# Overview

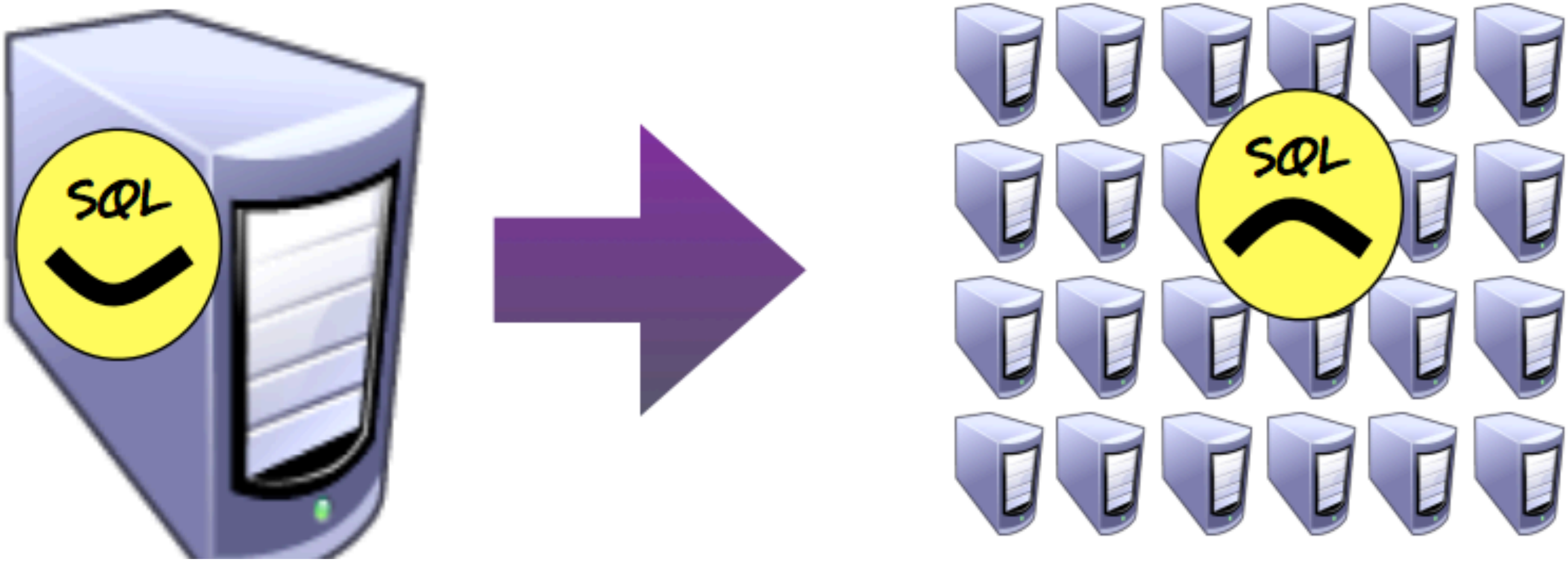
- Some History
- What is NoSQL?
- Why NoSQL?
- RDBMS vs NoSQL
- NoSQL Taxonomy
- Towards NewSQL

# Some History

- 1970's Relational Databases Invented
    - Fixed schema
    - Data is normalized
    - Expensive storage
    - Data abstracted away from apps
- 1980's RDBMS commercialized
    - Client/Server model
    - SQL becomes a standard
- 1990's Something new
    - 3-tier architecture
    - Rise of Internet
- 2000's
    - Web 2.0
    - Rise of social media and E-Commerce
    - Huge increase of collected data
    - Constant decrease of HW prices

Relational databases are designed to run on a single machine, so to scale, you need buy a bigger machine

But it's cheaper and more effective to scale horizontally by buying lots of machines.



# What is NoSQL?

NoSQL definition is evolving over time.

- Initially (2009) intended as *Absolutely No-SQL*
    - Most of the features offered by RDBMSes (global ACID, join, SQL, . . . ) considered useless or at least unnecessarily heavy
- Later on becomes *not only SQL*
    - Some RDBMSes features are recognized as useful, or even necessary, in many real application scenarios

# What is NoSQL?

There is no full agreement but nowadays we can summarize NoSQL definition as follows

- Next generation databases addressing some of the points:
    - non relational
    - schema-free
    - no Join
    - distributed
    - horizontally scalable with easy replication support
    - eventually consistent (*this will be clarified soon*)
    - open source

# Why NoSQL?

NoSQL databases first started out as in-house solutions to real problems:

- Amazon's Dynamo
- Google's BigTable
- LinkedIn's Voldemort
- Facebook's Cassandra
- Yahoo!'s PNUTS

# Why NoSQL? cont.

The listed companies didn't start off by rejecting relational technologies. They tried them and found that they didn't meet their requirements:

- Huge concurrent transactions volume
- Expectations of low-latency access to massive datasets
- Expectations of nearly perfect service availability while operating in an unreliable environment

# Why NoSQL? cont.

They tried the traditional approach

- Adding more HW
- Upgrading to faster HW as available

...and when it didn't work they tried to scale existing relational solutions:

- Simplifying DB schema
- De-normalization
- Introducing numerous query caching layers
- Separating read-only from write-dedicated replicas
- Data partitioning

# CAP Theorem

Formulated in 2000 by *Eric Brewer*

*It is impossible for a distributed computer system to simultaneously provide all three of the following guarantees:*

- **C***onsistency* (all nodes always see the same data at the same time)
- **A***vailability* (every request always receives a response about whether it was successful or failed)
- **P***artition Tolerance* (the system continues to operate despite arbitrary message loss or failure of part of the system)

# CAP Theorem and NoSQL

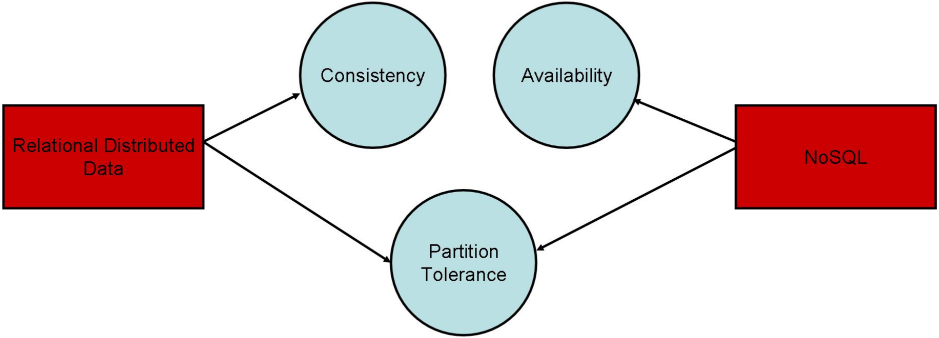
Most NoSQL database system architectures favour partition tolerance and availability over *strong* consistency

**Eventual Consistency**: inconsistencies between data held by different nodes are *transitory*. Eventually all nodes in the system will receive the latest consistent updates.

# RDBMS vs NoSQL

- RDBMSs enforce global ACID properties thus allowing multiple arbitrary operations in the context of a single transaction.
- NoSQL databases enforce only local BASE properties
    - **B***asically* **A***vailable* (data is always perceived as available by the user)
    - **S***oft State* (data at some node could change without any explicit user intervention. This follows from eventual consistency)
    - **E***ventually Consistent* (NoSQL guarantees consistency only at some undefined future time)

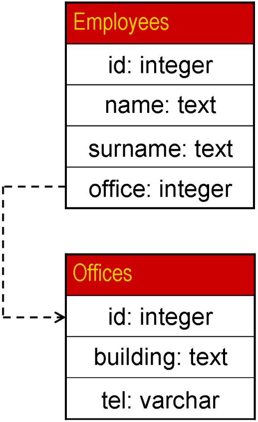
# RDBMS vs NoSQL



# NoSQL Taxonomy

- **Key/Value Store**
  - Amazon's Dynamo, LinkedIn's Voldemort, FoundationDB, . . .
- **Document Store**
  - MongoDB, CouchDB, . . .
- **Column Store**
  - Google's Bigtable, Apache's HBase, Facebook's Cassandra, . . .
- **Graph Store**
  - Neo4J, InfiniteGraph, . . .

# RDBMS Data



Employees

id: integer

name: text

surname: text

office: integer

Offices

id: integer

building: text

tel: varchar

| **id** | **name** | **surname** | **office** |
|---|---|---|---|
| 1 | Tom | Smith | 41 |
| 2 | John | Doe | 42 |
| 3 | Ann | Smith | 41 |

| **id** | **building** | **tel** |
|---|---|---|
| 41 | A4 | 45798 |
| 42 | B7 | 12349 |

# Key/Value Store

- Global collection of Key/Value pairs. Every item in the database is stored as an attribute name (*key*) together with its associated value
- Every key associated to *exactly* one value. No duplicates
- The value is simply a binary object. The DB does not associate any structure to stored values
- Designed to handle massive load of data
- Inspired by Distributed Hash Tables

# Key/Value Store

| **Key** | **Value** |
|---|---|
| employee_1 | name@Tom-surn@Smith-off@41-buil@A4-tel@45798 |
| employee_2 | name@John-surn@Doe-off@42-buil@B7-tel@12349 |
| employee_3 | name@Tom-surn@Smith |
| office_41 | buil@A4-tel@45798 |
| office_42 | buil@B7-tel@12349 |

# JSON

- Stands for **J**ava**S**cript **O**bject **N**otation
- Syntax for storing and exchanging text information
- Uses JavaScript syntax but it is language and platform independent
- Much like XML but smaller, faster and easier to parse than XML (and human readable)
- Basic data types(*Number*, *String*, *Boolean*) and supports data structures as objects and arrays

# JSON

```
{
 ”employees”: [
   { ”firstName”:”John” , ”lastName”:”Doe”}
   { ”firstName”:”Peter” , ”lastName”:”Jones”}
 ]
}
```

The ”employees” object is an array of two ”employee” records (objects).

