Discussion 13

NoSQL

Announcements

Vitamin 13 (NoSQL) due Monday, April 29 at 11:59pm

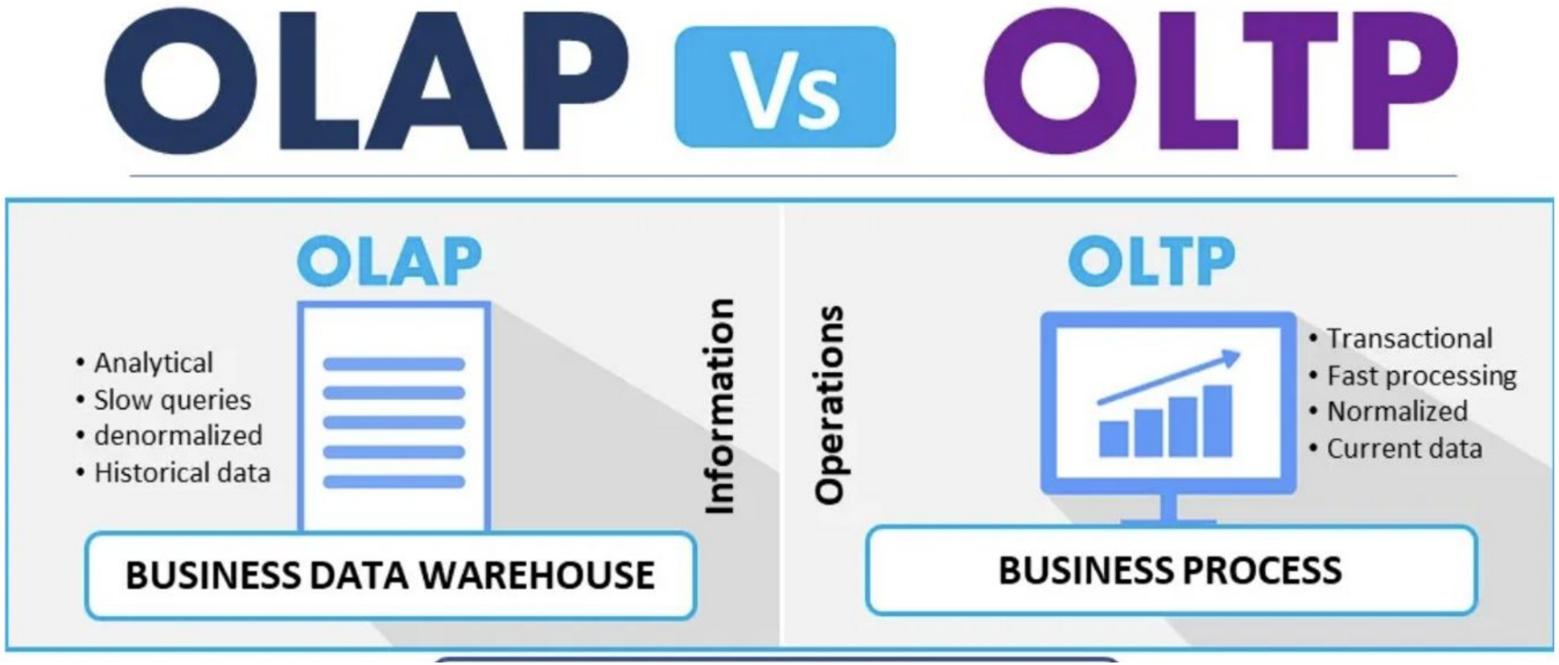
Agenda

Background

Types of Workloads

- **Online Transaction Processing (OLTP)**
  - Simple lookup and update queries with few joins and aggregations
    - E.g. Updating the “status” attribute on your social media page
  - Workload involves high numbers of transactions by many users
    - E.g. Modern “Web 2.0” applications with lots of user-generated content and user interactions
- **Online Analytical Processing (OLAP)**
  - Read-only queries that involve lots of joins and aggregations
  - Primarily used to support data-driven decision making
- **OLTP** and **OLAP** workloads are often handled by separate databases
  - **OLTP** data can be migrated to **OLAP** databases through a process called **Extract-Transform-Load (ETL)**

Quick Summary Guide:



Question 1: OLTP vs. OLAP

For each of these workloads, choose whether it's better characterized as Online Transaction Processing or Online Analytical Processing:

a.) A social media site with millions of users needs to track all the "likes" and "dislikes" that each post receives.

Question 1: OLTP vs. OLAP

For each of these workloads, choose whether it's better characterized as Online Transaction Processing or Online Analytical Processing:

a.) A social media site with millions of users needs to track all the "likes" and "dislikes" that each post receives.

OLTP: OLTP workloads involve high numbers of transactions executed by many different users. Queries in these workloads involve simple lookups more often than complex joins. In this case, when a user likes or dislikes a post, the site would need to do a lookup on that post and update its likes/dislikes value.

