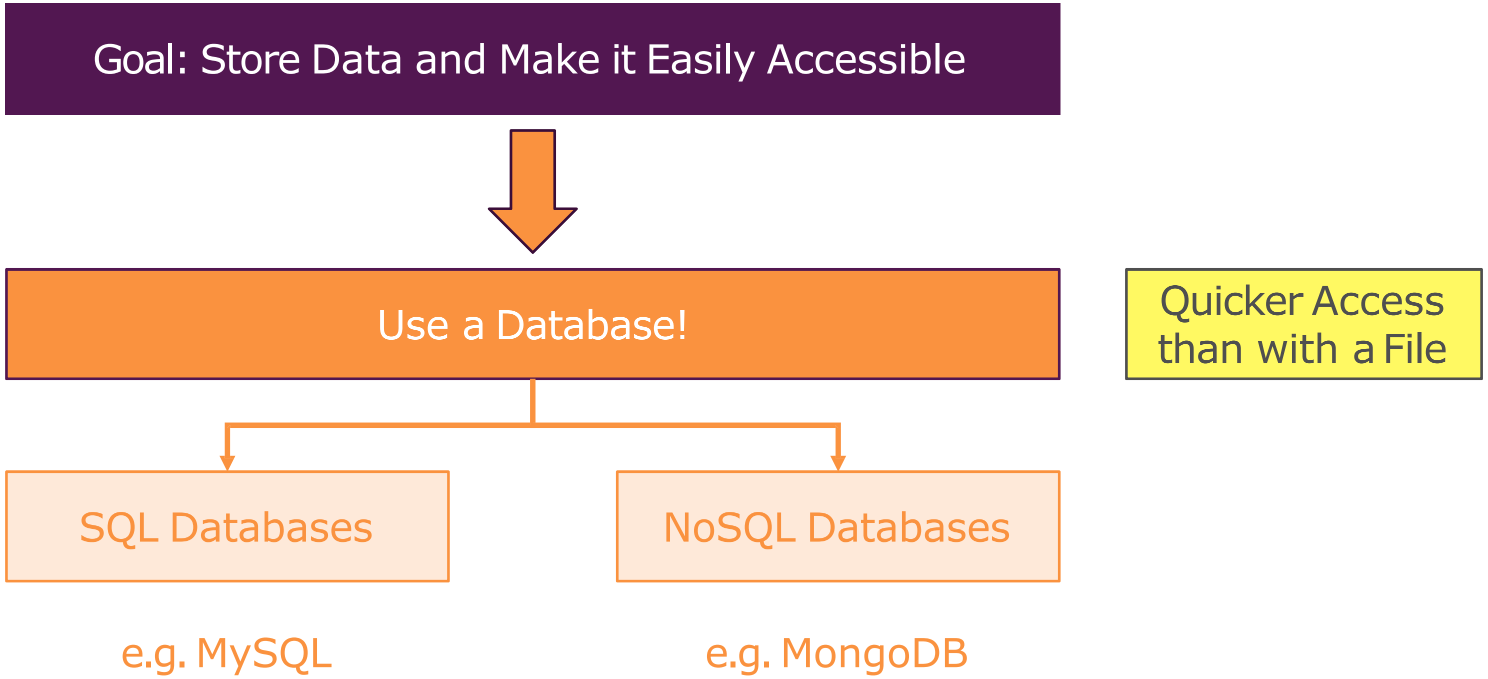
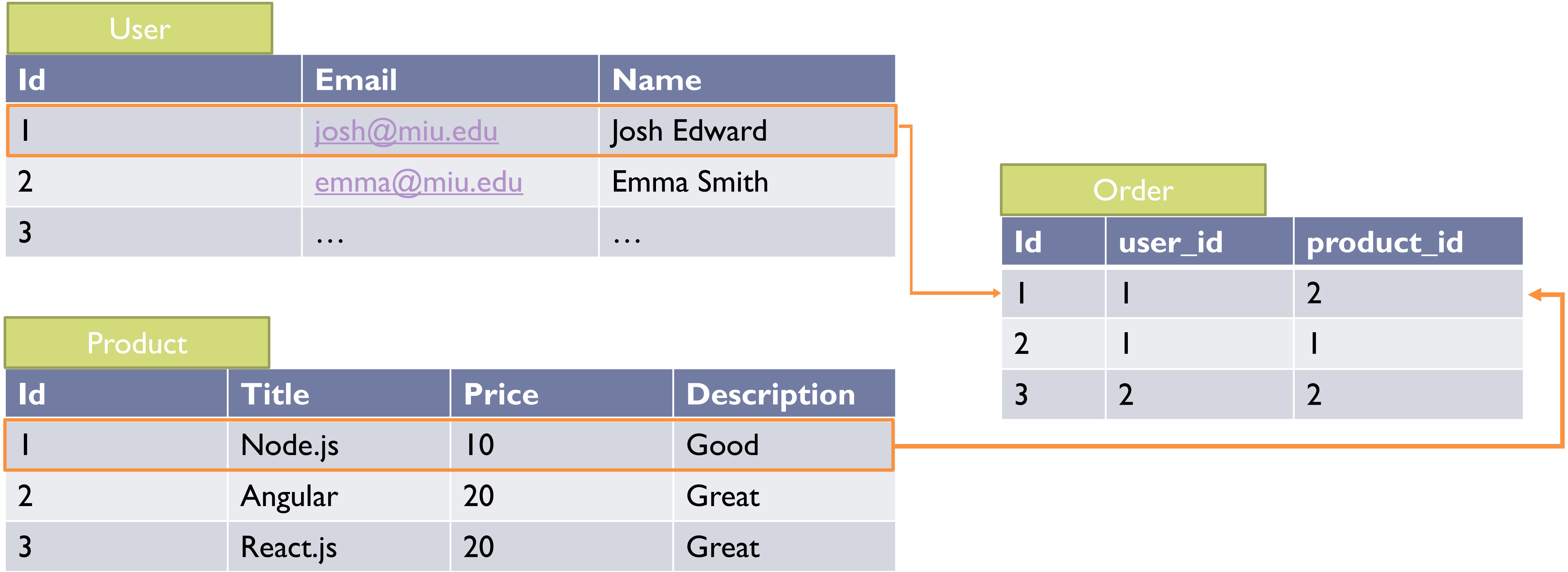
MongoDB – Intro & CRUD

SQL vs NoSQL


Goal: Store Data and Make it Easily Accessible
Use a Database!
Quicker Access than with a File
SQL Databases
NoSQL Databases
e.g. MySQL
e.g. MongoDB

What’s SQL?



