FullStack.Cafe - Kill Your Tech Interview

Q1: What are *Key Features* of MongoDB? ☆

Topics: MongoDB

Answer:

MongoDB is an open-source document database that provides high performance, high availability, and automatic scaling.

Its **Key Features** are:

- Document Oriented and NoSQL database.
- Supports Aggregation
- Uses BSON format
- Sharding (Helps in Horizontal Scalability)
- Supports Ad Hoc Queries
- Schema Less
- Capped Collection
- Indexing (Any field in MongoDB can be indexed)
- MongoDB Replica Set (Provides high availability)
- Supports Multiple Storage Engines
- ACID compliance including multi-document transactions support (starting from v. 4)

Q2: What is a *Replica Set*? ☆☆

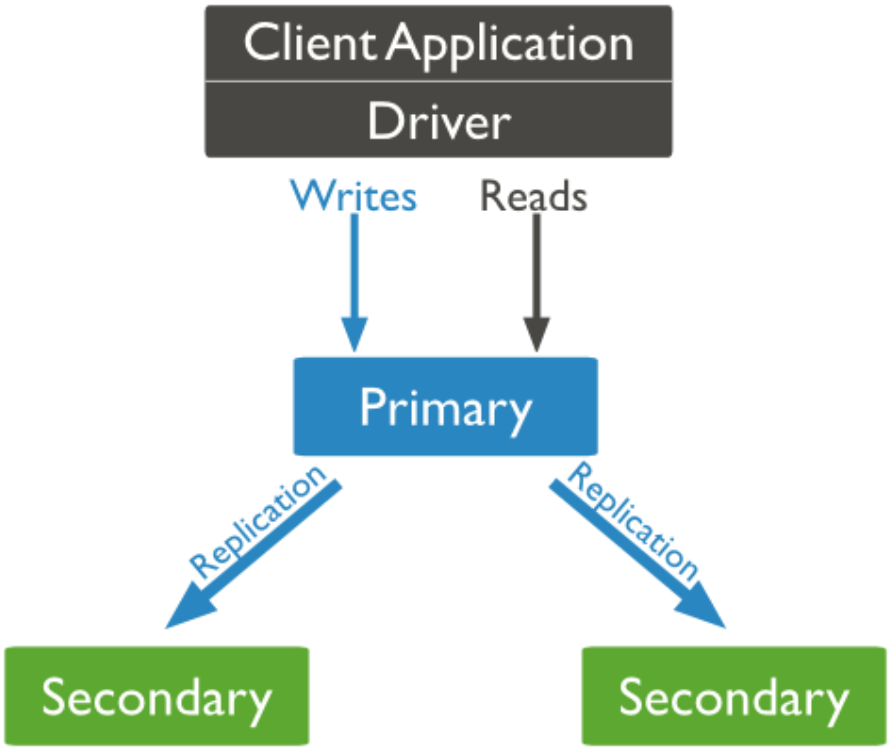
Topics: MongoDB

Answer:

A replica set is a group of `mongod` instances that maintain the same data set. A replica set contains several data-bearing nodes and optionally one arbiter node. Of the data-bearing nodes, one and only one member is deemed the **primary node**, while the other nodes are deemed **secondary nodes**.

The ideas for a replica set are :

- The primary node receives all write operations.
- The secondaries replicate the primary's oplog and apply the operations to their data sets such that the secondaries' data sets reflect the primary's data set asynchronously. If the primary is unavailable, an eligible secondary will hold an election to elect the new primary.



Q3: What Is *Replication* In MongoDB? ☆☆

Topics: MongoDB

Answer:

Replication is the process of synchronizing data across multiple servers.

Replication provides redundancy and increases data availability. With multiple copies of data on different database servers, replication provides a level of fault tolerance against the loss of a single database server.

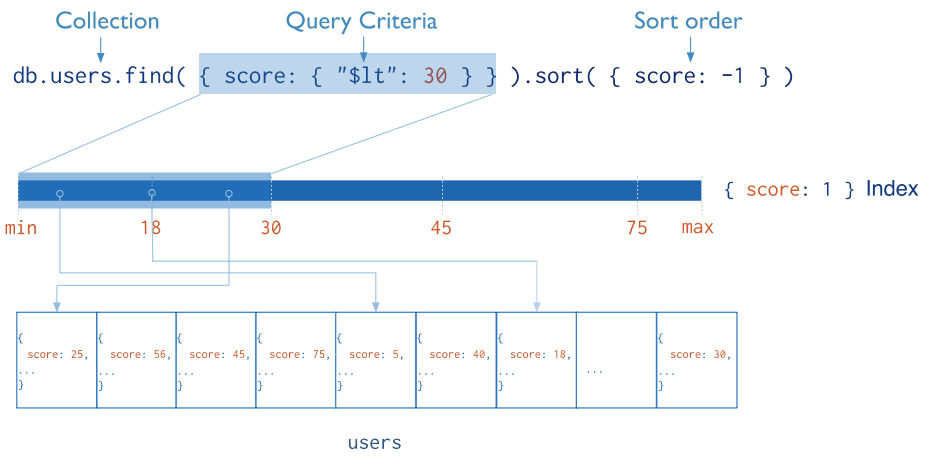
In some cases, replication can provide increased read capacity as clients can send read operations to different servers. Maintaining copies of data in different data centres can increase data locality and availability for distributed applications. You can also maintain additional copies for dedicated purposes, such as disaster recovery, reporting, or backup.

Q4: What are *Indexes* in MongoDB? ☆☆

Topics: MongoDB

Answer:

Indexes support the efficient execution of queries in MongoDB. Without indexes, MongoDB must perform a *collection scan*, i.e. scan every document in a collection, to select those documents that match the query statement. If an appropriate index exists for a query, MongoDB can use the index to limit the number of documents it must inspect.



Q5: What is *Sharding* in MongoDB? ☆☆

Topics: MongoDB

Answer:

Sharding is a method for distributing data across multiple machines. MongoDB uses sharding to support deployments with very large data sets and high throughput operations. MongoDB supports *horizontal scaling* through sharding. MongoDB shards data at the *collection* level, distributing the collection data across the *shards* in the *cluster*.

Q6: When should we *embed one document within another* in MongoDB? ☆☆

Topics: MongoDB

Answer:

You should consider embedded documents (subdocuments) for:

- When the relationship is `one-to-few` (not many, not unlimited). For unlimited use case, you should start considering separating subdocuments into another collection.
- When retrieval is likely to happen together, that will improve performance
- When updates are likely to happen at the same time. Although starting from MongoDB 4.0, you can use multi-documents transactions, a single document transaction would be more performant
- When the field is rarely updated

Q7: What is *BSON* in MongoDB? ☆☆

Topics: MongoDB

Answer:

BSON is a binary serialization format used to store documents and make remote procedure calls in MongoDB. BSON extends the JSON model to provide additional data types, and ordered fields, and to be efficient for encoding and decoding within different languages.

BSON encodes type and length information, too, making it easier for machines to parse. Consider:

```
{"hello": "world"} →

\x16\x00\x00\x00           // total document size
\x02                       // 0x02 = type String
hello\x00                  // field name
\x06\x00\x00\x00world\x00  // field value
\x00                       // 0x00 = type EOO ('end of object')
```

Q8: What's the difference between `replaceOne()` and `updateOne()` in MongoDB? ☆☆

Topics: MongoDB

Answer:

- With `replaceOne()` you can only *replace* the entire document
- While `updateOne()` allows for *updating* fields.
- Since `replaceOne()` replaces the entire document - *fields in the old document not contained in the new will be lost*.
- With `updateOne()` *new fields can be added without losing the fields* in the old document.

For example:

```
{
  "_id" : ObjectId("0123456789abcdef01234567"),
  "my_test_key3" : 3333
}

replaceOne({"_id" : ObjectId("0123456789abcdef01234567")}, { "my_test_key4" : 4})

// Output
{
   "_id" : ObjectId("0123456789abcdef01234567"),
   "my_test_key4" : 4.0
}

updateOne({"_id" : ObjectId("0123456789abcdef01234567")}, {$set: { "my_test_key4" : 4}})

// Output
{
   "_id" : ObjectId("0123456789abcdef01234567"),
   "my_test_key3" : 3333.0,
   "my_test_key4" : 4.0
}
```

Q9: Explain the structure of `ObjectID` in MongoDB ☆☆☆

Topics: MongoDB

Answer:

ObjectIds are small, likely unique, fast to generate, and ordered. ObjectId values consist of 12 bytes, where the first four bytes are a timestamp that reflect the ObjectId's creation. Specifically:

- a 4-byte value representing the seconds since the Unix epoch,
- a 5-byte random value, and

- a 3-byte counter, starting with a random value. In MongoDB, each document stored in a collection requires a unique _id field that acts as a primary key. If an inserted document omits the _id field, the MongoDB driver automatically generates an ObjectId for the _id field.

