Introduction to NoSQL with MongoDB

Imanuel Portalatin

Part 1

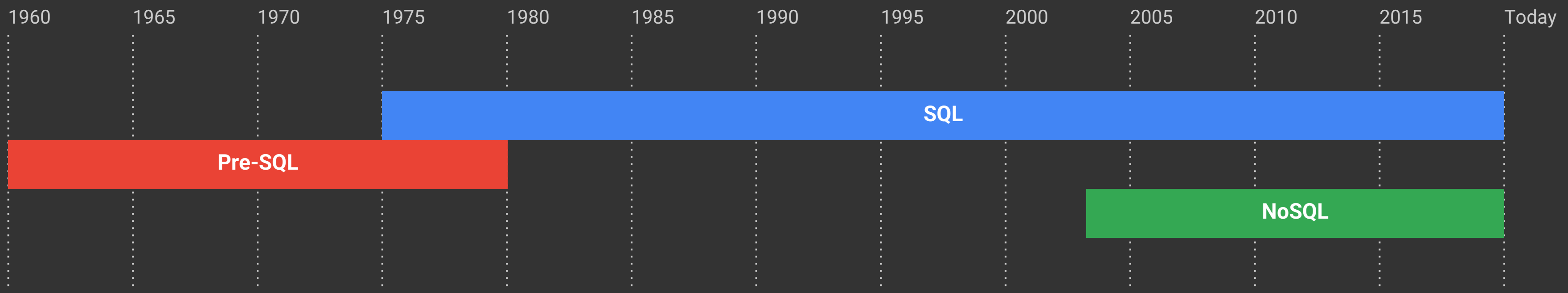
Introduction and Overview

NoSQL Databases

- Originally referring to “non SQL” or “non relational” databases
  - As NoSQL databases (like MongoDB) support more SQL-like features, the term has been redefined as "not only SQL”
- Used to denote mechanism for storage and retrieval of data modeled in non-tabular relations
- Motivations:
  - Simplicity of design
  - Simpler "horizontal" scaling to clusters of machines
  - Finer control over availability
- NoSQL stores may trade consistency (in the sense of the CAP theorem) in favor of availability, partition tolerance, and speed

Source: https://en.wikipedia.org/wiki/NoSQL

Brief History of SQL/NoSQL



- Programming and data are tightly integrated
- One kind of developer
- More control

- Strong data model
- *Two* kinds of developers
- Invariants **first** (ACID)

- Back to tight integration
- One kind of programmer
- Both models make sense

**From: http://www.slideshare.net/mongodb/mongodb-world-2016-mongodb-google-cloud

NoSQL Database Types

- Key-value stores
  - Simplest type of NoSQL database
  - Every item is stored as an attribute name (or 'key'), together with its value
  - Some key-value stores, such as Redis, allow each value to have a type, such as 'integer', which adds functionality
  - Examples include Riak and Berkeley DB
- Document databases
  - Pair each key with a complex data structure known as a document
  - Documents can contain many different key-value pairs, or key-array pairs, or even nested documents
- Graph stores
  - Used to store information about networks of data, such as social connections
  - Examples include Neo4J and Giraph.
- Wide-column stores
  - Optimized for queries over large datasets
  - Store columns of data together, instead of rows
  - Examples include Cassandra and HBase

What is MongoDB?

- From the MongoDB marketing material…
  - MongoDB is a “source available” free to use document-oriented database
  - Designed with scalability, flexibility and developer agility in mind
  - Data is stored in JSON-like documents instead of tables and rows
  - Supports ad hoc queries, indexing, and real time aggregation for analytics
  - Relies on flexible schemas that can evolve over time
  - Distributed architecture with built-in features to support high availability, horizontal scaling, and geographic distribution

https://www.mongodb.com/what-is-mongodb

RDBMS to MongoDB Translation

RDBMS	MongoDB
Database	Database
Table	Collection
Row	Document
Index	Index
JOIN	Embedded Document or Reference***

***NOTE: Aggregation pipeline now includes “lookup” stage, which is essentially a JOIN

Source: https://www.mongodb.com/blog/post/thinking-documents-part-1

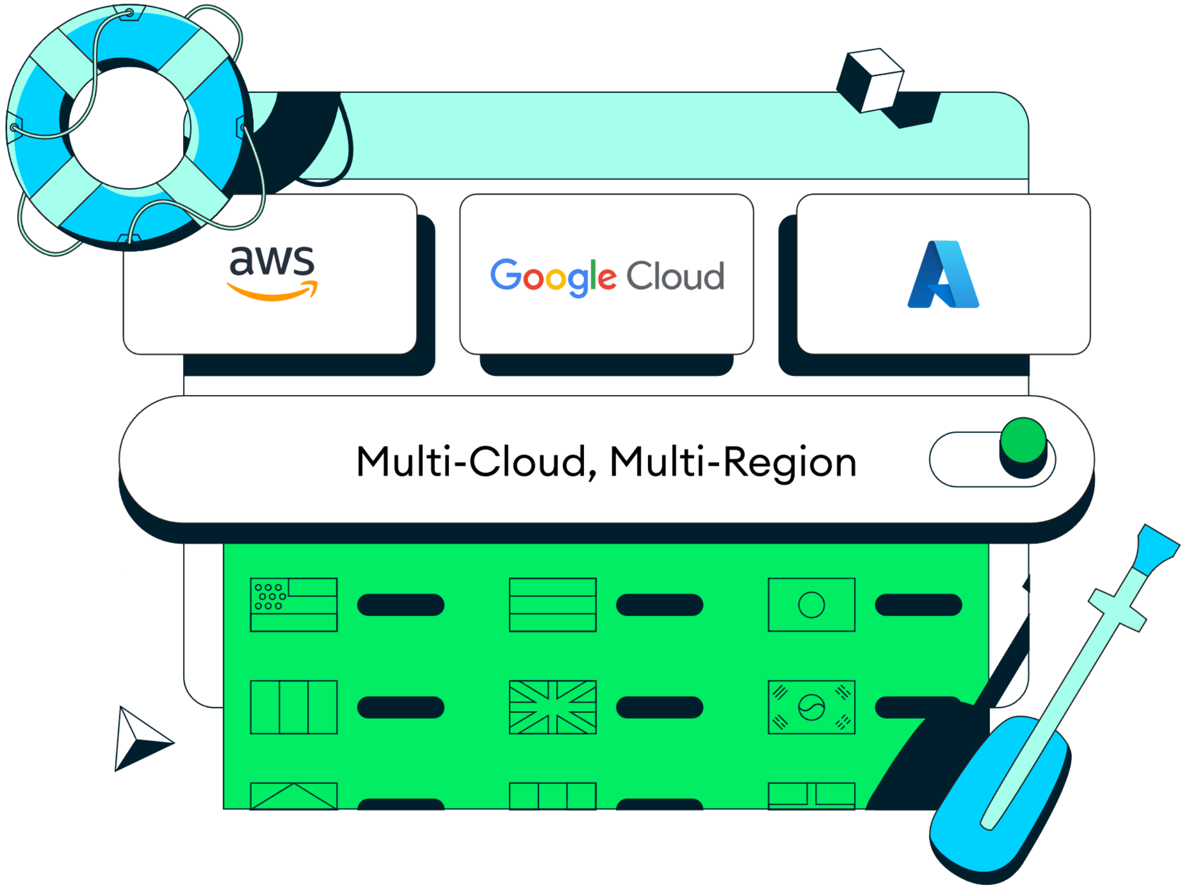
MongoDB Deployment Options

- MongoDB is “free to use”
  - Versions released prior to October 16, 2018 published under GNU AGPL v3.0.
  - Later versions are published under Server Side Public License (SSPL) v1.
    - Includes patch fixes for earlier versions
- Available environments:
  - MongoDB Atlas: Fully managed service for MongoDB cloud deployments
  - MongoDB Enterprise: Subscription-based, self-managed version of MongoDB
  - MongoDB Community: Source-available, free-to-use, and self-managed version of MongoDB

Install MongoDB Community Edition

- Installation Options
  - On “bare metal” or virtual server (I.e. AWS EC2):
    - Linux: https://docs.mongodb.com/manual/administration/install-on-linux/
    - macOS: https://docs.mongodb.com/manual/tutorial/install-mongodb-on-os-x/
    - Windows: https://docs.mongodb.com/manual/tutorial/install-mongodb-on-windows/
  - From official MongoDB server community Docker container image:
    - https://www.mongodb.com/docs/manual/tutorial/install-mongodb-community-with-docker/#std-label-docker-mongodb-community-install
  - Using Kubernetes operator with MongoDB Enterprise Docker image:
    - https://www.mongodb.com/docs/kubernetes-operator/master/kind-quick-start/

