Examensarbete

Kandidatexamen

Jämförelse av NoSQL-databas och SQL-baserad relationsdatabas

En förklarande studie för när NoSQL kan vara att föredra framför en relationsdatabas

Comparison of NoSQL database and SQL relational database

Författare: Jennifer Hedman & Mikael Holmberg
Handledare: Bo Sundgren
Samarbetspartner: Triona
Examinator: Pär Eriksson
Ämne/huvudområde: Informatik
Kurskod: IK2017
Poäng: 15 HP
Examinationsdatum: 2018-06-04

Vid Högskolan Dalarna har du möjlighet att publicera ditt examensarbete i fulltext i DiVA. Publiceringen sker Open Access, vilket innebär att arbetet blir fritt tillgängligt att läsa och ladda ned på nätet. Du ökar därmed spridningen och synligheten av ditt examensarbete.

Open Access är på väg att bli norm för att sprida vetenskaplig information på nätet. Högskolan Dalarna rekommenderar såväl forskare som studenter att publicera sina arbeten Open Access.

Jag/vi medger publicering i fulltext (fritt tillgänglig på nätet, Open Access):

Ja ☒ Nej ☐

Sammanfattning:

I och med den explosionsartade utvecklingen av den mobila världen, webbapplikationer och Big Data har nya krav inom lagringskapacitet och hastighet hos databassystem uppkommit. Den traditionella relationsdatabasen som länge dominerat har fått konkurrens då relationsdatabasen brister i hastighet och skalbarhet. NoSQL är ett samlingsnamn för databaser som inte bygger på den traditionella relationsmodellen. NoSQL-databaser är designade för att kunna expandera sin lagringskapacitet på ett enkelt sätt och samtidigt leverera hög prestanda. NoSQL-databaser har funnits i decennier men behovet för dem är relativt nytt. Vår samarbetspartner uttryckte ett behov om att få veta vilka skillnader som finns mellan NoSQL och den traditionella relationsdatabasen.

För att reda ut dessa skillnader har vi i detta arbete svarat på följande frågeställningar:

- När kan en NoSQL-databas vara att föredra framför en relationsdatabas?
- Vilka skillnader finns det i form av prestanda mellan databaserna?

För att svara på frågeställningen har en litteraturstudie gjorts tillsammans med experiment där vi testar vilka prestandaskillnader som finns mellan de valda databaserna. Prestandatester har utförts med testverktyget Yahoo Cloud Serving Benchmark för att verifiera eller falsifiera den ökade prestandan hos NoSQL-databaserna. Hypoteserna falsifierades i de båda NoSQL-databaserna. Resultatet visade att relationsdatabasen presterade bättre än de molnbaserade NoSQL-databaserna, men också att relationsdatabasens prestanda försämras vart eftersom belastningen ökade. Experimenten har kombinerats med litteraturstudien och svarar tillsammans på vår frågeställning. Slutsatsen är att det inte finns en databas som är bättre än någon annan utan allt handlar om de krav som ställs på den data som ska lagras. Utefter dessa krav kan en analys utföras för att dra slutsatser kring vilken typ av databas som är att föredra.

Nyckelord: Firebase Realtime database, CosmosDB, Relational database, comparison

Abstract:

With the explosive development of the mobile world, web applications and Big Data, new requirements for storage capacity and speed of database systems have arisen. The traditional relational database that has long dominated the marked has received competition because of its lack in speed and scalability. NoSQL is a collective name for databases that are not based on the traditional relational model. NoSQL databases are designed to easily expand their storage capacity while delivering high performance. NoSQL databases have been around for decades but the need for them is relatively new. Our partner expressed a desire to know what differences exist between NoSQL and the traditional relational database.

To clarify these differences, we have answered the following questions in this work:

- When can a NoSQL database be preferred to a relational database?
- What are the differences in database performance?

In order to answer these questions, a literature study has been conducted together with experiments where we test which performance differences exist between the selected databases. Performance tests have been performed with the benchmarking tool Yahoo Cloud Serving Benchmark, to verify or falsify the enhanced performance of the NoSQL databases. The hypotheses were falsified in both NoSQL databases. The results showed that the relational database performed better than the cloud-based NoSQL databases, but also that the relational database performance deteriorates when the load increased. The results of the experiments are combined with the literature study and together answer our questions. The conclusion is that no database performs better than another one, it is the requirements of the data to be stored. From these requirements, analyses can be made to draw conclusions about what kind of database is preferable.

Keywords: Firebase Realtime database, CosmosDB, Relational database, comparison

Förord

Vi vill tacka vår samarbetspartner Triona för allt stöd och hjälp med arbetet av rapporten, extra stort tack till David Hedman och Johan Larsson för idéer och synpunkter.
Vi vill även tacka vår handledare Bo Sundgren för hjälp och guidning genom arbetet.

Juni 2018
Mikael & Jennifer

Begreppslista

.NET	Är ett mjukvaru-ramverk som är utvecklat av Microsoft och körs huvudsakligen på Microsoft Windows (.NET, 2018)
API	Står för application programming interface och översätts till programmeringsgränssnitt på svenska. Regler för hur ett program ska interagera med andra mjukvaror, t ex andra program eller operativsystem (IDG, u.å.)
Datakrav	Regler eller direktiv för hur data ska lagras och hanteras. Kan vara olika för olika typer av data. Kan definieras av individer eller av lagar och regler (Data requirements, 2012)
DBMS	Är en förkortning för Database Management System och är ett program för uppläggning, underhåll och användning av databaser (IDG, u.å.)
Identifierare	Är ett namn på en funktion eller variabel i ett program (IDG, u.å.)
JSON	Står för Javascript Object Notation och är ett språk för överföring av datamängder på internet. Det är en inofficiell standard för NoSQL-databaser (IDG, u.å.)
Nästling	Någonting som är placerat inuti något annat, till exempel objekt A ligger i objekt B, som ligger inuti objekt C etc (IDG, u.å.)
SDK	Står för Software Development Kit och är utvecklingsverktyg för utvecklare. Består av program, handböcker och liknande från tillverkare som tillhandahåller information om produkter eller program för utomstående som till göra tillägg i dessa produkter/program (IDG, u.å.)

Innehåll

1 Inledning 1
1.1 Bakgrund 1
1.2 Problemformulering och frågeställning 2
1.3 Syfte 2
1.4 Mål 2
1.5 Avgränsning 2
1.6 Samarbetspartner 2
2 Teori 3
2.1 Relationsdatabas 3
2.2 NoSQL 3
2.2.1 Nyckel-värdeorienterad 4
2.2.2 Dokumentorienterad 5
2.2.3 Kolumnorienterad 5
2.2.4 Grafbaserad databas 6
2.3 De valda databaserna 7
2.3.1 MySQL 7
2.3.2 Firebase realtidsdatabas 7
2.3.3 Microsoft Azure Cosmos DB 8
2.4 Mätning av prestanda 8
2.4.1 Transaktion 8
2.4.2 Olika ramverk och standarder 8
2.5 Relaterat arbete 10
3 Metod 12
3.1 Forskningsprocess 12
3.2 Litteraturstudie 13
3.3 Forskningsstrategi 13
3.4 Datainsamling 14
3.5 Prestandamätning med YCSB 14
3.5.1 Konfiguration av databaser 14
3.5.2 Databasgränssnitt, belastningsuppgifter och parametrar 15
3.5.3 Inmatning av testdata och körning 15
3.5.4 Konfiguration 16

3.5.5 Analyering av prestandatester 16
3.6 Dataanalys 16
3.7 Metodkritik 17
4 Resultat av prestandatester 18
5 Analys 20
5.1 Analys av prestandatesterna 20
5.2 Framtagna datakrav 20
6 Diskussion och slutsats 23
6.1 Svar på frågeställning 23
6.2 Diskussion 23
6.3 Reflektion 24
6.4 Slutsats 24
6.5 Vidare forskning 24
Referenser 25

Figur 1 4
Figur 2 5
Figur 3 6
Figur 4 6
Figur 5 11
Figur 6 12
Figur 7 18
Figur 8 19

Tabell 1 10
Tabell 2 22

1 Inledning

I detta kapitel beskrivs bakgrunden till studien följt av en problemformulering med tillhörande frågeställningar samt syftet och målet för arbetet. Det tar även upp avgränsningar som har gjorts i arbetet och en kort beskrivning av vår samarbetspartner.

1.1 Bakgrund

Olika tekniker för att systematiskt lagra data har utvecklats sedan 1960-talet. Termen databas är benämningen för denna typ av samling strukturerade data. Hanteringen av en databas görs med ett *Database Management System* (DBMS). Under 1970-talet föreslog Edgar F. Codd (1970) relationsdatamodellen som modellerar entiteter och relationer för strukturerade data. Denna struktur har dominerat i alla större databehandlings-applikationer sedan 1980-talet (Database, 2018).

Utvecklingen av den mobila världen, webbapplikationer och Big Data har drivit fram nya krav inom lagringskapacitet och hastighet hos databassystem. Big Data är ett begrepp som används när vi talar om stora datamängder. Det handlar ofta om hundratals terabytes med ostrukturerade data, som behöver hanteras med hög hastighet. Några välkända applikationer som använder Big Data är Google Translate, Google Flu Trends och the PriceStats project (Sundgren, 2016). Den traditionella relationsdatabasen är inte designad för att skala effektivt och är därmed inte en optimal lösning för att hantera sådana datamängder. När lagringskapaciteten är nådd kan man antingen expandera den interna lagringen eller distribuera databasen till flera noder. Det betyder att man delar upp databasen över till exempel olika servrar eller datorer. Distribuering till flera noder i en relationsdatabas kan innebära ett stort och kostsamt arbete och resulterar ofta också i nedsatt prestanda. För att kunna hantera denna utveckling har ny teknik varit nödvändig och har resulterat i att så kallade NoSQL-databaser har fått ny luft (Sadalage & Fowler, 2012).

NoSQL står för *"not-only SQL"* och är ett samlingsnamn för många olika databaser. Databaserna som faller under namnet NoSQL har gemensamt att de inte lagrar data enligt relationer. Strukturen skiljer sig mellan olika NoSQL-databaser där de vanligaste använder antingen nyckel-värde-, dokument-, kolumn- eller en graf-modell som grund. Syftet med dessa databaser är att de ska kunna skala bra horisontellt och hantera stora mängder data och samtidigt bibehålla hög hastighet (Palovská, 2015). Firebase Realtime Database och Azure CosmosDB är två exempel på NoSQL databaser.

Tekniken är relativt ny och fortfarande under utveckling och nya NoSQL-databaser tillkommer på marknaden kontinuerligt. Relationsdatabasen är än idag dominant och används i de flesta applikationer (DB-Engines, 2018). Att välja det mest lämpade databassystemet kan vara svårt på grund av den stora mängden system som finns tillgängliga på marknaden idag. Det kan finnas fall där en relationsdatabas används men en NoSQL-databas skulle vara bättre anpassad för fallet. Vi upplever att en bidragande faktor till det är att många företag inte känner till hur NoSQL-databaser fungerar och används. Detta innebär en försämrad prestanda och därmed bortkastade resurser.

1.2 Problemformulering och frågeställning

NoSQL databasteknik kom till för att svara mot de framtida kraven inom datalagring i och med den snabba tillväxten av internet, mobila applikationer och webbapplikationer. Vår samarbetspartner har uttryckt att det finns för lite kunskap om NoSQL inom IT-företag. Detta bidrar till att det kan vara svårt att veta vilken typ av databas som är att föredra. Vi vill med detta arbete undersöka och svara på följande frågeställningar:

- När kan en NoSQL-databas vara att föredra framför en relationsdatabas?
- Vilka skillnader finns det i form av prestanda mellan databaserna?

1.3 Syfte

Syftet med arbetet är att förklara när en NoSQL-databas är att föredra framför en relationsdatabas. För att svara på den frågan görs en grundlig litteraturstudie tillsammans med explorativa experiment vars syfte är att verifiera de hypoteser vi införskaffat genom vår litteraturstudie. Teori och experiment kommer tillsammans att förklara när, och under vilka datakrav, en NoSQL-databas är att föredra framför den traditionella relationsdatabasen.

