MONGO DB LAB

## The createCollection()

* MongoDB **db.createCollection(name, options)** is used to create collection.
* Options

|  |  |  |
| --- | --- | --- |
| **Field** | **Type** | **Description** |
| capped | Boolean | (Optional) If true, enables a capped collection. Capped collection is a fixed size collection that automatically overwrites its oldest entries when it reaches its maximum size. **If you specify true, you need to specify size parameter also.** |
| autoIndexId | Boolean | (Optional) If true, automatically create index on \_id fields Default value is false. |
| size | number | (Optional) Specifies a maximum size in bytes for a capped collection. **If capped is true, then you need to specify this field also.** |
| max | number | (Optional) Specifies the maximum number of documents allowed in the capped collection. |

Creating simple collection collection:

>use test

>db.createCollection("customers")

>show collections

Indexing: (1=> ascending -1 => descending)

>db.post.ensureIndex({"title":1})

\_id: ObjectId(4 bytes timestamp, 3 bytes machine id, 2 bytes process id, 3 bytes incrementer) (Total : 12 bytes)

//compound index

>db.post.ensureIndex({"title":1,"description":-1})

Options for Indexing:

name : name of the index

background: builds index on background to avoid blocking db activities

unique : create unique index to avoid duplicate index keys

dropDups: create unique index that have duplicates.

expireAfterSeconds(): specify how long as TTL doc. Lives

Capped collection:

>db.createCollection("people", { capped : true, autoIndexId : true, size :6142800, max : 10000 } )

Dropping a collection:

>use test

>db.people.drop()

Capped Collection:

To limit the number of documents in the collection using the **max** parameter −

>db.createCollection("cappedLogCollection",

{capped:true,size:10000,max:1000})

To check whether a collection is capped or not, use the following **isCapped** command −

>db.cappedLogCollection.isCapped()

If there is an existing collection can be converted to capped

>db.runCommand({"convertToCapped":"posts",size:10000})

## Querying Capped Collection

## db.cappedLogCollection.insertOne( {name:"murthy",salary:5000,job:"engineer" } );

## {

## "acknowledged" : true,

## "insertedId" : ObjectId("59ee144379ebde33b753a1db")

## }

## >db.cappedLogCollection.insertOne( {name:"raju",salary:6000,job:"officer" } );

## > db.cappedLogCollection.insertOne( {name:"kiran",salary:8000,job:"manager" } );

By default, a find query on a capped collection will display results in insertion order.

Documents to be retrieved in reverse order, use the **sort**

>db.cappedLogCollection.find().sort({$natural:1})

{ "\_id" : ObjectId("59ee144379ebde33b753a1db"), "name" : "murthy", "salary" : 5000, "job" : "engineer" }

{ "\_id" : ObjectId("59ee148279ebde33b753a1dc"), "name" : "raju", "salary" : 6000, "job" : "officer" }

{ "\_id" : ObjectId("59ee14b079ebde33b753a1dd"), "name" : "kiran", "salary" : 8000, "job" : "manager" }

Points on Capped Collection:

* We cannot delete documents from a capped collection.
* While inserting a new document, MongoDB does not have to actually look for a place to accommodate new document on the disk. It can blindly insert the new document at the tail of the collection. This makes insert operations in capped collections very fast.
* while reading documents MongoDB returns the documents in the same order as present on disk. This makes the read operation very fast

Insert Command :insertOne(), insertMany() are added in mongodb 3.2 onwards

db.blog.insert({

\_id: ObjectId(7df78ad8902c),

title: 'Murthy blog',

description: 'About Murthy',

by: 'Murthy',

email: 'murthy@yahoo.com',

tags: ['mongodb', 'mongoose'],

likes: 45

})

>db.myCollection.insertOne( { x: 1 } );

InsertMany()

db.inventory.insertMany([

{ item: "journal", qty: 25, tags: ["blank", "red"], size: { h: 14, w: 21, uom: "cm" } },

{ item: "mat", qty: 85, tags: ["gray"], size: { h: 27.9, w: 35.5, uom: "cm" } },

{ item: "mousepad", qty: 25, tags: ["gel", "blue"], size: { h: 19, w: 22.85, uom: "cm" } }

])

db.inventory.find( { price: { $type: "decimal" } } )

Add more documents to collection using array syntax:

>db.post.insert([

{

title: 'Customer Management',

description: 'Customer management in Mongodb',

by: 'Murthy',

email: 'murthy@yahoo.com',

tags: ['mongodb', 'database'],

likes: 45

},

{

title: 'Service Managment',

description: "To work with ITSM",

by: 'Kumar',

email: 'Kumar@yahoo.com',

tags: ['mongodb', 'RPA', 'Cloud'],

likes: 20,

comments: [

{

user:'Kiran',

message: 'It is the future technology',

dateCreated: new Date(2017,11,10,2,35),

like: 10

}

]

}

])

* Delete document:

[db.collection.deleteOne()](https://docs.mongodb.com/manual/reference/method/db.collection.deleteOne/#db.collection.deleteOne) New in version 3.2

[db.collection.deleteMany()](https://docs.mongodb.com/manual/reference/method/db.collection.deleteMany/#db.collection.deleteMany) New in version 3.2

>db.post.remove() // will remove all documents

>db.post.remove({‘by’:’sriram’},1) // will remove one doc

Projections: (show or hide fields) 1=show 0=hide

>db.post.find({},{"title":1,\_id:0})

Limiting no. of records: limit(n)

>db.post.find({},{"title":1,\_id:0}).limit(2)

Skipping Records : skip(n)

>db.post.find({},{"title":1,\_id:0}).limit(1).skip(1)

Sorting : sort() 1= ascending -1=descending

>db.post.find({},{"title":1,\_id:0}).sort({"title":-1})

Relationships in MongoDB

Relationships can be modelled with  **Embedded** and **Referenced** approaches. Such relationships can be either 1:1, 1:N, N:1 or N:N.

User

{

"\_id":ObjectId("52ffc33cd85242f436000001"),

"name": "murthy",

"contact": "984839483",

"dob": "01-01-1966"

})

Address

{

"\_id":ObjectId("52ffc4a5d85242602e000000"),

"building": "22 A, Tarnaka",

"pincode": 500044,

"city": "Hyderabad",

"state": "Telangana"

})

Embedded approach:

>db.userInfo.insert({

"\_id":ObjectId("52ffc33cd85242f436000001"),

"contact": "984839848",

"dob": "01-01-1966",

"name": "Murthy",

"address": [

{

"building": "22 A, Tarnaka",

"pincode": 500013,

"city": “Hyderabad",

"state": "Telangana"

},

{

"building": "170 A, DD colony",

"pincode": 500012,

"city": "Chennai",

"state": "TN"

}

]

}

This approach maintains all the related data in a single document, which makes it easy to retrieve and maintain. The whole document can be retrieved in a single query such as −

**>db.userInfo.findOne({"name":"murthy"},{"address":1})**

Modeling Referenced Relationships

This is the approach of designing normalized relationship.

In this approach, both the user and address documents will be maintained separately but the user document will contain a field that will reference the address document's **id** field.

{

"\_id":ObjectId("52ffc33cd85242f436000001"),

"contact": "98450394",

"dob": "01-01-1996",

"name": "murthy",

"address\_ids": [

ObjectId("52ffc4a5d85242602e000000"),

ObjectId("52ffc4a5d85242602e000001")

]

}

>var result = db.users.findOne({"name":"murthy"},{"address\_ids":1})

>var addresses = db.address.find({"\_id":{"$in":result["address\_ids"]}})

# **MongoDB - Atomic Operations**

MongoDB does not support **multi-document atomic transactions**.

it does provide atomic operations on a single document.

So if a document has hundred fields the update statement will either update all the fields or none, hence maintaining atomicity at the document-level.

Products document:

"\_id":1,

"product\_name": "Samsung S3",

"category": "mobiles",

"product\_total": 5,

"product\_available": 3,

"product\_bought\_by": [

{

"customer": "murthy",

"date": "7-Jan-2017"

},

{

"customer": "kiran",

"date": "8-Jan-2016"

}

]

}

Atomic transaction (Update customer name and date only if product is available)

>db.products.findAndModify({

query:{\_id:2,product\_available:{$gt:0}},

update:{

$inc:{product\_available:-1},

$push:{product\_bought\_by:{customer:"Sriram ",date:"9-Jan-2017"}}

}

})

Shutdown already running MongoDB server.

Start the MongoDB server by specifying -- replSet option.

mongod --port "PORT" --dbpath "YOUR\_DB\_DATA\_PATH"

--replSet "REPLICA\_SET\_INSTANCE\_NAME"

mongod --port 27017 --dbpath "D:\setup\mongodb\data"

--replSet rset0

* It will start a mongod instance with the name rset0, on port 27017.
* Now start the command prompt and connect to this mongod instance.
* In Mongo client, issue the command **rs.initiate()** to initiate a new replica set.
* To check the replica set configuration, issue the command **rs.conf()**. To check the status of replica set issue the command **rs.status()**.

