Examensarbete

HÖGSKOLAN I SKÖVDE

WEBBRAMVERK FÖR SÖKNING AV HETEROGEN DATA UTIFRÅN SINGLE-PAGE SÖKFUNKTION

WEB FRAMEWORK FOR SEARCHING HETEROGENEOUS DATA BY SINGLE-PAGE SEARCH FUNCTION

Examensarbete inom huvudområdet Informationsteknologi
Grundnivå 30 högskolepoäng
Vårtermin 2019

Mattias Andersson

Handledare: Henrik Gustavsson
Examinator: Mikael Berndtsson

Sammanfattning

Detta arbete fokuserar på att utveckla två webbapplikationer i ramverken Django och Node.js tillsammans med Express för att besvara frågan av vilket ramverk som erhåller bäst söktider utifrån en Q&A plattform som använder en PostgreSQL databas vars innehåll är delar av Stack Overflow datasetet. Ramverken jämförs med hjälp av metoden experiment på grund av fördelarna som det erbjuder. Resultatet blev att Node.js gav bättre söktider för ett mindre webbapplikationer medan Django presterade bättre för större webbapplikationer. Genom att halvera storleken av sökresultatens kroppstext till 150 tecken fick Node.js söktider som i snitt var bättre än Django vid större projekt. Antalet sökresultat har en inverkan där vardera ramverk har sina egna intervaller där de ger bäst söktider. Kortsiktigt kan arbetet fortsätta genom att utföra ytterligare mätningar för respektive faktor, långsiktigt kan dessa ramverk jämföras med andra för att se om dessa två ligger bland de bättre eller sämre för denna tillämpningen.

Nyckelord: Django, Node.js, PostgreSQL, AJAX, Stack Overflow

Innehållsförteckning

1 Introduktion

Webbapplikationer har kommit till att bli en viktig del av många personers liv och används bland annat av företag för att visa produkter och förmedla information. Användare av webbapplikationer blir ofta frustrerade när laddtider går över 2 sekunder och därför är det viktigt att hålla korta laddtider (Butkiewicz m.fl. 2011). Genom att implementera en single-page applikation som använder AJAX, begränsas den data som skickas mellan klient och server och kan således påverka laddtider positivt eftersom endast delar av hemsidan behöver uppdateras. Den programkod som exekveras på servern måste fortfarande vara snabb, säker och strukturerad så att applikationens kodbas och prestanda kan skala väl när antalet användare och funktioner ökar. Ett beprövat ramverk kan genom ett lager programkod erbjuda just det - hög säkerhet samtidigt som en mjukvaruarkitektur hjälper till med strukturen av projektet (Li m.fl. 2017).

Q&A plattformar, som exempelvis Stack Overflow, har fått stor betydelse för utvecklare runtom i världen till deras arbete genom att ge svar på olika problem. Stack Overflow har över 3'000'000 aktiva användare varje månad och kräver således snabba svarstider för att kunna hantera alla http-förfrågningar (Ponzanelli m.fl. 2015). Detta gör ramverk för Q&A plattformar värt att undersöka för att hitta det ramverk som kan skala bäst sett till storleken av plattformen.

Detta arbete kommer fokusera på att evaluera webbramverken Django och Node.js tillsammans med Express, som båda är kraftiga ramverk enligt Schutt & Balci (2016), utifrån en identiskt implementerad Q&A plattform som lagrar delar av det öppna datasetet Stack Overflow i en PostgreSQL databas. Fokuset ligger på en AJAX sökfunktion som dynamiskt genererar en vy med de sökresultat som matchar de sökord som användaren har använt. För att göra sökningar snabbare så används fulltextsökning i form av en tsvector kolumn tillsammans med ett GiN index.

Webbramverken kommer jämföras med hjälp av ett teknikorienterat empiriskt experiment på grund av de fördelar som metoden erbjuder. Det vill säga en kontrollerad miljö som kan utesluta vissa externa faktorer samtidigt som det finns större kontroll av de faktorer som ska ändras för att rättvist kunna jämföra artefakterna utifrån insamlad empirisk data. Den empiriska datan kommer samlas in med hjälp av ett Tampermonkey-skript.

Implementationen av respektive artefakt gick för mesta del bra, förutom vissa mindre hinder som enkelt gick att lösa. Bland annat fanns det problem med att förstå vad Stack Overflow datasetets alla värden faktiskt betydde. Utan en beskrivning av datan gällde det att bara analysera alla kolumner för att se hur respektive fil och värde höll ihop. Det som fokuserades på under utvecklingen var allt som behövdes till sökfunktionen eftersom det var den som skulle ligga i fokus.

Genomförandet av pilotstudien påvisade att Tampermonkey-skriptet kunde användas för att mäta på den tänkta variabeln med en uppsättning sökord. Resultatet sparades i en CSV-fil som automatiskt laddas ner när en mätserie är slutförd. CSV-filen kunde sedan importeras i Excel för att skapa diverse olika diagram.

2 Bakgrund

Detta kapitel går igenom underliggande begrepp och tekniker som är relevanta för denna studie.

2.1 Webbapplikationer

Internet har kommit till att bli en viktig plats för företag att visa produkter och förmedla information (Patel m.fl. 2013). Företag som tidigare har haft skrivbordsapplikationer flyttar dem till webben för att göra applikationer mer lättillgängliga på olika typer av enheter, enligt Kiruthika m.fl. (2016). Detta betyder att applikationer inte behöver skrivas om för varje typ av enhet, utan istället kan en webbapplikation göras responsiv och användas på alla typer av enheter som har en modern webbläsare.

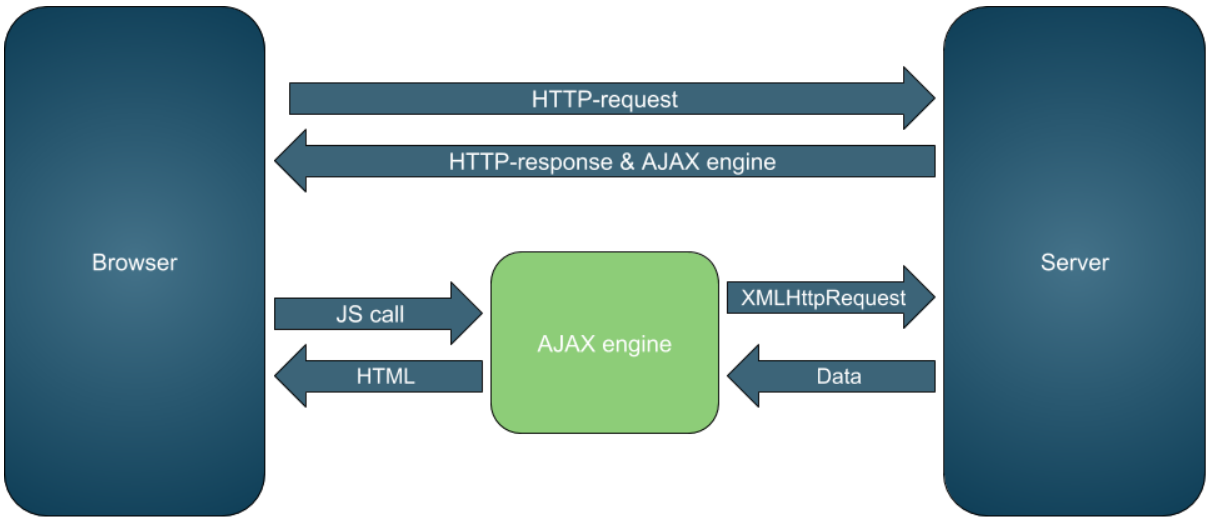
En webbapplikation är interaktiv och innehåller dynamiskt innehåll medans en webbplats är mestadels statisk (Kiruthika m.fl. 2016). Content Management System (CMS), som exempelvis Wordpress, Joomla och Drupal har blivit extremt populära på grund av enkelheten att skapa webbapplikationer utan några direkta webbkunskaper (Patel m.fl. 2013). Dessutom fann Patel m.fl. (2013) att de just tre nämnda CMS:en hade väldigt bra säkerhet och att det vanligaste sättet att hacka dem var genom osäkra tredjeparts plugins. Således kan CMS ses som en bra grund till en webbapplikation.

Dagens webbapplikationer är ofta single-page, vilket innebär att hela sidan laddas in en gång för att sedan endast byta ut huvudinnehållet på sidan (Mesbah & Deursen 2007). I praktiken kan det betyda att navigationsbaren och sidfoten aldrig laddas om. Mesbah och Deursen (2007) nämner också att single-page applikationer gör användarna mer nöjda med produkten genom att öka responsiviteten och interaktiviteten på sidan. Exempelvis så är det möjligt att göra en animation för att byta sida, vilket kan vara estetiskt tilltalande. För att kunna hämta en sida som användaren efterfrågar, utan att ladda om hela sidan, så måste AJAX användas.

2.2 AJAX

I webbapplikationer används Asynchronous JavaScript and XML (AJAX) för att kunna göra asynkrona HTTP-förfrågningar till webbservrar utan att behöva ladda om sidan (Brown A. & Harper 2011). Detta kan exempelvis användas för att ge förslag i formulär till användaren baserat på input eller att byta sida i single-page applikationer som nämndes tidigare. I sökmotorer är detta extremt användbart för användaren eftersom de kan få förslag på de mest sökta frågorna baserat på det som skrivs in i sökfältet. Användarupplevelsen påverkas negativt av siduppdateringar som inte använder AJAX eftersom tillstånd på sidan såsom formulärinput, scrollpositionen och annat återställs.

På figur 1 visas en tidslinje av en webbapplikation som använder AJAX för att hämta data från en webbserver efter första sidladdningen. AJAX är JavaScript beroende och använder ett XMLHttpRequest (XHR) objekt för att kommunicera med servrar. En initial sidladdning krävs för att kunna använda XHR objekt eftersom webbapplikationens AJAX funktioner ligger i tillhörande JavaScript filer.



Figur 1 Ett AJAX anrop till en webbserver efter initial sidladdning

I praktiken kan en HTTP-förfrågan gjord med AJAX se ut som på figur 2. Först och främst skapas det XHR-objektet som ska användas för att sedan sätta en callback funktion som ska hantera svaret från servern. Open funktionen kallas för att specificera vad som ska hämtas och med vilken HTTP-metod, exempelvis POST eller GET.

```
1   var httpRequest = new XMLHttpRequest();
2   httpRequest.onreadystatechange = function() {
3       if (this.readyState == 4 && this.status == 200) {
4           document.getElementById("data") = this.response;
5       }
6   };
7   httpRequest.open("GET", "data.txt", true);
8   httpRequest.send();
```

Figur 2 Ett AJAX anrop till en webbserver

Det kan dock finnas vissa svårigheter med att implementera AJAX i en webbapplikation. Exempelvis så måste servern kunna hantera förfrågningar som endast begär delar av en sida och generera en hel vy som innan (Fu m.fl. 2010). Det går att underlätta implementationen genom att exempelvis använda olika typer av serverramverk.

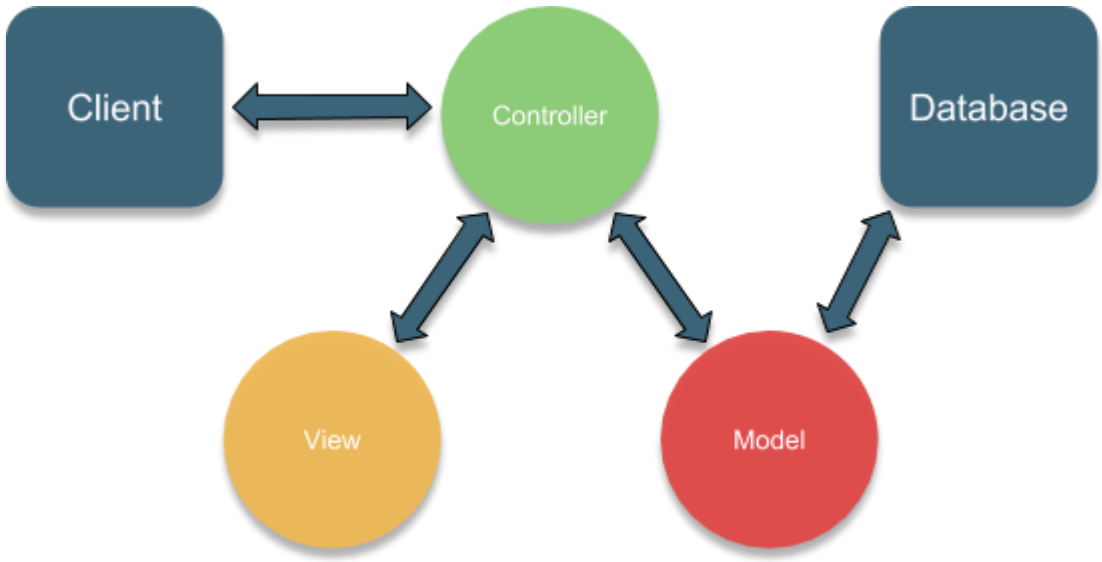
2.3 Ramverk

Ramverk är ett lager programkod som ligger ovanpå programmeringsspråket för att underlätta ytterligare för utvecklaren genom att bidra med en projektstruktur och bibliotek. Detta kan både underlätta och påskynda utveckling av programkod och är därför en nödvändig kunskap för att kunna välja ett lämpligt ramverk för en webbapplikation (Li m.fl. 2017). Dessutom menar Li, Karnan och Chishti (2017) att webbapplikationer blir säkrare och stabilare genom att använda ett beprövad ramverk. Ett sätt som ramverk förbättrar projektstrukturen är genom att implementera en arkitektur som MVC (Gupta m.fl. 2012).

En mjukvaruarkitektur är en övergripande beskrivning av systemets komponenter och relationen mellan dem och är menat att lösa vanliga problem med strukturen.

Mjukvaruarkitekturer tar bland annat hänsyn till effektivitet, tillförlitlighet och återanvändning (Coelho m.fl. 2005).

Webbapplikations-ramverk implementerar oftast en Model View Controller (MVC) struktur. Detta är att föredra eftersom MVC modulariseringen har bevisats vara effektivt och uppmuntrar kodåteranvändning (Li m.fl. 2017). På grund av modulaliseringen blir testbarheten av koden dessutom väldigt bra (Gupta m.fl. 2012).



Figur 3 MVC Arkitektur

På figur 3 så visas flödet av en MVC webbapplikation. Controllern tar emot förfrågningar av klienter och använder sig sen av en databasmodell för att hämta ut relevant information från databasen. Datan som finns tillgänglig passas vidare till en vy för att generera den sida som användaren har efterfrågat (Singh m.fl. 2018).

MVC-ramverk har på av just nämnda anledningar kommit till att bli en standard som serverplattform - det gör det intressant att undersöka om ett MVC-serverramverk implementerat i Python kan få bättre prestanda än ett implementerat i Node.js.

2.3.1 Node.js

Node.js är ett serverramverk som körs genom Googles V8 JavaScript motor. Ramverket är i synnerhet speciellt eftersom det tillåter utvecklare att skriva både klient- och serversidan, olikt andra ramverk, av en webbapplikation i JavaScript (Schutt & Balci 2016). Detta menar Schutt och Balci (2016) gör JavaScript till ett attraktivt språk för nybörjare då de endast måste lära sig det för att kunna hantera hela webbstacken.

