Databases and information management

Lecture 9

Piotr Syga

2022-11-24

NotOnly SQL

- designed with distributed data in mind
- capability of handling large number of 'records'
- adaptability to cope with lightweight as well as multimedia files
- no consistent schema required
- providing significant horizontal scalability (adding data nodes)

CAP

We replace the ACID requirements with ones adjusted to distributed environment:

- (C*onsistency*) after performing the operation,
- (A*vailability*) of the data without interrupts,
- (P*artition tolerance*) when data nodes cannot exchange messages in specific time periods.

In practice we fulfill two of the requirements, e.g., relational databases(CA), MongoDB/HBase(CP), CouchDB/Cassandra(AP).

NOSQL's downsides

- lack of standardization for all the languages (dialects)
- lack of a universal, declarative query language
- the capabilities of the queries is constrained, often requires additional programming language snippets
- BASE transactions

BASE

In place of a restrictive ACID model, we use BASE model, which allows some leniency:

Basically Available – we have access to the data withou *major* interrupts

Soft state – the data in a node may change even if there is no external data manipulation

Eventual consistency – means that without external data manipulation, given sufficient time, the nodes will synchronize and provide consistent data

Main types of the databases

Depending on the required functionalities and purpose of the database we may organize the data in various ways

- in dictionaries
- wide-column store
- in documents
- in graphs

Key-value store

Storing the data in a dictionary allows significant improvement of data retrieving time*, at the cost of lack of uniform datastructure and data duplication.

- Amazon Dynamo as a cornerstone
- storing data often implemented with hashtables
- fulfills Availability and Partition tolerance
- hashing allows operations on large datasets
- mimics object-oriented programming
- allows storing data of an arbitrary structure**
- strong scalability

Examples: Dynamo, Redis, Riak, Virtuoso

Columnstore

Storing data in columns allows easy aggregation for data analysis, in contrast to RDB the schema is not fixed

- data for the columns are stored in consecutive blocks in a single file
- each column is treated separately
- similar to two-way (2D) dictionary
- in a single column, there is only one datatype
- reducing time overhead on aggregation, BI analysis, data warehousing, however significantly reduced when using multi-column aggregation
- easy and efficient data compression

Examples: Bigtable, Cassandra, HBase, SimpleDB

Documentstore

Storing data in documents allows you to store incomplete or non-atomic data, at the expense of operation time and syntax differences for seemingly similar queries (cf. subdocuments, lists, arrays, sets).

- can be interpreted as a generalization of dictionary (multikey-multivalue store)
- data organized into document collections
- documents do not realize any fixed schema and allow nested (sub)documents, the field scope is (almost) unlimited
- allow connections between document collections
- one may find RDB analogies (a row–document, a column–dictionary, a table–collection,. . .)

Examples: Caché, CouchDB, Elasticsearch, MongoDB

Graphstore

Storing data in graphs allows the use of algorithms known from Graph Theory (checking the degree of a node, checking connectivity, distance and reachability, finding the shortest path between two nodes, . . .) to process the data, at the cost of time-consuming base joining and the requirement to search the entire graph to obtain an (accurate) answer.

- the base consists of ordered pairs of vertices (each pair uniquely identifies an edge)
- both vertices and edges have unique labels or IDs
- vertices represent (often complex) entities
- each vertex knows their neighbors
- one can easily identify the connection between the data
- the cost of traversing an edge (a single hop) is constant
- efficient for inter-connected (dense) data; easy indexing
- one may find RDB analogies (a row –vertex, a table–subgraph, JOIN–edges. . .)
- no universal graph query language (ongoing proposal: ISO/IEC 39075 Information Technology — Database Languages — GQL)

Examples: Amazon Neptune, Janus Graph, Neo4J, TerminusDB

Basics

- data organized in documents⇒collections
- can use indexes to speed up searches (as in RDB)
- flexible data representation
- in a single document all operations are atomic
- easy updates and replacing of documents
- server mirroring
- data sharding and live resharding
- hierarchical data segmentation – Map/Reduce
- supports bindings of many programming languages (np. pyMongo, mongocxx, mongo.jar)
- supports transactions and views
- client-side field level encryption

Current stable version: 6.0.3

BSON

All documents are stored in BSON format.

- the structure of the document is created dynamically and may be easily changed
- each document has its (eventually) unique 12B ID (4B timestamp, randomly generated 5B string, 3B counter)
- 21 datatypes (4 of them deprecated)
- default fields for UUID, MD5 and BSON client-side encryption field
- datatypes may be evoked by the name or their ID
- types might be assigned automatically, based on the value or manually (to verify the type of a field use `instanceof`, `typeof`)

Data bases

Note that if database or a collection does not exist, they are created at their first evocation

- creating or switching the database: `use newDataBase`
- using a different database (or collection from such) without switching: `db.getSiblingDB('secondDB').*` (e.g.,`db.getSiblingDB('secondDB').getCollectionNames()`)
- (in)direct creation of a collection: `db.firstCol.insertOne( { key: val } )` (or `db.createCollection(name, options)`)

The names of both databases and collections are restricted similarly to most programming languages – details. As opposed to SQL by default the names are case sensitive.

Documents

Structure

```
{ field1:  value,
field2:  value,
...
fieldN: value}
```

`_id` field is added automatically with `ObjectId` type.

Documents

Arrays and embedded documents

Fields may use non-atomic types as arrays, e.g., `color: ["white", "red"]`. You may address the field by its position `color.2` or reference all positions (e.g., at updating) with `color.[]`.
Matching: `["white", "red"]` vs `$all: ["white", "red"]`
Non-atomic fields may also be a document, e.g., `phone: {home: "XXX", cell: "YYY"}`. We reference the values hierarchically with dot notation `phone.cell`.
Naturally, one may create arrays of embedded documents and embedded documents that include arrays.

