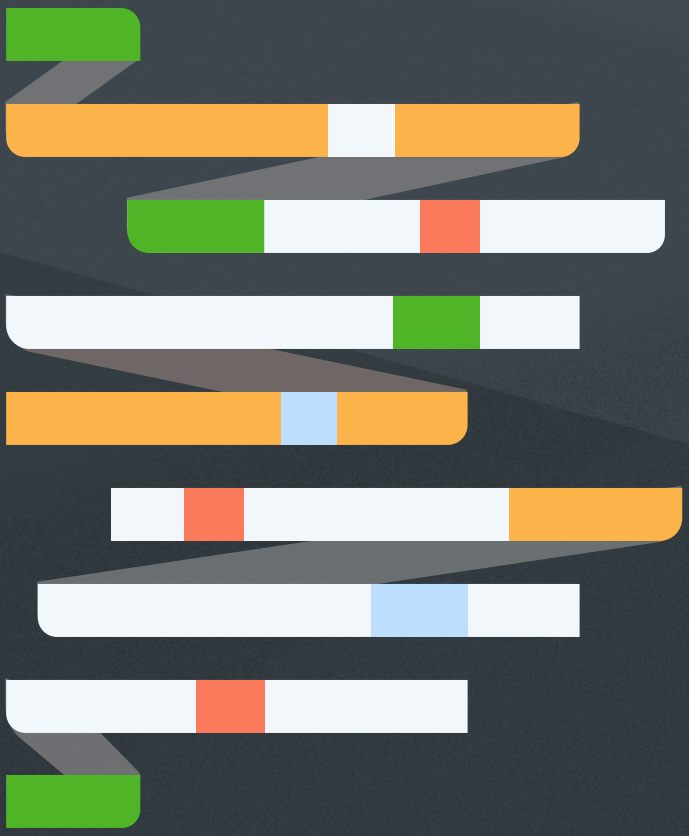
QUICK

REFERENCE CARDS

FOR MONGODB 3.4

Updated July 2017



mongoDB.

MONGODB IS AN OPEN-SOURCE, GENERAL PURPOSE DATABASE.

Instead of storing data in rows and columns as a relational database does, 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 with powerful analytics capabilities, atomic update modifiers, text search, 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.

MongoDB is fully secure, with authentication, authorization & role-based access control, integration with auth mechanisms like LDAP and Kerberos, end-to-end encryption, auditing, and field-level redaction.


