MongoDB Introduction

Jing Hua Ye

[1]Department of Computer Science
CIT

Outline

What is MongoDB?

- A document-oriented, NoSQL database
- Hash-based, schema-less database
  - No Data Definition Language
  - In practice, this means you can store hashes with any keys and values that you choose
    - Keys are a basic data type but in reality stored as strings
    - Document Identifiers (_id) will be created for each document, field name reserved by system
  - Application tracks the schema and mapping
  - Use JSON format or BSON format
- Written in C++
- Supports APIs (drivers) in many computer languages: JavaScript, Java, Python etc

Functionality of MongoDB

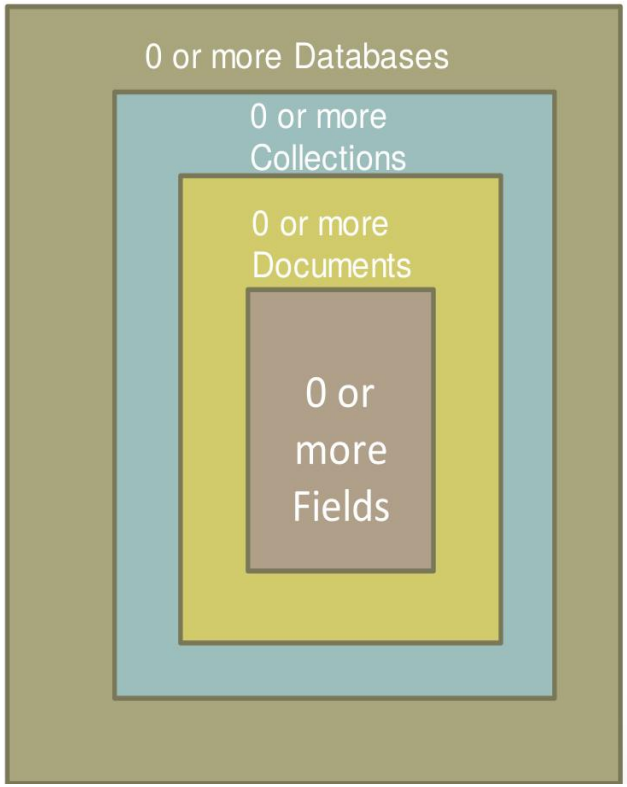
- Dynamic schema (No DDL)
- Designed with both scalability and developer agility.
- Secondary indexes
- Query language via an API
- No joins nor transactions
- Master-slave replication with automated failover (replica sets)
- Atomic writes and fully-consistent reads

Why use MongoDB?

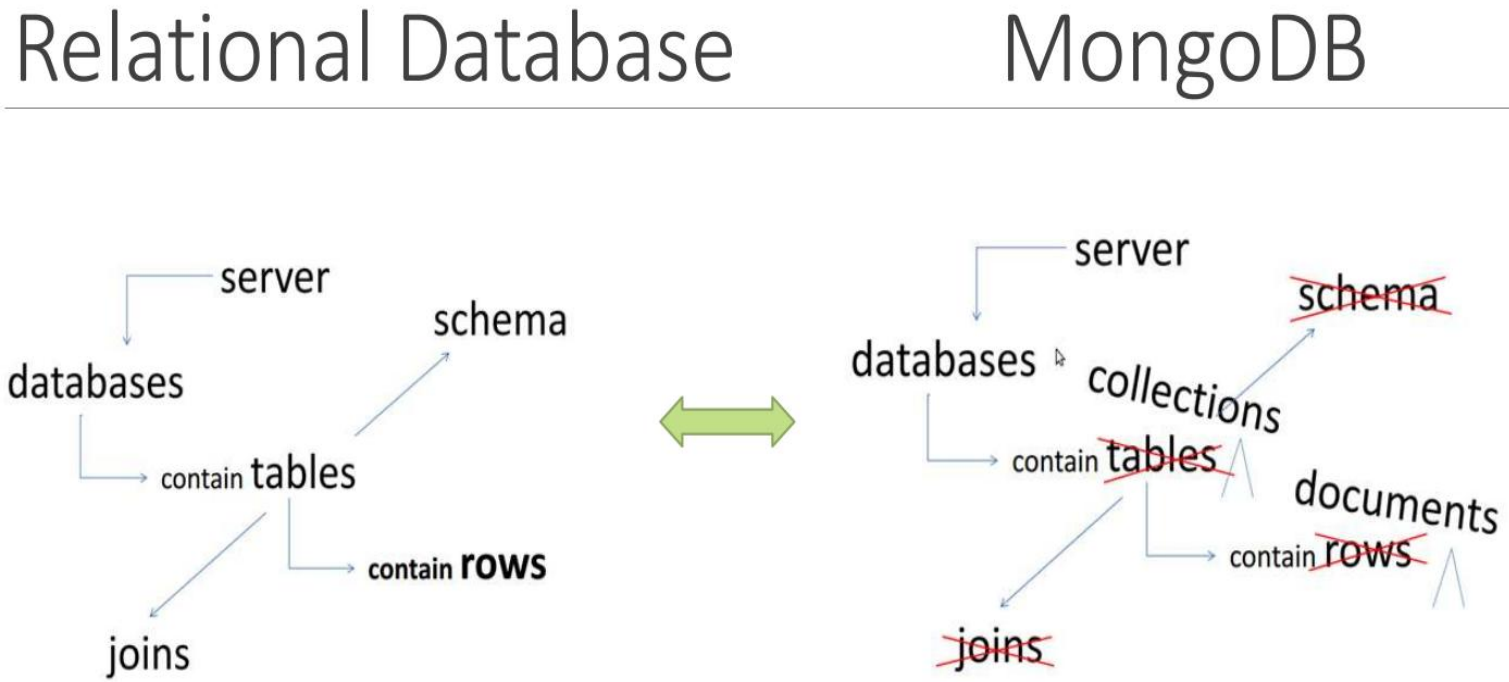
- Simple queries
- Functionality provided applicable to most web applications
- Easy and fast integration of data (No ERD diagram)
- Not well suited for heavy and complex transactions systems

Hierarchical Objects

- A MongoDB instance may have zero or more databases
- A database may have zero or more collections
- A collection may have zero or more documents
- A document may have one or more fields
- MongoDB Indexes function much like their RDBMS counterparts.



Relational vs MongoDB



SQL vs MongoDB

SQL Terms/Concepts	MongoDB Terms/Concepts
database	database
table	collection
row	document
column	field
index	index
Join	Embedded Document
Foreign Key	Reference
Primary Key	_id field is always the primary key
Aggregation (i.e. group by)	aggregation pipeline
Partition	Shard

MongoDB Processes and Configuration

mongod: database instance

mongos: sharding process

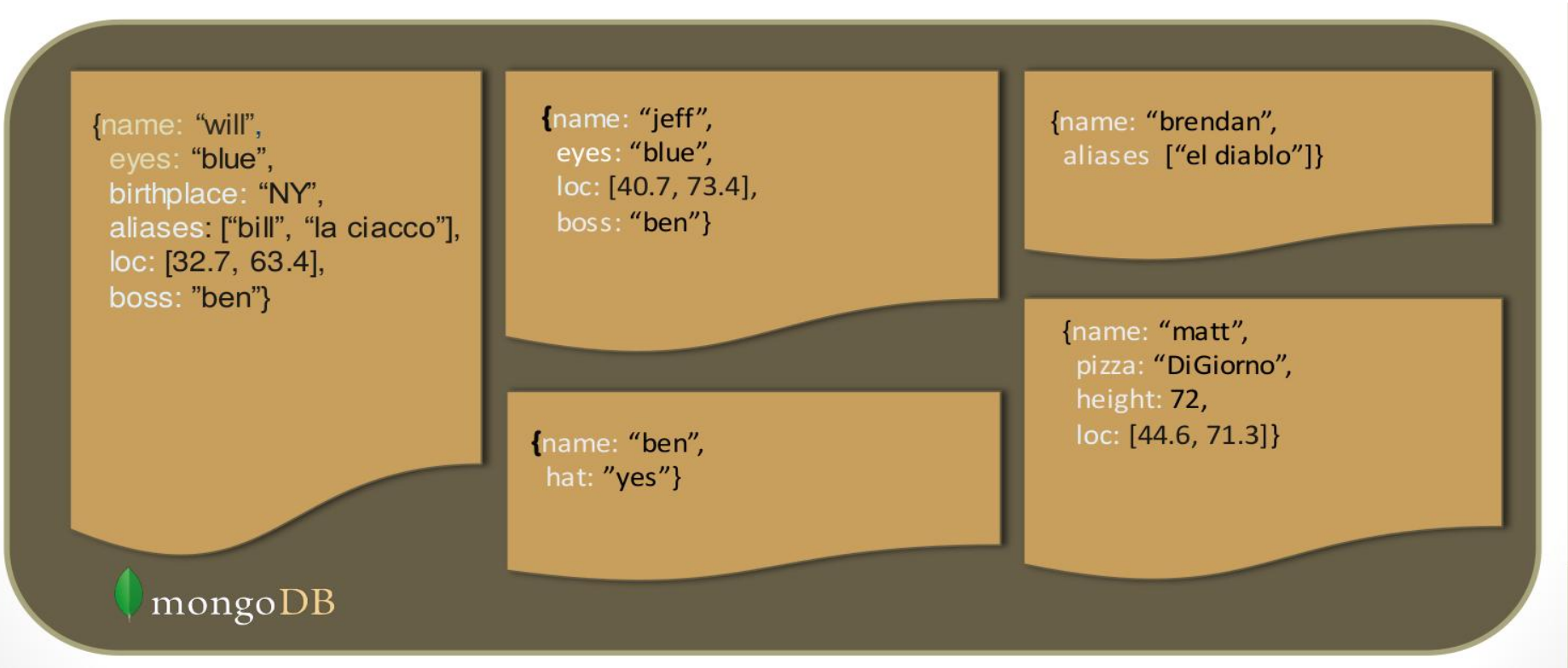
- Analogous to a database router
- Processes all requests
- Decides how many and which mongods should receive the query
- Mongos collates the results, and sends it back to the client

mongo: an interactive shell (a client)

