Chapter 24

Document Databases (MongoDB)

20 March 2017

Chris Sexton

TOC

Document Databases

Document Databases

* What is a document database?

- A collection of records/documents/objects
- A method of querying those documents

* Why MongoDB?

- high performance
- high availability
- automatic scaling
- datatypes that map to native programming language constructs
- embedded documents and arrays eliminate the need for expensive JOIN operations
- dynamic schema allows for polymorphic database storage
- rich query languages
- eventually consistent

NoSQL Standard?

The time for NoSQL standards is now (http://www.infoworld.com/article/2615807/nosql/the-time-for-nosql-standards-is-now.html)

- The wild west of databases.
- No standard connectors from programming languages
- No standard interface: REST vs sockets vs ODBC vs etc.
- No standard data storage types: JSON, BSON, private, etc.
- No standard data model: Document, Column, Key-Value, Graph, Multi-Model
- No standard query language: Many times, multiple query languages for the same database!

NoSQL Standard?

A "short" list of NoSQL databases...

- Column: Accumulo, Cassandra, Druid, HBase, Vertica, SAP HANA
- Document: CouchDB, ArrangoDB, Couchbase, DocumentDB, HyperDex, Domino, MongoDB, OrientDB, RethinkDB, Caché, ElasticSearch, Informix, Lotus Notes, PostgreSQL with JSON datatypes
- Key-Value: ArrangoDB, Couchbase, FoundationDB, HyperDex, MUMPS, Memcache, Oracle NoSQL, OrientDB, Redis, Riak, RocksDB, Dynamo, Project Voldemort, LMDB, InfinityDB
- Graph: Allegro, ArrangoDB, Neo4j, OrientDB, Cayley, DataStax, OpenCog, Oracle Spatial, SAP HANA, Teradata Aster

And more.

While there are many document database systems, we will focus on MongoDB.

Postgresql as a Nosql Document Store (http://withouttheloop.com/articles/2014-09-30-postgresql-nosql/)

MongoDB Architecture

Databases, Collections, Views, and Documents. Oh My!

- Database: a set of collections
- Collection: Similar to a table. A grouping of documents.
- Views: Read-only pre-stored queries that act as collections
- Capped Collections: Fixed-size collections with insertion order-dependendent insertion and retrieval speedups
- Documents: Data objects

Documents

- Stored as individual objects
- Vaguely translatable to JavaScript objects
- Each object has a set of labeled fields with values
- Example

```
var doc = {
    _id: ObjectId("5099803df3f4948bd2f98391"),
    name: { first: "Chris", last: "Sexton" },
    hireDate: new Date('Aug 01, 2014'),
    classes: ["C311", "B461", "C346"],
    office: "LF122",
    email: "cwsexton@ius.edu"
}
```

Where JSON fails, and why Mongo does not use JSON

JSON has no representation for

- dates
- regular expressions
- typed objects
- binary data

In Mongo:

```
{ "$date": "<date repr" }
{ "$regex": "<string regex", "$options": "string of options on a regex"}
{ "$oid": "<hex id string>" }
{ "$binary": "<base64 binary string>", "$type": "<hex string representing a type>" }
```

BSON vs JSON

BSON Spec (http://bsonspec.org/)

- MongoDB stores documents in BSON format
- Binary JSON
- Much more space and scan-speed efficient than JSON
- Compatible with JavaScript objects

```
                                    \x16\x00\x00\x00                  // total document size
                                    \x02                              // 0x02 = type String
{"hello": "world"}              →   hello\x00                         // field name
                                    \x06\x00\x00\x00world\x00         // field value
                                    \x00                              // 0x00 = type EOO ('end of object')
```

Types

* BSON Native Base Types

- byte
- int32
- int64
- uint64
- double
- decimal128

Types

* Some Constructed Types

- String (2): UTF-8 strings
- Object (3): Embedded documents
- Array (4): Embedded arrays of other types
- ObjectId (7): small/unique, generated via time, machineID, processID, and random counter
- Date (9): 64-bit integer repr. time since Jan 1, 1970; supports ~290M years into past and future

