

# Introduction to Databases

## Part 2: MongoDB

[ashok.dudhat@devugees.org](mailto:ashok.dudhat@devugees.org)

# What is NOSQL?

A variety of technologies that are alternatives to **Tables** and Structured Query Language (**SQL**). Stands for Not Only SQL.

NoSQL is a **non-relational database management systems**, different from traditional relational database management systems in some significant ways. It is designed for distributed data stores where very large scale of data storing needs (for example Google or Facebook which collects terabits of data every day for their users). These type of data storing may **not require fixed schema**, **avoid join operations** and typically **scale horizontally**.

# Why NoSQL?

In today's time data is becoming easier to access and capture through third parties such as Facebook, Google+ and others.

Personal user information, social graphs, geo location data, user-generated content and machine logging data are just a few examples where the data has been increasing exponentially.

To avail the above service properly, it is required to process huge amount of data. Which SQL databases were never designed. The evolution of NoSql databases is to handle these huge data properly.

Example :

---

## Social-network graph:

- |   |  |
|---|--|
| 1 | Each record: UserID1, UserID2  |
| 2 | Separate records: UserID, first_name, last_name, age, gender, ...            |
| 3 | Task: Find all friends of friends of friends of ... friends of a given user. |

## Wikipedia pages :

- |   |   |
|---|---|
| 1 | Large collection of documents   |
| 2 | Combination of structured and unstructured data                             |
| 3 | Task: Retrieve all pages regarding athletics of Summer Olympic before 1950. |

# The Big Picture Differences

## The Language

Think of a town - we'll call it **Town A** - where everyone speaks the **same language**. All of the businesses are built around it, every form of communication uses it - in short, it's the only way that the residents understand and interact with the world around them. **Changing that language** in one place would be **confusing and disruptive for everyone**.

Now, think of another town, **Town B**, where every home can speak a **different language**. Everyone interacts with the world differently, and there's no "universal" understanding or set organization. **If one home is different, it doesn't affect anyone else at all.**

# SQL databases

**SQL databases** use structured query language (SQL) for defining and manipulating data. On one hand, this is extremely powerful: SQL is one of the most versatile and widely-used options available, making it a safe choice and especially great for complex queries. On the other hand, it can be restrictive. SQL requires that you use **predefined schemas** to determine the structure of your data before you work with it. In addition, **all of your data must follow the same structure.**

This can require significant up-front preparation, and, as with **Town A**, it can mean that a **change in the structure** would be both **difficult and disruptive to your whole system.**

# NoSQL database

**A NoSQL database**, on the other hand, has **dynamic schema for unstructured data**, and data is stored in many ways: it can be column-oriented, document-oriented, graph-based or organized as a Key-Value store. This flexibility means that:

- You can create documents without having to first define their structure.
- Each document can have its own unique structure.
- The syntax can vary from database to database.
- You can add fields as you go.

# The Scalability

In most situations, **SQL databases are vertically scalable**, which means that you can increase the load on a single server by increasing things like CPU, RAM or SSD.

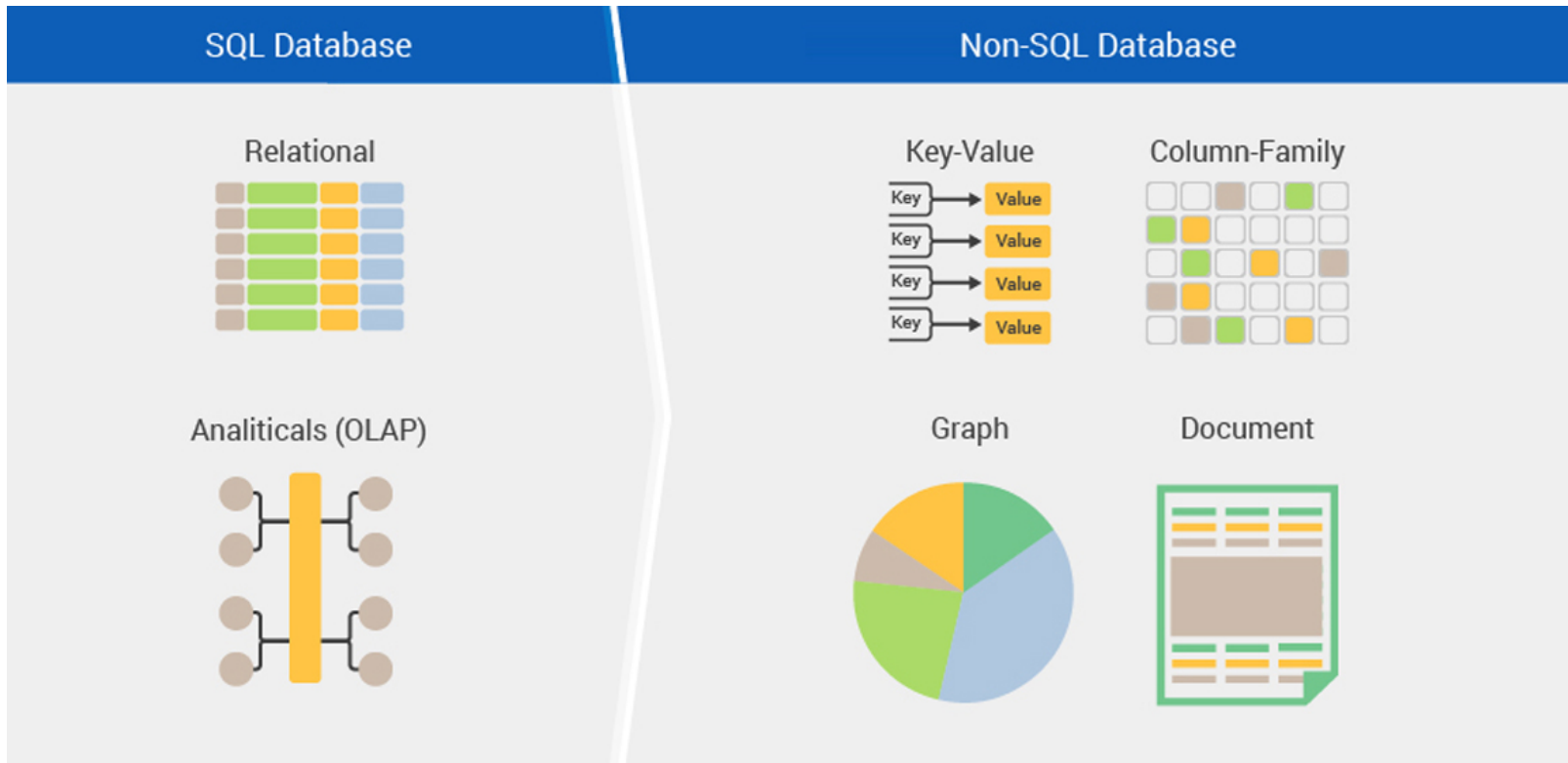
**NoSQL databases**, on the other hand, **are horizontally scalable**. This means that you handle more traffic by sharding, or adding more servers in your NoSQL database.

