endast genomför en insättning av data sker ett flertal i MySQL, en för dykdatan och sedan en för varje djur som sågs under dyket.

4.2.4 Prestandamätning

Den frågeställning arbetet grundats på kräver att prestandan på de bägge databaserna mäts. Resultaten på dessa mätningar ska sedan jämföras och analyseras för att sedan kunna dra en slutsats. För att mäta prestandan på sökningarna används verktyget Tampermonkey som är ett plugin i Google Chrome. Via Tampermonkey kan man exekvera javascriptkod direkt på en hemsida och gör det möjligt att automatisera vissa funktioner. Via Tampermonkey kan specifik kod köras som mäter tiden mellan den punkt att ett uppdrag, till exempel en sökning, startas och den punkt då ett uppdrag är slutfört. För att mäta denna tid lagras den tidpunkt uppdraget startas i millisekunder och sedan hämtas den tidpunkt uppdraget slutförts i millisekunder. När man sedan subtraherar sluttiden med starttiden får man den tid som det tog att exekvera uppdraget. Då samma typ av uppdrag ska ske med både MySQL och MongoDB kommer samma typ av mätning ske för bägge databaser. Figur 9 visar ett exempel på hur automatisering av sökningar kan se ut.

```
function runSearch(counter){
  var searchArray = ['Delfin' , 'Marulk', 'Barracuda', 'Muräna', 'Manta Ray', 'Alligator',
            'Delfin' , 'Marulk', 'Barracuda', 'Muräna', 'Manta Ray', 'Alligator',
            'Delfin' , 'Marulk', 'Barracuda', 'Muräna', 'Manta Ray', 'Alligator', 'Delfin',
'Marulk'];
  if(counter < 20){
    setTimeout(function(){
      $("#searchField").val(searchArray[counter]);
      $("#ajaxButton").click();
      counter = counter + 1;
      console.log("Counter  = " + counter);
      runSearch(counter);
    }, 1000);
  }
  else{
    console.log("Finished  searching");
  }
}
```

Figur 9 Automatiserad sökning i Tampermonkey

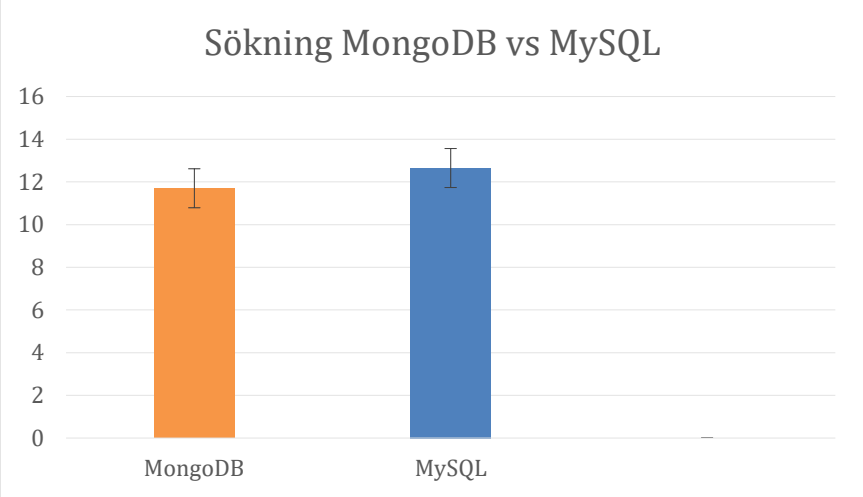
Scriptet påbörjas när knappen trycks ner och kommer sedan att genomföra en sökning till dess att countern når sin punkt. För att ändra mängden sökningar som ska genomföras ändrar man på värdet i if(counter < 20). Om man tillexempel vill genomföra 100 sökningar ändrar men helt enkelt värdet från 20 till 100. För mätning vid insättning av data definierar man mängden mätningar som ska ske direkt i koden.

Det är viktigt att den tid som lagras har mätt korrekt arbete och innehåller så få felfaktorer som möjligt. Från början fanns tanke att mäta tiden via Tampermonkey och man skulle då få tiden från dess att en sökning påbörjas till dess att det är dags för nästa sökning. Det beslutades däremot att detta mätte alldeles för många delar av processen då det skulle inkludera mängder av javascript-funktioner och det vi är mäta är databaser och inte javascript. I slutändan lades mätningen i javascript och mäter tiden från dess att AJAX anropar PHP till dess att datan har lagrats. Detta ger en väldigt korrekt mätning då AJAX-anropet tar extremt kort tid vilket resulterar i att det som främst mäts är den tid det tar för PHP att interagera med databasen. Datan lagras sedan i en textfil via PHP (**APPENDIX**).

4.3 Pilotstudie

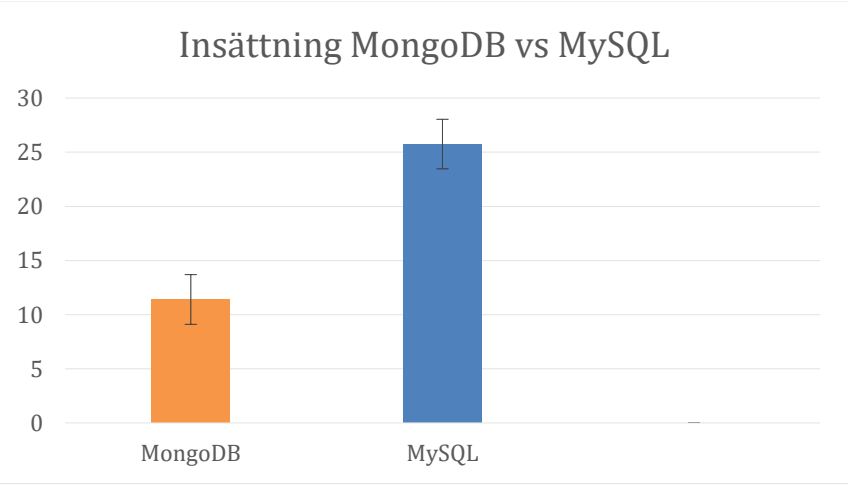
För att se till att mätningarna sker korrekt utfördes en pilotstudie, detta gav även en baseline för framtida tester. Pilotstudien bestod av 2 olika typer av mätningar för varje databas, insättning av data och sökningar, vilket resulterade i sammanlagt 4 mätningar. Varje test bestod av 20 mätningar och sökningarna skedde på den data som generades vid insättnings-

mätningen. Alla mätningar skedde i webbläsaren Google Chrome, via den installerade webbservern och på samma dator.



Figur 10 20 sökningar i millisekunder

I figur 10 kan vi se resultaten vid sökning i respektive databas. Tabellen visar att MongoDB presterar något bättre än MySQL, men det är väldigt lite skillnad mellan de bägge. Detta antyder att MongoDB presterar bättre än MySQL men skillnaden är så låg att det gör skillnaden är försumbar. Vid senare sökningar kommer både datamängd och antal sökningar öka vilket bör ge en större skillnad i prestanda mellan databaserna och en klarare bild på deras prestanda.



Figur 11 20 insättningar i millisekunder

I figur 11 kan vi se resultaten vid insättning av data i respektive databas. Här kan vi se en markant skillnad mellan MongoDB och MySQL, detta beror förmodligen på skillnaden i komplexitet vid insättning av data då MongoDB har stöd för att lagra arrayer medans man i MySQL måste lagra varje värde i en array individuellt.

MongoDB har presterat bättre än MySQL i bägge fall under pilotstudien vilket antyder att MongoDB kommer visa bättre resultat än MySQL i framtida mätningar.

5 Utvärdering

Pilotstudien jämförde prestandan mellan MongoDB och MySQL vid både insättning och sökning av data. Vid sökning presterade de väldigt jämt, MongoDB var någon enstaka millisekund snabbare, men vid insättning gav MongoDB betydligt bättre resultat än MySQL. Det intressanta nu är vad som händer när man ökar mängden data i databasen och se hur detta påverkar databaserna.

Mätningarna kommer ske med tre olika datamängder som bestäms via insättningar. Datamängderna består av 100, 1.000 och 10.000 insättningar av data, skillnaden i datamängd bör ge tydliga skillnader i våra mätningar då mängden data ökar stressnivån i en databas. Testerna är indelade i tre testfall och varje testfall består av sex mätningar.

5.1 Hård- och mjukvara

Alla mätningar har skett på en personlig dator, i tabellen nedan kan man se hårdvaruspecifikationerna för datorn.

Operativsystem	Microsoft Windows 10 Pro Version 10.0.16299 Build 16299
Processor	Intel® Core™ i7-47000Q CPU @2.40GHZ
Minne	8GB DDR3 DIMM 1600MHz
Grafik	NVIDIA Quadro K1100M 2GB
Lagring	Samsung SSD 128GB

Tabell 1 Hårdvaruspecifikation för host

Utöver hårdvara är det även viktigt att hålla koll på den mjukvara som använts, mjukvaran samt de versioner som använts presenteras i tabellen nedan.