Q10: Does Mongodb support *Foreign Key* constraints? ☆☆☆

Topics: MongoDB

Answer:

No.

One of the great things about relational database is that it is really good at keeping the data consistent within the database. One of the ways it does that is by using foreign keys. A foreign key constraint is that let's say there's a table with some column which will have a foreign key column with values from another table's column.

In `MongoDB`, there's no guarantee that foreign keys will be preserved. It's upto the programmer to make sure that the data is consistent in that manner. Constraints can not be enforced by MongoDB either. It can't even enforce a specific type for a field, due to the schemaless nature of MongoDB.

Q11: Does MongoDB support *ACID transaction management* and *Locking* functionalities? ☆☆☆

Topics: MongoDB

Answer:

Yes.

ACID stands that any update is:

- **Atomic**: it either fully completes or it does not
- **Consistent**: no reader will see a "partially applied" update
- **Isolated**: no reader will see a "dirty" read
- **Durable**: (with the appropriate write concern)

MongoDB is ACID-compilant at the document level. MongoDB added support for multi-document ACID transactions in version 4.0 in 2018 and extended that support for distributed multi-document ACID transactions in version 4.2 in 2019.

MongoDB's document model allows related data to be stored together in a single document. The document model, combined with atomic document updates, obviates the need for transactions in a majority of use cases. Nonetheless, there are cases where true multi-document, multi-collection MongoDB transactions are the best choice.

MongoDB transactions work similarly to transactions in other databases. To use a transaction, start a MongoDB session through a driver. Then, use that session to execute your group of database operations. You can run any of the CRUD (create, read, update, and delete) operations across multiple documents, multiple collections, and multiple shards.

```
try (ClientSession clientSession = client.startSession()) {
                  clientSession.startTransaction();
                  collection.insertOne(clientSession, docOne);
                  collection.insertOne(clientSession, docTwo);
                  clientSession.commitTransaction();
        }
```

Q12: What does MongoDB *not being ACID compliant* really mean?

☆☆☆

Topics: MongoDB

Answer:

It's actually not correct that MongoDB is not ACID-compliant. On the contrary, MongoDB is ACID-compilant at the document level.

Any update to a single document is

- **Atomic**: it either fully completes or it does not
- **Consistent**: no reader will see a "partially applied" update
- **Isolated**: again, no reader will see a "dirty" read
- **Durable**: (with the appropriate write concern)

What MongoDB doesn't have is *transactions* - that is, multiple-document updates that can be rolled back and are ACID-compliant.

Multi-document transactions, scheduled for MongoDB 4.0 in Summer 2018*, will feel just like the transactions developers are familiar with from relational databases – multi-statement, similar syntax, and easy to add to any application.

Q13: Should I *normalize* my data before storing it in MongoDB?

☆☆☆

Topics: MongoDB

Answer:

It depends from your goals. Normalization will provide an *update efficient data representation*. Denormalization will make data *reading efficient*.

In general, use embedded data models (denormalization) when:

- you have “contains” relationships between entities.
- you have one-to-many relationships between entities. In these relationships the “many” or child documents always appear with or are viewed in the context of the “one” or parent documents.

In general, use normalized data models:

- when embedding would result in duplication of data but would not provide sufficient read performance advantages to outweigh the implications of the duplication.
- to represent more complex many-to-many relationships.
- to model large hierarchical data sets.

Also normalizing your data like you would with a relational database is usually not a good idea in MongoDB. Normalization in relational databases is only feasible under the premise that JOINs between tables are relatively cheap. The $lookup aggregation operator provides some limited JOIN functionality, but it doesn't work with sharded collections. So joins often need to be emulated by the application through multiple subsequent database queries, which is very slow (see question MongoDB and JOINs for more information).

Q14: How is data stored in MongoDB? ☆☆☆

Topics: MongoDB

Answer:

Data in MongoDB is stored in BSON documents – JSON-style data structures. Documents contain one or more fields, and each field contains a value of a specific data type, including arrays, binary data and sub-documents. Documents that tend to share a similar structure are organized as collections. It may be helpful to think of documents as analogous to rows in a relational database, fields as similar to columns, and collections as similar to tables.

The advantages of using documents are:

- Documents (i.e. objects) correspond to native data types in many programming languages.
- Embedded documents and arrays reduce need for expensive joins.
- Dynamic schema supports fluent polymorphism.

Q15: How can you achieve *Primary Key - Foreign Key* relationships in MongoDB? ☆☆☆

Topics: MongoDB

Answer:

By default MongoDB does not support such primary key - foreign key relationships. However, we can achieve this concept by embedding one document inside another (aka subdocuments).

Embedded data models allow applications to store related pieces of information in the same database record. As a result, applications may need to issue fewer queries and updates to complete common operations.

In general, use embedded data models when:

- you have "contains" relationships between entities. See Model One-to-One Relationships with Embedded Documents.
- you have one-to-many relationships between entities. In these relationships the "many" or child documents always appear with or are viewed in the context of the "one" or parent documents. See Model One-to-Many Relationships with Embedded Documents.

Q16: Can you create an *index on an array* field in MongoDB? If yes, what happens in this case? ☆☆☆

Topics: MongoDB

Answer:

Yes. An array field can be indexed in MongoDB. In this case, MongoDB would index each value of the array so you can query for individual items:

```
> db.col1.save({'colors': ['red','blue']})
> db.col1.ensureIndex({'colors':1})

> db.col1.find({'colors': 'red'})
{ "_id" : ObjectId("4ccc78f97cf9bdc2a2e54ee9"), "colors" : [ "red", "blue" ] }
> db.col1.find({'colors': 'blue'})
{ "_id" : ObjectId("4ccc78f97cf9bdc2a2e54ee9"), "colors" : [ "red", "blue" ] }
```

Q17: What is a *Covered Query* in MongoDB? ☆☆☆

Topics: MongoDB

Answer:

A **covered query** is a query that can be satisfied entirely using an index and does not have to examine any documents. An index covers a query when all of the following apply:

- all the fields in the query are part of an index, **and**
- all the fields returned in the results are in the same index.
- no fields in the query are equal to `null` (i.e. {`"field" : null`} or {`"field" : {$eq : null}}`).

For example, a collection `inventory` has the following index on the `type` and `item` fields:

```
db.inventory.createIndex( { type: 1, item: 1 } )
```

This index will cover the following operation which queries on the `type` and `item` fields and returns only the `item` field:

```
db.inventory.find(   { type: "food", item:/^c/ },   { item: 1, _id: 0 })
```

Q18: How can you achieve *Transaction* in MongoDB? ☆☆☆

Topics: MongoDB

Answer:

In MongoDB, an operation on a single document is **atomic**.

Because you can use *embedded documents* and *arrays* to capture *relationships* between data in a single document structure instead of normalizing across multiple documents and collections, this single-document atomicity obviates the need for multi-document transactions for many practical use cases.

For situations that require atomicity of reads and writes to multiple documents (in single or multiple collections), MongoDB supports **multi-document (distributed) transactions**.

Q19: What is `oplog` and how is it relevant to *replication* process?

☆☆☆

Topics: MongoDB

Answer:

- The **oplog** (operations log) is a special capped collection that keeps a rolling record of all operations that modify the data stored in your databases.
- MongoDB applies database operations on the primary and then records the operations on the primary's oplog. The secondary members then copy and apply these operations in an asynchronous process.
- Each operation in the oplog is **idempotent**. That is, oplog operations produce the same results whether applied once or multiple times to the target dataset.

Q20: Is there an `upsert` option in the MongoDB insert command?

☆☆☆

Topics: MongoDB

Answer:

Since upsert is defined as operation that *creates a new document when no document matches the query criteria* there is no place for upserts in insert command. It is an option for the update command.

```
db.collection.update(query, update, {upsert: true})
```

FullStack.Cafe - Kill Your Tech Interview

Q1: What is an *Aggregation Pipeline* in MongoDB? ☆☆☆

Topics: MongoDB

Answer:

Aggregation operations process multiple documents and return computed results. You can use aggregation operations to:

- **Group** values from multiple documents together.
- **Perform operations** on the grouped data to return a single result.
- **Analyze** data changes over time.