1.4 Mål

Vår samarbetspartner Triona har i dagsläget bristande erfarenhet av NoSQL-databaser. Vårt arbete ska ge dem en grund i hur NoSQL-databaser kan utnyttjas genom att förklara skillnaderna mellan de olika typerna av databaser. Vi vill även ge exempel på vilka datakrav som är viktiga att tänka på vid val av databas. Samt även genomföra prestandatester och en jämförelse mellan två NoSQL-databaser och en traditionell relationsdatabas för att ge dem en bättre förståelse för de olika typerna av databaser.

1.5 Avgränsning

Det finns många olika databaser som går under namnet NoSQL. I denna undersökning kommer vi att jämföra och utföra tester på Firebase NoSQL realtidsdatabas, Azure Cosmos DB NoSQL realtidsdatabas och relationsdatabasen MySQL.

1.6 Samarbetspartner

Triona är ett IT-företag med stor kompetens inom transportinfrastruktur, trafik, transporter, skogsindustri och energi-/fordonsindustri. De arbetar med både systemutveckling och systemförvaltning av egna produkter samt även förvaltning av andra system. De vill skapa effektiva lösningar som ska bidra till att deras kunder når sina mål och erbjuder både egna produkter och konsulttjänster. Triona finns i Sverige, Finland och Norge och har ca 140 medarbetare (Triona, u.å.).

2 Teori

Teorikapitlet tar upp de relevanta begrepp som är viktiga och centrala för studien. Det tar även upp tidigare forskning och relevanta arbeten som framkommit under litteraturstudien.

2.1 Relationsdatabas

Relationsdatabasen tillkom under 1970-talet när Codd (1970) föreslog användning av en relationsmodell för att svara mot den motstridiga datastruktur som då användes där data organiserades och hanterades genom sparade sökvägar. Relationsdatabasen blev industrialiserad och blev en standard runt 1980-talet. Relationsdatabasen är dominant på marknaden än idag (DB-Engines, 2018).

Data i en relationsdatabas är strukturerad, där data lagras i tabeller representerande entiteter tillsammans med dess relationer och attribut. En instans av en entitet är en rad i dess tabell. Relationer mellan tabeller definieras av att tabellerna har ett gemensamt attribut. Databasens tabeller är fixerade där alla rader i en tabell har samma antal kolumner och attribut (Brander & Dakermandji, 2016). Förfrågningar och modifiering av data i relationsdatabaser sker vanligen med *Structured Query Language* (SQL) som är ett standardiserat språk inom databashantering och tillåter avancerade sökfrågor (Database, 2018).

Relationsdatabaser uppfyller i regel också ACID, som står för *Atomicity, Consistency, Isolation och Durability*. Atomicity innebär att om en transaktion har påbörjats men avbryts på grund av fel mitt i transaktionen så ska databasen kunna rulla tillbaka alla påbörjade förändringar så att systemet förblir oförändrat. Consistency innebär att databasen bara accepterar data som uppfyller definierade regler och constraints och därmed ser till att systemet är konsistent i sin helhet. Isolation är en kontroll för hur data ska vara tillgängligt för flera anrop som sker samtidigt. Durability är en garanti för att utförda transaktioner ska lagras i sekundärminne och ej gå förlorade vid strömförlust. ACID är en viktig aspekt vid val av databas (ACID, 2018). ACID-garantin kan däremot innebära att prestandan blir negativt påverkad. Med consistency innebär det att transaktioner kan behöva stå i kö ifall fler transaktioner sker samtidigt (Al Hinai, 2016).

Prestandan kan även bli negativt påverkad när tabellerna i en relationsdatabas blir för massiva, vilket kan medföra att utsökningar i databasen tar längre tid än innan. Genom att dela upp tabellerna över olika noder kringgår man detta problem och kan snabbare göra utsökningar när det inte krävs att hela databasen analyseras. Att dela upp eller fördela en relationsdatabas på ett sådant sätt kallas på engelska för "sharding" eller "partitioning" och översätts på svenska till att skala horisontellt. Detta bidrar inte bara till en ökad prestanda utan ökar även tillgänglighet, tillförlitlighet och garanterar feltolerans (Al Hinai, 2016). Relationsdatabaser designades inte med horisontell skalbarhet som ett krav och det är inte alltid lätt att dela upp en databas på detta sätt och det kräver stor planering. En rekommendation är att försöka lösa problemet på andra sätt innan ett arbete för att skala en relationsdatabas inleds. Därmed finns det en efterfrågan att få detta att fungera på ett smidigare sätt (Corbellini, Mateos, Zunino, Godoy, & Schiaffino, 2017).

2.2 NoSQL

Många databaser har utvecklats utöver relationsdatabasen. Carlo Strozzi skapade en databas som han namngav NoSQL som inte använde ett SQL gränssnitt. På senare tid har en rad nya databaser tillkommit som tillsammans landat under namnet NoSQL, även fast de inte på något

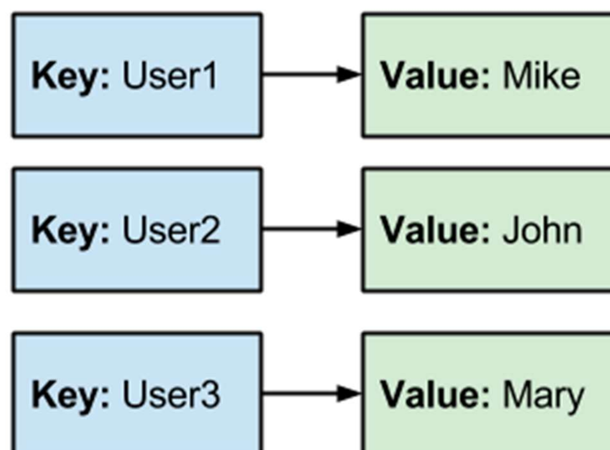
sätt bygger på Strozzis teknik. Generellt så har dessa databaser ingen fixerad datastruktur som relationsdatabasen. Relationsdatabasen kräver i förtid att tabeller och kolumner, inklusive relationsregler, är designade i förtid. NoSQL-databaser har inte denna strukturerade design, utan kan skapa 'kolumner' under körtid. Detta betyder att ett objekt i databasen kan ha varierande antal egenskaper, och det är då upp till utvecklarna att ta hand om logiken. Eventuella relations-samband som önskas programmeras då på applikationsnivå. NoSQL skalar bra horisontellt, men uppfyller i regel inte ACID, som relationsdatabasen (Lith & Mattsson 2010).

En databas som ofta förekommer i litteraturen är Google BigTable. Motiveringen för utvecklingen av Google BigTable var att skapa en databas med effektiv horisontell skalning och hög prestanda. Googles (u.å.) beskrivning av BigTable lyder: *"BigTable is designed to handle massive workloads at* ***consistent low latency and high throughput****, so it's a great choice for both operational and analytical applications, including* ***IoT, user analytics, and financial data analysis****"*. BigTable har inspirerat många av dagens NoSQL-databaser (Corbellini et. al., 2017).

De nya teknikerna kategoriseras baserat på den underliggande datastrukturen som databasen använder. De fyra mest omtalade strukturerna är nyckel-värdeorienterad (key-value store), dokumentorienterad (document store), kolumnorienterad (column-family store) och grafbaserad databas (graph database). Dessa databaser kan i regel skala effektivt över flera noder och samtidigt bibehålla tillgänglighet (Palovská, 2015).

2.2.1 Nyckel-värdeorienterad

Den nyckel-värdeorienterade databasen lagrar data enligt nycklar och värden, likt ett stort JSON-träd. En nyckel är en identifierare och används för hämtning av värdet kopplat till den identifieraren. Data lagras ostrukturerad där en rad, om vi depikterar data i tabeller, kan ha varierande mängder kolumner. Värdet för en identifierare består av ett objekt, tillsammans med dess attribut där attribut kan bestå av textsträngar och komplexa listor. (Baron, 2016). Nyckel-värde-databasens struktur är den simplaste formen av NoSQL och är lämplig där stora mängder data ska lagras som kräver effektiv horisontell skalning. Dessa databaser kan, med horisontell skalning, lagra petabytes av data och samtidigt vara tillgängligt för miljontals samtidiga anrop (Corbellini et al., 2017).

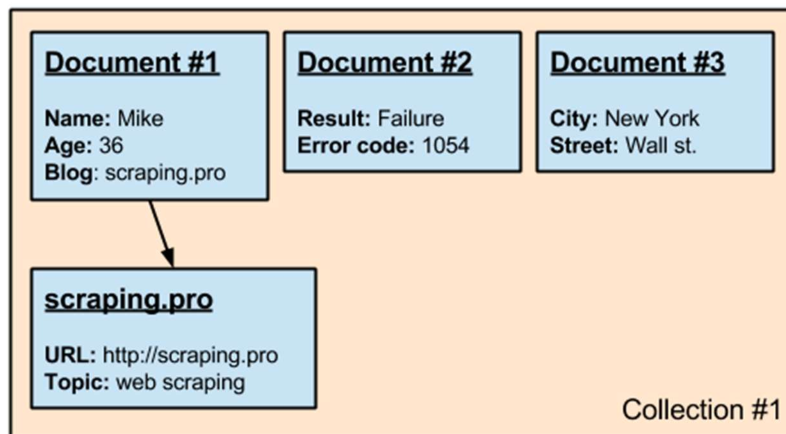


Figur 1

Nyckel-värdeorienterad struktur (Shilov, 2016)

2.2.2 Dokumentorienterad

En dokumentorienterad databas är semistrukturerad där data lagras i dokument. Dokument grupperas i dokumentsamlingar. Varje dokument i en samling har en unik identifierare och innehåller en egen nyckel-värde kedja. I den dokumentorienterade databasen definieras regler för vilken struktur och vilka datatyper som ska kunna lagras till skillnad mot den nyckel-värdeorienterade databasen där data är lagrat helt ostrukturerat (Sadalage & Fowler, 2012). Dokumentets fält kan ha numeriska och algebraiska datatyper, och även listor. Skillnaden mellan nyckel-värdeorienterade databasen och den dokumentorienterade är ibland otydlig, men den generella skillnaden är att nyckel-värde-databasen endast hämtar data baserat på tillhörande identifierare, medans den dokumentorienterade databasen kan göra mer flexibla förfrågningar baserat på den definierade data-strukturen (Sadalage & Fowler, 2012).

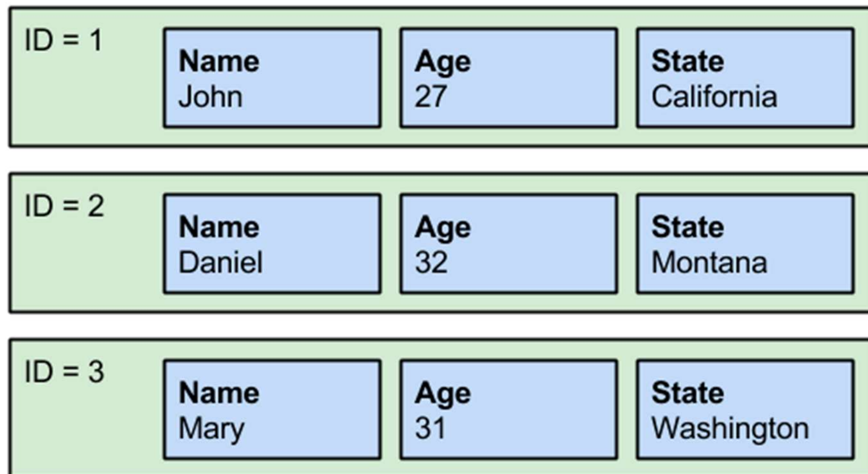


Figur 2

Dokumentorienterad struktur (Shilov, 2016)

2.2.3 Kolumnorienterad

I den kolumnorienterade *Column-family* databasen, även kallad *Wide Column,* lagras data i ett kluster av kolumner. Datamodellen kan se annorlunda ut i olika databaser men grundprincipen är densamma. De flesta kolumnorienterade databaserna är inspirerade av Google Big Table som var en av de första databaserna inom den här kategorin (Corbellini et al., 2017). Data lagras i rader och kolumner och en kolumn är nyckel-värdebaserad. Exempel på en kolumn kan vara 'namn' och dess värde 'Anna'. Kolumner som förväntas höra ihop grupperas i kolumnfamiljer. Exempel på en kolumnfamilj kan vara 'adress', och dess kolumner kan då vara gata, postnummer, stad etc. Varje rad innehåller data, som är grupperade i flera kolumnfamiljer. Varje kolumnfamilj innehåller en samling av kolumner (Sadalage & Fowler, 2012). Modellen kan ses som en *"Map"* fylld med *maps* (*map*<nyckel, *map*<nyckel, värde>). Data kan hämtas i hela rader såväl som specifika kolumner.

