UMD DATA605 - Big Data Systems

NoSQL Document Stores

MongoDB

CouchDB

Dr. GP Saggese

gsaggese@umd.edu

with thanks to Profs

Alan Sussman (UMD)

Amol Deshpande (UMD)

Nguyen Vo (Utah State U.)

Kathleen Durant (Northeastern U.)

UMD DATA605 - Big Data Systems

NoSQL Document Stores

MongoDB

Couchbase

Key-Value Store vs Document DBs

- **Key-value stores**
  - Basically a map or dictionary
    - E.g., HBase, Redis
  - Typically only look up values by key
    - Sometimes can do search in value field with a pattern
  - Uninterpreted value (e.g., binary blob) associated with a key
  - Typically one namespace for all key-values

- **Document DBs**
  - Collect sets of key-value pairs into *documents*
    - E.g., MongoDB, CouchDB
  - Documents typically represented in JSON, XML, or binary JSON
  - Documents organized into *collections*, similar to tables in relational DBs
    - Large collections can be partitioned and indexed

UMD DATA605 - Big Data Systems

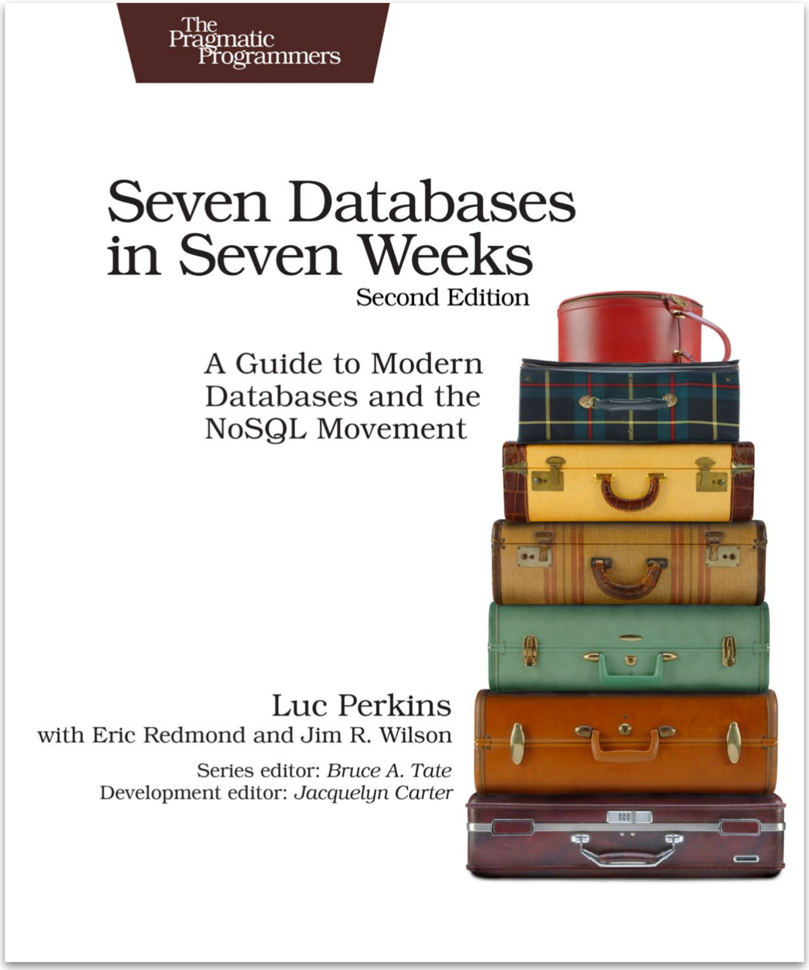
NoSQL Document Stores

MongoDB

Couchbase

Resources

- Concepts in slides
- MongoDB tutorial
- Web
  - https://www.mongodb.com/
  - [Official docs]()
  - [pymongo]()
- Book
  - [Seven Databases in Seven Weeks, 2e]()



MongoDB

- Developed by MongoDB Inc.
  - Founded in 2007
  - Based on DoubleClick experience with large-scale data
  - Mongo comes from “hu-mongo-us”
- One of the most used NoSQL DBs (if not the most used)
- Document-oriented NoSQL database
  - Schema-less
    - No Data Definition Language (DDL)
    - In practice, you can store maps with any keys and values that you choose
    - Application tracks the schema, mapping between documents and their meaning
  - Keys are hashes stored as strings
    - Document Identifiers `_id` created for each document (field name reserved by system)
  - Values use BSON format
    - Based on JSON (B stands for Binary)
- Written in C++
- Supports APIs (drivers) in many languages
  - E.g., JavaScript, Python, Ruby, Java, Scala, C++, ...

