**MongoDB Transactions**

MongoDB Transactions should be used in specific cases, when you absolutely need them: for example, because your application is aware of data consistency and atomicity. Transactions incur a greater performance cost over single document writes, so the denormalized data model will continue to be optimal in many cases and this helps to minimize the need for transactions.

Single writes are atomic by design: as long as you are able to embed documents in your collections you absolutely don’t need to use a transaction. Even so, transaction support is a very good and interesting feature that you can rely on in MongoDB from now on.

**MongoDB 4.0 provides fully ACID transactions support but remember:**

* multi-document transactions are available for replica set deployments only
  + you can use transactions even on a standalone server but you need to configure it as a replica set (with just one node)
* multi-document transactions are not available for sharded cluster
  + hopefully transactions will be available from version 4.2
* multi-document transactions are available for the Wired Tiger storage engine only

## ACID transactions in MongoDB 4.0

[ACID](https://urldefense.proofpoint.com/v2/url?u=https-3A__en.wikipedia.org_wiki_ACID-5F-28computer-5Fscience-29&d=DwMFaQ&c=j-EkbjBYwkAB4f8ZbVn1Fw&r=GgNNAm-b4BspomCvSnK34oPSD1DIwGTcDMtKTAecOiI&m=16YD70n8QIeLB9tjr_q5nGHfhSHnwsiAvSPLbZnXaNY&s=IQHVXJ1k4yjMwp7iswxN2XBgP_0FbQXcrISq9eZzA2I&e=) properties are well known in the world of relational databases and the acronym means.

* **Atomicity**: a group of commands inside the transaction must follow the “all or nothing” paradigm. If only one of the commands fails for any reason, the complete transaction fails as well.
* **Consistency**: if a transaction successfully executes, it will take the database from one state that is consistent to another state that is also consistent.
* **Isolation**: multiple transactions can run at the same time in the system. Isolation guarantees that each transaction is not able to view partial results of the others. Executing multiple transactions in parallel must have the same results as running them sequentially
* **Durability**: it guarantees that a transaction that has committed will remain persistent, even in the case of a system failure

## Limitations of transactions

The support for transactions introduced some limitations:

* a collection MUST exist in order to use transactions
* a collection cannot be created or dropped inside a transaction
* an index cannot be created or dropped inside a transaction
* non-CRUD operations are not permitted inside a transaction (for example, administrative commands like createUser are not permitted )
* a transaction cannot read or write in config, admin, and local databases
* a transaction cannot write to system.\* collections
* the size of a transaction is limited to 16MB
  + a single oplog entry is generated during the commit: the writes inside the transaction don’t have single oplog entries as in regular queries
  + the limitation is a consequence of the 16MB maximum size of any [BSON](https://urldefense.proofpoint.com/v2/url?u=https-3A__www.mongodb.com_json-2Dand-2Dbson&d=DwMFaQ&c=j-EkbjBYwkAB4f8ZbVn1Fw&r=GgNNAm-b4BspomCvSnK34oPSD1DIwGTcDMtKTAecOiI&m=16YD70n8QIeLB9tjr_q5nGHfhSHnwsiAvSPLbZnXaNY&s=3pKaE2P4hqOyBIUMItEKQZJQIJjCQsNQkxjkbZAVFBg&e=) document in the oplog
  + in case of larger transactions, you should consider splitting these into smaller transactions
* by default a transaction that executes for longer than 60 seconds will automatically expire
  + you can change this using the configuration parameter transactionLifetimeLimitSeconds
  + transactions rely on WiredTiger snapshot capability, and having a long running transaction can result in high pressure on WiredTiger’s cache to maintain snapshots, and lead to the retention of a lot of unflushed operations in memory

## Sessions

Sessions were deployed in version 3.6 in order to run the retryable writes (for example) but they are very important, too, for transactions. In fact any transaction is associated with an open session. Prior to starting a transaction, a session must be created. A transaction cannot be run outside a session.

At any given time you may have multiple running sessions in the system, but each session may run only a single transaction at a time. You can run transactions in parallel according to how many open sessions you have.

Three new commands were introduce for creating, committing, and aborting transactions:

* session.startTransaction()
  + starts a new transaction in the current session
* session.commitTransaction()
  + saves **consistently** and **durably** the changes made by the operations in the transaction
* session.abortTransaction()
  + the transaction ends without saving any of the changes made by the operations in the transaction

Note: in the following examples, we use two different connections to create two sessions. We do this for the sake of simplicity, but remember that you can create multiple sessions even inside a single connection, assigning each session to a different variable.

## Our first transaction

To test our first transaction if you don’t have a replica set already configured you can start a standalone server with –replSet option as shown below:

Shell

|  |  |
| --- | --- |
|  | [root@sd-0fa1-ff2c /]# mkdir d1 d2 d3  [root@sd-0fa1-ff2c /]# /opt/mongodb/bin/mongod --replSet ACID --dbpath d1/ --logpath ./27011.log --port 27011 --fork  about to fork child process, waiting until server is ready for connections.  forked process: 3185  child process started successfully, parent exiting  [root@sd-0fa1-ff2c /]#  [root@sd-0fa1-ff2c /]# /opt/mongodb/bin/mongod --replSet ACID --dbpath d2/ --logpath ./2702.log --port 27022 --fork  about to fork child process, waiting until server is ready for connections.  forked process: 3229  child process started successfully, parent exiting  [root@sd-0fa1-ff2c /]#  [root@sd-0fa1-ff2c /]# /opt/mongodb/bin/mongod --replSet ACID --dbpath d3/ --logpath ./27033.log --port 27033 --fork  about to fork child process, waiting until server is ready for connections.  forked process: 3272  child process started successfully, parent exiting  [root@sd-0fa1-ff2c /]#  [root@sd-0fa1-ff2c /]# /opt/mongodb/bin/mongo --port 27011 --quiet  # > rs.initiate()  {  "info2" : "no configuration specified. Using a default configuration for the set",  "me" : "localhost:27011",  "ok" : 1,  "operationTime" : Timestamp(1557128936, 1),  "$clusterTime" : {  "clusterTime" : Timestamp(1557128936, 1),  "signature" : {  "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAAAAAAAA="),  "keyId" : NumberLong(0)  }  }  }  # ACID:SECONDARY>  # ACID:PRIMARY>  # ACID:PRIMARY> rs.status()  {  "set" : "ACID",  "date" : ISODate("2019-05-06T07:49:12.869Z"),  "myState" : 1,  "term" : NumberLong(1),  "syncingTo" : "",  "syncSourceHost" : "",  "syncSourceId" : -1,  "heartbeatIntervalMillis" : NumberLong(2000),  "optimes" : {  "lastCommittedOpTime" : {  "ts" : Timestamp(1557128938, 5),  "t" : NumberLong(1)  },  "readConcernMajorityOpTime" : {  "ts" : Timestamp(1557128938, 5),  "t" : NumberLong(1)  },  "appliedOpTime" : {  "ts" : Timestamp(1557128938, 5),  "t" : NumberLong(1)  },  "durableOpTime" : {  "ts" : Timestamp(1557128938, 5),  "t" : NumberLong(1)  }  },  "lastStableCheckpointTimestamp" : Timestamp(1557128938, 4),  "members" : [  {  "\_id" : 0,  "name" : "localhost:27011",  "health" : 1,  "state" : 1,  "stateStr" : "PRIMARY",  "uptime" : 78,  "optime" : {  "ts" : Timestamp(1557128938, 5),  "t" : NumberLong(1)  },  "optimeDate" : ISODate("2019-05-06T07:48:58Z"),  "syncingTo" : "",  "syncSourceHost" : "",  "syncSourceId" : -1,  "infoMessage" : "could not find member to sync from",  "electionTime" : Timestamp(1557128936, 2),  "electionDate" : ISODate("2019-05-06T07:48:56Z"),  "configVersion" : 1,  "self" : true,  "lastHeartbeatMessage" : ""  }  ],  "ok" : 1,  "operationTime" : Timestamp(1557128938, 5),  "$clusterTime" : {  "clusterTime" : Timestamp(1557128938, 5),  "signature" : {  "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAAAAAAAA="),  "keyId" : NumberLong(0)  }  }  }  # ACID:PRIMARY>  # ACID:PRIMARY> rs.add("localhost:27022")  {  "ok" : 1,  "operationTime" : Timestamp(1557128985, 1),  "$clusterTime" : {  "clusterTime" : Timestamp(1557128985, 1),  "signature" : {  "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAAAAAAAA="),  "keyId" : NumberLong(0)  }  }  }  # ACID:PRIMARY> rs.add("localhost:27033")  {  "ok" : 1,  "operationTime" : Timestamp(1557128989, 1),  "$clusterTime" : {  "clusterTime" : Timestamp(1557128989, 1),  "signature" : {  "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAAAAAAAA="),  "keyId" : NumberLong(0)  }  }  }  # ACID:PRIMARY>  # ACID:PRIMARY> rs.status()  {  "set" : "ACID",  "date" : ISODate("2019-05-06T07:49:52.709Z"),  "myState" : 1,  "term" : NumberLong(1),  "syncingTo" : "",  "syncSourceHost" : "",  "syncSourceId" : -1,  "heartbeatIntervalMillis" : NumberLong(2000),  "optimes" : {  "lastCommittedOpTime" : {  "ts" : Timestamp(1557128989, 1),  "t" : NumberLong(1)  },  "readConcernMajorityOpTime" : {  "ts" : Timestamp(1557128989, 1),  "t" : NumberLong(1)  },  "appliedOpTime" : {  "ts" : Timestamp(1557128989, 1),  "t" : NumberLong(1)  },  "durableOpTime" : {  "ts" : Timestamp(1557128989, 1),  "t" : NumberLong(1)  }  },  "lastStableCheckpointTimestamp" : Timestamp(1557128938, 4),  "members" : [  {  "\_id" : 0,  "name" : "localhost:27011",  "health" : 1,  "state" : 1,  "stateStr" : "PRIMARY",  "uptime" : 118,  "optime" : {  "ts" : Timestamp(1557128989, 1),  "t" : NumberLong(1)  },  "optimeDate" : ISODate("2019-05-06T07:49:49Z"),  "syncingTo" : "",  "syncSourceHost" : "",  "syncSourceId" : -1,  "infoMessage" : "could not find member to sync from",  "electionTime" : Timestamp(1557128936, 2),  "electionDate" : ISODate("2019-05-06T07:48:56Z"),  "configVersion" : 3,  "self" : true,  "lastHeartbeatMessage" : ""  },  {  "\_id" : 1,  "name" : "localhost:27022",  "health" : 1,  "state" : 2,  "stateStr" : "SECONDARY",  "uptime" : 7,  "optime" : {  "ts" : Timestamp(1557128989, 1),  "t" : NumberLong(1)  },  "optimeDurable" : {  "ts" : Timestamp(1557128989, 1),  "t" : NumberLong(1)  },  "optimeDate" : ISODate("2019-05-06T07:49:49Z"),  "optimeDurableDate" : ISODate("2019-05-06T07:49:49Z"),  "lastHeartbeat" : ISODate("2019-05-06T07:49:51.949Z"),  "lastHeartbeatRecv" : ISODate("2019-05-06T07:49:50.953Z"),  "pingMs" : NumberLong(0),  "lastHeartbeatMessage" : "",  "syncingTo" : "localhost:27033",  "syncSourceHost" : "localhost:27033",  "syncSourceId" : 2,  "infoMessage" : "",  "configVersion" : 3  },  {  "\_id" : 2,  "name" : "localhost:27033",  "health" : 1,  "state" : 2,  "stateStr" : "SECONDARY",  "uptime" : 2,  "optime" : {  "ts" : Timestamp(1557128989, 1),  "t" : NumberLong(1)  },  "optimeDurable" : {  "ts" : Timestamp(1557128989, 1),  "t" : NumberLong(1)  },  "optimeDate" : ISODate("2019-05-06T07:49:49Z"),  "optimeDurableDate" : ISODate("2019-05-06T07:49:49Z"),  "lastHeartbeat" : ISODate("2019-05-06T07:49:51.951Z"),  "lastHeartbeatRecv" : ISODate("2019-05-06T07:49:52.469Z"),  "pingMs" : NumberLong(0),  "lastHeartbeatMessage" : "",  "syncingTo" : "localhost:27011",  "syncSourceHost" : "localhost:27011",  "syncSourceId" : 0,  "infoMessage" : "",  "configVersion" : 3  }  ],  "ok" : 1,  "operationTime" : Timestamp(1557128989, 1),  "$clusterTime" : {  "clusterTime" : Timestamp(1557128989, 1),  "signature" : {  "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAAAAAAAA="),  "keyId" : NumberLong(0)  }  }  }  # ACID:PRIMARY> |

**Create a new collection, and insert some data.**

[root@sd-0fa1-ff2c /]# /opt/mongodb/bin/mongo --port 27011 --quiet

# ACID:PRIMARY> use transactions\_db

switched to db transactions\_db

# ACID:PRIMARY> db.products.insertMany([{\_id:1,name:'T-Shirts',price:575},{\_id:2,name:'Shoes',price:3575},{\_id:3,name:'Sports',price:4575}])

{ "acknowledged" : true, "insertedIds" : [ 1, 2, 3 ] }

# ACID:PRIMARY>

# ACID:PRIMARY> db.products.find()

{ "\_id" : 1, "name" : "T-Shirts", "price" : 575 }

{ "\_id" : 2, "name" : "Shoes", "price" : 3575 }

{ "\_id" : 3, "name" : "Sports", "price" : 4575 }

# ACID:PRIMARY>

**Create a session**

# ACID:PRIMARY> session= db.getMongo().startSession()

session { "id" : UUID("69d7db78-0273-48ff-b1f5-aeb13feabce0") }

# ACID:PRIMARY>

# ACID:PRIMARY> session.startTransaction()

# ACID:PRIMARY>

# ACID:PRIMARY> session.getDatabase("transactions\_db").products.insert([{\_id:4,name:"Trousers", price: 2599 },{ \_id:5, name:"Accessories", price: 540}])

BulkWriteResult({

"writeErrors" : [ ],

"writeConcernErrors" : [ ],

"nInserted" : 2,

"nUpserted" : 0,

"nMatched" : 0,

"nModified" : 0,

"nRemoved" : 0,

"upserted" : [ ]

})

# ACID:PRIMARY> session.getDatabase("transactions\_db").products.find()

{ "\_id" : 1, "name" : "T-Shirts", "price" : 575 }

{ "\_id" : 2, "name" : "Shoes", "price" : 3575 }

{ "\_id" : 3, "name" : "Sports", "price" : 4575 }

{ "\_id" : 4, "name" : "Trousers", "price" : 2599 }

{ "\_id" : 5, "name" : "Accessories", "price" : 540 }

# ACID:PRIMARY>

# ACID:PRIMARY>

# ACID:PRIMARY> db.products.find()

{ "\_id" : 1, "name" : "T-Shirts", "price" : 575 }

{ "\_id" : 2, "name" : "Shoes", "price" : 3575 }

{ "\_id" : 3, "name" : "Sports", "price" : 4575 }

# ACID:PRIMARY>

As you might notice, since the transaction is not yet committed, you can see the modifications only from inside the session. You cannot see any of the modifications outside of the session, even in the same connection. If you try to open a new connection to the database, then you will not be able to see any of the modifications either.

Now, commit the transaction and see that you can now read the same data both inside and outside the session, as well as from any other connection.

# ACID:PRIMARY> session.commitTransaction()

# ACID:PRIMARY>

# ACID:PRIMARY> db.products.find()

{ "\_id" : 1, "name" : "T-Shirts", "price" : 575 }

{ "\_id" : 2, "name" : "Shoes", "price" : 3575 }

{ "\_id" : 3, "name" : "Sports", "price" : 4575 }

{ "\_id" : 4, "name" : "Trousers", "price" : 2599 }

{ "\_id" : 5, "name" : "Accessories", "price" : 540 }

# ACID:PRIMARY>

# ACID:PRIMARY> session.getDatabase("transactions\_db").products.find()

{ "\_id" : 1, "name" : "T-Shirts", "price" : 575 }

{ "\_id" : 2, "name" : "Shoes", "price" : 3575 }

{ "\_id" : 3, "name" : "Sports", "price" : 4575 }

{ "\_id" : 4, "name" : "Trousers", "price" : 2599 }

{ "\_id" : 5, "name" : "Accessories", "price" : 540 }

When the transaction is committed, all the data are written consistently and durably in the database, just like any typical write. So, writing to the journal file and to the oplog takes place in the same way it as for any single write that’s not inside a transaction. As long as the transaction is open, any modification is stored in memory.

## Isolation test

Let’s test now the isolation between two concurrent transactions.

Open the first connection, create a session and start a transaction:

And update the document on connection #1 to record with \_id: 1 and change something like below.  
Also update the same collection on connection #2 to record with \_id in 2,3,4 and change the pricing as shown below.

# ACID:PRIMARY>

# ACID:PRIMARY> session2.getDatabase("transactions\_db").products.updateMany({\_id: {$in:[2,3,4]}},{$set:{price:10999}})

{ "acknowledged" : true, "matchedCount" : 3, "modifiedCount" : 3 }

# ACID:PRIMARY> session2.getDatabase("transactions\_db").products.find()

{ "\_id" : 1, "name" : "T-Shirts", "price" : 999 }

{ "\_id" : 2, "name" : "Shoes", "price" : 10999 }

{ "\_id" : 3, "name" : "Sports", "price" : 10999 }

{ "\_id" : 4, "name" : "Trousers", "price" : 10999 }

{ "\_id" : 5, "name" : "Accessories", "price" : 540 }

# ACID:PRIMARY>

The two transactions are isolated, each one can see only the ongoing modifications that it has made itself.

Commit the transaction in connection #1:

# ACID:PRIMARY> db.products.find()

{ "\_id" : 1, "name" : "T-Shirts", "price" : 575 }

{ "\_id" : 2, "name" : "Shoes", "price" : 3575 }

{ "\_id" : 3, "name" : "Sports", "price" : 4575 }

{ "\_id" : 4, "name" : "Trousers", "price" : 2599 }

{ "\_id" : 5, "name" : "Accessories", "price" : 540 }

# ACID:PRIMARY>

# ACID:PRIMARY>

# ACID:PRIMARY> var session1 = db.getMongo().startSession()

# ACID:PRIMARY> session1.startTransaction()

# ACID:PRIMARY> session1.getDatabase("transactions\_db").products.update({\_id:1},{$set:{price:'TBD', status:'ON HOLD'}})

WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })

# ACID:PRIMARY> session1.getDatabase("transactions\_db").products.find()

{ "\_id" : 1, "name" : "T-Shirts", "price" : "TBD", "status" : "ON HOLD" }

{ "\_id" : 2, "name" : "Shoes", "price" : 3575 }

{ "\_id" : 3, "name" : "Sports", "price" : 4575 }

{ "\_id" : 4, "name" : "Trousers", "price" : 2599 }

{ "\_id" : 5, "name" : "Accessories", "price" : 540 }

# ACID:PRIMARY> session2.getDatabase("transactions\_db").products.find()

{ "\_id" : 1, "name" : "T-Shirts", "price" : 575 }

{ "\_id" : 2, "name" : "Shoes", "price" : 3575 }

{ "\_id" : 3, "name" : "Sports", "price" : 4575 }

{ "\_id" : 4, "name" : "Trousers", "price" : 2599 }

{ "\_id" : 5, "name" : "Accessories", "price" : 540 }

# ACID:PRIMARY> db.products.find()

{ "\_id" : 1, "name" : "T-Shirts", "price" : 575 }

{ "\_id" : 2, "name" : "Shoes", "price" : 3575 }

{ "\_id" : 3, "name" : "Sports", "price" : 4575 }

{ "\_id" : 4, "name" : "Trousers", "price" : 2599 }

{ "\_id" : 5, "name" : "Accessories", "price" : 540 }

# ACID:PRIMARY>

# ACID:PRIMARY>

# ACID:PRIMARY>

# ACID:PRIMARY> var session2 = db.getMongo().startSession()

# ACID:PRIMARY> session2.startTransaction()

# ACID:PRIMARY> session2.getDatabase("transactions\_db").products.updateMany({\_id: {$in:[2,3,4]}},{$set:{price:500, status:"DISCOUNTED\_SALE"}})

{ "acknowledged" : true, "matchedCount" : 3, "modifiedCount" : 3 }

# ACID:PRIMARY> session2.getDatabase("transactions\_db").products.find()

{ "\_id" : 1, "name" : "T-Shirts", "price" : 575 }

{ "\_id" : 2, "name" : "Shoes", "price" : 500, "status" : "DISCOUNTED\_SALE" }

{ "\_id" : 3, "name" : "Sports", "price" : 500, "status" : "DISCOUNTED\_SALE" }

{ "\_id" : 4, "name" : "Trousers", "price" : 500, "status" : "DISCOUNTED\_SALE" }

{ "\_id" : 5, "name" : "Accessories", "price" : 540 }

# ACID:PRIMARY> session1.getDatabase("transactions\_db").products.find()

{ "\_id" : 1, "name" : "T-Shirts", "price" : "TBD", "status" : "ON HOLD" }

{ "\_id" : 2, "name" : "Shoes", "price" : 3575 }

{ "\_id" : 3, "name" : "Sports", "price" : 4575 }

{ "\_id" : 4, "name" : "Trousers", "price" : 2599 }

{ "\_id" : 5, "name" : "Accessories", "price" : 540 }

# ACID:PRIMARY> db.products.find()

{ "\_id" : 1, "name" : "T-Shirts", "price" : 575 }

{ "\_id" : 2, "name" : "Shoes", "price" : 3575 }

{ "\_id" : 3, "name" : "Sports", "price" : 4575 }

{ "\_id" : 4, "name" : "Trousers", "price" : 2599 }

{ "\_id" : 5, "name" : "Accessories", "price" : 540 }

# ACID:PRIMARY>

# ACID:PRIMARY>

# ACID:PRIMARY>

# ACID:PRIMARY> session1.commitTransaction()

# ACID:PRIMARY> session1.getDatabase("transactions\_db").products.find()

{ "\_id" : 1, "name" : "T-Shirts", "price" : "TBD", "status" : "ON HOLD" }

{ "\_id" : 2, "name" : "Shoes", "price" : 3575 }

{ "\_id" : 3, "name" : "Sports", "price" : 4575 }

{ "\_id" : 4, "name" : "Trousers", "price" : 2599 }

{ "\_id" : 5, "name" : "Accessories", "price" : 540 }

# ACID:PRIMARY> session2.getDatabase("transactions\_db").products.find()

{ "\_id" : 1, "name" : "T-Shirts", "price" : 575 }

{ "\_id" : 2, "name" : "Shoes", "price" : 500, "status" : "DISCOUNTED\_SALE" }

{ "\_id" : 3, "name" : "Sports", "price" : 500, "status" : "DISCOUNTED\_SALE" }

{ "\_id" : 4, "name" : "Trousers", "price" : 500, "status" : "DISCOUNTED\_SALE" }

{ "\_id" : 5, "name" : "Accessories", "price" : 540 }

# ACID:PRIMARY> db.products.find()

{ "\_id" : 1, "name" : "T-Shirts", "price" : "TBD", "status" : "ON HOLD" }

{ "\_id" : 2, "name" : "Shoes", "price" : 3575 }

{ "\_id" : 3, "name" : "Sports", "price" : 4575 }

{ "\_id" : 4, "name" : "Trousers", "price" : 2599 }

{ "\_id" : 5, "name" : "Accessories", "price" : 540 }

# ACID:PRIMARY>

As you can see the second transaction still sees its own modifications, and cannot see the already committed updates of the other transaction. This kind of isolation works the same as the “REPEATABLE READ” level of MySQL and other relational databases.

Now commit the transaction in connection #2 and see the new values of the collection:

# ACID:PRIMARY>

# ACID:PRIMARY> session2.commitTransaction()

# ACID:PRIMARY> session1.getDatabase("transactions\_db").products.find()

{ "\_id" : 1, "name" : "T-Shirts", "price" : "TBD", "status" : "ON HOLD" }

{ "\_id" : 2, "name" : "Shoes", "price" : 500, "status" : "DISCOUNTED\_SALE" }

{ "\_id" : 3, "name" : "Sports", "price" : 500, "status" : "DISCOUNTED\_SALE" }

{ "\_id" : 4, "name" : "Trousers", "price" : 500, "status" : "DISCOUNTED\_SALE" }

{ "\_id" : 5, "name" : "Accessories", "price" : 540 }

# ACID:PRIMARY> session2.getDatabase("transactions\_db").products.find()

{ "\_id" : 1, "name" : "T-Shirts", "price" : "TBD", "status" : "ON HOLD" }

{ "\_id" : 2, "name" : "Shoes", "price" : 500, "status" : "DISCOUNTED\_SALE" }

{ "\_id" : 3, "name" : "Sports", "price" : 500, "status" : "DISCOUNTED\_SALE" }

{ "\_id" : 4, "name" : "Trousers", "price" : 500, "status" : "DISCOUNTED\_SALE" }

{ "\_id" : 5, "name" : "Accessories", "price" : 540 }

# ACID:PRIMARY> db.products.find()

{ "\_id" : 1, "name" : "T-Shirts", "price" : "TBD", "status" : "ON HOLD" }

{ "\_id" : 2, "name" : "Shoes", "price" : 500, "status" : "DISCOUNTED\_SALE" }

{ "\_id" : 3, "name" : "Sports", "price" : 500, "status" : "DISCOUNTED\_SALE" }

{ "\_id" : 4, "name" : "Trousers", "price" : 500, "status" : "DISCOUNTED\_SALE" }

{ "\_id" : 5, "name" : "Accessories", "price" : 540 }

# ACID:PRIMARY>

# ACID:PRIMARY>

## Conflicts

When two (or more) concurrent transactions modify the same documents, we may have a conflict. MongoDB can detect a conflict immediately, even while transactions are not yet committed. The first transaction to acquire the lock on a document will continue, the second one will receive the conflict error message and fail. The failed transaction can then be retried later.

Let’s see an example.

Create a new transaction in connection #1 to update. We want to change the status

# ACID:PRIMARY> db.products.find()

{ "\_id" : 1, "name" : "T-Shirts", "price" : "TBD", "status" : "DISCOUNTED\_SALE" }

{ "\_id" : 2, "name" : "Shoes", "price" : 500, "status" : "DISCOUNTED\_SALE" }

{ "\_id" : 3, "name" : "Sports", "price" : 500, "status" : "DISCOUNTED\_SALE" }

{ "\_id" : 4, "name" : "Trousers", "price" : 500, "status" : "DISCOUNTED\_SALE" }

{ "\_id" : 5, "name" : "Accessories", "price" : 540, "status" : "DISCOUNTED\_SALE" }

# ACID:PRIMARY>

# ACID:PRIMARY>

# ACID:PRIMARY> var session6 = db.getMongo().startSession()

# ACID:PRIMARY> session6.startTransaction()

# ACID:PRIMARY> session6.getDatabase("transactions\_db").products.updateMany({status:"DISCOUNTED\_SALE"},{$set:{status:"NO\_DISCOUNT"}})

{ "acknowledged" : true, "matchedCount" : 5, "modifiedCount" : 5 }

# ACID:PRIMARY> db.products.find()

{ "\_id" : 1, "name" : "T-Shirts", "price" : "TBD", "status" : "DISCOUNTED\_SALE" }

{ "\_id" : 2, "name" : "Shoes", "price" : 500, "status" : "DISCOUNTED\_SALE" }

{ "\_id" : 3, "name" : "Sports", "price" : 500, "status" : "DISCOUNTED\_SALE" }

{ "\_id" : 4, "name" : "Trousers", "price" : 500, "status" : "DISCOUNTED\_SALE" }

{ "\_id" : 5, "name" : "Accessories", "price" : 540, "status" : "DISCOUNTED\_SALE" }

# ACID:PRIMARY> session6.getDatabase("transactions\_db").products.find()

{ "\_id" : 1, "name" : "T-Shirts", "price" : "TBD", "status" : "NO\_DISCOUNT" }

{ "\_id" : 2, "name" : "Shoes", "price" : 500, "status" : "NO\_DISCOUNT" }

{ "\_id" : 3, "name" : "Sports", "price" : 500, "status" : "NO\_DISCOUNT" }

{ "\_id" : 4, "name" : "Trousers", "price" : 500, "status" : "NO\_DISCOUNT" }

{ "\_id" : 5, "name" : "Accessories", "price" : 540, "status" : "NO\_DISCOUNT" }

# ACID:PRIMARY>

# ACID:PRIMARY>

# ACID:PRIMARY> var session7 = db.getMongo().startSession()

# ACID:PRIMARY> session7.startTransaction()

# ACID:PRIMARY> session7.getDatabase("transactions\_db").products.updateMany({status:"DISCOUNTED\_SALE"},{$set:{status:"50%-DISCOUNT"}})

2019-05-06T10:04:28.955-0400 E QUERY [js] WriteCommandError: WriteConflict :

WriteCommandError({

"errorLabels" : [

"TransientTransactionError"

],

"operationTime" : Timestamp(1557151459, 1),

"ok" : 0,

"errmsg" : "WriteConflict",

"code" : 112,

"codeName" : "WriteConflict",

"$clusterTime" : {

"clusterTime" : Timestamp(1557151459, 1),

"signature" : {

"hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAAAAAAAA="),

"keyId" : NumberLong(0)

}

}

})

WriteCommandError@src/mongo/shell/bulk\_api.js:420:48

Bulk/executeBatch@src/mongo/shell/bulk\_api.js:902:1

Bulk/this.execute@src/mongo/shell/bulk\_api.js:1150:21

DBCollection.prototype.updateMany@src/mongo/shell/crud\_api.js:655:17

@(shell):1:1

# ACID:PRIMARY> session6.getDatabase("transactions\_db").products.find()

{ "\_id" : 1, "name" : "T-Shirts", "price" : "TBD", "status" : "NO\_DISCOUNT" }

{ "\_id" : 2, "name" : "Shoes", "price" : 500, "status" : "NO\_DISCOUNT" }

{ "\_id" : 3, "name" : "Sports", "price" : 500, "status" : "NO\_DISCOUNT" }

{ "\_id" : 4, "name" : "Trousers", "price" : 500, "status" : "NO\_DISCOUNT" }

{ "\_id" : 5, "name" : "Accessories", "price" : 540, "status" : "NO\_DISCOUNT" }

# ACID:PRIMARY> db.products.find()

{ "\_id" : 1, "name" : "T-Shirts", "price" : "TBD", "status" : "DISCOUNTED\_SALE" }

{ "\_id" : 2, "name" : "Shoes", "price" : 500, "status" : "DISCOUNTED\_SALE" }

{ "\_id" : 3, "name" : "Sports", "price" : 500, "status" : "DISCOUNTED\_SALE" }

{ "\_id" : 4, "name" : "Trousers", "price" : 500, "status" : "DISCOUNTED\_SALE" }

{ "\_id" : 5, "name" : "Accessories", "price" : 540, "status" : "DISCOUNTED\_SALE" }

# ACID:PRIMARY>

# ACID:PRIMARY>

# ACID:PRIMARY>

# ACID:PRIMARY> session6.getDatabase("transactions\_db").products.find()

{ "\_id" : 1, "name" : "T-Shirts", "price" : "TBD", "status" : "NO\_DISCOUNT" }

{ "\_id" : 2, "name" : "Shoes", "price" : 500, "status" : "NO\_DISCOUNT" }

{ "\_id" : 3, "name" : "Sports", "price" : 500, "status" : "NO\_DISCOUNT" }

{ "\_id" : 4, "name" : "Trousers", "price" : 500, "status" : "NO\_DISCOUNT" }

{ "\_id" : 5, "name" : "Accessories", "price" : 540, "status" : "NO\_DISCOUNT" }

# ACID:PRIMARY> session7.getDatabase("transactions\_db").products.find()

Error: error: {

"errorLabels" : [

"TransientTransactionError"

],

"operationTime" : Timestamp(1557151469, 1),

"ok" : 0,

"errmsg" : "Transaction 0 has been aborted.",

"code" : 251,

"codeName" : "NoSuchTransaction",

"$clusterTime" : {

"clusterTime" : Timestamp(1557151469, 1),

"signature" : {

"hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAAAAAAAA="),

"keyId" : NumberLong(0)

}

}

}

# ACID:PRIMARY>

We received an error and the transaction failed. We can retry it later.

## Other details

* the individual writes inside the transaction are not retry-able even if retryWrites is set to true
* each commit operation is a retry-able write operation regardless of whether retryWrites is set to true. The drivers retry the commit a single time in case of an error.
* [**Read Concern**](https://urldefense.proofpoint.com/v2/url?u=https-3A__docs.mongodb.com_v4.0_reference_read-2Dconcern_&d=DwMFaQ&c=j-EkbjBYwkAB4f8ZbVn1Fw&r=GgNNAm-b4BspomCvSnK34oPSD1DIwGTcDMtKTAecOiI&m=16YD70n8QIeLB9tjr_q5nGHfhSHnwsiAvSPLbZnXaNY&s=aJbTqk0sz5VmxbvChEttZesN2eNHFsj0FMFaktk14Vs&e=) supports snapshot, local and majority values
* [**Write Concern**](https://urldefense.proofpoint.com/v2/url?u=https-3A__docs.mongodb.com_v4.0_reference_write-2Dconcern_&d=DwMFaQ&c=j-EkbjBYwkAB4f8ZbVn1Fw&r=GgNNAm-b4BspomCvSnK34oPSD1DIwGTcDMtKTAecOiI&m=16YD70n8QIeLB9tjr_q5nGHfhSHnwsiAvSPLbZnXaNY&s=OkNXM75JQgFsX1vVDMWqjg2IiA5LMKFsk-KyKbPmnyM&e=) can be set at the transaction level. The individual operations inside the transaction ignore the write concern. Write concern is evaluated during the commit
* [**Read Preference**](https://urldefense.proofpoint.com/v2/url?u=https-3A__docs.mongodb.com_v4.0_reference_read-2Dpreference_index.html&d=DwMFaQ&c=j-EkbjBYwkAB4f8ZbVn1Fw&r=GgNNAm-b4BspomCvSnK34oPSD1DIwGTcDMtKTAecOiI&m=16YD70n8QIeLB9tjr_q5nGHfhSHnwsiAvSPLbZnXaNY&s=HOCYE7ZPG1NftmyWC6Tg83AwFXeb4fER5w-mlehgEAc&e=) supports only primary value

## Conclusions

Transaction support in MongoDB 4.0 is a very interesting new feature, but it isn’t fully mature yet, there are strong limitations at this stage: a transaction cannot be larger than 16MB, you cannot use it on sharded clusters and others. If you absolutely need a transaction in your application use it. But don’t use transactions only because they are cool, since in some cases a proper data model based on embedding documents in collections and denormalizing your data could be the best solution. MongoDB isn’t by its nature a relational database; as long as you are able to model your data keeping in mind that it’s a NOSQL database you should avoid using transactions. In specific cases, or if you already have a database with strong “informal relations” between the collections that you cannot change, then you could choose to rely on transactions.