13 MongoDB

Guirong Fu

January 28 2019

Keywords latency, index, replica, write concern, schema validation, operation level, document format, collection, language

MongoDB is a powerful, flexible, and scalable general-purpose database. It combines the ability to scale out with features such as secondary indexes, range queries, sorting, aggregations, and geospatial indexes.

1 Entities

Document basic unit of data, roughly equivalent to a **row** in RDMS. Each document has a **special key**, "_id", that is unique within a collection. It is an **ordered** set of **key-value pairs**.

Collection can be thought of as a **table with a dynamic schema**[1]

Databases a single instance of MongoDB can host zero or more independent databases, each of which can have its own collections.

1.1 Document

Document-oriented database, not a relational one: to make scaling out easier. Documents makes MongoDB possible to represent complex hierarchical relationships with a single record.

Keys strings, no duplicate. Any UTF-8 character is allowed in a key, with a few exceptions: \0(must not be contained, which delineates the end of a collection name), . , & (these two are reserved).

type-sensitive and case-sensitive "foo" and "Foo" are different.

ordered {"x":1,"y":2} is not the same as {"y":2,"x":1}. However, in some programming languages the default representation of a document does not even maintain ordering.

[1] what does the dynamic mean?

1.2 Collections

Dynamic Schemas The documents within a single collection can have any number of **different "shapes"**. Any document can be put into any collection. [2]

Naming name is an identifier for a collection, can be any UTF-8 string, with a few restrictions: empty string("") (is not a valid collection name), the null character \0(cannot be contained), prefix ("system") (reserved) and "$" (reserved).

"Subcollections" Some collection's name including ".", e.g. "blog.authors". This is for organizational purpose only. There is **no relationship** between the "blog" collection and its "children", like "blog.authors".

2 Data Types

Documents in MongoDB can be thought of as "JSON-like".

2.1 Basic Data Types

In addition to six basic data types in JSON, MongoDB also includes: date, several kinds of number (the shell defaults to using 64-bit floating point numbers, differentiating float, double, integer), regular expression and so on.

2.2 ObjectIds

The "_id" key's value can be any type, but it defaults to an ObjectId.

In a single collection, every document must have a unique value for "_id". However, if you had two collections, each one could have a document where the value for "_id" was 123.

3 Documents' Operations

3.1 Insert

single db.collection-name.insert({"key1":"value1"})

batch using an arrary: db.collection-name.insert([{"key1":"value1"},{"key2":"value2"},...])

Insert Validation MongoDB only does **minimal checks** on data being inserted: basic structure (size: smaller than 16 MB)[3] and adds an "_id" field if one does not exist.

All of the drivers for major languages (and most of the minor ones, too) do check for a variety of invalid data (documents that are too large, contain non-UTF-8 strings, or use unrecognized types) before sending anything to the database.

[2] what is the function of schema?

[3] This is a somewhat arbitrary limit (and may be raised in the future); it is mostly to prevent bad schema design and ensure consistent performance.

3.2 Remove

Once data has been removed, it is gone forever. There is no way to undo the remove or recover deleted documents.

Using "db.cname.remove({"key1":"value1"})". The remove function optionally takes a query document as a parameter.

If you want to clear an entire **collection**, it is faster to drop it

3.3 Updating

update takes two parameters:
- a query document, which locates documents to update,
- a modifier document, which describes the changes to make to the documents found.

Atomic Updating a document is atomic: if two updates happen at the same time, whichever one reaches the server first will be applied, and then the next one will be applied.

update modifiers special keys that can be used to specify complex update operations, such as altering, adding, or removing keys e.g.
- "$inc": can be used only on values of type integer, long, or double. If it is used on any other type of value, it will fail.
db.cname.update({"key":"value"},...{"$inc":{"key2":1}})
- "$set" sets the value of a field. If the field does not yet exist, it will be created.
db.cname.update({"key1" : "value1"}, ... {"$set" : {"key2" : "value2"}})
- "$push" adds elements to the end of an array if the array exists and creates a new array if it does not.
- "$ne": check values if they are not present.
- "$addToSet" to prevent duplicates.

Always use $ operators for modifying individual key/value pairs. Otherwise, update will do a full-document replacement.

Modifier speed Array modifiers might change the size of a document and can be slow.
Some modifiers are faster than others, which does not have to change the size of a document.

Padding factor It is the amount of extra space MongoDB leaves around new documents to give them room to grow.
Initially, documents are inserted with no space between them.
If a document grows and must be moved, free space is left behind and the padding size in increased.
If subsequent updates cause more moves, the padding factor will continue to grow (although not as dramatically as it did on the first move). If there aren't more moves, the padding factor will slowly go down.

3.4 Write Concern

Write concern is a client setting used to describe how safely a write should be stored before the application continues. By default, inserts, removes, and updates wait for a database response—did the write succeed or not?—before continuing.

The two basic write concerns are **acknowledged or unacknowledged** writes. Acknowledged writes are the default: you get a response that tells you whether or not the database successfully processed your write. Unacknowledged writes do not return any response, so you do not know if the write succeeded or not.