COMMAND HELPERS

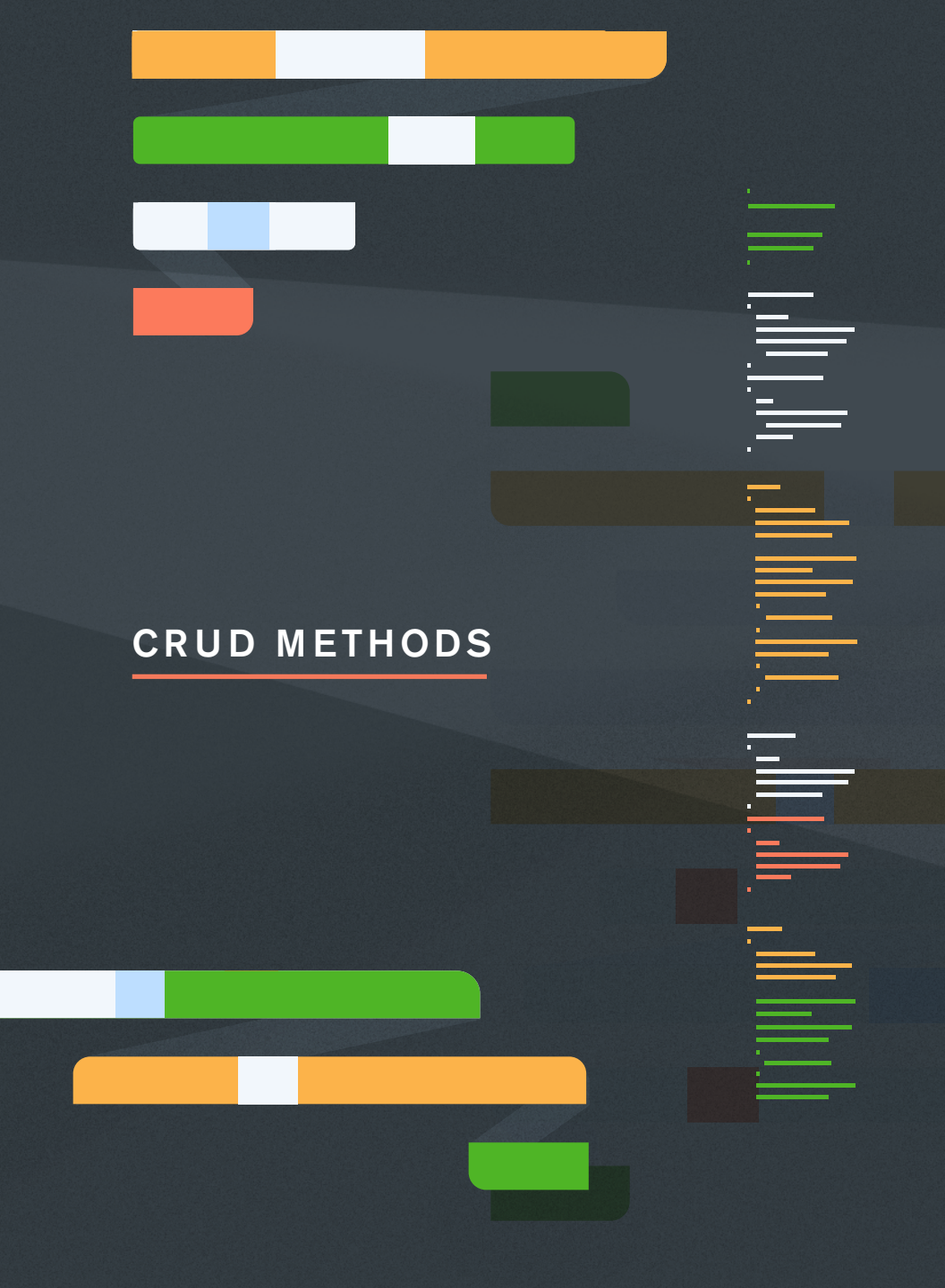
COMMAND HELPERS

The following table lists some common help commands which are available in the `mongo` shell:

`help`	Show help.
`db.help()`	Show help for database methods.
`db.<collection>.help()`	Show help on collection methods. The **`<collection>`** can be the name of an existing collection or a non-existing collection.
`show dbs`	Print a list of all databases on the server.
`use <db>`	Switch current database to **`<db>`**. The mongo shell variable db is set to the current database.
`show collections`	Print a list of all collections for the current database.
`show users`	Print a list of users for the current database.
`show roles`	Print a list of all roles, both user-defined and built-in, for the current database.
`show profile`	Print the five most recent operations that took 1 millisecond or more.
`show databases`	Print a list of all available databases.

FOR MORE INFORMATION

https://docs.mongodb.com/manual/reference/mongo-shell/


CRUD METHODS

CRUD METHODS

Queries typically take the following form:

```
db.<collection>.<method>( <filter>, <options>)
```

db refers to the current database. **<collection>** is the name of the target collection for your query. For **<method>**, substitute the desired query method, (examples below). Each method has its own **<options>** for what it will do with the matching document(s).

`db.collection.insertOne()`	Inserts a document into a collection.
`db.collection.insertMany()`	Inserts multiple documents into a collection.
`db.collection.find()`	Selects documents in a collection based on the filter and returns a cursor to the selected documents.
`db.collection.updateOne()`	Updates a single document within the collection based on the filter.
`db.collection.updateMany()`	Updates all documents within the collection that match the filter.
`db.collection.replaceOne()`	Replaces a single document within the collection based on the filter.
`db.collection.deleteOne()`	Removes a single document from a collection based on the filter.
`db.collection.deleteMany()`	Removes all documents that match the filter from a collection.

FOR MORE INFORMATION

https://docs.mongodb.com/manual/crud/

QUERY FILTER PARAMETERS AND WHAT THEY MATCH

QUERY FILTER PARAMETERS AND WHAT THEY MATCH

MongoDB uses a key-value structure to create query filter parameters, which you can use in the **mongo** shell or with a driver in a client application. For example, the following query finds all documents in the collection named **inventory** in which the **qty** field contains a value greater than 10:

```
db.inventory.find({ "qty" : { $gt: 10 }})
```

Queries take documents as query filter parameters, shown as examples below. Multiple filter parameters can be included in one document, separated by commas.

`{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: {$type: "string"}}`	Docs where **a** is a string.
`{$text: {$search: "hello"}}`	Docs that contain "**hello**" on a full text search.

FOR MORE INFORMATION

https://docs.mongodb.com/manual/reference/bson-types
https://docs.mongodb.com/manual/reference/operator/query/text/

```
{ a:
  { $near:
    { $geometry:
      {
        type: "Point",
        coordinates:
          [ -73.9876, 40.7574
]
      }
    }
  }
}
```

Docs sorted in order of **a** nearest to farthest from the given coordinates. *There must be a 2dsphere index on **a** for this type of query. Coordinates in GeoJSON are listed in the order longitude, latitude.*

```
{ a:
  { $geoWithin:
    { $geometry:
      {
        type : "Polygon",
        coordinates:
          [ [ [ 0, 0 ],
              [ 3, 6 ],
              [ 6, 1 ],
              [ 0, 0 ] ] ]
      }
    }
  }
}
```