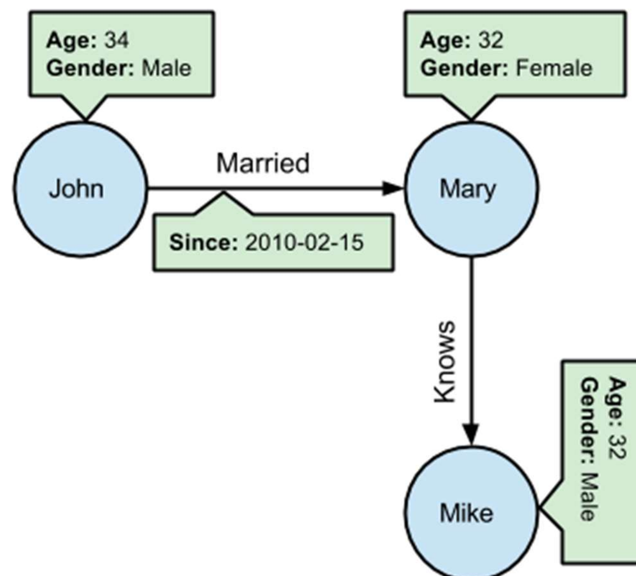


Figur 3

Kolumnorienterad struktur (Shilov, 2016)

2.2.4 Grafbaserad databas

Den grafbaserade strukturen skiljer sig från de övriga. Den enda likheten de egentligen har är att de inte använder sig av relationsmodellen. Databasens syfte är möjliggöra analyser av data där dess relationer är av intresse, som i exempelvis sociala nätverk. I sociala nätverk skulle vi med en grafdatabas kunna ställa frågor som "Hämta filmer med skådespelare som någon av mina vänner gillar". Det intressanta i databasen är relationerna mellan data, och inte data i sig. En fråga till grafdatabasen börjar vanligtvis med att utgå ifrån ett id, och därefter navigera sig igenom de olika relationerna och plocka ut sökta data. Databasen bedrivs ofta på en kärna och är inte designad för att skala (Sadalage & Fowler, 2012).



Figur 4

Grafbaserad struktur (Shilov, 2016)

2.3 De valda databaserna

Nedan följer beskrivningar av de valda databaserna som vårt arbete kommer att göra jämförelser mellan. Databaserna valdes dels utifrån önskemål från vår samarbetspartner samt även för att vi inte har hittat forskning som har innefattat någon av de två NoSQL-databaserna.

2.3.1 MySQL

Relationsdatabasen vi kommer att använda oss av i testerna är MySQL och är utvecklad av ett svenskt företag som heter MySQL AB. Idag ägs MySQL av Oracle Corporation. Databasen lanserades den 23 maj 1995 och har sedan dess släppts i flera versioner, den senaste släpptes den 19 april i år. MySQL är en populär databashanterare och kan köras på en rad olika plattformar. Några exempel är Linux, macOS, Microsoft Windows, OpenSolaris och Symbian (MySQL, 2018). Databasen har öppen källkod, vilket betyder att källkoden inte ägs av någon utan är tillgänglig för alla att använda och modifiera. Det betyder även att användare inte behöver betala en licensavgift för att använda produkterna, och kan anpassa koden efter behov (Open-source software, 2018).

MySQL använder sig av språket "Structured Query Language" för att hantera operationer mot databasen. Det är det vanligaste språket att använda sig av för att hantera data i databaser och har utvecklats sen och använts sedan 1986. MySQL använder sig av en relationsmodell för lagring som innefattar objekt som till exempel databas, tabeller, vyer, rader och kolumner. Användaren anger vilka relationer som ska finnas mellan de olika tabellerna samt även vilka regler som ska finnas för hur data struktureras i databasen. Exempel på dessa är till exempel ett-till-många-förhållande eller att en viss kolumn alltid måste innehålla ett unikt värde. Genom att ha en väldesignad datamodell ser databasen till att det inte förekommer ologisk, duplicerad, utdaterad eller ofullständig data (Oracle, u.å.).

2.3.2 Firebase realtidsdatabas

NoSQL-databasen Firebase som är en molnbaserad databas utvecklad av Google. Firebase lanserades år 2012 (Firebase, 2018) och är en nyckel-värdeorienterad databas. Data sparas ner i JSON-objekt och kan ses som ett enda stort JSON-träd med nycklar och värden. Det går att välja att ange egna nycklar i form av ID-nummer eller semantiska namn men Firebase kan även generera dessa nycklar automatiskt. För att på ett simpelt sätt kunna hämta data krävs det planering för hur data ska sparas och struktureras i databasen. Firebase accepterar nästling ner till 32 nivåer, men det är praktiskt bättre att försöka undvika alltför många nivåer av nästlad data. Dels för att hämtningar ska ske på ett smidigare sätt och dels för att enklare hantera säkerhetsrisker. När någon ges behörighet att skriva och läsa från ett objekt i databasen ges samma behörighet till alla nästlade eller underliggande objekt. Detta kan medföra säkerhetsrisker om data inte planeras på ett strukturerat sätt (Google, 2018).

Som standard så hämtas data på djupet. Från den punkt i JSON-trädet som ett visst objekt hämtas så kommer alla underliggande objekt även att följa med vid hämtningen. För att hantera vilka objekt som ska hämtas kan man välja att sortera eller filtrera bland objekten. Det går dock inte att både filtrera och sortera samtidigt i samma förfrågan (Google, 2018). Vill man använda mer komplexa sökningar så hanteras detta på applikationslagret och istället för på databas-nivån, då sådana operationer kan försämra prestandan (Corbellini et al., 2017).

Firebase stödjer integration med en rad olika applikationer uppbyggda av till exempel Android, iOS, JavaScript, Java, Objective-C, swift och Node.js. Genom ett REST API ges även integration

med några olika ramverk som AngularJS, React, Ember.js och Backbone.js. REST API:et använder sig av ett ytterligare API, Server-sent Events, som är en teknik för webbläsare. Webbläsare kan automatiskt hämta uppdateringar och pushnotiser från databasen via en HTTP-anslutning (Firebase, 2018).

2.3.3 Microsoft Azure Cosmos DB

Databasen är Cosmos DB som är utvecklad av Microsoft och lanserades under våren 2017. Databasen stödjer flera olika typer av datamodeller, vilket betyder att den kan användas som dokument-, graf-, nyckelvärde-, tabell- och kolumn-orienterad databas beroende på användarens syfte och ändamål. Denna typ av databas kan även kallas för multi-modell då den stödjer olika modell-strukturer. Cosmos DB låter användaren distribuera sin data över hela världen där Azure har datacenter och även sprida replikor av sin data för en snabb åtkomst av databasen på platser runt om i världen. Detta medför att Cosmos garanterat en hög latens vid både läs- och skrivoperationer (Microsoft, 2018).

Cosmos indexerar per automatik alla fält i databasen som grundinställning men det går att ställ in regler för hur indexering ska ske. Andra funktioner som går att reglera är hastighet, utrymme och följdriktighets-nivåer. Hastighet mäts i förfrågnings-enheter per sekund och ställs in av användaren och detsamma gäller utrymmet. Cosmos kan dynamiskt reglera dessa två parametrar vid tillfällen då det till exempel finns en högre belastning på databasen. Detta göra för att upprätthålla garantierna för läs- och skrivoperationerna. Kostnaden för databasen baseras även på hastigheten och utrymmet. Den sista funktionen att ställa in är följdriktigheten på databasen. Det finns fyra olika nivåer: strong, bounded-staleness, session och eventual. Desto högre nivå, desto mer "kostar" förfrågnings-enheterna per sekund. Den förinställda nivån för Cosmos är session. Databasen kan köras med ACID-principerna vid alla de fyra nivåerna, med förutsättning att operationerna körs från lagrade procedurer. Cosmos stödjer även en rad olika SDK:s och API:er för att kunna ge användaren ett brett utbud över hur data ska hanteras och hämtas. Några exempel är: .NET, Node.js (JavaScript), Java, Python, SQL API, MongoDB API och Cassandra API (Cosmos DB, 2018).

2.4 Mätning av prestanda

2.4.1 Transaktion

En transaktion är definierad som en atomisk händelse som tar databasen från ett fungerande tillstånd till ett annat (Yao & Hevner, 1984). En transaktion kan bestå av sökfrågor med och utan relationer, samt Data Manipulation Language (DDL) vilket berör transaktioner där data modifieras i databasen. DDL kan vara inmatning av data, uppdatering av lagrad data, läsning av data, och borttagning av data.

2.4.2 Olika ramverk och standarder

En mätning av prestanda kan göras på olika sätt beroende på vad för område som ska undersökas. Prestandamätning utförs vanligen för att antingen finna den optimala konfigurationen för ett tänkt databassystem, eller för att jämföra flera databaser för att hitta den bäst lämpade databaslösningen för ett system (Yao & Hevner, 1984). Flera verktyg finns tillgängliga för att utföra dessa mätningar. TPC (Transaction Processing Performance Council) har definierat flera standarder för olika typer av prestandamätningar.

HÖGSKOLAN
DALARNA

TPC-C förekommer ofta i litteraturen och är lämplig för mätning OLTP (On-Line Transaction Processing) system. TPC-C är en vidareutveckling av TPC-A och simulerar ett kundorder-system. Systemet består av nio tabeller där fem olika typer av samtidiga transaktioner som utförs i varierande kombination (TPC, u.å. a). TPC-C mäter hur många transaktioner per sekund som systemet hanterar och ett estimerat pris per transaktion.

TPC-E är en annan variant av OLTP prestandamätning. Detta system är mer komplext än TPC-C, med 33 tabeller. Systemet simulerar en mäklarfirma där användare skickar förfrågningar om konton och marknadsundersökningar. Det simulerade mäklarsystemet växlar uppgifter med en simulerad finansmarknad för genomförande av ordrar och uppdaterar sedan relevant kontoinformation. Antalet användare som simuleras kan styras av testaren för att simulera olika storlekar på ett mäklarbolag. Mätningen resulterar i antalet transaktioner per sekund som systemet lyckas hantera (TPC, u.å. b).

TPC-H är en standard för prestandamätning i ett OLAP (Online Analytical Processing) system, för exempelvis decision-support system och business intelligence. Systemet simulerar ett beslutsstödsystem där en stor mängd komplex data undersöks för att kunna svara på kritiska affärsfrågor. Transaktionerna består av samtidiga komplexa data mining (eller datautvinnings) förfrågor. Mätningen resulterar i TPC-H Composite Query-per-Hour Performance Metric, som beräknas utifrån olika aspekter av de utförda transaktionerna. Aspekterna omfattar databasens storlek som förfrågan berör, behandlingstider för transaktioner från en användare, och transaktioner per sekund för flera samtidiga användare (TPC, u.å. c).

Yahoo Cloud Serving Benchmark (YCSB) är ett av de mest använda open-source verktygen för att mäta och jämföra prestanda hos databaser (Barata, & Bernardino, 2016). Verktyget utvecklades för att kunna utföra mätningar på olika typer av databaser såsom NoSQL-databaser. Många av de standarder som definierats av TCP är baserade på relationsmodellen vilket inte alltid är möjligt att implementera i en NoSQL-databas. En nyckelfaktor med YCSB är att det enkelt kan byggas ut för att ge stöd åt nya databaser. Transaktionerna som genereras av YCSB efterliknar TPC-H där systemet som simuleras efterliknar ett OLAP-system. YCSB innehåller en datagenereringsklient och ett antal olika standard workloads eller belastningsuppgifter (Cooper, Silberstein, Tam, Ramakrishnan, & Sears, 2010). Användaren anger mängder av transaktioner som ska utföras och hur många samtidiga användare som ska simuleras. Belastningsuppgifterna består av olika proportioner av de samtidiga transaktioner som ska utföras och visas i tabellen nedanför (Github, 2018).