4 Querying

4.1 find

db.collection.find()
- first argument: single condition for a single value (**also called point query**[4]). single condition for multi-value, (**also called multi-value query**[5])multiple conditions separated by comma, e.g. db.users.find({"username":"joe","age":27}). By default, {} means matching everything in the collection
- second argument: which key/value pair to be returned. 1: means to be returned; 0: not return. By default, "_id" key is returned.
e.g. db.users.find({},{"username":1,"_id":0})
- limitations: the value of a query document must be a constant (can be a normal variable). That is, it cannot refer to the value of another key in the document.

4.2 Query criteria

Query Conditionals "$lt","$lte","$gt","$gte", can be combined. "$ne"
e.g. db.users.find({"age" : {"$gte" : 18, "$lte" : 30}})

OR queries "$in","$nin": returns documents that don't match any of the criteria in the array. "$or" takes an **array** of possible criteria.

Conditional VS modifiers $-prefix: conditionals are an inner document key, and modifiers are always a key in the outer document. e.g.
- conditional $gt, $lt: db.users.find({"age" : {"$lt" : 30, "$gt" : 20}})
- modifier $and: db.users.find({"$and" : [{"x" : {"$lt" : 1}}, {"x" : 4}]})

4.3 Type-specific queries

null 1. matches itself: null; 2. matches "does not exist, will return all documents lacking that key.
If we only want to find keys whose value is null, we can check that the key is

[4]Point query: searches for a single value (although there may be multiple documents with that value).

[5]mutli-value query e.g. find({"age" : {"$gte" : 21, "$lte" : 30}})

null and exists using the "$exists" conditional:
db.c.find({"z" : {"$in" : [null], "$exists" : true}})

Regular Expressions e.g. db.users.find({"name" : /joe/i})
/j means: J or j. /i: is the regular expression flag, which is allowed but not required.
Regular expressions can also match themselves.

Querying Arrays Querying for elements of an array is designed to behave the way querying for **scalars** does.
e.g. we do: db.food.insert({"fruit" : ["apple", "banana", "peach"]})
It will successfully match the following query:
db.food.find({"fruit" : "banana"})

"$all" Match arrays by **more than one element. Order does not matter.**

Entire array as value Exact match! Even **the order is important.** e.g. the item {"_id" : 1, "fruit" : ["apple", "banana", "peach"]} will match:
db.food.find({"fruit" : ["apple", "banana", "peach"]})
not match: db.food.find({"fruit" : ["banana", "apple", "peach"]})

"$size" To query for arrays of a given size. e.g.
db.food.find({"fruit" : {"$size" : 3}})
"$size" cannot be combined with another $ conditional (in this example, "$gt"), but this query can be accomplished by adding a "size" key to the document.

"$slice" To return a subset of elements for an array key.
- To return the first 10 comments: using {"$slice" : 10}
- To return the last 10 comments, use −10
- To return the middle of the results, take an offset and the number of elements to return. e.g. {"$slice" : [23, 10]}, will return the 24th through 33th.

Array and range query interactions A range may match any multi-element array.
- "$elemMatch" won't match non-array elements.
- Using min() and max() when querying for ranges over documents that may include arrays is generally a good idea.

On embedded documents - A query for a full subdocument must exactly match the subdocument, is also**order-sensitive**
- Using dot-notation .
- To correctly group criteria without needing to specify every key, use "$elemMatch" to partially specify criteria to match a single embedded document in an array.

"$where" allow you to execute arbitrary JavaScript as part of your query. Should not be used unless strictly necessary: **they are much slower than regular queries.**

4.4 Cursors

The database returns results from find using a cursor. By a cursor, You can limit the number of results, skip over some number of results, sort results by any combination of keys in any direction, and perform a number of other powerful operations.
e.g. cursor.hasNext(), cursor.next();

Skips, limits, and sort - limit: db.c.find().limit(3). sets an upper limit
- skip: db.c.find().skip(3).
- sort: Sort direction can be 1 (ascending) or -1 (descending). If multiple keys are given, the results will be sorted in that order.

4.5 Comparison Order

To compare different types: from least to greatest value, the ordering is:
1. minimum value; 2.**null**; 3. numbers; 4. strings; 5. object/document; 6. array; 7. binary data; 8. object ID; 9. Boolean; 10. Date; 11. Timestamp; 12. Regular expression; 13. maximum value.

5 Indexing

compound indexes: hierarchy! **Order matters!** e.g. {"username":1, "age":-1} is different from {"age":-1, "username":1}

Compound index can do "double duty" and act like different indexes for different queries. f we have an index that looks like {"a": 1, "b": 1, "c": 1, ..., "z": 1}, we effectively have indexes on {"a": 1}, {"a": 1, "b" : 1}, {"a": 1, "b": 1, "c": 1}, and so on.

Attention this doesn't hold for any subset of keys. only queries that can use a prefix of the index can take advantage of it.

5.1 How $-operators use indexes

Inefficient operators - There are a few queries that cannot use an index at all, such as "$where" queries and "$exists", checking if a key exists.
- In general, negation is inefficient. "$ne" queries can use an index, but not very well. "$not" can sometimes use an index but often does not know how.