# Document Store

- Same as Key/Value Store but pair each key with a arbitrarily complex data structure known as a *document*.
- Documents may contain many different key-value pairs or key-array pairs or even nested documents (like a JSON object).
- Data in documents can be *understood* by the DB: querying data is possible by other means than just a key (selection and projection over results are possible).

# Document Store

**Key**:”employee_1” →

```
{
 id:”1” .
 name:”Tom” .
 surname:”Smith” .
 office:{
       id:”41” .
       building:”A4” .
       telephone:”45798”
       }
}
```

**Key**:”office_1” →

```
{
 id:”41” .
 building:”A4” .
 telephone:”45798”
}
```

# Column Store

- "*A sparse, distributed multi-dimensional sorted map*"
- Store rows of data in similar fashion as typical RDBMSs do
- Rows are contained within a *Column Families*. Column Families can be considered as tables in RDBMSes
- Unlike table in RDBMSes, a Column Family can have different columns for each row it contains
- Each row is identified by a key that is unique in the context of a single Column Family. The same key can be however re-used within other Column Families, so it is possible to store unrelated data about the same key in different Column Families
- Each column is simply a key/value couple

# Column Store cont.

- It is also possible to organize data in *Super Columns*, that is columns whose values are themselves columns
- Usually data from the same Column Family are stored contiguously on disk (and consequently on the same node of the network)

# Column Store

**ColumnFamily**: Employees

| Key | id | name | surname | office | | |
|---|---|---|---|---|---|---|
| | | | | **id** | **buil.** | **tel.** |
| employee_1 | 1 | Tom | Smith | 41 | A4 | 45798 |

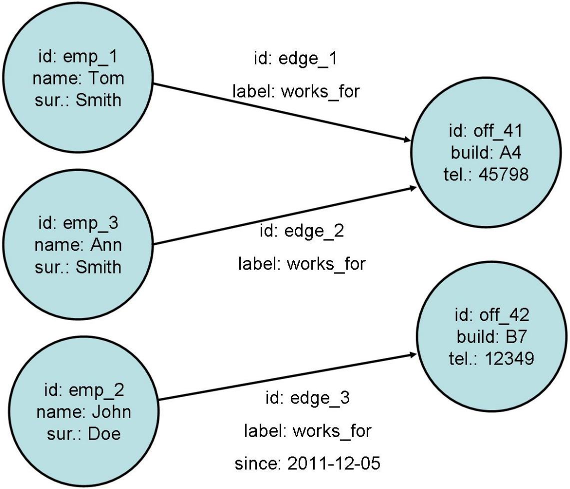
| Key | id | name | surname |
|---|---|---|---|
| employee_3 | 3 | Anna | Smith |

| Key | id | name | surname | office | |
|---|---|---|---|---|---|
| | | | | **id** | **buil.** |
| employee_2 | 2 | John | Doe | 42 | B7 |

# Graph Store

- Use graph structures with nodes, edges and properties to store pieces of data and relations between them
- Every element contains direct pointers to its adjacent elements. No index
- Computing answers to queries over the DB corresponds to finding suitable paths on the graph structure

# Graph Store


id: emp_1
name: Tom
sur.: Smith
id: edge_1
label: works_for
id: off_41
build: A4
tel.: 45798
id: emp_3
name: Ann
sur.: Smith
id: edge_2
label: works_for
id: off_42
build: B7
tel.: 12349
id: emp_2
name: John
sur.: Doe
id: edge_3
label: works_for
since: 2011-12-05

# Summarizing

- **Key/Value Store**
  - \+ Very fast lookups
  - – Stored data cannot have any schema
- **Document-Column Store**
  - \+ Tolerant of incomplete data
  - – Query performance
- **Graph Store**
  - \+ Exploit well known graph algorithms (shortest path, connectedness, . . . )
  - – Have to traverse the entire graph to achieve a definitive answer
- **Any NoSQL Store**
  - – NO JOIN! Pieces of related data have to be stored together
  - – no standard query language

Start the MongoDB JavaScript shell with:

```
# 'mongo' is shell binary. exact location might vary depending on
# installation method and platform
$ bin/mongo
```

By default the shell connects to database "test" on localhost.  You then see:

```
MongoDB shell version: <whatever>
url: test
connecting to: test
type "help" for help
>
```

`"connecting to:"` tells you the name of the database the shell is using. To switch databases, type:

```
> use mydb
switched to db mydb
```

# Dynamic schema

MongoDB has databases, collections, and indexes much like a traditional RDBMS.

