Use multi-document ACID transactions in MongoDB 4.0

November 7th 2018

Corrado Pandiani - Senior consultant
Percona

Thank You Sponsors

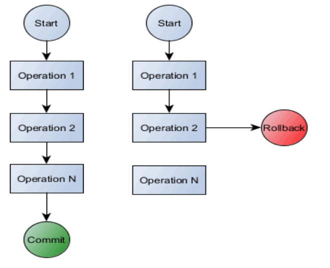
Shannon Systems

About me

really sorry for my face

- Italian (yes, I love spaghetti, pizza and espresso)
- 22 years spent in designing, developing and administering web sites, mainly in the football industry
- Joined Percona on February 2018 as a Senior Consultant
- MySQL / MongoDB DBA
- Perl/PHP/Javascript developer (but also some other languages)
- Open Source enthusiast
- 1 wife, 3 male kids, 1 maltese dog, 2 goldfishes
- Piano, synthesizers, pipe organ and bass guitar player

What is a transaction



all or nothing

- Legacy feature in relational databases
- A **transaction** symbolizes a unit of work performed within a database management system (or similar system) against a database, and treated in a coherent and reliable way independent of other **transactions**. A **transaction** generally represents any change in a database. (Wikipedia)

ACID

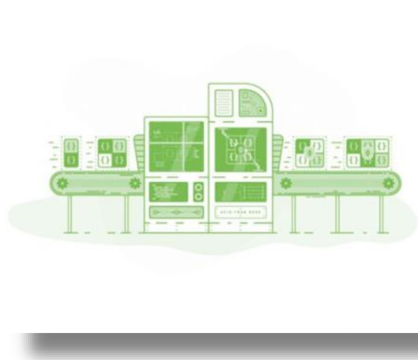
- **Atomicity :** guarantees that each transaction is treated as a single "unit", which either succeeds completely, or fails completely
- **Consistency :** ensures that a transaction can only bring the database from one valid state to another
- **Isolation :** ensures that concurrent execution of transactions leaves the database in the same state that would have been obtained if the transactions were executed sequentially
- **Durability :** guarantees that once a transaction has been committed, it will remain committed even in the case of a system failure

MongoDB 4.0 - transactions

new amazing features

- First release to provide multi-document ACID transactions
- New concept for a document-based database
- Multi-document transactions are available for Replica Sets only
  - standalone as well, but you need to configure it as RS
- Multi-document transactions for sharded clusters are not available. Scheduled for version 4.2
- Multi-document transactions are available for WiredTiger storage engine only

MongoDB 4.0 - transactions



new amazing features

- A transaction executes only on the PRIMARY
- In-memory
- After committing the replication takes place as usual
- Journal must be enabled for durability
  - Always enabled in any case
- Isolation is guaranteed by WiredTiger snapshot
- Setting WriteConcern to `majority` is required for data consistency
- Exclusive locks on the documents

PERCONA LIVE EUROPE FRANKFURT

Important notes

- Multi-document transaction incurs a greater performance cost over single document writes
- Multi-document transactions should not be a replacement for effective schema design
- For many scenarios, modeling your data appropriately will minimize the need for multi-document transactions, denormalized data model will continue to be optimal
- Remember that single document writes are atomic

PERCONA LIVE EUROPE FRANKFURT

Limitations

- A collection MUST exists 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 cannot be used inside a transaction; for example stuffs like `createUser`, `getParameter`, etc.
- Cannot read/write in `config`, `admin` and `local` databases
- Cannot write to `system.*` collections

Limitations

- A single transaction is limited to 16MB
  - The same for BSON objects and oplog entries
  - Larger transactions need to be splitted into smaller transactions

Is my app good for transactions ?