Det finns ett antal olika Node.js ramverk som används för att hantera routing av inkommande HTTP-förfrågningar i en webbapplikation som också tillåter utvecklaren att specificera en egen templatemotor (Schutt & Balci 2016). Enligt Schutt och Balci (2016) är Express.js ett bra ramverk i det syftet eftersom att det är ett tunt, robust men samtidigt kraftfullt ramverk. De lyfter även fram Pug (Jade) som templatemotor, vilket går att integrera tillsammans med Express.js, för att skapa en kraftig backend. Det finns fall där templatemotorer inte behövs - exempelvis när klienten använder sig av en templatemotor som Angular.js (Schutt & Balci 2016).

```
1  <!DOCTYPE html>
2  html(lang="en")
3      head
4          title=title
5      body
6          h1=message
7          ul
8              each item in [1, 2, 3, 4, 5]
9                  li= 'Item: ' + item
```

Figur 4 Exempel av Pug (jade) syntax

På figur 4 visas ett exempel av en enkel Pug fil. Syntaxen är känslig för indentering, likt Python, då den avgör vilken nivå element ska placeras i. Titeln och texten på sidan sätts med hjälp av två variabler som skickas med när template motorn renderar sidan och en lista skrivs ut med hjälp av en for loop.

2.3.2 Django

Django är ett serverramverk skrivet i Python som följer en MVC arkitektur och körs genom Apache eller NGINX tillsammans med en Python modul. Django följer konventioner och MVC arkitekturen på ett sätt som gör det lättförståeligt även för oerfarna utvecklare att förstå strukturen (Schutt & Balci 2016). Vyerna genereras med Djangos egna templatemotor - Django Template Language, enligt Schutt & Balci (2016), och följer en syntax mer lik vanlig HTML än Pug. Figur 5 visar ett exempel som motsvarar det givet för Pug fast skrivet i Django Template Language.

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <title>{{ title }}</title>
5  </head>
6  <body>
7      <h1>{{ message }}</h1>
8      <ul>
9          {% for item in items %}
10             <li>Item: {{ item }}</li>
11         {% endfor %}
12     </ul>
13 </body>
14 </html
```

Figur 5 Exempel av Django Template Language syntax

Django inkluderar en Object Relational Mapper (ORM) som är en slags interface till databasen. Varje tabell i databasen representeras av en klass bestående av ett antal variabler och funktioner och används i controllern av applikationen för att manipulera databasobjekt.

2.4 Fulltextsökning och dataset

I databashanteraren PostgreSQL är *tsvector* en datatyp som representerar ett behandlat dokument som är optimerat för fulltext sökningar (Arslan & Yilmazel 2008). Vanliga ord som ”a” och ”we”, i engelska språket, indexeras inte i det behandlade dokumentet eftersom de används i nästan alla texter. Orden används dock för att avgöra relevans av diverse olika sökträffar (Bartunov & Sigaev 2005). Därav ger en sökning på ordet ”we” noll sökträffar trots att många rader i databasen kan innehålla ordet. På figur 6 visas ett behandlat dokument som är lagrad i en *tsvector* kolumn i en databas.

```
1   'code':7 'compil':4 'delphi':15 'intermedi':12
2   'languag':13 'mean':3 'tri':14 'win32':6
```

Figur 6 Ett behandlat dokument lagrad i tsvector datatyp

Tsvector datatypen kan även innehålla ett behandlat dokument av flera kolumner från respektive rad i databasen – alltså kan exempelvis titeln och kroppstexten av ett inlägg lagras i samma tsvector som ger möjligheten att söka på båda kolumnerna samtidigt.

PostgreSQL erbjuder två typer av index, vid namn GiN och GiST, som kan användas på tsvector datatypen för att drastiskt påskynda söktider av dokumentet. Användningsområdet skiljer sig för respektive index; GiN indexet tar ungefär 10 gånger så lång tid att bygga som GiST men erbjuder 3 gånger så snabb söktid, enligt Bartunov & Sigaev (2005). Således är GiN ett bra index att välja i applikationer där dokumentet för mesta del hålls statiskt, eller inte uppdateras för ofta.

Verklig data är i synnerhet viktigt att använda för att utföra verklighetstrogna tester eftersom genererad data inte är garanterad att ge ett resultat som efterliknar verkligheten och kan därför påverka söktider av datasetet. Stack Overflow, som är den mest populära Q&A-plattformen för utvecklare (Baltes m.fl. 2018), släpper öppna data dumpar som är fria att använda i alla syften och kan därmed användas i detta syfte.

```
1   <?xml version="1.0" encoding="utf-8"?>
2   <posts>
3       <row id="1"
4           PostTypeId="1"
5           AcceptedAnswerId="879"
6           CreationDate="2013-09-24T18:51:54.593"
7           Body="The question"
8           LastEditorUserId="6"
9           LastEditDate="2013-11-01T11:23:02.907"
10          LastActivityDate="2014-02-03T16:49:55.200"
11          Title="Title of the question"
12          Tags="tags"
13          AnswerCount="2"
14          CommentCount="0"
15          FavoriteCount="3" />
16  </posts>
```

Figur 7 Stack Overflow inlägg representerat som XML extraherat från det öppna datasetet.

På figur 7 visas ett exempel på hur Stack Overflow datan ser ut när den har extraherats. I detta fall visas endast ett inlägg och innehållet av frågan och titeln har förkortats. Utifrån den öppna datan så är det möjligt att återskapa frågor i det format som de visas i på Stack Overflow. Datadumparna innehåller, förutom det som nämnts ovan, även tillhörande användare, kommentarer, tags, röster med mera. Det som är mest relevant för att återskapa inlägg och kommenterar är tabellerna filerna Posts och Comments.

3 Problemformulering

Schutt & Balci (2016) jämför diverse olika ramverk för mjukvara i molnet genom att diskutera ett antal punkter. Genom analysen av språken för molnet, av Schutt & Balci (2016), så kan skillnaderna diskuteras ytterligare för Q&A-plattformar som innehåller ett heterogent dataset.

Enligt Ponzanelli m.fl. (2015) så har Q&A-plattformar kommit till att bli en viktig del av det dagliga arbetet för många utvecklare. Stack Overflow, som är den mest populära Q&A-plattformen för utvecklare, har miljontals användare som tillsammans skapar tiotusentals inlägg varje dag och kräver således snabba svarstider för att kunna hantera alla http-förfrågningar. Frågorna är i grunden heterogena som blandar naturligt språk tillsammans med programkod för att kunna visa det problem som de ställs inför. Svar kan ges i form av kodförslag eller generella tankar kring lösningar till exempelvis stack traces (Ponzanelli m.fl. 2015). Stack Overflow erbjuder ett fritt tillgängligt dataset som används av många forskare eftersom det innehåller text, kodstycken, olika typer av konfigurationsfiler och stack traces. Att generera ett verkligt heterogent dataset, likt Stack Overflow, kan vara svårt och därför kan det vara fördelaktigt att använda ett öppet dataset av den naturen (Ponzanelli m.fl. 2015).

Quality of Service (QoS) är en term som används för att mäta kvalitén av en hemsida genom att ta hänsyn till responstid, tillgänglighet och tillförlitlighet (Kulnarattana & Rongviriyapanish 2009). Det är extra viktigt att upprätthålla hög QoS för webbapplikationer, för att behålla användarna, eftersom de ofta blir frustrerade när sidor tar för lång tid att ladda (Butkiewicz m.fl. 2011). Det tar en enorm mängd tid och energi att bygga en webbapplikation och således blir det viktigt att göra rätt från början innan användare hinner göra en negativ bedömning av webbapplikationen (Hsieh m.fl. 2008).

Utvecklare underlättar sitt arbete genom att använda sig av ramverk och bibliotek. Därför är det väldigt viktigt att kunna välja ett ramverk som passar den applikation som ska byggas (Li m.fl. 2017). **Problemet** ligger i svårigheter när det kommer till vilket ramverk som ska väljas eftersom utvecklare ifrågasätter vad som är bra och dåligt med olika ramverk och fastnar i en process där de kanske inte kan bestämma sig. Det är speciellt svårt eftersom det inte är hållbart att jämföra alla ramverk som finns tillgängliga och själv försöka göra en bedömning av vilket som passar bäst (Laakso & Niemi 2008). Detta arbetet kan hjälpa till i beslutningsprocessen för utvecklare genom att jämföra två av serverramverken i en specifik tillämpning.

Schutt & Balci (2016) jämförde Node.js och Django för att undersöka QoS i en webbapplikation. En aspekt av QoS, Svarstider, kommer nu även undersökas för en identisk implementerat Q&A plattform i webbramverken Node.js och Django. Likheter går att se i arbetet av Lei m.fl. (2014) som genomförde ett teknikorienterat experiment för att jämföra laddtider av Node.js, Python och PHP, i ett antal olika scenarion, på ett sätt som kommer att likna detta arbete.

Frågeställningen: Vilket ramverk av Django och Node.js ger bäst söktider av frågor för en Q&A-plattform utifrån Stack Overflow data lagrad i en PostgreSQL databas som hanterar fulltextsökningar?

H_0: Valet av ramverk ger inte en signifikant skillnad för söktider av inlägg på en Q&A plattform utifrån Stack Overflow data lagrad i en PostgreSQL databas som hanterar fulltextsökningar.

H_1: Valet av ramverk ger en signifikant skillnad för söktider av inlägg på en Q&A plattform utifrån Stack Overflow data lagrad i en PostgreSQL databas som hanterar fulltextsökningar.

3.1 Metodbeskrivning

Ett empiriskt experiment är den metod som kommer att användas för att besvara frågeställningen i arbetet. Anledningen till detta är för att testerna utförs i en sluten och kontrollerad miljö så att utomstående faktorer minimeras - vilket gör det optimalt (Wohlin 2012). Resultatet av ett experiment behöver dock inte nödvändigtvis vara korrekt eftersom det finns ett stort antal saker som kan gå fel och ge missvisande resultat (Berndtsson m.fl. 2008). Experiment kan vara teknikorienterade eller människoorienterat. I teknikorienterade experiment utförs exempelvis mätningar på en variabel före och efter en förutbestämd faktor för att kunna se den prestandaskillnad som skapas (Wohlin 2012). Detta menar Wohlin (2012) är bra eftersom ett program kan utföra mätningarna utan olika fördomar som kan påverka resultatet. Människoorienterade experiment går inte att kontrollera i samma grad som ett teknikorienterat på grund av den mänskliga faktorn (Wohlin 2012).

Variabler som är intressanta att undersöka är responstider av sökningar i applikationen, tid en SQL-fråga tar att köras och hur lång tid en vy tar att genereras i respektive ramverk. För sökningar kan antalet inlägg, som finns i databasen, vara en intressant faktor för att ta reda på hur respektive ramverk skalar. Hur lång tid en SQL-fråga tar att köra och en vy att genereras är också intressant eftersom databasdrivers och templatemotorerna jämförs. På så sätt är det möjligt att hitta delar av applikationen som kör långsamt och kanske utgör stora delar av en respons rent tidsmässigt.

I detta arbete kommer söktider av Stack Overflow data i en identiskt webbapplikation implementerat i ramverken Django och Node.js mätas för att statistiskt avgöra vilket som presterar bäst, om något. Svarstiden räknas ut genom att beräkna tiden det tar från att användaren klickar sök tills sökresultatet visas på sidan och innehåller således tiden det tar för respektive ramverk att kontakta PostgreSQL databasen och generera vyn. Resterande nämnda variabler och faktorer testas i mån av tid.

3.2 Alternativa metoder

Fallstudie är en alternativ metod som skulle kunna appliceras. En fallstudie används för att undersöka ett antal olika fall, ibland bara ett, i en verklig miljö under en begränsad period (Wohlin 2012). Wohlin (2012) påpekar också att fallstudie mestadels används när området som fenomenet faller inom är okänt och kan i det fallet användas för att utforska ny mark. Nackdelen med en fallstudie jämfört med ett teknikorienterat experiment är att miljön inte är under samma kontroll och kan därför påverka resultatet. Dessutom kan resultatet vara svårt att generalisera och kan, enligt Wohlin (2012), påverkas av partiska forskare. För att kunna rättvist jämföra ramverken i detta arbete så måste det finnas kontroll över hårdvaran och implementationen av programkoden och därför blir det svårt att kunna applicera detta i en fallstudie.

Ett intressant sätt att applicera fallstudie i denna studie hade vart att istället undersöka hur ett seniort utvecklingsteam upplever och hanterar utvecklingen av programkod i respektive ramverk i verkligheten. Från det går det att se vilken teknik som erbjuder bra med bibliotek och hur lång tid utvecklingen tar för typisk funktionalitet. På så sätt kan andra utvecklingsteam få reda på tankar kring respektive ramverk från erfarna utvecklare.

3.3 Forskningsetik

Det finns ett antal etiska aspekter som är värt att reflektera över i arbetet. Stack Overflow datan, som har publicerats under Creative Commons CC-BY-SA 3.0 licensen, är fri att använda i alla syften, men det kan ses olämpligt att med data som kan identifiera riktiga användare på sidan. Exempelvis går det att ta användar id:t och öppna personens riktiga profil på Stack Overflow. Den personliga integriteten tas på allvar och därför kommer delar av datan, som går att koppla till användare, filtreras bort innan den läggs in i en lokal databas inför testning då de kanske inte vill synas i detta syfte, trots att datan är öppen. Detta inkluderar bland annat användarnamn och profilbilder. Själva kommentarerna på frågor kommer fortfarande att finnas kvar och därmed går det att löst koppla användare till inlägget, men det är riktigt svårt att undvika. Profilbilder filtreras bort eftersom det kan identifiera en användare och för att profilbilderna ligger på diverse olika Content Delivery Networks (CDN) och kan därför påverka laddtider oförutsägbart.

Alternativt hade det gått att generera egen data i testningssyfte, men det introducerar svårigheten att garantera datan håller ett verkligt format. Fördelen med Stack Overflow datan är just det - den kommer från en riktig plattform som har många aktiva användare.

För att uppnå replikerbarhet av arbetet så kommer programkoden för respektive webbapplikation publiceras på GitHub så att flödet av utvecklingen går att följa. Specifikationer av hårdvara och mjukvara kommer att specificeras i både rapporten och på GitHub. Onödig funktionalitet av operativsystem på webbservern kommer att stängas av så att det inte påverkar mätningarna negativt.

4 Genomförande

4.1 Förstudie

Inspirationen till denna applikation kommer från flera olika källor och respektive källa beror mycket på vilken del av applikationen som skulle implementeras. När det kom till den tekniska artefakten som skulle implementeras i Django så visade sig boken *Django by Example* (Mele 2015) vara mycket behändig genom att ge enkla, så väl som komplicerade, kodexempel. Varje kapitel representerar utveckling, eller vidareutveckling, av olika applikationer och ger därför många olika tillvägagångssätt på hur problem kan lösas. Även Djangos egna dokumentation (Django 2019a) ger kodexempel så väl som utförliga förklaringar av olika Django koncept och var den sida som användes mest under implementationen. Ett komplett exempel av en röstningsapplikation finns också för att erbjuda en helhet av en applikation.

Node.js artefakten var lättare att påbörja med tanke på mängden JavaScript som har används under utbildningsprogrammet, men boken *Web Development with Node Express* (Brown E. 2014) ger bra insikt i hur Express funkar i en webbapplikation. Dock så tar boken även upp orelaterade aspekter som olika textredigerare, men den förklarar exempelvis request och reponse objects på ett utförligt sett, vilket är en essentiell del av en Express applikation.

Boken *Programming Web Applications with Node, Express and Pug* (Krause 2017) förklarar template motorn Pug och dess byggstenar på ett väldigt utförligt sätt och är således en bra kompletterande bok till den förstnämnda Node.js boken som endast nämner Jade som template motor. Express har, likt Django, väldigt bra dokumentation (Express 2019) som ger bra kunskap för att kunna ta emot förfrågningar, rendera en vy, byta template motor och använda middlewares.

Stack Overflow (Stack Overflow 2019) har använts för att hitta många lösningar till de problem som har uppstått under implementationens gång. De frågor som har vart särskilt användbara är ”How to do SELECT COUNT(*) GROUP BY and ORDER BY in Django?” (ninja 2019), ”Using a Django model method from a template” (Anteru 2019) och ” Node.js Express3 - Middleware to add render data to all render requests” (Magaluk 2019).

Till databasimplementationen behövdes inte så mycket inspiration eftersom att Djangos ORM skapar databasen och således var det endast mindre grejer som utfördes i testningssyfte innan de implementerades i Django. I de fall som svar behövdes var PostgreSQL egna dokumentation (PostgreSQL 2019) väldigt utförlig, speciellt när det kom till fulltextsökningar och olika typer av index som kunde användas för att förbättra prestandan av sökningar.

Från det urval kurser som Högskolan i Skövde erbjuder så gav Webbprogrammering mycket av inspirationen till detta arbete. I kursen så implementerades ett bokningssystem i ett single-page format med hjälp av AJAX. Dock gavs API-delen till uppgiften och således var det endast front-end delarna som behövde implementeras för att göra och visa bokningar. Även kursen XML API gav en komponent till helheten av detta arbete genom att ge grunderna i att läsa av och presentera XML data från ett ibland okänt format, som utfördes i projektuppgiften.

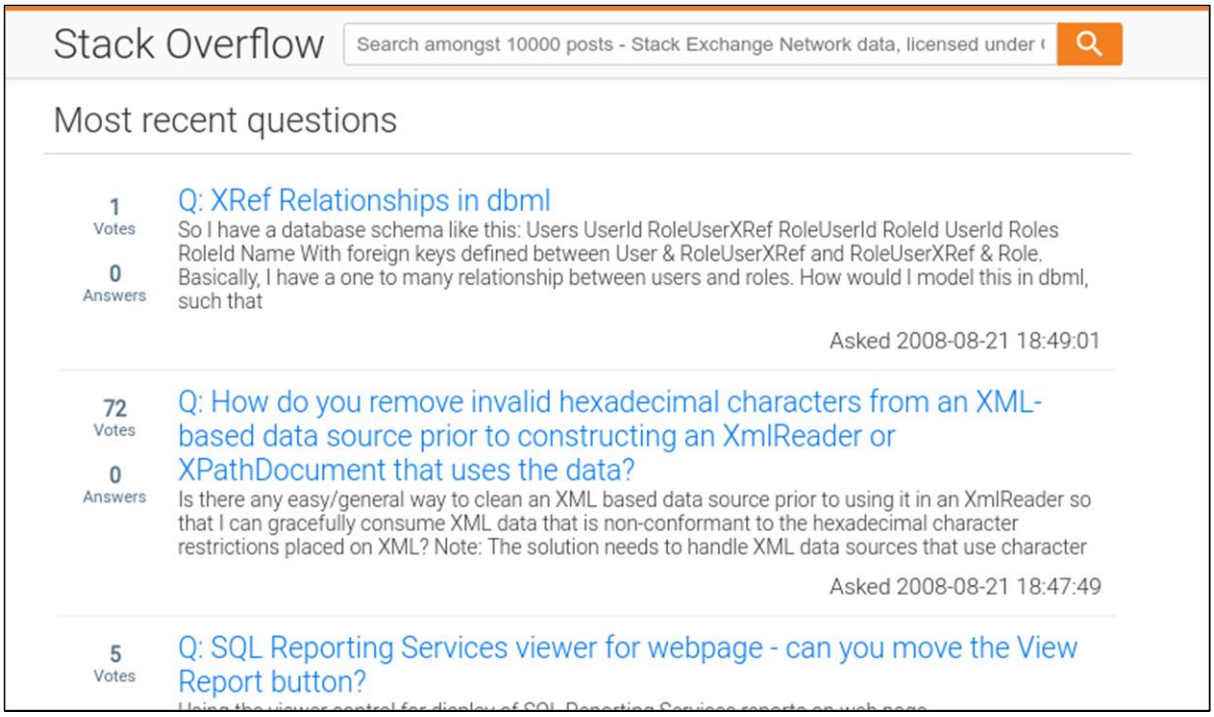
Till implementationen gav de just nämnda kurserna samt Webbutveckling – Mobilapplikation mycket användbar kunskap för att kunna läsa av och presentera XML data i en mobilanpassad webbapplikation som använder AJAX.

4.2 Implementation

4.2.1 Applikationerna

Under hela implementationens gång har sökfunktionen legat i tankarna eftersom det är huvudsakligen den som ska mätas på i pilotstudien och därför var det viktigt att resten av artefakterna var korrekt strukturerade när sökfunktionen väl skulle implementeras.

Utvecklingen började med att utveckla utseendet av Django applikationen med Django Template Language (DTL) för att sen konverteras till Pug så att utseendet såg identiskt ut. I detta stadie var det bara statiska strängar som användes på sidan eftersom Stack Overflow datan inte hade lagts in i databasen. På figur 8 går det att se startsidan av den slutgiltiga designen tillsammans med den sökfunktion som ska mätas på. Tanken var att designen skulle vara enkel och påminna om Stack Overflow samt att göra den mobilanpassad.



Figur 8 Slutgiltigt utseende på artefakten

Med en färdig design påbörjades utvecklingen av de modeller som Djangos ORM använder sig av för att skapa och göra förfrågningar mot databasen. Till detta var Djangos dokumentationen, närmare bestämt ”Model Field Types” (Django 2019b) till stor användning för att kunna välja rätt datatyper för att lägga in Stack Overflow datan. I samband med detta analyserades även datasetet för att avgöra vad som faktiskt skulle användas så att rätt tabeller kunde skapas och kunna lagra datan.

Controllern, som höll i den tidigare statiska datan, uppdaterades för att använda sig av de nya modellerna för att hämta inlägg. Detta var en väldigt enkel process eftersom den statiska datan redan var korrekt formaterad och därmed behövde inget extra arbete utföras i vyerna för att rendera datan.

Till slut implementerades fulltextsökning, ett GIN-index samt JavaScripten för sökfunktionen på sidan vilket behövdes för att kunna utföra pilotstudien på. I samband med

fulltextsökningsstödet konverterades också hela Django applikationen till Node.js. Vyerna innehåller väldigt lite kod och var därför inte svåra att konvertera över. Det som inte fanns från Django applikationen var de sökfrågor som Node.js applikationen behöver. Django har möjligheten att skriva ut sökfrågorna som är på väg att köras och användes därför för att jämföra de planerade sökfrågorna för Node.js. Dessa visade sig vara identiska och därför bör inte skillnader i prestanda bero på en dålig optimerad sökfråga utförd av Djangos ORM.

Dessa applikationer är inte direkt komplicerade, men det som har vart svårast att implementera kommer diskuteras närmare nedan. All kod finns även tillgänglig på projektets GitHub sida (Examensarbete 2019).

4.2.2 Datautvinning från Stack Overflow dataset

Stack Overflow datasetet innehåller en otroligt stor mängd data, endast inläggen på sidan var totalt 70GB, och därför behövde en mängd inlägg och kommenterar extraheras och lagras i mindre filer för att lättare kunna hanteras. Detta gjordes väldigt lätt genom att använda programmet ”head” i terminalen för att kunna extrahera de första X-tusen raderna av både inlägg och kommentarer.