MongoDB: Example of Document

- A **document** is a JSON data structure
- Correspond to a row in a relational DB
  - without schema
  - values nested to an arbitrary depth
  - primary key is `_id`

```
{
   "_id" : ObjectId("4d0b6da3bb30773266f39fea"),
   "country" : {
      "$ref" : "countries",
      "$id" : ObjectId("4d0e6074deb8995216a8309e")
   },
   "famous_for" : [
      "beer",
      "food"
   ],
   "last_census" : "Sun Jan 07 2018 00:00:00 GMT -0700 (PDT)",
   "mayor" : {
      "name" : "Ted Wheeler",
      "party" : "D"
   },
   "name" : "Portland",
   "population" : 582000,
   "state" : "OR"
}
```

MongoDB: Functionalities

- Design goals:
  - Performance
  - Scalability
  - Rich data access
- Dynamic schema
  - No DDL (Data Definition Language)
  - Secondary indexes
- Query language via an API
- Several levels of consistency
  - E.g., atomic writes and fully-consistent reads
- No joins nor transactions across multiple documents
  - Makes distributed queries easy and fast
- High availability through replica sets
  - E.g., primary replication with automated failover
- Built-in sharding
  - Horizontal scaling via automated range-based partitioning of data
  - Reads and writes distributed over shards

Relational DBs vs MongoDB: Terms and Concepts

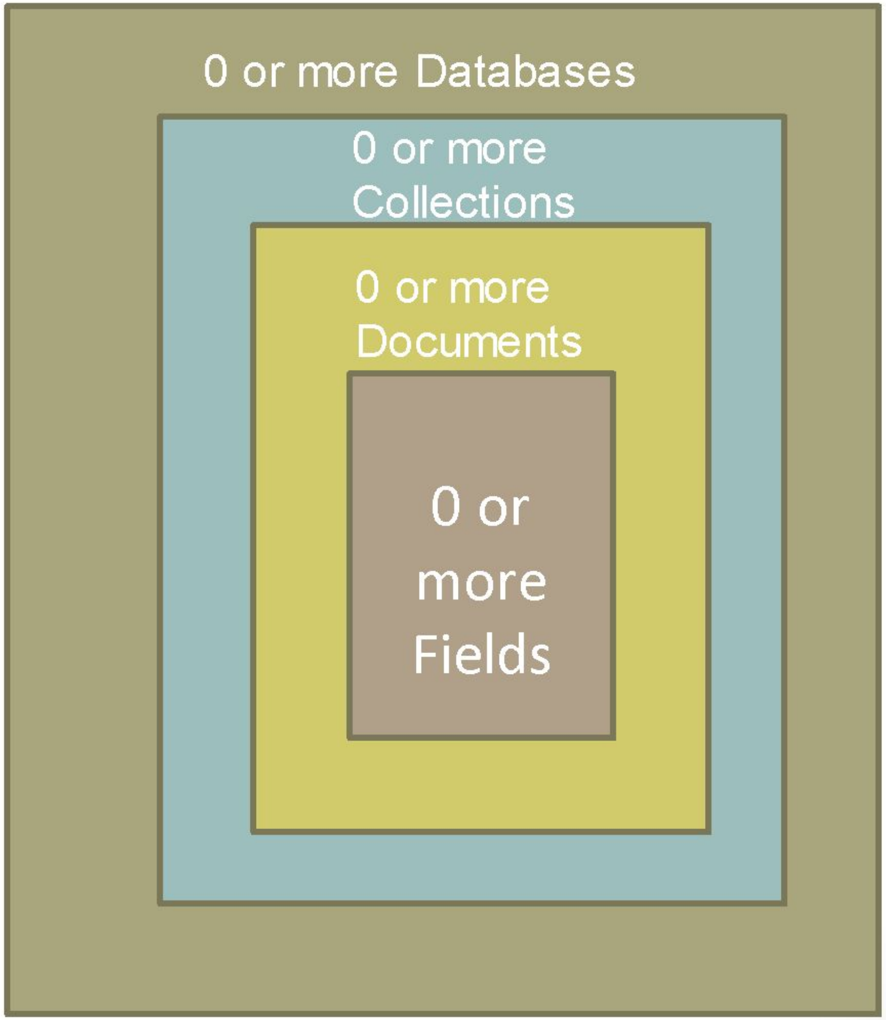
RDBMS Concepts	MongoDB Concepts	Meaning in MongoDB
database	database	Container for collections
relation / table, view	collection	Group of documents
row / instance	document (BSON)	Group of fields
column / attributes	field	A name-value pair
index	index	
primary keys	_*id* field	Always the primary key
foreign key	reference	
table joins	embedded documents and linking	

Relational vs Document DB Workflows

- **Relational DBs**
  - E.g., PostgreSQL
  - Know what you want to store
    - Tabular data
  - Do not know how to use it
    - Static schema allows query flexibility (e.g., joins)
  - Complexity is at insertion time
    - Decide how to represent the data
- **Document DBs**
  - E.g., MongoDB
  - No assumptions on what to store
    - E.g., irregular JSON data
  - Know a bit how to access
    - E.g., it's a nested key-value map
  - Complexity is at access time
    - Get the data from the server
    - Process data on the client side

