<https://university.mongodb.com/exam/practice/DBA/results/5ca5c7ac09a54d2fbf5e404b>

**Philosophy and Features**

1. Which writes are atomic in MongoDB 4.0? Check all that apply.

**An update to a single document in a replica set**

**An update to multiple documents in a replica set using transactions**

**An update to a single document in a sharded cluster**

An update to multiple documents in a sharded cluster

**Detailed Answer**

*MongoDB 4.0* introduced updates to multiple documents in a replica set through *transactions*.

In *MongoDB 3.6*, updates to single documents in replica sets or sharded clusters were atomic.

To learn more, visit the following MongoDB Documentation page:

* [Atomicity and Transactions](https://docs.mongodb.com/manual/core/write-operations-atomicity/)

1. In order to ensure that you can maintain high availability in the face of server failure, you should use which of the following MongoDB feature?

**Replication**

Sharding

Proper storage engine

Proper defined user roles

Indexes on all of the documents

**Detailed Answer**

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.

If a member of the replica set becomes unavailable due to maintenance or a hardware crash, the other members will still be able to provide the applications access to the documents.

To learn more, visit the following MongoDB Documentation page:

* [Redundancy and Data Availability](https://docs.mongodb.com/manual/replication/#redundancy-and-data-availability)

1. Why does MongoDB use BSON rather than JSON? Check all that apply.

**BSON includes metadata to describe a document/object**

**BSON supports more data types than JSON**

BSON is more human readable than JSON

**Detailed Answer**

BSON extends the JSON model to provide additional data types, ordered fields, and to be efficient for encoding and decoding within different languages.

To learn more, visit the following MongoDB Documentation page:

* [MongoDB Extended JSON](https://docs.mongodb.com/manual/reference/mongodb-extended-json/index.html)

1. Which of the following are valid types in BSON? Check all that apply.

**Int64**

**ObjectId**

**Decimal128**

**Detailed Answer**

These are all valid BSON types.

To learn more, visit the following MongoDB Documentation page:

* [BSON Types](https://docs.mongodb.com/manual/reference/bson-types/)

### CRUD

1. Which of the documents below will be retrieved by the following query? Assume the documents are stored in a collection called "sample". Check all that apply.

db.sample.find( { "$or" : [ { "a" : { "$in" : [ 3, 10] } }, { "b" : { "$lt" : 2 } } ] } )

{ "\_id" : 1, "a" : 0, "c" : 0, "b" : 2 }

**{ "\_id" : 2, "a" : 2, "c" : 0, "b" : 1 }**

{ "\_id" : 3, "a" : 4, "c" : 0, "b" : 14 }

{ "\_id" : 4, "a" : 5, "c" : 0, "b" : 17 }

**{ "\_id" : 5, "a" : 3, "c" : 0, "b" : 12 }**

{ "\_id" : 6, "a" : 1, "c" : 1, "b" : 5 }

{ "\_id" : 7, "a" : 8, "c" : 1, "b" : 7 }

**{ "\_id" : 8, "a" : 11, "c" : 1, "b" : 0 }**

**{ "\_id" : 9, "a" : 17, "c" : 1, "b" : 1 }**

**{ "\_id" : 10, "a" : 3, "c" : 1, "b" : 1 }**

**Detailed Answer**

The $or operator means that any document that satisfies one of the conditions will be retrieved.

To learn more, visit the following MongoDB Documentation page:

* [$or operator](https://docs.mongodb.com/manual/reference/operator/query/or)

1. You perform the following query on the sayings collection, which has the index { quote : “text” }:

db.sayings.find( { $text : { $search : "fact find" } } )

Assuming the following documents are in the collection, which ones will the query return? Check all that apply.

**{ \_id : 1, quote : “That’s a fact, Jack.” }**

**{ \_id : 2, quote : “Find out if that fact is correct.” }**

{ \_id : 3, quote : “Nobody will ever catch me.” }

**Detailed Answer**

Because the text "Nobody will ever catch me." contains neither "fact" nor "find" it won't be included in the results.

To learn more, visit the following MongoDB Documentation page:

* [Text Indexes](https://docs.mongodb.com/manual/core/index-text/)

1. Which of the following queries on the "team" collection will return only the first five elements of the array in the "scores" field?

Example document (as it exists in the collection):

> db.team.find()

{ "\_id" : 12, scores : [ 3, 5, 7, 2, 1, -4, 3, 12 ] }

Document you want returned by your query:

{ "\_id" : 12, "scores" : [ 3, 5, 7, 2, 1 ] }

db.team.find( { } , { scores : [ 0, 1, 2, 3, 4, 5 ] } )

db.team.find( { } , { scores : [ 0 , 5 ] } )

**db.team.find( { } , { scores : { $slice : [ 0 , 5 ] } } )**

db.team.find( { } , { scores : { $substr[ 0 , 5 ] } } )

db.team.find( { scores : [ 0 , 5 ] } )

**Detailed Answer**

Only the query

db.team.find( { } , { scores : { $slice : [ 0 , 5 ] } } )

will return the desired results.

The $slice projection operator is used to control how many elements of an array will be returned after fetching it.

To learn more, visit the following MongoDB Documentation page:

* [$slice (projection)](https://docs.mongodb.com/manual/reference/operator/projection/slice/index.html)

1. On which of the following CRUD operations can you specify a write concern? Check all that apply.

**db.collection.insertOne()**

db.collection.find()

**db.collection.updateMany()**

**Detailed Answer**

*write concerns* apply to write operations. insertOne and updateMany are write operations.

find is a read operation. Read operations can be influenced by *read concerns* or *read preferences*.

To learn more, visit the following MongoDB Documentation pages:

* [Write Concern for Replica Sets](https://docs.mongodb.com/manual/core/replica-set-write-concern/)
* [Write Concern](https://docs.mongodb.com/manual/reference/write-concern/)

1. Which of the documents below would be modified by the following update? Assume all documents are in the "people" collection. Check all that apply.

> db.people.updateMany( { "city" : "Seattle", "state" : "WA" }, { "$addToSet" : { "likes" : "forest" } } )

**{ "\_id" : ObjectId("57fd48257268886f789b33ff"), "firstName" : "Arthur", "lastName" : "Aaronson", "state" : "WA", "city" : "Seattle", "likes" : [ "dogs", "cats", "hiking" ] }**

{ "\_id" : ObjectId("57fd48257268886f789b3400"), "firstName" : "Beth", "lastName" : "Barnes", "state" : "WA", "city" : "Richland", "likes" : [ "forest", "desert" ] }

{ "\_id" : ObjectId("57fd48257268886f789b3401"), "firstName" : "Charlie", "lastName" : "Carlson", "state" : "CA", "city" : "San Diego", "likes" : [ "desert", "beach" ] }

{ "\_id" : ObjectId("57fd48257268886f789b3402"), "firstName" : "Dawn", "lastName" : "Davis", "state" : "WA", "city" : "Seattle", "likes" : [ "forest", "mountains", "hiking" ] }

**Detailed Answer**

Two documents satisfy the following criteria:

{ "city" : "Seattle", "state" : "WA" }

Out of these two documents, the document with \_id:ObjectId("57fd48257268886f789b3402") already contains forest in the likes array and therefore, $addToSet is not going to add anything to this document.

