SESSION 9
TRANSACTION MANAGEMENT

Learning Objectives

In this session, students will learn to:

- Explain transactions in MongoDB
- Describe the transaction Application Programming Interfaces (API) in MongoDB
- Explain various error labels in MongoDB
- Describe properties of transactions in MongoDB
- Explain sessions and transactions within the sessions in MongoDB

Consider that a buyer wants to place an order on an e-commerce Website by making a payment using a credit card. The steps in this case will be:

- Calculation of the total amount to be paid for the items in the shopping cart
- Selection of mode of payment and entry of the details of the card by the buyer
- Initiation of payment request to the payment gateway

Only after the payment gateway approves the payment, the order is placed. Otherwise, the buyer is taken back to the shopping cart. For this process to work, all these steps should be performed as a single unit. Otherwise, the order will be placed even if the payment is declined, or the payment will be made,

and the order will not be placed. A set of operations that are performed as a single unit is known as a transaction.

This session will provide an overview of the transactions in MongoDB. It will also explain various transaction error labels in MongoDB. This session will explore the properties of transactions in MongoDB. It will also explain the sessions and transactions within the sessions in MongoDB.

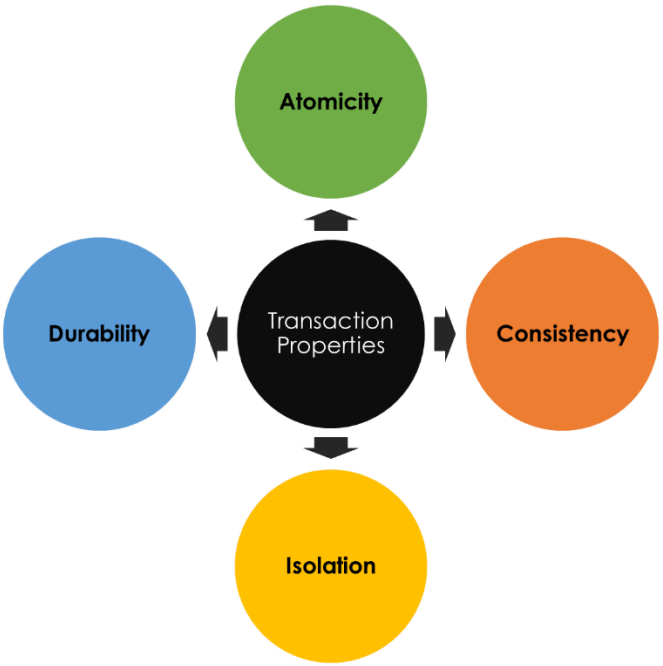
9.1 Introduction to MongoDB Transactions

Transaction is a set of database operations that are run against a database as a single unit of work. In a transaction either all the specified operations in the unit of work execute successfully or it does not execute at all. In case even one database operation in the set fails to execute successfully, all the changes made by other successful database operations are rolled back. The database then comes back to the state in which it was before the transaction was performed.

Transactions in MongoDB can involve operations on a single document or multiple documents.

9.1.1 Transaction Properties

Database transactions guarantee data integrity by complying with the four important properties of transactions:



These properties together are known as ACID properties.

Atomicity ensures that all operations specified in a transaction work as a single unit of work. All the operations in the transaction are executed and changes are made permanent. Otherwise, all the operations are not executed and the database returns to its original state prior to the transaction.

For example, when transferring money from Account A to Account B, if the amount is deducted from Account A and an error occurs when adding that amount to Account B. Atomicity ensures that the amount deducted from Account A is credited back into the account and the transaction fails.

Consistency ensures that the updates made by a transaction are consistent with the existing constraints in the database.

For example, consider an account that has a constraint that on a single day, only a maximum of $5,000 can be withdrawn. If a transaction was initiated to withdraw $10,000 from that account, then the transaction will fail because it is not consistent with the existing constraints.

Isolation ensures that in case of multiple transactions which concurrently update data in the same database, the updates from one transaction does not affect updates from the other.

For example, consider that a transaction has two operations—one operation adds $300 to Account A and the other withdraws $200 from Account A. Isolation prevents other transactions from accessing the details of Account A between these two operations.

Durability ensures that when a transaction is committed successfully, all the updates made by the transaction are saved permanently. Power outages or hardware failures will not impact the validity of the transaction.

For example, if the server of a bank crashes, the details of all the committed transactions will remain unaffected and can be recovered completely.

A transaction ends with a commit or a rollback. When a transaction is committed, all the changes made by the operations in the transaction are saved permanently and are visible outside the transaction. When a transaction is rolled back, all the changes made by the operations in the transaction are discarded. The database is then restored to the state in which it was before the beginning of the transaction.

