Linköping Studies in Science and Technology
Dissertations, No. 000001

ISRN: LIU-IDA/LITH-EX-G–24/049—SE

Undersökning av prestandaförbättring i geografiska databaser

av

Gustav Wibom & Pontus Kragmo

Akademisk avhandling
som för framläggande av teknologie doktorsexamen i databasteknik vid Linköpings universitet kommer att offentligt försvaras i sal Herbert Simon, hus B, campus Valla, den 14 juni 2024 kl. 13:15

Fakultetsopponent
Student Evan Hedbor & Oliver Waldemar
Linköping Universitet
Sverige

Abstract

Uppdraget för denna undersökning var att ta fram lösningar för att öka prestandan i en existerande databas som hanterar stora mängder av geografisk data. Databasen sparade blixtar och användare och användes för att notifiera användare om blixtar som slår ner nära deras position, vilket kräver frekventa uppslagningar i databasen för att matcha användares position med positionen av blixtnedslag. Vår metod var att sammanställa 4 olika lösningar: Rutnät, Polygoner, Caching och Indexering. Dessa lösningar utvärderades sedan med hjälp av en egenbyggd testmiljö med avseende på exekveringstid samt CPU- och minnesanvänding. Resultaten från dessa tester jämfördes med resultaten som den originella databasen fick och kunde på så sätt jämföra de olika metoderna för att se vilken av dom som gav störst ökning av prestandan. Det var flera av lösningarna som gav en märkbar ökning av prestandan men en kombination av polygoner och indexering gav den största ökningen av prestandan. Med resultaten från testningen så är det möjligt att få en inblick i hur olika metoder kan användas

,

för att öka prestandan i en databas som hanterar geografisk data och se vilken påverkan de faktiskt har under användning.

Avdelningen Software and Systems
Department of Computer and Information Science
Linköpings universitet, SE–581 83 Linköping, Sweden

Undersökning av prestandaförbättring i geografiska databaser

Pontus Kragmo
ponkr400@student.liu.se

Gustav Wibom
guswi424@student.liu.se

INLEDNING

Varje år slår det ner cirka 150 000 blixtar i Sverige över hela landet [7]. Är du en mark- eller skogsägare kan dessa åskoväder vara förödande om de leder till bränder. Därför är det bra med ett system som kan meddela om blixtnedslag i närheten av dig eller din mark. Ett system som gör detta kallas för Blixtvakt[1] som utvecklas av företaget Zenon[2], där du kan bli informerad om blixtar i närheten av dig genom din telefons position. För att göra detta får dom in GPS-positionen av blixtnedslag och söker igenom en databas över användare som är inom en viss radie om blixtnedslaget.

Vid åskoväder kan det slå ned upp mot 10 000 blixtar under en kortare period. Blixtvakt får in stora mängder data under liknande åskoväder vilket leder till ökad belastning på deras databas. I nuläget behöver Zenon, för varje blixt som slår ner, söka igenom alla aktiva användare i sin databas för att kontrollera om de är tillräckligt nära för att skicka ut en notifikation. Denna sökning genom databasen tar lång tid speciellt eftersom deras användare blir mer aktiva under större kraftiga åskoväder. Den långa söktiden leder till att blixtar slår ner snabbare än Blixtvakt kan hantera vilket skapar en kö av blixtnedslag som byggs upp desto längre åskovädret pågår, detta leder till mycket långa svarstider för användarna och en stor belastning på databasen.

Prestandan av databaser kan variera beroende på hur väl optimerade de är i många olika områden. Skillnader i ändringar av data via queries kan orsaka stora prestanda problem beroende på dess optimering [11]. Mängden data och dess struktur kan leda till både förbättrad eller försämrad prestanda. Indexering i databaser kan underlätta sökning men kräver extra utrymme. Det krävs också förståelse för hur det ska användas för att undvika att indexering skulle leda till sämre prestanda [9]. Alla dessa saker påverkar prestandan på databasen och det krävs kunskap om hur databaser fungerar för att minska ineffektivitet som uppstår.

[1] https://www.blixtvakt.com/
[2] https://www.zenon.se/

SYFTE

Målet med detta examensarbete är att replikera en existerande databas och utvärdera olika lösningar för att öka databas prestanda, primärt för uppslagning av positioner av blixtnedslag och användare. Lösningarna kommer att utvärderas på olika aspekter såsom minnesanvändning och prestanda.

FRÅGESTÄLLNING

Vi kommer testa olika metoder som kan användas för att förbättra databasens prestanda vid uppslagning. Utifrån dessa metoder kommer vi sedan att utforska varje metod för sig och utvärdera de med avseende till:

1. Hur skiljer sig de olika metoderna i avsikt på minnesanvändning, CPU-användning och exekveringstid?
2. Förbättrar någon av metoderna den nuvarande prestandan för databasen? Vilken metod visade störst ökad prestanda?

AVGRÄNSNING

Detta projekt kommer endast att undersöka och implementera lösningar relaterade till databas prestanda. Projektet kommer att ta fram olika lösningar och jämföra resultaten av de olika lösningarna. Hanteringen av att informera användaren om blixtnedslag kommer ej att undersökas i projektet.

BAKGRUND

I detta kapitel tas olika relevanta begrepp upp inom området databashantering.

Queries

Databaser ser alla olika ut, men den generella beskrivningen är en organiserad samling av information eller data. Även om databaser är olika finns det några tekniker som används av de flesta när det kommer till att förbättra prestandan. Det vanligaste sättet att hämta eller manipulera information i en databas kallas för quering. Queries skrivs med en specifik syntax, den mest använda är SQL, där du med en textsträng anger vilken information du vill hämta eller hur du vill ändra informationen. Men databaser kan vara stora, med flera hundratusen kolumner data, och är begränsade av sin egen hårdvara vilket betyder att dåligt skrivna queries kan leda till försämrad prestanda [6]. Att en query är dålig kan bero på många olika saker, Stefan Brass tar upp många exempel i sitt arbete kring semantiska fel i SQL [2]. Ett exempel är att det är bättre att undvika subqueries och istället använda sig av JOIN eller andra tekniker, ett annat exempel han tar upp är att querien kan innehålla för många wildcards vilket betyder att den oftast hämtar data som inte är behövd. Men att optimera alla queries kan däremot vara väldigt tidskrävande och ineffektivt, därför bör man analysera

queries innan optimering. Analyseringen kontrollerar flera mätvärden som utförandetid, logiska samt fysiska läsningar och CPU-tid som alla påverka queries prestanda på ett negativt sätt.

