

MongoDB

Alvin Cheung
Fall 2022



NoSQL databases

- We talked about NoSQL data models
- We will now study two implementations
 - MongoDB
 - JSON storage engine
 - Spark
 - A distributed key-value store

MongoDB: History 1

- A prototypical NoSQL database
- Short for humongous
- First version in 2009!
- Still very popular
 - IPO in 2017
 - Now worth >7B in market capital (as of 2020)



MongoDB: History 2



- Internet & social media boom led to a demand for
 - Rapid data model evolution: "a move fast and break things" mentality to system dev
 - E.g., adding a new attrib to a Facebook profile
 - Contrary to DBMS wisdom of declaring schema upfront and changing rarely (costly!)
 - Rapid txn support, even at the cost of losing some updates or non-atomicity
 - Contrary to DBMS wisdom of ACID, esp. with distribution/2PC (costly!)
- Early version centered around storing and querying json documents quickly
- Made several tradeoffs for speed
 - No joins → now support left outer joins
 - Limited query opt → still limited, but many improvements
 - No txn support apart from atomic writes to json docs → limited support for multi-doc txns
 - No checks/schema validation → now support json schema validation (rarely used!)

MongoDB: History 3



<https://www.mongodb.com/blog/post/what-about-durability>

- Most egregious: no durability or write ahead logging! *No recovery*

We get lots of questions about why MongoDB doesn't have full single server durability, and there are many people that think this is a major problem. We wanted to shed some light on why we haven't done single server durability, what our suggestions are, and our future plans.

Excuse 1:
Durability is
overrated

Excuse 2:
It's hard to
implement



Sure enough, this was fixed later (four years after the first version!)

MongoDB: History 4

Bottomline:

MongoDB has now evolved into a mature "DBMS" with some different design decisions, and relearning many of the canonical DBMS lessons

We'll focus on two primary design decisions:

- The data model
- The query language

Will discuss these two to start with, then some of the architectural issues

MongoDB Data Model

MongoDB	Relational DBMS
Database	Database
Collection	Relation
Document	Row/Record
Field	Column

Document = {..., field: value, ...}

JSON

Where value can be:

- Atomic
- A document
- An array of atomic values
- An array of documents

```
{ qty : 1, status : "D", size : {h : 14, w : 21}, tags :  
  ["a", "b"] },
```

Can also mix and match, e.g., array of atomics and documents, or array of arrays
[Same as the JSON data model]

Internally stored as BSON = Binary JSON

- Client libraries can directly operate on this natively

MongoDB Data Model 2

MongoDB	DBMS
Database	Database
Collection	Relation
Document	Row/Record
Field	Column

Document = {..., field: value, ...}

Can use JSON schema validation

- Some integrity checks, field typing and ensuring the presence of certain fields
- Rarely used, and we'll skip for our discussion

Special field in each document: `_id`

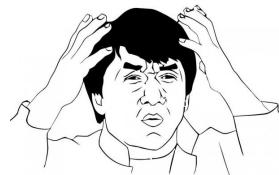
- Primary key
- Will also be indexed by default
- If it is not present during ingest, it will be added
- Will be first attribute of each doc.
- This field requires special treatment during projections as we will see later

MongoDB Query Language (MQL)

- Input = collections, output = collections
- Three main types of queries in the query language
 - Retrieval: Restricted SELECT-WHERE-ORDER BY-LIMIT type queries
 - Aggregation: A bit of a misnomer; a general pipeline of operators
 - Can capture Retrieval as a special case
 - But worth understanding Retrieval queries first...
ways to concatenate multiple retrieval operators together
 - Updates
- All queries are invoked as
 - db.collection.operation1(...).operation2(...)...
 - collection: name of collection
 - Unlike SQL which lists many tables in a FROM clause, MQL is centered around manipulating a single collection

Some MQL Principles : Dot (.) Notation

- `".` is used to drill deeper into nested docs/arrays
- Recall that a value could be atomic, a nested document, an array of atomics, or an array of nested documents
- Examples:
 - `"instock.qty"` → qty field within the instock field
 - Applies only when instock is a nested doc or an array of nested docs
 - If instock is a nested doc, then qty could be nested field
 - If instock is an array of nested docs, then qty could be a nested field within documents in the array
 - `"instock.1"` → second element within the instock array
 - Element could be an atomic value or a nested document
 - `"instock.1.qty"` → qty field within the second document within the instock array
 - Note: such dot expressions need to be in quotes



Some MQL Principles : Dollar (\$) Notation

- \$ indicates that the string is a special keyword
 - E.g., \$gt, \$lte, \$add, \$elemMatch, ...
- Used as the "field" part of a "field : value" expression
- If binary operator, it is usually written as:
 - {LOperand : { \$keyword : ROperand}}
 - e.g., {qty : {\$gt : 30}}
- Alternative: arrays
 - {\$keyword : [argument list]}
 - e.g., {\$add : [1, 2]}
- Exception: \$fieldName, used to refer to a previously defined field on the value side
 - Purpose: disambiguation
 - Only relevant for aggregation pipelines
 - Let's not worry about this for now.

Retrieval Queries Template

`db.collection.find(<predicate>, optional <projection>)`

returns documents that match <predicate>

keep fields as specified in <projection>

both <predicate> and <projection> expressed as documents
in fact, most things are documents!

`db.inventory.find({ })`

returns all documents

```
[> db.inventory.find()
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d993"), "item" : "journal", "qty" : 25, "size" : { "h" : 14, "w" : 21, "uom" : "cm" }, "status" : "A" }
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d994"), "item" : "notebook", "qty" : 50, "size" : { "h" : 8.5, "w" : 11, "uom" : "in" }, "status" : "A" }
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d995"), "item" : "paper", "qty" : 100, "size" : { "h" : 8.5, "w" : 11, "uom" : "in" }, "status" : "D" }
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d996"), "item" : "planner", "qty" : 75, "size" : { "h" : 22.85, "w" : 30, "uom" : "cm" }, "status" : "D" }
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d997"), "item" : "postcard", "qty" : 45, "size" : { "h" : 10, "w" : 15.25, "uom" : "cm" }, "status" : "A" }
```

Examples

Retrieval Queries: Basic Queries

```
[> db.inventory.find()
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d993"), "item" : "journal", "qty" : 25, "size" : { "h" : 14, "w" : 21, "uom" : "cm" }, "status" : "A" }
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d994"), "item" : "notebook", "qty" : 50, "size" : { "h" : 8.5, "w" : 11, "uom" : "in" }, "status" : "A" }
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d995"), "item" : "paper", "qty" : 100, "size" : { "h" : 8.5, "w" : 11, "uom" : "in" }, "status" : "D" }
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d996"), "item" : "planner", "qty" : 75, "size" : { "h" : 22.85, "w" : 30, "uom" : "cm" }, "status" : "D" }
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d997"), "item" : "postcard", "qty" : 45, "size" : { "h" : 10, "w" : 15.25, "uom" : "cm" }, "status" : "A" }
```

db.collection.find(<predicate>, optional <projection>)

