MongoDB and NoSQL Databases


```
{
"_id": ObjectId("5146bb52d8524270060001f3"),
“course": "csc443,
”campus": ”Byblos",
“semester": ”Fall 2017",
“instructor": ”Haidar Harmanani"
}
```


A look at the Database Market

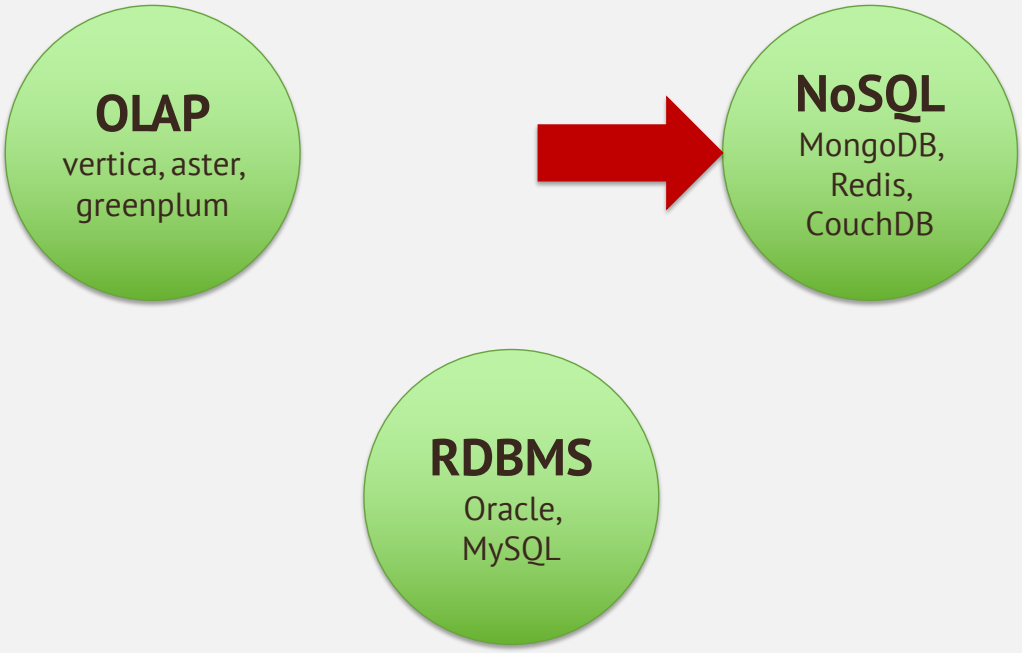


Table of Contents

What is NoSQL Database?

- Work extremely well on the web
- NoSQL (cloud) databases
  - Use document-based model (non-relational)
  - Schema-free document storage
    - Still support indexing and querying
    - Still support CRUD operations (create, read, update, delete)
    - Still supports concurrency and transactions
    - No joins
    - No complex transactions
  - Horizontally scalable
  - Highly optimized for append / retrieve
  - Great performance and scalability
  - NoSQL == "No SQL" or "Not Only SQL"?

Relational vs. NoSQL Databases

- Relational databases
  - Data stored as table rows
  - Relationships between related rows
  - Single entity spans multiple tables
  - RDBMS systems are very mature, rock solid
- NoSQL databases
  - Data stored as documents
  - Single entity (document) is a single record
  - Documents do not have a fixed structure

Relational vs. NoSQL Models

Relational Model

Name	Svetlin Nakov
Gender	male
Phone	+359333777555
Email	nakov@abv.bg
Site	www.nakov.com

* 1

Street	Al. Malinov 31
Post Code	1729

* 1

Town	Sofia

* 1

Country	Bulgaria

Document Model

Name: Svetlin Nakov

Gender: male

Phone: +359333777555

Address:

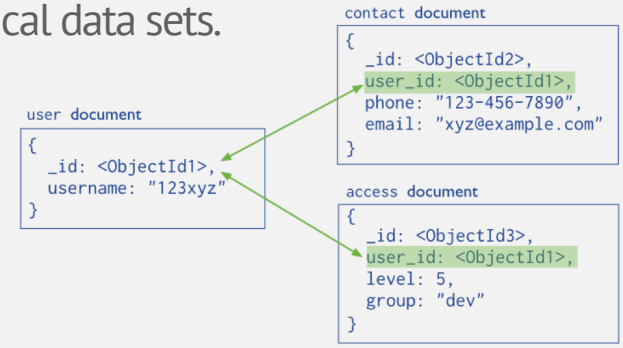
- **Street:** Al. Malinov 31
- **Post Code:** 1729
- **Town:** Sofia
- **Country:** Bulgaria

Email: nakov@abv.bg

Site:

Document oriented database – Normalized data model

- When to use:
  - 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.
  - Multiple queries!



Redis

What is Redis?

- Redis is
  - Ultra-fast in-memory key-value data store
  - Powerful data structures server
  - Open-source software: http://redis.io
- Redis stores data structures:
  - Strings
  - Lists
  - Hash tables
  - Sets / sorted sets

Hosted Redis Providers

- Redis Cloud
  - Fully managed Redis instance in the cloud
  - Highly scalable, highly available
  - Free 1 GB instance, stored in the Amazon cloud
  - Supports data persistence and replication
  - http://redis-cloud.com
- Redis To Go
  - 5 MB free non-persistent Redis instance
  - http://redistogo.com

CouchDB

What is CouchDB?