- Yes it is if
  - you have a lot of 1:N and/or N:N relationships between different collections
  - you are aware of data consistency because your app needs to be
  - You manage commercial/financial or really sensitive data
- No it’s not if
  - you shouldn’t be aware of data consistency
  - you can achieve as well your goals embedding documents and denormalizing

Sessions

- Transactions are associated with a session
- In order to use a transaction you must create a session at first
- When using a driver to connect to mongod, the session id must be passed to each operation in the transaction
- At any given time you can have only a single open transaction
- If a session ends for any reason the open transaction is automatically aborted
- Sessions were introduced in version 3.6

Commands available

- `Session.startTransaction()`
  - Starts a multi-document transaction associated with the session
- `Session.commitTransaction()`
  - Saves the changes made by the operations in the multi-document transaction and ends the transaction
- `Session.abortTransaction()`
  - The transaction ends without saving any of the changes made by the operations in the transaction

Our first transaction

Start mongod

- Start your Replica Set environment
- Even with a standalone host the Replica Set must be configured and initiated

```
#> mongod --dbpath /data/db40 --logpath /data/log40.log --
fork --replSet foo
```

Connection #1: create a collection and insert data

```
foo:PRIMARY> use percona
switched to db percona
foo:PRIMARY> db.createCollection('ple18')
{
    "ok" : 1,
    "operationTime" : Timestamp(1538483120, 1),
    "$clusterTime" : {
            "clusterTime" : Timestamp(1538483120, 1),
            "signature" : {
                    "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAAAAAAAA="),
                    "keyId" : NumberLong(0)
            }
    }
}
foo:PRIMARY> db.ple18.insert([{_id:1, name:"Corrado"},{_id:2, name:"Peter"},{_id:3,
name:"Heidi"}])
BulkWriteResult({
    "writeErrors" : [ ],
    "writeConcernErrors" : [ ],
    "nInserted" : 3,
    "nUpserted" : 0,
    "nMatched" : 0,
    "nModified" : 0,
    "nRemoved" : 0,
    "upserted" : [ ]
})
```

PERCONA LIVE EUROPE FRANKFURT

Connection #1: create a transaction and insert a new document

```
foo:PRIMARY> session = db.getMongo().startSession()
session { "id" : UUID("dcfa7de5-527d-4b1c-a890-53c9a355920d")
}

foo:PRIMARY> session.startTransaction()

foo:PRIMARY>
session.getDatabase("percona").ple18.insert([{_id: 4 , name :
"George"},{_id: 5, name: "Tom"}])
WriteResult({ "nInserted" : 2 })
```

PERCONA LIVE EUROPE FRANKFURT

Connection #1: read the documents from the collection

```
foo:PRIMARY> session.getDatabase("percona").ple18.find()
{ "_id" : 1, "name" : "Corrado" }
{ "_id" : 2, "name" : "Peter" }
{ "_id" : 3, "name" : "Heidi" }
{ "_id" : 4, "name" : "George" }
{ "_id" : 5, "name" : "Tom" }

foo:PRIMARY> db.ple18.find()
{ "_id" : 1, "name" : "Corrado" }
{ "_id" : 2, "name" : "Peter" }
{ "_id" : 3, "name" : "Heidi" }
```

The transaction is not yet committed: inserted and updated documents are visible only inside the session. Even in the same connection.

PERCONA LIVE EUROPE FRANKFURT

Connection #2: open a new connection and read the documents

```
foo:PRIMARY> use percona
switched to db percona

foo:PRIMARY> db.ple18.find()
{ "_id" : 1, "name" : "Corrado" }
{ "_id" : 2, "name" : "Peter" }
{ "_id" : 3, "name" : "Heidi" }
```

Connection #1: commit the transaction

```
foo:PRIMARY> session.commitTransaction()

foo:PRIMARY> session.getDatabase("percona").ple18.find()
{ "_id" : 1, "name" : "Corrado" }
{ "_id" : 2, "name" : "Peter" }
{ "_id" : 3, "name" : "Heidi" }
{ "_id" : 4, "name" : "George" }
{ "_id" : 5, "name" : "Tom" }

foo:PRIMARY> db.ple18.find()
{ "_id" : 1, "name" : "Corrado" }
{ "_id" : 2, "name" : "Peter" }
{ "_id" : 3, "name" : "Heidi" }
{ "_id" : 4, "name" : "George" }
{ "_id" : 5, "name" : "Tom" }
```