User

Id	Email	Name
1	josh@miu.edu	Josh Edward
2	emma@miu.edu	Emma Smith
3	…	…

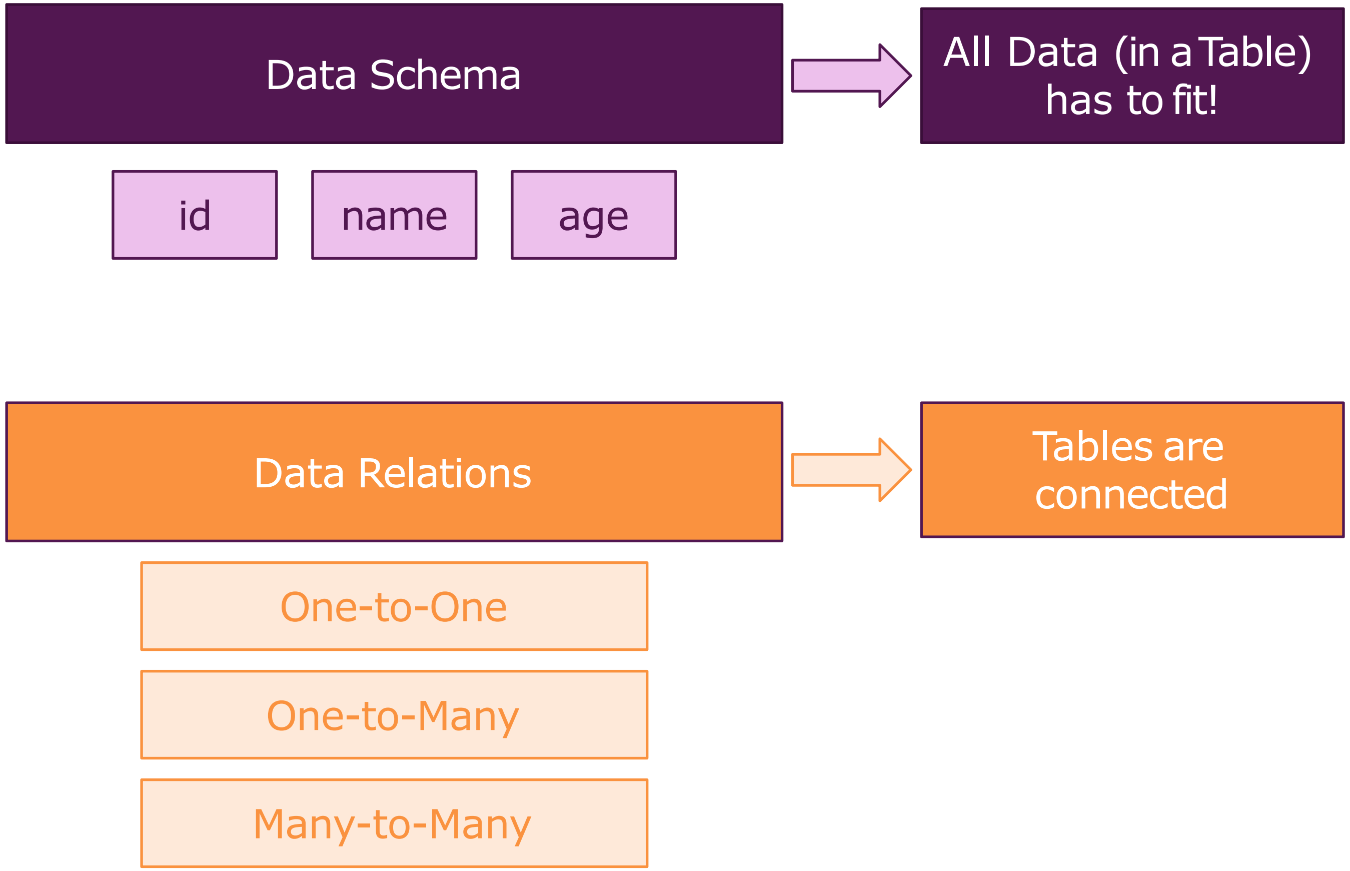
Product

Id	Title	Price	Description
1	Node.js	10	Good
2	Angular	20	Great
3	React.js	20	Great

Order

Id	user_id	product_id
1	1	2
2	1	1
3	2	2

Core SQL Database Characteristics



Data Schema → All Data (in a Table) has to fit!