- Apache CouchDB
  - Open-source NoSQL database
  - Document-based: stored JSON documents
  - HTTP-based API
  - Query, combine, and transform documents with JavaScript
  - On-the-fly document transformation
  - Real-time change notifications
  - Highly available and partition tolerant

Hosted CouchDB Providers

- Cloudant
  - Managed CouchDB instances in the cloud
  - Free $5 account – unclear what this means
  - https://cloudant.com
  - Has nice web-based administration UI

"Big Data" is two problems

- The analysis problem
  - How to extract useful info, using modeling, ML and stats.
- The storage problem
  - How to store and manipulate huge amounts of data to facilitate fast queries and analysis
- Problems with traditional (relational) storage
  - Not flexible
  - Hard to partition, i.e. place different segments on different machines

Example: E-Commerce

- Problem: Product catalogs store different types of objects with different sets of attributes.
- This is not easily done within the relational model, need a more "flexible schema"
- Relational Solutions
  - Create a table for each product category
  - Put everything in one table
  - Use inheritance
  - Entity-Attribute-Value
  - Put everything in a BLOB

RDBMS (1): Table per Product

```
CREATE TABLE `product_audio_album`
     ( `sku` char(8) NOT NULL, ...
       `artist` varchar(255) DEFAULT NULL,
       `genre_0` varchar(255) DEFAULT NULL,
       `genre_1` varchar(255) DEFAULT NULL, ...
       PRIMARY KEY(`sku`)) ...
CREATE TABLE `product_film`
     ( `sku` char(8) NOT NULL, ...
       `title` varchar(255) DEFAULT NULL,
       `rating` char(8) DEFAULT NULL, ...
        PRIMARY KEY(`sku`)) ...
```


RDBMS (2): Single table for all

```
CREATE TABLE `product`
    ( `sku` char(8) NOT NULL, ...
       `artist` varchar(255) DEFAULT NULL,
       `genre_0` varchar(255) DEFAULT NULL,
       `genre_1` varchar(255) DEFAULT NULL, ...
       `title` varchar(255) DEFAULT NULL,
       `rating` char(8) DEFAULT NULL, ...
        PRIMARY KEY(`sku`))
```

RDBMS (3): Inheritance

```
CREATE TABLE `product`
     ( `sku` char(8) NOT NULL,
       `title` varchar(255) DEFAULT NULL,
       `description` varchar(255) DEFAULT NULL,
       `price`, ...
        PRIMARY KEY(`sku`))
CREATE TABLE `product_audio_album`
     ( `sku` char(8) NOT NULL, ...
        `artist` varchar(255) DEFAULT NULL,
        `genre_0` varchar(255) DEFAULT NULL,
        `genre_1` varchar(255) DEFAULT NULL, ...
         PRIMARY KEY(`sku`),
         FOREIGN KEY(`sku`) REFERENCES `product`(`sku`))
….
```


RDBMS (4): Entity Attribute Value

Entity	Attribute	Value
sku_00e8da9b	Type	Audio Album
sku_00e8da9b	Title	A Love Supreme
sku_00e8da9b	…	…
sku_00e8da9b	Artist	John Coltrane
sku_00e8da9b	Genre	Jazz
sku_00e8da9b	Genre	General

MongoDB Solution

- A "collection" can contain heterogeneous "documents", e.g. for an audio album we could store as

```
{ sku: "00e8da9b",
  type: "Audio Album",
  title: "A Love Supreme",
  description: "by John Coltrane",
  shipping: { weight: 6,
      dimensions: { width: 10, height: 10, depth: 1 } },
  pricing: { list: 1200, retail: 1100, savings: 100},
  details: { title: "A Love Supreme [Original Recording]",
            artist: "John Coltrane",
            genre: [ "Jazz", "General" ]}
}
```


Hosted MongoDB Providers

- MongoLab
  - Free 0.5 GB instance
  - https://mongolab.com
- MongoHQ
  - Free 0.5 GB instance (sandbox)
  - https://www.mongohq.com
- MongoOd
  - Free 100 MB instance
  - https://www.mongood.com

History

- mongoDB = "Hu**mongo**us DB"
  - Open-source
  - Document-based
  - "High performance, high availability"
  - Automatic scaling
  - C-P on CAP

History

- 2007 - First developed (by 10gen)
- 2009 - Became Open Source
- 2010 - Considered production ready (v 1.4 >)
- 2013 - MongoDB closes $150 Million in Funding
- 2015 - version 3 released (v 3.0.7)
- 2016 – Latest stable version (v. 3.2.10)
- Today- More than $231 million in total investment since 2007

History

CNBC HOME U.S. NEWS MARKETS INVESTING TECH MAKE IT VIDEO

MongoDB 20. Synack 21. Teespring 22.