Mjukvara	**Version**
Apache	2.4.29
PHP	7.2.2
MySQL	5.7.21
MongoDB	3.6.2
Google Chrome	66.0.3359.139

Tabell 2 Mjukvaruspecifikationer

5.2 Presentation av undersökning

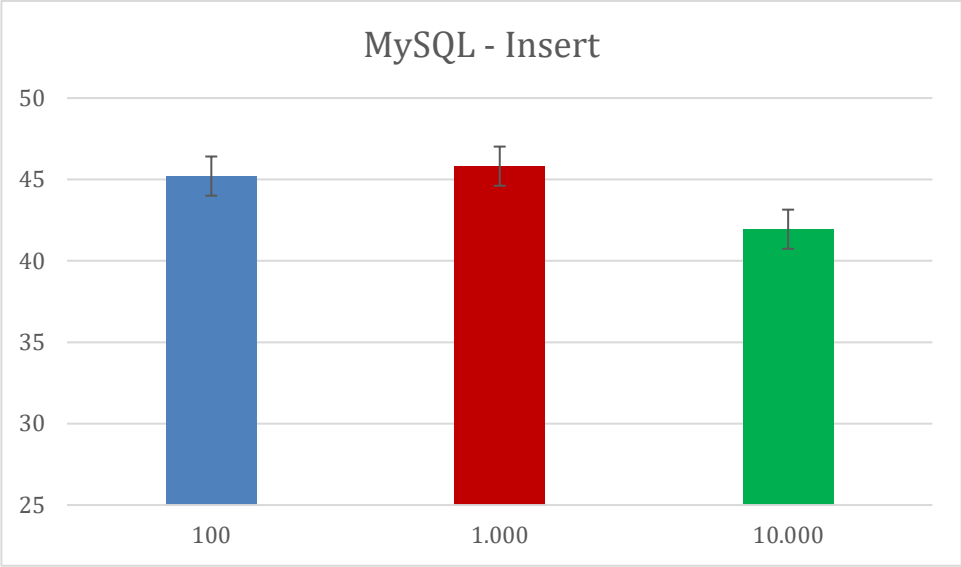
Testerna består av tre olika sorter mätningar, insättning av data, sökning på dykplats och sökning på skådade djur. Vid sökning på dykplats hämtar vi endast information om dyket och

inte de djur som skådats, vid sökning på skådade djur hämtar vi både information om dyket samt alla djur som skådats under dyket. Detta låter oss mäta hur stor skillnad det gör att arbeta med en eller flera tabeller i MySQL. I varje testfall sker en typ av mätning, denna mätning utförs sedan för alla datamängder och bägge databaser.

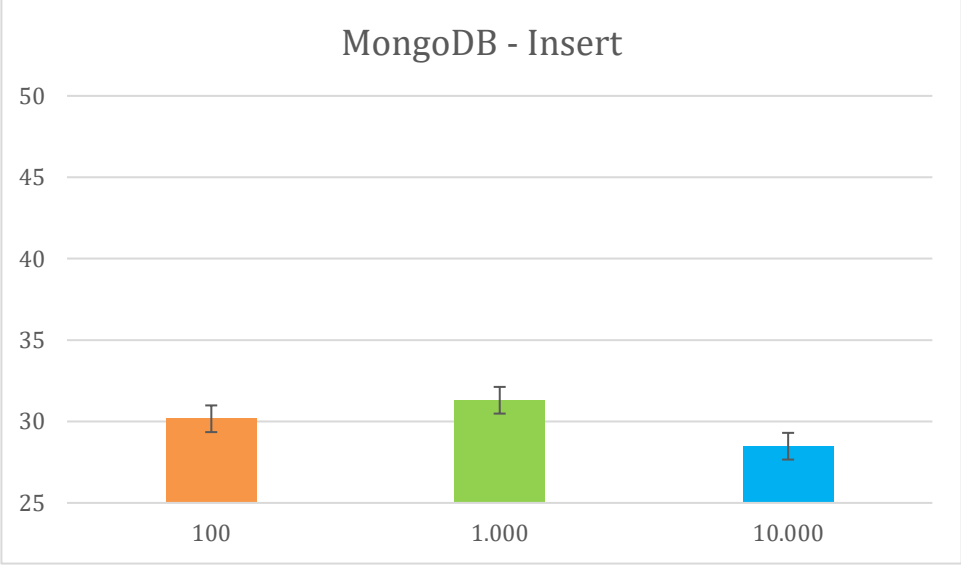
Testfallen undersöker hur väl databaserna presterar med olika datamängder. I varje testfall kommer det finnas tabeller som presenterar den genomsnittliga tiden för mätningarna i millisekunder.

5.2.1 Testfall 1 – Insättning av data

I testfall 1 mäter vi prestandan vid insättning av data. Datan som använts har genererats i javascript och tiden som mätts är den tid det tar från det att PHP anropas för insättning av data till dess att datan har lagrats och returnerar till Javascript. Figur 12 presenterar resultatet för MySQL och figur 13 presenterar resultatet för MongoDB.



Figur 12 Testfall 1 – Insättning i MySQL

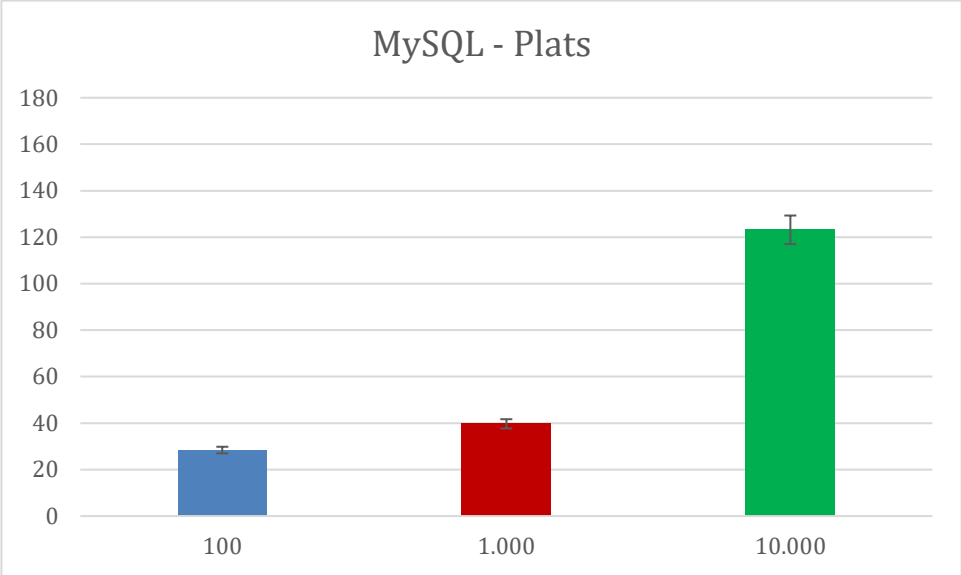


Figur 13 Testfall 1 – Insättning i MongoDB

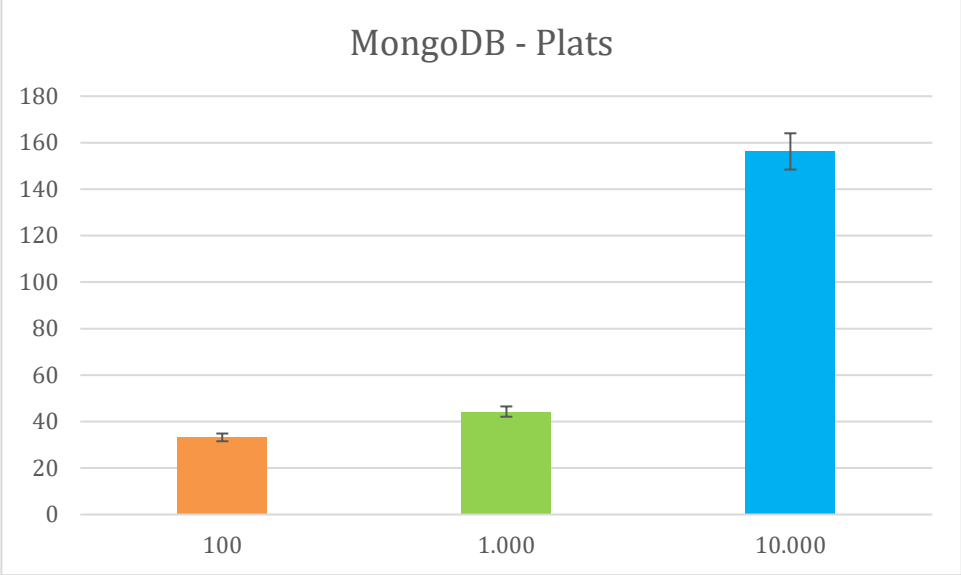
Vid insättning av data presterar MongoDB bättre än MySQL, det är inga större skillnader gällande tid, endast ca 15ms som mest, men procentmässigt blir det en betydlig skillnad då MongoDB genomför sina insättningar 160% snabbare. Något som är intressant är att insättningar av 10.000 rader ger bäst genomsnittlig tid för både MySQL och MongoDB. En faktor för detta kan vara att dyken under insättningen har färre skådade djur i snitt.

5.2.2 Testfall 2 – Sökning av plats

I testfall 2 mäts tiden det tar att söka på en plats och sedan få information om de dyk som skett på platsen. Sökningarna hämtar endast information om dyket och inte om de djur som skådats vilket betyder att endast MySQL endast interagerar med en tabell. I figur 14 presenteras resultatet för MySQL och i figur 15 resultatet för MongoDB.