Caching

I databas-världen hanteras det som sagt stora mängder data och det kan ofta användas många queries för att hämta eller manipulera den datan som finns sparad i databasen. När det kommer in många queries till databasen skapas det större arbetsmängd vilket kan sakta ner svarstiden på queries. För att undvika detta behövs sätt att minska svarstiden för queries. En lösning för detta är att använda sig utav caching. Caching inom databasteknik innebär att man sparar en viss andel queries i arbetsminnet istället för att endast ha allt på disk. Detta gör att den sparade datan i arbetsminnet går betydligt snabbare att nå än datan på disk eftersom att arbetsminnet är flera gånger snabbare än disken [4]. Då arbetsminnet inte har särskilt stor kapacitet brukar det endast sparas data i arbetsminnet som ofta efterfrågas eller som nyligen efterfrågats. När en query kommer in till databasservern kommer den först kolla igenom arbetsminnet och om den hittar en träff returneras värdet direkt från minnet utan att behöva leta igenom disken. Eftersom att många queries kan vara samma eller liknande queries kan detta minska tiden det tar att hitta resultat från queries [1, 4].

PostgreSQL

PostgreSQL[3] är ett open-source *Object-Relational Database Management System* (ORDBMS) och är ett av de mest använda systemen globalt inom databashantering [3, 10]. PostgreSQL använder sig av bland annat SQL för hantering av data via queries. Det som gör PostgreSQL till ett populärt databas system är dess många olika användbara tekniker såsom *Transactional Processing, Multiple Language Processing* och *Unique Data Types* [3]. Fördelen med att PostgreSQL är open-source är att användare kan bidra till utvecklingen av systemet. Detta har lett till utvecklingen av PostGIS[4].

PostGIS

PostGIS är ett plugin till PostgreSQL som lägger till möjligheten att hantera geografiska positioner, punkter och polygoner som egna datatyper. De två mest förekommande datatyperna är *geometry*[5] och *geography*[6]. Det som skiljer geography från geometry är att deras positioner baseras på olika saker. Geography baseras på jorden och dess bredare form kring ekvatorn medan geometry baseras på en perfekt sfär.

Med dessa två datatyper medföljer även så kallade *Spacial Functions*. Spacial Functions kan hantera och göra olika mätningar av dessa datatyper som att beräkna avstånd mellan punkter, hitta överlappande areor m.m. Beroende på om man använder geography eller geometry måste dessa funktioner beräknas på olika sätt då de baseras på två olika system. Geometry använder ett Kartesiskt koordinatsystem medan geography använder ett Sfäriskt koordinatsystem.

Indexering

När en SELECT query skickas till en databas måste databasen leta igenom ett visst antal rader i de tabeller som man har specifierat i sin query. I större databaser kan antalet rader som behövs sökas igenom bli väldigt många, upp mot flera miljoner. I en vanlig PostgreSQL-databas görs detta som standard genom en linjär sökning[8]. Ifall du ska hämta en rad i tabell som har N mängd rader behövs i genomsnitt läsa igenom (N+1)/2 rader. Ett bättre sätt att söka igenom en databas är genom en binär sökning som kräver läsningen av Log(N) rader i genomsnitt. För att ändra sättet att söka igenom databasen på använder man indexering. I fallet av vår databas som hanterar geografisk data är vi i huvudsak intresserade av Generalized Search Tree index, eller GiST index som tillsammans med postgis biblioteket kan användas för att förbättra söktiden i databaser som innehåller geografisk data[7].

GiST indexering fungerar speciellt bra för geografisk data då vanlig indexering inte kan hantera den mer komplexa datan som används. GiST indexering sorterar geometries och geographies utifrån deras *bounding box*, vilket är den minsta möjliga rektangel som omsluter figuren[8]. Om man sedan vill till exempel söka efter en geometry som är inuti någon annan geometry i en tabell kommer den att dela upp sökningen i två delar. Först att leta efter alla bounding boxes i tabellen och jämföra dom istället för att jämföra polygonerna mot varandra då dessa rektanglar är enklare att jämföra. Sedan kommer den att hämta ut alla bounding boxes som angränsade den geometrien och endast beräkna de faktiska polygonerna för just dem eftersom att denna beräkning är mycket dyrare.

Prestanda

Databaser måste kunna effektivt hantera och manipulera stora mängder data vilket ställer ett stort krav på prestanda och optimering för att de ska fungera på en hög nivå. Det som vi kommer mäta som prestanda i databasen är minnesanvändning, arbetsbelastning och söktid. Söktid syftar på hur snabbt en query blir avklarad, alltså hur lång tid det tar för en query från att du skickar den till att du får ut informationen du frågade efter. Med minnesanvändning menar vi hur mycket arbetsminne som krävs av systemet. Arbetsbelastning är hur mycket arbete databasen behöver utföra över en viss tid, exempelvis hur många queries behöver köras eller hur mycket data behöver bli manipulerad.

