NoSQL

(Or “Not Only SQL”?)

Databases that aren’t based on tabular models and SQL

Many different models:
Document, Key Value,
Column, Graph

Advantages

Performance

Scaling

Data model

OOP

Agile

Comes at a cost

CAP Theorem

- Impossible for a distributed system to simultaneously provide all three:
  - **C**onsistency
  - **A**vailability
  - **P**artition Tolerance

NoSQL usually prefers
the last 2 and have
eventual consistency

NoSQL models

Column based

- Column groups almost look like tables
- But queries are very simplified (e.g. only query by primary key)
- Allows for massive distribution and scale
- Examples:
  - Cassandra
  - HBase

Key-Value stores

- A lot like huge hash maps
- Bunch of keys, each key has its own values (numbers, strings, lists, etc.)
    - key: “user:1000”, value: {name: ‘Moshe’}
    - key: “user:1001”, value: {name: ‘Michael’}
    - key: “project:2”, value: {owner: 1000}
- Examples:
    - Redis
    - Riak

Graph databases

- Let’s you have nodes, links and properties
- For example: Facebook (person is a node, friendships are links)
- Examples:
  - neo4j

Document based

- Documents are the building blocks - basically objects
- There’s no schema
- Comfortable for ORM
- Very nice for REST
- Examples:
  - MongoDB
  - CouchBase

MongoDB

MongoDB

- One of the most popular NoSQL databases
- JS interface
- Great replication
- Great sharding
- Easy to start working with
- Strong queries

A database contains many collections

e.g. Users, Projects, Products

Each collection contains documents which are a lot like JSON objects

Documents can contain strings, numbers, lists, nested objects, etc.

No joins

Operations

Inserting

```
db.collection('Todos').insertOne(todo, function(err, result) {
    if (err) {
        // handle error
        return;
    }

    var savedTodo = result.ops[0];
});
```

ObjectID

- Automatically created for inserted objects under the key _id
- Look like this: ObjectID(‘549d6cbecac7b55bc0a9c7fb’)

Fetching single document

```
db.collection('Todos').findOne({_id: ObjectID(todoId)}, function(err, todo) {
    if (err) {
        // handle error
        return;
    }

    if (todo === null) {
        // no todo with specified id found
    } else {
        // use our todo
    }
});
```

Fetching multiple documents

```
db.collection('Todos').find({done: true}).toArray(function (err, completedTodos) {
    if (err) {
        // handle error
        return;
    }

    // Use our completedTodos
});
```

Updating a document

```
db.collection('Todos').updateOne({_id: ObjectId(todoId},
    {$set: {done: true}}, function(err, result) {
        if (err) {
            // handle error
        }

        // our update has finished
    });
```

Queries

AND

{done: false, user: ‘Aviv’}

OR

{$or:
[{done: false}, {user: ‘Aviv’}]}

```
{
    done: false,
    $or: [{user: ‘Aviv’}, {user: ‘Elad’}]
}
```

{users: {$gt: 3}}

{users: {$lt: 10}}

{user: {$in: [‘Aviv’,
‘Elad’]}}

```
{
    users: {
        $elemMatch: {
            $in: [‘Aviv’, ‘Elad’]
        }
    }
}
```

{counts: {$all: {$gt: 5}}}

Lots more

- $size
- $regex
- $where
- $exists
- And nicer functions (findAndModify)

Bringing promises to Node

Q library

Q.when(value);

```
Q.all([promise1,
promise2]);
```

```
Q.ninvoke(db.collection('Todos').find({}), 'toArray').then(
    function(todo) {
        // do something with todo
    }
).fail(
    function(err) {
        // handle error
    }
);
```