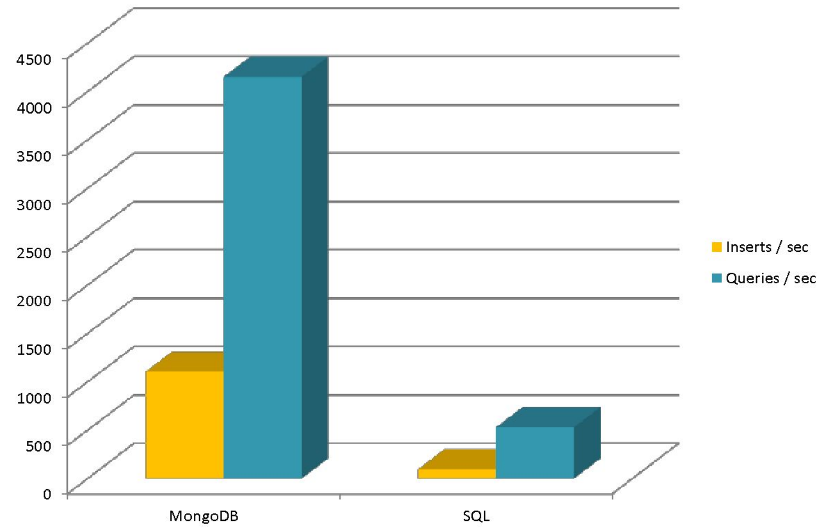
MongoDB: Hierarchical Objects

- An **instance** has:
  - zero or more "databases"
  - same as Postgres
- A **database** has:
  - zero or more "collections"
  - ~ Postgres tables
- A **collection** has:
  - zero or more "documents"
  - ~ Postgres rows
- A **document** has:
  - one or more "fields"
  - It has always the _id
  - ~ Postgres columns



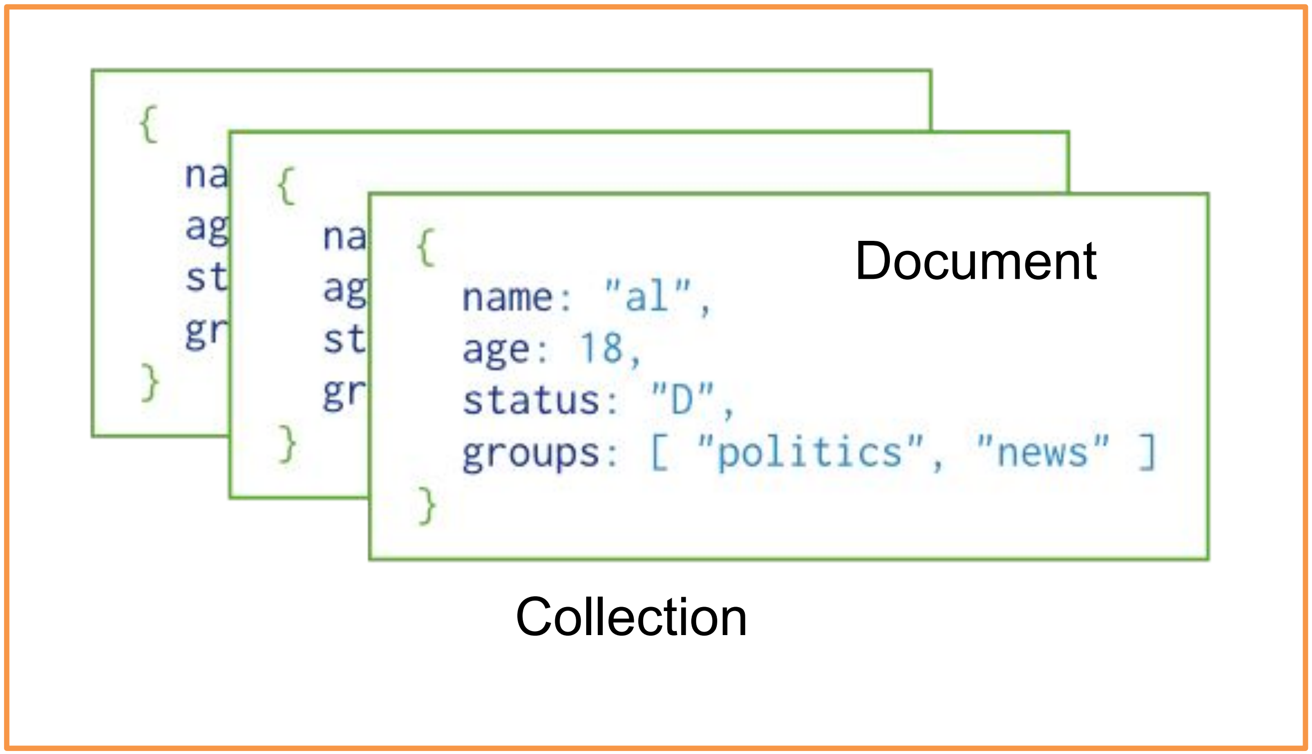
Why Use MongoDB?

