NoSQL

Based on Slides from Michael Franklin and Dan Suciu

Big data is not only databases

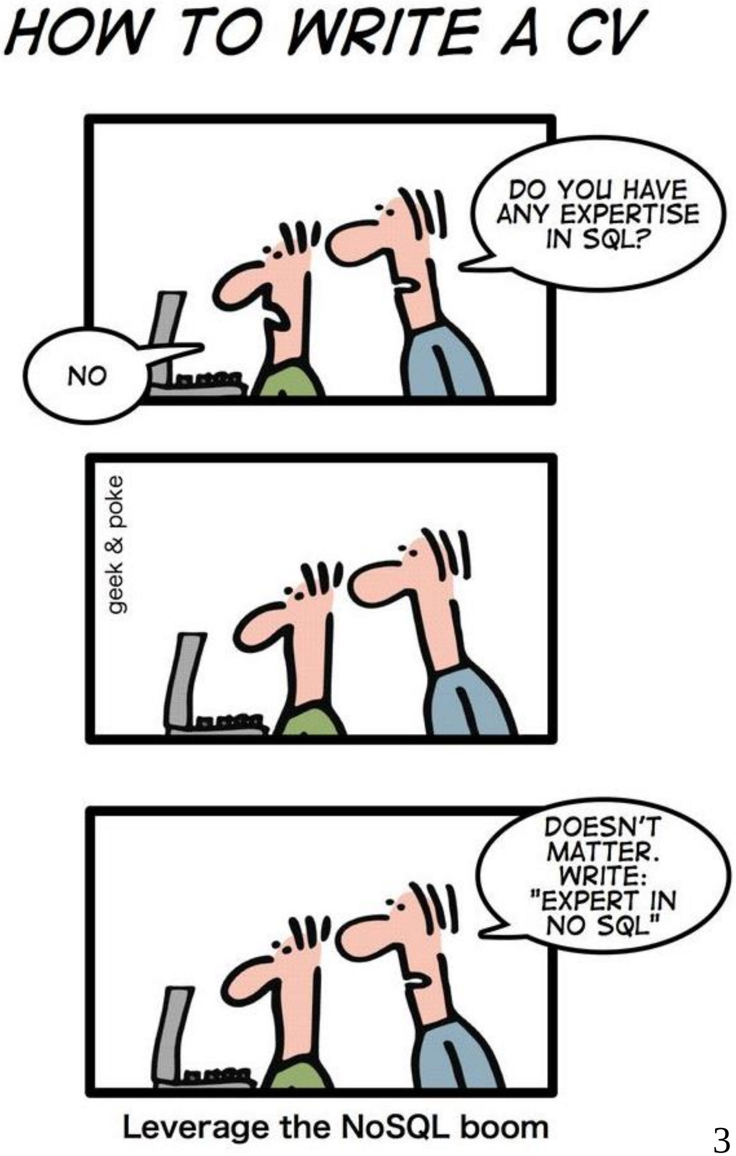
- Big data is more about data analytics and on-line querying

Many components:

- **Storage systems**
- **Database systems**
- **Data mining and statistical algorithms**
- **Visualization**

What is NoSQL?

from “Geek and Poke”



What is NoSQL?

- **An emerging "movement" around non-relational software for Big Data**
- **Roots are in the Google and Amazon homegrown software stacks**

Wikipedia: "A NoSQL database provides a mechanism for storage and retrieval of data that use looser consistency models than traditional relational databases in order to achieve horizontal scaling and higher availability. Some authors refer to them as "Not only SQL" to emphasize that some NoSQL systems do allow SQL-like query language to be used."