The document for \_id:ObjectId("57fd48257268886f789b33ff") is the only one that will be updated since it meets both criteria, and forest is not in the likes array.

To learn more, visit the following MongoDB Documentation page:

* [$addToSet](https://docs.mongodb.com/manual/reference/operator/update/addToSet/)

1. Consider the following example document from the *sample* collection. All documents in this collection have the same schema.

* {
* "\_id" : 3,
* "a" : 7,
* "b" : 4
* }
* Which of the following queries will replace this with the document,
* {
* "\_id" : 7,
* "c" : 4
* }

**This operation cannot be done with a single query.**

**Detailed Answer**

The correct answer is that this operation cannot be done in a single query.

To understand why, recall that the \_id field of a document is *immutable*.

In fact, trying this operation with:

updateOne({\_id: 3}, {$set: { \_id: 7, c: 4 }, $unset: { a: "", b: "" }})

produces the following error:

"Performing an update on the path '\_id' would modify the immutable field '\_id'"

To learn more, visit the following MongoDB Documentation page:

* [Documents in MongoDB](https://docs.mongodb.com/manual/core/document/)

### Indexes

1. You have the following index on the toys collection:

{

"manufacturer" : 1,

"name" : 1,

"date" : -1

}

Which of the following queries are able to use the index? Check all that apply.

**db.toys.find( { manufacturer : "Matteo", name : "Barbara", date : "2018-07-02" } )**

**db.toys.find( { name : "Big Rig Truck", date : "2018-02-01", manufacturer : "Tanko" } )**

**db.toys.find( { date : "2018-03-01", manufacturer : "Loggo", name : "Brick Set" } )**

**Detailed Answer**

All answer choices are correct!

Consider just the first choice. Any document that could match *"manufacturer is Matteo AND name is Barbara AND date is 2018-07-02* would have to match *date is 2018-07-02 AND, name is Barbara, and manufacturer is Matteo*.

Because of this fact, the optimizer is able to rearrange the the search terms, using the existing index for each query.

To learn more, visit the following MongoDB Documentation page:

* [Prefixes](https://docs.mongodb.com/manual/core/index-compound/#prefixes)

1. Which of the following must be true for a query to be a covered query? Check all that apply.

**All fields used in the selection filter of the query must be in the index that the query uses**

**All fields returned in the results must be in the index that the query uses**

All fields returned in the results must be fields in the selection filter of the query

**Detailed Answer**

All fields used in the selection filter of the query must be in the index, so the system can find the documents that satisfy the selection filter without having to retrieve the document from the collection.

All fields returned in the results must be in the index, so again there is no need to retrieve the full document. A common mistake is not to provide a projection that filters out the field \_id, which is returned by default. If the \_id field is not a field in the index definition, it is not available, and the query system will need to fetch the full document to retrieve the value.

On the other hand, it is OK to ask for more fields than the ones provided in the selection filter, as long as those are in the index values, the system has all the information needed to avoid fetching the full document from the collection.

To learn more, visit the following MongoDB Documentation page:

* [Covered Query](https://docs.mongodb.com/manual/core/query-optimization/#covered-query)

**3.** You have created the following index on the foo collection:

> db.foo.createIndex( { a : 1, b : -1, c : -1, d: 1 } )

Which of the following queries will be able to fulfill the query without an in-memory sort (i.e., it's able to use the index to sort)? Check all that apply.

db.foo.find( { a : { $gt : 100 } } ).sort( { c : -1 } )

**db.foo.find( { a : 100 } ).sort( { b : 1, c : 1 } )**

**db.foo.find( { a : 200, b : { $lt : 100 } } ).sort( { b : 1 } )**

**Detailed Answer**

Recall that compound indexes should be built in the order of *equality, range, sort* for common operational query patterns.

Also recall that as long as the query uses all keys of a compound index or a combination of index prefixes, it will make use of the existing index.

To learn more, visit the following MongoDB Documentation page:

* [Compound Indexes](https://docs.mongodb.com/manual/core/index-compound/)

1. Given the following example document:

{

"\_id": ObjectId("5360c0a0a655a60674680bbe"),

"user": {

"login": "ir0n",

"description": "Made of metal"

"date": ISODate("2014-04-30T09:16:45.836Z"),

}

}

and the following index:

db.users.createIndex( { "user.login": 1, "user.date": -1 }, "myIndex" )

When performing the following query:

db.users.find( { "user.login": /^ir.\*/ }, { "user":1, "\_id":0 } ).sort( { "user.date":1 } )

1. Which of the following statements correctly describes how MongoDB will handle the query?

As a covered query using "myIndex" because we are filtering out "\_id" and only returning "user.login"

**As an index scan that uses "myIndex" because field "user.login" is indexed**

As an optimized sort query (no explicit sort stage) using "myIndex" because we are sorting on an indexed field

MongoDB will need to do a table/collection scan to find matching documents

None of the above

**Detailed Answer**

Because the field user.login is indexed and the regex *beginning of line* operator is being used (^), the index myIndexwill be used for this query.

To learn more, visit the following MongoDB Documentation page:

* [Evaluation Query Operators - $regex](https://docs.mongodb.com/manual/reference/operator/query/regex/)

1. Suppose you have a collection "sample" with an index created as follows.

db.sample.createIndex( { "foo" : -1 } )

For which of the following queries can MongoDB efficiently look at only a subset of the index entries, rather than all of the index entries? Check all that apply.

db.sample.find( { "foo" : /a.\*b/ } )

**db.sample.find( { "foo" : /^c.\*d/ } )**

**db.sample.find( { "foo" : /^e.\*f/ } )**

**Detailed Answer**

By specifying the "beginning of line" regex operation, you constrain the query and MongoDB can efficiently navigate the index to the correct location.

To learn more, visit the following MongoDB Documentation page:

* [Index Use](https://docs.mongodb.com/manual/reference/operator/query/regex/index.html#index-use)

1. You have the following indexes on your collection:

[

{

"v" : 1,

"key" : {

"\_id" : 1

},

"name" : "\_id\_",

"ns" : "test.sample"

},

{

"v" : 1,

"key" : {

"name" : 1,

"date" : 1,

"phone" : 1

},

"name" : "name\_1\_date\_1\_phone\_1",

"ns" : "test.sample"

}

]

Which of the following queries will use an index? Check all that apply.

**db.sample.find( { \_id : 22, date: ISODate("2012-07-04" } )**

**db.sample.find( { date: ISODate("2011-07-04"), name : "Alice" } )**

db.sample.find( { title : "DBA" } )

db.sample.find( { phone : "123-456-7890"), info : "201-555-5792" } )

**Detailed Answer**

There are 2 indexes:

{ \_id: 1 }

{ "name" : 1, "date" : 1, "phone" : 1 }

The order of the fields in the index is important, however, the order of the fields in the query are not significant as the query planner will "reorder" the query terms to match a prefix of, or the full compound index.

The query on \_id will use the first index. Because \_id is guaranteed to be unique, it's possible for the planner to make this optimization. To be sure, there will still be a FETCH stage to get the document and ensure the date predicate is fulfilled.

The query on date, name will also use an index, the name\_1\_date\_1\_phone\_1 index, because a prefix is specified (name, date).

The query on title is using a field for which there is no index.