Beteckning	**Typ**	**Skriv**	**Läs**	**Uppdatering**	**Sökning**	**Borttagning**
a		0%	50%	50%	0%	0%
b		0%	95%	5%	0%	0%
c		0%	100%	0%	0%	0%
d		5%	95%	0%	0%	0%

e	5%	0%	0%	95%	0%
f	25%	50%	25%	0%	0%

Tabell 1

Belastningsuppgifter (Github, 2018)

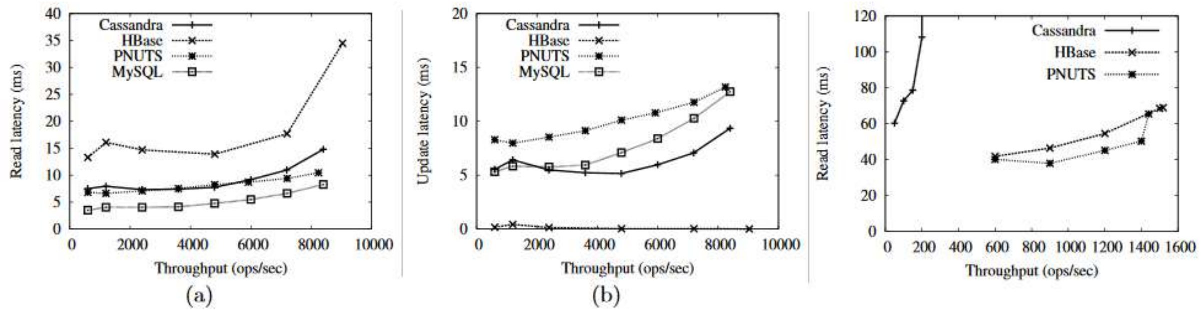
2.5 Relaterat arbete

Under litteraturstudien hittade vi tidigare arbeten som har utfört liknande prestanda-jämförelser mellan relationsdatabaser och NoSQL-databaser. Inget arbete som vi har stött på har använt sig av någon av de NoSQL-databaser vi undersöker i vårt arbete. Tidigare jämförelser tar upp att NoSQL-databaser utför skrivoperationer med än lägre svarstid jämfört med relationsdatabaser. NoSQL-databaserna presterade med lägre svarstider överlag även vid olika typer av operationer, till exempel uppdaterings- eller läsoperationer. Storleken på data som användes var av varierande mängd och visade liknande resultat oavsett mängden data (Pereira, Ourique de Morais, & Pignaton de Freitas, 2018). Det fanns vissa avvikelser vid skrivoperationer av låg mängd data, då NoSQL-databaser i vissa fall presterade med högre svarstider jämfört med relationsdatabaserna. Vid större mängder data var NoSQL-databaserna snabbare (Boicea, Radulescu, & Agapin, 2012). NoSQL-databaserna är designade för att vara tillgängliga samt att kunna hämta data med hög hastighet, vilket stödjer de arbeten med tillhörande resultat vi hittat under litteraturstudien. De fungerar även bra ihop med analysprogram, där stora mängder data behöver hämtas och analyseras. Det är därför big data spelar en stor roll när man talar om NoSQL-databaser då de är utvecklade för att hantera dessa stora datamängder (Khazaei et al., 2016).

Ramakrishnan et al. (2013) har använt YCSB för att jämföra vilka prestandaskillnader som finns mellan Cassandra, Hbase och MongoDB. Resultatet visade att Cassandra åstadkom fler transaktioner per sekund jämfört med de andra två i alla belastningsuppgifter utom en. MongoDB presterade i klass med Cassandra vid läsoperationer men betydligt långsammare i övriga belastningsuppgifter.

En annan prestandamätning som utförts med YCSB visade att Microsoft SQL Server uppnådde högre antal transaktioner per sekund än MongoDB vid uppdateringsoperationer och läsoperationer. Testerna utfördes med åtta datorer, 100 klienttrådar per maskin, mot en databas med 640 miljoner rader data (Floratou, Teletia, DeWitt, Patel, & Zhang, 2012).

Cooper et al. (2010) gjorde en prestandamätning av Cassandra, HBase, PNUTS, och MySQL. Resultatet visade att MySQL och PNUTS hade lägre svarstider än Cassandra och HBase. Cassandra och HBase hade däremot lägre svarstider när det gällde uppdateringsoperationer. Testet utfördes med upp till 500 klienttrådar och en databasstorlek på 120 miljoner rader.



Figur 5

Resultat av prestandamätning (Nelubin & Engber 2013)

En jämförelse av VoltDB, HBase, Redis, Voldemort och Cassandra med testverktyget YCSB visade även den en högre prestanda hos Cassandra. Svarstiden var däremot högre hos Cassandra än övriga databaser (Nelubin & Engber, 2013).

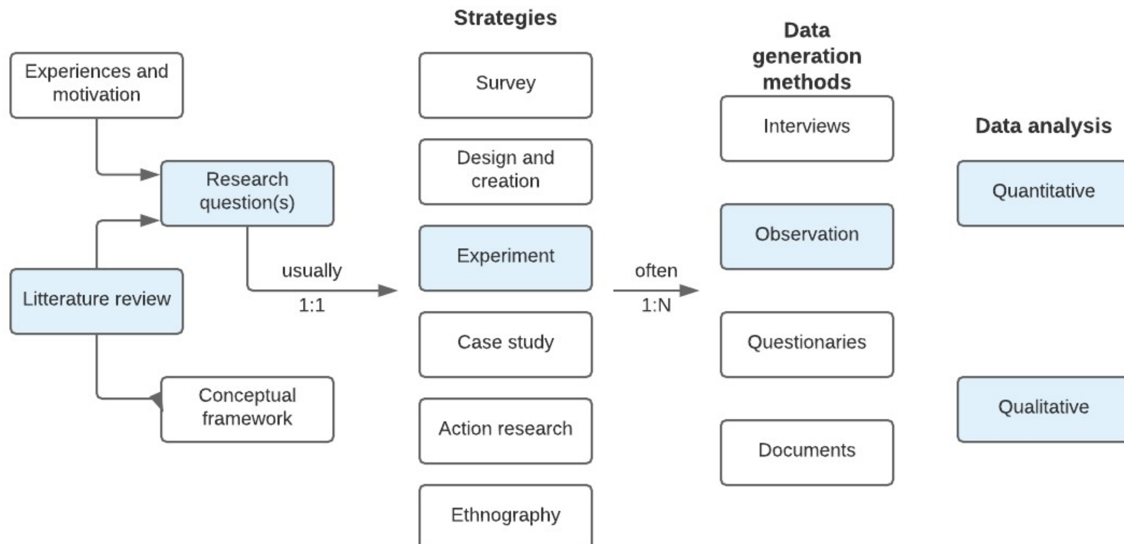
Vid mer komplexa operationer mot databaserna, som till exempel nästlade sökningar, presterar relationsdatabaser bättre. NoSQL-databaser stödjer i regel inte operationer som innefattar till exempel "join", "limit" eller "where" (Boicea et al., 2012). Databaser som faller under NoSQL-kategorin undviker ofta att använda "joins" och liknande på databas-lagret, då sådana operationer kan medföra sämre prestanda. Selektering av data som ska visas sker istället vid applikationslagret i en applikation. Alternativt struktureras data redan i datamodellen på ett sådant sätt att data hämtas och visas på ett önskvärt sätt (Corbellini et al., 2017).

3 Metod

Här beskrivs metoden vi använt för att gå tillväga med arbetet. Genomförande av litteraturstudie, val av forskningsstrategi, datainsamlingsmetoder samt dataanalysmetoder presenteras. Förklaring av systemet vi använt vid testning, testfallen och testmetoden av databaserna beskrivs och motiveras.

3.1 Forskningsprocess

Forskningsprocessen började med en omfattande litteraturstudie för att få en djupare förståelse inom området. Vi använde även litteraturstudien för att söka efter metoder över hur andra har genomfört och jämfört prestandatester av databaser tidigare. En djupare beskrivning av litteraturstudien finns under kapitel 3.2. För att kunna besvara en av våra frågeställningar samlade vi in data genom observationer av olika tester mot databaserna. Testerna gick ut på att mäta de olika databasernas genomströmningskapacitet. Datainsamlingsmetoderna finns beskrivna djupgående i kapitel 3.4. Efter datainsamlingen analyserades vår insamlade data både kvantitativt och kvalitativt. Testresultaten analyserades genom kvantitativa metoder och databasernas resultat jämfördes med varandra för att besvara frågeställningen rörande prestandaskillnader. För att besvara den andra frågeställningen gjordes en kvalitativ analys av den övriga data som framkommit under litteraturstudien. Detta finns vidare beskrivet under kapitel 3.6. Resultatet av experimentet resulterade i kunskap inom området. Nedan följer en modell över vår forskningsprocess och de delar vi har använt oss av för att skapa en metod för arbetet.



Figur 6

Forskningsprocess (Oates, 2006)

3.2 Litteraturstudie

En litteraturstudie gjordes för att undersöka vilken tidigare forskning som redan finns inom ämnet och för att öka vår förståelse inom området. Den användes även för att reda ut vilka skillnader det finns mellan de olika databaserna. Utförandet har gjorts enligt en sjustegsprocess av Oates (2006). Litteraturstudien avser att samla in och presentera tidigare forskningsresultat med syfte att ge stöd till arbetet samt att påvisa att vi bidragit med ny kunskap. Första steget är att söka efter litteraturen. Ämnet vi valt är i dagsläget relativt färskt, vilket innebär svårigheter att hitta litteratur i tryckt form i bibliotek. Vi har därför sökt all litteratur i sökmotorer på internet och gärna nyare källor då ämnet hela tiden utvecklas och äldre källor snabbt kan bli inaktuella. Sökmotorerna vi har använt oss av är Google Scholar och Summon. Sökorden som använts är *NoSQL database, realtime database, relational database, RDBMS, big data, database performance, database comparison*, kombinerade på olika sätt. Sökningarna är gjorda både på svenska och engelska. Utöver våra egna sökningar har vi använt oss av *backwards-search,* vilket innebär att forskaren följer referenser från en hittad källa till den refererade källan (Oates, 2006).

Primärt har vi sökt efter vetenskapliga artiklar som är peer reviewed, det vill säga att artikeln är granskad av andra forskare inom ämnet, men vi har också använt källor av mindre kaliber då ämnet är relativt nytt. Efter att ha gjort en snabb värdering om de böcker och artiklar vi hittat kan bidra till vårt arbete har vi gjort en kritisk granskning av källorna. Vi har sökt efter information om författarna där vi tittat på vilken bakgrund de har och om de är framstående inom området. För tidskrifter har vi också tittat på hur länge tidskriften har funnits. Tidskrifter som funnits en längre tid är mer etablerade och kan förväntas ha en högre standard (Oates, 2006).

3.3 Forskningsstrategi

Inom forskning använder man sig av olika strategier för att besvara sin frågeställning. Oates (2006) tar upp sex olika forskningsstrategier: undersökning, design och skapande, experiment, fallstudie, aktionsforskning och etnografi. I vårt arbete har vi använt oss av experiment som strategi för att besvara delar av vår frågeställning. Ett experiment är en strategi där forskaren verifierar eller falsifierar framställda hypoteser. Vårt experiment är ett *sant experiment* där experimentet är konstgjort och utförs i en kontrollerad miljö, till skillnad mot quasi-experiment och *okontrollerade test* som inriktar sig på sociala aspekter. Hypoteserna framställs genom att forskaren gör ett antagande, och verifierar eller falsifierar hypotesen baserat på observationer av resultat från de experiment som utförs (Oates, 2006).

Baserat på den litteraturstudie som utförts har vi definierat följande hypoteser:
- Inmatning av data utförs snabbare i NoSQL-databaserna.
- Hämtning av data utförs snabbare i NoSQL-databaserna.

Ett experiment går inte bara ut på att observera resultat. Forskaren ska också ta reda på varför resultatet blev som det blev och beskriva detta baserat på den teori som grundade hypoteserna (Oates, 2006). Vi kommer därmed att utföra våra testfall, observera resultaten och därefter återkoppla till de bakomliggande teoretiska grunderna för att förklara utfallet.