Some NoSQL Components

Analytics Interface
(Pig, Hive, …)

Imperative Lang
(RoR, Java,Scala, …)

Data Parallel Processing
(MapReduce/Hadoop)

Distributed Key/Value or Column Store
(Cassandra, Hbase, Redis, …)

Scalable File System
(GFS, HDFS, …)

Query Optimization and Execution

Relational Operators

Access Methods

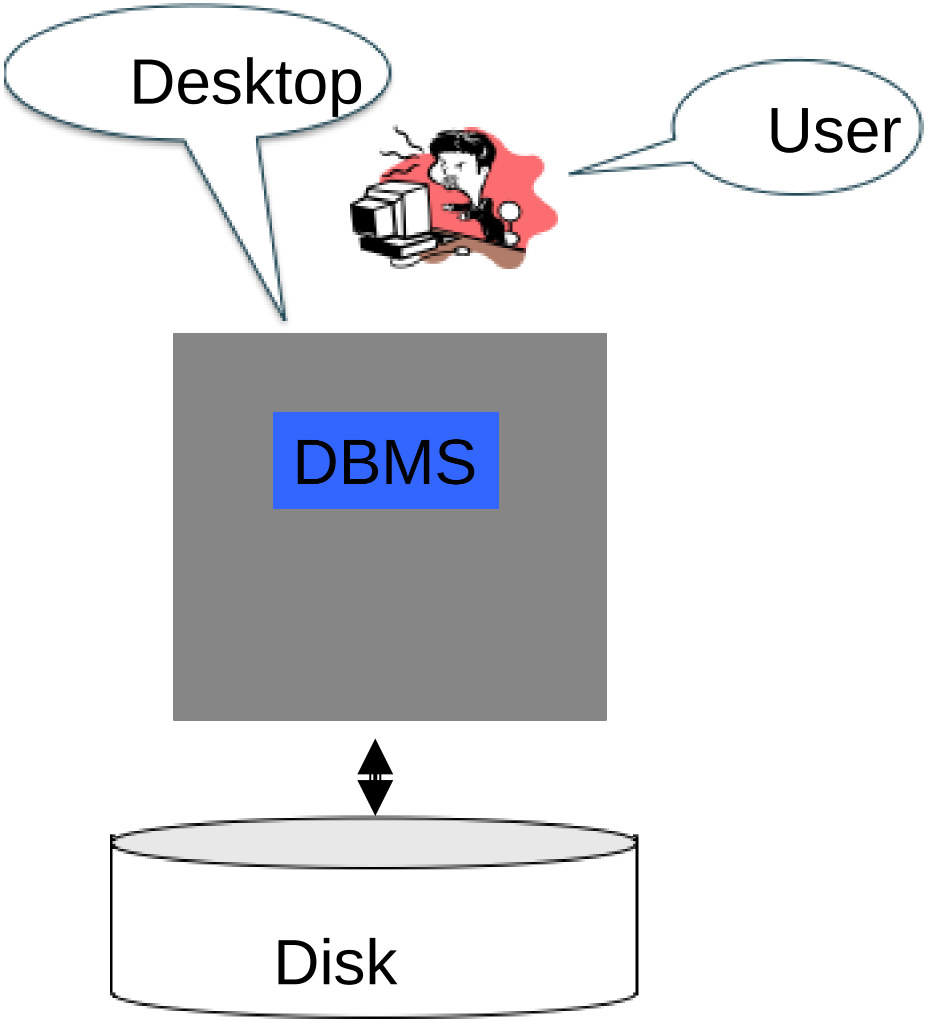
Buffer Management

Disk Space Management

What is the Problem?