In some cases (databases and collections) these objects can be implicitly created, however once created they exist in a system catalog (db.systems.collections, db.system.indexes).

# Schema free

- Collections contain (BSON) documents. Within these documents are fields.
- In MongoDB there is no predefinition of fields (what would be columns in an RDBMS).
- There is no schema for fields within documents – the fields and their value datatypes can vary.
- Thus there is no notion of an "alter table" operation which adds a "column".

# Schema free

- In practice, it is highly common for a collection to have a homogenous structure across documents; however this is not a requirement.
- This flexibility means that schema migration and augmentation are very easy in practice

## Inserting Data into A Collection

Let's create a test collection and insert some data into it. We will create two objects, `j` and `t`, and then save them in the collection *things*.

In the following examples, '>' indicates commands typed at the shell prompt.

```
> j = { name : "mongo" };
{"name" : "mongo"}
> t = { x : 3 };
{ "x" : 3  }
> db.things.save(j);
> db.things.save(t);
> db.things.find();
{ "_id" : ObjectId("4c2209f9f3924d31102bd84a"), "name" : "mongo" }
{ "_id" : ObjectId("4c2209fef3924d31102bd84b"), "x" : 3 }
>
```

Let's add some more records to this collection:

```
> for (var i = 1; i <= 20; i++) db.things.save({x : 4, j : i});
> db.things.find();
{ "_id" : ObjectId("4c2209f9f3924d31102bd84a"), "name" : "mongo" }
{ "_id" : ObjectId("4c2209fef3924d31102bd84b"), "x" : 3 }
{ "_id" : ObjectId("4c220a42f3924d31102bd856"), "x" : 4, "j" : 1 }
{ "_id" : ObjectId("4c220a42f3924d31102bd857"), "x" : 4, "j" : 2 }
{ "_id" : ObjectId("4c220a42f3924d31102bd858"), "x" : 4, "j" : 3 }
{ "_id" : ObjectId("4c220a42f3924d31102bd859"), "x" : 4, "j" : 4 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85a"), "x" : 4, "j" : 5 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85b"), "x" : 4, "j" : 6 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85c"), "x" : 4, "j" : 7 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85d"), "x" : 4, "j" : 8 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85e"), "x" : 4, "j" : 9 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85f"), "x" : 4, "j" : 10 }
{ "_id" : ObjectId("4c220a42f3924d31102bd860"), "x" : 4, "j" : 11 }
{ "_id" : ObjectId("4c220a42f3924d31102bd861"), "x" : 4, "j" : 12 }
{ "_id" : ObjectId("4c220a42f3924d31102bd862"), "x" : 4, "j" : 13 }
{ "_id" : ObjectId("4c220a42f3924d31102bd863"), "x" : 4, "j" : 14 }
{ "_id" : ObjectId("4c220a42f3924d31102bd864"), "x" : 4, "j" : 15 }
{ "_id" : ObjectId("4c220a42f3924d31102bd865"), "x" : 4, "j" : 16 }
{ "_id" : ObjectId("4c220a42f3924d31102bd866"), "x" : 4, "j" : 17 }
{ "_id" : ObjectId("4c220a42f3924d31102bd867"), "x" : 4, "j" : 18 }
has more
```

If we want to return the next set of results, there's the `it` shortcut. Continuing from the code above:

```
{ "_id" : ObjectId("4c220a42f3924d31102bd866"), "x" : 4, "j" : 17 }
{ "_id" : ObjectId("4c220a42f3924d31102bd867"), "x" : 4, "j" : 18 }
has more
> it
{ "_id" : ObjectId("4c220a42f3924d31102bd868"), "x" : 4, "j" : 19 }
{ "_id" : ObjectId("4c220a42f3924d31102bd869"), "x" : 4, "j" : 20 }
```

