Inledning och bakgrund

Det har pratats en del om NoSQL-databaser i organisationen, och många har hört att de är en uppgradering till vanliga relationella databaser. De sägs vara snabbare och enklare att arbeta med. Med detta som grund har ledningen bett mig, som är bekant med NoSQL-databaser att skriva en rapport som sammanfattar det mest väsentliga kring dessa databaser.

Från SQL till NoSQL

Den moderna benämningen NoSQL startade som en twitter-hashtag som användes i samband med ett möte 2009 för medlemmar i olika projekt som blivit inspirerade av Amazons DynamoDB och Googles BigTable. Dessa projekt hade börjat experimentera med alternativa datalagringsmodeller och ville diskutera hur de kunde påverka allt från utveckling till skalbarhet (Fowler & Sadalage, 2012, s. 9).

Problemområden

NoSQL växte fram som ett svar på vissa utmaningar de relationella databaserna stötte på – framförallt i fråga om att hantera stora mängder trafik. När en organisation behöver expandera sin infrastruktur för att hantera mer trafik finns två alternativ, att skala vertikalt och köpa starkare maskiner, eller att skala horizontellt och använda många mindre maskiner som kan vara av kommersiell kvalitet. Att skala horizontellt är oftast billigare vid stor skala. Det är ofta även mer pålitligt, då kluster kan byggas för att vara tåliga för att individuella maskiner går ner, och databasen blir inte ett single point of failure, d.v.s. att applikationen slutar fungera om databasservern går ner eller om anslutningen till den går ner.

De relationella databaserna var dock inte designade för att köras på kluster, och den problematiken var en motivation för utvecklingen av databaser som är byggda för att köra på kluster. Det går att utföra sharding, d.v.s. att olika delar av datan finns på olika maskiner, men det är ingen perfekt lösning. Man förlorar möjligheten till att söka mot en gemensam datakälla – applikationen måste hålla koll på vilken data som är lagrad var. Utöver det förloras även transaktioner, följdriktighet (consistency) och referentiell integritet mellan dessa shards (Fowler & Sadalage, 2012, s. 8).

NoSQL löser skalbarhetsproblemet det genom att kombinera sharding, att ditt dataset är uppdelat på olika noder i klustret och peer-to-peer replication, som betyder att två eller fler noder delar samma dataset, men kan alla ta emot både läsningar och skrivningar. Skrivningar koordineras sedan med de noder servern delar dataset med. Detta leder till en skalbar lösning som inte har någon single point of failure. Aggregat-modellen många NoSQL-databaser använder passar väldigt bra för sharding då aggregatet är en naturlig enhet för uppdeling (Fowler & Sadalage, 2012, s. 43-45) (Klein et al, 2015, s. 9).

I skrivande stund har dock Oracle en tjänst som erbjuder mycket av det som Fowler & Sadalage saknar i de relationella databaserna (Oracle, 2015).
Den heter MySQL Cluster och erbjuder bland annat föjande:

- Automatisk Failover av trasiga noder
- Geografisk skalbarhet
- Schemaförändringar på servrar i produktion, samt schemaless-modeller med hjälp av integration mot memcached.
- Synkron datareplikation mellan noder för att garantera att datan är lagrad på mer än en nod
- Automatisk sharding av tabeller, vilket leder till att det går att skriva till och läsa från alla noder i klustret.
- Stödjer ACID, transaktioner och sökningar som går över flera noder i klustret.

Det är dock värt att nämna av källkritiska skäl att ovan information kommer från Oracles eget marknadsföringsmaterial. Jag är dock inte särskillt skeptisk till att detta är möjligt, då både Google och Amazon erbjuder skalbara lagringsalternativ med SQL i molnet (Yegulalp, 2014). Uber bloggar även om hur de använder MySQL Cluster tillsammans med docker (Recht, 2016).

Jag är inte förvånad över att de relationella databastillverkarna har utvecklats för att möta användingsfall där de måste köra databaserna som individuella noder i ett kluster, men det tog lite längre tid än nya, fräsha projekt då de fortfarande värderade den relationella modellen och ville ha kvar den.

Den relationella databasmodellen har varit dominant sedan den kom, och har hela tiden samexisterat med sökspråket SQL. Det kan vara viktigt att reflektera på om dominansen av en relationell datamodell påverkar sättet vi ser på världen (Dourish, 2014). Detta kan leda till det analyseras från ”fel håll”, d.v.s. att problemområdet analyseras och därefter representeras i en relationell datamodell, istället för att problemområdet analyseras och därefter paras ihop med den databas som passar bäst.

Ett annat problem är något som kallas ”Impedence mismatch”. Det är skillnaden mellan den relationella datamodellen och hur datan representeras i applikationen (Fowler & Sadalage, 2012, s. 5). Om du exempelvis vill ha ett person-objekt i Java som har information om kreditkort, adress, ålder, namn o.s.v. kan detta med fördel lagras i själva objektet. Motsvarande data i en relationell databas lagras i relationer och tupler.

CAP-Teoremet
När distribuerade NoSQL-databaser diskuteras går det inte att undvika att nämna CAP-teoremet. CAP står för:
Följdriktighet *(eng: Consistency)* Varje läsning ska ge dig den senaste skrivna datan.
Tillgänglighet *(eng: Availability)* säger att om du kan nå en nod i ett system,

ska du kunna skriva data till och läsa data från den.
Partitionstolerans *(eng: Partition Tolerance)* innebär att ett kluster inte slutar vara följdriktigt och tillgängligt om någon form av nätverksproblem uppstår som separerar klustret i mindre delar. (Fowler & Sadalage, 2012, s. 53).

Availability vs Consistency
När CAP-teoremet diskuteras sägs det ofta att det bara går att uppnå en av de två ovan nämnda, förutsatt att systemet kräver partitionstolerans. Vad det innebär i praktiken är att i ett system som kan uppleva nätverkspartitioner (vilket alla distribuerade system kan), måste databasen välja mellan tillgänglighet och följdriktighet. Detta är dock inte svart och vitt, utan det går att justera för att göra vissa operationer högst tillgängliga medan andra är högst följdriktiga. När och hur dessa avvägningar ska göras är starkt kopplat till den domän som systemet tjänar – det är exempelvis ofta acceptabelt att överboka hotel om det innebär att ingen potentiell kund förloras. Detta för att hotel ofta har möjlighet att hantera överbokningar (Fowler & Sadalage, 2012, s. 54-56).

Oftast väljer systemet att kompromissa med är hur följdriktigt det är. Detta kan utföras genom att vara ”eventuellt följdriktig”, vilket är en mindre pålitlig form av följdriktighet, och ”starkt följdriktig”, vilket är mer pålitligt. Det senare kan uppnås med hjälp av quorums, vilket innebär att varje nod dubbelkollar med fler noder innan den läser och skriver, för att öka säkerheten att den har den senaste informationen (Fowler & Sadalage, 2012, s. 57-59).

Varför NoSQL?
En av de störa vinsterma med NoSQL-databaser är att utvecklare har möjlighet att välja en lagringsmodell som passar för applikationens behov. Bara för att en organisation har applikationer som redan använder en relationell databas, betyder det inte att det inte går att välja en NoSQL-databas för en ny applikation, om den datamodellen passar bättre.
Ett bra exempel är att användersessioner eller varukorgar väldigt bra representeras av nyckelvärdesdatabaser och de färdiga ordrarna enkelt kan lagras i en dokumentdatabas. Användardata kanske finns i en relationell databas av olika skäl. Detta koncept kallas för polyglot persistance (Fowler & Sadalage, 2012, s. 134-135). För att kunna implementera polyglot persistance måste organisationen först ta steget från att använda databaser som integrationsmekanism till att använda webtjänster som intergrationsmekanism. Om databaserna blir applikationsspecifika följer många fördelar – det garanterar bland annatatt reglerna för manipulering av data alltid följs, och utvecklare behöver inte ta hänsyn till att andra applikationer kommer använda databasen. Med valet att ha applikationsspecifika databaser kommer möjligheten att välja en databas som passar för just den applikationens ändamål – detta kan förenkla både utveckling och skalning (Fowler & Sadalage, 2012, s. 6-7).

Olika NoSQL Familjer och Databaser

Nyckelvärdesdatabaser

Fowler & Sadalage (2012, s. 81) skriver:

”The value is a blob that the data store just stores, without caring or knowing what's inside; it's the responsibility of the application to understand what was stored. Since key-value stores always use primary-key access, they generally have great performance and can be easily scaled”

Implementationer

Namn	Adress
Riak	http://basho.com/products/
Redis	http://www.redis.io
Memcached	https://memcached.org/

Datastrukturering

I nyckelvärdesdatabaser lagras all data i ett hash-table, där nyckeln pekar direkt på ett värde. Vad värdet är spelar ingen roll för databasen, utan är bara en blob av information. Det är applikationens ansvar att förstå och tolka den datastruktur som ligger där. Det fungerar lika bra att lagra JSON, XML, CSV eller valfri annan sträng i blobben (Fowler & Sadalage, 2012 s. 81-82).

Sökning

I en traditionell nyckelvärdesdatabas går det bara att söka på nyckeln. I undantagsfall, exempelvis riak, går det att simulera en dokumentdatabas och utföra sökningar mot lucene-index (Fowler & Sadalage, 2012 s. 84-85). Detta går dock mot principen av en nyckelvärdesdatabas, och är detta ett krav passar en dokumentdatabas som MondoDB bättre. Det finns inget stöd för joins eller liknande operationer då varje nyckelvärdespar är oberoende av de andra.

Integration mot databasen

Riak exponerar ett HTTP-gränssnitt ungefär som en REST Webbtjänst. Det går att komma åt all data genom detta gränssnitt (Fowler & Sadalage, 2012 s. 85).

Huvudsakliga tillämpningsomriden

De tre huvudsakliga tillämningsområden som Fowler & Sadalage (2012) nämner är sessionshantering, varukorg och inställningar. Alla dessa kan lagras som aggregat-objekt och applikationen kräver ofta helheten i blobben som den sedan kan tolka för att visa användaren. Alla dessa tre kan också korrekt identifieras bara genom att veta användarens UUID, vilket gör att det går väldigt snabbt att hämta dem. Detta är den största vinsten mot relationella databaser.

Dokumentdatabaser

Fowler & Sadalage (2012, s. 89) skriver:

”These documents are self-describing, hierarchical data structures which can consist of maps, collections and scalar values. The documents stores are similar to each other but do not need to be exactly the same.”

Implementationer

Namn	Adress
MongoDB	https://www.mongodb.com/
CouchDB	http://couchdb.apache.org/
OrientDB	http://orientdb.com/orientdb/

Datastrukturering

Dokumentdatabaser lagrar just det – dokument. Dessa dokument är rikt beskrivna och en enkel liknelse är XML eller JSON, där det går att ha maps, samlingar, arrayer, arrayer av arrayer med mera som attribut i dokumentet. Det är en flexibel typ av databas och varje dokument i samma dokument-samling måste inte ha exakt samma attribut , utan är en heterogen samling (Fowler & Sadalage, 2012 s. 89).

Sökning

Dokumentdatabaser tillhandahåller generellt väldigt olika sökfunktionalitet. CouchDB låter dig söka via views, likt vyer i relationella databaser. I MongoDB, och andra dokumentdatabaser, går det att söka på valfritt fält inom en samling. Detta är en stor skillnad mot exempelvis nyckelvärdesdatabaser, där detta inte är möjligt. Detta möjliggör för väldigt komplexa sökningar. Det finns dock inget stöd för joins, för det finns inga relationer mellan samlingar. (Fowler & Sadalage, 2012 s. 94).

Integration mot databasen

I MongoDB interagerar användaren med databasen med hjälp av ett JSON-liknande syntax. Det finns inbyggd funktionalitet som representerar SELECT, WHERE, INSERT, UPDATE och ORDER BY från SQL. (Fowler & Sadalage, 2012 s. 94).

Huvudsakliga tillämpningsomriden

MongoDB lämpar sig för heterogena datasamlingar. Vanliga exempel på detta är eventloggning, CMS, finansdata och analysdata. Lämpligheten ligger i att det går att ändra schemat beroende på hur behoven och verksamheen förändras (Fowler & Sadalage, 2012 s. 97-98). Om fyra attribut loggas på en maskin i en fabrik exempelvis, och ett femte behöver läggas till är detta problemfritt. Det är en stryka mot en relationell databas, där schemat måste ändras innan ny information kan lagras.

Columndatabaser

Fowler & Sadalage (2012, s. 111) skriver:

”Column families, such as Cassandra [etc.], allow you to store data with a keys mapped to values and the values grouped into multiple column families, each column family being a map of data.”

Implementationer

Namn	Adress
Cassandra	http://cassandra.apache.org/
Amazon SimpleDB	https://aws.amazon.com/simpledb/
Hypertable	http://www.hypertable.org/

Datastrukturering

I Cassandra struktureras datan i rader som identifieras av en nyckel. Raden består sedan av nyckelvärdespar där värdet kan vara individuella värden eller samlingar som sets, maps eller listor. Varje nyckelvärdepar har även en tidsstämpel som ändras varje gång data skrivs eller uppdateras där. Tidsstämpeln används sedan för expirera data eller hantera skriv-konflikter (Fowler & Sadalage, 2012 s. 100-101).

Sökning

I Cassandra går det bara att söka på indexerade rader och primärnyckeln. Av denna anledning rekommenderas det att optimera datan för läsning redan från början, då Cassandra inte än har ett rikt sökspråk. (Fowler & Sadalage, 2012 s. 105-106).

Integration mot databasen

Cassandra har ett query-språk som heter CQL, Cassandra Query Language. Syntaxmässigt liknar det SQL, och det har satser som ”INSERT INTO” och ”SELECT...”. CQL är dock mindre etablerat än SQL, och stöder exempelvis inte joins eller subqueries. Detta på grund av den grundläggande designen bakom Cassandra (Fowler & Sadalage, 2012 s. 100-101).

Huvudsakliga tillämpningsomriden

Kolumndatabasers tillämpningsområden är relativt lika dokumentdatabasernas, båda databastyper använder sig av aggregatorienterade datamodeller. Den huvudsakliga skillnaden ligger i hur schemat definieras. Cassandra har ett mindre flexibelt schema, och det krävs ett ”ALTER TABLE” för att lägga till en nytt nyckelvärdespar i columnfamiljen. En fördel med Cassandra är att det finns en inbyggd funktionalitet för att expirera data som läggs in. De användningsområden som finns är CMS, Event-loggning o.s.v (Fowler & Sadalage, 2012 s. 108-109). Den huvudsakliga vinsten mot en relationell databas här är att aggregatsmodellen är mer lämpad för den data som loggas.

Grafdatabaser

Grafdatabaser är den fjärde familjen av NoSQL-databaser, och skiljer sig väldigt markant från mängden. De andra tre familjerna är aggregatorienterade, medan grafdatabaser modellerar sin data på ett helt annat sätt. Grafdatabaser ryms under NoSQL-paraplyet för att det ingår i rörelsen bort från ”one-size-fits-all” till att välja lagringslösningar som passar användningsområdet bäst (Neo4J).

Fowler & Sadalage (2012, s. 106-107) skriver:
”Graph databases allow you to store entities and relationships between these entities. Entities are also known as nodes, which have properties. ”

Implementationer

Namn	Adress
Neo4J	https://neo4j.com/
OrientDB	http://orientdb.com/
FlockDB	https://github.com/twitter/flockdb

Datastrukturering

I grafdatabaser lagras data som noder och relationer mellan noder. Dessa relationer kan vara viktade och är riktade. Om applikationen exempelvis kräver en bidirektionell relation ”vän” mellan två person-noder, skapas två relationer, en från vän #1 till vän #2, och vice versa (Fowler & Sadalage, 2012, s. 112-113).

Sökning

I grafdatabaser går det inte att söka på samma sätt som i relationella eller aggregatorienterade databaser. Här traverserar sökningen grafer och letar efter relationer, läser av vikter och gör analyser baserat på detta. Det finns dock inget behov för joins (Fowler & Sadalage, 2012, s. 114-116).

Integration mot databasen

Neo4J stöder Cypher och Gremlin. Cypher är Neo4J's egna syntax, och Gremlin är ett generellt men domänspecifikt queryspråk. Båda språken tillåter traversering, sökning, läsning av relationer etc. Neo4J stödjer även ACID (Fowler & Sadalage, 2012, s. 112-113).

Huvudsakliga tillämpningsomriden

Tillämpningsområden är framförallt social data och geografisk data. Båda dessa kan beskrivas väldigt bra med hjälp av viktade och riktade grafer. Jag inkluderar rekommendationsmotorer i social data, då dessa använder användarens beslut för att skapa rekommendationer (Fowler & Sadalage, 2012, s. 120-121). Att använda en grafdatabas för dessa implementationer är bättre än en relationell databas för att det är svårt att beskriva grafer relationellt.

Programutvecklingsaspekter

Användandet av NoSQL-databaser kan leda till ett enklare tänk över den data som lagras. Utan relationer och med aggregatorienterade datamodeller kan databasen lagra de datastrukturer applikationen använder på ett liknande sätt. Detta gör att utvecklarne inte behöver ORMs, vilket kan förenkla utvecklingsarbetet. Detta realiseras genom att använda applikationsdatabaser. (Fowler & Sadalage, 2012, s. 5-6).

Schemalösa databaser förenklar förändringar i databasstrukturen i en applikation som är i produktion men fortfarande förändras frekvent. Det gör det även enklare att hantera inkonsekvent data. Detta gör dock att det krävs mer av applikationen, som måste hålla koll på denna inkonsekventa data. Detta leder till att utvecklare ofta använder ett ”implicit schema”, som applikationens logik följer (Fowler & Sadalage, 2012, s. 28-29).

MapReduce tillåter analytiker att utföra databehandling över stora kluster av information, vilket förenklar arbetet skapa dashboards och andra analysverktyg för datan, vilket kan hjälpa organisationens beslutsfattandeprocess (Fowler & Sadalage, 2012, s. 67-69).

Slutsatser

NoSQL erbjuder många nya intressanta möjligheter för att bygga system på bättre sätt, men de är inte en ersättning för den relationella databasmodellen, utan ett alternativ till den i de problemområden där en annan datamodell passar bättre. Det finns för och nackdelar vad gäller utvecklingsproduktivitet, och dessa bör vägas mot olika vinster och förluster vid val av olika databaser.

Referenslista

Fowler, M., & Sadalage, P. J. (2012). NoSQL distilled. Boston, MA: Addison-Wesley.

Klein, J., Gorton, I., Ernst, N., Donohoe, P., Pham, K., & Matser, C. (2015). Performance Evaluation of NoSQL Databases. *Proceedings of the 1st Workshop on Performance Analysis of Big Data Systems - PABS '15*. doi:10.1145/2694730.2694731

Dourish, P. (2014). No SQL: The Shifting Materialities of Database Technology. http://computationalculture.net/article/no-sql-the-shifting-materialities-of-database-technology (Hämtad 2016-11-17)

Oracle. (2015). *MySQL Cluster - Web Scalability, Carrier Grade Availability, and Developer Agility.* [online]. https://www.youtube.com/watch?v=A7dBB8_yNJI (Hämtad 2016-11-15)

Yegulalp, S. (2014). *Google Cloud SQL debuts, following Amazon's RDS lead* http://www.infoworld.com/article/2610430/cloud-computing/google-cloud-sql-debuts--following-amazon-s-rds-lead.html (Hämtad 2016-11-15)

Recht, J. (2016) *Dockerizing MySQL at Uber Engineering* https://eng.uber.com/dockerizing-mysql/ (Hämtad 2016-11-15)

Neo4J. *How Graph Databases Relate To Other NoSQL Data Models. https://neo4j.com/developer/graph-db-vs-nosql/ (Hämtad 2016-11-17)*