MongoDB – A document data store

MongoDB in short

Feature	MongoDB
Model	Document-based
Query language	Supported, aggregation framework
Reference scenarios	Transactional (read/write intensive) & analytical (read intensive)
Partitioning (Sharding)	Hash-based & range-based (not all the collections are sharded)
Indexes	Primary, secondary, multiattr, fulltext
Replication	**Master-slave**, replica set
Consistency	Strong, eventual at replicas
Availability	Can be mediated with consistency, through r/w concerns
Fault tolerance	By replica set, system remains operational on failing nodes
Transactions	ACID transactions (multidocument since 4.0) (read concern)
CAP theorem	CP
Distributed by	MongoDB Inc.

Mongo DB

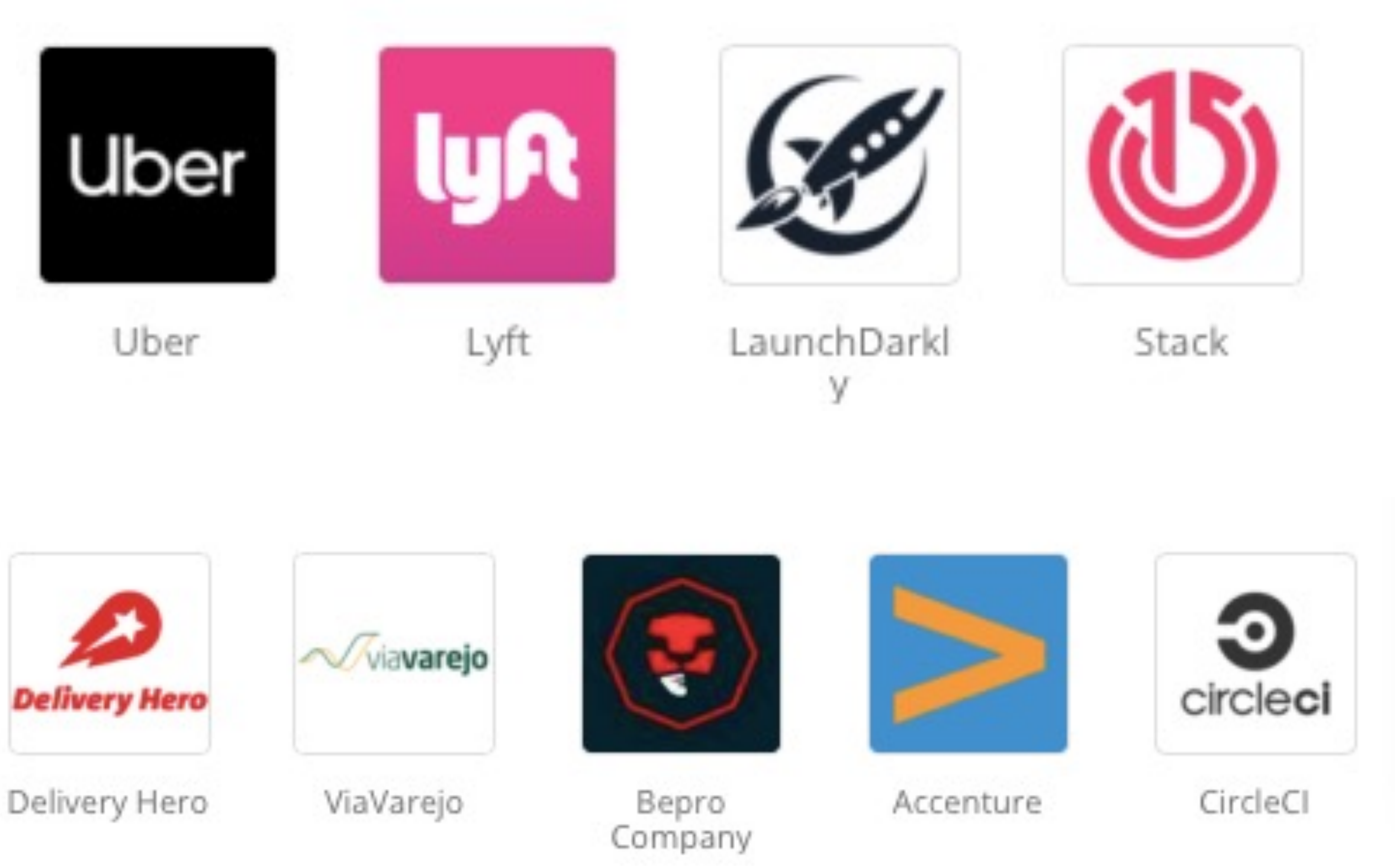
humongous

- History:
  - Development started by 10gen in 2007
  - Open sourced in 2009
  - In 2013, 10gen became MongoDB Inc.

- Uses BSON format
  - Based on JSON –B stands for Binary

- Written in C++, C, Javascript
- Supports APIs (drivers) in many languages
  - JavaScript, Python, Ruby, Perl, Java, JavaScala, C#, C++, Haskell, Erlang, …

Who uses mongoDB?



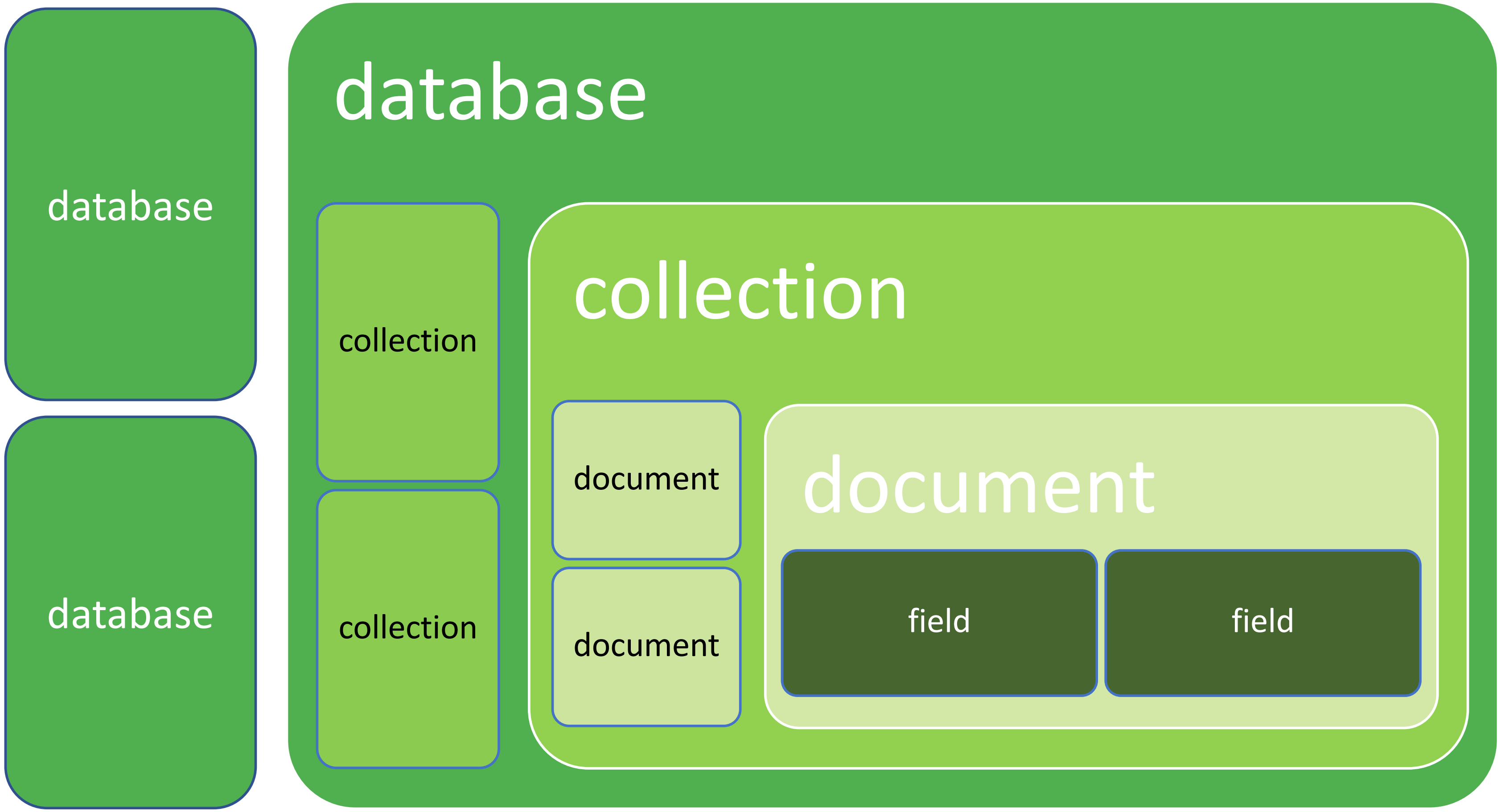
MongoDB

- MongoDB Compass (GUI to explore and manipulate documents)
- MongoDB Atlas (on cloud)
- Supported in Azure Cosmos DB

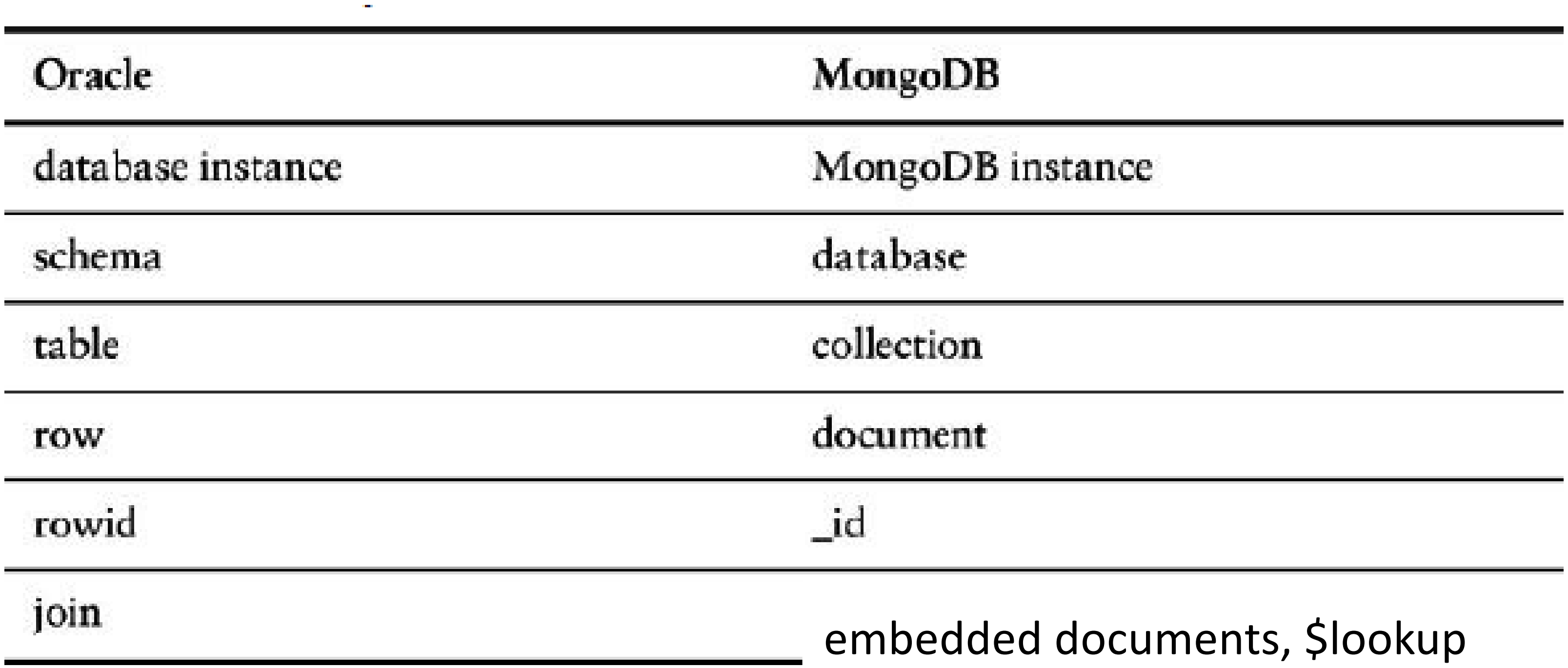
Data Model and Interaction

MongoDB: Basics

- Mongodb instance



MongoDB: Basics



Oracle	MongoDB
database instance	MongoDB instance
schema	database
table	collection
row	document
rowid	_id
join	embedded documents, $lookup