Technically, find() returns a cursor object. But in the cases above, we haven't assigned that cursor to a variable. So, the shell automatically iterates over the cursor, giving us an initial result set, and allowing us to continue iterating with the **it** command.

```
> var cursor = db.things.find();
> while (cursor.hasNext()) printjson(cursor.next());
{ "_id" : ObjectId("4c2209f9f3924d31102bd84a"), "name" : "mongo" }
{ "_id" : ObjectId("4c2209fef3924d31102bd84b"), "x" : 3 }
{ "_id" : ObjectId("4c220a42f3924d31102bd856"), "x" : 4, "j" : 1 }
{ "_id" : ObjectId("4c220a42f3924d31102bd857"), "x" : 4, "j" : 2 }
{ "_id" : ObjectId("4c220a42f3924d31102bd858"), "x" : 4, "j" : 3 }
{ "_id" : ObjectId("4c220a42f3924d31102bd859"), "x" : 4, "j" : 4 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85a"), "x" : 4, "j" : 5 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85b"), "x" : 4, "j" : 6 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85c"), "x" : 4, "j" : 7 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85d"), "x" : 4, "j" : 8 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85e"), "x" : 4, "j" : 9 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85f"), "x" : 4, "j" : 10 }
{ "_id" : ObjectId("4c220a42f3924d31102bd860"), "x" : 4, "j" : 11 }
{ "_id" : ObjectId("4c220a42f3924d31102bd861"), "x" : 4, "j" : 12 }
{ "_id" : ObjectId("4c220a42f3924d31102bd862"), "x" : 4, "j" : 13 }
{ "_id" : ObjectId("4c220a42f3924d31102bd863"), "x" : 4, "j" : 14 }
{ "_id" : ObjectId("4c220a42f3924d31102bd864"), "x" : 4, "j" : 15 }
{ "_id" : ObjectId("4c220a42f3924d31102bd865"), "x" : 4, "j" : 16 }
{ "_id" : ObjectId("4c220a42f3924d31102bd866"), "x" : 4, "j" : 17 }
{ "_id" : ObjectId("4c220a42f3924d31102bd867"), "x" : 4, "j" : 18 }
{ "_id" : ObjectId("4c220a42f3924d31102bd868"), "x" : 4, "j" : 19 }
{ "_id" : ObjectId("4c220a42f3924d31102bd869"), "x" : 4, "j" : 20 }
```

```
> db.things.find().forEach(printjson);
{ "_id" : ObjectId("4c2209f9f3924d31102bd84a"), "name" : "mongo" }
{ "_id" : ObjectId("4c2209fef3924d31102bd84b"), "x" : 3 }
{ "_id" : ObjectId("4c220a42f3924d31102bd856"), "x" : 4, "j" : 1 }
{ "_id" : ObjectId("4c220a42f3924d31102bd857"), "x" : 4, "j" : 2 }
{ "_id" : ObjectId("4c220a42f3924d31102bd858"), "x" : 4, "j" : 3 }
{ "_id" : ObjectId("4c220a42f3924d31102bd859"), "x" : 4, "j" : 4 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85a"), "x" : 4, "j" : 5 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85b"), "x" : 4, "j" : 6 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85c"), "x" : 4, "j" : 7 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85d"), "x" : 4, "j" : 8 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85e"), "x" : 4, "j" : 9 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85f"), "x" : 4, "j" : 10 }
{ "_id" : ObjectId("4c220a42f3924d31102bd860"), "x" : 4, "j" : 11 }
{ "_id" : ObjectId("4c220a42f3924d31102bd861"), "x" : 4, "j" : 12 }
{ "_id" : ObjectId("4c220a42f3924d31102bd862"), "x" : 4, "j" : 13 }
{ "_id" : ObjectId("4c220a42f3924d31102bd863"), "x" : 4, "j" : 14 }
{ "_id" : ObjectId("4c220a42f3924d31102bd864"), "x" : 4, "j" : 15 }
{ "_id" : ObjectId("4c220a42f3924d31102bd865"), "x" : 4, "j" : 16 }
{ "_id" : ObjectId("4c220a42f3924d31102bd866"), "x" : 4, "j" : 17 }
{ "_id" : ObjectId("4c220a42f3924d31102bd867"), "x" : 4, "j" : 18 }
{ "_id" : ObjectId("4c220a42f3924d31102bd868"), "x" : 4, "j" : 19 }
{ "_id" : ObjectId("4c220a42f3924d31102bd869"), "x" : 4, "j" : 20 }
```

## SELECT * FROM things WHERE name="mongo"

```
> db.things.find({name:"mongo"}).forEach(printjson);
{ "_id" : ObjectId("4c2209f9f3924d31102bd84a"), "name" : "mongo" }
```

## SELECT * FROM things WHERE x=4

```
> db.things.find({x:4}).forEach(printjson);
{ "_id" : ObjectId("4c220a42f3924d31102bd856"), "x" : 4, "j" : 1 }
{ "_id" : ObjectId("4c220a42f3924d31102bd857"), "x" : 4, "j" : 2 }
{ "_id" : ObjectId("4c220a42f3924d31102bd858"), "x" : 4, "j" : 3 }
{ "_id" : ObjectId("4c220a42f3924d31102bd859"), "x" : 4, "j" : 4 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85a"), "x" : 4, "j" : 5 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85b"), "x" : 4, "j" : 6 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85c"), "x" : 4, "j" : 7 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85d"), "x" : 4, "j" : 8 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85e"), "x" : 4, "j" : 9 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85f"), "x" : 4, "j" : 10 }
{ "_id" : ObjectId("4c220a42f3924d31102bd860"), "x" : 4, "j" : 11 }
```

MongoDB also lets you return "partial documents" - documents that have only a subset of the elements of the document stored in the database. To do this, you add a second argument to the `find()` query, supplying a document that lists the elements to be returned.

