Optimering av datastrukturer i SQL Server för loggningssystem

Mathias Lithén

Examensarbete för ingenjör (YH)-examen

Utbildningsprogrammet för informationsteknik

Vasa 2018

EXAMENSARBETE

Författare:	Mathias Lithén
Utbildningsprogram och ort:	Informationsteknik, Vasa
Handledare:	Kaj Wikman

Titel: Optimering av datastrukturer i SQL Server för loggningssystem

Datum 28.5.2018 Sidantal 21 Bilagor 7

Abstrakt

Syftet med detta examensarbete var att för ett företag undersöka huruvida man kan förbättra svarstider vid systemkörningar genom partitionering i Microsoft SQL Server Standard-edition. Företaget producerar ett loggsystem för övervakning av persondata inom offentlig förvaltning. Avsikten var att undersöka hur man kan optimera svarstider i systemet då datamängden rör sig kring 100 till 1000 miljoner händelser. Med denna datamängd kan sökningar i databasen vara långsamma. Detta kan motarbetas genom indexering, men många index i databasen kan producera långsamma söktider då data laddas med minst 60 000 händelser per dygn.

Företaget vill undvika dyrare licenskostnader för Enterprise-edition av Microsoft SQL Server, som har en inbyggd optimering för stora datamängder genom partitionering. Därför ville man undersöka om man kan nå bättre responstider genom att omstrukturera tabellerna i databasen. Den optimering som undersöktes i detta arbete var att dela data tidsmässigt i olika tabeller samt ändra källkod i systemet för att uppnå samma systemfunktioner som tidigare.

Mätningarna gjordes genom att mäta svarstiden för vissa systemsökningar för en av företagets kunder. Systemmiljön består av Windows Server samt Microsoft SQL Server, utvecklingsmiljön baserar sig på Python. Hårdvaran varierar bland företagets kunder och därför utfördes denna undersökning på samma hårdvara för en viss kunds räkning.

Språk: svenska Nyckelord: optimering, Microsoft SQL Server, loggsystem, Python

OPINNÄYTETYÖ

Tekijä:	Mathias Lithen
Koulutusohjelma ja paikkakunta:	Tietotekniikka, Vaasa
Ohjaaja:	Kaj Wikman

Nimike: Datarakenteiden optimointi SQL Serverissä lokienseurantajärjestelmälle

Päivämäärä 28.5.2018 Sivumäärä 21 Liitteet 7

Tiivistelmä

Opinnäytetyön tarkoitus oli tutkia kuinka voidaan parantaa vasteaikoja käyttämällä osiointia Microsoft SQL Server Standard Edition-ohjelmalla. Neotide kehittää lokienseurantajärjestelmää asiakkaille julkishallinnossa. Työssä haluttiin tutkia kuinka vasteajat voidaan optimoida järjestelmässä, joka voi sisältää 100 – 1000 miljoonaa tapahtumaa. Tällä datamäärällä tietokantahaut voivat olla hitaita. Tämä voidaan estää indeksoinnilla, mutta liika indeksointi voi johtaa hitaampiin vasteaikoihin, kun dataa ladataan vähintään 60 000 tapahtumaa vuorokaudessa.

Neotide pyrkii välttämään kalliimpaa lisensointikustannusta Microsoft SQL Server Enterprise -versiossa, joka sisältää optimointitoiminallisuuden suurille datamäärille osioinnin kautta. Tämän takia haluttiin tutkia, voidaanko paremmat vasteajat saavuttaa järjestämällä kannan taulukot uudelleen. Työssä tutkittavan optimoinnin tarkoituksena oli jakaa dataa ajallisesti eri taulukoihin.

Vasteajat mitattiin ajamalla valitut tietokantahaut tietyllä asiakkaalla. Järjestelmäympäristö koostuu Microsoft Windows Server -käyttöjärjestelmästä ja Microsoft SQL Server -tietokantajärjestelmästä. Kehitysympäristö on toteutettu Python-ohjelmointikielellä. Laitteisto vaihtelee asiakkaiden kesken ja tämän takia kaikki testaukset tehdään yhden asiakkaan palvelinympäristössä.

Kieli: ruotsi Avainsanat: optimointi, Microsoft SQL Server, lokiseurantajärjestelmä, Python

BACHELOR'S THESIS

Author: Mathias Lithen
Degree Programme: Information technology, Vasa
Supervisor: Kaj Wikman

Title: Optimization of Data Structures in Microsoft SQL Server for Log Monitoring System

---

Date May 28, 2018 Number of pages 21 Appendices 7

Abstract

The purpose of this thesis is on behalf of company to investigate the possibility to improve runtime response times through partitioning in Microsoft SQL Server Standard Edition. The company develops a log monitoring system for personal data in the public sector. The intent is to investigate if it is possible to improve response times when the system contains between 100 and 1000 million events. With these amounts of data queries against the database can be slow. This can be counteracted with indexing, though many indexes can slow down system performance as at least 60 000 events are fetched to the database every 24 hours.

The company wants to avoid the higher licensing costs for the Enterprise Edition of Microsoft SQL Server, which comes with an optimizing feature for large data amounts through partitioning. This is the reason for investigating possible lower response times through refactoring table structures in the database. The optimization method investigated in this thesis is to divide data temporally in different tables.

Measurements are done by registering response times executing queries in one of the customer's environment. The system environment consists of Microsoft Windows Server and Microsoft SQL Server, the development environment is based on the Python coding language. Hardware varies from customer to customer, so all measurements are done in the same environment.

Language: Swedish Key words: Optimization, Microsoft SQL Server, log monitoring system, Python

---

Innehållsförteckning

1 Inledning 1
1.1 Uppdragsgivare 1
1.2 Bakgrund 1
2 Uppgift 1
2.1 Systemets grundläggande funktion 2
2.2 Problembild 3
3 Tekniker 4
3.1 Index 4
3.2 Python baserat websystem mot SQL Server 5
3.3 Microsoft SQL Server 6
3.3.1 Graphical Execution Plans 6
3.4 Databas partitionering 7
3.5 Mätningar 7
4 Skapande och testande av partitioner 8
4.1 Originaltabellens struktur 9
4.2 Årsvis partitionering 9
4.3 Insättning av data till partitioner 11
4.4 Testande av partitioner 12
4.5 Ändringar i källkoden 12
4.6 Alternativt skapande av partitioner 14
5 Resultat 15
5.1 Mätresultat 16
5.2 Slutresultat 18
6 Diskussion 19
7 Källförteckning 20

Förklarningar

DBCC	Database Console Commands är T-SQL kommandon för hantering av en Microsoft SQL Server databas.
IIS	Internet Information Services är en webbserver utvecklad av Microsoft.
MS SQL Server	MS SQL Server är ett hanteringssystem för relationsdatabaser utvecklat av Microsoft. I denna text förkortat till MSSQL, om ingen version anges avses Standard versionen.
SQL	SQL eller Structured Query Langugae är ett specialiserat programmeringsspråk designat för datahantering i relationsdatabaser.
Index	Ett databas index är en data struktur som snabbar upp operationer som hämtar data från en databas. Detta fås mot en kostnad av fler skrivningar då data sparas och databasen tar fysiskt mer utrymme på disk.
LogMonitor	Ett loggуppföljningssystem utvecklat av Neotide Ab.
Windows Server	Windows Server är ett märkesnamn för operativsystem för servrar utvecklat av Microsoft.
AD	Active Directory, en Windows Server tjänst som innehåller en rad olika undertjänster, bland annat hantering av användare i ett nätverk.
Tabell	En tabell är en samling av relaterad data i ett strukturerat format. Tabeller består av rader och kolumner.
Tidskomplexitet	Beskriver hur krävande en algoritm är, ofta beroende på storleken av indata. För att presentera tidskomplexitet i detta arbete används ”Big O”-notationen.
Query hint	En MSSQL option eller strategi som kan tvingas att användas genom att specificera dem i SQL satsen som exekveras.
HTML	Hypertext markup language är ett standardiserat språk för skapandet av websidor.
HTTP	Hypertext transfer protocol, ett protokoll för överföring av websidor och information. Finns också som en krypterad version, HTTPS.
GNU	General Public License är en licens för gratis mjukvara som ger användaren rätt att använda, dela eller modifiera mjukvaran.