And many more (https://docs.mongodb.com/manual/reference/bson-types/)

The _id field

- Required to be a unique value as a key for the collection
- Automatically generated ObjectId if omitted on new documents
- May use the ObjectId constructor: `ObjectId("123456789")`
- Automatically indexed

ObjectId

`ObjectId()` Returns a new ObjectId value. The 12-byte ObjectId value consists of:

- a 4-byte value representing the seconds since the Unix epoch,
- a 3-byte machine identifier,
- a 2-byte process id, and
- a 3-byte counter, starting with a random value.

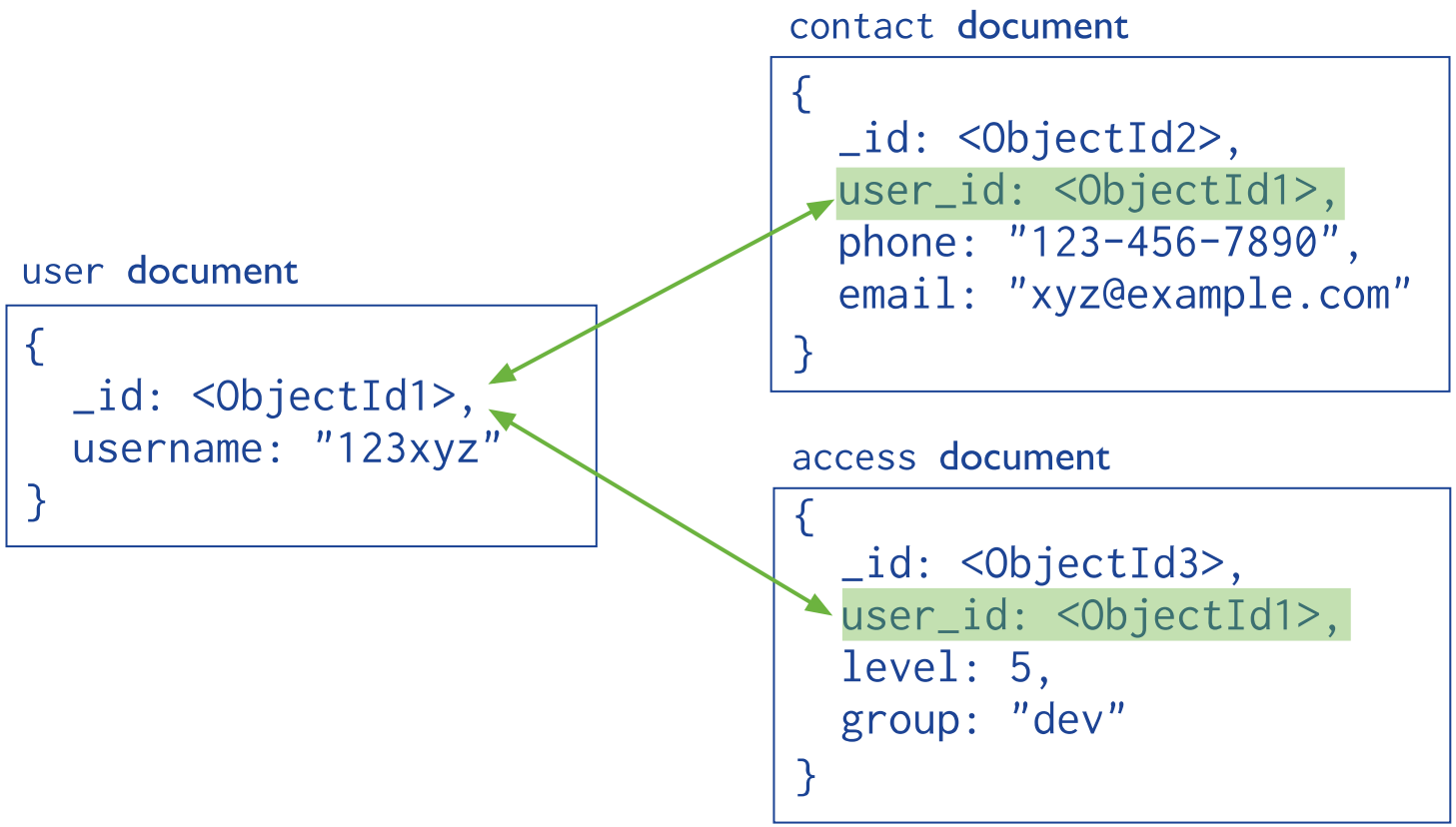
Can we be sure this is unique?

