Review: NoSQL

Alvin Cheung

Spring 2023

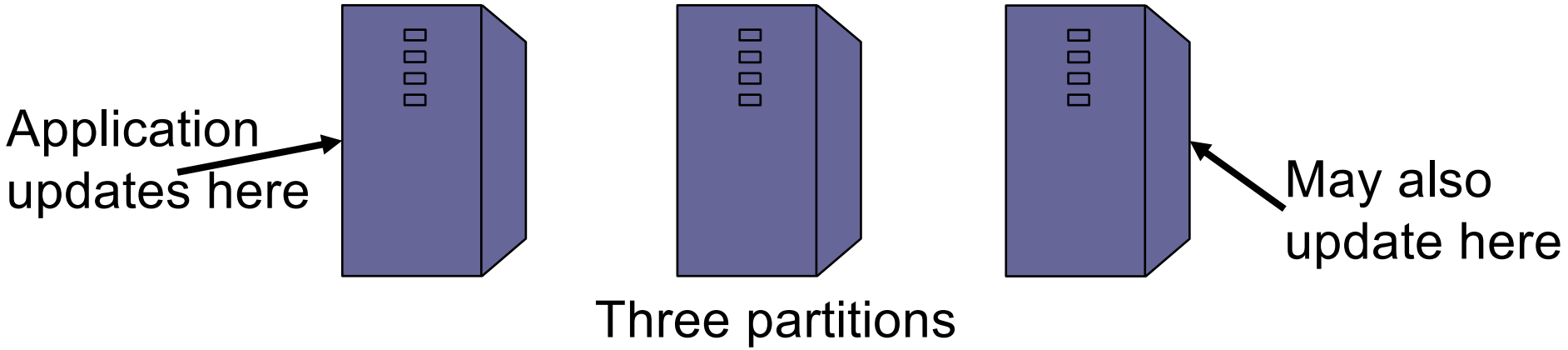
Replicating the Database

- Two basic approaches:
  - Scale up through partitioning – “sharding”
  - Scale up through replication

- Consistency is much harder to enforce

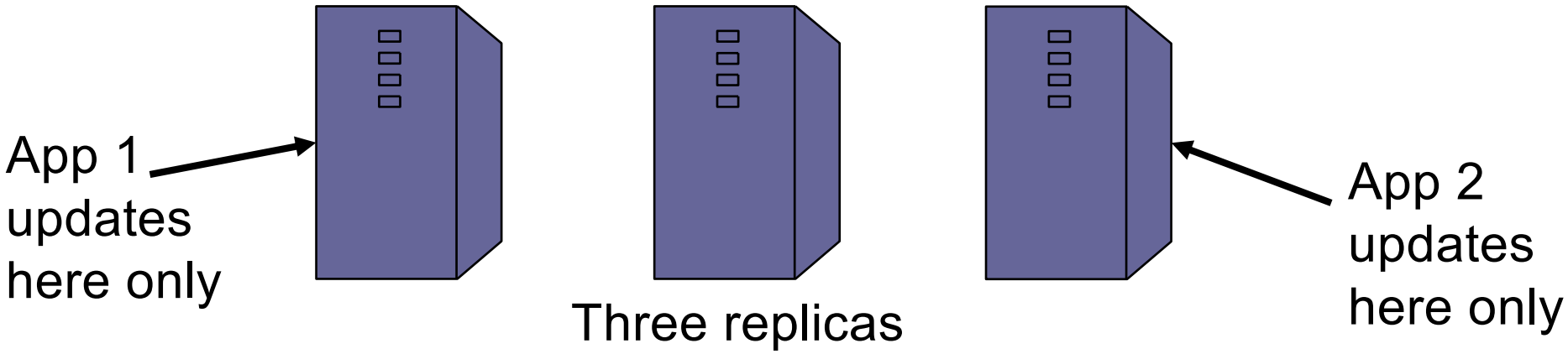
Scale Through Partitioning

- Partition the database across many machines in a cluster
  - Database now fits in main memory
  - Queries spread across these machines
- Can increase throughput
- Easy for writes but reads become expensive!



Scale Through Replication

- Create multiple copies of each database partition
- Spread queries across these replicas
- Can increase throughput and lower latency
- Can also improve fault-tolerance
- Easy for reads but writes become expensive!



Relational Model → NoSQL

- Relational DB: difficult to replicate/partition. E.g., `Supplier(sno,…),Part(pno,…),Supply(sno,pno)`
  - Partition: we may be forced to join across servers
  - Replication: local copy has inconsistent versions
  - Consistency is hard in both cases (why?)