- find({ status : "D" })
 - all documents with status D → paper, planner
- find ({ qty : {\$gte : 50} }) *greater or equal*
 - all documents with qty $\geq 50 \rightarrow$ notebook, paper, planner
- find ({ status : "D", qty : {\$gte : 50} })
 - all documents that satisfy both → paper, planner
- find({ \$or: [{ status : "D" }, { qty : { \$lt : 30 } }] })
 - all documents that satisfy either → journal, paper, planner

```
> db.inventory.find( { $or: [ { status: "D" }, { qty: { $lt: 30 } } ] } )
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d993"), "item" : "journal", "qty" : 25, "size" : { "h" : 14, "w" : 21, "uom" : "cm" }, "status" : "A" }
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d995"), "item" : "paper", "qty" : 100, "size" : { "h" : 8.5, "w" : 11, "uom" : "in" }, "status" : "D" }
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d996"), "item" : "planner", "qty" : 75, "size" : { "h" : 22.85, "w" : 30, "uom" : "cm" }, "status" : "D" }
```

Retrieval Queries: Nested Documents

```
[> db.inventory.find()
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d993"), "item" : "journal", "qty" : 25, "size" : { "h" : 14, "w" : 21, "uom" : "cm" }, "status" : "A" }
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d994"), "item" : "notebook", "qty" : 50, "size" : { "h" : 8.5, "w" : 11, "uom" : "in" }, "status" : "A" }
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d995"), "item" : "paper", "qty" : 100, "size" : { "h" : 8.5, "w" : 11, "uom" : "in" }, "status" : "D" }
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d996"), "item" : "planner", "qty" : 75, "size" : { "h" : 22.85, "w" : 30, "uom" : "cm" }, "status" : "D" }
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d997"), "item" : "postcard", "qty" : 45, "size" : { "h" : 10, "w" : 15.25, "uom" : "cm" }, "status" : "A" }
```

db.collection.find(<predicate>, optional <projection>)

No
other
way of
querying
nested
docs!

- find({ size: { h: 14, w: 21, uom: "cm" } })
 - exact match of nested document, including ordering of fields! → journal
- find ({ "size.uom" : "cm", "size.h" : {\$gt : 14} })
 - querying a nested field → planner
 - Note: when using . notation for sub-fields, expression must be in quotes
 - Also note: each argument of find is a document

Retrieval Queries: Arrays

```
> db.inventory.find()
{ "_id" : ObjectId("5fb59ab9f50b800678c0e196"), "item" : "journal", "instock" : [ { "loc" : "A", "qty" : 5 }, { "loc" : "C", "qty" : 15 } ], "tags" : [ "blank", "red" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e197"), "item" : "notebook", "instock" : [ { "loc" : "C", "qty" : 5 } ], "tags" : [ "red", "blank" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e198"), "item" : "paper", "instock" : [ { "loc" : "A", "qty" : 60 }, { "loc" : "B", "qty" : 15 } ], "tags" : [ "red", "blank", "plain" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e199"), "item" : "planner", "instock" : [ { "loc" : "A", "qty" : 40 }, { "loc" : "B", "qty" : 5 } ], "tags" : [ "blank", "red" ], "dim" : [ 22.85, 30 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e19a"), "item" : "postcard", "instock" : [ { "loc" : "B", "qty" : 15 }, { "loc" : "C", "qty" : 35 } ], "tags" : [ "blue" ], "dim" : [ 10, 15.25 ] }
```

Slightly different example dataset for Arrays and Arrays of Documents

db.collection.find(<predicate>, optional <projection>)

- `find({ tags: ["red", "blank"] })`
 - Exact match of array → notebook
- `find({ tags: "red" })`
 - If one of the elements matches red → journal, notebook, paper, planner
- `find({ tags: "red", tags: "plain" })`
 - If one matches red, one matches plain → paper
- `find({ dim: { $gt: 15, $lt: 20 } })`
 - If one element is >15 and another is < 20 → journal, notebook, paper, postcard
- `find({ dim: { $elemMatch: { $gt: 15, $lt: 20 } } })`
 - If a single element is >15 and < 20 → postcard
- `find({ "dim.1": { $gt: 25 } })`
 - If second item > 25 → planner
 - Notice again that we use quotes to when using . notation

Retrieval Queries: Arrays of Documents

```
> db.inventory.find()
{ "_id" : ObjectId("5fb59ab9f50b800678c0e196"), "item" : "journal", "instock" : [ { "loc" : "A", "qty" : 5 }, { "loc" : "C", "qty" : 15 } ], "tags" : [ "blank", "red" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e197"), "item" : "notebook", "instock" : [ { "loc" : "C", "qty" : 5 } ], "tags" : [ "red", "blank" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e198"), "item" : "paper", "instock" : [ { "loc" : "A", "qty" : 60 }, { "loc" : "B", "qty" : 15 } ], "tags" : [ "red", "blank", "plain" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e199"), "item" : "planner", "instock" : [ { "loc" : "A", "qty" : 40 }, { "loc" : "B", "qty" : 5 } ], "tags" : [ "blank", "red" ], "dim" : [ 22.85, 30 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e19a"), "item" : "postcard", "instock" : [ { "loc" : "B", "qty" : 15 }, { "loc" : "C", "qty" : 35 } ], "tags" : [ "blue" ], "dim" : [ 10, 15.25 ] }
```

db.collection.find(<predicate>, optional <projection>)

- `find({ instock: { loc: "A", qty: 5 } })`
 - Exact match of document [like nested doc/atomic array case] → journal
- `find({ "instock.qty": { $gte : 20 } })`
 - One nested doc has $\geq 20 \rightarrow$ paper, planner, postcard
- `find({ "instock.0.qty": { $gte : 20 } })`
 - First nested doc has $\geq 20 \rightarrow$ paper, planner
- `find({ "instock": { $elemMatch: { qty: { $gt: 10, $lte: 20 } } } })`
 - One doc has $20 \geq \text{qty} > 10 \rightarrow$ paper, journal, postcard
- `find({ "instock.qty": { $gt: 10, $lte: 20 } })`
 - One doc has $20 \geq \text{qty}$, another has $\text{qty} > 10 \rightarrow$ paper, journal, postcard, planner

Retrieval Queries Template: Projection

```
> db.inventory.find()
{ "_id" : ObjectId("5fb59ab9f50b800678c0e196"), "item" : "journal", "instock" : [ { "loc" : "A", "qty" : 5 }, { "loc" : "C", "qty" : 15 } ], "tags" : [ "blank", "red" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e197"), "item" : "notebook", "instock" : [ { "loc" : "C", "qty" : 5 } ], "tags" : [ "red", "blank" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e198"), "item" : "paper", "instock" : [ { "loc" : "A", "qty" : 60 }, { "loc" : "B", "qty" : 15 } ], "tags" : [ "red", "blank", "plain" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e199"), "item" : "planner", "instock" : [ { "loc" : "A", "qty" : 40 }, { "loc" : "B", "qty" : 5 } ], "tags" : [ "blank", "red" ], "dim" : [ 22.85, 30 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e19a"), "item" : "postcard", "instock" : [ { "loc" : "B", "qty" : 15 }, { "loc" : "C", "qty" : 35 } ], "tags" : [ "blue" ], "dim" : [ 10, 15.25 ] }
```

db.collection.find(<predicate>, optional <projection>)

- Use 1s to indicate fields that you want
 - Exception: _id is always present unless explicitly excluded
 - OR Use 0s to indicate fields you don't want
 - Mixing 0s and 1s is not allowed for non _id fields
 - `find({}, {item: 1})`

