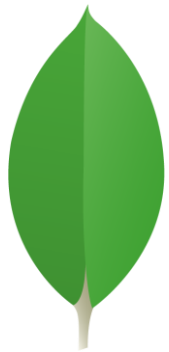


mongoDB

Bộ môn: Kỹ Thuật Phần Mềm

Giáo viên: Trần Thế Trung.
Email: tranthetrung@iuh.edu.vn

Transactions in MongoDB

1. Transactions in MongoDB.
2. Read Preference.
3. Read Concern.
4. Write Concern

```
const session = db.getMongo().startSession();
session.startTransaction();

try {
  session.getDatabase("shop").orders.insertOne({ item: "Book", qty: 2 });
  session.getDatabase("shop").inventory.updateOne(
    { item: "Book" },
    { $inc: { stock: -2 } }
  );

  session.commitTransaction();   // Lưu thay đổi
} catch (error) {
  session.abortTransaction();    // Rollback
}
session.endSession();
```

What is a transaction?

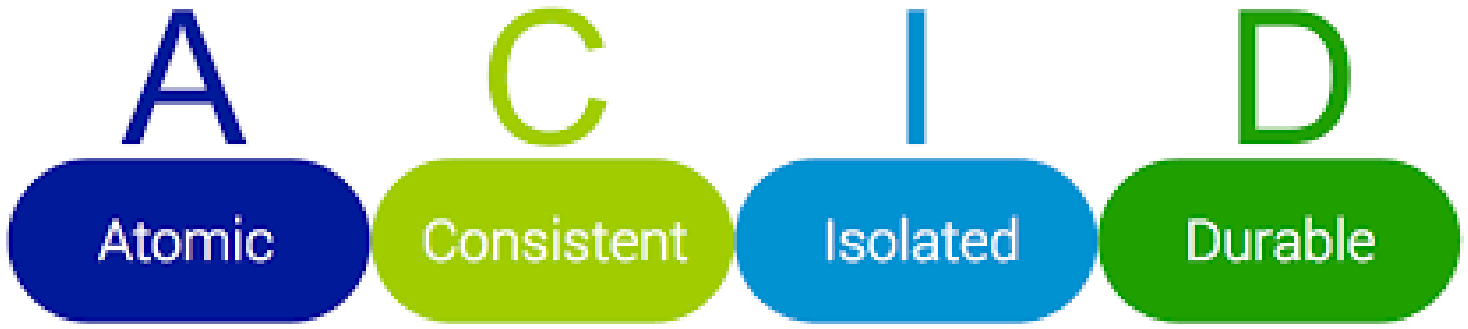
- **A database transaction** is a **unit of work**, designed to handle the changes of data in the database. It makes sure that the output of the data is **consistent** and **doesn't generate errors**. It helps with concurrent changes to the database, and makes the database more scalable.
- **A transaction** is a set of database operations *(reads and writes)* performed in a sequential order, all individual operations must succeed.
- If any operation is not executed correctly, the transaction will be aborted. The database will then be restored to its previous state

Transactions in MongoDB

- In MongoDB, an operation on a single document is **atomic**.
- For situations that require atomicity of reads and writes to multiple documents *(in a single or multiple collections)*, MongoDB supports distributed transactions. With distributed transactions, transactions can be used across multiple *operations, collections, databases, documents, and shards*.

Transactions in MongoDB

- Distributed transactions are **Atomic**:
  - When a transaction **commits**, all data changes made in the transaction **are saved** and visible outside the transaction.
  - When a transaction writes to **multiple shards**, not all outside read operations need to wait for the result of the committed transaction to be visible across the shards.
  - When a **transaction aborts**, all data changes made in the transaction are **discarded** without ever becoming visible.