Docs where **a** exists entirely within the specified GeoJSON Polygon.

```
{ a:
  { $geoIntersects:
    { $geometry:
      {
        type: "Polygon",
        Coordinates:
          [ [ [ 0, 0 ],
              [ 3, 6 ],
              [ 6, 1 ],
              [ 0, 0 ] ] ]
      }
    }
  }
}
```

Docs where **a** intersects with the specified GeoJSON Polygon, including cases where **a** and the polygon share an edge.

```
{ a:
  { $nearSphere:
    { $geometry:
      {
        type : "Point",
        Coordinates:
          [ -73.9876, 40.7574
]
      },
     $minDistance: 1000,
     $maxDistance: 5000
    }
  }
}
```

Docs where **a** is at least **1000** meters and at most **5000** meters from the specified point, ordered from nearest to farthest. ***$nearsphere*** *requires a geospatial index.*

NOT INDEXABLE QUERIES

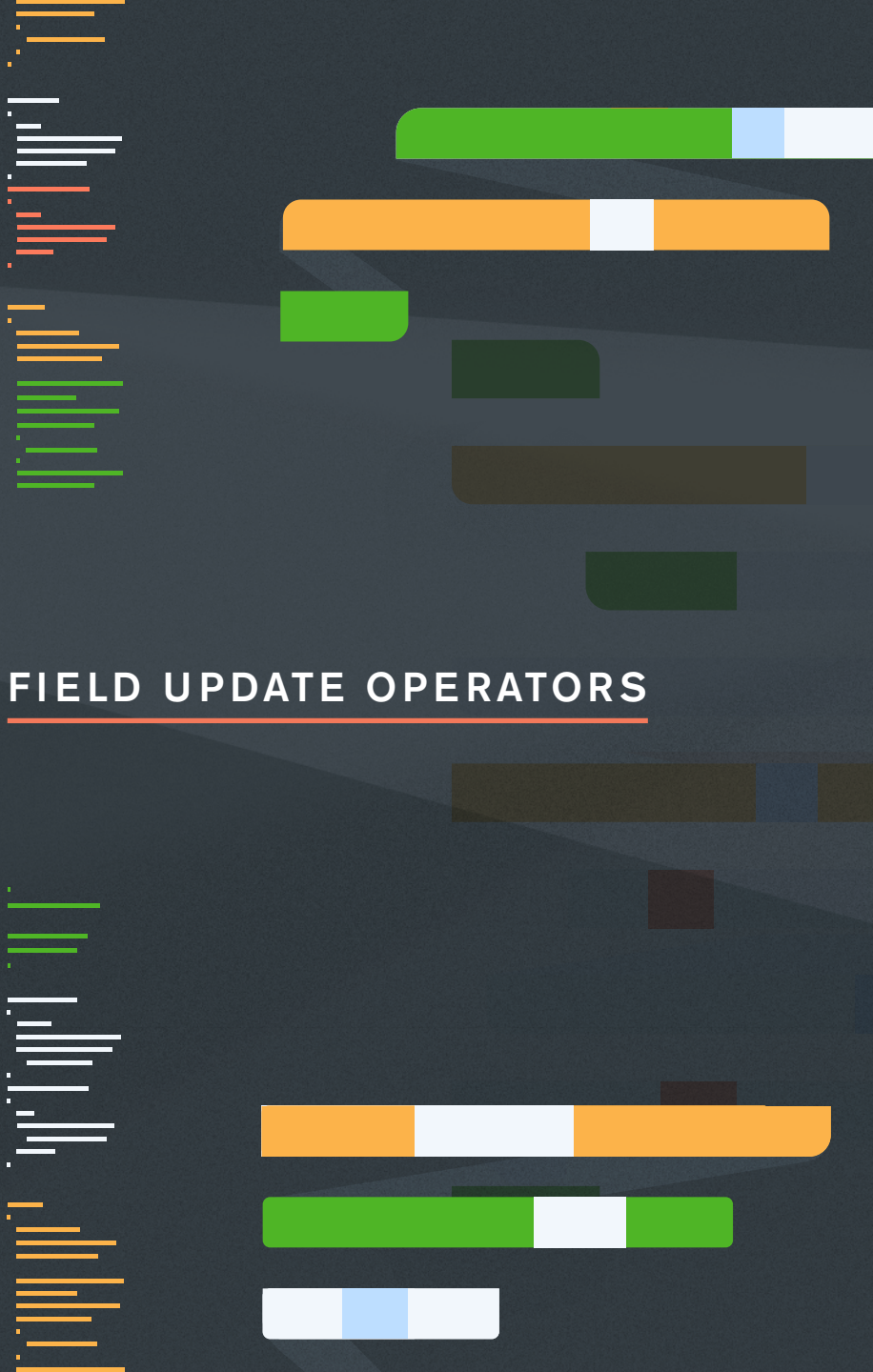
Queries that cannot use indexes will be executed as collection scans – scanning all documents in the collection – which will perform poorly at scale. The following are examples of query types which require a collection scan. To avoid collection scans, these query forms should normally be accompanied 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/