_id implicitly returned

```
{ "_id" : ObjectId("5fb59ab9f50b800678c0e196"), "item" : "journal" }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e197"), "item" : "notebook" }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e198"), "item" : "paper" }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e199"), "item" : "planner" }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e19a"), "item" : "postcard" }
```
 - `find({}, {item: 1, _id: 0})`

No mix of 0 and 1 in non-id field

```
{ "item" : "journal" }
{ "item" : "notebook" }
{ "item" : "paper" }
{ "item" : "planner" }
{ "item" : "postcard" }
```
- `find({}, {item: 1, tags: 0, _id: 0})`
- Error: error: {
"ok" : 0,
"errmsg" : "Cannot do exclusion on field tags in inclusion projection",
"code" : 31254,
"codeName" : "Location31254" }
- `find({}, {item: 1, "instock.loc": 1, _id: 0})`
- ```
{ "item" : "journal", "instock" : [{ "loc" : "A" }, { "loc" : "C" }] }
{ "item" : "notebook", "instock" : [{ "loc" : "C" }] }
{ "item" : "paper", "instock" : [{ "loc" : "A" }, { "loc" : "B" }] }
{ "item" : "planner", "instock" : [{ "loc" : "A" }, { "loc" : "B" }] }
{ "item" : "postcard", "instock" : [{ "loc" : "B" }, { "loc" : "C" }] }
```

# Retrieval Queries : Addendum

```
> db.inventory.find()
{ "_id" : ObjectId("5fb59ab9f50b800678c0e196"), "item" : "journal", "instock" : [{ "loc" : "A", "qty" : 5 }, { "loc" : "C", "qty" : 15 }], "tags" : ["blank", "red"], "dim" : [14, 21] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e197"), "item" : "notebook", "instock" : [{ "loc" : "C", "qty" : 5 }], "tags" : ["red", "blank"], "dim" : [14, 21] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e198"), "item" : "paper", "instock" : [{ "loc" : "A", "qty" : 60 }, { "loc" : "B", "qty" : 15 }], "tags" : ["red", "blank", "plain"], "dim" : [14, 21] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e199"), "item" : "planner", "instock" : [{ "loc" : "A", "qty" : 40 }, { "loc" : "B", "qty" : 5 }], "tags" : ["blank", "red"], "dim" : [22.85, 30] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e19a"), "item" : "postcard", "instock" : [{ "loc" : "B", "qty" : 15 }, { "loc" : "C", "qty" : 35 }], "tags" : ["blue"], "dim" : [10, 15.25] }
```

Two additional operations that are useful for retrieval:

- Limit (k) like LIMIT in SQL
  - e.g., db.inventory.find( {} ).limit(1)
- Sort ({} {}) like ORDER BY in SQL
  - List of fields, -1 indicates decreasing 1 indicates ascending
  - e.g., db.inventory.find( {}, { \_id : 0, instock : 0 } ).sort( { "dim.0": -1, item: 1 } )

```
{ "item" : "planner", "tags" : ["blank", "red"], "dim" : [22.85, 30] }
{ "item" : "journal", "tags" : ["blank", "red"], "dim" : [14, 21] }
{ "item" : "notebook", "tags" : ["red", "blank"], "dim" : [14, 21] }
{ "item" : "paper", "tags" : ["red", "blank", "plain"], "dim" : [14, 21] }
{ "item" : "postcard", "tags" : ["blue"], "dim" : [10, 15.25] }
```

# Retrieval Queries: Summary

find() = SELECT <projection>  
          FROM Collection  
          WHERE <predicate>

limit() = LIMIT

sort() = ORDER BY

db.inventory.find(  
          { tags : red },  
          {\_id : 0, instock : 0} )  
.sort ( { "dim.0": -1, item: 1 } )  
.limit (2)

          FROM  
          WHERE  
          SELECT  
          ORDER BY  
          LIMIT

# Other Features

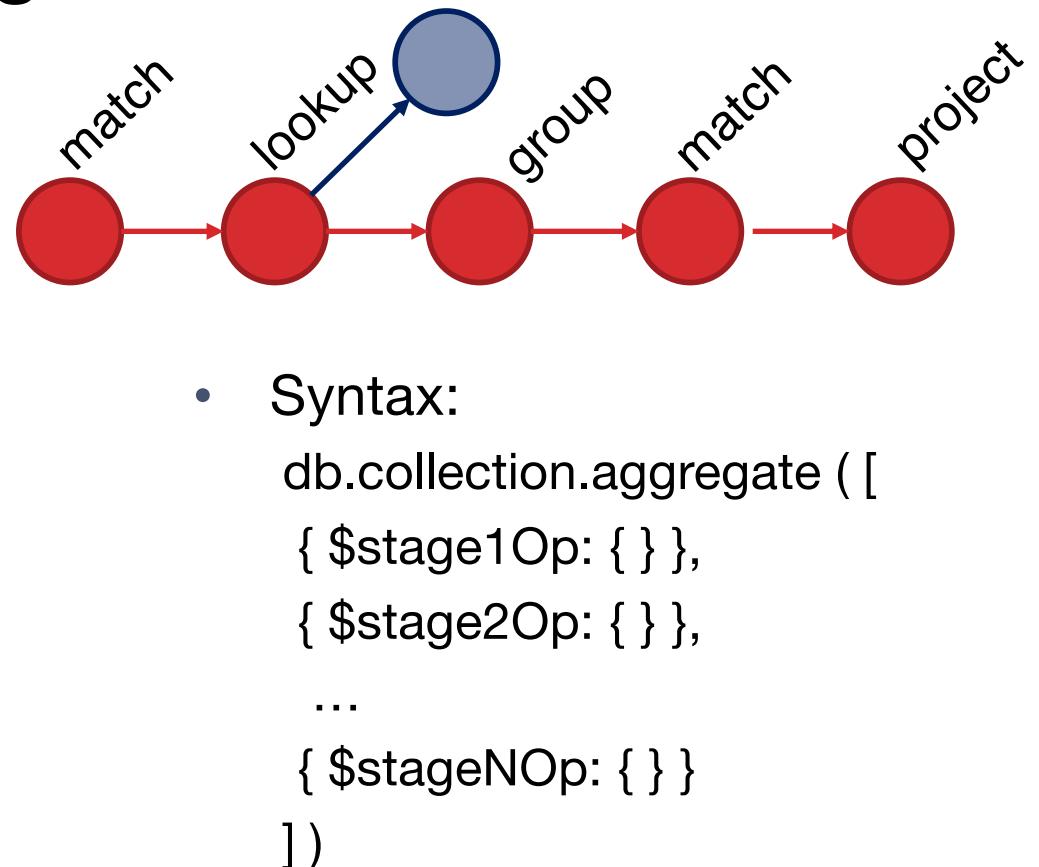
- The use of regexes for matching
- \$all : all entries in an array satisfy a condition
- \$in : checking if a value is present in an array of atomic values
- The presence or absence of fields
  - Can use special “null” values
  - {field : null} checks if a field is null or missing
  - \$exists : checking the presence/absence of a field

# MongoDB Query Language (MQL)

- Input = collections, output = collections
- Three main types of queries in the query language
  - Retrieval: Restricted SELECT-WHERE-ORDER BY-LIMIT type queries
  - **Aggregation:** A bit of a misnomer; a general pipeline of operators
    - Can capture Retrieval as a special case
    - But worth understanding Retrieval queries first...
  - Updates
- All queries are invoked as
  - db.collection.operation1(...).operation2(...)...
    - collection: name of collection
  - Unlike SQL which lists many tables in a FROM clause, MQL is centered around manipulating a single collection