- Fully functional JavaScript environment for use with a MongoDB

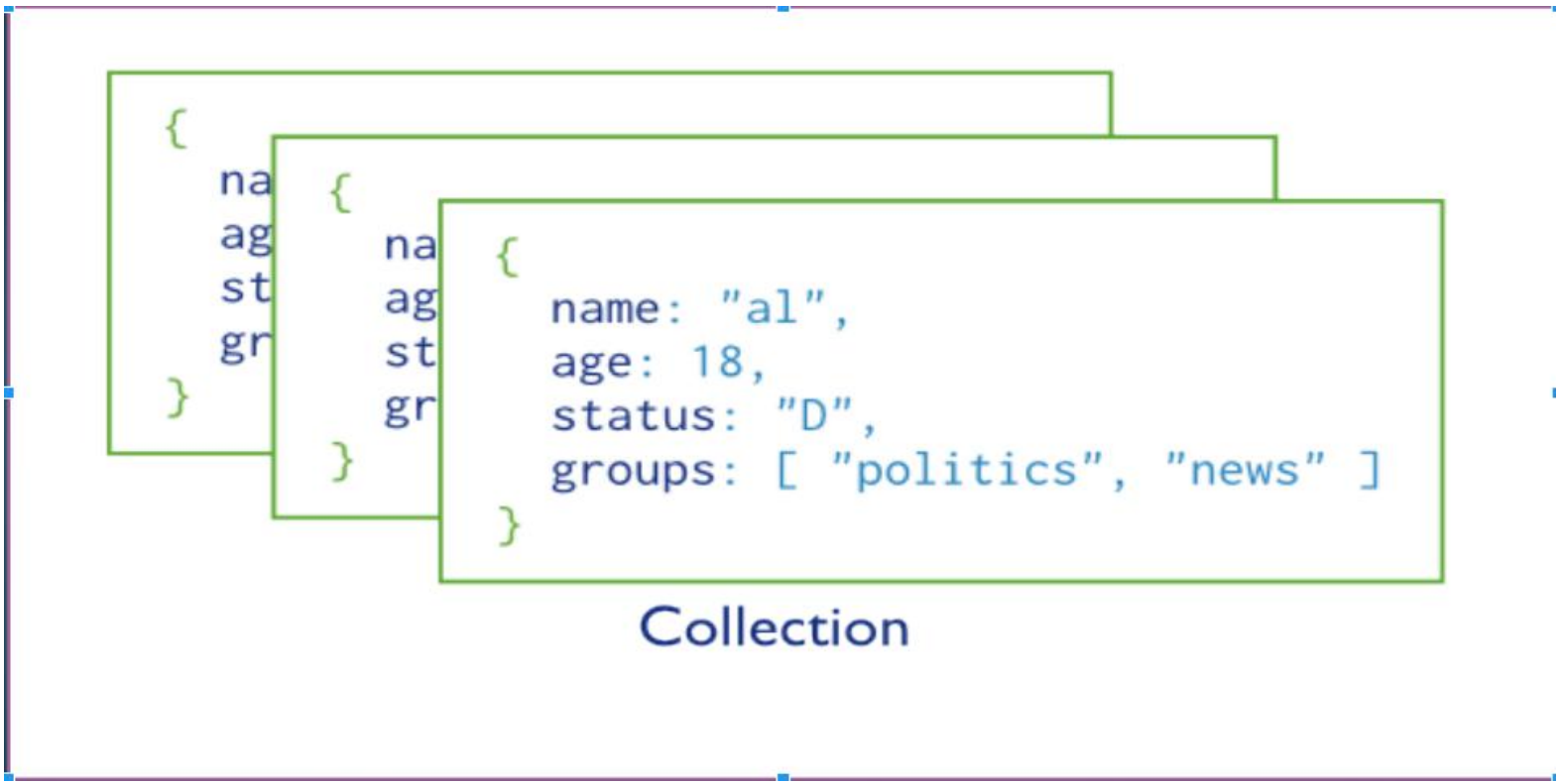
Schema Free

- MongoDB does not need any pre-defined data schema
- Every document in a collection could have different data



Collection

A collection includes documents.



JSON Format

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

Example

"name" : "R2 - D2"

- Data is separated by commas

Example

"name" : "R2 - D2", race : "Droid"

- Curly braces hold objects

Example

{"name" : "R2 - D2", race : "Droid", affiliation : "rebels"}

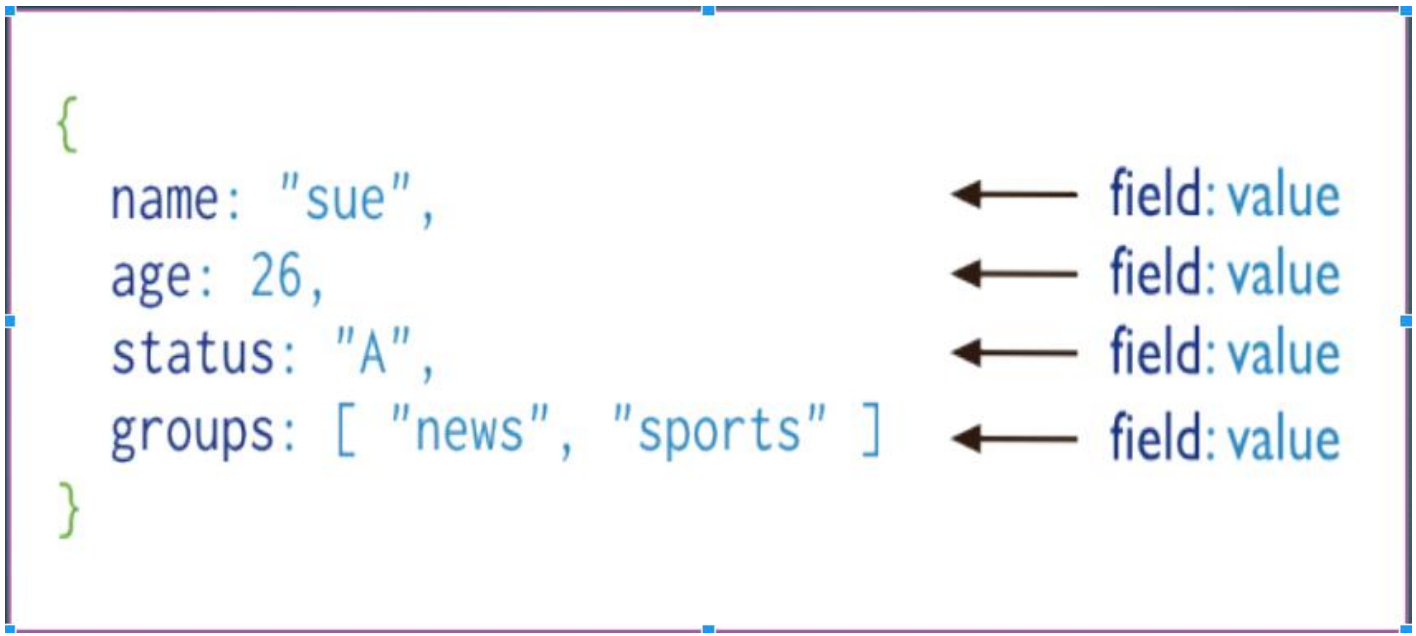
- An array is stored in brackets []

Example

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

JSON Document

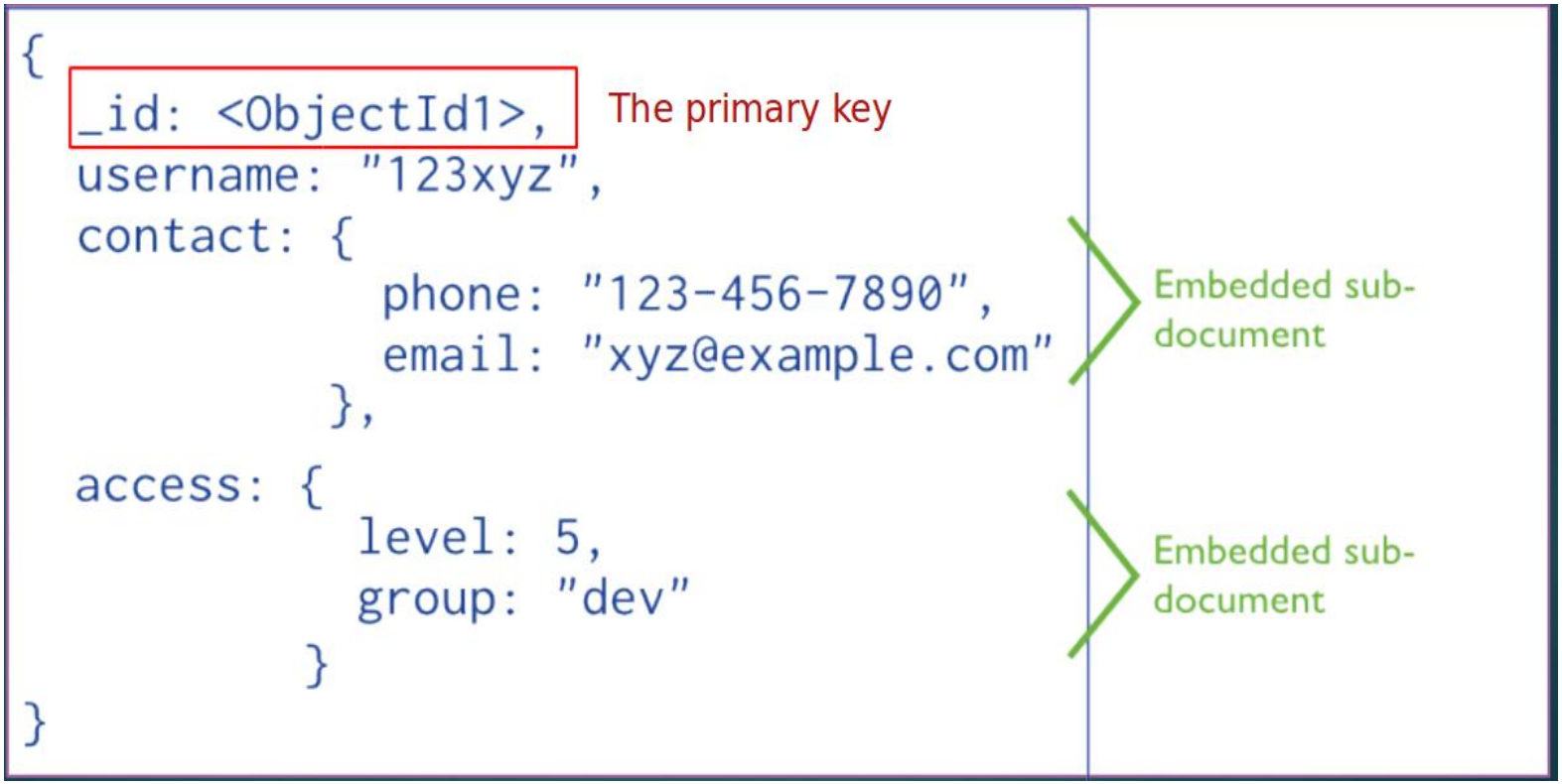
Structure of a JSON document:



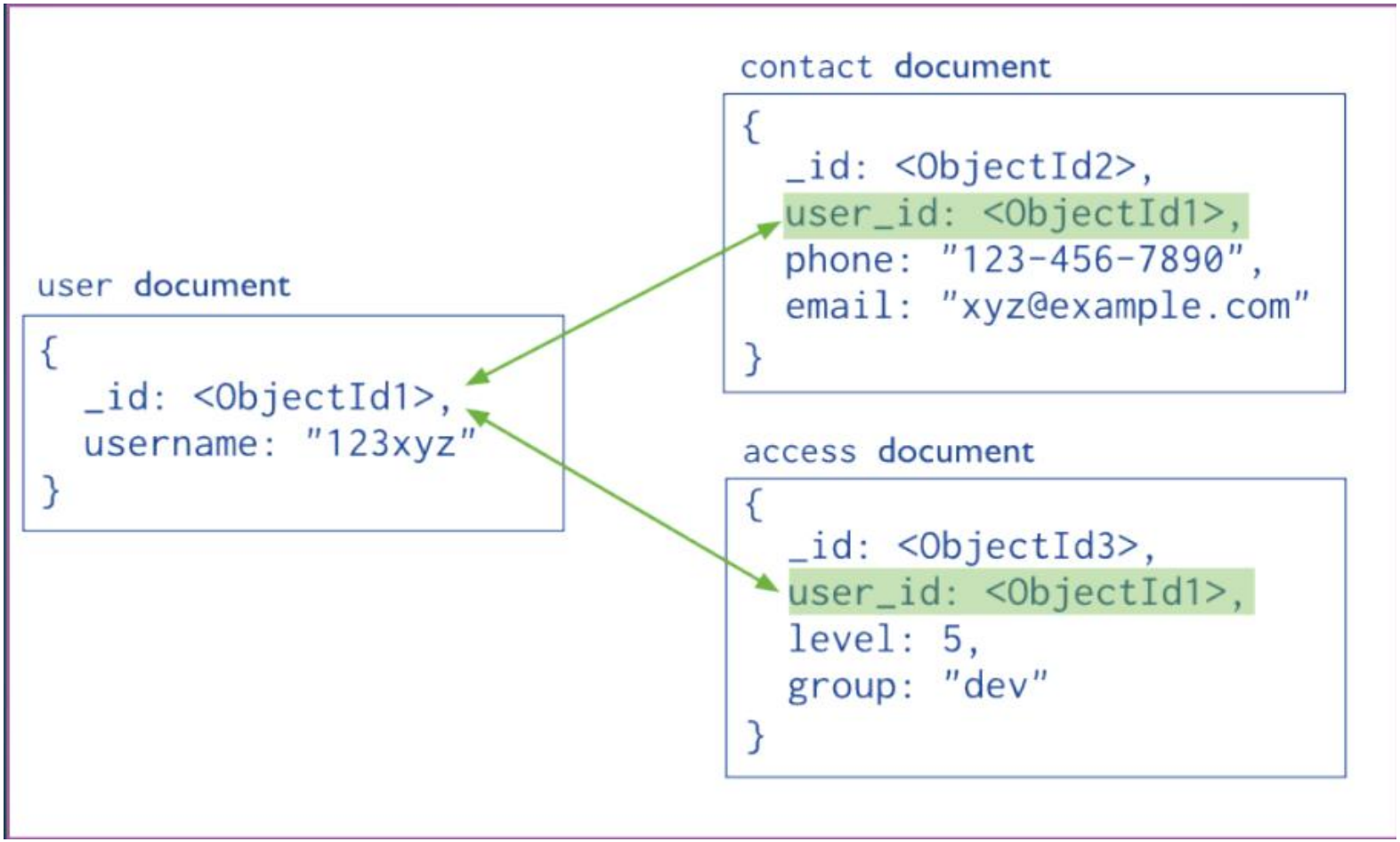
The value of field:

- Native data types
- Arrays
- Other documents

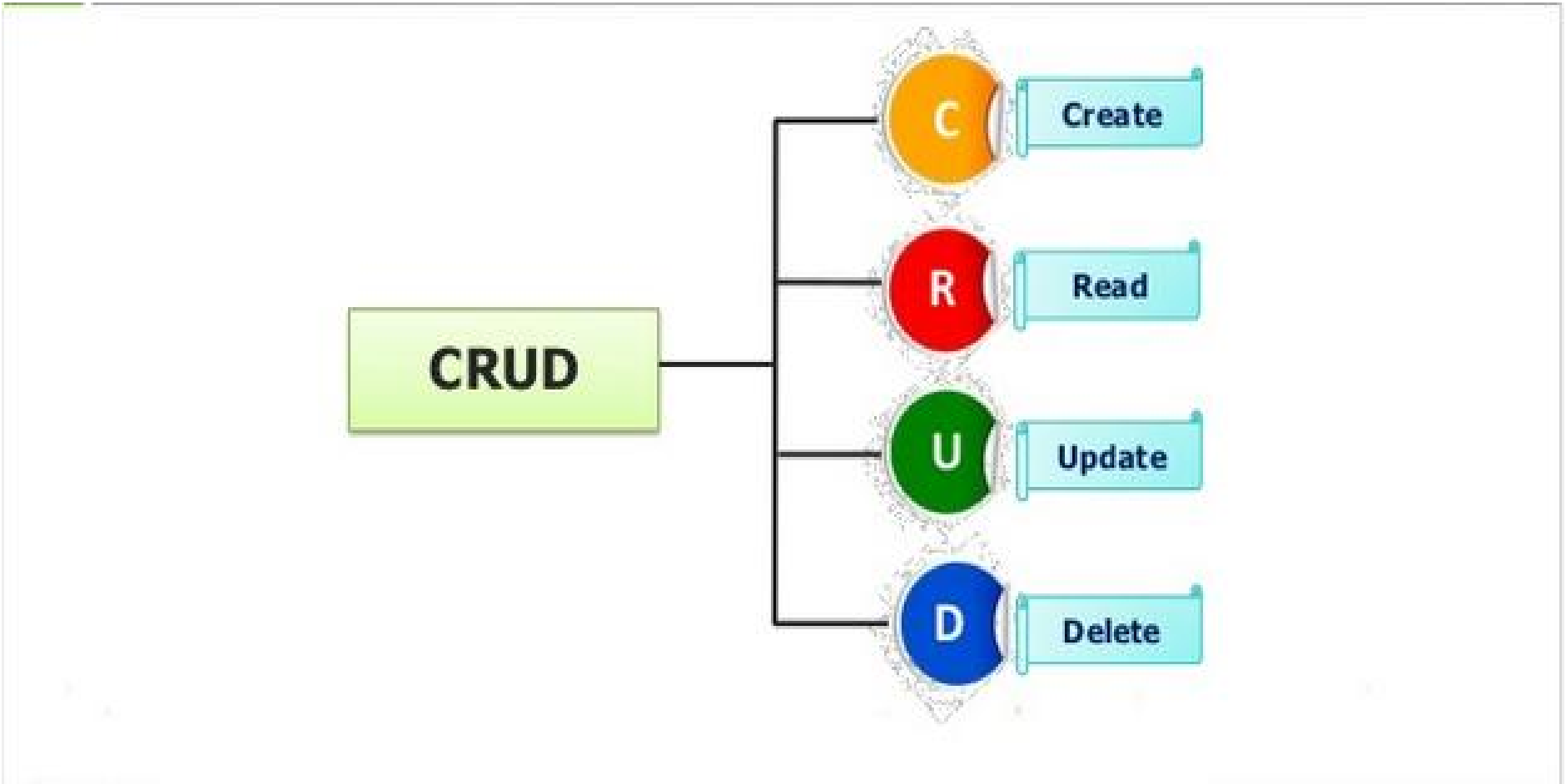
Embedded Documents



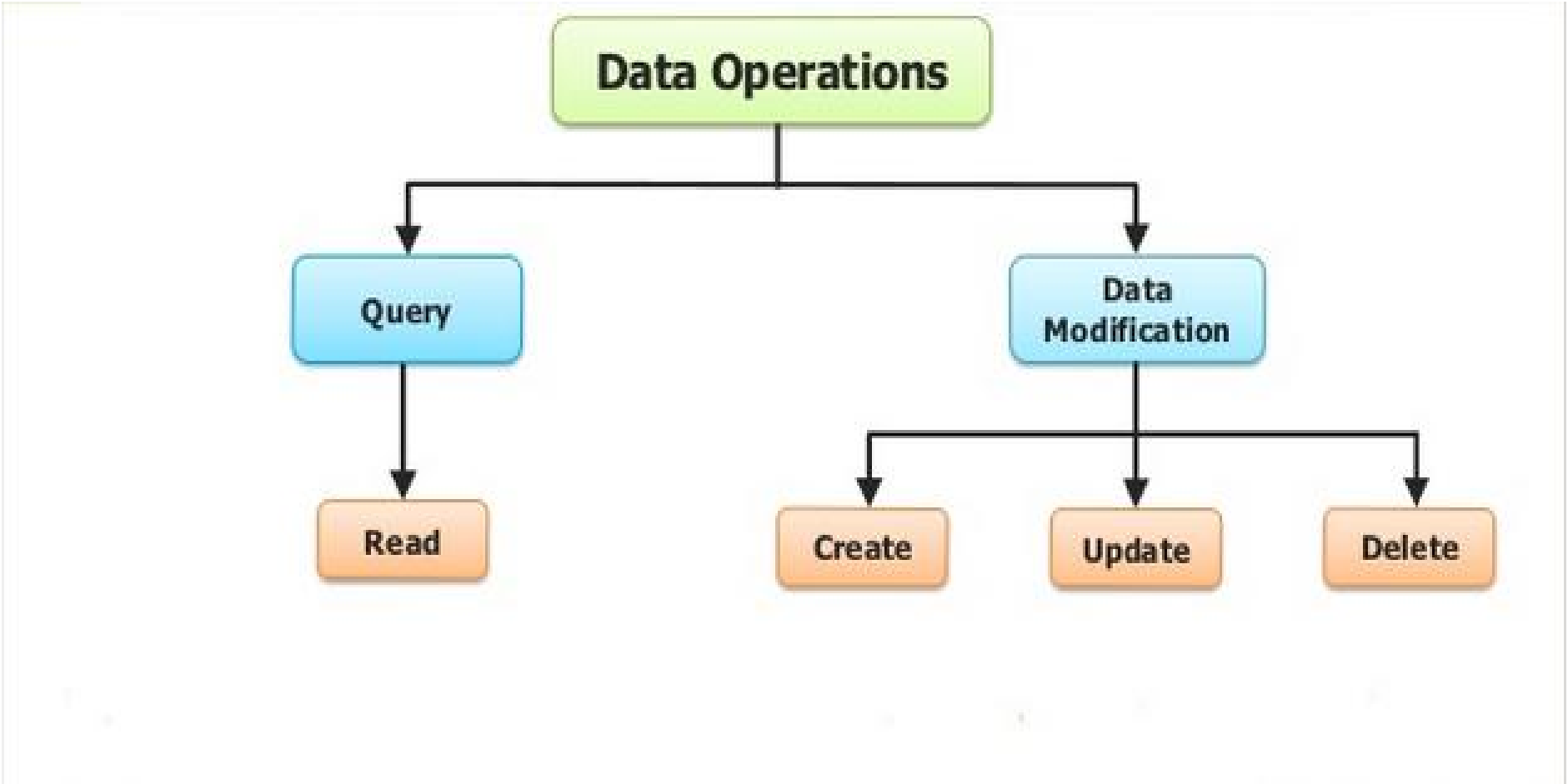
Linking Documents



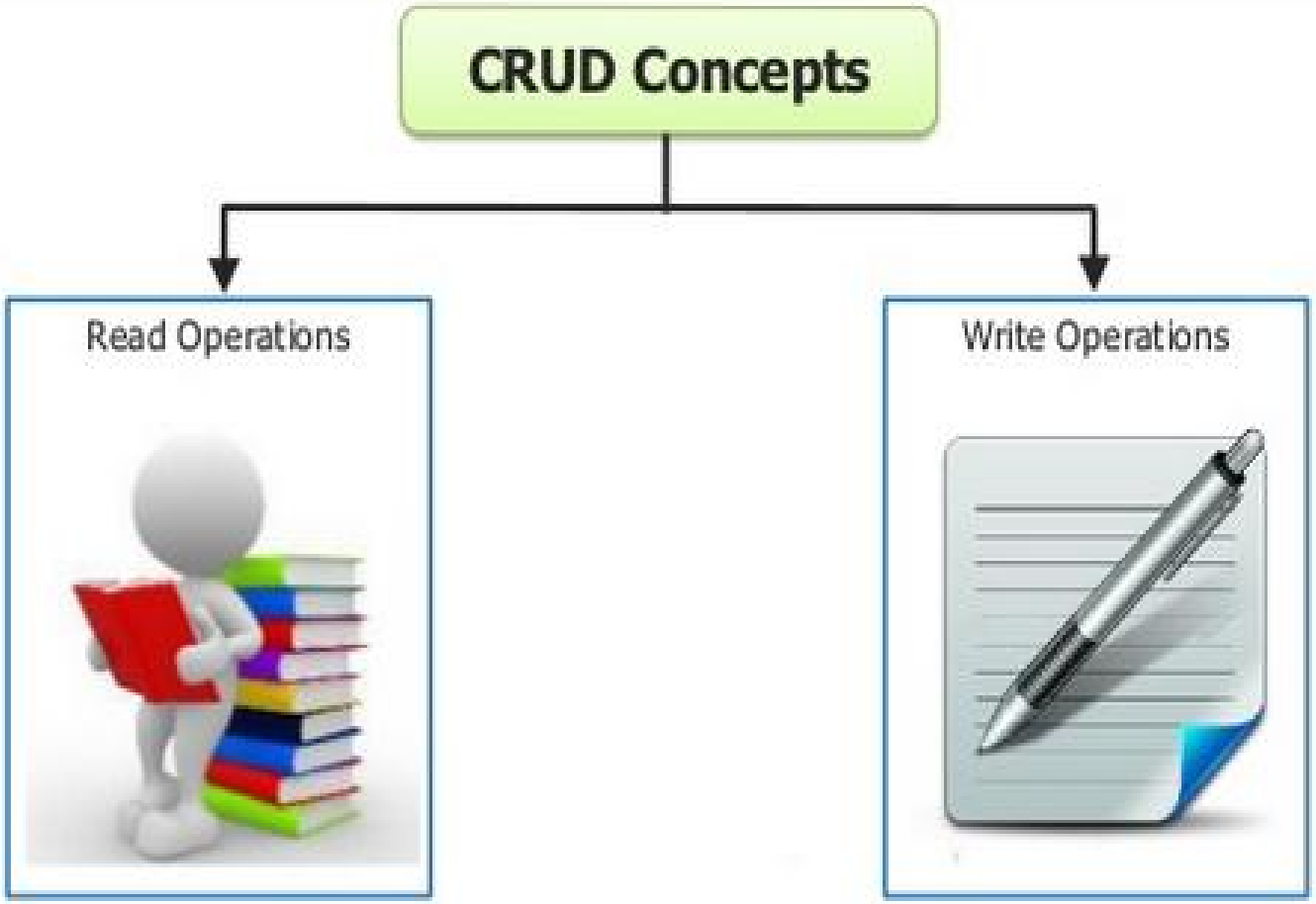
CRUD



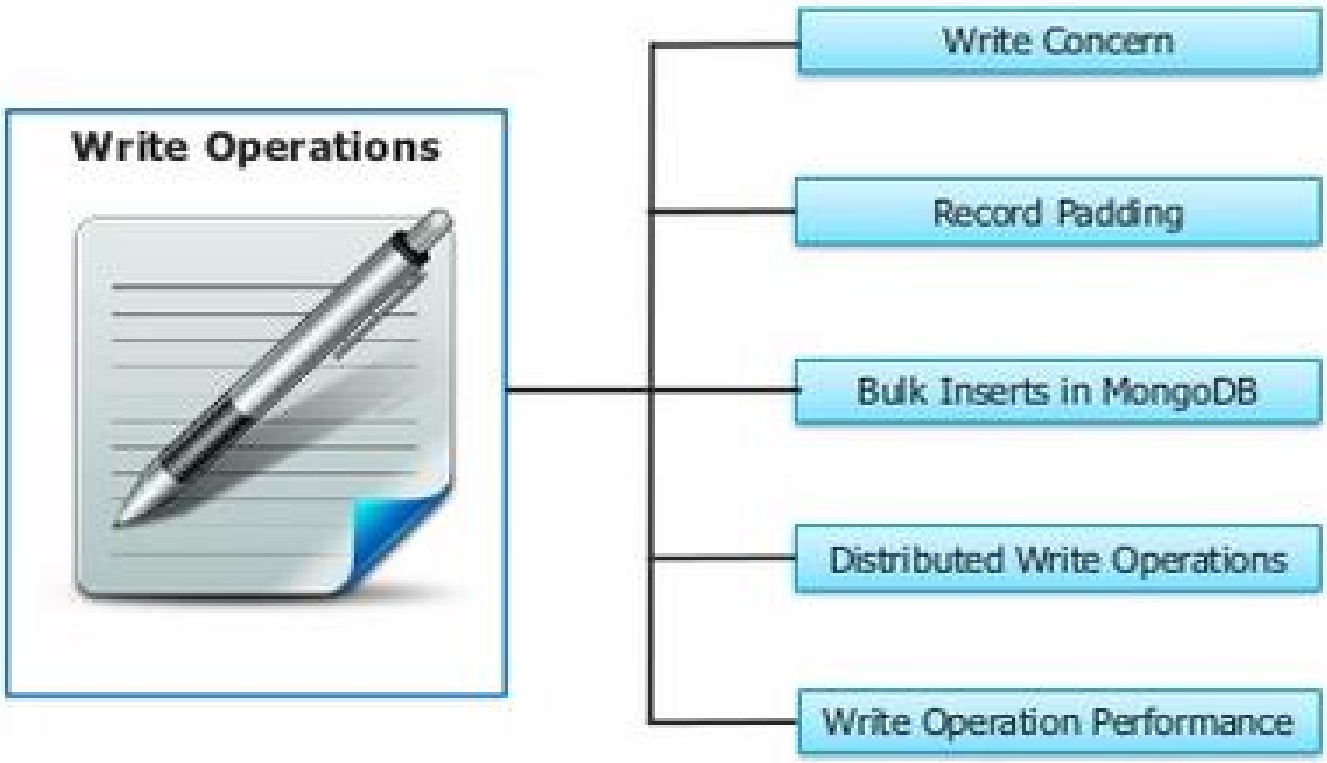
Data Operations



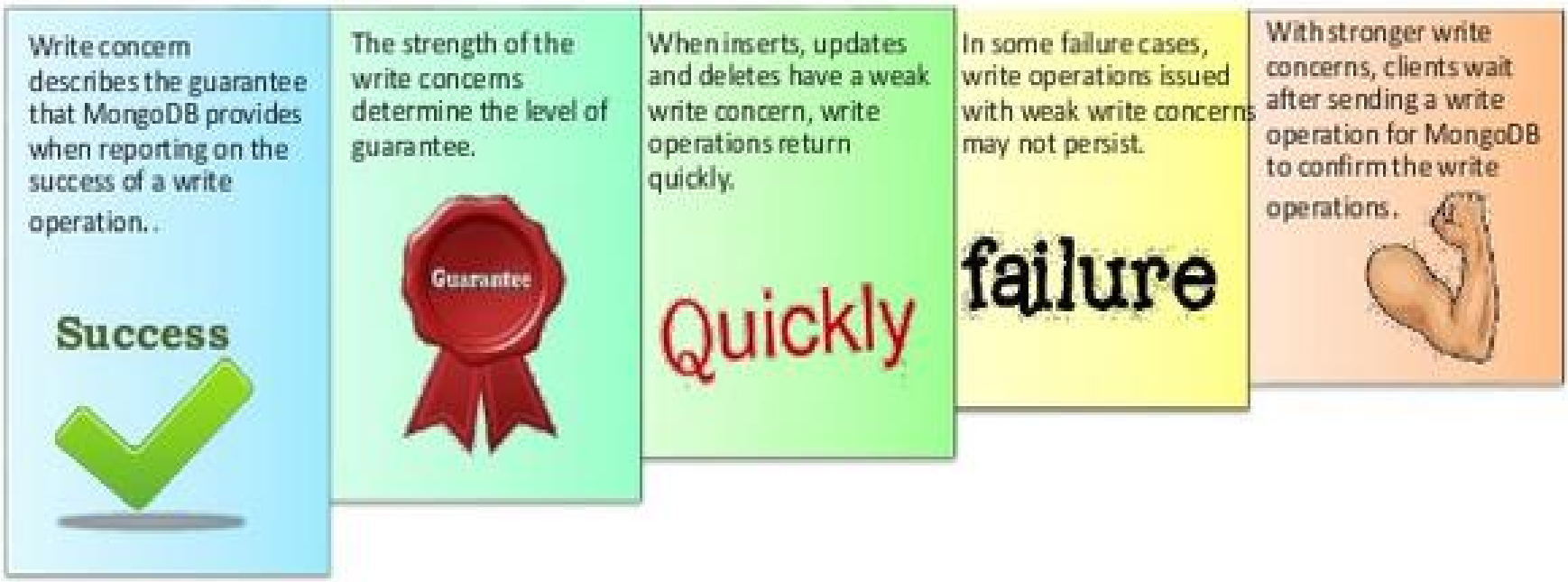
CRUD Concepts



Write Operations



Write Concerns



MongoDB Data Types

String: Used to store the string data. String in mongodb must be UTF-8 valid.	**Arrays:** This type is used to store arrays or list or multiple values into one key.	**Object:** This datatype is used for embedded documents.	**Code:** To store javascript code into document.	**Time Stamp:** For recording when a document has been modified or added.
Integer: For numerical value. Integer can be 32 bit or 64 bit depending upon your server.	**Boolean:** For a Boolean (true/ false) value.	**Double:** For floating point values.	**Regular expression:** To store regular expression.	**Null:** This type is used to store a Null value.
Date: For current date or time in UNIX time format. Can specify your own date time by creating object of Date and passing day, month, year into it.	**Min/ Max keys:** To compare a value against the lowest and highest BSON elements.	**Symbol:** It's generally reserved for languages that use a specific symbol type.	**Object ID:** Store the document's ID.	**Binary data:** To store binay data.

CRUD Syntax and Queries

- ✓ Insert Documents
- ✓ Query Documents
- ✓ Limit Fields to Return from a Query
- ✓ Iterate a Cursor in the mongo Shell
- ✓ Analyze Query Performance
- ✓ Modify Documents
- ✓ Remove Documents
- ✓ Perform Two Phase Commits
- ✓ Create Tailable Cursor
- ✓ Isolate Sequence of Operations
- ✓ Create an Auto-Incrementing Sequence Field
- ✓ Limit Number of Elements in an Array after an Update

Create and Delete Database

Create a database: use database_name

Example

use mydb

Check your currently selected database: db

Check your databases list: show dbs

Delete the selected database: db.dropDatabase()

- Your created database is not present in list. To display database, you need to insert at least one document into it.
- In MongoDB default database is test. If you didn't create any database, then collections will be stored in test database.
- When you delete a database, you need to make sure that you are currently in the selected database to be deleted.

Create, Rename, and Delete a Collection

- To create a collection with the following syntax:
  db.createCollection(name, options)
  where name is the name of a collection to be created, and options are optional.