FIELD UPDATE OPERATORS



FIELD UPDATE OPERATORS

`{$inc: {a: 2}}`	Increment **a** by **2**.
`{$set: {a: 5}}`	Set **a** to the value **5**.
`{$unset: {a: 1}}`	Delete the **a** key.
`{$max: {a: 10}}`	Set **a** to the greater value, either current or **10**. If **a** does not exist, set **a** to **10**.
`{$min: {a: -10}}`	Set **a** to the lowest value, either current or **-10**. If **a** does not exist, set **a** to **-10**.
`{$mul: {a: 2}}`	Set **a** to the product of the current value of **a** and **2**. If **a** does not exist set **a** to **0**.
`{$rename: {a: "b"}}`	Rename field **a** to **b**.
`{$setOnInsert: {a: 1}}, {upsert: true}`	Set field **a** to **1** in case of upsert operation.

`{$currentDate: {a: {$type: "date"}}}`	Set 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}}}`	Perform the bitwise **and** operation over **a** field: 1000 0100 --- 1100 *Supports* ***and | xor | or*** *bitwise operators.*

ARRAY UPDATE OPERATORS

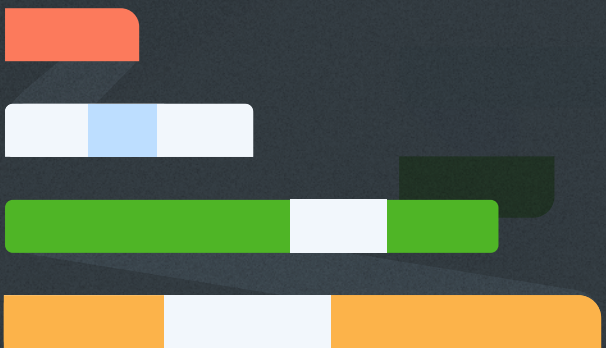
`{$push: {a: 1}}`	Append the value **1** to the array **a**.
`{$push: {a: {$each: [1, 2]}}}`	Append both **1** and **2** to the array **a**.
`{$push: {a: {$each: [10, 20, 30], $slice: -5}}}`	Append **10**, **20**, and **30** to the array **a**, then trim the resulting array to contain only the last 5 elements. *$slice can only be used with the $each modifier. Negative values trim to the last <num> elements, while positive values trim to the first <num> elements.*

`{$push: {a: {$each: [50, 60, 70], $position: 0}}}`	Insert **50**, **60**, and **70** starting at position **0** of the array **a**. *$position can only be used with the $each modifier.*
`{$addToSet: {a: 1}}`	Append the value **1** to the array **a** (if the value doesn't already exist).
`{$addToSet: {a: {$each: [1, 2]}}}`	Append both **1** and **2** to the array **a** (if they don't already exist).
`{$pop: {a: 1}}`	Remove the last element from the array **a**.
`{$pop: {a: -1}}`	Remove the first element from the array **a**.
`{$pull: {a: ($gt: 5}}}`	Remove all values greater than **5** from the array **a**.
`{$pullAll: {a: [5, 6]}}`	Remove 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, part of the MongoDB query language, 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( [ { <stage1> }, { <stage2> } ... ]
)
```

COMMON 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}
`{$group: {_id: "$a", names: {$addToSet: "$b"} }}`	Group 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.	{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.	

`{$skip: 10}`	Skips the first **10** documents and passes the rest.	
`{$sample: {size: 25}}`	Randomly selects **25** documents.	The sample size value must be a positive integer.

`{$lookup: {`
`  from: "inventory",`
`  localField: "item",`
`  foreignField: "sku",`
`  as: "inventory_docs"`
`}}`
	Performs an equality match from the **sku** field in the **inventory** collection to the **item** field in the documents passed into this stage, then adds an array **inventory_docs** to each document with matching documents from **inventory**.	This is a *left outer join.*
		
		The collection specified in **from** cannot be sharded, and must be in the same database.

`{ $graphLookup:`
`  {`
`  from: "airports",`
`  startWith: "$nearestAirport",`
`  connectFromField: "connects",`
`  connectToField: "airport",`
`  maxDepth: 2,`
`  depthField: "numConnections",`
`  as: "destinations"`
`  }`
`}`
	For each document passed into this stage, looks up its **nearestAirport** value in the **airports** collection and recursively matches the **connects** field to the **airport** field within the **airports** collection. The operation specifies a maximum recursion depth of **2**. The array **destinations** is added to each document with the results of the recursive match, including the field **numConnections** with the value of the depth of each match.	The collection specified in **from** cannot be sharded, and must be in the same database.

`{ $bucket: { groupBy: "$price", boundaries: [ 0, 200, 400 ], default: "Other", output: { "count": {$sum: 1}, "titles" : {$push: "$title"} } } }`	Categorizes incoming documents into groups, called buckets, as defined in **`boundaries`**. Documents will be grouped by **`price`** into buckets **`0-200`** and **`200-400`**, with inclusive lower bound and exclusive upper bound. Documents with a price outside of those bounds will be grouped into the bucket **`Other`**. The output is a set of documents, each with **`_id`** set as the lower bound of the bucket, a count field with the sum of documents, and an array **`titles`** of the **`title`** field of incoming documents.	Any input document with a value outside of specified boundaries of or a different BSON type will cause the operation to throw an error. This is avoided by specifying a **`default`** bucket. The **`count`** field is included by default when the **`output`** is not specified.
`{ $bucketAuto: { groupBy: "$price", buckets: 5 } }`	Categorizes incoming documents into buckets, grouping by **`price`** into 5 buckets with bucket boundaries automatically determined to attempt to evenly distribute documents.	Optionally, provide a *granularity* to ensure that bucket boundaries adhere to a particular number series. See the documentation for more details.

`{$sortByCount: $tags}`	Groups incoming documents based on the value of **`tags`**, then computes the count of documents in each distinct group.	The expression can not evaluate to an object.
`{$count: "a"}`	Counts the documents input to this stage and returns a document with a field **`a`** with the value of the count.	
`{$indexStats: {} }`	Returns statistics regarding the use of each index for the collection.	The **`$indexStats`** stage takes an empty document
`{ $geoNear: { near: { type: "Point", coordinates: [ -73.9876, 40.7574 ] }, distanceField: "dist" } }`	Outputs documents in order of nearest to farthest from the specified point, adding the field **`dist`** with a value of the distance from the specified point.	Must be the **`first stage`** of a pipeline. See documentation for other options to pass to **`$geoNear`**.
`{$out: "myResults"}`	Writes resulting documents of the pipeline into the collection "**`myResults`**".	Must be the **`last stage`** of the pipeline

FACETS

New in version 3.4, the `$facet` stage processes multiple aggregation pipelines within a single stage on the same set of input documents. Each sub-pipeline has its own field in the output document where its results are stored as an array of documents. The `$facet` stage has the following form:

```
db.collection.aggregate( [
   { $facet:
      {
         <outputField1>: [ { <stage1> }, { <stage2> }, ... ],
         <outputField2>: [ { <stage1> }, { <stage2> }, ... ],
         ...
      }
   }
] )
```

Facet-related aggregation stages (`$bucket`, `$bucketAuto`, `$sortByCount`) can be used in sub-pipelines for multi-faceted aggregations. All other aggregation stages can also be used in sub-pipelines, with the following exceptions: `$facet`, `$out`, `$geoNear`, `$indexStats`, `$collStats`.

FOR MORE INFORMATION

https://docs.mongodb.com/manual/aggregation/


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**, or a multikey index on an array **a**; it is not necessary to explicitly specify the multikey type for arrays.
`{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 hash-based sharding.

INDEX OPTIONS:

`{unique: true}`	Create an index that requires all values of the indexed field to be unique.
`{background: true}`	Create this index in the background; useful when you need to minimize index creation performance impact.

`{name: "foo"}`	Specify a custom name for this index. If not specified, the name will be derived from the key pattern.
`{expireAfterSeconds:3600 }`	Create 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'}`	Use with text indexes to define the default language used for stop words and stemming.
`{partialFilterExpression: { 'rating.grade': { $gte: 60 } }}`	Partial indexes only index the documents in a collection that meet a specified filter expression – here, where **rating.grade** is greater than **60**. *Partial indexes will only be used by queries that contain the filter expression or a subset thereof.*

EXAMPLES:

`db.products.createIndex( {'supplier':1}, {unique:true} )`	Creates 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( {style: 1, name: 1}, {partialFilterExpression: {rating: { $gt: 5}}} )`	Creates a compound index that indexes only the documents with a rating field greater than **5**. You can specify a **`partialFilterExpression`** option for all MongoDB index types.
`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

https://docs.mongodb.com/manual/indexes/

DOCUMENT VALIDATION

WHAT IS DOCUMENT VALIDATION?

MongoDB provides the capability to validate documents during updates and inserts. Validation rules are specified per collection using the **validator** option.

EXAMPLE

Add document validation to an existing collection using the **collMod** command with the validator option or when creating a new collection using **db.createCollection()**:

```
db.createCollection( "contacts",
   { validator: { $or:
      [
         { phone: { $type: "string" } },
         { email: { $regex: /@mongodb\.com$/ } },
         { status: { $in: [ "Unknown", "Incomplete" ] } }
      ]
   },
   validationLevel: "moderate"
   validationAction: "warn"
} )
```

MongoDB also provides the **validationLevel** option, which determines how strictly MongoDB applies validation rules to existing documents during an update, and the **validationAction** option, which determines whether MongoDB should **error** and reject documents that violate the validation rules or **warn** about the violations in the log but allow invalid documents.

FOR MORE INFORMATION

https://docs.mongodb.com/manual/core/document-validation/

VIEWS

WHAT ARE VIEWS?

Views are often used in relational databases to achieve both data security and a high level of abstraction, making it easier to retrieve data. Unlike regular tables, MongoDB views neither have a physical schema nor use disk space.

MongoDB views execute a pre-specified query. To create a view, use the **`db.createView('view_name','source',[pipeline])`** command, specifying the view name, the view source collection, and an aggregation pipeline that defines the view. This aggregation pipeline, as well as the other parameters, is saved in the **`system.views`** collection. This is the only space that the view will use in the system. A new document is saved in the **`system.views`** collection for each view created.

FOR MORE INFORMATION

https://docs.mongodb.com/manual/core/views/


REPLICATION

REPLICATION

A replica set in MongoDB is a group of `mongod` processes that maintain the same data set. Replica sets provide redundancy and high availability, and are the basis for all production deployments. Replication of data is handled automatically, and in the event of node failure or network partition, an automatic failover occurs.

SETUP:

Deploying a replica set for development or testing is a simple process. The example setup below should not be used for production; for instance, this setup involves only a single server, while a production environment should involve separate servers in different physical locations for high availability. See the documentation for production instructions.

The basic procedure is to start the `mongod` instances that will become members of the replica set, configure the replica set itself, and then add the `mongod` instances to it.

Before deploying a replica set, MongoDB must already be installed on each system that will be part of the replica set. Ensure that your network configuration allows all possible connections between each member.

EXAMPLE SETUP, 3-MEMBER REPLICA SET

1. Create data directories for each member with a command similar to the following.

```
mkdir -p /srv/mongodb/rs0-0 /srv/mongodb/rs0-1 /srv/mongodb/rs0-2
```

2. Start 3 `mongod` instances in their own shell windows with a command similar to the following. These commands start each `mongod` as a member of a replica set named `rs0`, each with a distinct port; if you are already using these ports, select different ones.

```
1. mongod --port 27017 --dbpath /srv/mongodb/rs0-0 --replSet rs0
2. mongod --port 27018 --dbpath /srv/mongodb/rs0-1 --replSet rs0
3. mongod --port 27019 --dbpath /srv/mongodb/rs0-2 --replSet rs0
```

3. Connect to any `mongod` instance through the `mongo` shell. Here, we connect to the first `mongod` created.

```
mongo --port 27017
```

4. While connected, use the command below to initiate a replica set. As written below, default configuration will be used. See the documentation for how to include a configuration document when initializing.

```
rs.initiate()
```

5. In the same `mongo` shell, add the two remaining `mongod` to the replica set using the commands below.