- NoSQL: simplified data model
  - Given up on functionality
  - Application must now handle joins and consistency

TANSTAAFL!

Chem 1A

- Relational DB
  - **A**tomicity
  - **C**onsistency
  - **I**solation
  - **D**urability
- NoSQL
  - **B**asic **A**vailability
    - Application must handle partial failures itself
  - **S**oft State
    - DB state can change even without inputs
  - **E**ventually Consistency
    - DB will “eventually” become consistent
- i.e., ACID vs BASE

Data Models

Taxonomy based on data models:

- Key-value stores
  - e.g., Amazon Dynamo, Voldemort, Memcached
- Extensible Record Stores
  - e.g., HBase, Cassandra, PNUTS
- Document stores
  - e.g., SimpleDB, CouchDB, MongoDB

Key-Value Stores Features

- **Data model**: (key,value) pairs
  - Key = string/integer, unique for the entire data
  - Value = can be anything (very complex object)
- **Operations**
  - `get(key)`, `put(key,value)`
  - Operations on value not supported
- **Distribution / Partitioning** – w/ hash function
  - No replication: key k is stored at server h(k)
  - Multi-way replication: e.g., key k stored at h1(k),h2(k),h3(k)

Example

Flights(fid, date, carrier, flight_num, origin, dest, ...)
Carriers(cid, name)

- How would you represent the Flights data as key, value pairs?
- Option 1: key=fid, value=entire flight record
- Option 2: key=date, value=all flights that day
- Option 3: key=(origin,dest), value=all flights between

Key-Value Stores Internals

- Partitioning:
  - Use a hash function h
  - Store every (key,value) pair on server h(key)
- Replication:
  - Store each key on (say) three servers
  - On update, propagate change to the other servers; *eventual consistency*
  - Issue: when an app reads one replica, it may be stale
- Usually: combine partitioning+replication

Data Models

Taxonomy based on data models:

- Key-value stores
  - e.g., Amazon Dynamo, Voldemort, Memcached
- Extensible Record Stores
  - e.g., HBase, Cassandra, PNUTS
- Document stores
- ☞ e.g., SimpleDB, CouchDB, MongoDB

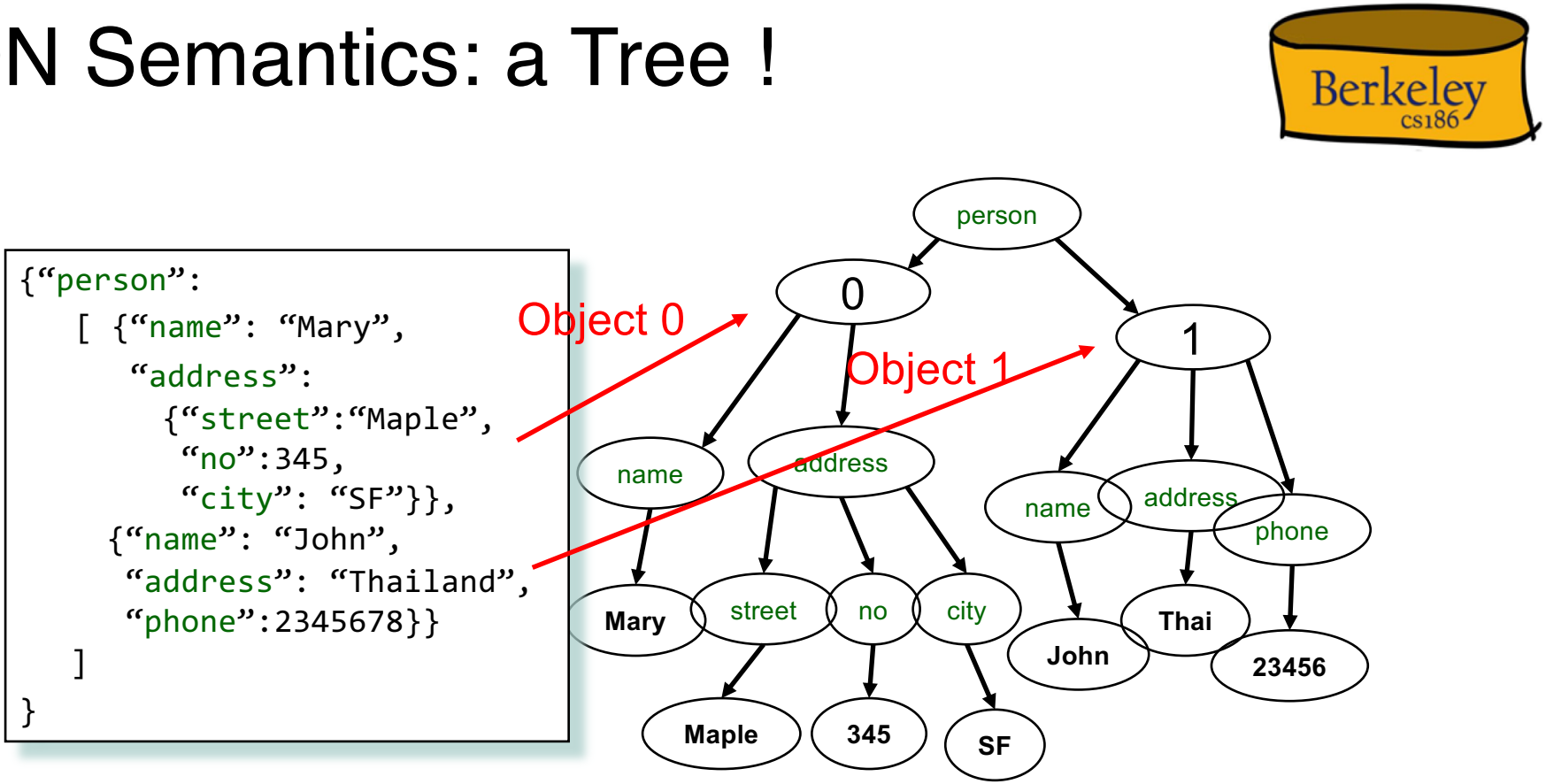
JSON Syntax


```
{  "book": [
      {"id":"01",
       "language": "Java",
       "author": "H. Javeson",
       "year": 2015
      },
      {"id":"07",
       "language": "C++",
       "edition": "second"
       "author": "E. Sepp",
       "price": 22.25
      }
   ]
}
```