Example

```
use test

db.createCollection("mycollection")
```

- You can check the created collection by using the command show collections.
- You can drop a collection by using the command db.COLLECTION_NAME.drop()

Example

```
db.mycollection.drop()
```

- Collections can also be renamed:
  db.COLLECTION_NAME.renameCollection("NEW_COLLECTION_NAME")

Example

db.products.renameCollection("store_products")

Insert Document

- To insert data into MongoDB collection, you need to use MongoDB's insert() or save() method.
- The basic syntax of insert() command is as follows: db.COLLECTION_NAME.insert(document)

Example

```
db.mycol.insert({_id: ObjectId(7df78ad8902c),
title: 'MongoDB Overview', description: 'MongoDB
is no sql database', by: 'tutorials point', url:
'http://www.tutorialspoint.com', tags: ['mongodb',
'database', 'NoSQL'], likes: 100})
```

- _id is 12 bytes hexadecimal number unique for every document in a collection. 12 bytes are divided as follows: _id: ObjectId(4 bytes timestamp, 3 bytes machine id, 2 bytes process id, 3 bytes incrementer)

Insert Multiple Documents

To insert multiple documents in a single query, you can pass an array of documents in insert() command.
db.COLLECTION_NAME.insert([document1, document2, . . .])

Example

```
db.post.insert([{title: 'MongoDB Overview', description:
'MongoDB is no sql database', by: 'tutorials
point', url: 'http://www.tutorialspoint.com', tags:
['mongodb', 'database', 'NoSQL'], likes: 100 },
{title: 'NoSQL Database', description: "NoSQL database
doesn't have tables", by: 'tutorials point', url:
'http://www.tutorialspoint.com', tags: ['mongodb',
'database', 'NoSQL'], likes: 20, comments: [ {user:'user1',
message: 'My first comment', dateCreated: new
Date(2013,11,10,2,35), like: 0 }] }])
```

Delete Document

- A group of documents can be removed from the collection with the following command:
  db.COLLECTION_NAME.remove(DELLETION_CRITTERIA) Consider the mycol collection has the following data:
  { "_id" : ObjectId(5983548781331adf45ec5), "title":"MongoDB Overview" }
  { "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview" }
  { "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point Overview" }

Example

```
db.mycol.remove({'title':'MongoDB Overview'})
```

- If there are multiple records and you want to delete only the first record:
  db.COLLECTION_NAME.remove(DELETION_CRITERIA,1)
- If you don't specify deletion criteria, then MongoDB will delete whole documents from the collection.

Example

```
db.mycol.remove({})
```

Update Document

- All updates require at least two arguments. The first specifies which documents to update, and the second defines how the selected documents should be modified.
- One type of update involves applying modification operations to a document or documents.
  - Passing a document with the $set operator: db.COLLECTION_NAME.update(SELECTION_CRITERIA,{$set : {UPDATE_DATA}})

Example

db.mycol.update({'title':'MongoDB Overview'},{$set:{'title':'New MongoDB Tutorial'}})

- To update multiple documents, you need to set a parameter 'multi' to true: db.COLLECTION_NAME.update(SELECTION_CRITERIA,{$set : {UPDATE_DATA}},{multi:true})

Example

db.mycol.update({'title':'MongoDB Overview'},{$set:{'title':'New MongoDB Tutorial'}},{multi:true})

Replace or Update Document

- Be sure to use the $set operator if you intend to add or set fields rather than to replace the entire document.

Example

Add an username into a record:

```
db.users.update({country:'Canada'},{$set:{username:'Smith'}})
```

- If you later decide that the country stored in the profile is no longer needed, the value can be removed using the $unset operator

Example

```
db.users.update({username:'Smith'},{$unset:{country:1}})
```

Be aware that the 1 in the {country:1} field means a boolean flag TRUE.

Update/Add Complex Data

Let's suppose that, in addition to storing profile information, your users can store lists of their favorite things.

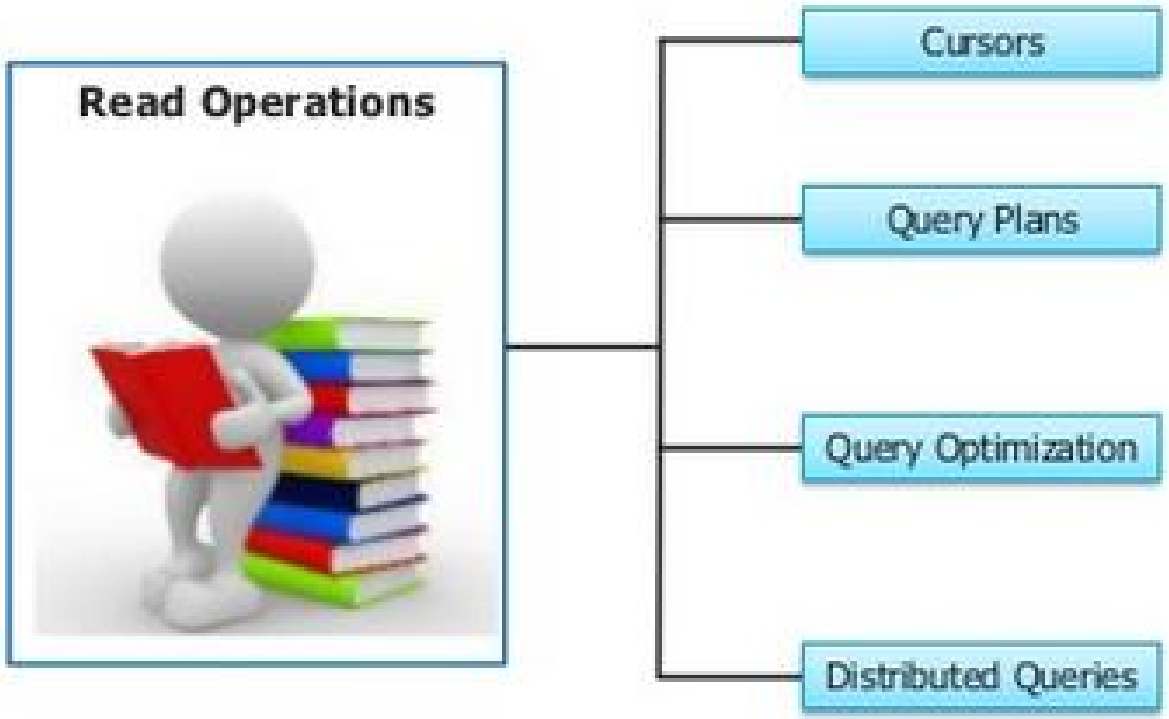
```
{username: "smith", favorites: {food:["apple","cherries"],
sports:["bungee jumping", "sky diving"] }}
```

The favorites key points to an object containing two other keys, which point to lists of favorite food and sports. Use $set operator to add a sub-document.

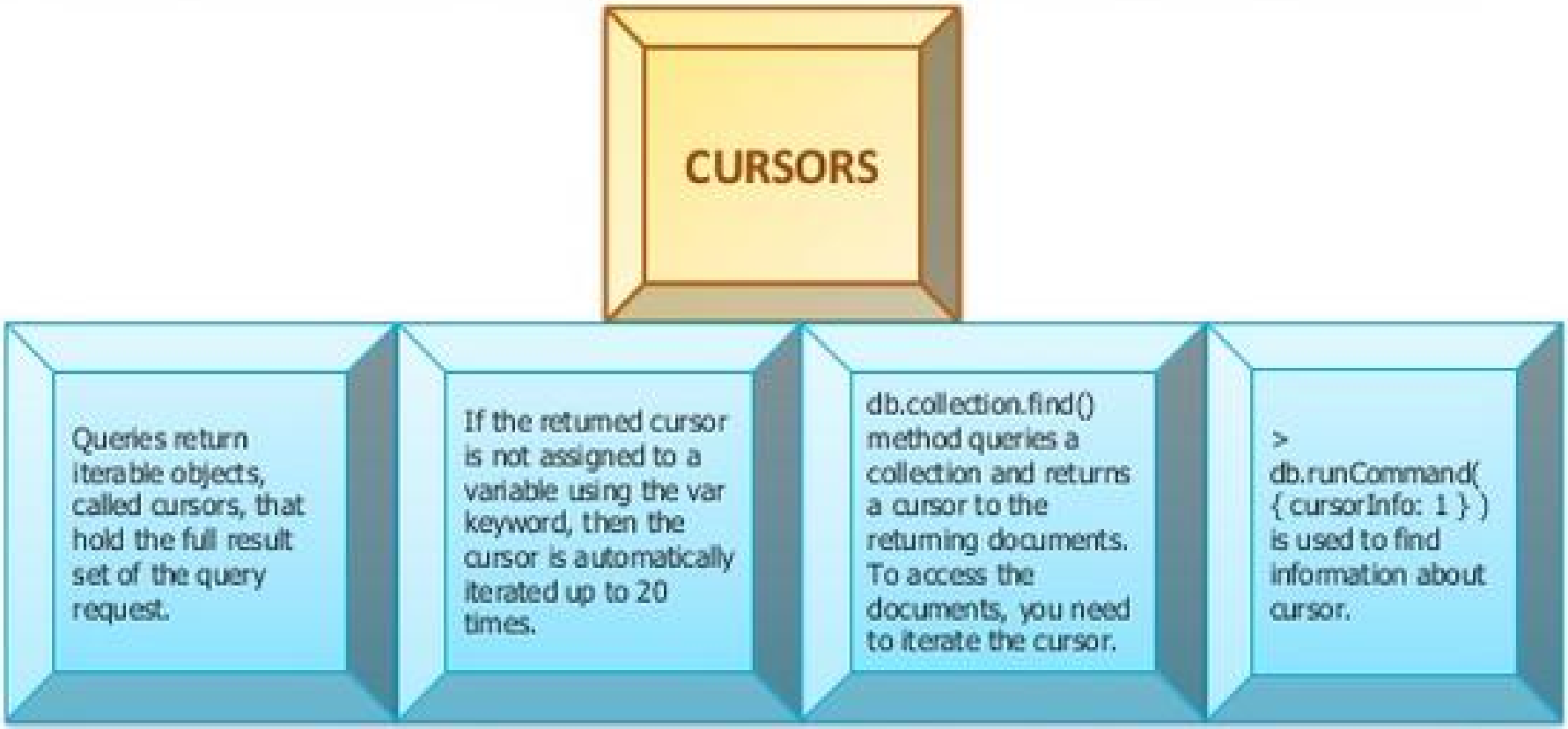
Example

db.users.update({username:'Smith'},{$set:{favorites:{
food:["apple","cherries"], sports:["bungee jumping", "sky diving"]}}})

Read Operations



Cursors


CURSORS
Queries return iterable objects, called cursors, that hold the full result set of the query request.
If the returned cursor is not assigned to a variable using the var keyword, then the cursor is automatically iterated up to 20 times.
db.collection.find() method queries a collection and returns a cursor to the returning documents. To access the documents, you need to iterate the cursor.
> db.runCommand({ cursorInfo: 1 }) is used to find information about cursor.