Default size limit of a single document (with embeddings) is 16MB, yet can be modified.

Basic operations

- adding a single document: `db.collection.insertOne()`
- adding an array of documents: `db.collection.insertMany()`
- queries: `db.collection.find()`
- updates: `db.collection.update()`, `db.collection.findAndModify()`
- adding fields: `db.collection.update()` with `$inc:{newField: value}`
- counting documents (with condition): `db.collection.count()`, `db.collection.find({speed:{$lt:240}}).count()`
- deleting documents: `db.collection.remove({condition})`,`db.collection.remove()`
- dropping a collection or a database: `db.collection.drop()`/`db.dropDatabase()`

Finds

SQL

```
SELECT * FROM students WHERE name='Doe'
```

```
db.students.find({"name":  "Doe"}).pretty()
```

Some of the conditions:

- comparison: $eq, $ne, $gt, $gte, $lt, $lte, $in, $nin
- logical conjunctions: $and, $not, $or, $nor
- array operators: $all, $elemMatch, $size
- evaluations, e.g., $regex, $mod
- selecting only fixed fields: `.find({condition},{field1:  0, field2:  true, field3:1, field:false})`

Finds

SQL

```
SELECT * FROM students WHERE name='Doe'
```

```
db.students.find({"name":  "Doe"}).pretty()
```

SQL

```
SELECT * FROM students
WHERE course="CS" AND (semester=1 or semester=2);
```

```
db.students.find({ course :  "CS" , $or :  [{ semester :
1 }, { semester :  2 }]}).pretty()
```

Update

Schema

`db.collection.update(`*query*`,` *modifications*`,`*options*`)`

Important: `upsert` and `multi` options are false by default

Example: Changing field value

```
db.studenci.update({semestr:3},{$set:{semestr:4}})
```

Update

Example: overwriting a document

```
db.students.save({"_id" :
ObjectId(6083341721031b0c459c2), "name":"John", "surname":
"Smith", "comment":"foreign exchange student"})
```

Example: Modifying all the fields

```
db.students.update({ name:  "John", surname:"Smith" },
{name:  "John", surname:"Smith", ID:123456789, semester:1,
faculty:  "W11", major:"BDA"})
```

Removing documents

Deleting with ID

Similarly to SQL, the safest way to delete a document is by using ID (a primary key) – `db.students.find({surname: {$regex:'Smi'}},{_id:1})` and then `db.students.deleteOne({ _id: XXX})`

Direct deleting

Alternatively, one can delete the result of the selection `db.students.deleteOne({surname: 'Smith'})`

- just as with inserts (`*.insertOne()`, `*.insertMany()`) one can delete one (first match) or more documents (`*.deleteOne()`, `*.deleteMany()`)
- note that even though the effect of `remove()` and `deleteMany()` is similat there are important differences – default number of deleted elements, the returned type (object vs document), both commands may not be viable in different programming languages (`remove()` is deprecated in many of them)

Aggregation

- simple aggregation is possible e.g., `db.collection.estimatedDocumentCount()`, `db.collection.count()`
- MongoDB utilizes aggregation pipelines, that are successors of Map/Reduce
- the aggregation map `db.collection.aggregate([])` consists of 2 main stages – matching and grouping
- `$match` filters out the documents that are passed to the grouping stage, e.g., `{ $match: { name: "John" } }`
- `$group` does two things: groups the documents based on the value of the grouping field and performs the aggregate function, e.g., `{ $group: { _id: "$major", averageMarks: { $avg: "$grade"} } }`
- additional stage – `$setWindowFields` may be performed additionally to add partitions, sorts, and document ranges

Aggregation

Selected aggregate functions for grouping operator (accumulator operators) include:

- known from SQL: $avg, $count, $max, $min, $sum and two variants of std: $stdDevPop, $stdDevSamp
- document list and set operations: $first, $last, $mergeObjects, $push, $addToSet
- custom $accumulator function

Aggregation

$accumulator definition

{ $accumulator: {
init: <code>,
initArgs: <array expression>,
accumulate: <code>,
accumulateArgs: <array expression>,
merge: <code>,
finalize: <code>,
lang: <string>
} }

Aggregation

$accumulator example – data

```
db.books.insertMany([ { '_id' : 8751, 'title' : 'The Banquet', 'author' :
'Dante', 'copies' : 2 }, { '_id' : 8752, 'title' : 'Divine Comedy', 'author' :
'Dante', 'copies' : 1 }, { '_id' : 8645, 'title' : 'Eclogues', 'author' :
'Dante', 'copies' : 2 }, { '_id' : 7000, 'title' : 'The Odyssey', 'author' :
'Homer', 'copies' : 10 }, { '_id' : 7020, 'title' : 'Iliad', 'author' :
'Homer', 'copies' : 10 } ])
```

Aggregation

```
db.books.aggregate([{$group : { '_id : "$author", avgCopies:{
$accumulator: init: function() { //Set the initial state
return { count: 0, sum: 0 } },
accumulate: function(state, numCopies) { // Define how to update the state
return {count: state.count + 1, sum: state.sum + numCopies } },
accumulateArgs: ["$copies"], // Argument required by the accumulate function
merge: function(state1, state2) { // When the operator performs a merge,
return { //add the fields from the two states
count: state1.count + state2.count,
sum: state1.sum + state2.sum } },
finalize: function(state) { //After collecting the results from all documents,
return (state.sum / state.count) // calculate the average
},
lang: "js" } }}}])
```

Aggregation

Additional aggregation stages

Aside from 2 main stages and `$setWindowFields` the aggregation pipeline allows additional stages, some that are worth checking inluce:

- $project
- $lookup
- $unwind
- $sort
- $limit
- $skip