MongoDB Atlas

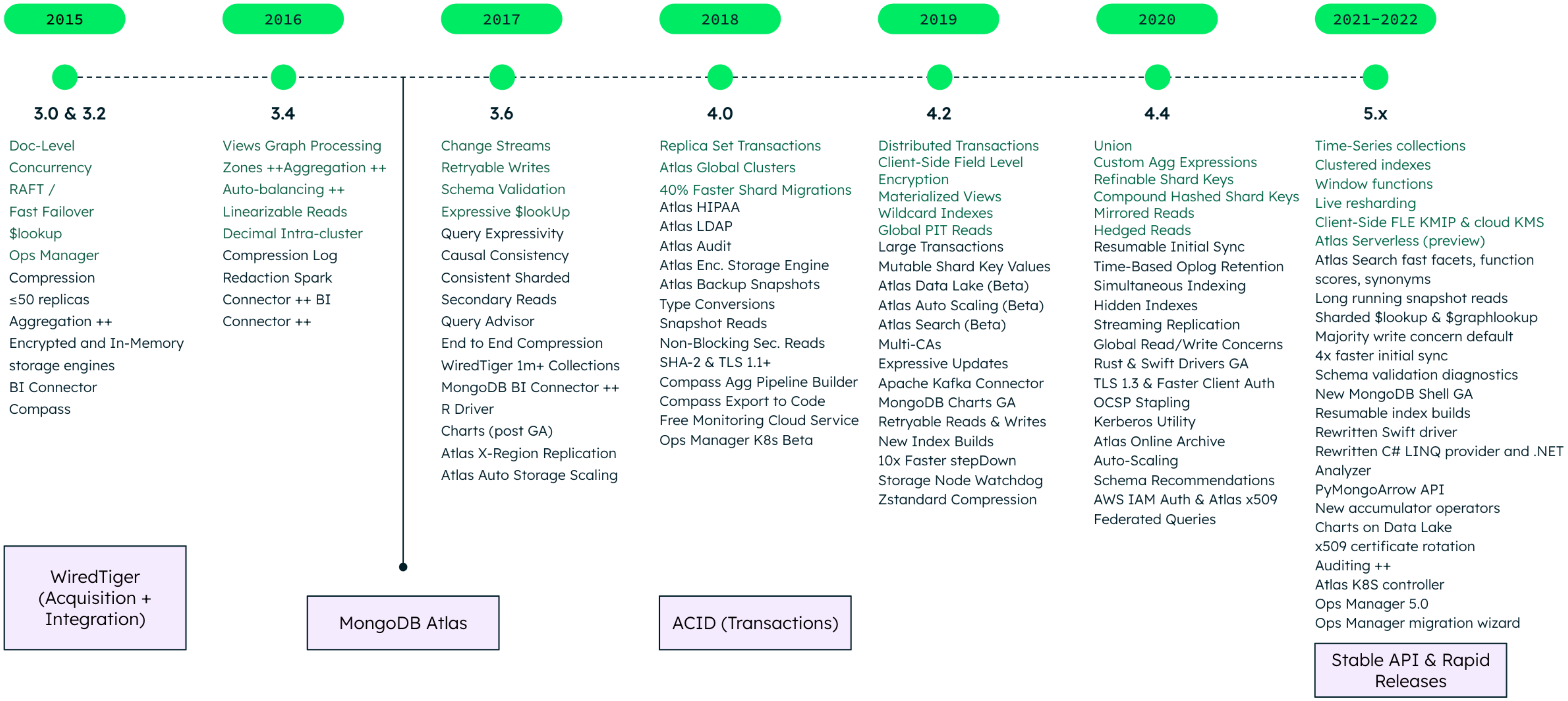


- Multi-cloud database service developed by MongoDB
- Simplifies on-demand deployment and management of document databases
- Free tier available

References:

- https://www.mongodb.com/docs/atlas/
- https://www.mongodb.com/docs/atlas/data-federation/config/config-aws-s3/

MongoDB Evolution



2015

3.0 & 3.2

Doc-Level Concurrency
RAFT / Fast Failover
$lookup
Ops Manager
Compression
≤50 replicas
Aggregation ++
Encrypted and In-Memory storage engines
BI Connector
Compass

WiredTiger (Acquisition + Integration)

2016

3.4

Views Graph Processing
Zones ++Aggregation ++
Auto-balancing ++
Linearizable Reads
Decimal Intra-cluster
Compression Log
Redaction Spark
Connector ++ BI
Connector ++

MongoDB Atlas

2017

3.6

Change Streams
Retryable Writes
Schema Validation
Expressive $lookUp
Query Expressivity
Causal Consistency
Consistent Sharded
Secondary Reads
Query Advisor
End to End Compression
WiredTiger 1m+ Collections
MongoDB BI Connector ++
R Driver
Charts (post GA)
Atlas X-Region Replication
Atlas Auto Storage Scaling

2018

4.0

Replica Set Transactions
Atlas Global Clusters
40% Faster Shard Migrations
Atlas HIPAA
Atlas LDAP
Atlas Audit
Atlas Enc. Storage Engine
Atlas Backup Snapshots
Type Conversions
Snapshot Reads
Non-Blocking Sec. Reads
SHA-2 & TLS 1.1+
Compass Agg Pipeline Builder
Compass Export to Code
Free Monitoring Cloud Service
Ops Manager K8s Beta

ACID (Transactions)

2019

4.2

Distributed Transactions
Client-Side Field Level Encryption
Materialized Views
Wildcard Indexes
Global PIT Reads
Large Transactions
Mutable Shard Key Values
Atlas Data Lake (Beta)
Atlas Auto Scaling (Beta)
Atlas Search (Beta)
Multi-CAs
Expressive Updates
Apache Kafka Connector
MongoDB Charts GA
Retryable Reads & Writes
New Index Builds
10x Faster stepDown
Storage Node Watchdog
Zstandard Compression

2020

4.4

Union
Custom Agg Expressions
Refinable Shard Keys
Compound Hashed Shard Keys
Mirrored Reads
Hedged Reads
Resumable Initial Sync
Time-Based Oplog Retention
Simultaneous Indexing
Hidden Indexes
Streaming Replication
Global Read/Write Concerns
Rust & Swift Drivers GA
TLS 1.3 & Faster Client Auth
OCSP Stapling
Kerberos Utility
Atlas Online Archive
Auto-Scaling
Schema Recommendations
AWS IAM Auth & Atlas x509
Federated Queries

2021-2022

5.x

Time-Series collections
Clustered indexes
Window functions
Live resharding
Client-Side FLE KMIP & cloud KMS
Atlas Serverless (preview)
Atlas Search fast facets, function scores, synonyms
Long running snapshot reads
Sharded $lookup & $graphlookup
Majority write concern default
4x faster initial sync
Schema validation diagnostics
New MongoDB Shell GA
Resumable index builds
Rewritten Swift driver
Rewritten C# LINQ provider and .NET Analyzer
PyMongoArrow API
New accumulator operators
Charts on Data Lake
x509 certificate rotation
Auditing ++
Atlas K8S controller
Ops Manager 5.0
Ops Manager migration wizard