# Aggregation Pipelines

- Composed of a linear *pipeline* of *stages*
- Each stage corresponds to one of:
  - match // first arg of find ()
  - project // second arg of find () but more expressiveness
  - sort/limit // same
  - group
  - unwind
  - lookup
  - ... lots more!!
- Each stage manipulates the existing collection in some way



# Next Set of Examples

```
> db.zips.find()
{ "_id" : "01022", "city" : "WESTOVER AFB", "loc" : [-72.558657, 42.196672], "pop" : 1764, "state" : "MA" }
{ "_id" : "01011", "city" : "CHESTER", "loc" : [-72.988761, 42.279421], "pop" : 1688, "state" : "MA" }
{ "_id" : "01026", "city" : "CUMMINGTON", "loc" : [-72.905767, 42.435296], "pop" : 1484, "state" : "MA" }
{ "_id" : "01020", "city" : "CHICOPEE", "loc" : [-72.576142, 42.176443], "pop" : 31495, "state" : "MA" }
{ "_id" : "01028", "city" : "EAST LONGMEADOW", "loc" : [-72.505565, 42.067203], "pop" : 13367, "state" : "MA" }
{ "_id" : "01030", "city" : "FEEDING HILLS", "loc" : [-72.675077, 42.07182], "pop" : 11985, "state" : "MA" }
{ "_id" : "01032", "city" : "GOSHEN", "loc" : [-72.844092, 42.466234], "pop" : 122, "state" : "MA" }
{ "_id" : "01012", "city" : "CHESTERFIELD", "loc" : [-72.833309, 42.38167], "pop" : 177, "state" : "MA" }
{ "_id" : "01010", "city" : "BRIMFIELD", "loc" : [-72.188455, 42.116543], "pop" : 3706, "state" : "MA" }
{ "_id" : "01034", "city" : "TOLLAND", "loc" : [-72.908793, 42.070234], "pop" : 1652, "state" : "MA" }
{ "_id" : "01035", "city" : "HADLEY", "loc" : [-72.571499, 42.36062], "pop" : 4231, "state" : "MA" }
{ "_id" : "01001", "city" : "AGAWAM", "loc" : [-72.622739, 42.070206], "pop" : 15338, "state" : "MA" }
{ "_id" : "01040", "city" : "HOLYOKE", "loc" : [-72.626193, 42.202007], "pop" : 43704, "state" : "MA" }
{ "_id" : "01008", "city" : "BLANDFORD", "loc" : [-72.936114, 42.182949], "pop" : 1240, "state" : "MA" }
{ "_id" : "01050", "city" : "HUNTINGTON", "loc" : [-72.873341, 42.265301], "pop" : 2084, "state" : "MA" }
{ "_id" : "01054", "city" : "LEVERETT", "loc" : [-72.499334, 42.46823], "pop" : 1748, "state" : "MA" }
```

One document per zipcode: 29353 zipcodes

# Grouping (with match/sort) Simple Example

```
> db.zips.find()
{ "_id" : "01022", "city" : "WESTOVER AFB", "loc" : [-72.558657, 42.196672], "pop" : 1764, "state" : "MA" }
{ "_id" : "01011", "city" : "CHESTER", "loc" : [-72.988761, 42.279421], "pop" : 1688, "state" : "MA" }
{ "_id" : "01026", "city" : "CUMMINGTON", "loc" : [-72.905767, 42.435296], "pop" : 1484, "state" : "MA" }
```

Find states with population > 15M, sort by decending order

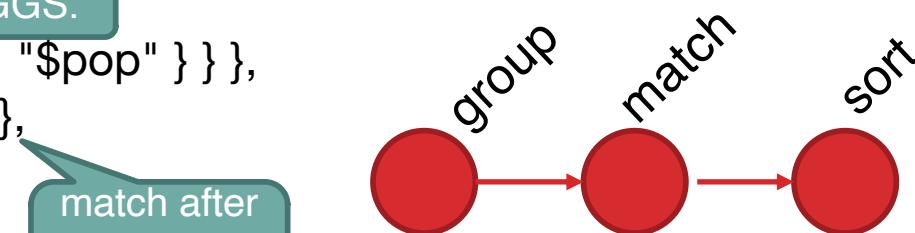
```
db.zips.aggregate([GROUP BY AGGS.
{ $group: { _id: "$state", totalPop: { $sum: "$pop" } } },
{ $match: { totalPop: { $gte: 15000000 } } },
{ $sort : { totalPop : -1 } }
])
```

```
{ "_id" : "CA", "totalPop" : 29754890 }
{ "_id" : "NY", "totalPop" : 17990402 }
{ "_id" : "TX", "totalPop" : 16984601 }
...
```

Q: what would the SQL query for this be?

match after  
group =  
HAVING

**SELECT state AS id, SUM(pop) AS totalPop  
FROM zips  
GROUP BY state  
HAVING totalPop >= 15000000  
ORDER BY totalPop DESCENDING**



# Grouping Syntax

```
$group : {
 _id: <expression>, // Group By Expression
 <field1>: { <aggfunc1> : <expression1> },
 ... }
```

Returns one document per unique group, indexed by \_id

Agg.func. can be standard ops like \$sum, \$avg, \$max

Also MQL specific ones:

- **\$first** : return the first expression value per group
  - makes sense only if docs are in a specific order [usually done after sort]
- **\$push** : create an array of expression values per group
  - didn't make sense in a relational context because values are atomic
- **\$addToSet** : like \$push, but eliminates duplicates

# Multiple Attrib. Grouping Example

```
> db.zips.find()
{ "_id" : "01022", "city" : "WESTOVER AFB", "loc" : [-72.558657, 42.196672], "pop" : 1764, "state" : "MA" }
{ "_id" : "01011", "city" : "CHESTER", "loc" : [-72.988761, 42.279421], "pop" : 1688, "state" : "MA" }
{ "_id" : "01026", "city" : "CUMMINGTON", "loc" : [-72.905767, 42.435296], "pop" : 1484, "state" : "MA" }
```

```
aggregate([
 { $group: { _id: { state: "$state", city: "$city" }, pop: { $sum: "$pop" } } },
 { $group: { _id: "$_id.state", avgCityPop: { $avg: "$pop" } } }
])
```

Q: Guesses on what this might be doing?

