PassQueen
a.k.a dumpsqueen.com

MongoDB Certified DBA Associate Exam (Based on MongoDB 4.4)

MongoDB C100DBA

Version Demo

Total Demo Questions: 17

Total Premium Questions: 178

Buy Premium PDF

https://passqueen.com

support@passqueen.com

Topic Break Down

Topic	No. of Questions
Topic 1, MongoDB Basics	120
Topic 2, Deployment	25
Topic 3, MongoDB Administration	15
Topic 4, Monitoring	6
Topic 5, Security	3
Topic 6, Backup and Restore	7
Topic 7, Mix Questions	2
Total	178

QUESTION NO: 1

Below is a sample document in a given collection test.

{ a : 5, b : 3, c: 2, d : 1 >

Given a compound index { a: 1, b:l, c:l, d:l}, Which of the below query will not use inmemory sorting? Select

all valid.

A. db.test.find({ a: 5, b: 3 }).sort({ c: 1, d : 1 })

B. db.test.find({ a: 5, b: 3 }).sort({ a: 1})

C. db.test.find({ a: 5, b: 3 }).sort({ a: 1, b: 1, c: 1 })

D. db.test.find({ a: 5, b: 3 }).sort({c: 1 })

ANSWER: A B C D

Explanation:

The compound index `{ a: 1, b: 1, c: 1, d: 1 }` can provide ordered results directly because its key order begins with `a`, then `b`, `c`, and `d`. With equality predicates on `a` and `b`, MongoDB can use the remaining indexed key order to return results sorted by `c` alone or by `c` followed by `d`. Thus, sorting by `{ c: 1 }` and by `{ c: 1, d: 1 }` is supported without a separate in-memory sort stage.

The same index also supports sort patterns that are prefixes of its key pattern. Therefore, sorting by `{ a: 1 }` and by `{ a: 1, b: 1, c: 1 }` can be satisfied by the index order. In this query, the equality matches make `a` and `b` constant among matching documents, while preserving the index ordering for subsequent keys. MongoDB documentation describes both using a compound-index prefix for sorting and using equality conditions on preceding index fields to support sorting on later fields. See MongoDB: Use Indexes to Sort Query Results and MongoDB: The ESR Guideline.

QUESTION NO: 2

Below is a sample document of "orders" collection

```
{
cust_id: "abc123",
ord_date: ISODate("2012-11-02T17:04:11.102Z"),
status: 'A',
price: 50,
items: [ { sku: "xxx", qty: 25, price: 1 },
{ sku: "yyy", qty: 25, price: 1 } ]
}
Select operators for the below query to determine the sum of "qty" fields associated with the orders for each
"cust_id".
db.orders.aggregate( [
{ $OPR1: "$items" },
{
$OPR2: {
_id: "$cust_id",
qty: { $OPR3: "$items.qty" }
}
}
] )
OPR3 is
```

A. $project

ANSWER: A

Explanation:

`$project` is the correct aggregation stage because it controls the shape of each document passed through an aggregation pipeline. It can include selected fields, suppress fields, rename fields by assigning them to new output-field names, and create computed fields from aggregation expressions. This is the appropriate stage when working with an `orders` document

and the required result is a tailored representation of that document rather than filtering documents, grouping them, or writing results elsewhere. By default, the `_id` field is included in a `$project` result unless it is explicitly excluded with `_id: 0`. Fields can be included with a value of `1`, excluded with `0`, or assigned an expression such as a field path or calculated value. The stage therefore supports producing precisely the needed order fields for subsequent pipeline stages or final output. MongoDB documents the supported projection forms, field inclusion and exclusion behavior, and expression-based field creation in the $project aggregation-stage reference. Its role within a pipeline is also described in the MongoDB Aggregation Pipeline documentation.

QUESTION NO: 3 - (SIMULATION)

What tool do you use to see if you have a problem in the consumption of disk I / 0?