1 Inledning

Uppdraget gick ut på att undersöka huruvida det är möjligt att optimera tabellstrukturer hos ett snabbt växande databasbaserat loggsystem. Undersökningen skulle visa huruvida bättre svarstider vid systemkörningar är möjliga.

1.1 Uppdragsgivare

Neotide Ab är ett privat aktiebolag som grundades år 1999 av Johan Kullas och Patrik Simons. Neotide grundades i Helsingfors men verksamheten flyttades 2001 till Vasa. Neotide har idag 11 heltidsanställda. Neotide har specialiserat sig på databas- och nätbaserade lösningar och erbjuder produkter så som ledningssystem, loggsystem, kontrollsystem för dieter och uppföljningssystem för antibiotika och infektioner. Bland Neotides kunder finns alla Finlands sjukvårdsdistrikt och städer som Helsingfors, Seinäjoki och Vasa.

1.2 Bakgrund

LogMonitor är ett logguppföljningssystem specialiserat för patientadministration utvecklat av Neotide. Systemet förebygger missbruk av känslig information och förbättrar datasäkerheten. Genom att sammanställa loggdata från en rad olika system som används inom offentlig förvaltning ger systemet användaren en överblick om vad som sker i de övervakade systemen. Systemets källkod är skriven i Python och körs på Microsofts integrerade webbserver IIS. Systemet är nätbaserat vilket betyder att det används via en webbläsare och fungerar utan installation hos klienten. Autentisering eller själva inloggningen till systemet sker mot AD. Systemet samlar in data från de övervakade systemen automatisk med önskat intervall, oftast en gång per dygn och analyserar data automatiskt. All data som samlats in lagras i en MSSQL databas.

2 Uppgift

Uppdraget som Neotide beställde var att undersöka huruvida man kan förbättra svarstider genom att optimera tabellstrukturer i MSSQL. Systemet som uppdraget gäller är ett loggsystem med datamängden 100 till 1000 miljoner händelser med en ökning på 60 000 händelser per dygn.

Med optimering av tabellstruktur avses i detta fall att dela upp data till separata tabeller tidsmässigt. Även källkod måste anpassas enligt tabelloptimeringarna för att uppnå samma systemfunktioner som tidigare.

Enterprise versionen av MSSQL erbjuder möjlighet till partitionering av tabeller, detta innebär att MSSQL logiskt (i bakgrunden) kan dela upp enskilda tabeller i flera delar och till och med på olika lagringsenheter. Enterprise versionen kan med en eller flera instanser bilda ett kluster och montera olika partioner till olika noder för att dela upp belastningen i systemet. Eftersom partitionerna används som en logisk enhet behöver man inte ändra på SQL-kommandon för användning av dessa tabeller. [19]

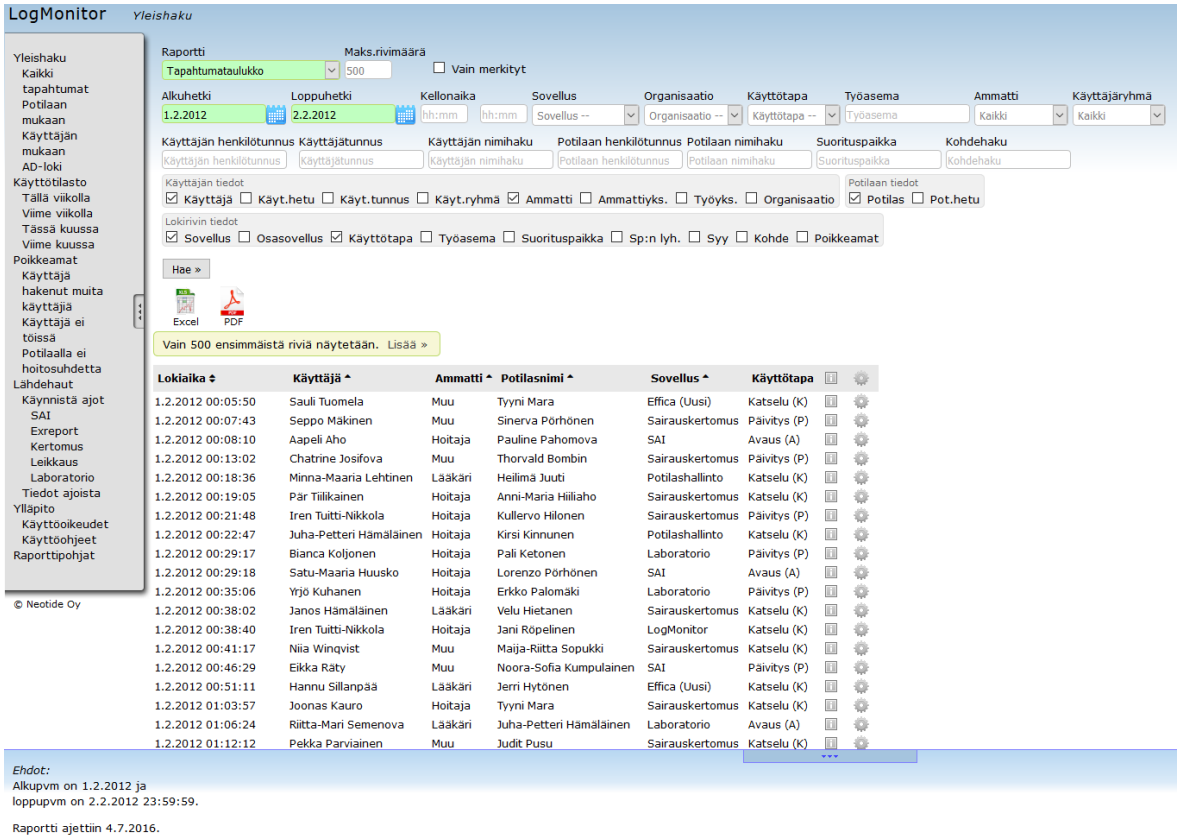
Enligt de prisuppgifter Microsoft publicerat för MSSQL 2016 är Enterprise versionen ca fyra gånger dyrare än Standard, 3717 mot 14256 amerikanska dollar. Prissättningen är per processorkärna. [20]

2.1 Systemets grundläggande funktion

LogMonitors grundläggande funktioner kan delas i tre logiska sektioner; insamling, analys och presentation av data. Insamling och analys sker automatiskt, typiskt en gång per dygn nattetid för att minimera belastningen hos databasen. De insamlade logghändelserna berikas med data från andra integrerade system, exempelvis vårddata och arbetstidsuppföljning.

LogMonitor ger användaren möjlighet att enkelt producera en rapport över händelser i ett övervakat system. Rapporter kan anpassas på en rad olika sätt och begränsas med ett flertal optioner, exempelvis med ett tidsintervall eller yrkesgrupper. Genom att se på antalet logghändelser och tider producerar systemet rapporter på belastningskurvor. Systemet flaggar automatiskt händelser som kan anses vara avvikande och producerar rapporter över dessa. Användaren kan enkelt exportera alla rapporter i Excel eller PDF format.

Eftersom användaren via LogMonitor har tillgång till känslig information, personuppgifter och patientdata, från de integrerade systemen. På grund av lagstadgade krav, beskrivna i första stycket i 2.2 Problembild, loggas all aktivitet inom LogMonitor där sådan information hanteras.