## Add Members to Replica Set

To add members to replica set, start mongod instances on multiple machines and then start a mongo client and issue a command **rs.add()**.

>rs.add(HOST\_NAME:PORT)

Suppose mongod instance name is **mongod1.net** and it is running on port **27017**. To add this instance to replica set, issue the command **rs.add()**in Mongo client.

>rs.add("mongod1.net:27017")

Add mongod instance to replica set only when connected to primary node.

To check whether connected to primary or not, issue the command **db.isMaster()** in mongo client.

Mongodb Administration

<https://sourceforge.net/projects/rockmongoadmin/>

After downloading, Open any web browser and access the **index.php** page from the folder rockmongo.

Enter admin/admin as username/password respectively.

## Creating New Database

To create a new database, click **Databases** tab.

Click **Create New Database**.

On the next screen, provide the name of the new database and click on **Create**. Observe new Database in left panel

## Creating New Collection

To create a new collection inside a database, click on that database from the left panel.

Click on the **New Collection** link on top. Provide the required name of the collection.

Click on **Create**. A new collection will be created and observe in panel

## Creating New Document

To create a new document, click on the collection to add documents.

To create a new document, click on the **Insert** link at the top.

Enter the document's data either in JSON or array format and click on **Save**.

## Export/Import Data

To import/export data of any collection, click on that collection and then click on **Export/Import** link on the top panel.

Follow the next instructions to export data in a zip format and then import the same zip file to import back data.

**GridFS**

**GridFS** is the MongoDB specification for storing and retrieving large files such as images, audio files, video files, etc.

It is kind of a file system to store files but its data is stored within MongoDB collections.

GridFS has the capability to store files even greater than its document size limit of 16MB.

GridFS divides a file into chunks and stores each chunk of data in a separate document, each of maximum size 255k.

GridFS by default uses two collections **fs.files** and **fs.chunks** to store the file's metadata and the chunks.

Each chunk is identified by its unique \_id ObjectId field. The fs.files severs as a parent document.

The **files\_id** field in the fs.chunks document links the chunk to its parent.

fs.files() document sample

{

"filename": "test.txt",

"chunkSize": NumberInt(261120),

"uploadDate": ISODate("2017-04-13T11:32:33.557Z"),

"md5": "7b762939321e146569b07f72c62cca4f",

"length": NumberInt(646)

}

fs.chunks() document sample

{

"files\_id": ObjectId("534a75d19f54bfec8a2fe44b"),

"n": NumberInt(0),

"data": "Mongo Binary Data"

}

## Adding Files to GridFS

Now, we will store an mp3 file using GridFS using the **put** command.

use the **mongofiles.exe** utility present in the bin folder of the MongoDB installation folder.

Open command prompt, navigate to the mongofiles.exe in the bin folder of MongoDB installation folder and type the following code −

>mongofiles.exe -d gridfs put song.mp3

Here, **gridfs** is the name of the database in which the file will be stored.

If the database is not present, MongoDB will automatically create a new document on the fly.

Song.mp3 is the name of the file uploaded. To see the file's document in database, find query −

>db.fs.files.find()

{

\_id: ObjectId('534a811bf8b4aa4d33fdf94d'),

filename: "song.mp3",

chunkSize: 261120,

uploadDate: new Date(1397391643474), md5: "e4f53379c909f7bed2e9d631e15c1c41",

length: 10401959

}

We can also see all the chunks present in fs.chunks collection related to the stored file with the following code, using the document id returned in the previous query −

>db.fs.chunks.find({files\_id:ObjectId('534a811bf8b4aa4d33fdf94d')})

query returned 40 documents meaning that the whole mp3 document was divided in 40 chunks of data.

