Reference Cards for MongoDB

What is MongoDB?

MongoDB is an open-source, general purpose database.

Instead of storing data in rows and columns as one would with a relational database, MongoDB uses a document data model, and stores a binary form of JSON documents called BSON. Documents contain one or more fields, and each field contains a value of a specific data type, including arrays and binary data. Documents are stored in collections, and collections are stored in databases. It may be helpful to think of documents as roughly equivalent to rows in a relational database; fields as equivalent to columns; and collections as tables. There are no fixed schemas in MongoDB, so documents can vary in structure and can be adapted dynamically.

MongoDB provides full index support, including secondary, compound, and geospatial indexes. MongoDB also features a rich query language, atomic update modifiers, text search, the Aggregation Framework for analytics similar to SQL GROUP BY operations, and MapReduce for complex in-place data analysis.

Built-in replication with automated failover provides high availability. Auto-sharding enables horizontal scaling for large deployments. MongoDB also provides native, idiomatic drivers for all popular programming languages and frameworks to make development natural.

Queries

Queries

Queries and What They Match

`{a: 10}`	Docs where **a** is **10**, or an array containing the value **10**.
`{a: 10, b: "hello"}`	Docs where **a** is **10** and **b** is **"hello"**.
`{a: {$gt: 10}}`	Docs where **a** is greater than **10**. *Also available:* ***$lt*** *(<),* ***$gte*** *(>=),* ***$lte*** *(<=), and* ***$ne*** *(!=).*
`{a: {$in: [10, "hello"]}}`	Docs where **a** is either **10** or **"hello"**.
`{a: {$all: [10, "hello"]}}`	Docs where **a** is an array containing both **10** and **"hello"**.
`{"a.b": 10}`	Docs where **a** is an embedded document with **b** equal to **10**.
`{a: {$elemMatch: {b: 1, c: 2}}}`	Docs where **a** is an array that contains an element with both **b** equal to **1** and **c** equal to **2**.
`{$or: [{a: 1}, {b: 2}]}`	Docs where **a** is **1** or **b** is **2**.
`{a: /^m/}`	Docs where **a** begins with the letter **m**. *One can also use the regex operator:* ***{a: {$regex: "^m"}}***.
`{a: {$mod: [10, 1]}}`	Docs where **a** mod **10** is **1**.
`{a: {$type: 2}}`	Docs where **a** is a string. *See bsonspec.org for more.*
`{ $text: { $search: "hello" } }`	Docs that contain **"hello"** on a text search. Requires a text index.

Queries and What They Match (continued)

`{a: {$near: {$geometry:{ type: "Point", coordinates: [ -73.98, 40.75 ] }} } }`	Docs sorted in order of nearest to farthest from the given coordinates. *For geospatial queries one can also use* ***`$geoWithin`*** *and* ***`$geoIntersects`*** *operators.*

Not Indexable Queries

The following queries cannot use indexes. These query forms should normally be accompanied by at least one other query term which does use an index.

`{a: {$nin: [10, "hello"]}}`	Docs where **a** is anything but **10** or **"hello"**.
`{a: {$size: 3}}`	Docs where **a** is an array with exactly **3** elements.
`{a: {$exists: true}}`	Docs containing an **a** field.
`{a: /foo.*bar/}`	Docs where **a** matches the regular expression **foo.*bar.**
`{a: {$not: {$type: 2}}}`	Docs where **a** is not a string. **$not** negates any of the other query operators.

For More Information

http://docs.mongodb.org/manual/tutorial/query-documents/

http://docs.mongodb.org/manual/reference/operator/query/

Updates

Updates

Field Update Modifiers

`{$inc: {a: 2}}`	Increments **a** by **2**.
`{$set: {a: 5}}`	Sets **a** to the value **5**.
`{$unset: {a: 1}}`	Deletes the **a** key.
`{$max: { a: 10 } }`	Sets **a** to the greater value, either current or **10**. If **a** does not exist sets **a** to **10**.
`{$min: {a: -10}}`	Sets **a** to the lower value, either current or **-10**. If **a** does not exist sets **a** to **-10**.
`{$mul: { a: 2 } }`	Sets **a** to the product of the current value of **a** and **2**. If **a** does not exist sets **a** to **0**.
`{$rename: { a: "b"} }`	Renames field **a** to **b**.
`{ $setOnInsert: { a: 1 } }, { upsert: true }`	Sets field **a** to **1** in case of upsert operation.
`{$currentDate: { a: { $type: "date"} } }`	Sets field **a** with the current date. ***$currentDate*** *can be specified as date or timestamp. Note that as of 3.0, date and timestamp are not equivalent for sort order.*
`{ $bit: { a: { and: 7 } } }`	Performs the bitwise **and** operation over **a** field: 1000 0100 ------- 1100 Supports **and|xor|or** bitwise operators.