MongoDB provides aggregation operations through aggregation pipelines — a series of operations that process data documents sequentially. An aggregation pipeline consists of one or more stages that process documents:

- Each stage performs an operation on the input documents. For example, a stage can **filter** documents, **group** documents, and **calculate** values.
- The documents that are output from a stage are passed to the next stage.
- An aggregation pipeline can return results for groups of documents. For example, return the total, average, maximum, and minimum values.

Consider:

```
db.orders.aggregate([
   // Stage 1: Filter pizza order documents by pizza size
   {
      $match: { size: "medium" }
   },
   // Stage 2: Group remaining documents by pizza name and calculate total quantity
   {
      $group: { _id: "$name", totalQuantity: { $sum: "$quantity" } }
   }
])
```

Q2: When to use MongoDB vs other document oriented database systems? ☆☆☆

Topics: MongoDB

Answer:

MongoDB is best suitable to store *unstructured data*. And this can organize your data into document format. These data stores don't enforce the ACID properties, and any schemas. This doesn't provide any transaction abilities. So this can scale big and we can achieve faster access (both read and write). If you wanted to work with structured data you can go ahead with RDBM.

Q3: What is use of *Capped Collection* in MongoDB? ☆☆☆

Topics: MongoDB

Answer:

Capped collections allow you to define a fix length/size collection. After the size/no of documents have been reached it will override the oldest document to accommodate new documents. It is like a circular buffer. Capped collections support high-throughput operations that insert and retrieve documents based on insertion order.

Consider the following potential use cases for capped collections:

- Store log information generated by high-volume systems. Inserting documents in a capped collection without an index is close to the speed of writing log information directly to a file system.
- Cache small amounts of data in a capped collections.

Q4: What is *BSON* and exactly how is it different from JSON? ☆☆☆

Topics: MongoDB

Answer:

BSON is the binary encoding of JSON-like documents that MongoDB uses when storing documents in collections. It adds support for data types like Date and binary that aren't supported in JSON.

In practice, you don't have to know much about BSON when working with MongoDB, you just need to use the native types of your language and the supplied types (e.g. ObjectId) of its driver when constructing documents and they will be mapped into the appropriate BSON type by the driver.

Q5: Explain advantages of *BSON* over *JSON* in MongoDB? ☆☆☆

Topics: MongoDB JSON

Answer:

- **BSON** is designed to be efficient in space, but in some cases is not much more efficient than JSON. In some cases BSON uses even more space than JSON. The reason for this is another of the BSON design goals: traversability. BSON adds some "extra" information to documents, like length of strings and subobjects. This makes traversal faster.
- BSON is also designed to be fast to encode and decode. For example, integers are stored as 32 (or 64) bit integers, so they don't need to be parsed to and from text. This uses more space than JSON for small integers, but is much faster to parse.
- In addition to compactness, BSON adds additional data types unavailable in JSON, notably the BinData and Date data types.

Q6: When to use CouchDB over MongoDB and vice versa? ☆☆☆

Topics: MongoDB

Answer:

- If you plan to have a mobile component, or need desktop users to work offline and then sync their work to a server you need CouchDB.
- If your code will run only on the server then go with MongoDB

That's it. Unless you need CouchDB's (awesome) ability to replicate to mobile and desktop devices, MongoDB has the performance, community and tooling advantage at present.

Some more factors to consider:

1. Do you need **master-master**? Then CouchDB. Mainly CouchDB supports master-master replication which anticipates nodes being disconnected for long periods of time. MongoDB would not do well in that environment.
2. Do you need **MAXIMUM R/W throughput**? Then MongoDB
3. Do you need ultimate **single-server durability** because you are only going to have a single DB server? Then CouchDB.
4. Are you storing a **MASSIVE data** set that needs sharding while maintaining insane throughput? Then MongoDB.
5. Do you need strong **consistency** of data? Then MongoDB.
6. Do you need high **availability** of database? Then CouchDB.
7. Are you hoping **multi databases** and multi tables/ collections? Then MongoDB
8. You have a **mobile app offline users** and want to sync their activity data to a server? Then you need CouchDB.
9. Do you need large variety of **querying** engine? Then MongoDB
10. Do you need large **community** to be using DB? Then MongoDB

Q7: What is a *Cluster* in MongoDB? ☆☆☆

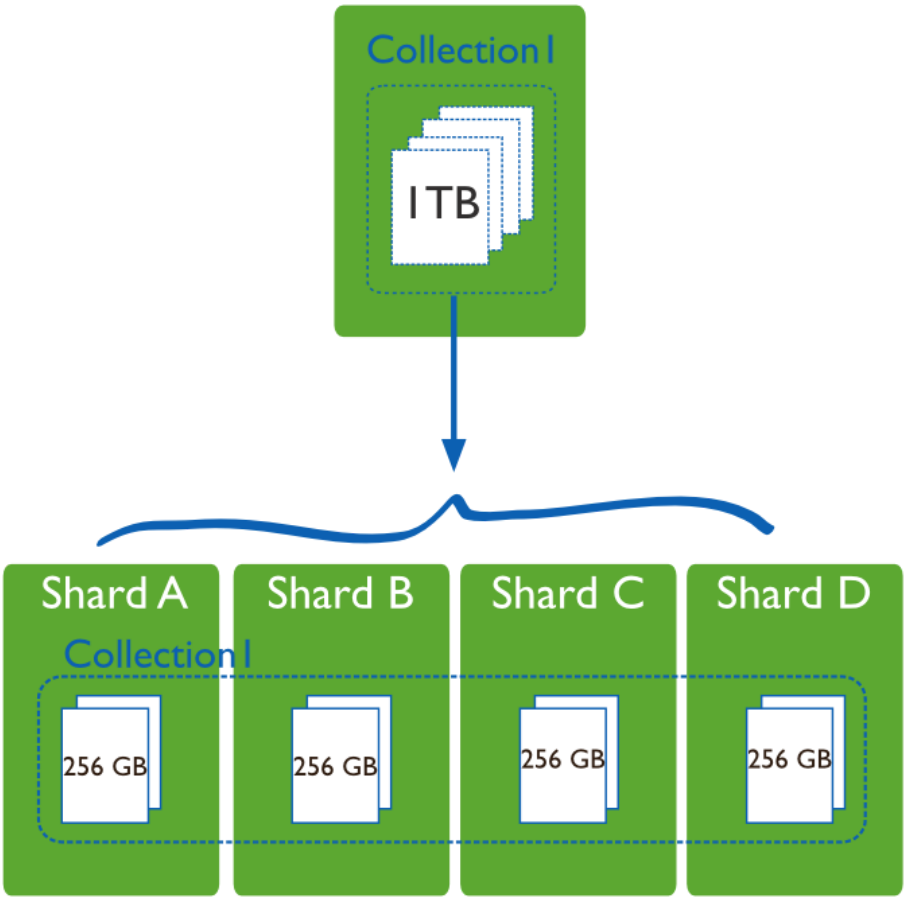
Topics: MongoDB

Answer:

A mongodb **cluster** is the word usually used for **sharded cluster** in mongodb. The main purposes of a sharded mongodb are:

- Scale reads and writes along several nodes
- Each node does not handle the whole data so you can separate data along all the nodes of the shard. Each node is a member of a shard and the data are separated on all shards.

A sharded cluster consists of config servers, shards, and one or more mongos routing processes.



Q8: When shall we use Azure Table Storage over Azure SQL? ☆☆☆

Topics: Azure MongoDB

Answer:

- **SQL Azure** is great when you want to work with structured data using relations, indexes, constraints, etc.
- **Azure Storage Table** is great when you need to work with centralized structured data without relations and usually with large volumes.

I would always use azure tables as a much cheaper solution if:

- I perform table selects ONLY by PK (select on the property is slow due to the of the whole deserialization),
- I can live with a limited Linq set (Query Operators (Table Service Support),
- I don't need to *join* tables and perform complex queries *on the server*,
- I need horizontal partitioning "sharding" of my data (SQL Azure Federations is a step in that direction by Tables have PartionKey from day 0),
- Azure Tables are only cheaper than SQL Azure if the data access pattern is relatively light since tables have a per-transaction fee and SQL Azure doesn't.