RELATERADE ARBETEN

Databasprestanda är ett område med mycket forskning, det har utförts en stor mängd arbeten för att utforska tekniker för att öka prestanda. Med alla dessa arbeten har det dykt upp en stor mängd olika sätt att förbättra prestandan. Veronika Šalgová testade i sitt arbete att använda indexering med partitionering för att förbättra prestandan av en databas med data från IMDB-hemsidan [12]. De hade en databas med väldigt mycket data

[3] https://www.postgresql.org/

[4] https://postgis.net/

[5] https://www.postgis.net/workshops/postgis-intro/geometries.html

[6] https://www.postgis.net/workshops/postgis-intro/geography.html

[7] https://postgis.net/workshops/postgis-intro/indexing.html

där de testade olika queries först utan indexering och sedan med indexering. Arbetet kom fram till att indexering kunde ha en positiv inverkan på query prestandan, men det kunde även öka kostnaden i vissa fall. De noterar däremot att prestandan med indexering kan avvika beroende på vilka queries och vilka sorters index som används, samt att icke-indexerade databasen faktist var snabbare när vissa queries skulle hanteras. Chunxia sammanfattar en konferens från 2016 där indexering i SQL-databaser diskuteras [9]. Där kom de fram till att indexering inte har en direkt koppling till ökad prestanda i databaser, utan att det kan leda till att prestandan minskar vid vissa fall. För att undvika detta presenterar dom ett antal principer man bör hålla sig till för indexering ska leda till en ökning av prestanda. Indexering kan alltså vara ett sätt att öka prestandan av sin databas men det krävs noggrann hållning till vissa riktlinjer för att undvika att man istället minskar prestandan.

Richard Hankins utförde tester relaterade till arbetsbelastning i en Oracle databas som kördes på ett multi-processors system [5]. Databasen var specialiserad för att ta emot transaktioner från tusentals kunder som kunde skapa och ändra beställningar. För att mäta arbetsbelastningen i databasen använde de sig utav sina egna mått för genomsnittliga antalet instruktioner per transaktion (IPX) och genomsnittliga antalet cykler per instruktion (CPI). Deras resultat visade på hur högre värden på IPX och CPI gav en högre arbetsbelastning på databasen. Då den testade databasen var fokuserad på transaktioner var dessa mått den största faktorn till prestanda men i databaser som inte är lika inriktade på kundtransaktioner kan dessa mått ha mindre inverkningar på prestandan.

METOD

I det här kapitlet beskrivs de olika metoder som använts för att förbättra olika aspekter av prestanda. Kapitlet börjar med en generell beskrivning av vårt tillvägagångsätt och följer senare in på de fyra olika lösningar som implementerades. Slutligen beskrivs de tester som utfördes.

Replikering av databasen

För att kunna utvärdera de metoder som ska användas så replikerades deras existerande PostgreSQL-databas för att sedan kunna implementera lösningar och köra tester. Databasen bestod utav två olika tabeller: "Devices" och "Flashes". "Devices" innehöll information om användarna och "Flashes" innehöll information om alla historiskt observerade inkomna blixtar (se Figur 1, 2). I alla stycken här efter relaterar flashes till blixtar och devices till användare. En PostgreSQL databas med två olika tabeller skapades där tabellerna var: "`testing_flashes`" och "`testing_devices`". Dessa två tabeller strukturerades utifrån de två tabellerna i den existerande databasen och fylldes sedan upp med exakt samma data för att skapa en exakt replika av databasen. Detta resulterade i en databas där "testning_devices" innehöll 400 000 rader och "`testing_flashes`" innehöll 1,7 miljoner rader.

Riktlinjer för Queries

Under arbetet med att förbättra databasens prestanda behövdes en del egna queries skapas för att använda olika metoder som tas upp senare. För att se till att dessa queries var välskrivna och inte påverkade de resultat som testerna returnerade följdes de 37 punkterna som Brass tar upp som semantiska fel i sin text [2]. Extra vikt sattes på att kontrollera de 8 punkter som Brass, efter en undersökning, kommit fram till var vanligast att stötta på:

1. Many Duplicates - *Queries that give give a lot of duplicate answers, often not using the "distinct" keyword without reason.*
2. Missing Join Conditions - *Joining tables without any conditions.*
3. Inconsistent Conditions - *Queries that are empty depending on database states, excluding empty relations.*
4. Unnecessary Joins - *If the variable in the join condition is the only one needed and it is the same in both tables.*
5. Unused Tuple Vars - *Retrieving information from the database that is not used.*
6. Singleton Groups - *Groups that consist of only a single row.*
7. Implied, tautological, or inconsistent subconditions - *Subconditions that can be replaced with TRUE or FALSE.*
8. Always Identical Tuple Vars - *If the key attributes of two tuple variables are the same, the two tuple variables always point to the same tuple.*

Att följa dessa riktlinjer ledde såklart inte till helt felfria queries men det hjälpte till att minimera risken av att de queries som skapades påverkade databasprestandan negativt.

Baslösning

Det orginella systemet som Zenon använde vid blixtnedslag var enkelt. När det slog ner en blixt lades den till i flashes-tabellen och kördes en SQL-query som hittar alla närliggande användare som syns i det underliggande kodstycket. Querien hämtar ut alla devices som både är inom 50 000 meter från blixten och är inom dess egna sparade distans i databasen (se Figur 1).

```
SELECT id, device_token, distance, last_updated, location
     FROM testing_devices AS d WHERE
     ST_DWithin(d.location, ST_SetSRID(ST_Point(:lon:,
     :lat:), 4326), 50000, false) AND
     ST_DWithin(d.location, ST_SetSRID(ST_Point(:lon:,
     :lat:), 4326), d.distance, false);
```

Denna query användes för att hitta närliggande devices från en flash. Querien innehåller funktionen `ST_DWithin()`[8] som tar in två olika geographies, G1 och G2, en distans i meter samt en boolean. Det funktionen gör är att den beräknar ifall G2 är helt inom den angivna distansen ifrån G1 och returnerar en boolean. Variablerna :lon: och :lat: i koden motsvarar blixtens longitud och latitud.

