CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

NoSQL Databases

LECTURE 11

Dr. Philipp Leitner

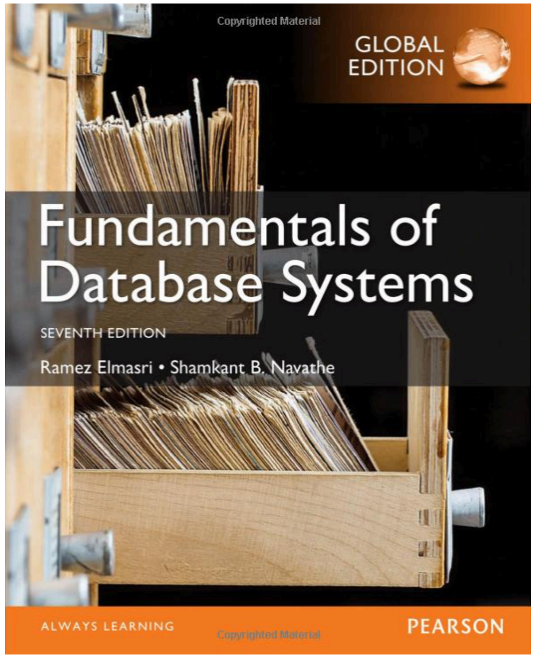
philipp.leitner@chalmers.se

@xLeitix

LECTURE 11

Covers …

Chapter 24



What we will be covering

Basic concepts of alternative databases (NoSQL)

Types of NoSQL databases

Some details on one concrete system
    MongoDB (document store)

Distributed Databases

So far we have never considered the case that a database may actually be **logically distributed** over 2 or more physical hosts

- Your database is actually a bunch of independent processes running on different physical machines

Two common reasons for such distribution:

- **Fault-tolerance** (avoiding a **single point of failure**)
- **Scalability** (if you have to manage enough data, even the largest server can't fit all at once anymore)

These are two **orthogonal issues** and should not be mixed up

Distributed Databases

SQL databases tend to have reasonable mechanisms for replication

… but not so much for scalability

—> NoSQL

Avoiding Single Points of Failure

Avoiding single points of failure (i.e., making the application fault tolerant) means that **the same data needs to be available** in multiple locations

Actually makes it **harder** to manage large volumes of data

Basic principle:

Replication

(mechanism to keep data in sync between different database instances)

Replication Models

Inherent trade-off of replication:

- It's not possible to have two or more DB instances that are always in sync

Common models:

- **Master/slave replication**
  - 1 master (authoritive copy), 1..N slaves (backups, get activated if the master is unavailable)
  - Election procedure is used to promote one slave to master of master becomes unavailable
- **Master/master replication**
  - Multiple masters, voting is used to resolve inconsistencies

Dealing with data at scale

Dealing with data at scale

- Imagine you are a company like Facebook
  - It’s evident that your data won’t all fit into a single computer

Dealing with data at scale

Imagine you are a company like Facebook

- It's evident that your data won't all fit into a single computer

Basic mechanism for dealing with data at scale: **sharding**

- Fancy name for splitting up data between multiple nodes

Dealing with data at scale

Imagine you are a company like Facebook
- It's evident that your data won't all fit into a single computer

Basic mechanism for dealing with data at scale: **sharding**
- Fancy name for splitting up data between multiple nodes

Like in replication, you have multiple database nodes
Unlike in replication, the goal is **not** that they keep the same data

Sharding strategies

- Basic principle:
  - Map content to hosts (shards) based on a **mapping function**
  - Most common:
    - Select an attribute / field value that all content to map has, figure out the domain of this field, and distribute all possible values of the domain evenly across shards
    - Maybe apply **shard rebalancing** if it turns out that some values / ranges are much more common than others

SQL Databases

So far we have focused on a **fairly narrow** type of database

Highly structured (in tables and rows)
Powerful query language (SQL)
Focus on ACID and consistency

These characteristics are **time-tested** and **valuable** for databases
… but not all database applications need the same features

NoSQL

NoSQL

“Not only SQL” databases

Catch-all term for a wide range of different database models that cover different needs and niches where SQL is not good enough:

For semi- or unstructured data

When we need sharding

…

Common Basic Approaches

Common Basic Approaches

Different NoSQL database are used for different purposes, but some commonalities are:

Common Basic Approaches

Different NoSQL database are used for different purposes, but some commonalities are:

+ :

Common Basic Approaches

Different NoSQL database are used for different purposes, but some commonalities are:

+ :

Semi- or unstructured data (as opposed to rigid RM schema)

Common Basic Approaches

Different NoSQL database are used for different purposes, but some commonalities are:

+ :

  Semi- or unstructured data (as opposed to rigid RM schema)

  Scalability and high performance at scale

Common Basic Approaches

Different NoSQL database are used for different purposes, but some commonalities are:

+ :

Semi- or unstructured data (as opposed to rigid RM schema)

Scalability and high performance at scale

Sharding (distribution of data between multiple physical nodes)

Common Basic Approaches

Different NoSQL database are used for different purposes, but some commonalities are:

+ :

    Semi- or unstructured data (as opposed to rigid RM schema)

    Scalability and high performance at scale

    Sharding (distribution of data between multiple physical nodes)

- :

Common Basic Approaches

Different NoSQL database are used for different purposes, but some commonalities are:

+ :
  - Semi- or unstructured data (as opposed to rigid RM schema)
  - Scalability and high performance at scale
  - Sharding (distribution of data between multiple physical nodes)

- :
  - Eventual consistency (rather than ACID)

Common Basic Approaches

Different NoSQL database are used for different purposes, but some commonalities are:

+ :
    - Semi- or unstructured data (as opposed to rigid RM schema)
    - Scalability and high performance at scale
    - Sharding (distribution of data between multiple physical nodes)

- :
    - Eventual consistency (rather than ACID)
    - No or much less sophisticated query languages

Common Use Cases

Increase in importance of NoSQL tightly linked to the Internet, Web 2.0, and Big Data

Social media
Web links
User profiles
Marketing and sales
Posts and tweets
Road maps and spatial data
Email
…

CAP Theorem

Well-known theoretical result for distributed databases:

CAP Theorem

You can have max. two of the following three properties at the same time in a distributed database

Consistency (all nodes have same copy of data)

Availability for Updates (you can at all times **write** to the database)

Partition tolerance (the system can deal graciously with a nodes being unable to talk for some time)

CAP Theorem

SQL DBMSs prioritize **Consistency** (ACID properties)
… and sacrifice availability for updates and partition tolerance

For instance, in a master/slave RDMS you typically can't write (but maybe read) while a master gets re-elected

CAP Theorem

Most NoSQL systems follow a concept of **eventual consistency**

- You can always read/write to the system, and the system can deal with partitions
- But you don't have a guarantee of consistency between all nodes in practice
  - Asking two replicas for the same data can lead to different results

Strictly speaking:

- **An eventually consistent database is guaranteed to reach a consistent state in finite time**
  - However, in practice there are little guarantees how long it will take
  - And other inconsistencies may come up will the previous ones are being fixed

BASE

Tongue-in-cheek alternative to ACID properties for NoSQL:

BASE

(**b**asically **a**vailable, **s**oft state, **e**ventually consistent)

Classes of NoSQL Database Systems

Key/Value Stores

E.g., Redis, DynamoDB, memcached

Document Stores

E.g., MongoDB, CouchDB

Column-Based Databases

E.g., HBase

Graph Databases

E.g., Neo4J

Classes of NoSQL Database Systems

Key/Value Stores

E.g., Redis, DynamoDB, memcached

Document Stores

E.g., MongoDB, CouchDB

Column-Based Databases

E.g., HBase

Graph Databases

E.g., Neo4J

Plus some hybrids that are not easily classified, such as Apache Cassandra

Key/Value Stores

- Extremely **simple** basic model:
  - A key/value database is a collection of **keys** and **values**
  - **Keys** are
    - (Usually) simple types (numbers, strings)
    - Unique in the entire database
  - **Values** are
    - Arbitrary data
    - (can be literally anything for most Key/Value Stores)

Basically like a Map<String,Object> in Java

Key/Value Stores

Extremely **simple** basic model:

- A key/value database is a collection of **keys** and **values**
- **Keys** are
  - (Usually) simple types (numbers, strings)
  - Unique in the entire database
- **Values** are
  - Arbitrary data
  - (can be literally anything for most Key/Value Stores)

Lookups in Key/Value Stores

Usually **no query support** in Key/Value Stores

Can look up data only **by key**

(but this can be done in *O(1)* complexity)

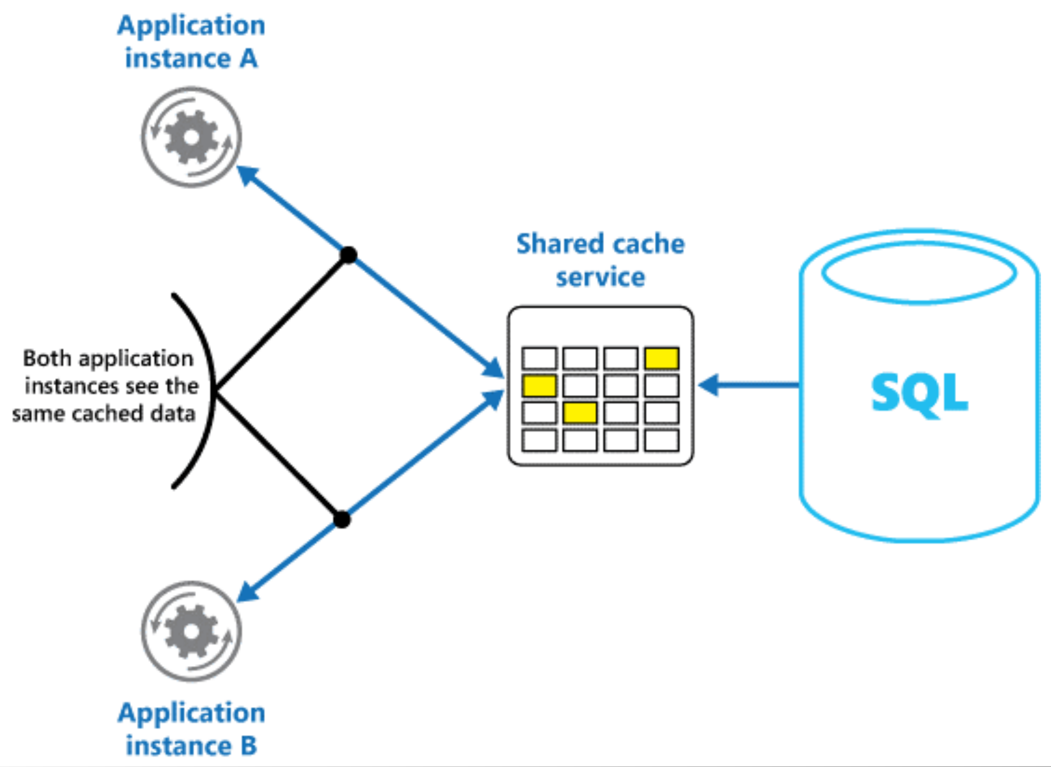
There is no feasible way to search for data where you don't know the key

This makes this model very suitable for one specific use cases:

Distributed Caching

(e.g., keeping session information consistent across multiple Web servers)

Common Use Case of Key/Value Store



Column-Based Databases

Table-based database, similar to SQL

Tables have rows, and each row has a **unique string key**

Tables are defined through **column families**, but different rows may have different concrete attributes for each column family

Data items are **versioned**

- You can retrieve an old version of a value by timestamp

Meant for **Big Data** applications (see next week)

Graph Databases

Data represented as a mathematical **graph**

Collection of vertices (nodes) and edges

Possible to store data associated at nodes and edges

Most useful to store highly interlinked data

E.g., social networks (Twitter data, Facebook interactions, …)

Most important **implementation**:

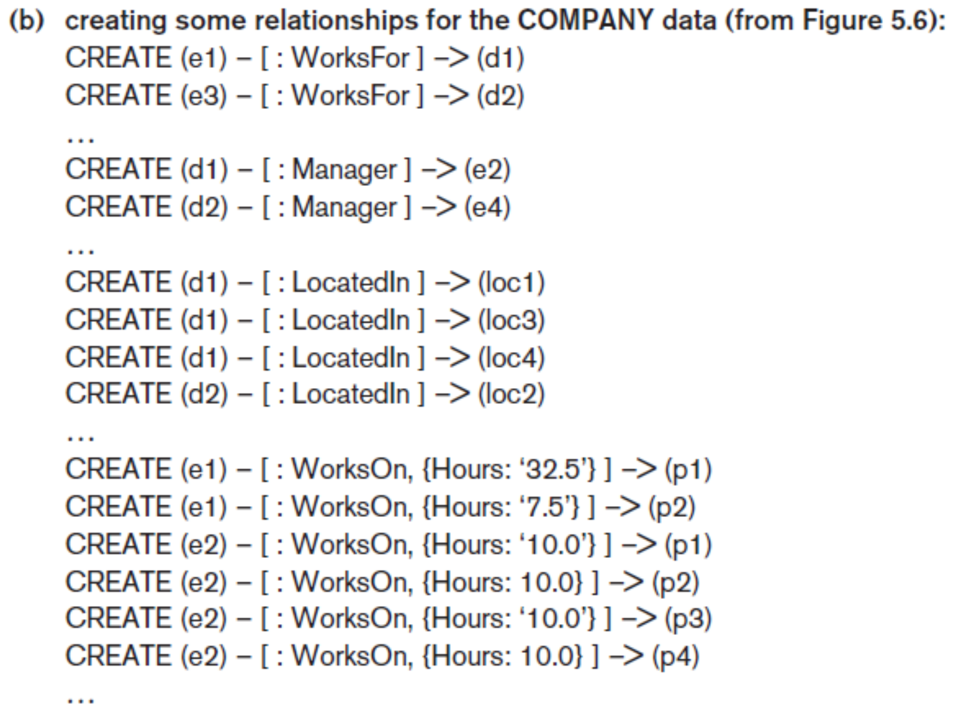
Neo4j

Creating Nodes in Neo4J

(a) creating some nodes for the COMPANY data (from Figure 5.6):

```
CREATE (e1: EMPLOYEE, {Empid: ‘1’, Lname: ‘Smith’, Fname: ‘John’, Minit: ‘B’})
CREATE (e2: EMPLOYEE, {Empid: ‘2’, Lname: ‘Wong’, Fname: ‘Franklin’})
CREATE (e3: EMPLOYEE, {Empid: ‘3’, Lname: ‘Zelaya’, Fname: ‘Alicia’})
CREATE (e4: EMPLOYEE, {Empid: ‘4’, Lname: ‘Wallace’, Fname: ‘Jennifer’, Minit: ‘S’})
…
CREATE (d1: DEPARTMENT, {Dno: ‘5’, Dname: ‘Research’})
CREATE (d2: DEPARTMENT, {Dno: ‘4’, Dname: ‘Administration’})
…
CREATE (p1: PROJECT, {Pno: ‘1’, Pname: ‘ProductX’})
CREATE (p2: PROJECT, {Pno: ‘2’, Pname: ‘ProductY’})
CREATE (p3: PROJECT, {Pno: ‘10’, Pname: ‘Computerization’})
CREATE (p4: PROJECT, {Pno: ‘20’, Pname: ‘Reorganization’})
…
CREATE (loc1: LOCATION, {Lname: ‘Houston’})
CREATE (loc2: LOCATION, {Lname: ‘Stafford’})
CREATE (loc3: LOCATION, {Lname: ‘Bellaire’})
CREATE (loc4: LOCATION, {Lname: ‘Sugarland’})
…
```