Database distributed amongst:

- Multiple threads
- Multiple processes on the same machine
- Multiple machines

Denormalization

Reference types:



Denormalization

Embedded types:

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

Embedded sub-document

Embedded sub-document

Denormalization

When to use normalization:

- when embedding would result in duplication of data but would not provide sufficient read performance advantages to outweigh the implications of the duplication.
- to represent more complex many-to-many relationships.
- to model large hierarchical data sets.

But realize this: client-side applications must issue follow-up queries to resolve the references. In other words, normalized data models can require more round trips to the server.

Using MongoDB

Installation

* Linux

  Installation on Linux (https://docs.mongodb.com/manual/administration/install-on-linux/)

Varies from distribution to distribution

* OS X

  Installation on OS X (https://docs.mongodb.com/manual/tutorial/install-mongodb-on-os-x/)

  Homebrew package manager (https://brew.sh/)

```
brew instal mongodb
```

* Windows

  Installation on Windows (https://docs.mongodb.com/manual/tutorial/install-mongodb-on-windows/)

Use the installer, may be to a custom location with no dependencies.

Hosted Installations

MongoDB Atlass (https://www.mongodb.com/cloud/atlas/pricing)

- FREE up to 512MB of storage

AWS MongoDB Quickstart (https://docs.aws.amazon.com/quickstart/latest/mongodb/welcome.html)

- FREE for one year

Azure for Students (https://azure.microsoft.com/en-us/pricing/member-offers/imagine/)

Using MongoDB on Azure (https://docs.mongodb.com/ecosystem/platforms/windows-azure/)

- FREE with student DreamSpark account

At IUS

`shannon.ius.edu`

- You may request access to shannon if you do not already have it
- You may connect to the `cscib461db` database

```
[15:08:30] cws:~ $ ssh sexton@shannon.ius.edu
sexton@shannon.ius.edu's password:
Last login: Mon Mar 20 15:06:07 2017 from 99-110-229-64.lightspeed.lsvlky.sbcglobal.net
[sexton@shannon ~]$ mongo -u cscib461user -p
MongoDB shell version v3.4.1
Enter password:
connecting to: mongodb://127.0.0.1:27017
MongoDB server version: 3.4.1
> use cscib461db
switched to db cscib461db
>
```

MongoDB Shell

`mongo` or `mongo` `[db address]`

- DB address may be a DNS name, IP address, or a combination of the former with a database identifier via `address/dbname`
- This is a JavaScript interpreter connected to the mongo instance

* Special commands

- `help` - reveal special commands available
- `show dbs` - must have admin access
- `use` <db> - switch dbs (may create one if using a non-exinstant DB)
- `show collections` - view collections in the currend database
- `exit` - quit the shell

Data Commands

insertOne/insertMany

```
db.users.insertOne(          ← collection
   {
      name: "sue",           ← field: value
      age: 26,               ← field: value
      status: "pending"      ← field: value
   }
)
```

} document

- insert into current db, must `use` <db> or connect to a specific db first
- specify collection in dot notation before insert method
- specify object to be inserted as a JavaScript object
- `insertMany` takes an array of objects

find

```
db.users.insertOne(
   {
      name: "sue",
      age: 26,
      status: "pending"
   }
)
```

collection

field: value

field: value

field: value

document

- Similar to insert (as most commands)
- returns object ID and status

```
> db.users.insertOne({ name: "Larry" });
{
    "acknowledged" : true,
        "insertedId" : ObjectId("58d00d69202496e2c3356fc8")
}
```

