Data 101: Data Engineering Note 9 – October 25, 2022

Semistructured Data

Lecturer: Lakshya Jain *Scribe: Shreya Shankar*

1 Introduction

In previous lectures, we've worked with relational data, where the schema is well-defined, and each row has a fixed set of attributes or columns. Each attribute's type is pretty simple (e.g., int), and there is no nesting. But not all data is so well-formed. In recent years, unstructured data has been stored in larger quantities than structured data, due to storage costs decreasing and the ease of logging unstructured data. How do we work with this kind of data?

2 Semi-Structured Data

Semi-structured data is a data representation or data model that is less "rigid" than structured or rectangular data. The schema is flexible and attributes can be nested. Common data models for semi-structured data are key-value stores and document stores. NoSQL databases provide the ability to store and query such semi-structured data.

3 Scaling

There are two ways to think about scaling a database (e.g., storing more data, processing more queries): partitioning and replicating. In both ways, we need to guarantee consistency, or the fact that the database must reflect the most recent changes (e.g., INSERTS).

- Partitioning spreads the database across many machines in the cluster. This is good for writes but bad for reads, since we have to read from the exact partition the data is written on.
- Replication creates multiple copies of the same piece of data and spreads them across different replicas, or machines in a cluster. This is good for reads (because you can query any of the relevant replicas for a piece of data) but bad for writes, since we must write to all the replicas.

There is a consistency-availability tradeoff: high consistency means data isn't always available (because all replica updates need to be synchronized). In general, consistency is easier to achieve with relational databases. Availability is easier to achieve with NoSQL databases.

4 Key-Value Stores

The data model here is, simply, key-value pairs. Keys must be unique and are typically strings or ints. Values can be any object. KV stores offer two operations—`get` and `put`—to add and query data based on a key. Querying based on value is not supported.

KV stores use both partitioning and replication, but they replicate on only a fraction of servers. They promise *eventual* consistency, so an app may read a stale version of data in its query. However, KV stores have high availability.

5 JSON and XML

Sometimes the values in a key-value store will be complex, and we can represent them via JSON or XML. JSON and XML formats are textual representations of nested data. JSON looks like a dictionary:

```
{
  key1: val1,
  key2: {key2.1: val2.1}
}
```

while XML is a markup language:

```
<head>
  <key1> val1 </key1>
  <key2>
    <key2.1> val2.1 </key2.1>
  </key2>
</head>
```

XML predated JSON.

6 Querying Semistructured Data

To query semi-structured data, we'll look into the syntax of MongoDB Query Language (MQL). There are three main types of queries:

- Retrieval: restricted SELECT-WHERE-ORDERBY-LIMIT queries
- Aggregation: a pipeline of operators
- Updates: adding, changing, or removing data

Queries are usually pretty messy, as we'll see. This is because JSON itself is pretty messy—you sacrifice simplicity when you give up "flatness" of the data.

6.1 Notation

There are two special forms of notation: dot (.) notation and dollar ($) notation. "." is used to drill deeper into nested objects/arrays, e.g., `person.address`. "$" is used to denote a special keyord, like an operator, e.g., `$gt` means greater than and `$elemMatch` refers to the elemMatch function.

6.2 Retrieval Queries

The most common query is of the form:

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

where the predicates can be combined. Combining predicates looks something like:

```
db.collection.find({ status : "D", qty : {$gte : 50} })
```

Retrieval queries end up looking like this:

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

```
sort()  =  ORDER BY
```

6.3 Aggregation Queries

Aggregation queries are essentially a linear pipeline of stages. Each stage corresponds to an operation like match, lookup, group, sort, limit, etc. An example to find states with population $>$ 15M, sort by descending order:

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

Note that there 3 pipelined operators, each of which follow the same JSON-like syntax.

6.4 Updates

Update queries follow the syntax of [insert/delete/update] [one/many]. For example:

```
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 ] }
] );
```

Most of the time, you will want to use insertMany or updateMany. Rarely do queries change or add only one thing at a time.