Question 1: OLTP vs. OLAP

For each of these workloads, choose whether it's better characterized as Online Transaction Processing or Online Analytical Processing:

b.) An online book store needs to aggregate and analyze its users book purchases by genre over the last eight months.

Question 1: OLTP vs. OLAP

For each of these workloads, choose whether it's better characterized as Online Transaction Processing or Online Analytical Processing:

b.) An online book store needs to aggregate and analyze its users book purchases by genre over the last eight months.

OLAP: OLAP workloads involve read-only queries and typically include lots of joins and aggregations. Often, workloads executed for analysis and decision making are OLAP workloads. In this case, the book store needs to read from data stored over the last eight months regarding book purchases, and may need to perform some joins to analyze those purchases by genre.

Question 1: OLTP vs. OLAP

For each of these workloads, choose whether it's better characterized as Online Transaction Processing or Online Analytical Processing:

c.) A multiplayer online game has added updated areas to its map and now wants to assess how users behave in those areas, and how user playtime has changed as a result.

Question 1: OLTP vs. OLAP

For each of these workloads, choose whether it's better characterized as Online Transaction Processing or Online Analytical Processing:

c.) A multiplayer online game has added updated areas to its map and now wants to assess how users behave in those areas, and how user playtime has changed as a result.

OLAP: In this case, the game designers will need to read data that likely includes which players logged in after the updated areas were released, their average playtimes, and any other data that may be relevant to how players behave in the new area.

Scaling

- To serve heavy workloads, a database needs to be scaled in one of two ways:
    1. **Partitioning** (aka **Sharding**): Split the data among multiple machines to increase parallelism.
        - Writes are quicker because different machines can be updated simultaneously. Reads are slower because they may need to access multiple machines.
    2. **Replication**: Copy the data among multiple machines to increase fault-tolerance.
        - Writes are slower because data needs to be replicated. Reads are faster because there are more copies of data to read.

Question 2: Scaling

a.) A small startup realizes that its current database can't sustain their growing workloads. Given that these workloads involve a lot of writes but few reads, should it invest in more **partitioning** or more **replication**?

Question 2: Scaling

a.) A small startup realizes that its current database can't sustain their growing workloads. Given that these workloads involve a lot of writes but few reads, should it invest in more **partitioning** or more **replication**?

Partitioning: More partitioning means that more queries can be executed in parallel on different machines. This is especially good for write-heavy workloads because these can often involve updates to only a few machines, meaning more of them can happen at once (higher throughput).

Question 2: Scaling

b.) A mechanical failure causes some of the startup's database machines to permanently crash, losing data in the process. If the startup wants to prevent similar losses in the future, should it invest more in **partitioning** or more **replication**?

Question 2: Scaling

b.) A mechanical failure causes some of the startup's database machines to permanently crash, losing data in the process. If the startup wants to prevent similar losses in the future, should it invest more in **partitioning** or more **replication**?

Replication: Replication means each database machine is no longer a single point of failure. Once data is replicated, the entire system is more resilient to data loss because if one machine crashes, its data can be recovered from the replicas. However, writes will become slower because the changes have to propagate through all replicas.

CAP Theorem and BASE

Distributed Systems– Desired Properties

- In any distributed system we would like to maintain these three properties:
  1. **Consistency**: Two clients making simultaneous requests to the database should get the same view of the data.
     - **Different from ACID consistency!**
  2. **Availability**: Every request must get a response. The only time it should get an error is if the input itself is erroneous.
  3. **Partition Tolerance**: The system must continue to operate even if messages between machines are delayed or dropped or even if certain machines get disconnected from the network. (i.e. Facebook datacenter in Europe going down doesn’t bring everything down)

CAP Theorem

- **CAP Theorem** states that it is **impossible** for a distributed system to provide more than 2 out of the 3 desired properties
- Practically speaking, most systems are designed to be **partition tolerant** since networks are usually unreliable
  - *Tradeoff is between **consistency** and **availability***
  - Choosing consistency over availability means the system responds with a time-out if it cannot guarantee that the data is up to date
  - Choosing availability over consistency means the system may respond with *stale data*
- In practice, many systems prioritize availability and provide **eventual consistency** which guarantees that eventually, all replicas will be consistent (i.e. once all updates stop, replicas will converge to same state)

BASE Semantics

- Eventually consistent systems opt for the 3 BASE guarantees instead of ACID guarantees:
    1. **B**asic **A**vailability: reads and writes are available as much as possible; **however**, they are not guaranteed to be consistent (i.e. a read may not get the latest data and a write might not persist after all updates propagate). This is up to the application to fix.
    2. **S**oft State: a database can change even without inputs (e.g. as updates propagate), so the application only has a probability of knowing its state.
    3. **E**ventually Consistent: given enough time (after all updates propagate), all reads will be consistent.