Transactions and ACID in MogoDB

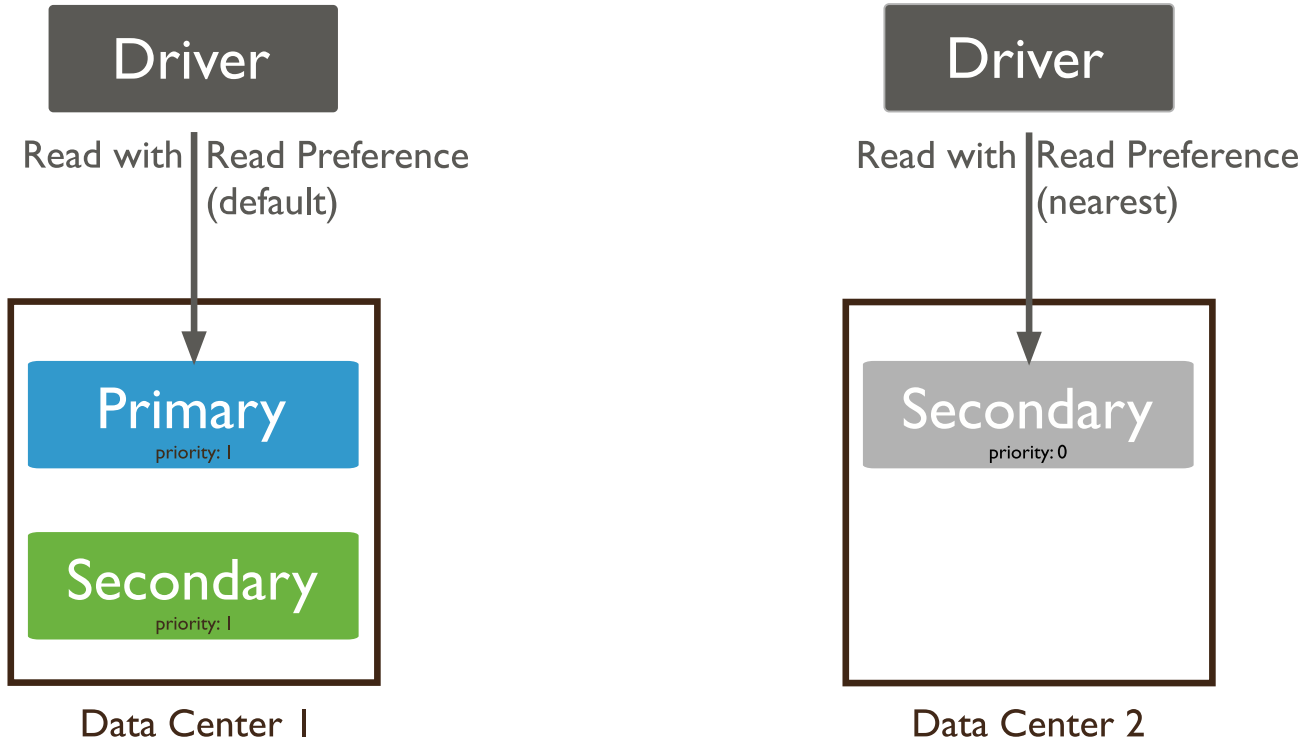
- ACID in MogoDB is a group of database operations that must happen together or not at all, ensuring database safety and consistency.
- ACID Transactions should be used in scenarios that involve the transfer of value from one record to another *(such as when exchanging currency, stock or even adding an item to an online shopping cart)*

ACID properties of transactions

- **A**tomicity: all operations will either succeed or fail together.
- **C**onsistency: all changes made by operations are consistent with database constraints.
  - *For example, in an application that transfers funds from one account to another, the consistency property ensures that the total value of funds in both the accounts is the same at the start and end of each.*
- **I**solation: multiple transactions can happen at the same time without affecting the outcome of the other transaction.
  - *For example, in an application that transfers funds from one account to another, the isolation property ensures that another transaction sees the transferred funds in one account or the other, but not in both, nor in neither.*
- **D**urability: all of the changes that are made by operations in a transaction with persist, no matter what.
  - *For example, in an application that transfers funds from one account to another, the durability property ensures that the changes made to each account will not be reversed.*