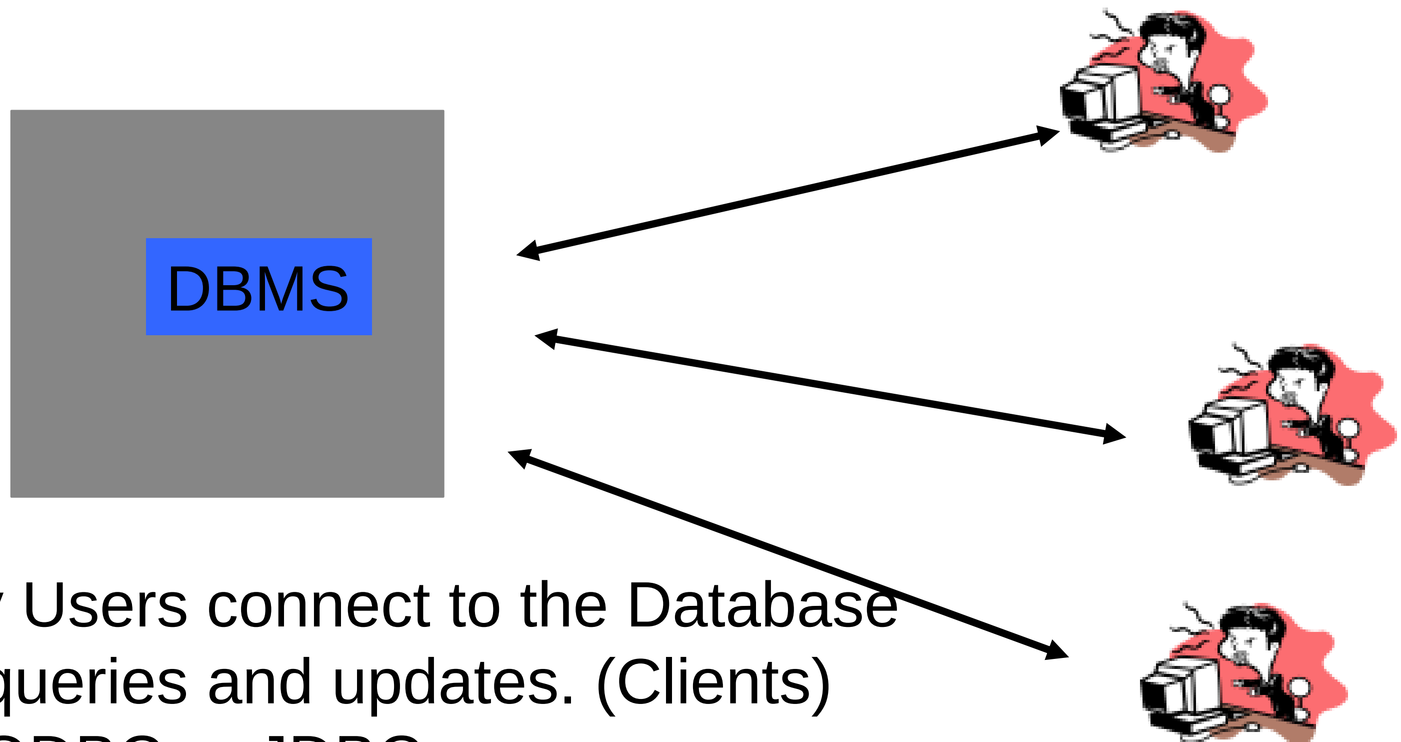
- Single server DBMSs are too small for Web data
- Solution: scale out to multiple servers
- This is hard for the *entire* functionality of DMBS
- NoSQL: reduce functionality for easier scaleup
  - Simpler data model
  - Very restricted updates

No Server



- A simple version of a DBMS works on a single machine with a single Disk and a local user