Cursor Information

```
> db.runCommand( { cursorInfo: 1 } )
```

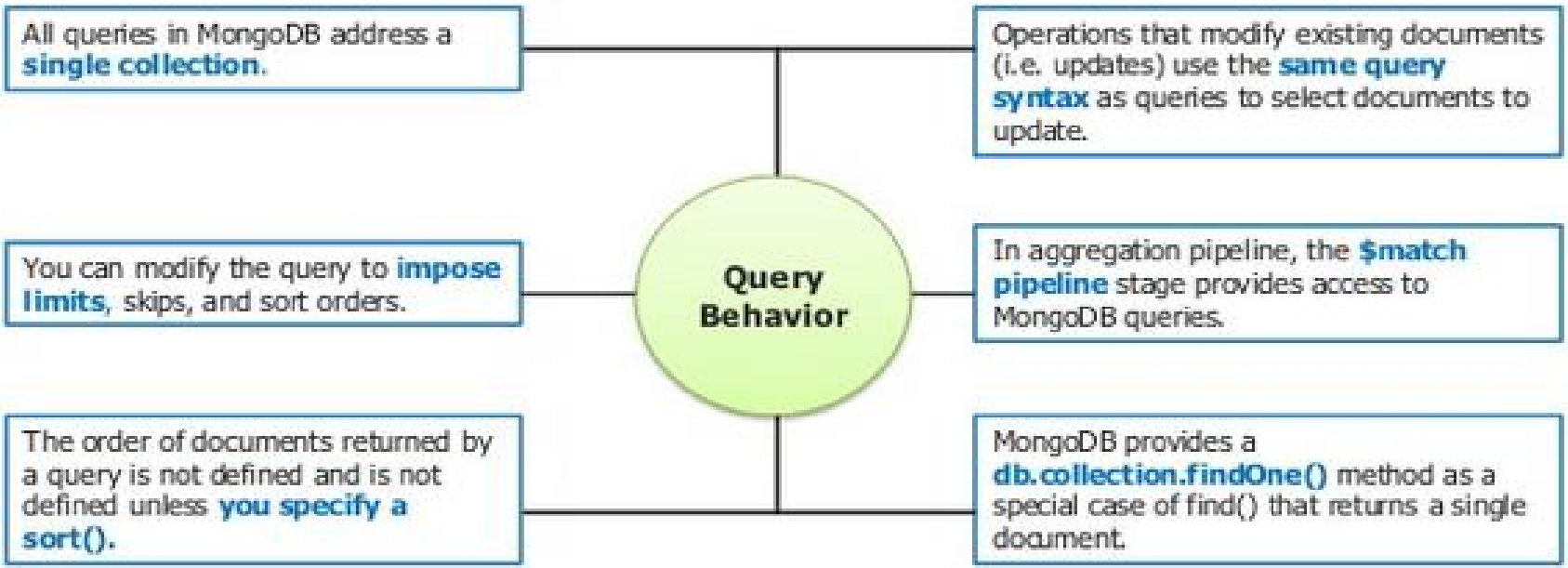
```
> db.runCommand< { cursor Info : 1 } >
{ " totalOpen" : 0, " client Cursors_size": 0, "timedout" : 0, "ok" :1}
>
```

```
> db.runCommand( { cursorInfo: 1 } )
```

```
> db.runCommand< { cursor Info : 1 } >
{ " totalOpen" : 1, " client Cursors_size": 1, "timedout" : 0, "ok" :1}
>
```

Query Optimization

Query Behavior



Query All Documents

- The find command returns a cursor to the returning documents.
- Display all the documents in a non-structured way: db.COLLECTION_NAME.find()

Example

```
db.mycol.find()
```

- To display the results in a formatted way, you can use pretty() method.

Example

```
db.mycol.find().pretty()
```

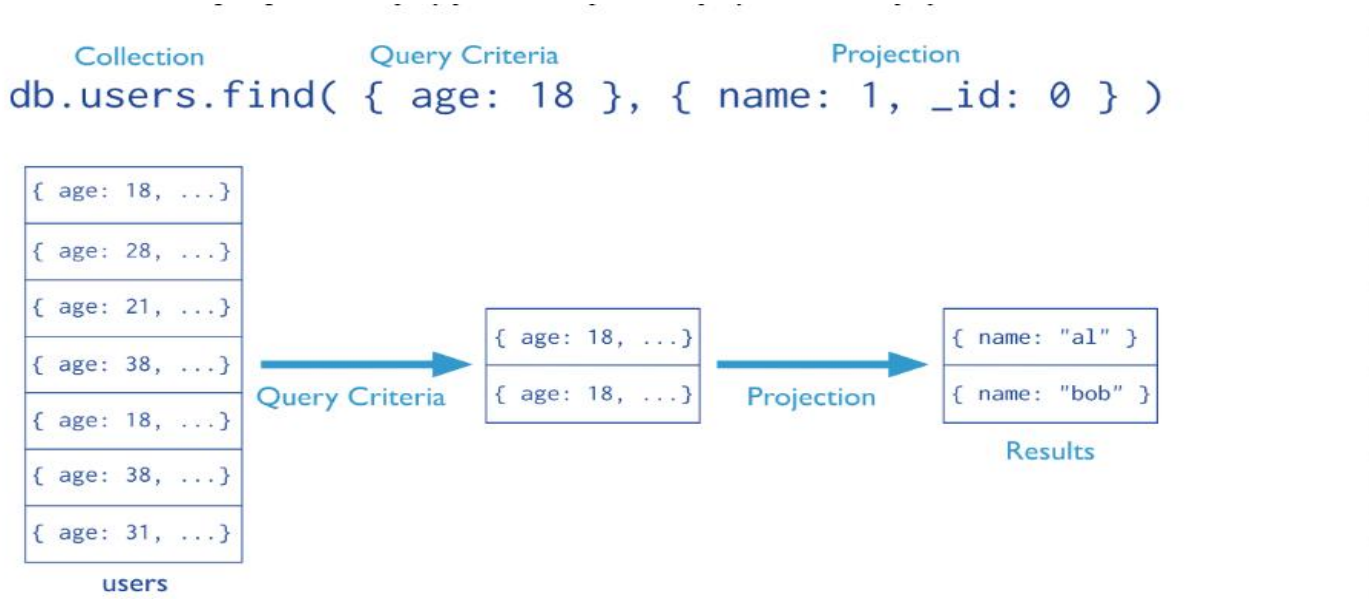
- Apart from find() method, there is findOne() method, that returns only one document.
- To count the number of documents in one collection, use count() function.

Example

```
db.mycol.count()
```

Projection

- Projections, which are the second argument to the find() method, may either specify a list of fields to return or list fields to exclude in the result documents.
- To limit this, you need to set a list of fields with value 1 or 0. 1 is used to show the field while 0 is used to hide the fields.



In the diagram, the query selects from the users collection. The criteria matches the documents that have age equal to 18. Then the projection specifies that only the name field should return in the matching documents.

Projection Examples

- Exclude One Field From a Result Set

Example

db.records.find({ "user_id": {$lt: 42}}, {"history": 0})
This query selects documents in the records collection that match the condition {"user_id":{$lt: 42}}, and uses the projection {"history": 0} to exclude the history field from the documents in the result set.

- Return Two fields and the _id Field

Example

db.records.find({ "user_id": {$lt: 42}}, {"name": 1, "email": 1})

- Return Two Fields and Exclude _id

Example

db.records.find({ "user_id": {$lt: 42}}, {"name":1, "email":1, "_id":0})

Projection Behavior

MongoDB projections have the following properties:

- By default, the _id field is included in the results. To suppress the _id field from the result set, specify _id : 0 in the projection document
- For fields that contain arrays, MongoDB provides the following projection operators: $elemMatch, $slice, and $.
- For related projection functionality in the aggregation pipeline, use the $project pipeline stage.

Limit Documents

To paginate the review documents in MongoDB, you need to use limit() method. The method accepts one number type argument, which is the number of documents that you want to be displayed. The basic syntax of limit() method is as follows:
db.COLLECTION_NAME.find().limit(NUMBER)

Consider the mycol collection has the following data:
{ "_id" : ObjectId(5983548781331adf45ec5), "title":"MongoDB Overview" }
{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview" }
{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point Overview" }

Example

Display only two documents while querying the document.
db.mycol.find({},{"title":1,_id:0}).limit(2)

Skip Documents

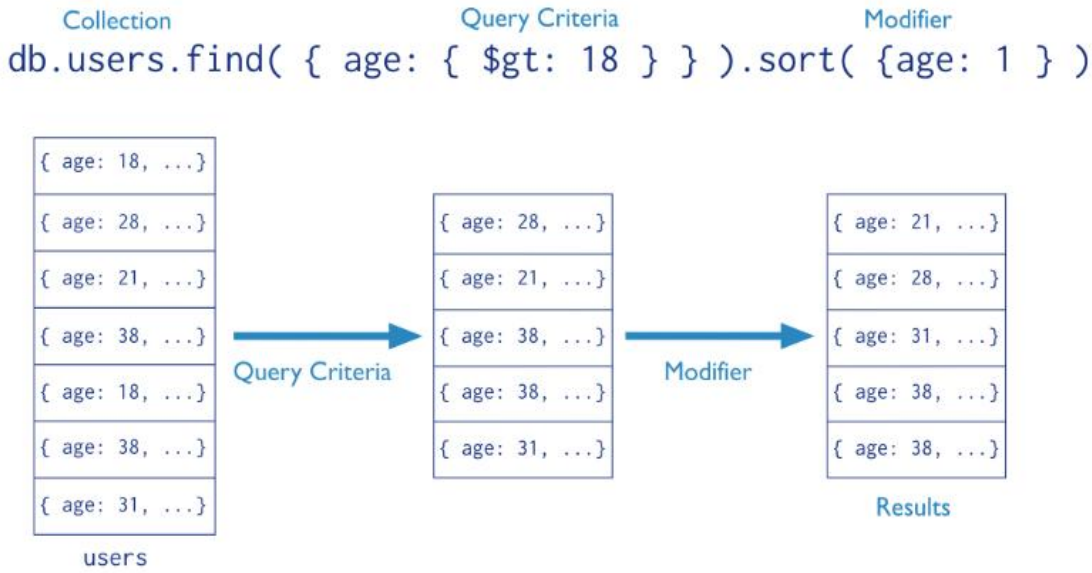
There is one more method skip() which also accepts number type argument and is used to skip the number of documents. The basic syntax of skip() method is as follows:
db.COLLECTION_NAME.find().skip(NUMBER)

Example

Display only the second document:
db.mycol.find(,"title":1,_id:0).limit(1).skip(1)
Display the rest documents except the first two:
db.mycol.find(,"title":1,_id:0).skip(2)

Sort Documents

To sort documents in MongoDB, you need to use sort() method. The method accepts a document containing a list of fields along with their sorting order. To specify sorting order 1 and -1 are used. 1 is used for ascending order while -1 is used for descending order. The basic syntax of sort() method is as follows: db.COLLECTION_NAME.find().sort(KEY:1) If you don't specify the sorting preference, then sort() method will display the documents in ascending order.



Example

Display the documents sorted by title in the descending order:
db.mycol.find({}, {"title" : 1, _id : 0}).sort({"title" : -1})

Specify Selective Condition

Operator	Syntax	Example
Equality	{key:value}	{"by":"tutorials"}
Less Than	{key:{$lt:value}}	{"likes":{$lt:50}}
Less Than Equals	{key:{$lte:value}}	{"likes":{$lte:50}}
Greater Than	{key:{$gt:value}}	{"likes":{$gt:50}}
Greater Than Equals	{key:{$gte:value}}	{"likes":{$gte:50}}
Not Equals	{key:{$ne:value}}	{"likes":{$ne:50}}
IN	{key:{$in:[val1, val2, ...]}}	{"likes":{$in:[1,2]}}
Not In	{key:{$nin:[val1, val2, ...]}}	{"likes":{$nin:[1,2]}}
Exists	{key:{$exists:true}}	{city:{$exists:true}}
All	{key:{$all:[val1, val2, ...]}}	{"likes":{$all:[1,2]}}
Logical OR	{$or:[{key1: value1}, {key2:value2}]	{$or:[{qty:{$gt:100}}, {price:{$lt:9.95}}]}
Logical AND	{$and:[{key1: value1}, {key2:value2}]	{$and:[{qty:{$gt:100}}, {price:{$lt:9.95}}]}
Logical NOT	{$not:{key: value}}	{$not:{qty:{$gt:100}}}
Logical NOR	{$nor:[{key1: value1}, {key2:value2}]	{$nor:[{qty:{$gt:100}}, {price:{$lt:9.95}}]}

Simple Query, Range Query

In Mongodb, all text string matches are case sensitive.