Figur 1. LogMonitors användargränssnitt.

2.2 Problembild

Enligt lagen om elektronisk behandling av klientuppgifter inom social- och hälsovården och lagen om social- och hälsovårdsministeriets förordning om journalhandlingar skall loggar föras över användning och utlämning av elektroniska patientuppgifter och sparas oförändrade i tolv år. [1] [2]

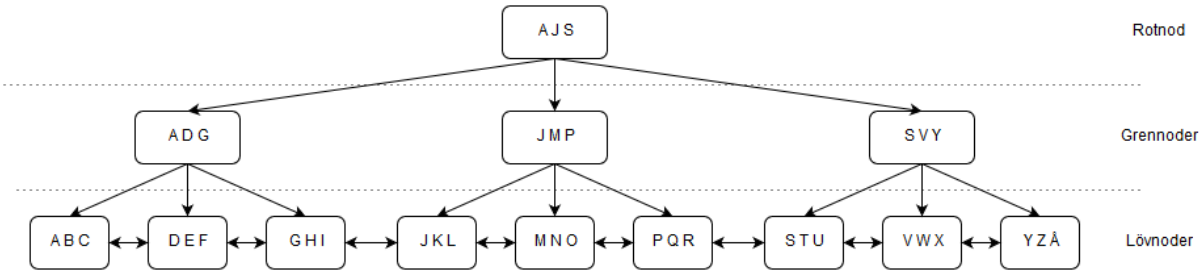
En av företagets kunder producerade ca 210 miljoner händelser året 2015, och ca 105 miljoner händelser i slutet på juni 2016. Med en linjär ökning på 210 miljoner händelser per år över en period på tolv år resulterar detta i ca 2520 miljoner logghändelser. Med en radstorlek i genomsnitt 103 byte (se Kodexempel 1) resulterar detta i en tabell på ca 260 gigabyte.

3 Tekniker

I detta kapitel presenteras de huvudsakliga teknikerna och mjukvarorna som använts i arbetet.

3.1 Index

Ett index är en datastruktur vars primära funktion är att representera data i en ordnad form. Den logiska ordningen skapas med en dubbelt länkad lista, noden har alltså pekare som hänvisar till den föregående noden och nästa nod i ordningen. Noden refererar också till var i filsystemet själva data existerar. Noderna sparas på en så kallad sida, en sida är den minsta lagringsenheten som MSSQL använder. Databasen försöker spara så många noder på en sida som möjligt för att minimera läsningar från filsystemet. Databasen hittar noderna via ett balanserat sökträd (refereras till som b-träd i fortsättningen). Egenskaperna hos ett b-träd är den största orsaken till vad som gör indexering snabbt. Den andra stora faktorn är att indexen sparas i kluster, indexnoder med liknande nycklar sparas med en logisk ordning nära varandra. [12]



Figur 2. Modell över ett b-träd, skapad med www.draw.io.

Ett b-träd är en datastruktur som sorterar data i en hierarkisk trädmodell. Trädet består av en rotnod, grennoder och lövnoder. Indexen i tabellen är i vårt fall lövnoderna, alla index ligger på samma nivå i trädet, detta gör att trädet är balanserat. Indexen hittas genom att från rotnoden via grennoderna jämföra en nyckel vid varje gren, grennoderna innehåller ett intervall med nycklar som leder till ett nytt intervall med nycklar eller själva lövnoderna. B-träd används av flera olika filsystem och databaser. [11]

Kodexempel 1. Information om originaltabellens b-träd.

```
SELECT      index_depth,
            index_level,
            record_count,
            avg_page_space_used_in_percent,
            min_record_size_in_bytes,
            max_record_size_in_bytes,
            avg_record_size_in_bytes
FROM sys.dm_db_index_physical_stats(DB_ID('logmonitor'),
OBJECT_ID('dbo.log'), 1, NULL, 'DETAILED')
```

4	0	324805517	89,32635	44	1079	102,865
4	1	4709706	91,2924	18	26	23,089
4	2	15987	65,54108	18	26	23,228
4	3	76	22,41166	18	26	21,894

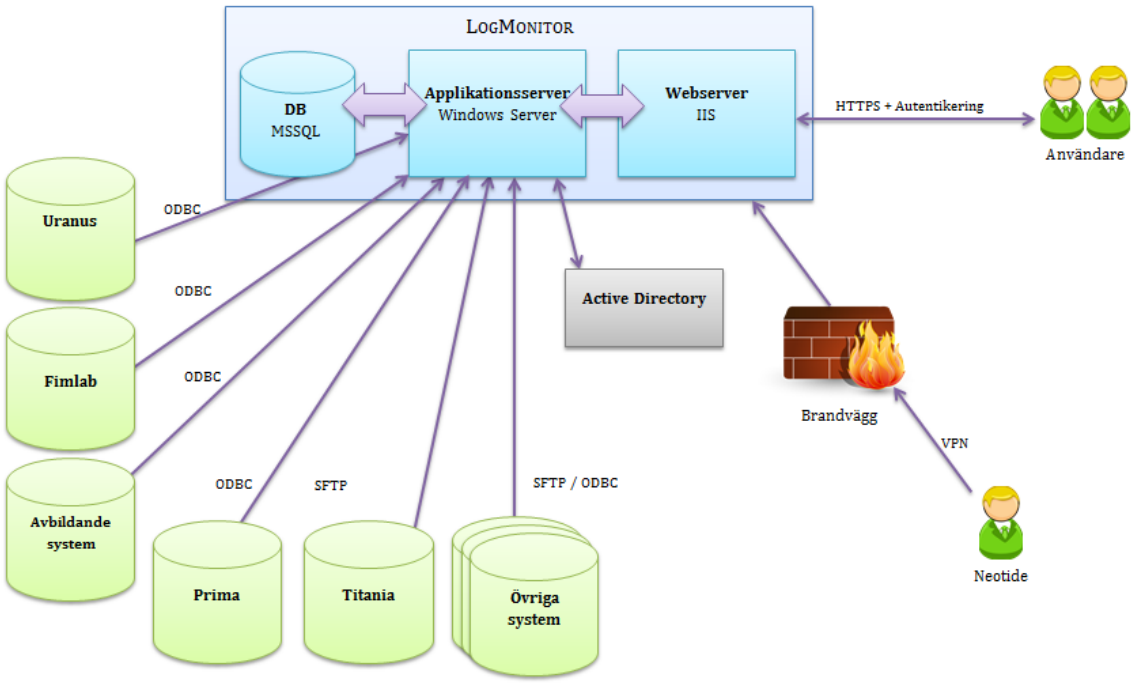
Antalet rader i originaltabellen närmar sig 325 miljoner, b-trädet består av en rotnod, två nivåer med grennoder och en nivå med lövnoder. Djupet eller antalet nivåer med grennoder växer logaritmiskt och lövnoderna sparas i en logisk ordning för sekventiell genomgång, detta bidrar till att b-trädet är väldigt effektivt. En sökning i ett b-träd har tidskomplexiteten $O(\log n)$. [12]

3.2 Python baserat websystem mot SQL Server

Systemet är byggt med Python och används genom ett grafiskt användargränssnitt via webbläsare. System är uppbyggt från grunden av Neotide och använder sig av en Python omgivning och en rad olika bibliotek som baserats på öppen källkod. [4]

Så gott som hela programlogiken utförs på serversidan. Eftersom det till stora delar är frågan om ett rapporteringssystem, så är exekvering av SQL-kommandon mot databasen en kritisk del av systemets prestanda.