19. MongoDB

Better uses for big data

Founders: Eliot Horowitz, Dwight Merriman, Kevin P. Ryan
Launched: 2007
Funding: $311.5 million
Valuation: $1.8 billion
Disrupting: Big data
Rival: Oracle

THE WALL STREET JOURNAL

Home World U.S. Politics Economy Business Tech Markets Opinion Arts

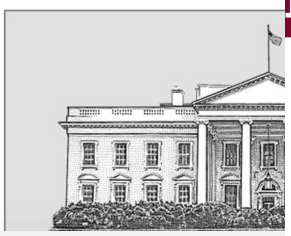
Dyn Says Cyberattack Has Ended, Investigation Continues

Visa Taps Blockchain for Cross-Border Payment Plan

Airbnb New York Possible Leg

DIGITS

Big-Data Startup MongoDB Is Now Valued at $1.6 Billion



InfoWorld FROM IDG

Home > Application Development

STRATEGIC DEVELOPER
By Andrew C. Oliver | Follow

Why MongoDB is worth $1.2 billion

If document database startup MongoDB is looking to thank someone for its hefty valuation, Larry Ellison should be first in line

Design Goals

- Scale horizontally over commodity systems
- Incorporate what works for RDBMSs
  - Rich data models, ad-hoc queries, full indexes
- Move away from what doesn't scale easily
  - Multi-row transactions, complex joins
- Use idomatic development APIs
- Match agile development and deployment workflows

To scale horizontally (or scale out/in) means to add more nodes to (or remove nodes from) a system, such as adding a new computer to a distributed software application. An example might involve scaling out from one Web server system to three.

Key Features

- Data stored as documents (JSON)
  - Dynamic-schema
- Full CRUD support (Create, Read, Update, Delete)
  - Ad-hoc queries: Equality, RegEx, Ranges, Geospatial
  - Atomic in-place updates
- Full secondary indexes
  - Unique, sparse, TTL
- Replication – redundancy, failover
- Sharding – partitioning for read/write scalability

Key Features

- All indexes in MongoDB are B-Tree indexes
- Index Types:
  - Single field index
  - Compound Index: more than one field in the collection
  - Multikey index: index on array fields
  - Geospatial index and queries.
  - Text index: Index
  - TTL index: (Time to live) index will contain entities for a limited time.
  - Unique index: the entry in the field has to b unique.
  - Sparse index: stores an index entry only for entities with the given field.

MongoDB Drivers and Shell

Drivers

Drivers for most popular programming languages and frameworks

Shell

Command-line shell for interacting directly with database

```
> db.collection.insert({product:"MongoDB",
type:"Document Database"})
>
> db.collection.findOne()
{
        "_id"        : ObjectId("5106c1c2fc629bfe52792e86"),
        "product"    : "MongoDB"
        "type"       : "Document Database"
}
```

Getting Started with Mongo

Installation

- Install Mongo from: http://www.mongodb.org/downloads
  - Extract the files
  - Create a data directory for Mongo to use
- Open your mongodb/bin directory and run the binary file (name depends on the architecture) to start the database server.
- To establish a connection to the server, open another command prompt window and go to the same directory, entering in mongo.exe or mongo for macs and Linuxes.
- This engages the mongodb shell—it's that easy!

MongoDB Design Model

Database	Database
Table	Collection
Row	Document

Mongo Data Model

- Document-Based (max 16 MB)
- Documents are in BSON format, consisting of field-value pairs
- Each document stored in a collection
- Collections
  - Have index set in common
  - Like tables of relational db's.
  - Documents do not have to have uniform structure

JSON

- "JavaScript Object Notation"
- Easy for humans to write/read, easy for computers to parse/generate
- Objects can be nested
- Built on
  - name/value pairs
  - Ordered list of values

BSON

- "Binary JSON"
- Binary-encoded serialization of JSON-like docs
- Also allows "referencing"
- Embedded structure reduces need for joins
- Goals
  - Lightweight
  - Traversable
  - Efficient (decoding and encoding)

BSON Example

```
{
"_id" : "37010",
"city" :     "ADAMS",
"pop" :     2660,
"state" :   "TN",
"councilman" : {
                name: "John Smith",
                address: "13 Scenic Way"
            }
}
```

BSON Types

Type	Number
Double	1
String	2
Object	3
Array	4
Binary data	5
Object id	7
Boolean	8
Date	9
Null	10
Regular Expression	11
JavaScript	13
Symbol	14
JavaScript (with scope)	15
32-bit integer	16
Timestamp	17
64-bit integer	18
Min key	255
Max key	127

The number can be used with the $type operator to query by type!

http://docs.mongodb.org/manual/reference/bson-types/

The _id Field

- By default, each document contains an _id field. This field has a number of special characteristics:
  - Value serves as primary key for collection.
  - Value is unique, immutable, and may be any non-array type.
  - Default data type is `ObjectId`, which is "small, likely unique, fast to generate, and ordered."
  - Sorting on an `ObjectId` value is roughly equivalent to sorting on creation time.

MongoDB vs. Relational Databases

RDBMS		MongoDB
Database	➜	Database
Table	➜	Collection
Row	➜	Document
Index	➜	Index
Join	➜	Embedded Document
Foreign Key	➜	Reference

mongoDB vs. SQL

MongoDB	SQL
Document	Tuple
Collection	Table/View
PK: _id Field	PK: Any Attribute(s)
Uniformity not Required	Uniform Relation Schema
Index	Index
Embedded Structure	Joins
Shard	Partition

Document Oriented, Dynamic Schema

Relational

Person:

Pers_ID	Surname	First_Name	City
0	Miller	Paul	London
1	Ortega	Alvaro	Valencia
2	Huber	Urs	Zurich
3	Blanc	Gaston	Paris
4	Bertolini	Fabrizio	Rom

— no relation

Car:

Car_ID	Model	Year	Value	Pers_ID
101	Bentley	1973	100000	0
102	Rolls Royce	1965	330000	0
103	Peugeot	1993	500	3
104	Ferrari	2005	150000	4
105	Renault	1998	2000	3
106	Renault	2001	7000	3
107	Smart	1999	2000	2