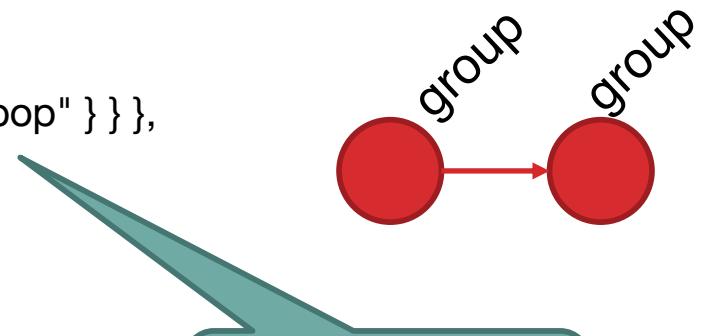
A: Find average city population per state

Group by 2 attrs,  
giving nested id

Group by  
previously  
def. id.state

Notice use of \$ to  
refer to previously  
defined fields

```
{ "_id" : "GA", "avgCityPop" : 11547.62210338681 }
{ "_id" : "WI", "avgCityPop" : 7323.00748502994 }
{ "_id" : "FL", "avgCityPop" : 27400.958963282937 }
{ "_id" : "OR", "avgCityPop" : 8262.561046511628 }
{ "_id" : "SD", "avgCityPop" : 1839.6746031746031 }
{ "_id" : "NM", "avgCityPop" : 5872.360465116279 }
{ "_id" : "MD", "avgCityPop" : 12615.775725593667 }
...
```



# Multiple Agg. Example

```
> db.zips.find()
{ "_id" : "01022", "city" : "WESTOVER AFB", "loc" : [-72.558657, 42.196672], "pop" : 1764, "state" : "MA" }
{ "_id" : "01011", "city" : "CHESTER", "loc" : [-72.988761, 42.279421], "pop" : 1688, "state" : "MA" }
{ "_id" : "01026", "city" : "CUMMINGTON", "loc" : [-72.905767, 42.435296], "pop" : 1484, "state" : "MA" }
```

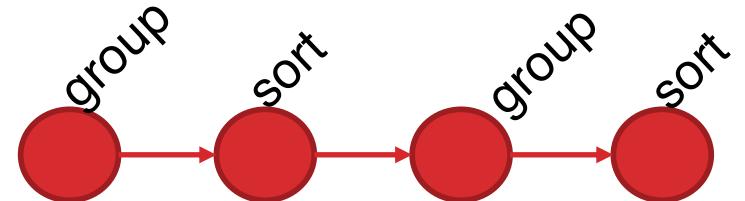
Find, for every state, the biggest city and its population

```
aggregate([
 { $group: { _id: { state: "$state", city: "$city" }, pop: { $sum: "$pop" } } },
 { $sort: { pop: -1 } },
 { $group: { _id : "$_id.state", bigCity: { $first: "$_id.city" }, bigPop: { $first: "$pop" } } },
 { $sort :{bigPop : -1} }
])
```

Approach:

- Group by pair of city and state, and compute population per city
- Order by population descending
- Group by state, and find first city and population per group (i.e., the highest population city)
- Order by population descending

```
{ "_id" : "IL", "bigCity" : "CHICAGO", "bigPop" : 2452177 }
{ "_id" : "NY", "bigCity" : "BROOKLYN", "bigPop" : 2300504 }
{ "_id" : "CA", "bigCity" : "LOS ANGELES", "bigPop" : 2102295 }
{ "_id" : "TX", "bigCity" : "HOUSTON", "bigPop" : 2095918 }
{ "_id" : "PA", "bigCity" : "PHILADELPHIA", "bigPop" : 1610956 }
{ "_id" : "MI", "bigCity" : "DETROIT", "bigPop" : 963243 }
...
```



Can list multiple aggregations after grouping id

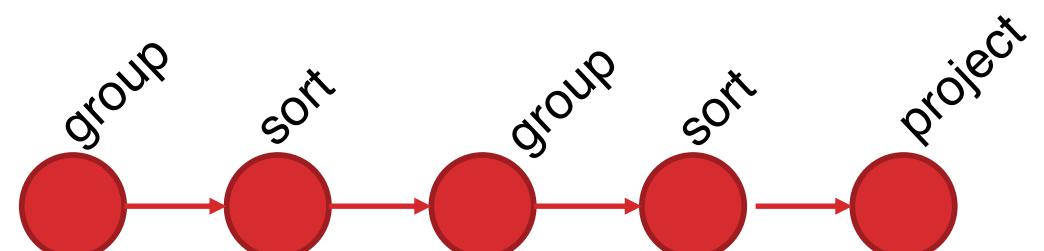
# Multiple Agg. with Vanilla Projection Example

```
> db.zips.find()
{ "_id" : "01022", "city" : "WESTOVER AFB", "loc" : [-72.558657, 42.196672], "pop" : 1764, "state" : "MA" }
{ "_id" : "01011", "city" : "CHESTER", "loc" : [-72.988761, 42.279421], "pop" : 1688, "state" : "MA" }
{ "_id" : "01026", "city" : "CUMMINGTON", "loc" : [-72.905767, 42.435296], "pop" : 1484, "state" : "MA" }
```

If we only want to keep the state and city ...

```
aggregate([
 { $group: { _id: { state: "$state", city: "$city" }, pop: { $sum: "$pop" } } },
 { $sort: { pop: -1 } },
 { $group: { _id : "$_id.state", bigCity: { $first: "$_id.city" }, bigPop: { $first: "$pop" } } },
 { $sort : {bigPop : -1} }
 { $project : {bigPop : 0} }
])
```

```
{ "_id" : "IL", "bigCity" : "CHICAGO" }
{ "_id" : "NY", "bigCity" : "BROOKLYN" }
{ "_id" : "CA", "bigCity" : "LOS ANGELES" }
{ "_id" : "TX", "bigCity" : "HOUSTON" }
{ "_id" : "PA", "bigCity" : "PHILADELPHIA" }
...
...
```



# Multiple Agg. with Adv. Projection Example

```
> db.zips.find()
{ "_id" : "01022", "city" : "WESTOVER AFB", "loc" : [-72.558657, 42.196672], "pop" : 1764, "state" : "MA" }
{ "_id" : "01011", "city" : "CHESTER", "loc" : [-72.988761, 42.279421], "pop" : 1688, "state" : "MA" }
{ "_id" : "01026", "city" : "CUMMINGTON", "loc" : [-72.905767, 42.435296], "pop" : 1484, "state" : "MA" }
```

If we wanted to nest the name of the city and population into a nested doc

```
aggregate([
 { $group: { _id: { state: "$state", city: "$city" }, pop: { $sum: "$pop" } } },
 { $sort: { pop: -1 } },
 { $group: { _id : "$_id.state", bigCity: { $first: "$_id.city" }, bigPop: { $first: "$pop" } } },
 { $sort : {bigPop : -1} },
 { $project : { _id : 0, state : "$_id", bigCityDeets: { name: "$bigCity", pop: "$bigPop" } } }
])
```

```
{ "state" : "IL", "bigCityDeets" : { "name" : "CHICAGO", "pop" : 2452177 } }
{ "state" : "NY", "bigCityDeets" : { "name" : "BROOKLYN", "pop" : 2300504 } }
{ "state" : "CA", "bigCityDeets" : { "name" : "LOS ANGELES", "pop" : 2102295 } }
{ "state" : "TX", "bigCityDeets" : { "name" : "HOUSTON", "pop" : 2095918 } }
{ "state" : "PA", "bigCityDeets" : { "name" : "PHILADELPHIA", "pop" : 1610956 } }
```

...