MongoDB: Basics

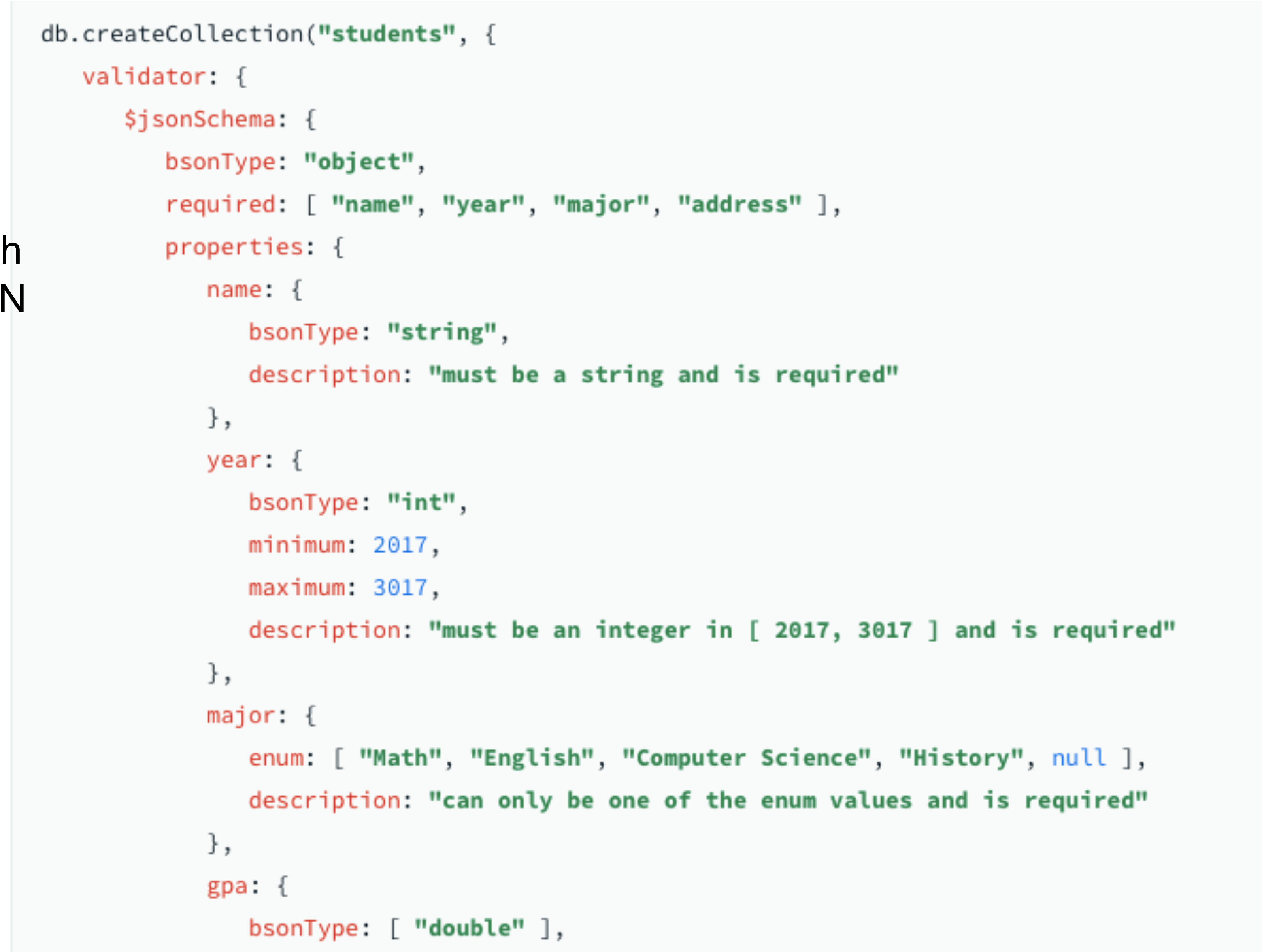
- Each MongoDB instance has multiple databases
  - Similar to a database schema in a RDBMS
- Each database can have multiple collections
  - Similar to a table in a RDBMS
- When we store a document, we have to choose which database and collection this document belongs

```
db.collection.insertOne(document)
```

- Document identifier `_id` will be created for each document, field name reserved by system

Schema validation

A collection may be associated with a schema in JSON schema

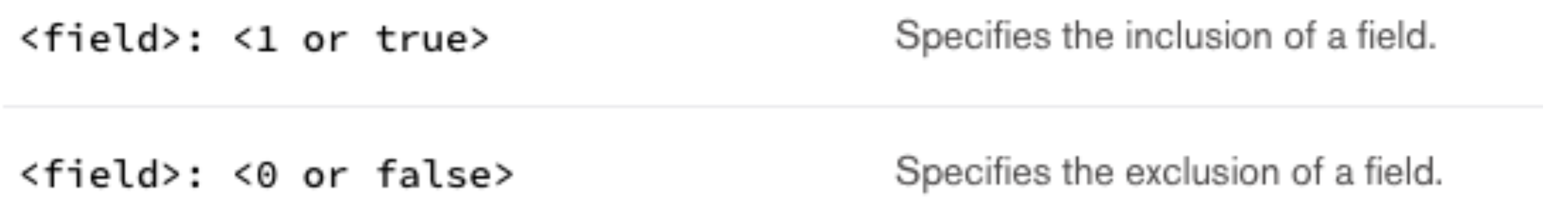


```
db.createCollection("students", {
   validator: {
      $jsonSchema: {
         bsonType: "object",
         required: [ "name", "year", "major", "address" ],
         properties: {
            name: {
               bsonType: "string",
               description: "must be a string and is required"
            },
            year: {
               bsonType: "int",
               minimum: 2017,
               maximum: 3017,
               description: "must be an integer in [ 2017, 3017 ] and is required"
            },
            major: {
               enum: [ "Math", "English", "Computer Science", "History", null ],
               description: "can only be one of the enum values and is required"
            },
            gpa: {
               bsonType: [ "double" ],
```

MongoDB query language

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

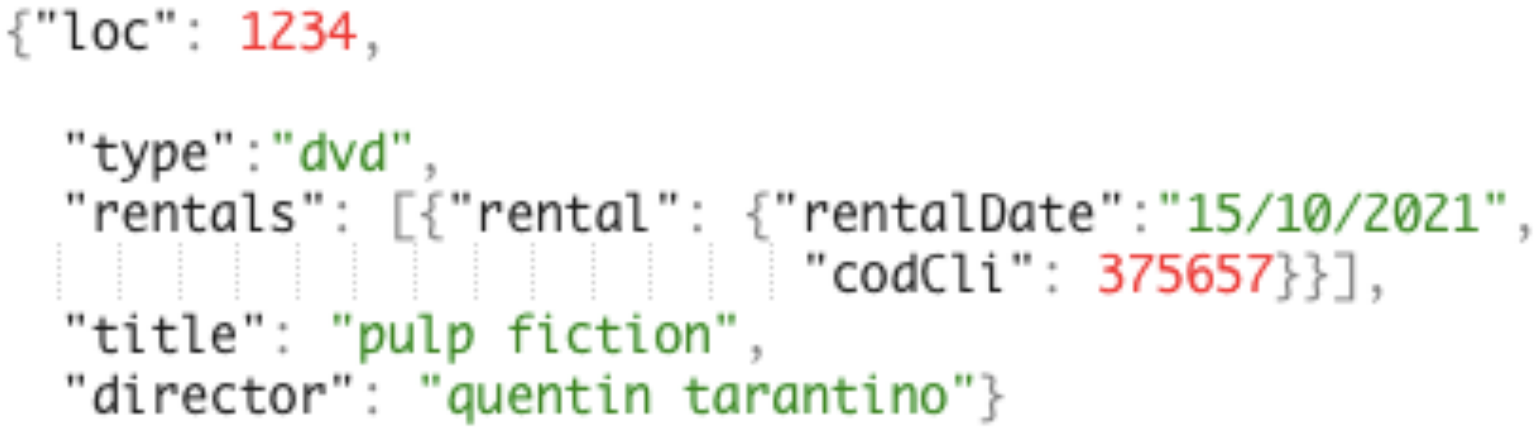
- Provides functionality similar to the SELECT command
  - `<query>` where condition
  - `<projection>` fields in result set



`<field>: <1 or true>`	Specifies the inclusion of a field.
`<field>: <0 or false>`	Specifies the exclusion of a field.

MongoDB query language

```
// in videos
```

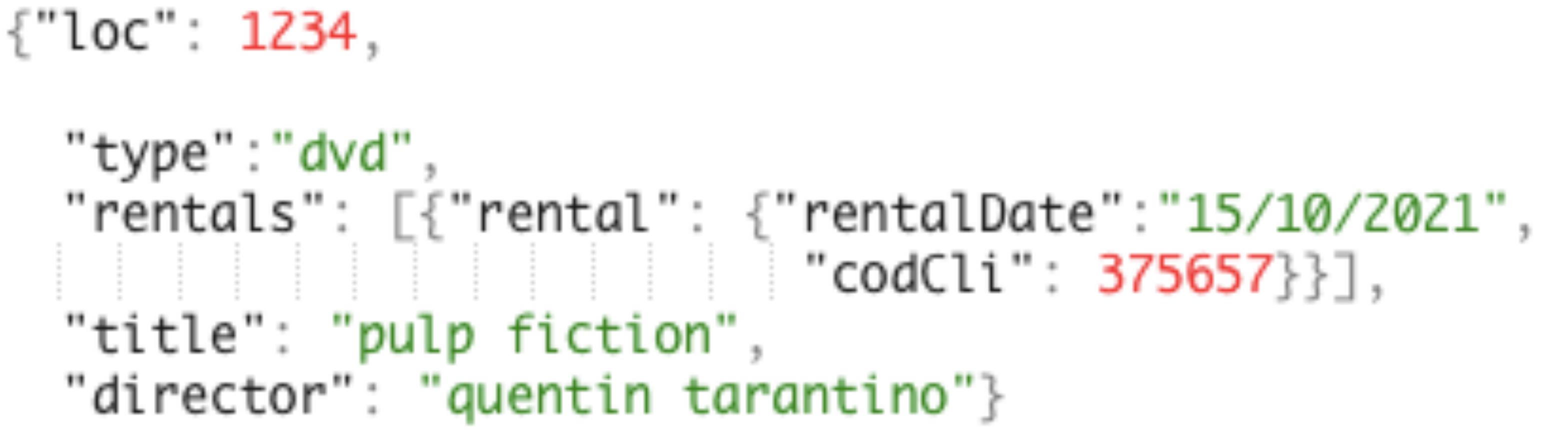


```
{"loc": 1234,

  "type":"dvd",
  "rentals": [{"rental": {"rentalDate":"15/10/2021",
                          "codCli": 375657}}],
  "title": "pulp fiction",
  "director": "quentin tarantino"}
```

```
db.videos.find()
```

MongoDB query language

```
// in videos
```



```
{"loc": 1234,

   "type":"dvd",
   "rentals": [{"rental": {"rentalDate":"15/10/2021",
                          "codCli": 375657}}],
   "title": "pulp fiction",
   "director": "quentin tarantino"}
```