Array Update Operators

`{$push: {a: 1}}`	Appends the value **1** to the array **a**.
`{$push: {a: {$each: [1, 2]}}}`	Appends both **1** and **2** to the array **a**.
`{$addToSet: {a: 1}}`	Appends the value **1** to the array **a** (if the value doesn't already exist).
`{$addToSet: {a: {$each: [1, 2]}}}`	Appends both **1** and **2** to the array **a** (if they don't already exist).
`{$pop: {a: 1}}`	Removes the last element from the array **a**.
`{$pop: {a: -1}}`	Removes the first element from the array **a**.
`{$pull: {a: 5}}`	Removes all occurrences of **5** from the array **a**.
`{$pullAll: {a: [5, 6]}}`	Removes multiple occurrences of **5** or **6** from the array **a**.

For More Information http://docs.mongodb.org/manual/reference/operator/update/

Aggregation Framework

Aggregation Framework

The aggregation pipeline is a framework for data aggregation modeled on the concept of data processing pipelines. Documents enter a multi-stage pipeline that transforms the documents into aggregated results. Pipeline stages appear in an array. Documents pass through the stages in sequence. Structure an aggregation pipeline using the following syntax:

```
db.collection.aggregate( [ { <stage> }, ... ] )
```

Aggregation Framework Stages

`{$match: { a: 10 }}`	Passes only documents where **a** is **10**.	Similar to `find()`
`{$project: { a: 1, _id:0}}`	Reshapes each document to include only field **a**, removing others.	Similar to `find()` projection
`{$project: { new_a: "$a" }}`	Reshapes each document to include only **_id** and the new field **new_a** with the value of **a**.	`{a:1}` => `{new_a:1}`
`{$project: { a: {$add:["$a", "$b"]}}}`	Reshapes each document to include only **_id** and field **a**, set to the sum of **a** and **b**.	`{a:1, b:10}` => `{a: 11}`
`{$project: { stats: { value: "$a", fraction: {$divide: ["$a", "$b"]} } } }`	Reshapes each document to contain only **_id** and the new field **stats** which contains embedded fields **value**, set to the value of **a**, and **fraction**, set to the value of **a** divided by **b**.	`{a: 10, b:2}` => `{ stats:{ value: 10, fraction: 5} }`

`{$group: { _id: "$a", count:{$sum:1} } }`	Groups documents by field **a** and computes the **count** of each distinct **a** value.	`{a:"hello"}, {a:"goodbye"}, {a:"hello"} => {_id:"hello", count:2}, {_id:"goodbye", count:1}`

Aggregation Framework Stages (continued)

`{$group: { _id: "$a", names: {$addToSet: "$b"}} }`	Groups documents by field **a** with new field **names** consisting of a set of **b** values.	`{a:1, b:"John"}, {a:1, b:"Mary"} => {_id:1, names:["John", "Mary"] }`
`{$unwind: "$a"}`	Deconstructs array field **a** into individual documents of each element.	`{a: [2,3,4]} => {a:2}, {a:3}, {a:4}`
`{$limit: 10}`	Limits the set of documents to **10**, passing the first **10** documents.	
`{$sort: {a:1}}`	Sorts results by field **a** ascending.	
`{$skip: 10}`	Skips the first **10** documents and passes the rest.	
`{$out: "myResults"}`	Writes resulting documents of the pipeline into the collection **"myResults"**.	Must be the **last stage** of the pipeline.

For More Information http://docs.mongodb.org/master/core/aggregation-introduction/

Indexing

Indexing

Index Creation Syntax `db.coll.createIndex(<key_pattern>, <options>)`

Creates an index on collection **`coll`** with given key pattern and options.

Indexing Key Patterns

`{a:1}`	Simple index on field **`a`**.
`{a:1, b:-1}`	Compound index with **`a`** ascending and **`b`** descending.
`{"a.b": 1}`	Ascending index on embedded field **`"a.b"`**.
`{a: "text"}`	Text index on field **`a`**. A collection can have at most one text index.
`{a: "2dsphere"}`	Geospatial index where the **`a`** field stores GeoJSON data. See documentation for valid GeoJSON formatting.
`{a: "hashed"}`	Hashed index on field **`a`**. Generally used with hashbased sharding.