Experimentet som utförs innebär att forskaren observerar hur en beroende variabel påverkas när forskaren förändrar en oberoende variabel (Oates, 2006). Den beroende variabeln i vårt fall är själva databasernas prestanda. Den oberoende variabeln är variationen i datamängder som testerna kommer behandla. Vi kommer alltså att observera svarstiderna från databaserna i flera

omgångar, där en omgång är en viss mängd data som behandlas. Experimentet utförs för att verifiera delar av den teoretiska grunden och svara på vilka skillnader som finns i form av prestanda och därmed stödja svaret till när en NoSQL-databas bör tillämpas.

3.4 Datainsamling

Det finns två olika typer av data vid en datainsamling: primärdata och sekundärdata. Primärdata är data som forskaren producerar och sammanställer själv. Exempel på metoder som genererar primärdata är intervjuer, enkäter och observationer. Sekundärdata är data som samlas in av forskaren och hit räknas bland annat litteraturstudier och presentationer vid föreläsningar eller konferenser. Vi har använt oss av litteraturstudier och observationer som datainsamlingsmetoder. En litteraturstudie ger tillgång till mycket information på kort tid men då det inte alltid framgår hur informationen man tar del av har skapats är det viktigt att ifrågasätta syftet för varför den skapats (Björklund & Paulsson, 2012). Litteraturstudien genomfördes för att ta reda på mer inom området samt vilka skillnader som finns mellan de olika databaserna.

En del av vår datainsamling är i form av observationer. Observation som datainsamlingsmetod kan användas när man vill undersöka hur exempelvis människor betér sig i en miljö där de inte tror att de blir utvärderade, hur ofta något inträffar under en viss tid eller hur lång tid något tar för en visst utförande. Observationer kan användas till många olika forskningsstrategier och kan göras enligt en systematisk observationsmodell eller en deltagarorienterad observation där forskaren deltar i situationen som studeras (Oates, 2006). Den här forskningen sker med en systematisk observationsmodell eftersom vi observerar svarstider mot databaser för att jämföra prestandaskillnader mellan databaserna. I den systematiska observationsmodellen definierar vi i förtid vad, hur och när ett test ska utföras, och observerar sedan resultaten.

3.5 Prestandamätning med YCSB

För prestandamätning har vi använt det verktyget Yahoo Cloud Serving Benchmark. Användningen av verktyget bygger på sex steg: konfiguration av databaser för YCSB, val av databas-gränssnitt, val av belastningsuppgifter, val av körningsparametrar, inmatning av testdata och körning av test (Github, 2018).

3.5.1 Konfiguration av databaser

Vid körning av prestandatest behöver YCSB kunna kommunicera med databasen som används. Varje databas har ett propriaritärt användargränssnitt och dessa behöver implementeras i YCSB. YCSB har officiellt stöd för över 50 databaser, men saknar stöd för många nyare databaser. Stöd fanns för MySQL och Cosmos DocumentDB, men inte för Firebase. Vi behövde därmed utveckla ett eget databaslager för Firebase för att kunna köra testerna. Detta görs genom att implementera gränssnittet com.yahoo.ycsb.DB och definiera skriv, läs, uppdaterings, söknings, och borttagningsmetoderna. Java-klassen för Firebase finns i bilaga 2. För att kompilera jar-filen måste core-0.12.0.jar laddas ned och inkluderas (Github, 2018). Den kompilerade jar-filen placeras i en mapp med namn ”firebase-binding” som placeras i YCSB rotkatalogen. Filen bindings.properties måste innehålla ”firebase:FirebaseClient” för att YCSB ska känna till filerna. Filer för MySQL och Cosmog DocumentDB finns tillgängliga på YCSB:s officiella Github. Anslutningsinställningar för MySQL måste placeras i en fil i YCSB-rotkatalog

enligt bilaga 3a. Anslutningsinställningar för Cosmos DocumentDB placeras i en fil i YCSB-rotkatalog enligt bilaga 3b.

3.5.2 Databasgränssnitt, belastningsuppgifter och parametrar

Gränssnitten för databaserna består av javaklasser som konfigurerats i föregående steg. Dessa klasser utför operationer mot respektive databas. I YCSB-klienten körs en av dessa konfigurationer åt gången. Vid körning specificeras också vilken *workload* (belastningsuppgift) som ska köras. Vi har utfört program **a** och b.

Vi körning av ett test måste klienten också ha ett antal parametrar. Parametrarna som använts är *threadcount*, som avgör hur många användare som simuleras, *recordcount* som används i laddningsfasen och avgör hur många rader data som ska genereras, och *target*, som anger hur många transaktioner per sekund vi önskar att systemet ska uppnå. Den sista parametern avgör inte om systemet kommer att klara av att leverera så många transaktioner per sekund, utan avgör hur länge varje användare ska vänta innan nästa transaktion påbörjas. Om vi anger ett *target* på 1 (en transaktion per sekund), *recordcount* på 1000, och *threads* på 1, och en transaktion tar 100 millisekunder att utföra så skulle det innebära att YCSB-klienten väntar i 900 ms innan nästa transaktion påbörjas. Med dessa parametrar kan man simulera olika tillstånd i ett tänkt system. I testerna som vi utfört är parametrarna *recordcount* på 1000000 i laddningsfasen som beskrivs nedan, och *threads* bevaras på 100 genom alla tester. Efter varje utfört test ökade vi sedan parametern *target* för att öka belastning på systemet. Parametern börjar med värdet 100, i andra testet 1000, därefter ökas värdet med 1000 tills dess att antalet transaktioner per sekund inte längre ökar. Dessa inställningar har vi valt baserat på YCSB:s rekommendationer, samt inspiration från tidigare forskning som beskrivs i kapitel 2.5.

3.5.3 Inmatning av testdata och körning

Ett test i YCSB är uppdelat i två steg; laddning av testdata och körning av test (Github 2018). Innan dessa två kan utföras måste databas och tabeller skapas för MySQL. Följande kod matas in i MySQL databashanteraren:

```
DROP TABLE IF EXISTS usertable;
CREATE TABLE usertable(YCSB_KEY VARCHAR (255) PRIMARY KEY,
  FIELD0 TEXT, FIELD1 TEXT,
  FIELD2 TEXT, FIELD3 TEXT,
  FIELD4 TEXT, FIELD5 TEXT,
  FIELD6 TEXT, FIELD7 TEXT,
  FIELD8 TEXT, FIELD9 TEXT);
```

Följande kommandon användes i kommandotolken i Windows Server för att utföra testerna:

```
bin\ycsb load jdbc -P ../workloads/workloadx -p recordcount=1000000
bin\ ycsb run jdbc -P ../workloads/workloadx -P MySQL -p target=y -p
threadcount=100

bin\ycsb load cosmosdb -P ../workloads/workloadx -P cosmos.dat -p
recordcount=1000000
```

```
bin\ ycsb run cosmosdb -P ../workloads/workloadx –P cosmos.dat -p
target=y -p threadcount=100

bin\ycsb load firebase –P ../workloads/workloadx –p
recordcount=100000
bin\ ycsb run firebase -P ../workloads/workloadx -p target=y -p
threadcount=100
```

X ersätts med vald belastningsuppgift och **y** ersätts med önskade transaktioner per sekund vid varje planerat test.

3.5.4 Konfiguration

Konfigurationen nedan avser den hårdvara och mjukvara som drivit den serverbaserade relationsdatabasen.

- **Primärminne**
    - 2048MB RAM
- **Operativsystem**
    - Windows Server 2012 R2 Standard 64-bit (6.3, Build 9600)
- **Processor**
    - Intel(R) Xeon(R) CPU E5-2660 2.20Ghz

3.5.5 Analyering av prestandatester

Experiment genererar normalt stora mängder data och det kan vara svårt att hantera och analysera om det är för massiv mängd. Därför är det viktigt att efter avslutat experiment summera och presentera resultaten på ett passande sätt. Resultaten bör summeras för att kunna ge en bra bild över vad som framkommit under experimentet. Data som framkommit under experiment presenteras ofta i grafer och diagram. Dessa diagram ligger sedan till grund för den analys som görs av resultaten. Metoden tar upp att man kan göra två olika analyser: en individuell systemanalys samt en jämförande systemanalys (Yao & Hevner, 1984). Vi genomför en jämförande systemanalys då vi kommer att jämföra prestandan mellan de tre olika databaserna. Vi kommer även att sammanställa resultaten från de olika testfallen i diagram för att på ett överskådligt sätt ge en bild över vad vi kom fram till.

3.6 Dataanalys

Oates (2006) tar upp att det finns två olika typer av analysmetoder: kvalitativ och kvantitativ metod för att analysera insamlad data. Kvantitativ analysmetod används när resultatet är numeriskt och det går att utföra statistiska beräkningar på insamlad data. När denna metod används studerar man högt strukturerade data, som framkommit under till exempel observationer eller enkäter. En kvalitativ analysmetod används när man har data som är lågt strukturerad och framkommer från till exempel dokument, intervjuer och enkäter med öppna svar. Data i denna analysmetod tolkas av forskaren för att komma fram till ett resultat för forskningen.

Vi använder oss av både kvalitativ och kvantitativ analysmetod i vårt arbete. Den kvantitativa analysdelen behandlar resultatet av testerna som vi utfört. Genomströmningskapaciteten av databaserna jämfördes mot varandra för att ta reda på vilken databas som presterade bättre än

de andra i respektive fråga. Varje testfall kommer att noteras i en tabell (se bilaga 1) för att ge forskningen transparens och god grund för repeterbarhet.

Litteraturstudien behandlas av den kvalitativa analysmetoden och ska besvara när man bör använda sig av en NoSQL-databas framför en relationsdatabas genom de datakrav som framkommit och analyserats under litteraturstudiens gång.

3.7 Metodkritik

Vi har använt oss av experiment som forskningsstrategi då vi anser att det passar vår forskning bäst. Alternativa strategier att använda oss av hade kunnat varit design och skapande. Då hade vårt syfte till exempel varit att skapa en modell eller guide över att välja databas baserat på krav. Hade vi istället valt att använda oss av en fallstudie hade vi kunnat studera en applikation hos vår samarbetspartner och sedan avgöra vilken databas som skulle lämpa sig bäst i den specifika situationen. Våra testfall kommer att testas i en kontrollerad miljö skapad av oss med testdata och då lämpar det sig bäst att vi använder oss av experiment som forskningsstrategi.

En nackdel med att utföra experiment i en kontrollerad miljö är att resultatet kan bli orealistiskt och ojämförbart med verkligheten (Oates, 2006). Syftet med våra tester är att verifiera prestandaskillnader mellan databaserna. Vi kan inte med våra tester påstå att resultatet vi kommit fram till är applicerbart på alla typer av applikationer. Testerna har genomförts med testdata då vi vill skydda vår samarbetspartners integritet och inte använda verkliga data som används inom företaget.

De datainsamlingsmetoder vi har använt oss av är litteraturstudie samt observationer. Vi ansåg att dessa räckte för att besvara vårt syfte och våra mål med arbetet. Vårt syfte var att reda ut när en NoSQL-databas är att föredra framför en relationsdatabas samt även vilka för- och nackdelar som finns och verifiera prestandaskillnader. Vi besvarar dessa frågor med vår litteraturstudie och våra tester mot databaserna. Oates (2006) skriver att metodtriangulering är ett bra sätt att öka trovärdigheten av sitt arbete. Med triangulering menas att forskaren använder två eller fler metoder för att samla in data för att ge mer stöd för sin forskning. Vi hade kunnat använda fler datainsamlingsmetoder, som till exempel intervjuer, för att generera mer data som sedan skulle ha analyserats och diskuterats i resultatet. Vi anser dock att området är så pass nytt och att hitta respondenter hade varit ett hinder. Vi anser även att det inte hade bidragit med mer relevanta data än det vi redan har för att uppnå syftet med arbetet. Tiden var också en faktor till varför fler metoder för datainsamling inte användes. Hade vi haft mer tid på oss att genomföra arbetet hade vi kunnat använda fler datakällor. Vi hade även kunna utfört tester på flera olika databaser än de tre vi testade.