It's like **adding more floors to the same building** versus **adding more buildings to the neighborhood**.

The latter can ultimately become larger and more powerful, making NoSQL databases the preferred choice for large or ever-changing data sets.









# The Structure

**SQL databases** are **table-based**, while **NoSQL databases** are either **document-based**, **key-value pairs**, **graph databases** or **wide-column stores**.





# Examples of NoSQL

Type	Example	
Key-Value Store	 redis	 riak
Wide Column Store	 H-BASE	 cassandra
Document Store	 mongoDB	 CouchDB relax
Graph Store	 Neo4j	 The Distributed Graph Database

# Core NOSQL Systems

**Riak** API: tons of languages, JSON, Protocol: REST, Query Method: MapReduce term matching, Scaling: Multiple Masters; Written in: Erlang, Concurrency: eventually consistent (stronger than MVCC via Vector Clocks).

**Redis** API: Tons of languages, Written in: C, Concurrency: in memory and saves asynchronously to disk after a defined time. Append only mode available. Different kinds of fsync policies. Replication: Master / Slave, Misc: also lists, sets, sorted sets, hashes, queues.

**Hadoop / HBase** API: Java / any writer, Protocol: any write call, Query Method: MapReduce Java / any exec, Replication: HDFS Replication, Written in: Java, Concurrency

**Cassandra** massively scalable, partitioned row store, masterless architecture, linear scale performance, no single points of failure, read/write support across multiple data centers & cloud availability zones. API / Query Method: CQL and Thrift, replication: peer-to-peer, written in: Java, Concurrency: tunable consistency, Misc: built-in data compression, MapReduce support, primary/secondary indexes, security features.

**MongoDB** API: **BSON**, Protocol: C, **Query Method: dynamic object-based language & MapReduce**, Replication: Master Slave & Auto-Sharding, **Written in: C++**, Concurrency: Update in Place. Misc: Indexing, GridFS, Freeware + Commercial License.

**CouchDB** API: JSON, Protocol: REST, Query Method: MapReduceR of JavaScript Funcs, Replication: Master Master, Written in: Erlang, Concurrency: MVCC.

**Neo4J** API: lots of langs, Protocol: Java embedded / REST, Query Method: SparQL, nativeJavaAPI, JRuby, Replication: typical MySQL style master/slave, Written in: Java, Concurrency: non-block reads, writes locks involved nodes/relationships until commit.

**Infinite Graph** (by Objectivity) API: Java, Protocol: Direct Language Binding, Query Method: Graph Navigation API, Predicate Language Qualification, Written in: Java (Core C++), Data Model: Labeled Directed Multi Graph, Concurrency: Update locking on subgraphs, concurrent non-blocking ingest, Misc: Free for Qualified Startups.

# SQL vs NoSQL: MySQL vs MongoDB

## MySQL: The SQL Relational Database

The following are some MySQL benefits and strengths:

- **Maturity:** MySQL is an extremely established database, meaning that there's a huge community, extensive testing and quite a bit of stability.
- **Compatibility:** MySQL is available for all major platforms, including Linux, Windows, Mac, BSD and Solaris. It also has connectors to languages like Node.js, Ruby, C#, C++, Java, Perl, Python and PHP, meaning that it's not limited to SQL query language.
- **Cost-effective:** The database is open source and free.
- **Replicable:** The MySQL database can be replicated across multiple nodes, meaning that the workload can be reduced and the scalability and availability of the application can be increased.
- **Sharding:** While sharding cannot be done on most SQL databases, it can be done on MySQL servers. This is both cost-effective and good for business.

# SQL vs NoSQL: MySQL vs MongoDB

## MongoDB: The NoSQL Non-Relational Database.

The following are some of MongoDB benefits and strengths:

- **Dynamic schema:** As mentioned, this gives you flexibility to change your data schema without modifying any of your existing data.
- **Scalability:** MongoDB is horizontally scalable, which helps reduce the workload and scale your business with ease.
- **Manageability:** The database doesn't require a database administrator. Since it is fairly user-friendly in this way, it can be used by both developers and administrators.
- **Speed:** It's high-performing for simple queries.
- **Flexibility:** You can add new columns or fields on MongoDB without affecting existing rows or application performance.