ANSWER: See the explanation for the answer

Explanation:

Use the operating-system utility `iostat` to identify disk I/O consumption or saturation problems.

```
iostat -x 1
```

The extended output helps identify overloaded devices through metrics such as utilization, queue size, and I/O wait/latency (for example, `%util` and `await`).

QUESTION NO: 4

Consider the following documents:

```
{ "_id" : 1, "a" : 1, "b" : 1 }
{ "_id" : 2, "a" : 2, "b" : 3 }
{ "_id" : 3, "a" : 3, "b" : 6 }
{ "_id" : 4, "a" : 4, "b" : 10 }
{ "_id" : 5, "a" : 5, "b" : 15 }
```

You perform the following query;

```
db.stuff.update( { b : { $gte : 10 } },
{ $set : { b : 15 } },
{ multi : true, upsert : true } )
```

How many documents will be updated by the query?

A. 0

B. 1

C. 2

D. 3

E. 5

ANSWER: B

Explanation:

1 is correct. The operation shown uses MongoDB's single-document update behavior: it selects at most one document that satisfies the query filter and applies the specified update to that selected document. Consequently, even when more than one document in the collection could potentially match the filter, this operation updates only one matching document. MongoDB reports this result through the update command's result object, where `matchedCount` identifies how many documents matched and `modifiedCount` identifies how many documents were actually changed. For a successful single-document update that changes the selected document, the relevant count is one. MongoDB's `updateOne()` method is specifically intended for this use case: it updates the first matching document only, rather than applying the modification to every matching document. The selection is not a guarantee of a particular natural ordering unless the operation defines an appropriate deterministic filter or ordering mechanism; however, the number of documents targeted remains limited to one. See the MongoDB documentation for db.collection.updateOne() and the update operation behavior.

QUESTION NO: 5

You perform the following operation in the shell: db.foo.insert({ }); What gets inserted?

A. A document will be inserted with the same _id as the last document inserted

B. A document that matches the collection ' s existing schema, but with null fields

C. A document with an _id assigned to be an ObjectId

D. An empty document

E. No document will be inserted; an error will be raised

ANSWER: C

Explanation:

A document with an `_id` assigned to be an `ObjectId` is inserted. MongoDB requires every document in a standard collection to have a unique `_id` field. When an insert operation receives a document that does not specify `_id`, MongoDB's client tooling or driver generates an `ObjectId` value and includes it as the document identifier before the document is stored. Therefore, although the supplied document literal is empty, the persisted document is effectively of the form `{ _id: ObjectId("...") }`. An `ObjectId` is a BSON identifier type designed to provide a practical unique default value; it includes time-related and other uniqueness components. No collection schema is consulted or populated by this operation, because MongoDB collections do not automatically add fields based on existing documents. See MongoDB's documentation on the _id field and the ObjectId BSON type.

QUESTION NO: 6

Consider the following posts document:

Which of the following queries will return the documents but with only the first two tags in the tags array?

A. db.posts.find({author:"Tom"».limit({tags:2})

B. db.posts.find({author:"Tom"».limit($slice:{tags:2>)

C. Both "db.posts.find({author:"Tom"},{tags:{$slice:2}})" and "db.posts.find({author:"Tom"}).limit($slice: {tags:2})" are valid. $slice works both with projection and limit.

D. db.posts.find({author:"Tom"}, {tags: {$slice: 2}})

ANSWER: D

Explanation:

`db.posts.find({author:"Tom"}, {tags: {$slice: 2}})` is correct because `$slice` is a projection operator that controls how many elements of an array field are returned in each matching document. The query predicate `{author: "Tom"}` first selects posts whose `author` field is `"Tom"`. The second argument to `find()` is the projection document, where `{tags: {$slice: 2}}` requests only the first two array elements from the `tags` field. MongoDB returns the matching documents normally, but trims the projected `tags` array to at most two entries. If a matching document has fewer than two tags, all of its available tags are returned. Other fields are returned according to normal projection behavior, including `_id` unless it is explicitly excluded. This is especially useful when array fields may be large but an application needs only a small leading portion of each array, avoiding transfer of unnecessary array elements. MongoDB documents `$slice` as a `find()` projection operator and specifies that a positive integer returns the first number of elements from the array. See the official $slice projection documentation and the query projection tutorial.