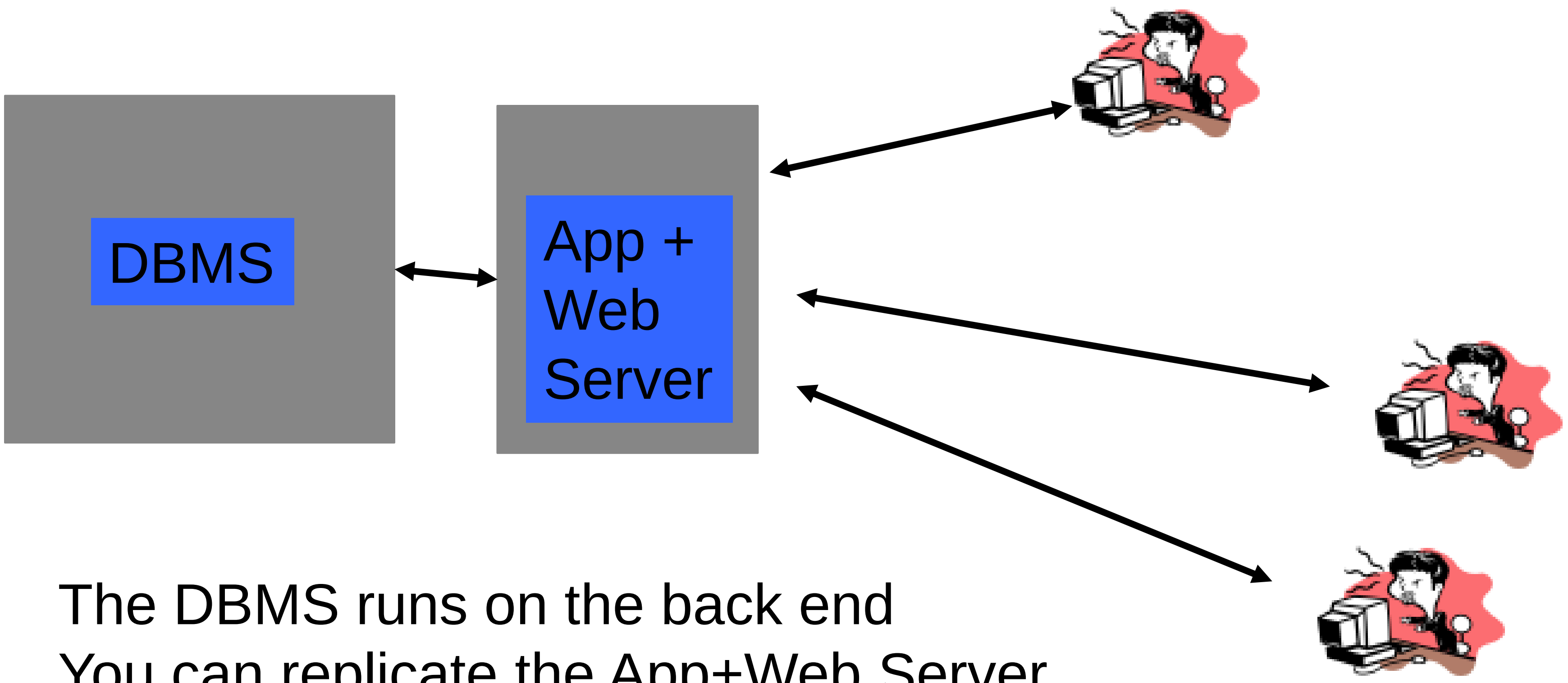
Client-Server



Many Users connect to the Database
with queries and updates. (Clients)
Use ODBC or JDBC
The Server runs the DBMS at

- -your laptop
- -dept server
- -cloud

3 Tier System: Web Apps



The DBMS runs on the back end
You can replicate the App+Web Server
if the number of users increases…
Keep the DBMS consistent using only one system
for it (or as consistent as possible)

Replicating the Database

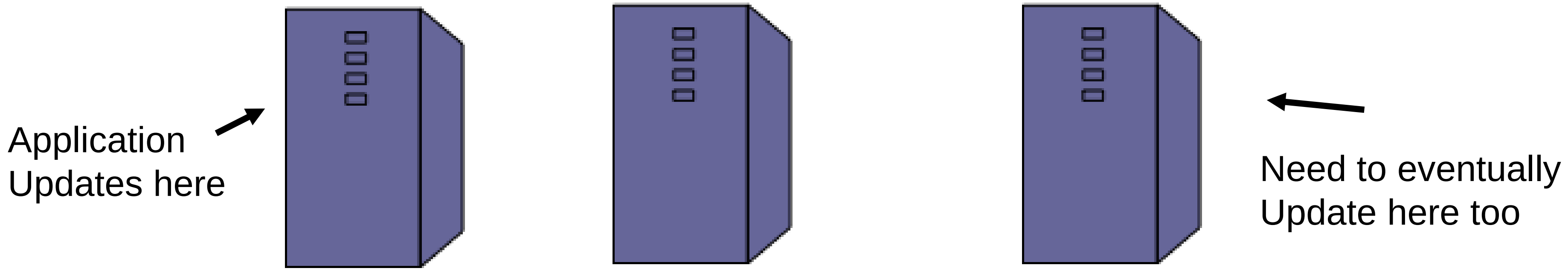
- **How to improve performance for very large databases?**

  - Two basic approaches:
    - Scale up through partitioning
    - Scale up through replication