As for the query on phone and info, phone is indexed, however as the third member of the compound index so won't be used.

To learn more, visit the following MongoDB Documentation page:

* [Query Optimization](https://docs.mongodb.com/manual/core/query-optimization/)

1. You have the following indexes on the *things* collection:

[

{

"v" : 1,

"key" : {

"\_id" : 1

},

"name" : "\_id\_",

"ns" : "test.things"

},

{

"v" : 1,

"key" : {

"a" : 1

},

"name" : "a\_1",

"ns" : "test.things"

},

{

"v" : 1,

"key" : {

"c" : 1,

"b" : 1,

"a" : 1

},

"name" : "c\_1\_b\_1\_a\_1",

"ns" : "test.things"

}

]

Which of the following queries will require that you load every document into RAM in order to fulfill the query? Assume that no data is being written during the query. Check all that apply.

**db.things.find( { b : 1 } ).sort( { c : 1, a : 1 } )**

db.things.find( { c : 1 } ).sort( { a : 1, b : 1 } )

db.things.find( { a : 1 } ).sort( { b : 1, c : 1 } )

**Detailed Answer**

The query

db.things.find( { b: 1} ).sort( {c: 1, a: 1} )

will require every document be loaded into RAM in order to fulfill the query. This is because the initial match on the b key does not use any existing index or index prefix.

To learn more, visit the following MongoDB Documentation page:

* [Compound Indexes - Prefixes](https://docs.mongodb.com/manual/core/index-compound/#prefixes)

1. Which of the following statements are true of unique indexes? Check all that apply.

The only possible unique index is the "\_id" field.

**The "unique" constraint on an index ensures that no two (or more) documents can share a value for that field in a collection**

**Hashed indexes cannot be unique.**

**Detailed Answer**

Unique indexes have certain properties and restrictions that you should be familiar with.

For example, they ensure that no documents have the same data at the same key that carries a unique index, and you may not specify a unique constraint on a field that is specified as a hashed index.

To learn more, visit the following MongoDB Documentation page:

* [Unique Indexes](https://docs.mongodb.com/manual/core/index-unique/)

1. Adding an index on { a : 1 } can potentially decrease the speed of which of the following operations? Check all that apply.

db.collection.find( { a : 232 } )

**db.collection.updateOne( { b : 456 }, { $inc : { a : 1 } } )**

**db.collection.insertOne( { a : 341 } )**

**Detailed Answer**

The operators updateOne and insertOne are correct because adding indexes does impact write performance.

Remember, write operations that modify an indexed field may require MongoDB to update the indexes associated with the document.

That said, not having the appropriate index for a given query will produce a *collection scan* on the collection, and those are undesirable.

To learn more, visit the following MongoDB Documentation page:

* [Indexes](https://docs.mongodb.com/manual/core/data-model-operations/#indexes)

1. Which of the following is true of covered queries? Check all that apply.

**MongoDB can satisfy covered queries using only index keys.**

MongoDB can satisfy covered queries efficiently without the use of an index.

Covered queries ordinarily have slower response times than queries that are not covered.

**Detailed Answer**

Covered queries are the best queries!

The underlying index supports the entire query, so no document information is required to be fetched from disk. With a covered query, you are servicing the operation entirely from the index, which is usually faster than examining each document.

To learn more, visit the following MongoDB Documentation page:

[Covered Query](https://docs.mongodb.com/manual/core/query-optimization/index.html#covered-query)

#### **AGGREGATION**

1. Which of the following statements are true about the $match pipeline stage? Check all that apply.

**You should use it as early as possible in the pipeline.**

**It can be used as many times as needed.**

**It has a syntax similar to find() commands.**

**Detailed Answer**

You should use the $match stage as early as possible in your pipeline. The reason is that by filtering documents that are not part of the answer early will mean the rest of the pipeline will process less documents and will be faster.

Most stages, except few exceptions, can be used many times in a pipeline.

To learn more, visit the following MongoDB Documentation page:

* [$match (aggregation)](https://docs.mongodb.com/manual/reference/operator/aggregation/match)

**2**. Suppose you have the following collection with only 2 documents:

> db.people.find()

{ "\_id" : "apples", "traits" : [ "sweet" , "crispy" ] }

{ "\_id" : "oranges", "traits" : [ "sweet" , "orange" , "juicy" ] }

You run an aggregation query and begin with { $unwind : "$traits" } .

How many documents will you have in the pipeline after the $unwind stage?

**5**

**Detailed Answer**

$unwind takes a field (array) as its argument. It then replaces the original document by N documents, one for each value in the array. So taking the "apple" document, you would get 2 resulting documents:

{ "\_id" : "apples", "traits" : "sweet" }

{ "\_id" : "apples", "traits" : "crispy" }

Similarly, the "oranges" document would unwind into 3 documents, for a total of 5 documents.

To learn more, visit the following MongoDB Documentation page:

* [$unwind (aggregation)](https://docs.mongodb.com/manual/reference/operator/aggregation/unwind)

1. Suppose you have the following collection in MongoDB. The collection is named *alpha*.

{"\_id" : 1, "region" : "NW1", "leads" : 1, "email" : "mlangley@co1.com"}

{"\_id" : 2, "region" : "NW1", "leads" : 1, "email" : "jpicoult@co4.com"}

{"\_id" : 3, "region" : "NW1", "leads" : 2, "email" : "zzz@company2.com"}

{"\_id" : 4, "region" : "SE1", "leads" : 8, "email" : "mary@hssu.edu" }

{"\_id" : 5, "region" : "SE2", "leads" : 4, "email" : "janet@col.edu"}

{"\_id" : 6, "region" : "SE2", "leads" : 2, "email" : "bill@uni.edu"}

{"\_id" : 7, "region" : "SE2", "leads" : 4, "email" : "iii@company1.com"}

{"\_id" : 8, "region" : "SW1", "leads" : 1, "email" : "phil@co3.com"}

{"\_id" : 9, "region" : "SW1", "leads" : 2, "email" : "thomas@company.com"}

{"\_id" : 10, "region" : "SW2", "leads" : 2, "email" : "sjohnson@uchi.edu"}

{"\_id" : 11, "region" : "SW2", "leads" : 5, "email" : "tsamuel@someco.com"}

How many documents will be returned in the result set in response to the following aggregation query?

db.alpha.aggregate( [

{ "$group" : { "\_id" : "$region" ,

"count" : { "$sum" : 1 } } } ,

{ "$match" : { "count" : { "$gte" : 3 } } }

] )

**2**

**Detailed Answer**

The first stage, $group, groups by the *region* and uses the $sum accumulator expression to count the number of documents in each group.

Next, these documents flow into the $match stage, where documents with a count that is less than 3 (3 out of the 5 groups) are filtered out, returning two documents.