# In Short : RDBMS vs NoSQL

## **RDBMS**

- Structured and organized data.
- Structured query language (SQL).
- Data and its relationships are stored in separate tables.
- Data Manipulation Language, Data Definition Language.
- Tight Consistency.

## **NoSQL**

- Stands for Not Only SQL.
- No declarative query language.
- No predefined schema.
- Key-Value pair storage, Column Store, Document Store, Graph databases.
- Eventual consistency rather ACID property.
- Unstructured and unpredictable data.
- Prioritises high performance, high availability and scalability.
- BASE Transaction.

# Which Database Is Right For Your Business?

MySQL is a strong choice for any business that will benefit from its pre-defined structure and set schemas. For example, applications that require **multi-row transactions** - like **accounting systems** or **systems that monitor inventory** - or that run on legacy systems will thrive with the MySQL structure.

MongoDB, on the other hand, is a good choice for businesses that have rapid growth or databases with no clear schema definitions. More specifically, **if you cannot define a schema for your database**, if you find yourself denormalizing data schemas, or **if your schema continues to change** - as is often the case with **mobile apps, real-time analytics, content management systems**, etc.- MongoDB can be a strong choice for you.

# Production Deployment Examples

There is a large number of companies using NoSQL.

- Google
- Facebook
- Mozilla
- Adobe
- Foursquare
- LinkedIn
- Digg
- McGraw-Hill Education
- Vermont Public Radio

# Document Oriented databases

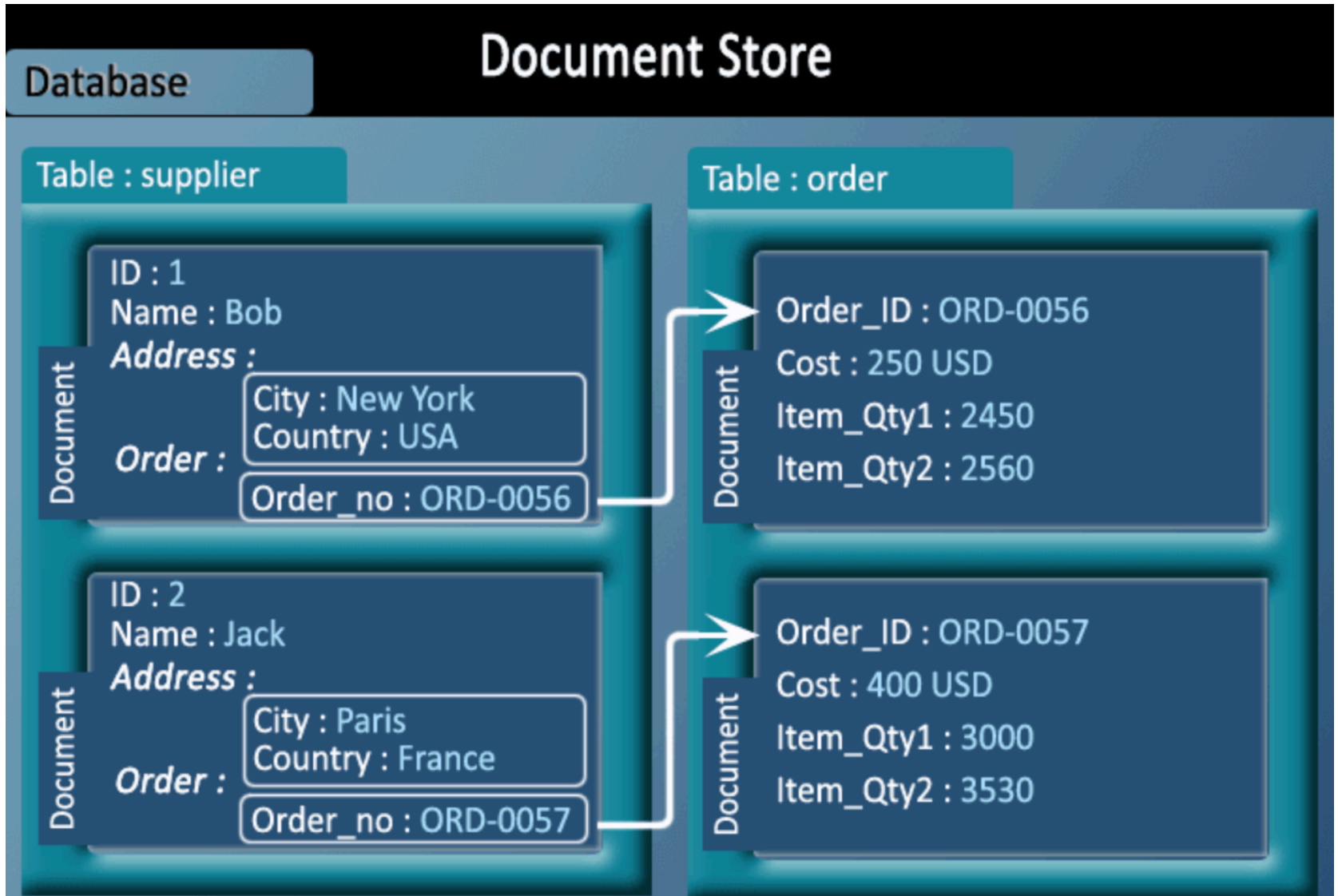
- A **collection** of **documents**.
- **Data** in this model is stored inside **documents**.
- A document is a **key value collection** where the key allows access to its value.
- Documents are **not typically forced to have a schema** and therefore are flexible and easy to change.
- **Documents are stored into collections** in order to group different kinds of data.
- Documents can contain many different key-value pairs, or key-array pairs, or even nested documents.