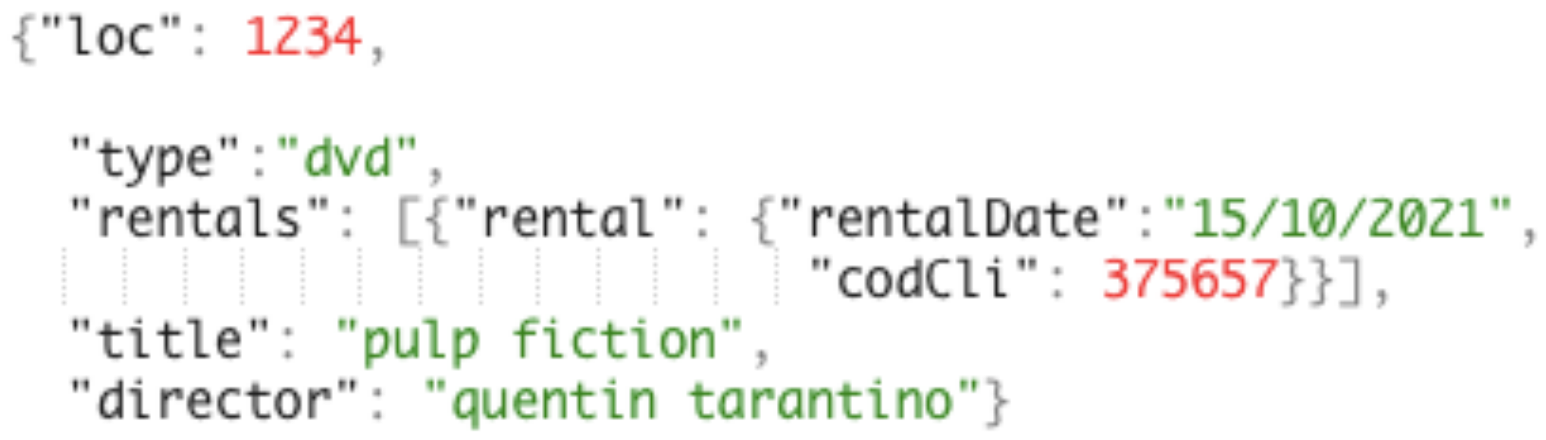
```
db.videos.find({"loc":1234},{title:1})
```

where (filtering - selection)

Select (projection)

MongoDB query language

```
// in videos
```



```
{"loc": 1234,

   "type":"dvd",
   "rentals": [{"rental": {"rentalDate":"15/10/2021",
                           "codCli": 375657}}],
   "title": "pulp fiction",
   "director": "quentin tarantino"}
```

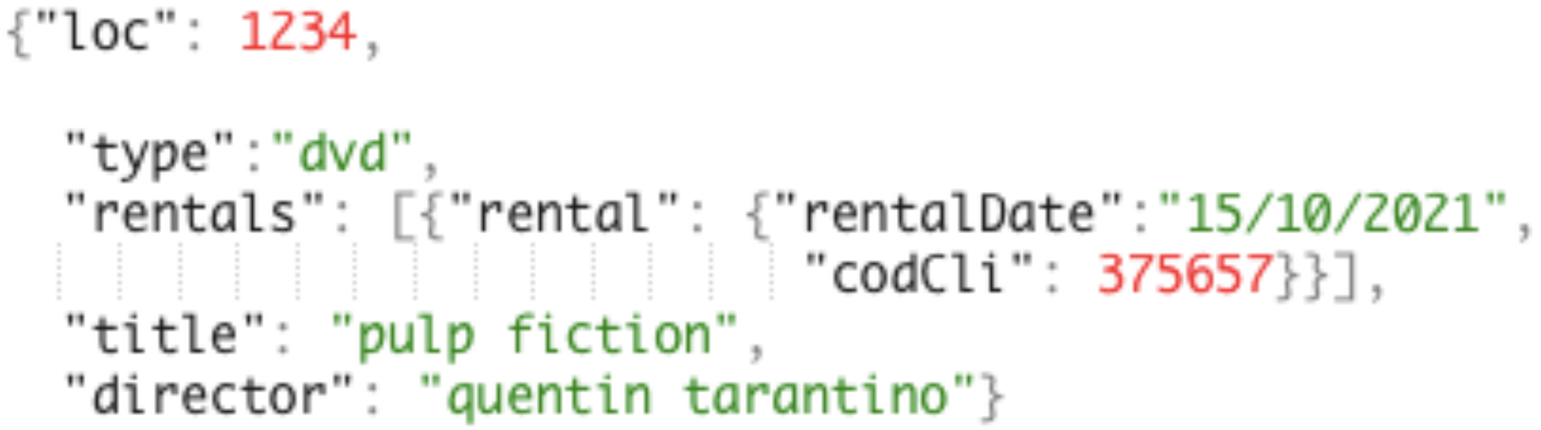
```
db.videos.find({},{title:1})
```

Select (projection)

Projection only, _id is included in the result if not explicitly excluded (i.e., {_id: 0})

MongoDB query language

```
// in videos
```



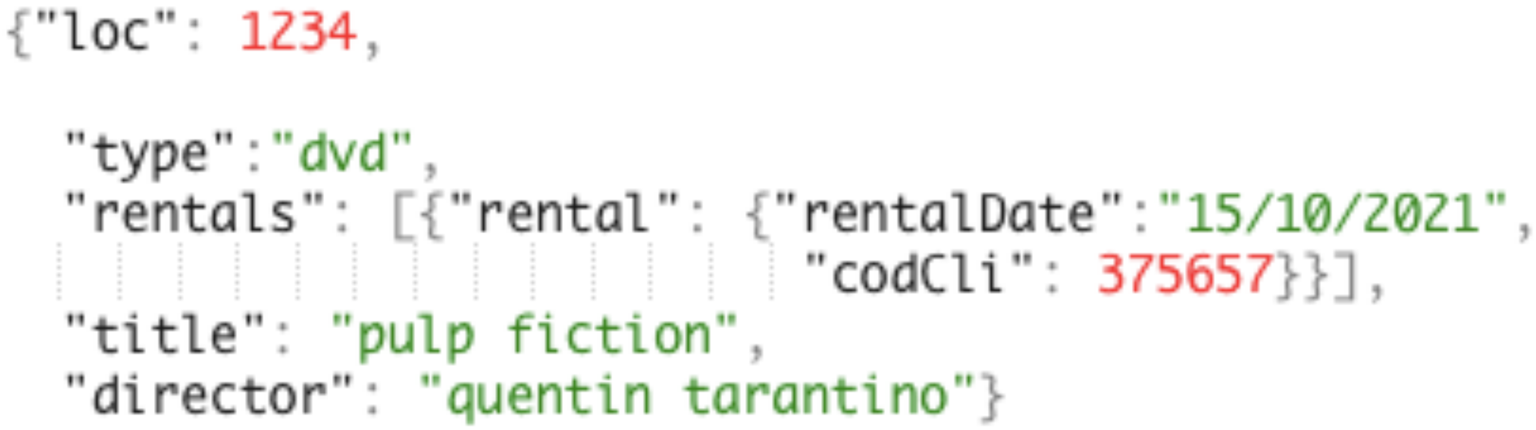
```
{"loc": 1234,

   "type":"dvd",
   "rentals": [{"rental": {"rentalDate":"15/10/2021",
                           "codCli": 375657}}],
   "title": "pulp fiction",
   "director": "quentin tarantino"}
```

```
db.videos.find({"title":"pulp fiction"})
```

where (filtering - selection)

MongoDB query language

```
// in videos
```



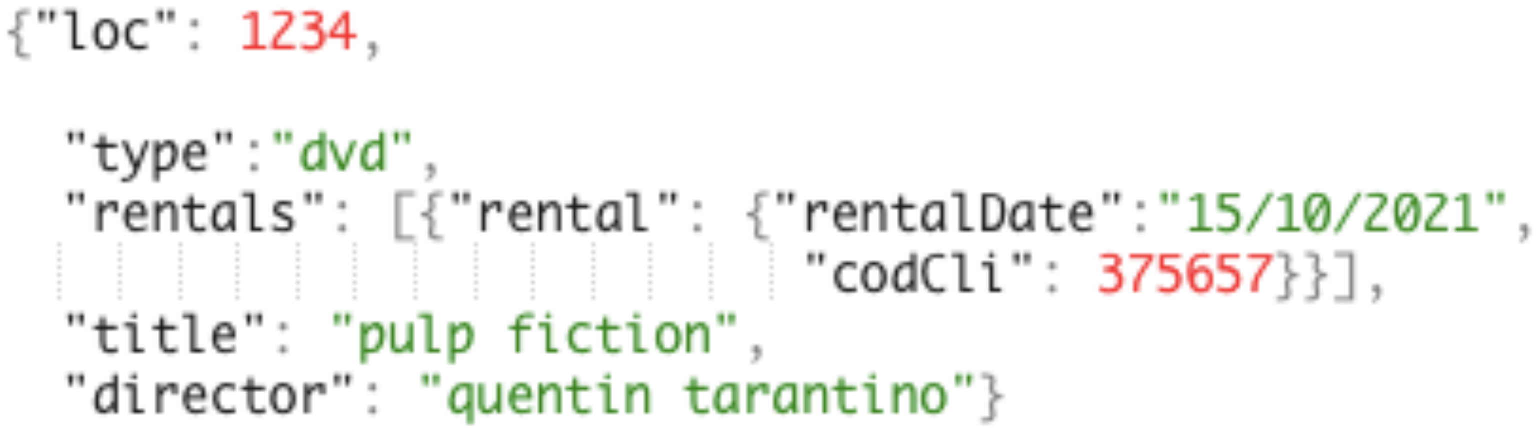
```
{"loc": 1234,

   "type":"dvd",
   "rentals": [{"rental": {"rentalDate":"15/10/2021",
                             "codCli": 375657}}],
   "title": "pulp fiction",
   "director": "quentin tarantino"}
```

```
db.videos.find({"rentals.rental.codCli":"375657")
```

path expression (dot notation)

MongoDB query language

```
// in videos
```

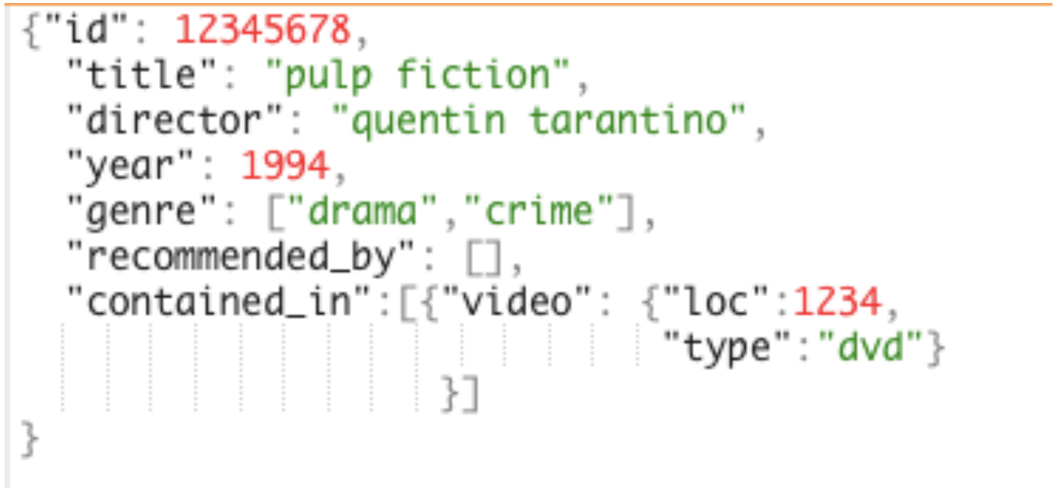


```
{"loc": 1234,

   "type":"dvd",
   "rentals": [{"rental": {"rentalDate":"15/10/2021",
                            "codCli": 375657}}],
   "title": "pulp fiction",
   "director": "quentin tarantino"}
```

```
db.videos.find({"type": {$in,["dvd", "vhs"]}})
```