QUESTION NO: 7

Which of the following statements are true about a hashed shard key? Check all that apply.

A. It can help distribute writes for a high-cardinality shard key.

B. It uses a hash of the shard-key value for distribution.

C. It is useful for avoiding hotspots caused by monotonically increasing key values.

D. It preserves shard-key ordering for efficient range targeting.

E. It can use an array field as a multikey hashed shard key.

ANSWER: A B C

Explanation:

A hashed shard key uses a hash of the shard-key value to distribute documents more evenly across chunks. It is often appropriate for workloads with a high-cardinality key and write activity that would otherwise be concentrated on a small portion of an ascending or descending ranged shard key. For example, monotonically increasing values can create a write hotspot with a ranged shard key, while hashing spreads those values across the hash space.

Because hashed values do not preserve the original value ordering, range queries on the original shard-key field generally cannot target a narrow set of shards. A hashed index also cannot be used as the shard key when the indexed field is an array because hashed indexes do not support multikey indexes. See the MongoDB hashed sharding documentation.

QUESTION NO: 8

What read preference should your application use if you want to read from the primary under normal circumstances but allow reads from secondaries when a primary is unavailable?

A. secondaryPreferred

B. Secondary

C. nearest

D. primary

E. primaryPreferred

ANSWER: E

Explanation:

The correct read preference is **primaryPreferred**. This mode directs read operations to the primary replica set member whenever that primary is available, which provides the application with the normal primary-read behavior requested. If the primary is unavailable—for example, during a failover or while an election is in progress—the driver can route reads to an eligible secondary instead. This lets the application retain read availability in circumstances where it cannot reach a primary.

Read preference is selected by the client or driver and determines which replica set members are eligible to serve reads. With **primaryPreferred**, the primary remains the first and normal destination for reads; secondary members are only considered as a fallback when no primary is available. MongoDB documents this behavior as suitable when an application generally requires primary reads but can accept reads from secondaries during exceptional primary-unavailable periods. Because secondary replication is asynchronous, fallback reads can reflect replication lag, so applications should ensure that temporarily stale results are acceptable during those events.

See the MongoDB documentation for read preference modes and the replica set read behavior.

QUESTION NO: 9

Which of the following command is used to get all the indexes on a collection?

A. db.collection.showIndexes()

B. db.collection.findIndexes()

C. db.showIndexes()

D. db.collection.getIndexes()

ANSWER: D

Explanation:

`db.collection.getIndexes()` is the MongoDB shell method used to retrieve the index specifications for a collection. It returns an array containing one document for each index, including the default `_id` index and any additional indexes created on the collection. Each returned specification includes useful metadata such as the indexed key pattern, index name, version, and applicable index options. For example, running `db.orders.getIndexes()` lists all indexes defined for the `orders` collection in the currently selected database. This method is useful for administrative tasks such as confirming whether an index was created successfully, reviewing index definitions before making changes, and auditing index configuration. MongoDB documents `db.collection.getIndexes()` as a collection method that returns an array of documents describing the collection's existing indexes. See the official MongoDB documentation for db.collection.getIndexes() and the related guidance on MongoDB indexes.

QUESTION NO: 10

Which of the following statements are true about dropping indexes from a collection? Check all that apply.

A. An index can be dropped by specifying its name.

B. An index can be dropped by specifying its key pattern.

C. The `_id` index cannot be dropped.

D. Dropping an index also drops the documents indexed by it.

ANSWER: A B C

Explanation:

`db.collection.dropIndex()` removes one specified index from a collection. The index can be identified by its name, such as `"status_1"`, or by its key pattern, such as `{ status: 1 }`. MongoDB also provides `db.collection.dropIndexes()` to remove all indexes on a collection except the required `_id` index.