id

name

age

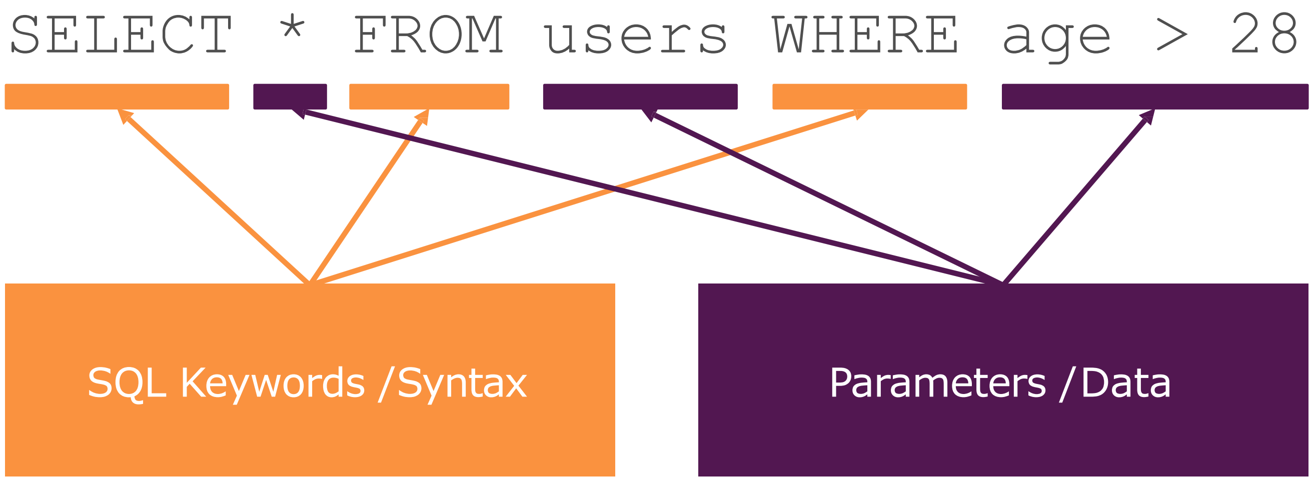
Data Relations → Tables are connected

One-to-One

One-to-Many

Many-to-Many

SQL Queries

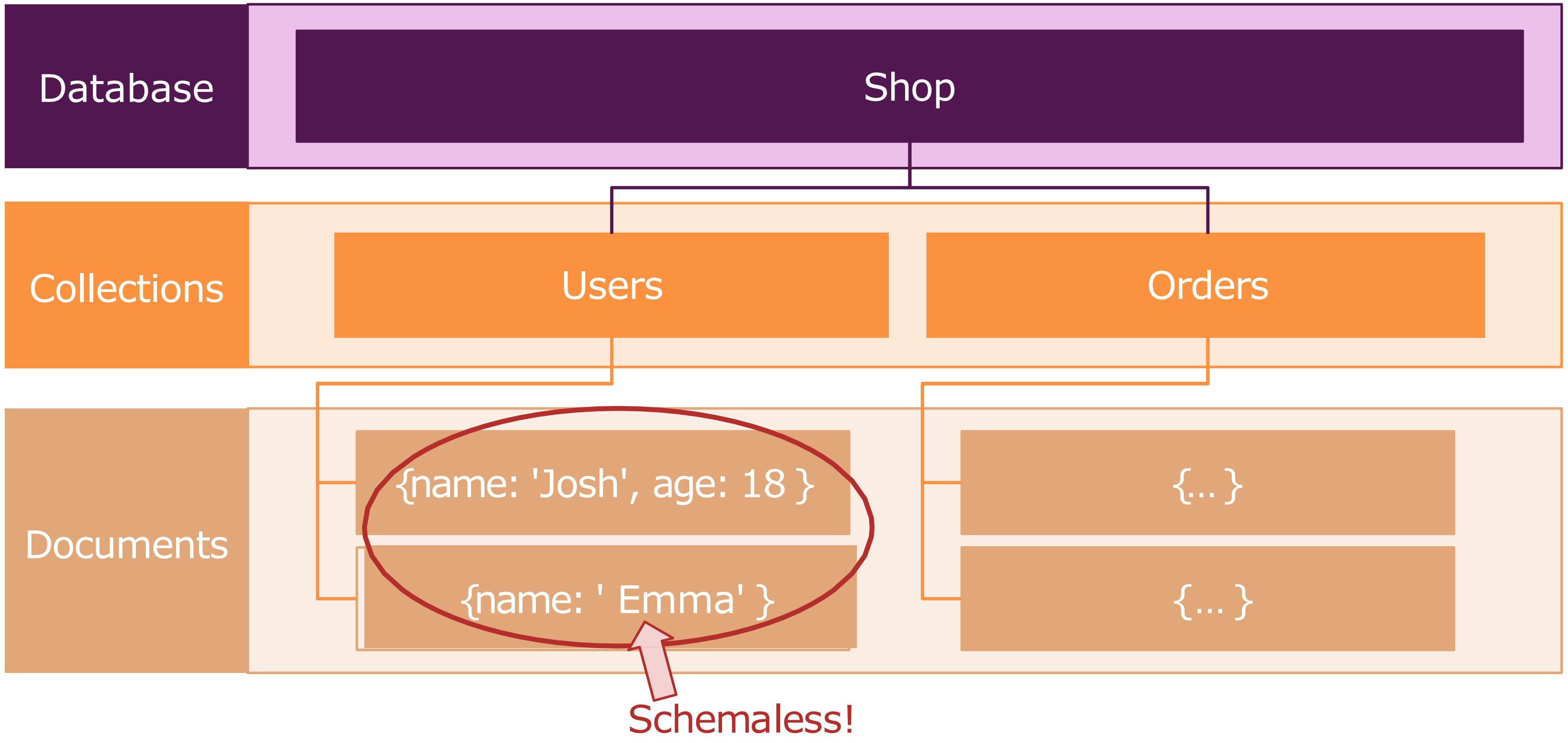


```
SELECT * FROM users WHERE age > 28
```

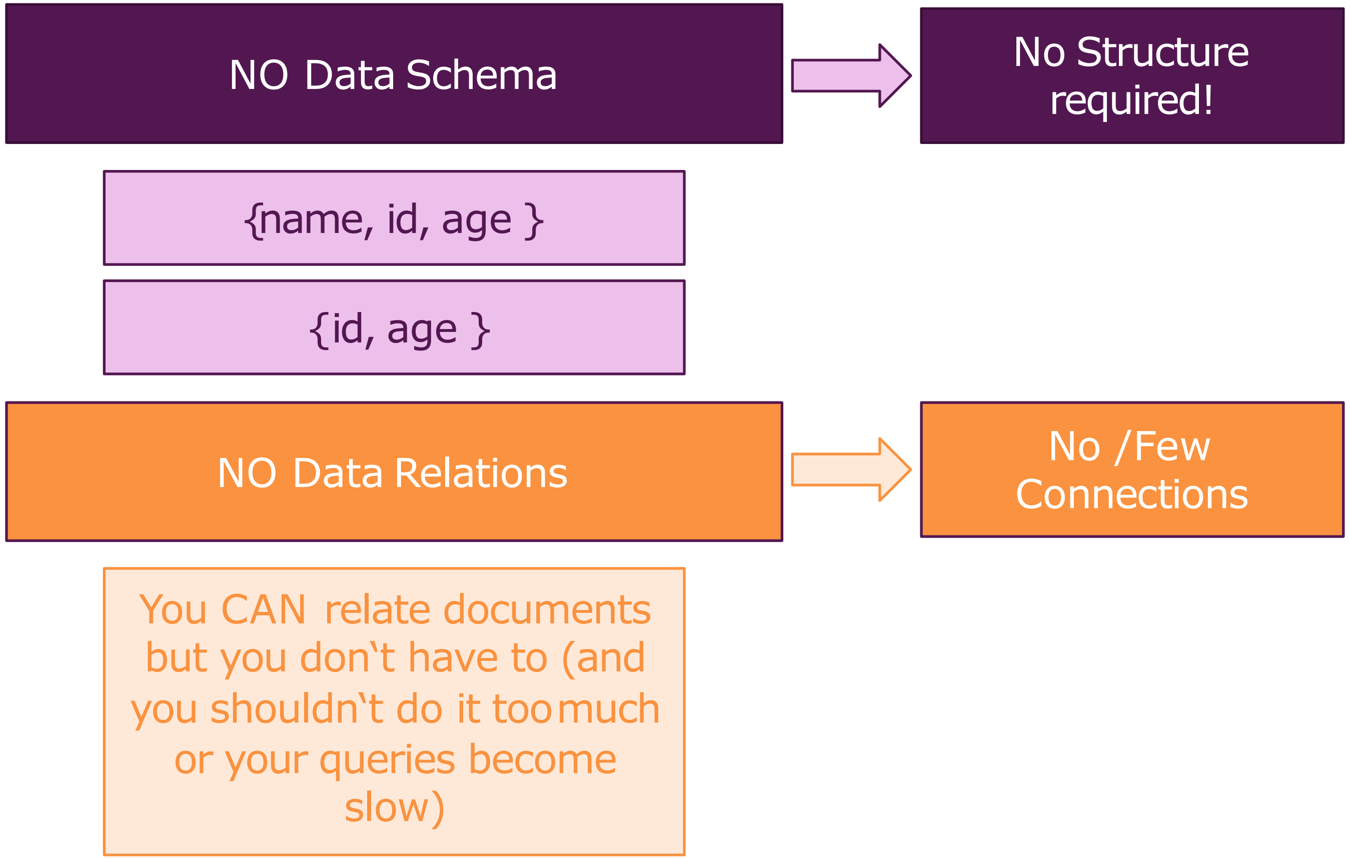
SQL Keywords /Syntax

Parameters /Data

NoSQL


Database
Shop
Collections
Users
Orders
Documents
{name: 'Josh', age: 18 }
{name: ' Emma' }
{… }
{… }
Schemaless!

NoSQL Characteristics



NO Data Schema

No Structure required!