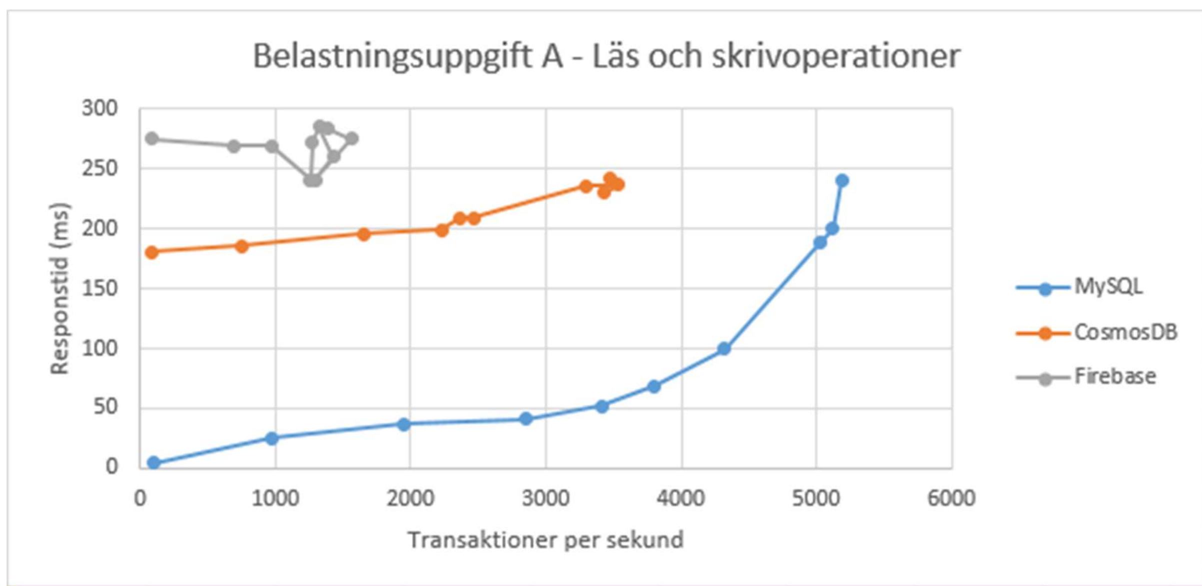
Nackdelar med att använda sig av en kvantitativ analysmetod är att forskarens val kan influera resultatet av en sådan forskning. Genom att till exempel välja ett opassande intervall på x- och y-axel kan resultatet framställas subjektivt. Den kvalitativa analysmetoden kan även den bli influerad av forskaren, oftast i större utsträckning än kvantitativ analys. Det är forskarens tolkning som analysen utgår ifrån vilket kan medföra mer subjektivitet (Oates, 2006). Vi anser att resultatet som framställs genom analyser framställs på ett objektivt sätt.

Det här examensarbetet bör inte generaliseras över flera NoSQL-databaser då testerna endast utförts på två NoSQL-databaser, som är molnbaserade. Det finns många variabler som påverkar belastningstester mot databaser som vi i detta arbete inte har forskat vidare på. Den prestanda som kan utvinnas i en molnbaserad databas beror på leverantören och därmed kan dessa experiment inte generaliseras.

4 Resultat av prestandatester

Kapitlet presenterar de resultat som framkom under prestandatesterna. Informationen är genererad från de experiment som utfördes genom olika testfall mot de valda databaserna.

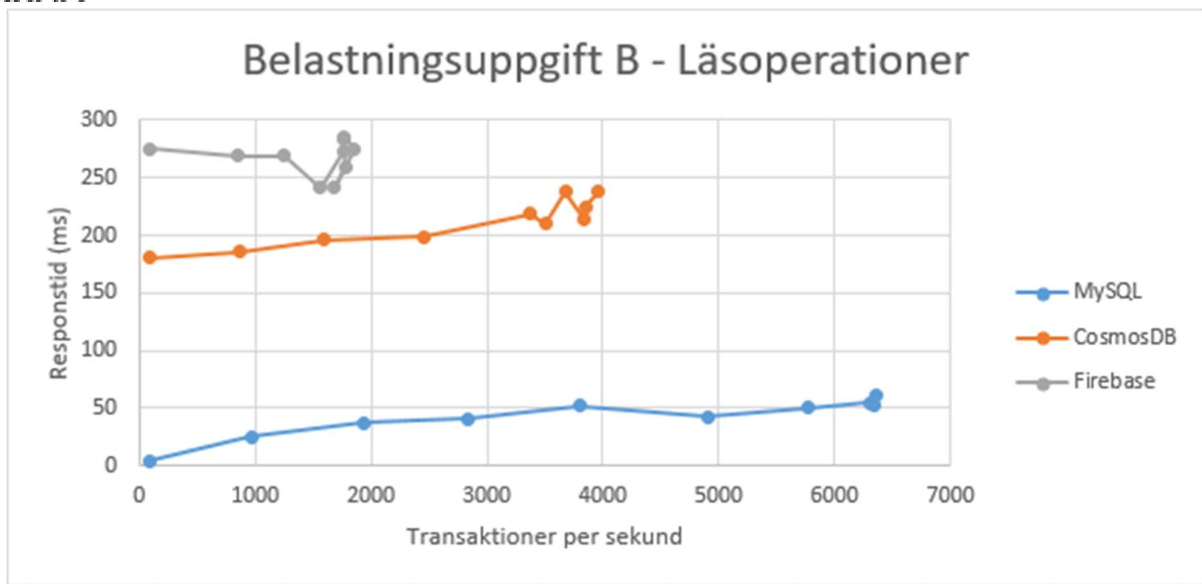
I det första testet användes belastningsuppgift A i YCSB. Testet består av samtidiga transaktioner där fördelningen mellan operationer är 50% läsoperationer och 50% skrivoperationer. Testet utfördes med 100 klienttrådar mot en databas på en miljon rader. Testet har utförts 10 gånger där parametern target har ändrats från 100 till 10000. Diagrammet visar det uppnådda antalet transaktioner per sekund, tillsammans med den genomsnittliga latenstiden för en transaktion. Varje punkt i diagrammet avser en körning av testet. Transaktionsantalet på x-axeln är antalet transaktioner som utförts varje sekund. Ett högre tal innebär att databasen har lyckats hantera fler transaktioner. Responstiden på y-axeln är i millisekunder och anger den genomsnittliga tiden för ett anrop, från det att YCSB påbörjat en generering av en transaktion, skickat till, samt fått tillbaka ett svar från databasen. En lägre responstid innebär ett snabbare utförande, och att klienttråden finns tillgänglig för en ny förfrågan av YCSB.



Figur 7

Belastningsuppgift A

Resultatet för MySQL visar att responstiden för ett anrop ökar i takt med att antalet samtidiga transaktioner som utförs ökar. Antalet transaktioner per sekund ökade tills antalet kom strax över 5000 transaktioner. Den lägsta latensen för CosmosDB låg på 181 millisekunder (ms) och varierade mellan 181 och 242ms. Latenstiden ökade i takt med att det begärda antalet transaktioner per sekund ökade, dock inte på samma nivå som MySQL. Maximalt uppnådda transaktioner per sekund landade på 3538 transaktioner. Maximalt uppnådda transaktioner per sekund för firebase var 1560 vilket var de lägsta siffrorna av de tre. Responstiden varierar ostabilt mellan 241 och 284 ms.
I det andra testet användes belastningsuppgift B i YCSB. Detta test består av 95% läsoperationer och 5% skrivoperationer. Testet är i övrigt utformat som det tidigare.



Figur 8

Belastningsuppgift B

Resultatet visar att det maximala antalet transaktioner per sekund ökade i MySQL till 6358 transaktioner per sekund. Responstiden för ett anrop påverkades inte lika negativt som i föregående test. Höjdpunkten för CosmosDB resulterade också högre från 3538 till 3962 transaktioner per sekund med en nästan oförändrad latenstid. Firebase gjorde också en ökning i antalet transaktioner per sekund till 1853 transaktioner. Latenstiden varierade mellan 285 till 241ms.

5 Analys

Nedan följer en analys av resultaten som framkom under prestandatesterna. Efter det följer en analys av de datakrav som kan spela in när man ska välja vilken typ av databas att använda sig av. Kraven som tas upp framkom under vår litteraturstudie och kan ses som riktlinjer för när vilken typ av databas är att föredra framför en annan.

5.1 Analys av prestandatesterna

Enligt litteraturstudien är två huvudsakliga syften med NoSQL att databaserna ska vara effektiva i form av prestanda och skalbarhet. Testerna som utförts med Yahoo Cloud Serving Benchmark har testat prestandan för tunga skrivoperationer och tunga läsoperationer hos de tre databaser vi har utvärderat. YCSB genererade värdefull data som genomsnittlig responstid per anrop, genomsnittligt mått på antalet transaktioner per sekund, samt enkla metoder för att skapa spridningsdiagram. För MySQL visade resultatet att responstiden per anrop ökade i takt med antalet transaktioner. Kurvan i spridningsdiagrammet visar att MySQL-databasen blir långsammare ju fler förfrågningar som ska hanteras. Liknande resultat kan ses i studien av Floratou et al. (2012) som jämför skillnaden mellan MS SQL Server och MongoDB. Den relationsbaserade databasen arbetar långsammare desto högre antal transaktioner som ska hanteras. Även studien av Cooper et al. (2010) visar på samma effekt. Denna effekt kan förklaras av ACID-principerna som beskrevs av litteraturstudien, och som även diskuteras i Floratou et al. (2012). Om en läsoperation sker strax efter en skrivoperation så måste läsoperationen placeras i kö av databashanteraren för att kunna garantera korrekta data.

Våra resultat visade en betydligt sämre prestanda för de båda NoSQL-databaserna relativt till MySQL-databasen. I spridningsdiagrammet kan vi se att responstiderna genomgående är högre i förhållande till relationsdatabasen. Resultat från Cooper et al. (2010) visar en betydligt högre prestanda, och lägre svarstider för både HBase och Cassandra vid körning av samma test. Vi testade att köra belastningstest A med en databasstorlek på 100 rader och noterade även där att responstiderna var avsevärt högre än MySQL. Med tanke på att de båda NoSQL databaserna är molnbaserade kan en högre responstid dock vara att förvänta. Vi tror därmed att resultatet som observerats hänger ihop med placeringen av de servrar vi anslutit till och därmed bör dessa tester inte generaliseras. Firebase är heller inte utvecklat för att utföra tunga belastningar, utan att leverera data till användare i realtid.

Hypoteserna som framställts genom litteraturstudien falsifieras enligt våra tester. MySQL-databasen toppade antalet utförda transaktioner per sekund i både belastningstest A och B. Den underliggande arkitekturen hos NoSQL-databaserna är utvecklad för att hantera in och ut data effektivt, vilket andra studier har verifierat att de gör. Våra tester visar därmed att prestandan hos NoSQL inte är självklar i de fall där databasen drivs i molnet.

5.2 Framtagna datakrav

Vi har analyserat fram ett antal krav som kan finnas på data som ska lagras och genom dessa ge en riktlinje för vilken typ av databas man bör välja som lagringslösning. Dessa datakrav har framkommit under vår litteraturstudie men även genom våra prestandamätningar. Kraven kan ses som ett stöd när man ska välja typ av lagringslösning. Oftast är det inte bara ett av dessa krav som bestämmer vilken typ av databas som passar bäst utan en kombination har krav. Det är heller inte alltid ett enkelt val att välja databas och valet bör därmed göras noggrant och genomtänkt innan en lösning implementeras.

All data som lagras i en relationsdatabas kommer att lagras på samma fördefinierade sätt med samma regler och följa samma mönster i och med att en modell måste definieras innan man börjar lagra data. Är data man ska lagra strukturerade och följer samma mönster är det naturliga valet att man bör använda en relationsdatabas. Är det dessutom viktigt att det finns relationer mellan de olika tabellerna så är en relationsdatabas ett givet val. En relationsdatabas kan dessutom hantera komplexa sökoperationer, något som inte finnas hos NoSQL-databaserna, vilket också är ett krav som talar för relationsdatabasen (Boicea et al., 2012). Det går att använda selektering av data tillsammans med en NoSQL-databas, men då sker detta vid applikationslagret jämfört med i datalagret hos relationsdatabasen (Corbellini et al., 2017). Skulle det däremot vara semi- eller ostrukturerade data som ska lagras är en NoSQL-databas ett mer passande val. Det kan även finnas situationer då man har både strukturerade och ostrukturerade data som ska lagras och beroende på om det finns fler krav som spelar in kan en kombination av dessa två typer vara att överväga.

