CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

En prestandajämförelse av Neo4j och SQL Server utifrån spatiala användningsområden

Datastrukturer till nya datalager och mixade datatyper för fastighetsvärderingar

Max Persson
Samuel Bach

INSTITUTIONEN FÖR DATA- OCH INFORMATIONSTEKNIK

CHALMERS TEKNISKA HÖGSKOLA
GÖTEBORGS UNIVERSITET
Göteborg, Sverige 2022
www.chalmers.se

En prestandajämförelse av Neo4j och SQL Server utifrån spatiala användningsområden

Institutionen för Data- och Informationsteknik
Chalmers Tekniska Högskola
Göteborgs Universitet
SE-412 96 Göteborg
Sverige
Telefon + 46 (0)31-772 1000

Abstract

Värderingsdata is a company which specializes in evaluating different types of properties in Sweden. To accomplish high accuracy in their evaluations, a considerable amount of data as well as complex and time consuming queries are utilized. In an effort to remedy the time aspect, this report's aim has been to determine the advantages and disadvantages of the currently-in-use relational database SQL Server and the graph database Neo4j. Furthermore, their individual performance is to be determined. The different advantages and disadvantages of the databases will be assessed in conjunction with tests as will their respective performance. The tests in question consist of several different queries based on three different scenarios. These are centered around retrieving a propertiey's closest point and area of interest as well as any relevant spatial information. The results from the tests conducted showed the relational database as performing better in all scenarios. Neo4j's lack of native support for spatial functionality was one of its downsides. This in combination with its performance compared to the relational database lead to the conclusion that Neo4j in its current state is not a viable substitution to SQL Server.

Keywords: Neo4j, SQL Server, comparison, spatial use cases

Förord

Denna rapport har skrivits för att ge en insikt i prestandaskillnaderna för spatiala användningsområden mellan relationsdatabasen SQL Server och grafdatabasen Neo4j. Syftet har varit att genom flertalet tydligt definierade tester undersöka prestanda och därigenom ge grund till fortsatt arbete kring de två databaserna genom att erbjuda en stabil faktagrund. Vidare skall rapporten erbjuda nyttig information om de två databaserna för att underlätta ett eventuellt val emellan dem.

Särskilt tack ges till handledaren på Värderingsdata Jon Larborn och handledare på Chalmers Ulf Norell, för deras kontinuerliga stöd och råd genom hela arbetsprocessen. Detta har uppskattats starkt och underlättat arbetet enormt.

Innehållsförteckning

1 Inledning

Inledningen ämnar att ge bakgrund kring arbetet samt dess syfte. Vidare beskrivs avgränsningar vilka gjorts för arbetet samt en precisering av frågeställningen.

1.1 Bakgrund

Värderingsdata är ett företag som erbjuder expertis inom automatiserade värderingsmodeller för bank och finansmarknaden, analys och värdering av kommersiella fastigheter samt analys och information om småhus och bostadsrätter [1]. För att uppnå tillförlitliga slutsatser från automatiserade värderingsmodeller används stora mängder spatial(geografisk) samt fastighetsrelaterad data, både nuvarande och historisk. Värderingsdata är den enda representanten i EAA (European AVM Alliance) från Sverige. EAA är en europeisk förening som arbetar med att representera och standardisera användandet av statistiska värderingsmodeller. För att statistiska värderingsmodeller ska certifieras av EAA med deras "EEA AVM Label" bör de uppfylla en del krav som listas på deras hemsida [2].

I Värderingsdatas nuvarande implementation utnyttjas relationsdatabasen MSSQL(Microsoft SQL Server) för att lagra de datamängder som används för deras automatiska värderingsmodeller. Elasticsearch, ett verktyg för att lagra och analysera data [3], används för att aggregera datamängder i förväg i syfte att förbättra tidsprestandan vid hämtning av data. På senare tid har däremot intresset för ett nytt sätt att lagra datan framkommit. Värderingsdata vill att en lösning tas fram där all data kan matchas i realtid utan att vissa datamängder behöver färdigställas i förväg. Detta för att underlätta uppdatering av data då man i nuläget behöver uppdatera de färdigställda datamängderna när grundläggande data ändras. Den nya lösningen bör även prestera bättre tidsmässigt och hantera nya datamängder för att kunna ge underlag för utvecklingen av framtida värderingsmodeller.

1.2 Syfte

Syftet med detta projekt är att skapa bättre förutsättningar för att framställa nya värderingsmodeller genom att undersöka sätt att strukturera och lagra data med en grafdatabas. Projektet syftar även till att förbättra den tidsmässiga prestandan för uthämtning av data i realtid utan att behöva aggregera vissa datamängder i förväg.

1.3 Avgränsningar

Innan starten av arbetet har avgränsningar definierats i syfte att anpassa det för tidsplanen som bestämts. Dessa är:

- Enbart två databastyper kommer jämföras.
- Testmiljön för de planerade testerna sätts upp av Värderingsdata.

- Data som ska användas för databaserna och testerna förses med hjälp av Värderingsdata.
- Arbetet använder delvis förvrängd data men ger fortfarande ett representativt urval av fastigheter.

1.4 Tidigare arbeten

Då arbetet är relativt specifikt kunde inte några tidigare arbeten med tillräcklig liknelse hittas. Arbeten kring Neo4j och spatialdata har tidigare gjorts men inga som kombinerar de två kunde hittas.

1.5 Precisering av frågeställning

Målet med projektet är att utveckla en Neo4j grafdatabas vilken kan hantera de spatialdata Värderingsdata tillhandahåller i deras nuvarande MSSQL databas samt klara av att hantera nya datamängder. Det ska även tas fram en MSSQL relationsdatabas med stöd från Värderingsdata vilken efterliknar deras nuvarande struktur. Dessa två databaser ska testas och jämföras utifrån ett tidsperspektiv med tester vilka baseras i tre olika scenarion. Dessa är:

- Matcha fastigheter/transaktioner till information om ett spatialområde.
- Matcha fastigheter/transaktioner till närmaste intresseområden.
- Matcha fastigheter/transaktioner till närmaste intressepunkter.

Vidare ska följande frågeställningar besvaras:

- Vilka är för och nackdelarna mellan Neo4j och MSSQL?
- Vilken databas presterar bäst tidsmässigt i de angivna scenarierna och testerna?

2 Teknisk bakgrund

Den tekniska bakgrunden ämnar att beskriva de tekniska verktyg, system och metoder arbetet inkluderar.

2.1 SQL Server

Relationsdatabasen har funnits sen 1970-talet och anses vara den mest accepterade databasmodellen idag. En relationsdatabas är en databas vilken hanterar och lagrar data utifrån en relationsmodell. Detta innebär att data struktureras med hjälp av tabeller, även kallade relationer. I en tabell representerar kolumnerna datans attribut och varje rad i tabellen är de sparade värdena. För att manipulera data i en relationsdatabas används språket SQL vilket står för "Structured Query Language". DBHS (databashanteringssystem) baserade på en relationsdatabas följer ACID-principer för transaktioner vilka sker när operationer på databasen utförs. A:et står för atomicitet som betyder att antingen utförs alla steg i transaktionen eller inga alls. C:et står för konsistens eller "consistency" och innebär att en transaktion lämnar databasen i ett konsistent tillstånd. I:et står för isolering och betyder att transaktioner inte ska kunna påverka varandra. Sist är D:et som står för hållbarhet och innebär att resultatet av en transaktion inte kan förloras [4]. MSSQL följer relationsmodellen då en databas i MSSQL består av en samling tabeller för att strukturera data [5]. SSMS (SQL Server Management Studio) används för att hantera SQL Server. Det ger en tillgång till verktyg för att designa, skapa, övervaka och genomföra queries [6]. En förfrågan till en databas för att hämta specificerad data kallas att utföra en query.

2.2 Neo4j

Neo4j är en grafdatabas med stöd för transaktioner vilka följer ACID-principerna. Grafdatabasen är skriven i Java och Scala. Dess källkod är öppen och tillgänglig på GitHub [7]. Neo4j släpptes ursprungligen 2010 och har varit under kontinuerlig utveckling sedan dess [8]. Inom Neo4j används noder, relationer och nodegenskaper för att organisera och lagra data. Noder kan associeras med ett märke vilket representerar den roll noden har. Noder kan erhålla egenskaper vilket är data och förvaras i nyckel-värde par. Relationer kopplar ihop noder och har alltid en typ, riktning samt start -och slutnod. Likt noder kan relationer ha egenskaper. Nämnvärt är att antalet typer av relationer på noder eller riktningen de navigeras inte påverkar prestandan [7].

Neo4js grafdatabas finns i två utgåvor varav deras "Neo4j Community Edition" använder sig av licensen GPL v3 vilket ger gratis åtkomst till mjukvaran. Den andra utgåvan "Neo4j Enterprise Edition" har en kommersiell inriktning. Tillgång kan fås genom en prenumeration men kan även fås gratis under en evaluerings licens eller vid användandet av Neo4j Desktop [9]. Neo4j är inte endast en grafdatabas utan kommer också med ett antal verktyg. Neo4j Desktop databashanteringssystem möjliggör skapandet och hanterandet av lokala Neo4j-databaser. Man kan skapa godtyckligt många projekt och lokala databaser samt ansluta till icke-lokala databaser. Hantering och nedladdning av olika plugins görs också enkelt.

Neo4j Browser är ett användargränssnitt till en Neo4j databas och tillhandahåller fulla CRUD förmågor. Efter förfrågningar kan resultat visas i form av en graf, JSON eller en tabell [10].

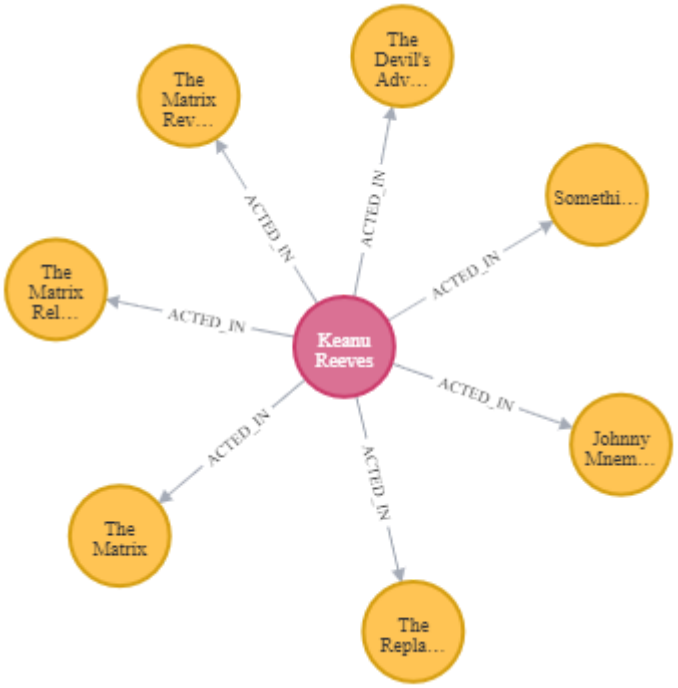
För att importera data till Neo4j från en relationsdatabas är det enklaste tillvägagångssättet "LOAD CSV". Det är en del av Cypher språket och är kompatibelt med datamängder upp till 10 miljoner rader. Med hjälp av kommandot kan man läsa in CSV filer, omvandla data, skapa noder och relationer för att importeras till grafdatabaser. Det stödjer att köras som en enda transaktion eller satsvis där transaktionen delas upp i mindre delar för att mängden data inte ska överskrida Neo4js minne [11].

2.2.1 Cypher

Trots att Neo4j är skrivet i Java används inte Java för att manipulera datan i dess grafdatabas. Neo4j har istället tagit fram ett eget språk, Cypher, vilket vid utvecklandet tagit inspiration av SQL och beskrivs av utvecklarna som "SQL för grafdatabaser". Syntaxen är visuell och av typen "ASCII-art" där noder representeras med parenteser och relationer med pilar [10]. I figur 2.1 kan detta ses i ett cypher-query vilken hittar alla filmer Keanu Reeves har varit skådespelare i. En nod presenteras med hjälp av parenteser, (p:Persson), och relationer med pilar, →.

```
neo4j$ MATCH (p:Person{name:"Keanu Reeves"})-[:ACTED_IN]→(m:Movie) RETURN p, m
```

Figur 2.1 - Cypher-query för att hitta alla filmer Keanu Reeves har varit med i.



Figur 2.2 - Grafen returnerad av cypher-query i figur 2.1.

Cyphers styrka och centralt för språket är dess förmåga att matcha och navigera mönster, något man kan se i dess syntax där ett query ritas upp visuellt. Satsen **MATCH** används för detta syfte. Cypher strukturerar queries linjärt från början av queries text till dess slut. Vad en query ska returnera bestäms i slutet med satsen **RETURN** till skillnad från SQL där **SELECT** används vid början av ett query. För att sätta samman queries används satsen **WITH** där resultatet från ett query innan satsen kan användas till nästkommande query.
I cypher finns det rikligt med satser för att uppdatera och modifiera en graf. Satserna **CREATE** och **DELETE** används för att skapa och ta bort noder eller relationer. För att kunna lagra data i noderna eller relationerna används satsen **SET** vilket skapar nod -och relationsegenskaper. Slår man ihop satserna **MATCH** och **CREATE** får man **MERGE**. Denna sats används för att säkerställa att noder och relationer hålls unika. **MERGE** försöker hitta den givna noden eller relationen och om ingen hittas skapas den [12].

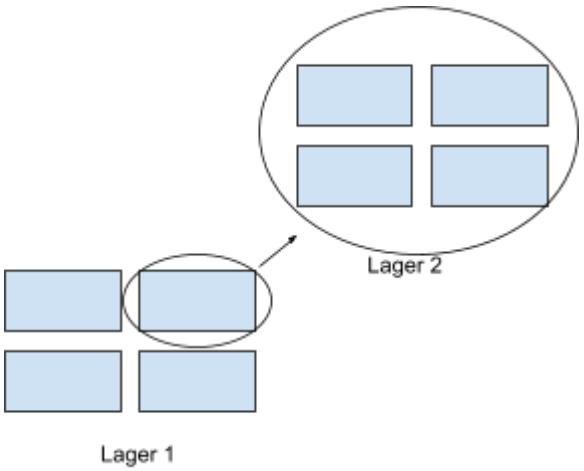
2.3 Index

Index används i databaser för att förbättra prestandan. Både SQL Server och Neo4j använder sig flitigt av dessa. SQL Server använder sig av bland annat klustrade index och spatialindex [13]. Klustrade index är baserade på en b^+-träd struktur [14]. Spatialindex baserar istället sitt index på en b-trädstruktur [15]. Dessa strukturer och dess funktionalitet beskrivs mer i detalj i 2.3.1. Klustrade index fungerar på det sättet att de sorterar datan i en tabell utifrån deras data. Detta innebär en viss begränsning i det att enbart ett klustrade index kan förekomma på en tabell då datan enbart kan sorteras i en ordning. Det är även i enbart klustrade index där data sorteras utifrån dess värden jämfört med andra index, exempelvis icke klustrade index [16].

Neo4j har fyra index: b-träd, full-text, text och token lookup [17]. B-träd index är dock på väg att bli utfasad och ska bland annat bytas ut med ett punktindex vilket är optimerat för spatiala queries. Inom cypher är det möjligt att skapa dessa index med ett eller flera värden som utgångspunkt. Dessa index kan skapas antingen för en nod eller relationer mellan noder [18].

2.3.1 Spatialindex

Spatiala index ämnar att minska exekveringstiden för beräkningstunga spatiala förfrågningar. För att uppnå detta finns olika metoder. SQL Server spatial index använder ett rutnätssystem bestående av olika lager av rutor [15]. Vardera ruta, bortsett från de tillhörande det lägsta lagret, leder till ett nytt lager bestående av samma uppsättning rutor. Lagren består av 4 nivåer och kan bestå av antingen 4x4, 8x8 eller 16x16 antal rutor för varje enskilt lager. I figur 2.3 förekommer ett fiktivt exempel av ett rutnätssystem i 2x2 konfigurering med enbart två lager i syfte att visualisera strukturen.



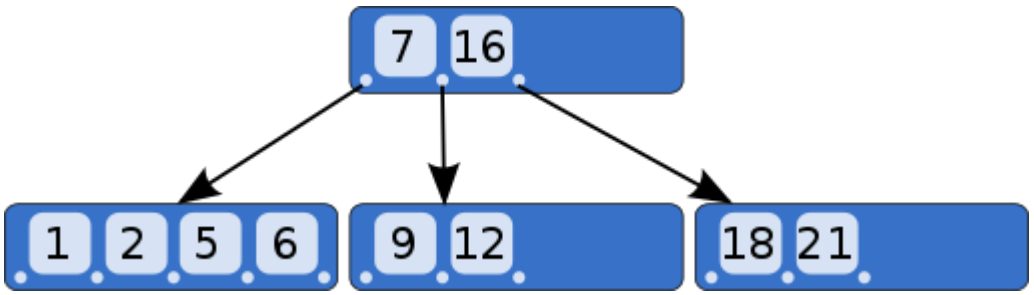
Figur 2.3 - Fiktivt exempel av rutnätssystem för spatialindex.

Detta index sorterar datan utifrån en b-träd struktur. Index baserade på en b-träd struktur fungerar likt ett balanserat binärt sökträd. De är självbalanserande och sorterar efter storleken på så kallade nycklar. Dessa nycklar används för att snabbt hämta specifik data utifrån den nyckel som ges vid sökningen.