```
rs.add( "rs0-1:27018" )
rs.add( "rs0-2:27019" )
```

6. The three-member replica is now running. At any time, use `rs.conf()` to check the replica set configuration or `rs.status()` to check the status.

ADMINISTRATION:

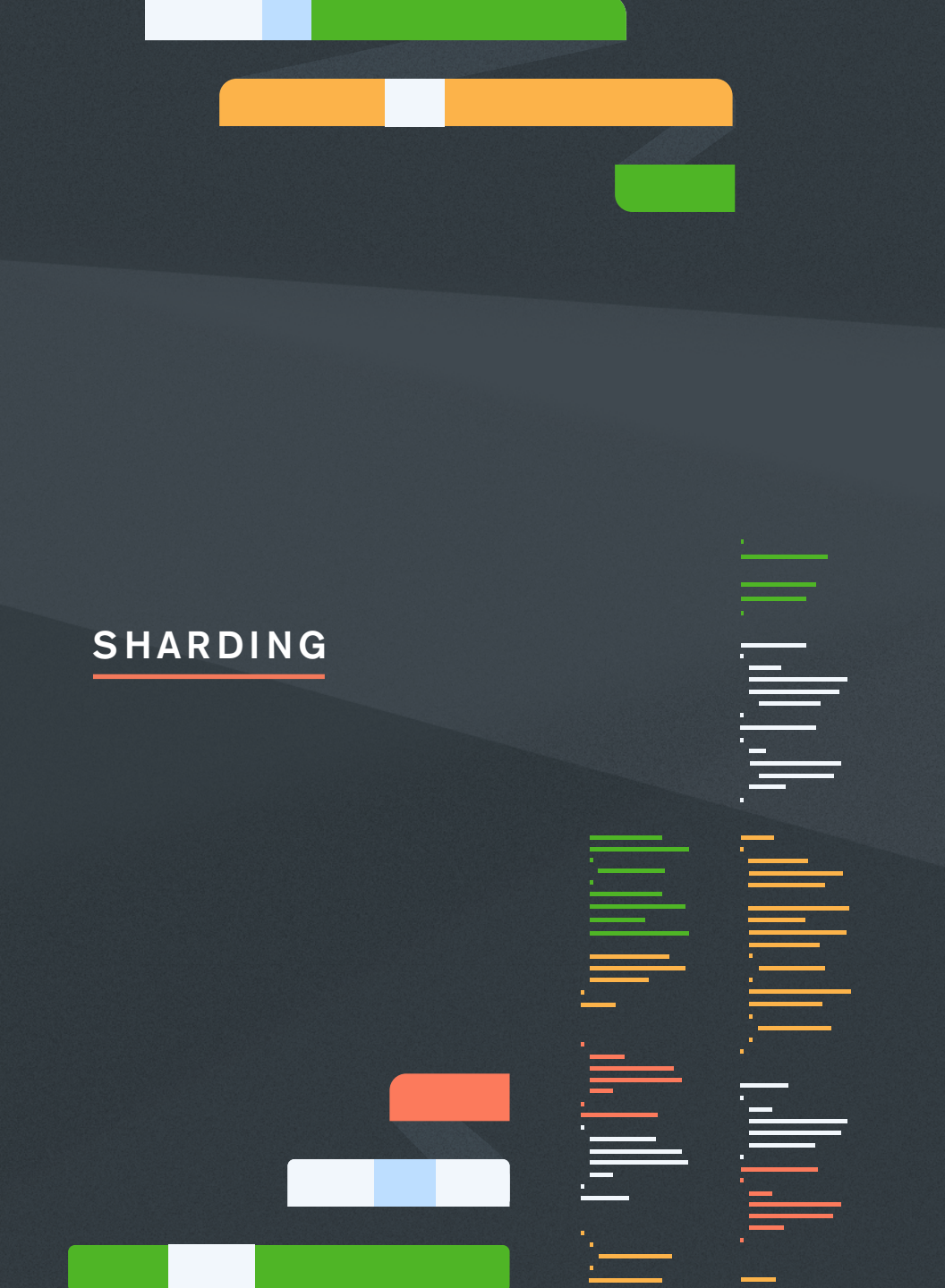
`rs.initiate()`	Create a new replica set with one member.
`rs.add("host:port")`	Add a member.
`rs.addArb("host:port")`	Add an arbiter.
`rs.remove("host:port")`	Remove a member.
`rs.status()`	Returns a document with information about the state of the replica set.

`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()`	See which member is primary.
`rs.stepDown(n)`	Force the primary to become a secondary for **n** seconds, during which time an election can take place.
`rs.freeze(n)`	Prevent the current member from seeking election as primary for **n** seconds. **n=0** *means unfreeze*.
`rs.printSlave ReplicationInfo()`	Prints a report of the status of the replica set from the perspective of the secondaries.

FOR MORE INFORMATION

https://docs.mongodb.com/manual/replication/

SHARDING



WHAT IS SHARDING?

Sharding is a method for distributing data across multiple machines. MongoDB uses sharding to support deployments with very large data sets and high throughput operations.

Database systems with large data sets or high throughput applications can challenge the capacity of a single server. For example, high query rates can exhaust the CPU capacity of the server. Working set sizes larger than the system's RAM stress the I/O capacity of disk drives.

There are two methods for addressing system growth: vertical and horizontal scaling.

VERTICAL SCALING

Vertical Scaling involves increasing the capacity of a single server, such as using a more powerful CPU, adding more RAM, or increasing the amount of storage space. Limitations in available technology may restrict a single machine from being sufficiently powerful for a given workload. Additionally, Cloud-based providers have hard ceilings based on available hardware configurations. As a result, there is a practical maximum for vertical scaling.

HORIZONTAL SCALING

Horizontal Scaling involves dividing the system dataset and load over multiple servers, adding additional servers to increase capacity as required. While the overall speed or capacity of a single machine may not be high, each machine handles a subset of the overall workload, potentially providing better efficiency than a single high-speed high-capacity server. Expanding the capacity of the deployment only requires adding additional servers as needed, which can be a lower overall cost than high-end hardware for a single machine. The trade off is increased complexity in infrastructure and maintenance for the deployment.

MongoDB supports horizontal scaling through sharding.

`sh.enableSharding( "products")`	Enable sharding on **products** database.
`sh.shardCollection( "products.catalog", { sku:1, brand:1 } )`	Shard collection **catalog** of **products** database with shard key consisting of the **sku** and **brand** fields. *This is an example of range-based sharding. Range-based sharding involves dividing data into contiguous ranges determined by the shard key values. In this model, documents with "close" shard key values are likely to be in the same chunk or shard. This allows for efficient queries where reads target documents within a contiguous range.*
`sh.shardCollection( "products.collection", { _id : "hashed" } )`	Shard collection **catalog** of **products** database with shard key consisting of a hash of the **_id** field. *This is an example of hashed sharding. Use hashed sharding for collections that do not naturally contain a key that will ensure an even distribution of documents across shards. Hashing offers even distribution of data at the likely expense of more broadcast operations. Uses a hashed index of a single field as the shard key to partition data across your sharded cluster.*

`sh.status()`	Print 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