Q9: Why would you use *Projection Document*? ☆☆☆

Topics: MongoDB

Answer:

- The *projection document* limits the fields to return for all matching documents.

- The *projection document* can specify the **inclusion** of fields or the **exclusion** of fields and has the following form:

```
{ field1: <value>, field2: <value> ... }
```

- `<value>` may be `0` (or `false`) to exclude the field, or `1` (or `true`) to include it.
- With the exception of the `_id` field, you may not have both *inclusions and exclusions* in the same projection document.

```
var collection = db.collection( 'restaurants' );
// Find some documents
var result = collection
     .find({ 'cuisine' : 'Brazilian' }, { 'name' : 1, 'cuisine' : 1 })
// returned documents
// [{ name: 'some...', cuisine: 'some...'}]
```

Q10: Does the *sort order* matter for *Compound Index* in MongoDB? ☆☆☆

Topics: MongoDB

Answer:

Yes. Since *indexes* contain *ordered* records, MongoDB can obtain the results of a sort from an index that includes the sort fields. If MongoDB cannot use an index or indexes to obtain the sort order, MongoDB must perform a blocking sort operation on the data.

You can specify a sort on all the keys of the index or on a subset; however, the sort keys must be listed in the *same order* as they appear in the index. For example, an index key pattern `{ a: 1, b: 1 }` can support a sort on `{ a: 1, b: 1 }` but *not* on `{ b: 1, a: 1 }`.

For a query to **use a compound index for a sort**, the specified sort direction for all keys in the `cursor.sort()` document must match the index key pattern *or* match the inverse of the index key pattern. For example, an index key pattern `{ a: 1, b: -1 }` can support a sort on `{ a: 1, b: -1 }` and `{ a: -1, b: 1 }` but **not** on `{ a: -1, b: -1 }` or `{a: 1, b: 1}`.

Q11: How do the MongoDB *Journal File* and *Oplog Differ*? ☆☆☆

Topics: MongoDB

Answer:

- **Oplog** stores high-level transactions that modify the database (queries are not stored for example), like insert this document, update that, etc. Oplog is kept on the master and slaves will periodically poll the master to get newly performed operations (since the last poll). Operations sometimes get transformed before being stored in the oplog so that they are idempotent (and can be safely applied many times).

- **Journal** on the other hand can be switched on/off on any node (master or slave), and is a low-level log of an operation for the purpose of crash recovery and durability of a single mongo instance. You can read *low-level op* like 'write these bytes to this file at this position'.

NOTE: Starting in MongoDB 4.0, you cannot turn journaling off for replica set members that use the WiredTiger storage engine.

Q12: How does MongoDB handle *caching*? ☆☆☆

Topics: MongoDB

Answer:

- MongoDB keeps most recently used data in RAM. If you have created indexes for your queries and your working data set fits in RAM, MongoDB serves all queries from memory.
- MongoDB does not cache the query results in order to return the cached results for identical queries.

Q13: What is *TTL Collection* in MongoDB? ☆☆☆

Topics: MongoDB

Answer:

TTL collections make it possible to store data in MongoDB and have the `mongod` automatically remove data after a specified number of seconds or at a specific clock time.

For example, the following operation creates an index on the `log_events` collection's `createdAt` field and specifies the `expireAfterSeconds` value of `10` to set the expiration time to be ten seconds after the time specified by `createdAt`.

```
db.log_events.createIndex( { "createdAt": 1 }, { expireAfterSeconds: 10 } )
```

When adding documents to the `log_events` collection, set the `createdAt` field to the current time:

```
db.log_events.insertOne( {
   "createdAt": new Date(),
   "logEvent": 2,
   "logMessage": "Success!"
} )
```

MongoDB will automatically delete documents from the `log_events` collection when the document's `createdAt` value [1] is older than the number of seconds specified in `expireAfterSeconds`.

To expire documents at a specific clock time, begin by creating a TTL index on a field that holds values of BSON date type or an array of BSON date-typed objects *and* specify an `expireAfterSeconds` value of `0`.

```
db.log_events.createIndex( { "expireAt": 1 }, { expireAfterSeconds: 0 } )
```

For each document, set the value of `expireAt` to correspond to the time the document should expire. For example, the following `insertOne()` operation adds a document that expires at `July 22, 2013 14:00:00`.

```
db.log_events.insertOne( {
   "expireAt": new Date('July 22, 2013 14:00:00'),
   "logEvent": 2,
   "logMessage": "Success!"
} )
```

MongoDB will automatically delete documents from the `log_events` collection when the documents' `expireAt` value is older than the number of seconds specified in `expireAfterSeconds`, i.e. `0` seconds older in this case. As

such, the data expires at the specified `expireAt` value.

Q14: In MongoDB what is the difference between *Sharding* and *Replication*? ☆☆☆

Topics: MongoDB

Answer:

In the context of scaling MongoDB:

- *replication* creates additional copies of the data and allows for automatic failover to another node. Replication may help with horizontal scaling of reads if you are OK to read data that potentially isn't the latest.
- *sharding* allows for horizontal scaling of data writes by partitioning data across multiple servers using a *shard key*. It's important to choose a good shard key. For example, a poor choice of shard key could lead to "hot spots" of data only being written on a single shard.

A sharded environment does add more complexity because MongoDB now has to manage distributing data and requests between shards -- additional configuration and routing processes are added to manage those aspects.

Replication and sharding are typically combined to create a *sharded cluster* where each shard is supported by a replica set.

To say simply, imagine you have a great music collection, so:

- **Sharding** `>>` Keeping your music files in *different folders*
- **Replication** `>>` *Syncing* your collection to other drives

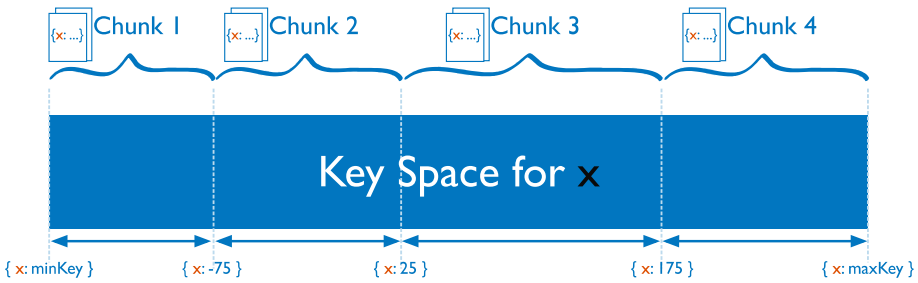
Q15: What is *Shard Key* in MongoDB and how does it affect development process? ☆☆☆

Topics: MongoDB

Answer:

The **shard key** is either a single indexed field or multiple fields covered by a compound index that determines the distribution of the collection's documents among the cluster's shards.

MongoDB divides the span of shard key values (or hashed shard key values) into non-overlapping ranges of shard key values (or hashed shard key values). Each range is associated with a chunk, and MongoDB attempts to distribute chunks evenly among the shards in the cluster.



The shard key has a direct relationship to the effectiveness of chunk distribution.

What do we need to really know as a developer?

- `insert` must include a shard key, so if it's a multi-parted shard key, we must include the entire shard key
- we've to understand what the shard key is on the collection itself
- for an `update`, `remove`, `find` - if `mongos` is not given a shard key - then it's going to have to broadcast the request to all the different shards that cover the collection.
- for an `update` - if we don't specify the entire shard key, we have to make it a multi update so that it knows that it needs to broadcast it

Q16: What is the difference between `One-to-Many` vs `One-to-Few` in MongoDB Schema Design? ☆☆☆

Topics: MongoDB

Answer:

An example of `one-to-few` might be the addresses for a person. This is a good use case for embedding – you'd put the addresses in an array inside of your Person object:

```
> db.person.findOne()
{
  name: 'Kate Monster',
  ssn: '123-456-7890',
  addresses : [
     { street: '123 Sesame St', city: 'Anytown', cc: 'USA' },
     { street: '123 Avenue Q', city: 'New York', cc: 'USA' }
  ]
}
```

An example of `one-to-many` might be parts for a product in a replacement parts ordering system. his is a good use case for referencing – you'd put the ObjectIDs of the parts in an array in product document.