För att det skall vara ett giltigt b-träd krävs det att tre specifika villkor uppfylls. Dessa är följande [19]:

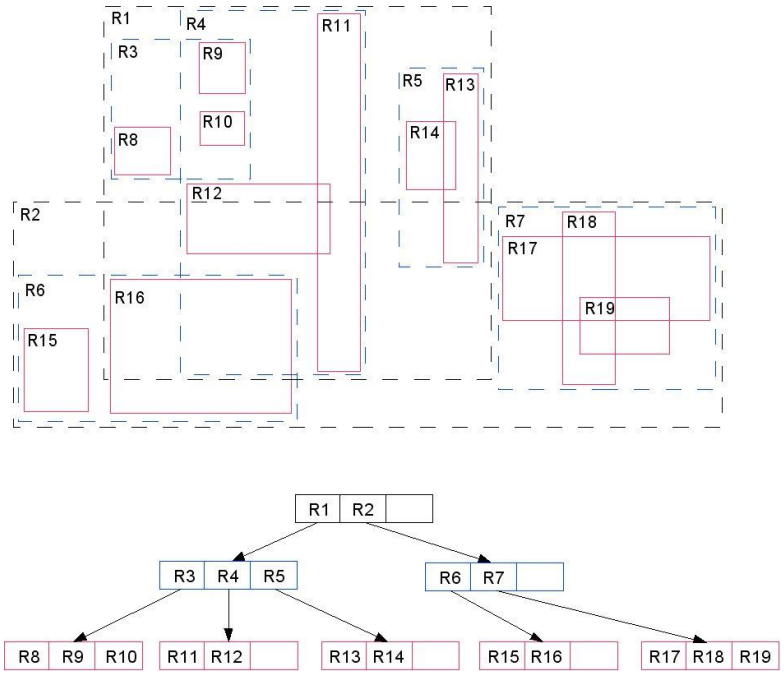
- Alla rötter är antingen ett löv (länkar inte till några noder) eller har från 2 till m stycken "barn" där m representerar maximala mängden barn en rot kan ha.
- Noder som ej är rot eller löv skall ha mellan m/2 till m antal "barn".
- Längden i antal noder från en rot till alla dess löv skall vara lika lång.

För exempel, se figur 2.4.



Figur 2.4 - B-träd av CyHawk [20], CC BY-SA 3.0.

En ytterligare metod vilket används för indexering av spatialdata är i form av r-träd struktur. Denna används av biblioteket "Neo4j Spatial". R-träd är en utbyggnad av b-träd. Skillnaden mellan r-träd och b-träd är att b-träd sorterar sin data utifrån definitiva värden (exempelvis siffervärden) medan r-träd använder sig av "boxar" för att organisera data. Se figur 2.5 nedan för visuell beskrivning.

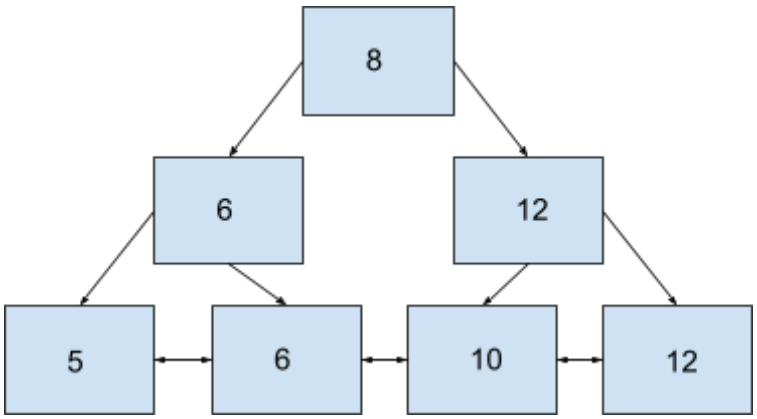


Figur 2.5 - Struktur av lagring av data i r-träd [21].

B^+-träd är en utöver de tidigare strukturerna b-träd och r-träd en struktur vilket används till spatialindex i Neo4j native [22]. Spatialindex i Neo4j skapas automatiskt i kombination med att ett index skapas för en nod eller relation med spatialdata. Datan lagras antingen utifrån ett tvådimensionellt eller tredimensionellt lager med hjälp av en B^+-träd struktur. Spatialindex appliceras på punkter, detta då det enbart är den formen av spatialdata Neo4j native erbjuder [22]. Dessa lagras sedan i fyra separata träd vilket baseras på den tidigare B^+-träd strukturen. De fyra olika träden representerar de fyra olika koordinatsystem formaten. Dessa är:

- WGS-84: longitud, latitud (x, y)
- WGS-84-3D: longitud, latitud, höjd(x, y, z)
- Kartesiska: x, y
- Kartesiska 3D: x, y, z

En B^+-träd struktur är relativt lik en b-träd struktur. Skillnaden mellan de två består av att B^+-träd sammankopplar de nedersta löven i dess struktur i form av en länkad lista. Detta möjliggör för sökning att fortsätta genom att gå vidare till nästa löv utan att först behöva gå tillbaka till roten för det nuvarande lövet [23]. Figur 2.6 följer ett exempel på ett B^+-träd.



Figur 2.6 - Struktur av lagring av data i B^+-träd.

2.4 Spatialdata

Spatialdata är data om ett geografiskt område och dess egenskaper. I syfte att specificera och identifiera området används koordinater av både longitud och latitud. Information förekommande inom spatialdata kan delas upp i två olika typer, raster- och vektordata. Relevant för arbetet är den spatiala datatypen vektordata vilken består av punkter, linjer och polygoner. Punkter beskrivs med koordinater, oftast longitud och latitud. Linjer består av ihopsatta punkter. En polygon skapas av linjer vilket sätts ihop i en särskild ordning och stängs där första och sista punkternas koordinater är samma [24]. Dessa kan representera flertalet olika typer av geografiska objekt. Exempelvis kan hus representeras som punkter, vägar eller åar kan representeras med linjer och polygoner kan representera sjöar eller områden. För att beskriva spatialdata i sträng format finns WKT (Well Known Text) vilket är en OGC (Open Geospatial Consortium) standard [25]. En punkt kan med detta format beskrivas på följande sätt, POINT(longitud, latitud).

2.5 Neo4j Spatial

I Neo4j finns det inbyggt tre spatiala funktioner. Dessa är “point.distance()” vilket returnerar det geodetiska avståndet mellan två punkter och “point.withinBbox()” vilket används för att avgöra om en punkt befinner sig inom en låda vilket begränsas av en angiven koordinat i lådans övre högra och nedre vänstra hörnen. Sist finns funktionen “point()” vilket används för att skapa punkter [26]. Endast en spatial datatyp stöds i Neo4j vilket är en punkt och representerar en koordinat. Punkter associeras med ett koordinatreferenssystem vilket definierar om en punkt är en geografisk koordinat eller en kartesisk koordinat.

För att erhålla utökat stöd för spatiala funktioner och datatyper finns ett spatialt bibliotek till Neo4j i form av ett plugin [27]. Biblioteket har sin början i 2010 och inspirerades av PostGIS till att ge geografiska förmågor till Neo4j. Ett antal funktioner stöds men viktigaste är

“spatial.intersect” vilket returnerar alla geometrier vilket korsar den angivna geometrin samt “spatial.withinDistance” vilket returnerar alla geometrier och deras avstånd, sorterade, inom ett angivet avstånd till en angiven koordinat. För att möjliggöra användandet av dessa funktioner på spatialdata krävs skapandet av ett lager. Man kan lägga in noder i ett lager och benämna vilken egenskap av noden där den spatiala datan lagras. Lagret utnyttjar en r-tree index vilket används för funktionerna. I figur 2.7 visas ett exempel där ett WKT lager skapas, insättning av noder till lagret samt ett funktionsanrop där lagret används. För att använda detta lager krävs det att den spatiala datan sparas i WKT-format [28].

```
CALL spatial.addWKTLayer("HallplatsLayer", "Geo")

MATCH (n:Hallplatser)
CALL spatial.addNode("HallplatsLayer", n) YIELD node
RETURN node

CALL spatial.withinDistance("HallplatsLayer", fk.Geo, 10.0) YIELD node, distance
```

Figur 2.7 - Ett WKT lager skapas för hållplatser och används med funktionen spatial.withinDistance.

2.6 SQL Server Spatial

SQL Server har stöd för lagrandet av spatialdata. Detta kommer huvudsakligen i två former, geometrisk och geografisk data. Geometrisk data representerar data utifrån ett “platt” koordinatsystem där det inte tar jordens utformning i åtanke. Geografisk data däremot representerar dess data med ett koordinatsystem där jordens utformning tas i åtanke [29].

SQL Server stödjer flertalet olika typer av spatialdata. Av dessa är, inom ramen av arbetet, punkter, polygoner och multipolygoner de mest intressanta. Punkter representerar en geografisk punkt i form av latitud och longitud. Det kan även innehålla ett z värde för att representera punktens höjd [30]. Polygoner representerar en tvådimensionell yta bestående av en samling ihopkopplade punkter [31]. Multipolygoner består av en samling av noll eller flera instanser av polygoner [32]. För att manipulera dessa typer av data erbjuder SQL Server en stor mängd funktioner. Likt datatyperna är några av dessa mer intressanta för arbetet vilket är STDistance och STIntersects. STDistance används för att beräkna det kortaste avståndet mellan två geografiska instanser t.ex avståndet mellan två hus eller avståndet från ett hus till en sjö. STIntersects används för att ta reda på om två geografiska instanser korsar varandra t.ex om ett hus är i ett specifikt område [33].

3 Genomförande

Genomförandet ämnar att beskriva hur arbetet genomförts. Den beskriver de val och vägar som tagits under arbetets gång och hur dessa påverkat det slutliga resultatet. Genomförandet baseras på det som beskrivs i 3.1 då detta beskriver den metod som framtagits och definierats i planeringsrapporten.

3.1 Metod

Arbetet avses utföras med hjälp av en agil arbetsmetod där projektet delas upp i sprints. Varje sprint sträcker sig en vecka och vid start av varje sprint hålls ett planeringsmöte. Där bestäms veckans arbetsuppgifter. Detta har avsikten att ge projektet god struktur och dokumentation där arbetet kan delas upp i mindre delar. Veckovis hålls separata handledarmöten med handledare från både Chalmers och Värderingsdata.

Majoriteten av arbetet kan delas upp kring fyra arbetsområden: Relationsdatabasen, grafdatabasen, hantering av data och testerna. Det första som skall göras är att ta fram en MSSQL relationsdatabas som efterliknar det system Värderingsdata använder sig av idag. När detta är gjort skall en Neo4j-grafdatabas, med utgångspunkt i relationsdatabasen, tas fram.

Vid färdigställandet av de två databaserna skall data inhämtas och formateras korrekt för att möjliggöra dess hantering av de två framtagna databaserna. Planerat är att aktuell data, nödvändig för genomförandet av tester, tillhandahålls av värderingsdata där privat information mörkläggs. En stor mängd data skall matas in i grafdatabasen, drygt 2 miljoner fastighetsobjekt samt strax under 3 miljoner transaktioner. Detta innebär att det kommer behövas ett script för att automatisera detta då manuell inmatning ej är rimligt för den mängden data.

När relations -och grafdatabasen är färdigställda ska testerna genomföras. För att säkerställa att testresultaten inte påverkas av hårdvara kommer dessa genomföras på en VM (virtuell maskin) framtagen av Värderingsdata. Testerna kommer baseras på de tre angivna scenarierna där likvärdiga queries för de två databaserna skapas och tiden noteras. Scenarierna vilket testerna skall baseras på är de följande.

- Matcha fastigheter/transaktioner till information om ett spatialområde.
- Matcha fastigheter/transaktioner till närmaste intresseområden.
- Matcha fastigheter/transaktioner till närmaste intressepunkter.

3.1.1 Val av databashanteringssystem

De två verktygen ämnade att användas är MSSQL och Neo4j. MSSQL har valts därför att det är det DBHS Värderingsdata i dagsläget huvudsakligen använder sig av. Detta ger därmed den bästa representationen av deras system vilket senare kan användas i testerna. Den grafdatabas vilket valts i syfte att jämföras med MSSQL är Neo4j. Detta beslut gjordes efter

viss efterforskning kring olika typer av grafdatabaser och vad de har att erbjuda. Neo4j är den populäraste och mest använda grafdatabasen med en stor och aktiv användarbas bakom sig. Många funktioner, exempelvis dess språk, skalbarhet och arkitektur anses väl implementerade och överträffar andra populära grafdatabaser som InfinityGraph, OrientDB och AllegroGraph [28]. Vidare har Värderingsdata funderat på Neo4j som ett möjligt alternativ för att hantera dess data.

3.1.2 Insamling av data

Datan för projektet kommer huvudsakligen bestå av de genomförda testerna av de två databaserna. Tiden det tar att utföra queries för de tre tidigare nämnda scenarierna är den delen av testerna vilket kommer vara av intresse för att jämföra prestandan av de två databastyperna.

3.2 Uppsättning av SSMS och SQL Server

För att påbörja arbetet med att ta fram en relationsdatabas installerades SSMS. Detta är gränssnittet varpå all interaktion skett med relationsdatabasen: Importering och exportering av data, designandet av relationsdatabasen, framtagandet av queries och utförandet av SQL-testerna. SSMS erbjuder verktyg för detta men också möjligheten att skriva scripts i ett fönster för att interagera med databasen. En kopia av en databas med tabeller innehållande data om fastigheter vilket är grunden för scenarierna togs emot från Värderingsdata och återställdes via SSMS. Varje individuell tabell analyserades för att förstå dess innehåll och hur datan relaterade till varandra. Databasen hade från början ej några satta primärnycklar, främmande nycklar eller indexes. Detta innebar att det behövdes undersökas vad för sådana alternativ fanns tillgängliga.

3.2.1 Datamängder

De ursprungliga datamängderna tilldelade från Värderingsdata bestod av sex stycken tabeller. Dessa tabeller inriktar sig på småhusbyggnader vilket är en delmängd av den data Värderingsdata hanterar. Denna datamängd användes för att prestandatesta de två databaserna och är grunden för de tre spatiala scenarierna som testats. Nedan beskrivs varje tabell.
Deso: Innehåller information om strax under 6000 områden i Sverige. Dessa geografiska områden representeras med polygoner. Syftas att användas för scenariot där en fastighet matchas till information på ett spatialområde.

DesoStatistik: Innehåller information om de specifika deso-områdena. Tabellen innehåller namnet för deso-området, ett id för statistiken, vilket år det representerar samt en värdesiffra för just det området och året.

FastighetsKoordinater: Innehåller information om en fastighets geografiska position i form av en punkt. Tabellen är central för databasen då den flitigt används för att relatera en fastighet till ett geografiskt intresseområde eller intressepunkt.

Fastigheter: Representerar en individuell fastighet och nödvändig information om den. Taxeringsvärde och nummervärde används för att identifiera en unik fastighet och koppla den till en byggnad. En fastighetsnyckel kopplar fastigheten till tabellen med transaktioner eller koordinater. Resterande information som lagras i denna tabell är län, kommun och area på fastigheten.

SmåhusByggnader: Identifierar enskilda småhus utifrån ett taxeringsvärde och nummervärde. En värdeyta lagras också i tabellen vilket representerar storleken på huset. Taxeringsvärdet och nummervärdet kan användas för att matcha byggnaden till den fastighet den tillhör.

Transaktioner: Innehåller information om enskilda transaktioner. Transaktionsid, datum för transaktionen, kommunnummer, fastighetsnyckel, latitud och longitud för fastigheten, pris och spatial information om fastighetens position i form av en punkt.

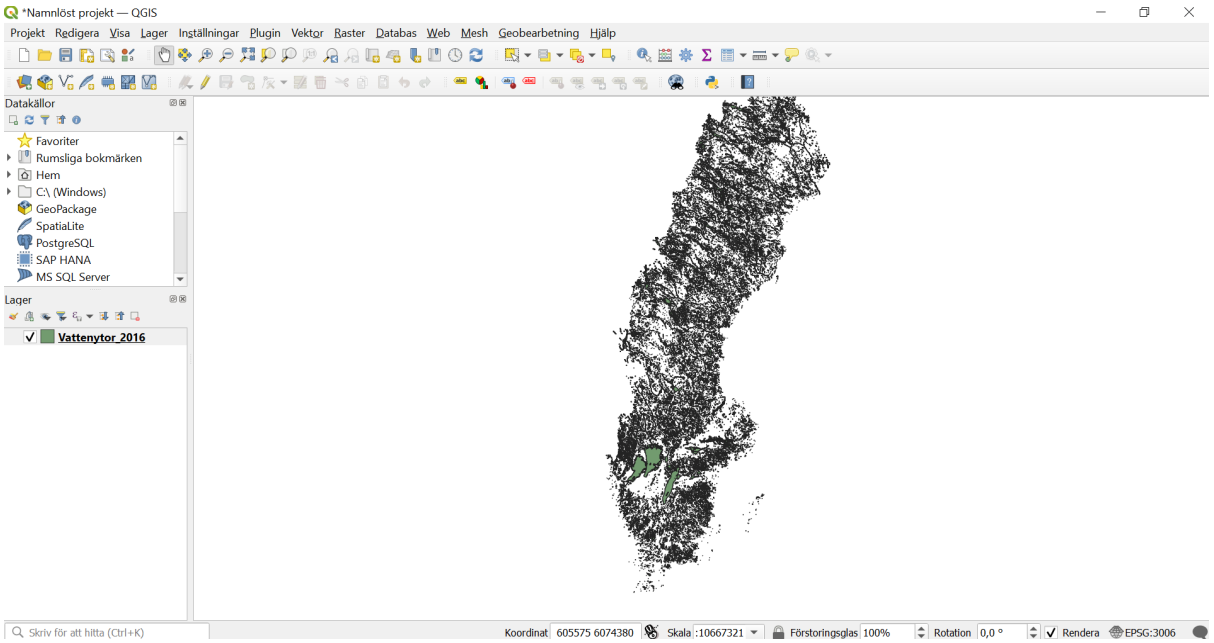
Vidare importerades två datamängder från öppna källor. Intresseområden i form av vattenytor och intressepunkter i form av hållplatser.
Hållplatser: Innehåller alla hållplatser i Sverige och varje hållplats har ett id, namn, latitud och longitud samt en spatial representation i form av en punkt bestående av dess koordinater. Datamängden för hållplatserna var tillgängliga att komma åt via Trafiklab [35] där den hämtades i form av en flat fil för att sedan importeras till databasen.

VattenYtor: Innehåller en publik datamängd från smhi som innefattar mindre och större samlingar vattenytor i Sverige [36]. Dessa ytor representeras med multipolygoner.