{name, id, age }

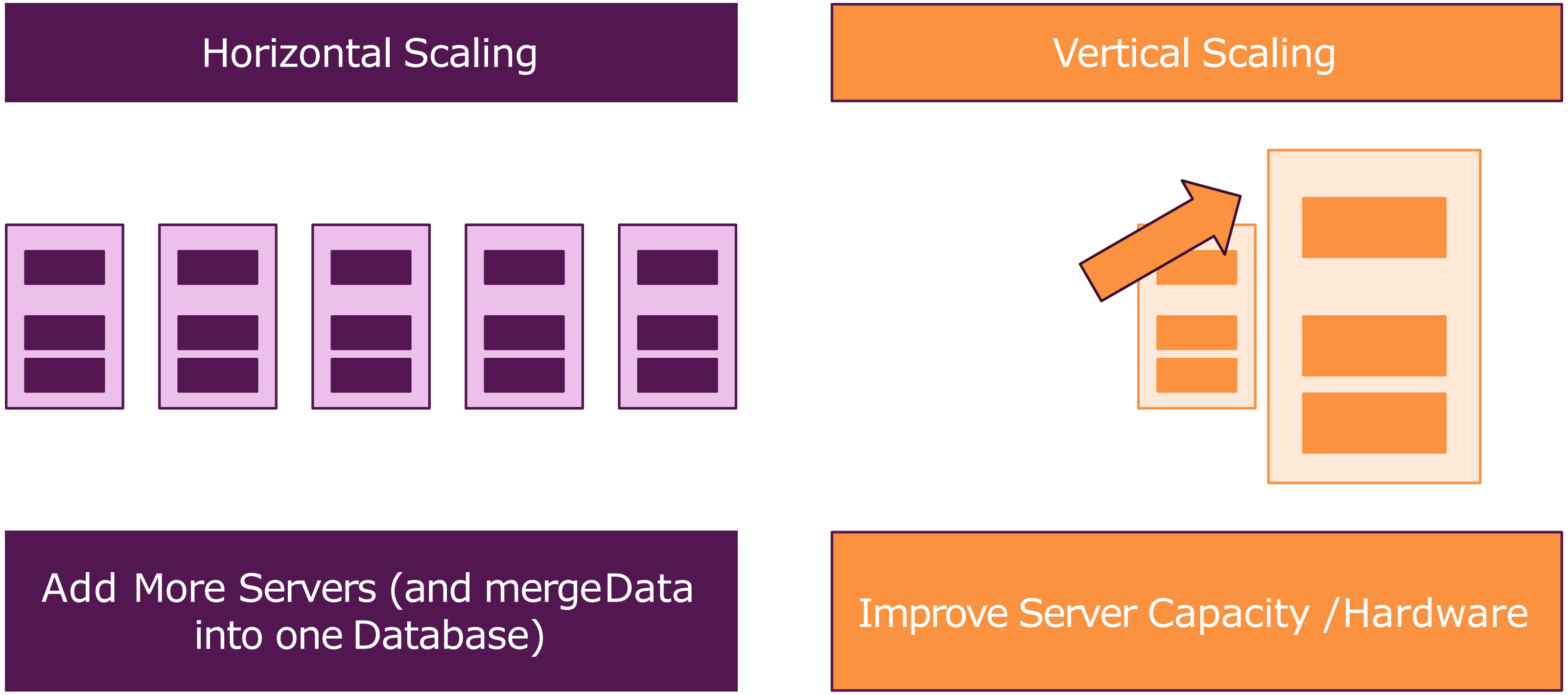
{id, age }

NO Data Relations

No /Few Connections

You CAN relate documents but you don‘t have to (and you shouldn‘t do it too much or your queries become slow)

Horizontal vs Vertical Scaling



Horizontal Scaling

Vertical Scaling

Add More Servers (and mergeData into one Database)

Improve Server Capacity /Hardware

SQL vs NoSQL

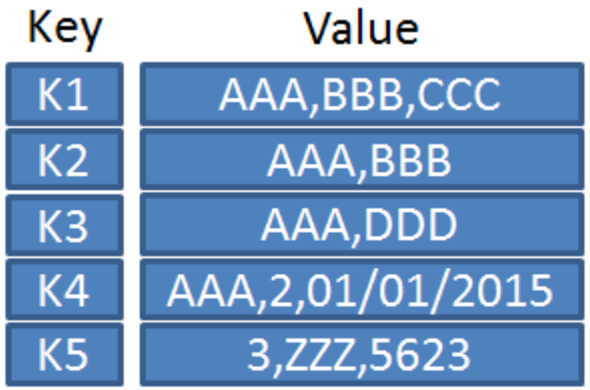
SQL	NoSQL
Data uses Schemas	Schema-less
Relations!	No (or very few) Relations
Data is distributed across multiple tables	Data is typically merged / nested in a few collections
Horizontal scaling is difficult / impossible; Vertical scaling is possible	Both horizontal and vertical scaling is possible
Limitations for lots of (thousands) read & write queries per second	Great performance for mass read & write requests

NoSQL Revolution

- NoSQL (originally referring to "non SQL" or "non relational") databases were created for "Big Data" and Real-Time Web Applications, it provides new data architectures that can handle the ever-growing velocity and volume of data.

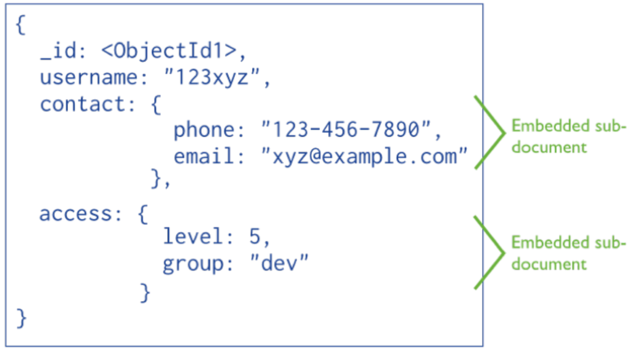
Name	Year	Type	Developer
MongoDB	**2008**	**Document**	**10Gen**
CouchDB	2005	Document	Apache
Cassandra	2008	Column Store	Apache
CouchBase	2011	Document	Couchbase
Riak	2009	Key-Value	Basho Technologies
SimpleDB	2007	Document	Amazon
BigTable	2015	Column Store	Google
Azure Cosmos DB	2017	Multi-Model	Microsoft

NOSQL Database Types



Key	Value
K1	AAA,BBB,CCC
K2	AAA,BBB
K3	AAA,DDD
K4	AAA,2,01/01/2015
K5	3,ZZZ,5623

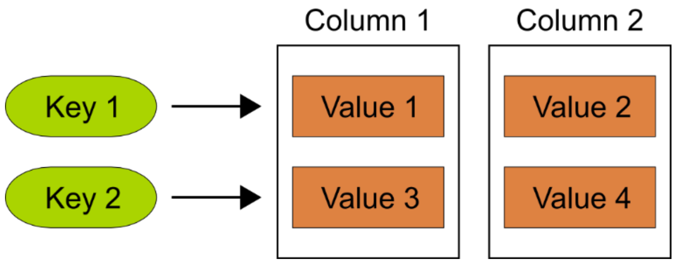
Key-Value Stores



```
{
   _id: <ObjectId1>,
   username: "123xyz",
   contact: {
               phone: "123-456-7890",
               email: "xyz@example.com"
            },
   access: {
              level: 5,
              group: "dev"
           }
}
```

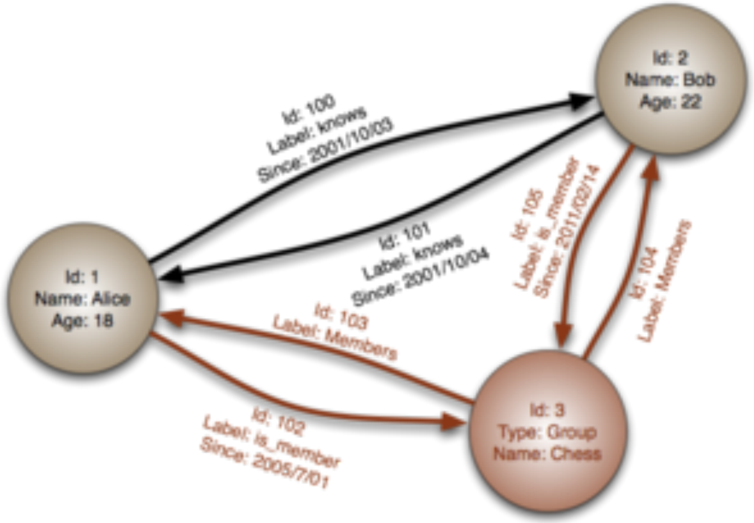
Embedded sub-document
Embedded sub-document

Document Databases



Column Family Stores

Key-Value pairs in hash table, always unique key. Logical group of keys are called: buckets
Document Databases uses Key-Value pairs in a document (JSON, BSON)
Column Stores data is stored in cells that are grouped in columns of data rather than rows (unlimited columns)
Graph Databases, uses flexible graphical representation (edges and nodes) instead of k/v pairs. Index free. Very fast for associative data sets and maps.