- Simple to query
  - Do the work on client side
- It's fast
  - 2-10x faster than Postgres
- Data model / functionalities are applicable to most web applications
  - Semi-structured data
  - Quickly evolving systems
- Easy and fast integration of data
- Not well suited for heavy and complex transactions systems



MongoDB Data Model

A **collection** (~table) includes **documents** (~rows)



From https://www.mongodb.com/docs/manual/core/data-modeling-introduction

MongoDB Data Model

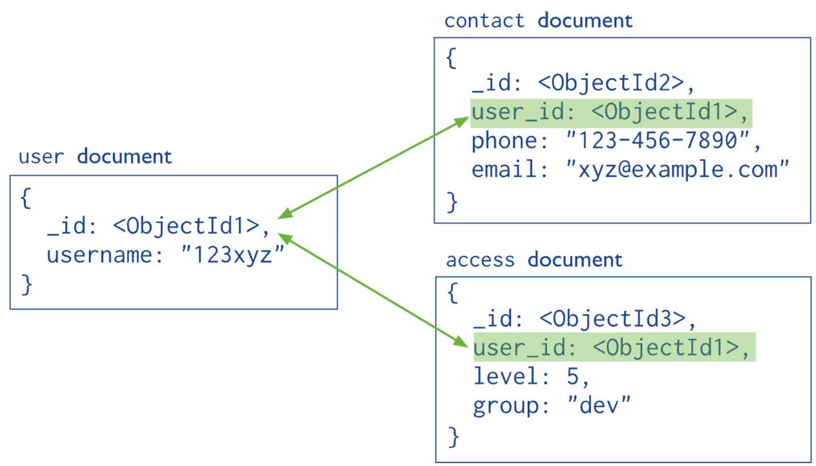
- **Documents** are composed of field and value pairs
  - Field names are strings
  - Each value is any BSON type (native data types, other documents, arrays of documents)
- E.g.,
  - `_id` holds an ObjectId
  - `name` holds a document that contains the fields `first` and `last`
  - `birth` and `death` are of Date type
  - `contribs` holds an array of strings
  - `views` holds a value of the NumberLong type

```
{
   name: "sue",                          ← field: value
   age: 26,                              ← field: value
   status: "A",                          ← field: value
   groups: [ "news", "sports" ]          ← field: value
}
```

```
{
   _id: ObjectId("5099803df3f4948bd2f98391"),
   name: { first: "Alan", last: "Turing" },
   birth: new Date('Jun 23, 1912'),
   death: new Date('Jun 07, 1954'),
   contribs: [ "Turing machine", "Turing test", "Turingery" ],
   views : NumberLong(1250000)
}
```