3.2.2 Importering och exportering av data

Det nämndes tidigare att data importerades från öppna källor och lades in i två tabeller, vattenytor och hållplatser. Denna process skedde i flera steg och skiljde sig åt mellan vattenytor och hållplatser då de hämtades med olika format.

Processen för införskaffning och importering av data relaterat till vattenytor gjordes först genom att hämta en shapefil samt associerade filer med datamängden vattenytor(SVAR2016) från smhi. Denna innehöll en representation av Sverige med sjöar inkluderade. QGIS, ett verktyg för att behandla och skapa geospatial information [37], användes där ett vektorlager skapades med den nedladdade shapefilen.



Figur 3.1 - Vektorlager bestående av vattenytor i Sverige skapat med shapefilen.

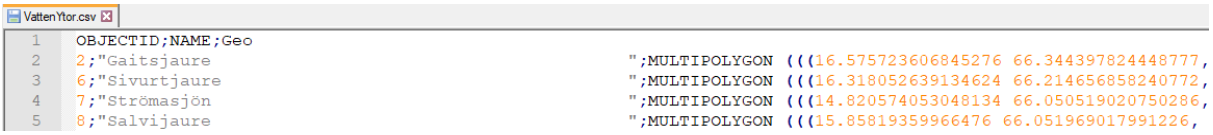
Därefter exporterades denna datan i form av en GeoJSON fil. Slutligen användes ett program skrivet i C# skapat med stöd från handledare för att importera datan från GeoJSON filen till en tom tabell vid namn VattenYtor vilken innehöll relevanta kolumner. Microsoft entity framework core och NetTopologySuite utnyttjades för detta ändamål.

Processen för att importera hållplatser till databasen gick ut på att erhålla en API nyckel från trafiklab [35]. Med denna nyckel kunde datasetet GTFS Sverige 2 laddas ner vilket innehöll en flat fil med hållplatser i Sverige. För att importera denna utnyttjades ett verktyg i SSMS som kopierar innehållet i flat-filen och möjliggör skapandet av en tabell [38]. Allra sist för att få en geografisk representation av hållplatserna skapades en kolumn i tabellen med punkter utifrån latitud och longitud. Detta gjordes med ett SQL-script vilket kan ses i figur 3.2 nedanför.

```
ALTER TABLE [dbo].[Hallplatser] ADD
[Geo] AS ([geography]::Point([stop_lat],[stop_lon],(4326)))
PERSISTED
```

Figur 3.2 - Tillägget av en Geo kolumn till hållplatser tabellen

Exportering av all data från MSSQL skedde genom användandet av csv-filer. Detta gjordes genom att hämta all tabelldata med hjälp av scripts. De geografiska datatyperna konverterades till WKT format för att användas av Neo4j:s spatial-bibliotek. Nedanför i figur 3.3 kan ett exempel på en csv-fil genererad i SSMS ses vilket i detta fall representerar tabellen VattenYtor.



Figur 3.3 - VattenYtor tabellens data representerad i en csv fil. OBJECTID, NAME och Geo är tabellens respektive kolumnnamn.

3.3 Uppsättning av Neo4j Desktop

För att påbörja arbetet med grafdatabasen behövdes Neo4j Desktop installeras. Genom Neo4j Desktop kunde en lokal grafdatabas skapas, hanteras och konfigureras. Neo4j Browser medföljde vilket var användargränssnittet där cypher queries kunde användas för att radera, uppdatera, skapa och avläsa grafdatabasen. Då det ursprungliga spatiala stödet från Neo4j var begränsat och endast hanterade punkter behövdes detta utökas för att även innefatta polygoner. Ett plugin, Neo4j Spatial, installerades för att kunna utföra spatial-beräkningar med polygoner vilket krävdes för scenarierna kring att matcha fastigheter till intresseområden.

För att möjliggöra användandet av funktioner från biblioteket “Neo4j Spatial” behövdes ett lager skapas till varje nod med en nodegenskap vilken lagrar geografisk information. Lagret var ett WKT lager vilket medföljer att den geografiska informationens format behövde vara WKT.

3.3.1 Importering av data

För att importera data till grafdatabasen användes csv-filer som tidigare exporterats från relationsdatabasen. För att göra detta följdes en guide på Neo4j:s hemsida kring hur importeringen av csv data går till [39]. Genom användandet av Neo4j Browser kunde eventuella index och constraints skapas. Ett exempel på båda kan ses nedan i figur 3.4 i form av noder av typen transaktioner. Indexerna användes vid framtagandet av queries för att snabba upp uthämtning av data. Ett constraint används för att förhindra att två identiska noder skapas.

```
CREATE CONSTRAINT UniqueTransaktioner FOR(n:Transaktioner) REQUIRE n.TransId IS UNIQUE

CREATE INDEX TransaktionsIndex FOR (n:Transaktioner) ON (n.FNKL)

CREATE INDEX TransaktionsGeo FOR (n:Transaktioner) ON (n.location)
```

Figur 3.4 - Skapandet av constraints och index för transaktionsnoder.

Efter skapandet av constraints och index laddades csv-filen in i grafdatabasen med cyphers “LOAD CSV” kommando. Transaktionsdatan kunde sen manipuleras med cypher för att

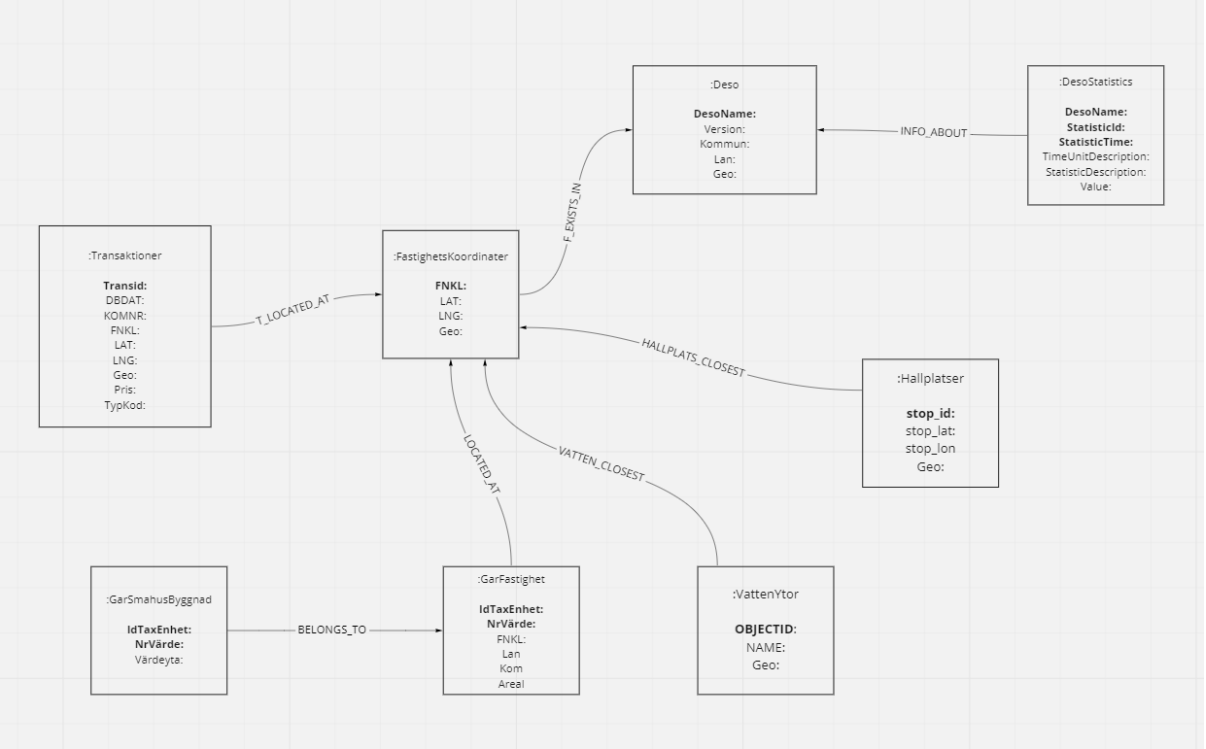
skapa noder, definiera dess egenskaper och konvertera datan. På grund av Neo4j:s ACID principer görs endast ändringar till databasen om en transaktion var lyckad och på det sättet hålls databasen konsistent efter att en transaktion har utförts. Vid importeringen av csv-filen transaktioner görs därför inga ändringar till databasen förrän alla ca 3 miljoner noder har skapats och deras data har konverterats korrekt. Den ännu inte inlagda datan för transaktionen sparas i Neo4j-minnet och för att inte få slut på plats behöver man dela upp en transaktion för att periodvis uppdatera databasen. Detta kan göras genom att utföra transaktionen i en subquery och lägga till "IN TRANSACTIONS OF 1000 ROWS" vilket kan ses nedan i figur 3.5.

```
:auto LOAD CSV WITH HEADERS FROM 'file:///Transaktioner.csv' AS row
CALL{
WITH row
CREATE (t:Transaktioner)
SET t.TransId = toInteger(row.Transid),
t.location = point({longitude: toFloat(row.LNG), latitude: toFloat(row.LAT)}), t.Geo = row.Geo,
t.latitude = toFloat(row.LAT), t.longitude = toFloat(row.LNG), t.TypKod = toInteger(row.TypKod), t.KOMNR =
toInteger(row.KOMNR), t.Date = apoc.date.parse(row.DBDAT, 'ms', 'yyyy-MM-dd'), t.Pris = toInteger(row.Pris), t.FNKL
= toInteger(row.FNKL)
} IN TRANSACTIONS OF 1000 ROWS
```

Figur 3.5 - Importering av data från csv filen Transaktioner där varje individuell rad i filen tilldelas en nod av typen Transaktioner.

3.3.2 Relationer

Något som uppmärksammades under inlärningen av Neo4j var att användandet av relationer var centralt och en av Neo4js styrkor. Detta för att cyphers visuella syntax och grafdatabasens grund i grafteori gör att skapandet och användandet blir intuitivt. Det togs därför tidigt fram en grafdatamodell med inspiration från relationsdatabasen. Se figur 3.6 för en visuell beskrivning av grafdatamodellen där alla relationer ritas upp.



Figur 3.6 - Relationsbeskrivning mellan olika typer av noder. Varje box representerar en typ av nod varpå relationer mellan kan ses på namnet på linjerna mellan två olika typer av noder.

Det stöttes dock på flertal problem vid skapandet av dessa relationerna. Detta berodde på att beräkningarna, specifikt de spatiala, var beräkningsmässigt tunga. En ytterligare bidragande faktor var de stora mängderna data som skulle beräknas, exempelvis relationen till transaktioner och fastighetskoordinater, två tabeller där vardera innehåller miljoner noder. Ett test gjordes därför där datamängden fastighetskoordinater skalades ner till 10000 noder och en relation skapades till det deso-område fastigheten befann sig inom. Datamängden deso innehöll cirka 6000 noder och att skapa dessa relationer tog 23 minuter att genomföra. I figur 3.7 kan detta cypher-query ses.

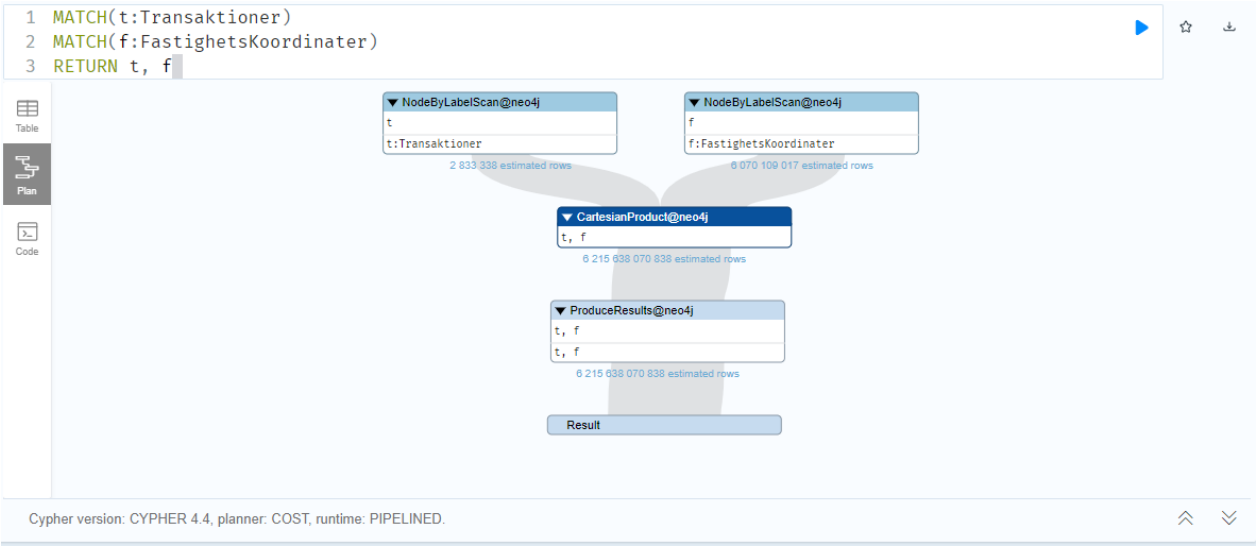
```
MATCH (n:FastighetsKoordinater)
CALL spatial.intersects('DesoLayer', n.Geo) YIELD node
MERGE (n)-[:F_EXISTS_IN]->(node)
```

Figur 3.7 - Cypher query för skapandet av relationer en fastighet och ett deso område.

Tiden tillägnad till att skapa relationer i Neo4j var ohållbar vilket resulterade i att de istället beräknades i MSSQL för att sedan exporteras och till slut importeras till Neo4j. Detta fungerade men var en omständig och fortfarande tidskrävande process då vissa beräkningar i MSSQL tog timmar. I denna fasen av arbetet tillkom insikten att en systemkonstruktion vilken utnyttjar relationer inte var optimal och gick emot arbetets syfte. Tanken var att undersöka om den tidsmässiga prestandan kunde förbättras utan att beräkna och aggregera datamängder i förväg. Av denna anledning användes inte relationer i systemkonstruktionen.

3.3.3 Arbete med prestanda

Något arbetet var centrerat kring var optimering av cypher queries. Det noterades tidigt att queries som involverade större mängder noder behövde utformas noggrant då de annars riskerar att bli för krävande för att kunna genomföras inom lämplig tid. Ett tydligt exempel på detta är när två matchningar sker i direkt följd, t.ex alla transaktioner och fastighetskoordinater. Detta resulterar i en kartesisk produkt vilket innebär att varje transaktion, runt 3 miljoner, matchas mot varje fastighetskoordinat, runt 2 miljoner. Se figur 3.8 för en visuell representation.



Figur 3.8 - Kartesisk produkt mellan Transaktioner och FastighetsKoordinater.

Ytterligare prestandaproblem som noterades var när stora mängder data skulle importeras. Detta resulterade i att heap:en (minne allokerat till queryn) tog slut vilket avbröt queryn. En lösning till detta var att genomföra queryn i transaktioner. Vad detta innebär är att queryn bearbetar en viss mängd rader, exempelvis tusen rader, importerar dessa vilket frigör minnet och sedan återupprepar detta tills all data har importerats [40].

En ytterligare prestandaförbättring kom i form av att öka det allokerade minnet Neo4j fick använda sig av. Den korrekta mängden kunde tas reda på genom att köra kommandot “neo4j-admin memrec” i en terminal tillhörande Neo4j. När väl detta gjorts visades rekommenderad mängd minnesallokering till olika delar av Neo4j. Detta erbjöd en förbättring av prestanda då Neo4j fick mer minne att arbeta med. Det noterades dock efter viss användning att den rekommenderade minnesmängden antog att systemet i fråga var dedikerat enbart till Neo4j. När väl andra tyngre processer kördes parallellt med ett krävande query frös datorn och behövdes startas om.

3.4 Framtagandet av queries för testning

Värdering av en fastighet görs genom att jämföra det aktuella objektet med närliggande transaktioner. Ju mer lik en närliggande transaktion är, desto mer väger dess pris in på värderingen av det aktuella objektet. Man vill därför skapa queries vilket tar fram de x närmaste objekten. Utifrån värderingsobjektet kan det också vara intressant att veta närmsta hållplats och vattenyta då avstånd till intresseområden och intressepunkter kan medfölja en viss värdeökning. Denna typen av query kallas för "closest neighbour query" och totalt har fem stycken queries av denna typ tagits fram. Information kring var värderingsobjektet ligger och i vilket område objektet befinner sig inom används också för att värdera en fastighet. Denna typ av query kallas för "point in polygon" eller "intersects" och totalt har två stycken queries av denna typ tagits fram. Utifrån de tre scenarierna har sju stycken queries tagits fram för varje databas. Dessa är de följande:

- Ta fram ett värderingsobjekt samt dess närmaste hållplats.
- Ta fram ett värderingsobjekt samt dess närmaste vattenyta.
- Ta fram ett värderingsobjekt samt relaterad deso information.
- Ta fram 50 transaktioner samt deras respektive närmaste hållplats.
- Ta fram 50 transaktioner samt deras respektive närmaste vattenyta.
- Ta fram 50 transaktioner samt deras respektive relaterad deso information.
- Ta fram de 50 närmaste transaktionerna från max 5 år tillbaka.