Gränssnittet är helt HTML baserat och kräver inga komponenter från klientsidan. Detta delvis för att systemet kräver hög säkerhet, man vill begränsa öppna nätverksportar och typen av trafik som går mot serversidan. Detta innebär att endast nätverksport 80 (HTTP) eller 443 (HTTPS) används för kommunikation mot klienten. För kontakt med databasen från serversidan eller vid inläsningar av data används nätverksport 1433. [5]



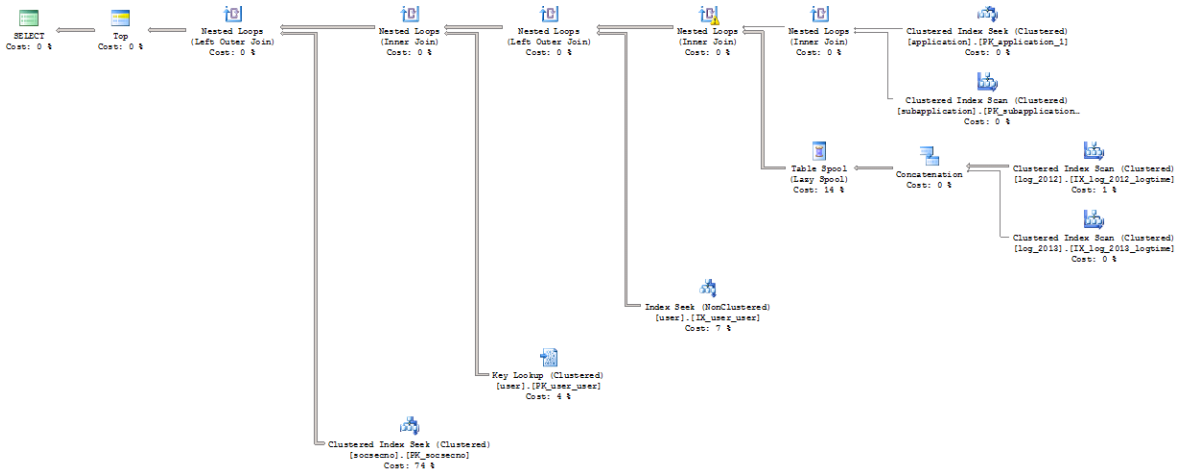
Figur 3. Teknisk vy över LogMonitor.

3.3 Microsoft SQL Server

Microsoft SQL Server (MSSQL) är en databashanterare för relationsdatabaser. MSSQL använder sig av en utvidgad dialekt av SQL som heter Transact-SQL (T-SQL). För användning, konfigurering, administrering av MSSQL används det grafiska verktyget SQL Server Management Studio (SMSS).

3.3.1 Graphical Execution Plans

För detta arbete var ett av de viktigaste verktygen möjligheten att analysera hur MSSQL verkligen exekverar SQL-kommandon. MSSQL har en inbyggd optimeringsfunktion, ”Query Optimizer”, vars inverkan inte alltid kan förutspås. Genom att aktivera ”Include Actual Execution Plan” kan man analysera vad som verkligen sker under exekveringen via ett grafiskt användargränssnitt. Verktyget producerar ett blockdiagram där alla operationer som utförts presenteras. Noggrannare information för varje steg fås också genom att inspektera de enskilda blocken.



Figur 4. MSSQL grafisk exekveringsplan.

3.4 Databas partitionering

Vid partitionering av en databas eller dess beståndsdelar delar man upp en större enhet till distinkta individuella enheter. Partitionering sker vertikalt eller horisontellt. Vid vertikal partitionering delar man upp en tabell genom att splittra raden i två eller flera delar och spara dem i skilda tabeller. Vid horisontell partitionering hålls raden intakt men raderna sparas i skilda tabeller enligt en nyckelkolumns värde.

För att partitionera logghändelserna i LogMonitor är det naturligt det görs en horisontell partitionering årsvis enligt händelsens tidpunkt.

3.5 Mätningar

MSSQL har ett antal olika inbyggda cache system och tiden för exekvering av ett SQL-kommando varierar kraftigt beroende på om det finns lagrat i cacheminnet. Med DBCC-kommandon finns möjligheten att tömma cacheminnet för att simulera en så kallad kall körning.

LogMonitor mäter tiden för alla exekveringar av SQL-kommandon. Om tiden överstiger en bestämd gräns skickas ett meddelande via e-post med information till utvecklarna. På så sett kan man över en tid analysera huruvida en ändring har en positiv inverkan eller inte.

Eftersom partitioneringen inte används i produktion är det huvudsakliga mätinstrumentet analys av grafiska exekveringsplaner och analys av mätresultat från tidsstatistik över exekveringar.

```
set statistics time on

SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 2 ms.

(0 row(s) affected)

(1 row(s) affected)

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
```

Figur 5. Tidsstatistik för ett enkelt SQL-kommando.

4 Skapande och testande av partitioner

I detta kapitel byggs en testmodell för partitionerade tabeller, en gemensam vy för partitionerna och för- och nackdelar med dessa undersöks. Alla tester görs på en och samma server, en virtuell server som har tillgång till en Intel Xeon E7-4830 processor och 16GB internminne. Serverns operativsystem är Windows Server 2008 R2 och som databasserver används Microsoft SQL Server 2008 R2 Enterprise Edition. Motsvarande SQL-kommandon testades i både en Standard och Express version av SQL Server, de olika versionerna producerade motsvarande exekveringsplaner. Eftersom SQL Server är en stängd programvara är det dock möjligt att skillnader mellan versionerna förekommer internt som inte syns via exekveringsplanerna. I databasens händelsetabell lagras ca 325 miljoner rader. Två partitioner har skapats, ”log_2012” och ”log_2013”, och innehåller ca 21 miljoner rader var. Partitionerna innehåller all data från respektive år som originaltabellen innehåller.

4.1 Originaltabellens struktur

Tabellen består av ett unikt id, tid för händelsen och ett antal olika nycklar till diverse uppslagstabeller. Kodexempel 2 visar en förenklad struktur för tabellen. Eftersom logghändelser läses in från flera system olika kan inte systemens egna primärnycklar användas utan en identitetskolumn används.

Kodexempel 2. SQL kod för förenklad händelsetabell.

```
create table log (
            rowid int identity(1, 1) not null
            ,event int not null
            ,logtime datetime not null
            ,constraint PK_log_rowid primary key nonclustered (rowid asc)
)
create clustered index IX_log_logtime on log (logtime)
create index IX_log_event on log (event)
```

4.2 Årsvis partitionering

Årsvisa tabeller skapas och namnges med ett suffix för att indikera från vilket år tabellens innehåll är. Tabellerna skapas precis som originaltabellen med samma nycklar och index.

Kodexempel 3. SQL-kommandon för att skapa partitioner.

```
create table log_2015 (
            rowid int not null
            ,event int not null
            ,logtime datetime not null
            ,constraint PK_log_2015_rowid primary key nonclustered (rowid asc)
)
create table log_2016 (
            rowid int not null
            ,event int not null
            ,logtime datetime not null
            ,constraint PK_log_2016_rowid primary key nonclustered (rowid asc)
)
create clustered index IX_log_2015_logtime on log_2015 (logtime)
create clustered index IX_log_2016_logtime on log_2016 (logtime)
create index IX_log_2015_event on log_2015 (event)
create index IX_log_2016_event on log_2016 (event)
```