Example

db.inventory.find({ type: "snacks" })
db.numbers.find({num : 500})
Find me all users such that the first name is Smith and was born in 1975:
db.users.find({'first_name' : "Smith", birth_year : 1975})

Note that whenever you pass more than one key-value pair, both must match; the query conditions function as a Boolean AND.

Example

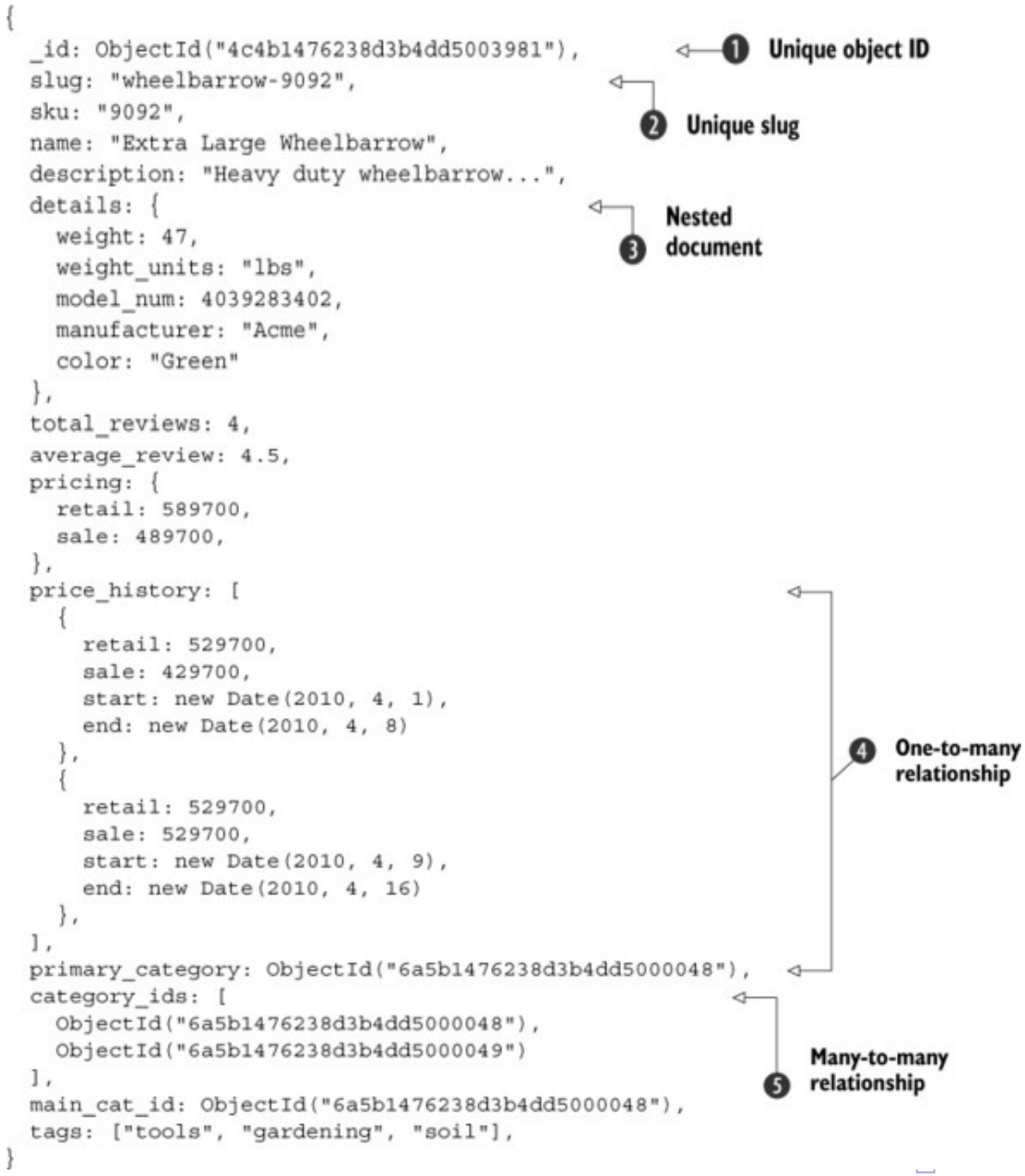
db.numbers.find({num : {$gt : 19995}})
db.numbers.find({num : {$gt : 20, $lt : 25}})
db.users.find({"zip" : {$gt: 10019, $lt : 10040}})
db.items.find({"value" : {$gte : "a"}})

Set Operators

Operator	Description
$in	Matches if any of the arguments are in the reference set
$all	Matches if all of the arguments are in the reference set and is used in documents that contain arrays
$nin	Matches if none of the arguments are in the reference set

- $in returns a document if any of the given values matches the search key.
- $nin returns a document only when none of the given elements matches.
- $all matches if every given element matches the search key.

A Sample Product Document



Set Operators Examples

Example

Selects all documents in the inventory collection where the value of the type field is either food or snacks: db.inventory.find({type:{$in:['food','snacks']}})
Find me all products of which the category is lawnmowers or hand tools or work clothing.
db.products.find({"main_cat_id" : {$in : ["lawnmower", "hand tools", "work clothing"]}})

Example

Find all products that are neither black nor blue:
db.products.find({"details.color" : {$nin : ["black", "blue"]}})

Example

Find all products tagged as gift and garden:
db.products.find({"tags" : {$all : ["gift", "garden"]}})

Logical Operators Examples

Example

Selects all documents where the type field has the value food and the value of the price field is less than 9.95:
db.inventory.find({$and:[{type:'food'},{price:{$lt:9.95}}]})

Example

Selects all documents in the collection where the field qty has a value greater than 100 or the value of the price field is less than 9.95:
db.inventory.find({$or:[{qty:{$gt:100}},{price:{$lt:9.95}}]})

Example

selects all documents in the collection where the value of the type field is food and either the qty has a value greater than 100 or the value of the price field is less than 9.95:
db.inventory.find({type:'food',{$or:[{qty:{$gt:100}},{price:{$lt:9.95}}]}})

Logical Operator Examples

Example

Find all products manufactured by Acme that aren't tagged with gardening:
db.products.find({"manufacturer" : "Acme", "tags" : {$ne : "gardening"}})

Example

db.users.find({"age" : {$not : {$lte : 30}}})

- $not includes documents that aren't evaluated by the given expression. This query returns documents where age greater than 30, it also returns documents with no age field.
- You should use $and operator only when you cannot express an AND in a simpler way.

Query for Null or Missing Fields

- The {item:null} query matches documents that either contain the item field whose value is null or that do not contain the item key.

Example

```
db.inventory.find({item: null})
```

- You need a way to query for documents containing a particular key. The {item : {$exists: false}} query matches documents that do not contain the item key. Even if a key exists, it can still be set to null value.

Example

```
db.inventory.find({item : {$exists: false}})
```

Partial Match Queries

Mongodb allows you to query using regular expressions:

Prefix Search: it starts with a caret ($\wedge$) or a left anchor (/), followed by a string of simple symbols

Example

/$\wedge$Ba/ means all user names begins with Ba

```
db.users.find({"last_name" : /^Ba/})
```

Suffix Search: it starts with a left anchor (/) and a string of simple symbols, followed by a $ and a left anchor.

Example

/Ba$/ means all user names ends with Ba

```
db.users.find({"last_name" : /Ba$/})
```

Pattern Search: it starts with a /.* and a string of simple symbold, followed by a .*/

Example

/.*Ba.*/ means all user names contain Ba

```
db.users.find({"last_name" : /.*Ba.*/})
```

Embedded Documents

When the field holds an embedded document, a query can either specify an exact match on the embedded document or specify a match by individual fields in the embedded document using the dot notation.

- Dot notation is to match by specific fields in an embedded document. When the field holds an array of embedded documents, then to match the field, concatenate the name of the field that contains the array, with a dot (.) and the name of the field in the embedded document: {array_field_name.embedded_document_field : value }

Example

Find all the products manufactured by Acme:
db.products.find({"details.manufacturer" : "Acme"})

- Such queries can be specified arbitrarily deep.

Example

db.products.find({details.manufacturer.id : 432})

Arrays

Operator	Description
$elemMatch	Matches if all supplied terms are in the same subdocument
$size	Matches if the size of array subdocument is the same as the supplied literal value

- To specify multiple criteria on an array of embedded documents such that AT LEAST ONE embedded document satisfies all the specified criteria, $elemMatch operator is used.

Example

```
db.schoolinfo.find ({'students': { $elemMatch:{ 'score' :{$gt : 85},
'section': 'A'}}}, {_id:0, 'name':1})
```

- $size allows you to query for an array by its size.

Example

```
db.users.find({"address" : {$size : 3}})
```

Analyze Query Performance

The $explain operator provides information on the query, indexes used in a query and other statistics. It is very useful when analyzing how well your indexes are optimized.
The db.collection_name.find(search_condition).explain("executionStats") methods provide statistics about the performance of a query.

Example

```
db.inventory.find({quantity:{$gte:100,$lte:200}}).explain("executionStats")
```

It returns an explain plan, which contains two important information:

1. executionStats.nReturned: indicates that the number of documents that the query matches and returns.
2. executionStats.totalDocsExamined: indicates that the number of documents that MongoDB had to scan (i.e. all documents in the collection) to find the matching documents.

Indexing

Indexes support the efficient resolution of queries. Without indexes, MongoDB must scan every document of a collection to select those documents that match the query statement. Indexes are special data structures, that store a small portion of the data set in an easy-to-traverse form. The index stores the value of a specific key or set of keys, ordered by the value of the key as specified in the index.
To create an index you need to use createIndex() method. The basic syntax of createIndex() method is as follows: db.COLLECTION_NAME.createIndex({KEY:1})
Here key is the name of the key on which you want to create index and 1 is for ascending order. To create index in descending order you need to use -1.