Varje query utgår från en byggnad som identifieras med dess taxeringsvärde och nummervärde. Denna byggnad matchas med dess fastighet vilket har en unik fastighetsnyckel, FNKL. För att få information kring den geografiska platsen matchas FNKL med tabellen FastighetsKoordinater där en punkt fås. Denna punkt matchas sen med ett subquery som utför "closest neighbour" beräkningen vilket kan vara att hitta närmsta vattenyta eller hållplats. Denna beräkning utnyttjar satserna **TOP** och **ORDER BY** [41]. Distansen mellan punkten och det spatiala objektet räknas ut med funktionen "STDistance( )" vilket används i satsen **ORDER BY** för att sortera listan utifrån avståndet i stigande ordning. **TOP** används för att ta ut den rad som är längst upp i tabellen då det är det närmsta objektet till punkten efter att ha sorterats av **ORDER BY**. Det är inte effektivt att titta på alla vattenytor eller hållplatser när man ska beräkna den närmsta då det görs icke relevanta beräkningar. En hållplats vars avstånd till ett hus är för långt adderar inget värde till huset. Det har därför applicerats ett filter i urvalet av spatiala objekt i **WHERE** satsen. Endast vattenytor eller hållplatser vars avstånd inte överstiger 10 km tittas på. I figur 3.9 kan ett "closest neighbour" query ses.

```
SELECT
    transaktioner.Transid,
    transaktioner.FNKL,
    V.NAME,
    V.VDistance
FROM transaktioner
CROSS APPLY(
    SELECT TOP(1)
        transaktioner.FNKL
        ,transaktioner.Geo.STDistance(VattenYtor.Geo) VDistance
        ,VattenYtor.NAME
    FROM  VattenYtor
    WHERE transaktioner.Geo.STDistance(VattenYtor.Geo)<10000
    ORDER BY(VDistance)) AS V
```

Figur 3.9 - SQL closest neighbour query.

För att matcha en fastighet med information kring ett intresseområde vilket fastigheten befinner sig inom användes funktionen “STIntersects( )”. I figur 3.10 kan detta ses när området deso och dess statistik matchas mot en fastighet.

```
SELECT
    transaktioner.Transid,
    transaktioner.FNKL,
    Deso.Geo,
    DesoStatistics.StatisticId,
    DesoStatistics.Value,
    DesoStatistics.StatisticTime
FROM transaktioner
JOIN Deso ON transaktioner.Geo.STIntersects(Deso.Geo) = 1
JOIN DesoStatistics ON Deso.DesoName = DesoStatistics.DesoName
```

Figur 3.10 - SQL intersect query.

I de scenarier då de femtio närmsta transaktionerna till ett objekt behövdes tas fram gjordes detta med ett “closest neighbour” query. Ett filter användes där endast transaktioner inom en radie av 20 km inkluderades. Detta var nödvändigt för att förbättra prestandan då det finns runt 3 miljoner transaktioner och att sortera alla på deras avstånd var inte rimligt då det tar för lång tid. Detta var mest märkbart för Neo4j då ett cypher query inte gick att slutföra om ett sådant filter inte användes.

```
DECLARE @FNK bigint;
SET @FNK = (SELECT FNKL FROM Fastighet WHERE (Fastighet.IdTaxEnhet = '105374-9' AND Fastighet.NrVarde = 1))

;WITH transaktioner (Transid, FNKL, Geo)
AS
(
SELECT TOP(50)
    Transaktioner.Transid,
    Transaktioner.FNKL,
    Transaktioner.Geo
FROM Transaktioner
JOIN FastighetsKoordinater ON FastighetsKoordinater.FNKL = @FNK AND Transaktioner.FNKL != @FNK
     AND Transaktioner.Geo.STDistance(FastighetsKoordinater.Geo) < 20000
ORDER BY(Transaktioner.Geo.STDistance(FastighetsKoordinater.Geo)))
```

Figur 3.11 - SQL 50 transaktioner subquery.

För att utföra “closest neighbour” beräkningen i cypher användes funktionen “spatial.withinDistance( )” vilket finns tillgängligt via biblioteket “Neo4j Spatial”. Denna funktion returnerar alla noder från det angivna lagret, sorterade efter avstånd, vilket finns inom ett angivet avstånd till en angiven koordinat. I figur 3.12 kan detta ses där alla noder i lagret “VattenLayer” returneras vilket finns inom 10 km till t.location vilket är koordinaten till en transaktion. De returnerade noderna aggregeras i en lista med funktionen “collect( )” och då de returnerade värderna sorteras efter avstånd får man den närmaste noden genom att ta det första värdet i listan med funktionen “head( )”.

```
WITH t CALL spatial.withinDistance("VattenLayer", t.location, 10.0) YIELD node, distance
RETURN t.TransId, head(collect([t.TransId, node.Name, distance])) AS ClosestVatten
```

Figur 3.12 - Cypher closest neighbour query.

För att uföra “intersect” beräkningen i cypher användes funktionen “spatial.intersect( )” vilket returnerar alla noder i det angivna lagret vilket korsar den angivna koordinaten.

```
WITH t CALL spatial.intersects("DesoLayer", t.location) YIELD node
RETURN t.TransId AS DesoTransId, node.DesoName AS desoname
```

Figur 3.13 - Cypher intersects query.

Vid framtagandet av närliggande transaktioner till en fastighet i cypher användes en inbyggd funktion i Neo4j, “point.withinBBox()”. Denna funktion kollar om en punkt finns inom en låda som definieras av en punkt i dess nedre vänstra och övre högra hörnen. Det var inte möjligt att skapa ett lager till datamängden Transaktioner då dess query inte gick att slutföra. Slutsatsen drogs att det var för många noder för spatial biblioteket att hantera.

```
WITH fk
MATCH (t:Transaktioner)
WHERE point.withinBBox(t.location,
point({longitude: (fk.Lng - 0.2), latitude: (fk.Lat - 0.2)}),
point({longitude: (fk.Lng + 0.2), latitude: (fk.Lat + 0.2)})) AND fk.FNKL <> t.FNKL
RETURN t, point.distance(t.location, fk.Geo) AS Dist ORDER by Dist ASC LIMIT 50
```

Figur 3.14 - Cypher transaktioner subquery.

3.5 Utförandet av testerna

Testerna hade tidigt planerats att genomföras på en dedikerad VM (virtuell maskin) för att kunna specificera den miljön och hårdvara som tilldelades de två databaserna. Namnet på den specifika typen av vm som användes var "Standard D4s v4" och dess specifikationer kan ses i figur 3.15. Det operativsystem som kördes på den virtuella maskinen var windows 11.

Standard_D4s_v4

Virtual Machine

Name	**Standard_D4s_v4** **Standard** is recommended tier **D** – General purpose compute **4** – The number of vCPUs **s** – Premium Storage capable **v4** – version
vCPUs	4
Memory	16 GiB
Hyper-V Generations	V1,V2
Azure Compute Units (ACUs)	195
Performance Score	65910
Proccessor	Intel(R) Xeon(R) Platinum 8272CL CPU @ 2.60GHz
NUMA Nodes	1
Max Network Interfaces	2
RDMA Enabled	no
Accelerated Networking	yes

OS disk size	1023 GiB
Resource (temporary) disk size	0 GiB
Max disks	8
Support Premium Disk	yes
Uncached Disk IOPS	6400

Figur 3.15 - Specifikationer över den dedikerade vm tillägnad åt testerna.

På den dedikerade virtuella maskinen installerades SQL Server 2019 samt SSMS 18 för att kunna genomföra testerna på relationsdatabasen. En kopia av relationsdatabasen som tagits fram under arbetets gång återställdes via SSMS. Neo4j Desktop 1.4.14 installerades varpå en lokal grafdatabas skapades med versionen 4.4.3. För att få tillgång till nödvändiga spatiala funktioner installerades biblioteket "Neo4j Spatial" i from av en .jar fil [42]. Detta krävde ändringar i konfigurationsfilen för grafdatabasen på grund av att viss funktionalitet i biblioteket använder sig av flertalet trådar, något Neo4j stoppar om det inte explicit tillåts [43]. Ett ytterligare bibliotek "apoc" installerades för att hantera och spara datum. Konfigureringen för den minnesallokering Neo4j fick tilldelat ändrades manuellt för att hjälpa med prestanda. Heap minnet tilldelades 5.1 gigabyte och page cache minnet tilldelades 7 gigabytes.

Allra sist importerades all data i form av noder med hjälp av cypher scripts, dessa kan hittas i appendix B. Testerna utfördes på tio olika fastigheter spridda runt om i Sverige där alla sju queries kördes på varje fastighet från fastighet 1 till 10. Totalt gjordes testerna i tre omgångar och under omgång 1 var de två databaserna i ett “uppvärmt” tillstånd. Att databasen är i ett uppvärmt tillstånd innebär att data finns i primärminnet och inte lika många disk I/O operationer behöver utföras. Under omgång 2 och 3 var de två databaserna inte uppvärmda utan kördes från en “kall” start. Detta gjordes för Neo4j genom att starta om grafdatabasen och för SQL server användes två kommandon, “DBCC FREEPRCCACHE” och “DBCC DROPCLEANBUFFER”.

Motivationen för hur testerna utfördes baseras i hur en värdering går till där angiven data matchas för en fastighet i syfte att användas i en värderingsmodell. Att titta på tio olika fastigheter ger en uppfattning av hur de två databaserna presterar vid hanterandet av olika mängder data. Att utföra testerna 3 gånger gjordes för att se om samma resultat kunde erhållas och huruvida resultatet kan hänvisas till den data vilket hanteras eller databasens tillstånd.

4 Systemkonstruktion

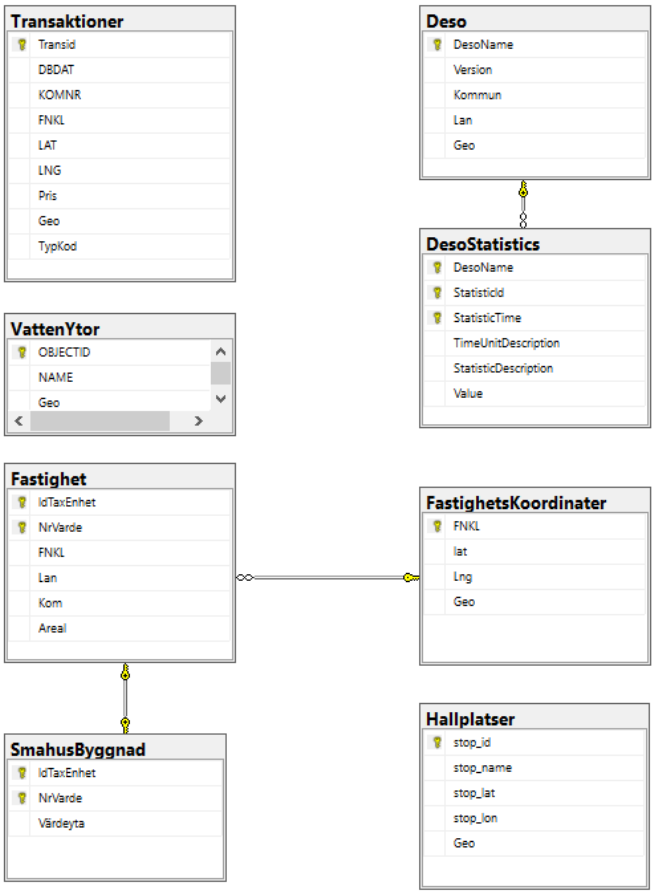
Här beskrivs den relationsdatabas och grafdatabas vilket använts och framtagits i syfte att utföra en jämförelse.

4.1 SQL Server

Här beskrivs relationsdatabasen, MSSQL, och dess konstruktion vars syfte är att jämföras med grafdatabasen. MSSQL består av åtta stycken tabeller. Dessa är följande:

- Deso
- DesoStatistics
- FastighetsKoordinater
- Fastighet
- SmahusByggnad
- Transaktioner
- Hallplatser
- VattenYtor

Relationerna mellan dessa tabeller kan beskrivas med hjälp av ett er-diagram vilket kan observeras i figur 4.1.



Figur 4.1 - ER-diagram över tabellerna i relationsdatabasen.

Datatyperna i relationsdatabasen består av tre olika spatiala datatyper. Dessa är punkter, polygoner och multipolygoner. De tabeller som använder sig av dessa datatyper är följande:

- Fastighetskoordinater, Transaktioner och Hållplatser
    - Använder sig av punkter för att beskriva dess geografiska position.
- Deso
    - Använder sig av polygoner för att beskriva det området deso innefattar.
- Vattenytor
    - Använder sig av multipolygoner för att beskriva vattenytans område.

I relationsdatabasen användes klustrade index och spatialindex flitigt för att snabba upp queries. Klustrade index skapades i och med att primärnycklar definierades för tabellerna. Spatialindex skapades individuellt för den spatialdata databasen hanterar.

4.2 Neo4j

Här beskrivs grafdatabasen och dess konstruktion vars syfte är att jämföras med relationsdatabasen. Grafdatabasen består av noder. Dessa representerar de tabeller relationsdatabasen består av. Noderna är följande:

- Deso
- DesoStatistik
- FastighetsKoordinater
- Fastighet
- SmahusByggnad
- Transaktioner
- Hållplatser
- VattenYtor

Då relationer ej använts finns inga direkta relationer mellan de olika typerna av noder likt de i relationsdatabasen i form av främmande nycklar. Alla noder är istället självstående och har ingen koppling till andra noder bortsett från vilken typ av nod de är.

Neo4j stödjer inte spatialdata utöver punkter och därför lagras den typen av data i WKT format. Detta innebär att data från vissa tabeller i MSSQL exporteras i WKT format för att göra det kompatibelt med Neo4j. Detta gäller för alla tabeller och nodtyper vilket innehåller någon typ av spatialdata. I grafdatabasen används även flertalet indexes med olika strukturer, beroende på dess användningsområde. Index framtagna med Neo4j native använder sig av index med en b-träd struktur. Index framtagna i samband med Neo4j Spatial använder sig däremot av en r-träd struktur.

Då grafdatabasen hade vissa prestandaproblem gjordes förändringar till dess minnesallokering. Detta gjordes i konfigurationsfilerna. Gränsen för mängden minne som kunde användas togs fram genom att köra kommandot “neo4j-admin memrec” via en terminal tillhörande grafdatabasen. Heap minnet tilldelades 5.1 gigabyte och page cache minnet tilldelades 7 gigabytes.

För Neo4j användes två plugins för att möjliggöra arbetet med spatialdata samt utöka antalet tillgängliga procedurer för queries. Dessa två plugins var Neo4j Spatial respektive Apoc.
I pluginet Neo4j spatial används lager för att bearbeta spatialdata. I grafdatabasen skapades tre lager varav varje representerade en nodtyp innehållande spatialdata.Vid skapandet av ett lager skapades också ett index för varje nod tillagd i lagret. Dessa tre lager är: DesoLayer, HallplatsLayer och VattenLayer.

5 Resultat

Här presenteras exekveringstiderna av de utförda testerna på grafdatabasen och relationsdatabasen. I det följande kommer i tabellerna MSSQL att benämnas SQL och Neo4j benämns cypher. Testresultaten baseras på de tre tidigare definierade scenarierna och utifrån dem har sju stycken queries framtagits vilka hämtar efterfrågad data från båda databaserna. Testerna har utförts manuellt genom Neo4j browser och SSMS i tre omgångar och med totalt tio fastigheter.

Testresultaten relaterade till fastighet 1 kommer uteslutas vid jämförelsen av de två databaserna gällande högsta och lägsta exekveringstid samt den genomsnittliga exekveringstiden för varje query. Detta görs för att få en rättvis jämförelse mellan omgångarna då testerna utfördes från en kall start för omgång 2 och 3. Högsta och lägsta exekveringstid presenteras för att få en uppfattning av det spann av exekveringstider vilket kan tillkomma medan den genomsnittliga exekveringstiden ger en övergripande uppfattning av hur de två databaserna presterar. En logaritmisk skala har använts för att presentera resultaten i diagrammen. Detta då både mycket små och stora resultat kan observeras från de olika testerna.

5.1 Testomgång 1

Resultaten för alla tester utförda på de två databaserna med ett okänt tillstånd där tidigare queries har utförts.

5.1.1 Tabell

Tabell 5.1 och 5.2 är tider för varje test utförda på tio fastigheter.

Cypher (ms)	1	2	3	4	5	6	7	8	9	10
Värdeobjekt + hållplatser	30	37	32	35	33	71	41	57	44	47
Värdeobjekt + vattenplatser	66	33	21	52	83	561	478	70	82	73
Värdeobjekt + deso	23	54	39	40	84	209	145	131	126	256
50 transaktioner + hållplatser	618	981	665	1128	1006	2157	865	1105	1307	887
50 transaktioner + vattenytor	6434	1225	761	3956	7764	13200	40420	3748	3954	887

50 transaktioner + deso	5961	8116	2110	5903	9774	10666	6769	10250	10650	15631
50 transaktioner + 5 år tillbaka	9	236	58	608	49	818	108	161	904	151

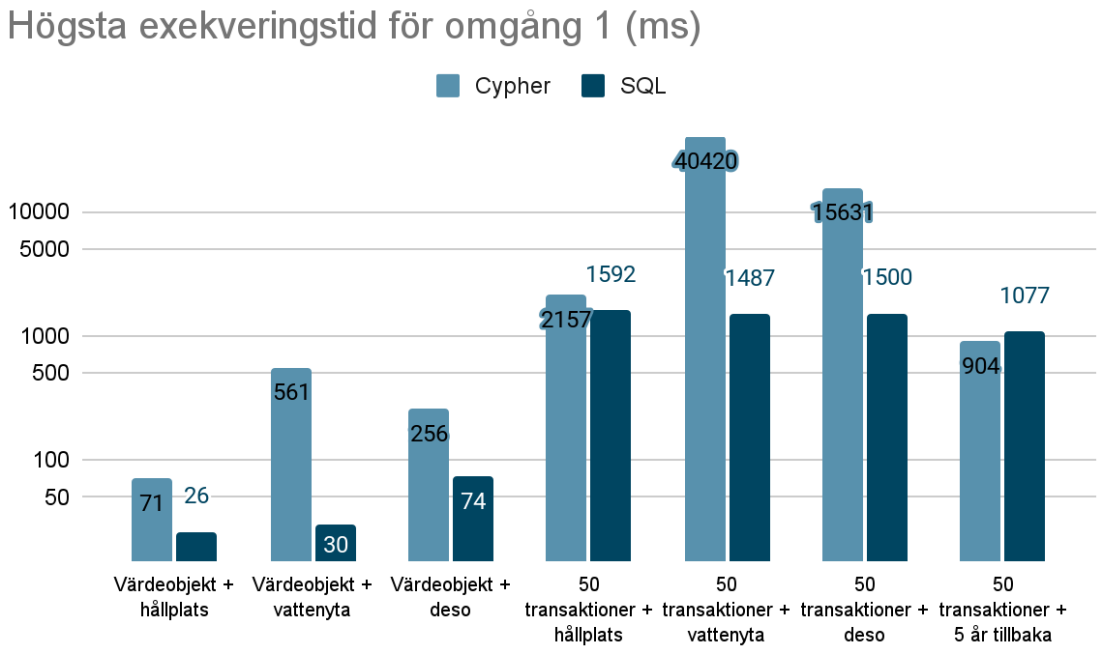
Tabell 5.1 - Cypher: Exekveringstid för testerna utförda på tio fastigheter. Tidsenheten är i ms.