Each Product would have its own document, which would contain an array of ObjectID references to the Parts that make up that Product:

```
> db.products.findOne()
{
    name : 'left-handed smoke shifter',
    manufacturer : 'Acme Corp',
    catalog_number: 1234,
    parts : [     // array of references to Part documents
        ObjectID('AAAA'),    // reference to the #4 grommet above
        ObjectID('F17C'),    // reference to a different Part
        ObjectID('D2AA'),
        // etc
    ]
```

Q17: What types of `One-to-N` schema designs you can use in MongoDB? ☆☆☆

Topics: MongoDB

Answer:

There are three basic `One-to-N` schema designs:

- *Embed* the `N` side if the cardinality is **one-to-few** and there is no need to access the embedded object outside the context of the parent object. For example, a list of person's addresses.
- Use an *array of references* to the `N`-side objects if the cardinality is **one-to-many** or if the `N`-side objects should stand alone for any reason. For example, parts for a product.
- Use a *reference* to the `One`-side in the `N`-side objects (parent-referencing) if the cardinality is **one-to-squillions**. For example, logs from N machines.

Q18: How *replication* works in MongoDB? ☆☆☆☆

Topics: MongoDB

Answer:

A replica set consists of a primary node and a secondary node too. With the help of a replica set, all the data from primary node to the secondary node replicates. Replication is a process of synchronizing the data. Replication provides redundancy and it also increases the availability of data with the help of multiple copies of data on the different database server. It also protects the database from the loss of a single server.

Q19: How does MongoDB ensure *high availability*? ☆☆☆☆

Topics: MongoDB

Answer:

MongoDB automatically maintains replica sets, multiple copies of data that are distributed across servers, racks and data centers. Replica sets help prevent database downtime using native replication and automatic failover.

A replica set consists of multiple replica set members. At any given time one member acts as the primary member, and the other members act as secondary members. If the primary member fails for any reason (e.g., hardware failure), one of the secondary members is automatically elected to primary and begins to process all reads and writes.

Q20: Is MongoDB *schema-less*? ☆☆☆☆

Topics: MongoDB

Answer:

No. In MongoDB schema design is still important. MongoDB's document model does, however, employ a different schema paradigm than traditional relational databases.

In MongoDB, documents are self-describing; there is no central catalog where schemas are declared and maintained. The schema can vary across documents, and the schema can evolve quickly without requiring the modification of existing data.

MongoDB's dynamic schema also makes it easier to represent semi-structured and polymorphic data, as documents do not all need to have exactly the same fields. For example, a collection of financial trading positions might have positions that are equity positions, and some that are bonds, and some that are cash. All may have some fields in common, but some fields ('ticker', “number_of_shares”) do not apply to all position types.

FullStack.Cafe - Kill Your Tech Interview

Q1: Find objects between *two dates* in MongoDB ☆☆☆

Topics: MongoDB

Answer:

```
db.CollectionName.find({"whenCreated": {
    '$gte': ISODate("2018-03-06T13:10:40.294Z"),
    '$lt': ISODate("2018-05-06T13:10:40.294Z")
}});
```

Q2: Why is a *Covered Query* important? ☆☆☆☆

Topics: MongoDB

Answer:

A covered query is a query that can be satisfied entirely using an index and does not have to examine any documents. Because the index contains all fields required by the query, MongoDB can both match the query conditions and return the results using only the index.

Querying only the index can be much **faster** than querying documents outside of the index. Index keys are typically smaller than the documents they catalog, and indexes are typically available in RAM or located sequentially on disk.

Q3: What are *Primary* and *Secondary* Replica sets? ☆☆☆☆

Topics: MongoDB

Answer:

- **Primary** and master nodes are the nodes that can accept writes. MongoDB's replication is `single-master`: only one node can accept write operations at a time.
- **Secondary** and slave nodes are read-only nodes that replicate from the primary asynchronously. If the primary is unavailable, an eligible secondary will hold an election to elect the new primary.

Q4: How does MongoDB provide *Concurrency*? ☆☆☆☆

Topics: MongoDB

Answer:

MongoDB uses multi-granularity locking that allows operations to lock at the global, database or collection level, and allows for individual storage engines to implement their own concurrency control below the collection level (e.g., at the document-level in WiredTiger).

When the storage engine detects conflicts between two operations, one will incur a write conflict causing MongoDB to transparently *retry* that operation.

When using the WiredTiger storage engine, MongoDB does not lock individual documents for either reads or writes. Instead, WiredTiger uses Multi-Version Concurrency Control, MVCC. When performing an update of a document, that update will succeed as long as the document still has the same version as it had when acquiring the snapshot. If not, WiredTiger will return an error (WT_ROLLBACK) indicating that the update had write conflicts. In this case, the update will abort and all pending changes are undone. MongoDB will then transparently retry the operation.

Q5: When to use Redis or MongoDB? ☆☆☆☆

Topics: MongoDB Redis

Answer:

- **Use MongoDB if you don't know yet how you're going to query your data or what schema to stick with.** MongoDB is suited for Hackathons, startups or every time you don't know how you'll query the data you inserted. MongoDB does not make any assumptions on your underlying schema. While MongoDB is schemaless and non-relational, this does not mean that there is no schema at all. It simply means that your schema needs to be defined in your app (e.g. using Mongoose). Besides that, MongoDB is great for prototyping or trying things out. Its performance is not that great and can't be compared to Redis.

- **Use Redis in order to speed up your existing application.** It is very uncommon to use Redis as a standalone database system (some people prefer referring to it as a "key-value"-store).

Q6: Explain what is *horizontal scalability*? ☆☆☆☆

Topics: MongoDB Scalability

Answer:

Horizontal scalability is the ability to increase capacity by connecting multiple hardware or software entities so that they work as a single logical unit. MongoDB provides horizontal scalability as part of its core functionality.

Q7: How would you choose between *Azure Table Storage* vs *MongoDB*? ☆☆☆☆

Topics: Azure MongoDB

Answer:

Azure Table Storage is a core Windows Azure storage feature, designed to be scalable (100TB 200TB 500TB per account), durable (triple-replicated in the data center, optionally geo-replicated to another data center), and schemaless (each row may contain any properties you want). A row is located by partition key + row key, providing a very fast lookup. All Table Storage access is via a well-defined REST API usable through any language (with SDKs, built on top of the REST APIs, already in place for .NET, PHP, Java, Python & Ruby).

MongoDB is a document-oriented database. To run it in Azure, you need to install MongoDB onto a web/worker roles or Virtual Machine, point it to a cloud drive (thereby providing a drive letter) or attached disk (for Windows/Linux Virtual Machines), and optionally turn on journaling (which I'd recommend), and optionally define an external endpoint for your use (or access it via virtual network). The Cloud Drive / attached disk, by the way, is actually stored in an Azure Blob, giving you the same durability and geo-replication as Azure Tables.

- if your queries are really simple (key->value) go with ATS.

- If you need more elaborate queries and don't want to go to SQL Azure, Mongo is likely your best bet.

Q8: What is *Selectivity* of the query in MongoDB? ☆☆☆☆

Topics: MongoDB NoSQL

Answer:

Selectivity is the ability of a query to **narrow results using the index**. Effective indexes are more selective and allow MongoDB to use the index for a larger portion of the work associated with fulfilling the query.

Suppose you have a field called `status` where the possible values are `new` and `processed`. If you add an index on `status` you've created a **low-selectivity** index. The index will be of little help in locating records.

A better strategy, depending on your queries, would be to create a **compound index** that includes the low-selectivity field and another field. For example, you could create a compound index on `status` and `created_at`.

Q9: How Compound Index *Prefix* affects *Sort* operation in MongoDB? ☆☆☆☆

Topics: MongoDB

Answer:

If the sort keys correspond to the index keys or an index *prefix*, MongoDB can use the index to sort the query results. A *prefix* of a compound index is a subset that consists of one or more keys at the start of the index key pattern.

For example, create a compound index on the `data` collection:

```
db.data.createIndex( { a:1, b: 1, c: 1, d: 1 } )
```

Then, the following are prefixes for that index:

```
{ a: 1 }{ a: 1, b: 1 }{ a: 1, b: 1, c: 1 }
```

The following query and sort operations use the index prefixes to sort the results. These operations do not need to sort the result set in memory.