Figur 14 Testfall 2 – Sökning på plats i MySQL

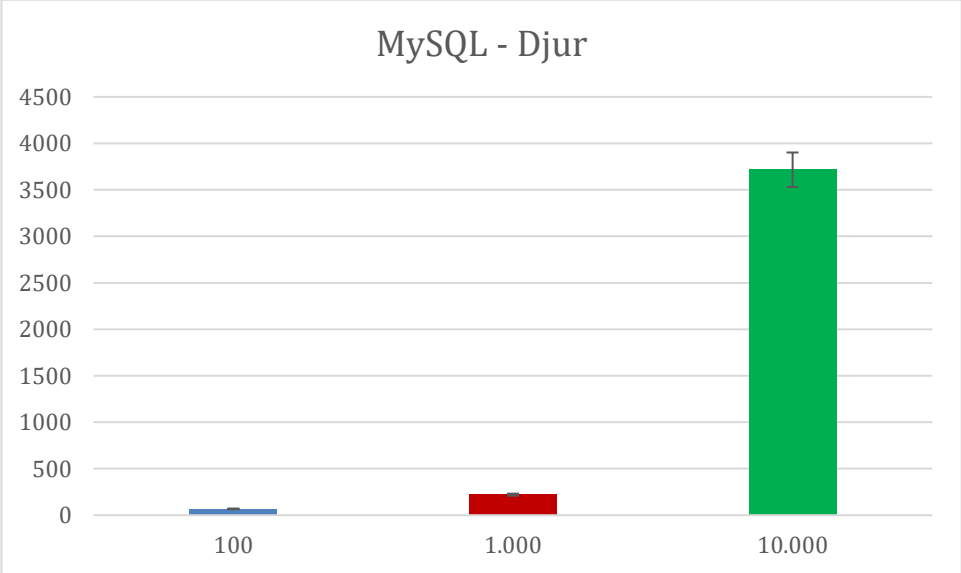


Figur 15 Testfall 2 – Sökning på plats i MongoDB

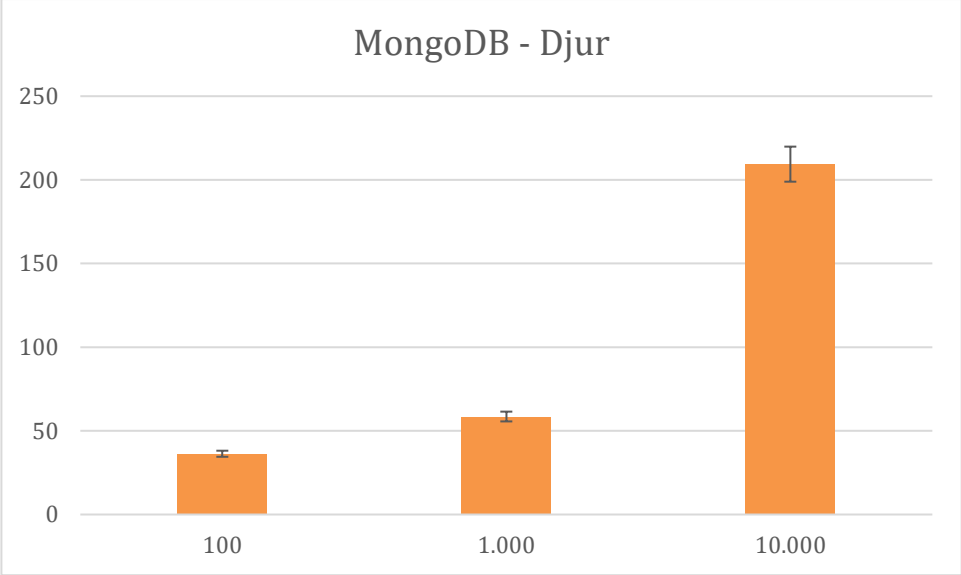
Vid sökning i en tabell presterar MySQL bättre än MongoDB, vid de lägre datamängderna är deras tider nästintill identiska men när datamängden når 10.000 rader tappar finns en tydlig skillnad på 20ms. I bägge fall kan man se en tydlig ökning i söktid när datamängden ökar.

5.2.3 Testfall 3 – Sökning på djur

I testfall 3 sker en sökning på ett djur, utifrån detta hämtas sedan de dyk som djuret has skådats samt alla andra djur som skådats under respektive dyk. I denna sökning interagerar MySQL med två tabeller och för att få fram alla djur för ett dyk kommer interaktion mot den ena tabellen ske ett flertal gånger. Då MongoDB lagrar all data för både dyk och djur i samma collection kommer interaktionen endast ske mot en collection, däremot kommer sökning ske mot inbäddad data och det ska bli intressant att se om detta påverkar responstid. I figur 16 presenteras resultatet för MySQL och i figur 17 resultatet för MongoDB.

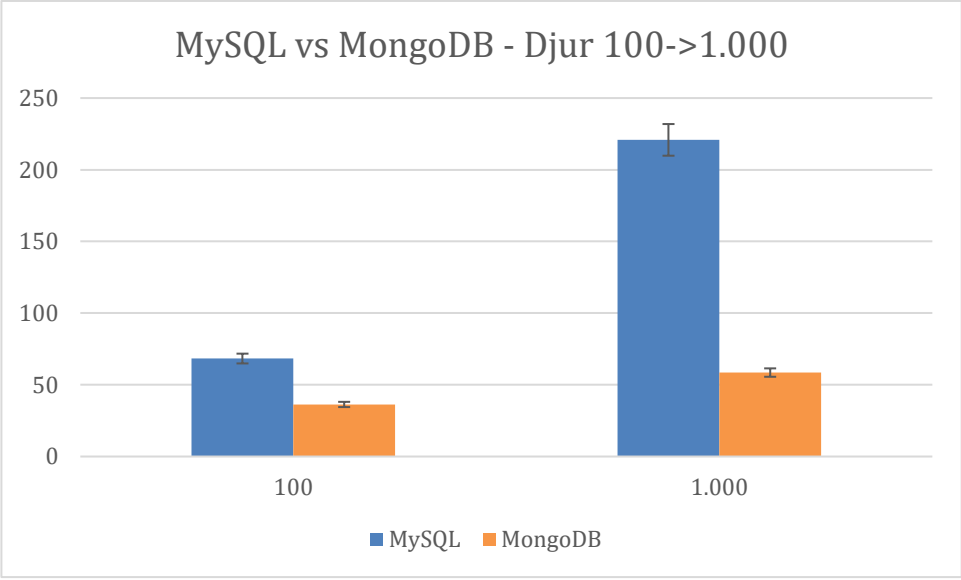


Figur 16 Testfall 3 – Sökning på djur i MySQL



Figur 17 Testfall 3 – Sökning på djur i MongoDB

Vid sökning på djur och interaktion med flera tabeller tappar MySQL stora mängder prestanda, särskilt vid stora datamängder. Man kan se en tydlig skillnad i söktid mellan 100 och 1.000 rader men det är vid 10.000 rader som responstiden skenar iväg och mätningarna tar flera sekunder i genomsnitt. Även i MongoDB sker en stor ökning vid större datamängd men den genomsnittliga söktiden är betydlig lägre för alla datamängder. I figur 18 kan man tydligare se skillnaden i responstid mellan MySQL och MongoDB vid sökning i 100 och 1.000 rader.



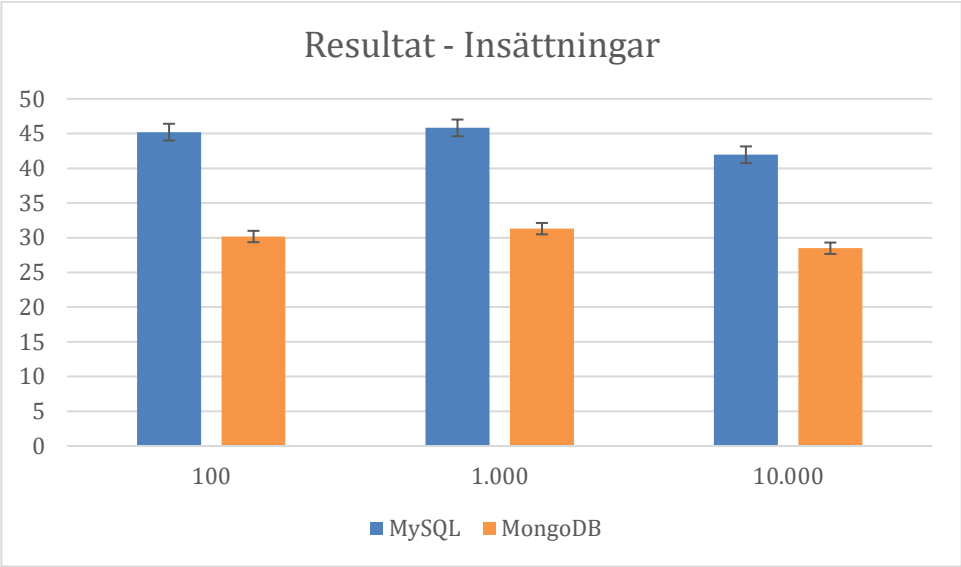
Figur 18 Testfall 3 – MySQL vs MongoDB sökning 100 och 1.00 rader

I figur Z ser vi tydligt hur MongoDB utklassar MySQL även vid de lägre datamängderna. Anledningen till att det blir så stor skillnad mellan de bägge databaserna är förmodligen att

MySQL hämtar data en gång från dyktabellen och sedan en går per skådat djur från djurtabellen medans MongoDB endast hämtar data från det dokument/dyk där djuret har skådats.