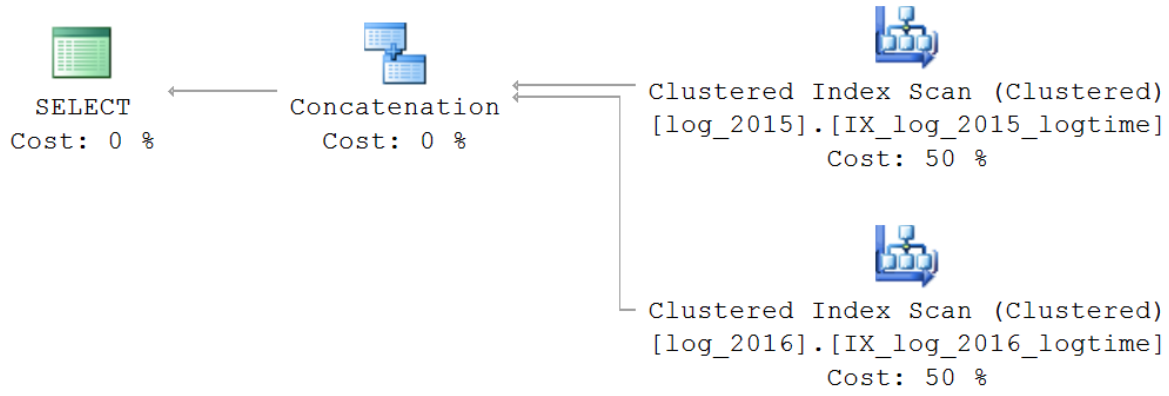
För att minimera ändringarna i källkoden skall SQL-kommandon inte köras mot enskilda partitioner därför skapas en vy över tabellerna. För att vara säker på att partitionernas tabelldefinition inte ändras används ”schemabinding” klausulen, detta betyder att inga ändringar till partitionerna som inverkar på vyns funktionalitet är tillåtna. Tabellerna som ingår kan inte raderas medan vyn existerar. [6]

Kodexempel 4. SQL-kommando för att skapa en vy över två tabeller.

```
create view logs with schemabinding
as
select rowid, event, logtime from dbo.log_2015
union all
select rowid, event, logtime from dbo.log_2016
```

Även om vyn består av flera tabeller så ses den som en tabell, och den används som en tabell. Genom att inspektera exekveringsplanen för ett enkelt select-kommando mot vyn konstateras att bägge partitioners index skannas för matchande rader, även om sökningen i detta fall är begränsad till år 2015. För ökad prestanda skall så få index som möjligt skannas så inget onödigt arbete utförs.

```
select * from logs where year(logtime) = 2015
```



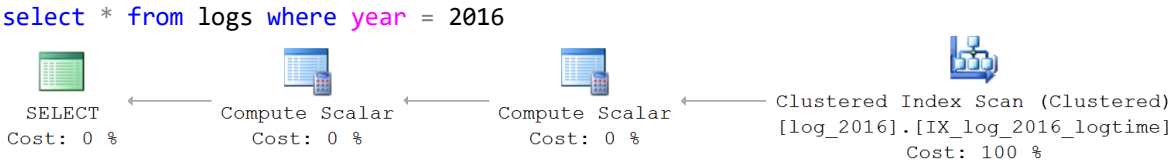
Figur 6. Exekveringsplan vid hämtning ur vy.

Med fler partitioner utförs mer onödigt arbete, en begränsning för att tvinga sökningen till så få partitioner som möjligt behövs. En restriktion för varje partition läggs till så det kan begränsas vilka partitioners index som skannas. Med en automatiskt beräknad årskolumn baserad på ”logtime” kan en restriktion skapas för tabellen, för att spara på utrymme skall också kolumnen tvingas till den minsta datatypen möjlig. Denna restriktion gäller vid insättning eller uppdatering av rader i tabellen och förbättrar exekveringsplanen.

Kodexempel 5. Skapa restriktion.

```
alter table log_2015 add year as cast((year(logtime)) as smallint) persisted not null
alter table log_2016 add year as cast((year(logtime)) as smallint) persisted not null

alter table log_2015 with check add constraint CK_log_2015_year check (year = 2015)
alter table log_2016 with check add constraint CK_log_2016_year check (year = 2016)
```

Då exekveringsplanen för SQL-kommandot granskas med restriktionerna på plats fås en förbättrad exekveringsplan där endast den tabells index som sökningen berör skannas. Detta upplägg utgör grunden för det manuella partitioneringssystemet.



Figur 7. Förbättrad exekveringsplan.

4.3 Insättning av data till partitioner

Partitionernas unika radnummer kan inte vara en identitetskolumn då systemet behöver ett unikt värde över alla partitioner för alla händelser. Detta skapar problem vid insättningar till partitionerna eftersom identitetskolumnen måste hanteras manuellt. MSSQL 2012 och nyare versioner har en lättanvänd inbyggd funktion för sekvenser som kan användas till detta ändamål. Efter att sekvensen är skapad kan man i ett insert-kommando kalla på ”next value for” och sekvensen namn.

Kodexempel 6. Skapa en sekvens i MSSQL 2012 eller nyare version.

```
create sequence id_seq start with 1 increment by 1

insert into log_2016 (id, ...) values (next value for id_seq, ...)
```

För äldre versioner av MSSQL är det möjligt att skapa en sparad procedur som returnerar nästa värde i en sekvens med hjälp av en tabell som består av en identitetskolumn och insert-kommandots ”output” klausul, vilket returnerar den insatta radens id i detta exempel. Detta är även möjligt i en miljö med mångkärniga processorer då ”output” klausulen används med insert-kommandot är det garanterat en atomisk operation. [7]

Kodexempel 7. Sparad procedur som returnerar nästa värde i en sekvens.

```
create table log_sequence (id int identity(1, 1) primary key)

create procedure next_value_log_sequence as
begin
            insert into log_sequence output inserted.id default values
end
```

4.4 Testande av partitioner

De uppmätta tiderna och detaljerna för den SQL-sats som använts för de olika försöken finns bokförda i bilaga 1 till 6. Resultatet finns sammanfattat på Bilaga 7. Tiderna är uppmätta med den inbyggda tidsstatistikfunktionen i MSSQL. Samma SQL-sats har exekverats mot originaltabellen och partitionen ett antal gånger, för varje exekvering har den totala tiden bokförts. Beroende på om försöket simulerar en kall körning eller inte körs ett antal DBCC-kommandon före varje exekvering av den SQL-sats som testas. För detta används ”checkpoint” och ”dbcc dropcleanbuffers” för att tömma buffercache i MSSQL. Därefter användes ”dbcc freeproccache” för att tömma cache med alla exekveringsplaner. [16][17]

Grafiska exekveringsplaner jämfördes ifall oväntade problem skulle uppstå. Planerna var väldigt lika i alla fall bortsett från då sökningen gjordes över flera år och på så sätt läste data från två partitioner, detta diskuteras i kapitel 5.1.

4.5 Ändringar i källkoden

En av systemets uppgifter är att lagra händelser separat från källsystemet på ett sådant sätt att de hålls intakta och orörda. Detta betyder att källkoden inte hanterar uppdatering eller borttagning av händelser utan endast insättning och hämtning av data.

Modulen som hanterar insättningar i loggtabellen använder redan på grund av varierande tabellstrukturer hos olika källsystem moduler och kan lätt anpassas för partitioner. Modulen tar emot en funktion som fungerar som en adapter för källsystemet i fråga, en adapterfunktion som analyserar logghändelsens tidpunkt och utgående efter den sparas händelsen i rätt partition.

Kodexempel 8. Förenklad adapterfunktion för insättningar till partitioner.

```
def insert_adapter_partitions(self, columns, cursor, values):
    sql = ("insert into log_%(year)d (%(columns)s)"
           " output inserted.rowid"
           " values (%(values)s)")
    args = {'columns': columns,
            'values': values,
            'year': values.get('logtime').year}
    cursor.execute(sql, args)
    rowid, = cursor.fetchone()
    return rowid
```

Systemet använder sig av en rapportgenerator som beroende på begränsningarna, dimensionerna och fälten i rapporten genererar de SQL-kommandon som behövs. Som tidigare diskuterat i detta kapitel producerar MSSQL en bättre exekveringsplan genom att ange ett eller flera årtal i SQL-kommandots where-sats. Eftersom alla SQL-kommandon systemet kör är tidsbegränsade med ett start- och slutdatum kan dessa användas för intervallet för att ange vilket eller vilka år sökningen gäller. Då systemet bygger upp ett SQL-kommando är de olika delarna logiskt uppdelade i listor tills de sammanställs och kommandot exekveras, detta gör det möjligt att enkelt uppdatera en viss del av kommandot.