To illustrate, lets repeat the last example `find({x:4})` with an additional argument that limits the returned document to just the "j" elements:

**SELECT j FROM things WHERE x=4**

```
> db.things.find({x:4}, {j:true}).forEach(printjson);
{ "_id" : ObjectId("4c220a42f3924d31102bd856"), "j" : 1 }
{ "_id" : ObjectId("4c220a42f3924d31102bd857"), "j" : 2 }
{ "_id" : ObjectId("4c220a42f3924d31102bd858"), "j" : 3 }
{ "_id" : ObjectId("4c220a42f3924d31102bd859"), "j" : 4 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85a"), "j" : 5 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85b"), "j" : 6 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85c"), "j" : 7 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85d"), "j" : 8 }
```

# findOne()

- For convenience, the mongo shell (and other drivers) lets you avoid the programming overhead of dealing with the cursor, and just lets you retrieve one document via the findOne() function.
- findOne() takes all the same parameters of the find() function, but instead of returning a cursor, it will return either the first document returned from the database, or null if no document is found that matches the specified query.

However, the `findOne()` method is both convenient and efficient:

```
> printjson(db.things.findOne({name:"mongo"}));
{ "_id" : ObjectId("4c2209f9f3924d31102bd84a"), "name" : "mongo" }
```

This is more efficient because the client requests a single object from the database, so less work is done by the database and the network. This is the equivalent of `find({name:"mongo"}).limit(1)`.

Another example of finding a single document by _id:

```
> var doc = db.things.findOne({_id:ObjectId("4c2209f9f3924d31102bd84a")});
> doc
{ "_id" : ObjectId("4c2209f9f3924d31102bd84a"), "name" : "mongo" }
```

# Architecture Replica Sets, Autosharding

- MongoDB uses replica sets to provide read scalability, and high availability.
- Autosharding is used to scale writes (and reads).
- Replica sets and autosharding go hand in hand if you need mass scale out.

# Replica sets

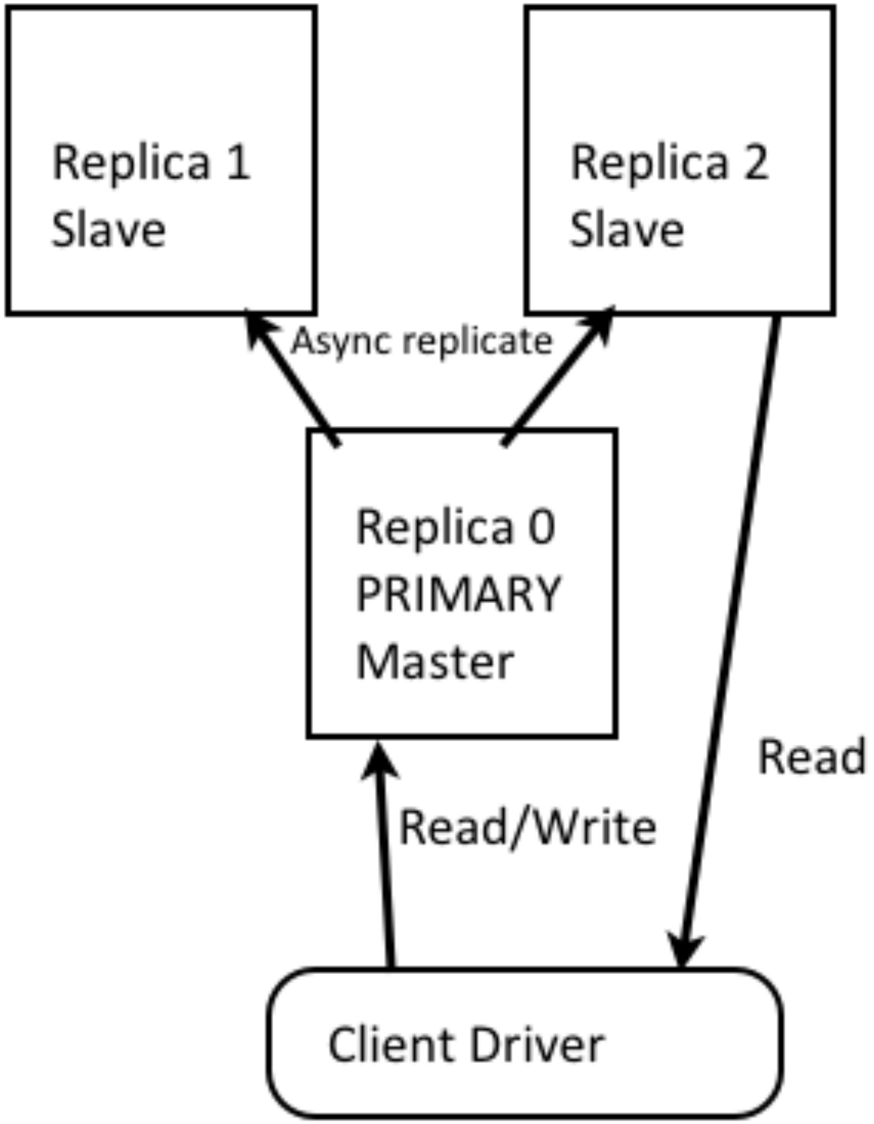
- The major advantages of replica sets are:
  - business continuity through high availability,
  - data safety through data redundancy,
  - read scalability through load sharing (reads).
- Replica sets use a share nothing architecture.