SQL vs Mongodb

| **SQL SELECT Statements** | **MongoDB find() Statements** |
| --- | --- |
| **SELECT** \*  **FROM** people | **db**.**people**.**find**() |
| **SELECT** id,  user\_id,  status  **FROM** people | **db**.**people**.**find**(  { },  { **user\_id**: 1, **status**: 1 }  ) |
| **SELECT** user\_id, status  **FROM** people | **db**.**people**.**find**(  { },  { **user\_id**: 1, **status**: 1, **\_id**: 0 }  ) |
| **SELECT** \*  **FROM** people  **WHERE** status = "A" | **db**.**people**.**find**(  { **status**: "A" }  ) |
| **SELECT** user\_id, status  **FROM** people  **WHERE** status = "A" | **db**.**people**.**find**(  { **status**: "A" },  { **user\_id**: 1, **status**: 1, **\_id**: 0 }  ) |
| **SELECT** \*  **FROM** people  **WHERE** status != "A" | **db**.**people**.**find**(  { **status**: { **$ne**: "A" } }  ) |
| **SELECT** \*  **FROM** people  **WHERE** status = "A"  **AND** age = 50 | **db**.**people**.**find**(  { **status**: "A",  **age**: 50 }  ) |
| **SELECT** \*  **FROM** people  **WHERE** status = "A"  **OR** age = 50 | **db**.**people**.**find**(  { **$or**: [ { **status**: "A" } ,  { **age**: 50 } ] }  ) |
| **SELECT** \*  **FROM** people  **WHERE** age > 25 | **db**.**people**.**find**(  { **age**: { **$gt**: 25 } }  ) |
| **SELECT** \*  **FROM** people  **WHERE** age < 25 | **db**.**people**.**find**(  { **age**: { **$lt**: 25 } }  ) |
| **SELECT** \*  **FROM** people  **WHERE** age > 25  **AND** age <= 50 | **db**.**people**.**find**(  { **age**: { **$gt**: 25, **$lte**: 50 } }  ) |
| **SELECT** \*  **FROM** people  **WHERE** user\_id **like** "%bc%" | **db**.**people**.**find**( { **user\_id**: /bc/ } )  -or-  **db**.**people**.**find**( { **user\_id**: { **$regex**: /bc/ } } ) |
| **SELECT** \*  **FROM** people  **WHERE** user\_id **like** "bc%" | **db**.**people**.**find**( { **user\_id**: /^bc/ } )  -or-  **db**.**people**.**find**( { **user\_id**: { **$regex**: /^bc/ } } ) |
| **SELECT** \*  **FROM** people  **WHERE** status = "A"  **ORDER** **BY** user\_id **ASC** | **db**.**people**.**find**( { **status**: "A" } ).**sort**( { **user\_id**: 1 } ) |
| **SELECT** \*  **FROM** people  **WHERE** status = "A"  **ORDER** **BY** user\_id **DESC** | **db**.**people**.**find**( { **status**: "A" } ).**sort**( { **user\_id**: -1 } ) |
| **SELECT** **COUNT**(\*)  **FROM** people | **db**.**people**.**count**()  or  **db**.**people**.**find**().**count**() |
| **SELECT** **COUNT**(user\_id)  **FROM** people | **db**.**people**.**count**( { **user\_id**: { **$exists**: **true** } } )  or  **db**.**people**.**find**( { **user\_id**: { **$exists**: **true** } } ).**count**() |
| **SELECT** **COUNT**(\*)  **FROM** people  **WHERE** age > 30 | **db**.**people**.**count**( { **age**: { **$gt**: 30 } } )  or  **db**.**people**.**find**( { **age**: { **$gt**: 30 } } ).**count**() |
| **SELECT** **DISTINCT**(status)  **FROM** people | **db**.**people**.**distinct**( "status" ) |
| **SELECT** \*  **FROM** people  **LIMIT** 1 | **db**.**people**.**findOne**()  or  **db**.**people**.**find**().**limit**(1) |
| **SELECT** \*  **FROM** people  **LIMIT** 5  SKIP 10 | **db**.**people**.**find**().**limit**(5).**skip**(10) |
| **EXPLAIN** **SELECT** \*  **FROM** people  **WHERE** status = "A" | **db**.**people**.**find**( { **status**: "A" } ).**explain**() |
| **SQL Update Statements** | **MongoDB updateMany() Statements** |
| **UPDATE people**  **SET status = "C"**  **WHERE age > 25** | **db.people.updateMany(**  **{ age: { $gt: 25 } },**  **{ $set: { status: "C" } }**  **)** |
| **UPDATE people**  **SET age = age + 3**  **WHERE status = "A"** | **db.people.updateMany(**  **{ status: "A" } ,**  **{ $inc: { age: 3 } }**  **)** |
| **SQL Delete Statements** | **MongoDB deleteMany() Statements** |
| **DELETE FROM people**  **WHERE status = "D"** | **db.people.deleteMany( { status: "D" } )** |
| **DELETE FROM people** | **db.people.deleteMany({})** |