Eftersom att devices är riktiga användare kommer det även in uppdateringar om devices positioner när användare rör på sig. Dessa updateringar hanteras av en vanligt UPDATE funktion:

[8] https://postgis.net/docs/ST_DWithin.html

	id [PK] integer	device_token text	distance integer	last_updated timestamp without time zone	feature_flags text	location geography	zone integer	polygon geography
1	12	djmUyTM8R1SstX2hg...	100	2021-05-21 15:19:08.798	{Ads}	0101000020E6100...	90	0103000020E61000000100000
2	30	ew-XGX_JS8SD_McE...	1000	2024-03-22 06:46:41.38	{Ads}	0101000020E6100...	90	0103000020E61000000100000
3	53	fXkBiNsjAEgRhlKwOb...	1000	2021-05-27 10:05:11.239	{Ads}	0101000020E6100...	90	0103000020E61000000100000
4	62	cMHmdAx8S0OHyM8...	1000	2021-05-27 13:45:39.862	{Ads}	0101000020E6100...	[null]	0103000020E61000000100000
5	64	cdCIRn_8R-yT1LFSUB...	1000	2021-05-27 13:45:51.38	{Ads}	0101000020E6100...	[null]	0103000020E61000000100000
6	65	coHQ5A9SSqGIhvKD7...	1000	2021-05-27 13:47:42.399	{Ads}	0101000020E6100...	[null]	0103000020E61000000100000
7	66	fduKaWlfQpODY_9q8...	1000	2021-05-27 13:50:01.988	{Ads}	0101000020E6100...	[null]	0103000020E61000000100000
8	70	fh0uzm0eezU:APA91b...	47800	2021-07-14 13:54:09.411	{Ads}	0101000020E6100...	90	0103000020E61000000100000
9	73	et3ftJYsSVOXi3JJu_C...	15000	2024-02-24 22:46:45.165	{Ads}	0101000020E6100...	90	0103000020E61000000100000
10	83	cQY2AEuoSsGMSivici...	1000	2021-05-28 09:10:25.346	{Ads}	0101000020E6100...	[null]	0103000020E61000000100000

Figure 1. Bild över `testing_devices`. Varje device har ett id och en device_token som identifierar den. De har även en kolumn för last_updated som är den tid som den användaren har uppdaterat sin position som representeras av kolumnen location. kolumnen feature_flags är inte relaterat till testerna och de tre slutliga kolumnerna: distance, zone och polygon tas upp i Lösning 1: Rutnät och Lösning 2: Positionering med polygoner

	id [PK] integer	lat double precision	lon double precision	proj text	received_time timestamp without time zone	flash_time timestamp without time zone	peak_current integer	cloud_indicator integer
1	1	60.0519	17.2274	EPSG:4326	2021-05-21 12:32:32.865	2021-05-21 12:32:15.498	11	1
2	2	60.0785	17.3989	EPSG:4326	2021-05-21 12:33:11.39	2021-05-21 12:32:54.449	6	1
3	3	55.0418	18.099	EPSG:4326	2021-05-21 12:33:42.252	2021-05-21 12:33:24.927	-16	1
4	4	60.034	17.3301	EPSG:4326	2021-05-21 12:33:45.539	2021-05-21 12:33:29.06	-14	0
5	5	60.0622	17.2128	EPSG:4326	2021-05-21 12:33:45.562	2021-05-21 12:33:29.06	9	1
6	6	56.5012	12.9705	EPSG:4326	2021-05-21 12:34:38.793	2021-05-21 12:34:20.998	-12	0
7	7	59.9638	17.3154	EPSG:4326	2021-05-21 12:34:43.89	2021-05-21 12:34:26.989	7	1
8	8	55.0575	18.3943	EPSG:4326	2021-05-21 12:34:49.961	2021-05-21 12:34:32.775	-31	1
9	9	60.0273	17.3126	EPSG:4326	2021-05-21 12:35:14.972	2021-05-21 12:34:57.849	8	1
10	10	60.146	16.9269	EPSG:4326	2021-05-21 12:35:35.881	2021-05-21 12:35:18.757	6	1

Figure 2. Bild över `testing_flashes`. Varje flash har ett unikt id den representeras av. kolumnerna lat och lon sparar de koordinater som blixten slog ned i och proj sparar vilken typ av koordinatsystem som används. Det finns två kolumner för olika tider, `received_time` och `flash_time`. `Received_time` innehåller tidpunkten då databasen tog emot en ny blixt medan flash_time är den faktiska tidpunkt som blixten slog ned. De två sista kolumnerna peak_current och cloud_indicator är extra väder data som inte kommer användas i testerna.

```
UPDATE testing_devices SET
     location=ST_SetSRID(ST_Point(:lon:, :lat:), 4326)
     WHERE id=:id:
```

När en uppdatering för devices kommer in till databasen körs denna query för att uppdatera positionen för en device. Här representerar :lon: och :lat: de nya värdena för longitud och latitud för användaren med :id: som id.

Lösningar

Här utvärderas lösningarna som användes i undersökningen till problemet för att öka prestandan.

Lösning 1: Rutnät

Eftersom det finns över 400 000 användare som använder blixtvakt blir det en stor mängd användare att söka igenom när en blixt slår ner. Alla dessa användare är sparade i en stor tabell som hanterar alla användare i databasen. För att minska antalet användare som behövs sökas igenom vid varje blixt kan en rutnäts-lösning användas. Först skapades ett rutnät över Sverige med sidostorlek på 50 kilometer för varje ruta och sedan ges varje ruta en kod baserat på dess position, A-L för x-axeln och 1-36 för y-axeln, varje ruta har då en kod som exempelvis ser ut såhär: `E25` eller `B3`. Dessa rutor sparas i en tabell där varje rad i tabellen har 3 kolumner: rutans geografiska position (`zone`), rutans kod (`zone_cord`) och en lista av koderna för de 8 rutorna runt sig (`close_zones`). En kolumn i devices-tabellen läggs även till där vilken ruta som användaren befinner sig i sparas, vilket även betyder att rutan som användaren befinner sig i måste kontrolleras varje gång den uppdaterar sin position. När en blixt sedan slår ner kollas först vilken ruta den slog ner i. Sedan, istället för att söka igenom alla användare efter matchningar, kontrolleras vilka användare som befinner sig i samma ruta som blixten slog ner i samt de 8 närliggande rutorna och då endast behöva leta efter matchningar med dem användarna. Detta ger en betydligt mindre mängd användare att leta igenom. Denna lösning använder sig av två olika queries:

```
SELECT id, close_zones FROM exjobb.zones WHERE
ST_Within(ST_SetSRID(ST_Point(:lon:, :lat:), 4326), zone);
```