Index options

`{unique: true}`	Creates an index that requires all values of the indexed field to be unique.

`{background: true}`	Creates this index in the background; useful when you need to minimize index creation performance impact.
`{name: "foo"}`	Specifies a custom name for this index. If not specified, the name will be derived from the key pattern.
`{sparse: true}`	Creates entries in the index only for documents having the index key.
`{expireAfterSeconds:360}`	Creates a time to live (TTL) index on the index key. This will force the system to drop the document after **3600** seconds expire. *Only works on keys of* ***date*** *type.*
`{default_language: 'portuguese'}`	Used with text indexes to define the default language used for stop words and stemming.

Examples

`db.products.createIndex( {'supplier':1}, {unique:true})`	Create ascending index on **supplier** assuring unique values.
`db.products.createIndex( {'description': 'text', {'default_language': 'spanish'})`	Creates text index on **description** key using **Spanish** for stemming.
`db.products.createIndex( { 'regions': 1 }, {sparse:true})`	Creates ascending sparse index on **regions** key. If regions is an array – e.g., `regions: ['EMEA', 'NA', 'LATAM']` – will create a multikey index.
`db.stores.createIndex( {location: "2dsphere"})`	Creates a **2dsphere** geospatial index on **location key**.

Administration

`db.products.getIndexes()`	Gets a list of all indexes on the **`products`** collection.
`db.products.reIndex()`	Rebuilds all indexes on this collection.
`db.products.dropIndex({x: 1, y: -1})`	Drops the index with key pattern **`{x: 1, y: -1}`**. *Use **`db.products.dropIndex('index_a')`** to drop index named **`index_a`**. Use **`db.products.dropIndexes()`** to drop all indexes on the products collection.*

For More Information http://docs.mongodb.org/master/core/indexes-introduction/

Replication

Replication

What is a Majority?

If your set consists of...
1 server, 1 server is a majority.
2 servers, 2 servers are a majority.
3 servers, 2 servers are a majority.
4 servers, 3 servers are a majority.
...

Setup

To initialize a three-node replica set including one arbiter, start three **`mongod`** instances, each using the **`--replSet`** flag followed by a name for the replica set. For example:

```
mongod --replSet cluster-foo
```

Next, connect to one of the **`mongod`** instances and run the following:

```
rs.initiate()
rs.add("host2:27017")
rs.add("host3:27017", true)
```

`rs.add()` can also accept a document parameter, such as **`rs.add({"_id": 4, "host": "host4:27017"})`**. The document can contain the following options:

`priority: n`	Members will be elected primary in order of priority, if possible. Higher values make a member more eligible to become a primary. **`n=0`** means this member will never be a primary.
`votes: n`	Assigns a member voting privileges (**`n=1`** for voting, **`n=0`** for nonvoting).
`slaveDelay: n`	This member will always be a secondary and will lag **`n`** seconds behind the master.

`arbiterOnly: true`	This member will be an arbiter.

Setup (continued)

`hidden: true`	Do not show this member in **`isMaster`** output. Use this option to hide this member from clients.
`tags: [...]`	Member location description. See *docs.mongodb.org/manual/data-center-awareness.*

Administration

`rs.initiate()`	Creates a new replica set with one member.
`rs.add("host:port")`	Adds a member.
`rs.addArb("host:port")`	Adds an arbiter.
`rs.remove("host:port")`	Removes a member.
`rs.status()`	Returns a document with information about the state of the replica set.
`rs.conf()`	Returns the replica set configuration document.
`rs.reconfig(newConfig)`	Re-configures a replica set by applying a new replica set configuration object.
`rs.isMaster()`	Indicates which member is primary.

`rs.stepDown(n)`	Forces the primary to become a secondary for **`n`** seconds, during which time an election can take place.

Administration (continued)

`rs.freeze(n)`	Prevents the current member from seeking election as primary for **`n`** seconds. ***`n=0`*** *means unfreeze.*
`rs.printSlaveReplicationInfo()`	Prints a report of the status of the replica set from the perspective of the secondaries.

For More Information http://docs.mongodb.org/master/core/replication-introduction/

Sharding

Sharding

`sh.enableSharding( 'products')`	Enables sharding on **`products`** database.
`sh.shardCollection( 'products.catalog', { sku:1, brand:1})`	Shards collection **`catalog`** of **`products`** database with shard key consisting of the **`sku`** and **`brand`** fields.
`sh.status()`	Prints a formatted report of the sharding configuration and the information regarding existing chunks in a sharded cluster.
`sh.addShard( 'REPLICA1/host:27017')`	Adds existing replica set **`REPLICA1`** as a shard to the cluster.