MongoDB

```
{
  first_name: 'Paul',
  surname: 'Miller'
  city: 'London',
  location: [45.123,47.232],
  cars: [
    { model: 'Bentley',
      year: 1973,
      value: 100000, … },
    { model: 'Rolls Royce',
      year: 1965,
      value: 330000, … }
  ]
}
```

MongoDB Marketing Spiel

- MongoDB (from "humongous") is a scalable, high-performance, open source, document-oriented database.
  - Fast querying & In-place updates
  - Full Secondary Index Support
  - Replication & High Availability
  - Auto-Sharding
- Currently used in a number of different applications
  - Craigslist, ebay, New York Times, Shutterfly, Chicago Tribune, Github, Disney…

CRUD:

Create, Read, Update, Delete

CRUD: Using the Shell

- To check which db you're using → db
- Show all databases → show dbs
- Switch db's/make a new one → use <name>
- See what collections exist → show collections

- Note: db's are not actually created until you insert data!

CRUD: Using the Shell (cont.)

- To insert documents into a collection/make a new collection:

- db.<collection>.insert(<document>)
- <=>
- INSERT INTO <table>
- VALUES(<attributevalues>);

CRUD: Inserting Data

- Insert one document
- db.<collection>.insert({<field>:<value>})
- Inserting a document with a field name new to the collection is inherently supported by the BSON model.
- To insert multiple documents, use an array.

CRUD: Querying

- Done on collections.
- Get all docs: db.<collection>.find()
  - Returns a cursor, which is iterated over shell to display first 20 results.
  - Add $limit(<number>) to limit results
  - SELECT * FROM <table>;
- Get one doc: db.<collection>.findOne()

CRUD: Querying

To match a specific value:

```
db.<collection>.find({<field>:<value>})
"AND"
db.<collection>.find({<field1>:<value1>, <field2>:<value2>
})
```

```
SELECT *
FROM <table>
WHERE <field1> = <value1> AND <field2> = <value2>;
```

CRUD: Querying

```
OR
db.<collection>.find({ $or: [
<field>:<value1>
<field>:<value2>        ]
})
```

```
SELECT *
FROM <table>
WHERE <field> = <value1> OR <field> = <value2>;
```

Checking for multiple values of same field

```
db.<collection>.find({<field>: {$in [<value>, <value>]}})
```


CRUD: Querying

Excluding document fields

```
db.<collection>.find({<field1>:<value>}, {<field2>: 0})
```

```
SELECT field1
FROM <table>;
```

Including document fields

```
db.<collection>.find({<field>:<value>}, {<field2>: 1})
```

Find documents with or w/o field

```
db.<collection>.find({<field>: { $exists: true}})
```

CRUD: Updating

```
db.<collection>.update(
{<field1>:<value1>},      //all docs in which field = value
{$set: {<field2>:<value2>}},          //set field to value
{multi:true} )      //update multiple docs
```

bulk.find.upsert(): if true, creates a new doc when none matches search criteria.

```
UPDATE <table>
SET <field2> = <value2>
WHERE <field1> = <value1>;
```


CRUD: Updating

To remove a field

```
db.<collection>.update({<field>:<value>},
            { $unset: { <field>: 1}})
```

Replace all field-value pairs

```
db.<collection>.update({<field>:<value>},
            { <field>:<value>, <field>:<value>})
```

*NOTE: This overwrites ALL the contents of a document, even removing fields.

CRUD: Removal

Remove all records where field = value

```
db.<collection>.remove({<field>:<value>})
```

```
DELETE FROM <table>
WHERE <field> = <value>;
```

As above, but only remove first document

```
db.<collection>.remove({<field>:<value>}, true)
```


CRUD: Isolation

- By default, all writes are atomic only on the level of a single document.
- This means that, by default, all writes can be interleaved with other operations.
- You can isolate writes on an unsharded collection by adding $isolated:1 in the query area:
  - db.<collection>.remove({<field>:<value>, $isolated: 1})

MongoDB Example II

```
db.users.insertMany(
 [
   {
    _id: 1,
    name: "sue",
    age: 19,
    type: 1,
    status: "P",
    favorites: { artist: "Picasso", food: "pizza" },
    finished: [ 17, 3 ],
    badges: [ "blue", "black" ],
    points: [
      { points: 85, bonus: 20 },
      { points: 85, bonus: 10 }
    ]
   },
   {
    _id: 2,
    name: "bob",
    age: 42,
    type: 1,
    status: "A",
    favorites: { artist: "Miro", food: "meringue" },
    finished: [ 11, 25 ],
    badges: [ "green" ],
    points: [
      { points: 85, bonus: 20 },
      { points: 64, bonus: 12 }
    ]
   },
   {
    _id: 3,
    name: "ahn",
    age: 22,
    type: 2,
    status: "A",
    favorites: { artist: "Cassatt", food: "cake" },
    finished: [ 6 ],
    badges: [ "blue", "red" ],
    points: [
      { points: 81, bonus: 8 },
      { points: 55, bonus: 20 }
    ]
   },
   {
    _id: 4,
    name: "xi",
    age: 34,
    type: 2,
    status: "D",
    favorites: { artist: "Chagall", food: "chocolate" },
    finished: [ 5, 11 ],
    badges: [ "red", "black" ],
    points: [
      { points: 53, bonus: 15 },
      { points: 51, bonus: 15 }
    ]
   },
)
```