Stable API & Rapid Releases

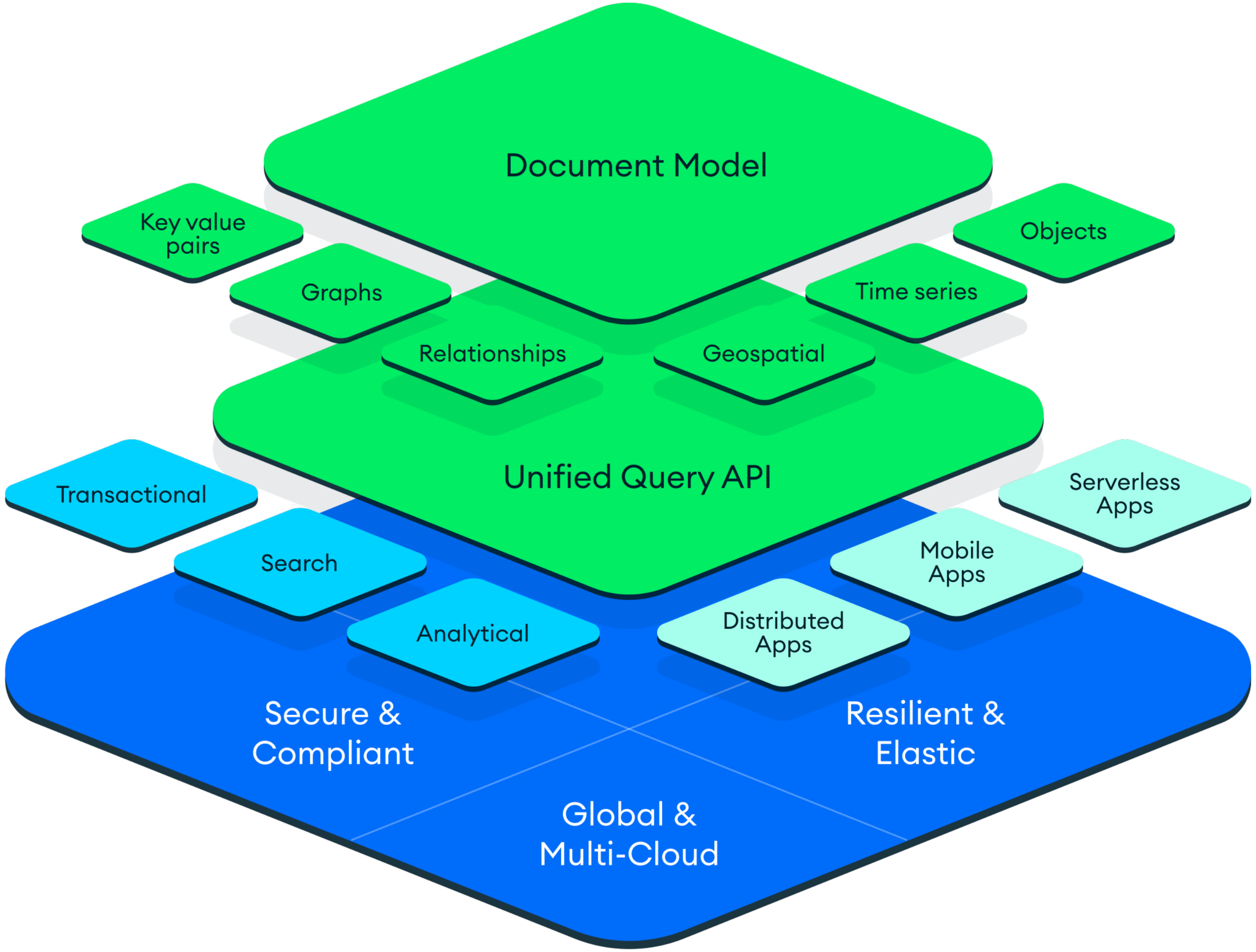
Source: https://www.mongodb.com/collateral/mongodb-evolved-becoming-the-worlds-most-wanted-database

MongoDB's "Developer Data Platform" evolution

Again, quoting from the marketing material…

- Early MongoDB releases were "focused on validating a new and largely unproven approach to database design":
  - JSON-like document data model
  - Elastic and distributed systems foundation
- Those releases "attracted adoption across startups and enterprises alike"
- MongoDB's "focus has since shifted to expanding the system beyond a niche NoSQL database into the industry's first developer data platform"
- "From operational and transactional workloads with integrated full-text search to real-time analytics and mobile computing at the network edge, MongoDB Atlas developer data platform accelerates and simplifies how developers build with data for any class of modern application, all accessed via a unified API."

Source: https://www.mongodb.com/evolved

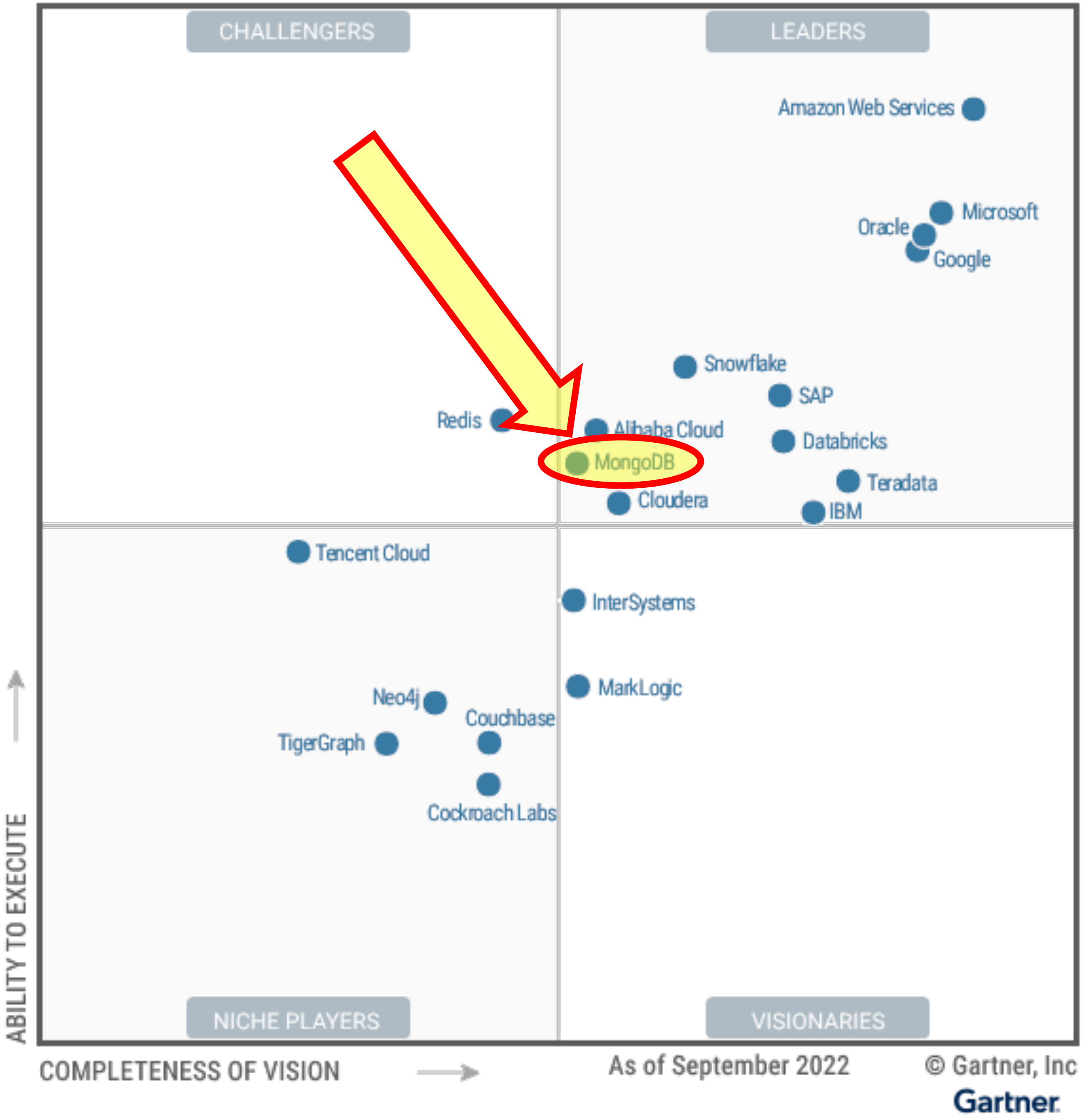


MongoDB and Gartner's "Magic Quadrant"(2022)

- As of 2022, MongoDB is still a "Leader" in Cloud Database Management Systems
- In this category, MongoDB is somewhat "edged out" by cloud providers, like AWS
- Due to MongoDB's flexibility, it may be more dominant in other categories

Bottom line…

- The NoSQL market is extremely competitive and MongoDB is constantly under threat
- There are many factors to consider when choosing a NoSQL database, such as
  - Cost and scale
  - Type of data
  - Access patterns
  - Developer familiarity
  - Many, many more…



Source: https://www.cloudera.com/campaign/2022-gartner-magic-quadrant-for-cloud-database-management-systems/2022-gartner-magic-quadrant-for-cloud-dbms-thank-you.html?cid=7012H000001Z0PxQAK