MongoDB query language

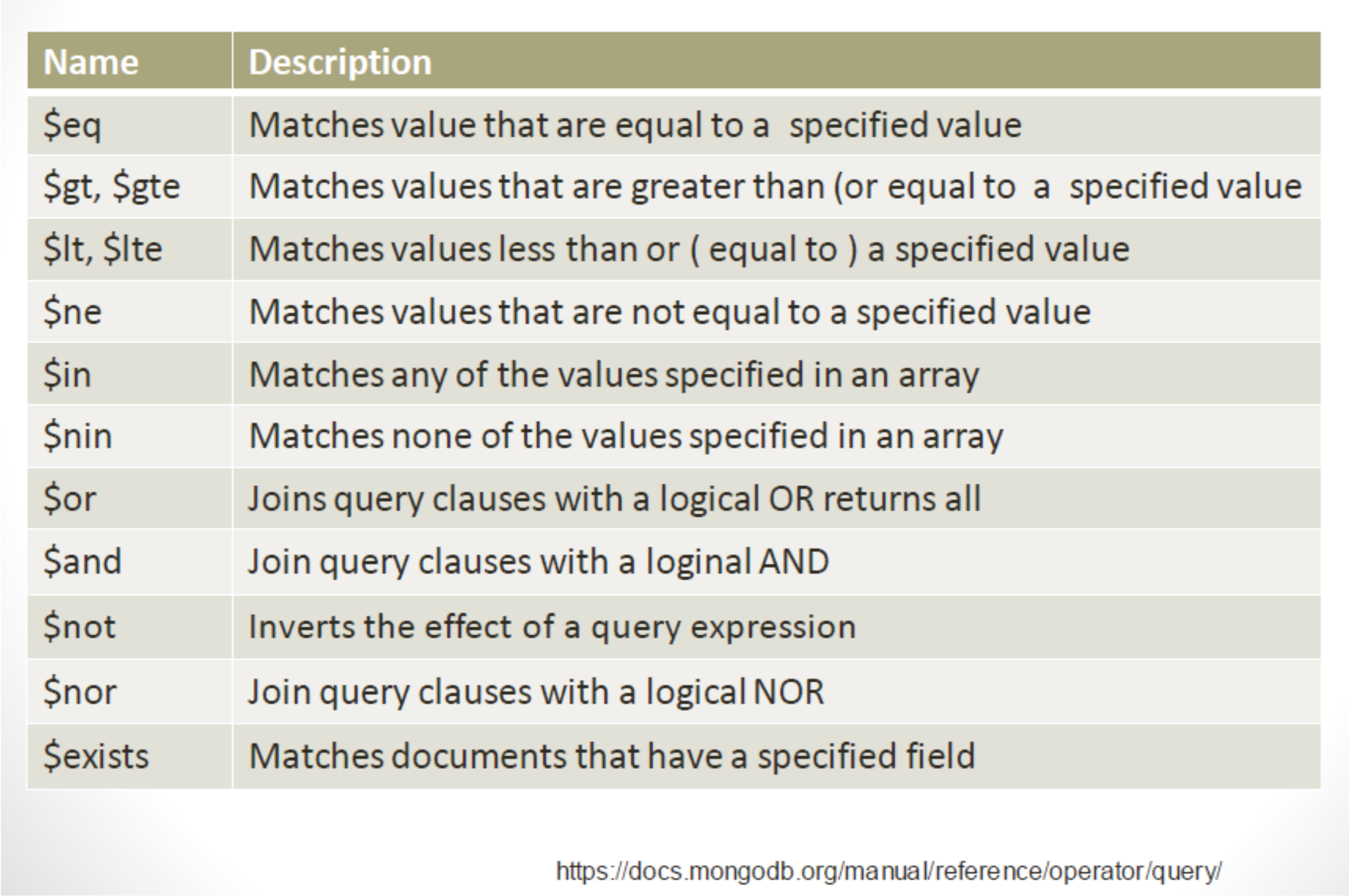
```
// in movies
```



```
{"id": 12345678,
  "title": "pulp fiction",
  "director": "quentin tarantino",
  "year": 1994,
  "genre": ["drama","crime"],
  "recommended_by": [],
  "contained_in":[{"video": {"loc":1234,
                             "type":"dvd"}
                 }]
}
```

```
db.movies.find({"year" : {$geq: 1990}})
```

MongoDB – Query Operators

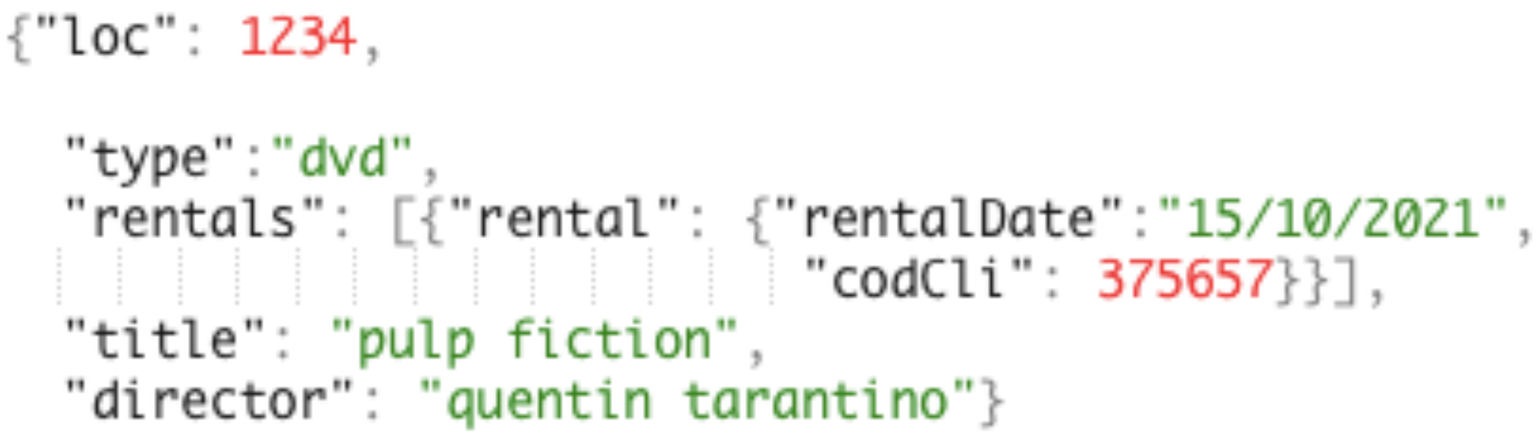


Name	Description
$eq	Matches value that are equal to a specified value
$gt, $gte	Matches values that are greater than (or equal to a specified value
$lt, $lte	Matches values less than or (equal to) a specified value
$ne	Matches values that are not equal to a specified value
$in	Matches any of the values specified in an array
$nin	Matches none of the values specified in an array
$or	Joins query clauses with a logical OR returns all
$and	Join query clauses with a loginal AND
$not	Inverts the effect of a query expression
$nor	Join query clauses with a logical NOR
$exists	Matches documents that have a specified field

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

MongoDB query language

```
// in videos
```

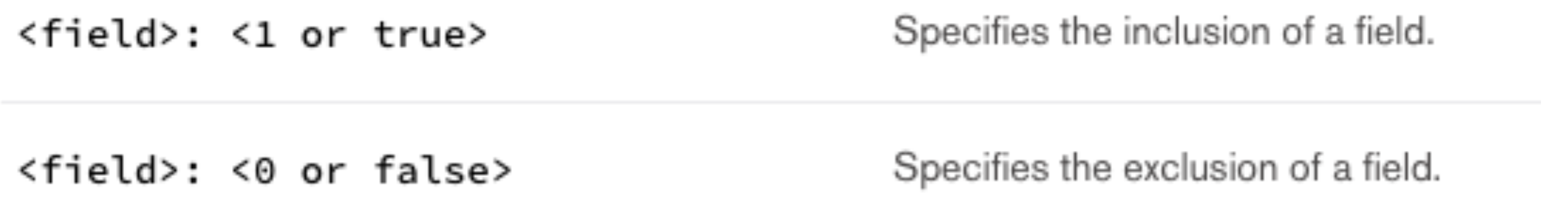


```
{"loc": 1234,

   "type":"dvd",
   "rentals": [{"rental": {"rentalDate":"15/10/2021",
                           "codCli": 375657}}],
   "title": "pulp fiction",
   "director": "quentin tarantino"}
```

```
db.videos.find({"title":"pulp fiction",
"director" : "gabriele salvatores" })


db.videos.find({$or: [{"title":"pulp fiction",
"director" : "gabriele salvatores" }] })
```

MongoDB query language

- `db.collection.find( <query>, <projection> )`
- Provides functionality similar to the SELECT command
  - `<query>` where condition
  - `<projection>` fields in result set



`<field>: <1 or true>`	Specifies the inclusion of a field.
`<field>: <0 or false>`	Specifies the exclusion of a field.