Connection #2: read the collection

```
foo:PRIMARY> db.ple18.find()
{ "_id" : 1, "name" : "Corrado" }
{ "_id" : 2, "name" : "Peter" }
{ "_id" : 3, "name" : "Heidi" }
{ "_id" : 4, "name" : "George" }
{ "_id" : 5, "name" : "Tom" }
```

Now we can see the effect of the committed transaction in all other connections.

Isolation test

Connection #1: create a new transaction for updating data

```
foo:PRIMARY> var session1 = db.getMongo().startSession()

foo:PRIMARY> session1.startTransaction()

foo:PRIMARY>
session1.getDatabase("percona").ple18.update({_id:3},{$set:{ gender:
"F" }})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })

foo:PRIMARY> session1.getDatabase("percona").ple18.find()
{ "_id" : 1, "name" : "Corrado" }
{ "_id" : 2, "name" : "Peter" }
{ "_id" : 3, "name" : "Heidi", "gender" : "F" }
{ "_id" : 4, "name" : "George" }
{ "_id" : 5, "name" : "Tom" }
```

PERCONA LIVE EUROPE FRANKFURT

Connection #2: create a new transaction for updating data

```
foo:PRIMARY> var session2 = db.getMongo().startSession()

foo:PRIMARY> session2.startTransaction()

foo:PRIMARY> session2.getDatabase("percona").ple18.update({_id:{$in:
[1,2,4,5]}},{$set:{ gender: "M" }},{multi:"true"})
WriteResult({ "nMatched" : 4, "nUpserted" : 0, "nModified" : 4 })

foo:PRIMARY> session2.getDatabase("percona").ple18.find()
{ "_id" : 1, "name" : "Corrado", "gender" : "M" }
{ "_id" : 2, "name" : "Peter", "gender" : "M" }
{ "_id" : 3, "name" : "Heidi" }
{ "_id" : 4, "name" : "George", "gender" : "M" }
{ "_id" : 5, "name" : "Tom", "gender" : "M" }
```

Connection #1: commit the transaction

```
foo:PRIMARY> session1.commitTransaction()

foo:PRIMARY> session1.getDatabase("percona").ple18.find()
{ "_id" : 1, "name" : "Corrado" }
{ "_id" : 2, "name" : "Peter" }
{ "_id" : 3, "name" : "Heidi", "gender" : "F" }
{ "_id" : 4, "name" : "George" }
{ "_id" : 5, "name" : "Tom" }
```

Connection #2: commit the transaction

```
foo:PRIMARY> session2.commitTransaction()

foo:PRIMARY> session2.getDatabase("percona").ple18.find()
{ "_id" : 1, "name" : "Corrado", "gender" : "M" }
{ "_id" : 2, "name" : "Peter", "gender" : "M" }
{ "_id" : 3, "name" : "Heidi", "gender" : "F" }
{ "_id" : 4, "name" : "George", "gender" : "M" }
{ "_id" : 5, "name" : "Tom", "gender" : "M" }
```

Conflicts

Connection #1: create a new transaction for updating data

Let's try to create two concurrent transaction that modify the same document

```
foo:PRIMARY> session.startTransaction()

foo:PRIMARY>
session.getDatabase("percona").ple18.update({name:"Heidi"},{$set:{na
me:"Luise"}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

PERCONA LIVE EUROPE FRANKFURT

Connection #2: create a new transaction for updating data

Let's try to modify the same document. The conflict is triggered before the commit.