# Replica sets

- Typically you have at least three MongoDB instances in a replica set on different server machines.
- You can add more replicas of the primary if you like for read scalability, but you only need three for high availability failover.

# Replica Sets

- Drivers know the primary
- If primary down, Drivers know how to get new primary
- Data is replicated after writing
- Typical to have three in a replica set
- You can do more
- Load sharing for reads
- Writes only to primary



Replica 1
Slave

Replica 2
Slave

Async replicate

Replica 0
PRIMARY
Master

Read

Read/Write

Client Driver

# Replica sets

- By default replication is non-blocking/async.
- This might be acceptable for some data category
  - descriptions in an online store
- but not other data
  - shopping cart's credit card transaction data
- For important data, the client can block until data is replicated on all servers or written to the journal.
- The client can force the master to sync to slaves before continuing.
  - This sync blocking is slower.

# Replica sets

Async/non-blocking is faster and is often described as eventual consistency. Waiting for a master to sync is a form of data safety.

Here is a list of some data safety options for MongoDB:

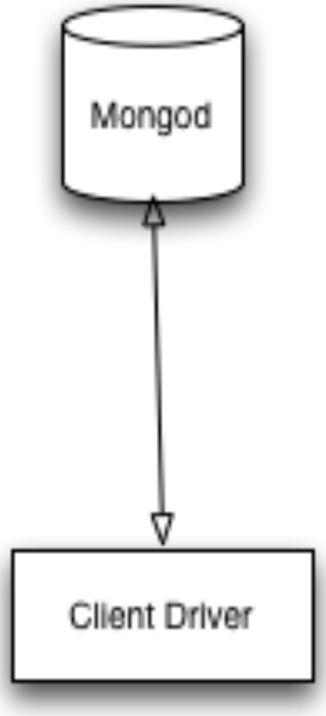
- Wait until write has happened on all replicas
- Wait until write is on two servers (primary and one other)
- Wait until write has occurred on majority of replicas
- Wait until write operation has been written to journal

# Sharding

- Sharding allows MongoDB to scale horizontally.
- Sharding is also called partitioning.
  - You partition each of your servers a portion of the data to hold or the system does this for you.
- MongoDB can automatically change partitions for optimal data distribution and load balancing, and it allows you to elastically add new nodes