Ett krav som kan vara avgörande vid valet av databas är ACID-principerna. Är det viktigt att data följer detta regler är det en relationsdatabas man bör välja som lagringslösning. Ett exempel på system som kräver att data är korrekt hela tiden och att det inte ska kunna ske samtidiga transaktioner är ett banksystem. Där är det viktigt att systemet har rätt information hela tiden för att det inte ska ske motstridigheter i systemet. Al Hinai (2016) tar upp att principerna kan påverka prestandan negativt, men är korrekt data viktigare än prestanda är ACID-principerna ett viktigt krav att ta hänsyn till. Det finns NoSQL-databaser som kan hantera ACID-principerna. Cosmo kan köras med principerna med förutsättning att alla operationer körs från lagrade procedurer. Ett givet val är ändå att välja en relationsdatabas vid stor vikt av detta krav.

Corbellini et al. (2017) skriver att relationsdatabaserna inte designades med horisontell skalbarhet i åtanke, vilket även detta kan vara ett viktigt krav att ta hänsyn till. Rör det sig om stora mängder data som ska hanteras i ett system kan en NoSQL-databas vara en bra lösning då dessa hanterar skalbarhet på ett bra sätt. Både de lokala och molnbaserade NoSQL-databaserna hanterar horisontell skalbarhet på ett effektivt sätt vilket kan öka prestanda i form av till exempel läshastighet. Detta då om en sökning sker i databasen så behöver man inte gå igenom hela databasen utan vet vilken nod som håller i den information som man ska ha. Att dela upp en databas ökar även tillgänglighet, tillförlitlighet och garanterar feltolerans (Al Hinai, 2016), vilket är en styrka hos NoSQL-databaserna. Databaser som är uppdelade över flera noder är mer tillgängliga, även om en nod eller enhet i systemet skulle vara nere. Om en nod skulle försvinna helt så skulle systemet inte förlora data vilket garanterar en tillförlitlighet.

NoSQL-databaser presterar generellt snabbt vid både läs- och skrivoperationer specifikt. De hanterar operationer över lag med låga svarstider. En anledning till detta kan vara för att de är designade för att vara tillgängliga samt för att hämta data med hög hastighet (Khazaei et al., 2016). I kombination med den horisontella skalbarheten hos databaserna passar de bra för att hantera stora mängder data. I och med att begreppet Big Data blir allt mer utbrett har NoSQL-databaser utvecklats för att kunna svara på de krav inom lagringskapacitet och hastighet som krävs för dessa datamängder. NoSQL-databaserna passar bra då stora mängder data behöver hämtas och analyseras med hjälp av analysprogram.

När det handlar om datakrav så behöver det inte bara vara ett krav som bestämmer vilken typ av databas som krävs. Genom att prioritera de viktigaste datakraven för varje specifik situation kan man ta fram riktlinjer för vilken typ av lagringslösning som passar bäst. I vissa fall kan det vara så att det krävs både en relations- och en NoSQL-databas för att uppnå optimal

funktionalitet. Nedan följer en tabell för att ge en bättre överblick över de krav som framkommit under vårt arbete.

Datakrav	**Relationsdatabas**	**NoSQL-databas**
Strukturerade data	x	
Semistrukturerade data		x
Ostrukturerade data		x
Komplexa sökoperationer	x	
ACID	x	
Horisontell skalbarhet		x
Tillgänglighet		x
Big Data		x
Snabb läshastighet		x
Snabb skrivhastighet		x
Analys		x

Tabell 2

Datakrav framtagna från litteraturstudien

6 Diskussion och slutsats

I kapitlet besvarar vi vårt syfte och våra frågeställningar. Resultatet diskuteras och innehåller även reflektion kring resultatet och förslag på vidare forskning.

6.1 Svar på frågeställning

Syftet med vårt arbete var att förklara när en NoSQL-databas kan vara att föredra framför en relationsdatabas. För att besvara detta formulerades frågeställningar som besvaras nedan. Svaren baseras genom litteraturstudien i kombination med de experiment som utfördes.

När kan en NoSQL-databas vara att föredra framför en relationsdatabas?
Vilken typ av databas som bör väljas beror på syftet och kraven som ställs på det data som ska lagras. Den främsta anledningen till att välja en NoSQL-databas är när mängden data som ska lagras förväntas bli så stor att en ensam enhet inte räcker till. En NoSQL-databas kan bekymmersfritt skala horisontellt till flera noder tack vare dess ostrukturerade datamodell. Den ostrukturerade datamodellen innebär även att nya attribut (kolumner i en relationsmodell) enkelt kan läggas till under utvecklingens gång. En applikation som förväntas förändra datastruktur frekvent, som att lägga till eller ta bort attribut kan tjäna på att använda NoSQL. Den kan även vara att överväga om den data man har är ostrukturerad. NoSQL bör också tillämpas när kraven på läs och skrivhastigheter är höga. Vid till exempel analysering, filtrering eller visualisering av data kan NoSQL-databaser leverera högre effektivitet.

Kravet på data som ska lagras är en viktig del vid valet av databas. Den traditionella relationsdatabasen uppfyller akronymen ACID vilket garanterar korrekt data även om fel uppstår under en transaktion. Dessa egenskaper är viktiga i till exempel ett banksystem där varje operation måste resultera i korrekt data. Om tillgänglighet är av större vikt är NoSQL ett bra val tack vare att data enkelt kan replikeras till flera noder. SQL-baserade databaser kan också hantera avancerade sökfrågor mot data. NoSQL-databaser är generellt begränsade och kan i de flesta fall bara utföra simpla frågor.

Vilka skillnader finns det i form av prestanda mellan databaserna?
NoSQL-databaser levererar högre prestanda till kostnad av den funktionalitet som finns i SQL-baserade relationsdatabaser. Tidigare forskning har påvisat effektiviteten hos en rad NoSQL-databaser. Våra experiment har verifierat en ökning i prestanda i en av de två NoSQL-databaserna vi använt. Dessa två databaser är molnbaserade och resultatet visade att högre prestanda kan uppnås om databasen drivs på egna servrar, då till kostnad av tillgänglighet.

6.2 Diskussion

Syftet med arbetet var att förklara när en NoSQL-databas kan vara att föredra framför en relationsdatabas. För att lyckas besvara syftet gjordes en grundlig litteraturstudie. I litteraturstudien framkom det att NoSQL fått ny luft på marknaden på grund av utvecklingen av IoT och Big Data. Vi upptäckte också att NoSQL är inte en typ av databas, utan en samling av en mängd olika typer av databaser med varierande datamodeller och designsyfte. Vi genomförde även experiment för att jämföra de tre databaserna för att ta reda på om det fanns några skillnader i form av prestanda. Metoden vi använde oss av för att utforma testerna och göra jämförelsen mellan databaserna kändes trovärdig. Dock fick vi göra lite modifikationer då den

inte var anpassad efter de nyare NoSQL-databaserna. Dessa databaser har sällan stöd för till exempel komplexare operationer (Boicea et al., 2012), vilket gjorde att vi inte kunde utföra sådana tester. Tidigare forskning tar även upp just detta, att det är en svaghet hos NoSQL-databaser att selektering av data istället sker på applikationsnivå (Corbellini et al., 2017). Våra tester visar att det finns skillnader mellan databaserna och att NoSQL-databaser kan prestera med högre genomströmning än relationsdatabaser. Mycket av den tidigare forskning som vi läst har påvisat samma resultat (Pereira et al., 2018). Dock så tror vi att vi hade fått mer tydliga svar om vi använt oss av NoSQL-databaser på dedikerade servrar. Vi anser även att våra tester visar att molnbaserade databaser kan dra ner prestandan för att väga upp den tillgänglighet de istället erbjuder genom molnet.

6.3 Reflektion

Att besvara vilken databas som bör väljas beroende på krav var ett betydligt bredare och djupare ämne än vi hade anat. Vår tidigare kunskap inom ämnet var begränsad vilket inneburit att varje del av arbetet har tagit långt tid. Vi hade velat göra fler tester mot databaserna men eftersom allt tog så pass mycket längre tid än vi anat blev vi tvungna att avgränsa oss till skriv- och läsoperationer.

Vårt bidrag med detta arbete är att på ett klargörande sätt beskriva skillnaderna mellan traditionella relationsdatabaser och NoSQL-databaser samt att ge en konkret bild över när NoSQL kan vara att föredra framför traditionella relationsdatabaser. Experimenten som utfördes kan inte verifiera eller falsifiera prestandan av NoSQL-databaser på ett generaliserbart plan. För att se den verkliga skillnaden mellan databaserna måste alla databaser drivas av samma konfiguration. Vi menar att resultatet ändå är ett värdigt bidrag då vi påvisat att högre prestanda kan utvinnas om systemet bedrivs på en dedikerad server.

6.4 Slutsats

Det som avgör om en NoSQL-databas är att föredra framför en relationsdatabas är kraven som ställs på det data som ska lagras. Är det ostrukturerade data eller om det är stora mängder data kan en NoSQL-databas vara att överväga. Dessa databaser ska hantera skalbarhet effektivt och då databasstrukturerna inte bygger på relationsmodellen är det enkelt att hantera ostrukturerade data. Bristen på ACID-garantier resulterar i snabbare hantering av förfrågningar mot databasen och högre prestanda. Resultat från utförda tester visar dock att högre prestanda inte är en garanti i de fall där databasen drivs i molnet. Resultatet visar också att relationsdatabasen presterar sämre under hög belastning när stora mängder data lagras i databasen. Då det finns olika typer av NoSQL-databaser finns det ett stort utbud att välja bland beroende på vad den data man har ska användas till. Det finns inte en databas som är bättre än någon annan utan allt handlar om de krav som ställd på den data som ska lagras. Utefter dessa krav kan analyser göra för att dra slutsatser kring vilken typ av databas som är att föredra.

6.5 Vidare forskning

Förslag på vidare forskning kan vara att utföra fler experiment mot NoSQL-databaserna. YCSB har fler belastningsprogram som inte rymdes i detta arbete. Ett annat förslag är att titta på något som vi stötte på under litteraturstudien, så kallade NewSQL-databaser (Corbellini et al., 2017). Dessa databaser påstods erbjuda liknande prestandaökning som NoSQL, i kombination med stöd för ACID-garantier och SQL-liknande förfrågningsmöjligheter.

Referenser

Tryckta källor och litteratur

Björklund, M. & Paulsson, U. (2012). *Seminarieboken.* Studentlitteratur. Lund.

Oates, B. J. (2006). *Researching information systems and computing*. Sage.

Sadalage, P. J., & Fowler, M. (2012). NoSQL distilled. *ISBN-10, 321826620*.

Artiklar

Barata, M., & Bernardino, J. (2016). Cassandra's Performance and Scalability Evaluation. In *DATA* (pp. 127-134). doi:10.5220/0005980101270134

Baron, C. A. (2016). NoSQL key-value DBs riak and redis. *Database Systems Journal,* (4), 3-10.

Codd. E. F. (1970). A relational model of data for large shared data banks. *Communications of ACM, 13*(6), 377-387

Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., & Sears, R. (2010, June). Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing* (pp. 143-154). ACM.

Corbellini, A., Mateos, C., Zunino, A., Godoy, D., & Schiaffino, S. (2017). Persisting big-data: The NoSQL landscape. *Information Systems, 63*, 1-23. 10.1016/j.is.2016.07.009

Floratou, A., Teletia, N., DeWitt, D. J., Patel, J. M., & Zhang, D. (2012). Can the elephants handle the nosql onslaught?. *Proceedings of the VLDB Endowment*, *5*(12), 1712-1723.

