

IN3020&4020 – Database Systems (2022)

Part 3: NoSQL

Lecture 24: Overview

25 April

Egor V. Kostylev

IFI, University of Oslo

egork@ifi.uio.no



Big Picture (reminder)

Syllabus of the course:

- (Extended) Intro and SQL recap ✓

Part 1. Query processing in relational databases ✓

Part 2. Transaction management in relational databases ✓

- Database security ✓

Part 3. **NoSQL DBMS**

- Advanced DBMS architecture (IN4020 only)

Lecture schedule

#	Date	Part	Topic
...	...	...	...
23	21.04 (Thu)	(no part)	Database security
24	25.04 (Mon)	Part 3: NoSQL	Overview
25	28.04 (Thu)	Part 3: NoSQL	Graph Databases
26	02.05 (Mon)	Part 3: NoSQL	RDF and SPARQL
27	05.05 (Thu)	Part 3: NoSQL	Neo4j and Cypher
28	09.05 (Mon)	for IN4020	Advanced DBMSs (Digital? V. Goebel?, IN5040)
29	12.05 (Thu)	Wrap-up	Summary

Exam: Monday, 30 May (see next slide)

Materials to read about NoSQL

1. *Fundamentals of Database Systems* by R. Elmasri and S.B. Navathe (7th edition):
Section 24 (and 25) (close to my presentation)
2. (English) Wikipedia
3. Systems' tutorials (MondoDB, SPARQL, etc.)
4. etc.

1. Introduction and classification
2. Document-based systems
3. Key-valued stores
4. Column-based systems
5. Graph-based (a quick look)

1. Introduction and classification

NoSQL: Intro

New main-line **types of applications** (data science, social media) impose **new requirements** upon the underlying systems (including DBMS)

NoSQL (**Not only** SQL) serves these needs

Most **NoSQL** systems are **distributed** over several nodes

Main **challenges**:

- **semi-structured** data (no or little schema)
- **high performance**
- data **availability** and **replication** (several unreliable nodes)
- **scalability** (as opposed to immediate consistency)
- **sharding** and **partitioning** (range portioning)

NoSQL and Big Data

NoSQL systems focus on **Big Data**:

- data sets too **large** or **complex** for traditional data-processing
- **challenges**: capturing data, data storage, data analysis, search, sharing, transfer, visualisation, querying, etc.
- **V-characteristics**:
 - **Volume** (currently, the size 10^{15}B – 10^{18}B),
 - **Velocity** (speed of creation, ingestion, and processing)
 - **Variety** (various types)
 - (Veracity, Value, Variability, etc.)

Some typical **applications** that use NoSQL:

- **Social media** (**Facebook**, etc.)
- **Web links** (**Google** search)
- **Marketing** and **sales** (**Amazon**, etc.)
- **Interactive maps** (**Google maps**, etc.)
- **Email** (**Gmail**, etc.)

NoSQL: distributed characteristics

NoSQL characteristics related to **distributed** database systems:

- **Scalability** (horizontal, i.e., adding more distributed data nodes)
- **Availability** via **replication** and **eventual consistency**
(c.f., **serialisable consistency**)
- **Replication models**
 - **Master-slave** (write to one master, propagate to slaves)
 - **Master-master** (different writes, reconciliation)
- **Sharding** of files (one table distributed over several files)
- High performance **data access** (e.g., via indexes)

NoSQL: data model and query language characteristics

NoSQL characteristics related to **data models** and **query languages**:

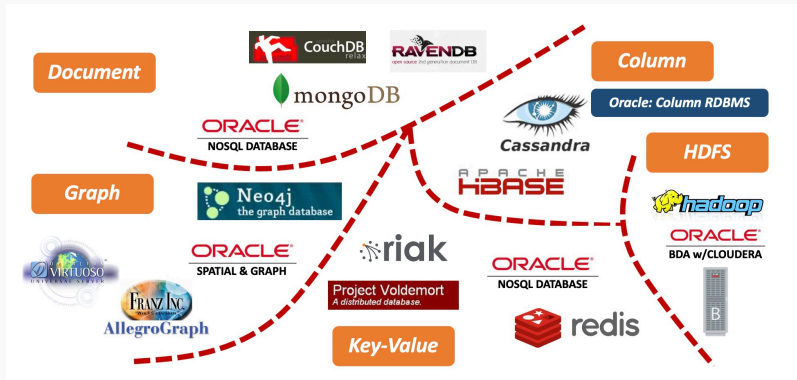
- Schema **not required**
 - data may be **semi-structured**
(e.g., **JSON** and **XML** data formats)
 - **self-describing** data ('schema' as a part of data)
- **Less powerful query** languages:
 - usually **search** rather than **querying** (e.g., no join)
 - **(S)CRUD**: (search), create, read, update, delete
- **Versioning**
 - e.g., timestamps

Categories of NoSQL systems

- Document-based systems: MongoDB, CouchDB, etc.
- Key-value stores: DynamoDB, Voldemort, etc.
- Column-based (or wide-column) systems: HBase, Cassandra, etc.
- Graph-based systems: RDF, Neo4j, etc.
- Hybrid systems: HADOOP, etc.
- (Object databases, XML databases)