- More on query criteria and projection later.

updateOne/updateMany

```
db.users.updateMany(                    ← collection
   { age: { $lt: 18 } },                ← update filter
   { $set: { status: "reject" } }       ← update action
)
```

- Uses similar query criteria to find
- Returns number of records matched, modified, and status

```
> db.users.updateMany({ name: "Larry" }, { $set: { status: "reject" } });
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }

> db.users.find({ name: "Larry" });
{ "_id" : ObjectId("58d00d69202496e2c3356fc8"), "name" : "Larry", "status" : "reject" }
```

deleteOne/DeleteMany

```
db.users.deleteMany(
   { status: "reject" }
)
```

collection

delete filter

- returns number of deleted records and status

```
> db.users.deleteMany({ status: "reject" });
{ "acknowledged" : true, "deletedCount" : 1 }

> db.users.find({ name: "Larry" });
> // no results
```

SQL to MongoDB Mapping

SQL to MongoDB Mapping Chart (https://docs.mongodb.com/manual/reference/sql-comparison/)

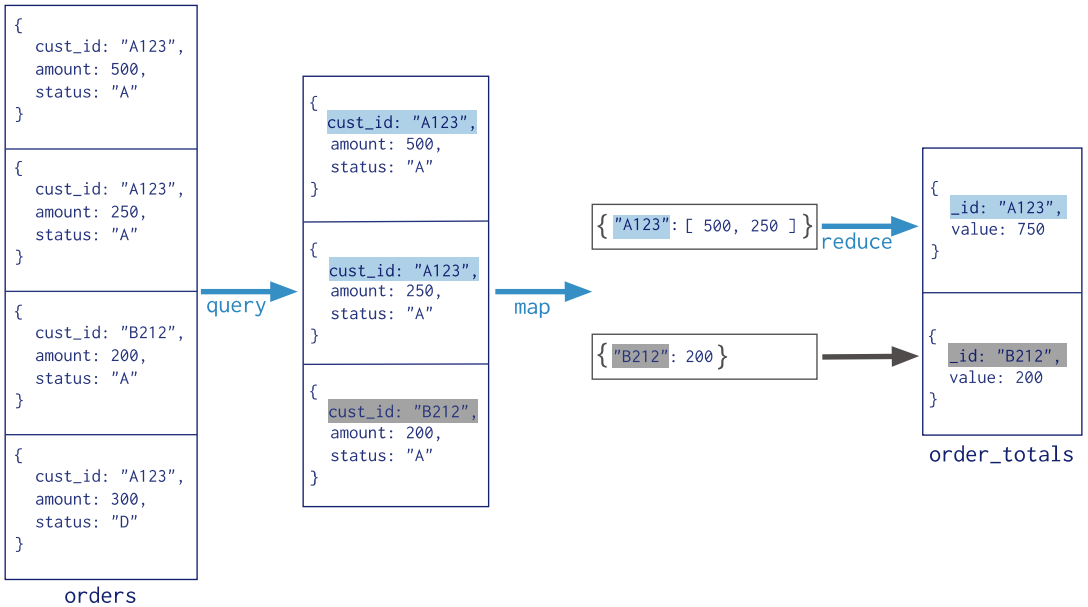
Map/Reduce

Map-Reduce is a method of aggregating large volumes of data into smaller results.