Effective queries Compound indexes can help MongoDB execute more effectively queries with multiple clauses.

MongoDB will use one of the indexes you created, not use both. The only exception to this rule is "$or". "$or" can use one index per $or clause, as $or preforms two queries and then merges the results.

Indexes often work well for	Table scans often work well for
Large collections	Small collections
Large documents	Small documents
Selective queries	Non-selective queries

Table 1: Properties that affect the effectiveness of indexes

Indexing Objects and Arrays - embedded docs: Indexes can be created on keys in embedded documents in the same way that they are created on normal keys. Indexing the entire subdocument will only help queries that are querying for the entire subdocument. - arrays: allows you to use the index to search for specific array elements efficiently. **Only one field** in an index entry can be from an array. This is to avoid the explosive number of index entries you'd get from multiple multikey indexes. e.g.:
we had an index on {"x" : 1, "y" : 1}
legal: db.multi.insert({"x" : [1, 2, 3], "y" : 1}) or db.multi.insert({"x" : 1, "y" : [4, 5, 6]})
illegal: db.multi.insert({"x" : [1, 2, 3], "y" : [4, 5, 6]})

5.2 Index Cardinality

Cardinality refers to how many distinct values there are for a field in a collection. In general, the greater the cardinality of a field, the more helpful an index on that field can be.

6 Topics

6.1 Schema

There are no predefined schemas: a document's keys and values are not of fixed types or sizes. Without a fixed schema, adding or removing fields as needed becomes easier.

6.2 Schema Validation

MongoDB provides the capability to perform schema validation during updates and insertions. More details in 3.1 Insert Validation.

6.3 Replications

With MongoDB, you set up replication by creating a replica set. A replica set is a group of servers with one primary, the server taking client requests, and multiple secondaries, servers that keep copies of the primary's data. If the primary crashes, the secondaries can elect a new primary from amongst themselves.

There are a few key concepts to remember:
- Clients can send a primary all the same operations they could send a standalone server (reads, writes, commands, index builds, etc.).
- Clients cannot write to secondaries.

- Clients, by default, cannot read from secondaries. By explicitly setting an "I know I'm reading from a secondary" setting, clients can read from secondaries.

7 Past Exam Questions

35. Which has the lowest latency?
A. A local MongoDB point query
B. Getting an object from Amazon S3 over the Atlantic
C. They both have the same order of magnitude of latency
D. Latency is irrelevant to retrieving data, it only matters for hosting websites.

98. Assume a very large MongoDB collection named "books" that has a tree index
{ "Title": 1, "Year": -1}

Which one of these queries cannot be executed efficiently using this index?
A. db.books.find({"Title": "Inferno" }) B. db.books.find({"Year" : 2016 }) C. db.books.find({"Title": "Inferno", "Year" : 2016, "Author": "Dan Brown" }) D. db.books.find({"Year" : 2016, "Title": "Inferno"})

9. In MongoDB, does each replica set contain all of the data in a collection?
A. Never.
B. Always.
C. It depends on the schema of the collection.
D. Not if there is more than one replica set.

What best describes a write concern in MongoDB?
A. It is the protocol for writing data to MongoDB, which is a blocking call until at least a certain number of replicas have acknowledged that the data was written.
B. It is the requirement that any query writing to a collection must follow the schema of this collection.
C. It is the situation in which MongoDB starts being emotional about too large an amount of queries sent in too little time.
D. It is the risk that data gets lost in case a server goes down.

2017: Which of the following statements about Document stores and MongoDB are correct? On the answer sheet, mark the correct answers with a checkmark (X).
a. In MongoDB, operations are atomic at the document level only.
b. Document stores expose only a key-value interface.
c. MongoDB does not support schema validation.
d. MongoDB stores documents in the XML format.
e. MongoDB performance degrades when the number of documents increases.
f. Document stores are column stores with flexible schema.

Which of the following statements about Document stores and MongoDB are correct?
a. Document stores support key-value-like queries
b. MongoDB supports schema validation
c. MongoDB can serialize documents in the JSON-like format
d. MongoDB can store different documents in the same collection regardless of their content

e. In MongoDB all documents have the _id field
f. Document stores and object stores are completely the same

(2 points) Consider the following requests to a newly created collection company: 1db.company.insert({ "name": ["David"], "job":"Engineer" })
2db.company.insert({ "room_number": 305, "type":"office" })
3db.company.index({"salary" : 1})
Which of the following statements are correct?
a. MongoDB will complain about the request 1 as "name" cannot be an array
b. MongoDB will complain about the request 2, since 305 is not quoted
c. MongoDB will complain about the request 2, because the intended document for insertion has different schema from the document inserted with the request
d. MongoDB will complain about the request 3, since there is no documents with the field "salary" in the collection yet

What does the following MongoDB query output?(Write what it would print in a console after execution.) Hint: To get all points, the answer must exactly match the output of MongoDB.
books.find({ "genre" : "fantasy" }, {"quantity" : 1}).sort({ "quantity" : 1 })