Graph Databases

What is MongoDB?

- MongoDB is an open-source document database that provides high performance, high availability, and automatic scaling.
- Non relational DB, stores BSON documents.
- Schemaless: Two documents don't have the same schema.

Document Data Model

- A record in MongoDB is a Document
- Structure of key/value pairs
- Values may contain other documents, arrays and arrays of documents.

```
{
    _id: 1,
    firstname: "Josh",
    lastname: "Edward",
    email: "test@mim.edu",
    phones: ["6414511111", "6414512222"]
}
```

BSON

- BSON, short for Binary JSON, is a binary-encoded serialization of JSON-like documents.
- Both JSON and BSON support Rich Documents (embedding documents and arrays within other documents and arrays).
- BSON also contains extensions that allow representation of data types that are not part of the JSON spec. (For example, BSON has a BinData ObjectId, 64 bits Integers and Date type…etc)

BSON characteristics

- Lightweight
  - Keeping spatial overhead to a minimum is important for any data representation format, especially when used over the network.
- Traversable
  - BSON is designed to be traversed easily. This is a vital property in its role as the primary data representation for MongoDB.
- Efficient
  - Encoding data to BSON and decoding from BSON can be performed very quickly in most languages. For example, integers are stored as 32 (or 64) bit integers and they don't need to be parsed to and from text.

Non-Relational

- Scalability and Performance *(embedded data models reduces I/O activity on database system)*
- Depth of Functionality *(Aggregation framework, Text Search, Geospatial Queries)*
- To retains scalability
  - MongoDB **does not support Joins** between two collections *($loopup)*
  - **No relational algebra:** tables/columns/rows *(SQL)*
  - **No Transactions** across multiple collections *(Do it programmatically, documents can be accessed atomically)*

Schema

- By default, a collection does not require its documents to have the same schema, the documents in a single collection do not need to have the same set of fields and the data type for a field can differ across documents within a collection.
- Starting of MongoDB 3.2, you can enforce document validation rules for a collection during update and insert operations

Document Structure

- The value of a field can be any of the BSON data types, including other documents, arrays, and arrays of documents.

```
const doc = {
    _id: new ObjectID('5e44ab7638d4f738f05c57a8'),
    name: { first: "Josh", last: "Edward" },
    birth: new Date('Oct 31, 1979'),
    email: "test@mim.edu",
    phones: ["6414511111", "6414512222"]
}
```

Setup

- Follow the link to install MongoDB
  - https://docs.mongodb.com/manual/tutorial/install-mongodb-on-windows/
- Two ways to start your MongoDB
  - as a Windows Service
    1. From the Services console, locate the MongoDB service.
    2. Right-click on the MongoDB service and click **Stop** (or **Pause**).
  - from the Command Interpreter
    1. Create database directory - C:/data/db
    2. Start your MongoDB database.
       - "C:\Program Files\MongoDB\Server\4.2\bin\mongod.exe" --dbpath="c:\data\db"
    3. Connect to MongoDB
       - "C:\Program Files\MongoDB\Server\4.2\bin\mongo.exe"

Collections

- MongoDB stores documents in collections. (Collections are similar to tables in relational databases)

```
use myDB
```

- If a database/collection does not exist, MongoDB creates the db/collection when you first store data for that collection

```
use myNewDB
db.myNewCollection.insert( { x: 1 } )
```

- *The insert() operation creates both the database myNewDB and the collection myNewCollection if they do not already exist.*

Shell Demo

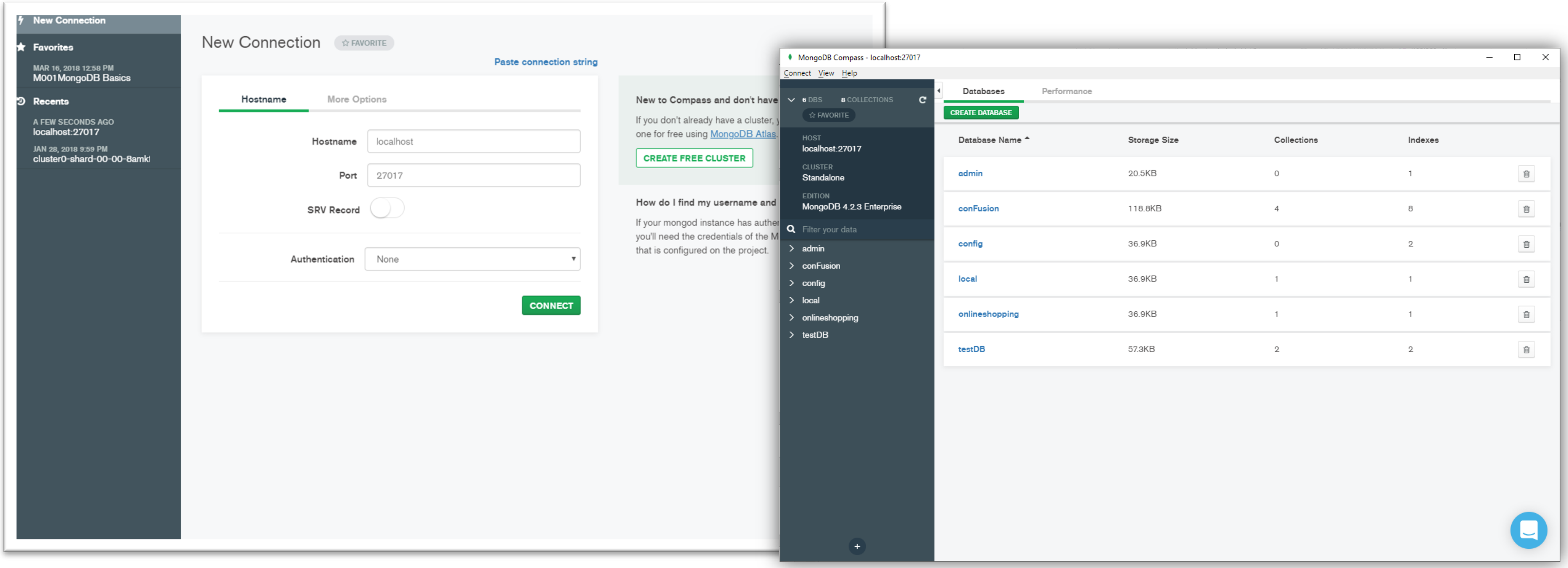
```
show dbs
use testDB // switch or create
show collections
db.testCol.insert({"name": "Josh"})// db var refers to the current database
db.testCol.find() // notice _id
// passing a parameter to find a document that has a property "name" and value "Josh"
db.testCol.find({"name":"Josh"})
// save() = upsert if _id provided
db.testCol.save({"name":"Mike"})
// insert 10 documents – Shell is C++ app that uses V8
for (var i=0; i<10; i++){ db.testCol.insert({"x": i}) }
```

Shell Demo

```
db.testCol.save({a:1, b:2})
db.testCol.save({a:3, b:4, fruit: ["apple", "orange"] })
db.testCol.save({name: "Josh", address: {city: "Fairfield",
                                zip: 52557,
                                street: "1000 N 4th street"}
})
// show documents in a nice way, it will only work when you have nested or larger documents:
db.testCol.find().pretty()
```

MongoDB Compass

- As the GUI for MongoDB, MongoDB Compass allows you to make smarter decisions about document structure, querying, indexing, document validation, and more. Commercial subscriptions include technical support for MongoDB Compass.