Importing Data into MongoDB

- The `mongoimport` tool is included with the MongoDB tools package
  - Imports content from an Extended JSON, CSV, or TSV export created by `mongoexport`, or potentially, another third-party export tool
  - Available in MongoDB Download Center as part of the Database tools package
- Examples
  - Import contacts from JSON file into `contacts` collection of `users` database:

```
$ mongoimport --db=users --collection=contacts --file=contacts.json
```

  - Import contacts from CSV with field names from header row:

```
$ mongoimport --db=users --collection=contacts --type=csv --headerline --file=/opt/backups/contacts.csv
```

- See documentation for more examples:
  - https://www.mongodb.com/docs/database-tools/mongoimport/

Connecting to MongoDB

- The mongo shell is included with the MongoDB Server installation***
  - Interactive JavaScript interface to MongoDB
  - Used to query and update data and perform administrative operations.
  - Also available from the tools section of the MongoDB Download Center
- To access the mongo shell…
  - Change to the `bin` directory in the MongoDB installation folder*:

```
$ cd <mongodb installation dir>/bin
```

(*Or add the `bin` directory to the executable PATH for your environment)

  - Connect to a local server using default port:

```
$ mongo
```

  - Connect to remote server on port 28015:

```
$ mongo "mongodb://mongodb0.example.com:28015"
```

***NOTE: There is now a new MongoDB Shell called `mongosh`. See the mongosh documentation page for info.

Browsing Databases and Collections

- From the mongo shell…
  - Use the `show dbs` command to list databases:

    ```
    > show dbs
    ```

  - Access databases with the `use` command followed by the database name:

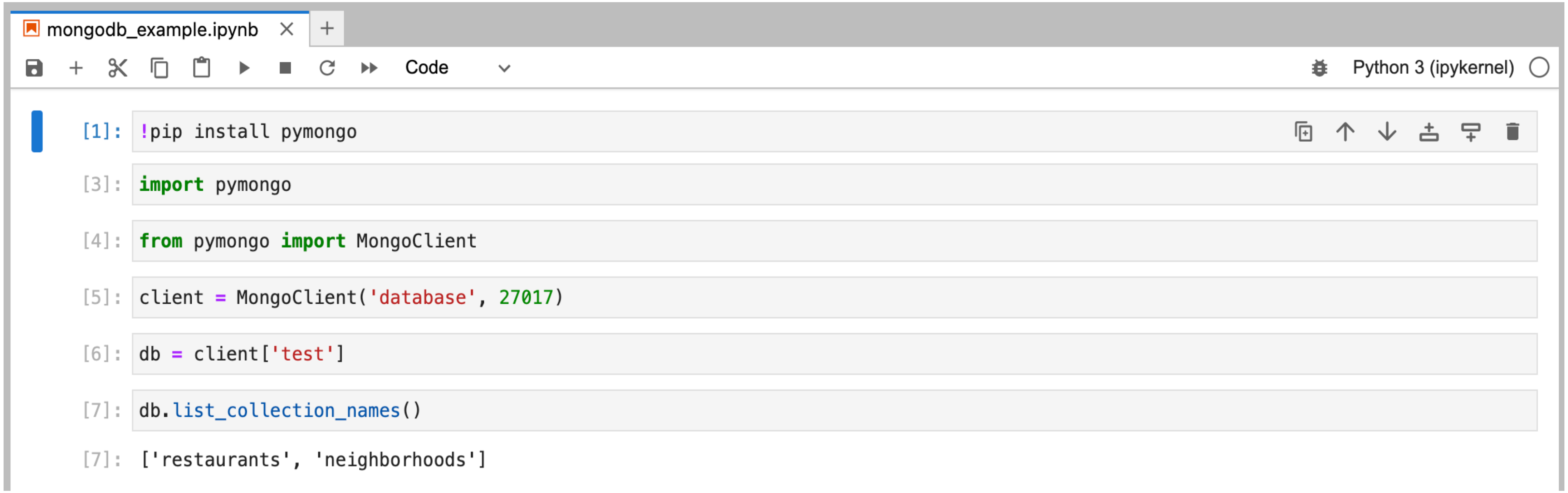
    ```
    > use test
    ```

  - Use `show collections` to list collections in the current database:

    ```
    > show collections
    ```

PyMongo and Jupyter

- PyMongo is a distribution of Python tools for working with MongoDB
  - Enables easy integration of MongoDB with Jupyter Notebooks
- Sample MongoDB connection from Jupyter:



```
[1]: !pip install pymongo

[3]: import pymongo

[4]: from pymongo import MongoClient

[5]: client = MongoClient('database', 27017)

[6]: db = client['test']

[7]: db.list_collection_names()

[7]: ['restaurants', 'neighborhoods']
```

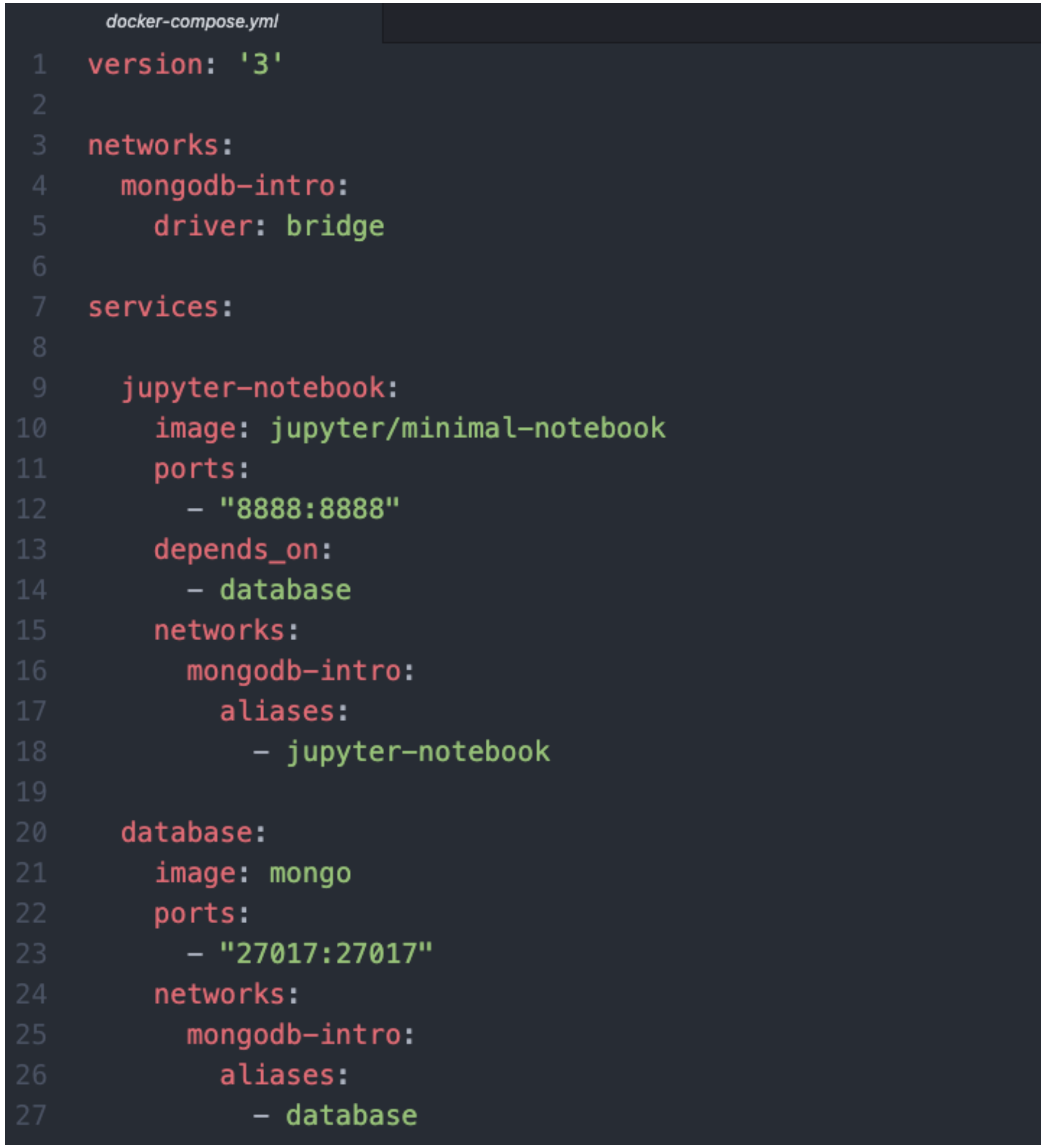
MongoDB & Jupyter Docker QuickStart

- Docker compose can be used to deploy both MongoDB and Jupyter
  - Only pre-requisite is to install `docker` and `compose` on the target system
- Steps to deploy:
  - Install docker
  - Install compose
  - Create a directory for your deployment:

```
> mkdir mongo-compose
> cd mongo-compose
```