Insert

- db.collection.insertOne()
- db.collection.insertMany()
- db.collection.insert()
- Example

```
db.users.insertMany(
  [
    { name: "bob", age: 42, status: "A", },
    { name: "ahn", age: 22, status: "A", },
    { name: "xi", age: 34, status: "D", }
  ]
)
```


Update

- db.collection.updateOne()
- db.collection.updateMany()
- db.collection.replaceOne()
- db.collection.update()
- Example

```
db.users.updateOne(
  { "favorites.artist": "Picasso" },
  {
    $set: { "favorites.food": "pie", type: 3 },
    $currentDate: { lastModified: true }
  }
)
```

Return All Fields in Matching Documents

- Retrieve from the users collection all documents where the status equals "A"
  - db.users.find({ status: "A" })

Return the Specified Fields and the _id Field Only

- A projection can explicitly include several fields
  - Return all documents that match the query
    - db.users.find({ status: "A" }, { name: 1, status: 1 })

- This will result in the following:

  { "_id" : 2, "name" : "bob", "status" : "A" }
  { "_id" : 3, "name" : "ahn", "status" : "A" }

Return the Specified Fields

- Remove the _id field from the results by specifying its exclusion in the projection
  - db.users.find({ status: "A" }, { name: 1, status: 1, _id: 0 })
- This will result in the following:

```
{ "name" : "bob", "status" : "A" }
{ "name" : "ahn", "status" : "A" }
{ "name" : "abc", "status" : "A" }
```


Return All But the Excluded Field

- Use a projection to exclude specific fields
  - db.users.find({ status: "A" }, { favorites: 0, points: 0 })
- Returns

```
{
  "_id" : 2,
  "name" : "bob",
  "age" : 42,
  "type" : 1,
  "status" : "A",
  "finished" : [ 11, 25 ],
  "badges" : [ "green" ]
}
...
```

Return Specific Fields in Embedded Documents

- Use the dot notation to return specific fields in an embedded document
  - db.users.find({ status: "A" }, { name: 1, status: 1, "favorites.food": 1 })
- Returns the following fields inside the favorites document

```
{ "_id" : 2, "name" : "bob", "status" : "A", "favorites" : { "food" : "meringue" } }
{ "_id" : 3, "name" : "ahn", "status" : "A", "favorites" : { "food" : "cake" } }
```


Suppress Specific Fields in Embedded Documents

- Exclude the food field inside the favorites document

```
db.users.find(
  { status: "A" },
  { "favorites.food": 0 }
)
```

- Returns

```
{
  "_id" : 2,
  "name" : "bob",
  "age" : 42,
  "type" : 1,
  "status" : "A",
  "favorites" : { "artist" : "Miro" },
  "finished" : [ 11, 25 ],
  "badges" : [ "green" ],
  "points" : [ { "points" : 85, "bonus" : 20 }, { "points" : 64, "bonus" : 12 } ]
}
...
```

SQL Schema Statements	MongoDB Schema Statements

`CREATE TABLE users (`
`  id MEDIUMINT NOT NULL`
`  AUTO_INCREMENT,`
`  user_id Varchar(30),`
`  age Number,`
`  status char(1),`
`  PRIMARY KEY (id)`
`)`
	Implicitly created on first insert() operation. The primary key _id is automatically added if _id field is not specified.
	`db.users.insert( {`
	`  user_id: "abc123",`
	`  age: 55,`
	`  status: "A"`
	`  } )`
	However, you can also explicitly create a collection:
	`db.createCollection("users")`

`ALTER TABLE users`
`ADD join_date DATETIME`
	Collections do not describe or enforce the structure of its documents; i.e. there is no structural alteration at the collection level.
	However, at the document level, update() operations can add fields to existing documents using the $set operator.
	`db.users.update(`
	`  { },`
	`  { $set: { join_date: new Date() } },`
	`  { multi: true }`
	`)`

`ALTER TABLE users`
`DROP COLUMN join_date`
	Collections do not describe or enforce the structure of its documents; i.e. there is no structural alteration at the collection level.
	However, at the document level, update() operations can remove fields from documents using the $unset operator.
	`db.users.update(`
	`  { },`
	`  { $unset: { join_date: "" } },`
	`  { multi: true }`
	`)`

`CREATE INDEX idx_user_id_asc` `ON users(user_id)`	`db.users.createIndex( { user_id: 1 } )`

`CREATE INDEX`
`  idx_user_id_asc_age_des`
`c`
`ON users(user_id, age DESC)`
	`db.users.createIndex( { user_id: 1, age: -1 } )`

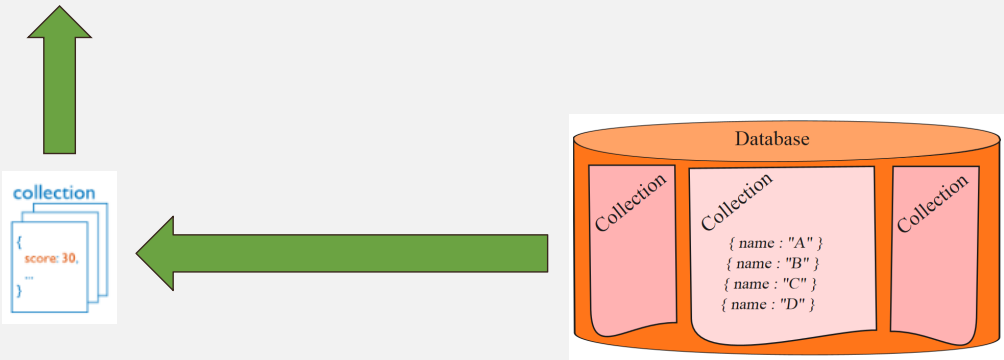
`DROP TABLE users`	`db.users.drop()`

Index in MongoDB

Before Index

- What does database normally do when we query?
  - MongoDB must scan every document.
  - Inefficient because process large volume of data