NoSQL: Landscape

Some **examples** in the landscape:



CAP theorem

Levels of **consistency** among **replicated** data items

- the **strongest** form: **serialisability**
high **overhead**: can reduce read/write performance

CAP theorem (more like **principle**, since only informally):

The three desired properties
of distributed systems with replicated data:

- **consistency** (among replicas, not as ACID)
- **availability** (for read and write)
- **partition tolerance** (works after a split)

are **not possible** to guarantee simultaneously (in distributed system with data replication)

CAP theorem: Consequences

CAP theorem:

consistency, availability, and partition tolerance
are **not possible** to guarantee simultaneously

NoSQL system designers can choose two of three to guarantee

- **availability** and **partition tolerance** more important
- **weaker consistency** levels is often acceptable
- **eventual consistency** is often adopted

Document-based NoSQL systems

Document-based systems: Idea

Document stores:

Collections of similar documents

Individual documents may be complex objects:

- Documents are complex and self-describing
- Can have different data elements (attributes)

Documents can be specified in various formats

- XML (Extensible Markup Language)
file format for storing, transmitting, and reconstructing arbitrary data
tree-like semi-structured data
- JSON (JavaScript Object Notation)
open standard human-readable file format and data interchange format
attribute-value pairs and arrays

MongoDB data model

MongoDB

- a popular **document-based** approach
- documents stored in **binary JSON (BSON)** format

Individual documents stored in **collections**

- collection creation **example**:

```
db.createCollection("project", { capped : true, size : 1310720, max : 500 } )
```

- **first** parameter specifies the collection name
- **collection options** may include limits
on size and number of documents
- (each document in collection has **unique ID**)

MongoDB data model: CRUD

Collection of documents **does not** have a **schema**

- structure of documents are chosen based on how documents will be accessed

Document **creation** using insert operation (documents in BSON)

```
_ db.<collection_name>.insert(<document(s)>)
```

Document **reading** using find operation with a Boolean condition (and indexing)

```
_ db.<collection_name>.find(<condition>)
```

Document **deletion** using remove operation

```
_ db.<collection_name>.remove(<condition>)
```

MongoDB: Example

Two **collections** (projects and workers):

```
{
  _id:      "P1",
  Pname:    "ProductX",
  Plocation: "Bellaire"
}
{
  _id:      "W1",
  Ename:    "John Smith",
  ProjectId: "P1",
  Hours:    32.5
}
{
  _id:      "W2",
  Ename:    "Joyce English",
  ProjectId: "P1",
  Hours:    20.0
}
```

Note: Different workers can have different keys

Can be achieved via **inserts**:

```
db.project.insert( { _id: "P1", Pname: "ProductX", Plocation: "Bellaire" } )
db.worker.insert( [ { _id: "W1", Ename: "John Smith", ProjectId: "P1", Hours: 32.5 },
                    { _id: "W2", Ename: "Joyce English", ProjectId: "P1",
                      Hours: 20.0 } ] )
```

MongoDB Distributed Systems: Concurrency and Replication

Concurrency in MongoDB:

- Two-phase commit for atomicity and consistency

Replication in MongoDB:

- Replica sets for **multiple copies** on different nodes
- Variation of **master-slave** approach:
 - primary copy, secondary copy, and arbiter
 - (arbiter participates in elections of a new primary if needed)
- All write operations applied to the primary copy and propagated to the secondaries
- User can choose read preference
 - (read requests can be processed at any replica)

Sharding in MongoDB:

- horizontal partitioning divides the documents into disjoint partitions (shards)
- shards stored on different nodes to achieve load balancing
- partitioning field (shard key) must exist in every document in the collection and be indexed
- range or hash partitioning is used

Key-Value NoSQL stores

Key-Value stores: Overview

Key-value stores:

focus on **high performance**, **availability**, and **scalability**
(as opposed to complex querying and search)

Key: **unique identifier** associated with a data item
Used for **fast retrieval**

Value: the **data item** itself
can be **string** or **array of bytes**

No built-in **query language**
application is responsible for searching, querying, etc.

Several similar systems, including

- **DynamoDB** (Amazon, proprietary)
- **Voldemort** (open-source)

DynamoDB:

part of Amazon Web Services and SDK platform (**proprietary**)

Uses its own **terminology**:

- **Table**: collection of items (with no schema)
- **Item**: set of attribute-value pairs
- **Attribute**: simple or complex unique identifier (primary key)
- **Value**: arbitrary data values

Implements **quick access** via hashing and indexing

Voldemort

open-source key-value system similar to **DynamoDB**

focus: horizontal scalability, replication, sharding

Voldemort **features:**

- Simple **basic** CRUD operations (get, put, and delete)
- High-level formatted **data values** (input as **JSON**)
- **Hashing** for distributing (key, value) pairs
- **Consistency and versioning:**
 - **concurrent** writes allowed
 - each read may return **several versions**

Other Key-Value stores

Redis key-value cache and store

- caches data in main memory to improve performance
- master-slave replication and high availability
- persistence by backing up cache to disk

Apache Cassandra

- offers features from several NoSQL categories
- used by Facebook and others

Oracle NoSQL database

- high-level of integration into other Oracle products

etc.