5.3 Analys

I analysen jämförs de olika resultaten mot varandra för att se vilken databas som visat bäst resultat. Resultaten kommer allt kombineras och visas upp i grafer för att visa ett en helhet av de mätningar som har gjort. I figur 19 presenteras resultatet från testfall 1.

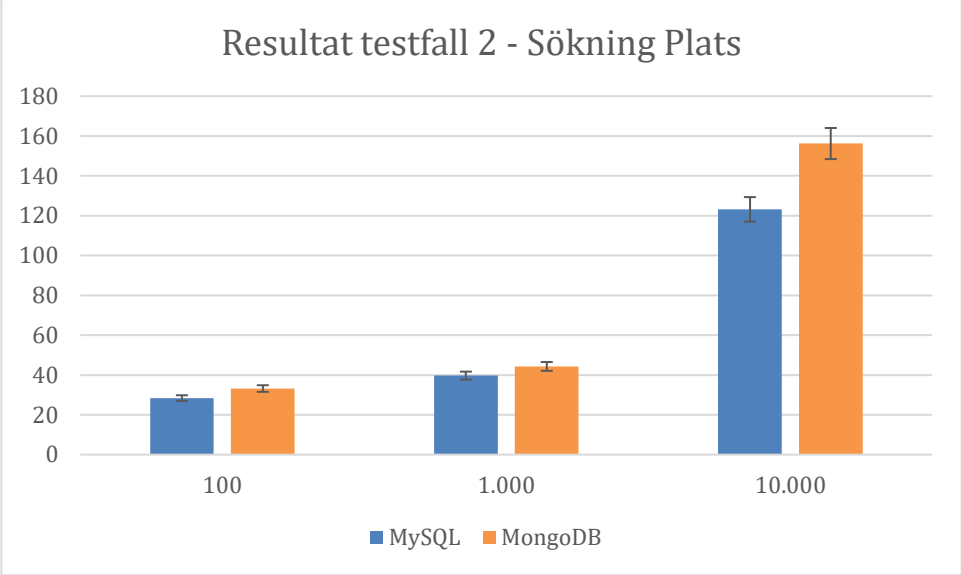


Figur 19 Resultat testfall 1

Från det resultat vi fått ifrån testfall 1 kan vi se att MongoDB presterar bättre än MySQL vid insättning av data. Bägge databaser verkar hålla en jämn prestanda oavsett datamängd, de visar faktiskt bättre prestanda vid 10.000 rader än vid de lägre datamängderna. En anledning för detta kan vara att datagenerering genererade fler dyk med en låg mängd sedda djur jämfört med de lägre datamängdernas data. Detta bör dock inte vara fallet då generering sker på så sätt att datan bör bli så lik som möjligt. En mer trolig anledning till den ökade prestandan vid högre datamängd är att det tar en stund för databasen att komma igång och vid större datamängd sker fler insättningar i det skede då databasen arbetar som bäst.

Utifrån testfall 1 kan vi då säga att vid insättning av data i min artefakt är MongoDB är det bättre valet för en dykapplikation.

I figur 20 kan vi se resultatet från testfall 2 där vi sökte på en plats och hämtade information om de dyk som skedde där.

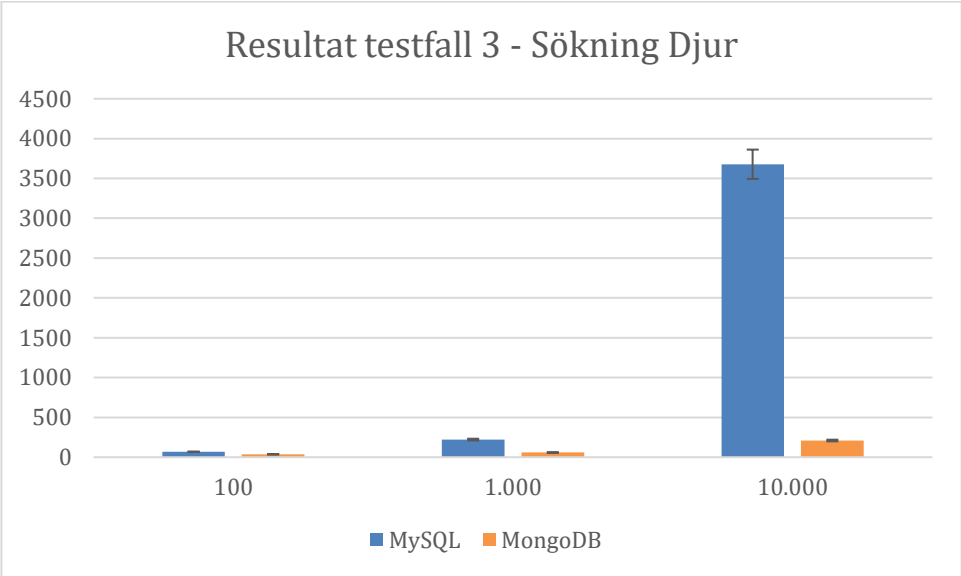


Figur 20 Resultat testfall 2

I testfall 2 presterade MySQL bättre än MongoDB och skillnaden blir tydligare vid högre datamängder. En anledning till detta kan vara strukturen på databaserna. MySQL tillåter att jag endast hämtar information om dyket och inte några av de skådade djuren, detta är inte fallet för MongoDB då djuren är inbäddade i dykdatan och det går inte separera på de bägge vid hämtning. Detta innebär att MongoDB hämtar mer data än MySQL vilket kan påverka prestandan men då bägge databaser hämtar datan utan att använda sig av referenser bör detta inte vara den enda faktorn.

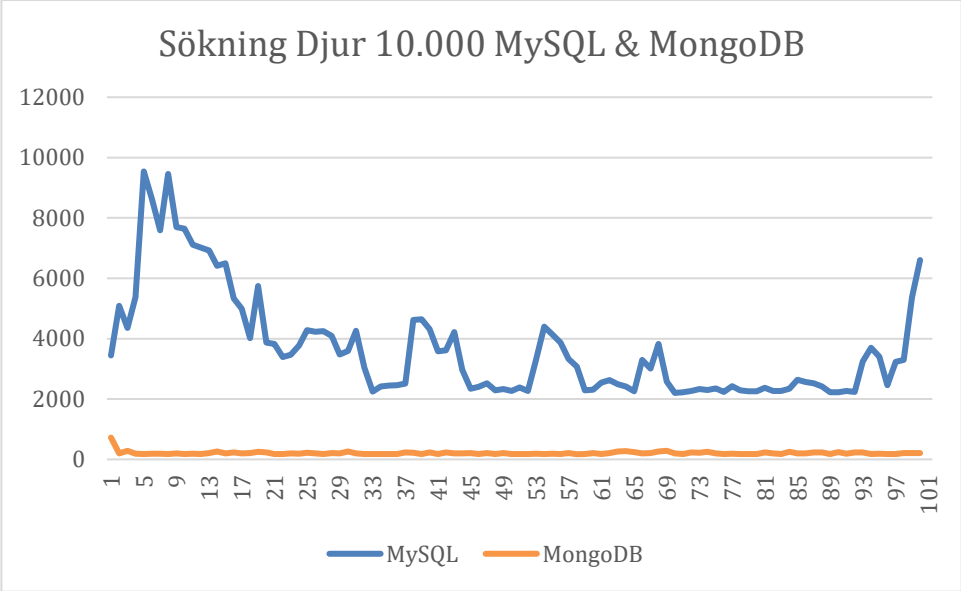
Utifrån testfall 2 kan vi då säga att MySQL är det bättre valet av databas när man ska hämta data från en enskild tabell.

I figur 21 presenteras resultatet från testfall 3 där vi söte efter ett djur och sedan hämtade både information om det dyk som djuret skådades samt alla djur som skådats under dyket.



Figur 21 Resultat testfall 3

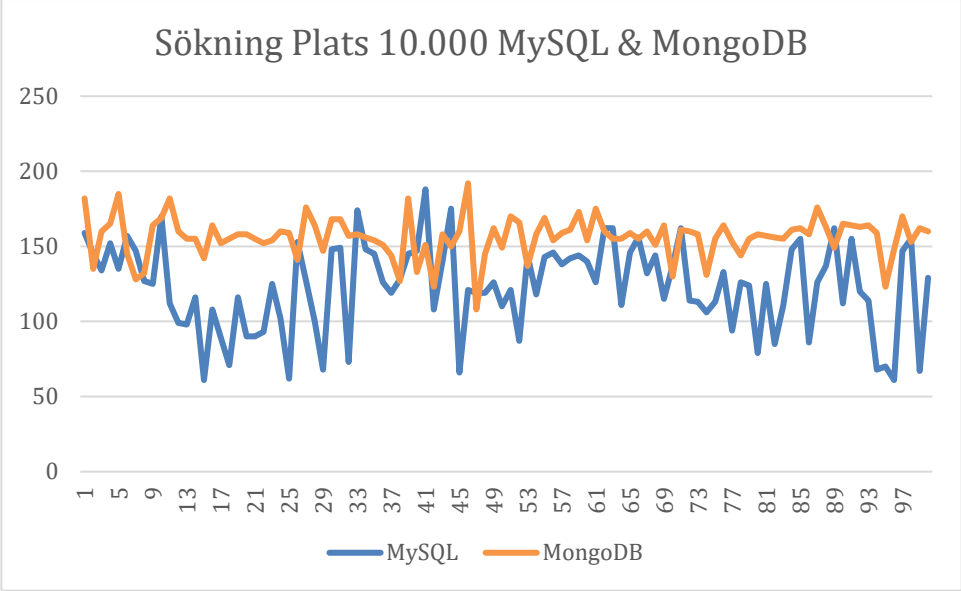
Resultatet i testfall 3 visar tydligt hur MySQL förlorar prestanda när flera tabeller ska användas vid stora datamängder och kurvan ökar kraftig desto högre datamängd man använder. Prestandan hos MongoDB försämras även den vid högre datamängder men jämfört med MySQL är försämringen väldigt låg. En anledning till den kraftiga försämringen av prestanda i MySQL är att MySQL måste hämta data från djurtabellen flera gånger för att hämta alla djur. I figur 22 och 23 presenteras alla sökningar som skett i respektive databas med datamängden 10.000.