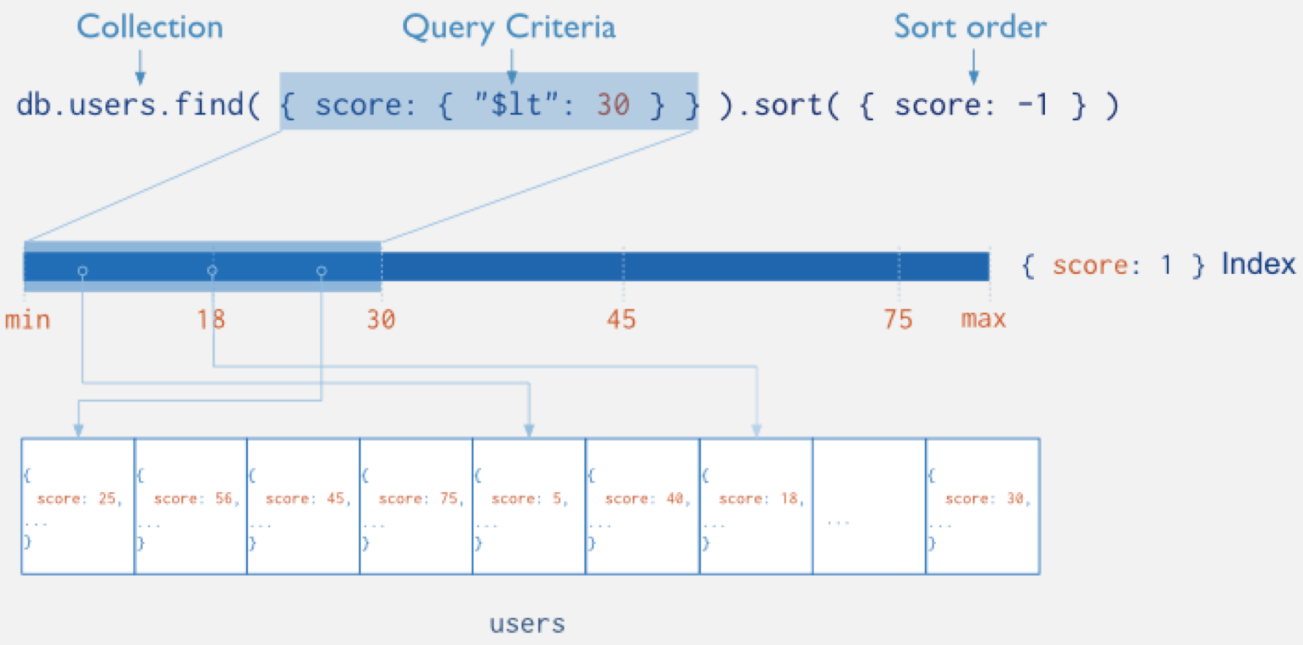
db.users.find({ score: { “$lt” : 30} })



Index in MongoDB: Operations

- Creation index
  - db.users.ensureIndex({ score: 1 })
  - db.people.createIndex({ zipcode: 1}, {background: **true**})
- Show existing indexes
  - db.users.getIndexes()
- Drop index
  - db.users.dropIndex({score: 1})
- Explain—Explain
  - db.users.find().explain()
  - Returns a document that describes the process and indexes
- Hint
  - db.users.find().hint({score: 1})
  - Overide MongoDB's default index selection

Index in MongoDB: Operations



Index in MongoDB

- Types
  - **Single Field Indexes**
  - Compound Field Indexes
  - Multikey Indexes
- Single Field Indexes
  - db.users.ensureIndex({ score: 1 })

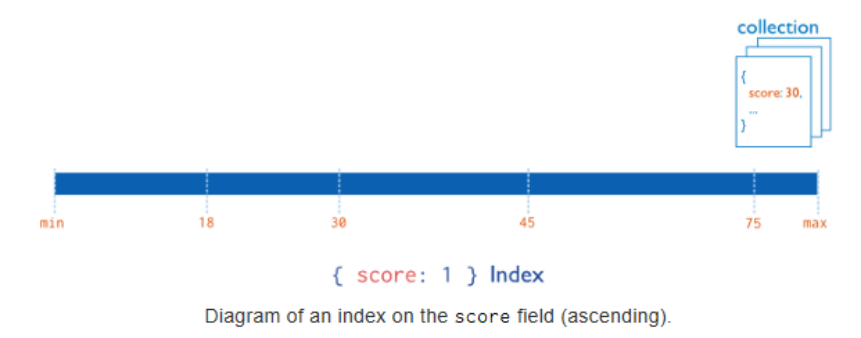


Diagram of an index on the score field (ascending).

