CS143: MongoDB (NoSQL)

Book Chapters

(7th) Chapter 10.2

MongoDB

- Database for JSON objects
  - "NoSQL database"
- Schema-less: no predefined schema
  - MongoDB will store anything with no complaint!
  - No normalization or joins
  - Use `Mongoose` for ensuring structure in the data
- Adopts JavaScript philosophy
  - "Laissez faire" policy
    - Don't be too strict! Handle user request in a "reasonable" way
  - Both blessing and curse

Document in MongoDB

- Data is stored as a *collection* of *documents*
  - *Document*: (almost) JSON object
  - *Collection*: group of "similar" documents
- Example

```
{
    "_id": ObjectId(8df38ad8902c),
    "title": "MongoDB",
    "description": "MongoDB is NoSQL database",
    "tags": ["mongodb", "database", "NoSQL"],
    "likes": 100,
    "comments": [
        { "user":"lover", "comment": "Great book!" },
        { "user":"hater", "comment": "Worst ever!" }]
}
```

- `_id` field: primary key
  - Its value must be unique in the collection
  - May be of any type other than array
  - If not provided, `_id` is automatically added with a unique `ObjectId` value
- Stored as BSON (Binary representation of JSON)
  - Supports more data types than JSON
  - Does not require double quotes for field names
- Analogy
  - Document in MongoDB ≈ row in RDB
  - Collection in MongoDB ≈ table in RDB

MongoDB vs RDB

MongoDB document

- Preserves structure
  - Nested objects
- Potential redundancy
- Hierarchical view of a particular app
- Retrieving data with different "view" is difficult

RDB relation

- "Flattens" data
  - Set of flat rows
- Removes redundancy
- Flat schema based on the intrinsic nature of data
- Easy to obtain different "view" using efficient "joins"

Basic MongoDB Commands

- Basic administration
  - `mongo`: start MongoDB shell
  - `use <dbName>`: use the database
  - `show dbs`: show list of databases
  - `show collections`: show list of collections
  - `db.colName.drop()`: delete `colName` collection
  - `db.dropDatabase()`: delete current database
- CRUD operations
  - Create: `insertOne()`, `insertMany()`

  - Retrieve: `findOne()`, `find()`
  - Update: `updateOne()`, `updateMany()`
  - Delete: `deleteOne()`, `deleteMany()`

MongoDB commands for CRUD

- Create: `insertX(doc(s))`

```
db.books.insertOne({title: "MongoDB", likes: 100})
db.books.insertMany([{title: "a"}, {title: "b"}])
```

- Retrieve: `findX(condition)`

```
db.books.findOne({likes: 100})
db.books.find({$and: [{likes: {$gte: 10}}, {likes: {$lt: 20}}]})
```

  - `findOne()` returns the first (?) matching document for multiple matches
  - Other boolean/comparison operators: `$or`, `$not`, `$gt`, `$ne`, ...

- Update: `updateX(condition, update_op)`

```
db.books.updateOne({title: "MongoDB"}, {$set: {title: "MongoDB II"}})
db.books.updateMany({title: "MongoDB"}, {$inc: {likes: 1}})
```

  - Other update operators: `$mul` (multiply), `$unset` (remove the field), ...

- Delete: `deleteX(condition)`

```
db.books.deleteOne({title: "MongoDB"})
db.books.deleteMany({likes: {$lt: 100}})
```

MongoDB Queries: Aggregates

- MongoDB allows posing complex queries using "aggregates"

  - MongoDB aggregates ≈ SQL select queries
  - An "aggregate pipeline" consists of multiple "aggregate stages"
    * pipeline ≈ select statement
    * stage ≈ select clause

- Example

```
{ _id: 1, cust_id: "a", status: "A", amount: 50 }
{ _id: 2, cust_id: "a", status: "A", amount: 100 }
{ _id: 3, cust_id: "c", status: "D", amount: 25 }
{ _id: 4, cust_id: "d", status: "C", amount: 125 }
{ _id: 5, cust_id: "d", status: "A", amount: 25 }
```

```
db.orders.aggregate([
    { $match: { status: "A" } },
    { $group: { _id: "$cust_id", total: { $sum: "$amount" }, count: {
       $sum: 1 }}},
    { $sort: { total: -1 } }
]);
```

  - `$match` ≈ where
  - `$group` ≈ group by
    - `_id` is the group by attribute
  - `$sort` ≈ order by
  - `$limit` ≈ fetch first
  - `$project` ≈ select
  - `$unwind`: replicate document per every element in the array
    - `{ $unwind: "y"}` converts `{"x": 1, "y": [1, 2] }` to `{"x": 1, "y": 1}, {"x": 1, "y": 2 }`
  - `$lookup`: "look up and join" another document based on attribute value
    - {$lookup: { from: <collection to join>, localField: <local join attr>, foreignField : <remote join attr>, as: <output field name> }}
    - matching documents are returned as an array in <output field name>

- More on MongoDB aggregates
  - Short tutorial: https://studio3t.com/knowledge-base/articles/mongodb-aggregation-framework/
  - Reference: https://docs.mongodb.com/manual/reference/method/db.collection.aggregate/

Index

- Indexes can be built for efficient retrieval
- `db.books.createIndex({title:1, likes:-1})`
  - Create one index on combined attributes "title" and "likes"
  - 1 means ascending order, -1 means descending order

More on MongoDB

- We learned just the basic
  - Enough for our project
- But MongoDB has many more features:
  - Aggregate queries
  - Transactions
  - Replication
  - (Auto)sharding

  - ...
- Read MongoDB documentation and online tutorials to learn more

CS143: Map Reduce (Spark)

Book Chapters

(7th) Chapters 10.3-4

Distributed Computing on Cluster

- Often, our data is non-relational (e.g., flat file) and huge
  - Billions of query logs
  - Billions of web pages
  - ...
- Q: Can we perform analytics on large data quickly using thousands of machines? How can we help programmer write parallel code running in distributed clusters?