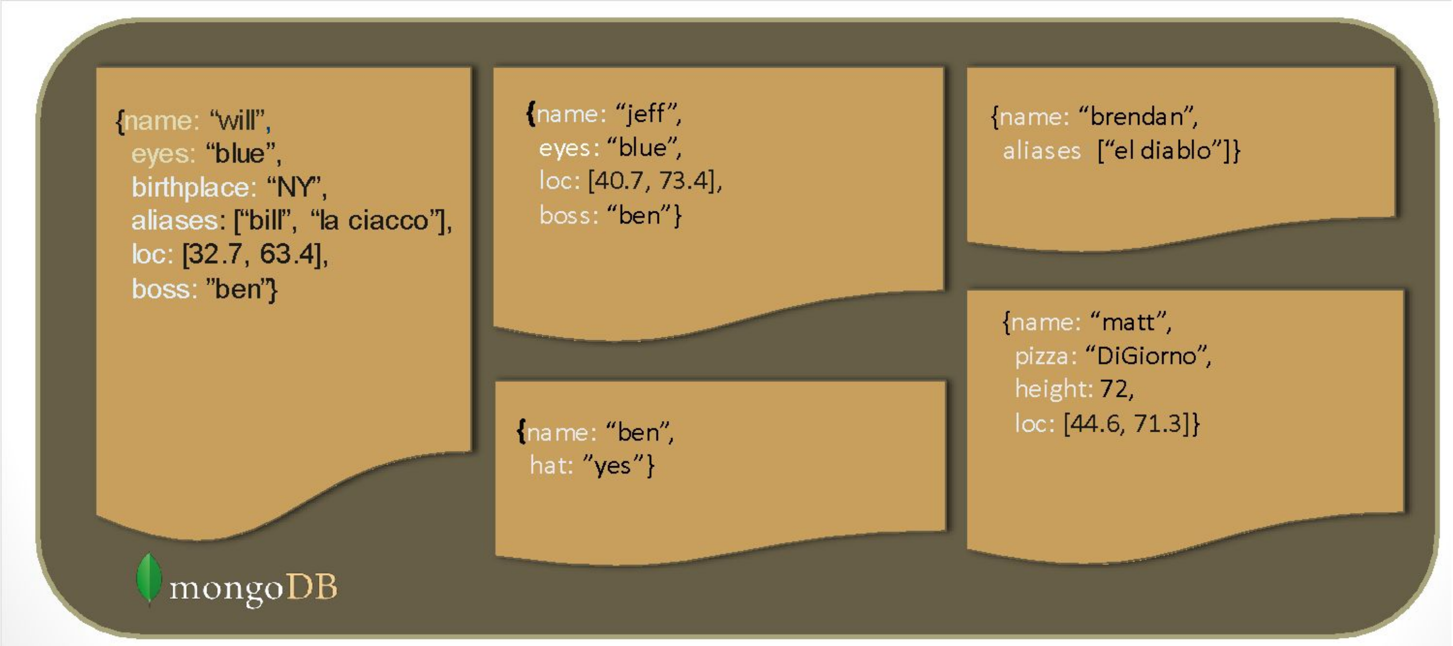
MongoDB Data Model

- Documents can be nested
- **Denormalized data models**
  - Allow to retrieve and manipulate data in a single operation
  - Store multiple related pieces of information in the same record
- **Normalized data models**
  - Eliminate duplication
  - Represent many-to-many relationships



Schema Free

- MongoDB does not need any pre-defined data schema
- Every document in a collection can have different fields and values
  - No need for NULL data fields



JSON Format

- JavaScript Object Notation
- Data is in name / value pairs
- A name / value pair consists of a field name followed by a colon, followed by a value

```
“name”: “R2-D2”
```

- Data is separated by commas

```
“name”: “R2-D2”, race : “Droid”
```

- Curly braces {} hold objects

```
{“name”: “R2-D2”, race : “Droid”, affiliation: “rebels”}
```

- An array is stored in brackets []

```
[{“name”: “R2-D2”, race: “Droid”, affiliation: “rebels”},
 {“name”: “Yoda”, affiliation: “rebels”}]
```

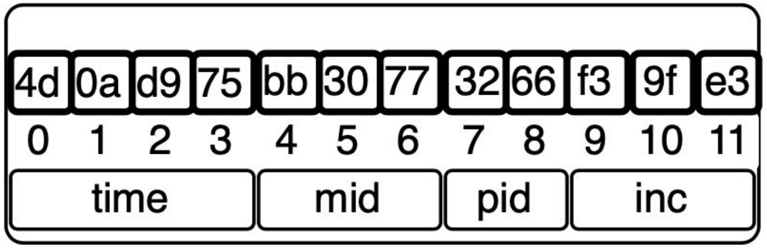
- Support embedding of nested objects within other objects, or just references

BSON Format

- Binary-encoded serialization of JSON-like documents
  - https://bsonspec.org
- Zero or more key/value pairs are stored as a single entity
- Each entry consists of:
  - a field name
  - a data type
  - a value
- Large elements in a BSON document are prefixed with a length field to facilitate scanning
- MongoDB understands the internals of BSON objects, even nested ones
  - Can build indexes and match objects against query expressions for BSON keys

ObjectID

- Each JSON data contains an `_id` field of type ObjectId
  - Like a `SERIAL` incrementing a numeric primary key in PostgreSQL
- An ObjectId is always 12 bytes, composed of:
  - a timestamp
  - client machine ID
  - client process ID
  - a 3-byte auto-incremented counter
- Each Mongo process can handle its own ID generation without colliding
  - Mongo has a distributed nature
- Details [here]



MongoDB Features

- Document-oriented NoSQL store
- Rich querying
  - Full index support
- Fast in-place updates
- Agile and scalable
  - Replication and high availability
  - Auto-Sharding
  - Map-reduce functionality
- Scale horizontally over commodity hardware
  - Lots of relatively inexpensive servers

MongoDB vs Relational DBs

- Keep the functionality that works well in RDBMSs
  - Ad-hoc queries
  - Fully featured indexes
  - Secondary indexes
- Do not offer RDBMS functionalities that don't distribute
  - Long running multi-row transactions
  - Joins
  - Both artifacts of the relational data model (row x column)

Indexes

- Primary index automatically created on the `_id` field
  - B+ tree indexes
- Users can create secondary indexes to:
  - Improve query performance
  - Enforce unique values for a particular field
- Support single field index and compound index
  - Order of the fields in a compound index matters (like SQL)
  - If you index a field that holds an array value, MongoDB creates separate index entries for every element of the array
- Sparse property of an index
  - The index contains only entries for documents that have the indexed field
  - Ignore records that do not have the field defined
- If an index is unique and sparse
  - Reject records that have a duplicate key value
  - Allow records that do not have the indexed field defined
- Details at https://www.mongodb.com/docs/manual/indexes/

CRUD Operations

- CRUD = Create, Read, Update, Delete

- **Create**

```
db.collection.insert(<document>)
db.collection.save(<document>)
db.collection.update(<query>, <update>, {upsert: true})
```

- **Read**

```
db.collection.find(<query>, <projection>)
db.collection.findOne(<query>, <projection>)
```

- **Update**

```
db.collection.update(<query>, <update>, <options>)
```

- **Delete**

```
db.collection.remove(<query>, <justOne>)
```

Details at https://www.mongodb.com/docs/manual/crud/

Create Operations

- `db.collection` specifies the collection (like a 'table') to store the document

  ```
  db.collection_name.insert(<document>)
  ```

