Introduction to Mongo DB

MongoDB offers both a *Community* and an *Enterprise* version of the database:

- MongoDB Community is the source available and free to use edition of MongoDB.
- MongoDB Enterprise is available as part of the MongoDB Enterprise Advanced subscription and includes comprehensive support for your MongoDB deployment. MongoDB Enterprise also adds enterprise-focused features such as LDAP and Kerberos support, on-disk encryption, and auditing.

Document Database

A record in MongoDB is a document, which is a data structure composed of field and value pairs. MongoDB documents are similar to JSON objects. The values of fields may include other documents, arrays, and arrays of documents.

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.

Collections/Views/On-Demand Materialized Views

MongoDB stores documents in collections. Collections are analogous to tables in relational databases

Key Features

High Performance

MongoDB provides high performance data persistence. In particular,

- Support for embedded data models reduces I/O activity on database system.
- Indexes support faster queries and can include keys from embedded documents and arrays.

Rich Query Language

MongoDB supports a rich query language to support read and write operations (CRUD)

MONGO DB Commands

1. CREATE DATABASE:

SYNTAX: use DATABASE_Name

assume name of database: "myDB"

```
use myDB;
```

Output: switched to db myDB

To confirm the existence of your databse, type command at the MongoDB shell:

```
db
```

Output: myDB

To get a list of all databses, type the below command:

```
show dbs
```

Output:

```
admin(empty)
local 0.078GB
test  0.078GB
```

2. DROP DATABASE:

SYNTAX: db.dropDatabase();

To drop the database, "myDB", first ensure that you are currently placed in "myDB" database and then use db.dropDatabase() command to drop the database.

```
use myDB;
db.dropDatabase();
```

If no databse is selected, the default databse "test" is dropped.

QUERY 1: TO REPORT THE NAME OF THE CURRENT DATABASE

SOLUTION:

```
db
```

Output: test

Query 2: To display the list of databases:

SOLUTION:

```
show dbs
```

Output:

```
admin(empty)
local 0.078GB
test  0.078GB
```

QUERY 3: TO SWITCH TO A NEW DATABASE, FOR EXAMPLE: myDB1:

SOLUTION:

```
use myDB1
```

Output: switched to db myDB1

Query 4: To display the list of collections (tables) in the current databse:

SOLUTION:

```
show collections
```

Output:

```
system.indexes
system.js
```

Query 5: To display the current version of MongoDB Server:

SOLUTION:

```
db.version()
```

Output: 2.6.1

DATA TYPES IN MONGODB

String---- Must be UTF-8 Valid

Integer---- Can be 32 bit 0r 64 bit (depends on the server)

Boolean--- To store true/false values.

Double--- To store floating points (real values)

Min/Max Keys---- To compare a value against the lowest or highest BSON elements.

Arrays---- To store arrays or list or multiple values in a key

Timestamp---- To record when a document has been modified or added.

Null---- To store a null value, A null is missing or unknown value.

Date---- To store the current date or time i Unix time fromat. One can create object of date and

and pas days, month and year

Object ID--- To store the document's ID

Binary Data--- To store binary data (images, binaries etc.)

Code--- To store javascript code into the document.

Regular expression--- To store regular expression.

Note:

UTF-8 can represent any character in the Unicode Standard. UTF-8 is the preferred encoding for email and web pages.

Bson: Binary JSON

Query: Type in db.help() in the mongoDB client to get a list of commands

```
db.help();
```

MONGO DB QUERY LANGUAGE:

CRUD--> CREATE READ UPDATE AND DELETE OPERATIONS IN MONGO DB

CREATE: Creation of data is done using insert() or update() or save() method

Read: Reading the data is performed using find() method.

Update: using update() method

Delete: using remove() method

Qurey: To create a collection by the name "Person"?

Solution:

```
db.createCollection("Person");
```

Output: {"ok" :1}

Query" to show collections?

Solution:

```
show collections;
```

Query: To drop a collection by the name "food".

Solution:

```
show collections;
```

Output:

```
Person
Students
food
system.indexes
system.js
```

Now to drop type command

```
db.food.drop();
```

Output: true

Now to confirm type commad:

```
show collections;
```

Output:

```
Person
Student
system.indexes
system.js
```

Insert Method:

Syntax:

```
db.student.insert(        ---------Collection
{
RollNo: 101;                ---------Field:Value
Age: 19;                    ---------Field:Value
ContactNo:0123456789;       ---------Field:Value
}
)
```

Query:Create a collection by the name "Students" and store the following data in it:

db.Students.insert({_id:1,StudName:"Michelle Jacintha:, Grade:"VII", Hobbies:"Internet Surfing"});

Solution:

```
show collections;
```

Output:

```
system.indexs
system.js
```

Now to insert type command:

```
db.Students.insert(
{
_id:1;
StudName:"Michelle Jacintha;
Grade:"VII";
Hobbies:"Internet Surfing";
}
)
```

Query: Check if the collection has been successfully created

```
show collections;
```

Output:

```
Students
system.indexs
sytem.js
```

Read:

Query: Check if the Student "Michelle Jacintha" has been successfully inserted into "Students" collection.

Solution:

```
db.Students.find();
```

```
Output: {_id:1,StudName:"Michelle Jacintha", Grade:"VII",

Hobbies:"Internet Surfing"}
```

To format the result, one can add pretty() method to the operation:

```
db.Students.find().pretty();
```

```
Output:

{

   _id:1,

   StudName:"Michelle Jacintha",

   Grade:"VII",

   Hobbies:"Internet Surfing"

}
```

Objective: Insert another document into collection.

Step1: ChecK the document in "Students" collection before proceeding.

Solution:

```
db.Students.find.pretty();
```

```
Output: {

   _id:1,

   StudName:"Michelle Jacintha",

   Grade:"VII",

   Hobbies:"Internet Surfing"

}
```

Step 2: Insert another document in same collection

Solution:

```
db. Students.insert({_id:2, StudName:"Mabel Mathews", Grade: "VII", Hobbies:"Baseball"});
```

```
Output: writeResult({"nInserted:1})
```

Step3: using pretty command to find document is successfully inserted

```
db.Students.find().pretty();
```

Output:

```
{
   _id:1,
   StudName:"Michelle Jacintha",
   Grade:"VII",
   Hobbies:"Internet Surfing"
}
{
   _id:2,
   StudName:"Mabel Mathews",
   Grade: "VII",
   Hobbies:"Baseball"
}
```

Query: Insert the document for "Aryan David" into the students collection only if it does not exist in the collection. However, if it is already present in the collection, then update the document with new values. (Update his hobbies from "Skating" to "Chess".). Use "Update else insert" (if there is an existing document, it will attempt to update it, if there is no existing document then it will insert it).

Solution:

Step1: Check the document in the "Students" collection before proceeding.

```
db.Students.find().pretty;
```

Output:

```
{
   _id:1,
   StudName:"Michelle Jacintha",
   Grade:"VII",

   Hobbies:"Internet Surfing"

}
```

Step 2:

```
db.Students.update({_id:3, StudName:"Aryan David", Garde: "VII"},
{$set:{Hobbies:"Skating"}},{upsert:true});
```

Output: writeResult({nMatched" :0, "nUpserted" :1, "nModified":0, "_id":3})

Step 3: Confirm the presence of the document of "Aryan David" in the

"Students" Collection.

Solution:

```
db.Students.find({_id:3});
```

Objective: To demonstrate Save method to insert a document for student “Vamsi Bapat” in the “Students” collection. Omit providing value for the _id key.

Solution:

Step 1: Check the documents in the “Students” collection before proceeding.

```
db.Students.find().pretty();
```

Output:

```
{

        “_id”:1,

        “StudName” : “Michelle Jacintha”,

        “Grade”: “VII”,

        “Hobbies”: “Internet Surfing”

}

{

        “_id”:2,

        “StudeName” : “Mabel Mathews”,

        “Grade”: “VII”,

        “Hobbies”: “Baseball”
}
{
        "_id":3,
        "StudeName" : "Aryan David",
        "Grade": "VII",
        "Hobbies": "Chess"
}
>
```

Step 2:

```
db.Students.save({StudName:"Vamsi Bapat", Grade: "VII"})
```

Output: WriteResult({ "nInserted":1})

Step 3: Confirm the presence of the document of "Vamsi Bapat" in the "Students" collection

```
db.Students.find().pretty;
```

Output:

```
 {
        "_id":1,
        "StudName" : "Michelle Jacintha",
        "Grade": "VII",
        "Hobbies": "Internet Surfing"
}
{
        "_id":2,
        "StudName" : "Mabel Mathews",
        "Grade": "VII",
        "Hobbies": "Baseball"
}
{
        "_id":3,
        "StudName" : "Aryan David",
            "Grade": "VII",
            "Hobbies": "Chess"
}
{           "_id" :ObjectId("546dd0e0a799bb94d"),
            "StudName" :"Vamsi Bapat",
            "Grade" :"VII"
}
>
```

ADDIND A NEW FIELD TO AN EXISTING DOCUMENT-UPDATE METHOD

Syntax:

```
db.students.update(        ------------------------------Collection
{Age:{$gt 18}},            --------------------------Update Criteria
{$set: {Status: "A"}}      --------------------------Update Action
{multi:true}               ----------------------------Update Option
}
```

Query: To add a new field "Location" with Value "Newark" to the document (_id:4) of "Students" collection.

Solution:

Check the document (_id:4) in the "Students" collection before proceeding.

```
db.Students.find({_id:4}).pretty();
```

Output:

```
{
            "_id":4,
            "Grade":"VII",
            "StudName": Herris Gibbs",
            "Hobbies": "Graffiti"
}
```

Step2:

```
db.Students.update({_id:4}, {$set:{Location:"Newark"}});
```

Output: WriteResult ({ “nMatched”:0, “nModified”:1})

Step3: Confirm that new field “Loaction” has been added to document(_id:4)

db.Students.find({_id:4}).pretty();

Output:

```
{
        “_id”:4,
        “Grade”:”VII”,
        “StudName”: “Hersh Gibbs”,
        “Hobbies” : “Graffiti”,
        “Location” : “Newark”
}
```

REMOVING AN EXISTING FIELS FROM AN EXISTING DOCUMENT-REMOVE METHOD

Syntax:

```
db.Students.remove(            ---------Collection
{Age: {$dt 18}},               -----------Remove Criteria
)
```

Query: To remove the fiels “Location” with “Newark” in the document(_id:4) of “Students” collection.

Solution:

Step1: Check the document (_id:4)

db.Students.find({_id:4}).pretty();

Output:

```
{
        “_id”:4,
        “Grade”:”VII”,
        “StudName”: “Hersh Gibbs”,
        “Hobbies” : “Graffiti”,
        “Location” : “Newark”
}
```

Stpe 2:

```
db.Students.update({_id:4}, {$unset:{Location: “Newark”}});
```

Output:

```
{
        “_id”:4,
        “Grade”:”VII”,
        “StudName”: “Hersh Gibbs”,
        “Hobbies” : “Graffiti”,
}
```

Or

```
db.Students.remove({_id:4}, {$set:{Location: “Newark”}});
```

Query: To display only the StudName and Grade from all the documents of the Students collection. The identifier _id should be suppressed and NOT displayed.

Solution:

```
db.Students.find({}, {StudName:1, Grade:1, id:0});
```

Output:

```
{“StudName” :”Michelle Jacintha, “Grade” : “VII”}
{“StudName” :”Aaryan David, “Grade” : “VII”}
```

Query: To find those documents from the Students Collection where the Hobbies is set to either “Chess” OR is set to “Skating”.

Solution:

```
db.Students.find({Hobbies:{$in:[“Chess”,”Skating”]}}).pretty();
```

Query: To find those documents from the Students Collection where the Hobbies is neither set to “Chess” nor is set to “Skating”.

Solution:

```
db.Students.find({Hobbies:{$nin:[“Chess”,”Skating”]}}).pretty();
```

Query: To find those documents from the Students Collection where the Hobbies is set to “Graffiti” and the StudName is set to “Hersh Gibbs” (AND Condition)

Solution:

db.Students.find({Hobbies:"Graffiti", StudName:"Hersch Gibbs"}).pretty();

Query: To find the documents from the Students Collection where the StudName begins with "M"

Solution:

db.Students.find({StudName:/^M/}).pretty();

or

db.Students.find({StudName:{$regex:"^M"}}).pretty();

Query: To find the documents from the Students Collection where the StudName ends in "s"

Solution:

db.Students.find({StudName:/s$/}).pretty();

or

db.Students.find({StudName:{$regex""a$"}}).pretty();

Query: To find the documents from the Students Collection where the StudName has an "e" in any position.

Solution:

db.Students.find({StudName:/e/}).pretty();

or

db.Students.find({StudName:/.*e.*/}).pretty();

or

db.Students.find({StudName:{$regex:"e"}}).pretty();

Relational operators on MongoDB:

$eq → equal to

$ne → not equal to

$gte → greater than or equal to

$lte → less than or equal to

$gt → greater than

$lt →less than

COUNT, LIMIT, SORT AND SKIP

QUERY: TO FIND THE NUMBER OF DOCUMENTS IN THE STUDENTS COOLECTION

Solution:

```
db.Students.count()
```

QUERY : TO RETRIEVE THE FIRST 3 DOCUMENTS FROM THE STUDENTS COLLECTION WHEREIN THE GRADE IS VII

SOLUTION:

```
db.Students.find({Grade: “VII”}).limit(3).pretty();
```

QUERY: TO SORT THE DOCUMENTS FROM THE STUDENTS COLLECTION IN THE ASCENDING ORDER OF STUDNAME.

SOLUTION:

In ascending order:

```
db.Students.find().sort({StudName:1}).pretty();
```

In descending order:

```
db.Students.find().sort({StudName:-1}).pretty();
```

Query: TO skip the first 2 documents from the students collection:

Solution:

```
db.Students().skip(2).pretty();
```

Query: To sort the documents from the Students Collection and skip the first document from the output.

Solution:

```
db.Students.find().skip(1).pretty().sort({StudName:1});
```

Query: To display the last 2 records from the Students Collection

Soution:

```
db.Students.find().pretty().skip(db.Students.count()-2);
```

Query: To retrieve the third, fourth and fifth document from the Students Collection:

Solution:

```
db.Students.find().pretty().skip(2).limit(3);
```

DEALING WITH NULL VALUES

A null value is a missing or unknown value. When we place NULL as a value for a field, it implies that currently we don’t know the value or the value is missing. We can always update the value once we know it.

Before we execute the coomans the update documents with a null value in a column, let us first view 2 documents:

```
db.Students.find({$or:[{_id:3},{_id:4}]})
```

output:

```
{“_id”:3, “Grade”:”VII”, “StudName”:”Aryan David”, “Hobbies”:”Chess”}

{“_id”:4, “Grade”:”VII”, “StudName”:”Hersh Gibbs”, “Hobbies”:”Grafitti”}
```

Now update the documents with NULL value in the “Location” column.

```
db.Students.update({_id:3}, {$set:{Location:null}});

db.Students.update({_id:4}, {$set:{Location:null}});
```

Query: To search for NULL value in “Location” column

```
db.Students.find({Location:{$eq:null}});
```

ARRAYS:

Query: To create a collection by the name “food” and then insert documents into the “food” collection. Each document should have “fruit”array.

Solution:

```
db.food.insert({_id:1,fruits:[“banana”,”apple”,”cherry”]});
```

Output: writeResult({ “nInserted”:1})

```
 db.food.insert({_id:2,fruits:[“orange”,”butterfruit”,”mango”]});
```

Output: writeResult({ “nInserted”:1})

```
db.food.insert({_id:3,fruits:[“pineapple”,”strawberry”,”grapes”]});
```

Output: writeResult({ “nInserted”:1})

```
db.food.insert({_id:4,fruits:[“banana”,”strawberry”,”grapes”]});
```

Output: writeResult({ “nInserted”:1})

```
db.food.insert({_id:5,fruits:[“orange”,”grapes”]});
```

Output: writeResult({ “nInserted”:1})

Query: To find those documents from the “food” collection which has the “fruit array” constituted of “banana”, “apple” and “cherry”.

Solution:

db.food.find({fruits:[“banana”,”apple”,”cherry”]}).pretty();

Query: To find those documents from the “food” collection which have the “fruits” array having “grapes” in the first index position. The index position begins at 0.

Solution:

db.food.find({“fruits.1”:”grapes”});

Query: To find those documents from the “food” collection where the size of the array is two. The size implies that the array holds only 2 values.

Solution:

db.food.find({“fruits”:{$size:2}});

Query: To find those documents with (_id:1) from the “food” collection and display the first two elements from the array “fruits”.

Solution:

db.food.find({_id:1},{“fruits”:{$slice:2}});

UPDATE ON THE ARRAY:

Query: To update the document with “_id:4” and replace the element present in the 1st index position of the “fruits”array with “apple”

Solution:

db.food.update({_id:4},{$set:{“fruits.1”:”apple”}})

Query: To update the documents with “_id:2” and push new key value pairs in the “fruits” array.

Solution:

db.food.update({_id:2}, {$push:{price:{orange:60,butterfruit:200,mango:120}}})

Query: To update the documents with “_id:4” by adding an elememt “orange” to the list of elements in the array “fruits”.

Solution:

db.food.update({_id:4},{$addToSet:{fruits:”orange”}});

Query: To update the document with “_id:4” by popping an element from the list of elements present in the array “fruits”. The element popped is the one from the end of the array.

Solution:

db.food.update({_id:4},{$pop:{fruits:1}});

Query: To update the document with “_id:4” by popping an element from the list of elements present in the array “fruits”. The element popped is the one from the beginning of the array.

Solution:

db.food.update({_id:4},{$pop:{fruits:-1}});

Query: To update the document with “_id:4” by popping an element from the list of elements present in the array “fruits”. The element popped are “pineapple” and “grapes”.

Solution:

db.food.update({_id:4},{$pullAll:{fruits:[“pineapple”,”grapes”]}});

Indexes

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.

Indexes are special data structures that store a small portion of the collection's data set in an easy to traverse form. The index stores the value of a specific field or set of fields, ordered by the value of the field. The ordering of the index entries supports efficient equality matches and range-based query operations. In addition, MongoDB can return sorted results by using the ordering in the index.

The following diagram illustrates a query that selects and orders the matching documents using an index:

click to enlarge

Fundamentally, indexes in MongoDB are similar to indexes in other database systems. MongoDB defines indexes at the collection level and supports indexes on any field or sub-field of the documents in a MongoDB collection.

Default _id Index

MongoDB creates a unique index on the _id field during the creation of a collection. The _id index prevents clients from inserting two documents with the same value for the _id field. You cannot drop this index on the _id field.

NOTE

In sharded clusters, if you do *not* use the _id field as the shard key, then your application must ensure the uniqueness of the values in the _id field to prevent errors. This is most-often done by using a standard auto-generated ObjectId.

Create an Index

To create an index in the Mongo Shell, use db.collection.createIndex().

```
db.collection.createIndex( <key and index type specification>, <options> )
```

The following example creates a single key descending index on the name field:

```
db.collection.createIndex( { name: -1 } )
```

The db.collection.createIndex() method only creates an index if an index of the same specification does not already exist.

[1] MongoDB indexes use a B-tree data structure.

Index Names

The default name for an index is the concatenation of the indexed keys and each key's direction in the index (i.e. 1 or -1) using underscores as a separator. For example, an index created on { item : 1, quantity: -1 } has the name item_1_quantity_-1.

You can create indexes with a custom name, such as one that is more human-readable than the default. For example, consider an application that frequently queries the products collection to populate data on existing inventory. The following createIndex() method creates an index
on item and quantity named query for inventory:

```
db.products.createIndex(
{ item: 1, quantity: -1 } ,
{ name: "query for inventory" }
)
```

You can view index names using the db.collection.getIndexes() method. You cannot rename an index once created. Instead, you must drop and re-create the index with a new name.

Index Types

MongoDB provides a number of different index types to support specific types of data and queries.

Single Field

In addition to the MongoDB-defined _id index, MongoDB supports the creation of user-defined ascending/descending indexes on a single field of a document.

For a single-field index and sort operations, the sort order (i.e. ascending or descending) of the index key does not matter because MongoDB can traverse the index in either direction.

See Single Field Indexes and Sort with a Single Field Index for more information on single-field indexes.

Compound Index

MongoDB also supports user-defined indexes on multiple fields, i.e. compound indexes.

The order of fields listed in a compound index has significance. For instance, if a compound index consists of { userid: 1, score: -1 }, the index sorts first by userid and then, within each userid value, sorts by score.

For compound indexes and sort operations, the sort order (i.e. ascending or descending) of the index keys can determine whether the index can support a sort operation. See Sort Order for more information on the impact of index order on results in compound indexes.

See Compound Indexes and Sort on Multiple Fields for more information on compound indexes.

Multikey Index

MongoDB uses multikey indexes to index the content stored in arrays. If you index a field that holds an array value, MongoDB creates separate index entries for *every* element of the array. These multikey indexes allow queries to select documents that contain arrays by matching on element or elements of the arrays. MongoDB automatically determines whether to create a multikey index if the indexed field contains an array value; you do not need to explicitly specify the multikey type.

See Multikey Indexes and Multikey Index Bounds for more information on multikey indexes.

Geospatial Index

To support efficient queries of geospatial coordinate data, MongoDB provides two special indexes: 2d indexes that uses planar geometry when returning results and 2dsphere indexes that use spherical geometry to return results.

See 2d Index Internals for a high level introduction to geospatial indexes.

Text Indexes

MongoDB provides a text index type that supports searching for string content in a collection. These text indexes do not store language-specific *stop* words (e.g. "the", "a", "or") and *stem* the words in a collection to only store root words.

See Text Indexes for more information on text indexes and search.

Hashed Indexes

To support hash based sharding, MongoDB provides a hashed index type, which indexes the hash of the value of a field. These indexes have a more random distribution of values along their range, but *only* support equality matches and cannot support range-based queries.

Index Properties

Unique Indexes

The unique property for an index causes MongoDB to reject duplicate values for the indexed field. Other than the unique constraint, unique indexes are functionally interchangeable with other MongoDB indexes.

Partial Indexes

New in version 3.2.

Partial indexes only index the documents in a collection that meet a specified filter expression. By indexing a subset of the documents in a collection, partial indexes have lower storage requirements and reduced performance costs for index creation and maintenance.

Partial indexes offer a superset of the functionality of sparse indexes and should be preferred over sparse indexes.

Sparse Indexes

The sparse property of an index ensures that the index only contain entries for documents that have the indexed field. The index skips documents that *do not* have the indexed field.

You can combine the sparse index option with the unique index option to prevent inserting documents that have duplicate values for the indexed field(s) and skip indexing documents that lack the indexed field(s).

TTL Indexes

TTL indexes are special indexes that MongoDB can use to automatically remove documents from a collection after a certain amount of time. This is ideal for certain types of information like machine generated event data, logs, and session information that only need to persist in a database for a finite amount of time.

See: Expire Data from Collections by Setting TTL for implementation instructions.

Hidden Indexes

New in version 4.4.

Hidden indexes are not visible to the query planner and cannot be used to support a query.

By hiding an index from the planner, users can evaluate the potential impact of dropping an index without actually dropping the index. If the impact is negative, the user can unhide the index instead of having to recreate a dropped index. And because indexes are fully maintained while hidden, the indexes are immediately available for use once unhidden.

Except for the _id index, you can hide any indexes.

Capped Collections

Overview

Capped collections are fixed-size collections that support high-throughput operations that insert and retrieve documents based on insertion order. Capped collections work in a way similar to circular buffers: once a collection fills its allocated space, it makes room for new documents by overwriting the oldest documents in the collection.

See createCollection() or create for more information on creating capped collections.

TIP

As an alternative to capped collections, consider MongoDB's TTL (Time To Live) indexes. As these indexes allow you to expire and remove data from normal collections based on the value of a date-typed field and a TTL value for the index.

TTL indexes are not compatible with capped collections.

Behavior

Insertion Order

Capped collections guarantee preservation of the insertion order. As a result, queries do not need an index to return documents in insertion order. Without this indexing overhead, capped collections can support higher insertion throughput.

Automatic Removal of Oldest Documents

To make room for new documents, capped collections automatically remove the oldest documents in the collection without requiring scripts or explicit remove operations.

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. Furthermore, the built-in *first-in-first-out* property maintains the order of events, while managing storage use.
- Cache small amounts of data in a capped collections. Since caches are read rather than write heavy, you would either need to ensure that this collection *always* remains in the working set (i.e. in RAM) *or* accept some write penalty for the required index or indexes.

For example, the oplog.rs collection that stores a log of the operations in a replica set uses a capped collection. Starting in MongoDB 4.0, unlike other capped collections, the oplog can grow past its configured size limit to avoid deleting the majority commit point.

_id Index

Capped collections have an _id field and an index on the _id field by default.

Restrictions and Recommendations

Updates

If you plan to update documents in a capped collection, create an index so that these update operations do not require a collection scan.

Document Size

Changed in version 3.2.

If an update or a replacement operation changes the document size, the operation will fail.

Document Deletion

You cannot delete documents from a capped collection. To remove all documents from a collection, use the drop() method to drop the collection and recreate the capped collection.

Sharding

You cannot shard a capped collection.

Query Efficiency

Use natural ordering to retrieve the most recently inserted elements from the collection efficiently. This is similar to using the tail command on a log file.

Aggregation $out

The aggregation pipeline stage $out cannot write results to a capped collection.

Transactions

Starting in MongoDB 4.2, you cannot write to capped collections in transactions. Reads
from capped collections are still supported in transactions.

Procedures

Create a Capped Collection

You must create capped collections explicitly using the db.createCollection() method, which is a helper in the mongo shell for the create command. When creating a capped collection you must specify the maximum size of the collection in bytes, which MongoDB will pre-allocate for the collection. The size of the capped collection includes a small amount of space for internal overhead.

```
db.createCollection( "log", { capped: true, size: 100000 } )
```

If the size field is less than or equal to 4096, then the collection will have a cap of 4096 bytes. Otherwise, MongoDB will raise the provided size to make it an integer multiple of 256.

Additionally, you may also specify a maximum number of documents for the collection using
the max field as in the following document:

```
db.createCollection("log", { capped : true, size : 5242880, max : 5000 } )
```

IMPORTANT

The size argument is *always* required, even when you specify max number of documents. MongoDB will remove older documents if a collection reaches the maximum size limit before it reaches the maximum document count.

TIP

See:

db.createCollection() and create.

Query a Capped Collection

If you perform a find() on a capped collection with no ordering specified, MongoDB guarantees that the ordering of results is the same as the insertion order.

To retrieve documents in reverse insertion order, issue find() along with the sort() method with
the $natural parameter set to -1, as shown in the following example:

```
db.cappedCollection.find().sort( { $natural: -1 } )
```

Check if a Collection is Capped

Use the isCapped() method to determine if a collection is capped, as follows:

```
db.collection.isCapped()
```

Convert a Collection to Capped

You can convert a non-capped collection to a capped collection with the convertToCapped command:

```
db.runCommand({"convertToCapped": "mycoll", size: 100000});
```

The size parameter specifies the size of the capped collection in bytes.

This holds a database exclusive lock for the duration of the operation. Other operations which lock the same database will be blocked until the operation completes. See What locks are taken by some common client operations? for operations that lock the database.

Tailable Cursor

You can use a tailable cursor with capped collections. Similar to the Unix tail -f command, the tailable cursor "tails" the end of a capped collection. As new documents are inserted into the capped collection, you can use the tailable cursor to continue retrieving documents.