Question 3: BASE

a.) Database designer Doug is annoyed with his distributed database because for some time after issuing a write, all his reads return different values. Does this violate any of the BASE properties?

Question 3: BASE

a.) Database designer Doug is annoyed with his distributed database because for some time after issuing a write, all his reads return different values. Does this violate any of the BASE properties?

No.

This is in fact the **soft state** and **eventual consistency** properties in action. As the write propagates through the system, reads may return different values because the database is inconsistent.

Question 3: BASE

Which properties of BASE do these scenarios violate?

b.) All reads and writes always have the same views of data, but they sometimes respond to valid inputs with timeout errors.

Question 3: BASE

Which properties of BASE do these scenarios violate?

b.) All reads and writes always have the same views of data, but they sometimes respond to valid inputs with timeout errors.

Basic Availability.

This database seems to prioritize consistency over availability. Since valid inputs sometimes receive an error response, it violates basic availability.

Question 3: BASE

Which properties of BASE do these scenarios violate?

c.) Writes propagate to only 3 replicas, but the system has 5 replicas of each piece of data.

Question 3: BASE

Which properties of BASE do these scenarios violate?

c.) Writes propagate to only 3 replicas, but the system has 5 replicas of each piece of data.

Eventual Consistency.

Since writes do not propagate to all 5 replicas, different pieces of data will always be stale on different replicas, so reads will never be consistent.

Question 3: BASE

Which properties of BASE do these scenarios violate?

d.) An empty database that has never been populated responds to a read query on some specific key with the message "Error: key nonexistent!"

Question 3: BASE

Which properties of BASE do these scenarios violate?

d.) An empty database that has never been populated responds to a read query on some specific key with the message "Error: key nonexistent!"

No properties are violated.

The **basic availability** property only guarantees that *valid* queries always get a non-error response. However, since the database is empty, a read query is invalid, so the error response is OK.

NoSQL Motivation

- The relational databases from class:
  - are better for OLAP workloads than OLTP workloads
  - struggle to scale due to
    - lack of partitioning and
    - RDBMS overhead that may be unnecessary for OLTP (e.g. fancy query optimization)
- To better handle OLTP workloads and scale more efficiently, we can simplify our data model to be less structured
  - This gives us the **NoSQL** data model!

NoSQL Data Models

Key-Value Stores

- **Data model:** (Key, Value) pairs
  - Key = String/Integer, unique for the entire data
  - Value = Can be anything (very complex object)
- Among the most flexible data models
- **Operations:**
  - `get(key), put(key, value)`
  - Operations on value are not supported due to flexibility of value type
- **Distribution / Partitioning:** (with a hash function)
  - No replication: key k is stored at server h(k)
  - Multi-way replication: e.g. key k stored at $h_1(k)$, $h_2(k)$, $h_3(k)$. On update, propagate change to other servers, *eventual consistency*
- e.g. Amazon Dynamo, Voldemort, Memcached

Key-Value Stores Example

Database Doug now has the following tables:

```
Sales (sid, date, quantity, customer, product)
Product (pid, name, price)
Customer (cid, name, address)
```

The `Sales` data can be represented in multiple ways as key, value pairs:

- Option 1: Key = `sid`, Value = entire Sales record
- Option 2: Key = `date`, Value = list of all Sales for that day
- Option 3: Key = `customer`, Value = list of all purchases of this customer
- Option 4: Key = `(customer, product)`, Value = list of all purchases of this product by this customer

Choose depending on your use case!

Key-Value Stores Q1a

Database Doug now has the following tables:

```
Sales (sid, date, quantity, customer, product)
Product (pid, name, price)
Customer (cid, name, address)
```

`Sales` data is stored with Key = `sid`, Value = entire Sales record, partitioned on hash function h and replicated across 3 servers. **Describe how the operation `get(sid1)` would be executed. (Assume a Sale with `sid1` exists in the data).**

Key-Value Stores Q1a

Database Doug now has the following tables:

```
    Sales (sid, date, quantity, customer, product)
    Product (pid, name, price)
    Customer (cid, name, address)
```

`Sales` data is stored with Key = `sid`, Value = entire Sales record, partitioned on hash function h and replicated across 3 servers. **Describe how the operation `get(sid1)` would be executed. (Assume a Sale with `sid1` exists in the data).**

We must first hash the key, `h(sid1)`, to find which **partition** the data is stored on. Then we can retrieve the value from any of the **replicas/servers**.