# **Perform Two Phase Commits**

**On this page**

* [Synopsis](https://docs.mongodb.com/manual/tutorial/perform-two-phase-commits/#synopsis)
* [Background](https://docs.mongodb.com/manual/tutorial/perform-two-phase-commits/#background)
* [Pattern](https://docs.mongodb.com/manual/tutorial/perform-two-phase-commits/#pattern)
* [Recovering from Failure Scenarios](https://docs.mongodb.com/manual/tutorial/perform-two-phase-commits/#recovering-from-failure-scenarios)
* [Multiple Applications](https://docs.mongodb.com/manual/tutorial/perform-two-phase-commits/#multiple-applications)
* [Using Two-Phase Commits in Production Applications](https://docs.mongodb.com/manual/tutorial/perform-two-phase-commits/#using-two-phase-commits-in-production-applications)

## Synopsis

This document provides a pattern for doing multi-document updates or “multi-document transactions” using a two-phase commit approach for writing data to multiple documents. Additionally, you can extend this process to provide a [rollback-like](https://docs.mongodb.com/manual/tutorial/perform-two-phase-commits/#phase-commits-rollback) functionality.

## Background

Operations on a single [document](https://docs.mongodb.com/manual/reference/glossary/#term-document) are always atomic with MongoDB databases; however, operations that involve multiple documents, which are often referred to as “multi-document transactions”, are not atomic. Since documents can be fairly complex and contain multiple “nested” documents, single-document atomicity provides the necessary support for many practical use cases.

Despite the power of single-document atomic operations, there are cases that require multi-document transactions. When executing a transaction composed of sequential operations, certain issues arise, such as:

* Atomicity: if one operation fails, the previous operation within the transaction must “rollback” to the previous state (i.e. the “nothing,” in “all or nothing”).
* Consistency: if a major failure (i.e. network, hardware) interrupts the transaction, the database must be able to recover a consistent state.

For situations that require multi-document transactions, you can implement two-phase commit in your application to provide support for these kinds of multi-document updates. Using two-phase commit ensures that data is consistent and, in case of an error, the state that preceded the transaction is [recoverable](https://docs.mongodb.com/manual/tutorial/perform-two-phase-commits/#phase-commits-rollback). During the procedure, however, documents can represent pending data and states.

**NOTE**

Because only single-document operations are atomic with MongoDB, two-phase commits can only offer transaction-like semantics. It is possible for applications to return intermediate data at intermediate points during the two-phase commit or rollback.

## Pattern

### Overview

Consider a scenario where you want to transfer funds from account A to account B. In a relational database system, you can subtract the funds from A and add the funds to B in a single multi-statement transaction. In MongoDB, you can emulate a two-phase commit to achieve a comparable result.

The examples in this tutorial use the following two collections:

1. A collection named accounts to store account information.
2. A collection named transactions to store information on the fund transfer transactions.

### Initialize Source and Destination Accounts

Insert into the accounts collection a document for account A and a document for account B.

db.accounts.insert(

[

{ \_id: "A", balance: 1000, pendingTransactions: [] },

{ \_id: "B", balance: 1000, pendingTransactions: [] }

]

)

The operation returns a [BulkWriteResult()](https://docs.mongodb.com/manual/reference/method/BulkWriteResult/#BulkWriteResult) object with the status of the operation. Upon successful insert, the [BulkWriteResult()](https://docs.mongodb.com/manual/reference/method/BulkWriteResult/#BulkWriteResult) has [nInserted](https://docs.mongodb.com/manual/reference/method/BulkWriteResult/#BulkWriteResult.nInserted) set to 2 .

### Initialize Transfer Record

For each fund transfer to perform, insert into the transactions collection a document with the transfer information. The document contains the following fields:

* source and destination fields, which refer to the \_id fields from the accounts collection,
* value field, which specifies the amount of transfer affecting the balance of the source and destination accounts,
* state field, which reflects the current state of the transfer. The state field can have the value of initial, pending, applied, done, canceling, and canceled.
* lastModified field, which reflects last modification date.

To initialize the transfer of 100 from account A to account B, insert into the transactions collection a document with the transfer information, the transaction state of "initial", and the lastModified field set to the current date:

db.transactions.insert(

{ \_id: 1, source: "A", destination: "B", value: 100, state: "initial", lastModified: **new** Date() }

)

The operation returns a [WriteResult()](https://docs.mongodb.com/manual/reference/method/WriteResult/#WriteResult) object with the status of the operation. Upon successful insert, the [WriteResult()](https://docs.mongodb.com/manual/reference/method/WriteResult/#WriteResult) object has [nInserted](https://docs.mongodb.com/manual/reference/method/WriteResult/#WriteResult.nInserted) set to 1.