# Comparison between Classic relational model and Document model

Relational model	Document model
Tables	Collections
Rows	Documents
Columns	Key/value pairs
Joins	not available

# Pictorial Presentation



# Introduction : MongoDB

Document: „A record in a MongoDB collection and the basic unit of data in MongoDB. Documents look like JSON objects but exist as BSON“.

- BSON

```
{  
  "title": "Article two",  
  "category": "Education",  
  "body": "this is the body"  
}
```

# Documents

- BSON is JSON saved binarily

{

“title”: “Article two”,

“category”: “Education”,

“body”: “this is the body”

}

- We do not have a JSON file, we have binary BSON file. Not human-readable.

# Documents

- BSON is JSON saved binarily

{

“title

“cat

“bo

}

Each document is JSON  
which is saved as a BSON  
file.

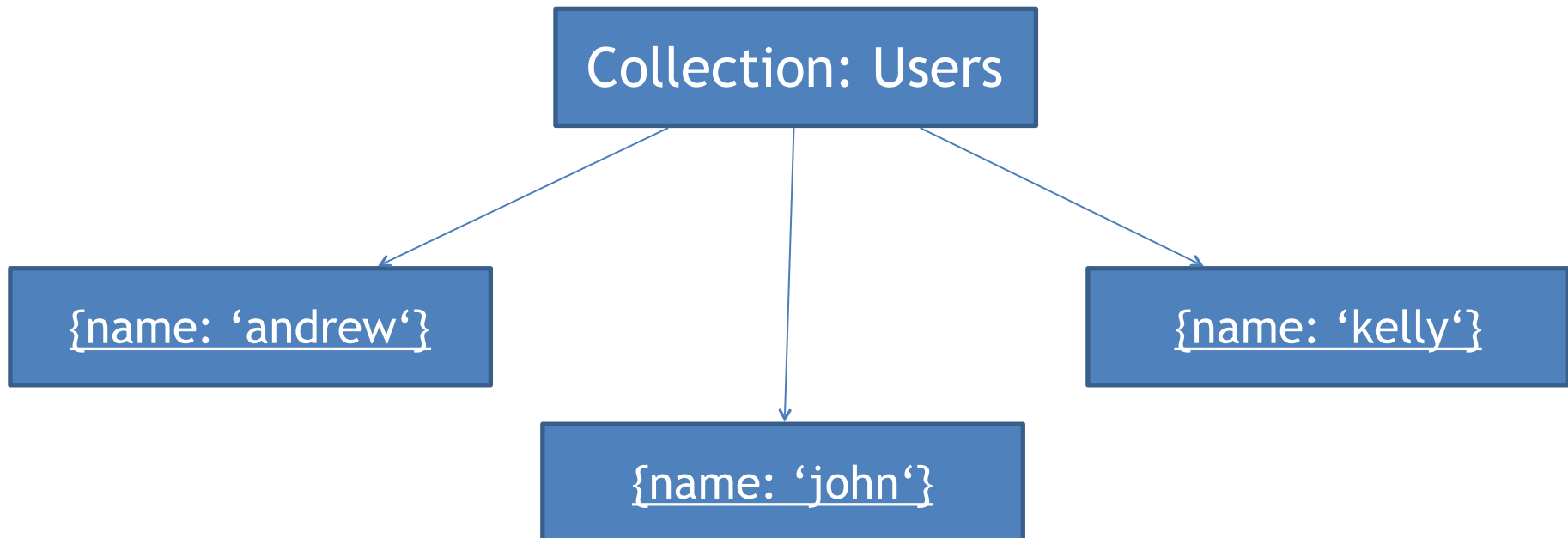
- We do not have a JSON file, we have binary BSON file. Not human-readable.

# Collections

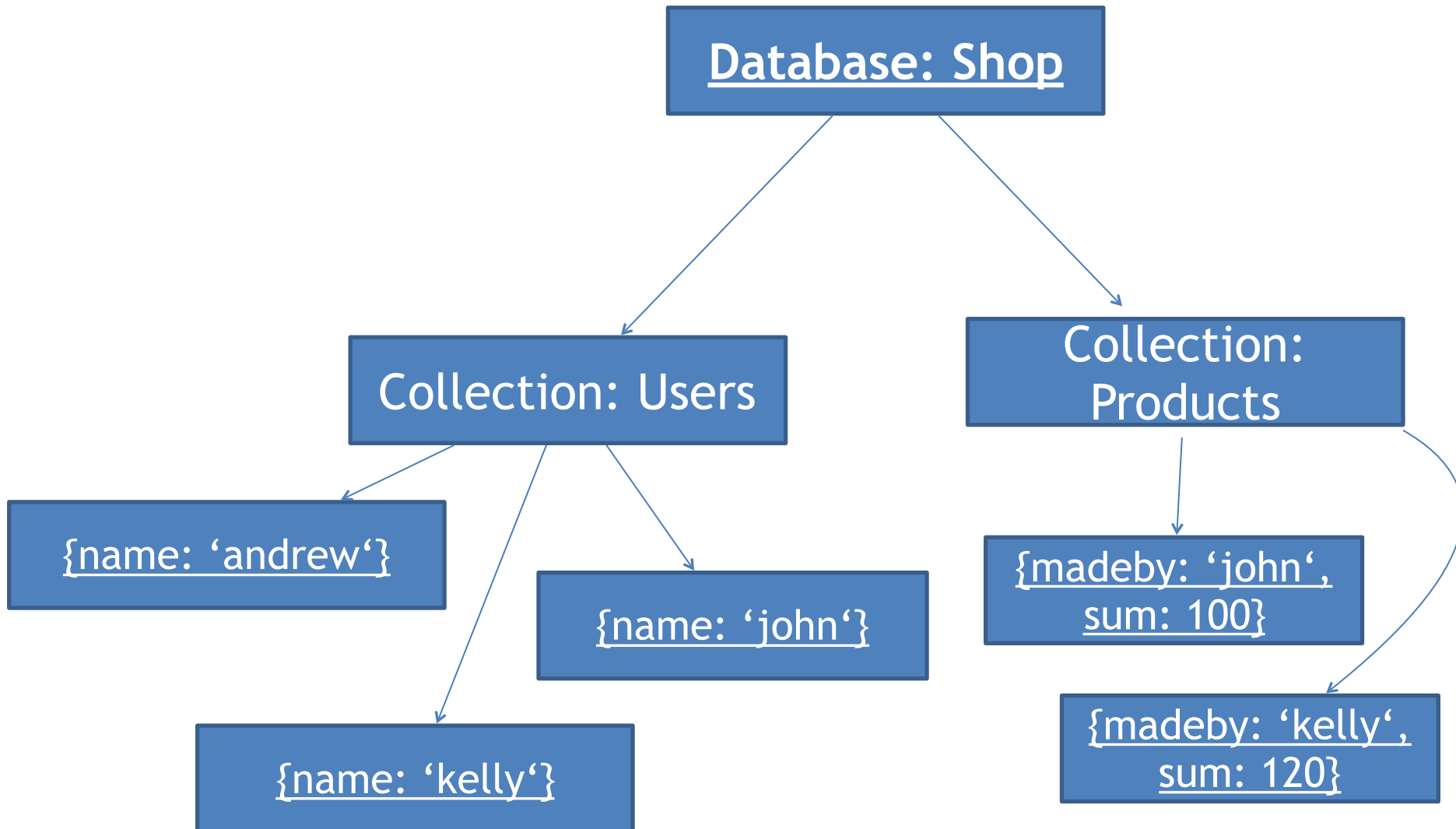
- Collection: A group of MongoDB documents. Typically, all documents in a collection have a similar related purpose.