Key-Value Stores Q1b

Database Doug now has the following tables:

```
    Sales (sid, date, quantity, customer, product)
    Product (pid, name, price)
    Customer (cid, name, address)
```

`Sales` data is stored with Key = `sid`, Value = entire Sales record, partitioned on hash function h and replicated across 3 servers. **Describe how the operation `put(sid2, saleRecord)` would be executed.**

Key-Value Stores Q1b

Database Doug now has the following tables:

```
    Sales (sid, date, quantity, customer, product)
    Product (pid, name, price)
    Customer (cid, name, address)
```

`Sales` data is stored with Key = `sid`, Value = entire Sales record, partitioned on hash function h and replicated across 3 servers. **Describe how the operation `put(sid2, saleRecord)` would be executed.**

We must first hash the key, `h(sid2)`, to find which partition the data should be stored on. Then, we will insert the record into that partition, and propagate the change to the other replicas/servers. Note: Propagation of changes may not happen immediately. We only need to enforce *eventual consistency*.

Key-Value Stores Q1c

Database Doug now has the following tables:

```
Sales (sid, date, quantity, customer, product)
Product (pid, name, price)
Customer (cid, name, address)
```

`Sales` data is stored with Key = `sid`, Value = entire Sales record, partitioned on hash function h and replicated across 3 servers. **After `put(sid2, saleRecord)` is executed, is it guaranteed that every app will be able to access that new Sale data?**

Key-Value Stores Q1c

Database Doug now has the following tables:

```
    Sales (sid, date, quantity, customer, product)
    Product (pid, name, price)
    Customer (cid, name, address)
```

`Sales` data is stored with Key = `sid`, Value = entire Sales record, partitioned on hash function h and replicated across 3 servers. **After `put(sid2, saleRecord)` is executed, is it guaranteed that every app will be able to access that new Sale data?**

No. Since we're only enforcing *eventual consistency*, the changes from the put operation may not have propagated to all replicas yet. Note: It *is* possible to perform checks to see whether the replica an app is pulling data from is up-to-date, if the app requires non-stale data. Requires communication with other replicas.

Extensible Record Stores

- Aka Wide-Column Stores
- **Data model:** (like a 2-D key-value store)
    - Variant 1: key = rowID, value = record
    - Variant 2: key = (rowID, columnID), value = field
        - Can have multiple columnIDs in the key
- Compromise between structured relational model and flexible key-value store
    - Do not require each row in the table to have the same columns
- **Operations:**
    - `get(key) or get(key, [columns]), put(key, value)`
- e.g. HBase, Cassandra, PNUTS

Document Stores

- The value in key-value stores is often a very complex object due to the data stores’ flexible nature
  - E.g. key = ‘2020/7/1’, value = [all sales for that date]
- **Data Model:**
  - Document: semi-structured data format (e.g. JSON, Protobuf, or XML)
  - Store documents in collections
- Document stores are among the most structured data models
  - Called a “document” but it’s just data
- e.g. SimpleDB, CouchDB, MongoDB

JSON

JSON Overview

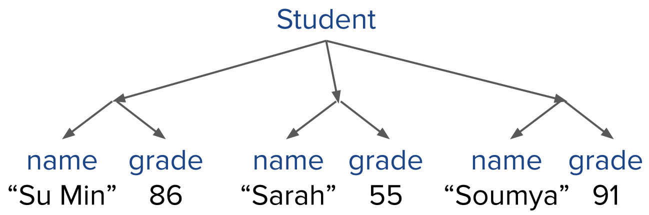
- **J**ava**S**cript **O**bject **N**otation: A text format widely adopted as a native representation for many NoSQL databases (among many other use cases)
- Supported types:
  - **Object**: collection of key (string) - value (object/array/atomic) pairs
    - Denoted with “{“ and “}”
    - Should not have duplicate keys
  - **Array**: ordered list of values (object/array/atomic)
    - Denoted with “[“ and “]”
  - **Atomic**: a number (64-bit float), string, boolean, or null
- Can be interpreted as a tree due to its nested structure
- **Self-describing**: schema elements are part of the data itself, which allows each document to have its own schema

JSON vs Relational

	JSON	**Relational**
Flexibility	Very flexible, can represent complex structures and nested data	Less flexible
Schema Enforcement	Self-describing; Each document can have unique structure	Schema is fixed
Representation	Text-based (easily parsed and manipulated by many languages)	Binary representation (designed for efficient storage and retrieval from disk)
	“Enforcing schema on read”	“Enforcing schema on write”

Relational ➔ JSON

- Single table can be represented as an object where **key**: name of table and **value**: array of objects
- Tip: Draw out a tree, then translate into JSON



Student

name	grade
Su Min	86
Sarah	55
Soumya	91

$\longrightarrow$

```
{"student": [
    {"name": "Su Min", "grade": 86},
    {"name": "Sarah", "grade": 55},
    {"name": "Soumya", "grade": 91}
  ]
}
```

Relational ➔ JSON

For a many-to-many Relationship:

- Each relation as flat JSON array, or
- Inline relations based on keys (ex. Student ➔ Classes Taken ➔ Subjects - put all subject info into each classes taken object, and put all classes taken for a student into the student object)