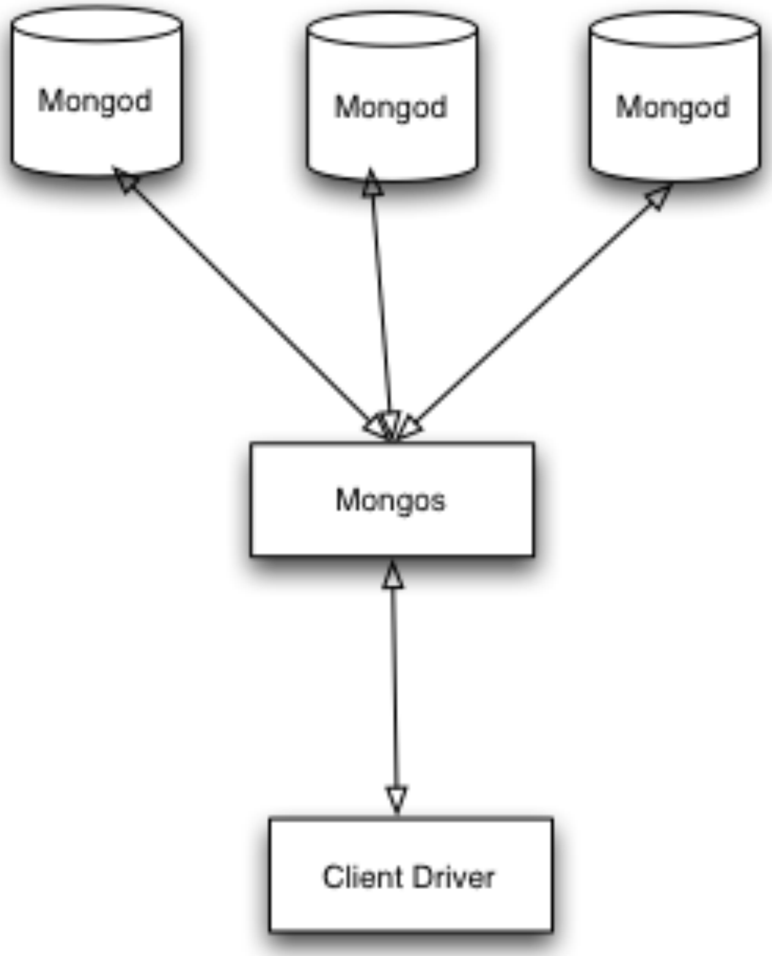
# Non-sharded client connection

- Client Driver talks directly to mongod process



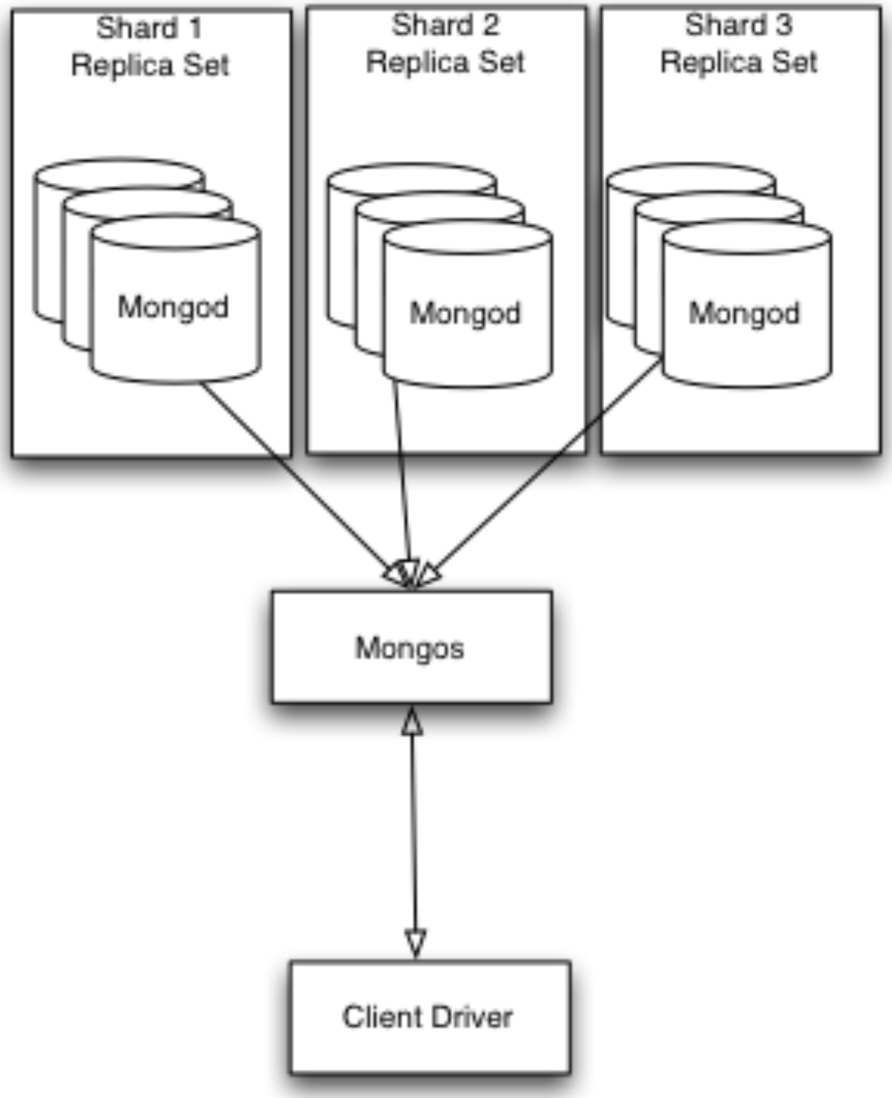
# Autosharded

- Three actors now: mongod, mongos, and Client Driver library
- Mongod is the process
- Mongos is a router, it routes writes to correct mongod instance
- Shares writing



# Autosharding plus Replica Set

- Autosharding increases writes, helps with scale out
- Replica Sets are for high availability, and read scaling not write scaling
- Each shard/partition has its own replica set



# Large deployment