Kodexempel 9. Förenklad funktion som modifierar where-satsen för sökning från partitioner.

```
def modify_sql(self, values):
    select, top, table, where, joins, group, order = self.sql_parts
    start_year = values.get('start_time').year
    end_year = values.get('end_time').year
    years = ','.join([str(y) for y in range(start_year, end_year + 1)])
    where.append("%%syear in (%s)" % years)
    self.sql_parts = select, top, table, where, joins, group, order
```

Systemet använder sig av speciella databasklasser för upprätthållande av databasens struktur, klasserna innehåller också namnet på tabellen i databasen. Med dessa klasser utför systemet automatiserade ”Code first” databasmigrationer, en strategi som också används av bland annat Microsofts Entity Framework. Detta görs med ett bibliotek utvecklat av Neotide för de olika Python baserade produkterna. Då händelsetabellen definieras i källkoden används vyn över partitionerna som tabell, detta minimerar ändringarna som behövs för att hämta data från partitionerna eftersom vyn nu används automatiskt, t.ex. skulle värdet för variabeln ”table” i kodexempel 9 automatiskt vara namnet på partitionernas vy. [18]

4.6 Alternativt skapande av partitioner

För att sätta perspektiv på manuellt skapande av partitioner i MSSQL undersöks hur partitionerade tabeller skapas i MySQL. MySQL Community Server finns gratis tillgänglig under en GNU licens, partitioner stöds i denna version. Enligt DB-Engines är MySQL rankad som den näst populäraste databashanteraren efter Oracle och före MSSQL. I dessa exempel används MySQL 5.7 och MySQL Workbench 6.3. [21]

MySQL stöder en rad olika partitioneringstyper men för motsvarande partitioner används så kallad ”range partitiong” eller intervall partitionering. För att skapa partitioner definieras en partitioneringsfunktion då tabellen skapas. Intervall definieras i detta fall enligt logtime med ”less than” kommandot. Partitioner skapas automatiskt enligt de intervall som definieras.

```
CREATE TABLE log (
    id INT NOT NULL,
    logtime DATETIME NOT NULL DEFAULT '1900-1-1',
    CONSTRAINT PRIMARY KEY PK (logtime)
)
PARTITION BY RANGE (YEAR(logtime)) (
    PARTITION p2014 VALUES LESS THAN (2015) ,
    PARTITION p2015 VALUES LESS THAN (2016) ,
    PARTITION p2016 VALUES LESS THAN MAXVALUE
);
```

Info | Columns | Indexes | Triggers | Foreign keys | Partitions | Grants

Name	Ordinal Pos	Partition Method	Partition expres...	Partition description
p2014	1	RANGE	YEAR(logtime)	2015
p2015	2	RANGE	YEAR(logtime)	2016
p2016	3	RANGE	YEAR(logtime)	MAXVALUE

Figur 8. SQL-kommando för att skapa en tabell med partitioner i MySQL.

Då ett select-kommando körs mot tabellen utan where-sats utförs en ”Full table scan”, detta motsvarar beteendet i MSSQL där alla partitioners index skannades (beskrivet i 4.2 Årsvis partitionering). Detta löses i MySQL genom att i where-satsen använda ”logtime between <startdatum> and <slutdatum>”, sökningen utför då endast en ”Index range scan” för de partitioner som infaller i intervallet som angivits.

Partitioner läggs enkelt till genom ett ”alter table”-kommando där ett nytt partitionsintervall definieras, partitioner kan också raderas utan att resten av tabellen berörs. En kolumn med ”auto increment” egenskapen, motsvarande ”identity” i MSSQL, kan också användas som nyckel i partitionerade tabeller, dock måste kolumnen som används i partitioneringsfunktionen vara med i primärnyckeln. Det klustrade indexet kan inte separeras från primärnyckeln som i MSSQL. Med tanke på LogMonitors logtabell finns det alltså både för- och nackdelar med MySQL. [22]

5 Resultat

Det ursprungliga antagandet var att eftersom en partitions index är en bråkdel av originaltabellens måste sökningar vara snabbare. I kapitel 3.1 diskuterades index och vad som gör indexering snabb. Eftersom skillnaden mellan tidskomplexiteten för en sökning i originaltabellen (8,5) och partitionen (7,3) är väldigt liten så kommer också storleken på tabellen ha en väldigt liten inverkan. Även om originaltabellen växer som beskrivits i kapitel 2.2 till över 2500 miljoner händelser är b-trädets tidskomplexitet endast 9,4. Det skiljer ca 300 miljoner rader mellan originaltabellen och partitionen men bägge tabellers b-träd har fortfarande endast djupet 4.

```
SELECT
            index_depth,
            index_level,
            record_count,
            avg_page_space_used_in_percent,
            min_record_size_in_bytes,
            max_record_size_in_bytes,
            avg_record_size_in_bytes
FROM sys.dm_db_index_physical_stats(DB_ID('logmonitor'),
OBJECT_ID('dbo.log_2013'), 1, NULL, 'DETAILED')
```

4	0	21772365	99,41376	46	1079	83,75
4	1	231967	98,41441	15	23	19,227
4	2	618	20,30208	15	23	19,297
4	3	8	1,655547	15	15	15

Figur 9. Partitionstabellen log_2013s b-träd.

5.1 Mätresultat

Tabell 1. Resultatet av de uppmätta exekveringstiderna.

Försök	Bilaga	Exekv.tid mot originaltabell	Exekv.tid mot partition	SQL	Kall körning
1	1	(86 ± 9) ms	(85 ± 4) ms	1	Ja
2	2	(285 ± 22) ms	(279 ± 28) ms	2	Ja
3	3	(50,6 ± 0,1) ms	(47,4 ± 0,4) ms	2	Nej
4	4		(282 ± 20) ms	2	Ja
5	5	(53,1 ± 3,8) ms	(83,5 ± 1,4) ms	2	Nej
6	6	(281 ± 32) ms	(292 ± 16) ms	2	Ja

Att Exekveringstiderna är inom varandras osäkerhet verkar vara ett återkommande fenomen för resultaten där en kall körning simuleras. Slutsatsen för dessa körningar är att exekvering mot partition eller originaltabell har motsvarande prestanda.

Försök 3 visar en liten men tydlig fördel till exekvering mot partitionen som förväntat, även om indexets storlek inte är avgörande förväntas en kortare exekveringstid mot partitionen.

Försök 4 jämförs med tiderna i försök 2. Som tidigare diskuterat är en sida den minsta enhet som databasen sparar data i. Sidorna är 8192 byte stora varav 8060 byte används för data, om en rad inte ryms på en sida delas inte raden utan hela raden flyttas till nästa sida. Originaltabellens sidor är i medeltal fyllda till 89,3% medan partitionens sidor fylls till 99,4%. Detta betyder att varje läsning från disk hämtar ca tio procentenheter mer data vid exekvering mot partitionen. För att granska om detta har någon märkbar inverkan byggdes indexet om för log_2013 och exekveringstider uppmättes. I kapitel 3.1 visas att originaltabellens sidor är fyllda till ca 89 procent (Kodexempel 1), därför används ”fillfactor” 89.