Mongo supports map-reduce as a function on collections

```
db.orders.mapReduce(
    map      → function() { emit( this.cust_id, this.amount ); },
    reduce   → function(key, values) { return Array.sum( values ) },
             → {
    query    →   query: { status: "A" },
    output   →   out: "order_totals"
             → }
)
```

Collection



Map/Reduce

- Map phase: associate many documents with a particular key value (to be reduced)
- Analogous to the GROUP BY keyword in SQL
- Reduce phase: perform some action as a rollup on the mapped data
- Analogous to the aggregate functions available in SQL statements
- Query: filter documents to be mapped
- Output: send results to a particular collection

Map/Reduce Example

```
> db.users.insertMany([
        ... { name: "Larry", age: 20, job: "idiot" },
        ... { name: "Joe", age: 32, job: "idiot" },
        ... { name: "Curly", age: 43, job: "idiot" },
        ... { name: "Graham", age: 34, job: "actor" },
        ... { name: "John", age: 67, job: "actor" },
        ... { name: "Terry", age: 45, job: "actor" },
        ... { name: "Eric", age: 31, job: "actor" },
        ... { name: "Michael", age: 28, job: "actor" },
        ... ]);
{
    "acknowledged" : true,
        "insertedIds" : [
            ObjectId("58d01c0a202496e2c3356fe1"),
        ObjectId("58d01c0a202496e2c3356fe2"),
        ObjectId("58d01c0a202496e2c3356fe3"),
        ObjectId("58d01c0a202496e2c3356fe4"),
        ObjectId("58d01c0a202496e2c3356fe5"),
        ObjectId("58d01c0a202496e2c3356fe6"),
        ObjectId("58d01c0a202496e2c3356fe7"),
        ObjectId("58d01c0a202496e2c3356fe8")
        ]
}
```

Map/Reduce Example

```
> db.users.mapReduce(
... function() { emit( this.job, this.age ); },
... function(key, values) { return Array.sum(values) / values.length; },
... { out: { inline: 1} }
... );
{
    "results" : [
       { "_id" : "actor", "value" : 41 },
       { "_id" : "idiot", "value" : 31.666666666666668 }
    ],
    "timeMillis" : 15,
    "counts" : {
        "input" : 8,
        "emit" : 8,
        "reduce" : 2,
        "output" : 2
    },
    "ok" : 1
}
```

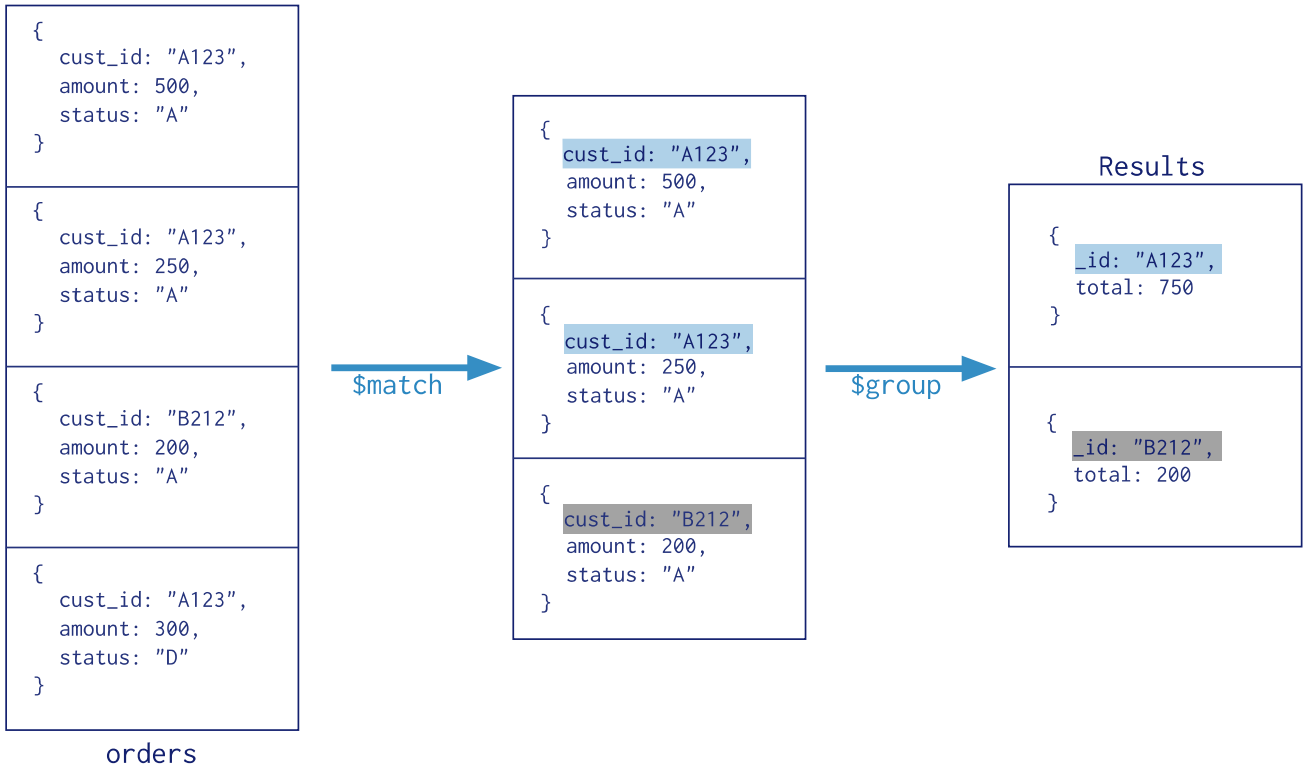
Map/Reduce Example

```
> db.users.mapReduce(
        ... function() { emit( this.job, this.age ); },
        ... function(key, values) { return Array.sum(values) / values.length; },
        ... {
        ...   query: { job: "actor" },
        ...   out: { inline: 1}
        ... }
        ... );
{
    "results" : [
        { "_id" : "actor", "value" : 41 }
    ],
    "timeMillis" : 14,
    "counts" : {
        "input" : 5,
        "emit" : 5,
        "reduce" : 1,
        "output" : 1
    },
    "ok" : 1
}
```