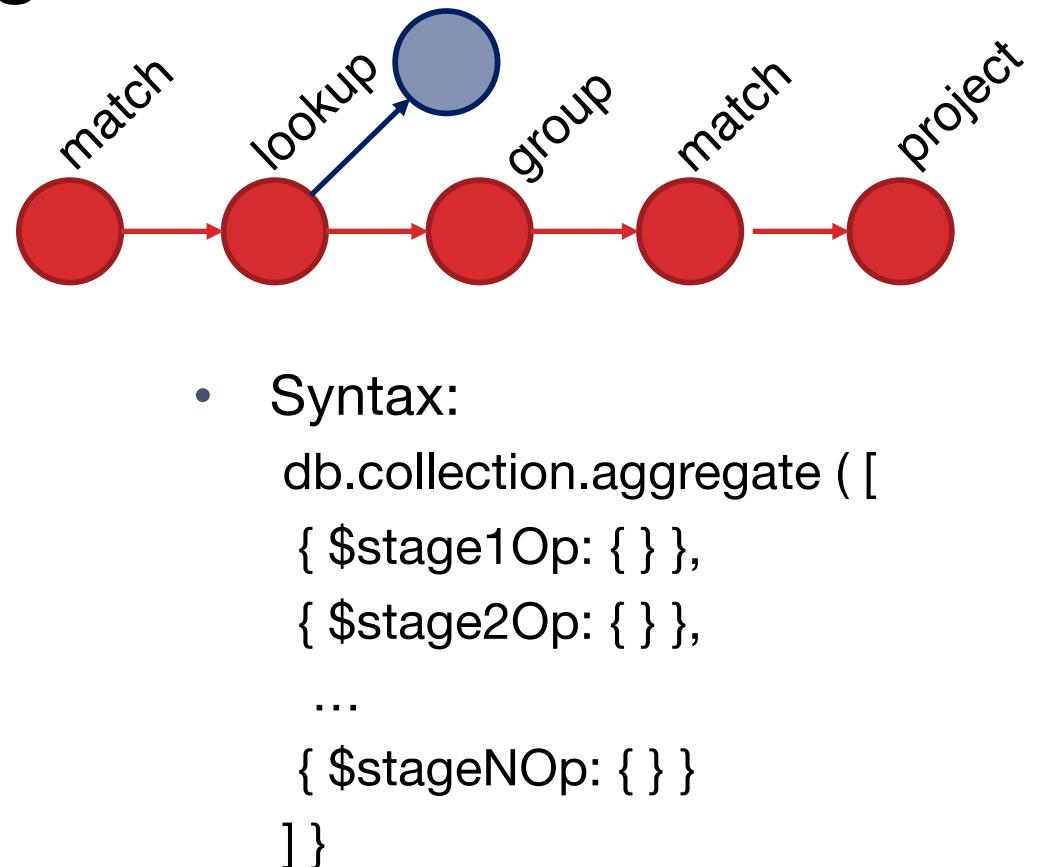
Can construct new nested documents in output, unlike vanilla projection

# Advanced Projection vs. Vanilla Projection

- In addition to excluding/including fields like in projection during retrieval (find), projection in the aggregation pipeline allows you to:
  - Rename fields
  - Redefine new fields using complex expressions on old fields
  - Reorganize fields into nestings or unnestings
  - Reorganize fields into arrays or break down arrays
- Try them at home!

# Aggregation Pipelines

- Composed of a linear *pipeline* of *stages*
- Each stage corresponds to one of:
  - match // first arg of find ()
  - project // second arg of find () but more expressiveness
  - sort/limit // same
  - group
  - **unwind**
  - **lookup**
  - ... lots more!!
- Each stage manipulates the existing collection in some way



# Unwinding Arrays

```
> db.inventory.find()
{ "_id" : ObjectId("5fb59ab9f50b800678c0e196"), "item" : "journal", "instock" : [{ "loc" : "A", "qty" : 5 }, { "loc" : "C", "qty" : 15 }], "tags" : ["blank", "red"], "dim" : [14, 21] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e197"), "item" : "notebook", "instock" : [{ "loc" : "C", "qty" : 5 }], "tags" : ["red", "blank"], "dim" : [14, 21] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e198"), "item" : "paper", "instock" : [{ "loc" : "A", "qty" : 60 }, { "loc" : "B", "qty" : 15 }], "tags" : ["red", "blank", "plain"], "dim" : [14, 21] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e199"), "item" : "planner", "instock" : [{ "loc" : "A", "qty" : 40 }, { "loc" : "B", "qty" : 5 }], "tags" : ["blank", "red"], "dim" : [22.85, 30] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e19a"), "item" : "postcard", "instock" : [{ "loc" : "B", "qty" : 15 }, { "loc" : "C", "qty" : 35 }], "tags" : ["blue"], "dim" : [10, 15.25] }
```

Unwind expands an array by constructing documents one per element of the array

Going back to our old example with an array of tags

Notice no relational analog here: no arrays so no unwinding  
[in fact, some RDBMSs do support arrays, but not in the rel. model]

```
aggregate([
 { $unwind : "$tags" },
 { $project : { _id : 0, instock: 0 } }
])
```

```
{ "item" : "journal", "tags" : "blank", "dim" : [14, 21] }
{ "item" : "journal", "tags" : "red", "dim" : [14, 21] }
{ "item" : "notebook", "tags" : "red", "dim" : [14, 21] }
{ "item" : "notebook", "tags" : "blank", "dim" : [14, 21] }
{ "item" : "paper", "tags" : "red", "dim" : [14, 21] }
{ "item" : "paper", "tags" : "blank", "dim" : [14, 21] }
{ "item" : "paper", "tags" : "plain", "dim" : [14, 21] }
{ "item" : "planner", "tags" : "blank", "dim" : [22.85, 30] }
{ "item" : "planner", "tags" : "red", "dim" : [22.85, 30] }
{ "item" : "postcard", "tags" : "blue", "dim" : [10, 15.25] }
```

# Unwind: A Common Template

```
> db.inventory.find()
{ "_id" : ObjectId("5fb59ab9f50b800678c0e196"), "item" : "journal", "instock" : [{ "loc" : "A", "qty" : 5 }, { "loc" : "C", "qty" : 15 }], "tags" : ["blank", "red"], "dim" : [14, 21] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e197"), "item" : "notebook", "instock" : [{ "loc" : "C", "qty" : 5 }], "tags" : ["red", "blank"], "dim" : [14, 21] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e198"), "item" : "paper", "instock" : [{ "loc" : "A", "qty" : 60 }, { "loc" : "B", "qty" : 15 }], "tags" : ["red", "blank", "plain"], "dim" : [14, 21] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e199"), "item" : "planner", "instock" : [{ "loc" : "A", "qty" : 40 }, { "loc" : "B", "qty" : 5 }], "tags" : ["blank", "red"], "dim" : [22.85, 30] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e19a"), "item" : "postcard", "instock" : [{ "loc" : "B", "qty" : 15 }, { "loc" : "C", "qty" : 35 }], "tags" : ["blue"], "dim" : [10, 15.25] }
```

Q: Imagine if we want to find sum of qtys across items. How would we do this?