```
head -n 1000002 posts.xml > extractedPosts.xml
echo "</posts>" >> extractedPosts.xml
```

Figur 9 Kommando för att extrahera första 1 000 002 raderna från filen posts.xml och lagra i filen extractedPosts.xml

På figur 9 går det att se programmet ”head” i action som extraherar de första 1 000 002 raderna från en XML fil som sen lagras i en ny XML fil. Anledningen till att det är just 1 000 002 inlägg istället för en jämn miljon är för att de två första raderna av en XML är specifikationen av XML samt öppningen för rotnoden; alltså hade det resulterat i 999 998 inlägg. Genom att extrahera datan med ”head” så kommer inte rotnod att vara stängd och är därför inte i ett giltigt XML format. För att fixa det så körs den andra raden som echoar ut stängningsnoden i slutet av filen. Det är betydligt enklare att köra echo istället för att försöka öppna en fil på 2GB i en textredigerare eftersom många textredigerare inte kan hantera att öppna så stora filer. Samma procedur utfördes också för att extrahera 5 000 000 tillhörande kommentarer på inlägg.

4.3 Progression

4.3.1 Databas och fulltextsökning

Eftersom Django använder en Object-Relational Mapper (ORM) som automatiskt skapar en databas togs valet att Django ska få skapa databasen men att båda applikationer använder den. Tanken med det var att minska risken för att databasen ser olika ut mellan applikationerna som därmed hade kunnat skapa en prestandaskillnad som inte beror på ramverken.

Databashanteraren PostgreSQL använder en datatyp som heter tsvector för att göra ett textstycke sökbart med fulltextsökningar och en annan vid namn plainto_tsquery för att skapa en sökning mot en tsvector. Båda dessa kan skapas i en SQL-fråga, men tsvector kan också skapas som en kolumn i databasen. Eftersom det ska vara möjligt att söka på både titeln och innehållet av frågan så valdes en kolumn som då innehåller en gemensam tsvector för båda kolumnerna. Problemet med att ha en separat kolumn är att den inte uppdateras när

exempelvis titeln eller frågan uppdateras, vilket resulterar i att sökningar blir inkorrekta. PostgreSQL dokumentationsida hade en lösning på detta, vid namn ”Creating indexes” (PostgreSQL 2019) genom en trigger som automatiskt uppdaterar ts_vector fältet när antingen titeln eller kroppen av frågan uppdateras.

```
3   from django.db import migrations
4
5
6   class Migration(migrations.Migration):
7
8       dependencies = [
9           ('questions', '0007_auto_20190331_1324'),
10      ]
11
12      operations = [
13          migrations.RunSQL(
14              "CREATE TRIGGER tsvectorupdate BEFORE INSERT OR UPDATE"
15              + " ON questions_post FOR EACH ROW EXECUTE PROCEDURE"
16              + " tsvector_update_trigger(search_vector, 'pg_catalog.english', title, body);"
17          )
18      ]
```

Figur 10 En Django migration som automatiskt skapar en trigger på sökfältet när en databas skapas

Om triggern skulle skapas manuellt på databasen så hade det behövt göras manuellt varje gång den byggs om, därför har en migrationsfil skapats som Django automatiskt applicerar när en databas skapas på en ny maskin eller en redan befintlig databas ska uppdateras. På figur 10 visas migration 0008 av applikation som automatiskt skapar en tsvectorupdate trigger som PostgreSQL dokumentationssida rekommendera. Ett Generalized Inverted Index (GIN) index skapades även för tsvector kolumnen för att optimera söktider av datasetet. Hela lösningen och implementationen av fulltextsökning tillsammans med ett GIN-index går att i se i commit b4ab7be[1].

4.3.2 Stack Overflow dataset

Datasetet kom inte med en beskrivning på vad alla värden faktiskt betyder och det resulterade i att datan behövde analyseras för att förstå hur den hängde ihop. Alla inlägg på sidan sparas exempelvis i samma tabell, men alla rader har inte samma attribut ifyllda. Det upptäcktes relativt snabbt att uppsättningen av attribut som är ifyllt bestäms av PostTypeId vilket bestämmer om ett inlägg är en fråga eller ett svar på en fråga. Varje inlägg som hade PostTypeId=1 har också attributet AcceptedAnswerId och title vilket gör det till en fråga medans PostTypeId=2 inte har en titel, men har istället ett ParentId attribut som är en FK till id:t av den frågan.

Kommenterar på inlägg var enklare att förstå sig på eftersom varje rad innehåller samma kolumner och har bara en FK till id:t av raden. Datasetet innehåller mer data än det som precis förklarats, men resten har inte använts till de tekniska artefakterna.

4.3.3 Lägga in Stack Overflow data i databasen

Med den extraherade datan från implementationsdelen skrevs ett Python-skript som läser av XML filerna och lägger in ett specificerat antal inlägg i databasen samt alla kommentarer.

[1] https://github.com/c16matan/examensarbete/commit/b4ab7be

Filen med inlägg innehåller i detta stadie 1 000 000 inlägg och i Python-skriptet går att det specificera att exempelvis 10 000 ska läggas in i databasen.

Som tidigare nämnts så innehåller svar till frågor ett attribut som är en FK till id:t av frågan, och i XML filen är raderna sorterade på id. Konstigt nog finns det svar som pekar på ett inlägg som inte finns. Exempelvis så pekar kanske inlägg med id 5 mot fråga med id 10, men om då XML filen parsas i logisk ordning så finns inte inlägg 10 när inlägg 5 ska skapas vilket skapar ett FK constraint fel. Detta togs till hänsyn i skriptet genom att hoppa över de rader som inte går att skapa på grund av det just nämnda felet med hjälp av en enkel if-sats som går att se på figur 11 och även i commit b4ab7be[2].