  - Create `docker-compose.yml` file and copy sample YAML code to the right
  - Launch compose deployment:

```
> docker compose up
```

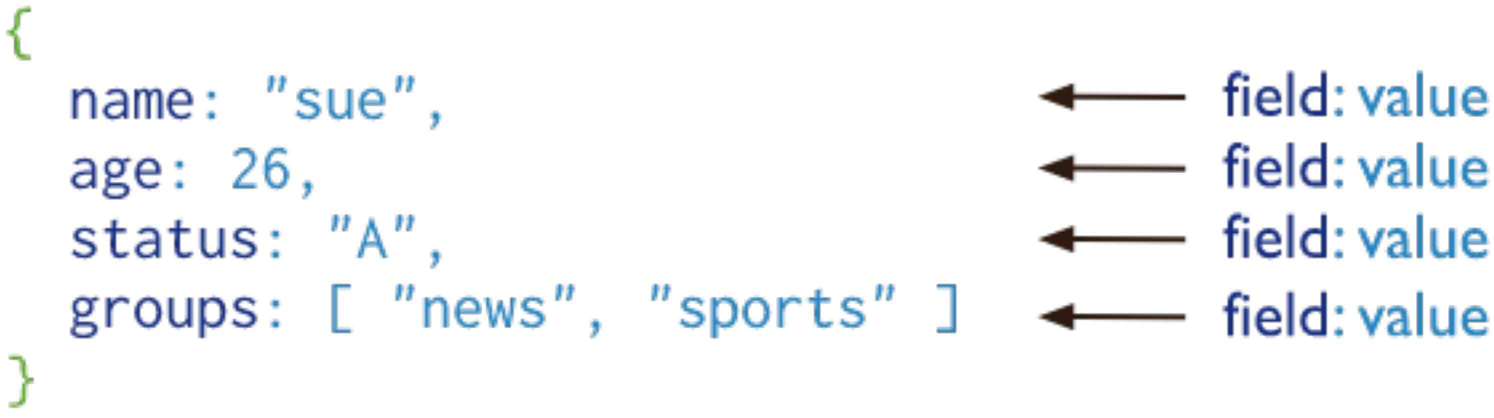


```yaml
version: '3'

networks:
  mongodb-intro:
    driver: bridge

services:

  jupyter-notebook:
    image: jupyter/minimal-notebook
    ports:
      - "8888:8888"
    depends_on:
      - database
    networks:
      mongodb-intro:
        aliases:
          - jupyter-notebook

  database:
    image: mongo
    ports:
      - "27017:27017"
    networks:
      mongodb-intro:
        aliases:
          - database
```

Part 2

MongoDB Documents and Simple Queries

BSON Overview



```
{
   name: "sue",
   age: 26,
   status: "A",
   groups: [ "news", "sports" ]
}
```

- MongoDB stores data records as BSON document
- BSON is a binary representation of JSON documents
    - BSON contains more data types than JSON
    - For the BSON spec, see bsonspec.org

BSON Types

Type	Number	Alias	Notes
Double	1	“double”	
String	2	“string”	
Object	3	“object”	
Array	4	“array”	
Binary data	5	“binData”	
Undefined	6	“undefined”	Deprecated.
ObjectId	7	“objectId”	
Boolean	8	“bool”	
Date	9	“date”	
Null	10	“null”	
Regular Expression	11	“regex”	
DBPointer	12	“dbPointer”	Deprecated.
JavaScript	13	“javascript”	
Symbol	14	“symbol”	Deprecated.
JavaScript (with scope)	15	“javascriptWithScope”	
32-bit integer	16	“int”	
Timestamp	17	“timestamp”	
64-bit integer	18	“long”	
Decimal128	19	“decimal”	New in version 3.4.
Min key	-1	“minKey”	
Max key	127	“maxKey”	

Source: https://docs.mongodb.com/manual/reference/bson-types/

MongoDB Document Example

```
var mydoc = {
        _id: ObjectId("5099803df3f4948bd2f98391"),
        name: { first: "Alan", last: "Turing" },
        birth: new Date('Jun 23, 1912'),
        death: new Date('Jun 07, 1954'),
        contribs: [ "Turing machine", "Turing test" ],
        views : NumberLong(1250000)
        }
```

- `_id` holds an ObjectId
- `name` holds an *embedded document* that contains the fields first and last
- `birth` and death hold values of the *Date* type
- `contribs` holds an *array of strings*
- `views` holds a value of the *NumberLong* type

Dot Notation for Arrays

- Specify or access an array element by zero-based index position
  - Concatenate the array name with dot (.) and index position, and enclose in quotes:

```
"<array>.<index>"
```

- For Example, Given the document:

```
{
        ...
        contribs: [ "Turing machine", "Turing test", "Turingery" ],
        ...
}
```

- The third element in the `contribs` array is `"contribs.2"`

Dot Notation for Embedded Documents

- Specify or access a field of an embedded document
  - Concatenate embedded document name with the dot (.) and the field name, and enclose in quotes:

```
"<embedded document>.<field>"
```

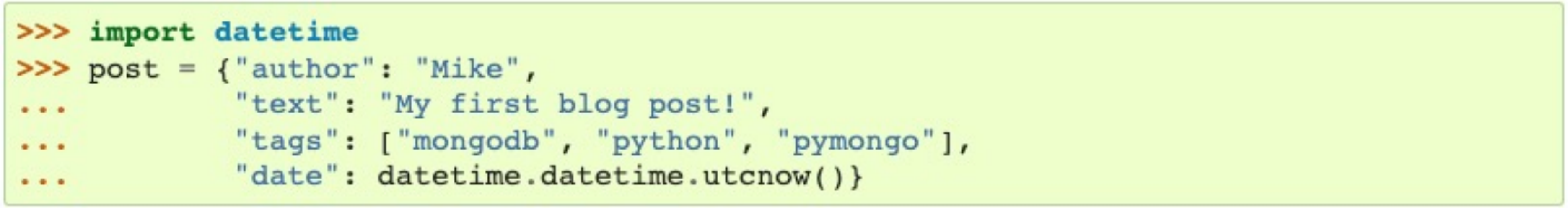
- For example, given the following field in a document:

```
{
        ...
        name: { first: "Alan", last: "Turing" },
        contact: {
                phone: { type: "cell", number: "111-222-3333" }
                },
...
}
```

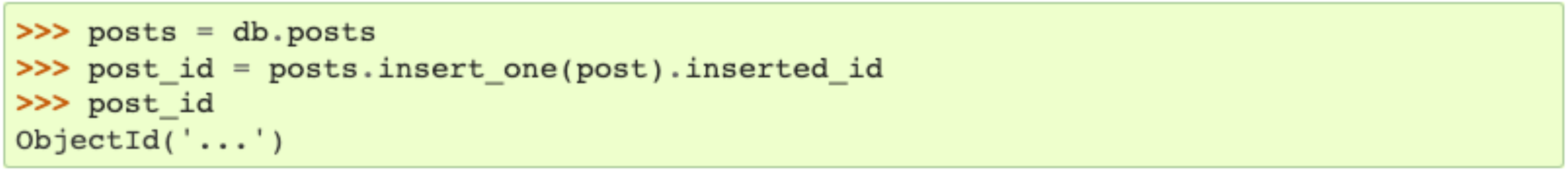
- The field named `last` in the `name` object is `"name.last"`.
- The `number` field in the `phone` embedded document contained in the `contact` field is `"contact.phone.number"`

Create Operations

- Methods to insert documents into a collection:
  - `db.collection.insert_one()` – Takes a single BSON Object
  - `db.collection.insert_many()` – Takes an array of BSON Objects
- Example:



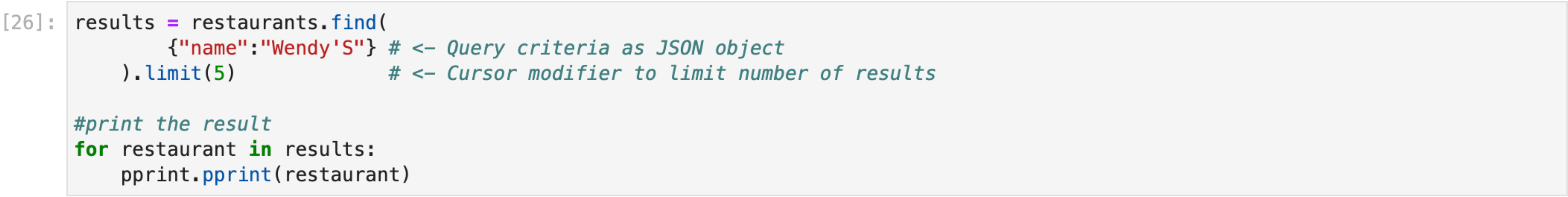
```
>>> import datetime
>>> post = {"author": "Mike",
...         "text": "My first blog post!",
...         "tags": ["mongodb", "python", "pymongo"],
...         "date": datetime.datetime.utcnow()}
```



```
>>> posts = db.posts
>>> post_id = posts.insert_one(post).inserted_id
>>> post_id
ObjectId('...')
```

Read Operations

- Method to find documents in a collection:
  - `db.collection.find_one()` – Find a single document
  - `db.collection.find()` – Find a multiple documents
- Specify query filters or criteria to identify documents to return:



[26]:

```
results = restaurants.find(
        {"name":"Wendy'S"} # <- Query criteria as JSON object
    ).limit(5)             # <- Cursor modifier to limit number of results

#print the result
for restaurant in results:
    pprint.pprint(restaurant)
```

- Other useful cursor modifiers
  - count() – Returns the number of documents in result set
  - pretty() – Configures results to make them more readable
  - sort() – Returns results sorted according to sort specification
  - forEach() – Applies JavaScript function to each document

MongoDB Queries

- Query all documents in a collection
  - Pass an empty document {} as the query filter parameter:

```
db.inventory.find( {} )
```

  - Equivalent SQL:

```
SELECT * FROM inventory
```

- Limit query results
  - Use the `limit()` cursor modifier to limit the number of documents returned by a query
  - Example, select the first 5 documents from the `inventory` collection:

```
db.inventory.find( {} ).limit(5)
```

  - Equivalent SQL:

```
SELECT * FROM inventory LIMIT 5
```

Query on equality conditions

- Use `<field>:<value>` expressions in the query filter document:

```
{ <field1>: <value1>, ... }
```

- Example, select from the `inventory` collection all documents where the `status` equals `"D"`:

```
db.inventory.find( { status: "D" } )
```

- Equivalent SQL:

```
SELECT * FROM inventory WHERE status = "D"
```

Part 3

Advanced MongoDB Queries

Query Operators

- A query document can use query operators to specify conditions:

```
{ <field1>: { <operator1>: <value1> }, ... }
```

- Example, retrieve all documents from the `inventory` collection where `status` equals either `"A"` or `"D"`:

```
db.inventory.find( { status: { $in: [ "A", "D" ] } } )
```

- Equivalent SQL:

```
SELECT * FROM inventory WHERE status in ("A", "D")
```

- For complete list of query operators see:
    - https://docs.mongodb.com/manual/reference/operator/query/

AND/OR Conditions

- Compound queries imply an AND logical conjunction
  - Example, retrieve all documents in the `inventory` collection where the `status` equals `"A"` **and** `qty` is less than (`$lt`) 30:

```
db.inventory.find( { status: "A", qty: { $lt: 30 } } )
```

  - Equivalent SQL:

```
SELECT * FROM inventory WHERE status = "A" AND qty < 30
```

- Use $or operator to specify compound queries with logical OR conjunction
  - Example, retrieve documents with `status` `"A"` **or** `qty` less than (`$lt`) 30:

```
db.inventory.find( { $or: [ { status: "A" }, { qty: { $lt: 30 } } ] } )
```

  - Equivalent SQL:

```
SELECT * FROM inventory WHERE status = "A" OR qty < 30
```

Query Embedded Documents

- Equality conditions on a field that is an embedded/nested document
  - Use **`{<field>:<value>}`** where **`<value>`** is the exact document to match
  - Equality matches on the whole embedded document require an *exact* match of the specified **`<value>`**
  - Example, select documents where **`size`** is equal to the document **`{ h: 14,w: 21, uom: "cm"`** }:

```
db.inventory.find( { size: { h: 14, w: 21, uom: "cm" } } )
```

- Equality condition on a nested field
  - Use the dot notation: **`”field.nestedField”`**
  - Example, select documents where **`uom`** in **`size`** field equals **`”in”`**:

```
db.inventory.find( { "size.uom": "in" } )
```

- Query nested field using operators
  - Example, match documents where h nested field is less than 15:

```
db.inventory.find( { "size.h": { $lt: 15 } } )
```

Query An Array

- Match an Array
  - Example, match documents where field **`tags`** value is an array with exactly two elements, **`"red"`** and **`"blank"`**, in that order:

```
db.inventory.find( { tags: ["red", "blank"] } )
```

  - To match array in any order, use the $all operator:

```
db.inventory.find( { tags: { $all: ["red", "blank"] } } )
```

- Match an Array Element
  - Example, match documents where **`tags`** is an array that contains the string **`"red"`** as one of its elements:

```
db.inventory.find( { tags: "red" } )
```

Query Array with Conditions

- Match Compound Filter Conditions on the Array Elements
  - Example, match documents where **`dim_cm`** array contains elements that, in some combination, satisfy the query conditions:

```
db.inventory.find( { dim_cm: { $gt: 15, $lt: 20 } } )
```

  - NOTE: A single element does not have to match both conditions

- Query for an Array Element that Meets Multiple Criteria
  - Use $elemMatch operator to specify multiple criteria that at least one array element must satisfy
  - Example, matches documents where **`dim_cm`** array contains at least one element greater than **`($gt) 22`** **and** less than **`($lt) 30`**:

```
db.inventory.find( { dim_cm: { $elemMatch: { $gt: 22, $lt: 30 } } } )
```

Query Documents in Arrays (1/2)

- Query a Field in the Embedded Document by Array Index
  - Use dot notation with zero-based index to specify query conditions for field in a document at a particular position of the array
  - Example, match documents where first element of `instock` array is a document where `qty` is less than or equal to `20`:

```
db.inventory.find( { 'instock.0.qty': { $lte: 20 } } )
```

- Query a Field Embedded in an Array of Documents
  - Concatenate array name, with a dot (.) and the name of the field in nested document to specify conditions on at least one element
  - Example, match documents where the `instock` array has at least one embedded document where `qty` is less than or equal to `20`:

```
db.inventory.find( { 'instock.qty': { $lte: 20 } } )
```

Query Documents in Arrays (2/2)

- Multiple Query Conditions on an Array of Documents
  - Use $elemMatch to specify multiple criteria on array of embedded documents where at least one element satisfies all the criteria
  - Example, match documents where `instock` array has at least one embedded document where `qty` is `5` **and** `warehouse` equals `A`:

```
db.inventory.find( { "instock": { $elemMatch: { qty: 5, warehouse: "A" } } } )
```

- Query for Combination of Elements that Satisfies the Criteria
  - To selects documents whose array contains any combination of elements that satisfies the conditions do not use $elemMatch
  - Example, match documents where any nested document inside the `instock` array has `qty` greater than `10` and any nested document in the `instock` array (but not necessarily the same) has `qty` less than or equal to `20`:

```
db.inventory.find( { "instock.qty": { $gt: 10, $lte: 20 } } )
```

Part 4

MongoDB Query Projections

Query Projections

- Return the Specified Fields and the `_id` Field Only
  - Include fields by specifying **`<field>: 1`** in the projection document
  - Example, return only **`item`**, **`status`** and **`_id`** fields of matches:

```
db.inventory.find( { status: "A" }, { item: 1, status: 1 } )
```

  - SQL equivalent:

```
SELECT _id, item, status from inventory WHERE status = "A"
```

- Suppress `_id` Field
  - Specify **`<field>: 0`** in the projection, as in the following example:

```
db.inventory.find( { status: "A" }, { item: 1, status: 1, _id: 0 } )
```

- Return All But the Excluded Fields
  - Exclude fields by specifying **`<field>: 0`** in projection document:

```
db.inventory.find( { status: "A" }, { status: 0, instock: 0 } )
```

Part 5

MongoDB Indexes

MongoDB Indexes Overview

- Indexes support the efficient execution of queries in MongoDB
  - Without indexes, MongoDB must scan every document in a collection
  - If appropriate index exists, fewer documents must be inspected
- By default, MongoDB inserts an index of type `ObjectId` as the `_id` field of every new document
- Other useful index types include:
  - Text Indexes
    - Do not store language-specific *stop* words (e.g. “the”, “a”, “or”)
    - *Stem* the words in a collection to only store root words
  - Geospatial Indexes
    - 2d indexes that use planar geometry when returning results
    - 2dsphere indexes that use spherical geometry to return results.