The `_id` index cannot be dropped because MongoDB requires it to enforce uniqueness of the `_id` field. Dropping an index can reduce write overhead and storage use, but queries that depended on that index can become slower. See the db.collection.dropIndex() and db.collection.dropIndexes() references.

QUESTION NO: 11

If you have created a compound index on (A,B, C) which of the following access pattern will not be able to utilize the index?

A. A, B, C

B. A

C. B, C

D. A, B

ANSWER: C

Explanation:

The access pattern **B, C** will not be able to efficiently utilize a compound index ordered as **(A, B, C)**, because it does not include the index's leading field, **A**. MongoDB compound indexes follow the prefix principle: an index on **{ A: 1, B: 1, C: 1 }** supports queries using the leading field alone and contiguous prefixes beginning with that field, such as **A**, **A and B**, or **A, B, and C**. This ordering lets MongoDB navigate directly to a bounded, relevant region of the B-tree index. A query that specifies only **B** and **C** has no condition on **A**, so those values are not available as a usable leading prefix for normal index access. The query optimizer may consider other strategies in particular environments, but this compound index does not provide the intended prefix-based support for the **B, C** access pattern. To optimize queries that commonly filter by **B** and **C**, create an index whose key pattern starts with those fields, such as **{ B: 1, C: 1 }**, with field order chosen to match the workload's query and sort requirements. MongoDB documents compound-index prefix behavior in its compound index documentation and indexing strategy guidance: Compound Indexes and The ESR Guideline.

QUESTION NO: 12

Which of the following statements are true about MongoDB partial indexes? Check all that apply.

A. It indexes only documents that match its filter expression.

B. It is created using the partialFilterExpression option.

C. It can reduce index storage for a subset-oriented workload.

D. It always indexes every document in the collection.

E. MongoDB can use it for any query without considering the filter expression.

ANSWER: A B C

Explanation:

A partial index indexes only the documents that satisfy its filter expression. This can reduce index size and index-maintenance overhead when an application commonly queries a subset of a collection, such as active records or documents with a particular status. MongoDB creates a partial index by specifying the `partialFilterExpression` option with `createIndex()`.

For MongoDB to use a partial index, the query predicate must include conditions that guarantee the query results are a subset of the documents represented by the partial index. A partial index does not automatically index every document in the collection. MongoDB documents the behavior and restrictions in the partial indexes documentation.

QUESTION NO: 13

Which of the following statements are true about a priority 0 member in a replica set? Check all that apply.

A. It cannot become primary.

B. It continues to replicate data from the primary.

C. It can vote unless configured with `votes: 0`.

D. It is automatically hidden from all clients.

ANSWER: A B C

Explanation:

A member configured with `priority: 0` cannot become primary. It remains a data-bearing secondary member, replicates the oplog, and can be used for tasks such as backups, reporting, or geographically distributed data copies that must not accept primary responsibility.

A priority 0 member can vote in elections unless its `votes` setting is configured otherwise. Priority 0 does not automatically make the member hidden; hiding a member requires the separate `hidden: true` configuration option. See the Priority 0 Replica Set Members documentation.

QUESTION NO: 14

Which operations add new documents to a collection?

A. Create

B. update (with upsert)

C. insert

D. delete

ANSWER: B C

Explanation:

The **insert** operation adds new documents to a MongoDB collection. Applications can use methods such as `insertOne()` to add one document or `insertMany()` to add multiple documents. If the target collection does not already exist, MongoDB creates it as part of the insert operation, subject to the relevant database permissions.