Index in MongoDB

- Types
  - Single Field Indexes
  - **Compound Field Indexes**
  - Multikey Indexes
- Compound Field Indexes
  - db.users.ensureIndex({ userid:1, score:-1 })

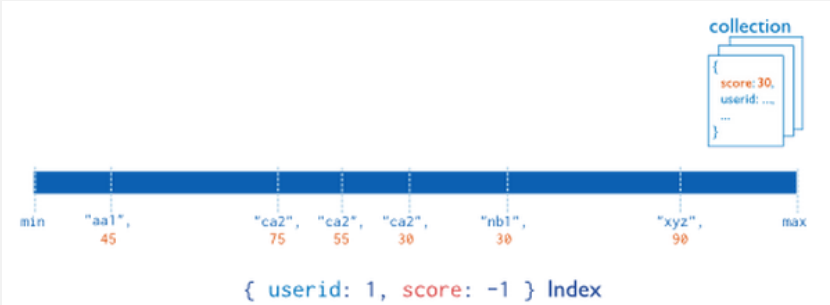


Diagram of a compound index on the userid field (ascending) and the score field (descending). The index sorts first by the userid field and then by the score field.

Index in MongoDB

- Types
  - Single Field Indexes
  - Compound Field Indexes
  - **Multikey Indexes**
- Multikey Indexes
  - db.users.ensureIndex({ addr.zip:1})

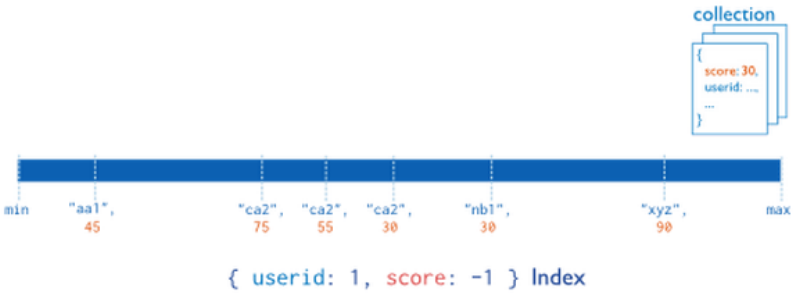


Diagram of a compound index on the `userid` field (ascending) and the `score` field (descending). The index sorts first by the `userid` field and then by the `score` field.

Other Indexes in MongoDB

- Geospatial Index
- Text Indexes
- Hashed Indexes

Mongo Example I

Documents

```
> var new_entry = {
  firstname: “John”,
  lastname: “Smith”,
  age: 25,
  address: {
    street: “21 2nd Street”,
    city: “New York”,
    state: “NY”,
    zipcode: 10021
  }
}
> db.addressBook.save(new_entry)
```

Querying

```
> db.addressBook.find()
{
  _id: ObjectId(“4c4ba5c0672c685e5e8aabf3”),
  firstname: “John”,
  lastname: “Smith”,
  age: 25,
  address: {
    street: “21 2nd Street”, city: “New York”,
    state: “NY”, zipcode: 10021
  }
}
// _id is unique but can be anything you like
```


Indexes

```
// create an ascending index on “state”
> db.addressBook.ensureIndex({state:1})

> db.addressBook.find({state:”NY”})
{
  _id: ObjectId(“4c4ba5c0672c685e5e8aabf3”),
  firstname: “John”,
  …
}

> db.addressBook.find({state:”NY”, zip: 10021})
```

Queries

```
// Query Operators:
// $all, $exists, $mod, $ne, $in, $nin, $nor, $or,
// $size, $type, $lt, $lte, $gt, $gte

// find contacts with any age
> db.addressBook.find({age: {$exists: true}})

// find entries matching a regular expression
> db.addressBook.find( {lastname: /^smi*/i } )

// count entries with “John”
> db.addressBook.find( {firstname: ‘John’} ).count()
```


Updates

```
// Update operators
// $set, $unset, $inc, $push, $pushAll, $pull,
// $pullAll, $bit

> var new_phonenumber = {
  type: “mobile”,
  number: “646-555-4567”
}

> db.addressBook.update({ _id: “...” }, {
  $push: {phonenumbers: new_phonenumber}
});
```

Nested Documents

```
{
  _id: ObjectId("4c4ba5c0672c685e5e8aabf3"),
  firstname: “John”, lastname: “Smith”,
  age: 25,
  address: {
    street: “21 2nd Street”, city: “New York”,
    state: “NY”, zipcode: 10021
  }
  phonenumbers : [ {
    type: “mobile”, number: “646-555-4567”
  } ]
}
```


Secondary Indexes

```
// Index nested documents
> db.addressBook.ensureIndex({“phonenumbers.type”:1})

// Geospatial indexes, 2d or 2dsphere
> db.addressBook.ensureIndex({location: “2d”})
> db.addressBook.find({location: {$near: [22,42]}})

// Unique and Sparse indexes
> db.addressBook.ensureIndex({field:1}, {unique:true})
> db.addressBook.ensureIndex({field:1}, {sparse:true})
```

Additional Features

- Geospatial queries
  - Simple 2D plane
  - Or accounting for the surface of the earth (ellipsoid)
- Full Text Search
- Aggregation Framework
  - Similar to SQL GROUP BY operator
- Javascript MapReduce
  - Complex aggregation tasks