Consider the following example in which the prefix keys of the index appear in both the query predicate and the sort:

```
db.data.find( { a: { $gt: 4 } } ).sort( { a: 1, b: 1 } )
```

In such cases, MongoDB can use the index to retrieve the documents in the order specified by the sort. As the example shows, the index prefix in the query predicate can be different from the prefix in the sort.

Q10: How does *Sharding* affect *Concurrency* in MongoDB? ☆☆☆☆

Topics: MongoDB

Answer:

- Sharding **improves concurrency** by distributing collections over multiple *mongod* instances, allowing shard servers (i.e. mongos processes) to perform any number of operations concurrently to the various downstream mongod instances.
- In a sharded cluster, locks apply to each individual shard, not to the whole cluster; i.e. each mongod instance is independent of the others in the sharded cluster and uses its own locks.
- The operations on one mongod instance do not block the operations on any others.

Q11: What does it mean to fit *working set* into RAM for MongoDB? ☆☆☆☆

Topics: MongoDB

Answer:

Working set is basically the amount of data AND indexes that will be active/in use by your system.

So for example, suppose you have 1 year's worth of data. For simplicity, each month relates to 1GB of data giving 12GB in total, and to cover each month's worth of data you have 1GB worth of indexes again totalling 12GB for the year.

If you are always accessing the last 12 month's worth of data, then your working set is: 12GB (data) + 12GB (indexes) = 24GB.

However, if you actually only access the last 3 month's worth of data, then your working set is: 3GB (data) + 3GB (indexes) = 6GB. In this scenario, if you had 8GB RAM and then you started regularly accessing the past 6 month's worth of data, then your working set would start to exceed past your available RAM and have a performance impact.

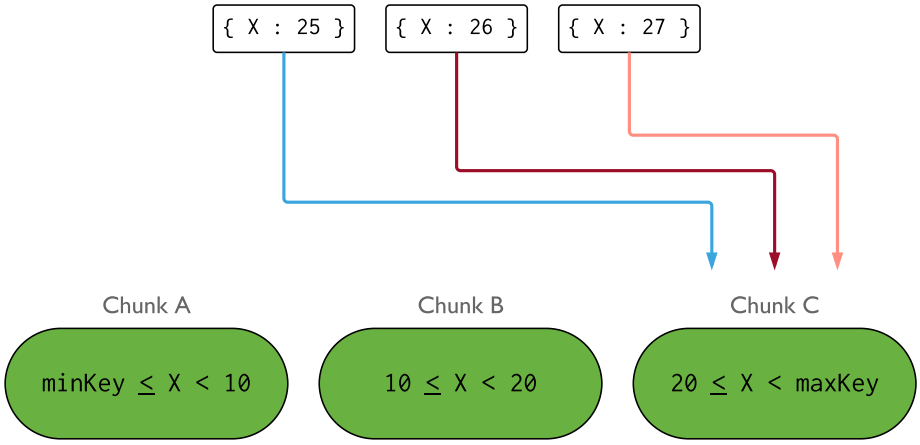
But generally, if you have enough RAM to cover the amount of data/indexes you expect to be frequently accessing then you will be fine.

Q12: When and why to use *Hashed Sharding* in MongoDB? ☆☆☆☆

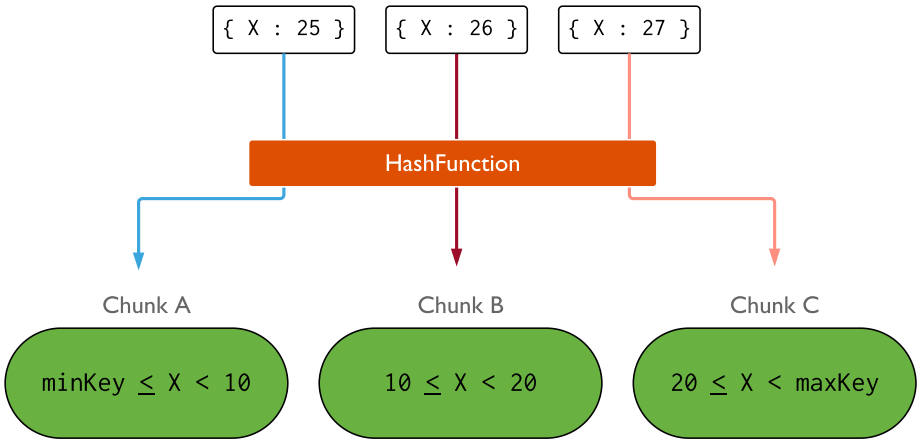
Topics: MongoDB

Answer:

A **monotonically** increasing **shard key**, is an increasing function such as a counter or an ObjectId. When writing documents to a sharded system using an incrementing counter as it’s shard key, all documents will be written to the same shard and chunk. This restricts insert operations to the single shard containing this chunk, which reduces or removes the advantage of distributed writes in a sharded cluster.



Hashed sharding provides a more even data distribution across the sharded cluster. Post-hash, documents with "close" shard key values are unlikely to be on the same chunk or shard - the `mongos` is more likely to perform Broadcast Operations to fulfill a given ranged query. By using a hashed index on `X`, the distribution of inserts is similar to the following:



Since the data is now distributed more evenly, inserts are efficiently distributed throughout the cluster.

The field you choose as your hashed shard key should have a good cardinality, or large number of different values. Hashed keys are ideal for shard keys with fields that change monotonically like ObjectId values or timestamps. A good example of this is the default `_id` field, assuming it only contains ObjectId values.

Q13: What are the differences between MongoDB and MySQL?

☆☆☆☆☆

Topics: MongoDB MySQL

Answer:

The Major Differences between MongoDB and MySQL are:

1. There is a difference in the representation of data in the two databases. In MongoDB, data represents in a collection of JSON documents while in MySQL, data is in tables and rows. JSON documents can compare to associative arrays when using PHP and directory objects when using Python.
2. When it comes to querying, you have to put a string in the query language that the DB system parses. The query language is called Structured Query Language, or SQL,from where MySQL gets its name. This exposes your DB susceptible to SQL injectionattacks. On the other hand, MongoDB's querying is object-oriented,

which means you pass MongoDB a document explaining what you are querying. There is no parsing whatsoever, which will take some time getting used to if you already use SQL.

3. One of the greatest benefits of relational databases like MySQL is the JOIN operation. The operation allows for the querying across several tables. Although MongoDB doesn't support joints, it supports multi-dimensional data types like other documents and arrays.
4. With MySQL, you can have one document inside another (embedding). You would have to create one table for comments and another for posts if you are using MySQL to create a blog. In MongoDB, you will only have one array of comments and one collection of posts within a post.
5. MySQL supports atomic transactions. You can have several operations within a transaction and you can roll back as if you have a single operation. There is no support for transactions in MongoDB and the single operation is atomic.
6. One of the best things about MongoDB is that you are not responsible for defining the schema. All you need to do is drop in documents. Any 2 documents in a collection need not be in the same field. You have to define the tables and columns before storage in MySQL. All rows in a table share the same columns.
7. MongoDB's performance is better than that of MySQL and other relational DBs. This is because MongoDB sacrifices JOINS and other things and has excellent performance analysis tools. Note that you still have to index the data and the data in most applications is not enough for them to see a difference. MySQL is criticized for poor performance, especially in ORM application. However, you are unlikely to have an issue if you do proper data indexing and you are using a database wrapper.
8. One advantage of MySQL over NoSQL like MongoDB is that the community in MySQL is much better than NoSQL. This is mostly because NoSQL is relatively new while MySQL has been around for several years.
9. There are no reporting tools with MongoDB, meaning performance testing and analysis is not always possible. With MySQL, you can get several reporting tools that help you rove the validity of your applications.
10. RDBSs function on a paradigm called ACID, which is an acronym for (Atomicity, Consistency, Isolation, and Durability). This is not present in MongoDB database.
11. MongoDB has a Map Reduce feature that allows for easier scalability. This means you can get the full functionality of MongoDB database even if you are using low-cost hardware.
12. You do not have to come up with a detailed DB model with MongoDB because of is non-relational. A DB architect can quickly create a DB without a fine-grained DB model, thereby saving on development time and cost.

Q14: What's the advantage of the backup features in *Ops Manager* versus traditional backup strategies? ☆☆☆☆☆

Topics: MongoDB

Answer:

Ops Manager offers a lot of advantages, including:

- Point-in-Time-Recovery, Scheduled Backups. You can generate a restore image from an exact time in the past, which can be very helpful for restoring operations just prior to a catastrophic event.
- Continuous, Incremental Backup. Data is backed up continuously as the cluster updates. Your backups are always up-to-date.
- Sharded Cluster Backup. Backing up sharded clusters can be hard. Ops Manager Backup automates this for you.
- Queryable Backup. Query backups directly without having to restore them. Find the backup you need or examine how data has changed over time.