Den första querien kontrollerar vilken ruta blixtnedslaget skedde i. Den använder funktionen `ST_Within()`[9] som tar in två geometries, G1 och G2, och kontrollerar ifall G1 är inuti G2. G1 är i detta fall blixten och G2 är rutan vilket betyder att vid en matchning identifieras vilken ruta som blixten slog ner i. Vid en matchning plockas den identifierade rutan ut tillsammans med de 8 närliggande rutorna för att använda dom i nästa query.

```
SELECT id, device_token, distance, last_updated, location
     FROM exjobb.devices AS d WHERE zone IN :close_zones:
     AND ST_DWithin(d.location,
     ST_SetSRID(ST_Point(:lon:, :lat:), 4326), 50000,
     false) AND ST_DWithin(d.location,
     ST_SetSRID(ST_Point(:lon:, :lat:), 4326),
     d.distance, false);
```

[9] https://postgis.net/docs/ST_Within.html

Denna query hämtar ut de devices som är påverkade av blixtnedslaget. Den börjar med att filtrera bort alla devices som inte är i någon av de 9 zonerna som är nära blixtens position vilket kraftigt minskar mängden devices som söks igenom. Efter det söker den igenom de kvarvarande devices genom användning av två `ST_DWithin()` funktioner, likt baslösningen.

Lösning 2: Positionering med polygoner

När det kom till just beräkningen av distansen mellan devices och flashes utvärderades två olika algoritmer inom postGIS biblioteket: `ST_DWithin()` och `ST_Within()`. Dessa funktioner fungerar på två snarlika sätt. `ST_Within()` tar in 2 geometries, `A` och `B`, och kollar ifall geometry `A` är inuti geometry `B` och returnerar sant. `ST_DWithin()` tar in 2 geographies, `A` och `B`, och en distance som är en distans i meter. Funktionen tar sedan och kollar ifall distansen mellan `A` och `B` är mindre än den givna distansen och returnerar sant.

Då både devices och flashes representeras som individuella punkter kommer `ST_Within()` inte kunna användas då en device och en flash skulle behöva ha exakt samma position för att den ena skulle kunna vara "inuti" den andra. För att jämföra `ST_Within()` med `ST_DWithin()` lades en kolumn "polygon" till i `testing_devices`-tabellen som sparar en cirkel-formad geography av positionen på en device istället för en punkt. Denna cirkel har samma värde för sin radie som värdet sparat i distance-kolumnen. Detta gör att vi endast behöver kolla ifall en blixt slår ner inuti den cirkeln vilket ledde till denna underliggande query:

```
SELECT id, device_token, distance, last_updated, location
     FROM exjobb.testing_devices AS d WHERE
     ST_Within(ST_SetSRID(ST_Point(:lon:, :lat:),
     4326)::geometry, d.polygon::geometry);
```

Då `ST_Within()` tar in två geometries behövdes `d.polygon` och blixtens position konverteras till geometries från geographies.

Eftersom att en kolumn lades till i `testing_devices`-tabellen för devices positioner som cirklar behövdes en extra query för att uppdatera den nya kolumnen:

```
UPDATE testing_devices SET polygon =
     ST_Buffer(ST_SetSRID(ST_Point(:lon:, :lat:),
     4326)::geography, distance, \'quad_segs=:value:\')
     WHERE id = :id:
```

Med hjälp av ST_Buffer() skapas ett område runt den specifierade positionen med en radie som representeras av distance i querien. ST_Buffer() returnerar dock inte en perfekt cirkel utan den returnerar en månghörning. Med hjälp av parametern quad_segs definieras hur många hörn polygonen ska ha, desto flera hörn den har, desto närmare en cirkel blir polygonen. Det som behövs ta hänsyn till är att flera hörn på polygonen kommer att göra att den tar mer utrymme, då den representeras med med hörnen som koordinatpunkter. Dubblar man hörnen så dubblar man minnet den tar upp. Standard värdet för quad_segs är 8 och 3 olika värden för quad_segs testades: 8, 16 och 32, vilket nämns mer i Testning.

Lösning 3: Caching
För att förbättra söktiden för att en query ska bli klar användes en lösning som liknar caching för att minska antalet spatial functions som behövs utföras för att en query ska bli klar. Anledningen att en normal caching lösning inte kan implementeras är att det i stort sett aldrig kommer bli en träff som matchar helt en annan blixts position. Det som istället görs är att en ny tabell skapas, recent_flashes, som agerar som ett cache genom att spara alla blixtar som slagit ner den senaste minuten (se Figur 3).

```
SELECT devices FROM exjobb.recent_flashes WHERE
     ST_DWithin(ST_SetSRID(ST_Point(lon, lat), 4326),
     ST_SetSRID(ST_Point(:lon:, :lat:), 4326), 500,
     false) LIMIT 1;
```

Den överstående querien används sedan för att leta efter flashes som har slagit ner inom en 500 meter radie från den nya flashen. Ifall det blir en matchning returneras värdet i devices-kolumnen som innehåller alla id:n för devices i närheten. Om ingen flash är inom 500 meter går den över till att använda baslösningen.

Lösning 4: Indexering
Förbättring av söktiden i en tabell implementerades genom att använda indexering i tabellerna. För att använda detta skapades ett GiST index på location-kolumnen i `testing_devices` eftersom att det är den kolumnen som kommer att sökas igenom när den söker efter närliggande devices vid en ny flash.

```
CREATE INDEX IF NOT EXISTS
     testing_devices_location_index_geog ON
     testing_devices USING gist (((location)::geography))
     TABLESPACE pg_default;
```

Detta ledde då till att när databasen söks igenom efter användare som kan matcha med blixtar som slår ner kan en binär sökning användas istället för en linjär sökning. Denna lösning kräver även inte någon ändring på vilka queries som används då detta sker internt i databasen. Ett index skapades även för polygon-kolumnen i `testing_devices`:

```
CREATE INDEX IF NOT EXISTS
     testing_devices_polygon_index_geog ON
     testing_devices USING gist (((polygon)::geometry))
     TABLESPACE pg_default;
```

Detta andra index implementerades för att använda med Lösning 2: Positionering med polygoner, vilket tas upp mer i nedanstående stycke.