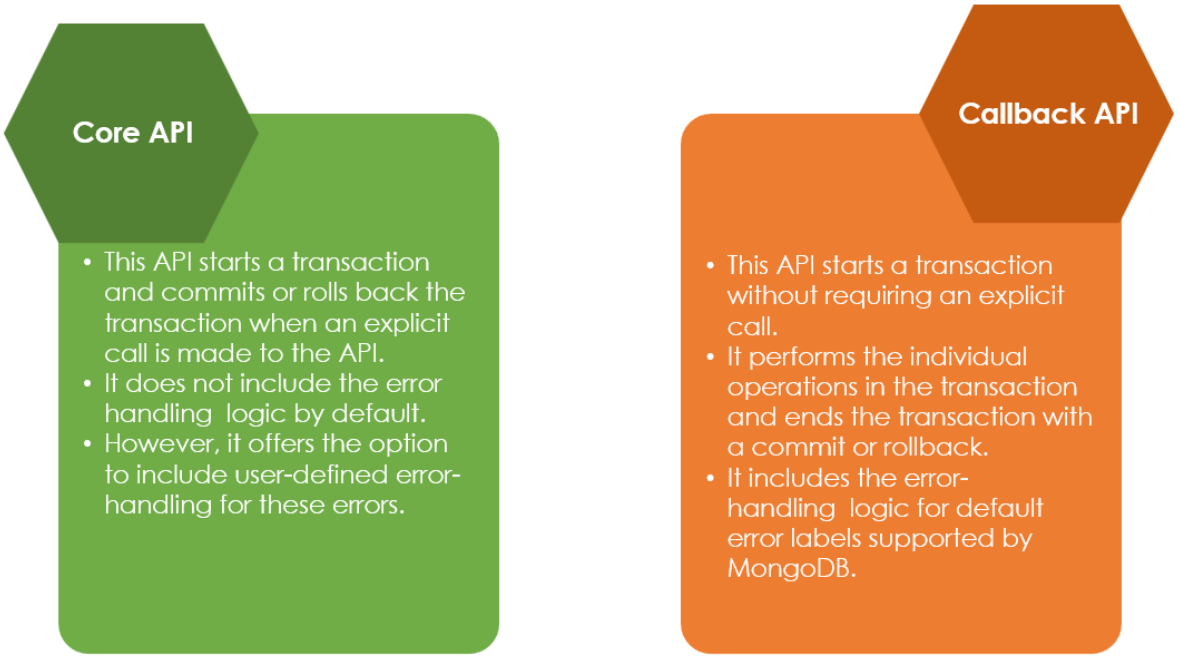
9.1.2 Replica Sets

MongoDB uses the concept of replica sets to support multi-document transactions. A replica set is a collection of `mongod` instances that provide data redundancy and availability by maintaining identical datasets.

The multiple copies of data in a replica set ensures that a multi-document transaction is committed or rolled back completely.

9.1.3 Transaction Application Programming Interface (API)

The two transaction APIs provided by MongoDB are:

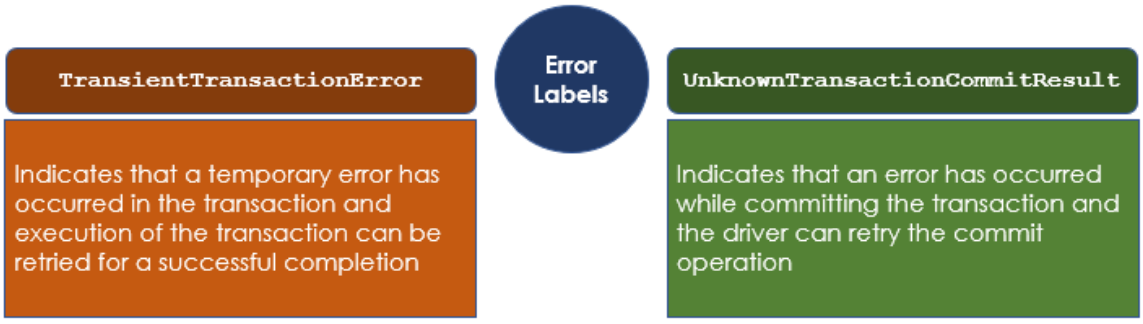


9.1.4 Transaction Error Handling

Consider a scenario where a user is using a credit card to make a payment for a purchase on a shopping Website. In this case, the payment will be successful after the user provides the correct card number, expiry date, Card Verification Value (CVV) number, and credentials. If any of these details are incorrect or if the specified card has expired the transaction is aborted with an error message stating that the payment could not be completed. This happens because these errors are captured in the transaction code and appropriate actions to be taken for each of these errors are also specified with the transaction code. For example, in this case, the transaction code would have specified if any of this incorrect information is provided, terminate the transaction, and display an error message.

Capturing the errors and specifying the actions to be taken in case of an error is referred to as error handling. Error handling is crucial in a transaction to ensure the consistency and integrity of data in the database. For example, in the scenario discussed earlier, consider that the card number is incorrect, but the card number exists for some other user. Then, the payment will be charged to the person who is holding that card instead of the person who made the payment.