### Transfer Funds Between Accounts Using Two-Phase Commit

**1**

#### Retrieve the transaction to start.

From the transactions collection, find a transaction in the initial state. Currently the transactions collection has only one document, namely the one added in the [Initialize Transfer Record](https://docs.mongodb.com/manual/tutorial/perform-two-phase-commits/#initialize-transfer-record) step. If the collection contains additional documents, the query will return any transaction with an initial state unless you specify additional query conditions.

**var** t = db.transactions.findOne( { state: "initial" } )

Type the variable t in the [mongo](https://docs.mongodb.com/manual/reference/program/mongo/#bin.mongo) shell to print the contents of the variable. The operation should print a document similar to the following except the lastModified field should reflect date of your insert operation:

{ "\_id" : 1, "source" : "A", "destination" : "B", "value" : 100, "state" : "initial", "lastModified" : ISODate("2014-07-11T20:39:26.345Z") }

**2**

#### Update transaction state to pending.

Set the transaction state from initial to pending and use the [$currentDate](https://docs.mongodb.com/manual/reference/operator/update/currentDate/#up._S_currentDate) operator to set the lastModified field to the current date.

db.transactions.update(

{ \_id: t.\_id, state: "initial" },

{

$set: { state: "pending" },

$currentDate: { lastModified: **true** }

}

)

The operation returns a [WriteResult()](https://docs.mongodb.com/manual/reference/method/WriteResult/#WriteResult) object with the status of the operation. Upon successful update, the [nMatched](https://docs.mongodb.com/manual/reference/method/WriteResult/#WriteResult.nMatched) and [nModified](https://docs.mongodb.com/manual/reference/method/WriteResult/#WriteResult.nModified) displays 1.

In the update statement, the state: "initial" condition ensures that no other process has already updated this record. If [nMatched](https://docs.mongodb.com/manual/reference/method/WriteResult/#WriteResult.nMatched) and [nModified](https://docs.mongodb.com/manual/reference/method/WriteResult/#WriteResult.nModified) is 0, go back to the first step to get a different transaction and restart the procedure.

**3**

#### Apply the transaction to both accounts.

Apply the transaction t to both accounts using the [update()](https://docs.mongodb.com/manual/reference/method/db.collection.update/#db.collection.update) method if the transaction has not been applied to the accounts. In the update condition, include the condition pendingTransactions: {$ne: t.\_id } in order to avoid re-applying the transaction if the step is run more than once.

To apply the transaction to the account, update both the balance field and the pendingTransactions field.

Update the source account, subtracting from its balance the transaction value and adding to its pendingTransactions array the transaction \_id.

db.accounts.update(

{ \_id: t.source, pendingTransactions: { $ne: t.\_id } },

{ $inc: { balance: -t.value }, $push: { pendingTransactions: t.\_id } }

)

Upon successful update, the method returns a [WriteResult()](https://docs.mongodb.com/manual/reference/method/WriteResult/#WriteResult) object with [nMatched](https://docs.mongodb.com/manual/reference/method/WriteResult/#WriteResult.nMatched) and [nModified](https://docs.mongodb.com/manual/reference/method/WriteResult/#WriteResult.nModified) set to 1.

Update the destination account, adding to its balance the transaction value and adding to its pendingTransactions array the transaction \_id .

db.accounts.update(

{ \_id: t.destination, pendingTransactions: { $ne: t.\_id } },

{ $inc: { balance: t.value }, $push: { pendingTransactions: t.\_id } }

)

Upon successful update, the method returns a [WriteResult()](https://docs.mongodb.com/manual/reference/method/WriteResult/#WriteResult) object with [nMatched](https://docs.mongodb.com/manual/reference/method/WriteResult/#WriteResult.nMatched) and [nModified](https://docs.mongodb.com/manual/reference/method/WriteResult/#WriteResult.nModified) set to 1.

**4**

#### Update transaction state to applied.

Use the following [update()](https://docs.mongodb.com/manual/reference/method/db.collection.update/#db.collection.update) operation to set the transaction’s state to applied and update thelastModified field:

db.transactions.update(

{ \_id: t.\_id, state: "pending" },

{

$set: { state: "applied" },

$currentDate: { lastModified: **true** }

}

)

Upon successful update, the method returns a [WriteResult()](https://docs.mongodb.com/manual/reference/method/WriteResult/#WriteResult) object with [nMatched](https://docs.mongodb.com/manual/reference/method/WriteResult/#WriteResult.nMatched) and [nModified](https://docs.mongodb.com/manual/reference/method/WriteResult/#WriteResult.nModified) set to 1.

**5**

#### Update both accounts’ list of pending transactions.

Remove the applied transaction \_id from the pendingTransactions array for both accounts.

Update the source account.

db.accounts.update(

{ \_id: t.source, pendingTransactions: t.\_id },

{ $pull: { pendingTransactions: t.\_id } }

)

Upon successful update, the method returns a [WriteResult()](https://docs.mongodb.com/manual/reference/method/WriteResult/#WriteResult) object with [nMatched](https://docs.mongodb.com/manual/reference/method/WriteResult/#WriteResult.nMatched) and [nModified](https://docs.mongodb.com/manual/reference/method/WriteResult/#WriteResult.nModified) set to 1.

Update the destination account.

db.accounts.update(

{ \_id: t.destination, pendingTransactions: t.\_id },

{ $pull: { pendingTransactions: t.\_id } }

)

Upon successful update, the method returns a [WriteResult()](https://docs.mongodb.com/manual/reference/method/WriteResult/#WriteResult) object with [nMatched](https://docs.mongodb.com/manual/reference/method/WriteResult/#WriteResult.nMatched) and [nModified](https://docs.mongodb.com/manual/reference/method/WriteResult/#WriteResult.nModified) set to 1.

**6**

#### Update transaction state to done.

Complete the transaction by setting the state of the transaction to done and updating the lastModified field:

db.transactions.update(

{ \_id: t.\_id, state: "applied" },

{

$set: { state: "done" },

$currentDate: { lastModified: **true** }

}

)

Upon successful update, the method returns a [WriteResult()](https://docs.mongodb.com/manual/reference/method/WriteResult/#WriteResult) object with [nMatched](https://docs.mongodb.com/manual/reference/method/WriteResult/#WriteResult.nMatched) and[nModified](https://docs.mongodb.com/manual/reference/method/WriteResult/#WriteResult.nModified) set to 1.

## Recovering from Failure Scenarios

The most important part of the transaction procedure is not the prototypical example above, but rather the possibility for recovering from the various failure scenarios when transactions do not complete successfully. This section presents an overview of possible failures and provides steps to recover from these kinds of events.

### Recovery Operations

The two-phase commit pattern allows applications running the sequence to resume the transaction and arrive at a consistent state. Run the recovery operations at application startup, and possibly at regular intervals, to catch any unfinished transactions.

The time required to reach a consistent state depends on how long the application needs to recover each transaction.

The following recovery procedures uses the lastModified date as an indicator of whether the pending transaction requires recovery; specifically, if the pending or applied transaction has not been updated in the last 30 minutes, the procedures determine that these transactions require recovery. You can use different conditions to make this determination.

#### Transactions in Pending State

To recover from failures that occur after step “[Update transaction state to pending.](https://docs.mongodb.com/manual/tutorial/perform-two-phase-commits/#update-transaction-state-to-pending)” but before “[Update transaction state to applied.](https://docs.mongodb.com/manual/tutorial/perform-two-phase-commits/#update-transaction-state-to-applied)” step, retrieve from the transactions collection a pending transaction for recovery:

**var** dateThreshold = **new** Date();

dateThreshold.setMinutes(dateThreshold.getMinutes() - 30);

**var** t = db.transactions.findOne( { state: "pending", lastModified: { $lt: dateThreshold } } );

And resume from step “[Apply the transaction to both accounts.](https://docs.mongodb.com/manual/tutorial/perform-two-phase-commits/#apply-the-transaction-to-both-accounts)“

#### Transactions in Applied State

To recover from failures that occur after step “[Update transaction state to applied.](https://docs.mongodb.com/manual/tutorial/perform-two-phase-commits/#update-transaction-state-to-applied)” but before “[Update transaction state to done.](https://docs.mongodb.com/manual/tutorial/perform-two-phase-commits/#update-transaction-state-to-done)” step, retrieve from the transactions collection an applied transaction for recovery:

**var** dateThreshold = **new** Date();

dateThreshold.setMinutes(dateThreshold.getMinutes() - 30);

**var** t = db.transactions.findOne( { state: "applied", lastModified: { $lt: dateThreshold } } );

And resume from “[Update both accounts’ list of pending transactions.](https://docs.mongodb.com/manual/tutorial/perform-two-phase-commits/#update-both-accounts-list-of-pending-transactions)“

### Rollback Operations

In some cases, you may need to “roll back” or undo a transaction; e.g., if the application needs to “cancel” the transaction or if one of the accounts does not exist or stops existing during the transaction.

#### Transactions in Applied State

After the “[Update transaction state to applied.](https://docs.mongodb.com/manual/tutorial/perform-two-phase-commits/#update-transaction-state-to-applied)” step, you should **not** roll back the transaction. Instead, complete that transaction and [create a new transaction](https://docs.mongodb.com/manual/tutorial/perform-two-phase-commits/#initialize-transfer-record) to reverse the transaction by switching the values in the source and the destination fields.

#### Transactions in Pending State

After the “[Update transaction state to pending.](https://docs.mongodb.com/manual/tutorial/perform-two-phase-commits/#update-transaction-state-to-pending)” step, but before the “[Update transaction state to applied.](https://docs.mongodb.com/manual/tutorial/perform-two-phase-commits/#update-transaction-state-to-applied)” step, you can rollback the transaction using the following procedure:

**1**

#### Update transaction state to canceling.

Update the transaction state from pending to canceling.

db.transactions.update(

{ \_id: t.\_id, state: "pending" },

{

$set: { state: "canceling" },

$currentDate: { lastModified: **true** }

}

)

Upon successful update, the method returns a [WriteResult()](https://docs.mongodb.com/manual/reference/method/WriteResult/#WriteResult) object with [nMatched](https://docs.mongodb.com/manual/reference/method/WriteResult/#WriteResult.nMatched) and[nModified](https://docs.mongodb.com/manual/reference/method/WriteResult/#WriteResult.nModified) set to 1.

**2**

#### Undo the transaction on both accounts.

To undo the transaction on both accounts, reverse the transaction t if the transaction has been applied. In the update condition, include the condition pendingTransactions: t.\_id in order to update the account only if the pending transaction has been applied.

Update the destination account, subtracting from its balance the transaction value and removing the transaction \_id from the pendingTransactions array.

db.accounts.update(

{ \_id: t.destination, pendingTransactions: t.\_id },

{

$inc: { balance: -t.value },

$pull: { pendingTransactions: t.\_id }

}

)

Upon successful update, the method returns a [WriteResult()](https://docs.mongodb.com/manual/reference/method/WriteResult/#WriteResult) object with [nMatched](https://docs.mongodb.com/manual/reference/method/WriteResult/#WriteResult.nMatched) and [nModified](https://docs.mongodb.com/manual/reference/method/WriteResult/#WriteResult.nModified) set to 1. If the pending transaction has not been previously applied to this account, no document will match the update condition and [nMatched](https://docs.mongodb.com/manual/reference/method/WriteResult/#WriteResult.nMatched) and [nModified](https://docs.mongodb.com/manual/reference/method/WriteResult/#WriteResult.nModified) will be 0.

Update the source account, adding to its balance the transaction value and removing the transaction \_id from the pendingTransactions array.

db.accounts.update(

{ \_id: t.source, pendingTransactions: t.\_id },

{

$inc: { balance: t.value},

$pull: { pendingTransactions: t.\_id }

}

)

Upon successful update, the method returns a [WriteResult()](https://docs.mongodb.com/manual/reference/method/WriteResult/#WriteResult) object with [nMatched](https://docs.mongodb.com/manual/reference/method/WriteResult/#WriteResult.nMatched) and [nModified](https://docs.mongodb.com/manual/reference/method/WriteResult/#WriteResult.nModified) set to 1. If the pending transaction has not been previously applied to this account, no document will match the update condition and [nMatched](https://docs.mongodb.com/manual/reference/method/WriteResult/#WriteResult.nMatched) and [nModified](https://docs.mongodb.com/manual/reference/method/WriteResult/#WriteResult.nModified) will be 0.

**3**

#### Update transaction state to canceled.

To finish the rollback, update the transaction state from canceling to cancelled.

db.transactions.update(

{ \_id: t.\_id, state: "canceling" },

{

$set: { state: "cancelled" },

$currentDate: { lastModified: **true** }

}

)

Upon successful update, the method returns a [WriteResult()](https://docs.mongodb.com/manual/reference/method/WriteResult/#WriteResult) object with [nMatched](https://docs.mongodb.com/manual/reference/method/WriteResult/#WriteResult.nMatched) and[nModified](https://docs.mongodb.com/manual/reference/method/WriteResult/#WriteResult.nModified) set to 1.