https://docs.mongodb.com/manual/sharding/


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 people ( id MEDIUMINT NOT NULL AUTO_INCREMENT, user_id Varchar(30), age Number, status char(1), PRIMARY KEY (id) )`	`db.people.insertOne( { user_id: "abc123", age: 55, status: "A" } )`
`ALTER TABLE people ADD join_date DATETIME`	`db.people.updateMany( { }, { $set: { join_date: new Date() } } )`
`ALTER TABLE people DROP COLUMN join_date`	`db.people.updateMany( { }, { $unset: { "join_date": "" } } )`
`CREATE INDEX idx_user_id_asc ON people(user_id)`	`db.people.createIndex( { user_id: 1 } )`
`CREATE INDEX idx_user_id_ asc_age_desc ON people(user_ id, age DESC)`	`db.people.createIndex( { user_id: 1, age: -1 } )`
`DROP TABLE people`	`db.people.drop()`
`INSERT INTO people(user_ id, age, status) VALUES ("bcd001", 45, "A")`	`db.people.insertOne( { user_id: "bcd001", age: 45, status: "A" } )`
`SELECT * FROM people`	`db.people.find()`

`SELECT id, user_id, status FROM people`	db.people.find({ }, { user_id: 1, status: 1 })
`SELECT user_id, status FROM people`	db.people.find({ }, { user_id: 1, status: 1, _id: 0 })
`SELECT * FROM people WHERE status = "A"`	db.people.find({ status: "A" })
`SELECT user_id, status FROM people WHERE status = "A"`	db.people.find({ status: "A" }, { user_id: 1, status: 1, _id: 0 })
`SELECT * FROM people WHERE status != "A"'`	db.people.find({ status: { $ne: "A" } })
`SELECT * FROM people WHERE status = "A" AND age = 50`	db.people.find({ status: "A", age: 50 })
`SELECT * FROM people WHERE status = "A" OR age = 50`	db.people.find({ $or: [{ status: "A" } , { age: 50 }] })
`SELECT * FROM people WHERE age > 25`	db.people.find({ age: { $gt: 25 } })
`SELECT * FROM people WHERE age < 25`	db.people.find({ age: { $lt: 25 } })
`SELECT * FROM people WHERE age > 25 AND age <= 50`	db.people.find({ age: { $gt: 25, $lte: 50 } })
`SELECT * FROM people WHERE user_id like "%bc%"`	db.people.find({ user_id: /bc/ })

`SELECT * FROM people WHERE user_id like "bc%"`	`db.people.find( { user_id: { $regex: /^bc/ } } )`
`SELECT * FROM people WHERE status = "A" ORDER BY user_id ASC`	`db.people.find( { status: "A" } ).sort( { user_id: 1 } )`
`SELECT * FROM people WHERE status = "A" ORDER BY user_id DESC`	`db.people.find( { status: "A" } ).sort( { user_id: -1 } )`
`SELECT COUNT(*) FROM people`	`db.people.count()`
`SELECT COUNT(user_id) FROM people`	`db.people.count( { user_id: { $exists: true } } )`
`SELECT COUNT(*) FROM people WHERE age > 30`	`db.people.count( { age: { $gt: 30 } } )`
`SELECT DISTINCT(status) FROM people`	`db.people.distinct( "status" )`
`SELECT * FROM people LIMIT 1`	`db.people.findOne()`
`SELECT * FROM people LIMIT 5 SKIP 10`	`db.people.find().limit(5).skip(10)`
`EXPLAIN SELECT * FROM people WHERE status = "A"`	`db.people.find( { status: "A" } ).explain()`
`UPDATE people SET status = "C" WHERE age > 25`	`db.people.updateMany( { age: { $gt: 25 } }, { $set: { status: "C" } } )`

`UPDATE people SET age = age + 3 WHERE status = "A"`	db.people.updateMany({ status: "A" } , { $inc: { age: 3 } })
`DELETE FROM people WHERE status = "D"`	db.people.deleteMany({ status: "D" })
`DELETE FROM people`	db.people.deleteMany({ })

FOR MORE INFORMATION

http://docs.mongodb.org/manual/reference/sql-comparison/

RESOURCES

LEARN

Downloads - mongodb.com/download-center
Enterprise Advanced - mongodb.com/enterprise
MongoDB Manual - docs.mongodb.com
Free Online Education - university.mongodb.com
Presentations - mongodb.com/presentations
In-person Training - university.mongodb.com/training

SUPPORT

Stack Overflow - stackoverflow.com/questions/tagged/mongodb
Google Group - groups.google.com/group/mongodb-user
Bug Tracking - jira.mongodb.org
MongoDB Management Service - mms.mongodb.com
Commercial Support - mongodb.com/support

COMMUNITY

MongoDB User Groups (MUGs) - mongodb.com/user-groups
MongoDB Events - mongodb.com/events

SOCIAL

Twitter - @MongoDB
Facebook - facebook.com/mongodb
LinkedIn - linkedin.com/groups/MongoDB-2340731

CONTACT

Contact MongoDB - mongodb.com/contact