Student

name	gpa
Su Min	3.0
Sarah	3.5

Classes Taken

name	subject
Su Min	Math
Su Min	English
Sarah	Math

Subjects

name	teacher
Math	Max
English	Toby

Relational ➜ JSON

For a many-to-many Relationship:

- Each relation as flat JSON array, or
- Inline relations based on keys (ex. Student ➜ Classes Taken ➜ Subjects - put all subject info into each classes taken object, and put all classes taken for a student into the student object)

```
{"Students": [
  {"name": "Su Min",
   "gpa": 3.0,
   "Classes Taken": [
    {"Subject":
       {"name": "Math",
       "teacher": "Max"},
     "grade": "A"},
    {"Subject":
       {"name": "English",
       "teacher": "Toby"},
     "grade": "B"}]},
  ...
]}
```

Students

name	gpa
Su Min	3.0
Sarah	3.5

Classes Taken

name	subject	grade
Su Min	Math	A
Su Min	English	B
Sarah	Math	C

Subjects

name	teacher
Math	Max
English	Toby

JSON ➔ Relational

- Can be tricky because of variation in structure of documents
- No obvious “best” way to translate into relational

<u>Example JSON Document that would be difficult to translate to a relational table</u>

```
{“student”: [
    {“name”: “Su Min”, “grade”: 86,
     “address”: {“street”: “21 Milan St.”, “city”: “Berkeley”, “state”: ”CA”}},
    {“name”: “Sarah”, “grade”: 55,
     “address” : “12 Franklin St., Berkeley, CA”,
     “email”: “sk@berkeley.edu”},
   ]
}
```

Structure of “address” is inconsistent across data, and so is the schema!

JSON Q1

Convert the following relational table into a JSON document.

Players

name	debut	goals
Tony	10/12/09	43
Katy	1/20/14	22

JSON Q1

Convert the following relational table into a JSON document.

Players

name	debut	goals
Tony	10/12/09	43
Katy	1/20/14	22

```
{“players”: [
    {“name”: “Tony”, “debut”: “10/12/09”,
     “goals”: 43},
    {“name”: “Katy”, “debut”: “1/20/14”,
     “goals”: 22}
   ]
}
```

JSON Q2

Convert the following JSON document into **two** relational tables, Players(name, debut) and Goals(name, goals).

```
{“players”: [
    {“name”: “Abby”, “debut”: “10/12/09”,
     “goals”: 43},
    {“name”: “Babby”, “debut”: “1/20/14”,
     “goals”: 22},
    {“name”: “Cabby”, “debut”: “1/21/14”,
     “goals”: 23}
   ]
}
```

JSON Q2

Convert the following JSON document into **two** relational tables: Players(name, debut) and Goals(name, goals).

```
{“players”: [
   {“name”: “Abby”, “debut”: “10/12/09”,
    “goals”: 43},
   {“name”: “Babby”, “debut”: “1/20/14”,
    “goals”: 22},
   {“name”: “Cabby”, “debut”: “1/21/14”,
    “goals”: 23}
  ]
}
```

Players

name	debut
Abby	10/12/09
Babby	1/20/14
Cabby	1/21/14

Goals

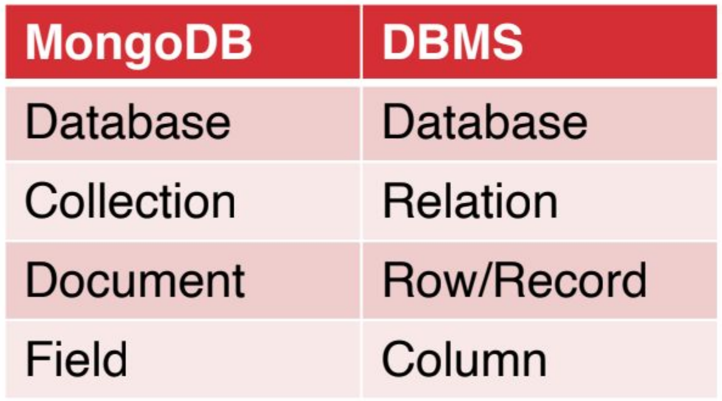
name	goals
Abby	43
Babby	22
Cabby	23

MongoDB Query Language (MQL)

MQL Data Model

- Same as JSON – Stores documents which are dictionaries of **field:value** pairs
  - E.g. Document = {Field_1: Value_1, Field_2:Value_2, ...}

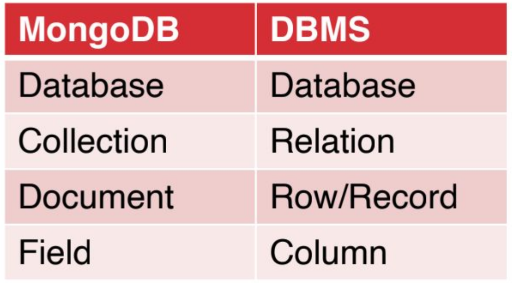
- Each document has a special "**_id**" field which represents its primary key
  - Indexed
  - First attribute of each doc