Gilbert, S., & Lynch, N. (2002). Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News, 33*(2), 51-59. doi:10.1145/564585.564601

Khazaei, H., Fokaefs, M., Zareian, S., Beigi-Mohammadi, N., Ramprasad, B., Shtern, M., ... & Litoiu, M. (2016). How do I choose the right NoSQL solution? A comprehensive theoretical and experimental survey. *Big Data & Information Analytics*, *1*(2/3), 185-216. doi: 10.3934/bdia.2016004

Palovská, H. (2015). What can NoSQL serve an enterprise. *Journal of Systems Integration, 6*(3), 44-49. 10.20470/jsi.v6i3.240. Hämtad från http://www.si-journal.org/index.php/JSI/article/viewFile/240/227

Pereira, D. A., Ourique de Morais, W., & Pignaton de Freitas, E. (2018). NoSQL real-time database performance comparison. *International Journal of Parallel, Emergent and Distributed Systems*, *33*(2), 144-156.

Yao, S. B., & Hevner, A. R. (1984). A Guide to Performance Evaluation of Database Systems. *NBS Special Publication, 500-118*. Washington, DC: U. S. Government Printing Office.

Övriga källor

.NET. (2018, 13 maj). *I Wikipedia*. Hämtad 2018-05-21 från https://en.wikipedia.org/wiki/.NET_Framework

ACID (2018, 23 april). I *Wikipedia.* Hämtad 2018-04-24, från https://en.wikipedia.org/wiki/ACID

Al Hinai, A. H. (2016). *A Performance Comparison of SQL and NoSQL Databases for Large Scale Analysis of Persistent Logs* (Examensarbete, Uppsala Universitet, Uppsala). Hämtad från https://uu.diva-portal.org/smash/get/diva2:957015/FULLTEXT01.pdf

Boicea, A., Radulescu, F., & Agapin, L. I. (2012, September). MongoDB vs Oracle--database comparison. In *Emerging Intelligent Data and Web Technologies (EIDWT), 2012 Third International Conference on* (pp. 330-335). IEEE.

Brander, T., & Dakermandji, C. (2016). *En jämförelse mellan databashanterare med prestandatester och stora datamängder*. (Examensarbete, Kungliga Tekniska Högskolan, Stockholm) Hämtad från http://kth.diva-portal.org/smash/get/diva2:934333/FULLTEXT01.pdf

Cosmos DB. (2018, 7 april). I *Wikipedia*. Hämtad 2018-04-12 från https://en.wikipedia.org/wiki/Cosmos_DB

Data requirements. (2012, 28 mars). *I SemWebQuality.* Hämtad 2018-05-21 från http://semwebquality.org/mediawiki/index.php?title=Create_Data_Requirements

Database. (2018, 2 april). I *Wikipedia.* Hämtad 2018-04-03, från https://en.wikipedia.org/wiki/Database

DB-Engines. (2018). *DB-Engines Ranking*. Hämtad 2018-05-10 från https://db-engines.com/en/ranking

Firebase. (2018, 25 April). *I Wikipedia*. Hämtad 2018-05-08 från https://en.wikipedia.org/wiki/Firebase

Github. (2018). *YCSB*. Hämtad 2018-06-07 från https://github.com/brianfrankcooper/YCSB

Google. (2018). *Firebase Realtime Database*. Hämtad 2018-05-08 från https://firebase.google.com/docs/database/

Google. (u.å.). *Cloud BigTable.* Hämtad 2018-04-22 från https://cloud.google.com/bigtable/

IDG. (u.å.). *DBMS, identifierare, JSON, nästling, programmeringsgränssnitt, SDK.* Hämtad 2018-05-21 från https://it-ord.idg.se/ord/

Lith, A., & Mattsson, J. (2010). *Investigating storage solutions for large data-A comparison of well performing and scalable data storage solutions for real time extraction and batch insertion of data.* (Master's thesis, Chalmers University of Technology, Göteborg). Hämtad från http://publications.lib.chalmers.se/records/fulltext/123839.pdf

Microsoft. (2018). *Welcome to Azure Cosmos DB*. Hämtad 2018-04-11 från https://docs.microsoft.com/en-us/azure/cosmos-db/introduction

MySQL. (2018, 8 maj). *I Wikipedia*. Hämtad den 2018-05-11 från https://en.wikipedia.org/wiki/MySQL

Nelubin, D., & Engber, B. (2013). Ultra-high performance nosql benchmarking: Analyzing durability and performance tradeoffs. *Thumbtack Technology, Inc., White Paper*.

Open-source software. (2018, 14 maj). *I Wikipedia*. Hämtad den 2018-05-15 från https://en.wikipedia.org/wiki/Open-source_software

Oracle. (u.å.). *What is MySQL?*. Hämtad den 2018-05-15 från https://dev.mysql.com/doc/refman/8.0/en/what-is-mysql.html

Ramakrishnan, L., Mantha, P. K., Yao, Y., & Canon, R. S. (2013). Evaluation of nosql and array databases for scientific applications. In *DataCloud Workshop*.

Shilov, M. (2016, 16 juni). *Where is NoSQL practically used?*. Hämtad 2018-05-09 från http://scraping.pro/where-nosql-practically-used/

Sundgren, B. (2016). *Big Data and predictive analytics - a scientific paradigm shift?* Pro Libera Scio.

TPC, Web-Design and Maintenance (u.å. a). *TPC-C is an On-Line Transaction Processing Benchmark*. Hämtad 2018-06-06 från http://www.tpc.org/tpcc/

TPC, Web-Design and Maintenance (u.å. b). *TPC-E is an On-Line Transaction Processing Benchmark*. Hämtad 2018-06-06 från http://www.tpc.org/tpce/

TPC, Web-Design and Maintenance (u.å. c). *TPC-H is a Decision Support Benchmark*. Hämtad 2018-06-06 från http://www.tpc.org/tpch/

Triona. (u.å.). *Om Triona*. Hämtad 2018-04-23 från https://www.triona.se/om_triona/

HÖGSKOLAN
DALARNA

Bilaga 1

Data från testfallen

Nedan följer de noterade data från testerna som utförts med YCSB.

MySQL (a)		CosmosDB (a)		Firebase (a)	
Transaktioner	Latens (ms)	Transaktioner/s	Latens (ms)		
98,28	4	79,46	181	81,42	275
974,42	25	745,34	186	689,45	269
1953,65	37	1649,42	196	969,79	269
2849,53	41	2229,3	199	1258,13	241
3409,32	52	2366,64	209	1269,9	273
3804,41	69	2464,84	209	1323,48	285
4322,87	100	3302,48	236	1423,68	260
5032,96	189	3538,53	237	1296,94	241
5116,96	200	3428,56	230	1560,2	275
5182,64	240	3475,89	242	1389,34	284
MySQL (b)		CosmosDB (b)		Firebase (b)	
97,71	4	85,25	181	86,57	275
963,63	25	863,58	186	850,12	269
1942,83	37	1596,67	196	1247,82	269
2841,32	41	2459,6	199	1563,8	241
3796,87	52	3378,34	219	1763,31	273
4899,9	43	3499,75	210	1762,92	285
5769,73	50	3675,27	238	1776,74	260
6296,71	55	3839,76	214	1681,2	241
6347,86	53	3846,61	225	1853,18	275
6358,69	61	3962,74	238	1765,64	284

Bilaga 2

YCSB klientklass för Firebase

```
package org.ycsb.firebase;

import com.google.firebase.database.FirebaseDatabase;
import com.google.firebase.database.util.JsonMapper;
import com.mashape.unirest.http.Unirest;
import com.yahoo.ycsb.ByteIterator;
import com.yahoo.ycsb.DBException;
import com.yahoo.ycsb.Status;
import com.yahoo.ycsb.StringByteIterator;
import org.apache.commons.io.IOUtils;
import org.json.JSONObject;

import java.io.IOException;
import java.net.URL;
import java.nio.charset.Charset;
import java.util.*;

public class FirebaseClient extends com.yahoo.ycsb.DB {

    FirebaseDatabase db;

    public FirebaseClient() {

    }

    @Override
    public void init() throws DBException {
        super.init();
        System.out.println(getProperties().toString());
    }

    public Status read(String table, String key, Set<String> fields,
HashMap<String, ByteIterator> result) {
        try {
            JSONObject value = new JSONObject(IOUtils.toString(
                    new URL("https://firestoretest-
200007.firebaseio.com/flight_data/" + table + "/" + key + ".json"),
                    Charset.forName("UTF-8")));
            if (result != null && fields != null) {
                for (String field : fields) {
                    result.put(field, new
StringByteIterator(value.getString(field)));
                }
            }
            return Status.OK;
        } catch (IOException e) {
            e.printStackTrace();
            return Status.ERROR;
        }
    }

    public Status scan(String table, String key, int recordCount,
Set<String> fields, Vector<HashMap<String, ByteIterator>> result) {
```

```
        try {
            JSONObject dbvalues = new JSONObject(IOUtils.toString(
                    new URL("https://firestoretest-
200007.firebaseio.com/flight_data/" + table + ".json"),
                    Charset.forName("UTF-8")));
            for (int i = 0; i < recordCount && i < dbvalues.length(); i++) {
                if (dbvalues.get(key) != null && result != null && fields !=
null) {
                    HashMap<String, ByteIterator> values = new
HashMap<String, ByteIterator>();
                    for (String field : fields) {
                        values.put(field, new
StringByteIterator(dbvalues.getString(field)));
                    }
                    result.add(values);
                }
            }
            return Status.OK;
        } catch (IOException e) {
            e.printStackTrace();
            return Status.ERROR;
        }
    }

    @SuppressWarnings("ALL")
    public Status update(String table, String key, HashMap<String,
ByteIterator> values) {
        try {
            int status = Unirest.put("https://firestoretest-
200007.firebaseio.com/flight_data/" + table + "/" + key + ".json")
                    .body(JsonMapper.serializeJson(toMap(values))).asString(
).getStatus();
            return status == 200 ? Status.OK : Status.ERROR;
        } catch (Exception e) {
            e.printStackTrace();
            return Status.ERROR;
        }
    }

    @SuppressWarnings("ALL")
    public Status insert(String table, String key, HashMap<String,
ByteIterator> values) {
        try {
            int status = Unirest.put("https://firestoretest-
200007.firebaseio.com/flight_data/" + table + "/" + key + ".json")
                   .body(JsonMapper.serializeJson(toMap(values))).asString()
.getStatus();
            return status == 200 ? Status.OK : Status.ERROR;
        } catch (Exception e) {
            e.printStackTrace();
            return Status.ERROR;
        }
    }
    @SuppressWarnings("ALL")
    public Status delete(String table, String key) {
        try {
            int status = Unirest.delete("https://firestoretest-
200007.firebaseio.com/flight_data/" + table + "/" + key + ".json")
                    .asString().getStatus();
            return status == 200 ? Status.OK : Status.ERROR;
```

```
        } catch (Exception e) {
            e.printStackTrace();
            return Status.ERROR;
        }
    }
    @Override
    public Properties getProperties() {
        return super.getProperties();
    }
    private HashMap<String, Object> toMap(Map<String, ByteIterator> values)
{
        HashMap<String, Object> map = new HashMap<String, Object>();
        for (Map.Entry<String, ByteIterator> entry : values.entrySet()) {
            map.put(entry.getKey(), entry.getValue());
        }
        return map;
    }
}
```

Bilaga 3a

Anslutningsinställningar för MySQL placeras i en fil i YCSB-rotkatalog och används enligt följande:

db.driver=com.mysql.jdbc.Driver
db.url=jdbc:mysql://<Sökväg>/yscb
db.user=<Användarnamn>
db.passwd=<Lösenord>

Exempel:
db.driver=com.mysql.jdbc.Driver
db.url=jdbc:mysql://localhost/yscb
db.user=root
db.passwd=root

Bilaga 3b

Anslutningsinställningar för CosmosDB placeras i en fil i YCSB-rotkatalog och används enligt följande:

documentdb.host=<URL till Azure>
documentdb.masterKey=<Primärnyckel till databas>

Exempel:
documentdb.host=https://a0b5fd23-0ee0-4-231-b9ee.documents.azure.com
documentdb.masterKey=3QjvWV4zxxZPxwLYmT2RiuP5RO3ILOqt5VoBH5G4Z7m6D4XF15X8aD7x