Figur 22 Linjediagram - Sökning djur 10.00 MongoDB & MySQL

I figur 22 kan vi se hur mätningarna tar olika tid. De tider i botten av grafen är troligtvis de mätningar där vi sökt på djur som förekommer sällan i databasen och de högre tiderna sker när vi sökt på djur som förekommer ofta i databasen. Den kraftiga ökningen i tid i början av mätningarna beror troligtvis på den stress som databasen utför för när mätningarna börjar, därefter sjunker mätningarna ner till en något mer stadig nivå med man kan fortfarande se stora skillnader i mättid. Stressen vid början av sökningarna syns för bägge databaser, i MongoDB kan man se ett högt värde direkt i början som sedan går ner till en stadigare nivå direkt. Hos MySQL däremot så sker stressen en liten bit in i mätningarna, anledningen till detta är att MySQL är snabbare med att påbörja hämtningen av data men då detta tar längre tid än förväntat hamnar databasen efter med uppgifterna och stressen ökar därefter. När databasen väl kommit igång med hämtningarna håller den en någorlunda jämn nivå. Anledningen till det höga värdet i slutet av MySQL sökningen är okänt men eftersom resultatet mellan databaserna är så tydligt som det är gjordes beslutet att ignorera det.

I figur 23 kan vi se hur mätningarna varierar vid sökning av plats för respektive databas.

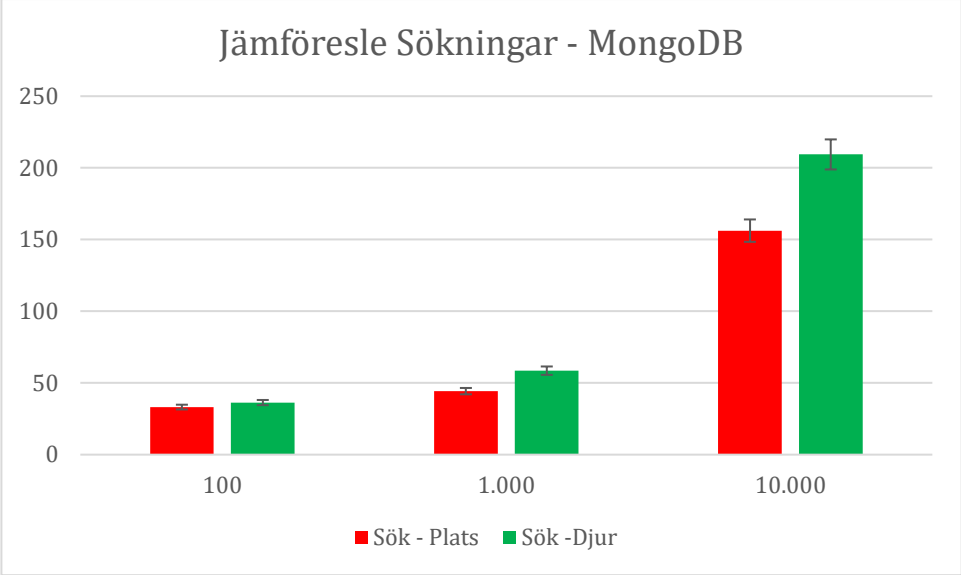


Figur 23 Linjediagram – Sökning plats 10.000 MySQL & MongoDB

Även här kan vi se att bägge databaserna har en liten ökning vid starten som sedan sjunker. Det som skiljer sig åt jämfört med sökning av djur är att mättiderna når upp till den initiella startiden senare i mätningarna. För MySQL beror detta troligen på att databasen inte upplever samma stress som den gör vid sökning i flera tabeller. För MongoDB däremot är den enda skillnaden at vi söker i inbäddad data, man får då fundera ifall den höga initieringstiden hos MongoDB i figur 22 beror på initiering av inbäddad data eller om den höga tiden beror på något annan faktor.

Det som påverkar spikes hos figur 23 är troligtvis mängden resultat på den sökning som sker. På vissa platser kan det ha skett 100 dyk medans andra har 1.000 dyk vilket påverkar mättiden.

En intressant sak att notera är skillnaden mellan de olika sökningarna i MongoDB, i testfall 2 söker vi på ett värde som lagras direkt i ett dokument, i testfall 3 däremot söker vi på ett värde som lagrats i samma dokument fast i form av inbäddad data. I figur 24 kan vi se skillnaden i resultat mellan dessa två sökningar.



Figur 24 Sökning på Plats vs Djur i MongoDB

I figuren kan vi se att trots det faktum att bägge sökningar hämta all data från samma dokument så tar det olika lång tid att hämta datan. När vi söker på en plats, vilket är ett värde som ligger direkt i dokumentet, så tar det mindre tid än när vi söker på ett djur, vars värde ligger i den inbäddade datan i dokumentet. Detta är främst en intressant notering då det ger mer information om hur MongoDB arbetar med datan.

Vi kan utifrån testfall 3 säga att MongoDB är det klart bättre valet för interaktion av data i flera tabeller vilket kommer vara nödvändigt för en dykapplikation.

Utifrån de resultat som testfallen har gett kan vi se att när man bör använda MongoDB före MySQL när man ska bygga en dykapplikation. MongoDB presterade bättre vid insättning av data vilket är en viktig del av en applikation. MySQL presterade däremot bättre vid hämtning av data från en tabell men skillnaden i prestandan vid sökning i en tabell är nästintill obefintlig och när man arbetar i en dykapplikation kommer man oftast behöva data från en mängd olika tabeller.

5.4 Slutsatser

Frågeställningen som ställdes i studien var:

”*Är MongoDB mer effektiv är MySQL vid interaktion av data för en dykapplikation”,*

och studiens hypotes var:

”Genom att använda sig av MongoDB kan man få bättre svarstider än med MySQL, särskilt vid användning av större datamängder”.

Utifrån de resultat som tagits fram kan vi dra slutsatsen att hypotesen stämmer. MongoDB presterade bättre än MySQL i två av tre testfall och MySQL visade betydligt större svårigheter vid stora datamängder. Vid insättning av data kommer MongoDB hålla en jämn nivå som alltid är snäppet snabbare är MySQL, MongoDB har en genomsnittlig insättningstid runt 30ms medans MySQL har en genomsnittlig tid runt 45ms, detta betyder att MongoDB presterar hela 50% snabbare än MySQL vid insättning av data.

Vid hämtning av data får vi två olika resultat, MySQL presterade bättre än MongoDB vid sökning av plats där MySQL har en genomsnittlig tid på 120ms medans MongoDB har en genomsnittlig tid på 155ms vid sökning i 10.000 rader, MySQL presterar då ca 30% bättre än MongoDB. När det däremot kommer till sökning av djur och interaktion av flera tabeller presterar MongoDB betydligt bättre med en genomsnittlig tid på ca 200ms jämfört med MySQLs tid på 3600ms vid sökning i 10.000 rader, MongoDB är då 18 gånger snabbare än MySQL, vilket är en enorm skillnad.

Detta betyder att även om MySQL visar bättre resultat vid ett av testfallen så är det även det testfall med minst skillnad i resultaten och även om MySQL skulle vara den databas som presterade bättre så skulle den gigantiska skillnaden i testfall 3 fortfarande avgöra då sökningar sker oftare än insättningar i de flesta applikationer.

Detta svarar även på frågeställningen, utifrån resultaten är MongoDB mer effektiv än MySQL vid interaktion av data för en dykapplikation. MongoDB kan hämta data under sekunden även vid större datamängder medans MySQL har svarstider på flera sekunder.

6 Avslutande diskussion

6.1 Sammanfattning

För att enkelt interagera med dykdata både för personligt bruk och inom forskning behöver man använda sig av en databas. Det finns en enorm mängd olika databaser av olika slag och beroende på vilken databas man använder kan den tid det tar att interagera med datan variera enormt och eftersom mängden data i en dykapplikation kan nå store mängder behöver databasen kunna hantera dessa mängder data. Det första steget för att hitta den optimala databasen för applikationen är att jämföra två olika typer av databas, en SQL databas mot en NoSQL databas, vilket är det studien undersöker. Den frågeställning som studien undersöker är *"Är MongoDB mer effektiv är MySQL vid interaktion av data för en dykapplikation"*.