  - Omit the _id field to have MongoDB generate a unique key

  ```
  db.parts.insert({type: "screwdriver", quantity: 15})
  db.parts.insert({_id: 10, type: "hammer", quantity: 1})
  ```

- Update 1 or more records in a collection satisfying *query*

```
db.collection_name.update(<query>, <update>, {upsert: true})
```

- Update an existing record or creates a new record

```
db.collection_name.save(<document>)
```

Read Operations

- find provides functionality similar to the SQL SELECT command, with
  - <query> = where condition
  - <projection> = fields in result set

```
db.collection.find(<query>, <projection>).cursor
```

- `db.parts.find({parts: “hammer”}).limit(5)`
  - Has cursors to handle a result set
  - Can modify the query to impose limits, skips, and sort orders
  - Can specify to return the ‘top’ number of records from the result set
- `db.collection.findOne(<query>, <projection>)`

More Query Examples

SQL	Mongo
`SELECT * FROM users WHERE age>33`	`db.users.find({age:{$gt:33}})`
`SELECT * FROM users WHERE age!=33`	`db.users.find({age:{$ne:33}})`
`SELECT * FROM users WHERE name LIKE "%Joe%"`	`db.users.find({name:/Joe/})`
`SELECT * FROM users WHERE a=1 and b='q’`	`db.users.find({a:1,b:'q’})`
`SELECT * FROM users WHERE a=1 or b=2`	`db.users.find({$or: [{a: 1}, {b: 2}]})`
`SELECT * FROM foo WHERE name=‘bob’ and (a=1 or b=2 )`	`db.foo.find({name: "bob", $or : [{a: 1}, {b: 2}]})`
`SELECT * FROM users WHERE age>33 AND age<=40`	`db.users.find({'age':{$gt:33,$lte:40}})`

Query Operators

Command	Description
$regex	Match by any PCRE-compliant regular expression string (or just use the // delimiters as shown earlier)
$ne	Not equal to
$lt	Less than
$lte	Less than or equal to
$gt	Greater than
$gte	Greater than or equal to
$exists	Check for the existence of a field
$all	Match all elements in an array
$in	Match any elements in an array
$nin	Does not match any elements in an array
$elemMatch	Match all fields in an array of nested documents
$or	or
$nor	Not or
$size	Match array of given size
$mod	Modulus
$type	Match if field is a given datatype
$not	Negate the given operator check

Update Operations

- `db.collection_name.insert(<document>)`
  - Omit the _id field to have MongoDB generate a unique key

```
db.parts.insert( {{type: “screwdriver”, quantity: 15 } )
db.parts.insert({_id: 10, type: “hammer”, quantity: 1 })
```

- `db.collection_name.save(<document>)`
  - Updates an existing record or creates a new record
- `db.collection_name.update(<query>, <update>, {upsert: true})`
  - Will update 1 or more records in a collection satisfying query
- db.collection_name.findAndModify(<query>, <sort>, <update>,<new>, <fields>, <upsert>)
  - Modify existing record(s) – retrieve old or new version of the record

Delete Operations

- `db.collection_name.remove(<query>, <justone>)`
  - Delete all records from a collection or matching a criterion
  - <justone> specifies to delete only 1 record matching the criterion
- Remove all parts starting with h

  ```
  db.parts.remove(type: /^h/ })
  ```
- Delete all documents in the parts collections

  ```
  db.parts.remove()
  ```

MongoDB Tutorial

Tutorial is at GitHub

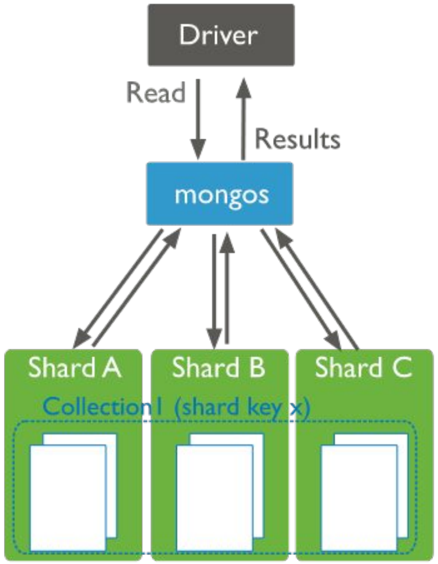
The instructions are here

> cd $GIT_REPO/tutorials/tutorial_mongodb

> vi tutorial_mongo.md

MongoDB Processes and Configuration

- ***mongod***: database instance (i.e., the server process)
- ***mongosh***: an interactive shell (i.e., a client)
  - Fully functional JavaScript environment for use with a MongoDB
- ***mongos***: database router
  - Process all requests
  - Decide how many and which *mongod*s should receive the query (sharding / partitioning)
  - Collate the results
  - Send result back to the client
- You should have:
  - One *mongos* for the whole system no matter how many *mongod*s you have; or
  - One local *mongos* for every client if you wanted to minimize network latency