Hostname: localhost

Port: 27017

SRV Record

Authentication: None

HOST
localhost:27017

CLUSTER
Standalone

EDITION
MongoDB 4.2.3 Enterprise

Database Name ▲	Storage Size	Collections	Indexes
admin	20.5KB	0	1
conFusion	118.8KB	4	8
config	36.9KB	0	2
local	36.9KB	1	1
onlineshopping	36.9KB	1	1
testDB	57.3KB	2	2

General Rules

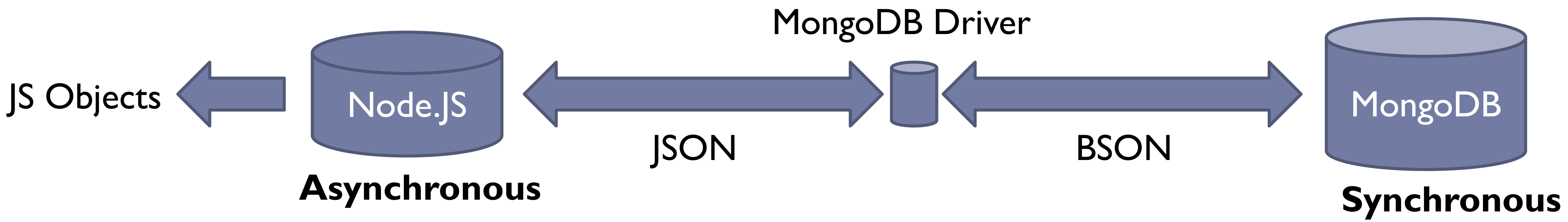
- Field names are strings.
- The field name `_id` is reserved for use as a primary key. It is immutable and always the first field in the document. It may contain values of any BSON data type, other than an array.
- The field names cannot start with the dollar sign ($) character and cannot contain the dot (.) character or `null`. Field names cannot be duplicated.
- The maximum BSON document size is 16 megabytes. *(To store documents larger than the maximum size, MongoDB provides the GridFS API)*

MongoDB Driver

- A library written in JS to handle the communication, open sockets, handle errors and talk with MongoDB Server.

```
npm install mongodb
```

- Note that Mongo Shell is **Synchronous** while Node.JS is **Asynchronous**.



Connect to MongoDB – 3.0+

```
const MongoClient = require('mongodb').MongoClient;

MongoClient.connect('mongodb://localhost:27017', { useUnifiedTopology: true })
    .then(client => {
        console.log('Connected......');
        const db = client.db('testDB');
        db.collection('testCol').find().each(function(err, doc) {
            if (err) throw err;
            // Print the result.
            // Will print a null if there are no documents in the db.
            console.log(doc);
            // Close the DB
            client.close();
        });
    })
    .catch(err => console.log('Error: ', err));
```

`db.collection.findOne({query}, {projection: {} })`

Returns **one document** that satisfies the specified **query** criteria. If multiple documents satisfy the query, this method returns the first document according to the **natural order** which reflects the order of documents on the disk. If no document satisfies the query, the method returns null.

- The **`query`** is equivalent to **`where`** in SQL, it takes the form of JSON object.
- The **`project`** method accepts JSON of the following form:

```
{ field1: <boolean>, field2: <boolean> ... }
```

Notes:

- The **`findOne()`** method always includes the **`_id`** field even if the field is not explicitly specified in the projection parameter, unless you explicitly exclude it.
- The projection argument cannot mix include and exclude specifications, with the exception of excluding the **`_id`** field.

Examples - findOne()

```
// return one document with all fields
db.collection.findOne({})

// return one document with two fields "_id" and "name"
db.collection.findOne({}, { projection: {name: 1} )

// return one document that has "name" property with value ”Umur",
   this document will have all fields but "_id" and "birth"
db.collection.findOne({name: ‘Umur'},{ projection: { _id: 0, birth: 0 } })
```

`db.collection.find({query}).project({projection})`

Selects documents in a collection and returns a **`cursor`** to the selected documents.

`cursor`: A pointer to the result set of a query. Clients can iterate through a cursor to retrieve results. By default, cursors timeout after 10 minutes of inactivity.

Notes:

- Executing **`find()`** in the mongo shell automatically iterates the cursor to display the first 20 documents. Type **`it`** to continue iteration.

Examples - `find()`

```
// returns all documents in a collection
db.collection.find({})

// It works also for Array type fields:
   return all documents where the tags field value is CS572
   { _id: 1, tags: [ "CS472", "CS572", "CS435" ] }
db.collection.find({ tags: "CS572" })
```

count()

We can use **count()** method exactly like **find()** to get the count of all the documents that match a certain criteria.

```
// returns number of all documents in the collection
db.collection.count()

// returns number of students who received A
db.collection.count({ "grade": "A" })
```

Example - Using `findOne()`

```
db.collection.findOne({'grade' : 100}, function(err, doc) {
        console.dir(doc);
});
```

`console.dir` vs `console.log`

`console.log()` only prints out a string, whereas `console.dir()` prints out a navigable object tree

Example - Using `find()`

```
db.collection.find({'grade' : 100}).toArray(function(err, docsArr) {
        console.dir(docsArr);
});
```

`toArray()` will buffer all data in memory as array before processing the callback function.

Example - Using `find()` with cursors

```
const cursor = db.collection.find({'grade' : 100});

cursor.forEach(function(err, doc) {
        console.dir(doc.student);
});
```

Behind the scene, MongoDB sends the data in batches (stream) is doesn’t send everything at once. The cursor will send a new request every time it finishes processing the batch.

Example - Using `find()` with projection

```
var query = { 'grade' : 100 };
var projection = { 'student' : 1, '_id' : 0 };

db.collection('grades').find(query).project(projection).toArray(function(err, docsArr){
        if(err) throw err;
        docsArr.forEach(function (doc) {
                console.dir(doc.student);
        });
        db.close();
});
```

Note: Projection is a good practice to save bandwidth and retrieve only the data we need.

sort() limit() skip()

Similar to SQL language, MongoDB provides certain methods on the collection object, they work as instructions sent to DB to affect the retrieval of data, all these methods will return a cursor back (chain):

SQL	MongoDB Method
Order by	sort()
Limit	limit()
Skip	skip()

Note: These will set instructions to DB server to process the information before its being sent to client.
No processing will ever happen at the client side.