Under studien utförs experiment för att svara på frågeställningen. MySQL och MongoDB installeras på en server och sedan sker olika testfall för att mäta deras prestanda. I det första testfallen undersöks prestandan vid insättning av data. Det genereras data som har baserats på dykloggar, denna data läggs sedan in i de bägge databaserna i olika datamängder för att se hur väl de hanterar dels insättningen av data samt hur mängden data i databasen påverkar prestandan. I de övriga testfallen hämtas data ur denna datamängd. Den ena sökningen hämtar data om ett specifikt dyk och använder endast en av tabellerna i MySQL. Den andra sökningen hämtar inte bara data om dyket utan inkluderar även alla de djur som skådats under dyket. Detta innebär att MySQL interagerar med fler än en tabell och MongoDB söker i inbäddad data. Genom de olika sökningarna kan man se hur databaserna hanterar mer komplexa förfrågningar samt hur datamängd påverkar prestandan vid sökning.

Det resultat som studien tagit fram, och presenterar i kapitel 5, visar att MongoDB är det bättre valet av databas för en dykapplikation. MongoDB presterade bättre vid både insättning av data och hantering av komplexa förfrågningar. MySQL var något snabbare än MongoDB vid interaktion med endast en tabell men då en dykapplikation kräver en mängd tabeller är hantering av komplexa förfrågningar viktigare än interaktion med en tabell och resultatet för det var väldigt tydligt vilket syns i **FIGUR SOM VISAR TESTFLL 3**

6.2 Diskussion

Studien undersöker vilken av två databaser som vore det bättre alternativet för en dykapplikation. När man arbetar med tekniska aspekter så som databaser ligger i prestanda i fokus vilket gör experiment en bra metod för att undersöka och jämföra databaserna. Experiment är repeterbara och sker i en sluten miljö vilket gör det möjligt att fokusera och undersöka specifika delar för att få fram så klara resultat som möjligt (Wohlin et al. 2012). Ett problem med experiment är den kod som ligger i grunden och det förtroende man har i koden. Vid experiment är det viktigt att ha i åtanke alla de olika faktorer som kan påverka och förbättra eller försämra resultaten. När man utför ett experiment måste man även vara säker att det man utför har någon form av nytta. Man får oftast fram ett klart resultat där man kan jämföra värden men om skillnaden i resultatet endast är ett par millisekunder så får man även fråga sig själv om det gör någon skillnad (Wohlin et al. 2012).

I jämförelse med den studie som utfördes av Györödi et al. (2015) så var de databaser som användes i studien simpla. I studien användes endast 2 tabeller och ett dokument medans Györödi et al. (2015) använde sig utav 5 tabeller och dokument och fick väldigt tydliga resultat

även vid insert, där MongoDB presterade betydligt bättre än MySQL (440sek mot 0,29sek) för sammanlagd tid vid insättning av 10.000 användare. I studien var skillnaden vid insert mellan MySQL och MongoDB tydlig (43ms för MySQL mot 28ms för MongoDB per insättning) men resultatet hos Györödi et al. (2015) tyder att skillnaden blir ännu större när databasernas komplexitet ökar.

De databaser som använts i studien representerade en specifik del av en komplett applikation, om man sedan fortsätter bygga ut databaserna kan resultaten påverkas och frågeställningarna mot databaserna kan göras mer och mer komplexa. Detta är något man måste ha i åtanke när man läser resultatet från denna studie då studien visar vad som bör vara det bättre alternativet men ju mer man lägger till i en applikation desto fler faktorer skapas som kan påverka prestandan.

Det man kan ta med sig ifrån studien är hur det bägge databaserna presterar och vad man bör tänka på när man ska bygga en dykapplikation och strukturera databaserna för dykdata. Först och främst kan man tydligt se att MySQL har problem med att hämta data från en mängd tabeller och tappar prestanda när detta ska ske. Man bör därför använda sig utav t.ex views för att koppla samman tabellerna. Via views kan man skapa en tabell som innehåller all data från bägge tabellerna och när man sedan utför en sökning kommer endast en tabell behövas istället för att man använder flera stycken. Detta skulle öka prestandan enormt och utifrån testfall 2 kan man till och med anta att MySQL skulle prestera bättre än MongoDB. Anledningen till att views inte användes i studien var för att specifikt testa prestandan vid interaktion av många tabeller för att se hur detta skulle påverka prestandan och när man gör mer komplexa frågeställningar finns det ingen garanti att man kan lösa problemet via views. Man kan även se att MongoDB presterar något sämre vid sökning av inbäddad data vilket kan vara bra att ha i åtanke när man vill uppnå optimal prestanda i databasen.

Sammanfattat är samhällsnyttan med studien att man kan se vissa för- och nackdelar med databaserna för att man sedan ska kunna se vad man bör tänka på för att få fram en så optimal databas som möjligt.

6.2.1 Etik

Eftersom studien publicerat all hård- och mjukvara samt all programkod (se appendix) som använts är det möjligt att upprepa studien ur ett forskningsetiskt perspektiv. Andra utvecklare kan enkelt upprepa det arbete som genomförts och genomföra te tester som utförts. Detta gör det även enklare att fortsätta studien ytterligare. Dock skapar detta även ett etiskt problem eftersom studien inte kan garantera samma resultat vid användning av t.ex en annan typ av server. Det vore därför intressant att uppfölja studien med annan hård- och mjukvara.

Studien forskar inom vilken typ av databas man bör använda sig av vid utveckling av en dykapplikation. Studien visar vilka resultat man kan förvänta sig från databaserna samt hur de arbetar med olika dataset.

Något som är viktigt att ha i åtanke är de datamängder som använts. Den största datamängd i studien är 10.000 rader men då erfarna dykare har över hundra dyk kan man snabbt uppnå datamängden. Det är därför viktigt att inte bara kolla på de tider som presenterats utan även hur dessa tider påverkas av mängden data. I studien kan man se att datamängd gör stor skillnad vid interaktion av data och man kan se en över hur prestandan sjunker när datamängden ökar.

Det är även viktigt att ha i åtanke att studien endast undersöker en liten del av en dykapplikation och även om resultatet påvisar att MongoDB bör prestera bättre än MySQL finns det ingen garanti att resultatet blir detsamma i en fullt utvecklad presentation. Studien visar dock tydligt att MySQL har problem med större datamängder och vid en fullt utvecklad applikation kommer denna datamängd troligtvis nå större mängder än de som använts i studien.

MongoDB har haft en historia med säkerhetsrisker vilket inte är allt för förvånande då den första versionen släpptes 2009 vilket gör det till en relativt ny databas. Utöver detta tillhör MongoDB även NoSQL gruppen som har haft problem med säkerhet i helhet. Detta har givetvis förbättrats med åren men till skillnad från MySQL som har särskilda inbyggda skydd bör man lägga ner extra tid och dubbelkolla säkerheten gällande MongoDB vid lagring av känslig data.

6.3 Framtida arbete

Syftet med studien var att hitta den optimala databasen för en dykapplikation och fokus lades på att jämföra MySQL med MongoDB. Vi har fått vårt resultat gällande specifikt dessa databaser men det finns ingen garanti att någon av dom är den optimala databasen då det finns en mängd databaser utöver MySQL och MongoDB, tillexempel kan man använda sig utav PostgreSQL istället för MySQL och Cassandra istället för MongoDB för att genomföra studien med andra databaser. I framtida arbeten kan man då utföra studier gällande andra databaser och fortsätta arbetet med att hitta den optimala databasen. Då studien använt sig av endast en form av webbläsare och server kan man enkelt upprepa studien med en annan databas. Det vore även intressant att upprepa studien med en annan form av webbläsare och/eller server för att även hitta den optimala lösningen där.

Man skulle även kunna bygga ut applikationen och databaserna med t. ex användare och inloggning och på så sätt skapa en mer komplex databas som tillåter mer komplexa operationer mot databasen för att sedan undersöka hur komplexitet påverkar databaserna.

Då MySQL vid sökning av djur hämtar data flera gånger från olika tabeller vore det även intressant försöka optimera databaserna och se om man kan få ett annat resultat. Det vore även intressant att försöka optimera saker utöver databaser som t.ex kommunikationen mellan databaserna och PHP.

Man skulle även kunna skapa en dykdator som automatiskt fyller databaserna med data. Detta skulle se till att datan alltid håller korrekt struktur och man skulle få dom exakta värdena för dyket. Om man sedan även byggt ut applikationen skulle en sådan dator kunna kopplas till en användare och på så sätt skulle dyket kunna kopplas till individen automatiskt och man skulle få in alla sina dyk på ett smidigt sätt. Med nuvarande teknologi skulle det dock inte vara möjligt att fylla in de djur som skådats under ett dyk då det inte finns teknologi som kan hämta information från det som ses. Om man då tänker långt in i framtiden kanske det även går att få fram teknologi som kan se ett djur och sedan få fram information om djuret och koppla det djuret till det nuvarande dyket.

I framtiden kan det dyka upp ytterligare saker att undersöka då det hela tiden utvecklas nya teknologier och det finns möjlighet att hitta betydligt bättre metoder som ännu inte används.

Referenser

Abramova, V. and Bernardino, J., 2013, July. NoSQL databases: MongoDB vs cassandra. In Proceedings of the international C* conference on computer science and software engineering (pp. 14-22). ACM.

Adermann. N, Boggiano. J (2018) Composer v1.6.5. https://getcomposer.org/

Berndtsson, M., Hansson, J., Olsson, B. and Lundell, B., 2007. Thesis projects: a guide for students in computer science and information systems. Springer Science & Business Media.

Brewer, E., 2012. CAP twelve years later: How the" rules" have changed. Computer, 45(2), pp.23-29.

Cattell, R., 2011. Scalable SQL and NoSQL data stores. Acm Sigmod Record, 39(4), pp.12-27.