SQL (ms)	1	2	3	4	5	6	7	8	9	10
Värdeobjekt + hållplatser	21	25	20	26	20	25	13	14	17	10
Värdeobjekt + vattenplatser	49	15	5	21	29	30	19	20	17	14
Värdeobjekt + deso	53	74	13	48	53	59	39	52	40	53
50 transaktioner + hållplatser	293	1146	152	1176	61	326	89	146	1592	299
50 transaktioner + vattenytor	119	291	83	1091	108	257	158	185	1487	283
50 transaktioner + deso	889	1244	648	1389	981	750	947	481	1500	620
50 transaktioner + 5 år tillbaka	17	197	48	746	15	124	37	81	1077	137

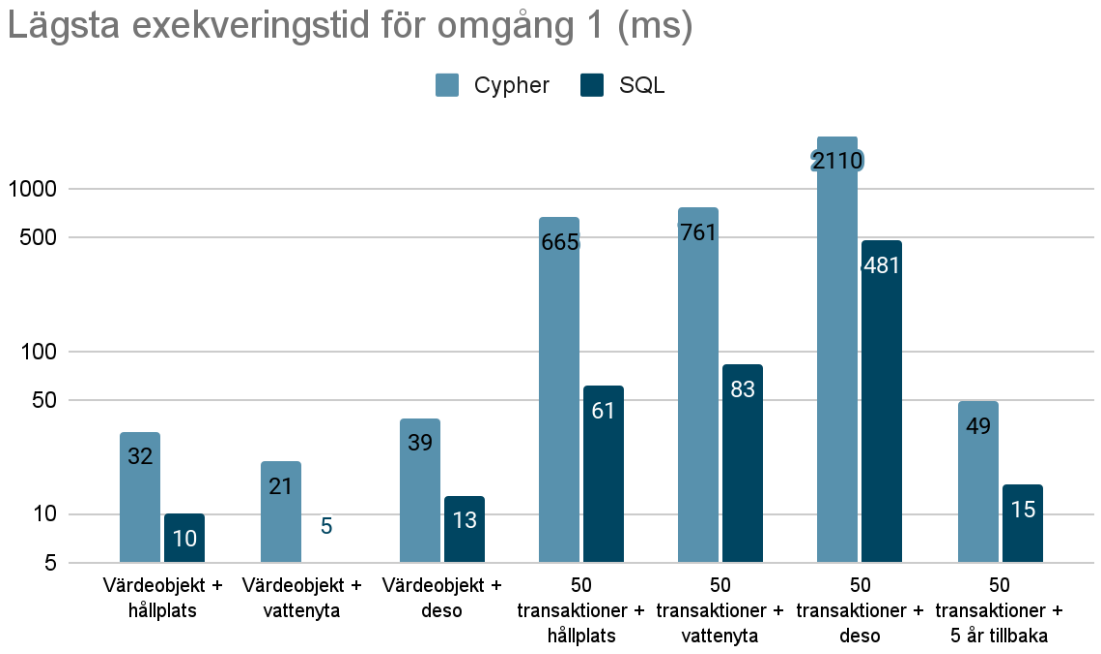
Tabell 5.2 - SQL: Exekveringstid för testerna utförda på tio fastigheter. Tidsenheten är i ms.

5.1.2 Diagram

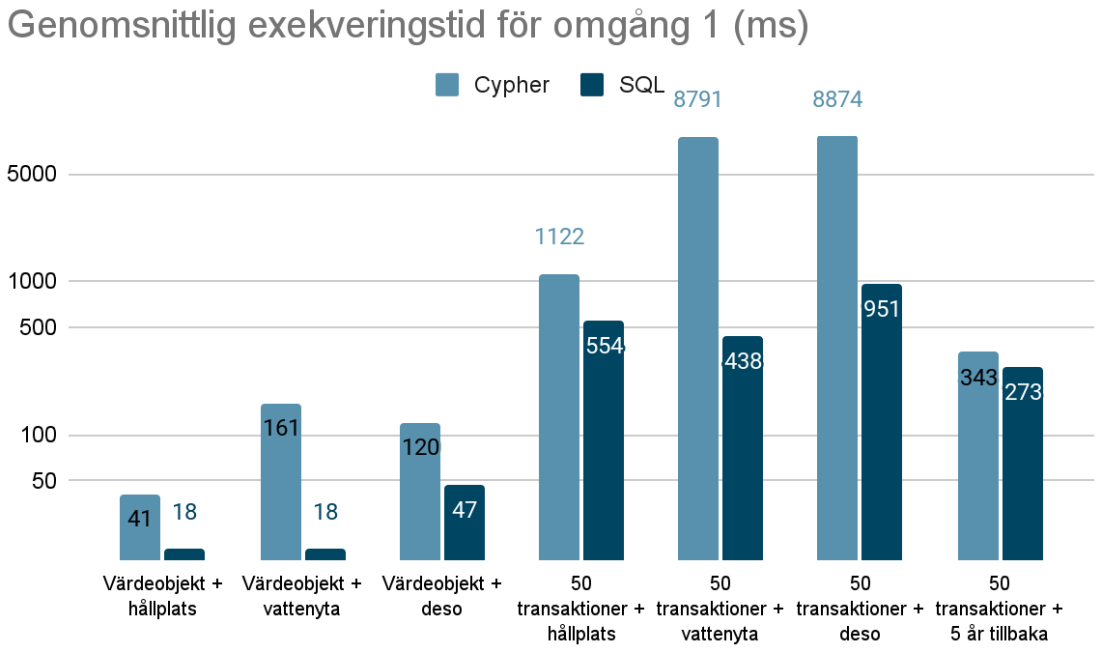
Figur 5.3, 5.4 och 5.5 är diagram skapade utifrån datan i tabell 5.1 och 5.2. De visar högsta, lägsta och genomsnittliga exekveringstiderna för de två databaserna.



Figur 5.3 - Högsta exekveringstid för testerna utförda på fastighet två till tio.



Figur 5.4 - Lägsta exekveringstid för testerna utförda på fastighet två till tio.



Figur 5.5 - Den genomsnittliga exekveringstid av testerna utförda på fastighet två till tio.

5.2 Testomgång 2

Resultaten för alla tester utförda på de två databaserna utifrån en kall start.

5.2.1 Tabeller

Tabell 5.6 och 5.7 är tider för varje test utförda på tio fastigheter.

Cypher (ms)	1	2	3	4	5	6	7	8	9	10
Värdeobjekt + hållplatser	703	89	44	53	46	51	47	38	49	45
Värdeobjekt + vattenplatser	249	51	59	79	114	307	534	69	65	109
Värdeobjekt + deso	133	152	54	66	148	206	183	174	290	519
50 transaktioner + hållplatser	1718	1971	972	1393	1101	1147	980	931	1701	1099
50 transaktioner	8740	1771	1028	4879	10945	16919	54369	5208	4982	5489

+ vattenytor										
50 transaktioner + deso	8643	10726	2879	8080	12703	14557	9133	13622	13972	21722
50 transaktioner + 5 år tillbaka	625	1037	108	674	50	166	69	104	1022	121

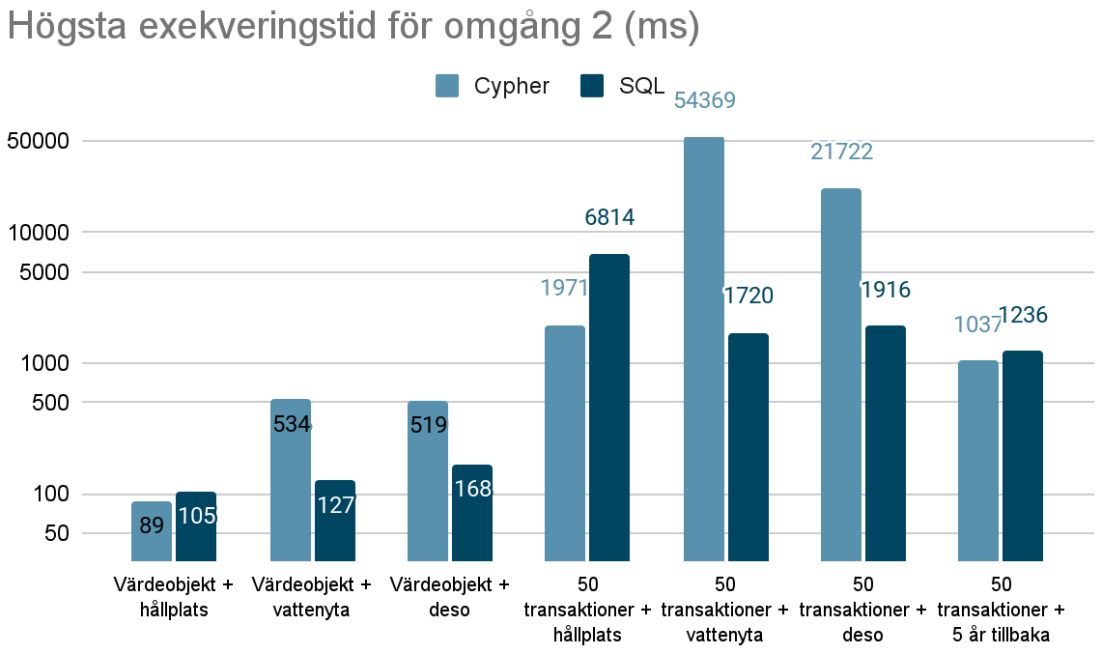
Tabell 5.6 - Cypher: Exekveringstid för testerna utförda på tio fastigheter. Tidsenheten är i ms.

SQL (ms)	1	2	3	4	5	6	7	8	9	10
Värdeobjekt + hållplatser	179	27	29	56	105	25	14	14	19	12
Värdeobjekt + vattenplatser	125	24	15	21	7	30	127	18	17	15
Värdeobjekt + deso	98	149	21	67	117	104	71	168	51	58
50 transaktioner + hållplatser	1908	6814	637	1960	91	468	109	193	2046	404
50 transaktioner + vattenytor	227	427	153	1426	254	333	231	257	1720	430
50 transaktioner + deso	1066	1565	776	1842	1225	863	1109	621	1916	810
50 transaktioner + 5 år tillbaka	16	202	57	1207	16	153	55	97	1236	233

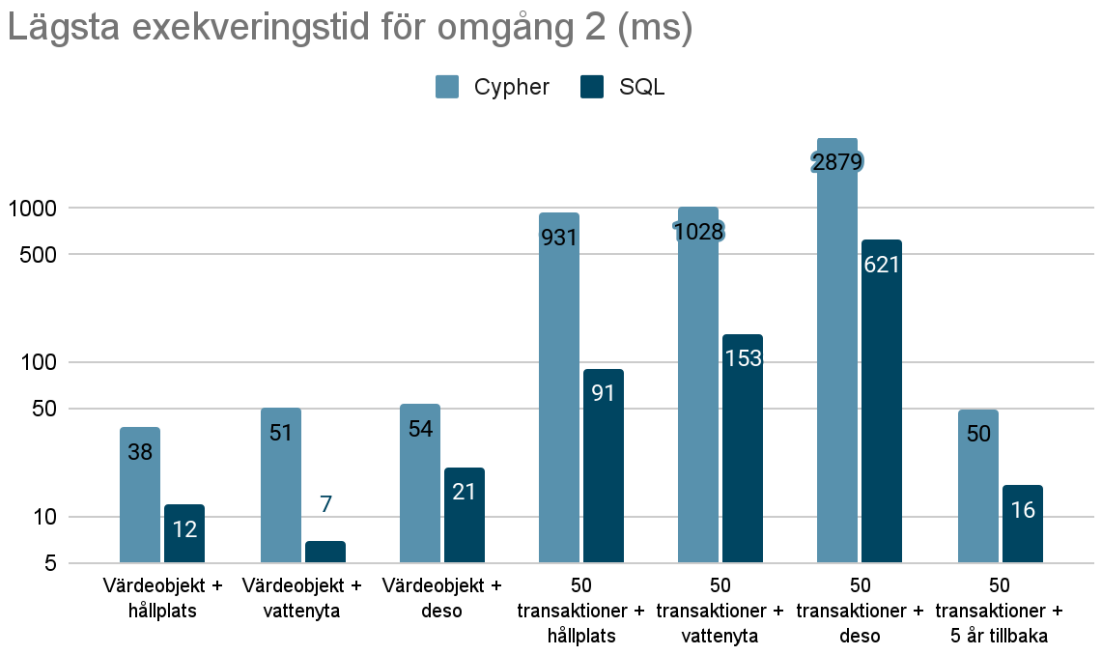
Tabell 5.7 - SQL: Exekveringstid för testerna utförda på tio fastigheter. Tidsenheten är i ms.

5.2.2 Diagram

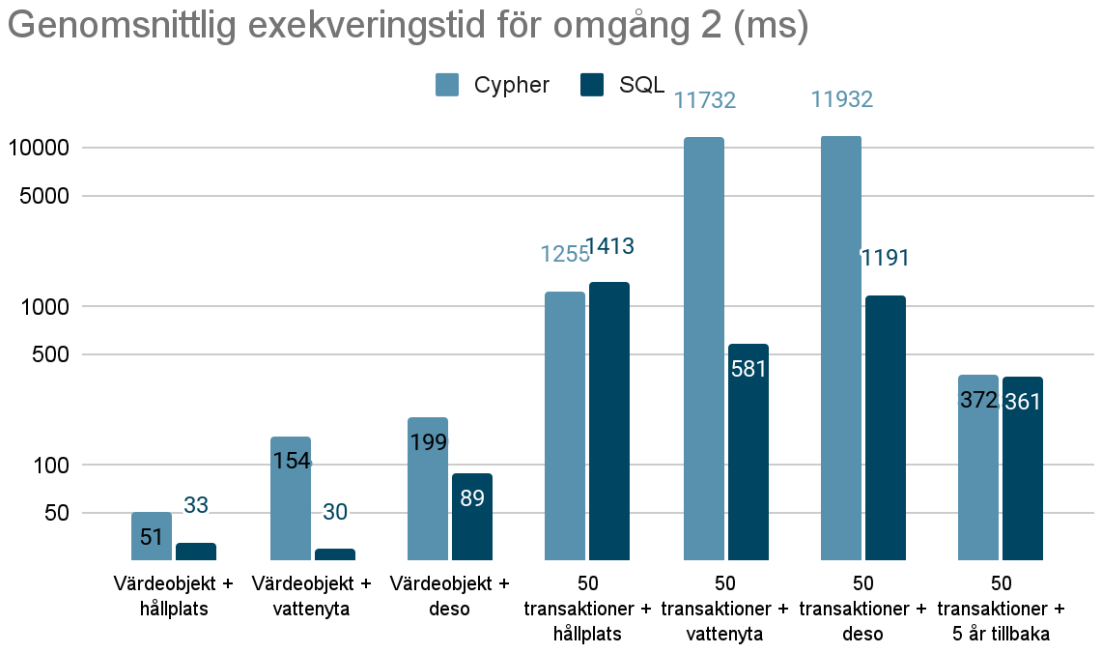
Figur 5.8, 5.9 och 5.10 är diagram skapade utifrån datan i tabell 5.6 och 5.7. De visar högsta, lägsta och genomsnittliga exekveringstiderna för de två databaserna.



Figur 5.8 - Högsta exekveringstid för testerna utförda på fastighet två till tio.



Figur 5.9 - Lägsta exekveringstid för testerna utförda på fastighet två till tio.



Figur 5.10 - Den genomsnittliga exekveringstid av testerna utförda på fastighet två till tio.

5.3 Testomgång 3

Resultaten för alla tester utförda på de två databaserna utifrån en kall start.

5.3.1 Tabeller

Tabell 5.11 och 5.12 är tider för varje test utförda på tio fastigheter.

Cypher (ms)	1	2	3	4	5	6	7	8	9	10
Värdeobjekt + hållplatser	662	53	43	54	56	46	41	57	52	40
Värdeobjekt + vattenplatser	300	50	50	66	79	305	424	85	82	84
Värdeobjekt + deso	144	130	48	95	156	177	198	222	161	425
50 transaktioner + hållplatser	1870	2134	883	1337	883	1320	847	1233	1581	1118
50	9088	2066	1197	5050	9243	17591	54244	4901	4840	5730

transaktioner + vattenytor										
50 transaktioner + deso	8219	10440	2898	7624	12208	14653	10933	13380	15456	21187
50 transaktioner + 5 år tillbaka	636	974	149	735	56	216	104	99	986	133

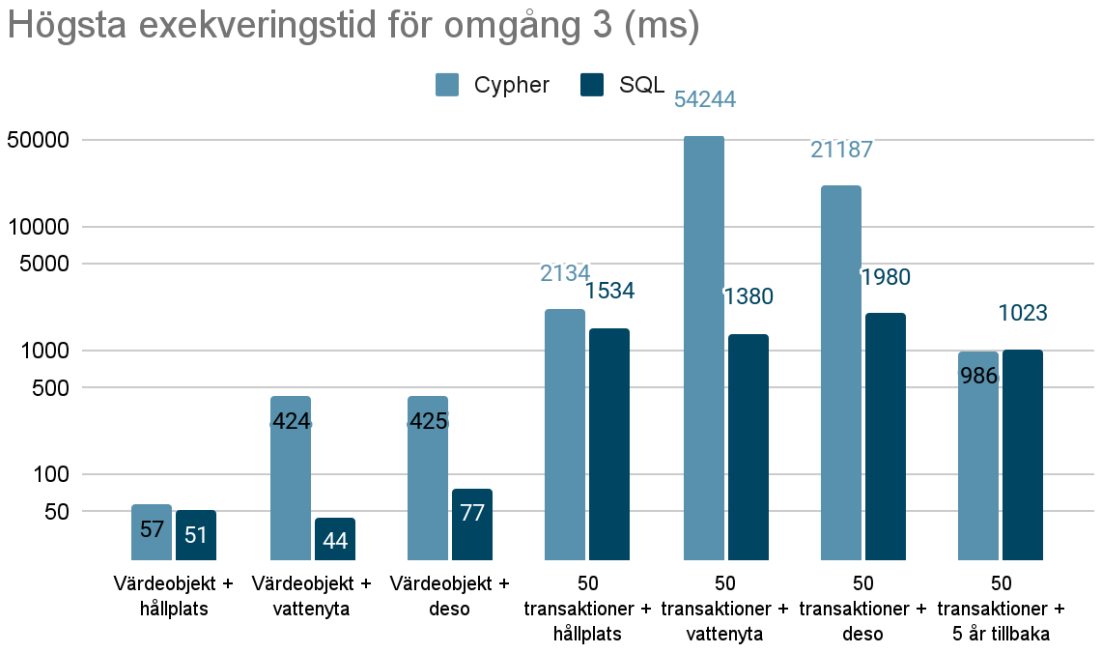
Tabell 5.11 - Cypher: Exekveringstid för testerna utförda på tio fastigheter. Tidsenheten är i ms.