# Collections

- Collection: A group of MongoDB documents. Typically, all documents in a collection have a similar related purpose.



# Collections





# Biggest difference between SQL and MongoDB

SQL

JOINS

MongoDB

REFERENCES

# Installation : MongoDB

- `sudo apt-get update`
- `sudo apt-get install -y mongodb-org`
- `sudo systemctl start mongod`
- `sudo systemctl status mongod`

# Data-Types

## DATA TYPES



### STRING

name: String

```
{  
  name: "John"  
}
```



### NUMBER

likes: Number

```
{  
  likes: 5  
}
```



### DATE

timeStamp: Date

```
{  
  timeStamp: ISODate("...")  
}
```



### ARRAY

tags: Array

OR

```
tags: []  
{  
  tags: ["tag1", "tag2"]  
}
```



### BOOLEAN

published: Boolean

```
{  
  published: true  
}
```



### ObjectId

\_creator: Schema.ObjectId

```
{  
  _creator: "41239878"  
}
```

# Mongo Shell

Let Create Database, Collection, Document  
and play with some function/queries...

# Quiz

1. How do we access the shell?

- A: by typing 'mongoddb'
- B: by typing 'mongo'
- C: by loading the browser
- D: by typing 'mongo start'

# Quiz

1. How do we access the shell?

A: by typing 'mongoddb'

B: by typing 'mongo'

C: by loading the browser

D: by typing 'mongo start'

# Quiz

2. How do we add a new document if the to-be-updated document is not found?

A: \$set

B: \$in

C: upsert=true

D: \$upsert=true

# Quiz

2. How do we add a new document if the to-be-updated document is not found?

A: \$set

B: \$in

C: `upsert=true`

D: `$upsert=true`



# Quiz

3. What command do we use to indicate which database we want to access?

A: use

B: show

C: find

D: list

# Quiz

3. What command do we use to indicate which database we want to access?

A: use

B: show

C: find

D: list

# Quiz

4. What method is used to display our documents in a clean and organized way?

A: insert

B: find

C: pretty

D: clean

E: style

# Quiz

4. What method is used to display our documents in a clean and organized way?

A: insert

B: find

C: pretty

D: clean

E: style

# Quiz

5. Which one of these is not one of the 6 main data types commonly used within the model of our collection?

- A: String
- B: Boolean
- C: Number
- D: Date
- E: Buffer
- F: Array

# Quiz

5. Which one of these is not one of the 6 main data types commonly used within the model of our collection?

- A: String
- B: Boolean
- C: Number
- D: Date
- E: Buffer
- F: Array

# Task

1. Create a new database “medialib” that is supposed to save infos about videos and songs. both are saved in the same collection “mediaitem”. Find 5 proper keys (title and type is a must, whereas type can either be „movie“ or „song“) and add 3 songs and 3 movies.
2. Write a function/query list Titles that lists all titles and types of each item in your collection. Therefore, take a look at the collection.count() method.

# Node.js + MongoDB

Install MongoDB Driver

```
$ npm install mongodb
```



# Import require package

Node.js can use this module to manipulate MongoDB databases:

```
var mongo = require('mongodb');
```

# Creating a Database

To create a database in MongoDB, start by creating a **MongoClient** object, then specify a connection URL with the correct ip address and the name of the database you want to create.