A common recipe in MQL queries is to *unwind* and then *group by*

```
aggregate([
 { $unwind : "$instock" },
 { $group : { _id : "$item", totalqty : { $sum : "$instock.qty" } } }
])

{ "_id" : "notebook", "totalqty" : 5 }
{ "_id" : "postcard", "totalqty" : 50 }
{ "_id" : "journal", "totalqty" : 20 }
{ "_id" : "planner", "totalqty" : 45 }
{ "_id" : "paper", "totalqty" : 75 }
```

# Looking Up Other Collections

```
> db.inventory.find()
{ "_id" : ObjectId("5fb59ab9f50b800678c0e196"), "item" : "journal", "instock" : [{ "loc" : "A", "qty" : 5 }, { "loc" : "C", "qty" : 15 }], "tags" : ["blank", "red"], "dim" : [14, 21] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e197"), "item" : "notebook", "instock" : [{ "loc" : "C", "qty" : 5 }], "tags" : ["red", "blank"], "dim" : [14, 21] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e198"), "item" : "paper", "instock" : [{ "loc" : "A", "qty" : 60 }, { "loc" : "B", "qty" : 15 }], "tags" : ["red", "blank", "plain"], "dim" : [14, 21] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e199"), "item" : "planner", "instock" : [{ "loc" : "A", "qty" : 40 }, { "loc" : "B", "qty" : 5 }], "tags" : ["blank", "red"], "dim" : [22.85, 30] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e19a"), "item" : "postcard", "instock" : [{ "loc" : "B", "qty" : 15 }, { "loc" : "C", "qty" : 35 }], "tags" : ["blue"], "dim" : [10, 15.25] }
```

```
{ $lookup: {
 from: <collection to join>,
 localField: <referencing field>,
 foreignField: <referenced field>,
 as: <output array field>
}}
```

Conceptually, for each document

- find documents in other coll that join (equijoin)
  - local field must match foreign field
- place each of them in an array

Thus, a left outer equi-join, with the join results stored in an array

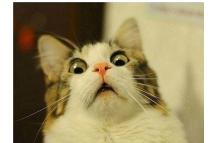
Straightforward, but kinda gross. Let's see...

Say, for each item, we want to find other items located in the same location = self-join

```
db.inventory.aggregate([
 { $lookup : {from : "inventory", localField: "instock.loc",
 foreignField: "instock.loc", as:"otheritems"}},
 { $project : {_id : 0, tags : 0, dim : 0}}
])

{ "item" : "journal", "instock" : [{ "loc" : "A", "qty" : 5 }, { "loc" : "C", "qty" : 15 }], "otheritems" : [
 { "_id" : ObjectId("5fb6f9605f0594e0227d3c24"), "item" : "journal",
 "instock" : [{ "loc" : "A", "qty" : 5 }, { "loc" : "C", "qty" : 15 }],
 "tags" : ["blank", "red"], "dim" : [14, 21] },
 { "_id" : ObjectId("5fb6f9605f0594e0227d3c25"), "item" :
 "notebook", "instock" : [{ "loc" : "C", "qty" : 5 }], "tags" : ["red",
 "blank"], "dim" : [14, 21] },
 { "_id" : ObjectId("5fb6f9605f0594e0227d3c26"), "item" : "paper",
 "instock" : [{ "loc" : "A", "qty" : 60 }, { "loc" : "B", "qty" : 15 }],
 "tags" : ["red", "blank", "plain"], "dim" : [14, 21] },
 ...
]
```

And many other records!



# Lookup... after some more projection

```
> db.inventory.find()
{ "_id" : ObjectId("5fb59ab9f50b800678c0e196"), "item" : "journal", "instock" : [{ "loc" : "A", "qty" : 5 }, { "loc" : "C", "qty" : 15 }], "tags" : ["blank", "red"], "dim" : [14, 21] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e197"), "item" : "notebook", "instock" : [{ "loc" : "C", "qty" : 5 }], "tags" : ["red", "blank"], "dim" : [14, 21] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e198"), "item" : "paper", "instock" : [{ "loc" : "A", "qty" : 60 }, { "loc" : "B", "qty" : 15 }], "tags" : ["red", "blank", "plain"], "dim" : [14, 21] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e199"), "item" : "planner", "instock" : [{ "loc" : "A", "qty" : 40 }, { "loc" : "B", "qty" : 5 }], "tags" : ["blank", "red"], "dim" : [22.85, 30] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e19a"), "item" : "postcard", "instock" : [{ "loc" : "B", "qty" : 15 }, { "loc" : "C", "qty" : 35 }], "tags" : ["blue"], "dim" : [10, 15.25] }
```

```
db.inventory.aggregate([
 { $lookup : {from:"inventory", localField:"instock.loc", foreignField:"instock.loc", as:"otheritems"}},
 {$project : { _id : 0, tags :0, dim :0, "otheritems._id":0, "otheritems.tags":0, "otheritems.dim":0,
 "otheritems.instock.qty":0}}])

{ "item" : "journal", "instock" : [{ "loc" : "A", "qty" : 5 }, { "loc" : "C", "qty" : 15 }], "otheritems" : [
 { "item" : "journal", "instock" : [{ "loc" : "A" }, { "loc" : "C" }] },
 { "item" : "notebook", "instock" : [{ "loc" : "C" }] },
 { "item" : "paper", "instock" : [{ "loc" : "A" }, { "loc" : "B" }] },
 { "item" : "planner", "instock" : [{ "loc" : "A" }, { "loc" : "B" }] },
 { "item" : "postcard", "instock" : [{ "loc" : "B" }, { "loc" : "C" }] }] }

{ "item" : "notebook", "instock" : [{ "loc" : "C", "qty" : 5 }], "otheritems" : [
 { "item" : "journal", "instock" : [{ "loc" : "A" }, { "loc" : "C" }] },
 { "item" : "notebook", "instock" : [{ "loc" : "C" }] },
 { "item" : "postcard", "instock" : [{ "loc" : "B" }, { "loc" : "C" }] }] }

...
```

# Some Rules of Thumb when Writing Queries

- `$project` is helpful if we want to construct or deconstruct nestings (in addition to removing fields or creating new ones)
- `$group` is helpful to construct arrays (using `$push` or `$addToSet`)
- `$unwind` is helpful for unwinding arrays
- `$lookup` is our only hope for joins. Be prepared for a mess. Lots of `$project` needed

# MongoDB Query Language (MQL)

- Input = collections, output = collections
- Three main types of queries in the query language
  - Retrieval: Restricted SELECT-WHERE-ORDER BY-LIMIT type queries
  - Aggregation: A bit of a misnomer; a general pipeline of operators
    - Can capture Retrieval as a special case
    - But worth understanding Retrieval queries first...
  - **Updates**
- All queries are invoked as
  - db.collection.operation1(...).operation2(...)...
    - collection: name of collection
  - Unlike SQL which lists many tables in a FROM clause, MQL is centered around manipulating a single collection

# Update Queries: InsertMany

[Insert/Delete/Update] [One/Many]

- Many is more general, so we'll discuss that instead

```
db.inventory.insertMany([
 { item: "journal", instock: [{ loc: "A", qty: 5 }, { loc: "C", qty: 15 }], tags: ["blank", "red"], dim: [14, 21] },
 { item: "notebook", instock: [{ loc: "C", qty: 5 }], tags: ["red", "blank"] , dim: [14, 21]},
 { item: "paper", instock: [{ loc: "A", qty: 60 }, { loc: "B", qty: 15 }], tags: ["red", "blank", "plain"] , dim: [14, 21]},
 { item: "planner", instock: [{ loc: "A", qty: 40 }, { loc: "B", qty: 5 }], tags: ["blank", "red"], dim: [22.85, 30] },
 { item: "postcard", instock: [{loc: "B", qty: 15}, { loc: "C", qty: 35 }], tags: ["blue"] , dim: [10, 15.25] }
]);
```

Several actions will be taken as part of this insert:

- Will create inventory collection if absent [No schema specification/DDL needed!]
- Will add the `_id` attrib to each document added (since it isn't there)
- `_id` will be the first field for each document by default