- **Consistency is much harder to enforce**

Scale Through Replication

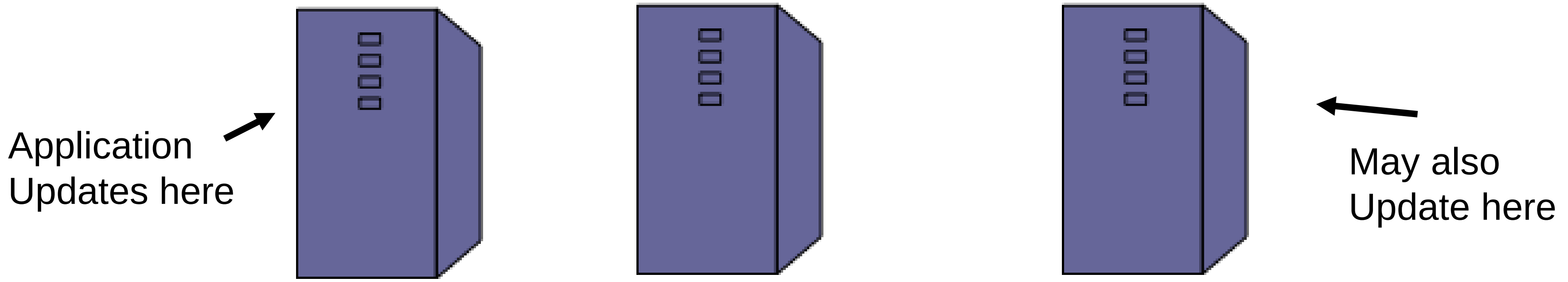
- Create multiple copies of each database partition
- Spread queries across these replicas
- Can increase throughput and lower latency
- Can also improve fault-tolerance
- Easy for reads but writes become expensive!



Three replicas

Scale Through Partitioning

- **Partition the database across many machines in a cluster**
  - Database now fits in main memory
  - Queries spread across these machines
    - Can increase throughput
    - Easy for writes but reads become expensive!



Three partitions

Relational Model → NoSQL

- **Relational DB: difficult to replicate/partition**
- **Given**

  Supplier(sno,...),Part(pno,...),Supply(sno,pno)
  – Partition: we may be forced to join across servers
  – Replication: local copy has inconsistent versions
  – Consistency is hard in both cases (why?)

- **NoSQL : simplified data model**
  - Given up on functionality
  - Application must now handle joins and consistency not the system!

Data Models

Taxonomy based on data models:

- **Key-value stores**
  - e.g., Redis, Memcached, DynamoDB
- **Document stores**
  - e.g., SimpleDB, CouchDB, MongoDB
- **Extensible Record Stores**
  - e.g., HBase, Cassandra, PNUTS

Key-Value Stores Features

- **Data model: (key, value) pairs**
  - Key = string/integer, unique for the entire data
  - Value = can be anything (very complex object)
- **Operations**
  - get(key), put(key,value)
  - Operations on value not supported
- **Distribution/Partitioning–w/hash function**
  - No replication: key k is stored at server h(k)
  - 3-way replication: key k stored at h1(k),h2(k),h3(k)

Example

Flights(fid, date, carrier, flight_num, origin, dest, ...)
Carriers(cid, name)

- **How would you represent the Flights data as (key, value) pairs?**

Option 1: key=fid, value=entire flight record

Option 2: key=date, value=all flights that day

Option 3: key=(origin, dest), value=all flights between

Key-Value Stores Internals

- Partitioning:
  - Use a hash function h
  - Store every (key,value) pair on server h(key)
- Replication:
  - Store each key on (say) three servers
  - On update, propagate change to the other servers; *eventual consistency*
  - Issue: when an app reads one replica, it may be stale

Usually: combine partitioning+replication

Document stores

- Extension of Key-Value stores but the Value is a document (and specific semantics)

- Examples: SimpleDB, CouchDB, MongoDB

Document Stores Features

- **Data model:** (key, document) pairs
  - Key = string/integer, unique for the entire data
  - Document = JSon, BSON, or XML

- **Operations**
  - Get/put document by key
  - Query language over JSon

- **Distribution/Partitioning**
  - Entire documents, as for key/value pairs
- **We will discuss MongoDB soon**

Extensible Record Stores

- Based on Google's BigTable
- Data model is rows and columns
- Scalability by splitting rows and columns over nodes
  – Rows partitioned through sharding on primary key
  – Columns of a table are distributed over multiple nodes by using "column groups"
- HBase is an open source implementation of BigTable

MongoDB (An example of a Document Database)

-Data are organized in **collections.** A collection stores a set of **documents**.