**MongoDB will create the database if it does not exist, and make a connection to it.**

```
var MongoClient = require('mongodb').MongoClient;  
var url = "mongodb://localhost:27017/testdb";
```

```
MongoClient.connect(url, function(err, db) {  
  if (err) throw err;  
  console.log("Database created!");  
  db.close();  
});
```

# Creating a Collection

To create a collection in MongoDB, use the `createCollection()` method:

Create a collection called **"customers"**:

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("testdb");
  dbo.createCollection("customers", function(err, res) {
    if (err) throw err;
    console.log("Collection created!");
    db.close();
  });
});
```

**Important:** A **collection** in MongoDB is the same as a **table** in MySQL. In MongoDB, a collection is not created until it gets content! MongoDB waits until you have inserted a document before it actually creates the collection.

# Insert Document : Insert One

To insert a record, or *document* as it is called in MongoDB, into a collection, we use the `insertOne()` method.

The first parameter of the `insertOne()` method is an object containing the name(s) and value(s) of each field in the document you want to insert. It also takes a callback function where you can work with any errors, or the result of the insertion:

Insert a document in the "**customers**" collection:

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("testdb");
  var myobj = { name: "Company Inc", address: "Highway 37" };
  dbo.collection("customers").insertOne(myobj, function(err, res) {
    if (err) throw err;
    console.log("1 document inserted");
    db.close();
  });
});
```

**Important:** A **document** in MongoDB is the same as a **record** in MySQL. If you try to insert documents in a collection that do not exist, MongoDB will create the collection automatically.

# Insert Document : Insert Many

To insert multiple documents into a collection in MongoDB, we use the `insertMany()` method.

The first parameter of the `insertMany()` method is an **array of objects**, containing the data you want to insert.

Insert multiple documents in the "**customers**" collection:

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("testdb");
  var myobj = [
    { name: 'John', address: 'Highway 71'},
    { name: 'Peter', address: 'Lowstreet 4'},
    { name: 'Amy', address: 'Apple st 652'},
    { name: 'Hannah', address: 'Mountain 21'},
    { name: 'Michael', address: 'Valley 345'},
    { name: 'Sandy', address: 'Ocean blvd 2'},
    { name: 'Betty', address: 'Green Grass 1'},
    { name: 'Richard', address: 'Sky st 331'},
    { name: 'Susan', address: 'One way 98'},
    { name: 'Vicky', address: 'Yellow Garden 2'},
    { name: 'Ben', address: 'Park Lane 38'},
    { name: 'William', address: 'Central st 954'},
    { name: 'Chuck', address: 'Main Road 989'},
    { name: 'Viola', address: 'Sideway 1633'}
  ];
  dbo.collection("customers").insertMany(myobj, function(err, res) {
    if (err) throw err;
    console.log("Number of documents inserted: " + res.insertedCount);
    db.close();
  });
});
```

# The `_id` Field

If you do not specify an `_id` field, then MongoDB will add one for you and assign a unique id for each document.

In the example above no `_id` field was specified, and as you can see from the result object, MongoDB assigned a **unique** `_id` for each document.

If you *do* specify the `_id` field, the value must be unique for each document:

Insert three records in a "**products**" table, with specified `_id` fields:

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("testdb");
  var myobj = [
    { _id: 154, name: 'Chocolate Heaven' },
    { _id: 155, name: 'Tasty Lemon' },
    { _id: 156, name: 'Vanilla Dream' }
  ];
  dbo.collection("products").insertMany(myobj, function(err, res) {
    if (err) throw err;
    console.log(res);
    db.close();
  });
});
```

When executing the `insertMany()` method, a result object is returned. The result object contains information about how the insertion affected the database.

# Find

In MongoDB we use the **find** and **findOne** methods to find data in a collection.

Just like the **SELECT** statement is used to find data in a table in a MySQL database.

# Find Data : Find One

To select data from a collection in MongoDB, we can use the `findOne()` method. The `findOne()` method returns the first occurrence in the selection.

**The first parameter** of the `findOne()` method is a query object. In this example we use an empty query object, which selects all documents in a collection (but returns only the first document).

Find the first document in the **customers** collection:

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("testdb");
  dbo.collection("customers").findOne({}, function(err, result) {
    if (err) throw err;
    console.log(result.name);
    db.close();
  });
});
```



# Find Data : Find All

To select data from a table in MongoDB, we can also use the `find()` method. The `find()` method returns all occurrences in the selection.

**The first parameter** of the `find()` method is a query object. In this example we use an empty query object, which selects all documents in the collection.

No parameters in the `find()` method gives you the same result as **SELECT \*** in MySQL.

Find all documents in the **customers** collection:

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("testdb");
  dbo.collection("customers").find({}).toArray(function(err, result)
  {
    if (err) throw err;
    console.log(result);
    db.close();
  });
});
```

# Find Data : Find Some

The **second parameter** of the `find()` method is an object describing which fields to include in the result.

This parameter is optional, and if omitted, all fields will be included in the result.

Return the fields "**name**" and "**address**" of all documents in the **customers** collection:

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("testdb");
  dbo.collection("customers").find({}, { _id: 0, name: 1, address:
1 }).toArray(function(err, result) {
    if (err) throw err;
    console.log(result);
    db.close();
  });
});
```

You are not allowed to specify both 0 and 1 values in the same object (except if one of the fields is the `_id` field). If you specify a field with the value 0, all other fields get the value 1, and vice versa: Set only `{ address: 0 }` and check the result.

To exclude the `_id` field, you must set its value to 0: Set the `{ _id: 0, name: 1 }` and check the result.

Also please check the result by set `{ _id: 0 }` and `{ name: 1, address: 0 }`.

# Filter the Result

When finding documents in a collection, you can filter the result by using a **query object**.

**The first argument** of the `find()` method is a **query object**, and is used to limit the search.

Find documents with the **address "Park Lane 38"**:

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("testdb");
  var query = { address: "Park Lane 38" };
  dbo.collection("customers").find(query).toArray(function(err,
result) {
    if (err) throw err;
    console.log(result);
    db.close();
  });
});
```

# Filter With Regular Expressions

You can write regular expressions to find exactly what you are searching for.