```
1   for event, element in parser:
2       if event == 'end' and element.tag == 'row':
3           if 'ParentId' in element.attrib and \
4                   int(element.attrib['ParentId']) > int(element.attrib['Id']):
5               element.clear()
6               continue
```

Figur 11 If-sats för att hoppa över inlägg som skulle ge FK constraint fel

XML datan itereras igenom som en stream med hjälp av Pythons cElementTree klass vilket är att föredra framför att ladda hela filen till minnet eftersom det är betydligt snabbare vilket märktes i ett tidigt stadie av skriptet vilket resulterade i att den tidigare versionen av skriptet inte pushades till GitHub.

4.4 Pilotstudie

Innan de riktiga mätningarna körs så har en pilotstudie utförts för att se till att upplägget av databehandling och att applikationerna kan mätas på ett tillförlitligt. Tanken är att detta kan hjälpa till att hitta fel i processen och utförs därför på en mindre mängd data och med ett mindre antal mätpunkter eftersom den statistiska signifikansen inte är relevant i detta fall. Inför pilotstudien fylldes PostgreSQL databasen med 10’000 inlägg samt tillhörande kommenterar.

Ett mätskript har implementerats i Tampermonkey för att automatisera processen av att mäta hur lång tid olika sökningar tar med hjälp av *performance.now()* funktionen vilket retunerar en tid som tar precision upp till mikrosekunder (Mozilla 2019). Detta gör timern lämplig att använda för att göra mätningar av hög precision. Mätskriptet använder detta till sin fördel genom att lagra *performance.now()* precis när en sökning sker och sen en gång till när resultatet kommit tillbaka och placerats på sidan. Laddtiden kan beräknas genom att ta värdet som hämtas ut i samband med resultatet minus startvärdet och kan sen lagras undan för att sen kunna laddas ner. För att avgöra när AJAX-förfrågningar har fått svar och placerats på sidan används en MutationObserver för att köra en funktion i skriptet när DOM trädet på sidan uppdateras inuti en specifik nod. När svaret behandlats och laddtiden beräknats görs en ny sökning till ett visst antal sökningar har gjorts.

Pilotstudien utfördes i en helt lokal miljö, utan internetuppkoppling, där både klienten och servern kör på maskin för att undvika utomstående faktorer. Hårdvaruspecifikationen för systemet presenteras i tabell 1 tillsammans med operativsystemet och webbläsaren. Inför

[2] https://github.com/c16matan/examensarbete/commit/b4ab7be#diff-91ebbe8

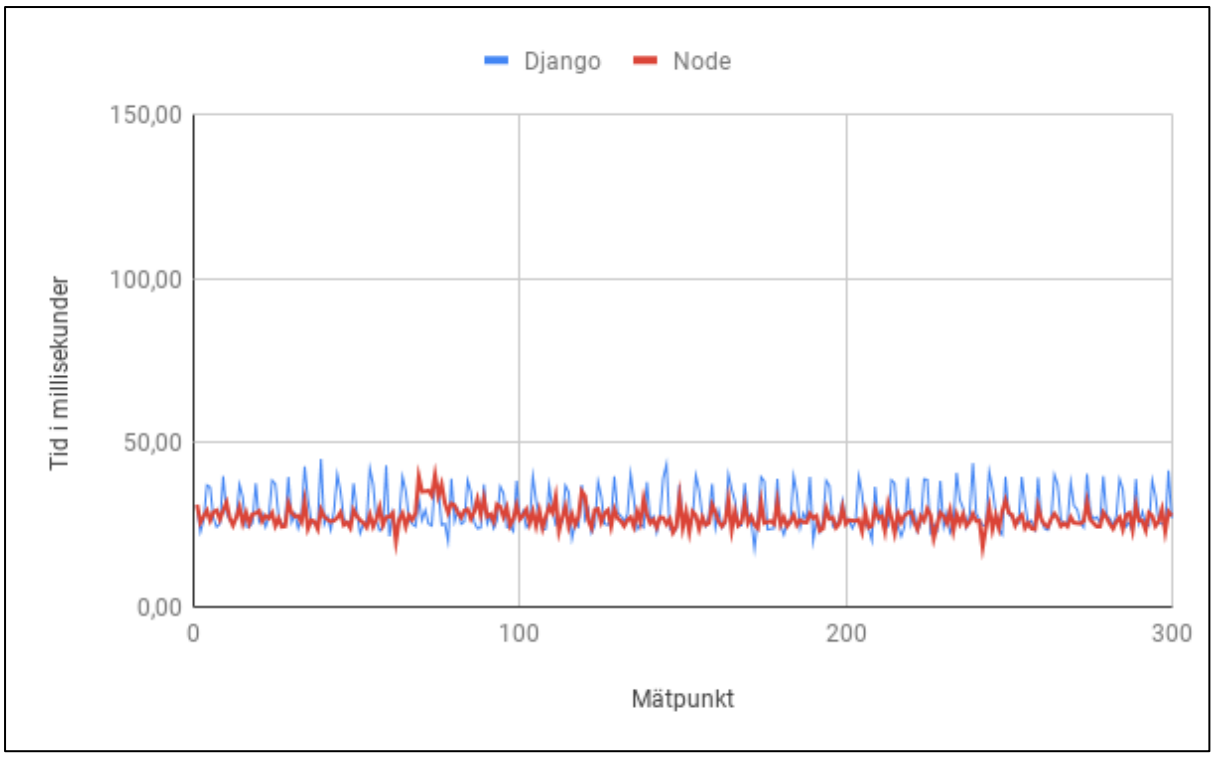
mätningen stängdes alla öppna program i operativsystemet förutom ett Chromium fönster som endast har en flik öppen och tillägget Tampermonkey aktiverat för att kunna utföra mätningarna. I detta stadie kördes ramverken genom utvecklingsservern som respektive ramverk erbjuder.

Hårdvara/Mjukvara	**Modell/Version**
Operativsystem	Arch Linux - GNU/Linux
Google Chromium	73.0.3683.103 (Official Build) (64-bit)
PostgreSQL	11.2
Moderkort	ASUS Z97-a
Processor	Intel Core i7 4790k @ 4.40GHz
Grafik	NVIDIA GeForce GTX 970
RAM	Corsair 16GB (2x8GB) DDR3 CL10 1600MHz
Lagring	Corsair Force GS 240GB

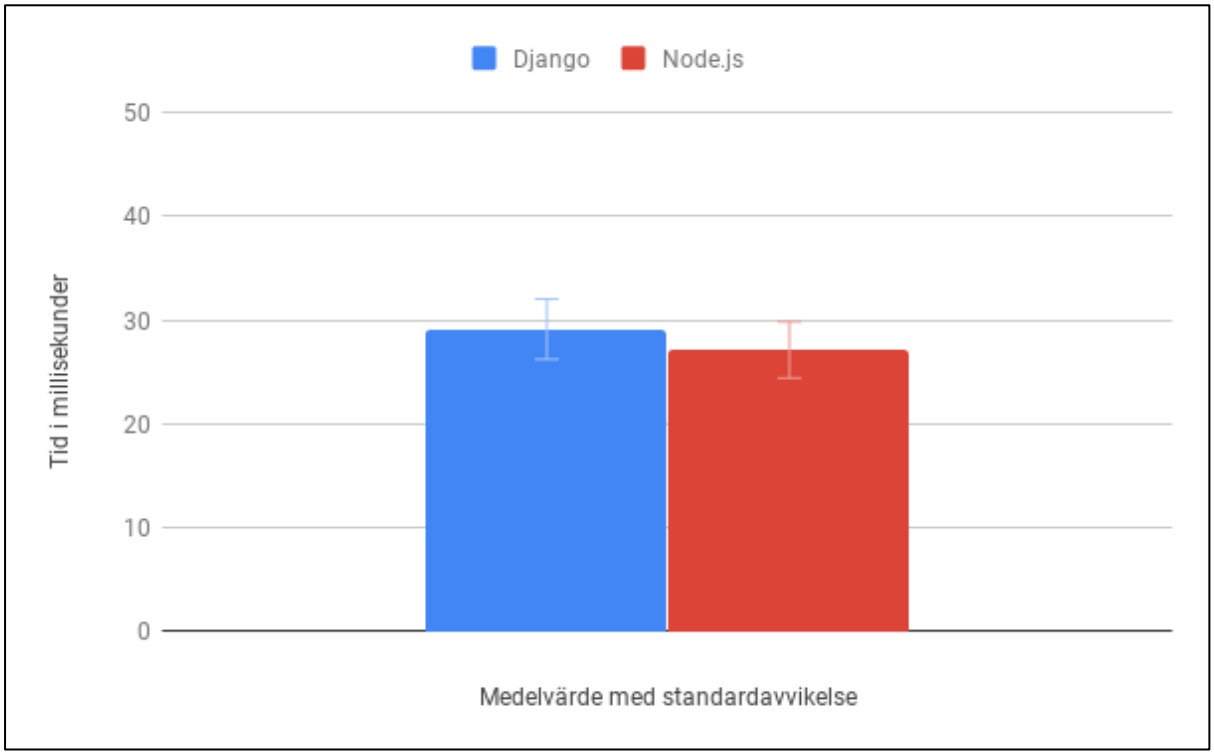
Tabell 1 Mjukvaru- och hårdvaruspecifikation av mätdator

Resultatet för respektive ramverk går att se i form av ett linje- och stapeldiagram på figur 12-13. Mätpunkterna ser lovande ut och det verkar vara relativt jämnt när det gäller sökprestandan i ramverken, men det går inte att bevisa att något av ramverken presterar bättre i detta stadiet eftersom antalet mätpunkter är för lågt. Söktiden för Django och Node.js var 29,14 respektive 27,16 millisekunder och var därmed väldigt nära, men återigen, pilotstudien används endast för att säkerhetsställa att processen för att mäta applikationen fungerar. Här är det också tydligt att båda ramverk har överlappande mätpunkter som går att se på standardavvikelsens intervall – Alltså kan det vara svårt att avgöra om ett mätvärde av 29,44 tillhör ett av ramverken.

Mätningarna kunde förövrigt utföras utan problem för båda artefakterna och gick lätt att importera i Google Kalkylark för att skapa olika diagram. När CSV filen importeras är det viktigt att välja en annan avgränsare än ett komma, exempelvis ett streck, så att mätvärdena importeras med decimaler. Om en annan avgränsare inte väljs så separeras decimalerna mot antalet millisekunder och placeras i två olika kolumner.



Figur 12 Resultat av 300 sökningar i pilotstudien



Figur 13 Resultat från 300 sökningar i pilotstudien

5 Utvärdering

5.1 Uppdatering av mätskript

Efter utförd pilotstudie uppdaterades mätskriptet för att spara mer information för varje mätpunkt. Till pilotstudien sparades endast söktiden, men nu sparas även sökordet tillsammans med antal returnerade svar. Tanken bakom denna uppdatering var att kunna göra en mer grundlig analys av respektive mätserie och kunna specificera hur respektive ramverk hanterar olika situationer.

Skriptet uppdateras även med 490 nya sökord, nu totalt 500, som används sekventiellt vid mätningar. Processen för att ta fram sökorden var att försöka hitta sökord som faktiskt skulle kunna användas på Stack Overflow, och andra plattformar med ett fokus på programmering. Exempel på detta är bland annat reserverade nyckelord av olika programmeringsspråk, designmönster, tekniker och verktyg.

5.2 Flytt till produktionsmiljö

För att göra en rättvis bedömning av ramverkens prestanda så behövde de flyttas till en produktionsmiljö, och inte köras från utvecklingsservern av respektive ramverk, som de gjorde under pilotstudien. Webbservern NGINX installerades för att hantera HTTP förfrågningar av statiskt innehåll såsom bilder, CSS, JavaScript och fonter till båda ramverken.

Till Django krävs även ett paket som heter Gunicorn och finns tillgängligt via pip, pakethanteraren till Python, för att skapa en socket-fil som NGINX använder för att kunna svara med det dynamiska innehåll som Django genererar. Pm2 är ett paket som kan starta, visa information och automatiskt starta om en Node.js applikation om den skulle krascha av någon anledning.

Pm2 som paket kräver ingen särskild konfiguration, utan det räcker med att starta applikationen och välja att starta pm2 som en service med datorn. Till NGINX skapades en konfiguration för respektive ramverk som hanterar förfrågningar av statiskt innehåll såväl som förfrågningar av dynamiskt innehåll. NGINX konfigurationen för Node.js skickar vidare förfrågan till den port som Node.js applikationen kör på lokalt för att sedan returnera innehållet. NGINX konfigurationen för Django, Node.js samt Gunicorn finns tillgängligt i appendix B, C och D. Förutom de nämnda ändringarna så är miljön identiskt med den som pilotstudien utfördes på.

Mjukvara	**Version**
NGINX	1.16.0
Gunicorn	19.9.0
Pm2	3.5.0

Tabell 2 Mjukvaruspecifikation av produktionsmiljö

5.3 Presentation av undersökning

Med en uppdaterat miljö och mätskript var dags att utföra experimentet utifrån ett antal olika faktorer för att kunna besvara frågeställningen. Till detta har två primära faktorer valts. Den första faktorn är antalet inlägg som finns i databasen; tanken är att kunna göra en analys från antalet returnerade inlägg och faktiskt visade inlägg. Som nämnts ovan så visar applikationen som mest 50 inlägg av en sökning på grund av en LIMIT som satts på SQL-frågan, men den totala mängden sökträffar skrivs också ut på sidan. På så sätt kan en analys göras utifrån olika sökningar där båda returnerar 50 inlägg, men totala mängden sökträffar skiljer sig med tusentals.

Den andra faktorn är antalet tecken som visas för i kroppstexten för varje sökresultat på sidan. Standarden för applikationen är 300 tecken per inlägg, men denna kommer även halveras till 150 tecken för att se skillnader mot mängden inlägg som returneras. Detta resulterade i 3 planerade mätserier per ramverk men det hade kunnat vara fler beroende på resultatet. Det behöver exempelvis inte vara nödvändigt att utföra fler mätserier där antalet inlägg skiljer ytterligare för första faktorn om skillnaden är minimal utifrån ökningen av 90’000 inlägg. De planerade mätserierna som kommer utföras på båda artefakterna presenteras i tabell 3. Varje mätserie kommer innehålla 20’000 mätpunkter för att kunna svara på frågeställningen med en viss säkerhet.

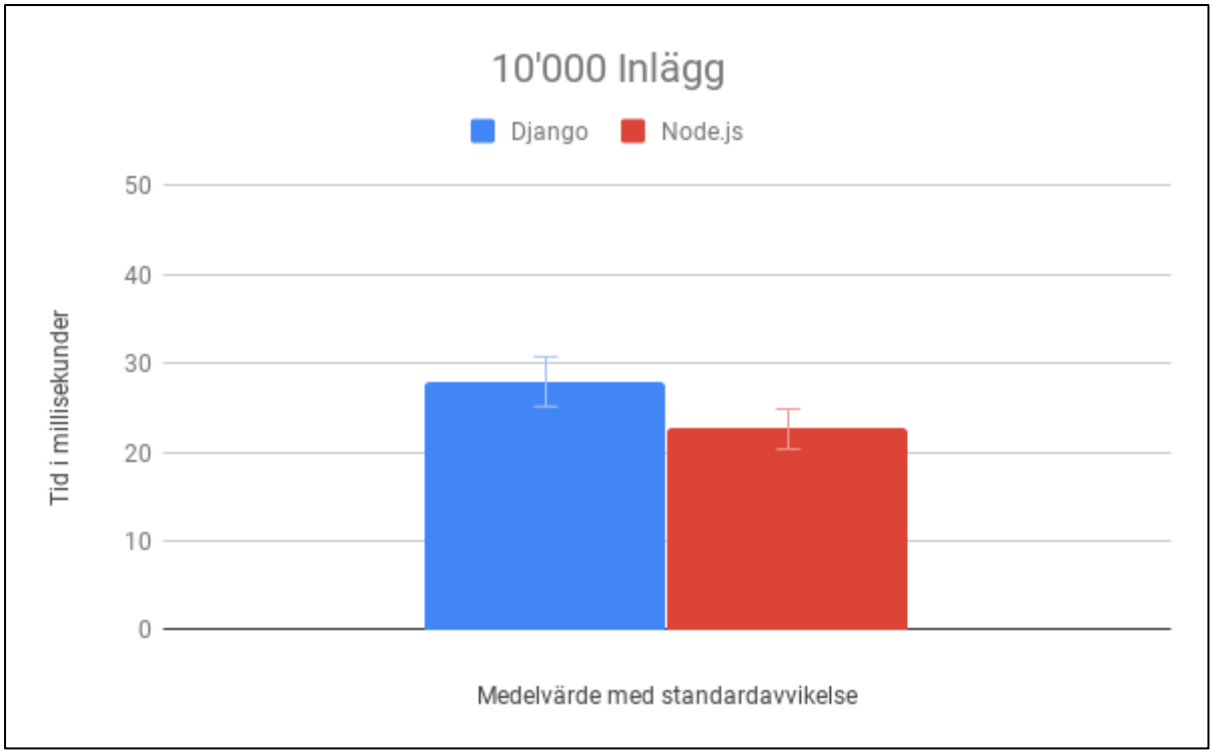
Mätserie
10’000 inlägg, 300 tecken kroppstext
100’000 inlägg, 300 tecken kroppstext
100’000 inlägg, 150 tecken kroppstext

Tabell 3 Mjukvaruspecifikation av produktionsmiljö

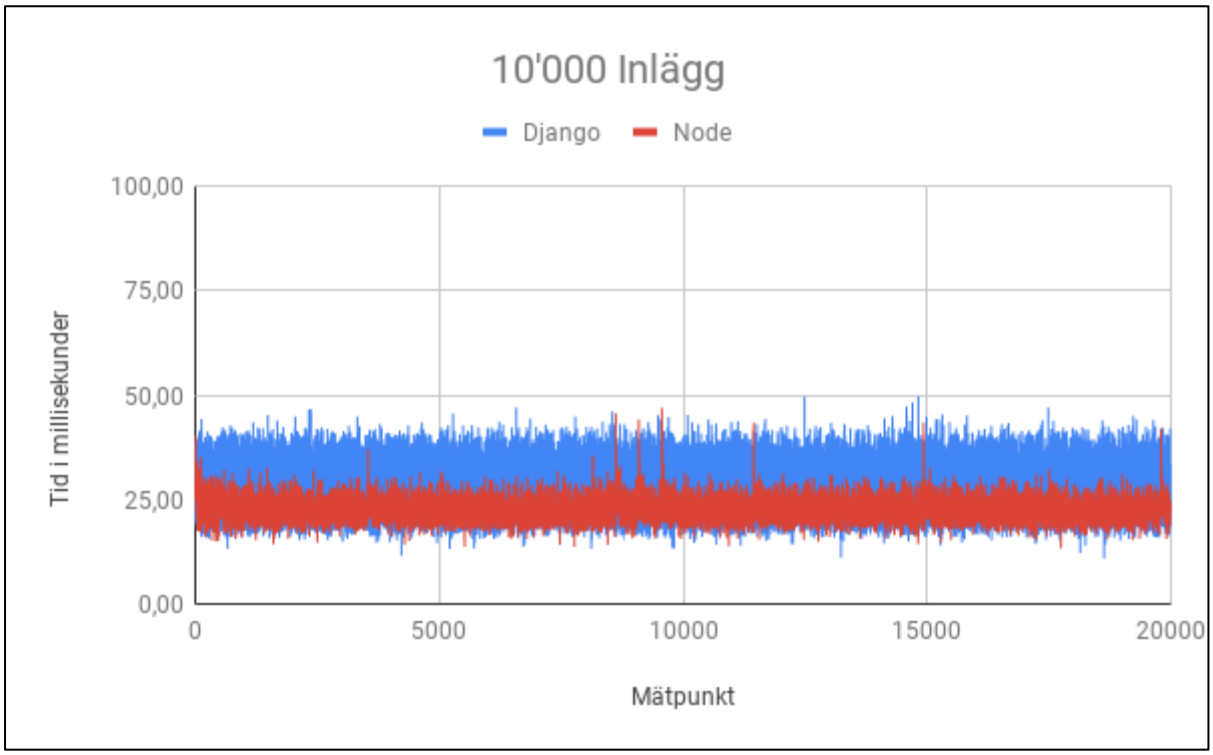
Mellan alla mätserier har databasen tagits bort och byggts om för att undvika problem med att databasen effektiviserar de SQL-frågor som används till varje mätserie. Detta innebär även att skriptet för att lägga in Stack Overflow data i databasen har körts mellan mätningar. Med hjälp av en klassvariabel som finns i skriptet så går det välja antal inlägg som ska läggas in; denna har modifierats mellan olika mätserier.

5.4 Analys

Resultatet för första mätserien av respektive ramverk, där databasen innehåller 10’000 inlägg, presenteras på figur 14 och 15 på nästa sida i form av ett stapel- och linjediagram. Django och Node.js har i detta fall ett medelvärde av 27,96 respektive 22,61 millisekunder. Linje-diagrammet ser ut att visa samma tendenser som mätserien utfört till pilotstudien, som också hade 10’000 inlägg i databasen, fast nu med betydligt fler mätpunkter. Node.js artefaktens alla mätpunkter är placerade i nedre delen av Djangos hela spann på vertikala axeln och har således en mer stabil söktid som inte varierar lika mycket på den sökterm som har använts. Node.js mätserien verkar innehålla några värden som sticker iväg mer än resterande värden från båda mätserierna av respektive applikationer, men totalt sett så är det inte många millisekunders skillnad.



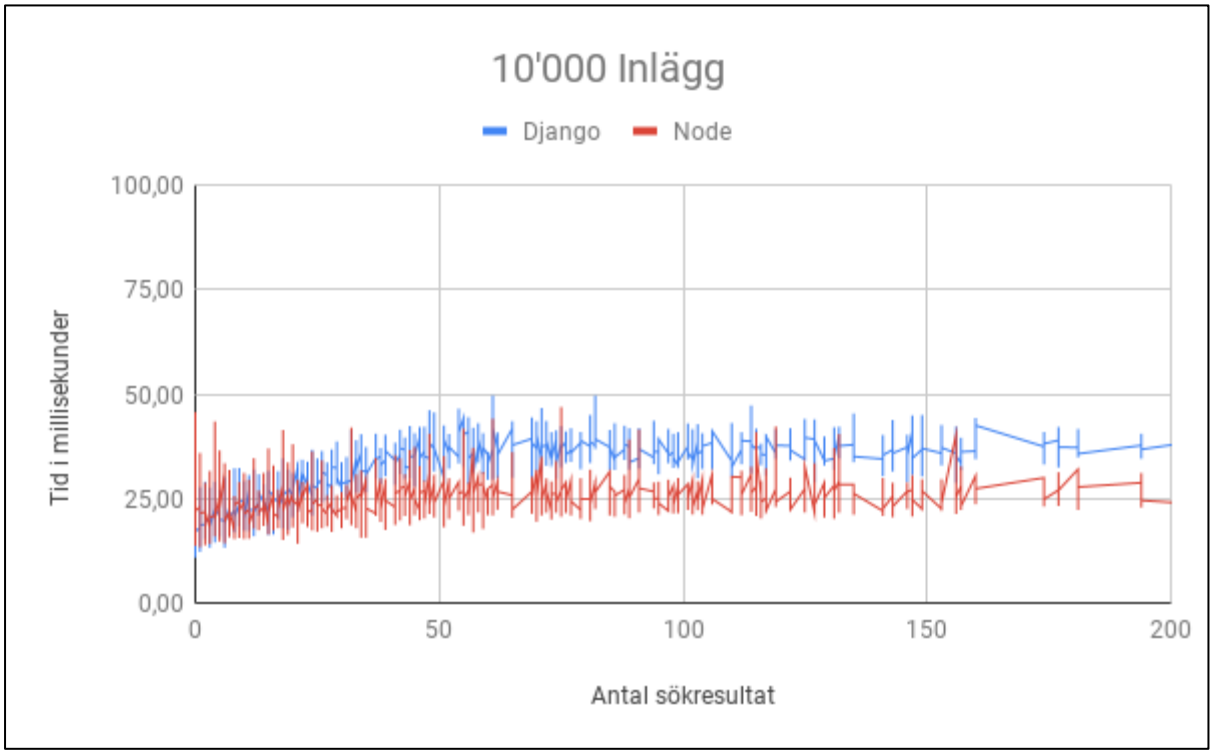
Figur 14 Resultat från experiment med 10’000 inlägg



Figur 15 Resultat från experiment med 10’000 inlägg

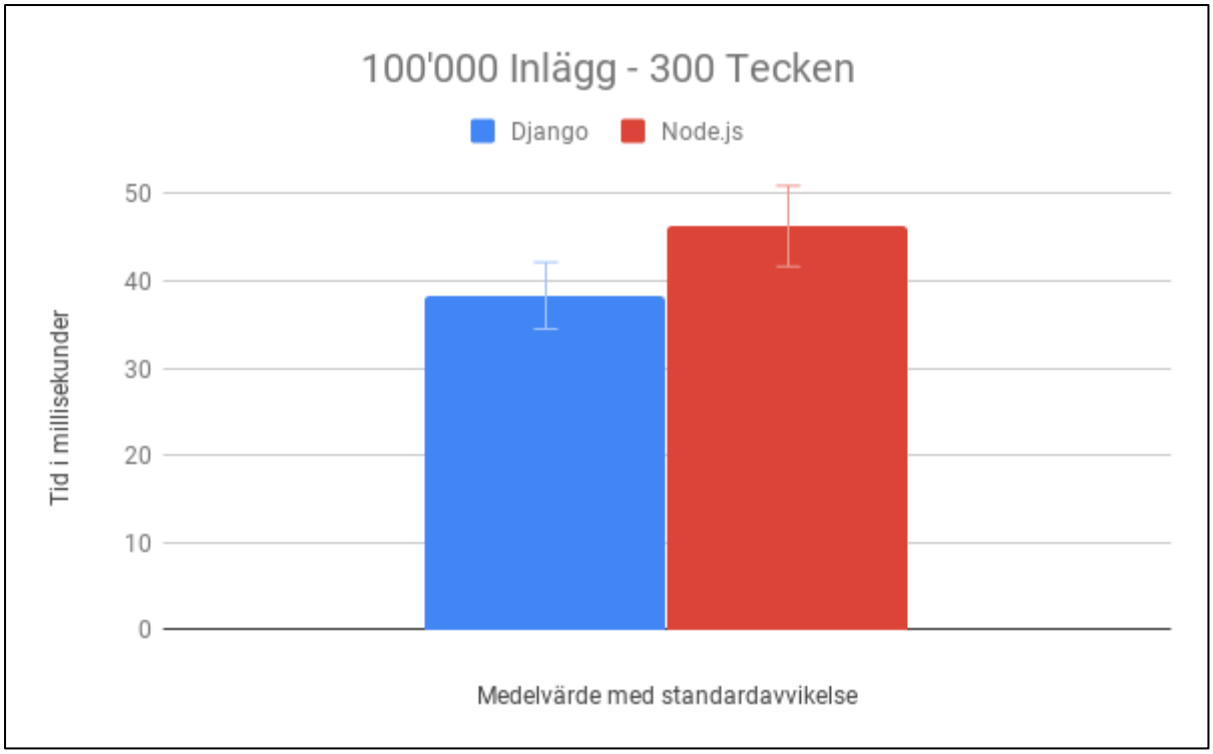
På figur 16 presenteras det hur sökresultatet förhåller sig till svarstiden när databasen innehåller 10’000 inlägg. Denna figur bekräftar att de lägsta söktiderna innehåller lägst antal sökträffar för båda ramverken samt att båda ramverken når sin topp när antalet sökträffar är 50 eller högre. Denna trend fortsatte även över 200 sökresultat, men figuren begränsades för att kunna visa mindre antal sökresultat mer detaljerat. Detta låter rimligt i och med SQL-frågan begränsar antalet sökresultat till 50 med hjälp av en LIMIT.

Django har en responstid som är något längre än Node.js när toppen av 50 sökresultat har nåtts; detta bekräftas även på figur 15 där topparna av respektive ramverk har ett gap mellan sig.

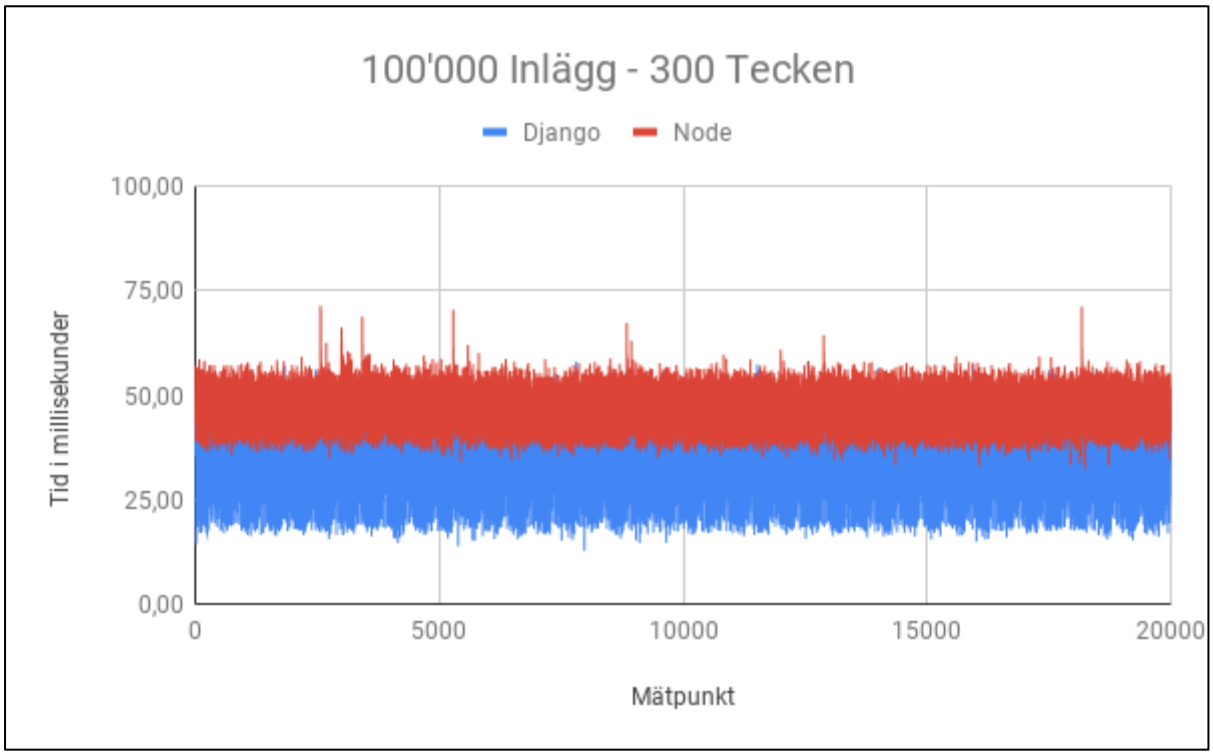


Figur 16 Resultat för hur svarstid förhåller sig mot antal sökresultat

Nästa mätserie för respektive ramverk utfördes med 100’000 inlägg i databasen och kroppstexten för alla sökresultat behölls kvar vid 300 för att kunna kontrasteras mot 150 tecken i nästa steg. På figur 17 och 18 presenteras resultatet av mätserierna. Med fler inlägg i databasen har nu Django fått en genomsnittlig responstid av 38,34ms medan Node.js applikation har mer än dubblat responstiden till ett snitt av 46,31ms. Det finns ett litet överlapp i standardavvikelsen och därmed kan det finnas mätvärden som inte helt bestämt går att avgöra vilken serie de tillhör.