Ops Manager is included with MongoDB Enterprise Advanced, and provides continuous backup and point-in-time restore for MongoDB. Those interested in a cloud-based backup solution should consider MongoDB Cloud Manager, which provides continuous, online backup and point-in-time restore for MongoDB as a fully managed service.

Q15: How to *condense* large volumes of data in Mongo? ☆☆☆☆☆

Topics: MongoDB

Answer:

Map-reduce is a data processing paradigm for condensing large volumes of data into useful aggregated results. For map-reduce operations, MongoDB provides the `mapReduce` database command.

Q16: What are *three primary concerns* when choosing a data management system? ☆☆☆☆☆

Topics: MongoDB

Answer:

There are three primary concerns you must balance when choosing a data management system: consistency, availability, and partition tolerance.

- **Consistency** - means that each client always has the same view of the data.
- **Availability** - means that all clients can always read and write.
- **Partition tolerance** - means that the system works well across physical network partitions.

According to the CAP Theorem, you can only pick two.

In addition to CAP configurations, another significant way data management systems vary is by the data model they use: relational, key-value, column-oriented, or document-oriented (there are others, but these are the main ones).

- *Relational* systems are the databases we've been using for a while now. RDBMSs and systems that support ACIDity and joins are considered relational.
- *Key-value* systems basically support get, put, and delete operations based on a primary key.
- *Column-oriented* systems still use tables but have no joins (joins must be handled within your application). Obviously, they store data by column as opposed to traditional row-oriented databases. This makes aggregations much easier.
- *Document-oriented* systems store structured "documents" such as JSON or XML but have no joins (joins must be handled within your application). It's very easy to map data from object-oriented software to these systems.

Q17: Where does MongoDB stand in the *CAP theorem*? ☆☆☆☆☆

Topics: MongoDB

Answer:

MongoDB is strongly consistent by default - if you do a write and then do a read, assuming the write was successful you will always be able to read the result of the write you just read. This is because MongoDB is a single-master system and all reads go to the primary by default.

On the other hand you can't just say that MongoDB is CP/AP/CA, because it actually is a trade-off between C, A and P, depending on both database/driver configuration and type of disaster: here's a visual recap, and below a more detailed explanation.

```
Scenario                   | Main Focus | Description
---------------------------|------------|-----------------------------------
No partition               |     CA     | The system is available
                           |            | and provides strong consistency
---------------------------|------------|-----------------------------------
partition,                 |     AP     | Not synchronized writes
majority connected         |            | from the old primary are ignored
---------------------------|------------|-----------------------------------
partition,                 |     CP     | only read access is provided
majority not connected     |            | to avoid separated and inconsistent systems
```

Consistency - MongoDB is strongly consistent when you use a single connection or the correct Write/Read Concern Level (Which will cost you execution speed). As soon as you don't meet those conditions (especially when you are reading from a secondary-replica) MongoDB becomes Eventually Consistent.

Availability - MongoDB gets high availability through Replica-Sets. As soon as the primary goes down or gets unavailable else, then the secondaries will determine a new primary to become available again. There is an disadvantage to this: Every write that was performed by the old primary, but not synchronized to the secondaries will be rolled back and saved to a rollback-file, as soon as it reconnects to the set(the old primary is a secondary now). So in this case some consistency is sacrificed for the sake of availability.

Partition Tolerance - Through the use of said Replica-Sets MongoDB also achieves the partition tolerance: As long as more than half of the servers of a Replica-Set is connected to each other, a new primary can be chosen. Why? To ensure two separated networks can not both choose a new primary. When not enough secondaries are connected to each other you can still read from them (but consistency is not ensured), but not write. The set is practically unavailable for the sake of consistency.

Q18: How much faster is Redis than MongoDB? ☆☆☆☆☆

Topics: Redis MongoDB

Answer:

- Rough results are **2x write, 3x read**.
- The general answer is that Redis 10 - 30% faster when the data set fits within working memory of a single machine. Once that amount of data is exceeded, Redis fails.
- But your best bet is to benchmark them yourself, in precisely the manner you are intending to use them. As a corollary you'll probably figure out the best way to make use of each.

Q19: What is *Index Cardinality* and why does it matter? ☆☆☆☆☆

Topics: Databases MongoDB SQL

Answer:

The **Index Cardinality** refers to how many possible values there are for a field.

Imagine we have a collection of all humans on earth with the following field:

```
- "sex" // 99.9% of the time "male" or "female", but string nonetheless (not binary)
- "name"
- "phone"
- "email"
```

The field `sex` only has two possible values. It has a very **low cardinality**. Other fields such as `names, usernames, phone numbers, emails`, etc. will have a more unique value for every document in the collection, which is considered **high cardinality**.

It's a rule of thumb to create indexes on `high-cardinality` keys or put `high-cardinality` keys *first* in the compound index.

Why?

The greater the cardinality of a field the more helpful an index will be, **because indexes narrow the search space, making it a much smaller set**.

Suppose you have an index on `sex` and are looking for men named John. You would only narrow down the resulting space by approximately `%50` if you indexed by `sex` first. Conversely, if you indexed by `name`, you would immediately narrow down the result set to a minute fraction of users named John, and then you would refer to those documents to check the gender.

Q20: When shall you *Shard* your MongoDB data? ☆☆☆☆☆

Topics: MongoDB

Answer:

One of the typical errors people commit is to shard too early. The reason this can be a problem is that sharding requires the developer to pick a shard key for the distribution of the writes and it’s easy to pick the wrong key early on due to not knowing how the data will be accessed over time.

There is some reason you might want to `Shard`.

1. Your **Working Set** no longer fits in the memory of your server. In this case, sharding can help make more of your Working Set stay in memory by pooling the RAM of all the shards. If you have a 20GB Working Set on a 16GB machine, sharding can split this across 2 machines for a total of 32GB of RAM available, potentially keeping all of the data in RAM and avoiding disk IO.
2. Scaling the write IO. You need to perform more write operations than what a single server can handle. By `Sharding` you can scale out the writes across multiple computers, increasing the total write throughput.

FullStack.Cafe - Kill Your Tech Interview

Q1: How to query MongoDB with `like`? ☆☆☆

Topics: MongoDB

Answer:

I want to query something as SQL's like query:

```
select *
from users
where name like '%m%'
```

How to do the same in MongoDB?

Answer

```
db.users.find({"name": /.*m.*/})
// or
db.users.find({"name": /m/})
```

You're looking for something that contains "m" somewhere (SQL's `'%'` operator is equivalent to Regexp's `'.*'`), not something that has "m" anchored to the beginning of the string.

Q2: How can I *combine data* from multiple collections into one collection? ☆☆☆

Topics: MongoDB

Answer:

MongoDB 3.2 allows one to combine data from multiple collections into one through the `$lookup` aggregation stage.

```
db.books.aggregate([{
    $lookup: {
            from: "books_selling_data",
            localField: "isbn",
            foreignField: "isbn",
            as: "copies_sold"
        }
}])
```

Q3: How do I perform the SQL `JOIN` equivalent in MongoDB? ☆☆☆

Topics: MongoDB

Answer:

Mongo is not a relational database, and the devs are being careful to recommend specific use cases for $lookup, but at least as of 3.2 doing join is now possible with MongoDB. The new$ lookup operator added to the aggregation pipeline is essentially identical to a left outer join:

```
{
   $lookup:
     {
       from: <collection to join>,
       localField: <field from the input documents>,
       foreignField: <field from the documents of the "from" collection>,
       as: <output array field>
     }
}
```

Q4: How to query MongoDB with `%like%`? ☆☆☆

Topics: MongoDB

Problem:

I want to query something as SQL's like query:

```
select *
from users
where name like '%m%'
```

How to do the same in MongoDB?

Solution:

```
db.users.find({name: /a/})  //like '%a%'
db.users.find({name: /^pa/}) //like 'pa%'
db.users.find({name: /ro$/}) //like '%ro'
```

Or using Mongoose:

```
db.users.find({'name': {'$regex': 'sometext'}})
```

Q5: Write equivalent MongoDB statement for this SQL aggregation statement ☆☆☆

Topics: MongoDB

Problem:

Consider the SQL query:

```
SELECT by, SUM(price) AS total FROM book GROUP BY price ORDER BY total
```

Write down the MongoDB equivalent.