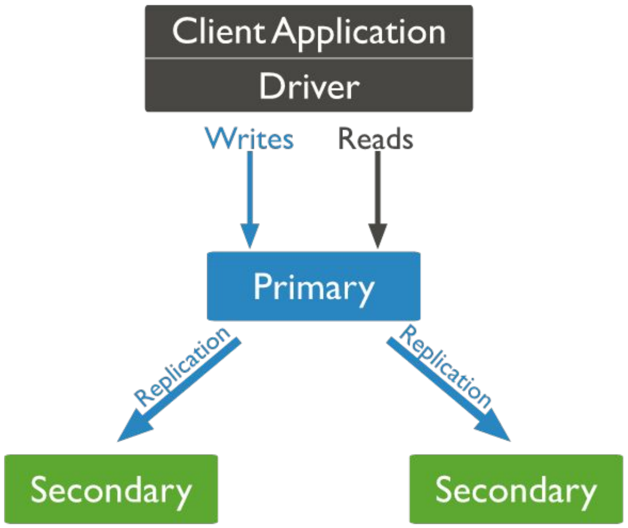
MapReduce Functionality

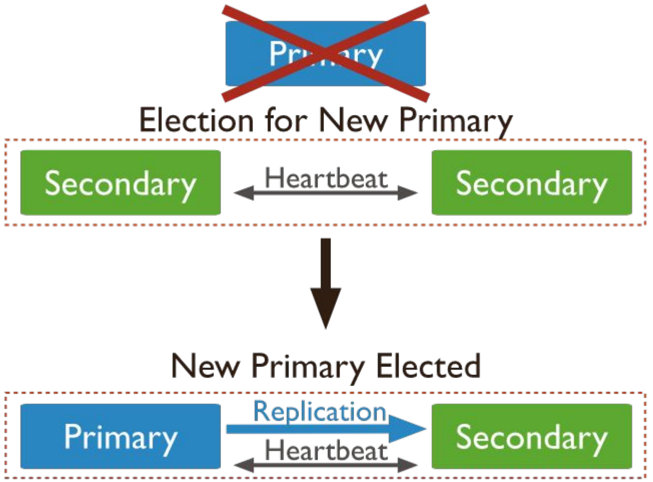
- Perform aggregator functions given a collection of (keys, value) pairs
- Must provide at least a map function, reduction function, and the name of the result set

```
db.collection.mapReduce(
    <mapfunction>,
    <reducefunction>,
    {
        out: <collection>,
        query: <document>,
        sort: <document>,
        limit: <number>,
        finalize: <function>,
        scope: <document>,
        jsMode: <boolean>,
        verbose: <boolean>
    })
```

Data Replication

- Ensure redundancy, backup, and automatic failover
- Replication occurs through groups of servers known as **replica sets** (for each shard)
  - **Primary set**: set of servers that client asks direct updates to
  - **Secondary set**: set of servers used for duplication of data
  - Different properties can be associated with a secondary set, e.g., secondary-only, hidden delayed, arbiters, non-voting
- If the primary set fails the secondary sets "vote" to elect the new primary set



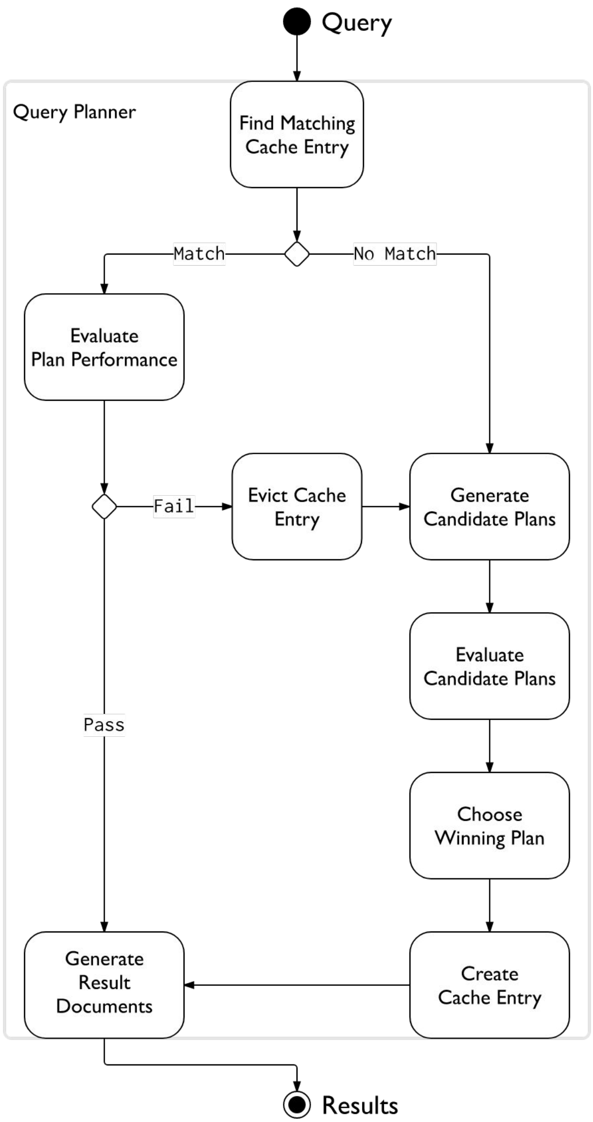


Data Consistency

- Client decides how to enforce consistency for reads
  - All writes and *consistent* reads go to the primary
  - All *eventually consistent* reads are distributed among the secondaries
- Reads to a primary have strict consistency
  - Reads reflect the latest changes to the data
- Reads to a secondary have eventual consistency
  - Updates propagate gradually
  - Client may read a previous state of the database