JSON Types

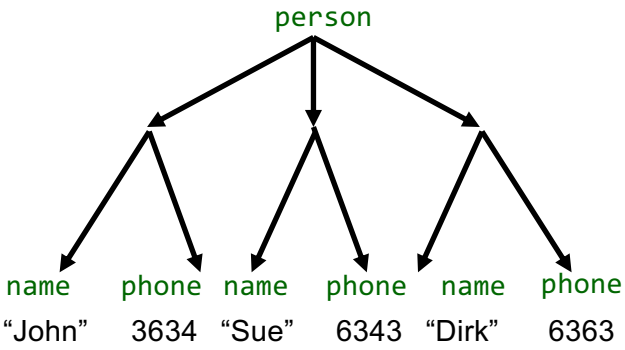
- Primitive: number, string, Boolean, null
- Object: collection of name-value pairs:
  - `{“name1”: value1, “name2”: value2, …}`
  - “`name`” is also called a “key”
- Array: *ordered* list of values:
  - `[obj1, obj2, obj3, ...]`

JSON Semantics: a Tree !



Recall: arrays are *ordered* in JSON!

Mapping Relational Data to JSON



Person

name	phone
John	3634
Sue	6343
Dirk	6363

```
{“person”: [
   {“name”: “John”, “phone”:3634},
   {“name”: “Sue”,  “phone”:6343},
    {“name”: “Dirk”, “phone”:6383}
   ]
}
```

Mapping Relational Data to JSON

May inline multiple relations based on foreign keys

Person

name	phone
John	3634
Sue	6343

Orders

personName	date	product
John	2002	Gizmo
John	2004	Gadget
Sue	2002	Gadget

```
{“Person”:
 [{"name": "John",
   "phone":3646,
   "Orders":[
    {"date":2002,"product":"Gizmo"},
    {“date”:2004,"product":"Gadget"}
   ]
  },
  {"name": "Sue",
   "phone":6343,
   "Orders":[
   {"date":2002,"product":"Gadget"}
   ]
  }
 ]
}
```

Mapping Relational Data to JSON

Many-many relationships are more difficult to represent

Person

name	phone
John	3634
Sue	6343

Product

prodName	price
Gizmo	19.99
Phone	29.99
Gadget	9.99

Orders

personName	date	product
John	2002	Gizmo
John	2004	Gadget
Sue	2002	Gadget

Options for the JSON file:

- 3 flat relations:
  Person, Orders, Product
- Person→Orders→Products
  products are duplicated
- Product→Orders→Person
  persons are duplicated

Mapping Semi-structured Data to Relations

- Missing attributes:

```
{“person”:
     [{“name”:“John”, “phone”:1234},
      {“name”:“Joe”}]
}
```

no phone !