Testning
För att testa de olika lösningarna sattes det upp en testningsmiljö i Python som tillsammans med databasen som beskrevs tidigare är sättet som prestandan påverkan av våra lösningar undersöks. Testet är uppbyggt som en simulation av en specificerad tidpunkt eller dag där de inkommande blixtarna återskapas baserat på när och var de faktiska blixtarna slog ner i realtid. Då ingen historisk data över devices positionsuppdateringar fanns sattes mängden device-uppdateringar till antalet flashes i testerna så att antalet anslutningar till databasen för flashes och devices skulle vara lika.

Innan testet startar anges vilket datum och tid som ska simuleras och även hur länge testet ska pågå. För alla tester valdes att simulera 10-minuters tester. När testet börjar plockas alla blixtar ut som slog ner i det angivna datumet och tidsintervallet från databasen. Dessa blixtar läggs sedan in i en scheduler från scheduler biblioteket i Python[10]. Schedulern gör det möjligt att simulera datan som att blixtarna slår ner i realtid och att blixtarna kommer in med samma mellanrum som de gjorde i verkligheten. Lika många uppdateringar av devices lades till i schedulern fast dessa uppdateringar gavs en slumpmässig tid då inget tidigare klockslag fanns för dem.

Efter detta körs testet i det tidsintervall som angavs och vid varje tidpunkt som en blixt slår ner eller en device uppdaterar sin position startas en ny tråd som skapar en ny anslutning mot databasen med hjälp av `psycopg2.connect()` från psycopg2 biblioteket. Innan queries sedan börjar skickas mot databasen noteras startiden med `time.process_time_ns()` från time biblioteket som sedan kommer att användas igen för att få fram exekveringstiden. `cur.execute()` används sedan för att skicka queries mot databasen för att matcha blixtens position mot positionerna av användare. Beroende på vilken lösning som testas skickas de olika tidigare nämna queriesen och hittar alla användare som är inom rätt avstånd från blixten för att få en notifikation om den. När den har hittat alla passande användare noteras slutiden med hjälp av `time.process_time_ns()` igen och hämtas differensen ut mellan startiden och slutiden för att få fram exekveringstiden, vilket är hur lång tid det tar för databasen att returnera ett resultat. Dessa tider skrivs ut samt sparas för att i slutet av testet kunna få ut den genomsnittliga exekveringstiden över alla blixtar.

När det var dags att börja med den slutgiltiga testningen av lösningarna sattes en extern databas upp för att undvika att själva testprogrammet skulle påverka databasens prestanda. Detta gjordes genom tjänsten Aiven[11] därett system med 8 GB RAM, 175 GB lagringsutrymme och 4 CPUs kunde sättas upp. Genom denna tjänst kunde även enkelt minnes- och CPU-användning som krävdes av de olika lösningarna mätas av under testerna. För varje lösning kördes 5 stycken 10-minuters tester för att få fram en rimlig genomsnitts prestanda. Till att börja med kördes testerna för baslösningen för att få fram en baslinje för prestandan att jämnföras med. Efter det kördes tester för alla olika lösningar var för sig, men även olika kombinationer av lösningar.

RESULTAT
I det här kapitlet beskrivs och redovisas de resultat som testningen gav.

[10] https://docs.python.org/3/library/sched.html
[11] https://aiven.io/

	id [PK] integer	lat double precision	lon double precision	received_time timestamp without time zone	zone integer	devices integer[]
1	1	62.4331	13.1856	2024-05-13 15:48:41.373879	195	
2	2	60.3022	11.9671	2024-05-13 15:48:41.379879	134	
3	3	60.3498	12.4172	2024-05-13 15:48:41.345879	134	
4	4	60.3683	12.4079	2024-05-13 15:48:41.392879	134	
5	5	57.7793	13.2619	2024-05-13 15:48:42.478879	75	{496680,357115,370265,345288,472370,48209,125775,

Figure 3. Bild över simcache. Här sparas data från `testing_flashes` som slagit ner under senaste minuten. Kolumnen devices sparar även vilka devices som var i närheten när blixten slog ner.

	Flash (ms)	Device (ms)
Bas:	21,8376	4,6626
Rutnät:	12,3302	5,0770
Polygon:	96,5323	61,6806
Polygon(16):	78,6126	44,9947
Polygon(32):	140,0766	51,8819
Caching:	22,1694	5,2343
Index:	4,6517	4,2653
Polygon + Index:	2,6748	4,4414
Polygon(16) + Index:	4,0659	7,5757
Polygon(32) + Index:	4,5302	6,9962
Alla:	3,072	5,8870

Table 1. Genomsnittliga exekveringstiden för flashes och devices i millisekunder

	CPU Låg	CPU Hög
Bas:	51,76%	78,42%
Rutnät:	39,1%	63,18%
Polygon:	57,84%	83,38%
Polygon(16):	67,14%	92%
Polygon(32):	86,34%	100%
Caching:	51,88%	78,4%
Index:	15,34%	24,52%
Polygon + Index:	4,46%	6,5%
Polygon(16) + Index:	4,56%	6,46%
Polygon(32) + Index:	4,9%	7,14%
Alla:	6,9%	10,61%

Table 2. CPU användningen av de testade lösningarna och för en omodifierad databas

Exekveringstid

Flera av lösningarna har en minskning av den genomsnittliga exekveringstiden jämfört med baslösningen som kan ses i tabell 1. Indexlösningen samt alla kombinationer som innehöll index hade en minskning runt 80% av flash exekveringstiden jämfört med testerna mot den omodifierade databasen. Däremot hade den fristående polygonlösningen runt 300-500% ökning av exekveringstiden för flashes. Rutnätslösningen hade en minskning på 40% medans cachinglösningen hade nära inpå 0% påverkan.

Nästan alla lösningar hade däremot en ökning av device exekveringstiden, det var endast index samt den enkla kombinationen av polygon och index som såg en minskad exekveringstid runt 4-8%. De andra kombinationerna av polygon och index hade en ökning på ungefär 50% medan de fristående polygonlösningarna hade en kraftig ökning på ungefär 1000% av exekveringstiden. Caching- och rutnätslösningen hade en mindre påverkan med runt 10% ökning.