- Return a cursor 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> )`

Sort

- `db.movies.find({"year" : {$geq: 1990}}).sort ({"title": 1})`

- `db.movies.find({"director" : "quentin tarantino"}).sort ({"year": -1})`

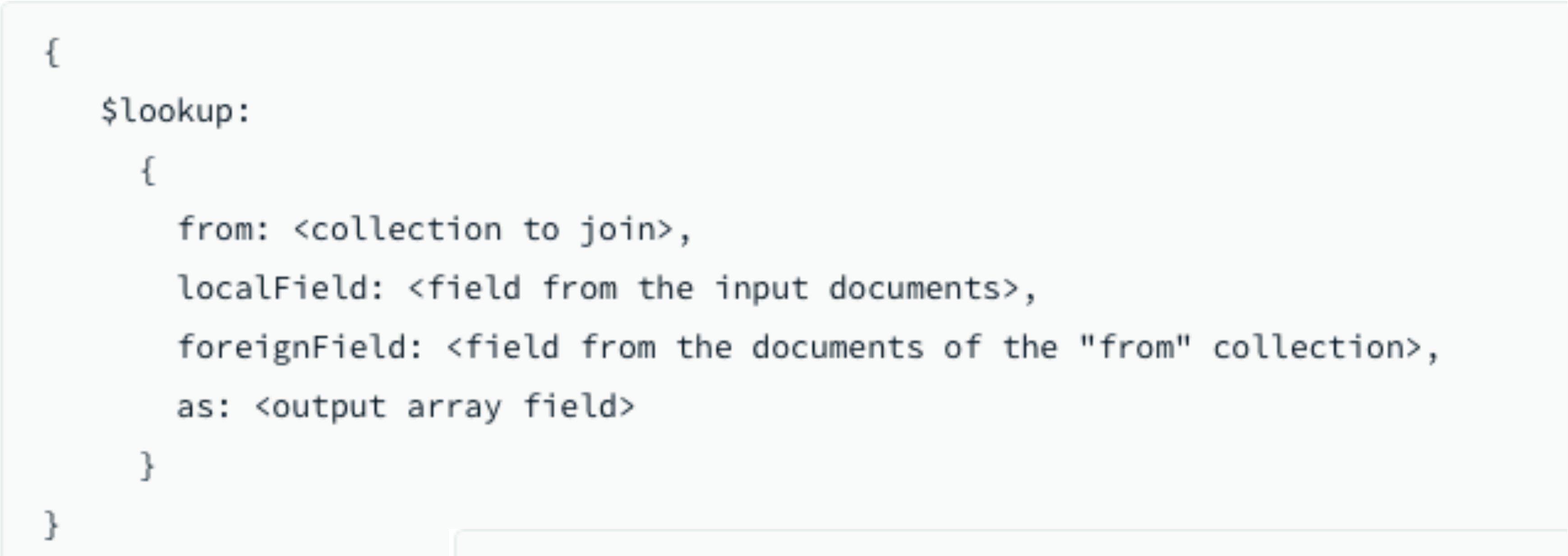
Count

- `count()`or `find().count()`

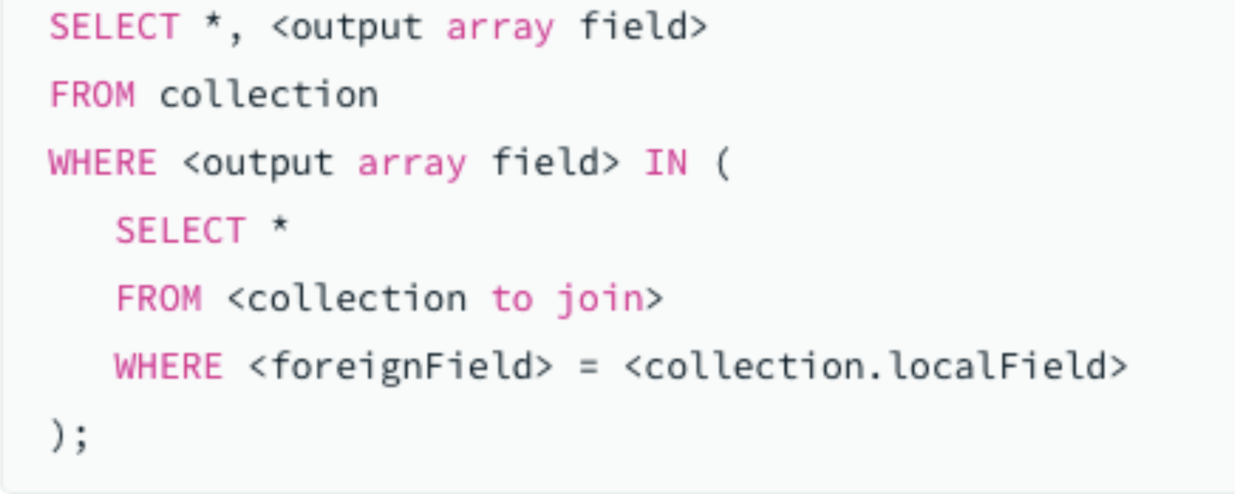
- Can have the same arguments as find

- **`db.movies.count({"year" : {$geq: 1990}})`**

Lookup

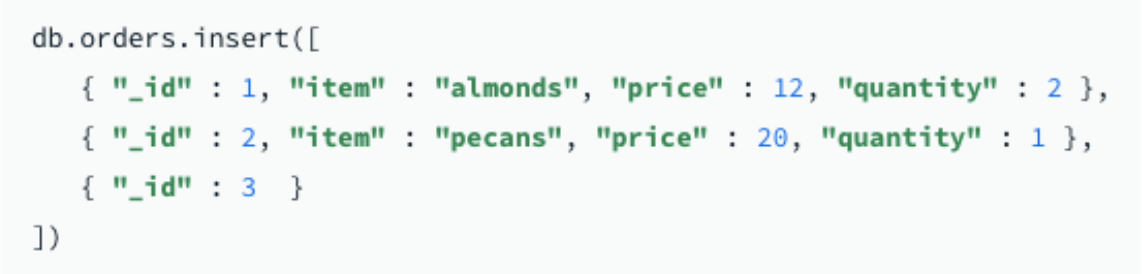


```
{
   $lookup:
     {
       from: <collection to join>,
       localField: <field from the input documents>,
       foreignField: <field from the documents of the "from" collection>,
       as: <output array field>
     }
}
```

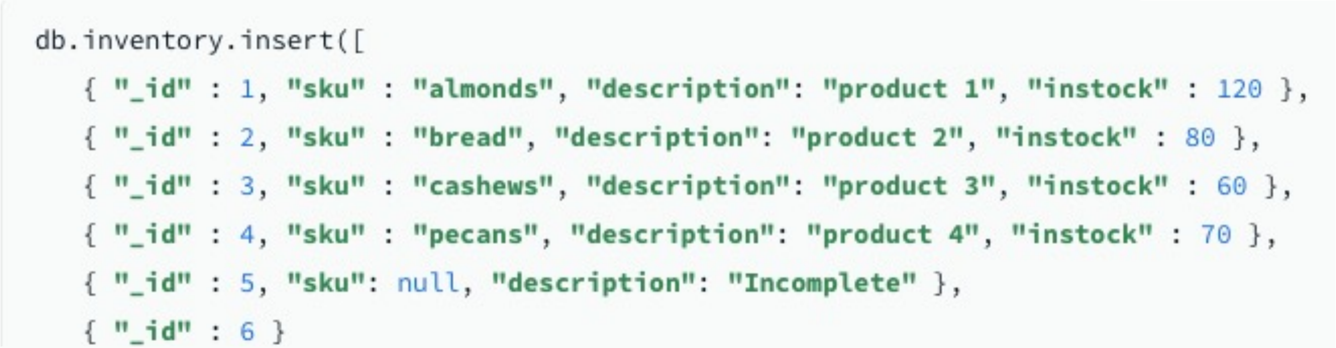


```
SELECT *, <output array field>
FROM collection
WHERE <output array field> IN (
   SELECT *
   FROM <collection to join>
   WHERE <foreignField> = <collection.localField>
);
```

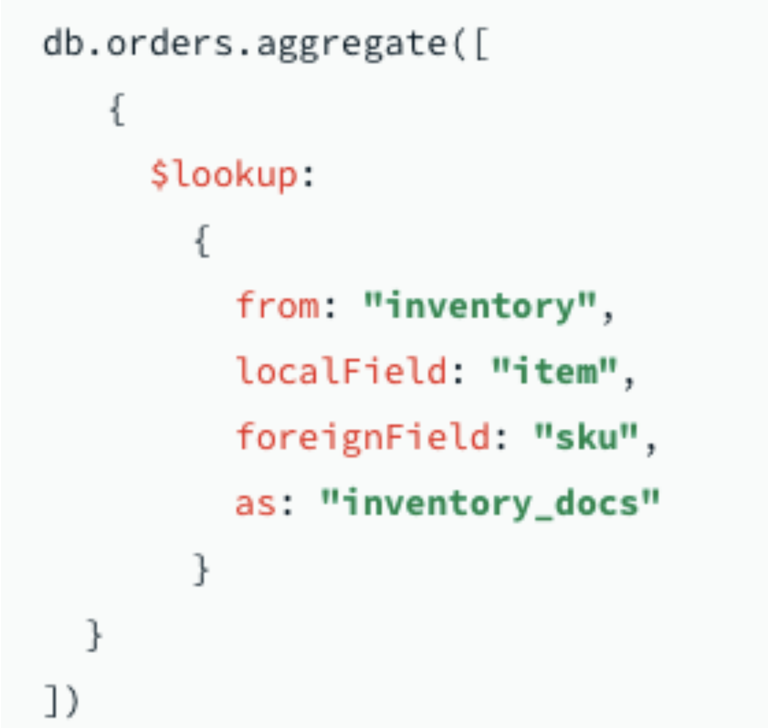
Lookup



```
db.orders.insert([
   { "_id" : 1, "item" : "almonds", "price" : 12, "quantity" : 2 },
   { "_id" : 2, "item" : "pecans", "price" : 20, "quantity" : 1 },
   { "_id" : 3  }
])
```

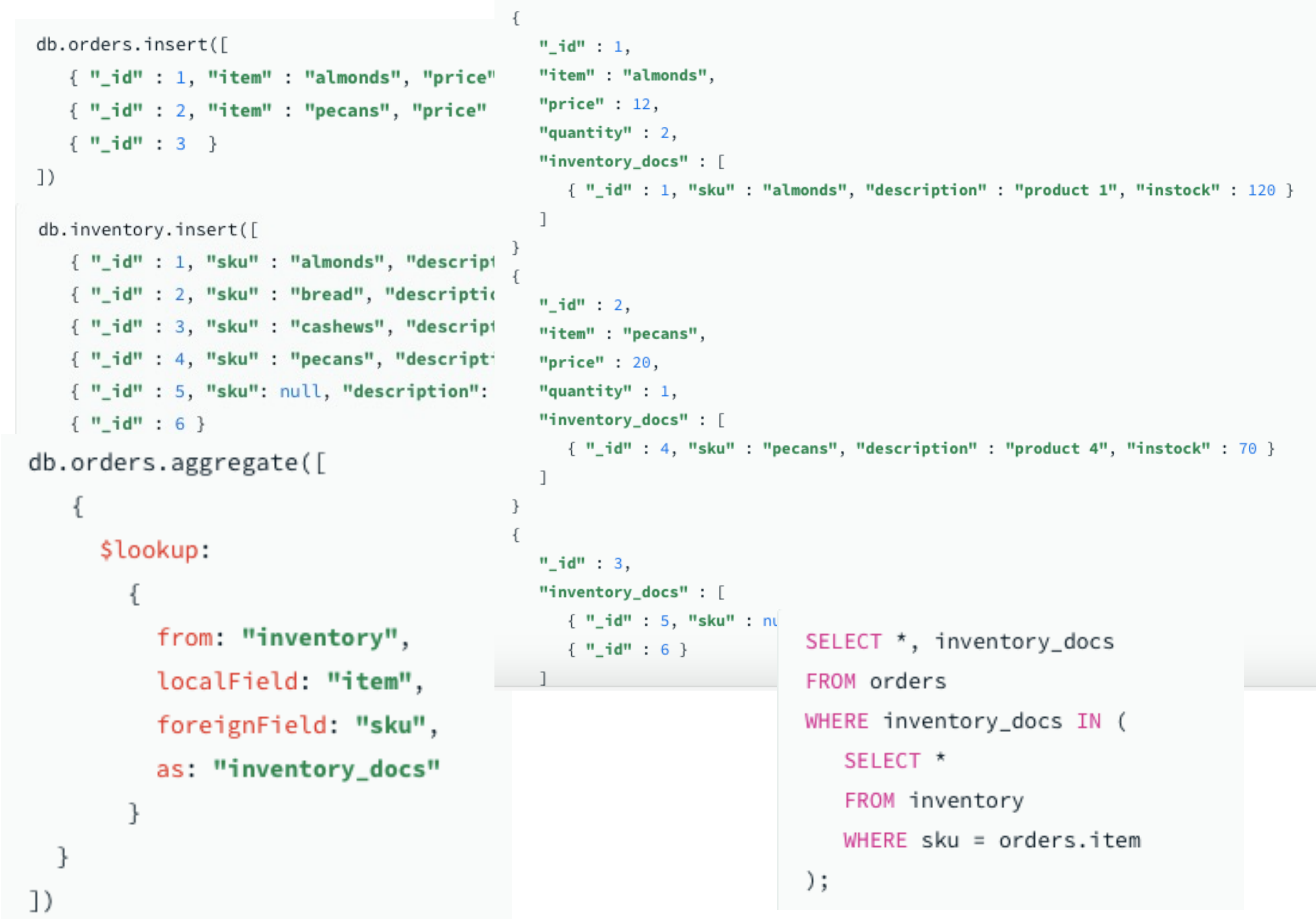


```
db.inventory.insert([
   { "_id" : 1, "sku" : "almonds", "description": "product 1", "instock" : 120 },
   { "_id" : 2, "sku" : "bread", "description": "product 2", "instock" : 80 },
   { "_id" : 3, "sku" : "cashews", "description": "product 3", "instock" : 60 },
   { "_id" : 4, "sku" : "pecans", "description": "product 4", "instock" : 70 },
   { "_id" : 5, "sku": null, "description": "Incomplete" },
   { "_id" : 6 }
```



```
db.orders.aggregate([
   {
     $lookup:
       {
         from: "inventory",
         localField: "item",
         foreignField: "sku",
         as: "inventory_docs"
       }
  }
])
```

Lookup



```
db.orders.insert([
   { "_id" : 1, "item" : "almonds", "price"
   { "_id" : 2, "item" : "pecans", "price"
   { "_id" : 3  }
])
```

```
db.inventory.insert([
   { "_id" : 1, "sku" : "almonds", "descript
   { "_id" : 2, "sku" : "bread", "descriptic
   { "_id" : 3, "sku" : "cashews", "descript
   { "_id" : 4, "sku" : "pecans", "descript
   { "_id" : 5, "sku": null, "description":
   { "_id" : 6 }
```

```
db.orders.aggregate([
   {
     $lookup:
       {
         from: "inventory",
         localField: "item",
         foreignField: "sku",
         as: "inventory_docs"
       }
  }
])
```

```
{
   "_id" : 1,
   "item" : "almonds",
   "price" : 12,
   "quantity" : 2,
   "inventory_docs" : [
      { "_id" : 1, "sku" : "almonds", "description" : "product 1", "instock" : 120 }
   ]
}
{
   "_id" : 2,
   "item" : "pecans",
   "price" : 20,
   "quantity" : 1,
   "inventory_docs" : [
      { "_id" : 4, "sku" : "pecans", "description" : "product 4", "instock" : 70 }
   ]
}
{
   "_id" : 3,
   "inventory_docs" : [
      { "_id" : 5, "sku" : nu
      { "_id" : 6 }
   ]
```