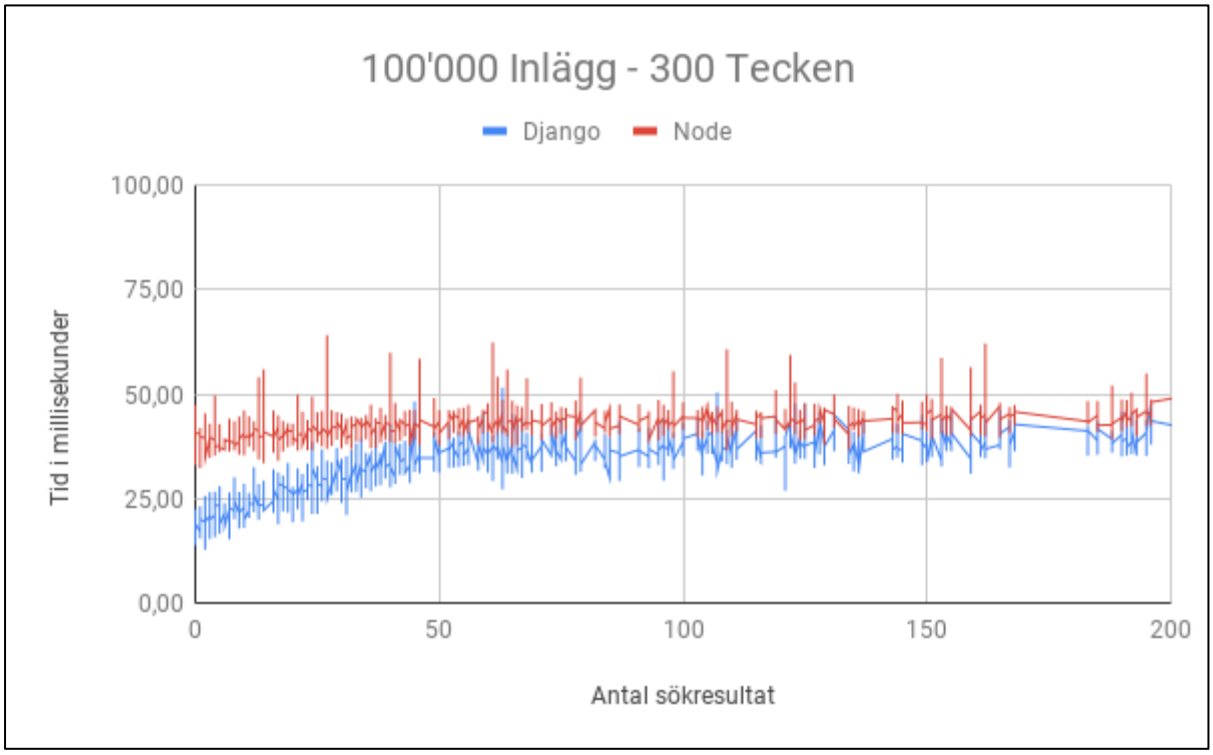


Figur 17 Resultat från experiment med 100’000 inlägg, 300 tecken



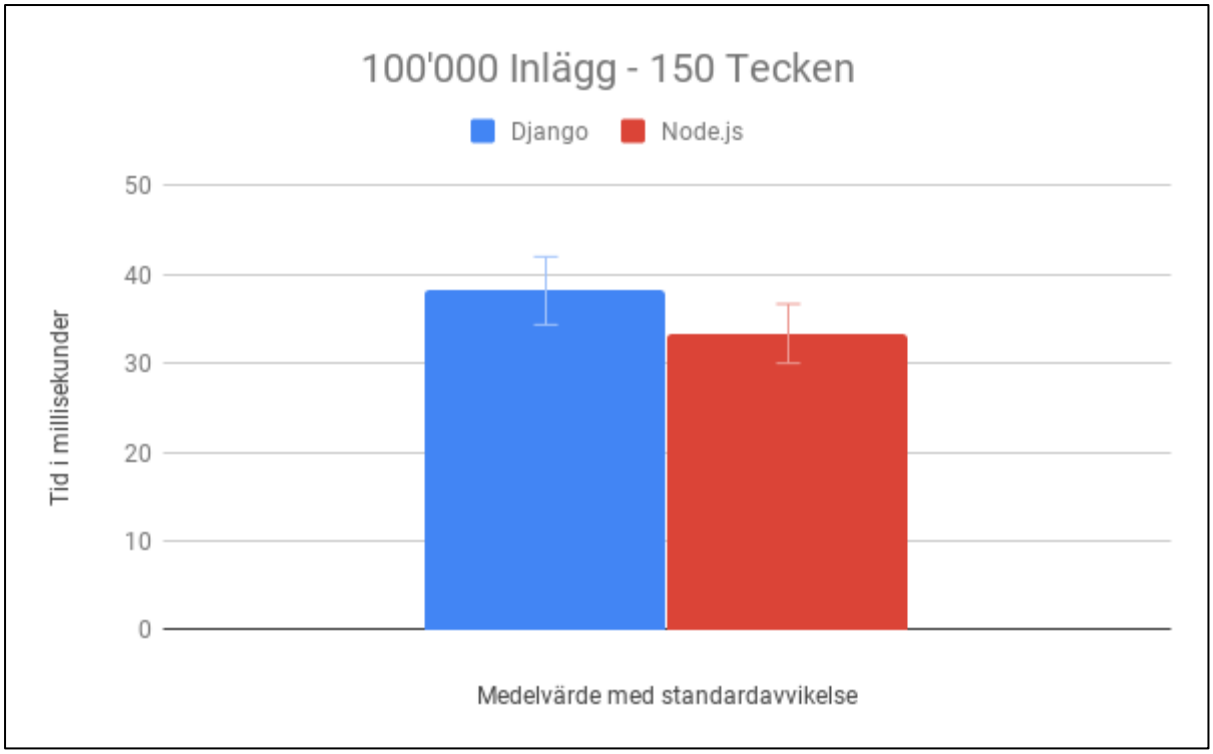
Figur 18 Resultat från experiment med 100’000 inlägg, 300 tecken

Med en databas som innehåller 100’000 inlägg så finns det vissa skillnader mot en databas som innehåller 10’000 inlägg när det kommer till förhållandet av antal söktider. Den första skillnaden som visade sig i början av diagrammet när varje applikation svarar med 0 sökträffar. På figur 16 tog det ungefär lika lång tid att få tillbaka ett sökresultat med 0 sökträffar, men i detta fall ligger Node.js applikationen väl över Django. Båda når dock sin topp vid 50 sökresultat igen, men denna gång svarar Django i snitt snabbare oberoende av antal sökträffar. Node.js ger i detta fall inte samma skala av trappeffekten upp till 50 sökträffar strecket, utan håller på en relativ jämn nivå för alla sökord.

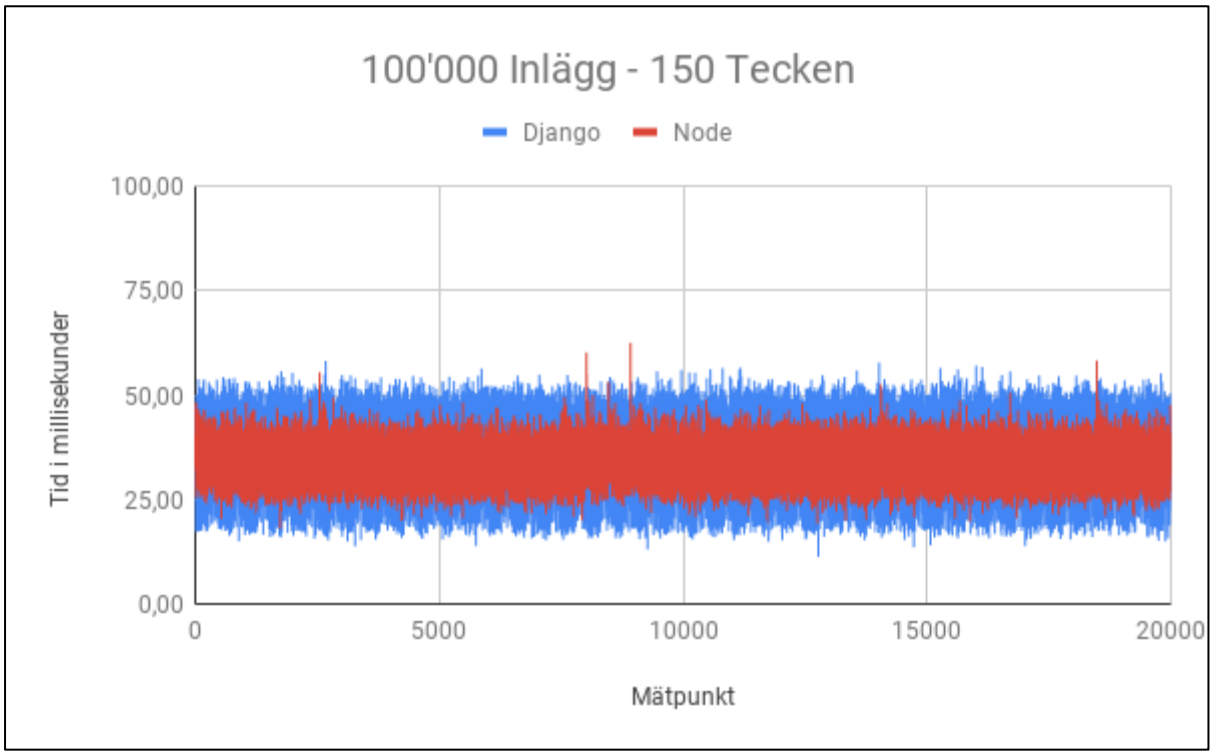


Figur 19 Resultat för hur svarstid förhåller sig mot antal sökresultat

På figur 20 och 21 visas de mätserier där databasen innehöll 100’000 inlägg samt att kroppstexten halverats till 150 tecken från de ursprungliga 300. Medelvärdet av Django och Node.js applikationernas responstid är nu 38,17 respektive 33,34 millisekunder och har således förbättrats för båda applikationer från föregående mätserie av respektive ramverk. Det intressanta här är hur lite det faktiskt förbättrade snittiden för Django jämfört med Node.js. Speciellt Node.js artefakten har fått se en betydlig förbättring i söktid av samtliga sökningar som nu har ett lägre snitt än Django applikationen. Standardavvikelsen är fortfarande till stor del överlappande, men de högsta och lägsta mätpunkterna från Django applikationen går till stor del avgöra att de inte tillhör Node.js applikationen.

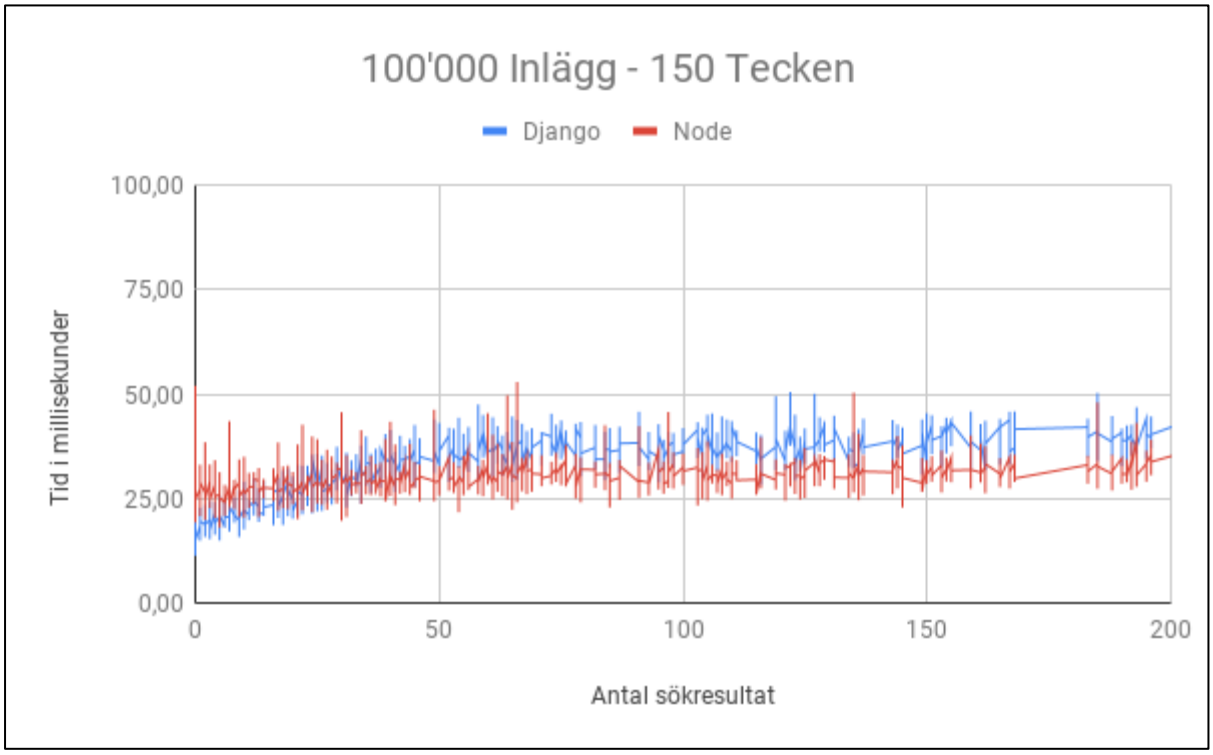


Figur 20 Resultat från experiment med 100’000 inlägg, 150 tecken



Figur 21 Resultat från experiment med 100’000 inlägg, 150 tecken

Med antalet tecken nu satt till 150 för sökresultatens kroppstext så har läget ändrats för hur förhållandet av responstider mot antal sökresultat ser ut. Figur 22 presenterar detta för en databas med 100’000 inlägg. I detta fall är svarstiden för 0 sökresultat närmare samma värde, således har gapet som hade skapades på figur 19 blivit mindre. Förutom det så har båda ramverken en trapplik effekt upp till 50 sökresultat som innan, trots att den för Node.js är väldigt minimal som på figur 19. Django har i detta fall bättre svarstider när det gäller ett få antal sökresultat, men snittet blev ändå i förmån för Node.js eftersom det uppstod en brytpunkt runt 40 sökresultat.



Figur 22 Resultat för hur svarstid förhåller sig mot antal sökresultat

5.5 Slutsats

Det som skulle besvaras med detta arbetet var ” Vilket ramverk av Django och Node.js ger bäst söktider av frågor för en Q&A-plattform utifrån Stack Overflow data lagrad i en PostgreSQL databas som hanterar fulltext sökningar?”. Genom de figurer och analyser som gjorts under 5.4 tillsammans med utförda ANOVA-tester så går det att dra några slutsatser av det experiment som har utförts på artefakterna som kan svara på frågeställningen.

ANOVA-tester var i detta fall nödvändigt eftersom de alltid ska utföras om det finns viss överlapp av standardavvikelse. ANOVA-testerna, som finns tillgängliga i appendix e, påvisar en signifikant skillnad genom ett P-värde av 0 på alla mätserier samtidigt som F-värdet är större än F-critical som är ytterligare en indikation på att det är en statistisk skillnad.

Node.js gav snabbare söktider av frågor när totala mängden inlägg i databasen vår lågt och skulle motsvara en mindre Q&A plattform utifrån den heterogena datan. Django var absolut inte helt ute ur leken, men från diagrammen gick det lätt att avläsa att Node.js var en signifikant förbättring.

Men med en databas som var 10 gånger så stor så vändes det helt. Django fick svarstider som genomgående var betydligt snabbare än motsvarande Node.js applikation, och hade dessutom svarstider som nästan var dubbelt så snabba för 0–10 sökresultat.

När en mindre mängd data behövde hanteras för varje sökning, på grund av minskningen i antal tecken per inlägg som visas i sökresultaten, så blev snittet av Node.js svarstider ca 4,8ms snabbare än Djangos. Värt att notera i detta fall är att Djangos medelvärde knappt ändrades medan Node.js medelvärde blev betydligt bättre.

Svaret till frågeställningen är således inte helt binär utifrån datan till hands, utan det beror lite på utformningen av applikationen. Men om samma trend skulle fortsätta som observerades när applikationerna gick från 10'000 till 100'000 inlägg i databasen, alltså att Django blev betydligt snabbare i jämförelse med Node.js, så skulle Django vara en klar vinnare för Q&A plattformar som har långt över 100'000 inlägg.

Men från det som har presenterats så erhåller Node.js bättre svarstider av frågor när antalet inlägg motsvarar en mindre plattform och när storleken av sökresultatet, den heterogena datan, är mindre än de 300 tecken som har satts som standard i dessa applikationer. Om samma ökning även kan ses för 10'000 inlägg när kroppstexten av varje sökresultat halveras så skulle vinsten av Node.js vara ännu större för mindre applikationer.

Således är Django ett bra val för utveckling av Q&A plattformar som hanterar större textmängder än Node.js och kan förväntas skala bättre när antalet inlägg ökar i samband med längre sökresultat. Node.js är däremot riktigt bra för mindre Q&A plattformar samt plattformar som har runt 100'000 inlägg, men i det fallet ska helst teckenantalet av kroppstexten för alla sökresultat hållas lågt.

Med detta sagt så måste noll hypotesen, H_0, stötas bort i fördel för mothypotesen, H_1, eftersom det finns tydliga skillnader mellan Django och Node.js för en Q&A plattform utifrån ett Stack Overflow data som lagras i en PostgreSQL databas som hanterar fulltextsökningar.

6 Avslutande diskussion

6.1 Sammanfattning

I detta arbete jämförs svarstider av en sökfunktion i Django och Node.js tillsammans med Express utifrån en identiskt implementerad Q&A plattformar som använder Stack Overflow data lagrad i en PostgreSQL databas som hanterar fulltextsökningar. Arbetet grundar sig från att användare förväntar sig en hög Quality of Service (Kulnarattana & Rongviriyapanish 2009), som bland annat innefattar laddtider av hemsidor. Eftersom det kostar mycket tid och pengar att bygga webbapplikationer så är det fördelaktigt att välja tekniker som kan erbjuda en hög Quality of Service för att slippa en ombyggnation. Valet av ramverk och tillhörande paket, exempelvis Express, har till stor del inspirerats av Schutt & Balci (2016) som bland annat jämför just Django och Node.js, fast utifrån ett molnplattform-perspektiv.

Ramverken jämförs med hjälp av ett teknikorienterat empiriskt experiment för att kunna svara på frågeställningen. Detta är fördelaktigt eftersom de flesta utomstående faktorer kan uteslutas och vissa specifika faktorer kan kontrolleras så att en rättvis jämförelse av ramverken kan utföras. Sökfunktionen jämförs bland annat genom att se hur väl applikationernas svarstider kan skala sett från antal inlägg i databasen, antal tecken som visas av varje inläggs kroppstext i sökresultatet och hur svarstiderna förhåller sig till antal sökresultat.

Utifrån diagrammen och ett antal ANOVA-tester gick det att konstatera att nollhypotesen kunde stötas bort i fördel för mothypotesen eftersom det tydligt gick att se en signifikant skillnad mellan båda ramverken i alla utförda mätserier. Samma slutsats gick även att ta i arbetet av Lei m.fl. (2014) där Node.js fick snabbast svarstider för en simpel applikation likt hur Node.js hanterar mindre antal inlägg bättre. Node.js presterade även bättre i det just nämnda arbetet när det kom till beräkningstunga http-frågor där varje http-fråga beräknande olika Fibonacci nummer medan Python fick de längsta svarstiderna.

Både Django eller Node.js fick söktider som var väldigt korta och är inte i närheten av de två sekunder som Butkiewicz m.fl. (2011) påpekade att användare ofta blir frustrerade vid. Alltså skulle båda ramverken i den skala som mätningarna har utförts på vara tillfredsställande för användarna.

6.2 Diskussion

Syftet med detta arbete var att komma fram till vilket ramverk av Django och Node.js tillsammans med Express ger best söktider av Stack Overflow data utifrån en Q&A plattform. Utmed vägen har ett antal etiska beslut tagits; bland annat har det lagts tid på att försöka få arbetet återupprepningsbart genom att redovisa tankegångar samt viktiga delar av programkoden som kan ha stor betydelse. Flödet av hela utveckling finns på GitHub i form av commits som visar vad som har ändrats i ordning från projektets start. Det är dock inte säkert att GitHub eller andra online-källor som har används kommer finnas tillgängligt i framtiden, därför har aspekten av att ta upp vad respektive källa innehåller samt att beskriva tankegångarna vart viktigt för att öka återupprepningsbarheten av arbetet.

En faktor som eventuellt har påverkat resultatet är max antal inlägg som visas vid sökningar, som för dessa artefakter är satta till 50. Söktidernas tak nåddes vid 50 resultat för alla mätserier, således kan det vara så att situationen mellan ramverken hade sett helt annorlunda

ut, om det istället var satt till exempelvis 20, eftersom brytpunkter och tak hade kunnat skilja från de diagram som presenterats.

Inga testpersoner eller testresultat har samlats in under denna studie, dock så har ett öppet dataset som innehåller identifierbara inlägg som går att koppla till användare använts. Inläggen har anonymiserats så att de inte längre går att koppla direkt till en specifik användare. Problemet som inte går att undkomma med detta är att man från inläggen kan söka sig fram till samma inlägg på riktiga Stack Overflow, och på så sätt hitta alla användare som har svarat och kommenterat på svar. Datasetet är helt tillåtet att använda i alla syften och ansågs vara speciellt användbart i detta syfte eftersom datan faktiskt är verklig och är vanlig i forskning syfte (Ponzanelli m.fl. 2015). Således finns särskilda etiska problem med datan, som det hade kunnat vara om datan var genererad eftersom den kanske inte hade motsvarat ett verkligt inlägg. Speciellt svårt hade det kunnat vara att generera data som innehåller ett heterogent dataset som detta arbete fokuserar på.

Alla presenterade diagram, förutom de som visar förhållandet mellan söktid och antal svar, innehåller inga trappor, spikar eller annan data som pekar mot att något har gått fel under mätningarna. Diagrammen för söktider förhållandet till antal svar innehåller dock vissa trappor upp till taket av 50 sökträffar och fortsätter efter det i samma nivå, men detta är helt förväntat.

Koden av respektive artefakt är baserad på bra tillvägagångssätt som bland annat nämns på respektive ramverks hemsida och i ett antal olika böcker som nämns under 4.1. För att kunna göra en rättvis bedömning så löser de olika kodproblem på samma sätt där det inte går mot bästa tillvägagångssätten som kan ge en prestandavinst. Det är viktigt att inte medvetet välja bort inbyggda effektiva funktioner för att lösa problem på samma sätt, utan varje ramverk och språk har sina fördelar som de borde använda för att göra en rättvis bedömning.