Transactions and Read Preference

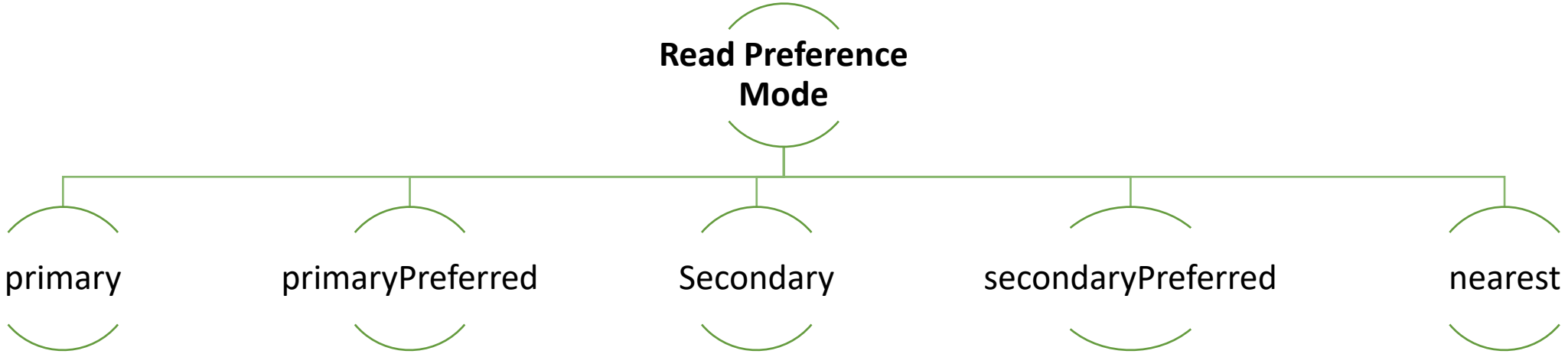
Read preference describes how MongoDB clients route read operations to the members of a replica set.



Transactions and Read Preference

Set the transaction-level read preference at the transaction start:

- If the transaction-level read preference is unset, the transaction uses the session-level read preference.
- If transaction-level and the session-level read preference are unset, the transaction uses the client-level read preference. By default, the client-level read preference is primary.
- Multi-document transactions that contain read operations must use read preference primary. All operations in a given transaction must route to the same member.



(Read more)

Transactions and Read Concern

- The **readConcern** option allows you to control the consistency and isolation properties of the data read from replica sets and replica set shards.
- Through the effective use of write concerns and read concerns, you can adjust the level of consistency and availability guarantees as appropriate, such as waiting for stronger consistency guarantees, or loosening consistency requirements to provide higher availability.

Transactions and Read Concern

Read Concern Levels

- **Local**: returns the most recent data available from the node but can be rolled back.
  - *For transactions on sharded cluster, "local" read concern cannot guarantee that the data is from the same snapshot view across the shards. If snapshot isolation is required, use "snapshot" read concern*
- **Majority**: returns data that has been acknowledged by a majority of the replica set members *(i.e. data cannot be rolled back)* if the transaction commits with write concern "majority".
  - *If the transaction does not use **write concern** "majority" for the commit, the "majority" **read concern** provides **no** guarantees that read operations read majority-committed data.*
  - *For transactions on sharded cluster, "majority" read concern cannot guarantee that the data is from the same snapshot view across the shards. If snapshot isolation is required, use "snapshot" read concern.*
- **Snapshot**: Read concern "snapshot" returns data from a snapshot of majority committed data if the transaction commits with write concern "majority".
  - *If the transaction does not use **write concern** "majority" for the commit, the "snapshot" **read concern** provides **no** guarantee that read operations used a snapshot of majority-committed data.*
  - *For transactions on sharded clusters, the "snapshot" view of the data is synchronized across shards.*

Transactions and Read Concern

Read Concern Option:

- For operations not in multi-document transactions, you can specify a readConcern level as an option to commands and methods that support read concern:

  readConcern: { level: <level> }

- To specify the read concern level for mongosh use the method:

  db.collection.find().readConcern(<level>)

Command/Method	"local"	"available"	"majority"	"snapshot" [3]	"linearizable"
count	✓	✓	✓		✓
distinct	✓	✓	✓	✓ [2]	✓
find	✓	✓	✓	✓	✓
db.collection.find() via cursor.readConcern()	✓	✓	✓	✓	✓
geoSearch	✓	✓	✓	✓	✓
getMore	✓				✓
aggregate db.collection.aggregate()	✓	✓	✓	✓	✓ [1]
Session.startTransaction()	✓		✓	✓	

Transactions and Write Concern

- **Write Concern** describes the level of acknowledgment requested from MongoDB for write operations to a standalone mongod or to replica sets or to sharded clusters.
- **Transactions** use the transaction-level write concern to commit the write operations.
  - **Write operations** inside transactions must be issued without explicit write concern specification and use the default write concern.
  - **At commit time**, the writes are then committed using the transaction-level write concern**.**

Transactions and Write Concern

- You can set the transaction-level write concern at the transaction start:
  - If the transaction-level write concern is unset, the transaction-level write concern defaults to the session-level write concern for the commit.
  - If the transaction-level write concern and the session-level write concern are unset, the transaction-level write concern defaults to the client-level write concern

Transactions and Write Concern

- Write Concern Specification: Write concern can include the following fields:

  { w : <value>, j : <boolean>, wtimeout : <number> }

  - The **w** option to request acknowledgment that the write operation has propagated to a specified number of mongod instances or to mongod instances with specified tags.
  - The **j** option to request acknowledgment that the write operation has been written to the on-disk journal, and
  - The **wtimeout** option to specify a time limit to prevent write operations from blocking indefinitely

Using a Transaction

- Here is a recap of the code that's used to complete a multi-document transaction:

```
const session = db.getMongo().startSession()

session.startTransaction()

const account = session.getDatabase('< add database name here>').getCollection('<add collection name here>’)

//Add database operations like .updateOne() here

session.commitTransaction()
```

Aborting a Transaction

- Here is a recap of the code that's used to cancel a transaction before it completes:

```
const session = db.getMongo().startSession()

session.startTransaction()

const account = session.getDatabase('< add database name here>').getCollection('<add collection name here>’)

//Add database operations like .updateOne()

here session.abortTransaction()
```

Transactions in MogoDB

Two phase commit in database management systems is a technique used to ensure atomicity and consistency

- Set up a collection called *transactions*
  {**Target document**, **source document**, **value**, **state**}
- Add a *pendingTransactions=[ ]* field to documents
- Create a new transaction with *state=initial*
- When transaction starts, set *state=pending*

- Store transaction id in *pendingTransactions[ ]*
- Apply transactions to both documents
- Set *state =applied*
- Use **find()** to see if documents are correct
- If so, set *state=done*

Examples

- **Add *pendingTransactions* field to accounts documents**:

```
db.accounts.insert(
   [
          {_id: "A", balance: 1000, pendingTransactions: [ ]},
          {_id: "B", balance: 1000, pendingTransactions: [ ]}
   ]
)
```

- Add document to **transactions** collection:

```
db.transactions.insert(
   {_id: 1, source: “A”, destination: “B”, value: 100, state: “initial”},
)
```

- Get the **transaction** from the collection

```
t = db.transactions.findOne({state: "initial“})
```

Examples

- Update the balances – put transaction id in ***pendingTransactions*** array.

```
db.accounts.update(
        {_id: t.source, pendingTransactions: {$ne:t._id}},
        {$inc:{balance: -t.value}, $push: {pendingTransactions: t._id}}
)
db.accounts.update(
        {_id: t.destination, pendingTransactions: {$ne:t._id}},
        {$inc: {balance: t.value}, $push: {pendingTransactions: t._id}}
)
```

- Update transaction state to **applied**

```
db.transactions.update(
  { _id: t._id, state: "pending" },
  { $set: { state: "applied" }, $currentDate: { lastModified: true } }
)
```

Examples

- Update both accounts' list of pending transactions. we remove the applied transaction_id from the pendingTransactions array for both accounts.

```
db.accounts.update(
  { _id: t.source, pendingTransactions: t._id },
  { $pull: { pendingTransactions: t._id } }
)

db.accounts.update(
  { _id: t.destination, pendingTransactions: t._id },
  { $pull: { pendingTransactions: t._id } }
)
```

- Update transaction state to **done**

```
db.transaction.update(
      {_id: t._id, state: "applied"},
      { $set:{state: "done"}, $currentDate: {lastModified: true}}
)
```