



AARHUS UNIVERSITET

# Microservices and DevOps

DevOps and Container Technology

NoSQL

Henrik Bærbak Christensen

- Relational Databases has ruled the ‘persistent storage’ universe for decades
  - XML and Object-oriented databases were hot – and fizzled out...
- Why did NoSQL databases then succeed?
  - RDB’s did not scale well for massive, web, data
    - Unstructured and evolving data
    - Massive amounts of data required *scaling fast*
    - Consistency makes RDB’s slow and increase likelihood of failures
  - Adopted by Google, Twitter, Facebook...
    - BigTable (Google) began around 2004

# Key features

- NoSQL: "Not Only SQL" / Not Relational
  - Horizontal scaling of *simple* operations over many servers
  - Replication and partitioning data over many servers
  - Simple call level interface (contrast: SQL)
  - Weaker concurrency model than ACID
  - Efficient use of RAM and dist. indexes for storage
  - Ability to dynamically add new attributes to records
- Architectural Drivers:
  - Performance
  - Scalability

[Cattell, 2010]

# Clarifications

- NoSQL focus on
  - *Simple operations*
    - Key lookup, read/write of one or a few records
    - Opposite: Complex joins over many tables (SQL)
    - NoSQL joins are handled **client side !!!**
      - Why is the argument for that?
  - *Horizontal scaling*
    - Many servers with no RAM nor disk sharing
      - Any server may serve a read request => balances load
    - Commodity hardware
      - Cheap but more prone to failures

# NoSQL DB Types



# Basic types

- Four types
  - Key-value stores
    - Memcache, Riak, **Redis**, ...
  - Document stores
    - **MongoDB**, ...
  - Extensible record (Hu: Column)
    - Cassandra, Google BigTable
  - Graph stores
    - Neo4J



# Overview by Hu

Data Model	Name	Producer	Data Storage	Concurrency Control	CAP Option	Consistency
Key-Value	Dynamo	Amazon	Plug-in	MVCC	AP	Eventually Consistent
	Voldemort	LinkedIn	RAM	MVCC	AP	Eventually Consistent
	Redis	Salvatore Sanfilippo	RAM	Locks	AP	Eventually Consistent
Column	BigTable	Google	Google File Systems	Locks + stamps	CP	Eventually Consistent
	Cassandra	Facebook	Disk	MVCC	AP	Eventually Consistent
	Hbase	Apache	HDFS	Locks	CP	Eventually Consistent
	Hypertable	Hypertable	Plug-in	Locks	AP	Eventually Consistent
Document	SimpleDB	Amazon	S3 (Simple Storage Solution)	None	AP	Eventually Consistent
	MongoDB	10gen	Disk	Locks	AP	Eventually Consistent
	CouchDB	Couchbase	Disk	MVCC	AP	Eventually Consistent
Row	PNUTS	Yahoo	Disk	MVCC	AP	Timeline consistent

# Key-value

- Basically the Java `Map<KeyType, ValueType>` datastructure:
  - `Map.put("pid01-2012-12-03-11-45", measurement);`
  - `m = Map.get("pid01-2012-12-03-11-45");`
- Schema: Any value under any key...
- Supports:
  - Automatic replication
  - Automatic sharding
  - *Both using hashing on the key*
- Only lookup using the (primary) key

Often memory based, with periodic flushing to disk. Thus RAM may become a bottleneck!



- Multi-Version Concurrency Control
  - To avoid reading an item while others are updating it
  - To avoid classic concurrency ‘locking’
- Data update
  - No overwrite but...
  - Mark old version as *obsolete*
  - Add new version with timestamp of entry
    - Periodic sweep to erase obsolete data
  - *Point in time* consistent read view
    - *Read with a timestamp*

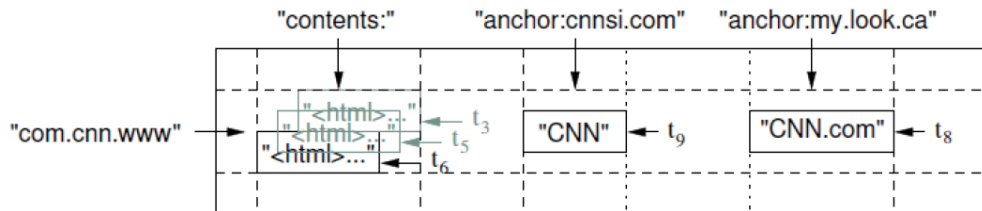
# Document

- Stores "documents"
  - MongoDB: JSON objects.
  - Stronger queries, also in document contents
  - Schema: Any JSON object may be stored!
  - Atomic updates, otherwise *no concurrency control*
- Supports
  - Master-slave replication, automatic failover and recovery
  - Automatic sharding
    - Range-based, on shard key (like zip-code, CPR, etc.)

```
> db.movies.findOne();
{
  "_id" : 1,
  "title" : "Toy Story (1995)",
  "genres" : [
    "Animation",
    "Children's",
    "Comedy"
  ]
}
```

# Extensible Record/Column

- First kid on the block: Google BigTable
- Datamodel
  - Table of rows and columns
  - Scalability model: splitting both on rows and columns
  - Row: (Range) Sharding on primary key
  - Column: Column groups – domain defined clustering
  - No Schema on the columns, change as you go
  - Generally *memory-based* with periodic flushes to disk



