Why mongo Db?

Open source document oriented database

High performance database highly available.

It is horizontally scalable. We can add a server to increase its capacity instead of vertically scalable which is buying a bigger server.

Where to use?

1. High demanding application (real-time application)
2. Diverse ,mixed set of data
3. Massive concurrency
4. Application which are globally deployed in multiple sites
5. no down time is tolerated
6. Able to grow with user needs should not be stuck with data models should be able to evolve as per the market need.
7. High uncertainty in sizing (not sure how much the system will grow) needs fast scaling.
8. Where ever we need seamless and consistence experience
9. High performance while read, write and update happens at the same time.
10. For Real-time analytics mongo Db is the best where Hadoop is good doing analytics on existing data (batch analysis).
11. Straight forward replication( we can create replicas of mongo db in very less time)
12. Scaling on demand (we can add or remove mongo servers on demand)
13. Location based deployment ( singe cluster can expand over entire globe)
14. Geo spatial queries (location based queries like nearest restaurant , within a circle)
15. High Availability and auto failover(if there is any failure of any datacenter then it will be automatically failover to one of the replicas so there will be no down time )
16. Mongo db can be deployed not only commodity hardware but also virtual hardware like aws.
17. Text indexing
18. Compression
19. Full table scan can disrupt if lot of operations going on.

What MongoDB is not for?

Mongo is not a good choice for:

1. Billing system or general ledger system. Example Oracle RAC
2. Search engine Example Elastic search, SOLR

OLTP and OLAP

OLTP (On-line Transaction Processing) is characterized by a large number of short on-line transactions (INSERT, UPDATE, and DELETE). The main emphasis for OLTP systems is put on very fast query processing, maintaining data integrity in multi-access environments and an effectiveness measured by number of transactions per second. In OLTP database there is detailed and current data, and schema used to store transactional databases is the entity model (usually 3rdNormalForm).

Example: RDBMS, NoSQL DB (Mongo Db)

- OLAP (On-line Analytical Processing) is characterized by relatively low volume of transactions. Queries are often very complex and involve aggregations. For OLAP systems a response time is an effectiveness measure. OLAP applications are widely used by Data Mining techniques. In OLAP database there is aggregated, historical data, stored in multi-dimensional schemas (usually star schema).

Example: Hadoop

Why mongo Db came in to picture?

1. Usage of mobile devices increased drastically in recent years
2. Social media usage in business application increased a lot.
3. Unstructured data increasing at the rate of 2x as compared to structured data.
4. Classic database (RDBMS) does not know how to scale gracefully in a distributed fashion on commodity hardware.

Bson -> binary json in mongo db

Collection similar to tables

Document similar to record

Commands

db 🡪 shows current database

Graphical user interface, text, application

Description automatically generated

show dbs or show databsases 🡪 show databases

Text

Description automatically generated

use dbname 🡪 creates and switches the database

Graphical user interface, text, application, chat or text message

Description automatically generated

**Create collection and insert data**

**insert**

db.mycollection.insert([

{"firstName":"Niranjan","lastName":"Panigrahi"},

{"firstName":"Amit","lastName":"Saha"}

])

**insertOne**

db.collection.insertOne(

<document>,

{

writeConcern: <document>

}

)

writeConcern is optional if not specified takes the default write concern.

{ w: "majority" } is the default write concern for most MongoDB deployments

**insertMany**

db.collection.insertMany(

[ <document 1> , <document 2>, ... ],

{

writeConcern: <document>,

ordered: <boolean>

}

)

|  |  |  |
| --- | --- | --- |
| ordered | boolean | Optional. A boolean specifying whether the [mongod](https://www.mongodb.com/docs/manual/reference/program/mongod/#mongodb-binary-bin.mongod) instance should perform an ordered or unordered insert. Defaults to true. |

db.collection.updateOne() when used with the upsert: true option.

db.collection.updateMany() when used with the upsert: true option.

db.collection.findAndModify() when used with the upsert: true option.

db.collection.findOneAndUpdate() when used with the upsert: true option.

db.collection.findOneAndReplace() when used with the upsert: true option.

db.collection.bulkWrite().

**updateOne**

**syntax:**

db.collection.updateOne(

<filter>,

<update>,

{

upsert: <boolean>,

writeConcern: <document>,

collation: <document>,

arrayFilters: [ <filterdocument1>, ... ],

hint: <document|string> // Available starting in MongoDB 4.2.1

}

)

**Example:**

Will update first matching document only.

db.products.updateOne({ price: 899 }, { $set: { price: null } })

{

acknowledged: true,

insertedId: null,

matchedCount: 1,

modifiedCount: 1,

upsertedCount: 0

}

**updateMany**

**syntax:**

db.collection.updateMany(

<filter>,

<update>,

{

upsert: <boolean>,

writeConcern: <document>,

collation: <document>,

arrayFilters: [ <filterdocument1>, ... ],

hint: <document|string> // Available starting in MongoDB 4.2.1

}

)

**Bulkwrite:**

db.collection.bulkWrite(

[

{ insertOne : <document> },

{ updateOne : <document> },

{ updateMany : <document> },

{ replaceOne : <document> },

{ deleteOne : <document> },

{ deleteMany : <document> }

],

{ ordered : false }

)

**Get all data from collection**

db.getCollection('testcollection').find({})

**show collections**

Graphical user interface, text, application, chat or text message

Description automatically generated

**Create collection**

db.createCollection("mycollection2")

**Drop collection**

db.mycollection2.drop()

db.getCollection('mycollection').find().pretty()

db.getCollection('mycollection').findOne()

db.mycollection.findOne()

**Importing collection**

mongoimport --db dbName --collection collectionName --file fileName.json

mongoimport --db dbName --collection collectionName --file fileName.json –jsonArray

**Drop and restore collection**

C:\project\softwares\mongo>mongoimport --db test --collection students --file C:\students.json

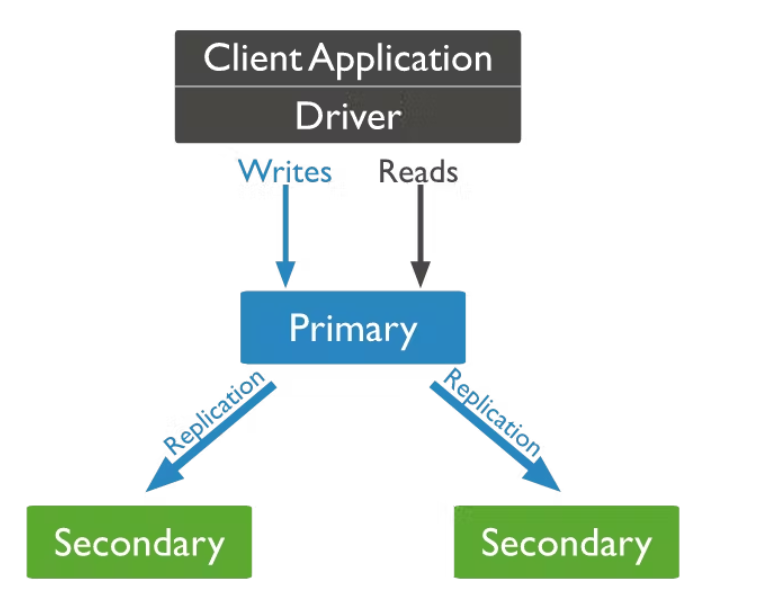
C:\project\softwares\mongo>mongorestore --drop -d test -c tweets C:\tweets.bson

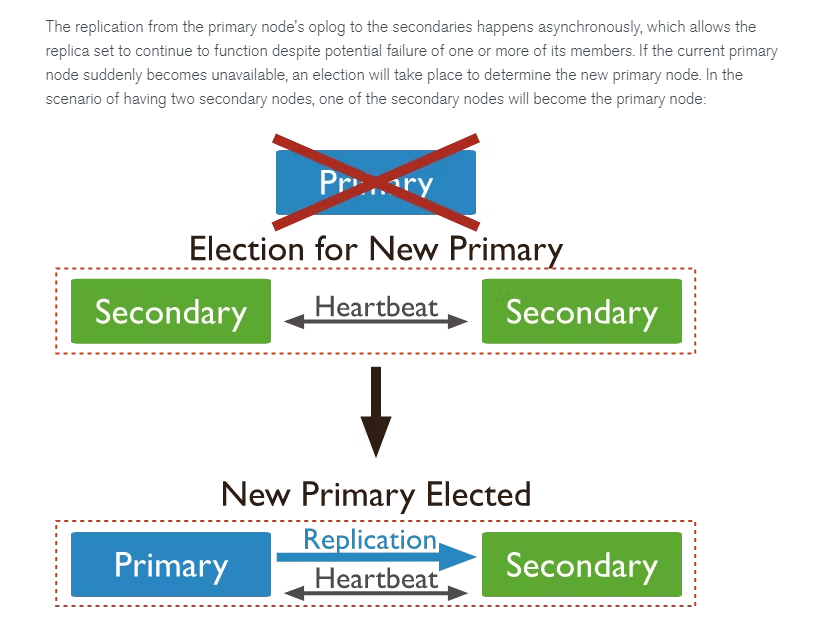
**Exporting collection**

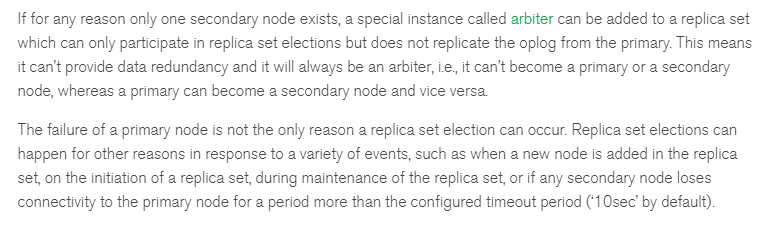
mongoexport –db test –collection students –out C:\students.json

**Mongo Replication**

1. Group of mongod processes that maintain same dataset
2. Redundancy and High availability across multiple server.
3. Increased read capacity
4. If replica set is enabled it ensures that data will not be lost in case primary node fails.







**Replication concept**

1. Write operations go to the primary node.
2. All changes are written recorded in operations log(oplog)
3. Asynchronous replication happens from primary oplog.

**Read write operation**

1. Read operation can optionally directed to specific replica-set.
2. By default, all read operation goes to primary node.

**Read preference:**

Configuration to read from specific secondary can be done in mongo cluster ,

Which is called read preference.

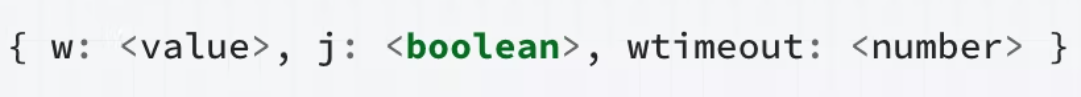
1. if a client application is configured to go directly to secondaries, then the mode parameter in the read preference should be set to [secondary](https://docs.mongodb.com/manual/core/read-preference/#secondary)
2. If there are specific needs for the least network latency irrespective of whether that happens in the primary or any secondary node, then the [nearest](https://docs.mongodb.com/manual/core/read-preference/#nearest) read preference mode should be configured.

potentially stale data comes into play (if the nearest node is a secondary node) due to the nature of [asynchronous replication](https://docs.mongodb.com/manual/replication/#asynchronous-replication) from primary to secondaries.

1. read preference mode can be set to [primary preferred](https://docs.mongodb.com/manual/core/read-preference/#primaryPreferred) or [secondary preferred](https://docs.mongodb.com/manual/core/read-preference/#secondaryPreferred). These two modes also make use of another property called [maxStalenessSeconds](https://docs.mongodb.com/manual/core/read-preference-staleness/#replica-set-read-preference-max-staleness) to determine to which node of the replica set should the read operation be directed.
2. Replica set members can lag behind the [primary](https://www.mongodb.com/docs/manual/reference/glossary/#std-term-primary) due to network congestion, low disk throughput, long-running operations, etc. The read preference  maxStalenessSeconds option lets you specify a maximum replication lag, or "staleness", for reads from [secondaries](https://www.mongodb.com/docs/manual/reference/glossary/#std-term-secondary). When a secondary's estimated staleness exceeds maxStalenessSeconds, the client stops using it for read operations.

**Write concern:**

  level of acknowledgement we desire to have from the cluster upon each write operation is called write concern.



1. MongoDB cluster, you can include additional options to ensure that the write has propagated successfully throughout the cluster. This involves adding a [write concern](https://docs.mongodb.com/manual/reference/write-concern/) property alongside an insert operation.

W: 0/1 by default 1

If W = 0 no write acknowledgement is needed

W= 1 only requires the primary node to acknowledge the write

W = 4 requires **3** secondaries need to signal as well as the primary node to acknowledge

The **j** value corresponds to whether MongoDB has been written on disk in a special area called the [journal](https://docs.mongodb.com/manual/reference/glossary/#term-journal). This is used from MongoDB for recovery purposes in case of a hard shutdown, and it is enabled by default.

**wtimeout** value is the time the command should wait before returning any result. If this is not specified and if for any reason the actual write has any network issues, then the command would block indefinitely, so it is a good practice to set this value. It is measured in milliseconds, and it is only applicable for **w** values greater than 1

Graphical user interface, text, application

Description automatically generated

**Sharding in MongoDB:**

https://www.mongodb.com/docs/manual/sharding/

https://www.bmc.com/blogs/mongodb-sharding-explained/#:~:text=Sharding%20is%20the%20process%20of,sets%20across%20multiple%20MongoDB%20instances.

Mongo cluster:

3 comp:

1.Config server

stores meta data for cluster.

can be created replica for this as well. Irrespective of shard data or query router. Meaning this is totally independent.

2.Mongos (Query router):

A interface for user to query. this will route the query to corresponding shards based on shard key defined. if key not in query then it

will broadcast and do scatter gather for this.

3:shard node

it will store distributed data.

Shard Key Index

To shard a populated collection, the collection must have an index that starts with the shard key.

When sharding an empty collection,

MongoDB creates the supporting index if the collection does not already have an appropriate index for the specified shard key

How store:

based on shard key

technique:

Hash(create hash and deside chunk where to go) 2. Range(1-10,10-20)

Note: chunk is set of documents that will go to shard in one go.

it can use any of baove strategy to store data.

What is shard key and how to choose:

A.Cardinality: no. of combination of data for any field

ex: for continent field there will be 7 diff value. so cardinality is 7.

prefer high cardinality always. so that in future we can add multi shard nodes. else it will become irrelevant in case less cardinality < shard nodes

B: Frquency:

Now data frequency should also be considerd while creating shard key

ex: if there are 7 continent menaing 7 cardinality then suppose data frequency for Asia is more then other shard will be idle and all chunk will go to

corresponding shard for asia only.

sharding query pattern:

if query doesnt have shard key then it will be distributed across shards that is called scatter gather.

For aggregation query always go with scatter gather

Notes: In case of Monotonic(Always in one direction(either inc or decrease) or constant) shard key prefer using hash

else in case of rang if it is monotoninc then other shard will not be in use

ex: shard1 : 0-10 shard2:10 - 20, shard 3: 20-30

if montonic increasing: 27 , 30 40, will go to alwysa shard 3.

in dereasing : 10,8, 7, 6 will alwyas go to shard 1

**Query Optimization in MONGODB**

**1)Use Index**

If you have a posts collection containing blog posts, and if you regularly issue a query that sorts on the author\_name field, then you can optimize the query by creating an index on the author\_name field:

db.posts.createIndex( { author\_name : 1 } )

If you regularly issue a query that sorts on the timestamp field, then you can optimize the query by creating an index on the timestamp field:

Creating this index:

|  |
| --- |
| db.posts.createIndex( { timestamp : 1 } ) |

Optimizes this query:

|  |
| --- |
| db.posts.find().sort( { timestamp : -1 } ) |

Because MongoDB can read indexes in both ascending and descending order, the direction of a single-key index does not matter.

**2)Limit the Number of Query Results to Reduce Network Demand**

MongoDB [cursors](https://www.mongodb.com/docs/manual/reference/glossary/#std-term-cursor) return results in groups of multiple documents. If you know the number of results you want, you can reduce the demand on network resources by issuing the [limit()](https://www.mongodb.com/docs/manual/reference/method/cursor.limit/#mongodb-method-cursor.limit) method.

This is typically used in conjunction with sort operations. For example, if you need only 10 results from your query to the posts collection, you would issue the following command:

|  |
| --- |
| db.posts.find().sort( { timestamp : -1 } ).skip(1).limit(10) |

**3)** **Use Projections to Return Only Necessary Data**

When you need only a subset of fields from documents, you can achieve better performance by returning only the fields you need:

For example, if in your query to the posts collection, you need only the timestamp, title, author, and abstract fields, you would issue the following command:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| db.posts.find( {}, { timestamp : 1 , title : 1 , author : 1 , abstract : 1} ).sort( { timestamp : -1 } ).   1. **Hint**   In most cases the [query optimizer](https://www.mongodb.com/docs/manual/core/query-plans/#std-label-read-operations-query-optimization) selects the optimal index for a specific operation; however, you can force MongoDB to use a specific index using the [hint()](https://www.mongodb.com/docs/manual/reference/method/cursor.hint/#mongodb-method-cursor.hint) method. Use [hint()](https://www.mongodb.com/docs/manual/reference/method/cursor.hint/#mongodb-method-cursor.hint) to support performance testing, or on some queries where you must select a field or field included in several indexes.  **6)Use the Increment Operator to Perform Operations Server-Side**  Use MongoDB's [$inc](https://www.mongodb.com/docs/manual/reference/operator/update/inc/#mongodb-update-up.-inc) operator to increment or decrement values in documents. The operator increments the value of the field on the server side, as an alternative to selecting a document, making simple modifications in the client and then writing the entire document to the server. The [$inc](https://www.mongodb.com/docs/manual/reference/operator/update/inc/#mongodb-update-up.-inc) operator can also help avoid race conditions, which would result when two application instances queried for a document, manually incremented a field, and saved the entire document back at the same time.   |  | | --- | | db.products.insertOne( | | { | | \_id: 1, | | sku: **"abc123"**, | | quantity: 10, | | metrics: { orders: 2, ratings: 3.5 } | | } | | ) |   The following [updateOne()](https://www.mongodb.com/docs/manual/reference/method/db.collection.updateOne/#mongodb-method-db.collection.updateOne) operation uses the [$inc](https://www.mongodb.com/docs/manual/reference/operator/update/inc/#mongodb-update-up.-inc) operator to:   * increase the "metrics.orders" field by 1 * increase the quantity field by -2 (which decreases quantity)  |  | | --- | | db.products.updateOne( | | { sku: **"abc123"** }, | | { $inc: { quantity: -2, **"metrics.orders"**: 1 } } | | ) |   **Index in MongoDb**  Indexes are special data structures [[1]](https://www.mongodb.com/docs/manual/indexes/#footnote-b-tree) that store a small portion of the collection's data set in  an easy to traverse form.  index type: single field index compound index user 1 score -1 first user wise then on score multi key index on array field geospatial for location text  for text field hashed to use for shard in dsitribution but suitable for exact match not for range  property: unique partial will store on basis of given filter cond  it wont use index if query expression is not match with partialexprn  ESR rule  first key should be exact match  second should be sort key and then  range should be specified in index with multi fields **TTL Indexes** [TTL indexes](https://www.mongodb.com/docs/manual/core/index-ttl/) 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.  **Hidden Indexes**  [Hidden indexes](https://www.mongodb.com/docs/manual/core/index-hidden/) are not visible to the [query planner](https://www.mongodb.com/docs/manual/core/query-plans/) 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.  **Index and Collation**  [Collation](https://www.mongodb.com/docs/manual/reference/collation/) allows users to specify language-specific rules for string comparison, such as rules  for lettercase and accent marks.  For example, the collection myColl has an index on a string field category with the collation locale "fr".   |  | | --- | | db.myColl.createIndex( { category: 1 }, { collation: { locale: **"fr"** } } ) |   The following query operation, which specifies the same collation as the index, can use the index:   |  | | --- | | db.myColl.find( { category: **"cafe"** } ).collation( { locale: **"fr"** } ) |   However, the following query operation, which by default uses the "simple" binary collator, cannot use the index:   |  | | --- | | db.myColl.find( { category: **"cafe"** } )  **Gridfs for large file store**  https://www.mongodb.com/docs/manual/core/gridfs |   [GridFS](https://www.mongodb.com/docs/manual/reference/glossary/#std-term-GridFS) is a specification for storing and retrieving files that exceed the [BSON](https://www.mongodb.com/docs/manual/reference/glossary/#std-term-BSON)-document  [size limit](https://www.mongodb.com/docs/manual/reference/limits/#std-label-limit-bson-document-size) of 16 MB.   * If your filesystem limits the number of files in a directory, you can use GridFS to store as many files as needed. * When you want to access information from portions of large files without having to load whole files into memory, you can use GridFS to recall sections of files without reading the entire file into memory. * When you want to keep your files and metadata automatically synced and deployed across a number of systems and facilities, you can use GridFS. When using [geographically distributed replica sets](https://www.mongodb.com/docs/manual/core/replica-set-architecture-geographically-distributed/#std-label-replica-set-geographical-distribution), MongoDB can distribute files and their metadata automatically to a number of [mongod](https://www.mongodb.com/docs/manual/reference/program/mongod/#mongodb-binary-bin.mongod) instances and facilities.   [GridFS](https://www.mongodb.com/docs/manual/reference/glossary/#std-term-GridFS) stores files in two collections:   * chunks stores the binary chunks. For details, see [The chunks Collection](https://www.mongodb.com/docs/manual/core/gridfs/#std-label-gridfs-chunks-collection). * files stores the file's metadata. For details, see [The files Collection](https://www.mongodb.com/docs/manual/core/gridfs/#std-label-gridfs-files-collection).   GridFS places the collections in a common bucket by prefixing each with the bucket name.  By default, GridFS uses two collections with a bucket named fs:   * fs.files * fs.chunks   Go to mongo bin directory to run  mongofiles.exe -d gridfs put song.mp3  **Mongodb storage engine**  The [storage engine](https://www.mongodb.com/docs/manual/reference/glossary/#std-term-storage-engine) is the component of the database that is responsible for managing how  data is stored, both in memory and on disk.  2 types of storage engine.  **➤ WiredTiger Storage Engine (Default)**  [WiredTiger](https://www.mongodb.com/docs/manual/core/wiredtiger/) is the default storage engine starting in MongoDB 3.2. It is well-suited for  most workloads and is recommended for new deployments.  WiredTiger provides a document-level concurrency model, checkpointing, and  compression, among other features.  In MongoDB Enterprise, WiredTiger also supports [Encryption at Rest](https://www.mongodb.com/docs/manual/core/security-encryption-at-rest/).  See [Encrypted Storage Engine](https://www.mongodb.com/docs/manual/core/security-encryption-at-rest/#std-label-encrypted-storage-engine).  **➤ In-Memory Storage Engine**  [In-Memory Storage Engine](https://www.mongodb.com/docs/manual/core/inmemory/) is available in MongoDB Enterprise. Rather than storing  documents on-disk, it retains them in-memory for more predictable data latencies.  The in-memory storage engine is non-persistent and does not write data to a persistent storage.  Non-persisted data includes application data and system data, such as users, permissions,  indexes, replica set configuration, sharded cluster configuration, etc.  [Capped collections](https://www.mongodb.com/docs/manual/reference/glossary/#std-term-capped-collection) 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.  **Mongodb transactions**  MongoDB supports multi-document transactions. With distributed transactions, transactions  can be used across multiple operations, collections, databases, documents, and shards.   |  | | --- | | */\** | | *For a replica set, include the replica set name and a seedlist of the members in the URI string; e.g.* | | *String uri = "mongodb://mongodb0.example.com:27017,mongodb1.example.com:27017/admin?replicaSet=myRepl";* | | *For a sharded cluster, connect to the mongos instances; e.g.* | | *String uri = "mongodb://mongos0.example.com:27017,mongos1.example.com:27017:27017/admin";* | | *\*/* | |  | | final MongoClient client = MongoClients.create(uri); | |  | | */\** | | *Create collections.* | | *\*/* | |  | | client.getDatabase(**"mydb1"**).getCollection(**"foo"**) | | .withWriteConcern(WriteConcern.MAJORITY).insertOne(new Document(**"abc"**, 0)); | | client.getDatabase(**"mydb2"**).getCollection(**"bar"**) | | .withWriteConcern(WriteConcern.MAJORITY).insertOne(new Document(**"xyz"**, 0)); | |  | | */\* Step 1: Start a client session. \*/* | |  | | final ClientSession clientSession = client.startSession(); | |  | | */\* Step 2: Optional. Define options to use for the transaction. \*/* | |  | | TransactionOptions txnOptions = TransactionOptions.builder() | | .readPreference(ReadPreference.primary()) | | .readConcern(ReadConcern.LOCAL) | | .writeConcern(WriteConcern.MAJORITY) | | .build(); | |  | | */\* Step 3: Define the sequence of operations to perform inside the transactions. \*/* | |  | | TransactionBody txnBody = new TransactionBody<String>() { | | public String execute() { | | MongoCollection<Document> coll1 = client.getDatabase(**"mydb1"**).getCollection(**"foo"**); | | MongoCollection<Document> coll2 = client.getDatabase(**"mydb2"**).getCollection(**"bar"**); | |  | | */\** | | *Important:: You must pass the session to the operations.* | | *\*/* | | coll1.insertOne(clientSession, new Document(**"abc"**, 1)); | | coll2.insertOne(clientSession, new Document(**"xyz"**, 999)); | | return **"Inserted into collections in different databases"**; | | } | | }; | | try { | | */\** | | *Step 4: Use .withTransaction() to start a transaction,* | | *execute the callback, and commit (or abort on error).* | | *\*/* | |  | | clientSession.withTransaction(txnBody, txnOptions); | | } catch (RuntimeException e) { | | *// some error handling* | | } finally { | | clientSession.close(); | | } | |