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

MQL

- Operate on collections (i.e. input: collection, output: collection)
- Use dot (.) notation to construct queries
  - i.e. `db.collection.operation1(...).operation2(...)`
    - where `collection` is the name of the collection and `operation1/2` is the name of the actual operation
- SQL aggregates multiple tables in the FROM clause, but MQL typically focuses on manipulating a single collection at a time



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

MQL Syntax

- **Values** in field:value pairs can be
  - Atomic
  - Nested document
  - Array of atomics
  - Array of nested documents

MQL Syntax– Dot (.) notation

- **Dot (.) notation** indexes into nested docs and arrays
  - “country.population” ➔ population field within the country field
    - Only applies when country’s value is a nested doc or array of nested docs
      - If it’s a nested doc, return the doc’s population field
      - If it’s an array of nested docs, return an array of all the docs’ population fields
  - “shopping_list.1” ➔ second element of shopping_list’s array
    - Applies when shopping_list’s value is an array
  - “shopping_list.1.price” ➔ price field of second element of shopping_list’s array
    - Applies when shopping_list’s value is an array where each value is a nested doc or array of nested docs
- **Note: Dot expressions need to be in quotes!**

MQL Syntax– Dollar ($) Notation

- **$** indicates the string is a special keyword
  - E.g. $gt, $lte, $add
- Used in the “field” part of a “field:value” expression
- For example, if the $ keyword is a binary operator, it’ll be used as {Left_Operand: {$keyword: Right_Operand}}
  - E.g. {“population”: {$gte: 1000}}

MQL Queries

- 3 main types of queries:
  1. **Retrieval**– essentially SELECT-WHERE-ORDER BY-LIMIT queries
  2. **Aggregation**– misnomer; in MQL it refers to a general pipeline of operators
  3. **Updates**

MQL: Retrieval Queries

MongoDB Query Language	Relational Database Equivalent

**find(<predicate>,
optional <projection>)**
	SELECT <projection>
	FROM Collection
	WHERE predicate

limit(<integer>)	LIMIT
sort(<list of fields>)	ORDER BY

MQL Retrieval: find()

- **find()** Template:

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

  - Both <predicate> and <projection> are expressed as documents {...}
- Note: db.collection.find({}) returns all documents
  - **Remember to replace "collection" with the collection's name**

MQL Retrieval: find() Examples

> db.dealership.find()

{“_id” : 1, “car” : “Honda”, “status” : “Sold”, “seats” : {“num”: 5, “type”: “leather”}, “reviews” : [5, 4, 5, 3, 1] }

{“_id” : 2, “car” : “Audi”, “status” : “In Stock”, “seats” : {“num”: 7, “type”: “polyester”}, “reviews” : [2, 4, 3, 4, 2] }

- **find(<predicate>, *optional* <projection>)**
  - *find({status : “In Stock”})* ➔ finds *exact* match ➔ returns second document
  - *find({seats:{“num”: 5, “type”: “leather”}})* ➔ finds exact match *including order of fields* ➔ returns first document

MQL Retrieval: find() Examples

```
> db.dealership.find()

{“_id” : 1, “car” : “Honda”, “status” : “Sold”, “seats” : {“num”: 5, “type”: “leather”}, “reviews” : [5, 4, 5, 3, 1] }

{“_id” : 2, “car” : “Audi”, “status” : “In Stock”, “seats” : {“num”: 7, “type”: “polyester”}, “reviews” : [2, 4, 3, 4, 2] }
```

- **find(<predicate>, *optional* <projection>)**
  - *find({reviews : [5, 5]})* → finds exact match → returns nothing
  - *find({reviews : 5, reviews : 4})* → finds *anything that has both elements* → returns first document

MQL Retrieval: find() Examples

> db.dealership.find()

{"_id" : 1, "car" : "Honda", "status" : "Sold", "seats" : {"num": 5, "type": "leather"}, "reviews" : [5, 4, 5, 3, 1] }

{"_id" : 2, "car" : "Audi", "status" : "In Stock", "seats" : {"num": 7, "type": "polyester"}, "reviews" : [2, 4, 3, 4, 2] }

- **find(<predicate>, *optional* <projection>)**
  - *find({$or: [{car: "Honda"}, {status: "In Stock"}]})* ➜ finds documents with car : "Honda" or status : "In Stock" ➜ returns both rows

MQL Retrieval: find() Examples

```
> db.dealership.find()

{“_id” : 1, “car” : “Honda”, “status” : “Sold”, “seats” : {“num”: 5, “type”: “leather”}, “reviews” : [5, 4, 5, 3, 1] }

{“_id” : 2, “car” : “Audi”, “status” : “In Stock”, “seats” : {“num”: 7, “type”: “polyester”}, “reviews” : [2, 4, 3, 4, 2] }
```