SQL (ms)	1	2	3	4	5	6	7	8	9	10
Värdeobjekt + hållplatser	63	15	51	12	9	14	6	5	8	8
Värdeobjekt + vattenplatser	33	44	29	40	10	10	13	8	11	8
Värdeobjekt + deso	38	50	46	57	29	29	77	69	27	37
50 transaktioner + hållplatser	328	1534	239	1463	202	958	135	152	1330	226
50 transaktioner + vattenytor	270	353	109	1380	112	277	175	183	1265	270
50 transaktioner + deso	1052	1753	784	1980	1159	642	1011	507	1447	691
50 transaktioner + 5 år tillbaka	29	256	55	1023	14	158	61	113	843	157

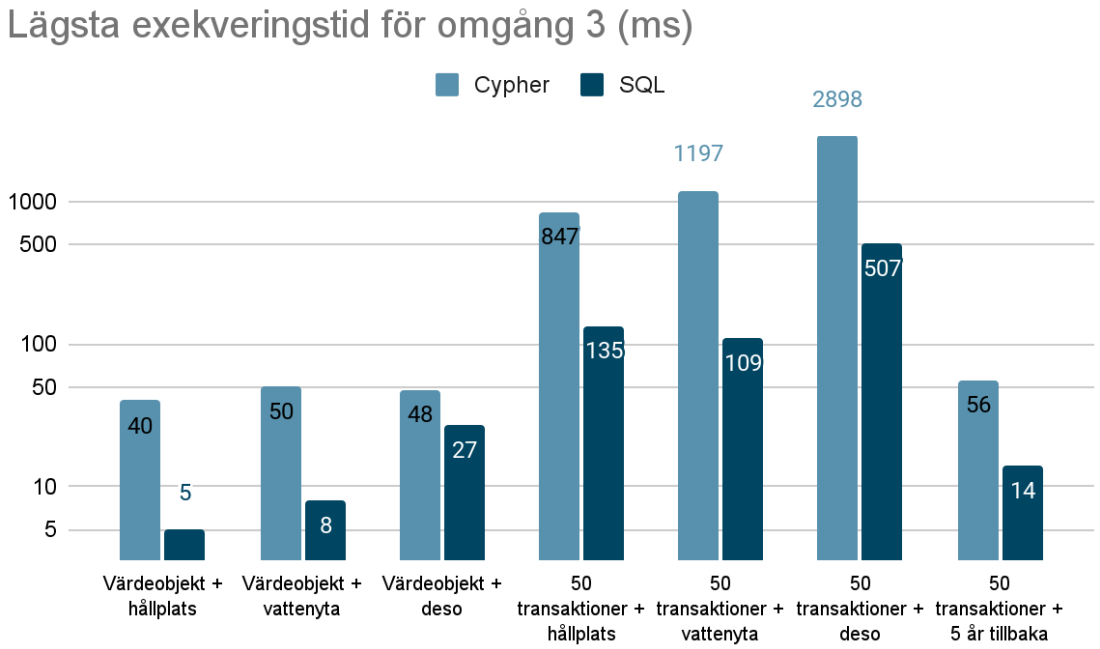
Tabell 5.12 - SQL: Exekveringstid för testerna utförda på tio fastigheter. Tidsenheten är i ms.

5.3.2 Diagram

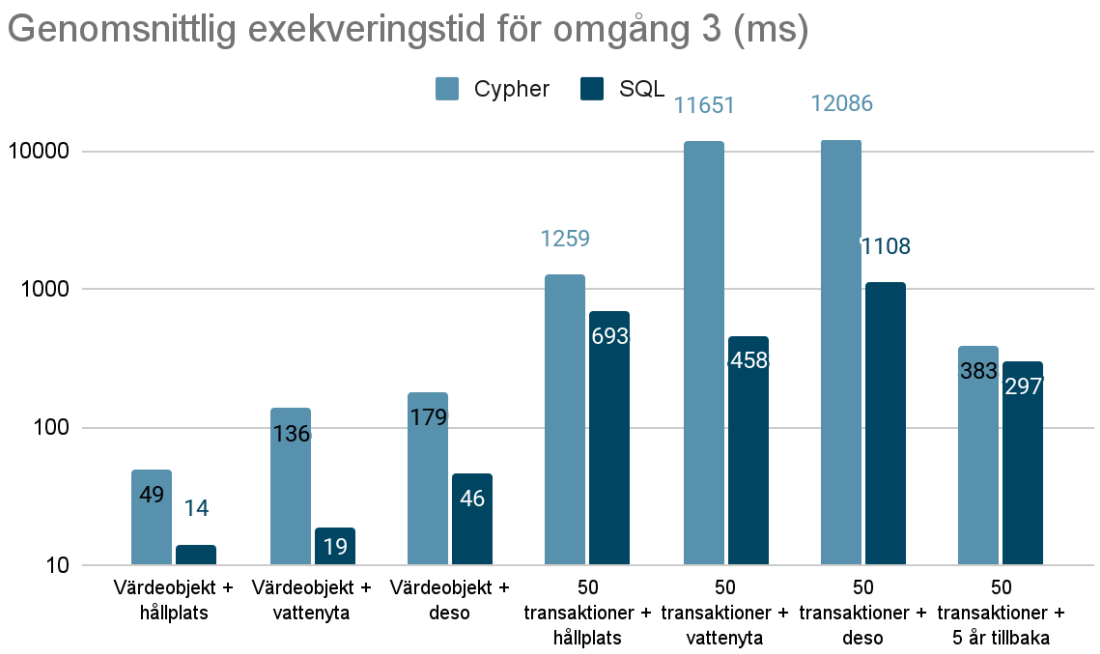
Figur 5.13, 5.14 och 5.15 är diagram skapade utifrån datan i tabell 5.11 och 5.12. De visar högsta, lägsta och genomsnittliga exekveringstiderna för de två databaserna.



Figur 5.13 - Högsta exekveringstid för testerna utförda på fastighet två till tio.



Figur 5.14 - Lägsta exekveringstid för testerna utförda på fastighet två till tio.



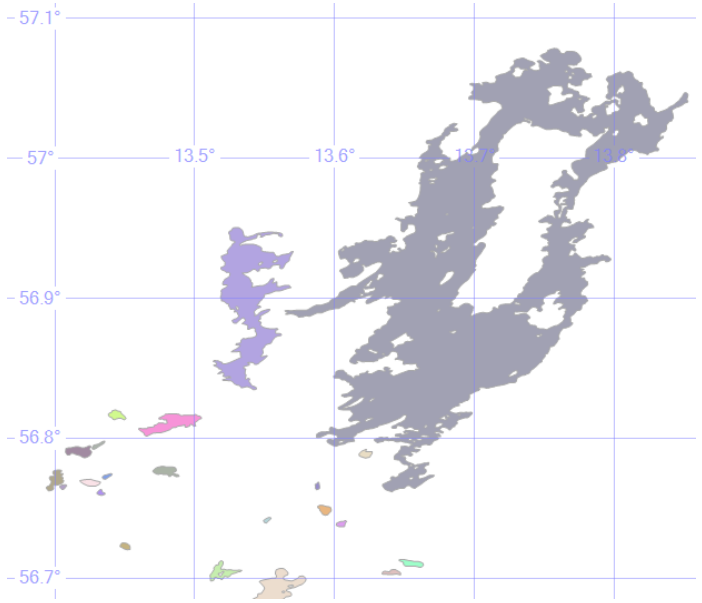
Figur 5.15 - Den genomsnittliga exekveringstid av testerna utförda på fastighet två till tio.

6 Diskussion

Här diskuteras de olika delarna av arbetet och de olika bidragande faktorerna till hur väl de genomfördes. För och nackdelar med de olika delar av arbetet och dess resultat diskuteras samt utvärderas arbetsprocessen.

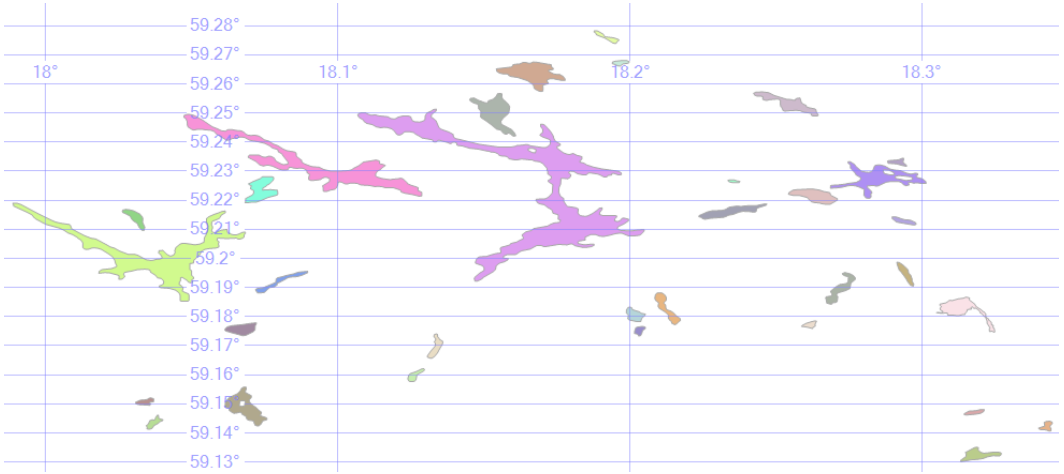
6.1 Utvärdering av resultat

De högst uppmätta exekveringstiderna för cypher var på 40420, 54369 och 54244 ms. Dessa tider tillkom av testet 50 transaktioner + vattenyta vilket matchar närmsta vattenytan till femtio fastigheter. Denna beräkning gjordes för fastighet 7 och i figur 6.1 nedan kan de 27 multipolygonerna vilket omringar fastigheten ses. Den största multipolygonen innehåller 4321 punkter och det är den multipolygonen i datamängden med flest punkter. Samma beräkning i SQL tog 158, 231 och 175 ms.



Figur 6.1 - Sjöar i omgivningen av fastighet 7.

För samma test gav fastighet 9 två av de högsta exekveringstiderna i SQL, 1487 och 1720 ms. I figur 6.2 nedan kan de multipolygoner som bidrog till tiden ses. Totalt finns 33 stycken vattenytor där de tre största innehåller 386, 244 och 209 punkter. För cypher gav samma beräkning 3954 och 4982 vilket är cirka dubbla tiden av SQL men långt ifrån den högsta tiden i cypher. Utifrån detta kan man dra slutsatsen att SQL hanterar beräkningar med komplexa multipolygoner bättre jämfört med cypher.



Figur 6.2 - Sjöar i omgivningen av fastighet 9.

Den genomsnittliga exekveringstiden visar att SQL presterar bättre för alla tester med ett undantag för testet 50 transaktioner + hållplats i testomgång två. Detta kan hänvisas till att beräkningen av de närmaste hållplatserna till de femtio närmaste transaktionerna till fastighet två gav den högsta tiden av alla SQL tester, 6814 ms. Anledningen till detta har inte kunnat besvaras då mängden data som hanteras inte sticker ut jämfört med alla andra fastigheter.

Det enda testet som inte utnyttjade biblioteket “Neo4j Spatial” var testet 50 transaktioner + 5 år tillbaka. Det kan noteras att den genomsnittliga exekveringstiden för detta testet gav den minsta skillnaden i exekveringstid av samtliga tester. Utifrån detta kan man ställa frågan huruvida stor inverkan användandet av biblioteket har på testerna. Om större stöd för spatiala datatyper och funktioner fanns inbyggt i Neo4j skulle eventuellt prestandan mer efterlikna den hos SQL. I ett system där endast punkter används för att representera geografiska objekt skulle Neo4j kunna ge en fördel. Detta skulle kunna ske genom att lagra punkter i en array i en nod för att representera en polygon.

6.2 Utvärdering av arbetsprocess

Arbetsprocessen gick i sin helhet bra och enligt plan. Det uppsatta schemat följdes relativt väl. Inga större avvikelser från det noterades. Arbetsprocessen och vad ämnades göras var tydligt genom nästintill hela arbetsprocessen. Anledningarna till varför arbetsprocessen gick för det mesta bra och enligt plan var bland annat en tydligt definierad planeringsrapport. Mycket arbete lades ner på denna vilket senare visade lönsamt. I denna beskrevs det i detalj vad som bör göras och i vilken ordning.

Oundvikligt blev det dock vissa problem med arbetet vilket krävde avvik från planeringsrapporten. Ett tydligt exempel på detta var att scenarierna definierade i planeringsrapporten under en stor del av arbetets gång ändrades och framkom i flertalet iterationer. Anledningen till detta grundade sig till viss del i missuppfattningar. Dessa scenarier och deras associerade queries visades för sent till handledaren på Värderingsdata.

Detta resulterade i att vissa brister och fel i dessa upptäcktes senare i arbetet än om de hade presenterats tidigare. En ytterligare bidragande faktor till problem kring scenarierna och deras queries berodde på bristande kunskap om databasen i sin helhet.

Den praktiska delen av arbetet gick, förutom tidigare nämnda komplikationer, enligt plan och stötte inte på några större problem. Det noterades dock efter drygt hälften av arbetsprocessen att det hade varit önskvärt att påbörja skrivandet av rapporten kring arbetet tidigare. Visserligen hade det gjorts men inte i den utsträckning vilket hade varit optimal.

6.3 Utvärdering av Systemkonstruktion

Systemkonstruktionen är grundpelaren för testerna och bör därmed evalueras för att få ökad förståelse kring hur testerna påverkats av den. De huvudsakliga för- och nackdelarna har framkommit av grafdatabasen. Detta då relationsdatabasen redan varit etablerad från arbetets start. Vidare är kunskapen begränsad kring grafdatabasen och det är därmed lättare att begå misstag kring dess utformning.

6.3.1 Systemkonstruktionens olika iterationer

Systemkonstruktionen har kontinuerligt ändrats under arbetets gång. Både relationsdatabasen och grafdatabasen har ändrats både i form av dess data samt dess konstruktion. Relationsdatabasen konstruktion bestod i början av arbetet enbart utav tabeller utan någon uppenbart koppling mellan dem. Därefter har det under arbetets gång kontinuerligt gjorts ändringar och tillägg i form av primärnycklar, främmande nycklar, borttagande av irrelevant data samt importering av ny data. Slutresultatet blev därmed en relationsdatabas bestående av sammankopplade tabeller.

Grafdatabasens data ändrades inte då den hade sitt ursprung från relationsdatabasen. Strukturen på grafdatabasen ändrades däremot betydelsefullt. Ursprungligen var tanken att använda relationer. Därmed bestod de första iterationerna av grafdatabasen av relationer mellan noderna. När väl relationer inte ansågs vara en lämplig lösning då de består av föraggregerad data mellan noder behövdes grafdatabasens struktur designas om. Därefter baserades den istället på enskilda noder av olika typer.

6.3.2 Styrkor och Svagheter

Den största svagheten med den slutliga systemkonstruktionen är att den ej använder sig av relationer för grafdatabasen. Detta resulterar i längre exekveringstider för queries i grafdatabasen och tar därmed bort dess största styrka i form av relationer. Denna förlängda exekveringstid blir betydligt noterbar vid beräkningar av avancerad spatialdata i form av polygoner eller multipolygoner.

Den fördel som däremot framkommer av avsaknandet av relationer är enklare och mindre tidskrävande underhåll av datan. Att vid varje tillägg av data behöva räkna om alla relationer är den del som minst motiverar användandet av dem. Detta hade begränsat när data skulle kunna läggas till i grafdatabasen för att undvika att störa tjänster som använder sig av grafdatabasen.

6.3.3 Påverkan på testerna

Den huvudsakligen påverkan denna systemkonstruktionen medförde till testerna var i form av avsaknandet av relationer. Då relationerna grundas i föraggregerad data, hade det inneburit ett betydande prestandalyft för grafdatabasen och hade sannolikt presterat bättre än vad relationsdatabasen gör. Detta då prestandan i nuläget är relativt lik, bortsett från de tyngre beräkningarna av spatialdata.

6.3.4 Potentiella förbättringsmöjligheter

Ett annat spatial bibliotek vilket ej användes eller utfördes tester med är "neo4j spatial algorithms". Detta är i vissa drag likt Neo4j Spatial men har jämförelsevis begränsad spatial funktionalitet. Det erbjuder "en ny implementation av algoritmer utan användandet av tredje parts bibliotek" [44]. Att pröva detta biblioteket och jämföra dess prestanda med det bibliotek som används nu, Neo4j Spatial, hade gett intressanta resultat. Huruvida Neo4j Spatial algorithms hade presterat bättre än Neo4j Spatial är oklart men hade varit värdefullt att undersöka. Detta kunde dock inte genomföras på grund av tidsbrist och att biblioteket ej kunde installeras då dokumentation var begränsad.