Example

```
db.mycol.createIndex({"title" : 1})
db.mycol.ensureIndex({"title" : 1,"description" : -1})
```

You can verify that the index has been created by calling the getIndexes() method:
db.collection_name.getIndexes()

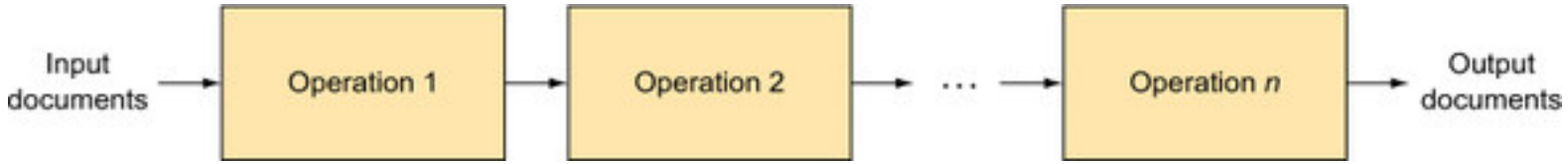
Aggregation

Aggregations operations process data records and return computed results. Aggregation operations group values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result. In SQL count(*) and with group by is an equivalent of mongodb aggregation. It allows you to transform and combine data from multiple documents to generate new information not available in any single document. For example, you might use the aggregation framework to determine sales by month, sales by product, or order totals by user. MongoDB provides three ways to perform aggregation:

- aggregation pipeline
- map-reduce function
- single purpose aggregation methods

Aggregation Pipeline

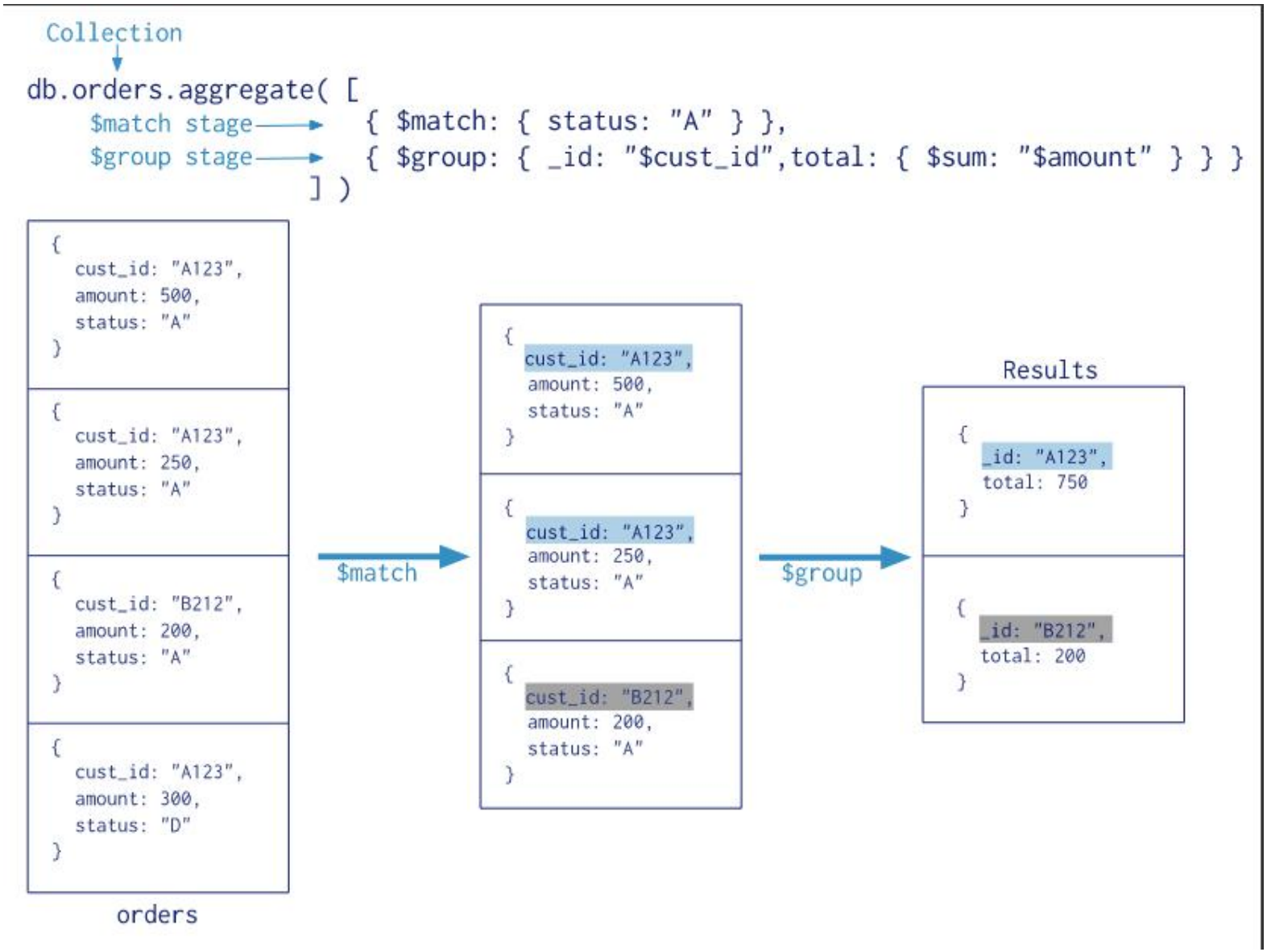
In aggregation pipeline, the output from each step in the pipeline provides input to the next step. Each step executes a single operation on the input documents to transform the input and generate output documents.



Aggregation pipeline operations include the following:

- $project - Specify keys to be placed in the output document (projected).
- $match - Select documents to be processed, similar to find().
- $limit - Limit the number of documents to be passed to the next steps.
- $skip - Skip a specified number of documents.
- $unwind - Expand an array, generating one output document for each array entry.
- $group - Group documents by a specified key
- $sort - Sort documents
- $geoNear - Select documents near a geospatial location
- $out - Write the results of the pipeline to a collection
- $redact - Control access to certain data

Aggregation Pipeline

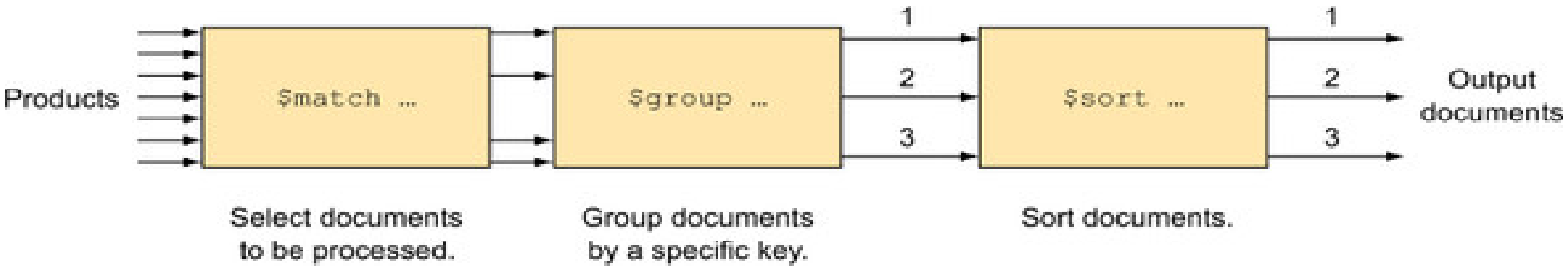


Aggregation Pipeline

SQL command	Aggregation Framework Operator
select	$project
	$group functions: $sum, $min, $avg, etc
from	db.collectionName.aggregate(. . .)
join	$unwind
where	$match
group by	$group
having	$match

Aggregation Pipeline

For example, an aggregation pipeline consists of a match, a group, and then a sort: db.products.aggregate.([{$match : . . . }, {$group : . . . }, {$sort : . . . }]) This series of operations is illustrated as below:



The code defines a pipeline where:

- The entire products collection is passed to the $match operation, which then selects only certain documents from the input collection.
- The output from $match is passed to the $group operator, which then groups the output by a specific key to provide new information such as sums and averages.
- The output from $group operator is then passed to a final $sort operator to be sorted before being returned as the final result.

$group Functions

Operator	Description
$addToSet	Creates an array of unique values for the group
$first	The first value in a group. Makes sense only if preceded by a $sort
$last	Last value in a group. Makes sense only if preceded by a $sort
$max	Maximum value of a field for a group
$min	Minimum value of a field for a group
$avg	Average value for a field
$push	Returns an array of all values for the group. Doesn't eliminate duplicate values.
$sum	Sum of all values in a group

$group Functions

Operator	Description
$addToSet	Creates an array of unique values for the group
$first	The first value in a group. Makes sense only if preceded by a $sort
$last	Last value in a group. Makes sense only if preceded by a $sort
$max	Maximum value of a field for a group
$min	Minimum value of a field for a group
$avg	Average value for a field
$push	Returns an array of all values for the group. Doesn't eliminate duplicate values.
$sum	Sum of all values in a group

- The elements in a set are guaranteed to be unique. A given value doesn't appear twice in the set, and this is enforced by $addToSet
- An array like one created by the $push operator doesn't require each element to be unique. Therefore, the same element may appear more than once in an array created by $push.

A Document Representing a Product Review

```
{_id: ObjectId("4c4b1476238d3b4dd5000041"),
product_id: ObjectId("4c4b1476238d3b4dd5003981"),
date: new Date(2010, 5, 7),
title: "Amazing",
text: "Has a squeaky wheel, but still a darn good wheelbarrow.",
rating: 4,
user_id: ObjectId("4c4b1476238d3b4dd5000042"),
username: "dgreenthumb",
helpful_votes: 3,
voter_ids: [ObjectId("4c4b1476238d3b4dd5000033"),
ObjectId("7a4f0376238d3b4dd5000003"),
ObjectId("92c21476238d3b4dd5000032") ]
}
```

Aggregation Pipeline Examples

```
db.reviews.aggregate([
 {$group : { _id:'$product_id',
             count:{$sum:1} }}
]);
```

Group the input documents by product_id.

Count the number of reviews for each product.

```
{ "_id" : ObjectId("4c4b1476238d3b4dd5003982"), "count" : 2 }
{ "_id" : ObjectId("4c4b1476238d3b4dd5003981"), "count" : 3 }
```

Outputs one document for each product

```
product  = db.products.findOne({'slug': 'wheelbarrow-9092'})

ratingSummary = db.reviews.aggregate([
 {$match : { product_id: product['_id']} },
 {$group : { _id:'$product_id',
             count:{$sum:1} }}
]).next();
```

Select only a single product.

Return the first document in the results.

```
product  = db.products.findOne({'slug': 'wheelbarrow-9092'})

ratingSummary  = db.reviews.aggregate([
    {$match : {'product_id': product['_id']}},
    {$group : { _id:'$product_id',
        average:{$avg:'$rating'},
        count: {$sum:1}}}
]).next();
```

Calculate the average rating for a product.

Aggregation Pipeline Example

- This example returns the one product you're interested in and assigns it to the variable ratingSummary.
- The result from the aggregation pipeline is a cursor, a pointer to your results that allows you to process results of almost any size, one document at a time.
- To retrieve the single document in the result, you use the next() function to return the first document from the cursor:
  { "_id" : ObjectId("4c4b1476238d3b4dd5003981"), "count" : 3 }
- Most query operators we covered are also available in the $match operator.
- It is important to have $match before $group.
- Uses the $avg function to calculate the average rating for the product.
- The field being averaged, rating, is specified using "$rating" in the $avg function. This is the same convention used for specifying the field for the $group _id value, where you used this: _id : "$product_id".

Summary of Aggregation Pipeline Operators

The aggregation framework supports 10 operators:

- $project - Specify document fields to be processed
- $group - Group document fields to be processed
- $match - Select documents to be processed, similar to find(...)
- $limit - Limit the number of documents passed to the next step.
- $skip - Skip a specified number of documents and don't pass them to the next step.
- $unwind - Expand an array, generating one output document for each array entry.
- $sort - Sort documents
- $geoNear - Select documents near a geospatial location
- $out - Write the results of the pipeline to a collection
- $redact - Control access to certain data.