- Collection like table and document like record
  - but: each document can have a different set of attributes even in the same collection
  - Semi-structured schema!
- Only requirement: every document should have an **"_id"** field

  - hu**mongo**us => Mongo

MongoDB

Key features include:

- JSON-style documents
  - actually uses BSON (JSON's binary format)
- replication for high availability
- auto-sharding for scalability
- document-based queries
- can create an index on any attribute for faster reads

JSON

- **JSON is an alternative data model for semi-structured data.**
  - JavaScript Object Notation
- **Built on two key structures:**
  - an object, which is a sequence of name/value pairs

```
{ "id": "1000",
  "name": "Sanders Theatre",
  "capacity": 1000 }
```

  - an array of values ["123", "222", "333"]
- A value can be:
  - an atomic value: string, number, true, false, null
  - an object
  - an array

Example mongodb: JSON

{ "_id”:ObjectId("4efa8d2b7d284dad101e4bc9"),
"Last Name": ” Cousteau",
"First Name": ” Jacques-Yves",
"Date of Birth": ”06-1-1910" },

{ "_id": ObjectId("4efa8d2b7d284dad101e4bc7"),
"Last Name": "PELLERIN",
"First Name": "Franck",
"Date of Birth": "09-19-1983",
"Address": "1 chemin des Loges",
"City": "VERSAILLES" }

JSon vs Relational

- Relational data model
  - –Rigid flat structure(tables)
  - – Schema must be fixed in advance
  - - Binary representation: good for performance, bad for exchange
  - – Query language based on Relational Calculus
- Semistructured data model / JSon
  - – Flexible, nested structure (trees)
  - – Does not require predefined schema("self describing”)
  - – Text representation: good for exchange, bad for performance
  - – Most common use: Language API; query languages emerging

JSon Terminology

- Data is represented in name/value pairs.
- Curly braces hold objects
  - Each object is a list of name/value pairs separated by , (comma)
  - Each pair is a name is followed by :(colon) followed by the value
- Square brackets hold *ordered* arrays and values are separated by ,(comma).

JSon Data Structures

- Objects, i.e., collections of name-value pairs:

  – {“name1”: value1, “name2”: value2, ...}
  – “name” is also called a “key”

- *Ordered* lists of values:
  – [obj1, obj2, obj3, ...]

JSon Primitive Datatypes

- Number
- String
  - Denoted by double quotes
- Boolean
  - Either true or false
- nullempty
- The document/database is actually a tree!

MongoDB: The _id Field

Every MongoDB document must have an _id field.

- its value must be unique within the collection
- acts as the primary key of the collection
- it is the key in the key/value pair

- If you create a document without an _id field:
  - MongoDB adds the field for you
  - assigns it a unique BSON ObjectID
  - example from the MongoDB shell:

```
> db.test.save({ rating: "PG-13" })
> db.test.find() { "_id" :ObjectId("528bf38ce6d3df97b49a0569"), "rating" : "PG-13" }
```

- Note: quoting field names is optional (see rating above)

Data Modeling in MongoDB

Need to determine how to map entities and relationships => collections of documents

- Could in theory give each type of entity:
  - its own (flexibly formatted) type of document
  - those documents would be stored in the same collection
- However, it can make sense to group different types of entities together.
  - create an aggregate containing data that tends to be accessed together

Capturing Relationships in MongoDB

- **Two options:**
  - 1. store references to other documents using their _id values
  - 2. embed documents within other documents