# Update Queries: UpdateMany

```
> db.inventory.find({}, {_id:0})
{ "item" : "journal", "instock" : [{ "loc" : "A", "qty" : 5 }, { "loc" : "C", "qty" : 15 }], "tags" : ["blank", "red"], "dim" : [14, 21] }
{ "item" : "notebook", "instock" : [{ "loc" : "C", "qty" : 5 }], "tags" : ["red", "blank"], "dim" : [14, 21] }
{ "item" : "paper", "instock" : [{ "loc" : "A", "qty" : 60 }, { "loc" : "B", "qty" : 15 }], "tags" : ["red", "blank", "plain"], "dim" : [14, 21] }
{ "item" : "planner", "instock" : [{ "loc" : "A", "qty" : 40 }, { "loc" : "B", "qty" : 5 }], "tags" : ["blank", "red"], "dim" : [22.85, 30] }
{ "item" : "postcard", "instock" : [{ "loc" : "B", "qty" : 15 }, { "loc" : "C", "qty" : 35 }], "tags" : ["blue"], "dim" : [10, 15.25] }
```

Syntax: updateMany ( {<condition>}, {<change>} )

```
db.inventory.updateMany (
```

```
 { "dim.0": { $lt: 15 } },
```

```
 { $set: { "dim.0": 15, status: "InvalidWidth" } }
```

```
) // if any width <15, set it to 15 and set status to InvalidWidth.
```

```
> db.inventory.find({}, {_id:0})
{ "item" : "journal", "instock" : [{ "loc" : "A", "qty" : 5 }, { "loc" : "C", "qty" : 15 }], "tags" : ["blank", "red"], "dim" : [15, 21], "status" : "InvalidWidth" }
{ "item" : "notebook", "instock" : [{ "loc" : "C", "qty" : 5 }], "tags" : ["red", "blank"], "dim" : [15, 21], "status" : "InvalidWidth" }
{ "item" : "paper", "instock" : [{ "loc" : "A", "qty" : 60 }, { "loc" : "B", "qty" : 15 }], "tags" : ["red", "blank", "plain"], "dim" : [15, 21], "status" : "InvalidWidth" }
{ "item" : "planner", "instock" : [{ "loc" : "A", "qty" : 40 }, { "loc" : "B", "qty" : 5 }], "tags" : ["blank", "red"], "dim" : [22.85, 30] }
{ "item" : "postcard", "instock" : [{ "loc" : "B", "qty" : 15 }, { "loc" : "C", "qty" : 35 }], "tags" : ["blue"], "dim" : [15, 15.25], "status" : "InvalidWidth" }
```

Analogous to: UPDATE R SET <change> WHERE <condition>

# Update Queries: UpdateMany 2

```
> db.inventory.find({}, {_id:0})
{ "item" : "journal", "instock" : [{ "loc" : "A", "qty" : 5 }, { "loc" : "C", "qty" : 15 }], "tags" : ["blank", "red"], "dim" : [14, 21] }
{ "item" : "notebook", "instock" : [{ "loc" : "C", "qty" : 5 }], "tags" : ["red", "blank"], "dim" : [14, 21] }
{ "item" : "paper", "instock" : [{ "loc" : "A", "qty" : 60 }, { "loc" : "B", "qty" : 15 }], "tags" : ["red", "blank", "plain"], "dim" : [14, 21] }
{ "item" : "planner", "instock" : [{ "loc" : "A", "qty" : 40 }, { "loc" : "B", "qty" : 5 }], "tags" : ["blank", "red"], "dim" : [22.85, 30] }
{ "item" : "postcard", "instock" : [{ "loc" : "B", "qty" : 15 }, { "loc" : "C", "qty" : 35 }], "tags" : ["blue"], "dim" : [10, 15.25] }
```

Syntax: updateMany ( {<condition>}, {<change>} )

```
db.inventory.updateMany (
 {"dim.0": { $lt: 15 } },
 { $inc: { "dim.0": 5 },
 $set: {status: "InvalidWidth"} }
 // if any width <15, increment by 5 and set status to InvalidWidth.
```

```
> db.inventory.find({}, {_id:0})
{ "item" : "journal", "instock" : [{ "loc" : "A", "qty" : 5 }, { "loc" : "C", "qty" : 15 }], "tags" : ["blank", "red"], "dim" : [19, 21], "status" : "InvalidWidth" }
{ "item" : "notebook", "instock" : [{ "loc" : "C", "qty" : 5 }], "tags" : ["red", "blank"], "dim" : [19, 21], "status" : "InvalidWidth" }
{ "item" : "paper", "instock" : [{ "loc" : "A", "qty" : 60 }, { "loc" : "B", "qty" : 15 }], "tags" : ["red", "blank", "plain"], "dim" : [19, 21], "status" : "InvalidWidth" }
{ "item" : "planner", "instock" : [{ "loc" : "A", "qty" : 40 }, { "loc" : "B", "qty" : 5 }], "tags" : ["blank", "red"], "dim" : [22.85, 30] }
{ "item" : "postcard", "instock" : [{ "loc" : "B", "qty" : 15 }, { "loc" : "C", "qty" : 35 }], "tags" : ["blue"], "dim" : [15, 15.25], "status" : "InvalidWidth" }
```

Analogous to: UPDATE R SET <change> WHERE <condition>

# MongoDB Query Language (MQL)

- Input = collections, output = collections
- Three main types of queries in the query language
  - Retrieval: Restricted SELECT-WHERE-ORDER BY-LIMIT type queries
  - Aggregation: A bit of a misnomer; a general pipeline of operators
    - Can capture Retrieval as a special case
    - But worth understanding Retrieval queries first...
  - Updates
- All queries are invoked as
  - db.collection.operation1(...).operation2(...)...
    - collection: name of collection
  - Unlike SQL which lists many tables in a FROM clause, MQL is centered around manipulating a single collection

# MongoDB: History 4

Bottomline:

*MongoDB has now evolved into a mature "DBMS" with some different design decisions, and relearning many of the canonical DBMS lessons*

We'll focus on two primary design decisions:

- The data model
- The query language

Will discuss these two to start with, then **some of the architectural issues**

# MongoDB Internals

- MongoDB is a distributed NoSQL database
- Collections are partitioned/sharded based on a field [range-based]
  - Each partition stores a subset of documents
- Each partition is replicated to help with failures
  - The replication is done asynchronously
  - Failures of the main partition that haven't been propagated will be lost
- Limited heuristic-based query optimization (will discuss later)
- Atomic writes to documents within collections by default. Multi-document txns are discouraged (but now supported).

# MongoDB Internals

- Weird constraint: intermediate results of aggregations must not be too large (100MB)
  - Else will end up spilling to disk
  - Not clear if they perform any pipelining across aggregation operators
- Optimization heuristics
  - Will use indexes for \$match if early in the pipeline [user can explicitly declare]
  - \$match will be merged with other \$match if possible
    - Selection fusion
  - \$match will be moved early in the pipeline sometimes
    - Selection pushdown
    - But: not done always (e.g., not pushed before \$lookup)
  - No cost-based optimization as far as we know

# MongoDB: Summary

Bottomline:

*MongoDB has now evolved into a mature "DBMS" with some different design decisions, and relearning many of the canonical DBMS lessons*

MongoDB has a flexible data model and a powerful (if confusing) query language.

Many of the internal design decisions as well as the query & data model can be understood when compared with DBMSs

- DBMSs provide a "gold standard" to compare against.
- In the "wild" you'll encounter many more NoSQL systems, and you'll need to do the same thing that we did here!