Creating Edges in Neo4J


(b) creating some relationships for the COMPANY data (from Figure 5.6):

```
CREATE (e1) – [ : WorksFor ] –> (d1)
CREATE (e3) – [ : WorksFor ] –> (d2)
…
CREATE (d1) – [ : Manager ] –> (e2)
CREATE (d2) – [ : Manager ] –> (e4)
…
CREATE (d1) – [ : LocatedIn ] –> (loc1)
CREATE (d1) – [ : LocatedIn ] –> (loc3)
CREATE (d1) – [ : LocatedIn ] –> (loc4)
CREATE (d2) – [ : LocatedIn ] –> (loc2)
…
CREATE (e1) – [ : WorksOn, {Hours: '32.5'} ] –> (p1)
CREATE (e1) – [ : WorksOn, {Hours: '7.5'} ] –> (p2)
CREATE (e2) – [ : WorksOn, {Hours: '10.0'} ] –> (p1)
CREATE (e2) – [ : WorksOn, {Hours: 10.0} ] –> (p2)
CREATE (e2) – [ : WorksOn, {Hours: '10.0'} ] –> (p3)
CREATE (e2) – [ : WorksOn, {Hours: 10.0} ] –> (p4)
…
```

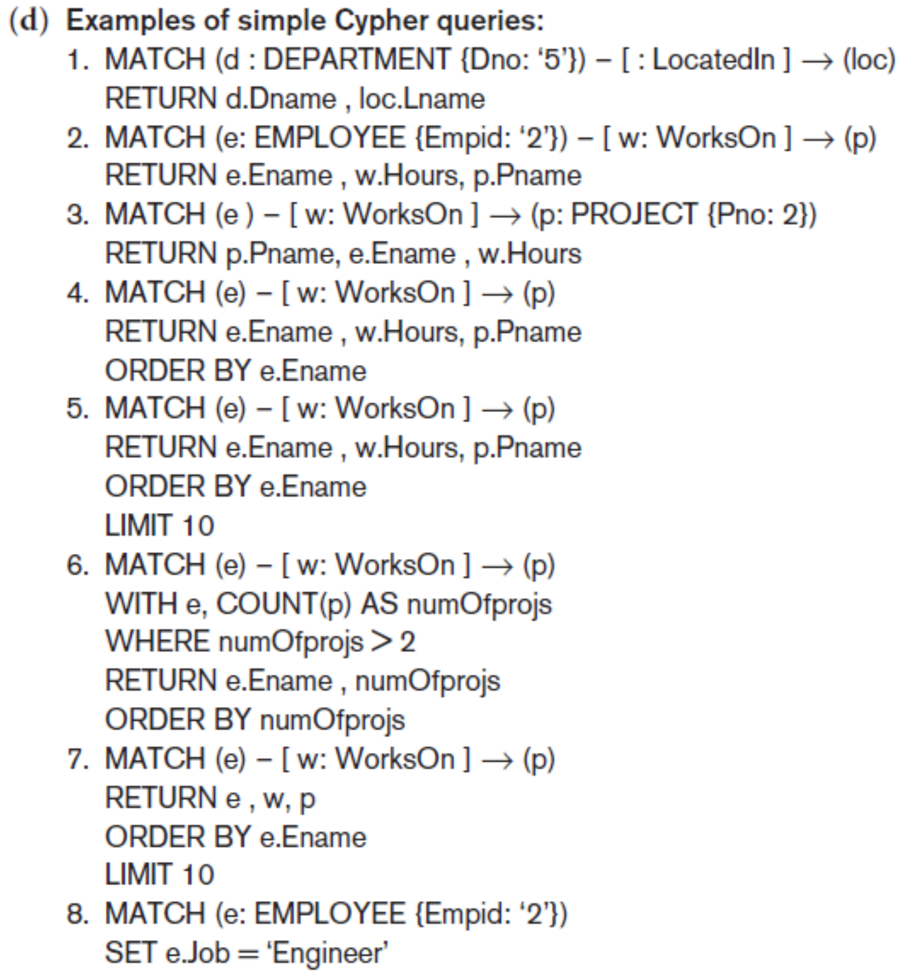
Cypher Query Language

Neo4J has a fairly powerful query language

Queries are defined as **clauses** (restrictions) on the graph

Can be executed quite **efficiently**
(much more efficiently than joins in RDBMs)