- Could represent in a table with nulls

name	phone
John	1234
Joe	NULL

Mapping Semi-structured Data to Relations

- Repeated attributes

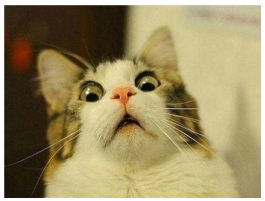
```
{“person”:
     [{“name”:”John”, “phone”:1234},
      {“name”:”Mary”, “phone”:[1234,5678]}]
}
```

Two phones !

- Impossible in one table:

name	phone	
Mary	2345	3456

???



Mapping Semi-structured Data to Relations

- Attributes with different types in different objects

```
{“person”:
     [{“name”:“Sue”, “phone”:3456},
      {“name”:{“first”:“John”, “last”:“Smith”},“phone”:2345}
     ]
}
```

Structured name !

- Nested collections
- Heterogeneous collections

- These are difficult to represent in the relational model

MongoDB

MongoDB Data Model

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

Document = {…, field: value, …}

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, …}

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
  - 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 instock
    - Applies only when instock is a nested doc or an array of nested docs
      - If instock is a nested doc or object, 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
- So if it is a binary operator, it is *usually* done as:
  - {LOperand : { $keyword : ROperand}}
  - e.g., {qty : {$gt : 30}}
- Alternative: arrays
  - {$keyword : [argument list]}
  - e.g., {$add : [1, 2]}

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" }
```

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>)

- 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: binary operator handled via a nested document

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 >= 20 ➔ paper, planner, postcard
- find({ "instock.0.qty": { $gte : 20 } })
  - First nested doc has >= 20 ➔ paper, planner
- find({ "instock": { $elemMatch: { qty: { $gt: 10, $lte: 20 } } } })
  - One doc has 20 >= qty >10➔ paper, journal, postcard
- find({ "instock.qty": { $gt: 10, $lte: 20 } })
  - One doc has 20 >= qty, another has qty>10 ➔ 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" : 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})

```
{ "item" : "journal" }
{ "item" : "notebook" }
{ "item" : "paper" }
{ "item" : "planner" }
{ "item" : "postcard" }
```

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

MongoDB Query Language (MQL)

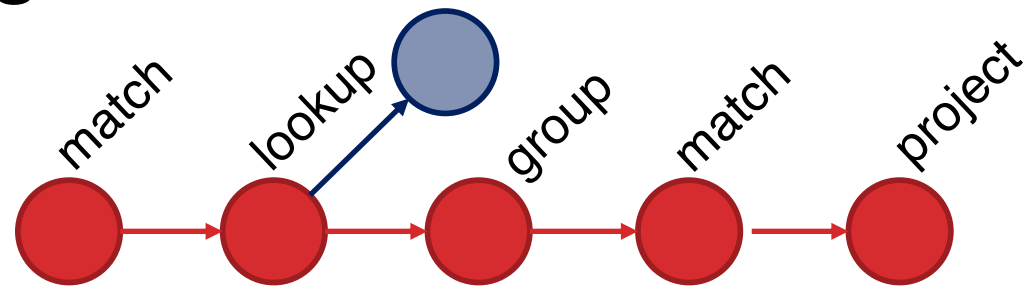
- Input = collections, output = collections

- 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
  - 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



- Syntax:

```
db.collection.aggregate ( [
 { $stage1Op: { } },
 { $stage2Op: { } },
  …
 { $stageNOp: { } }
] )
```

Grouping (with match/sort) 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" }
```

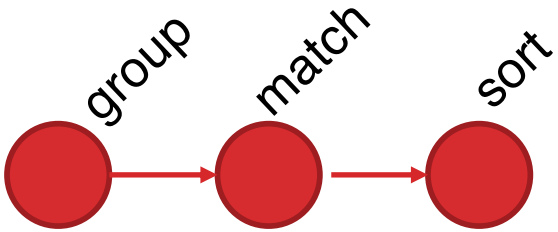
Find states with population > 15M, sort by decending order

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

GROUP BY

AGGS.

match after group = HAVING




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

Q: what would the SQL query for this be?

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

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 attribute 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>

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
- Atomic writes to documents within collections by default. Multi-document txns are discouraged (but now supported).

Make sure you understand these!

- Why NoSQL
- ACID vs BASE
- Compare and contrast relational vs various NoSQL data models
- JSON
- MQL