Example - Skip, Limit and Sort

```
const cursor = db.collection.find({});
cursor.skip(10);
cursor.limit(5);
cursor.sort('grade', 1); //cursor.sort([['grade', 1], ['student', -1]]);

cursor.forEach(function(err, doc) {
       console.dir(doc);
});
```

Note: These will be implemented in the DB in a very specific order: **1. sort, 2. skip, 3. limit** no matter how we put them in the code

Example - Skip, Limit and Sort

```
const MongoClient = require('mongodb').MongoClient;
const client = new MongoClient('mongodb://localhost:27017’);

client.connect(function(err) {
  const db = client.db('myDB’);
  const collection = db.collection('myCollection’);

  const options = { 'skip' : 10, 'limit' : 5, 'sort' : ['grade', 1] };
  const cursor = collection.find({}, options);

  cursor.forEach(function(err, doc) {
            console.dir(doc);
  });
});
```

Example - Using insert()

```
var doc = { 'student' : ‘Umur', 'grade' : 100 };

db.collection.insert(doc, function(err, docInserted) {
        console.dir(`Success: ${docInserted}`);
});



var docs = [ { 'student' : 'Kevin', 'grade' : 90 },
             { 'student' : 'Susie', 'grade' : 95 } ];

db.collection.insert(docs, function(err, docsInserted) {
        console.dir(`Success: ${docInserted}`);
});
```

Delete documents db.collection.remove()

```
// delete all documents - One by One
db.col.remove({})

// delete all students whose names start with N-Z
db.col.remove({"student": {$gt: "M"}})

// drop the collection - Faster than remove()
db.col.drop()
```

Notes

- When we want to delete large number of documents, it's faster to use drop() but we will need to create the collection again and create all indexes as drop() will take the indexes away (while remove() will keep them)
- Multi-docs remove are not atomic isolated transactions to other R/Ws and it will yield in between.
- Each single document is atomic, no other R/Ws will see a half removed document.

Example - Using `remove()`

```
var query = { 'assignment' : 'hw3' };

// remove all documents that have 'hw3' value in 'assignment'
db.collection.remove(query, function(err, removed) {
        console.dir( removed + " documents removed!");
});
```

```
{field: {operator: value} }
```

Comparison Query Operators

Can be applied on **numeric** and **string** field values

- **`$eq`** equal to →

```
{ field: { $eq: value } }
{ field: value }
```

- **`$gt`** greater than
- **`$gte`** greater than or equal to
- **`$lt`** less than
- **`$lte`** less than or equal to
- **`$ne`** not equal to
- **`$in`** matches any of the values specified in an array (implicit OR)
- **`$nin`** matches none of the values specified in an array.
- **Comma** between operators works as (implicit AND)

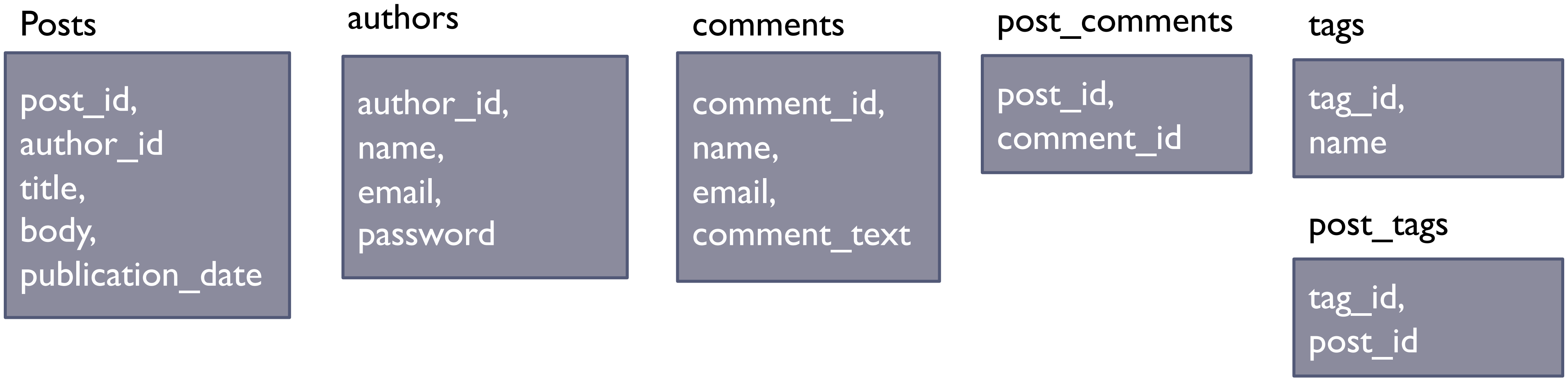
Examples - Comparison Query Operators

```
// return all documents that the score property is greater than 85
db.col.find({score: {$gt: 85}})

// return all documents where the qty field value is either 5 or 15
   { _id: 1, qty : 3 }
   { _id: 2, qty : 5 }
 db.col.find( { qty: { $in: [ 5, 15 ] } } ) // returns _id: 2

// return all documents where courses field value is either CS472 or CS572
   { _id: 1, courses: [ "CS472", "CS572", "CS435" ] }  (implicit OR)
db.col.find( { courses: { $in: ["CS572", "CS472"] } } )
```

Note: Because different values types for the same field is possible, MongoDB will do strongly/dynamically typed comparison operations.

Modeling

Let’s assume that we want to model a blog with these relational tables



Posts

post_id,
author_id
title,
body,
publication_date

authors

author_id,
name,
email,
password

comments

comment_id,
name,
email,
comment_text

post_comments

post_id,
comment_id

tags

tag_id,
name

post_tags

tag_id,
post_id

In order to display a blog post with its comments and tags, how many tables will need to be accessed?

Modeling Introduction

```
// posts collection
{ title: '',
  body: '',
  user: '', // no need for ID
  date: '',
  comments: [ {user: '', email: '', comment: ''},
              {user: '', email: '', comment: ''}]
  tags: ['', '', ''] }

// authors collection
{ user:'',
  password:''}
```

Why did we embed *tags* or *comments*? Rather than have them in separate collection? Because they need to be accessed at the same time we access the *post*. We don't need to access *comments* or *tags* independently without accessing the *post*.

Given the document schema that we proposed for the blog, how many collections would we need to access to display the blog home page?

MongoDB Schema Design

In MongoDB we use **Application-Driven Schema**, which means we design our schema based on **how we access the data**.

Note: The only scenario we cannot embed is when data exceeds 16 MB and we need to put it in separate collection.