{ "\_id" : "SE2", "count" : 3 }

{ "\_id" : "NW1", "count" : 3 }

To learn more, visit the following MongoDB Documentation page:

* [$group (aggregation)](https://docs.mongodb.com/manual/reference/operator/aggregation/group)
* [$match (aggregation)](https://docs.mongodb.com/manual/reference/operator/aggregation/match)

**4.** The users collection contains the following documents:

* > db.users.find()
* { "\_id" : 1, "name" : "dave123", "q1" : true, "q2" : true }
* { "\_id" : 2, "name" : "dave2", "q1" : false, "q2" : false }
* { "\_id" : 3, "name" : "ahn", "q1" : true, "q2" : true }
* { "\_id" : 4, "name" : "li", "q1" : true, "q2" : false }
* { "\_id" : 5, "name" : "annT", "q1" : false, "q2" : true }
* { "\_id" : 6, "name" : "li", "q1" : true, "q2" : true }
* { "\_id" : 7, "name" : "ty", "q1" : false, "q2" : true }
* You perform the following query:
* db.users.aggregate( [ { $sample : { size : 3 } } ] )
* Which of the following are possible outputs from the database? Check all that apply.

$sample возвращает **любые** 3 док-та

**Detailed Answer**

The $sample stage in the *Aggregation Framework* returns a subset of documents in a random fashion. In the above pipeline, we ask the stage to return 3 documents, so the documents returned could be any document in any order.

This $sample stage is useful when you want to test something against a large dataset, as processing all the documents would take too much time.

To learn more, visit the following MongoDB Documentation page:

* [$sample (aggregation)](https://docs.mongodb.com/manual/reference/operator/aggregation/sample)

### Replication

1. Which of the following is a correct definition for idempotence?

Affecting the same fields in successive queries. e.g., db.collection.updateOne( { \_id : 3 } , { $inc : { a : 7 , b : 6 } } ) and db.collection.updateOne( { \_id : 3 } , { $inc : { a : 1 , b : 2 } } ) are idempotent with respect to one another

**If an action may be performed multiple times and have the same effect as if it had been performed once, it is idempotent. For example, $set: {a: 3} is idempotent, but $inc: {a: 1} is not.**

Having the property that order does not matter. I.e., if A and B are update operations, then they are idempotentif A(B(document)) = B(A(document))

Requiring the same amount of data to be transferred makes two queries idempotent with respect to one another

Setting an entire document's state with an update, rather than just modifying a subset in the argument of the update, is an idempotent operation.

**Detailed Answer**

The desire for the *Oplog* to be *idempotent* is to ensure that if the server needs to resume applying *Oplog* entries it will always get to the same end state, regardless if it reapplies some that entry already applied. For example, if the server crashes applying *oplog5* and it is difficult to identify if *oplog5* is applied, then idempotency let you restart at *oplog4* without issues.

Another goal is to have the new state of the document be independent of a previous state. This means all operators like $em, which relies on the previous value to determine the new value, needs to be transformed to the actual values seen. For example, if an increment operation results in modifying a field from the value '4' to the value '5', the operation should be transformed to simply set '5' on that field. Replaying this operation many times always lead to the same result.

To learn more, visit the following MongoDB Documentation pages:

* [Idempotency definition](https://docs.mongodb.com/manual/reference/glossary/#term-idempotent)
* [Replica Set Oplog](https://docs.mongodb.com/manual/core/replica-set-oplog/)

1. Which of the following describe the primary reasons MongoDB supports replication? Check all that apply.

**To provide high availability**

To enable horizontal scaling

**To prevent downtime in case of a disaster at a data center**

**Detailed Answer**

*Horizontal scaling* is defined as adding more servers, while *vertical scaling* is defined as increasing the resources of a server. *Horizontal scaling* is achieved by *sharding*, not *replication*

A good practice when using *replication* is to have replicas in different geographical regions. If one region becomes unavailable due to a major failure in a data center or the network connection to it, the applications will continue to operate without downtime.

Note that replication helps in case of physical disasters, but does not prevent against logical disasters like the deletion of a database. For that reason, *replication* does not replace *backups*.

To learn more, visit the following MongoDB Documentation page:

* [Replication](https://docs.mongodb.com/manual/replication)

1. What settings can be controlled by rs.reconfig()? Check all that apply.

**Priority for each replica set member**

**Which replica set members are hidden**

Which version of MongoDB each replica set member is running

**Detailed Answer**

You can not set the version of *MongoDB* a given replica is using. This would be a little difficult to control, as nodes may not have the desired version installed yet.

The other options make more sense to be controlled in a global configuration, as you want to be able to change them from one location (the *Primary*), and have the changes being effective without having to restart the *mongod* processes.

To learn more, visit the following MongoDB Documentation page:

* [rs.reconfig()](https://docs.mongodb.com/manual/reference/method/rs.reconfig/)

1. The people collection contains the following documents:

{ "\_id" : ObjectId("57fd59a2d630a0fd9685a148"), "firstName" : "Arthur", "lastName" : "Aaronson", "state" : "WA", "city" : "Seattle", "likes" : [ "dogs", "cats" ] }

{ "\_id" : ObjectId("57fd59a2d630a0fd9685a149"), "firstName" : "Beth", "lastName" : "Barnes", "state" : "WA", "city" : "Richland", "likes" : [ "forest", "desert" ] }

{ "\_id" : ObjectId("57fd59a2d630a0fd9685a14a"), "firstName" : "Charlie", "lastName" : "Carlson", "state" : "CA", "city" : "San Diego", "likes" : [ "desert", "beach" ] }

{ "\_id" : ObjectId("57fd59a2d630a0fd9685a14b"), "firstName" : "Dawn", "lastName" : "Davis", "state" : "WA", "city" : "Seattle", "likes" : [ "forest", "mountains" ] }

You perform the following query:

db.people.deleteMany( { state : "NB" } )

How many oplog entries are created as a result of this query?

**0**

**Detailed Answer**

The *Oplog* collection only contains an entry for a given write query if the operation has modified a document.

Because the deleteMany operation is not deleting any document, there will be no *Oplog* entry to record.

1. What is the principal advantage of having a delayed replica set member?

It allows the load on the secondary servers to be more evenly spread.

It allows you to perform queries against historical versions of the data.

It increases write speed to the primary.

It makes it easier to upgrade the system without downtime.

**It provides a window of time to recover from an operator error.**

**Detailed Answer**

If you have a delayed member in your replica set, for example, a delay of one hour, it will take one hour before changes on the *Primary* are replicated to this member.

If a user were to drop a collection or database on the *Primary*, you would have one hour to go to this delayed member to retrieve the destroyed data.

You can also query older versions of your documents, however, you can't choose a historical version to retrieve as you only get the one that existed one hour ago.

To learn more, visit the following MongoDB Documentation page:

* [Configure a Delayed Replica Set Member](https://docs.mongodb.com/manual/tutorial/configure-a-delayed-replica-set-member/)

1. A replica set is under a moderate write load. A new data bearing member with no data is added to an existing replica set. Which of the following might be fetched by the new member before it can become a secondary? Check all that apply.

**Entries in the primary's oplog**

**BSON documents from the primary's database(s)**

Write queries as received by the primary

**Detailed Answer**

When the new member is added, it will undergo what is called an *initial sync*. During that phase, the *Secondary* will start pulling all the documents from the *Primary*.

In parallel of pulling the documents, the *Secondary* will also pull the *Oplog* entries, which reflect modifications done on those documents, or new documents being inserted. Applying those *Oplog* entries after it fetched all documents will ensure a coherent state of the documents. This guarantee is based on the omnipotence of the operations put in the *Oplog*.