Examples

Example 1: Search log analysis

- Log of billions of queries. Count frequency of each query

```
Input query log:
    cat,time,userid1,ip1,referrer1
    dog,time,userid2,ip2,referrer2
    ...
Output query frequency:
    cat 200000
    dog 120000
    ...
```

  - Log file is spread over many machines

- Questions
  - Q: How can we do this?
  - Q: How can we run it on thousands of machines in parallel?
    - Q: Can we process each query log entry independently?
    - Q: How can we combine the results?

Example 2: Web Indexing

- 1 billion pages. build inverted index

```
Input documents:
    1: cat chases dog
    2: dog hates zebra
    ...
Output index:
    cat 1,2,5,10,20
    dog 2,3,8,9
    ...
```

- Questions
  - Q: How can we do this?
  - Q: How can we run it on thousands of machines?
    - Q: Can we process each page independently?
    - Q: How can we aggregate extracted (word, docid)'s?

Generalization of Examples

- Common pattern in the two examples
  - Input data consists of multiple independent units
    - Each line of query log
    - Each web page
  - Partition input data into multiple "chunks" and distribute them to multiple machines
  - Transformation/map input into (key, value) tuples
    - Query log: query_log_line → (query, 1)
    - Indexing: web_page → (word1, page_id), (word2, page_id), ...
  - Reshuffle tuples of the same key to the same machine
  - Aggregate/reduce the tuples of same keys
    - Query log: (query, 1), (query, 1), ... → (query, count)
    - Indexing: (word, 1), (word, 3), ... → (word, [1, 3, ...])
  - Collect and output the aggregation results
- The two examples are almost the same except
  - "The mapping function"
    - Query log: query_log_line → (query, 1)
    - Indexing: web_page → (word1, page_id), (word2, page_id), ...
  - "The reduction function"
    - Query log: (query, 1), (query, 1), ... → (query, count)
    - Indexing: (word, 1), (word, 3), ... → (word, [1, 3, ...])

Map/Reduce Programming Model

- Many data processing jobs can be done as a sequence of
  1. Map: $(k, v) \to (k', v'), (k'', v''), ...$
  2. Reduce: partition/group by k and "aggregate" v's of the same k
  - Output of map function depends only on the input (k, v), not any other input
    * Each map task can be executed independently of others
  - Reduction on different keys are independent of each other
    * Each reduce task can be executed independently of others
    * Reduce function should be agnostic to the order of v's
- If any data processing follows the previous pattern, they can be parallelized by
  1. Split the input into independent chunks
  2. Run "map" tasks on the chunks in parallel on multiple machines
  3. Partition the output of the map task by the output key
  4. Move data of the same partition to the same node
  5. Run one reduce task per each partition
  - Only the map and reduce functions are different per app
- Under Map/Reduce programming model:
  - Programmer provides
    * Map function $(k, v) \to (k', v')$
    * Reduce function $(k, [v1, v2, ...]) \to (k, aggr([v1, v2, ...]))$
  - MapReduce handles the rest
    * Automatic data and task, partition, distribution, and collection
    * Failure and speed disparity handling

Systems

Hadoop

- First open source implementation of GFS (Google File System) and MapReduce
  - Implemented in Java
- Map and reduce functions are implemented by:
  - `Mapper.map(key, value, output, reporter)`
  - `Reducer.reduce(key, value, output, reporter)`

Spark

- Open source cluster computing infrastructure
- Supports MapReduce and SQL
  - Supports data flow more general than simple MapReduce
- Input data is converted into RDD (resilient distributed dataset)

  - A collection of independent tuples
  - The tuples are automatically distributed and shuffled by Spark

- Supports multiple programming languages

  - Scala, Java, Python, ...
  - Scala and Java are much more performant than others

Spark: Example

- Count words in a document

```
lines = sc.textFile("input.txt")
words = lines.flatMap(lambda line: line.split(" "))
word1s = words.map(lambda word: (word, 1))
wordCounts = word1s.reduceByKey(lambda a,b: a+b)
wordCounts.saveAsTextFile("output")
```

  - `map()`: one output per one input
  - `flatMap()`: multiple outputs per one input

Key Spark Functions

- Transformation: Convert RDD tuple into RDD tuple(s)

  - `map()`: convert one input tuple into one output tuple
  - `flatMap()`: convert one input into multiple output tuples
  - `reduceByKey()`: specify how two input "values" should be aggregated
  - `filter()`: filter out tuples based on condition

- Action: Perform "actions" on RDD

  - `saveAsTextFile()`: save RDD in a directory as text file(s)
  - `collect()`: create Python tuples from Spark RDD
  - `textFile()`: create RDD from text (each line becomes an RDD tuple)