Diviac (2018) https://logbook.diviac.com/

Győrödi, C., Győrödi, R., Pecherle, G. and Olah, A., 2015, June. A comparative study: MongoDB vs. MySQL. In Engineering of Modern Electric Systems (EMES), 2015 13th International Conference on (pp. 1-6). IEEE.

Hows, D., Membrey, P., Plugge, E., Hawkins, T. (2013). The Definitive Guide to MongoDB. Publisher: Apress. ISBN: 978-1-4302-5821-6.

Lutz, R., Ameri, P., Latzko, T., Meyer, J., Gómez, J.M., Sonnenschein, M., Vogel, U., Winter, A. and Rapp, B., 2014, September. Management of Meteorological Mass Data with MongoDB. In EnviroInfo (pp. 549-556).

Mahmood, K., Risch, T. and Zhu, M., 2015, July. Utilizing a nosql data store for scalable log analysis. In Proceedings of the 19th International Database Engineering & Applications Symposium (pp. 49-55). ACM.

MongoDB docs (2018). https://docs.mongodb.com/v3.6

MySQL docs (2018). https://dev.mysql.com/doc/refman/8.0/en/

Swaminathan, S.N. and Elmasri, R., 2016, June. Quantitative analysis of scalable nosql databases. In Big Data (BigData Congress), 2016 IEEE International Congress on (pp. 323-326). IEEE.

The PHP Group (2018) http://php.net/manual/en/set.mongodb.php

Vaish, G. (2013). Getting started with NoSQL. Publisher: Packt Publishing. ISBN: 10:1- 84969-499-0.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., & Wesslén, A. (2012). Experimentation in Software Engineering. Publisher: Springer-Verlag Berlin Heidelberg. ISBN:978-3-642-29043-5.

Appendix A - GenerateData.js

```
// Clear databases and textfiles
function prepareInsert(){
	var loop = 0;
	$.ajax({
		type: "POST",
		url: "resetDB.php",
		cache: false,
		data: {
			dbType: db,
			searches: $("#dataAmount").val(),
			storeType: "insert"
		},
		success: function(data){
			console.log(data);
			generateData(loop);
		},
		error: function(exception){
			console.log(exception.responseText);
		}
	})
}

// Function to generate data
function generateData(){
	var counter = 0;
	var id = 0;
	var dives = [];
```

```
var allDives = [];

// Interval loops until close with 700ms delay
var loop = setInterval(function () {
	id++;

	allDives = randomValues();	// Generate diveLog data

	// Generate sightings
	var sightings = [];
	var numOfSightings = Math.floor((Math.random() * 4) + 4)
	for(var i = 0; i<numOfSightings; i++){
		var newSighting = randomSighting();
		if(sightings.includes(newSighting)){
			i--;
		}
		else{
			sightings.push(newSighting);
		}

	}
	var sights = [id, sightings];

	insertData(allDives, sights);

	console.log(counter);
	counter++;
	if(counter >= $("#dataAmount").val()){ clearInterval(loop);}
```

```
        }, 700)
}

// Function for calling PHP-Insertion
function insertData(dives, sightings){
        var startTime = (new Date).getTime();

        JSON.stringify(dives);
        $.ajax({
                type: "POST",
                url: activeURL,
                cache: false,
                data: {
                        diveData : dives,
                        sightData: sightings},
                        success: function(data){
                                var diffTime = (new Date).getTime() - startTime;
                                storeInsertTime(diffTime);
                        },
                        error: function(exception){
                                console.log(exception.responseText);
                        }
                })
}

// Function for storing time
function storeInsertTime(searchTime){
        $.ajax({
                type: "POST",
```

```
                url: "writeTime.php",
                cache: false,
                data: {
                        storeTime: searchTime,
                        dbType: db,
                        searches: $("#dataAmount").val(),
                        storeType: "insert"
                },
                success: function(data){
                        console.log("Search time: " + data + "ms stored.");
                },
                error: function(exception){
                        console.log(exception.responseText);
                }
        })
}

// Functions for generating data

function randomDate(start, end) {
        var d = new Date(start.getTime() + Math.random() * (end.getTime() - start.getTime()));
        month = '' + (d.getMonth() + 1),
        day = '' + d.getDate(),
        year = d.getFullYear();

        if (month.length < 2) month = '0' + month;
        if (day.length < 2) day = '0' + day;
```

```
        return [year, month, day].join('-');
}

function randomLocation(){
        var Locations =
        [
        "Phuket",
        "Oslo",
        "Miami",
        "Phi Phi Islands",
        "Kairo"
        ];
        var rand = Locations[Math.floor(Math.random() * Locations.length)];
        return rand;
}

function randomSighting(){
        var Animal =
        [
        "Barracuda",
        "Manta Ray",
        "Leopard Shark",
        "Moray Eel",
        "Clownfish",
        "Black Tip Reef Shark",
        "Leopard Shark",
        "Parrot Fish",
        "Eagle Ray",
        "Dolphin",
```

```
        "Tuna"
        ];
        var rand = Animal[Math.floor(Math.random() * Animal.length)];
        return rand;
}

function randomValues(){

        var date = randomDate(new Date(2000, 0, 1), new Date());
        var diveLocation = randomLocation();
        var depth = Math.floor((Math.random() * 16) + 15);
        var diveTime = Math.floor((Math.random() * 31) + 30);

        var pressStart = Math.floor((Math.random() * 31) + 170);
        var pressEnd = Math.floor((Math.random() * 61) + 15);

        dives = [date, diveLocation, depth, diveTime, pressStart, pressEnd];
        return (dives);
}
```

Appendix B - fetchData.js

```
function prepareSearch(){
  clearFile();
  var counter = 0;
  getSearchValue(counter);
}

function getSearchValue(counter){
  if(searchType == 'sight'){
    var searchArray = [
    "Barracuda",
    "Manta Ray",
    "Leopard Shark",
    "Moray Eel",
    "Clownfish",
    "Black Tip Reef Shark",
    "Leopard Shark",
    "Parrot Fish",
    "Eagle Ray",
    "Dolphin",
    "Tuna"
    ];
  }
  else if(searchType == 'location'){
    var searchArray = [
    "Phuket",
    "Oslo",
    "Miami",
```

```
  "Phi Phi Islands",
  "Kairo"
  ];
 }
 else{console.log('Searchtype not defines!');}



 if(counter < 100){
  setTimeout(function(){
   var searchValue = searchArray[Math.floor(Math.random() * searchArray.length)];
   searchData(searchValue);
   counter++;;
   console.log("Counter = " + counter);
   getSearchValue(counter);
  }, 500);
 }
 else{
  console.log("Finished searching");
 }
}

function searchData(value){
 var startTime = (new Date).getTime();
 $.ajax({
  type: "POST",
  url: activeURL,
  data: {
   searchData : value,
```

```
  searchType : searchType
},
dataType: "json",
success: function(data) {
  var obj = data;
  var result = "<ul>"
  if(obj.length>0){
    $.each(obj, function() {
      if(db == "mysql"){
        if(this['id'] != null){
          result += "<li>Id : " + this['id'] + ", Date: " + this['diveDate'] +
          ", Location: " + this['Location'] + ", Depth: " + this['Depth'] +
          ", Dive Time: " + this['diveTime'] + ", Pressure Start: "
          + this['pressureStart'] + ", Pressure End: " + this['pressureEnd'] + "</li>";
        }
        else if(this['diveID'] != null){
          result += "<li>Sight : " + this['sight'] + "</li>";
        }
      }
      else if(db == "mongodb"){
        result += "<li> Date: " + this['diveDate'] +
        ", Location: " + this['Location'] + ", Depth: " + this['Depth'] +
        ", Dive Time: " + this['diveTime'] + ", Pressure Start: "
        + this['pressureStart'] + ", Pressure End: " + this['pressureEnd'] +
        ", Sightings: " + this['Sightings'] + "</li>" ;
      }

    });
  }
```

```
      else{
        result += "Location not found!";
      }
      $("#result").html(result);

      var diffTime = (new Date).getTime() - startTime;
      storeTime(diffTime);
    },
    error: function(exception){
     console.log("ERROR " + exception.responseText);
    }
  });
}

function storeTime(searchTime){
  $.ajax({
    type: "POST",
    url: "writeTime.php",
    cache: false,
    data: {
      storeTime: searchTime,
      dbType: db,
      searches: $("#dataAmount").val(),
      storeType: "search-" + searchType
    },
    success: function(data){
      console.log("Search time: " + data + "ms stored.");
    },
    error: function(exception){
```

```
    console.log(exception.responseText);
   }
  })
}

function clearFile(){
  $.ajax({
   type: "POST",
   url: "clearFile.php",
   cache: false,
   data: {
    dbType: db,
    searches: $("#dataAmount").val(),
    storeType: "search-" + searchType
   },
   success: function(data){
    console.log("File cleared!");
   },
   error: function(exception){
    console.log(exception.responseText);
   }
  })
}
```

Appendix C - resetDB.php

```
<?php
$currDB = $_POST['dbType'];
$searches = $_POST['searches'];
$storeType = $_POST['storeType'];

if($currDB == 'mysql'){
	try{
		include('connect_MySQL.php');
		$sql = "DELETE FROM diveLog";
		$command = $PDO->prepare($sql);
		$command->execute();
	}
	catch(PDOException $e)
  {
	echo $sql . "<br>" . $e->getMessage();
  }

}
elseif($currDB == 'mongodb'){
	require 'C:\Users\Emil\vendor/autoload.php';
	$m = new MongoDB\Client("mongodb://localhost:27017");
	$db = $m->ExjobbDB->dive;
	$db->drop();
}

$file = $currDB . "-" . $storeType . "-" . $searches . ".txt";
```

```
$content = file_get_contents($file);


$content = "";
file_put_contents($file, $content);


echo $file;
?>
```