Write queries, as received by the *Primary*, are not guaranteed to be omnipotent, and may need to be transformed. For example, a write operation modifying many documents is not omnipotent and will be transformed to a series of writes, one per modified document.

1. Suppose you have a three-node replica set distributed across two data centers: *dc1* and *dc2*. Which of the configurations below meets the following requirements:
2. Your replica set maintains three copies of the data.
3. Either dc1-01 or dc1-02 may become primary.
4. dc2-01 should never be primary.
5. Clients may read from dc2-01.

**{ "\_id" : "rs0",**

**"version" : 1,**

**"members" : [**

**{ "\_id" : "dc1-01", "host" : "mongodb0.example.net:27017" },**

**{ "\_id" : "dc1-02", "host" : "mongodb1.example.net:27017" },**

**{ "\_id" : "dc2-01", "host" : "mongodb2.example.net:27017",**

**"priority" : 0 } ] }**

**Detailed Answer**

The first requirement eliminates any three-node replica set where one node is an *arbiter*, as *arbiters* do not have a copy of the data. Please note, the three-node replica set recommendation from *MongoDB, Inc* does not include arbiters.

The second and third requirements eliminate the choices that do not specify a *Priority* of **0** for *dc2-01*. As for *dc1-01* and *dc1-02*, they can be assigned any positive integer, or the default value of **1** to be electable as *Primary*.

Finally, the fourth requirement eliminates the configuration where *dc2-01* is listed as hidden, as this will prevent reading from this replica member.

To learn more, visit the following MongoDB Documentation page:

* [Replica Set Configuration](https://docs.mongodb.com/manual/reference/replica-configuration/)

1. Which of the following are true of the *Oplog* entries created as a result of the below write operations in MongoDB?

db.collection.insertMany()

db.collection.updateMany()

db.collection.deleteMany()

Check all that apply.

**A single write query may result in multiple oplog entries.**

**Each oplog entry specifies whether a document is inserted, updated, or deleted.**

Each oplog entry may affect multiple documents.

**Detailed Answer**

When a write operation modifies many documents on a *Primary*, the *Primary* needs to insert a separate entry in the *Oplog* for each modified document. This is the only way the system can ensure the *Oplog* remains omnipotent.

A single command writing to many documents, running on a secondary which fails in the middle of the command, could not be guaranteed to be replayed correctly.

To learn more, visit the following MongoDB Documentation page:

* [db.collection.insertMany()](https://docs.mongodb.com/manual/reference/method/db.collection.insertMany/)
* [db.collection.updateMany()](https://docs.mongodb.com/manual/reference/method/db.collection.updateMany/)
* [db.collection.deleteMany()](https://docs.mongodb.com/manual/reference/method/db.collection.deleteMany/)

1. Which of the following are best practice reasons to read from secondaries?

To increase read throughput when the primary is under heavy write load

**To offload batch processing work from the primary (e.g., data analytics)**

**To provide local reads in geographically distributed replica sets**

**Detailed Answer**

Reading from *Secondaries* may not help when the *Primary* is under a heavy write load because each replica member is doing about the same number of writes. A given write on the *Primary* is also done on a *Secondary* through replication.

*Sharding* is the functionality that would help scale the read operations, as it distributes the writes across many servers in order to reduce the load.

The below two answer choices are considered the main reasons to read from secondary.:

* To offload batch processing work from the primary (e.g., data analytics)
* To provide local reads in geographically distributed replica sets

To learn more, visit the following MongoDB Documentation page:

* [Use Cases for reading from Secondaries](https://docs.mongodb.com/manual/core/read-preference/#use-cases)

### Sharding

1. Which of the following are traits of a hashed \_id as a shard key, relative to an unhashed \_id? Check all that apply.

**Inserts on auto-generated ObjectId’s will be distributed across shards**

**Range queries (to find documents created on a particular week, for example) may be less efficient**

Increased security due to hashing

**Detailed Answer**

*Auto-generated ObjectId's* are monotonically increasing values. Using those as *shard keys* results in one shard getting all the insert operations. Hashing those ObjectId values will create a uniform distribution of values, therefore sending insert operations on all shards in a distributed fashion.

Unfortunately, a range of documents who may have been colocated in the same *chunks*, will now be distributed randomly in all the *chunks*. The consequence is that any range query on a *hashed shard key* needs to be sent to all shards, making those queries less efficient, and also impacting the scalability of the system.

To learn more, visit the following MongoDB Documentation page:

* [Hashed Sharding](https://docs.mongodb.com/manual/core/hashed-sharding/)

1. An insufficiently granular (“low cardinality”) shard key can result in which type of problems? Check all that apply.

**Large chunks that cannot be split.**

More shards getting hit on certain queries.

Limits to document size.

**Detailed Answer**

Documents with the same values for their shard key will be colocated in the same *chunk*. If a lot of documents have the same values, this may result in a very big *chunk*. The system is unable to split this *chunk* as there is no value between the bounds of the *chunk*. For example if a shard key is the name of a country, all documents with *USA* are placed in the same *chunk*, and this *chunk* can't be split, as there is no other value between *USA* and *USA*.

*Chunks* that can not be split are called *jumbo chunks*.

To learn more, visit the following MongoDB Documentation page:

* [Shard Key Cardinality](https://docs.mongodb.com/manual/core/sharding-shard-key/#shard-key-cardinality)

1. Suppose you have a sharded cluster with a status as follows:

mongos> sh.status()

--- Sharding Status ---

sharding version: {

"\_id" : 1,

"version" : 3,

"minCompatibleVersion" : 3,

"currentVersion" : 4,

"clusterId" : ObjectId("51de8630162b88d59cd7b006")

}

shards:

{ "\_id" : "shard0000", "host" : "localhost:30000" }

{ "\_id" : "shard0001", "host" : "localhost:30001" }

{ "\_id" : "shard0002", "host" : "localhost:30002" }

{ "\_id" : "shard0003", "host" : "localhost:30003" }

{ "\_id" : "shard0004", "host" : "localhost:30004" }

{ "\_id" : "shard0005", "host" : "localhost:30005" }

{ "\_id" : "shard0006", "host" : "localhost:30006" }

{ "\_id" : "shard0007", "host" : "localhost:30007" }

databases:

{ "\_id" : "admin", "partitioned" : false, "primary" : "config" }

{ "\_id" : "test", "partitioned" : true, "primary" : "shard0001" }

test.products

shard key: { "productId" : 1 }

chunks:

shard0000 2

shard0002 2

shard0003 1

shard0004 1

shard0005 1

shard0006 1

shard0007 1

shard0001 2

{ "productId" : { "$minKey" : 1 } } -->> { "productId" : 9294 } on : shard0000 { "t" : 2, "i" : 0 }

{ "productId" : 9294 } -->> { "productId" : 18684 } on : shard0002 { "t" : 3, "i" : 0 }

{ "productId" : 18684 } -->> { "productId" : 27851 } on : shard0003 { "t" : 4, "i" : 0 }

{ "productId" : 27851 } -->> { "productId" : 36852 } on : shard0004 { "t" : 5, "i" : 0 }

{ "productId" : 36852 } -->> { "productId" : 46047 } on : shard0005 { "t" : 6, "i" : 0 }

{ "productId" : 46047 } -->> { "productId" : 55450 } on : shard0006 { "t" : 7, "i" : 0 }

{ "productId" : 55450 } -->> { "productId" : 64644 } on : shard0007 { "t" : 8, "i" : 0 }

{ "productId" : 64644 } -->> { "productId" : 73769 } on : shard0000 { "t" : 9, "i" : 0 }

{ "productId" : 73769 } -->> { "productId" : 82950 } on : shard0002 { "t" : 10, "i" : 0 }

{ "productId" : 82950 } -->> { "productId" : 91983 } on : shard0001 { "t" : 10, "i" : 1 }

{ "productId" : 91983 } -->> { "productId" : { "$maxKey" : 1 } } on : shard0001 { "t" : 1, "i" : 10 }

Which shards would be involved in answering the following query:

use test;

db.products.find({"productId" : {"$gte" : 20000, "$lte" : 40000}});

Check all that apply.