Experiment valdes som metod på grund av de fördelar som kommer med metoden, som exempelvis fallstudie, inte erbjuder. Ett experiment utförs i en kontrollerad miljö vilket ger fördelen av att kunna utesluta utomstående faktorer såsom internetanslutning och andra program som körs på maskinen. En fallstudie körs i en verklig miljö och används ofta för att undersöka ett antal olika fall, ibland bara ett, för att utforska ny mark. Därför var metoden experiment mer lämpligt till detta arbete med tanke på som skulle undersökas.

Arbetet har med resultatet från experimentet och diagrammen kunnat svara på vilka situationer respektive ramverk är lämpligt att använda, vilket i sin tur svarar på ett av grundproblemen som detta arbete bygger på – Användare förväntar sig en hög Quality of Service vilket bland annat innebär svarstider (Kulnarattana & Rongviriyapanish 2009) av webbapplikationer; därför är det viktigt att välja ett ramverk som kan skala väl och erbjuder snabba responstider av dynamiskt genererat innehåll på förfrågan. Enligt Butkiewicz m.fl. (2011) kan långa laddtider vara kritiskt för att behålla användare, speciellt om det är deras första besök, då användare kan bli frustrerade och lämna om det är en låg QoS. Detta arbete svarar på just det - vilket ramverk som erbjuder bäst söktider för den specifika tillämpning som detta arbete fokuserar på.

Arbetet har vissa samhällsnyttor; exempelvis kan företag som ska starta en plattform med heterogen data, lik den av en Q&A plattform som Stack Overflow, ta det ramverk som presterar bäst utifrån det beräknade antalet användare. Utifrån resultatet av detta arbete kan det innebära att de väljer Node.js om antalet användare beräknas vara relativt lågt eftersom det

är fokuserat på en nisch community eller liknande. Arbetet främjar även användandet av ramverk som i praktiken kan betyda att fler utvecklare väljer att använda ett ramverk på grund av de tidigare nämnda anledningarna såsom bättre skalbarhet och säkerhet som i sin tur skapar en säkrare web för användare.

6.3 Framtida arbete

Utifrån detta arbete finns det en del framtida arbeten som kan utföras på båda kort och lång sikt. På kort sikt hade fler mätningar kunnat utföras på de presenterade applikationerna för att avgöra hur de skulle skala med miljontals inlägg i databasen, som lätt finns tillgängligt i Stack Overflow datasetet. Det hade även vart intressant att ta och inkludera en till faktor, som hur antalet sökord påverkar söktiden av respektive applikation.

I ett långsiktigt perspektiv hade resultatet av denna studie kunnat jämföras med andra ramverk för samma tillämpning för att avgöra om just dessa två ramverk förhåller sig bra till andra ramverk. Det kan bli intressant att ta reda på eftersom det är svårt att veta om just dessa två ramverken ligger bland de som presterar bäst eller sämst för den givna tillämpningen. Även andra variabler, som hur lång tid det tar att visa inlägg med tillhörande kommentarer och hur lång tid det kan ta att skapa ett inlägg med heterogen data, kan vara intressant att ta reda på.

Om de ska jämföras för ett bredare område så kan bland annat exekveringstiden av SQL-frågor och Djangos ORM jämföras. Resultatet av ett sådant arbete skulle kunna appliceras för flera typer av tillämpningar beroende på typen av data som använts. Det finns vissa paket som erbjuder en ORM även till Node.js, exempelvis Sequelize, som skulle kunna jämföras mot Djangos ORM, men även mot vanliga SQL-frågor i Node.js.

Utifrån resultatet av detta arbete så skulle användartester kunna utföras för att ta reda på vad användare tycker är en rimlig förhandsvisning av sökresultat. Exempelvis behöver inte alltid början av varje sökresultat visas, utan sökordet som har använts kan bland annat markeras i förhandsvisning tillsammans med ett X antal tecken, både före och efter sökordet, för att visa det primära användandet av ordet. På så sätt skulle utvecklare komma undan med att visa en mindre mängd text för varje sökresultat. Det skulle bland annat Node.js applikationer ha nytta av, som inte kunde hantera lika stora textmängder, av heterogen data, lika bra som Django. På så sätt kan Node.js applikationer anpassa sig efter det alternativ som innehåller ett lägre antal tecken men som samtidigt användare tycker är mest informativt vilket i sin tur hade gjort Node.js mer attraktivt även för större applikationer utifrån resultatet av detta arbete.

Referenser

Anteru. (2019). *Using a Django model method from a template*. URL https://stackoverflow.com/questions/31359165/using-a-django-model-method-from-a-template (åtkomstdatum 2019.4.21.).

Arslan, A. & Yilmazel, O. (2008). A comparison of Relational Databases and information retrieval libraries on Turkish text retrieval. *2008 International Conference on Natural Language Processing and Knowledge Engineering*. ss. 1–8. doi:10.1109/NLPKE.2008.4906748

Baltes, S., Dumani, L., Treude, C. & Diehl, S. (2018). SOTorrent: Reconstructing and Analyzing the Evolution of Stack Overflow Posts. *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. ss. 319–330

Bartunov, O. & Sigaev, T. (2005). Full-Text Search in PostgreSQL, 77.

Berndtsson, M., Hansson, J., Olsson, B. & Lundell, B. (2008). *Thesis Projects*. London: Springer London. doi:10.1007/978-1-84800-009-4

Brown, A. & Harper, S. (2011). AJAX time machine. *W4A '11 Proceedings of the International Cross-Disciplinary Conference on Web Accessibility*. ACM Press, ss. 1. doi:10.1145/1969289.1969325

Brown, E. (2014). *Web development with Node and Express*. First edition. Beijing ; Sebastopol, CA: O'Reilly

Butkiewicz, M., Madhyastha, H. V. & Sekar, V. (2011). Understanding website complexity: measurements, metrics, and implications. *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*. ACM, ss. 313–328. doi:10.1145/2068816.2068846

Coelho, R., Kulesza, U. & von Staa, A. (2005). Improving architecture testability with patterns. *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM, ss. 114–115. doi:10.1145/1094855.1094890

Django. (2019a). *Django documentation*. URL https://docs.djangoproject.com/en/2.2/ (åtkomstdatum 2019.4.21.).

Django. (2019b). *Model field types*. URL https://docs.djangoproject.com/en/2.2/ref/models/fields/#model-field-types (åtkomstdatum 2019.4.22.).

Examensarbete. (2019). *Github repository*. URL https://github.com/c16matan/examensarbete (åtkomstdatum 2019.4.22.).

Express. (2019). *Express 4.x - API Reference*. URL https://expressjs.com/en/api.html (åtkomstdatum 2019.4.21.).

Fu, Y., Kowalczykowski, K., Ong, K. W., Papakonstantinou, Y. & Zhao, K. K. (2010). Ajax-based report pages as incrementally rendered views. Presenterad vid Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, ACM, ss. 567–578. doi:10.1145/1807167.1807229

Gupta, P., Mata-Toledo, R. & Monger, M. (2012). Utilizing ASP.NET MVC in web development courses. *Journal of Computing Sciences in Colleges*. ss. 10–14

Hsieh, M., Kao, Y. & Yuan, S. (2008). Web 2.0 Toolbar: Providing Web 2.0 Services for Existence Web Pages. *2008 IEEE Asia-Pacific Services Computing Conference*. ss. 507–512. doi:10.1109/APSCC.2008.137

Kiruthika, J., Khaddaj, S., Greenhill, D. & Francik, J. (2016). User Experience Design in Web Applications. *2016 IEEE Intl Conference on Computational Science and Engineering (CSE) and IEEE Intl Conference on Embedded and Ubiquitous Computing (EUC) and 15th Intl Symposium on Distributed Computing and Applications for Business Engineering (DCABES)*. ss. 642–646. doi:10.1109/CSE-EUC-DCABES.2016.253

Krause, J. (2017). *Programming web applications with Node, Express and Pug*. Berkeley, CA: Apress

Kulnarattana, L. & Rongviriyapanish, S. (2009). A client perceived QoS model for web services selection. *2009 6th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology*. ss. 731–734. doi:10.1109/ECTICON.2009.5137151

Laakso, T. & Niemi, J. (2008). An evaluation of AJAX-enabled java-based web application frameworks. *MoMM '08 Proceedings of the 6th International Conference on Advances in Mobile Computing and Multimedia*. Linz, Austria: ACM Press, ss. 431–437. doi:10.1145/1497185.1497278

Lei, K., Ma, Y. & Tan, Z. (2014). Performance Comparison and Evaluation of Web Development Technologies in PHP, Python, and Node.js. *2014 IEEE 17th International Conference on Computational Science and Engineering*. Presenterad vid 2014 IEEE 17th International Conference on Computational Science and Engineering, ss. 661–668. doi:10.1109/CSE.2014.142

Li, X., Karnan, S. & Chishti, J. A. (2017). An empirical study of three PHP frameworks. *2017 4th International Conference on Systems and Informatics (ICSAI)*. ss. 1636–1640. doi:10.1109/ICSAI.2017.8248546

Magaluk, A. (2019). *Node.js Express3 - Middleware to add render data to all render requests*. URL https://stackoverflow.com/questions/15057857/node-js-express3-middleware-to-add-render-data-to-all-render-requests (åtkomstdatum 2019.4.21.).

Mele, A. (2015). *Django By Example*

Mesbah, A. & Deursen, A. van. (2007). Migrating Multi-page Web Applications to Single-page AJAX Interfaces. *11th European Conference on Software Maintenance and Reengineering (CSMR'07)*. ss. 181–190. doi:10.1109/CSMR.2007.33

Mozilla. (2019). *performance.now()*. URL https://developer.mozilla.org/en-US/docs/Web/API/Performance/now (åtkomstdatum 2019.4.22.).

ninja. (2019). *How to do SELECT COUNT(*) GROUP BY and ORDER BY in Django?* URL https://stackoverflow.com/questions/19101665/how-to-do-select-count-group-by-and-order-by-in-django (åtkomstdatum 2019.4.21.).

Patel, S. K., Rathod, V. R. & Prajapati, J. B. (2013). Comparative analysis of web security in open source content management system. *2013 International Conference on Intelligent Systems and Signal Processing (ISSP)*. ss. 344–349. doi:10.1109/ISSP.2013.6526932

Ponzanelli, L., Mocci, A. & Lanza, M. (2015). StORMeD: Stack Overflow Ready Made Data. *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. Presenterad vid 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, ss. 474–477. doi:10.1109/MSR.2015.67

PostgreSQL. (2019). *PostgreSQL: Documentation: 9.5: Tables and Indexes - Creating Indexes*. URL https://www.postgresql.org/docs/9.5/textsearch-tables.html#TEXTSEARCH-TABLES-INDEX (åtkomstdatum 2019.4.22.).

Schutt, K. & Balci, O. (2016). Cloud software development platforms: A comparative overview. *2016 IEEE 14th International Conference on Software Engineering Research, Management and Applications (SERA)*. ss. 3–13. doi:10.1109/SERA.2016.7516122

Singh, A., Chawla, P., Singh, K. & Singh, A. K. (2018). Formulating an MVC Framework for Web Development in JAVA. *2018 2nd International Conference on Trends in Electronics and Informatics (ICOEI)*. ss. 926–929. doi:10.1109/ICOEI.2018.8553746

Stack Overflow. (2019). *Stack Overflow - Where Developers Learn, Share, & Build Careers*. URL https://stackoverflow.com/ (åtkomstdatum 2019.4.21.).

Wohlin, C. (2012). *Experimentation in software engineering*. New York: Springer

Appendix A - Mätskript