Text Queries

- Query on a string data requires exact match on value
- To search for a word or phrase appearing anywhere on a string field use regex:

```
db.articles.find( { status: /foo bar/ } )
```

- Perform full text search on fields indexed with a text index
  - Create text index on a single field:

```
db.articles.createIndex( { subject: "text" } )
```

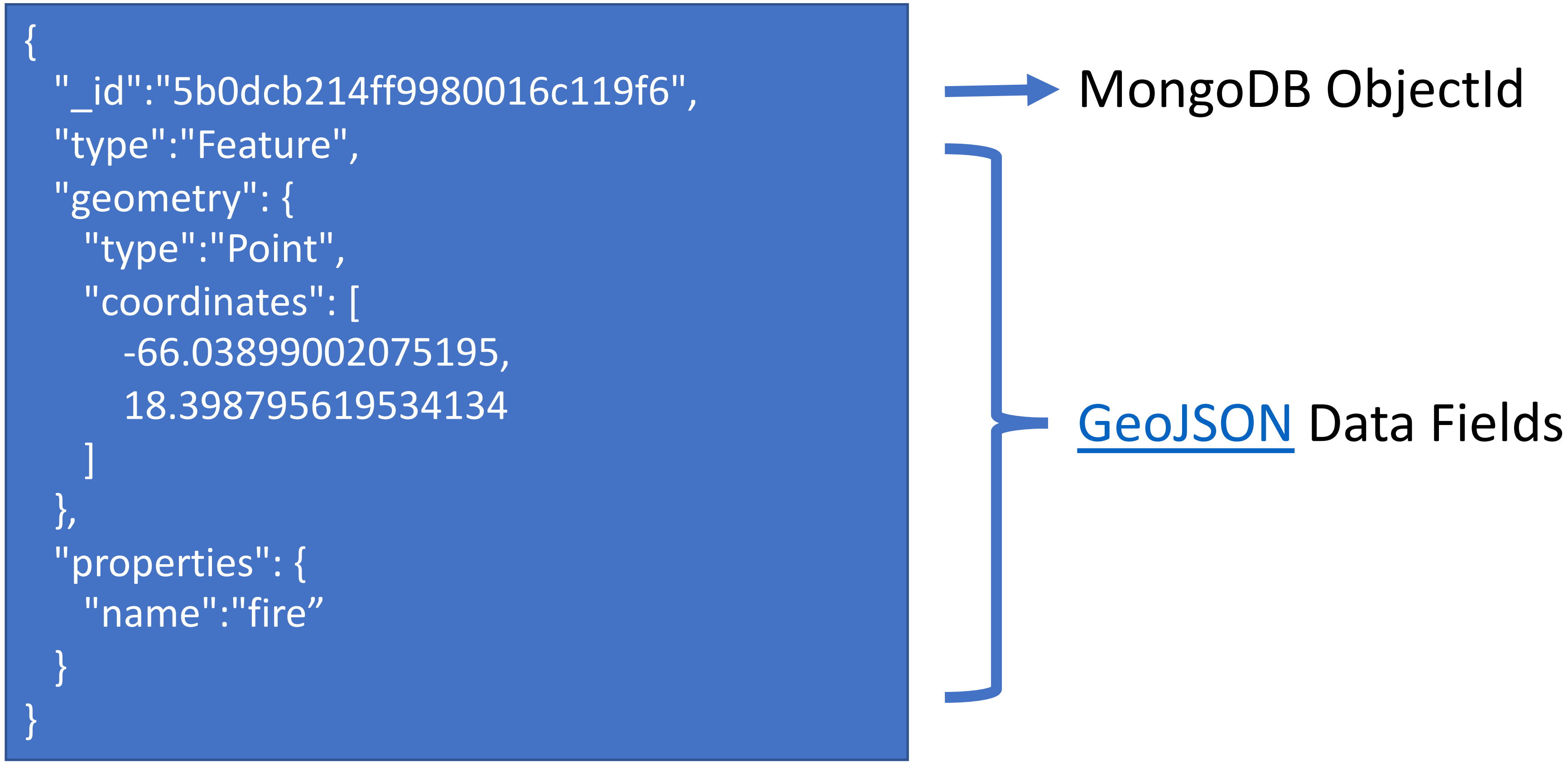
  - Or, create text index over all fields with string data in collection

```
db.articles.createIndex( { “$**” : "text" } )
```

  - Use $text query operator:

```
db.articles.find( { $text: { $search: "coffee" } } )
```

Storing Geospatial Data in MongoDB



```
{
  "_id":"5b0dcb214ff9980016c119f6",
  "type":"Feature",
  "geometry": {
    "type":"Point",
    "coordinates": [
       -66.03899002075195,
       18.398795619534134
    ]
  },
  "properties": {
    "name":"fire”
  }
}
```

MongoDB ObjectId

GeoJSON Data Fields

MongoDB Geospatial Queries

- Create a 2dsphere index on field(s) with geographic data
  - Create index on field containing coordinates in the form `[ long, lat ]`:

```
db.restaurants.createIndex({ coordinates: "2dsphere" })
```

  - Create index on field containing GeoJSON geometry object (E.g. point or polygon):

```
db.restaurants.createIndex({ geometry: "2dsphere" })
```

- Find documents with geospatial fields that intersect a point:

```
db.neighborhoods.findOne({ geometry: { $geoIntersects: { $geometry: { type: "Point", coordinates: [ -73.93414657, 40.82302903 ] } } } })
```

- Find documents with geospatial fields within a circular region:

```
db.restaurants.find({ location: { $geoWithin: { $centerSphere: [ [ -73.93414657, 40.82302903 ], 5 / 3963.2 ] } } })
```

- Return documents within sphere in sorted order (near to far):

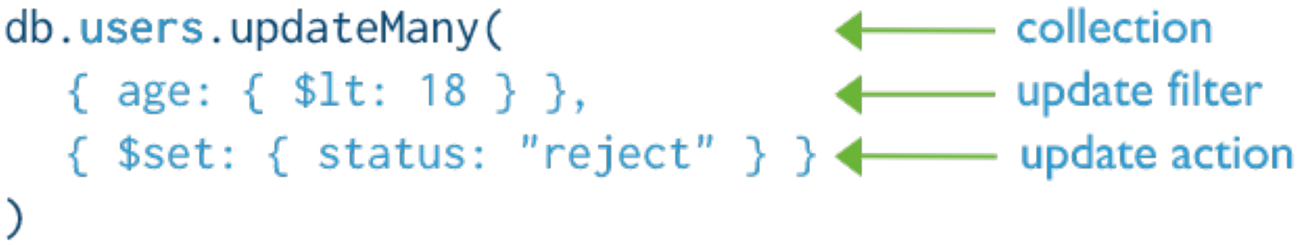
```
db.restaurants.find({ location: { $nearSphere: { $geometry: { type: "Point", coordinates: [ -73.93414657, 40.82302903 ] },
                                                  $maxDistance: 5 * METERS_PER_MILE } } })
```

Part 6

MongoDB Update Operations

Update Operations

- Methods to update documents of a collection:
  - db.collection.updateOne() – Update only first matching document
  - db.collection.updateMany() – Update all matching documents
  - db.collection.replaceOne() – Replace first matching document
- Specify criteria, or filters, to identify documents to update using the same syntax as read operations:



```
db.users.updateMany(
   { age: { $lt: 18 } },
   { $set: { status: "reject" } }
)
```

- Use `{ upsert : true }` to create document if it doesn't exist:

```
db.inventory.updateMany( <filter>, <update>, {upsert : true} )
```