Solution:

Consider:

```
db.book.aggregate([
  {
    $group: {
      _id: "$by",
      total: { $sum:"$price" }
    }
  },
  {
    $sort: { total:1 }
  }
])
```

Q6: Is possible in MongoDB to select collection's documents like in SQL `SELECT * FROM collection WHERE _id IN (1,2,3,4)`? ☆☆☆

Topics: MongoDB

Answer:

Consider:

```
db.collection.find({ _id: { $in: [1, 2, 3, 4] } });
```

Q7: Update MongoDB field *using value of another field* ☆☆☆☆

Topics: MongoDB

Answer:

Consider SQL command:

```
UPDATE Person SET Name = FirstName + ' ' + LastName
```

In Mongo, is it possible to update the value of a field using the value from another field?

Answer

You cannot refer to the document itself in an update (yet). You'll need to iterate through the documents and update each document using a function like:

```
db.person.find().snapshot().forEach(
    function (elem) {
        db.person.update(
            {
                _id: elem._id
```

```
            },
            {
                $set: {
                    name: elem.firstname + ' ' + elem.lastname
                }
            }
        );
    }
);
```

Q8: How to *remove a field* completely from a MongoDB document?

☆☆☆☆

Topics: MongoDB

Answer:

Suppose this is a document.

```
{
    name: 'book',
    tags: {
        words: ['abc','123'], // <-- remove it comletely
        lat: 33,
        long: 22
    }
}
```

How do I remove `words` completely from all the documents in this collection?

Answer

```
db.example.update({}, {$unset: {words:1}}, false, true);
```

Q9: MongoDB relationships. What to use - *embed* or *reference*?

☆☆☆☆

Topics: MongoDB

Problem:

I want to design a question structure with some comments, but I don't know which relationship to use for comments: embed or reference? Explain me pros and cons of both solutions?

Solution:

In general,

- **embed** is good if you have one-to-one or one-to-many relationships between entities, and
- **reference** is good if you have many-to-many relationships.

Also consider as a general rule, if you have a lot of [child documents] or if they are large, a separate collection might be best. Smaller and/or fewer documents tend to be a natural fit for embedding.

Q10: How to get the last `N` records from `find`? ☆☆☆☆

Topics: MongoDB

Answer:

Use `sort` and `limit` in chain:

```
db.foo.find().sort({_id:1}).limit(50);
```

Q11: How to find MongoDB records where array field is *not empty*? ☆☆☆☆

Topics: MongoDB

Answer:

Consider some variants:

```
collection.find({ pictures: { $exists: true, $not: { $size: 0 } } })
collection.find({ pictures: { $exists: true, $ne: [] } })
collection.find({ pictures: { $gt: [] } }) // since MongoDB 2.6
collection.find({'pictures.0': {$exists: true}}); // beware if performance is important - it'll do a full
collection scan
```

Q12: How to check if a field contains a *substring*? ☆☆☆☆

Topics: MongoDB

Answer:

Consider:

```
db.users.findOne({"username" : {$regex : ".*subsctring.*"}});
```

Q13: How do I create a *Compound Index* in MongoDB? ☆☆☆☆

Topics: MongoDB

Answer:

To specify a *compound index*, use the `compoundIndex` method.

For example:

```
var createCompoundIndex = function(db, callback) {
  // Get the documents collection
  var collection = db.collection('users');
  // Create the index
```

```
  collection.createIndex(
    { lastName : -1, dateOfBirth : 1 }, function(err, result) {
    console.log(result);
    callback(result);
  });
};
```

The above example specifies a *compound index* key composed of the `lastName` field sorted in descending order, followed by the `dateOfBirth` field sorted in ascending order.

Q14: Will you create *Compound Index* with `sex` first or `name` first? Explain. ☆☆☆☆

Topics: MongoDB

Problem:

Imagine we have a collection of all humans on earth with an index on `sex` (99.9% of the time "male" or "female", but string nontheless (not binary)) and an index on `name`.

If we would want to be able to select all people of a certain `sex` with a certain `name`, e.g. all "male"s named *"John"*, is it better to have a compound index with `sex` first or `name` first? Why (not)?

Solution:

The **index cardinality** refers to how many possible values there are for a field. The field `sex` only has two possible values. It has a very **low cardinality**. Other fields such as `names, usernames, phone numbers, emails`, etc. will have a more unique value for every document in the collection, which is considered **high cardinality**.

The greater the **cardinality** of a field the more helpful an index will be, because **indexes narrow the search space, making it a much smaller set**. Try to create indexes on **high-cardinality** keys or put high-cardinality keys first in the compound index.

When you create a compound index, **1 Index** will hold multiple fields. So if we index a collection by `{"sex" : 1, "name" : 1}`, the index will look roughly like:

```
["male","Rick"] -> 0x0c965148
["male","John"] -> 0x0c965149
["male","Sean"] -> 0x0cdf7859
["male","Bro"] ->> 0x0cdf7859
...
["female","Kate"] -> 0x0c965134
["female","Katy"] -> 0x0c965126
["female","Naji"] -> 0x0c965183
["female","Joan"] -> 0x0c965191
["female","Sara"] -> 0x0c965103
```

If we index a collection by `{"name" : 1, "sex" : 1}`, the index will look roughly like:

```
["John","male"] -> 0x0c965148
["John","female"] -> 0x0c965149
["John","male"] -> 0x0cdf7859
["Rick","male"] -> 0x0cdf7859
...
["Kate","female"] -> 0x0c965134
["Katy","female"] -> 0x0c965126
```

```
["Naji","female"] -> 0x0c965183
["Joan","female"] -> 0x0c965191
["Sara","female"] -> 0x0c965103
```

Having `{name:1}` as the Prefix will serve you much better in using compound indexes because you allowed MongoDB to be **selective**. Selectivity is the ability of a query to *narrow* results using the index. Effective indexes are more **selective**.

Q15: How to find document with array that contains a specific value? ☆☆☆☆☆

Topics: MongoDB

Problem:

You have this schema:

```
person = {
    name : String,
    favoriteFoods : Array
}
```

where the `favoriteFoods` array is populated with strings. How can I find all persons that have `sushi` as their favorite food using MongoDB?

Solution:

Consider:

```
PersonModel.find({ favouriteFoods: "sushi" }, ...);
PersonModel.find({ favouriteFoods: { "$in" : ["sushi"]} }, ...);
```

Q16: Is it possible to update MongoDB field using value of another field? ☆☆☆☆☆

Topics: MongoDB

Problem:

In SQL we will use:

```
UPDATE Person SET Name = FirstName + ' ' + LastName
```

Is it possible with MongoDB?

Solution:

You cannot refer to the document itself in an update (yet). You'll need to iterate through the documents and update each document using a function. So consider:

```
db.person.find().snapshot().forEach(
  function(elem) {
    db.person.update({
      _id: elem._id
    }, {
      $set: {
        name: elem.firstname + ' ' + elem.lastname
      }
    });
  }
);
```

Q17: How would you design the `One-to-Squillions` relationship in MongoDB? ☆☆☆☆☆

Topics: MongoDB

Answer:

An example of `one-to-squillions` might be an event logging system that collects log messages for different machines. Any given host could generate enough messages to overflow the 16 MB document size, even if all you stored in the array was the `ObjectID`.

This is the classic use case for “parent-referencing” – you’d have a document for the host, and then store the `ObjectID` of the host in the documents for the log messages.

```
> db.hosts.findOne()
{
    _id : ObjectID('AAAB'),
    name : 'goofy.example.com',
    ipaddr : '127.66.66.66'
}

>db.logmsg.findOne()
{
    time : ISODate("2014-03-28T09:42:41.382Z"),
    message : 'cpu is on fire!',
    host: ObjectID('AAAB')       // Reference to the Host document
}
```

You’d use a *application-level* join to find the most recent 5,000 messages for a host:

```
  // find the parent ‘host’ document
> host = db.hosts.findOne({ipaddr : '127.66.66.66'});  // assumes unique index
   // find the most recent 5000 log message documents linked to that host
> last_5k_msg = db.logmsg.find({host: host._id}).sort({time : -1}).limit(5000).toArray()
```