Aggregation Pipelines

Collection

```
db.orders.aggregate( [
    $match stage →   { $match: { status: "A" } },
    $group stage →   { $group: { _id: "$cust_id",total: { $sum: "$amount" } } }
                   ] )
```



Aggregation Example

Without `$match`

```
> db.users.aggregate([
        ... {$project: { job: 1, age: 1 }},
        ... {$group: { _id: "$job", avg: {$avg: "$age" } }}
        ... ]);
{ "_id" : "actor", "avg" : 41 }
{ "_id" : "idiot", "avg" : 31.666666666666668 }
```

With `$match`

```
> db.users.aggregate([
        ... {$match: { job: "actor" }},
        ... {$project: { job: 1, age: 1 }},
        ... {$group: { _id: "$job", avg: {$avg: "$age" } }}
        ... ]);

{ "_id" : "actor", "avg" : 41 }
```

$match

- Match conditions using query operators

```
{ <field>: { <operator>: <value> } }
```

For example:

```
db.users.find({ job: {$in: ["idiot", "actor" ]}});
```

or

```
db.users.aggregate([
    {$match: {
        job: {$in: ["idiot", "actor"]}
    }}
])
```

Match operators

Work in `find` queries and `$match` expressions

- $in
- $and
- $or
- $not
- $eq
- $gt
- $lte
- $nin
- $exists

And more (https://docs.mongodb.com/manual/reference/operator/query/#query-selectors)

$skip, $limit

- $skip: Skip the first *n* documents
- $limit: Return the first *n* documents
- Both useful on ordered queries or summaries of data

$project

- Specify which fields will be included in the output of a pipeline
- May be a 1 or 0 for explicit inclusion/exclusion
- Must explicitly exclude `_id` with `_id: 0`
- Can use expressions such as `$add`, `$divide`, `$concat`, etc

Expressions (https://docs.mongodb.com/manual/meta/aggregation-quick-reference/#aggregation-expressions)