**Regular expressions can only be used to query *strings*.**

To find only the documents where the **"address" field starts with the letter "S"**, use the **regular expression** `/^S/`:

Find documents where the **address starts** with the letter **"S"**:

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("testdb");
  var query = { address: /^S/ };
  dbo.collection("customers").find(query).toArray(function(err, result) {
    if (err) throw err;
    console.log(result);
    db.close();
  });
});
```

# Sort the Result

Use the `sort()` method to sort the result in **ascending or descending order**.

The `sort()` method takes one parameter, an object defining the sorting order.

Sort the result alphabetically by **name**:

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("testdb");
  var mysort = { name: 1 };
  dbo.collection("customers").find().sort(mysort).toArray(function(err,
result) {
    if (err) throw err;
    console.log(result);
    db.close();
  });
});
```

Also Check, Using the value -1 in the sort object to sort descending.

```
{ name: 1 } // ascending
{ name: -1 } // descending
```

# Delete Document : Delete One

To delete a record, or document as it is called in MongoDB, we use the **deleteOne()** method.

The first parameter of the **deleteOne()** method is a query object defining which document to delete.

Note: If the query finds more than one document, only the first occurrence is deleted.

Delete the document with the address "Mountain 21":

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("testdb");
  var myquery = { address: 'Mountain 21' };
  dbo.collection("customers").deleteOne(myquery, function(err, obj) {
    if (err) throw err;
    console.log("1 document deleted");
    db.close();
  });
});
```

# Delete Document : Delete Many

To delete more than one document, use the `deleteMany()` method.

The first parameter of the `deleteMany()` method is a query object defining which documents to delete.

Delete all documents where the address starts with the letter "O":

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("testdb");
  var myquery = { address: /^O/ };
  dbo.collection("customers").deleteMany(myquery, function(err, obj) {
    if (err) throw err;
    console.log(obj.result.n + " document(s) deleted");
    db.close();
  });
})
```

# The Result Object

The `deleteMany()` method returns an object which contains information about how the execution affected the database.

Most of the information is not important to understand, but one object inside the object is called "result" which tells us if the execution went OK, and how many documents were affected.

The result object looks like this:

```
{ n: 2, ok: 1 }
```



# Update Document: Update One

You can update a record, or document as it is called in MongoDB, by using the **updateOne()** method.

The first parameter of the **updateOne()** method is a query object defining which document to update.

Note: If the query finds more than one record, only the first occurrence is updated.

The second parameter is an object defining the new values of the document.

Update the document with the address "Valley 345" to name="Mickey" and address="Canyon 123":

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://127.0.0.1:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("testdb");
  var myquery = { address: "Valley 345" };
  var newvalues = { $set: { name: "Mickey", address: "Canyon 123" } };
  dbo.collection("customers").updateOne(myquery, newvalues, function(err, res) {
    if (err) throw err;
    console.log("1 document updated");
    db.close();
  });
});
```

When using the **\$set** operator, only the specified fields are updated:

# Update Document: Update Many

To update all documents that meets the criteria of the query, use the **updateMany()** method.

Update all documents where the name starts with the letter "S":

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://127.0.0.1:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("testdb");
  var myquery = { address: /^S/ };
  var newvalues = {$set: {name: "Minnie"} };
  dbo.collection("customers").updateMany(myquery, newvalues,
function(err, res) {
  if (err) throw err;
  console.log(res.result.nModified + " document(s) updated");
  db.close();
});
});
```

# The Result Object

The `updateOne()` and the `updateMany()` methods return an object which contains information about how the execution affected the database.

Most of the information is not important to understand, but one object inside the object is called "result" which tells us if the execution went OK, and how many documents were affected.

The result object looks like this:

```
{ n: 1, nModified: 2, ok: 1 }
```

# Limit the Result

To limit the result in MongoDB, we use the **limit()** method.

The **limit()** method takes one parameter, a number defining how many documents to return.

Limit the result to only return **5 documents**:

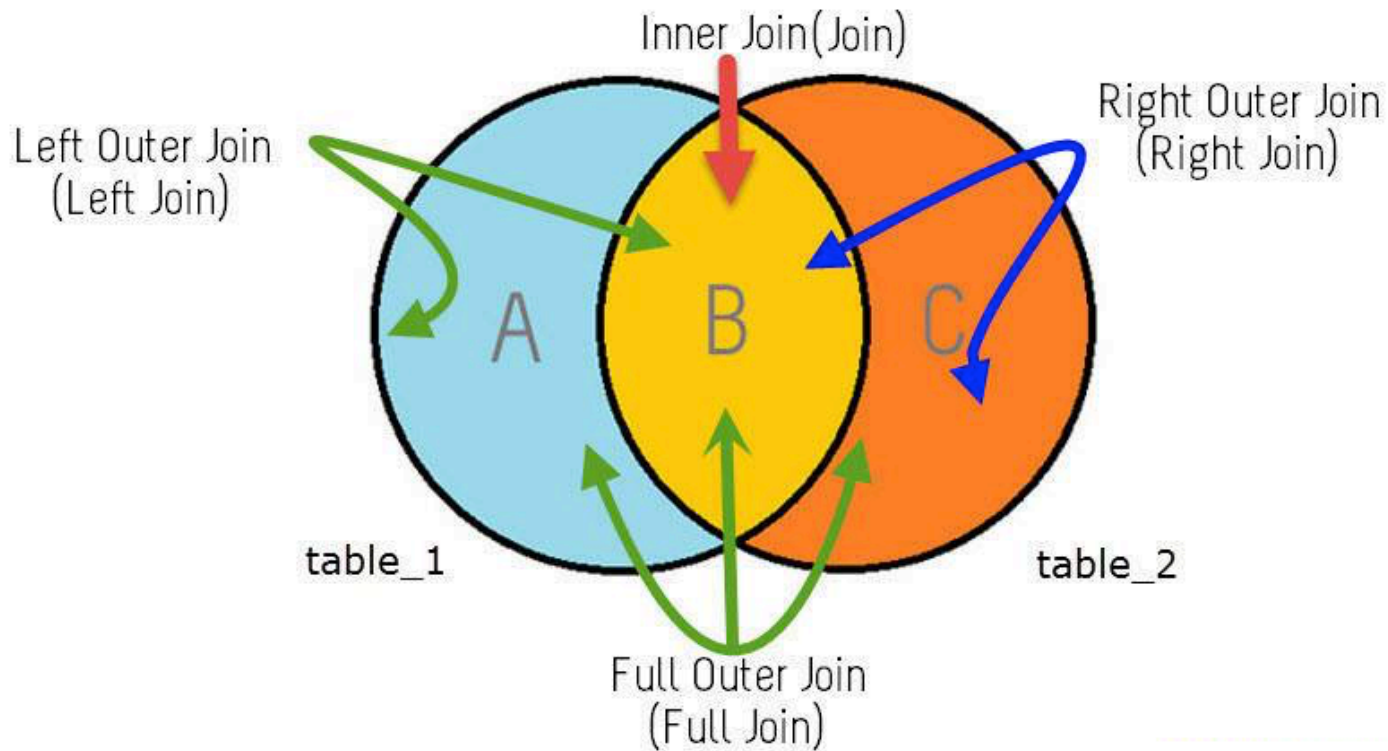
```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("testdb");
  dbo.collection("customers").find().limit(5).toArray(function(err, result) {
    if (err) throw err;
    console.log(result);
    db.close();
  });
})
```

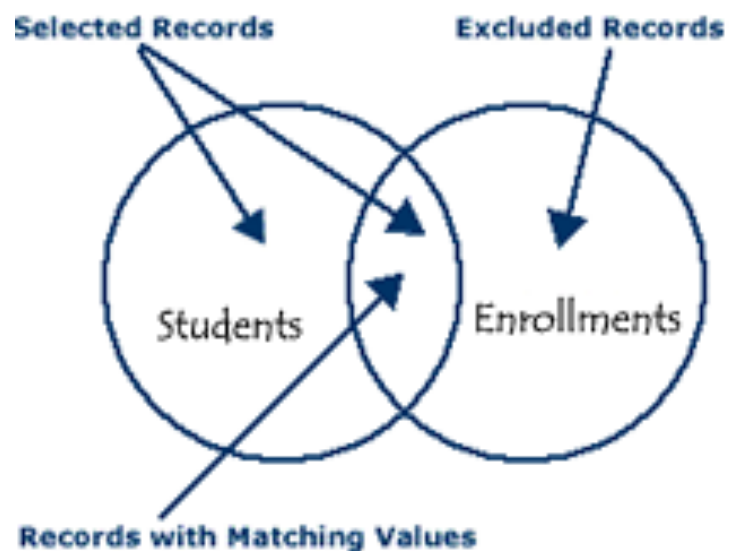
# Join Collections

MongoDB is **not a relational database**, but you can perform a **left outer join** by using the **\$lookup** stage.

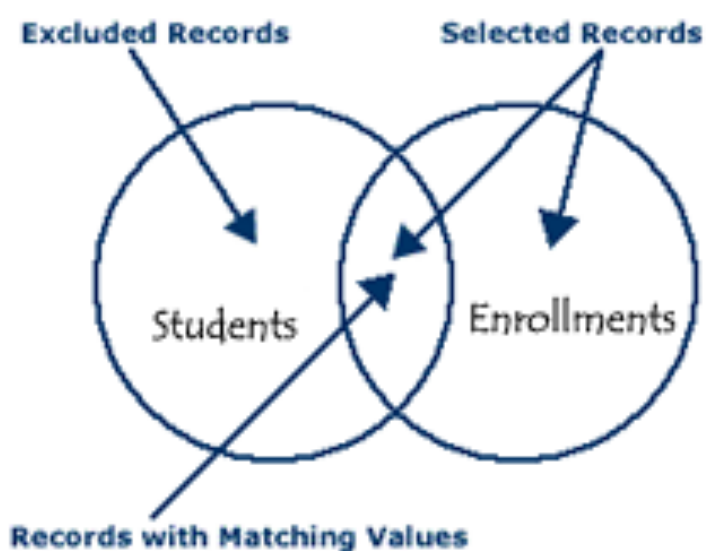
# Joins



## Left Outer Join



## Right Outer Join



# Join Collections

The **\$lookup** stage lets you specify which collection you want to join with the current collection, and which fields that should match.

Consider you have a "**orders**" collection and a "**products**" collection:

**orders :**

```
[  
  { _id: 1, product_id: 154, status: 1 }  
]
```

**products :**

```
[  
  { _id: 154, name: 'Chocolate Heaven' },  
  { _id: 155, name: 'Tasty Lemons' },  
  { _id: 156, name: 'Vanilla Dreams' }  
]
```



Join the matching "**products**" document(s) to the "**orders**" collection:

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://127.0.0.1:27017/";
```

```
MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("testdb");
  dbo.collection('orders').aggregate([
    { $lookup:
      {
        from: 'products',
        localField: 'product_id',
        foreignField: '_id',
        as: 'orderdetails'
      }
    }
  ]).toArray(function(err, res) {
    if (err) throw err;
    console.log(JSON.stringify(res));
    db.close();
  });
});
```