Array Update Operators

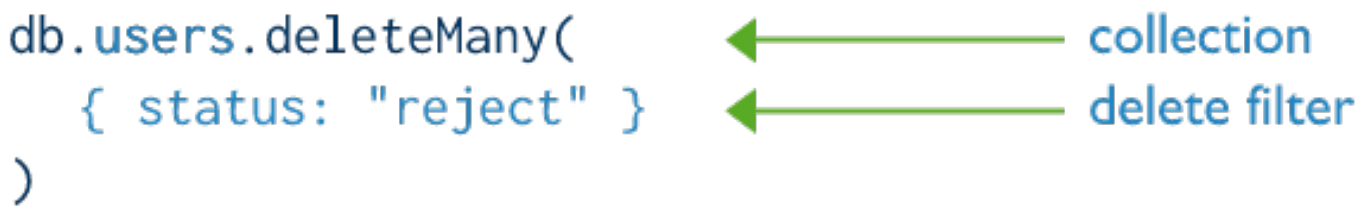
- $ - Acts as a placeholder to update first element matching query
  - Example, update the value of the **std** field in the embedded document with the **grade** of **85**:

```
db.students.update(
        { _id: 4, "grades.grade": 85 },
        { $set: { "grades.$.std" : 6 } }
)
```

- $addToSet - Adds elements to an array if they do not exist
- $push - Adds an item to an array even if it exists
  - If the value is an array, it is added as a single element
- $pop - removes the first or last element of an array
  - Pass a value of -1 to remove the first element
  - Pass a value of 1 to remove the last element
- Full list of update operators:
  - https://docs.mongodb.com/manual/reference/operator/update/

Delete Operations

- Methods to delete documents of a collection:
  - db.collection.deleteOne() – Deletes only first matching document
  - db.collection.deleteMany() – Deletes all matching documents
- Specify criteria, or filters, to identify the documents using the same syntax as read operations:



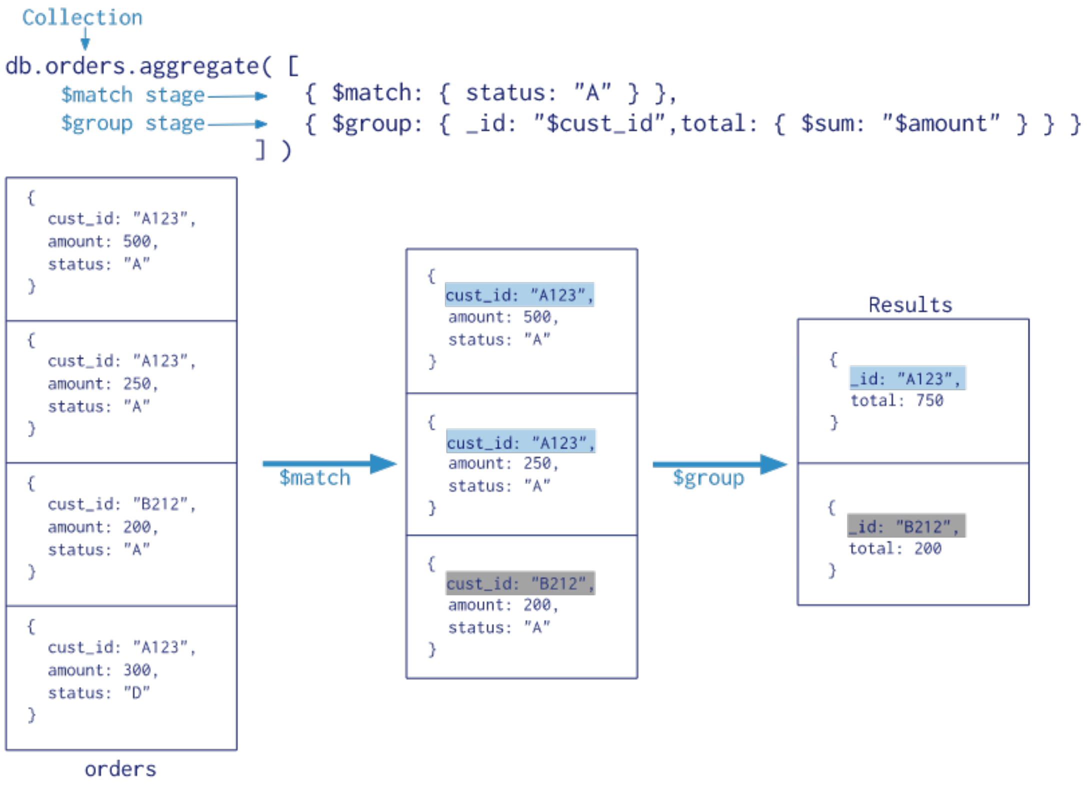
```
db.users.deleteMany(
   { status: "reject" }
)
```

Part 7

MongoDB Aggregation Pipeline

MongoDB Aggregation Pipeline

- MongoDB framework for data aggregation
  - Possible alternative to map-reduce for aggregation tasks where the complexity of map-reduce may be unwarranted
  - Documents enter multi-stage pipeline to be transformed into aggregated results:



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

Collection
{ 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
$match
{ cust_id: "A123", amount: 500, status: "A" }
{ cust_id: "A123", amount: 250, status: "A" }
{ cust_id: "B212", amount: 200, status: "A" }
$group
Results
{ _id: "A123", total: 750 }
{ _id: "B212", total: 200 }

Aggregation Pipeline Stages

- MongoDB aggregation pipelines consists of stages:

```
db.collection.aggregate( [ { <stage> }, ... ] )
```

- Each stage transforms documents as they pass through the pipeline
  - Stages do not have to produce one output for every input
  - Documents may be created or filtered out during stages
- Pipeline stages can appear multiple times in the pipeline
- For a full list of pipeline stages see:
  - https://docs.mongodb.com/manual/reference/operator/aggregation/#aggregation-pipeline-operator-reference

Sample Pipeline Stages

Stage	Description
`$match`	Filters document stream using standard MongoDB to allow only matching documents to pass unmodified into the next stage queries.
`$group`	Groups input documents by a specified identifier expression and applies the accumulator expression(s), if specified, to each group. Consumes all input documents and outputs one document per each distinct group. The output documents only contain the identifier field and, if specified, accumulated fields.
`$sort`	Reorders the document stream by a specified sort key. Only the order changes; the documents remain unmodified. For each input document, outputs one document.
`$project`	Reshapes each document in the stream, such as by adding or removing fields. For each input document, outputs one document.
`$lookup`	Performs a left outer join to another collection in the *same* database to filter in documents from the “joined” collection for processing.
`$unwind`	Deconstructs an array field from the input documents to output a document for *each* element. Each output document replaces the array with an element value.
`$out`	Writes the resulting documents of the aggregation pipeline to a collection. To use the `$out` stage, it must be the last stage in the pipeline.

SQL to Aggregation Mapping

SQL Terms, Functions, and Concepts	MongoDB Aggregation Operators
WHERE	$match
GROUP BY	$group
HAVING	$match
SELECT	$project
ORDER BY	$sort
LIMIT	$limit
SUM()	$sum
COUNT()	$sum
join	$lookup

Aggregation Pipeline Example

- Suppose we have a collection of documents like:

```
{
      "_id":  "10280",
      "city": "NEW YORK",
      "state": "NY",
      "pop":  5574,
      "loc":  [ -74.016323,  40.710537  ]
}
```

- This pipeline returns states with population above 10 million:

```
db.zipcodes.aggregate( [
     { $group: { _id: "$state", totalPop: { $sum: "$pop" } } },
     { $match: { totalPop: { $gte: 10*1000*1000 } } }
] )
```

- Notes:
  - The `$group` stage groups documents by the `state` field and uses the `$sum` operator to add the population field (`pop`) for each state
  - The output of the `$group` stage is a set of documents with two fields:
    1. `_id`: contains the grouped state field value
    2. `totalPop`: contains the total population of each state
  - The `$match` stage filters these grouped documents to output only those where totalPopvalue is greater than or equal to 10 million

Aggregation Pipeline Example #2

- Return the average city population by state:

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

- Notes:
  - The first $group stage groups the documents by the combination of city and state and uses the $sum expression to calculate the population for each combination to create a document like:

```
{
    "_id"  :  { "state" : "CO", "city" : "EDGEWATER"  },
    "pop"  :  13154
}
```

  - The second $group stage groups the documents in the pipeline by the _id.state field (i.e. the state field inside the _id document) and uses the $avg expression to calculate the average city population:

```
{  "_id" : "MN", "avgCityPop"  :  5335  }
```