```
// ==UserScript==
// @name         Measurement script
// @namespace    http://tampermonkey.net/
// @version      0.1
// @description  Measurement script for thesis project
// @author       You
// @match        http://localhost/*
// @grant        none
// ==/UserScript==

(function () {
    'use strict';

    let words = ['world', 'django', 'engine', 'block', 'c++', 'java', 'vim', 'git', 'image', 'html',
'asp.net', 'track', 'keep', 'php', 'what', 'search', 'mac', 'text', 'editor', 'implement', 'javascript',
'less', 'than', 'someone', 'either', 'fire', 'entity', 'parse', 'phone', 'anyone', 'increase', 'data',
'carry', 'xml', 'json', 'myself', 'certainly', 'function', 'call', 'declare', 'create', 'algorithm', 'tree',
'maven', 'gradle', 'large', 'projects', 'subversion', 'svn', '.net', 'best', 'practices', 'dry', 'ruby',
'closure', 'style', 'javac', 'sql', 'able', 'year', 'string', 'variable', 'silverlight', 'testing', 'pc',
'support', 'delete', 'move', 'end', 'line', 'normal', 'mode', 'insert', 'tldr', 'character', 'selection',
'shift', 'control', 'colon', 'dash', 'useful', 'resource', 'paste', 'code', 'language', 'replace',
'improve', 'answer', 'share', 'building', 'lego', 'sign', 'structure', 'meaning', 'noted', 'thanks',
'documentation', 'matches', 'see', 'brackets', 'semicolon', 'portal', 'python', 'google',
'simplicity', 'bug', 'open', 'source', 'crowdsource', 'server', 'file', 'scala', 'mysql', 'mapreduce',
'important', 'array', 'ascii', 'server-side', 'gpl', 'activex', 'actionscript', 'apk', 'applet',
'autohotkey', 'flag', 'life', 'goto', 'gcc', 'true', 'false', 'undefined', 'unit', 'operator', 'unary',
'webassembly', 'xna', 'xslt', 'zombie', 'tk', 'thread', 'theoretical', 'subroutine', 'counter', 'race',
'asc', 'desc', 'case', 'column', 'constraint', 'table', 'procedure', 'unique', 'index', 'exec', 'exists',
'join', 'key', 'rownum', 'where', 'view', 'truncate', 'typedef', 'union', 'extern', 'extends',
'unsigned', 'signed', 'long', 'enum', 'continue', 'auto', 'printf', 'std', 'using', 'namespace',
'include', 'throw', 'require', 'interface', 'echo', 'endfor', 'finally', 'instanceof', 'protected',
'static', 'return', 'try', 'unset', 'use', 'die', 'clone', 'break', 'callable', 'abstract', 'background',
'assert', 'import', 'global', 'elif', 'nonlocal', 'raise', 'volatile', 'register', 'note', 'inline',
'parameter', 'access', 'specifier', 'command', 'arguments', 'handling', 'header', 'interview',
'library', 'reserved', 'const', 'amd', 'intel', 'gpu', 'cpu', 'settings', 'config', 'increment',
'inheritence', 'haskel', 'heroku', 'heuristic', 'hex', 'hdml', 'gtk', 'expression', 'bash', 'ssh', 'event',
'listener', 'floating', 'generation', 'elixir', 'ellipsis', 'eclipse', 'problem', 'intellij', 'jetbrains',
'dataflow', 'dart', 'dark', 'cygwin', 'curry', 'cvs', 'compute', 'comment', 'bytecode', 'bool',
'camel', 'chaos', 'clos', 'allocate', 'agile', 'waterfall', 'ado', 'time', 'system', 'administration',
'choice', 'single', 'cut', 'college', 'interest', 'leader', 'init', 'user', 'password', 'localhost',
'127.0.0.1', 'from', 'path', 'connection', 'internet', 'docs', 'post', 'self', 'lib', 'edge', 'initial', 'scale',
'utf-8', 'console', 'without', 'level', 'young', 'allow', 'win', 'research', 'actually', 'begin', 'opengl',
'directx', 'still', 'private', 'behavior', 'emacs', 'patient', 'var', 'performance', 'goal', 'tech', 'quick',
'service', 'result', 'slow', 'fast', 'keyboard', 'keyword', 'definition', 'three', 'two', 'one', 'fixed',
'predefined', 'latter', 'play', 'game', 'default', 'mail', 'processor', 'gof', 'design', 'patterns',
'subject', 'unfortunately', 'client', 'website', 'linux', 'arch', 'debian', 'ubuntu', 'stream', 'pen',
'class', 'seperating', 'work', 'spent', 'coding', 'slack', 'barebones', 'encountered', 'puzzle',
'optimal', 'rectangle', 'quad', 'permission', 'print', 'powershell', 'terminal', 'framework', 'fly',
'email', 'recursive', 'deploy', 'production', 'node', 'syntax', 'figure', 'photoshop', 'illustrator',
'adobe', 'book', 'http', 'raw', 'content', 'learn', 'speech', 'pay', 'vmware', 'legacy', 'modern',
'virtual', 'windows', 'safety', 'zend', 'compile', 'bin', 'src', 'ip', 'address', 'domain', 'ftp', 'form',
'authentication', 'network', 'shell', 'commit', 'mime', 'bultin', 'office', 'graph', 'binding',
'tomcat', 'api', 'mouse', 'button', 'hyperlink', 'wiki', 'dev', 'integer', 'double', 'pointer', 'curl',
```