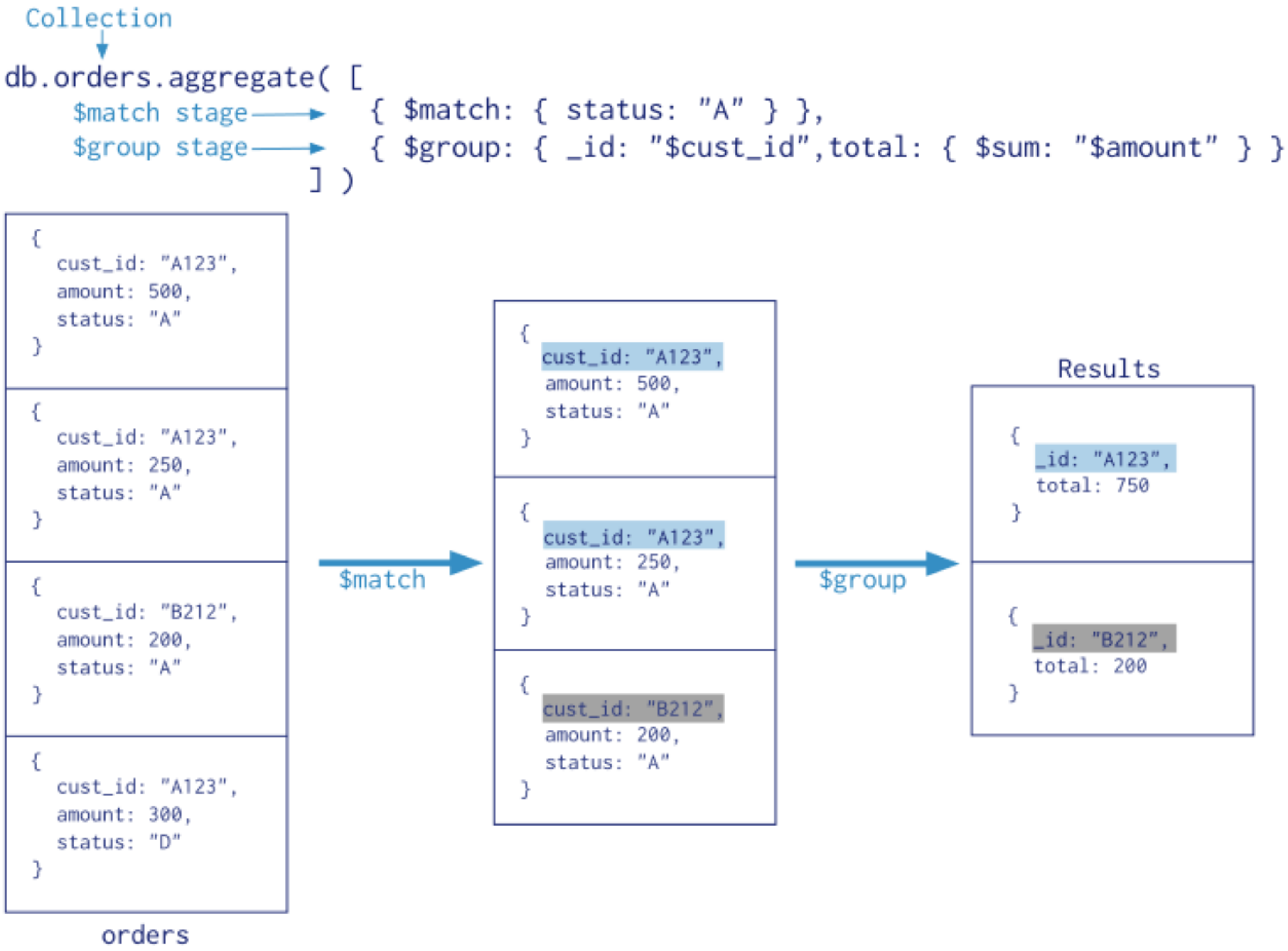
```
SELECT *, inventory_docs
FROM orders
WHERE inventory_docs IN (
   SELECT *
   FROM inventory
   WHERE sku = orders.item
);
```

Aggregates

- Aggregation framework provides SQL-like aggregation functionality
- Documents from a collection pass through an aggregation pipeline, which transforms these objects as they pass through
- Expressions produce output documents based on calculations performed on input documents
- Example

-
  ```
  db.orders.aggregate( {$group : {_id: type,
                    totalquantity: { $sum: quantity} } } )
  ```

Aggregates



```
db.orders.aggregate( [
      { $match: { status: "A" } },
      { $group: { _id: "$cust_id",total: { $sum: "$amount" } } }
] )
```

```
{
   cust_id: "A123",
   amount: 500,
   status: "A"
}
{
   cust_id: "A123",
   amount: 250,
   status: "A"
}
{
   cust_id: "B212",
   amount: 200,
   status: "A"
}
{
   cust_id: "A123",
   amount: 300,
   status: "D"
}
```

orders

```
{
   cust_id: "A123",
   amount: 500,
   status: "A"
}
{
   cust_id: "A123",
   amount: 250,
   status: "A"
}
{
   cust_id: "B212",
   amount: 200,
   status: "A"
}
```

Results

```
{
   _id: "A123",
   total: 750
}
{
   _id: "B212",
   total: 200
}
```

REMARK: partition* is called shard* in mongodb terminology

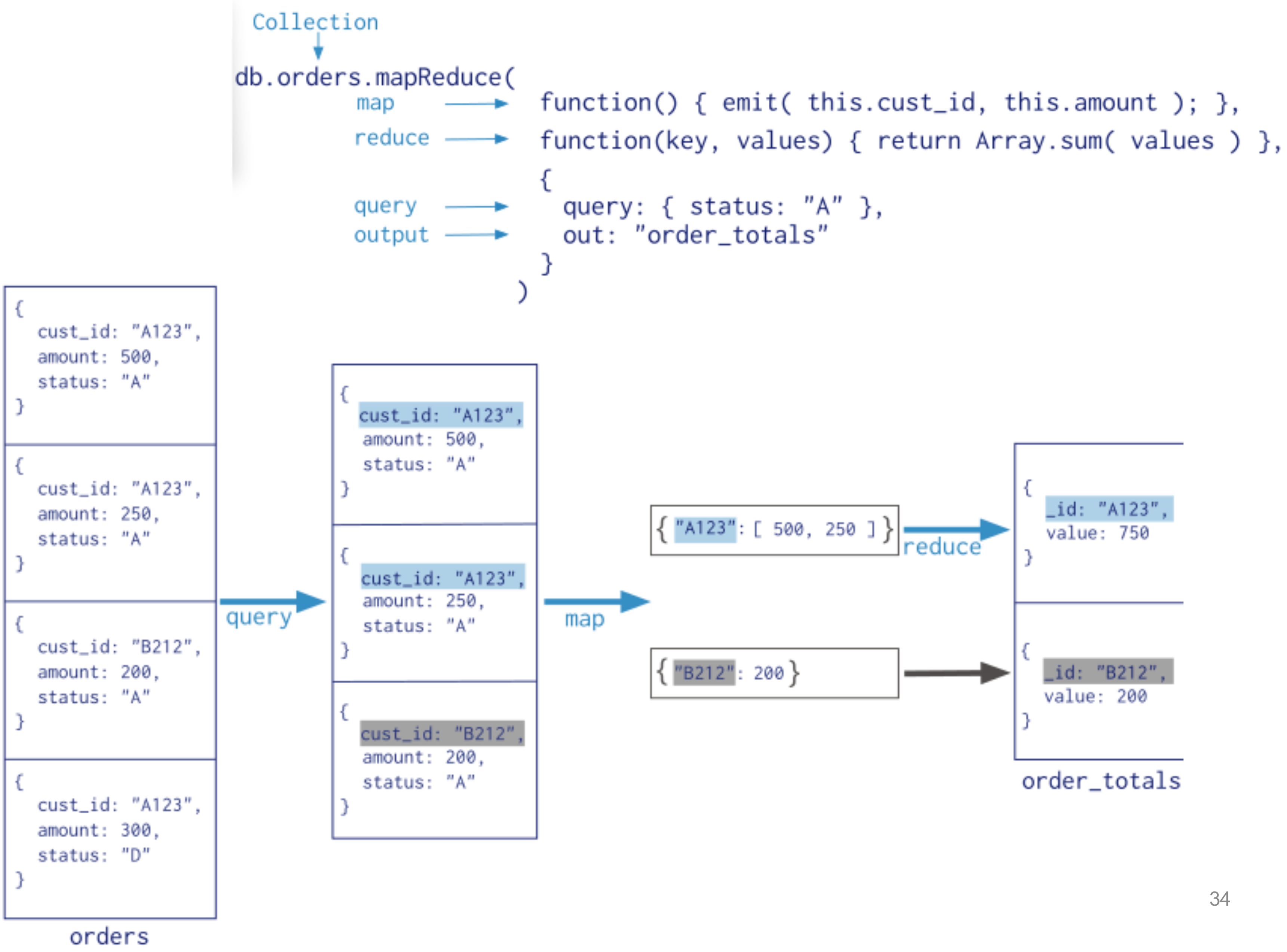
Map Reduce

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

```
db.collection.mapReduce( <mapfunction>,
                         <reducefunction>,
                { out: <collection>,
                  query: <document>,
                  sort: <document>,
                  limit: <number>} )
```

- If the input is a partitioned collection, mongos will automatically dispatch the map-reduce job to each partition in parallel
- if the out field for mapReduce has the partitioning value, the output collection is partitioned using the `_id` field as the partition key

Map Reduce



```
db.orders.mapReduce(
                        function() { emit( this.cust_id, this.amount ); },
                        function(key, values) { return Array.sum( values ) },
                        {
                          query: { status: "A" },
                          out: "order_totals"
                        }
                      )
```

orders

```
{
  cust_id: "A123",
  amount: 500,
  status: "A"
}
{
  cust_id: "A123",
  amount: 250,
  status: "A"
}
{
  cust_id: "B212",
  amount: 200,
  status: "A"
}
{
  cust_id: "A123",
  amount: 300,
  status: "D"
}
```

```
{
  cust_id: "A123",
  amount: 500,
  status: "A"
}
{
  cust_id: "A123",
  amount: 250,
  status: "A"
}
{
  cust_id: "B212",
  amount: 200,
  status: "A"
}
```

```
{ "A123": [ 500, 250 ] }
{ "B212": 200 }
```

order_totals

```
{
  _id: "A123",
  value: 750
}
{
  _id: "B212",
  value: 200
}
```

CRUD Operations

All write operations in MongoDB are atomic on the level of a single document.

- Create

```
db.collection.insertOne( <document> )
db.collection.insertMany([<document>,<document>,...])
```

- Update

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

   … **updateOne and updateMany**

- Delete

```
db.collection.deleteMany( <query> )
db.collection.deleteOne( <query> )
```