**shard0003**

**shard0004**

**shard0005**

**Detailed Answer**

Any chunk that covers the values in the range of 20,000 to 40,000 can be accessed by the find() query. Looking at the provided output for *rs.status()*, we identify the chunks for this range as:

{ "productId" : 18684 } -->> { "productId" : 27851 } on : shard0003 { "t" : 4, "i" : 0 }

{ "productId" : 27851 } -->> { "productId" : 36852 } on : shard0004 { "t" : 5, "i" : 0 }

{ "productId" : 36852 } -->> { "productId" : 46047 } on : shard0005 { "t" : 6, "i" : 0 }

Those three chunks cover all values for 20,000 to 40,000. Because they are on three different shards, the query will have to be routed to those three shards, each shard returning the corresponding documents it has for the range.

To learn more, visit the following MongoDB Documentation page:

* [Insert Documents](https://docs.mongodb.com/manual/tutorial/insert-documents/#insert-id-field)

1. Consider a collection of users with the following fields and possible values:

* phone\_number -- a 10-digit telephone number (string)
* eye\_color -- "brown", "hazel", "blue", "green", or "other" (string)
* weight -- an integer in pounds; known for about half the users
* started\_driving\_at -- the age at which the user got their driver's license in years. For most users this is 15, 16, 17, or 18. (integer)
* \_id -- automatically created on insert (ObjectId)

Assuming the data-access patterns also support your choice, which of these fields would make the best shard key?

**phone\_number**

**Detailed Answer**

Phone number is the best selection here.

With weight, eye\_color, and started\_driving\_at, we run the risk of low *cardinality* and wouldn't get a good distribution.

\_id would not make a good shard key because it isn't something meaningful we could query the database with in normal circumstances, like searching for a customer record when they call into a call center.

To learn more, visit the following MongoDB Documentation pages:

* [Choosing a shard key](https://docs.mongodb.com/manual/core/sharding-shard-key/#choosing-a-shard-key)
* [Blog Post on selecting a shard key](https://www.mongodb.com/blog/post/on-selecting-a-shard-key-for-mongodb)

1. In a sharded collection, which of the following is true of the primary shard?

It processes queries from the application layer and determines the location of this data in the sharded cluster.

It is a shard on which the primary for a replica set is found.

**It holds the unsharded collections for the database.**

It is simply the first shard initialized in a sharded cluster.

It holds configuration information for the sharded cluster.

**Detailed Answer**

For a given database in a cluster, not all collections may be sharded. As a matter of fact, you are likely to shard only the very large collections. For the ease of management and to provide features like $lookup across collections, it makes sense to group all non-sharded collections together, and this location is referred to as the *Primary Shard* for this given database. Other databases in the cluster are likely to have a different *Primary Shard* to level the space and load between the shards.

As a note, the term *Primary* Shard is used here, so be careful not to confuse this notion with the *Primary* replica in a replica set.

To learn more, visit the following MongoDB Documentation page:

* [Primary Shard](https://docs.mongodb.com/manual/core/sharded-cluster-shards/#primary-shard)

1. In which of the following situations can we assume sharding will be an effective strategy?

A single MongoDB instance cannot keep up with your application's write load and you have exhausted other options.

Your data set is too big to fit in a single MongoDB instance.

You would like to improve read performance for your application.

The data set is taking too much time to backup and restore.

**All of the above**

**Detailed Answer**

The common thread in the above answer choices is a bottleneck on resources, or for "taking too much time to backup and restore" a function of your operational requirements. Breaking the dataset over shards will mean that each server will have more resources available to handle the subset of data it owns, and operations of moving data across machines for replication, backups, restores will also be faster

To learn more, visit the following MongoDB Documentation page:

* [Advantages of Sharding](https://docs.mongodb.com/manual/sharding/#advantages-of-sharding)

1. When a chunk is in flight from one shard to another during a migration process, where are reads to that chunk directed?

**To the shard from which it is being migrated**

To both the shard from which it is being migrated and the shard to which it is being migrated

To the shard to which it is being migrated

Documents in the chunk are locked during migration. Reads are queued.

To the config server

**Detailed Answer**

When a chunk is in flight, reads and writes from the application can still access the documents in that chunk. Modifications on documents are propagated to the shard where it is migrated.

Until the chunk is fully migrated, the shard (donor) that is sending it to another shard (receiver) is the only location where the all documents are present in their latest form. For that reason, the *donor* shard is processing the reads.

To learn more, visit the following MongoDB Documentation page:

* [Chunk Migration Procedure](https://docs.mongodb.com/manual/core/sharding-balancer-administration/#chunk-migration-procedure)

### Indexes

1. In a sharded cluster, which of the following indexes must contain only unique values? Check all that apply.

**The \_id index**

The shard key index

A single-field index

**Detailed Answer**

The \_id must be unique in a replica set, as those values are used in the *Oplog* to reference documents to update. This characteristic of uniqueness is enforced by the system.

In a sharded cluster, \_id must also be unique across the sharded collection because documents may migrate to another shard, and identical values would prevent the document to be inserted in the receiver shard, failing the migration of the chunk. It is the responsibility of the application to ensure uniqueness on \_id for a given collection in a sharded cluster if it is not the *shard key*.

Note that if \_id is used as the shard key, the system will automatically enforce the uniqueness of the values, as chunk ranges are assigned to a single shard, and the shard can ensure uniqueness on the values in that range.

As for the shard key index, if it's not \_id, it is perfectly acceptable to have identical values for different documents. However, beware of having too many documents with the same values, as this will lead to *jumbo chunks*.

To learn more, visit the following MongoDB Documentation pages:

* [Unique Indexes in a Sharded Cluster](https://docs.mongodb.com/manual/core/sharding-shard-key/#unique-indexes)
* [Unique Indexes](https://docs.mongodb.com/manual/core/index-unique/)

1. Your top three queries for a collection consist of the following query patterns:

90% db.employees.find( { lastName : 1, firstName : 1, currentEmployee : 1, company : 1 }, { supervisor : 1, teamName : 1, position: 1, duties : 1, salary : 1 } )

5% db.employees.updateOne( { employeeId: 1 }, { $set : { position : 1, teamName : 1, salary : 1, duties : 1 } } )

1% db.employees.updateOne( { employeeId: 1 }, { $set : { currentEmployee : 1, hireDate: 1 } } )

Your system is outgrowing the servers you currently have it running on, and you’d like to shard the collection. Which of the following shard keys will be most performant?