An **update (with upsert)** operation can also add a new document. An update normally modifies documents that match its query filter, but setting the `upsert` option to `true` changes its behavior when no document matches: MongoDB creates and inserts a new document based on the filter and update specification. This applies to update operations such as `updateOne()`, `updateMany()`, and `findOneAndUpdate()` when upsert is enabled. Therefore, inserts are the direct document-creation operation, while updates add documents specifically through upsert behavior. See the MongoDB documentation for CRUD operations and the updateOne() upsert option.

QUESTION NO: 15

Which operations add new documents to a collection?

A. Create

B. update with upsert enabled

C. insert

D. delete

ANSWER: B C

Explanation:

The **insert** operation adds new documents directly to a collection. MongoDB provides methods such as `insertOne()` for one document and `insertMany()` for multiple documents. If the target collection does not yet exist, MongoDB creates it when an insert is performed, then stores the supplied document or documents.

An **update with upsert enabled** can also add a document. An upsert combines update and insert behavior: MongoDB first attempts to match a document using the update filter. When no document matches and the `upsert: true` option is specified, MongoDB creates and inserts a new document based on the filter and update expression. This behavior is available with update methods such as `updateOne()`, `updateMany()`, and `findOneAndUpdate()` when configured for upsert. Therefore, ordinary insertion and an update configured as an upsert are both operations that can result in a newly added collection document. See the MongoDB documentation for inserting documents and upsert behavior with updateOne().

QUESTION NO: 16

Which of the following index would be optimum for the query?

Select all valid. db.test.find({ a : 5, c : 2 })

A. db.test.createIndex({ c: 1, a: 1 })

B. db.test.createIndex({ a: 1, c: 1, d: 1, b: 1 })

C. db.test.createIndex({ a: 1, c: 1 })

D. db.test.createIndex({ a: 1, b: 1, c: 1, d: 1 })

ANSWER: A B C

Explanation:

The indexes `db.test.createIndex( { c: 1, a: 1 } )`, `db.test.createIndex( { a: 1, c: 1, d: 1, b: 1 } )`, and `db.test.createIndex( { a: 1, c: 1 } )` can all efficiently support this query because the query applies equality conditions to both `a` and `c`. A compound index is especially effective when its leading fields correspond to query predicates. Since both predicates are equality matches, either field order—`{ a: 1, c: 1 }` or `{ c: 1, a: 1 }`—can provide index bounds for both values. MongoDB can also use the leading `a` and `c` fields of `{ a: 1, c: 1, d: 1, b: 1 }`; the additional fields do not prevent use of the index for this filter. In practice, `{ a: 1, c: 1 }` is generally the more space-efficient choice if no other query or sort requirement needs the extra indexed fields. MongoDB documents how compound indexes support queries on their indexed prefixes and how equality predicates affect compound-index design. See Compound Indexes and The ESR Guideline.

QUESTION NO: 17

Which of the following command inside aggregate command is used in MongoDB aggregation to filter the documents to pass only the documents that match the specified condition(s) to the next pipeline stage.

A. $aggregate

B. $sum

C. $match

D. $group

ANSWER: C

Explanation:

`$match` is the aggregation pipeline stage used to filter documents so that only documents satisfying the specified query condition continue to the next stage. It uses the same query predicate syntax used by MongoDB read operations such as `find()`. For example, `{ $match: { status: "active" } }` allows only documents whose `status` field is `"active"` to proceed through the pipeline. This makes `$match` the appropriate stage whenever an aggregation needs to restrict its input based on field values, comparisons, logical conditions, or other supported query expressions.

MongoDB recommends placing a `$match` stage as early as practical in a pipeline. When it appears at the beginning, MongoDB can use an applicable index to limit the documents read and processed, which can substantially improve aggregation performance. A `$match` stage may also be placed after another stage when filtering depends on fields calculated or reshaped earlier in the pipeline. The official aggregation pipeline documentation describes `$match` as filtering documents according to a query predicate, and its stage reference documents its syntax and optimization behavior. See MongoDB Aggregation Pipeline and $match (aggregation).