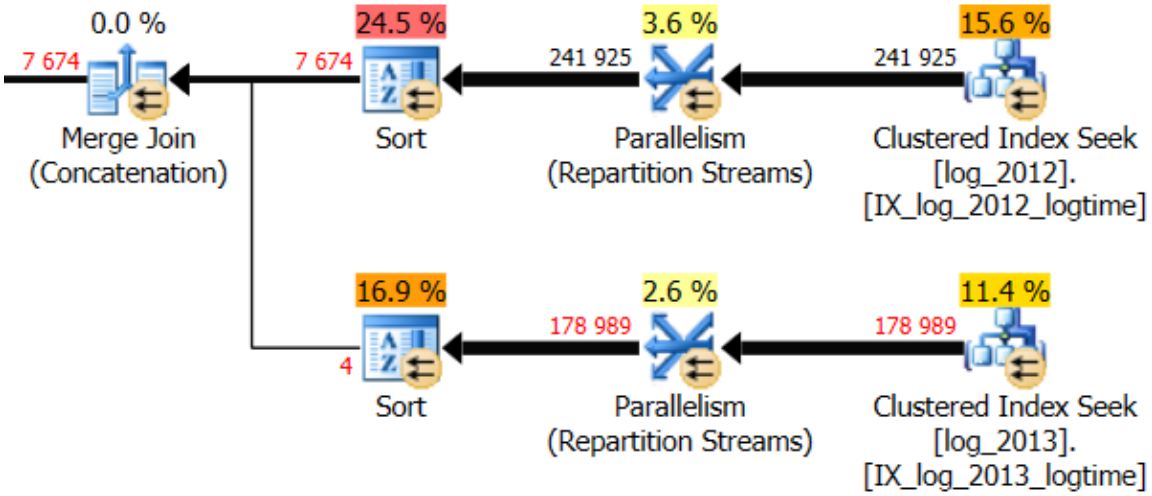
```
alter index IX_log_2013_logtime on log_2013 rebuild with (fillfactor = 89)

SELECT
           index_depth,
           index_level,
           record_count,
           avg_page_space_used_in_percent,
           min_record_size_in_bytes,
           max_record_size_in_bytes,
           avg_record_size_in_bytes
FROM sys.dm_db_index_physical_stats(DB_ID('logmonitor'),
OBJECT_ID('dbo.log_2013'), 1, NULL, 'DETAILED')
```

4	0	21772365	89,55859	46	1079	83,75
4	1	257486	98,80525	15	23	19,28
4	2	685	20,18918	15	23	19,496
4	3	9	1,865579	15	15	15

Figur 10. log_2013 index byggdes om för att motsvara originaltabellens index.

Resultatet i försök 4 motsvarar de tider och osäkerheter som förekommer i försök 2, ingen märkbar inverkan kunde påvisas i detta exempel.

Det största problem som uppstår med partitioner är hur MSSQL exekverar SQL-kommandon för sökningar som sträcker sig över flera år. Exekveringsplanen nedan visar en del av en exekveringsplan för en hämtning av de första 5000 resultaten mellan december 2012 och januari 2013. Till vänster är en "Merge Join" operation för att konkatenera resultaten från de två partitionerna, operationen är väldigt effektiv men kräver att indatat är sorterat och sortering är däremot en väldigt dyr operation. [8]



Figur 11. Exekveringsplan för sökning ur två partitioner.

SQL-kommandot som körs använder endast ”logtime” som sorteringskolumn men sorteringsoperationerna som syns i figur 10 sorterar raderna enligt alla kolumner i raden. Sorteringen kunde undvikas genom att använda ett unikt klusterindex, vilket i detta fall inte är ett alternativ. Även om sorteringarna sker parallellt ser detta ut som en nackdel. Exekveringstider för detta visas i försök 5 och försök 6 bokförda i bilaga 5 och 6. I försök 5 framkommer en klar fördel för originaltabellen där resultatet från index sökningen inte behöver sorteras. Försök 6 ger samma resultat som de övriga försöken där en kall körning simulerades, tiderna är väldigt nära varandra och inom varandras osäkerhet. [15]

5.2 Slutresultat

Undersökningen visar att förbättrade svarstider vid systemsökningar är möjliga genom partitionering i en miljö där liknande SQL-exekveringar finns i cacheminnet och körningarna endast hämtar data från en partition. Motsvarande svarstider fås vid körningar mot en partition som mot en större tabell vid körningar där liknande SQL-exekveringar inte finns i cacheminnet. Vid sökning från flera partitioner förekommer en negativ

inverkan på svarstiden vid körningar där liknande SQL-exekveringar finns i cacheminnet och motsvarande svarstider vid körningar där liknande SQL-exekveringar inte finns i cacheminnet.

6 Diskussion

Undersökningen skulle visa om det är möjligt att förbättra svarstider vid systemkörningen genom att optimera tabellstrukturer. Detta examensarbete har visat hur partitioner kan skapas och hur ändringar i källkod kunde göras för att lösa de problem som partitionerna för med sig. Mätresultaten har visat hur vi kan vänta oss att exekvering mot en partition eller partitioner beter sig jämfört mot originaltabellen. I examensarbetet förklaras också varför de resultat som uppmätts verkar troliga, därför är jag nöjd med undersökningen och resultatet. En intressant aspekt som inte kunde testas var att flytta partitionerna till ett snabbare lagringsmedia så som ett SSD-minne.

Under arbetets gång har jag fördjupat mig i flera områden inom databashantering. Indexering är inget nytt i sig själv men hur de verkligen fungerar och hur databasen lagrar indexen var nytt för mig. De grafiska exekveringsplanerna och hur man analyserar dem har varit en intressant del i arbetet och kommer att vara till nytta vid problemlösning i framtiden. Kunskap om vyer, DBCC-kommandon och allmän funktionalitet i MSSQL har jag också fått under arbetet.

7 Källförteckning

[1] Finlex 5 § 9.2.2007/159 Lag om elektronisk behandling av klientuppgifter inom social- och hälsovården http://www.finlex.fi/sv/laki/ajantasa/2007/20070159 [Hämtat 28.06.2016]

[2] Finlex 24 § 298/2009 Social- och hälsovårdsministeriets förordning om journalhandlingar http://www.finlex.fi/sv/laki/alkup/2009/20090298 [Hämtat 28.06.2016]

[3] Microsoft SQL Server https://en.wikipedia.org/wiki/Microsoft_SQL_Server [Hämtat 29.06.2016]

[4] Python https://www.python.org/ [Hämtat 29.06.2016]

[5] Nätverksportar https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers [Hämtat 29.06.2016]

[6] Views https://msdn.microsoft.com/en-us/library/ms187956.aspx [Hämtat 01.07.2016]

[7] Output https://msdn.microsoft.com/en-us/library/ms177564.aspx [Hämtat 01.07.2016]

[8] Understanding Merge Joins https://technet.microsoft.com/en-us/library/ms190967(v=sql.105).aspx [Hämtat 05.07.2016]

[9] Configure the max degree of parallelism Server Configuration Option https://msdn.microsoft.com/en-us/library/ms189094.aspx [Hämtat 05.07.2016]

[10] Query hints https://msdn.microsoft.com/en-us/library/ms181714.aspx [Hämtat 06.07.2016]

[11] DBCC (T-SQL) https://msdn.microsoft.com/en-us/library/ms188796.aspx [Hämtat 07.07.2016]

[12] Index och b-träd http://use-the-index-luke.com/sql/anatomy/the-tree/ [Hämtat 08.07.2016]

[13] B-träd https://en.wikipedia.org/wiki/B-tree [Hämtat 08.07.2016]

[14] Understanding Pages and Extents https://technet.microsoft.com/en-us/library/ms190969(v=sql.105).aspx [Hämtat 11.07.2016]

[15] Avoiding sorts with merge join concatenation http://sqlperformance.com/2014/09/t-sql-queries/avoiding-sorts-merge-join-concatenation [Hämtat 14.07.2016]

[16] DBCC DROPCLEANBUFFERS https://msdn.microsoft.com/en-us/library/ms187762.aspx [Hämtat 20.07.2016]

[17] DBCC FREEPROCCACHE https://msdn.microsoft.com/en-us/library/ms174283.aspx [Hämtat 20.07.2016]