Det var endast två lösningar som hade en minskning av både flash och device exekveringstiden, vilket var indexlösningen och enkla kombinationen av index och polygonlösningen. De flesta lösningar hade minskning av flash exekveringstiden men en ökning av device exekveringstiden istället.

CPU-användning

CPU användningen av lösningarna är liknande resultaten av exekveringstiden och kan ses i tabell 2. Indexlösningen och alla lösningar där indexlösningen ingår har en skillnad av ungefär 80-90% minskad CPU användning jämfört med bas testerna. Rutnätslösningen hade mindre påverkan med ungefär 20% minskad användning medans cachinglösningen, likt exekveringstiden, hade en påverkan som låg på nästan 0%. Den enkla fristående polygonlösningen hade runt 10% ökad CPU användning och de mer komplicerade fristående polygonlösningarna ökade CPU användningen med 30-60% där den mest komplicerade polygonlösningen kom upp i så högt som 100% CPU användning.

	Minne Låg	Minne Hög
Bas:	9,04%	16,38%
Rutnät:	8,8%	15,42%
Polygon:	8,1%	17,3%
Polygon(16):	8,38%	20,7%
Polygon(32):	11,8%	58,96%
Caching:	8,62%	19,04%
Index:	10%	10%
Polygon + Index:	8,78%	10,06%
Polygon(16) + Index:	8,72%	10,2%
Polygon(32) + Index:	8,74%	10,66%
Alla:	10%	10%

Table 3. Minnes användningen av de testade lösningarna och för en omodifierad databas

Minnesanvändning

De flesta lösningarna hade en minskning av minnesanvändning som kan ses i tabell 3. Indexlösningen och alla lösningar som använde index hade återigen bästa resultaten med en minskning på ungefär 30% jämfört med bas testerna. Rutnätslösningen och cachinglösningen hade minimala skillnader mot bas testerna med runt 5% minskning respektive ökning av minnesanvändningen. Den simpla fristående polygonlösningen hade liknande resultat till cachinglösningen medans de två mer komplicerade polygonlösningarna presterade sämre med den mindre komplicerade av de två som hade en ökning

runt 20% medans den mest komplicerade hade en ökning på ungefär 200% jämfört med bastesterna.

DISKUSSION

Här diskuteras olika aspekter av metoden samt resultaten av testerna, vilka faktorer som ledde till dem resultaten och de slutsatser som kan dras utifrån resultaten.

Testning

Testningen av lösningar fungerade i stora drag som planerat, men det finns saker i efterhand som vi insett skulle kunnat ge testerna en bättre precision och ännu mer tillförlitlig data. Enligt vår bedömning finns det däremot inget som skulle invalidera datan vi har fått fram genom testerna eller göra den opålitlig, endast åtgärder som är bra att ha i åtanke vid jämförelse med annan data eller vid replikation och/eller förbättring av testningen.

Eftersom testningen var relativt tidsbegränsad beslutade vi att endast göra tester som var 10 minuter långa eftersom det gav oss möjligheten att köra varje test mer än 1 gång för att undvika avvikande data från att komma in i resultaten. Men det betyder också att lösningarna inte garanterat fungerar lika bra, eller dåligt, som dom ser ut att göra enligt den data som vi har fått fram om de används över en längre period med mer varierande densitet av blixtar. En lösning som kan ha drabbats av detta är cachinglösningen eftersom den beror på att blixtar måste ske i närheten av varandra för att den ska kunna användas på ett betydelsefullt sätt, vilket ett längre test skulle vara fördelaktigt för eftersom det då är större chans att situationer som dessa uppstår. Däremot gav den korta test tiden också oss möjligheten att upptäcka och notera denna begränsning som caching metoden har vilket kanske inte skulle vara uppenbart vid ett längre test.

Blixtdatan som vi använde för testningen skulle också kunna bli justerad för en mer nyanserad bedömning av lösningarna. I våra tester beslutade vi att använda den period av blixtdatan vi hade tillgång till där det slog ner mest blixtar under en timme, av data som sträckte sig över 3-4 år skulle detta då kunna ses som en ovanlig mängd blixtar. Detta kan ha lett till att dom lösningar som bättre hanterar koncentrerade blixtoväder fick fördelaktiga resultat i testningen, jämfört med lösningar som fungerar bättre under en lugnare period. För att undvika detta kunde vi ha gjort flera uppsättningar av tester där vi kör alla lösningar men vi ändrar blixtdatan som används så att vi får testdata som har både lugnare perioder och mer koncentrerade perioder av blixtar. På detta sätt skulle vi då kunna se om det påverkar lösningarna i någon större grad beroende på hur mycket aktivitet som de behöver hantera.

Eftersom vi hade begränsad data över devices jämfört med datan vi hade tillgång till över blixtnedslag blev vi tvungna att själva försöka simulera positionändringar på den devicedatan som vi använde under testningen. Detta betyder då att device delen av testningen kan skilja sig mycket mot hur det ser ut i verkligheten, specifikt hur ofta devices uppdaterar sin position och hur långt dom brukar röra sig. Ifall det är så att vi uppdaterar positionen av devices för ofta under testningen skulle det kunna ha en påverkan på datan eftersom databasen då behöver hanterar mer queries än den borde få in. Men eftersom alla tester kördes med positions uppdateringar fungerande på samma sätt ska det ändå gå att jämföra datan vi fått ut med varandra. Test datan borde däremot inte jämföras med verklig data eftersom device positions systemet inte är speciellt optimiserat.

I våra tester hade vi endast ett fåtal kombinationer av olika lösningar, vilket var index och polygonlösningarna samt alla lösningar tillsammans. Detta är något som vi helst skulle ha gjort mer av där vi testade hur prestandan av alla olika kombinationer av alla olika lösningar skulle fungera, men eftersom vårat arbete gav oss en begränsad mängd tid att hålla oss till behövde vi begränsa oss till en mindre mängd kombinationer. Vi valde just kombinationen av polygon och indexlösningarna eftersom vi märkte tidigt att polygon lösning självstående hade väldigt dålig prestanda och ville då se om den kunde understödas på ett effektivt sätt av en annan lösning, något som fungerade mycket väl. De kombinationer som vi inte hann testa är däremot något som skulle vara mycket intressant att se resultaten av vid eventuella framtida replikeringar och/eller utökningar av dessa tester.