**{ company : 1, lastName: 1 }**

{ supervisor : 1, teamName : 1, position : 1, duties : 1, salary : 1 }

{ employeeId : 1 }

{ currentEmployee : 1, hireDate : 1 }

{ currentEmployee : 1, company : 1 }

**Detailed Answer**

Because the first query account for 90% of your read workload, it should be the one driving the selection of the shard key.

Combination of fields from that query would make the best shard key. Out of the two solutions that use a subset of those fields, the one using company and lastName is preferred to currentEmployee and company, as the currentEmployee field is likely a boolean leading to few values, and potentially a lot of documents with the same values, resulting in jumbo chunks (chunks too big to be splitted).

To learn more, visit the following MongoDB Documentation page:

* [Picking a Shard Key](https://docs.mongodb.com/manual/core/sharding-shard-key/#choosing-a-shard-key)

1. When should you pre-split data for a sharded cluster? Check all that apply.

**When you expect to set and leave the balancer inactive**

**When you are performing a bulk initial load**

When your collection is initially empty and you expect to be only occasionally adding documents

**Detailed Answer**

If you are using a *hashed shard key*, or know the distribution of your data very well such that you can arrange for a distribution of data to be equilibrated between shards, you may want to pre-split the data and not use the *balancer*. Please note the *balancer* is there to help by doing the migrations of chunks when needed.

A common mistake is to insert a lot of data sorted by the *shard key*. For example, migrating from a SQL database with the data sorted by userid, and using the same userid field as your *shard key*. By doing so, you will insert on a single shard (the one with the upper chunk), and then those documents are likely going to migrate to another shard later, doubling the number of writes. Alternatively, if you pre-split the collection across the values of userid, documents will not be written twice. Note that this is still not the optimal way to load the documents. For better performance, have parallel queries inserting each on one shard within the range of documents belonging to the chunks on a given shard.

To learn more, visit the following MongoDB Documentation page:

* [Create Chunks in a Sharded Cluster](https://docs.mongodb.com/manual/tutorial/create-chunks-in-sharded-cluster)

1. In the context of a sharded cluster, which of the following is NOT true of chunks?

Chunks may exist that contain no documents

Chunk ranges are inclusive of the lower boundary and exclusive of the upper boundary

The ranges of two chunks do not overlap

Every chunk is assigned to a particular shard

**The default chunk size changes automatically to meet the needs of an application**

**Detailed Answer**

The *chunk size* is a parameter in the config.settings collection in the sharded cluster. It is a unique value for the sharded cluster. It defaults to 64MB and usually does not need to be changed. To change the value, create or update the document in which this setting is.

To learn more, visit the following MongoDB Documentation page:

* [Tutorial on how to modify the chunk size](https://docs.mongodb.com/manual/tutorial/modify-chunk-size-in-sharded-cluster/)

1. Which of the following are recommended numbers of config servers for a sharded cluster? Check all that apply.

Минимум 3

**3**

**5**

**7**

**Detailed Answer**

The recommendation for the number of config servers is a minimum of 3. However if you have clusters with a lot of data centers, using 5 or 7 is perfectly acceptable. As for the replica set, you should have an odd number of replicas.

To learn more, visit the following MongoDB Documentation page:

* [Replica Set Deployment Architectures](https://docs.mongodb.com/manual/core/replica-set-architectures/)

### Application Administration

1. You are required to create a user for your organization that manages all other users.

Which role would be required for this user? Check all that apply.

**userAdminAnyDatabase**

clusterAdmin

readWriteAnyDatabase

**Detailed Answer**

In order to create a user, you need to be able to perform actions such as createUser and changePassword. These actions are available to the *role* userAdminAnyDatabase.

readWriteAnyDatabase comes close but it does not have the ability to write to the system.users collection. It is used to manage users in the other databases.

To learn more, visit the following MongoDB Documentation page:

* [Built-In Roles](https://docs.mongodb.com/manual/reference/built-in-roles)

1. Which of the following is true about MongoDB's Role-Based Access Control (RBAC) system?

A user can only be granted one role

MongoDB enables role-based access control by default

A role can only be assigned to a user after the user's creation

**A role can inherit privileges from other roles associated with that database**

MongoDB stores role information for each database within that database

**Detailed Answer**

Here is some information about roles, to help you understand why some choices of answers are incorrect.

*role-based access control* is only available once you turn on authorization and have user accounts. You can run the cluster without any user account and privileges, however, it is strongly recommended to enable *authentication* and *authorization*.

While creating a user, you can assign a role or many roles at the creation time. As a matter of fact, this would be the recommended way, as it makes it easier to keep a tight control on all accounts.

MongoDB associates users and their roles to different databases, however all the information about users and roles is kept in the admin database.

To learn more, visit the following MongoDB Documentation page:

* [Role-Based Access Control](https://docs.mongodb.com/manual/core/authorization/)

### Server Administration

1. Which of the following must you do before backing up a running sharded cluster using a file system snapshot?

Stop all the config servers

**Disable the balancer**

Disable journaling

Force an election

Backup secondaries

**Detailed Answer**

One of the requirements for doing a backup of a sharded cluster is to ensure that no group of documents (chunks) are getting migrated by one shard to another shard while you are copying the data for the given shard.

For this reason, you need to ensure the balancer is disabled while you take the file system snapshots.

If you are using *Ops Manager* or *Cloud Manager* for your backups, then those tools will stop the balancer for you, and they will also insert a synchronization token in all shards and in the *config server*, so you can have a consistent backup.

To learn more, visit the following MongoDB Documentation page:

* [Back Up a Sharded Cluster with File System Snapshots](https://docs.mongodb.com/manual/tutorial/backup-sharded-cluster-with-filesystem-snapshots/)

1. You have a three-member replica set. If your secondaries are falling behind, which of the following are plausible causes? Check all that apply.

Application is writing to the secondaries but not the Primary

**Network issues**

**Slower hardware on the secondaries**

**Detailed Answer**

*Application is writing to the secondaries but not the Primary* is clearly wrong because an application can only write to the *Primary*.

*Network issues* may lead to the replication subsystem not being able to quickly get the changes happening on the *Primary* resulting in *replication lag*.

Having faster hardware for the *Primary* can also lead to *replication lag*. Imagine the *Primary* operating at full capacity. While this is happening, the secondaries with slower hardware may not be able to apply all the writes happening on the *Primary* at the same speed.

To learn more, visit the following MongoDB Documentation page:

* [Check the Replication Lag](https://docs.mongodb.com/manual/tutorial/troubleshoot-replica-sets/#check-the-replication-lag)

1. What are the uses of the mongo shell? Check all that apply.

**Allow people to use MongoDB with a simple command line interface.**

**Perform administrative tasks.**

Use as a javascript framework for your production system.

**Detailed Answer**

Compass is the easiest interface to *MongoDB*, however, it does not allow you to perform all the tasks you may need to do. The *shell* lets you execute and perform all the administrative tasks on your servers.

An alternative to using the *shell* to perform the administration tasks is to put your cluster under the automation feature of *Ops Manager* or *Cloud Manager*. Under those tools, you can do most of the administrative tasks from the UI of those tools.

If you want to go further and do as little administrative tasks as possible, consider using *Atlas*, the *Database as a Service*.