Appendix D - clearFile.php

```
<?php
$currDB = $_POST['dbType'];
$searches = $_POST['searches'];
$storeType = $_POST['storeType'];

$file = $currDB . "-" . $storeType . "-" . $searches . ".txt";

$content = file_get_contents($file);

$content = "";
file_put_contents($file, $content);

echo $file;
?>
```

Appendix E - writeTime.php

```
<?php
$time = $_POST['storeTime'];
$currDB = $_POST['dbType'];
$searches = $_POST['searches'];
$storeType = $_POST['storeType'];

$file = $currDB . "-" . $storeType . "-" . $searches . ".txt";
$content = file_get_contents($file);

$content .= $time . "\r\n";
file_put_contents($file, $content);

echo $time;
?>
```

Appendix F - connect_MySQL.php

```
<?php
$dbhost = "localhost";
$dbname = "MySQL_DB";
$dbuser = "root";
$dbpass = "root";
try{
   $PDO = new PDO('mysql:host='.$dbhost.'; dbname='.$dbname, $dbuser, $dbpass);
   //echo "Connected successfully";
   $PDO->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
   //echo "Connected successfully";
}
catch(PDOException $connectionError){
   //echo "Failed to connect to database: ".$connectionError->getMessage();
}

?>
```

Appendix G - insertMySQL.php

```
<?php

include_once('connect_MySQL.php');

$diveData = json_encode($_POST['diveData']);

$dives = json_decode($diveData);

$sightData = json_encode($_POST['sightData']);

$sightings = json_decode($sightData);


$insert = 1;

if($insert == 1){

   try{

   $sql = $PDO->prepare('INSERT INTO diveLog(id, diveDate, Location, Depth, diveTime, pressureStart, pressureEnd) Values(:id, :date, :loc, :depth, :time, :pStart, :pEnd)');

   $sql->bindParam(':id', $sightings[0]);

   $sql->bindParam(':date', $dives[0]);

   $sql->bindParam(':loc', $dives[1]);

   $sql->bindParam(':depth', $dives[2]);

   $sql->bindParam(':time', $dives[3]);

   $sql->bindParam(':pStart', $dives[4]);

   $sql->bindParam(':pEnd', $dives[5]);

   $sql->execute();

   }

catch(PDOException $e)

   {

   echo $sql . "<br>" . $e->getMessage();

   }


   $insert = 2;

}
```

```
if($insert == 2){


   $id = $sightings[0];
   for($i = 0; $i<count($sightings[1]); $i++){
         $animal = $sightings[1][$i];
      try{
         $sql = $PDO->prepare('INSERT INTO Sightings(sight, diveID)
            Values(:sight, :id)');
         $sql->bindParam(':sight', $animal);
         $sql->bindParam(':id', $id);
         $sql->execute();
      }
      catch(PDOException $e)
      {
      echo $sql . "<br>" . $e->getMessage();
      }


   }
}




?>
```

Appendix H - fetchMySQL.php

```
<?php
include('connect_MySQL.php');
$searchVal = $_POST['searchData'];
$searchType = $_POST['searchType'];

if($searchType == 'sight'){
	$sql = $PDO->prepare('SELECT diveID FROM Sightings WHERE sight =:searchVal');
	$sql->bindParam(':searchVal', $searchVal, PDO::PARAM_STR);
	$sql->execute();

	$resultID = array();
	$result = array();
	while($row = $sql->fetch())
	{
		array_push($resultID, $row);
	}

	for($i=0; $i<count($resultID); $i++){
		$id = $resultID[$i][0];

		$sql = $PDO->prepare('SELECT * FROM diveLog WHERE id =:searchVal');
		$sql->bindParam(':searchVal', $id);
		$sql->execute();

		while($row = $sql->fetch())
		{
			array_push($result, $row);
```

```
                }

                $sql = $PDO->prepare('SELECT * FROM Sightings WHERE diveID =:searchVal');
                $sql->bindParam(':searchVal', $id);
                $sql->execute();

                while($row = $sql->fetch())
                {
                        array_push($result, $row);
                }

        }
}
else if($searchType == 'location'){
        $sql = $PDO->prepare('SELECT * FROM diveLog WHERE Location =:searchVal');
        $sql->bindParam(':searchVal', $searchVal, PDO::PARAM_STR);
        $sql->execute();

        $result = array();
        while($row = $sql->fetch())
        {
                array_push($result, $row);
        }
}

echo json_encode($result);

?>
```

Appendix I - insertMongo.php

```
<?php
require 'C:\Users\Emil\vendor/autoload.php';
$m = new MongoDB\Client("mongodb://localhost:27017");

$db = $m->ExjobbDB->dive;

$diveData = json_encode($_POST['diveData']);
$dives = json_decode($diveData);
$sightData = json_encode($_POST['sightData']);
$sightings = json_decode($sightData);

        $document = array(
                "diveDate" => $dives[0],
                "Location" => $dives[1],
                "Depth" => $dives[2],
                "diveTime" => $dives[3],
                "pressureStart" => $dives[4],
                "pressureEnd" => $dives[5],
                "Sightings" => $sightings[1]
        );
        try{
                $db->insertOne($document);
        }
        catch (MongoCursorException $mce){
                echo $mce;
        }
    echo "Success";
```

?>

Appendix J - fetchMongo.php

```
<?php
require 'C:\Users\Emil\vendor/autoload.php';
$m = new MongoDB\Client("mongodb://localhost:27017");
$db = $m->ExjobbDB->dive;

$searchVal = $_POST['searchData'];
$searchType = $_POST['searchType'];

if($searchType == 'sight'){
	$searchQuery = array('Sightings' => $searchVal);
	$result = $db->find($searchQuery);
}
else if($searchType == 'location'){
	$searchQuery = array('Location' => $searchVal);
	$result = $db->find($searchQuery);
}

echo json_encode(iterator_to_array($result));
?>
```

Appendix K - actionHandler.js

```
var db = "";
var activeURL = "";
var searchType ='';
$(document).ready(function() {

        // Set mysql as standard database
        db = 'mysql';
        activeURL = 'MySQL.php';

 // Function for changing database
 $("#switchDB").click(function(){
        if(db == 'mysql'){
                db = 'mongodb';
                console.log(db);
        }
        else if(db == 'mongodb'){
                db = 'mysql';
                console.log(db);
        }
        alert('changed database to ' + db + '!');
 });

 $("#generateButton").click(function() {
  if(db == 'mysql'){
    activeURL = 'insertMySQL.php';
  }
  else if(db == 'mongodb'){
```

```
    activeURL = 'insertMongo.php';
  }
        prepareInsert();
});

// Functions for fetching data
$("#animalSearch").click(function() {
  if(db == 'mysql'){
    activeURL = 'fetchMySQL.php';
  }
  else if(db == 'mongodb'){
    activeURL = 'fetchMongo.php';
  }
  searchType = 'sight';
  prepareSearch();
});

$("#diveSearch").click(function() {
  if(db == 'mysql'){
    activeURL = 'fetchMySQL.php';
  }
  else if(db == 'mongodb'){
    activeURL = 'fetchMongo.php';
  }
  searchType = 'location';
  prepareSearch();
});
/*function createTable(myArray) {
```

```
	var table = document.getElementById("diveTable").getElementsByTagName('tbody')[0];
	var array = myArray;
	for (var i = 0; i < array.length; i++){
		var row = table.insertRow(i);
		var numValues = Object.keys(array[i]).length;
		var cells = [];
		for(var j = 0; j < numValues; j++){
			var value = Object.keys(array[i])[j];
			cells[j] = row.insertCell(j);
			cells[j].innerHTML = array[i][value];
		}
	}
 }*/

});
```

Appendix L - Index.html

```
<html>

<head>

	<link rel="stylesheet" type="text/css" href="style.css">


	<script
src="http://ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min.js"></script>

	<script language="javascript" type="text/javascript"
src="actionHandler.js"></script>

	<script language="javascript" type="text/javascript" src="fetchData.js"></script>

	<script language="javascript" type="text/javascript"
src="generateData.js"></script>

</head>

<body>

	<nav id="mainMenu">

		<ul>

			<li><a href="#">Search</a></li>

			<li><a href="#" id="switchDB">Switch Database</a></li>

		</ul>

	</nav>

	<div id="tableDiv">

		<table id="diveTable">

			<thead>

				<th>Date</th>

				<th>Location</th>

				<th>Depth</th>

				<th>Dive Time</th>

				<th>Sighting</th>

				<th>Pressure Start</th>

				<th>Pressure End</th>
```

```
                </thead>
                <tbody id="diveBody">
                </tbody>
            </table>
        </div>
</body>
        <div>
                Data:
            <select id="dataAmount">
                <option>100</option>
                <option>1000</option>
                <option>10000</option>
            </select>
            <input type="button" value="Generate Data" id="generateButton"/>
        </div>
        <div>
        <input type="button" value="Search Animals" id="animalSearch"/>
        <input type="button" value="Search Dives" id="diveSearch"/>
        <div id="result"></div>
</div>
</body>
</html>
```