- Read

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

https://docs.mongodb.com/manual/tutorial/update-documents/

Indexing

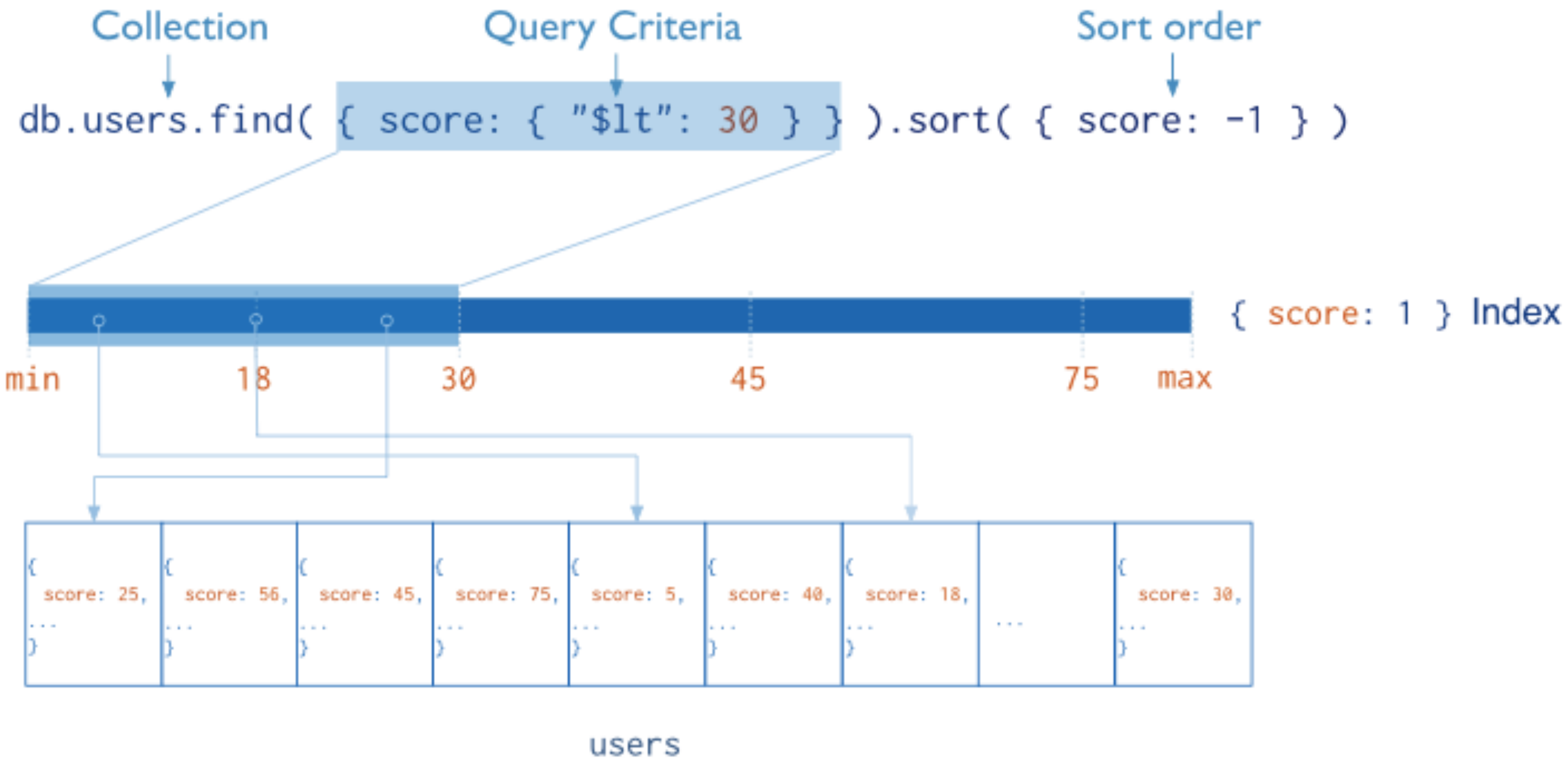
MongoDB - Indexes

```
> db.clients.find(firstname: "alice"}).explain()
{
"cursor" : "BasicCursor",
"nscanned" : 1000000,
"nscannedObjects" : 1000000,
"n" : 1,
"millis" : 721,
"nYields" : 0,
"nChunkSkips" : 0,
"isMultiKey" : false,
"indexOnly" : false,
"indexBounds" : {}
}
```

MongoDB - Indexes

```
>db.users.ensureIndex({"firstname" : 1})

> db.clients.find(firstname: "alice"}).explain()
{
"cursor" : "BtreeCursor username_1",
"nscanned" : 1,
"nscannedObjects" : 1,
"n" : 1,
"millis" : 3,
"nYields" : 0,
"nChunkSkips" : 0,
"isMultiKey" : false,
"indexOnly" : false,
"indexBounds" : {"firstname" : [ ["alice", "alice" ] ] }
}
```

Index functionalities



```
db.users.find( { score: { "$lt": 30 } } ).sort( { score: -1 } )
```

Index functionalities

- An index is automatically created on the `_id` field (the primary key)
- Users can create other indexes
  - to improve query performance (filter conditions, sorting on the field) or
  - to enforce unique values for a particular field `(unique)`

```
db.users.ensureIndex({"username" : 1}, {"unique" :
true})
```

Index functionalities

- Supports single field index as well as compound index

```
db.users.ensureIndex({"age" : 1, "username" : 1})
```

- Like SQL: order of the fields in a compound index matters
- If you index a field that holds an array value, MongoDB creates separate index entries for *every* element of the array
- Also supports hash indexes

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

Full-text Indexes and Search

- MongoDB has a special type of index for searching for text within documents
  - built-in support for multi-language stemming and stop words
- Heavyweight, be cautious
- `db.stores.createIndex( { name: "text", description: "text" } )`

- Text search

-
  ```
  db.stores.find( { $text:
                  { $search: "java coffee shop" } } )
  ```
- It is possible to exclude words, and to sort on relevance ranking `textScore`

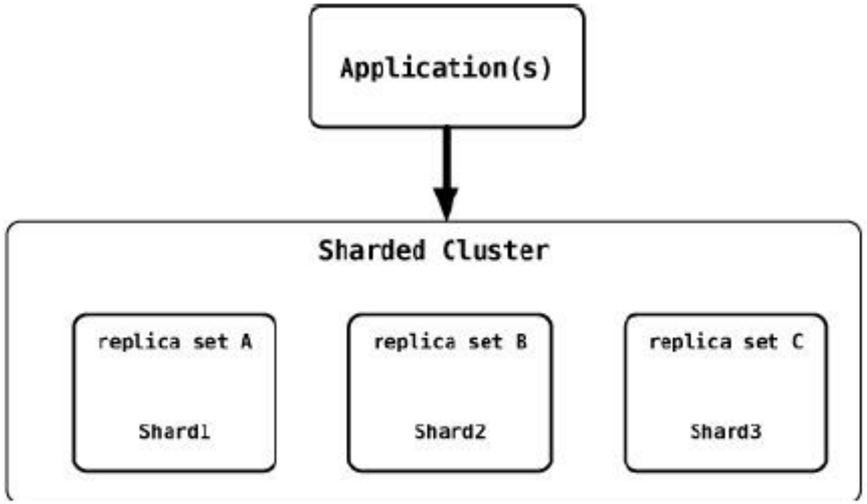
Architecture

MongoDB in short

Feature	**MongoDB**
Model	Document-based
Query language	Supported, aggregation framework
Reference scenarios	Transactional (read/write intensive) & analytical (read intensive)
Partitioning (Sharding)	Hash-based & range-based (not all the collections are sharded)
Indexes	Primary, secondary, multiattr, fulltext
Replication	**Master-slave**, replica set
Consistency	Strong, eventual at replicas
Availability	Can be mediated with consistency, through r/w concerns
Fault tolerance	By replica set, system remains operational on failing nodes
Transactions	ACID transactions (multidocument since 4.0) (read concern)
CAP theorem	CP
Distributed by	MongoDB Inc.

Collection sharding/data partitioning

- By sharding/partitioning the data is split by certain field and moved to different nodes

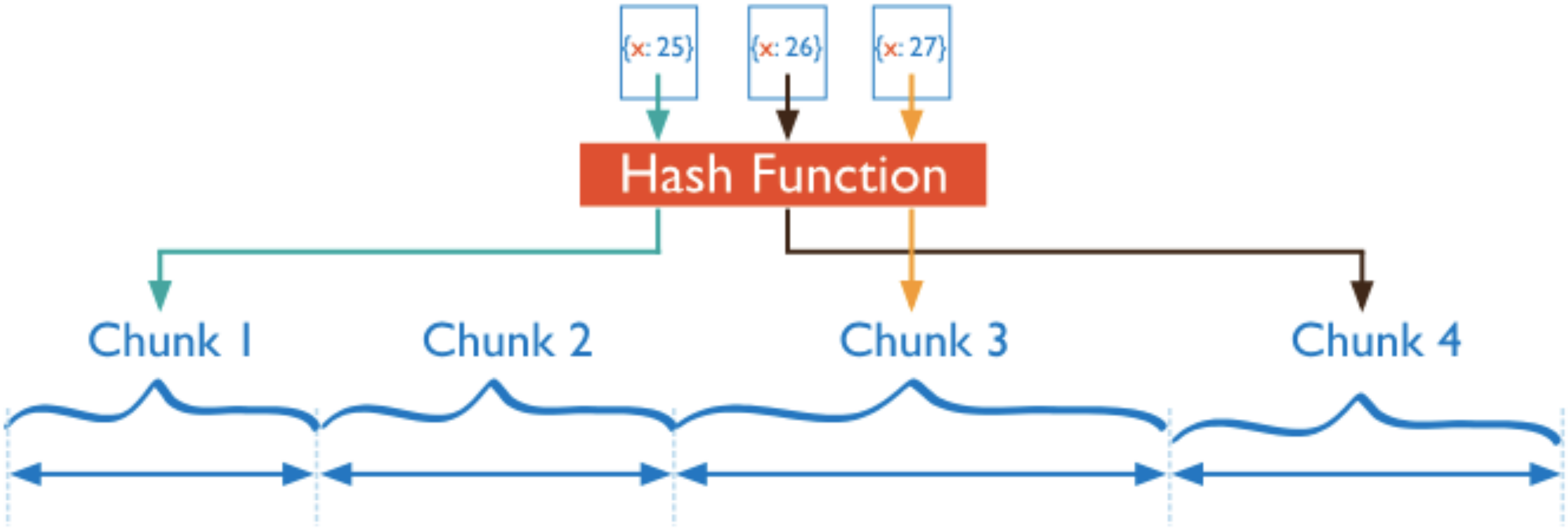


- Partitioning on the first name of the customer:

```
db.runCommand({shardcollection:”videorental.clients",
                   key:{firstname:1}})
```