Mongo Example II

Another Sample Document

```
d={
      _id :
ObjectId(“4c4ba5c0672c685e5e8aabf3”),
      author : “Kevin”,
      date : new Date(“February 2, 2012”),
      text : “About MongoDB...”,
      birthyear:  1980,
      tags : [ "tech", "databases" ]
      }

> db.posts.insert(d)
```


Find

- db.posts.find()
  - returns entire collection in posts
- db.posts.find({ “author” : “Kevin” , “birthyear” : 1980})

```
{
        _id :
ObjectId("4c4ba5c0672c685e5e8aabf3"),
        author : "Kevin",
        date : Date(“February 2, 2012”),
        birthyear:  1980,
        text : "About MongoDB...",
        tags : [ "tech", "databases" ]
  }
```

Specifying Which Keys to Return

- db.mydoc.find({}, { "name" , { "contribs" })

  _id: 1,
  name: { first: "John", last: "Backus" },
  contribs: ["Fortran", "ALGOL", "Backus-Naur Form", "FP"]
  }

- db.mydoc.find({}, { "_id" :0, { "name" :1})

  name: { first: "John", last: "Backus" }
  }

Ranges, Negation, OR-clauses

- Comparison operators: $lt, $lte, $gt, $gte
  - db.posts.find({ "birthyear" : { "$gte" : 1970, "$lte" : 1990}})
- Negation: $ne
  - db.posts.find({ "birthyear" : { "$ne" : 1982}})
- Or queries: $in (single key), $or (different keys)
  - db.posts.find({ "birthyear" : { "$in" : [1982, 1985]}})
  - db.posts.find({ "$or" : [{ "birthyear" : 1982}, { "name" : "John" }]})

Arrays

- db.posts.find({ “tags” : “tech” })
  - Print complete information about posts which are tagged “tech”
- db.posts.find({ “tags” : {$all: [“tech” , “databases”]},{ “author” :1, “tags” :1})
  - Print author and tags of posts which are tagged with both “tech” and “databases” (among other things)
  - Contrast this with:
  - db.posts.find({ “tags” : [“databases” , “tech”]})

Querying Embedded Documents

- db.people.find({ “name.first” : “John” })
  - Finds all people with first name John
- db.people.find({ “name.first” : “John” , “name.last” : “Smith”)
  - Finds all people with first name John and last name Smith.
  - Contrast with (order is now important):
  - db.people.find({ “name” : { “first” : “John” , “last” : “Smith” }})

Limits, Skips, Sort, Count

- db.posts.find().limit(3)
  - Limits the number of results to 3
- db.posts.find().skip(3)
  - Skips the first three results and returns the rest
- db.posts.find().sort({ “author” :1, “title” :-1})
  - Sorts by author ascending (1) and title descending (-1)
- db.people.find(…).count()
  - Counts the number of documents in the people collection matching the find(…)

Revisiting Sample Document

```
mydoc =   {
     _id: 1,
    name: { first: “John”, last: “Backus” },
    birthyear: 1924,
    contribs: [ “Fortran”, “ALGOL”, “Backus-Naur Form”, “FP” ],
    awards: [ { award_id: “NMS001”,
                       year: 1975 },
                    { award_id: “TA99”,
                       year: 1977} ]
                         }
> db.people.insert(mydoc)
```

Also assume…

```
award1=
 {_id: “NMS001”,
  title:  “National Medal of Science” ,
  by: “National Science Foundation”}
award2=
 {_id: “TA99”,
  title:  “Turing Award”,
  by: “ACM” }
db.awards.insert(award1)
db.awards.insert(award2)
```


“SemiJoins”

- Suppose you want to print people who have won Turing Awards
  - Problem: object id of Turing Award is in collection “awards” ,collection “people” references it.

```
turing= db.awards.findOne({title: “Turing Award”})
db.people.find({"awards.award_id": turing["_id"]})
```

Aggregation

- A framework to provide "group-by" and aggregate functionality without the overhead of map-reduce.
- Conceptually, documents from a collection pass through an aggregation pipeline, which transforms the objects as they pass through (similar to UNIX pipe "|")
- Operators include: $project, $match, $limit, $skip, $sort, $unwind, $group

Unwind

- db.article.aggregate({ $project : { author : 1, tags : 1 }}, { $unwind : "$tags" })

```
{ "result" : [ { "_id" : ObjectId("4e6e4ef557b77501a49233f6"),
                 "author" : "bob",
                 "tags" : "fun" },
               { "_id" : ObjectId("4e6e4ef557b77501a49233f6"),
                 "author" : "bob",
                 "tags" : "good" },
               { "_id" : ObjectId("4e6e4ef557b77501a49233f6"),
                 "author" : "bob",
                 "tags" : "fun" } ],
  "OK" : 1 }
```

$group

- Every group expression must specify an _id field.
- For example, suppose you wanted to print the number of people born in each year

```
> db.people.aggregate( { $group :
        { _id : "$birthyear", birthsPerYear : { $sum : 1}} )
{ "result" : [ { "_id" : 1924, "count" : 1 } ], "ok" : 1 }
```


MongoDB Development

Open Source

- MongoDB source code is on Github
  - https://github.com/mongodb/mongo
- Issue tracking for MongoDB and drivers
  - http://jira.mongodb.org

Summary of MongoDB

- MongoDB is an example of a document-oriented NoSQL solution
- The query language is limited, and oriented around "collection" (relation) at a time processing
  - Joins are done via a query language
- The power of the solution lies in the distributed, parallel nature of query processing
  - Replication and sharding