Example relationships

Consider the following documents examples:

```
{
  "_id":ObjectId("52ffc33cd85242f436000001"),
  "name": "Tom Hanks",
  "contact": "987654321",
  "dob": "01-01-1991"
}
```

```
{
  "_id":ObjectId("52ffc4a5d85242602e000000"),
  "building": "22 A, Indiana Apt",
  "pincode": 123456,
  "city": "Los Angeles",
  "state": "California"
}
```

Here is an example of embedded relationship:

```
{
  "_id":ObjectId("52ffc33cd85242f436000001"),
  "contact": "987654321",
  "dob": "01-01-1991",
  "name": "Tom Benzamin",
  "address": [
    {
      "building": "22 A, Indiana Apt",
      "pincode": 123456,
      "city": "Los Angeles",
      "state": "California"
    },
    {
      "building": "170 A, Acropolis Apt",
      "pincode": 456789,
      "city": "Chicago",
      "state": "Illinois"
    }
  ]
}
```

And here an example of reference based

```
{
  "_id":ObjectId("52ffc33cd85242f436000001"),
  "contact": "987654321",
  "dob": "01-01-1991",
  "name": "Tom Benzamin",
  "address_ids": [
    ObjectId("52ffc4a5d85242602e000000"),
    ObjectId("52ffc4a5d85242602e000001")
  ]
}
```

Queries in MongoDB

Each query can only access a single collection of documents.

- Use a method called

```
db.collection.find(<selection>, <projection>)
```

- Example: find the names of all R-rated movies:

```
> db.movies.find({ rating: 'R' }, { name: 1 })
```

Projection

- **Specify the name of the fields that you want in the output with 1 (0 hides the value)**

- Example:
  - >db.movies.find({},{"title":1,_id:0})

  (will report the title but not the id)

Selection

- **You can specify the condition on the corresponding attributes using the find:**

```
>db.movies.find({ rating: "R", year: 2000 }, { name: 1, runtime: 1 })
```

- Operators for other types of comparisons:

MongoDB	SQL equivalent
$gt, $gte	>, >=
$lt, $lte	<, <=
$ne	!=

Example: find the names of movies with an earnings <= 200000

```
> db.movies.find({ earnings: { $lte: 200000 }})
```

- **For logical operators $and, $or, $nor**
  - use an array of conditions and apply the logical operator among the array conditions:

```
> db.movies.find({ $or: [ { rating: "R" }, { rating: "PG-13" } ] })
```

Aggregation

- **Recall the aggregate operators in SQL: AVG(), SUM(), etc.**
  More generally, aggregation involves computing a result
  from a collection of data.

- **MongoDB supports several approaches to aggregation:**
  - single-purpose aggregation methods
  - an aggregation pipeline
  - map-reduce

Aggregation pipelines are more flexible and useful (see next):
https://docs.mongodb.com/manual/core/aggregation-pipeline/

Simple Aggregations

- **db.collection.count(<selection>)**

  returns the number of documents in the collection

  that satisfy the specified selection document

Example: how may R-rated movies are shorter than 90 minutes?

```
>db.movies.count({ rating: "R", runtime: { $lt: 90 }})
```

- **db.collection.distinct(<field>, <selection>)**

returns an array with the distinct values of the specified field

in documents that satisfy the specified selection document

if omit the query, get all distinct values of that field

- which actors have been in one or more of the top 10 grossing movies?

```
>db.movies.distinct("actors.name", { earnings_rank: { $lte: 10 }})
```

Aggregation Pipeline

- A very powerful approach to write queries in MongoDB is to use pipelines.
- We execute the query in stages. Every stage gets as input some documents, applies filters/aggregations/projections and outputs some new documents. These documents are the input to the next stage (next operator) and so on
- Example for the zipcodes database:

```
> db.zipcodes.aggregate( [
   { $group: { _id: "$state", totalPop: { $sum: "$pop" } } },
   { $match: { totalPop: { $gte: 10*1000*1000 } } }
] )
```

Here we use group_by to group documents per state, compute sum of population and output documents with _id, totalPop (_id has the name of the state). The next stage finds a match for all states the have more than 10M population and outputs the state and total population.

More here: https://docs.mongodb.com/v3.0/tutorial/aggregation-zip-code-data-set/