Design Considerations

Posts

```
{ _id,
user_id
title,
body,
shares_no
date }
```

users

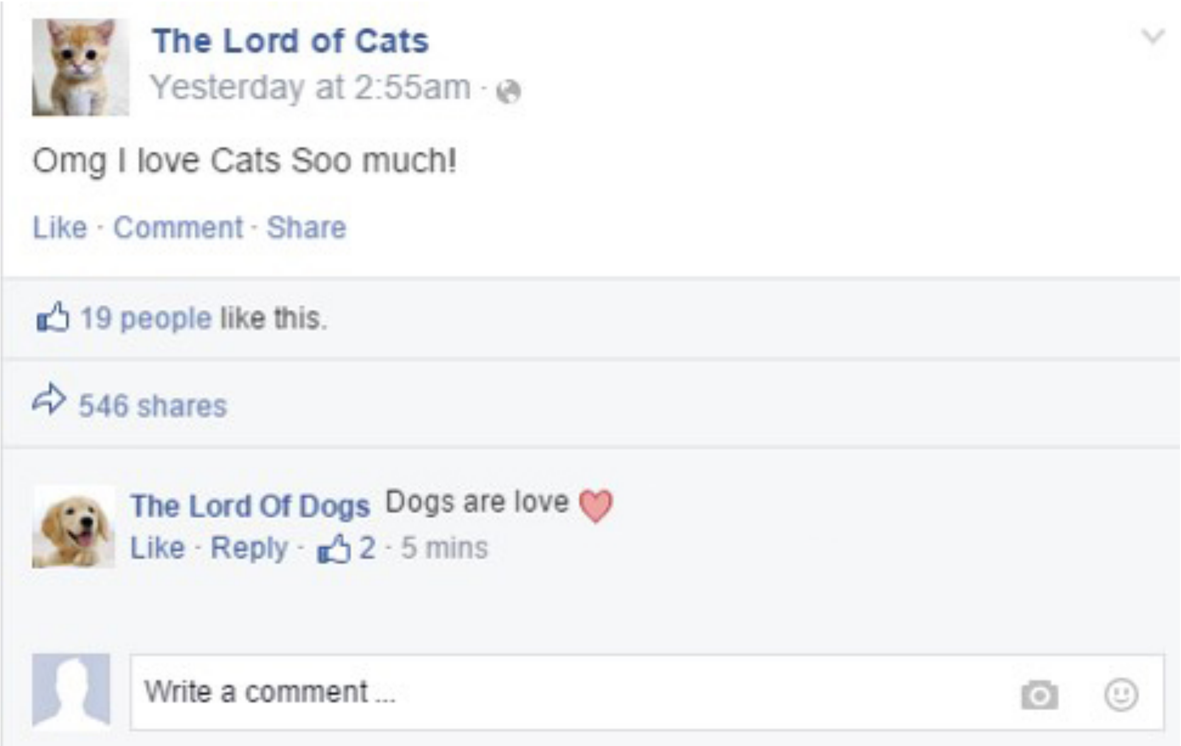
```
{ _id,
name,
email,
password }
```

comments

```
{ _id,
user_id,
post_id
comment_text
order }
```

post_likes

```
{ post_id,
user_id }
```



The Lord of Cats
Yesterday at 2:55am ·

Omg I love Cats Soo much!

Like · Comment · Share

19 people like this.

546 shares

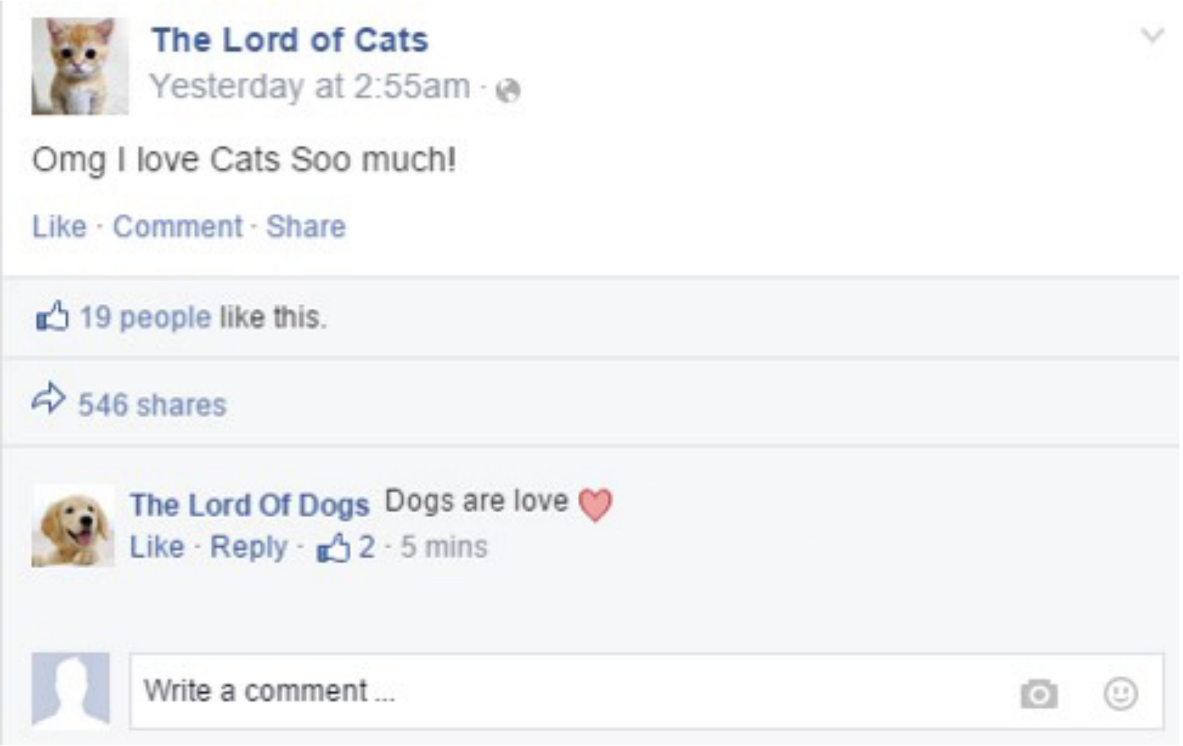
The Lord Of Dogs Dogs are love
Like · Reply · 2 · 5 mins

Write a comment ...

Remember that we don’t have constrains, so this design will not work as we need to perform too much work (4 joins) in the code to retrieve our data

Design Considerations

```
{
  _id: objectId(),
  user: 'user1', // use it as ID
  title: '',
  body: '',
  shares_no: 0,
  date: ,
  comments: [
     {user:'user2', comment_text:''},
     {user:'user3', comment_text:''}]
  likes: [ 'user1', 'user2']
}
```



The Lord of Cats
Yesterday at 2:55am ·

Omg I love Cats Soo much!

19 people like this.

546 shares

The Lord Of Dogs Dogs are love

This design is optimized for data access pattern so we can access the information much faster with 1 query. Especially that there is no need for data to be updated later.

Using MongoDB CRUD operations in Node/Express

`<code examples>`

Example - Using find() with query & projection

```
const MongoClient = require('mongodb').MongoClient;
const { ObjectID } = require('mongodb');

MongoClient.connect('mongodb://localhost:27017', { useUnifiedTopology: true }, function (err,
client) {
    if (err) throw err;

    const db = client.db('onlineshopping');

    const query = { _id: new ObjectID('5e44ab7638d4f738f05c57a8') };
    const projection = { title: 1, imageUrl: 1, _id: 0 };

    db.collection('products').find(query).project(projection).toArray(function (err, docArr) {
        if (err) throw err;
        docArr.forEach(function (doc) {
            console.log(doc);
        });
        client.close();
    });
});
```

Note: Projection is a good practice to save bandwidth and retrieve only the data we need.

Resources

- SQL vs NoSQL: https://academind.com/learn/web-dev/sql-vs-nosql/
- Mongo Shell: https://docs.mongodb.com/manual/mongo/
- MongoDB CRUD Operations: https://docs.mongodb.com/manual/crud/
- Node.js MongoDB Driver API: https://mongodb.github.io/node-mongodb-native/3.5/api/