To learn more, visit the following MongoDB Documentation page:

* [The mongo Shell](https://docs.mongodb.com/manual/mongo/)

1. What is an indication that your disk speed is causing a performance bottleneck?

**High IO wait times in the CPU stats**

High number of page faults

Resident memory approaches physical memory

If you are using spinning disks, there will always be a bottleneck.

High CPU load

**Detailed Answer**

*IO wait* is the key piece of information. That means the disk is unable to promptly take all the requests sent to it.

SSD are usually faster than spinning disks, however you can have a system performing very well with spinning disks if they are not used at full capacity.

*High number of page faults* and *Resident memory approaches physical memory* are usually symptoms that the system does not have enough physical memory.

To learn more, visit the following MongoDB Documentation page:

* [MongoDB Performance](https://docs.mongodb.com/manual/administration/analyzing-mongodb-performance/)

1. Which of the following is **not** a feature of the WiredTiger storage engine component in MongoDB?

Compression of data files

Index prefix compression

Document level concurrency

**Replication**

A dedicated cache of RAM

**Detailed Answer**

*Replication* is a feature handled at a higher level in the *mongod* process. A storage engine has the mission to store and retrieve documents from cache and memory. *Replication*, *Sharding*, processing of *MongoDB Query Language* queries and more, are all done in higher layers in the *mongod* process.

To learn more, visit the following MongoDB Documentation page:

* [The WiredTiger Storage Engine](https://docs.mongodb.com/manual/core/wiredtiger/)

1. In MongoDB, the WiredTiger storage engine provides concurrency at what level?

Server-level concurrency

Database level concurrency

Collection level concurrency

**Document level concurrency**

Field level concurrency

**Detailed Answer**

The *WiredTiger* storage engine supports document-level concurrency, allowing multiple documents from the same collection to be written to, simultaneously.

To learn more, visit the following MongoDB Documentation page:

* [WiredTiger Concurrency](https://docs.mongodb.com/manual/core/wiredtiger/#document-level-concurrency)

1. Which of the following is true of the file system cache when using WiredTiger as your storage engine? Check all that apply.

**The size of the File System Cache is tunable**

**The File System Cache is used by MongoDB**

The File System Cache can be re-allocated to the WT cache without shutting down the mongod process

**Detailed Answer**

You can set the size of the *File System Cache*, however this is an *Operating System* setting, outside of *MongoDB*. Note that the memory allocated to the *WiredTiger* cache is not available as *File System Cache*. The default values in *MongoDB*should work well for most workloads, so there is rarely a need to change the value of those caches.

If you were to change the size of the *WiredTiger* cache, you would need to shut down the mongod processes.

If there is space available in the *File System Cache*, *MongoDB* will use it indirectly, meaning the *Operating System* will use it to prevent fetching the same data from disk all the time.

To learn more, visit the following MongoDB Documentation page:

* [Memory Use by WiredTiger](https://docs.mongodb.com/manual/core/wiredtiger/#memory-use)

1. Which of the following are compression algorithms available for WiredTiger in MongoDB 4.0? Check all that apply.

**Zlib**

**Snappy**

bzip2

**Detailed Answer**

There is no such algorithm called *bzip2*.

As for the other two compression algorithms, they are used by the *WiredTiger* storage engine.

To learn more, visit the following MongoDB Documentation page:

* [Compression in WiredTiger](https://docs.mongodb.com/manual/core/wiredtiger/#compression)

1. Which of the following are features of zlib compression with WiredTiger? Check all that apply.

**Disk I/O will probably be lower with zlib than without compression.**

zlib will prioritize speed over compression factor.

zlib will not use any cpu cycles for compression/decompression.

**Detailed Answer**

The *zlib* algorithm provides higher compression rates, at the cost of more CPU.

For that reason, it is likely that the compressed data will have a smaller foot print, resulting in less *disk I/O*.

To learn more, visit the following MongoDB Documentation pages:

* [zlib](https://docs.mongodb.com/manual/reference/glossary/#term-zlib)
* [Compression in WiredTiger](https://docs.mongodb.com/manual/core/wiredtiger/#compression)

1. Which of the following commands are correct for exporting data into a CSV file format from a MongoDB server?

mongodump --host localhost:27017 -d catalog -c shoes > shoes.csv

**mongoexport --host localhost:27017 -d catalog -c shoes --type=csv -f fields -o shoes.csv**

mongoexport --host localhost:27017 -d catalog -c shoes -o shoes.csv

**Detailed Answer**

mongodump will export the documents in *BSON*. It is also the preferred way to transfer documents from one instance of *MongoDB* to another instance.

However, if you need to export to a *CSV* file, you would use mongoexport.

The correct answer is the one that includes --type=csv, which tells which format we want mongoexport to use for the output. The default type is *JSON*.

To learn more, visit the following MongoDB Documentation page:

* [mongoexport supported types](https://docs.mongodb.com/manual/reference/program/mongoexport/#cmdoption-mongoexport-type)

1. In MongoDB, which of the following has the option of using the *dbpath* where your database had put its data files, rather than talking to the *mongod* process itself?

Mongorestore

Mongodump

Mongoexport

Mongoimport

**None of these have the options of using the dbpath; you need to connect to a running mongod for all of them.**

**Detailed Answer**

Many versions ago, mongodump and mongoexport could understand the format of the files in the *dbpath*, however to increase security, this feature has been removed from those tools.

As for mongorestore and mongoimport, they are not reading the database files, but providing the documents to a mongod instance, allowing it to create or add to the database files.

To learn more, visit the following MongoDB Documentation pages:

* [mongodump](https://docs.mongodb.com/manual/reference/program/mongodump/)
* [mongoexport](https://docs.mongodb.com/manual/reference/program/mongoexport/)
* [mongorestore](https://docs.mongodb.com/manual/reference/program/mongorestore/)
* [mongoimport](https://docs.mongodb.com/manual/reference/program/mongoimport/)

1. You would like to store a .gif file in MongoDB using GridFS. Which of the following can help you to do this?

Mongodump

**Mongofiles**

Mongoimport

Mongorestore

None of these will work.

**Detailed Answer**

mongofiles is the only tool from the list that operates on *files*.

All other tools operate on *BSON* documents.

To learn more, visit the following MongoDB Documentation page:

* [mongofiles](https://docs.mongodb.com/manual/reference/program/mongofiles/)

1. Which of the following is true regarding the explain() method?

By default, the explain() method executes the query to collect statistics

**The explain() method provides information about which query plan would be selected for a given query**

The explain() method provides information about all of the indexes in a collection

**Detailed Answer**

The default mode for the explain() method is to report which candidate plans would be executed, without doing the execution. The reason to default to this behavior is not to add an additional load on a server unless it is explicitly requested.

The explain() method will give some info about indexes but only for the ones in the considered plans. To get the full information about all the indexes, the recommended method is db.collection.stats() or the *Aggregation Framework* stage $indexStats.

To learn more, visit the following MongoDB Documentation pages:

* [Explain Results](https://docs.mongodb.com/manual/reference/explain-results/)
* [db.collection.stats()](https://docs.mongodb.com/manual/reference/method/db.collection.stats/)
* [$indexStats (aggregation)](https://docs.mongodb.com/manual/reference/operator/aggregation/indexStats/)