Cypher Example



(d) Examples of simple Cypher queries:

```
1. MATCH (d : DEPARTMENT {Dno: ‘5’}) – [ : LocatedIn ] → (loc)
   RETURN d.Dname , loc.Lname
2. MATCH (e: EMPLOYEE {Empid: ‘2’}) – [ w: WorksOn ] → (p)
   RETURN e.Ename , w.Hours, p.Pname
3. MATCH (e ) – [ w: WorksOn ] → (p: PROJECT {Pno: 2})
   RETURN p.Pname, e.Ename , w.Hours
4. MATCH (e) – [ w: WorksOn ] → (p)
   RETURN e.Ename , w.Hours, p.Pname
   ORDER BY e.Ename
5. MATCH (e) – [ w: WorksOn ] → (p)
   RETURN e.Ename , w.Hours, p.Pname
   ORDER BY e.Ename
   LIMIT 10
6. MATCH (e) – [ w: WorksOn ] → (p)
   WITH e, COUNT(p) AS numOfprojs
   WHERE numOfprojs > 2
   RETURN e.Ename , numOfprojs
   ORDER BY numOfprojs
7. MATCH (e) – [ w: WorksOn ] → (p)
   RETURN e , w, p
   ORDER BY e.Ename
   LIMIT 10
8. MATCH (e: EMPLOYEE {Empid: ‘2’})
   SET e.Job = ‘Engineer’
```

Document Stores

A database in a document store consists of **collections** (or bags) of **documents**

Documents are **semi-structured** objects.

- There is no requirement that all documents in a given collection need to share the same structure.

Short Aside On Semi-Structured Data

(Somewhat informal) distinction between different data models:

- (Highly) structured data:
  - There is a **predefined schema** that all data items adhere to. Deviations, if allowed at all, are highly regulated (e.g., NULL values).
  - Example: relational model
- Unstructured data:
  - There are **no rules or assumptions** about what each data item looks like. Two data items can be very similar, or entirely different (a JPG image vs. a freeform text).
  - Example: file system, Key/Value Stores
- Semi-structured data:
  - There is **no strict schema**, but there are rules that govern the structure of data (e.g., needs to be JSON, specific fields need to be available). Documents are often assumed to be **self-descriptive**.
  - Example: document stores

JSON and XML

Two examples of typical formats (representations) for semi-structured data:

JSON (Javascript Object Notation)

XML (eXtensible Markup Language)

We will focus on JSON here, but there is going to be a separate lecture on data representations