Något som insågs under senare delar av arbetet var att relationer som ett koncept potentiellt förkastades utan att ta i åtanke att de skulle kunna erbjuda förbättringar även i en begränsad implementering. Det sättet vi närmade oss relationer var att försöka skapa dem genom att utifrån varje fastighet beräkna det närmaste geografiska objektet. Istället för detta skulle man kunna endast skapa relationer mellan fastigheter och geografiska objekt vilket befinner sig inom en viss radie. Relationerna består sedan av det aktuella avståndet mellan fastigheten och det geografiska objektet vilket kan användas för att till exempel hitta den närmaste sjön till en fastighet genom att sortera på relationerna.

6.4 Fördelar och nackdelar

En stor fördel med Neo4j är att det finns många likheter till MSSQL. En person med tidigare erfarenhet med relationsdatabaser skulle haft en fördel av detta vid inlärning och användandet av Neo4j. Språket cypher är likt SQL då inspiration av SQL har tagits vid dess utvecklande. Cypher som språk är även mer visuellt och intuitivt tack vare dess syntax och linjära struktur. Under arbetets gång upplevdes även att testerna i cypher blev mer läsbara och kompakta jämfört med SQL.

Den största nackdelen med Neo4j är att den endast stödjer en spatial datatyp vilket är punkter och som resultat finns det endast funktioner till punkter. För att utöka det spatiala stödet för att kunna hantera datatyper och funktioner till polygoner och multipolygoner krävdes användandet av ett externt bibliotek. MSSQL har istället stöd för detta inbyggt.

6.5 Etiska och Ekologiska aspekter

De etiska aspekter detta arbete har är hanteringen av data. Då arbetet innefattar känsliga uppgifter kring fastigheter i Sverige innebär det ett fokus på integritet. Trots det faktumet att en del av uppgifterna är mörkade och därmed inte exakta bör de ej falla i händer på obehöriga.

Det har för detta arbetet ej påfunnits några uppenbara ekologiska aspekter. En parallell vilket skulle kunna dras är att arbetet har syftat att undersöka en förbättring av prestanda. Vid bättre prestanda görs beräkningar snabbare vilket resulterar i mindre strömåtgång vid bland annat nedkylning av servrar vilket är positivt ekologiskt.

6.6 Fortsatt arbete

Ett intressant område för fortsatt arbete är Neo4j native. Det är planerat att införa en ny index till endast punkter. Detta tyder på ett potentiellt intresse att utöka den spatiala funktionalitet Neo4j native erbjuder och kan därmed vara värt att följa nära under dess fortsatta utveckling.

En ytterligare punkt är kring utvecklandet av nya spatiala algoritmer och optimeringen av redan existerande spatiala algoritmer. Detta hade kunnat bidra till att korta ner query tider för polygoner och multipolygoner. Jämfört mellan Neo4j och MSSQL är det i nuläget dessa queries där de mest noterbara tidsskillnaderna noteras. Detta i kombination med implementationen av nya indexes för punkter i Neo4j native tar fram ett behov av nya tester av de olika databaserna.

7 Slutsats

Arbetet med de två databaserna SQL Server samt Neo4j har gett värdefull information kring deras prestanda och uppbyggnad. Neo4j har visat sig vara en lovande databas med mycket potential. Enkel syntax, lättförståeligt koncept för lagring av data samt ett fullständigt paket i form av Neo4j desktop.

Dess relativa nya framträdande på scenen för databaser i jämförelse med SQL innebär dock vissa begränsningar. Detta är tydligt i dess begränsade funktionalitet och hantering av spatialdata och dess behov av externt bibliotek av funktioner för att hantera denna datan. SQL har fördelen av ett rikare utbud spatial funktionalitet.

Testerna talade i fördel för SQL över Neo4j. I alla tester för de definierade scenarierna presterade SQL bättre. I de tester och scenarier vilka involverade större mängder data och komplex spatialdata var SQL överlägset Neo4j då det tog betydligt längre för Neo4j att färdigställa sina queries. Detta kan till stor del härledas till biblioteket “Neo4j Spatial” vilket utnyttjades till alla tester förutom ett. Det testet vars funktionalitet inte använde detta bibliotek gav mest likt resultat med SQL.

Referenser

[1] Värderingsdata, “Om,” 2022. [Online]. Tillgänglig: https://www.varderingsdata.se/om/ (hämtad: 2022-01-19).

[2] European AVM Alliance, “Key Requirements,” [Online]. Tillgänglig: https://www.europeanavmalliance.org/eaa-avm-label-accreditation/key-requirements.html (hämtad: 2022-01-24).

[3] Elastic, “Elasticsearch Intro,”[Online]. Tillgänglig: https://www.elastic.co/guide/en/elasticsearch/reference/current/elasticsearch-intro.html (hämtad: 2022-01-28).

[4] Hector. Garcia-Molina, *Database systems: the complete book*, Storbritannien: Pearson Education, 2013. [Online]. Tillgänglig: https://www.vlebooks.com/Product/Index/437580?page=0, Hämtad: 2022-05-12.

[5] Microsoft, “Databases,” 2022. [Online]. Tillgänglig: https://docs.microsoft.com/en-us/sql/relational-databases/databases/databases?view=sql-server-ver15 (hämtad: 2022-05-12).

[6] Microsoft, “Download SQL Server Management Studio (SSMS),” 2022. [Online]. Tillgänglig:https://docs.microsoft.com/en-us/sql/relational-databases/databases/databases?view=sql-server-ver15 (hämtad: 2022-05-12).

[7] Neo4j, “Graph Database,” [Online]. Tillgänglig: https://neo4j.com/developer/graph-database/ (hämtad: 2022-02-16).

[8] Neo4j, “Company,” [Online]. Tillgänglig: https://neo4j.com/company/ (hämtad: 2022-05-25).

[9] Neo4j, “Licensing,”[Online]. Tillgänglig: https://neo4j.com/licensing/(hämtad: 2022-05-11).

[10] Neo4j, “Product,”[Online]. Tillgänglig: https://neo4j.com/product/ (hämtad: 2022-05-04).

[11] Neo4j, “Data Import,”[Online]. Tillgänglig: https://neo4j.com/developer/data-import/ (hämtad: 2022-03-15).

[12] Nadime Francis et al., "Cypher: An Evolving Query Language for Property Graphs," in SIGMOD'18 Proceedings of the 2018 International Conference on Management of Data, Houston, United States, Jun 2018. pp.1433. [Online]. Tillgänglig: https://dl.acm.org/doi/10.1145/3183713.3190657
(hämtad: 2022-05-11).

[13] Microsoft, "Indexes," 2022. [Online]. Tillgänglig: https://docs.microsoft.com/en-us/sql/relational-databases/indexes/indexes?view=sql-server-ver16 (hämtad: 2022-06-04).

[14] Microsoft, "SQL Server and Azure SQL index architecture and design guide," 2022. [Online]. Tillgänglig: https://docs.microsoft.com/en-us/sql/relational-databases/sql-server-index-design-guide?view=sql-server-ver16 (hämtad: 2022-06-04).

[15] Microsoft, "Spatial indexes overview," 2022. [Online]. Tillgänglig: https://docs.microsoft.com/en-us/sql/relational-databases/spatial/spatial-indexes-overview?view=sql-server-ver15
(hämtad: 2022-06-04).

[16] Microsoft, "Clustered and nonclustered indexes described," 2022. [Online]. Tillgänglig: https://docs.microsoft.com/en-us/sql/relational-databases/indexes/clustered-and-nonclustered-indexes-described?view=sql-server-ver16
(hämtad: 2022-06-05).

[17] Neo4j, "Index configuration,"[Online]. Tillgänglig: https://neo4j.com/docs/operations-manual/4.4/performance/index-configuration/
(hämtad: 2022-05-31).

[18] Neo4j, "Indexes for search performance," [Online]. Tillgänglig: https://neo4j.com/docs/cypher-manual/current/indexes-for-search-performance/
(hämtad: 2022-04-24).

[19] Kensuke Kouno, Joo Kooi Tan, Seiji Ishikawa, "High-speed Data Retrieval in an Eigenspace Employing a B-tree Structure," in 2006 SICE-ICASE International Joint Conference, Busan, Korea, 2006, pp. 18-21. [Online]. Tillgänglig: https://ieeexplore.ieee.org/document/4108106 Hämtad: 2022-04-28.

[20] Wikipedia, "B-tree," 2010 [Online]. Tillgänglig: https://sv.wikipedia.org/wiki/Fil:B-tree.svg (hämtad: 2022-05-24).

[21] Wikipedia, "R-tree," 2010 [Online]. Tillgänglig: https://commons.wikimedia.org/wiki/File:R-tree.svg (hämtad: 2022-05-24).

[22] Neo4j, “Spatial values,” [Online]. Tillgänglig: https://neo4j.com/docs/cypher-manual/current/syntax/spatial/ (hämtad: 2022-06-05).

[23] Baeldung, “The difference between B-trees and B+trees,” 2020. [Online]. Tillgänglig: https://www.baeldung.com/cs/b-trees-vs-btrees (hämtad: 2022-06-05)

[24] Gisgeography, “Vector vs Raster: What’s the Difference Between GIS Spatial Data Types?,” 2022. [Online]. Tillgänglig: https://gisgeography.com/spatial-data-types-vector-raster/ (hämtad : 2022-06-01).

[25] Microsoft, “Well Known Text Module,” 2022. [Online]. Tillgänglig: https://docs.microsoft.com/en-us/bingmaps/v8-web-control/modules/well-known-text-module (hämtad: 2022-05-24).

[26] Neo4j, “Spatial Functions,” [Online]. Tillgänglig: https://neo4j.com/docs/cypher-manual/current/functions/spatial/ (hämtad: 2022-04-28).

[27] Neo4j Spatial, “Neo4j Spatial,” [Online]. Tillgänglig: https://neo4j-contrib.github.io/spatial/ (hämtad: 2022-04-28).

[28] Neo4j Spatial, “Neo4j Spatial v0.24-neo4j-3.1.4,” 2017. [Online]. Tillgänglig: https://neo4j-contrib.github.io/spatial/0.24-neo4j-3.1/index.html (hämtad: 2022-04-28).

[29] Microsoft, “Spatial Data,” 2022. [Online]. Tillgänglig: https://docs.microsoft.com/en-us/sql/relational-databases/spatial/spatial-data-sql-server?view=sql-server-ver15 (hämtad: 2022-05-11).

[30] Microsoft, “Point,” 2021. [Online]. Tillgänglig: https://docs.microsoft.com/en-us/sql/relational-databases/spatial/point?view=sql-server-ver15 (hämtad: 2022-05-12).

[31] Microsoft, “Polygon,” 2020. [Online]. Tillgänglig: https://docs.microsoft.com/en-us/sql/relational-databases/spatial/polygon?view=sql-server-ver15 (hämtad: 2022-05-12).

[32] Microsoft, “MultiPolygon,” 2020. [Online]. Tillgänglig: https://docs.microsoft.com/en-us/sql/relational-databases/spatial/multipolygon?view=sql-server-ver15 (hämtad: 2022-05-12).

[33] Microsoft, “Create, construct, and query geography instances,” 2021. [Online]. Tillgänglig:https://docs.microsoft.com/en-us/sql/relational-databases/spatial/create-construct-and-query-geography-instances?view=sql-server-ver15 (hämtad: 2022-05-12).

[34] J. Lach, "SBFS: Steganography based file system," in Proceedings of the 7th International Conference on Data Science, Technology and Applications (DATA 2018), Porto, Portugal, 2018, pp. 373-380. [Online]. Tillgänglig: https://www.scitepress.org/papers/2018/69102/69102.pdf, (hämtad: 2022-01-28).

[35] Trafiklab, “Static Data,” 2021. [Online]. Tillgänglig: https://www.trafiklab.se/api/trafiklab-apis/gtfs-sverige-2/static-data/ (hämtad: 2022-03-02).

[36] Dataportal, “Vattenytor (SVAR2016),” 2020. [Online]. Tillgänglig: https://www.dataportal.se/sv/datasets/78_3203/vattenytor-svar2016#ref=?p=1&q=vattenytor&s=2&t=20&f=&rt=dataset%24esterms_IndependentDataService%24esterms_ServedByDataService&c=false (hämtad: 2022-03-02).

[37] QGIS, “Site,” [Online]. Tillgänglig: https://qgis.org/en/site (hämtad: 2022-05-29).

[38] Microsoft, “Import Flat File to SQL Wizard,” 2020. [Online]. Tillgänglig: https://docs.microsoft.com/en-us/sql/relational-databases/import-export/import-flat-file-wizard?view=sql-server-ver16 (hämtad: 2022-05-26)

[39] Neo4j, “Importing CSV Data into Neo4j,” [Online]. Tillgänglig: https://neo4j.com/developer/guide-import-csv/ (hämtad: 2022-05-19).

[40] Neo4j, “Subquery call in transactions,”. [Online]. Tillgänglig: https://neo4j.com/docs/cypher-manual/current/clauses/call-subquery/#subquery-call-in-transactions (hämtad: 2022-04-20).

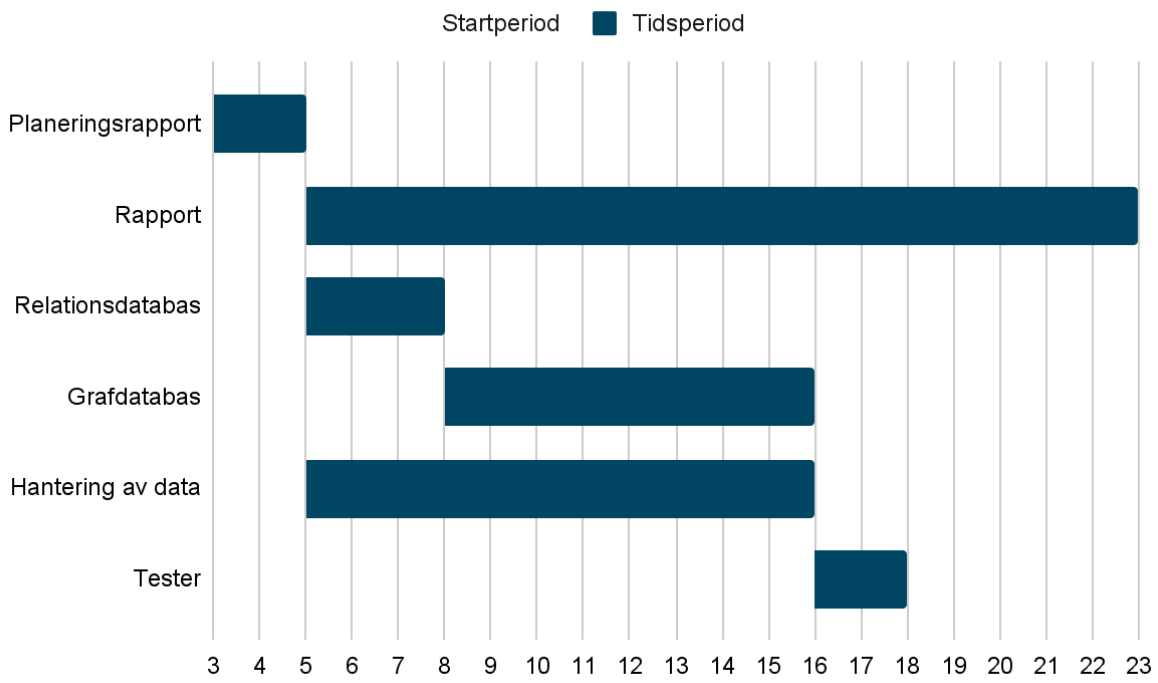
[41] Microsoft, “Query spatial data for nearest neighbor,” 2020. [Online]. Tillgänglig: https://docs.microsoft.com/en-us/sql/relational-databases/spatial/query-spatial-data-for-nearest-neighbor?view=sql-server-ver16 (hämtad: 2022-03-21)

[42] Neo4j Spatial, Version: Spatial 0.28.1-neo4j-4.4.3, [Software], Helsingborg, Sweden: Craig Taverner, 2022. (hämtad: 2022-04-24).

[43] Github, “neo4j-contrib spatial,” 2022. [Online]. Tillgänglig: https://github.com/neo4j-contrib/spatial (hämtad: 2022-04-24).

[44] Github, “Neo4j Spatial Algorithms,” 2022. [Online]. Tillgänglig: https://github.com/neo4j-contrib/spatial-algorithms (hämtad: 2022-06-01).