[18] Automatic Code First Migrations https://msdn.microsoft.com/en-us/data/jj554735.aspx [Hämtat 20.07.2016]

[19] Partitioned Tables and Indexes https://msdn.microsoft.com/en-us/library/ms190787.aspx [Hämtat 22.08.2016]

[20] SQL Server licensing https://www.microsoft.com/en-us/cloud-platform/sql-server-pricing [Hämtat 22.08.2016]

[21] DB-Engines Ranking http://db-engines.com/en/ranking [Hämtat 30.8.2016]

[22] Restrictions and Limitations on Partitioning https://dev.mysql.com/doc/refman/5.5/en/partitioning-limitations.html [Hämtat 30.8.2016]

Bilaga 1

Exekveringstid mot originaltabell [ms]	Exekveringstid mot partition [ms]
76	77
77	108
85	77
76	103
147	76
118	78
84	77
77	93
77	77
84	108
81	79
76	78
78	92
76	80
76	77

Resultat	
Originaltabell	Partition
(86 ± 9) ms	(85 ± 4) ms

SQL 1

top 500
8x left join
73 dagars intervall över samma år

DBCC-kommandon

CHECKPOINT;
DBCC DROPCLEANBUFFERS;
DBCC FREEPROCCACHE;

$$x_{avg} = \frac{x_1 + x_2 + \ldots + x_N}{N}$$

$$R = x_{max} - x_{min}$$

$$\Delta x = \frac{R}{2} = \frac{x_{max} - x_{min}}{2}$$

$$\Delta x_{avg} = \frac{\Delta x}{\sqrt{N}}$$

$$x_m = x_{avg} \pm \Delta x_{avg}$$

	Originaltabell	Partition
	85.9	85.3
	71	32
	35.5	16
	9.2	4.1
	(86 ± 9) ms	(85 ± 4) ms

Bilaga 2

Exekveringstid mot originaltabell [ms]	Exekveringstid mot partition [ms]
274	276
266	397
280	249
254	203
272	250
347	361
328	400
351	184
228	206
393	321
259	220
281	244
260	221
268	326
219	329

Resultat	
Originaltabell	Partition
(285 ± 23) ms	(279 ± 28) ms

SQL 2

top 5000
9x left join
159 dagars intervall över samma år

DBCC-kommandon

CHECKPOINT;
DBCC DROPCLEANBUFFERS;
DBCC FREEPROCCACHE;

$$x_{avg} = \frac{x_1 + x_2 + \ldots + x_N}{N}$$

$$R = x_{max} - x_{min}$$

$$\Delta x = \frac{R}{2} = \frac{x_{max} - x_{min}}{2}$$

$$\Delta x_{avg} = \frac{\Delta x}{\sqrt{N}}$$

$$x_m = x_{avg} \pm \Delta x_{avg}$$

	Originaltabell	Partition
	285.3	279.1
	174	216
	87	108
	22.5	27.9
	(285 ± 23) ms	(279 ± 28) ms

Bilaga 3

Exekveringstid mot originaltabell [ms]	Exekveringstid mot partition [ms]
51	47
50	46
51	48
50	49
50	47
50	48
51	47
51	47
51	47
51	48

Resultat	
Originaltabell	Partition
(50,6 ± 0,1) ms	(47,4 ± 0,4) ms

SQL 2

top 5000
9x left join
159 dagars intervall över samma år

$$x_{avg} = \frac{x_1 + x_2 + \ldots + x_N}{N}$$

$$R = x_{max} - x_{min}$$

$$\Delta x = \frac{R}{2} = \frac{x_{max} - x_{min}}{2}$$

$$\Delta x_{avg} = \frac{\Delta x}{\sqrt{N}}$$

$$x_m = x_{avg} \pm \Delta x_{avg}$$

	Originaltabell	Partition
x_{avg}	50.6	47.4
R	1	3
Δx	0.5	1.5
Δx_{avg}	0.1	0.4
x_m	(50,6 ± 0,1) ms	(47,4 ± 0,4) ms

Bilaga 4

Exekveringstid mot originaltabell [ms]	Exekveringstid mot partition [ms]
	270
	242
	326
	272
	265
	259
	371
	288
	322
	216
	311
	218
	273
	306
	295

Resultat	
Originaltabell	Partition
	(282 ± 20) ms

SQL 2

top 5000
9x left join
159 dagars intervall över samma år

OBS! Partitionens index ombyggt med fillfactor = 89

DBCC-kommandon

CHECKPOINT;
DBCC DROPCLEANBUFFERS;

$$x_{avg} = \frac{x_1 + x_2 + \ldots + x_N}{N}$$

$$R = x_{max} - x_{min}$$

$$\Delta x = \frac{R}{2} = \frac{x_{max} - x_{min}}{2}$$

$$\Delta x_{avg} = \frac{\Delta x}{\sqrt{N}}$$

$$x_m = x_{avg} \pm \Delta x_{avg}$$

Originaltabell	Partition
	282.3
	155
	77.5
	20.0
	(282 ± 20) ms

Bilaga 5

Exekveringstid mot originaltabell [ms]	Exekveringstid mot partition [ms]
51	82
50	82
50	88
50	82
50	90
53	81
51	82
74	82
51	83
51	83

Resultat	
Originaltabell	Partition
(53 ± 3) ms	(84 ± 1) ms

SQL 2

top 5000
9x left join
10 dagars intervall mellan dec 2012 och jan 2013

$$x_{avg} = \frac{x_1 + x_2 + \ldots + x_N}{N}$$

$$R = x_{max} - x_{min}$$

$$\Delta x = \frac{R}{2} = \frac{x_{max} - x_{min}}{2}$$

$$\Delta x_{avg} = \frac{\Delta x}{\sqrt{N}}$$

$$x_m = x_{avg} \pm \Delta x_{avg}$$

	Originaltabell	Partition
x_{avg}	53.1	83.5
R	24	9
Δx	12	4.5
Δx_{avg}	3.1	1.2
x_m	(53 ± 3) ms	(84 ± 1) ms

Bilaga 6

Exekveringstid mot originaltabell [ms]	Exekveringstid mot partition [ms]
248	310
187	248
431	273
290	313
300	313
279	326
239	272
315	293
266	239
197	227
211	352
219	254
421	327
243	295
366	345

Resultat	
Originaltabell	Partition
(285 ± 23) ms	(279 ± 28) ms

SQL 2

top 5000
9x left join
10 dagars intervall mellan december 2012 och januari 2013

DBCC-kommandon

CHECKPOINT;
DBCC DROPCLEANBUFFERS;
DBCC FREEPROCCACHE;

$$x_{avg} = \frac{x_1 + x_2 + \ldots + x_N}{N}$$

$$R = x_{max} - x_{min}$$

$$\Delta x = \frac{R}{2} = \frac{x_{max} - x_{min}}{2}$$

$$\Delta x_{avg} = \frac{\Delta x}{\sqrt{N}}$$

$$x_m = x_{avg} \pm \Delta x_{avg}$$

	Originaltabell	Partition
x_{avg}	280.8	292.5
R	244	125
Δx	122	62.5
Δx_{avg}	31.5	16.1
x_m	(285 ± 23) ms	(279 ± 28) ms

Bilaga 7

Försök	Bilaga	Exekv.tid mot originaltabell	Exekv.tid mot partition	SQL	Kall körning
1	1	(86 ± 9) ms	(85 ± 4) ms	1	Ja
2	2	(285 ± 22) ms	(279 ± 28) ms	2	Ja
3	3	(50,6 ± 0,1) ms	(47,4 ± 0,4) ms	2	Nej
4	4		(282 ± 20) ms	2	Ja
5	5	(53,1 ± 3,8) ms	(83,5 ± 1,4) ms	2	Nej
6	6	(281 ± 32) ms	(292 ± 16) ms	2	Ja