- Neo4J

# (Graph)

latest

Graph Setup:

```

create matrix1={id : '603', title : 'The Matrix', year : '1999-03-31'},
matrix2={id : '604', title : 'The Matrix Reloaded', year : '2003-05-07'},
matrix3={id : '605', title : 'The Matrix Revolutions', year : '2003-10-27'},

neo={name:'Keanu Reeves'},
morpheus={name:'Laurence Fishburne'},
trinity={name:'Carrie-Anne Moss'},

matrix1<-[:ACTS_IN {role : 'Neo'}]-neo,
matrix2<-[:ACTS_IN {role : 'Neo'}]-neo,
matrix3<-[:ACTS_IN {role : 'Neo'}]-neo,

matrix1<-[:ACTS_IN {role : 'Morpheus'}]-morpheus,
matrix2<-[:ACTS_IN {role : 'Morpheus'}]-morpheus,
matrix3<-[:ACTS_IN {role : 'Morpheus'}]-morpheus,

matrix1<-[:ACTS_IN {role : 'Trinity'}]-trinity,
matrix2<-[:ACTS_IN {role : 'Trinity'}]-trinity,
matrix3<-[:ACTS_IN {role : 'Trinity'}]-trinity

start n=node:node_auto_index(name="Keanu Reeves")
return n

```

Show 10

entries Search:

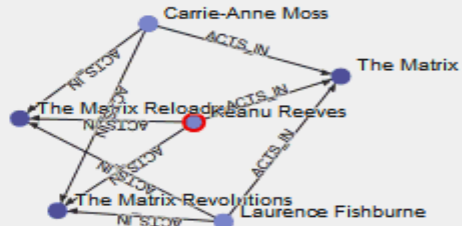
n

```

start n=node:node_auto_index(name="Keanu Reeves")
return n

```

Run



The graph visualization shows the following nodes and relationships:

- Nodes:** Keanu Reeves (red), Laurence Fishburne (blue), Carrie-Anne Moss (blue), The Matrix (blue), The Matrix Reloaded (blue), The Matrix Revolutions (blue).
- Relationships:**
  - Keanu Reeves is connected to The Matrix, The Matrix Reloaded, and The Matrix Revolutions via the relationship `ACTS_IN` with the role `Neo`.
  - Laurence Fishburne is connected to The Matrix, The Matrix Reloaded, and The Matrix Revolutions via the relationship `ACTS_IN` with the role `Morpheus`.
  - Carrie-Anne Moss is connected to The Matrix, The Matrix Reloaded, and The Matrix Revolutions via the relationship `ACTS_IN` with the role `Trinity`.

# CAP Theorem

# Reviewing ACID

- Basic RDBM teaching talks on ACID
- **Atomicity**
  - Transaction: All or none succeed
- **Consistency**
  - DB is in valid state before and after transaction
- **Isolation**
  - N transactions executed concurrently = N executed in sequence
- **Durability**
  - Once a transaction has been committed, it will remain so (even in the event of power loss, crash)

- Eric Brewer: *only get two of the three in scaled system:*
- **Consistency**
  - Reads and writes see the same valid and consistent state
    - "Every read receives the most recent write or an error" [wikipedia]
- **Availability**
  - Able to serve when it's needed
    - "Every request receives a (non-error) response – without the guarantee that it contains the most recent write"
- **Partition tolerance**
  - Operation will complete, even if components are unavailable
    - "The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes"

# Horizontal Scaling: P taken

- We have already taken P, so we have to relax either
  - **C**onsistency
  - **A**vailability
- RDBM prefer *consistency* over *availability*
- NoSQL prefer *availability* over *consistency*
  - *replacing it with eventual consistency*



# Achieving ACID

- Two-phase Commit
  1. All partitions *pre-commit*, report result to master
  2. If success, master tells each to *commit*; else roll-back
- Guaranty consistency, but availability suffer
- Example
  - Two partitions, 99.9% availability
    - $\Rightarrow 99.9^2 = 99.8\%$  (+43 min down every month)
  - Five partitions:
    - 99,5% (36 hours down time in all)

- Replace ACID with BASE
- BA: **B**asically available
- S: **S**oft state
- E: **E**ventual consistent
- Availability achieved by *partial failures* not leading to *system failures*
  - In two-phase commit, what would master do if one partition does not respond?

# Eventual Consistency

- So what does this mean?
  - Upon write, an immediate read may retrieve the old value – or not see the newest added item!
    - Why? Gets data from a replica that is not yet updated...
  - Eventual consistency:
    - *Given sufficiently long period of time which no updates, all replicas are consistent, and thus all reads consistently return the same data...*
- System always available, but state is 'soft' / cached

- Web applications
  - Is it a problem that a facebook update takes some minutes to appear at my friends?
- As Newman points out [p. 236]
  - Consistency is a ‘computerish’ thing, *usually the real world is pretty full of issues that invalidate consistency anyway*
    - While picking the The Brakes album from inventory of 100 records, the staff knocks on album on the floor and accidentally breaks it. Computer says 99 copies left, inventory contains 98. Who is right?
  - And real world counterparts have correcting mechanisms
    - *Sorry, we cannot deliver at the promised date...*

# Summary

- NoSQL – born to address real issues arriving from modern web applications
  - Horizontal scaling on commodity hardware
  - Sacrifices Consistency for Availability
    - CAP theorem states that this is the trade-off
    - Solution: BASE rather than ACID
      - Or rather ‘eventual consistency’
  - Types
    - Key-value, documents, column
  - Schema less – or rather dynamic schemas