For More Information

http://docs.mongodb.org/master/core/sharding-introduction/

Mapping SQL to MongoDB

Mapping SQL to MongoDB

Converting to MongoDB Terms

MYSQL Executable	Oracle Executable	MongoDB Executable
mysqld	oracle	mongod
mysql	sqlplus	mongo

SQL Term	MongoDB Term
database (schema)	database
table	collection
index	index
row	document
column	field
joining	linking & embedding
partition	shard

Queries and other operations in MongoDB are represented as documents passed to **`find()`** and other methods. Below are examples of SQL statements and the analogous statements in MongoDB JavaScript shell syntax.

SQL	MongoDB
`CREATE TABLE users (name VARCHAR(128), age NUMBER)`	`db.createCollection("users")`
`INSERT INTO users VALUES ('Bob', 32)`	`db.users.insert({name: "Bob", age: 32})`
`SELECT * FROM users`	`db.users.find()`
`SELECT name, age FROM users`	`db.users.find({}, {name: 1, age: 1, _id:0})`
`SELECT name, age FROM users WHERE age = 33`	`db.users.find({age: 33}, {name: 1, age: 1, _id:0})`
`SELECT * FROM users WHERE age > 33`	`db.users.find({age: {$gt: 33}})`
`SELECT * FROM users WHERE age <= 33`	`db.users.find({age: {$lte: 33}})`
`SELECT * FROM users WHERE age > 33 AND age < 40`	`db.users.find({age: {$gt: 33, $lt: 40}})`
`SELECT * FROM users WHERE age = 32 AND name = 'Bob'`	`db.users.find({age: 32, name: "Bob"})`

`SELECT * FROM users WHERE age = 33 OR name = 'Bob'`	`db.users.find({$or:[{age:33}, {name:"Bob"}]})`
`SELECT * FROM users WHERE age = 33 ORDER BY name ASC`	`db.users.find({age: 33}). sort({name: 1})`
`SELECT * FROM users ORDER BY name DESC`	`db.users.find().sort({name: -1})`
`SELECT * FROM users WHERE name LIKE '%Joe%'`	`db.users.find({name: /Joe/})`

SQL	MongoDB
`SELECT * FROM users WHERE name LIKE 'Joe%'`	`db.users.find({name: /^Joe/})`
`SELECT * FROM users LIMIT 10 SKIP 20`	`db.users.find().skip(20).limit(10)`
`SELECT * FROM users LIMIT 1`	`db.users.findOne()`
`SELECT DISTINCT name FROM users`	`db.users.distinct("name")`
`SELECT COUNT(*) FROM users`	`db.users.count()`
`SELECT COUNT(*) FROM users WHERE AGE > 30`	`db.users.find({age: {$gt: 30}}).count()`

`SELECT COUNT(AGE) FROM users`	`db.users.find({age: {$exists: true}}).count()`
`UPDATE users SET age = 33 WHERE name = 'Bob'`	`db.users.update({name: "Bob"}, {$set: {age: 33}}, {multi: true})`
`UPDATE users SET age = age + 2 WHERE name = 'Bob'`	`db.users.update({name: "Bob"}, {$inc: {age: 2}}, {multi: true})`
`DELETE FROM users WHERE name = 'Bob'`	`db.users.remove({name: "Bob"})`
`CREATE INDEX ON users (name ASC)`	`db.users.createIndex({name: 1})`
`CREATE INDEX ON users (name ASC, age DESC)`	`db.users.createIndex({name: 1, age: -1})`

`EXPLAIN SELECT * FROM users WHERE age = 32`	`db.users.find({age: 32}).explain()`
	
	`(db.users.explain().find({age: 32}) for 3.0)`

`SELECT age, SUM(1) AS counter FROM users GROUP BY age`	`db.users.aggregate( [ {$group: {'_id': '$age', counter: {$sum:1}} } ])`

SQL	MongoDB
`SELECT age, SUM(1) AS counter FROM users WHERE country = "US" GROUP BY age`	`db.users.aggregate([ {$match: {country: 'US'} }, {$group: {'_id': '$age', counter: {$sum:1}} } ])`

```
SELECT age AS “how_old” FROM
users
```

```
db.users.aggregate( [
{$project: {“how_old”:
“$age”}}
])
```