```
'maps', 'uml', 'workstation', 'drive', 'partition', 'database', 'explain', 'export', 'prevent',
'vs2008', 'delphi', 'mailto', 'def', 'dns', 'permanently', 'format', 'audio', 'enforce', 'decoration',
'web', 'report', 'crystal', 'recycle', 'retrieve', 'new', 'old', 'random', 'temp', 'encoded',
'encrypted', 'flex', 'like', 'realistic', 'pdo', 'mysqli', 'glew', 'arrow', 'document', 'loop', 'school',
'mixed', 'bit', 'byte', 'unix', 'clear', 'mock', 'endpoint', 'tcp', 'udp', 'jpeg', 'png', 'alpha',
'animation', 'coordinate', 'accelerate', 'render', 'input', 'decode', 'serialize', 'xps', 'driver', 'usb',
'execute', 'tool', 'signature', 'preserve', 'bind', 'loose', 'sockets', 'mfc', 'rgb', 'blend', 'menustrip',
'color', 'module', 'imap', 'secure', 'orm', 'avoid', 'mass', 'polymorphism', 'foreign', 'null', 'child',
'parent', 'live', 'sentence', 'diff', 'merge', 'identified', 'intellisense', 'browser'];

  let startTime;
  let currentWord = 0;
  let amountOfMeasures = 0;
  const TIMES_TO_MEASURE = 20000;

  var performance = [];

  function initObserver() {
    let container = document.getElementById('main-container');
    let observer = new MutationObserver(checkDOM);
    observer.observe(container, { childList: true });
  }

  function checkDOM(mutationsList, observer) {
    for (let mutation of mutationsList) {
      if (mutation.type == 'childList') {
        let completionTime = window.performance.now() - startTime;
        let searchWords = document.getElementById("search-words").innerHTML;
        let amountOfResults = document.getElementById("total-amount-of-
results").innerHTML;
        performance.push({
          time: completionTime,
          searchWords: searchWords,
          amountOfResults: amountOfResults
        });
        amountOfMeasures++;
        if (amountOfMeasures >= TIMES_TO_MEASURE) {
          let csv = "data:text/csv;charset=utf-8,";
          performance.forEach(function (value) {
            csv += value.time.toString().replace('.', ',');
            csv += ":" + value.searchWords;
            csv += ":" + value.amountOfResults;
            csv += "\n";
          });
          let uri = encodeURI(csv);
          let link = document.createElement("a");
          link.setAttribute("href", uri);
          link.setAttribute("download", "result.csv");
          document.body.appendChild(link);
          link.click();
          document.body.removeChild(link);
        } else {
          setTimeout(measure, 200);
        }
      }
```

```
    }
  }

  function measure() {
    document.getElementById('search-box').value = words[currentWord];
    currentWord++;
    if (currentWord >= words.length) currentWord = 0;
    startTime = window.performance.now();
    document.getElementById('search-icon').click();
  }

  function ready() {
    initObserver();
    setTimeout(measure, 1000);
  }

  if (document.attachEvent ? document.readyState === 'complete' : document.readyState
!== 'loading') {
    ready();
  } else {
    document.addEventListener('DOMContentLoaded', ready);
  }
})();
```

Appendix B - Nginx Django konfiguration

```
server {
        listen 8000;
        server_name localhost;

        location /static/ {
                root /path/to/project/root;
        }
        location / {
                include proxy_params;
                proxy_pass http://unix:/path/to/django-project.sock;
        }
}
```

Appendix C - Nginx Node.js konfiguration

```
server {
        listen 8080;
        server_name localhost;

        location ~ ^/(fonts/|images/|js/|css/|favicon.ico) {
                root /path/to/project/root/public;
                access_log off;
                expires max;
        }
        location / {
                proxy_pass http://localhost:3000/;
        }
}
```

Appendix D - Gunicorn

```
[Unit]
Description=gunicorn daemon
After=network.target

[Service]
User=user
Group=group
WorkingDirectory=/path/to/project/root
ExecStart=/path/to/project/root/venv/bin/gunicorn --access-logfile - --workers 3 --bind
unix:/path/to/project/root/django-project.sock django-project.wsgi:application

[Install]
WantedBy=multi-user.target
```

Appendix E - ANOVA Tester

10'000 Inlägg, 300 tecken

SUMMARY

Groups	*Count*	*Sum*	*Average*	*Variance*
Column 1	20000	452166,16	22,608308	7,74189277
Column 2	20000	559143,495	27,9571747	57,3996658

ANOVA

ırce of Variati	*SS*	*df*	*MS*	*F*	*P-value*	*F crit*
Between Gro	286103,755	1	286103,755	8784,06232	0	3,84169132
Within Group	1302766,03	39998	32,5707793			
Total	1588869,78	39999				

100'000 Inlägg, 300 tecken

SUMMARY

Groups	*Count*	*Sum*	*Average*	*Variance*
Column 1	20000	926154,685	46,3077343	29,9716842
Column 2	20000	766714,34	38,335717	84,132704

ANOVA

ırce of Variati	*SS*	*df*	*MS*	*F*	*P-value*	*F crit*
Between Gro	635530,59	1	635530,59	11139,4592	0	3,84169132
Within Group	2281973,66	39998	57,0521941			
Total	2917504,25	39999				

100'000 Inlägg, 150 tecken

SUMMARY

Groups	*Count*	*Sum*	*Average*	*Variance*
Column 1	20000	666852,115	33,3426057	30,953638
Column 2	20000	763480,795	38,1740398	86,8946793

ANOVA

ırce of Variati	*SS*	*df*	*MS*	*F*	*P-value*	*F crit*
Between Gro	233427,545	1	233427,545	3961,49135	0	3,84169132
Within Group	2356848,5	39998	58,9241587			
Total	2590276,04	39999				