Partitioning / Sharding

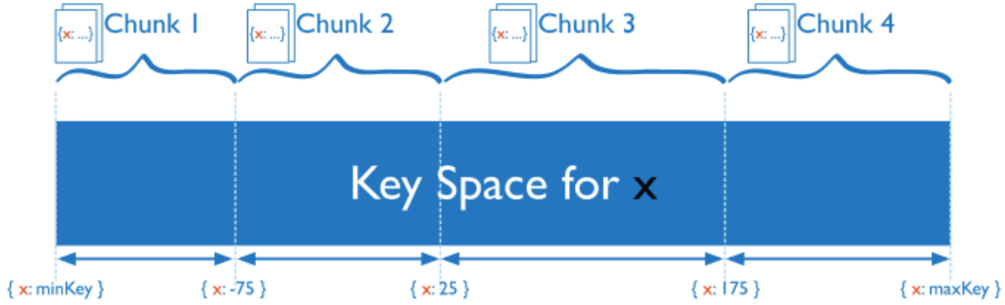
- Hash-based sharding

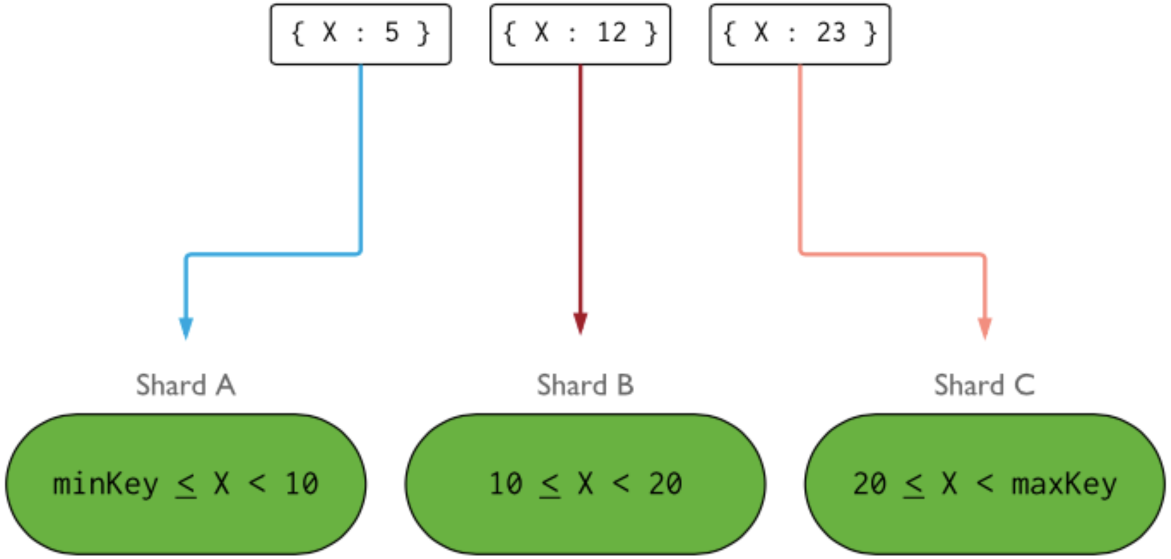


Partitioning / Sharding

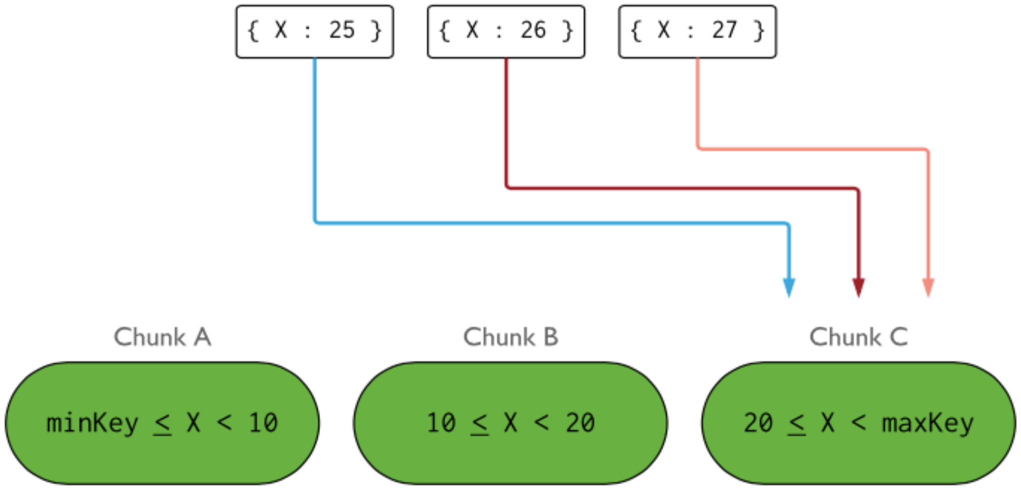
- Ranged sharding

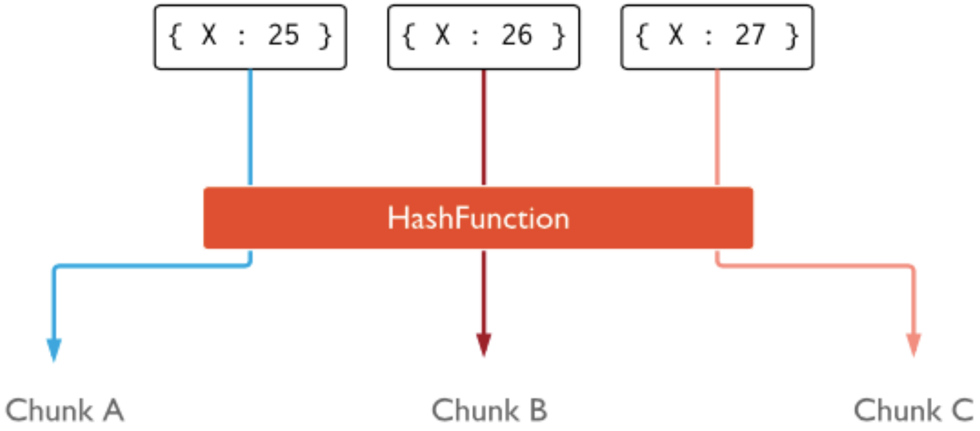
Ranged sharding is most efficient
when the shard key is
Large Cardinality
Low Frequency
Non-Monotonically Changing





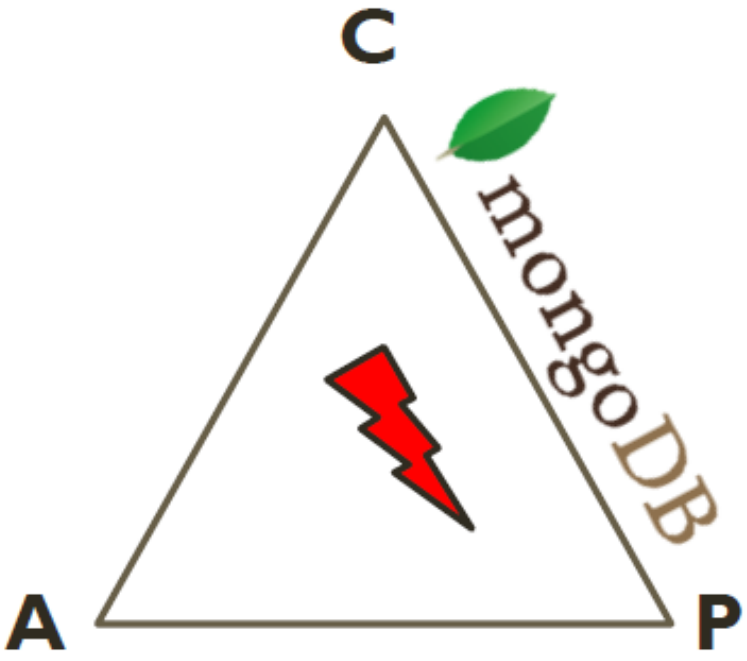
Partitioning with monotonically increasing keys





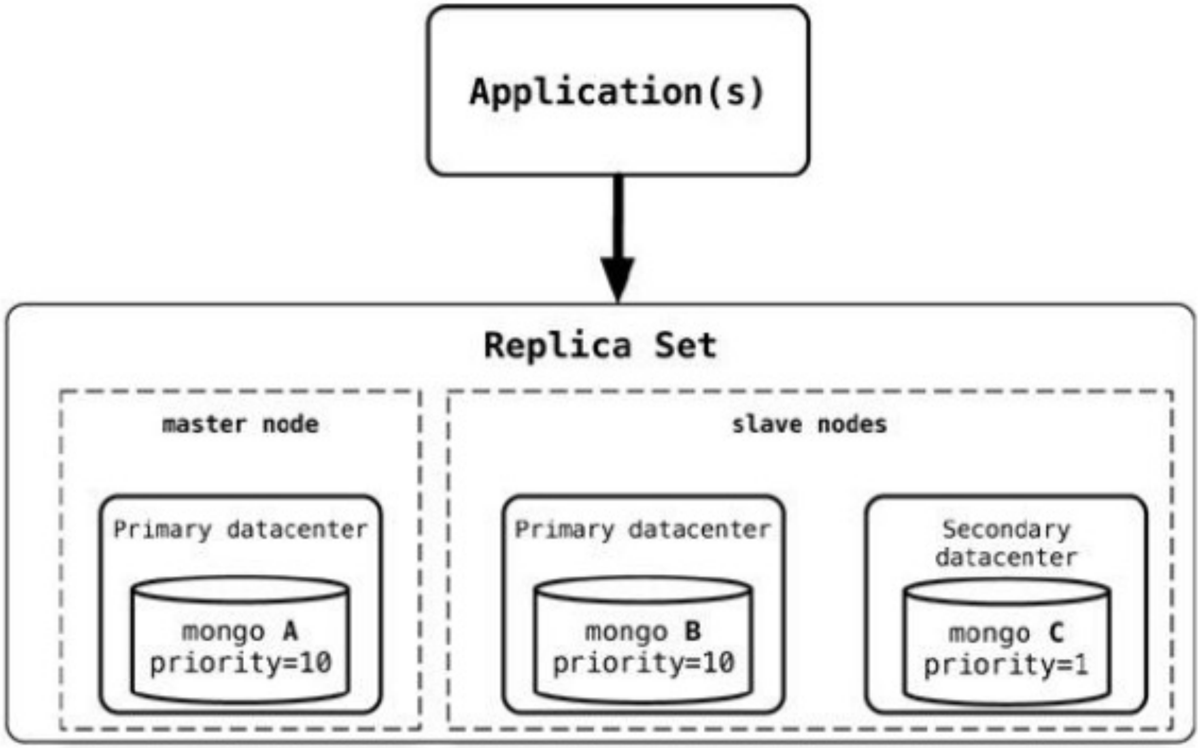
CAP in MongoDB

- **Focus on Consistency and Partition tolerance**
- Consistency
  - all replicas contain the same version of the data
- Partition tolerance
  - multiple entry points
  - system remains operational on system split
- Availability
  - system remains operational on failing nodes
  - traded off with consistency



Replication

- A MongoDB database makes use of replica sets for consistency and availability following a master-slave approach

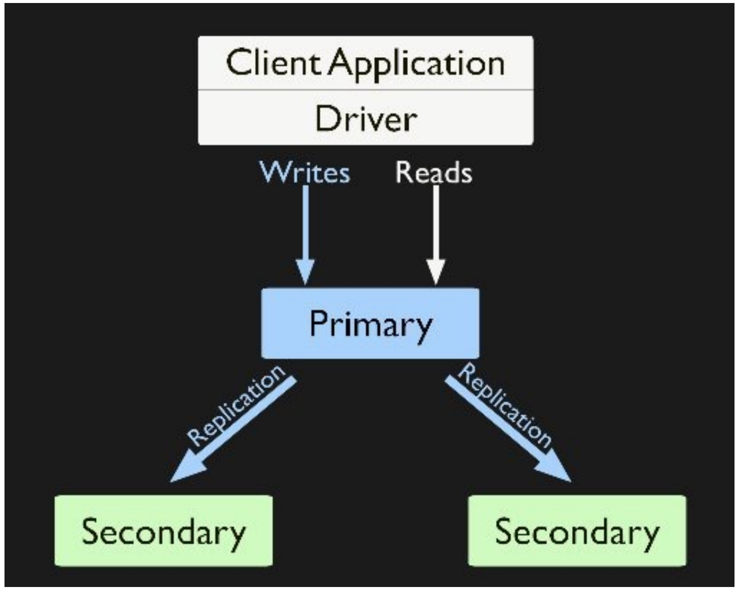


Replica sets

- The replica-set nodes elect the master, or primary, among themselves
  - The one closer to the other servers or having more RAM
  - Users can affect this by assigning a priority to a node
- All requests go to the master node
- Data is replicated to the slave nodes and the clients can get to the data even when the primary node is down
- If the master node goes down, the remaining nodes in the replica set vote among themselves to elect a new master

Replication & fault tolerance

- Primary accepts all read and write operations
- Secondaries only accepts read operations not write



- Secondaries replicate the primary's oplog and apply the operations to their data sets asynchronously
  - With a replication lag

- By having the secondaries' data sets reflect the primary's data set, the replica set can continue to function despite the failure of one or more members

Consistency vs availability

- Through the effective use of write concerns and read concerns, the level of consistency and availability, can be adjusted such as
  - waiting for stronger consistency guarantees, or
  - loosening consistency requirements to provide higher availability

Transactions

- All write operations in MongoDB are atomic on the level of a single document

- When a single write operation modifies multiple documents (e.g. `db.collection.updateMany()`), the modification of each document is atomic, but the operation as a whole is not atomic.

- For situations that require atomicity of reads and writes to multiple documents (in a single or multiple collections), MongoDB supports multi-document transactions

Transactions

- Atomic transactions are possible at the multi-document level since version 4.0 (2018)
- All transactions that contain read operations must use read preference primary
  - All operations in a given transaction must route to the same member
- Until a transaction commits, the data changes made in the transaction are not visible outside the transaction

Use cases

Suitable use cases

- Event Logging
  - Storing logs of events, acting as a central data store for event storage
  - Events can be sharded by
    - the application that generated the event
    - the type of the event (e.g., order_processed, customer_logged)
- Content Management Systems, Blogging Platforms
  - content management systems or applications for publishing websites, managing user comments, user registrations, profiles, web-facing documents.
- Web Analytics or Real-Time Analytics
  - store data for real-time analytics; since parts of the document can be updated, it’s very easy to store page views or unique visitors, and new metrics can be easily added without schema changes
- E-Commerce Applications
  - E-commerce applications often need to have flexible schema for products and orders, as well as the ability to evolve their data structure without expensive database refactoring or data migration

When not to use

- Complex Transactions Spanning Different Documents
  - Document data stores are not suited for atomic cross-document operations.

- Queries against Varying Aggregate Structure
  - In document databases data is saved as an aggregate in the form of application entities.
  - If the design of the aggregate is constantly changing, you need to save the aggregates at the lowest level of granularity.

References & Credits

- References:
- Kristina Chodorow, MongoDB – The definitive guide, 3rd Ed., O'Reilly, 2019

- https://docs.mongodb.com/

- Credits:
- Riccardo Torlone, Big Data, Università di Roma Tre
- Kathleen Durant, CS 3200, Northeastern University