- **find(<predicate>, *optional* <projection>)**
  - Use 1’s to indicate fields you want, OR use 0’s to indicate fields you don’t want
    - *Can’t mix them* UNLESS you use 1’s for fields you want and a 0 ONLY for the “_id” field (it will be included by default otherwise)

MQL Retrieval: find() Examples

```
> db.dealership.find()

{"_id" : 1, "car" : "Honda", "status" : "Sold", "seats" : {"num": 5, "type": "leather"}, "reviews" : [5, 4, 5, 3, 1] }

{"_id" : 2, "car" : "Audi", "status" : "In Stock", "seats" : {"num": 7, "type": "polyester"}, "reviews" : [2, 4, 3, 4, 2] }
```

- **find(<predicate>, *optional* <projection>)**
  - *find({}, {car : 1, _id : 0})* → returns {"car" : "Honda"}, {"car" : "Audi"}
  - *find({}, {car : 1})* → returns {"_id" : 1, "car" : "Honda"}, {"_id" : 2, "car" : "Audi"}
  - *find({}, {car : 1, reviews : 0})* → returns Error (cannot mix 1's and 0's except for with _id)

MQL Retrieval: limit() Examples

```
> db.dealership.find()

{“_id” : 1, “car” : “Honda”, “status” : “Sold”, “seats” : {“num”: 5, “type”: “leather”}, “reviews” : [5, 4, 5, 3, 1] }

{“_id” : 2, “car” : “Audi”, “status” : “In Stock”, “seats” : {“num”: 7, “type”: “polyester”}, “reviews” : [2, 4, 3, 4, 2] }
```

- **limit()**
  - Like LIMIT in SQL
  - *db.dealership.find({}).limit(1)* ➜ Returns 1 document

MQL Retrieval: sort() Examples

```
> db.dealership.find()

{"_id" : 1, "car" : "Honda", "status" : "Sold", "seats" : {"num": 5, "type": "leather"}, "reviews" : [5, 4, 5, 3, 1] }

{"_id" : 2, "car" : "Audi", "status" : "In Stock", "seats" : {"num": 7, "type": "polyester"}, "reviews" : [2, 4, 3, 4, 2] }
```

- **sort()**
    - Like ORDER BY in SQL
    - Input is list of fields, -1 means descending, 1 means ascending
    - *db.dealership.find({}).sort({"reviews.0":-1}).limit(2)* ➔ Returns both documents ➔ First the Honda, then the Audi

MQL Aggregation

- **Aggregations** are another way of querying MongoDB
  - Consist of a pipeline of **stages** where each stage manipulates the collection in some way
    - Depending on which stages are used, an aggregation can also do a retrieval

MQL Aggregation: Stage Types

- A **stage** can be one of the following **types**:
  - match – the first argument of find()
  - project – the second argument of find() (has more expressiveness)
  - sort – same as retrieval
  - limit – same as retrieval
  - group
  - lookup
  - unwind
  - more...

MQL Aggregation Syntax

- **Aggregations** are structured as

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

MQL Aggregation: Grouping

Grouping is done with the **$group** operator:

```
    $group : {
        _id : <expression>, // Same as a relational GROUP BY
        <field1> : {<aggregation_func1> : <expression1>},
    ...}
```

- The **aggregation_func1** can be standard operations like $sum, $avg, $max
    - Some MQL specific ones include
        - $first: return 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
        - $addToSet: like $push, but eliminates duplicates

MQL Grouping Example

```
> db.dealership.find()

{“_id” : 1, “car” : “Honda”, “status” : “In Stock”, “seats” : {“num”: 5, “type”: “leather”}, “price” : 12.1 }

{“_id” : 2, “car” : “Audi”, “status” : “In Stock”, “seats” : {“num”: 7, “type”: “polyester”}, “price” : 15.2}

{“_id” : 2, “car” : “Audi”, “status” : “Sold”, “seats” : {“num”: 7, “type”: “polyester”}, “price” : 14.8}
```

```
db.dealership.aggregate([

        {$group : {_id : “$status”, avgPrice : {$avg : “$price”}}},

        {$match : {avgPrice : {$lte : 14}}

])
```

Note: remember to use **“$field_name”** format when referring to fields in the values of the $group stage.