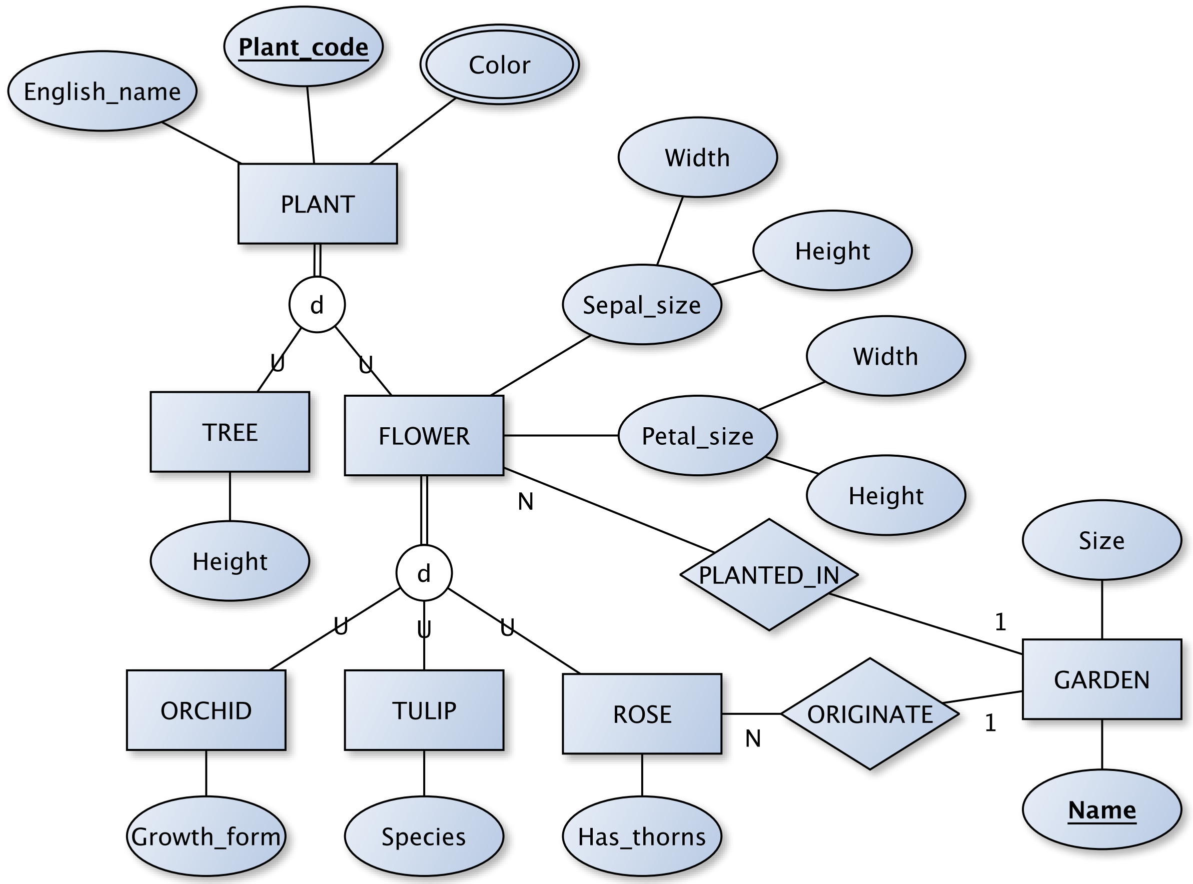
MongoDB

MongoDB is just one implementation of the Document Store model

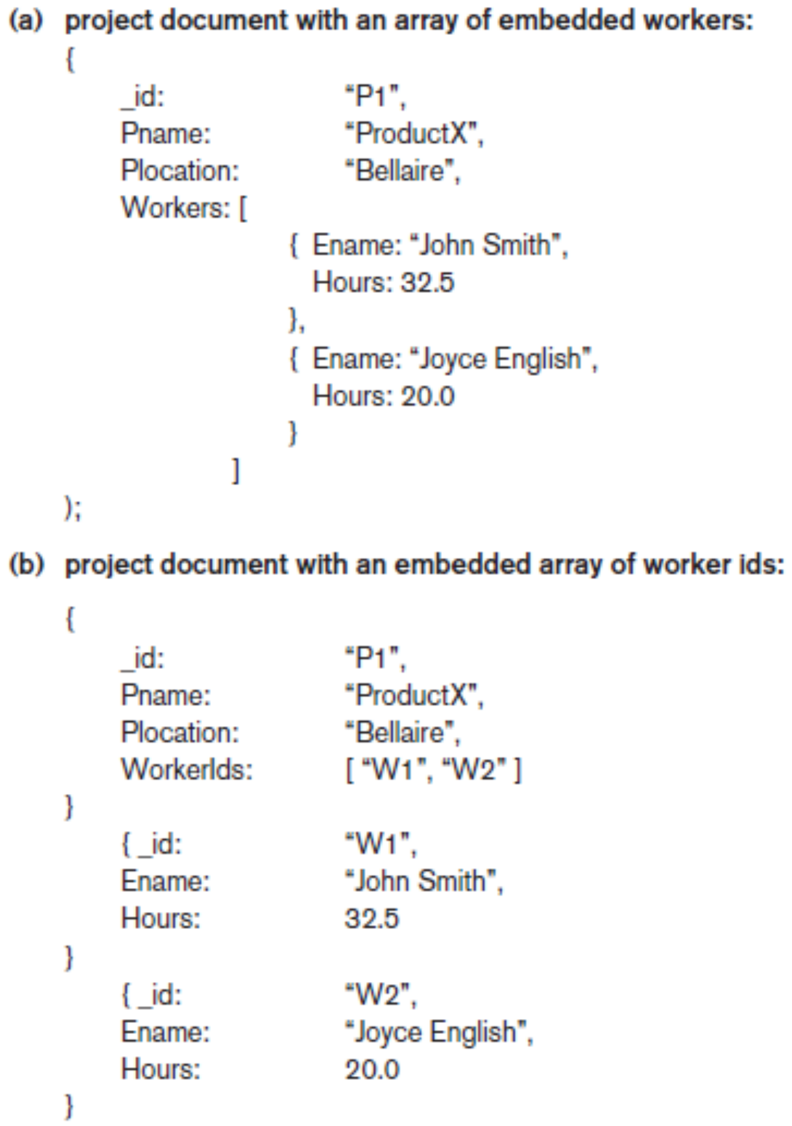
Basic idea:

- Databases contain collections, which in turn contain documents
- Each document is a JSON document
  - Commonly the documents in the same collection will look similar, but this is in no way enforced by MongoDB

Example (from the assignments)



Example JSON



(a) project document with an array of embedded workers:

```
{
    _id:            "P1",
    Pname:          "ProductX",
    Plocation:      "Bellaire",
    Workers: [
                  { Ename: "John Smith",
                    Hours: 32.5
                  },
                  { Ename: "Joyce English",
                    Hours: 20.0
                  }
              ]
);
```

(b) project document with an embedded array of worker ids:

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

Interacting with MongoDB

Interaction with MongoDB follows the CRUD principles:

Creating documents:

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

Removing documents:

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

Query and Update Languages

MongoDB has a fairly comprehensive **language** for querying and updating documents.

Basic principle is **query-by-example:**

- You construct a **partial document** that has the fields that you are interested in set to the value you are interested in, and the query returns all documents with the same fields and values.
- For operations other than equality, separate syntax exists

Conditions

Example of a **find** operation

(this is to be executed in the **Mongo Shell** - this is not Java code!)

```
db.collection.find( { qty: { $gt: 4 } } )
```

Returns all JSON documents that have a field ‘qty’ where the value is larger than 4.

Projections

The **find** operation also supports **projection**.

- Basically a list of fields that the query return document should have.

```
db.collection.find(
   { qty: { $gt: 4 } },
   { qty: true },
)
```

Projections

The **find** operation also supports **projection**.

Basically a list of fields that the query return document should have.

```
db.collection.find(
    { qty: { $gt: 4 } },
    { qty: true },
)
```

Returns the same documents as before, but only the _id and qty fields.

Updating

Updating consists of a **find** with subsequent modification instructions

```
db.books.update(
   { _id: 1 },
   {
     $inc: { stock: 5 },
     $set: { item: "ABC123" }
   }
)
```

Updating

Updating consists of a **find** with subsequent modification instructions

```
db.books.update(
    { _id: 1 },
    {
      $inc: { stock: 5 },
      $set: { item: "ABC123" }
    }
)
```

$set sets one or more fields to a specific value, $inc increases an (integer) field value.

Required Reference for Assignments

https://docs.mongodb.com/manual/reference/operator/

Distributed MongoDB

MongoDB assumes that it is run as a **distributed database**

Variation of Master/Slave replication with three roles:

- **Primary** (master)
- **Secondary** (slave) - can be multiple
- **Arbiter** - does not have a copy of data, but participates in elections of new primaries (tie breaker)

Data can also be sharded based on a **shard key field**

- Specific JSON field that all documents need to have
- Can be "_id" which all documents have anyway, or something else

Querying with Map/Reduce

MongoDB natively supports the execution of **Map/Reduce** queries on collections.

This is particularly useful for highly sharded databases, as all maps can be done on different instances in parallel.

```
db.orders.mapReduce(
                     mapFunction,
                     reduceFunction,
                     { out: "map_reduce_example" }
                   )
```

`mapFunction` and `reduceFunction` are functions defined in JavaScript.

Typical Use Cases

Unlike some other NoSQL databases, Document Stores have shown to be **broadly useful** for a number of different cases:

- To store data that is not inherently very structured
  - Example: user contributions in social media
- To support **schema evolution**
  - It is much easier to deal with program updates (which also often mean data updates) in a Document Store than in SQL

Short 6-Question Kahoot!

Key Takeaways

Even though we focus on SQL databases in this course (and they remain by far the most important database principle to know!), **there are plenty of alternatives** that you should be aware of.

Basic (alternative) models:

- Document Stores
- Key/Value Stores
- Graph Databases
- Document Stores

Key Takeaways

Further important concepts that we learned about:

- Replication (master/slave, master/master)
- Sharding (distribution of data across multiple nodes)
- Eventual consistency and the CAP theorem