Column-based (wide-column) NoSQL systems

Column-based systems: Overview

Column-based systems:

for storing large amount of data

similar to key-based, but with multi-dimensional keys
(e.g., table name, row key, column info and time-stamp)

BigTable: Google's distributed storage system for big data

- Used in Gmail
- Uses Google File System for data storage and distribution
- (omitted in our presentation)

Apache Hbase: a similar, open-source system

- Uses Hadoop Distributed File System (HDFS) for data storage
- Can also use Amazon's Simple Storage System (S3)

Hbase Data Model

Data organisation concepts:

- Namespace: collection of tables
- Named table: set of self-describing rows with row keys (ordered lexicographically)
- Named column families (associated with tables)
 - created on creation of table and cannot be changed
 - used for grouping together related columns (attributes)
- each column family can be associated with column qualifiers
 - make the model self-describing
 - not specified on creation, can be different in rows
- a column is a combination ColumnFamily:ColumnQualifier

Each column family stored in its own file(s)
using the HDFS file system

Hbase Provides only **low-level CRUD** operations
(Applications can implement more complex operations)

Create operation:

Creates a **new table** (with associated **column families**)

Put operation:

Inserts **new data** (or **new versions**) of existing data items

Get operation:

Retrieves data associated with a **single row**
(most recent by default, but older versions also possible)

Scan operation:

Retrieves **all rows** from a table

Delete operation:

Make an item 'invisible'

HBase: Example

(a) creating a table:

```
create 'EMPLOYEE', 'Name', 'Address', 'Details'
```

(b) Inserting some row data in the EMPLOYEE table:

```
put 'EMPLOYEE', 'row1', 'Name:Fname', 'John'  
put 'EMPLOYEE', 'row1', 'Name:Lname', 'Smith'  
put 'EMPLOYEE', 'row1', 'Name:Nickname', 'Johnny'  
put 'EMPLOYEE', 'row1', 'Details:Job', 'Engineer'  
put 'EMPLOYEE', 'row1', 'Details:Review', 'Good'  
put 'EMPLOYEE', 'row2', 'Name:Fname', 'Alicia'  
put 'EMPLOYEE', 'row2', 'Name:Lname', 'Zelaya'  
put 'EMPLOYEE', 'row2', 'Name:MName', 'Jennifer'  
put 'EMPLOYEE', 'row2', 'Details:Job', 'DBA'  
put 'EMPLOYEE', 'row2', 'Details:Supervisor', 'James Borg'  
put 'EMPLOYEE', 'row3', 'Name:Fname', 'James'  
put 'EMPLOYEE', 'row3', 'Name:Minit', 'E'  
put 'EMPLOYEE', 'row3', 'Name:Lname', 'Borg'  
put 'EMPLOYEE', 'row3', 'Name:Suffix', 'Jr.'  
put 'EMPLOYEE', 'row3', 'Details:Job', 'CEO'  
put 'EMPLOYEE', 'row3', 'Details:Salary', '1,000,000'
```

(c) Some Hbase basic CRUD operations:

Creating a table: create <tablename>, <column family>, <column family>, ...

Inserting Data: put <tablename>, <rowid>, <column family>:<column qualifier>, <value>

Reading Data (all data in a table): scan <tablename>

Retrieve Data (one item): get <tablename>,<rowid>

HBase: Storage

Each Hbase table **divided** into several **regions**

- each region holds a **range** of the row keys in the table
- row keys must be **lexicographically ordered**
- each region has **several stores**
(column families are assigned to stores)

Regions assigned to **region servers** for storage

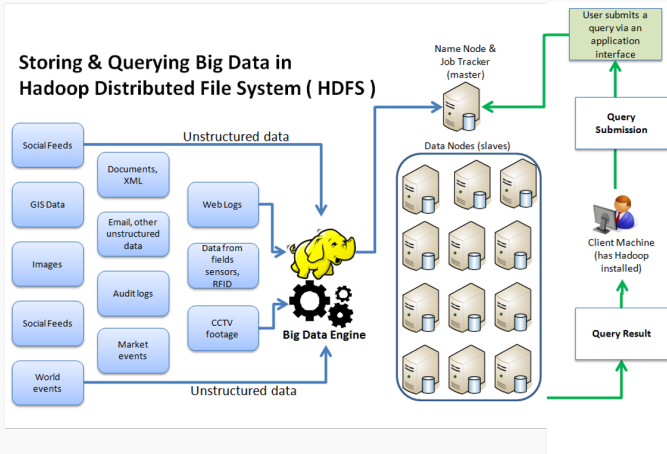
- master server responsible for monitoring the region servers

Hbase uses

- **Apache Zookeeper** (open source system for services related to data naming, distribution, and synchronisation)
- **HDFS (Hadoop Distributed File System)** for file services

Apache HADOOP: Idea

HADOOP: Scalable 'reservoir' for big any-structured data
(based on **MapReduce** model)



Graph databases: A quick look

Next time

What have we learned?

Brief intro to NoSQL

Next time: Graph databases (in detail)