$sort

- Exactly what it says on the label!
- Specify a set of fields with values 1 or -1 for ascending or descending orders

```
> db.users.aggregate([
... {$match: { job: "idiot" }},
... {$project: { _id: 0, name: 1, age: 1 }},
... {$sort: { age: 1 }}
... ])

{ "name" : "Larry", "age" : 20 }
{ "name" : "Joe", "age" : 32 }
{ "name" : "Curly", "age" : 43 }
```

```
> db.users.aggregate([
... {$match: { job: "idiot" }},
... {$project: { _id: 0, name: 1, age: 1 }},
... {$sort: { age: -1 }}
... ])

{ "name" : "Curly", "age" : 43 }
{ "name" : "Joe", "age" : 32 }
{ "name" : "Larry", "age" : 20 }
```

$group

- Aggregate operations such as `$avg`, `$sum`, `$max`, `$min`, `$push`
- Specify _id field as GROUP BY

```
> db.users.aggregate([
        ... {$match: { job: "actor" }},
        ... {$project: { job: 1, age: 1 }},
        ... {$group: { _id: "$job", avg: {$avg: "$age" } }}
        ... ]);

{ "_id" : "actor", "avg" : 41 }
```

- Use `_id:` ` null` to group by the entire set

```
> db.users.aggregate([
... {$project: { job: 1, age: 1 }},
... {$group: { _id: null, avg: {$avg: "$age" } }}
... ]);

{ "_id" : null, "avg" : 37.5 }
```

$push

- Adds values to arrays in a $group stage

```
> db.users.aggregate([
... {$project: { job: 1, age: 1 }},
... {$group: { _id: "$job", job: {$first: "$job" }, ages: {$push: "$age" } }}
... ]);

{ "_id" : "actor", "job" : "actor", "ages" : [ 34, 67, 45, 31, 28 ] }
{ "_id" : "idiot", "job" : "idiot", "ages" : [ 20, 32, 43 ] }
```

$unwind

- Used to unpack array values
- Often used in conjunction with $push

```
> db.users.aggregate([
... {$project: { job: 1, age: 1 }},
... {$group: { _id: "$job", job: {$first: "$job" }, ages: {$push: "$age" } }},
... {$unwind: "$ages" }
... ]);

{ "_id" : "actor", "job" : "actor", "ages" : 34 }
{ "_id" : "actor", "job" : "actor", "ages" : 67 }
{ "_id" : "actor", "job" : "actor", "ages" : 45 }
{ "_id" : "actor", "job" : "actor", "ages" : 31 }
{ "_id" : "actor", "job" : "actor", "ages" : 28 }
{ "_id" : "idiot", "job" : "idiot", "ages" : 20 }
{ "_id" : "idiot", "job" : "idiot", "ages" : 32 }
{ "_id" : "idiot", "job" : "idiot", "ages" : 43 }
```

$unwind

We will fix the "ages" key with a project

```
> db.users.aggregate([
... {$project: { job: 1, age: 1 }},
... {$group: { _id: "$job", job: {$first: "$job" }, ages: {$push: "$age" } }},
... {$unwind: "$ages" },
... {$project: { _id: 0, job: 1, age: "$ages" } }
... ]);

{ "job" : "actor", "age" : 34 }
{ "job" : "actor", "age" : 67 }
{ "job" : "actor", "age" : 45 }
{ "job" : "actor", "age" : 31 }
{ "job" : "actor", "age" : 28 }
{ "job" : "idiot", "age" : 20 }
{ "job" : "idiot", "age" : 32 }
{ "job" : "idiot", "age" : 43 }
```

Other Pipeline Stages

- $geoNear
- $count
- $bucket
- $sample
- $redact
- $out

Aggregation Quick Reference (https://docs.mongodb.com/manual/meta/aggregation-quick-reference/)

Summary

- Wide variety of Document Databases
- No standard for NoSQL
- MongoDB is one example
- Map-Reduce: A common query language for many document databases
- Aggregation pipeline: A query language for MongoDB