A. Första appendix: Tidsplan



B. Andra appendix: Script för uppsättning av Neo4j

```
CREATE CONSTRAINT UniqueFastighet FOR (n:GarFastighet) REQUIRE (n.IdTaxEnhet,
n.NrVarde) IS NODE KEY

CREATE CONSTRAINT UniqueSmahusByggnad FOR (n:GarSmahusByggnad) REQUIRE
(n.IdTaxEnhet, n.NrVarde) IS NODE KEY

:auto LOAD CSV WITH HEADERS FROM 'file:///GarFastighetSmahusByggnad.csv' as row
CALL {
WITH row
CREATE (s:GarSmahusByggnad {IdTaxEnhet: row.IdTaxEnhet, NrVarde:
toInteger(row.NrVarde)})
SET s.VardeYta = toInteger(row.Värdeyta)
CREATE (f:GarFastighet {IdTaxEnhet: row.FIdTaxEnhet, NrVarde: toInteger(row.FNrVarde)})
SET f.Areal = toInteger(row.Areal), f.Kom = toInteger(row.Kom), f.Lan = toInteger(row.Lan),
f.FNKL = toInteger(row.FNKL)
} IN TRANSACTIONS OF 100 ROWS

CREATE INDEX FastighetIndex FOR (n:GarFastighet) ON (n.FNKL)

CREATE CONSTRAINT UniqueFastighetsKoordinater FOR (n:FastighetsKoordinater) REQUIRE
n.FNKL IS UNIQUE

:auto LOAD CSV WITH HEADERS FROM 'file:///FastighetsKoordinater.csv' AS row
CALL {
WITH row
CREATE(f:FastighetsKoordinater{FNKL: toInteger(row.FNKL)})
SET f.Lng = toFloat(row.Lng), f.Lat = toFloat(row.lat),
f.Geo = point({longitude: toFloat(row.Lng), latitude: toFloat(row.lat)})
} IN TRANSACTIONS OF 100 ROWS

CREATE INDEX FastighetsKoordinaterGeo FOR (n:FastighetsKoordinater) ON (n.Geo)

CREATE CONSTRAINT UniqueDeso FOR (n:Deso) REQUIRE n.DesoName IS UNIQUE

:auto LOAD CSV WITH HEADERS FROM 'file:///Deso.csv' AS row
CALL{
   WITH row
   CREATE(n:Deso{DesoName: row.DesoName, Version: row.Version, Kommun: row.Kommun,
Lan: row.Lan, Geo: row.Geo})
} IN TRANSACTIONS OF 100 ROWS

call spatial.addWKTLayer("DesoLayer", "Geo")

MATCH (n:Deso)
CALL spatial.addNode("DesoLayer", n) YIELD node
RETURN node

CREATE CONSTRAINT UniqueDesoStatistics FOR (n:DesoStatistics) REQUIRE (n.DesoName,
n.StatisticTime) IS NODE KEY
```

```
:auto LOAD CSV WITH HEADERS FROM 'file:///DesoStatistics.csv' AS row
CALL {
WITH row
CREATE (n:DesoStatistics)
SET n.DesoName = row.DesoName, n.StatisticTime = toInteger(row.StatisticTime),
n.StatisticDescription = row.StatisticDescription, n.StatisticId = row.StatisticId, n.Value =
toFloat(row.Value), n.TimeUnitDescription = row.TimeUnitDescription
}IN TRANSACTIONS OF 100 ROWS

CREATE INDEX DesoStatisticsIndex FOR (n:DesoStatistics) ON (n.DesoName)

CREATE CONSTRAINT UniqueHallplats FOR (n:Hallplatser) REQUIRE n.stop_id IS UNIQUE

:auto LOAD CSV WITH HEADERS FROM 'file:///Hallplatser.csv' AS row
CALL{
WITH row
CREATE(h:Hallplatser{stop_name: row.stop_name, stop_id: toInteger(row.stop_id), stop_lng:
toFloat(row.stop_lon), stop_lat: toFloat(row.stop_lat),
location: point({longitude: toFloat(row.stop_lon), latitude: toFloat(row.stop_lat)})
, Geo: row.Geo})
} IN TRANSACTIONS OF 1000 ROWS

CALL spatial.addWKTLayer("HallplatsLayer", "Geo")

MATCH (n:Hallplatser)
CALL spatial.addNode("HallplatsLayer", n) YIELD node
RETURN node

CREATE CONSTRAINT UniqueVatten FOR (n:VattenYtor) REQUIRE n.ObjectId IS UNIQUE

:auto LOAD CSV WITH HEADERS FROM 'file:///VattenYtor.csv' AS row
CALL{ WITH row
CREATE (n:VattenYtor {ObjectId: toInteger(row.OBJECTID), Name: row.NAME, Geo: row.Geo})
}IN TRANSACTIONS OF 100 ROWS

call spatial.addWKTLayer("VattenLayer", "Geo")

MATCH (v:VattenYtor)
CALL spatial.addNode("VattenLayer", v) YIELD node
RETURN node

CREATE CONSTRAINT UniqueTransaktioner FOR(n:Transaktioner) REQUIRE n.TransId IS
UNIQUE

CREATE INDEX TransaktionsIndex FOR (n:Transaktioner) ON (n.FNKL)

CREATE INDEX TransaktionsGeo FOR (n:Transaktioner) ON (n.location)

:auto LOAD CSV WITH HEADERS FROM 'file:///Transaktioner.csv' AS row
CALL{
WITH row
```

```
CREATE (t:Transaktioner)
SET t.TransId = toInteger(row.Transid),
t.location = point({longitude: toFloat(row.LNG), latitude: toFloat(row.LAT)}), t.Geo = row.Geo,
t.latitude = toFloat(row.LAT), t.longitude = toFloat(row.LNG), t.TypKod = toInteger(row.TypKod),
t.KOMNR = toInteger(row.KOMNR), t.Date = apoc.date.parse(row.DBDAT, 'ms', 'yyyy-MM-dd'),
t.Pris = toInteger(row.Pris), t.FNKL = toInteger(row.FNKL)
} IN TRANSACTIONS OF 1000 ROWS
```

C. Tredje appendix: Testerna

Värderingsobjekt + hållplats:

```
SET STATISTICS TIME ON;
DECLARE @FNK bigint;
SET @FNK = (SELECT FNKL FROM GarFastighet WHERE (GarFastighet.IdTaxEnhet =
'105374-9' AND GarFastighet.NrVarde = 1))
SELECT
        fk.FNKL
        ,H.stop_name AS Hallplats
        ,H.Distance AS HallplatsAvstånd
        ,H.hallplatsGeo AS HallplatsGeo
FROM FastighetsKoordinater AS fk
CROSS APPLY (
        SELECT TOP(1)
                Hallplatser.stop_name
        ,Hallplatser.Geo AS hallplatsGeo
        ,fk.Geo.STDistance(Hallplatser.Geo) AS Distance
        FROM Hallplatser
        WHERE fk.Geo.STDistance(Hallplatser.Geo) < 10000
        ORDER BY(Distance)) AS H
WHERE fk.FNKL = @FNK
SET STATISTICS TIME OFF;
```

Värderingsobjekt + vattenyta:

```
SET STATISTICS TIME ON;
DECLARE @FNK bigint;
SET @FNK = (SELECT FNKL FROM GarFastighet WHERE (GarFastighet.IdTaxEnhet =
'105374-9' AND GarFastighet.NrVarde = 1))

SELECT
        fk.FNKL
        ,V.NAME AS Vatten
        ,V.Distance AS VattenAvstånd
        ,V.VattenGeo AS VattenGeo
FROM FastighetsKoordinater AS fk
CROSS APPLY (
        SELECT TOP(1) VattenYtor.NAME
                ,VattenYtor.Geo AS VattenGeo
                ,fk.Geo.STDistance(VattenYtor.Geo) AS Distance
        FROM VattenYtor
        WHERE fk.Geo.STDistance(VattenYtor.Geo) < 10000
        ORDER BY(Distance)
        ) AS V
WHERE fk.FNKL = @FNK
SET STATISTICS TIME OFF;
```

Värderingsobjekt Deso:

```
SET STATISTICS TIME ON;
DECLARE @FNK bigint;
SET @FNK = (SELECT FNKL FROM GarFastighet WHERE (GarFastighet.IdTaxEnhet =
'105374-9' AND GarFastighet.NrVarde = 1))

SELECT
        FastighetsKoordinater.FNKL
        ,Deso.DesoName
        ,Deso.Geo AS DesoGeo
        ,DesoStatistics.TimeUnitDescription
        ,DesoStatistics.StatisticTime
        ,DesoStatistics.StatisticDescription
        ,DesoStatistics.Value
FROM FastighetsKoordinater
JOIN Deso ON FastighetsKoordinater.FNKL = @FNK AND
FastighetsKoordinater.Geo.STIntersects(Deso.Geo) = 1
JOIN DesoStatistics ON Deso.DesoName = DesoStatistics.DesoName
SET STATISTICS TIME OFF;
```

50 transaktioner + hållplats:

```
SET STATISTICS TIME ON;
DECLARE @FNK bigint;
SET @FNK = (SELECT FNKL FROM GarFastighet WHERE (GarFastighet.IdTaxEnhet =
'105374-9' AND GarFastighet.NrVarde = 1))
;WITH Test (Transid, FNKL, Geo) AS(
SELECT TOP(50)
        Transaktioner.Transid,
        Transaktioner.FNKL,
        Transaktioner.Geo
FROM Transaktioner
JOIN
        FastighetsKoordinater ON FastighetsKoordinater.FNKL = @FNK AND
Transaktioner.FNKL != @FNK
        AND Transaktioner.Geo.STDistance(FastighetsKoordinater.Geo) < 20000
ORDER BY(Transaktioner.Geo.STDistance(FastighetsKoordinater.Geo)))
SELECT
        Test.Transid,
        Test.FNKL,
        H.stop_name,
        H.HDistance
FROM Test
CROSS APPLY(
   SELECT TOP(1)
         test.FNKL
         ,test.Geo.STDistance(Hallplatser.Geo) HDistance
```

```
        ,Hallplatser.stop_name
   FROM  Hallplatser
        WHERE test.Geo.STDistance(Hallplatser.Geo)<10000
        ORDER BY(HDistance)) AS H
SET STATISTICS TIME OFF;
```

50 transaktioner + vatten:

```
SET STATISTICS TIME ON;
DECLARE @FNK bigint;
SET @FNK = (SELECT FNKL FROM GarFastighet WHERE (GarFastighet.IdTaxEnhet =
'105374-9' AND GarFastighet.NrVarde = 1))
;WITH Test (Transid, FNKL, Geo)AS(
SELECT TOP(50)
        Transaktioner.Transid,
        Transaktioner.FNKL,
        Transaktioner.Geo
FROM Transaktioner
JOIN
        FastighetsKoordinater ON FastighetsKoordinater.FNKL = @FNK AND
Transaktioner.FNKL != @FNK
        AND Transaktioner.Geo.STDistance(FastighetsKoordinater.Geo) < 20000
ORDER BY(Transaktioner.Geo.STDistance(FastighetsKoordinater.Geo)))
SELECT
        Test.Transid,
        Test.FNKL,
        V.NAME,
        V.VDistance
FROM Test
CROSS APPLY(
   SELECT TOP(1)
         test.FNKL
         ,test.Geo.STDistance(VattenYtor.Geo) VDistance
         ,VattenYtor.NAME
   FROM  VattenYtor
        WHERE test.Geo.STDistance(VattenYtor.Geo)<10000
        ORDER BY(VDistance)) AS V
SET STATISTICS TIME OFF;
```

50 transaktioner + Deso:

```
SET STATISTICS TIME ON;
DECLARE @FNK bigint;
SET @FNK = (SELECT FNKL FROM GarFastighet WHERE (GarFastighet.IdTaxEnhet =
'105374-9' AND GarFastighet.NrVarde = 1))

;WITH Test (Transid, FNKL, Geo)
AS
(
```

```
SELECT TOP(50)
        Transaktioner.Transid,
        Transaktioner.FNKL,
        Transaktioner.Geo
FROM Transaktioner
JOIN
        FastighetsKoordinater
        ON
        FastighetsKoordinater.FNKL = @FNK
        AND
        Transaktioner.FNKL != @FNK
   AND
   Transaktioner.Geo.STDistance(FastighetsKoordinater.Geo) < 20000
ORDER BY(Transaktioner.Geo.STDistance(FastighetsKoordinater.Geo))
)
SELECT
        Test.Transid,
        Test.FNKL,
        Deso.Geo,
        DesoStatistics.StatisticId,
        DesoStatistics.Value,
        DesoStatistics.StatisticTime
FROM Test
JOIN Deso ON Test.Geo.STIntersects(Deso.Geo) = 1
JOIN DesoStatistics ON Deso.DesoName = DesoStatistics.DesoName
SET STATISTICS TIME OFF;
```

50 transaktioner + 5 år tillbaka:

```
SET STATISTICS TIME ON;
DECLARE @FNK bigint;
SET @FNK = (SELECT FNKL FROM GarFastighet WHERE (GarFastighet.IdTaxEnhet =
'105374-9' AND GarFastighet.NrVarde = 1))

SELECT TOP(50)
        Transaktioner.Transid,
        Transaktioner.FNKL,
        Transaktioner.Geo,
        Transaktioner.Pris,
        Transaktioner.DBDAT
FROM Transaktioner
JOIN
        FastighetsKoordinater
        ON
        FastighetsKoordinater.FNKL = @FNK
        AND
        Transaktioner.FNKL != @FNK
   AND
   Transaktioner.Geo.STDistance(FastighetsKoordinater.Geo) < 20000
```

```
WHERE Transaktioner.DBDAT >= '2017-01-01 00:00:00'
ORDER BY(Transaktioner.Geo.STDistance(FastighetsKoordinater.Geo))
SET STATISTICS TIME OFF;
```

Värderingsobjekt + hållplats:

```
MATCH (n:GarSmahusByggnad {IdTaxEnhet: '289780-1', NrVarde: 1})
WITH n
MATCH (f:GarFastighet {IdTaxEnhet: n.IdTaxEnhet, NrVarde: n.NrVarde})
WITH f
MATCH (fk:FastighetsKoordinater {FNKL: f.FNKL})
WITH fk
CALL spatial.withinDistance("HallplatsLayer", fk.Geo, 10.0) YIELD node, distance
RETURN node AS HallplatsNode, distance AS HallplatsDistance LIMIT 1
```

Värderingsobjekt + vattenyta:

```
MATCH (n:GarSmahusByggnad {IdTaxEnhet: '289780-1', NrVarde: 1})
WITH n
MATCH (f:GarFastighet {IdTaxEnhet: n.IdTaxEnhet, NrVarde: n.NrVarde})
WITH f
MATCH (fk:FastighetsKoordinater {FNKL: f.FNKL})
WITH fk
CALL spatial.withinDistance("VattenLayer", fk.Geo, 10.0) YIELD node, distance
RETURN node AS VatteNnode, distance AS VattenDistance LIMIT 1
```

Värderingsobjekt + Deso:

```
MATCH (n:GarSmahusByggnad {IdTaxEnhet: '289780-1', NrVarde: 1})
WITH n
MATCH (f:GarFastighet {IdTaxEnhet: n.IdTaxEnhet, NrVarde: n.NrVarde})
WITH f
MATCH (fk:FastighetsKoordinater {FNKL: f.FNKL})
WITH fk
CALL spatial.intersects('DesoLayer', fk.Geo) YIELD node
WITH node
MATCH (stat:DesoStatistics)
WHERE node.DesoName = stat.DesoName
RETURN stat
```

50 transaktioner + hållplats:

```
MATCH (n:GarSmahusByggnad {IdTaxEnhet: '289780-1', NrVarde: 1})
WITH n
MATCH (f:GarFastighet {IdTaxEnhet: n.IdTaxEnhet, NrVarde: n.NrVarde})
WITH f
MATCH (fk:FastighetsKoordinater {FNKL: f.FNKL})
CALL{
   WITH fk
   MATCH (t:Transaktioner)
   WHERE point.withinBBox(t.location,
```

```
    point({longitude: (fk.Lng - 0.2), latitude: (fk.Lat - 0.2)}),
    point({longitude: (fk.Lng + 0.2), latitude: (fk.Lat + 0.2)})) AND fk.FNKL <> t.FNKL
    RETURN t, point.distance(t.location, fk.Geo) AS Dist ORDER by Dist ASC LIMIT 50
}
WITH t CALL spatial.withinDistance("HallplatsLayer", t.location, 10.0) YIELD node, distance
RETURN t.TransId, head(collect([t.TransId, node.stop_name, distance])) AS ClosestHallplats
```

50 transaktioner + vattenyta:

```
MATCH (n:GarSmahusByggnad {IdTaxEnhet: '289780-1', NrVarde: 1})
WITH n
MATCH (f:GarFastighet {IdTaxEnhet: n.IdTaxEnhet, NrVarde: n.NrVarde})
WITH f
MATCH (fk:FastighetsKoordinater {FNKL: f.FNKL})
CALL{
    WITH fk
    MATCH (t:Transaktioner)
    WHERE point.withinBBox(t.location,
    point({longitude: (fk.Lng - 0.2), latitude: (fk.Lat - 0.2)}),
    point({longitude: (fk.Lng + 0.2), latitude: (fk.Lat + 0.2)})) AND fk.FNKL <> t.FNKL
    RETURN t, point.distance(t.location, fk.Geo) AS Dist ORDER by Dist ASC LIMIT 50
    }
WITH t CALL spatial.withinDistance("VattenLayer", t.location, 10.0) YIELD node, distance
RETURN t.TransId, head(collect([t.TransId, node.Name, distance])) AS ClosestVatten
```

50 transaktioner + Deso:

```
MATCH (n:GarSmahusByggnad {IdTaxEnhet: '289780-1', NrVarde: 1})
WITH n
MATCH (f:GarFastighet {IdTaxEnhet: n.IdTaxEnhet, NrVarde: n.NrVarde})
WITH f
MATCH (fk:FastighetsKoordinater {FNKL: f.FNKL})
CALL{
    WITH fk
    MATCH (t:Transaktioner)
    WHERE point.withinBBox(t.location,
    point({longitude: (fk.Lng - 0.2), latitude: (fk.Lat - 0.2)}),
    point({longitude: (fk.Lng + 0.2), latitude: (fk.Lat + 0.2)})) AND fk.FNKL <> t.FNKL
    RETURN t, point.distance(t.location, fk.Geo) AS Dist ORDER by Dist ASC LIMIT 50
    }
CALL {
WITH t CALL spatial.intersects("DesoLayer", t.location) YIELD node
RETURN t.TransId AS DesoTransId, node.DesoName as desoname
}
MATCH (stat:DesoStatistics)
WHERE stat.DesoName = desoname
RETURN DesoTransId, stat
```

50 transaktioner + 5 år tillbaka:

```
MATCH (n:GarSmahusByggnad {IdTaxEnhet: '289780-1', NrVarde: 1})
WITH n
MATCH (f:GarFastighet {IdTaxEnhet: n.IdTaxEnhet, NrVarde: n.NrVarde})
WITH f
MATCH (fk:FastighetsKoordinater {FNKL: f.FNKL})
CALL{
WITH fk
MATCH (t:Transaktioner)
WHERE point.withinBBox(t.location,
point({longitude: (fk.Lng - 0.2), latitude: (fk.Lat - 0.2)}),
point({longitude: (fk.Lng + 0.2), latitude: (fk.Lat + 0.2)})) AND
fk.FNKL <> t.FNKL AND
t.Date > apoc.date.parse("2017-01-01", "ms", "yyyy-MM-dd")
RETURN t, point.distance(t.location, fk.Geo) AS Dist ORDER by Dist ASC LIMIT 50
}
RETURN t
```

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2022

www.chalmers.se

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY