Inleiding tot NoSQL

Not only SQL

NoSQL: Hoe? Waarom?

Recente (vooral open source) beweging

Gedreven door

- Performantie (vooral als er veel wordt weggeschreven in vergelijking tot het aantal leesoperaties)
- Grote hoeveelheden data
- clusters

Verzamelterm voor allerlei recente databanksystemen (21ste eeuw) die niet klassiek relationeel zijn

Typisch: werken op grote datasets

NoSQL: Hoe? Waarom?

Niet-relationele databanken

- Geen nieuw verschijnsel!
  - Er zijn bijv. pogingen geweest om object-geöriënteerde databanksystemen te lanceren
    - dit was vooral om inherent probleem met RDBMS op te lossen (=> impedance mismatch, zie later)

NoSQL databanken:

- Typisch ontwikkeld voor gedistribueerde en parallelle omgevingen

RDBMS

RDBMS: voorziet gebruikers met de beste mix van eenvoud, robuustheid, flexibiliteit, performantie, schaalbaarheid en compatibiliteit

Echter: het is een mix, en blinkt niet specifiek uit in één van de opgesomde domeinen

Tegenwoordig is schaalbaarheid een probleem! => hoge workloads

NoSQL: Hoe? Waarom?

RDBMS: goed zolang # gebruikers niet te groot is en zolang data goed gestructureerd is

Indien # gebruikers en hoeveelheid data stijgt: duurdere hardware aanschaffen

Niet evident om RDBMS op verschillende servers (cluster) te verspreiden

NoSQL: kan groot # gebruikers en grote hoeveelheid data aan zonder dure hardware (bijv. cluster)

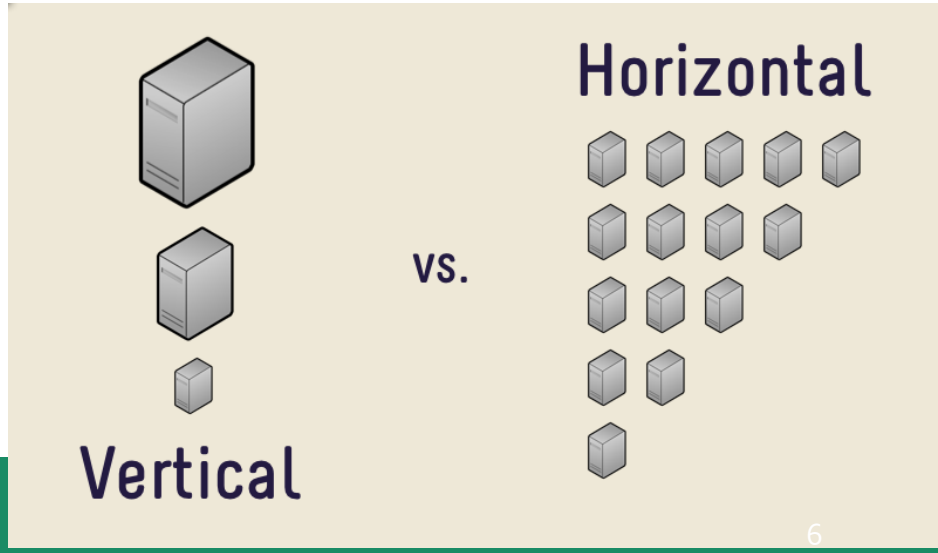
Begrip verticale versus horizontale schaalbaarheid

Begrip schaalbaarheid

Schaalbaarheid: mogelijkheid van een systeem om de throughput te doen toenemen door extra resources toe te voegen om toenemende belasting te counteren

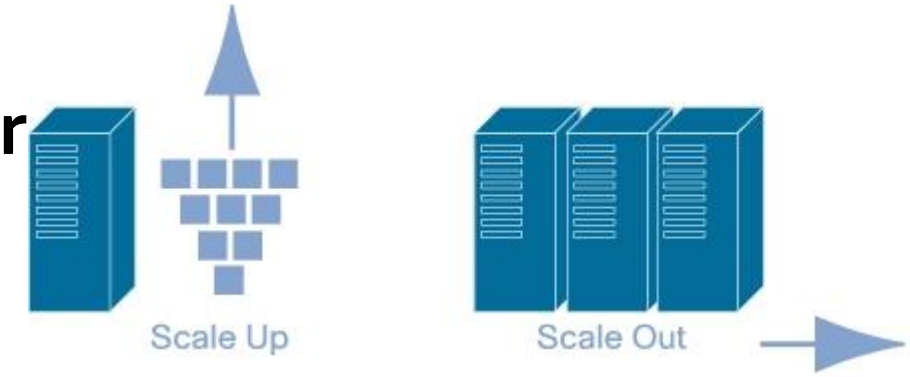
Horizontale versus verticale schaalbaarheid

- Verticaal: grotere, snellere supercomputer kopen
- Horizontaal: extra pc toevoegen aan cluster



Terminologie: “Scale out” vs “scale up”

Scaling up: grotere server => duurder!



Scaling out: extra server toevoegen in cluster

Probleem met relationele databanken

Schaalbaarheid: moeilijk op te lossen met traditionele RDBMS'en

- Oplossing: grotere (dus duurdere) server kopen
- Maar indien meerdere servers nodig zijn: niet evident meer!

Tegenwoordig: cloud-diensten zijn populair => schaalbaarheid is belangrijk

- Dus toevlucht nemen tot andere types van databanksystemen
- Consequentie: eigenschappen van RDBMS komen in gedrang:
  - typisch is dat consistentie (bijv. zelfde data op alle nodes…)

Impedance Mismatch

Eén van de grootste frustaties bij ontwikkelaars

Dit is het verschil tussen relationeel model (databank) en de datastructuren (applicatie)

Het is moeilijk om klassen of objecten 1-op-1 te mappen op tabellen of records in databank (en vice versa)

- Bijv. sommige objecten zijn verspreid over verschillende tabellen omwille van normalisatie
- Door normaliseren: veel joinen (is traag)

Eén van de oplossingen is gebruik maken van een ander soort databanksysteem!

Impedance Mismatch

In een rij van een tabel:

- Enkel eenvoudige data opslaan (zg. tuples)
- Niet mogelijk om een list in één record op te slaan
- Dit is wel mogelijk in een datastructuur

Ingewikkelde datastructuur dient wel op een of andere manier vertaald te worden naar relationeel model

Dus: impedance mismatch: 2 verschillende representaties die vertaling nodig hebben

Impedance Mismatch

In jaren 90 proberen op te lossen met object-geöriënteerde databanken

- Geen succes

Succes van RDBMS:

- SQL (quasi standaard)
- Eenvoudig aan te leren
- Leidde ook tot onderscheid tussen software ontwikkelaars en database admins

Nu:

- Proberen op te lossen via bijv. document-based NoSQL databanken zoals MongoDb (zie straks)

Kenmerken NoSQL databanken

NoSQL databanken gebruiken geen SQL

- Sommige hebben eigen querytaal die goed op SQL lijkt: bijv. Cassandra heeft CQL.
- Maar geen enkele heeft querytaal die zo flexibel is als SQL

Zijn over het algemeen open source projecten

Gemaakt om op clusters te lopen (in tegenstelling tot klassieke RDBMS)

- Gevolgen:
  - RDBMS gebruiken ACID transacties om consistentie te garanderen
  - *Garanderen van consistentie is niet vanzelfsprekend op clusters*

Uitzondering: graafdatabanken => single server

Kenmerken NoSQL databanken

Typisch gemaakt in 21ste eeuw

NoSQL: gebruiken geen schema (schemaless)

- Kan zonder problemen velden toevoegen zonder databankstructuur te moeten aanpassen (dus geen ALTER table zoals in SQL)

Polyglot persistentie

Verschillende databanksystemen gebruiken in verschillende omstandigheden

Ipv standaard RDBMS te nemen

- Eerst proberen te begrijpen wat de data is die we willen opslaan en hoe we die kunnen manipuleren
- Bedrijven zullen meerdere databanksystemen gebruiken afhankelijk van de omstandigheid

Verschuiving van *integratie*databanken naar *applicatie*databanken

Polyglot persistentie

Integratiedatabanken vs applicatiedatabanken

- Integratie: centraal instrument waarvan verschillende applicaties gebruik maken (zie http://martinfowler.com/bliki/IntegrationDatabase.html)
- Applicatiedatabank: één geschikte databank per applicatie gebruiken (zie http://martinfowler.com/bliki/ApplicationDatabase.html)

SOORTEN NOSQL DATABANKEN

Nieuwe generatie databanken

Datamodel

- model waardoor we data zien en manipuleren
- Beschrijft hoe we interageren met data in databank

Meest gekende:

- Relationele datamodel
  - Verzameling tabellen, met rijen, en elke rij stelt een entiteit voor
  - Relaties tussen tabellen

Nieuwe generatie databanken

NoSQL (Not only SQL) databanktypes

- Document based
- Key-value
- Column-oriented
- Graph

} aggregatiedatabanken

Aggregatiedatabanken

Document-based, key-value en column-oriented vallen hieronder

Laat toe om data op te slaan in complexere structuur dan rijen in een tabel

Aggregatie: *collectie van gerelateerde objecten die we willen behandelen als één geheel*

Makkelijker om met clusters te werken:

- één collectie van objecten (aggregatie) kan nl. op één node van cluster

Ook gemakkelijker voor ontwikkelaars

- Makkelijker om data via aggregatiestructuur te manipuleren

Key-value en document datamodellen

Key-value:

- Aggregatie is een grote blob bestaande uit betekenisloze bits
- Je kan gelijk wat opslaan in blob
- Opzoeken mbv sleutel (vandaar de naam)

Document-based:

- Structuur in aggregatie
- Definieert wat wel en niet mag van data
- Opzoeken op basis van wat er in aggregatievelden zit, je kan een deel van aggregatie opvragen

Document based

Bestaat uit documenten

Kunnen bestaan uit collecties, scalaire waarden, …

Documenten die opgeslagen worden zijn gelijkaardig, maar zijn niet noodzakelijk allemaal hetzelfde opgebouwd

In het valuedeel worden documenten opgeslagen

Key-value databanken waarbij het valuedeel kan ondervraagd worden

Spelers

MongoDB

- Claimt: hoge performantie

CouchDB:

- Mikt op hoge concurrency scenario's

Dit komt met een prijs!!!

- Geen ondersteuning voor transacties
  - MongoDB instantie zal geen stroomuitval overleven

Systemen die ACID nodig hebben => opteren beter voor relationele systemen

Mission-critical data (banken, nucleaire systemen) zal je niet snel in NoSQL systemen vinden

Typisch toepassingsgebied

MongoDB

- Werkt goed voor applicaties en componenten die data opslaan die vaak en snel gebruikt wordt
- Typische voorbeelden:
  - Website analytics, gebruikersvoorkeuren en settings, CMS, blogs, system configuratie, log data, event logging, e-commerce applicaties

MongoDb gebruikers

Zie https://www.mongodb.com/who-uses-mongodb

Is één van de populairste NoSQL databanksystemen

Lees ook http://www.theregister.co.uk/2012/03/27/picking_a_no_sql_winner/

JSON en BSON

BSON: Binary JSON

MongoDB gebruikt BSON om documenten op te slaan

JSON: JavaScript Object Notation

Wordt gebruikt voor het uitwisselen van datastructuren

Alternatief voor XML

Kan rechtstreeks ingelezen worden in JavaScript applicatie, er is geen aparte parser (zoals bij XML) nodig

JSON voorbeeld

Toegelaten datatypes zijn:

- Number
- String
- Boolean
- Array
- Een ander JSON Object
- Functions
- Null

JSON Voorbeeld

```
var User = {
            Age: 50,
            Name: “Filip”,
            IsUnderAge: false
};

var User = {
            Friends:[
                  “Albert”,
                  “Willem-Alexander”,
                  “Harry”
            ]
};
```

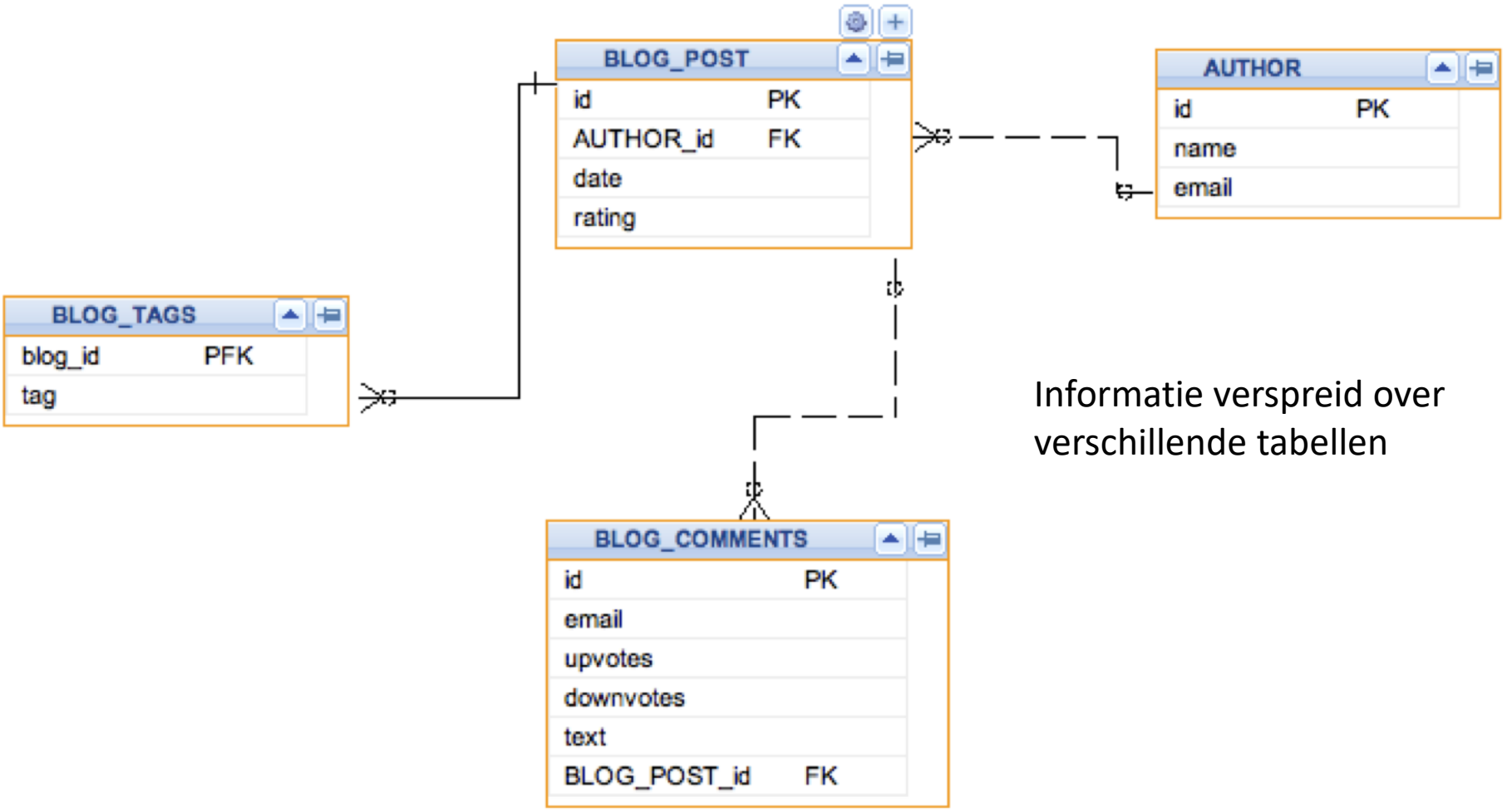
JSON voorbeeld

Je kan ook elke friend als een User object toevoegen

```
var User= {
        Name: “Filip”,
        Age: 50,
            Friends:[
            {Name: “Albert”, Age: 53}, {Name: “Willem-Alexander”, Age:
            40}, {Name: “Harry”, Age: 25}
            ],
        IsUnderAge: function(){
            return this.Age > 18 ? true: false;
        }
};
```

Zie http://www.codeproject.com/Articles/210568/Quick-JSON-Tutorial

Blog post (relationeel databankmodel)



Informatie verspreid over verschillende tabellen

MongoDb: blog post (JSON)

```
{
    _id: 1234,
    author: { name: "Bob Davis", email : "bob@bob.com" },
    post: "In these troubled times I like to …",
    date: { $date: "2010-07-12 13:23UTC" },
    location: [ -121.2322, 42.1223222 ],
    rating: 2.2,
    comments: [
       { user: "jgs32@hotmail.com",
         upVotes: 22,
         downVotes: 14,
         text: "Great point! I agree" },
       { user: "holly.davidson@gmail.com",
         upVotes: 421,
         downVotes: 22,
         text: "You are a moron" }
    ],
    tags: [ "Politics", "Virginia" ]
 }
```

Data wordt opgeslagen in één enkele collectie

MongoDB en C#

- Maak een directory data, met daarin subdirectory db
- starten van mongodb (via mongod.exe)
- Nieuw project aanmaken en via NuGet (`MongoDB.Driver`) driver voor C# afhalen

Merk op: er bestaat ook nog een mongocsharpdriver op NuGet maar dat is een vroegere versie.

MongoDB en C#: connecteren

```
MongoClient client= new
                    mongoClient(“mongodb://localhost”);
var db = client.GetDatabase(“test”);
    ...
```

Klasse Person

Nieuwe klasse maken:

```
public class Person {
    public int Id { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
    public string Profession { get; set; }
}
```

MongoDB en C#: collectie van entiteiten opvragen

```
MongoClient client = new
          MongoClient(“mongodb://localhost”);
var db = client.GetDatabase(“test”);
IMongoCollection<Person> collection =
                        db.GetCollection<Person>(“persons");
```

MongoDB en C#: data toevoegen

```
MongoClient client = new  MongoClient(“mongodb://localhost”);
var db = client.GetDatabase(“test”);
IMongoCollection<Person> collection =
                              db.GetCollection<Person>(“people");
Person jane = new Person(1, “Jane van Tarzan“, 24, “Avonturier” };
collection.InsertOneAsync(jane);
```

MongoDB en C#: een document zoeken

```
...
IMongoCollection<Person> collection =
                        db.GetCollection<Person>(“people");
Person jane = new Person(1, “Jane van Tarzan“, 24,”Avonturier” };
collection.InsertOneAsync(jane);
var people = collection.Find(x => x.Age < 42).ToListAsync();
```

MongoDB en C#: update + deleten

```
var result = collection.UpdateOneAsync( x => x.Name == "Tom",
Builders<Person>.Update.Set(x => x.Profession, "Musician"));
```

Alternatief:

```
var tom = await collection.Find(x => x.Id ==
        ObjectId.Parse("550c4aa98e59471bddf68eef")).SingleAsync();
tom.Name = "Thomas";
tom.Age = 43;
tom.Profession = "Hacker";
var result = collection.ReplaceOneAsync(x => x.Id == tom.Id, tom);


var result = collection.DeleteOneAsync(x => x.Id == tom.Id);
```

Meer informatie:

http://mongodb.github.io/mongo-csharp-driver/2.5/getting_started/ of
https://docs.mongodb.com/getting-started/csharp/

Real-world voorbeeld

Large Hadron Collider (CERN)

- Enorme hoeveelheden data verwerken (10 PB/jaar => 1 PB = 1 000 TB = 1 000 000 GB)
- Gebruiken CouchDB (speciaal ontwikkeld voor gedistribueerde omgevingen => voorziet zelf replicatie van data)

Beslissing voor document-based baseren op

Data is zwaar document georiënteerd

Ontwikkelingsomgeving is object-georiënteerd

Data opslag is goedkoop

Je belangrijkste zorg is on-demand, en schaalbaarheid

Key-value

Key-value

- Alle toegang tot databank gebeurt via primaire sleutel

Gebruiker kan enkel via sleutel waarde opvragen, de sleutel verwijderen, of een waarde toekennen aan de sleutel

Waardedeel is een blob van data

Verantwoordelijkheid van applicatie om te begrijpen wat precies is opgeslagen

Key-value: goed?

Geschikt voor cloud toepassingen

- Wegens beter schaalbaar

Past beter bij ontwikkelde code

- Relationeel datamodel en object model zijn vaak verschillend
- Soms moeilijk om beiden op elkaar te mappen (zie ORM)
- Gaat makkelijker bij key-value databanken

Key-value: slecht?

Data-integriteit:

- Bij RDBMS: automatisch afgedwongen
- Bij NoSQL: verantwoordelijkheid van ontwikkelaar

Datamodelling => logische structuur

- Verplicht bij RDBMS
- Niet vanzelfsprekend bij key-value databanken

Compatibiliteit: niet vanzelfsprekend bij NoSQL!

- Elke vendor heeft zijn eigen systeem. Niet eenvoudig om opeens een andere vendor te nemen

Toepassingsgebied

Gebruik key-value databanken bij voorkeur

- Om sessie informatie op te slaan (sessionid als key gebruiken)
- Gebruikersprofielen en –voorkeuren (via userID)
- Online shopping (winkelmandje) met userid als key

Key-value beter niet gebruiken

Om relaties tussen data te bewaren

Als je wilt zoeken op data die in value-deel zit

Redis (REmote DIctionary Server)

- Is één van gekendste key-value NoSQL databanken
- "Redis is an open-source in-memory database project implementing a distributed, in-memory key-value store with optional durability."
- Heeft een C# Client: https://github.com/StackExchange/StackExchange.Redis
- Wordt bijv gebruikt voor Stack Overflow

Voorbeeldcode

- **Connectie maken**

```
using StackExchange.Redis;
...
ConnectionMultiplexer redis =
ConnectionMultiplexer.Connect("localhost");
// ^^^ store and re-use this!!!
```

Voorbeeldcode

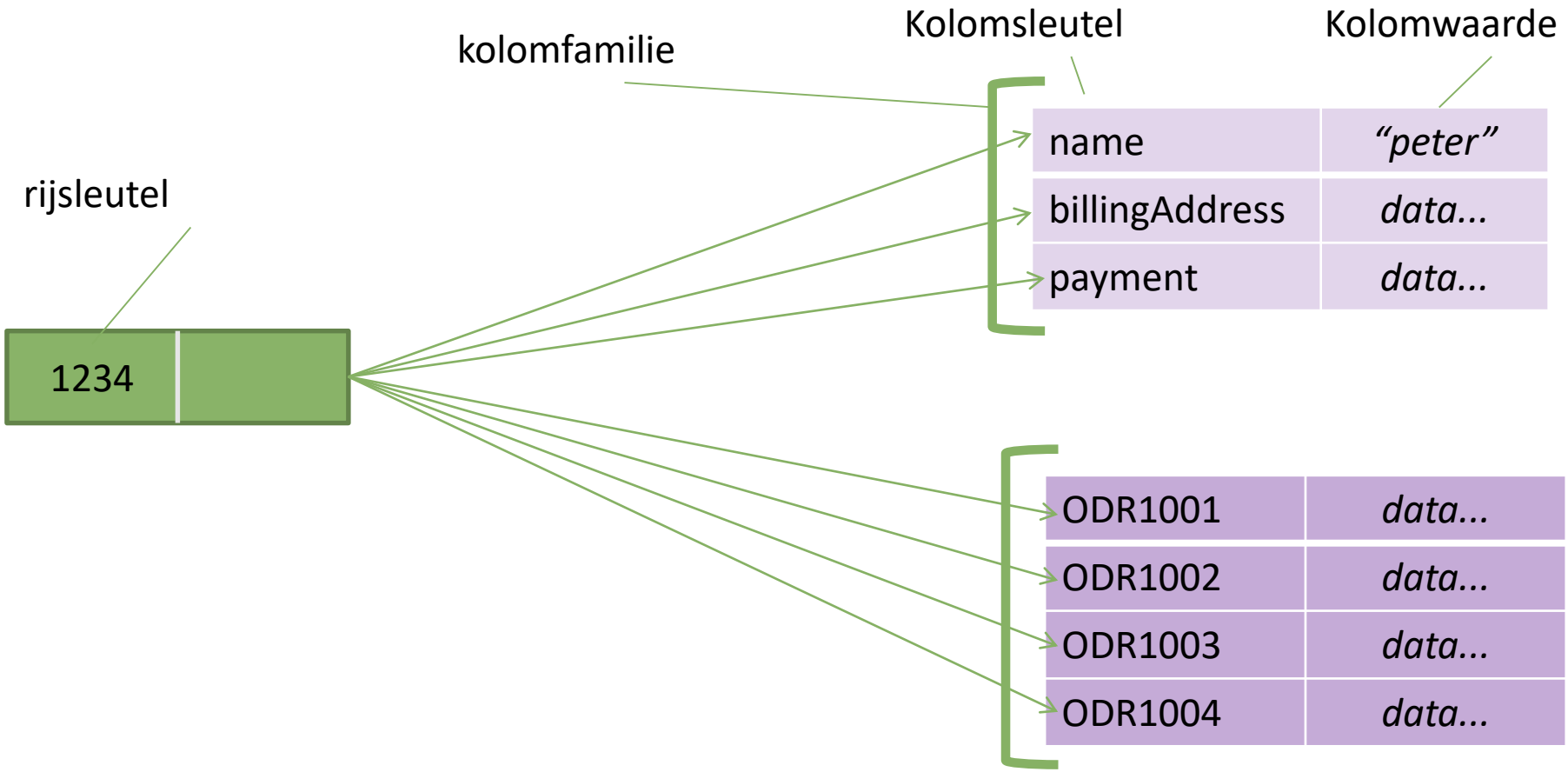
- **Redis databank gebruiken**

```
IDatabase db = redis.GetDatabase();
string value = "abcdefg";
db.StringSet("mykey", value);
...
string value = db.StringGet("mykey");
Console.WriteLine(value); // writes: "abcdefg"
```

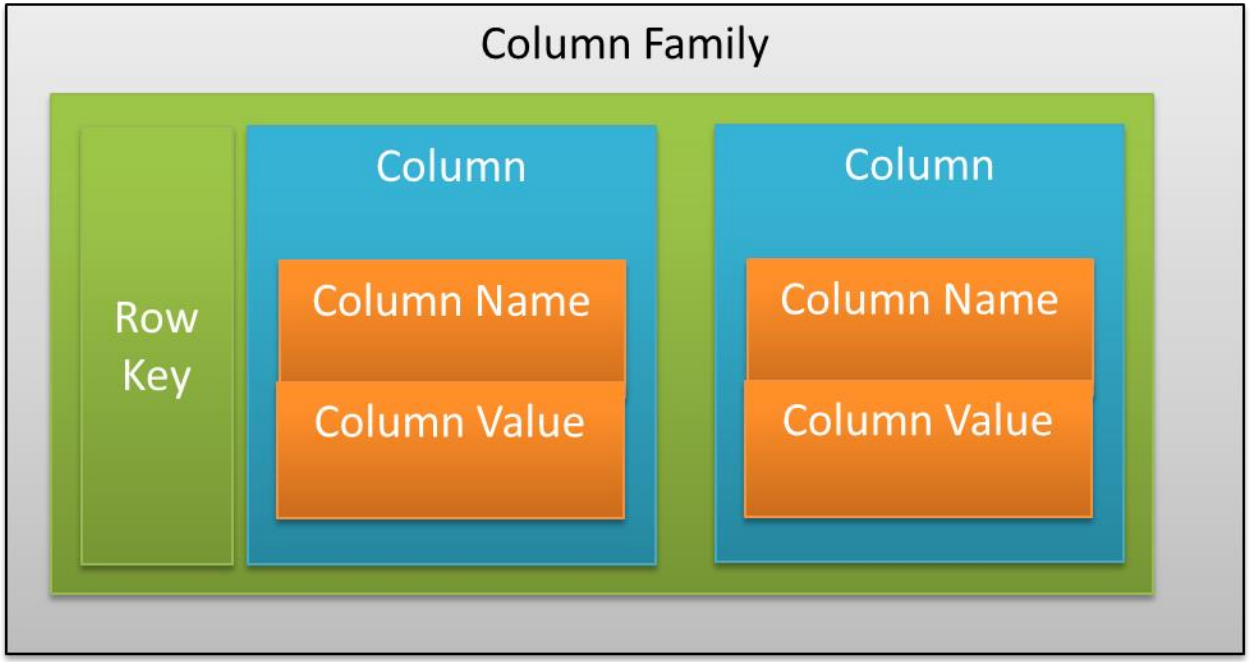
Kolomfamilie databanken

Data wordt opgeslagen als groepen van kolommen van alle rijen

Kolomfamilie databanken



Kolomfamilie: Cassandra



Waarom Cassandra?

"Relational databases typically structure tables in order to keep data duplication at a minimum. The various pieces of information needed to satisfy a query are stored in various related tables that adhere to a pre-defined structure. Because of the way data is structured in a relational database, writing data is expensive, as the database server has to do additional work to ensure data integrity across the various related tables. As a result, relational databases usually are not performant on writes."

Waarom Cassandra?

"Cassandra is optimized for write throughput. Cassandra writes are first written to a commit log (for durability), and then to an in-memory table structure called a *memtable*. A write is successful once it is written to the commit log and memory, so there is very minimal disk I/O at the time of write. Writes are batched in memory and periodically written to disk to a persistent table structure called an *SSTable* (sorted string table). Memtables and SSTables are maintained per column family. Memtables are organized in sorted order by row key and flushed to SSTables sequentially (no random seeking as in relational databases)."

Cassandra installeren en gebruiken

Download Cassandra van:

http://cassandra.apache.org/

Volg tutorial op

https://academy.datastax.com/resources/getting-started-apache-cassandra-and-c-net

Of op https://www.guru99.com/cassandra-tutorial.html

Cassandra: demo

```
public class SimpleClient {
        public Cluster Cluster { get; private set; }
        public ISession Session { get; private set; }

        public void Connect(String node) {
                Cluster = Cluster.Builder().AddContactPoint(node).Build();
                Metadata metadata = Cluster.Metadata;
                Console.WriteLine("Connected to cluster: " +
                                        metadata.ClusterName.ToString());
                foreach (var host in Cluster.Metadata.AllHosts()) {
                    Console.WriteLine("Data Center: " + host.Datacenter + ", " +
                    "Host: " + host.Address + ", " +
                    "Rack: " + host.Rack);
                }
                Session = Cluster.Connect();
        }
```

Cassandra: demo

```
public void Close(){
      Cluster.Shutdown();
      Session.Dispose();
}
public void CreateSchema(string keyspace) {
       Session.Execute("CREATE KEYSPACE "+ keyspace +" WITH replication " +
                   "= {'class':'SimpleStrategy', 'replication_factor':3};");
       Session.Execute(
           "CREATE TABLE " + keyspace+".songs (" +
               "id uuid PRIMARY KEY," +
               "title text," +
               "album text," +
               "artist text," +
               "tags set<text>," +
               "data blob" +
           ");");
```

Cassandra: demo

```
    Session.Execute(
            "CREATE TABLE " + keyspace + ".playlists (" +
                "id uuid," +
                "title text," +
                "album text, " +
                "artist text," +
                "song_id uuid," +
                "PRIMARY KEY (id, title, album, artist)" +
            ");");
    }
```

Cassandra: demo

```
public virtual void LoadData(string keyspace) {
            Session.Execute("INSERT INTO " + keyspace +
              ".songs (id, title, album, artist, tags) " +
                 "VALUES (" +
                     "756716f7-2e54-4715-9f00-91dcbea6cf50," +
                     "'La Petite Tonkinoise'," +
                     "'Bye Bye Blackbird'," +
                     "'Joséphine Baker'," +
                     "{'jazz', '2013'})" +
                     ";");
```

Cassandra: demo

```
    Session.Execute(
                "INSERT INTO " + keyspace + ".playlists (id,
                       title, album, artist, song_id) " +
                "VALUES (" +
                    "2cc9ccb7-6221-4ccb-8387-f22b6a1b354d," +
                    "'La Petite Tonkinoise'," +
                    "'Bye Bye Blackbird'," +
                    "'Joséphine Baker'," +
                    "756716f7-2e54-4715-9f00-91dcbea6cf50" +
                    ");");
    }
```

Cassandra: demo

```
public void QuerySchema(string keyspace) {
    RowSet results = Session.Execute("SELECT * FROM " + keyspace +
    ".playlists "+"WHERE id = 2cc9ccb7-6221-4ccb-8387-f22b6a1b354d;");
    Console.WriteLine(String.Format("{0,-30}\t{1,-20}\t{2,-20}\t{3,-30}",
    "title", "album", "artist", "tags"));
     Console.WriteLine("------------------------------+----------------
------+--------------------+------------------------------");
     foreach (Row row in results.GetRows()){
         Console.WriteLine(row.GetValue<String>("title") + " " +
          row.GetValue<String>("album") + " " + row.GetValue<String>
          ("artist"));
      }
  }
```

Cassandra: demo

```
public void DropSchema(string keyspace) {
     Session.Execute("DROP KEYSPACE " + keyspace);
     Console.WriteLine("Finished dropping " + keyspace + "
                        keyspace.");
 }
```

Cassandra: demo

```
    public static void Main(String[] args)  {
              SimpleClient client = new SimpleClient();
               client.Connect("127.0.0.1");
              client.DropSchema("simplex");
               client.CreateSchema("simplex");
              client.LoadData("simplex");
               client.QuerySchema("simplex");
              Console.ReadKey(); // pause the console before exiting
               client.DropSchema("simplex");
              client.Close();
      }
}
```

Wanneer gebruiken?

Event logging

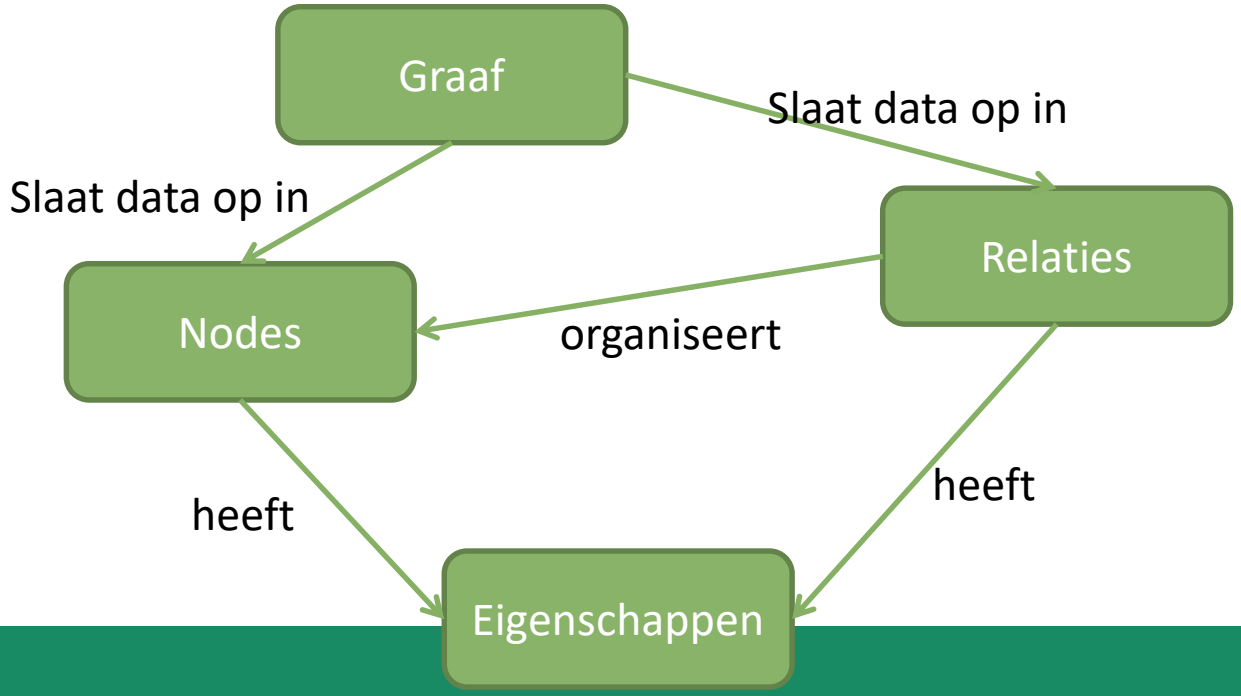
CMS, blogging omgevingen

Counters

Graph databases

Niet gemaakt om op clusters te draaien

Wel om relaties tussen kleine stukjes data bij te houden



Graph databases

Wordt vooral gebruikt in sociale netwerken (Twitter, Facebook,…) om relaties tussen personen te achterhalen

Wanneer gebruiken?

Geconnecteerde data

- Data met verschillende relaties

Locatiegebaseerde diensten

Recommendation engine

BEDENKINGEN

Minpuntje voor NoSQL

Je kan niet alle NoSQL systemen voor alles gebruiken

- Sommigen zijn goed wanneer er veel weggeschreven moet worden en weinig gelezen
- Andere vice versa (veel lezen, weinig wegschrijven)

Komt er op aan de goede tool voor de juiste job te kiezen!

Voorbeelden (Adam Wiggins)

Frequently-written, rarely read statistical data (for example, a web hit counter) should use an in-memory key/value store like Redis, or an update-in-place document store like MongoDB.

Big Data (like weather stats or business analytics) will work best in a freeform, distributed db system like Hadoop.

Binary assets (such as MP3s and PDFs) find a good home in a datastore that can serve directly to the user's browser, like Amazon S3.

Transient data (like web sessions, locks, or short-term stats) should be kept in a transient datastore like Memcache

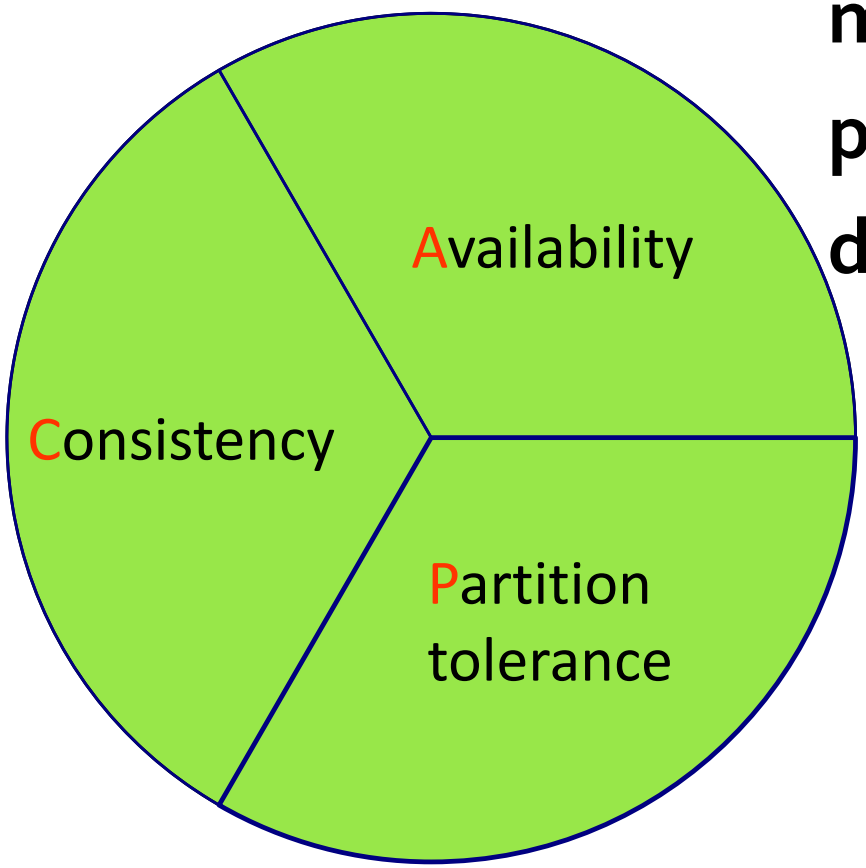
If you need to be able to replicate your data set to multiple locations (such as syncing a music database between a web app and a mobile device), you'll want the replication features of CouchDB.

High availability apps, where minimizing downtime is critical, will find great utility in the automatically clustered, redundant setup of datastores like Cassandra and Riak.

CAP

CAP theorema wordt in NoSQL wereld vaak aangehaald als reden om consistency (aCid) af te zwakken in gedistribueerde omgeving

Het CAP Theorema



Theorema: You can have at most two of these properties for any shared-data system

Betekenis

Consistentie: zie Gegevensbanken 1 en transacties

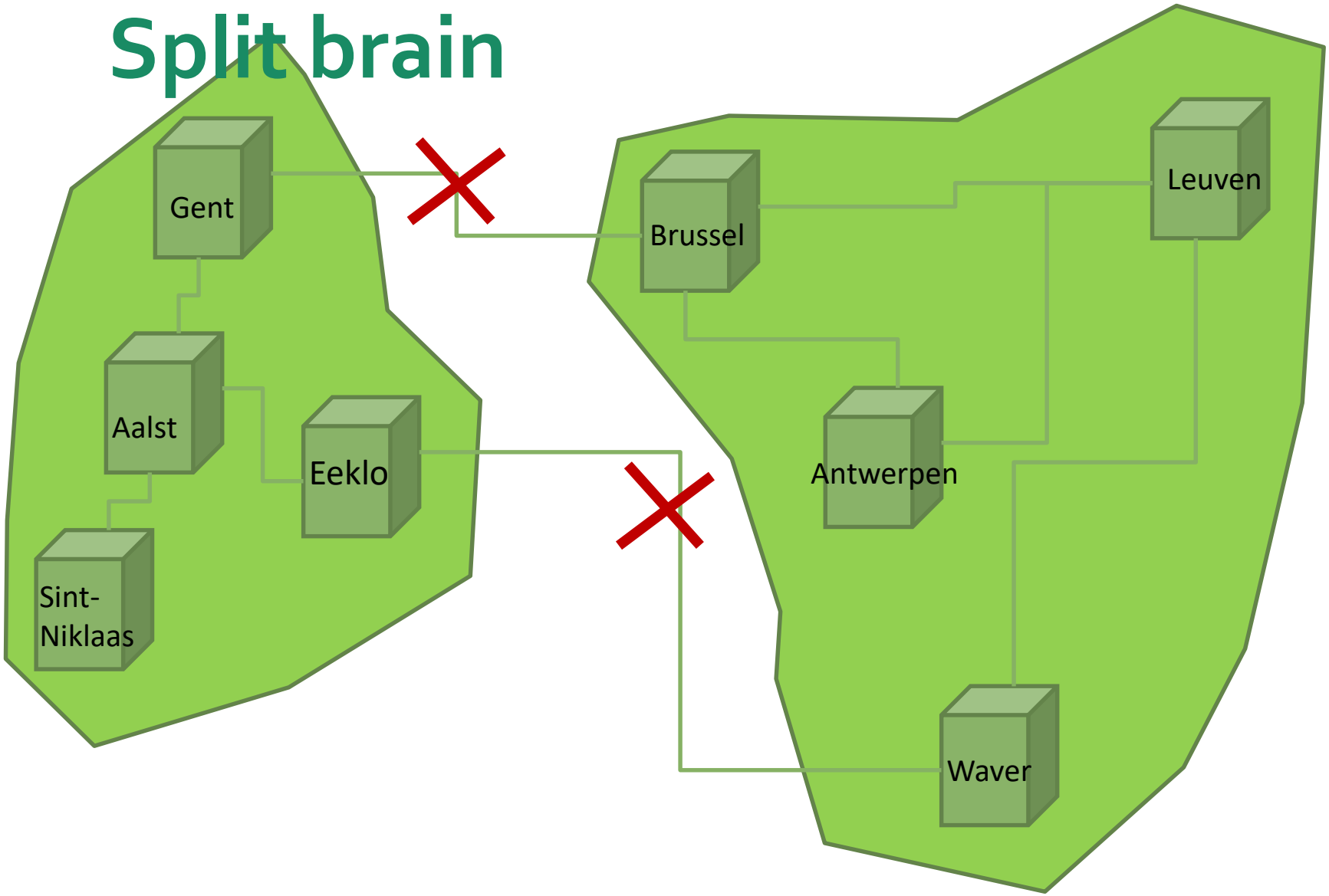
Availability: eigen betekenis in CAP

- "Elke vraag ontvangen van een niet-falende node in de cluster, moet resulteren in een antwoord"

Partitie tolerantie:

- Cluster kan onderbrekingen in de communicatielijnen overleven die de cluster in verschillende partities verdelen, waardoor onderdelen van cluster niet met elkaar kunnen communiceren
- Ook soms split brain genoemd

Split brain


Gent
Aalst
Eeklo
Sint-
Niklaas
Brussel
Leuven
Antwerpen
Waver

CAP

Databanksysteem die op één server draait:

- CA, dus geen partitie tolerantie, aangezien er geen partities kunnen zijn
- Typisch voorbeeld: meeste RDBMS

Praktische betekenis van het theorema:

- In een systeem dat onderhevig is aan partities (zoals gedistribueerde systemen) moet je consistency afwegen tov beschikbaarheid
- Voorbeeld: beetje aan consistency inboeten, om een beetje beschikbaarheid terug te krijgen
- Systeem is dan niet perfect consistent, noch perfect beschikbaar, maar het is een combinatie

CAP: voorbeeld

Bob (in Londen) en Alice (in Gent) willen beiden de laatste hotelkamer boeken

Met een systeem dat peer-to-peer distributie gebruikt met 2 nodes

- 1 in Londen
- 1 in Gent

Consistentie

- Als Bob kamer boekt, moet Londen node met Gent node communiceren alvorens boeking te bevestigen

CAP: voorbeeld

Stel geen netwerkverbinding

- Ook geen boeking meer mogelijk
- Dus geen beschikbaarheid meer

Mogelijke oplossing

- één van de nodes master maken, en alle boekingen moeten via master node gebeuren
- Als master node in Gent is, dan kan Gent node nog altijd boekingen doen (indien er geen netwerk meer is) en Alice heeft laatste hotelkamer

CAP: voorbeeld

Met deze oplossing

- Londen node kan geen hotelkamer boeken

In CAP termen: falen in beschikbaarheid

- Bob kan spreken met de Londen node, maar Londen node kan niet updaten

Alternatieve oplossing (om beschikbaarheid te verhogen):

- Beide systemen kunnen reserveringen blijven accepteren, zelfs als netwerk wegvalt
- Gevaar: laatste hotelkamer meerdere keren boeken
  - Maar meestal is er een kleine overboeking bij hotels
  - Of men excuseert zich later bij één klant voor overboeking
    - Deze kost is lager dan boekingen mislopen door netwerkfalen

CAP: voorbeeld

Ander voorbeeld van inconsistente "writes" naar databank

- Online winkelkarretje
  - Als klant kan je altijd wegschrijven naar winkelkarretje, zelfs bij netwerkfalen
  - Stel door falen: meerdere winkelkarretjes
  - Unie nemen: duplicaten gaan eruit
  - En nadien: laten checken door klant

CAP: wat hebben we geleerd?

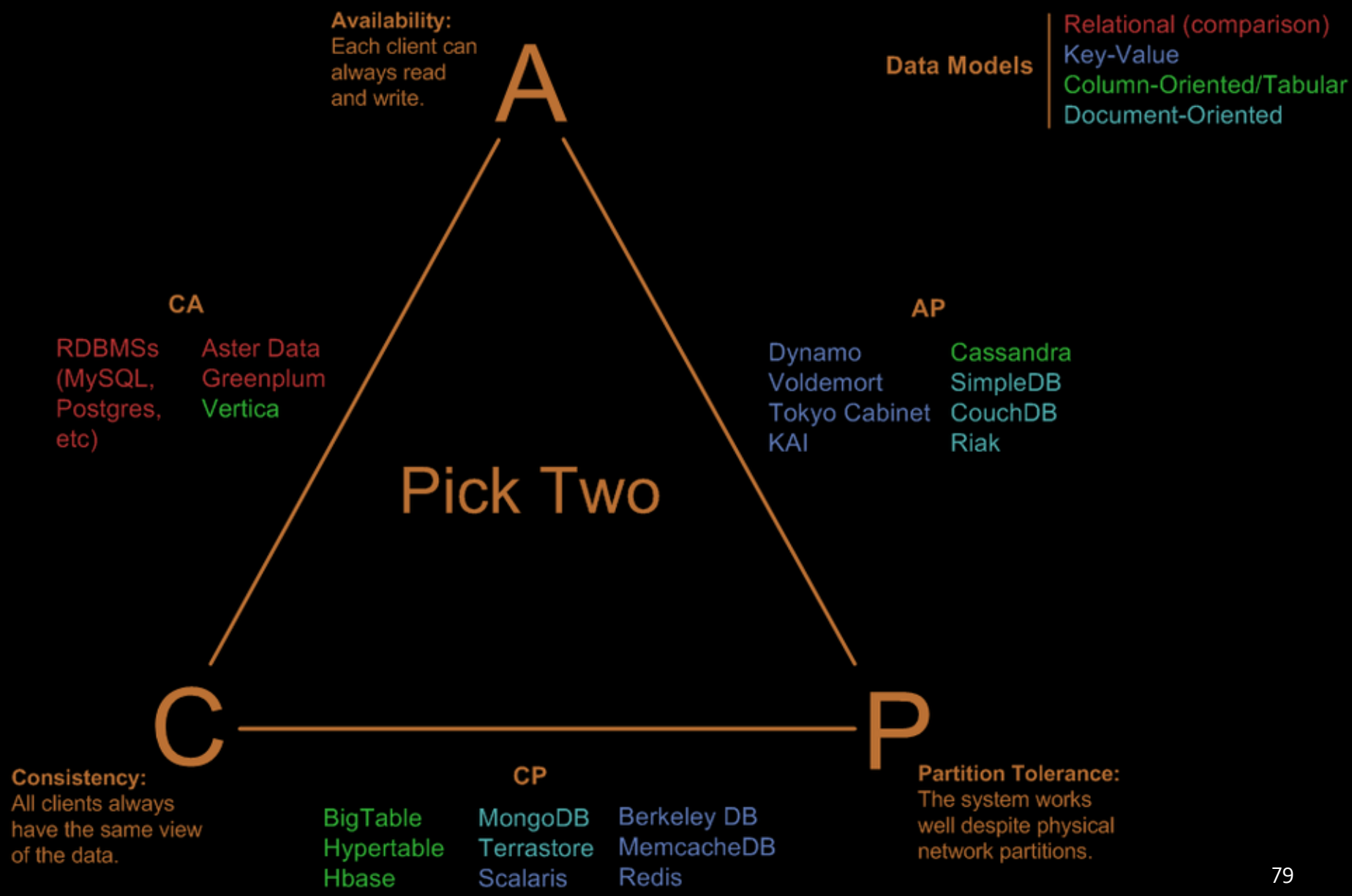
Typisch

- Voor consistentie zorgen bij updaten van data

In sommige gevallen:

- Kan je inconsistentie elegant oplossen
- Als je manier vindt om inconsistente update op te lossen => geeft je meer opties om beschikbaarheid en performantie te verbeteren

Visual Guide to NoSQL Systems



MAP-REDUCE

MAP-REDUCE

Wanneer er gewerkt op een cluster

- Proberen om dataverkeer tussen nodes verminderen, door zoveel mogelijk uitvoeren op de node waar de data zich bevindt

Map-reduce patroon is een manier om dit te bekomen

Geïmplementeerd in Apache Hadoop

Voorbeeld: Map-Reduce (Google)

Map-Reduce:

- Parallel programmeermodel dat toelaat om grote datasets op een cluster van computers gedistribueerd te verwerken

Mosterd werd gehaald bij functionele programmeertalen

Daar worden de functies Map en Reduce vaak gebruikt (zie ook volgend jaar in Advanced Programming)

Voorbeeld Map

Map functie voert een functie of operatie uit op elk element uit een lijst

- Stel: functie `vermenigvuldig_maal_twee op lijst [1,2,3,4]`
- Leidt tot een andere lijst `[2,4,6,8]`

In functioneel programmeren wordt oorspronkelijk lijst niet aangepast (immutable) en data wordt niet gedeeld tussen processen of threads

Ons voorbeeld mapfunctie kan dus zonder problemen uitgevoerd wordt door 2 of meer threads, zonder in elkaars weg te lopen

Voorbeeld Reduce

Reduce functie is in functioneel programmeren gekend als een Fold functie

Reduce past een functie toe op alle elementen van een lijst en geeft één enkel getal terug

Bijv: Reduce (hier bijv. som nemen) toepassen op ons voorbeeld zal 20 teruggeven

MapReduce

Deze ideëen zijn door Google toegepast op grote datasets

Eenvoudig voorbeeld: verzameling van key-value paren

```
[{“9031”:”Peter”}, {“9031”:”Johan”}, {“9820”: “Kristien”}, {“9260”: “Davy”}]
```

Na Map:

```
[{“9031”:[”Peter”,”Johan”]}, {“9820”: [“Kristien”]}, {“9260”: [“Davy”]}]
```

Na Reduce:

```
[{“9031”:2}, {“9820”: 1}, {“9260”: 1}]
```

HBase

Hadoop: gedistribueerd bestandssysteem

Map-Reduce: raamwerk voor gedistibueerd programmeren

Hbase: databanksysteem dat gebouwd is boven Hadoop

Wanneer HBase gebruiken?

First, make sure you have enough data. If you have hundreds of millions or billions of rows, then HBase is a good candidate. If you only have a few thousand/million rows, then using a traditional RDBMS might be a better choice due to the fact that all of your data might wind up on a single node (or two) and the rest of the cluster may be sitting idle.

Second, make sure you can live without all the extra features that an RDBMS provides (e.g., typed columns, secondary indexes, transactions, advanced query languages, etc.) An application built against an RDBMS cannot be "ported" to HBase by simply changing a JDBC driver, for example. Consider moving from an RDBMS to HBase as a complete redesign as opposed to a port.

Referentie:
http://hbase.apache.org/book/architecture.html#arch.overview

Referenties

http://tweakers.net/reviews/2354/1/nosql-maar-wat-is-het-dan-wel-inleiding.html

http://readwrite.com/2009/02/12/is-the-relational-database-doomed

http://readwrite.com/2010/08/26/lhc-couchdb

https://blog.heroku.com/archives/2010/7/20/nosql

http://blog.nahurst.com/visual-guide-to-nosql-systems

http://www.youtube.com/watch?v=qI_g07C_Q5I

NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence, Sadalage & Fowler, 2013, 0-321-82662-0

http://code.msdn.microsoft.com/Getting-started-with-37dbd5bd

http://blog.mongolab.com/2012/08/why-is-mongodb-wildly-popular/