MQL Grouping Example

```
> db.dealership.find()

{"_id" : 1, "car" : "Honda", "status" : "In Stock", "seats" : {"num": 5, "type": "leather"}, "price" : 12.1 }

{"_id" : 2, "car" : "Audi", "status" : "In Stock", "seats" : {"num": 7, "type": "polyester"}, "price" : 15.2}

{"_id" : 2, "car" : "Audi", "status" : "Sold", "seats" : {"num": 7, "type": "polyester"}, "price" : 14.8}
```

```
db.dealership.aggregate([

        {$group : {_id : "$status", avgPrice : {$avg : "$price"}}},

        {$match : {avgPrice : {$lte : 14}}

])
```

Output:

```
{"_id" : "In Stock",
"avgPrice" : 13.65}
```

MQL Aggregation: Lookup

- Lookup is the only way to do joins in MongoDB but it's messy:

```
{$lookup : {

    from : <collection to join>,

    localField : <referencing field>,

    foreignField : <referenced field>,

    as : <output array field>

}}
```

- This query is essentially saying:
  - For each document in this collection,
    - Find documents in the "**from**" collection whose "**foreignField**" matches the "**localField**"
    - Put all these matching documents into an array in the calling document under the "**as**" key
- Use $project afterward to clean up the output if needed

MQL Aggregation: Lookup Example

```
 db.dealership.find()
{“_id” : 1, “car” : “Honda”, “status” : “In Stock”, “seats” : {“num”: 5, “type”: “leather”}, “price” : 12.1 }
{“_id” : 2, “car” : “Audi”, “status” : “In Stock”, “seats” : {“num”: 7, “type”: “polyester”}, “price” : 15.2}
{“_id” : 2, “car” : “Audi”, “status” : “Sold”, “seats” : {“num”: 7, “type”: “polyester”}, “price” : 14.8}

db.dealership.aggregate([
        {$lookup : {from : “dealership”, localField : “status”, foreignField : “status”, as : “stock”},
        {$project : {_id : 0, seats: 0, “stock._id” : 0, “stock.seats” : 0}}
])
```

Output:

```
{“car” : “Honda”, “status” : “In Stock”, “price” : 12.1, “stock” : [
        {“car” : “Honda”, “status” : “In Stock”, “price” : 12.1, “stock”},
        {“car” : “Audi”, “status” : “In Stock”, “price” : 15.2}
]}
...
```

MQL Updates

- **Updates** are the 3rd major type of MongoDB query
- Types of updates:
  - InsertOne/InsertMany
  - UpdateOne/UpdateMany
  - DeleteOne/DeleteMany
- The Many case is more general, so we’ll use that as an example

MQL Updates: insertMany

- Similar to an insert from relational databases
  - Insert list of documents into collections
- For example:

```
db.aquarium.insertMany([
    {“fish” : “yellowtail”, “price” : “30”, status : “friend”},
    {“fish” : “tuna”, “price” : 20, status : “food”}
])
```

  - Will create aquarium collection if needed
  - Will add _id attributes to each document since it doesn’t already exist
  - _id will be first field by default

MQL Updates: updateMany

- Syntax: db.Collection.**updateMany**({<Condition>}, {<Change>})
- Example:

```
db.aquarium.updateMany(
        {price : {“$lt” : 35}},
        {$set : {status : “food”}}
)
```

```
db.aquarium.find()  ➜  Output
                        {“fish” : “yellowtail”, “price” : “30”, status : “food”},
                        {“fish” : “tuna”, “price” : 20, status : “food”}
```

MQL Q1

Consider the following MongoDB collection **teams**.

(**teamId**: int, **divisionId**: int, **stadiumCapacity**: int, **wins**: int, **losses**: int, **coach**: string, **captain:** string)

Write an MQL query to find the **coach** and **captain** of all teams from **division** 1 with at least 10 **wins**, sorted by **coach** DESC and ties broken by **captain** ASC.

MQL Q1

Consider the following MongoDB collection **teams**.

Write an MQL query to find the **coach** and **captain** of all teams from **division** 1 with at least 10 **wins**, sorted by **coach** DESC and ties broken by **captain** ASC.

```
db.teams.aggregate([
   {$match: {wins: {$gte: 10}, divisionId: 1}},
   {$sort: {"coach": -1, "captain": 1}},
   {$project: {"coach": 1, "captain": 1, "_id": 0}}
])
```

MQL Q2

Translate the following SQL query into an MQL query:

```
SELECT divisionId AS div, MAX(wins) AS maxWins
FROM teams
WHERE stadiumCapacity >= 20000
GROUP BY divisionId
ORDER BY MAX(wins), COUNT(*) DESC;
```

MQL Q2

Translate the following SQL query into an MQL query:

SELECT divisionId **AS** div, **MAX**(wins) **AS** maxWins
FROM teams
WHERE stadiumCapacity >= 20000
GROUP BY divisionId
ORDER BY MAX(wins), **COUNT**(*) **DESC**;

```
db.teams.aggregate([
    { $match: {
        stadiumCapacity: {$gte: 20000}
    }},
    { $group: {
        _id: "$divisionId",
        maxWins: {$max: "$wins"},
        count: {$sum: 1}
    }},
    { $sort: {
        "maxWins": 1,
        "count": -1
    }},
    { $project: {
        div: "$_id",     ← similar to aliasing in SQL
        _id: 0,
        maxWins: 1
    }}
])
```

Attendance Link

https://cs186berkeley.net/attendance