Query Optimizer

- Not static and cost-based optimizer (like in RDBMSs)
- MongoDB optimizer tries different query plans and learns which ones perform well
  - Because there are no joins, the space of query plans is not so large
  - When testing new plans
    - Execute multiple query plans in parallel
    - As soon as one plan finishes, terminate the other plans
  - If a plan that was working well starts performing poorly try again different plans
    - E.g, data in the DB has changed, parameter values to a query are different

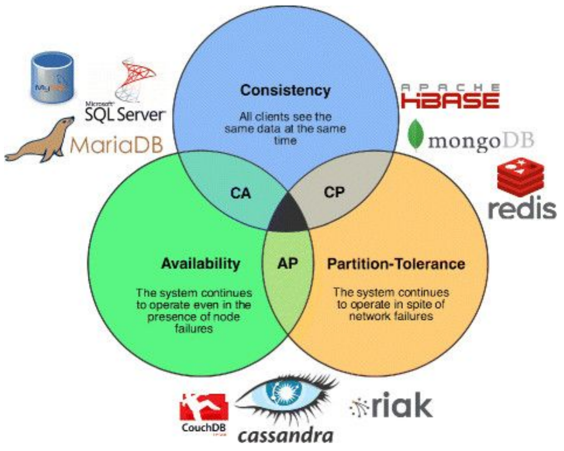


Strengths

- Provide a query language
- High-performance
  - Implemented in C++
- Very rapid development, open source
  - Support for many platforms
  - Many language drivers
- Built to address a distributed database system
  - Sharding
  - Replica sets of data
- Tunable consistency
- Useful when working with a huge quantity of data not requiring a relational model
  - What really matters is the ability to store and retrieve great quantities of data
  - The relationships between the elements does not matter

Limitations

- No referential integrity
- Lack transactions, joins
- High degree of denormalization
  - Updating something in many places instead of one
- Lack of predefined schema is a double-edged sword
  - You must have a data model in your application
  - Objects within a collection can be completely inconsistent in their fields
- CAP Theorem: targets consistency and partition tolerance



UMD DATA605 - Big Data Systems

NoSQL document stores

MongoDB

Couchbase

Couchbase

- NoSQL document-oriented DB like MongoDB
- Couchbase = merge of CouchDB and membase
  - *CouchDB*
    - Open source document store
    - HTTP RESTful API for reading and updating (add, edit, delete) documents
    - Support all 4 ACID properties
  - *membase*
    - Distributed key-value store
    - Designed to scale both up and down
    - Highly available and partition tolerant
- Couchbase
  - Uses HTTP protocol to query and interact with objects in the DB
  - Objects stored in *buckets*
    - Just a collection of documents (in JSON), with no special relation to one another
- For CAP, get consistency and partition tolerance by default, high availability instead of consistency through use of multiple clusters

Architecture

- Every Couchbase node consists of:
  - a data service
  - index service
  - query service
  - cluster manager component
- Services can run on separate nodes of the cluster, if needed
- Data replication across nodes of a cluster (and across data centers)
- Data manager *asynchronously* writes data to disk after acknowledging to the client
  - Optionally can ensure data is written to more than one server before acknowledging a write

Queries

- Can create multiple views over documents
  - Views are optimized/indexed by Couchbase for fast queries
  - Only re-indexed when underlying documents change a lot
- Perform well when there are infrequent changes to the structure of documents
  - And know in advance what kinds of queries you want to execute
- Uses a custom query language called N1QL, based on SQL
  - But runs on JSON documents
  - And queries over multiple documents using joins
- Can also do full-text searches using the indexes
- Map-reduce support
  - First define a view with the columns of the document your are interested in
    - Called the *map*
  - Optionally define aggregate functions over the data
    - The *reduce* step

Couchbase vs MongoDB

- According to Couchbase advocates
- **MongoDB**: hard to scale from single replica set to fully distributed environment
- **MongoDB**: performance degrades with increasing numbers of clients/users
  - **Couchbase**: Scales seamlessly, with an in-memory architecture, and able to scale across multiple nodes
- **MongoDB**: susceptible to data loss from failures
  - **Couchbase**: no master, no single point of failure
  - During failover, prevents different nodes from accepting simultaneous reads of writes of same data (maintains consistency)
- **MongoDB**: require a 3rd party cache to help it perform well
  - **Couchbase**: has integrated in-memory cache (memcached)
  - Keeps frequently accessed documents, metadata, and indexes in RAM, yielding high read/write throughput at low latency