Resultat

Det vi kan se utifrån resultaten som vi fått från testerna är att indexeringen verkar vara den mest påverkande lösningen för att optimisera prestandan. Även om att bara använda indexering inte ger det bästa resultatet så är det ändå den lösningen som ger störst påverkan på både exekveringstid och CPU-användning.

Något som var förvånande var hur stor skillnad det var mellan resultaten på polygonlösningen utan indexering och med indexering. Vi förväntade oss att polygon utan indexering skulle inte vara särskilt mycket sämre än baslösningen men den visade på upp mot 440% sämre exekveringstid. Vi antar att det beror på att utan indexering kan man inte använda bounding boxes för att göra beräkningarna billigare. Detta verkar stämma överens med när man ser att för testerna med quad_segs=16 och 32 blir polygonerna mer komplicerade och exekveringstiden går upp.

Testet av cachinglösningen verkar inte ge några förbättringar jämfört med baslösningen utifrån våra tester. Vi antar att detta kan bero på att vår lösning för att hitta närliggande blixtar faktiskt inte sparade särskilt mycket exekveringstid och att det därför inte gav något bättre resultat. Det kan också bero på att vi inte hittade närliggande blixtar särskilt ofta så att den gick över till baslösningen istället. Om vi hade haft en databas där vi sökte mot stationära devices istället hade vi kanske fått bättre resultat då vi inte hade behövt anpassa oss efter att devices rör sig.

Det vi kan se i resultaten är att det verkar som att indexeringen ger bäst resultat som individuell lösning. Exekveringstiden för flashes går ner med 78,6% och CPU-användningen går ner med 36,4-53,9 procentenheter. Den stora förbättringen kan komma från att vi hade en relativt stor databas med över 400 000 devices eftersom att indexering ger större påverkan vid större databaser. Hade databasen varit mycket mindre hade det antagligen gett liknande resultat fast inte lika stora skillnader som vi har observerat.

Den lösningen som presterade bäst var dock inte endast indexeringen utan det var polygon + index som presterade bäst. Jämfört med endast indexering presterade den bättre i flash exekveringstid och CPU-användning

SLUTSATS

Med resultaten från testningen kan vi se att det finns flera olika metoder som kan användas för att förbättra prestandan primärt i blixtvakt databasen men troligtvis på många databaser som använder sig av liknande datatyper och struktur. Den metod som fungerade bäst i ansende till alla olika delar av prestandan som vi undersökte var vår polygonlösning tillsammans med användningen av index. Denna lösning gav den bästa prestanda ökningen för både CPU- och minnesanvändningen men även för flash exekveringstiden. Hade vi haft mer tid till undersökningen hade vi testat flera olika kombinationer mellan lösningarna för att få fram mer resultat kring hur man kan kombinera olika metoder för bäst resultat. Detta kan alltså vara en väg att föra denna linje av forskning vidare, genom att antingen hitta nya metoder, eller testa de olika kombinationer som inte täcks av detta arbete.

Det kan däremot vara viktigt att notera att detta arbete handlar om databaser som hanterar mycket geografisk data vilket kan betyda att databaser som hanterar enklare datatyper kanske inte kan använda sig av samma metoder och förvänta sig en liknande förändring av prestandan.

REFERENCES

[1] Khalil Amiri, Sanghyun Park, Renu Tewari, and Sriram Padmanabhan. 2003. DBProxy: A dynamic data cache for Web applications. In *Proceedings 19th International Conference on Data Engineering (Cat. No. 03CH37405)*. IEEE Computer Society, 821–821.

[2] S. Brass and C. Goldberg. 2004. Semantic errors in SQL queries: a quite complete list. In *Fourth International Conference onQuality Software, 2004. QSIC 2004. Proceedings.* 250–257. DOI: http://dx.doi.org/10.1109/QSIC.2004.1357967

[3] Korry Douglas and Susan Douglas. 2003. *PostgreSQL: a comprehensive guide to building, programming, and administering PostgresSQL databases*. SAMS publishing.

[4] Goetz Graefe. 2007. The five-minute rule twenty years later, and how flash memory changes the rules. In *Proceedings of the 3rd international workshop on Data management on new hardware*. 1–9.

[5] R. Hankins, T. Diep, M. Annavaram, B. Hirano, H. Eri, H. Nueckel, and J.P. Shen. 2003. Scaling and characterizing database workloads: bridging the gap between research and practice. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.* 151–162. DOI: http://dx.doi.org/10.1109/MICRO.2003.1253191

[6] Matthias Jarke and Jurgen Koch. 1984. Query Optimization in Database Systems. *ACM Comput. Surv.* 16, 2 (jun 1984), 111–152. DOI: http://dx.doi.org/10.1145/356924.356928

[7] MSB. 2022. *Handbok i kommunal krisberedskap*. MSB, Chapter 4. Riskkatalog.

[8] Thanh Thi Nguyen. 2009. Indexing PostGIS databases and spatial Query performance evaluations. *International Journal of Geoinformatics* 5, 3 (2009), 1.

[9] Chunxia Qi. 2016. On index-based query in SQL Server database. (2016). DOI: http://dx.doi.org/10.1109/ChiCC.2016.7554868

[10] John C. Worsley and Joshua D. Drake. 2002. *Practical PostgreSQL*. O'Reilly Associates, Inc.

[11] Cong Yan, Alvin Cheung, Junwen Yang, and Shan Lu. 2017. Understanding Database Performance Inefficiencies in Real-world Web Applications. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management (CIKM '17)*. Association for Computing Machinery, New York, NY, USA, 1299–1308. DOI: http://dx.doi.org/10.1145/3132847.3132954

[12] Veronika Šalgová, Michal Kvet, Marek Kvet, Matej Čajka, and Pavol Grofčík. 2022. IMDB database performance analysis. In *2022 IEEE 10th Jubilee International Conference on Computational Cybernetics and Cyber-Medical Systems (ICCC)*. 000323–000328. DOI: http://dx.doi.org/10.1109/ICCC202255925.2022.9922783