```
foo:PRIMARY> session.startTransaction()
foo:PRIMARY> session.getDatabase("percona").ple18.update({name:"Heidi"},{$set:{name:"Marie"}})
WriteCommandError({
	"errorLabels" : [
		"TransientTransactionError"
	],
	"operationTime" : Timestamp(1538495683, 1),
	"ok" : 0,
	"errmsg" : "WriteConflict",
	"code" : 112,
	"codeName" : "WriteConflict",
	"$clusterTime" : {
		"clusterTime" : Timestamp(1538495683, 1),
		"signature" : {
			"hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAAAAAAAA="),
			"keyId" : NumberLong(0)
		}
	}
})
```

PERCONA LIVE EUROPE FRANKFURT

Transaction on different collections

PERCONA LIVE EUROPE FRANKFURT

Connection #1: create a new collection

```
foo:PRIMARY> db.createCollection("presentation")

foo:PRIMARY> db.presentation.insert( [{ _id:1, title:"Amazing
Transactions" }, { _id:2, title:"MongoDB for dummies" } ])

foo:PRIMARY> db.presentation.find()
{ "_id" : 1, "title" : "Amazing Transactions" }
{ "_id" : 2, "title" : "MongoDB for dummies" }
```

Connection #1: start a transaction to modify 2 different collections

```
foo:PRIMARY> session1.startTransaction()

foo:PRIMARY> session1.getDatabase("percona").ple18.insert( [ {
_id:6, name: "Bruce" }, {_id:7, name: "Steve" } ])
BulkWriteResult({
    "writeErrors" : [ ],
    "writeConcernErrors" : [ ],
    "nInserted" : 2,
    …

foo:PRIMARY> session1.getDatabase("percona").presentation.update(
{_id:1}, { $set : { attendees: [ 6, 7 ]  }  }  )
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

PERCONA LIVE EUROPE FRANKFURT

Connection #2: you shouldn’t see the modifications

```
foo:PRIMARY> db.ple18.find()
{ "_id" : 1, "name" : "Corrado", "gender" : "M" }
{ "_id" : 2, "name" : "Peter", "gender" : "M" }
{ "_id" : 3, "name" : "Heidi", "gender" : "F" }
{ "_id" : 4, "name" : "George", "gender" : "M" }
{ "_id" : 5, "name" : "Tom", "gender" : "M" }

foo:PRIMARY> db.presentation.find()
{ "_id" : 1, "title" : "Amazing Transactions" }
{ "_id" : 2, "title" : "MongoDB for dummies" }
```

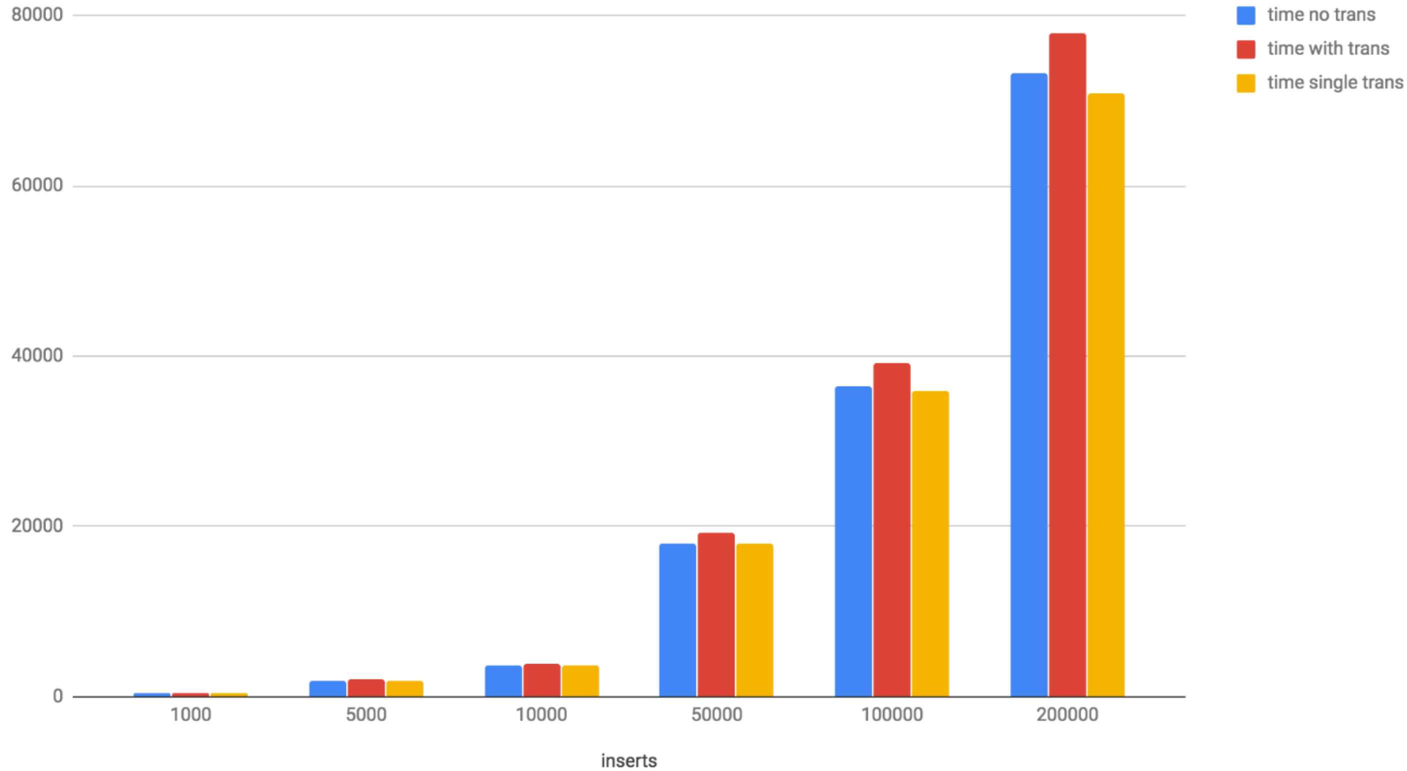
Connection #1: commit and check

```
foo:PRIMARY> session1.commitTransaction()

foo:PRIMARY> db.ple18.find()
{ "_id" : 1, "name" : "Corrado", "gender" : "M" }
{ "_id" : 2, "name" : "Peter", "gender" : "M" }
{ "_id" : 3, "name" : "Heidi", "gender" : "F" }
{ "_id" : 4, "name" : "George", "gender" : "M" }
{ "_id" : 5, "name" : "Tom", "gender" : "M" }
{ "_id" : 6, "name" : "Bruce" }
{ "_id" : 7, "name" : "Steve" }

foo:PRIMARY> db.presentation.find()
{ "_id" : 1, "title" : "Amazing Transactions", "attendees" : [ 6, 7 ] }
{ "_id" : 2, "title" : "MongoDB for dummies" }
```

Trivial
Benchmark tests

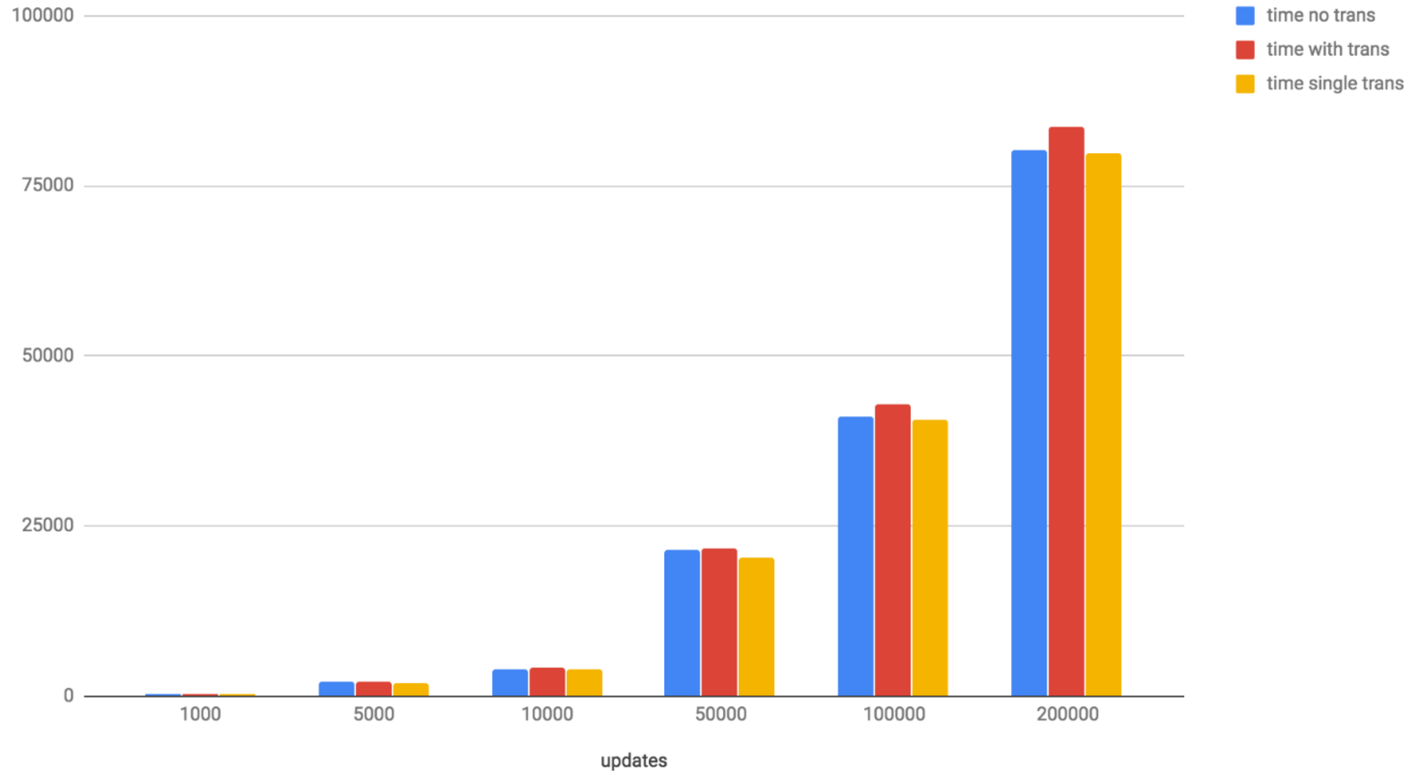


Test environment

Mac OSX
3.1GHz Intel i5 2 cores
8GB RAM
SSD disk

Time in milliseconds

PERCONA LIVE EUROPE FRANKFURT



Test environment

Mac OSX
3.1GHz Intel i5 2 cores
8GB RAM
SSD disk

Time in milliseconds

PERCONA LIVE EUROPE FRANKFURT

Other details

Other details

- The individual writes inside the transaction are not retryable, regardless of whether `retryWrites` is set to true. Look at the manual for helper functions.
- The commit operations are retryable write operations. If the commit operation encounters an error, MongoDB drivers retry the operation a single time regardless of whether `retryWrites` is set to true.
  - Look at the manual for helper functions.
- **Read Concern**: `snapshot`, `local` and `majority` are supported
- **Write Concern**: you can set the WC at the transaction level, not on the individual operation. At the time of the commit, transactions use the transaction level WC to commit all the writes. Individual operations inside the transaction ignore WC.
- **Read Preference**: must use `primary`.

PERCONA LIVE EUROPE FRANKFURT

Thank you

corrado.pandiani@percona.com
https://www.percona.com/blog/author/corrado-pandiani/