The two major error labels in MongoDB are:



A temporary error can be temporary network error or conflict between two sessions that try to modify the same document. When the `TransientTransactionError` error is encountered, the entire transaction must be retried; individual write operations cannot be retried. When the `UnknownTransactionCommitResult` error is encountered, the commit operation can be retried; individual write operations in the transaction cannot be retried.

9.2 Transactions and Sessions

MongoDB supports the feature of sessions. A session is a logical grouping of related read and write operations that must be executed sequentially. Transactions exist and get executed within a session. A session can contain multiple transactions. These transactions within a session get executed sequentially. At a given time, only one transaction remains active in a session. If a session ends before a transaction completes, the transaction is aborted, and all changes are rolled back.

Session provides methods for the transactions to access the database. Let us learn about some of the important methods provided by MongoDB to create, commit, and abort transactions in a session.

9.2.1 Start Transaction

MongoDB provides the `Session.startTransaction` method to start a transaction within a session. The syntax for this method is:

```
Session.startTransaction(options)
```

This method takes an argument named `options` which is of type `document`. Table 9.1 describes the parameters of `options`.

Parameter	Description
readConcern	• Specifies how the data should be read during a read operation • Takes three values: o `snapshot`: Reads the data that has been duplicated to majority of the members in the replica set at a specific point in time recently. o `local`: Reads the most recent data that is available when the query is executed. It does not check if the data has been persisted and duplicated to the members of the replica set. There is a possibility that after the data is read, the changes are rolled back. This is the default setting that is applicable when no `readConcern` option is specified. o `majority`: Reads the most recent data that is persisted and duplicated to the majority of the members in the replica set. This data will not be rolled back. • Overrides the read concern levels specified in the operations
writeConcern	• Specifies how MongoDB must acknowledge the success of write operation • Takes two values: o `majority`: Checks for acknowledgement from a majority of the members in the replica set o `<number>`: Checks for acknowledgement from the specified number of members in the replica set

Table 9.1: Parameters of `option`

9.2.2 Commit Transaction

MongoDB provides the `Session.commitTransaction` method to permanently save the changes made by the operations in a transaction and to make the changes visible outside the transaction. The value for `writeConcern` specified in the method to start the transaction is used here.

The `Session.commitTransaction` transaction ensures atomicity of the transaction.

9.2.3 Abort Transaction

MongoDB provides the `Session.abortTransaction` method to rollback the changes made by the operations in a transaction. This method ensures that the changes made by the operations in the transaction are discarded and are not visible outside the transaction.

9.3 Working with Transactions

MongoDB supports transactions only on replica set or sharded cluster. It does not support transactions on standalone deployments.

Therefore, to work with transactions consider that the user has created a three-member replica set (Primary-Secondary-Secondary) named `replicaset`.

Now, consider that in `replicaset`, the user has the `Employee` database, and this database has the `EmpDetail` collection. The documents in this collection are shown in Figure 9.1.

```
replicaset [direct: primary] Employee> db.EmpDetail.find()
[
  {
    _id: ObjectId("647778437e0714d84f821a47"),
    Emp_ID: 101,
    Emp_Name: 'Oliver Smith',
    Gender: 'Male',
    Designation: 'Software Engineer'
  },
  {
    _id: ObjectId("647778437e0714d84f821a48"),
    Emp_ID: 103,
    Emp_Name: 'Richard Franklin',
    Gender: 'Male',
    Designation: 'HR Manager'
  },
  {
    _id: ObjectId("647778437e0714d84f821a49"),
    Emp_ID: 105,
    Emp_Name: 'Linda Michael',
    Gender: 'Female',
    Designation: 'Accountant'
  }
]
```

Figure 9.1: Documents in `EmpDetail` Collection

To avoid duplicates in the `Emp_ID` field, the user has created a unique index using the command:

```
db.EmpDetail.createIndex( { "Emp_ID": 1 },
          { "unique": true } )
```

To allow read operation on the secondary nodes, the user runs the command as:

```
rs.secondaryOk()
```

The command executes as shown in Figure 9.2.

```
replicaset [direct: secondary] test> rs.secondaryOk()
Leaving read preference unchanged (is already "primaryPreferred")
```

Figure 9.2: Allow Read Operation on Secondary

To verify that the documents reflect in the secondary node connected at port `27019`, on the secondary node, run the commands as:

```
use Employee
```

```
db.EmpDetail.find()
```

The commands execute as shown in Figure 9.3.

```
replicaset [direct: secondary] Employee> use test
switched to db test
replicaset [direct: secondary] test> rs.secondaryOk()
Leaving read preference unchanged (is already "primaryPreferred")

replicaset [direct: secondary] test> use Employee
switched to db Employee
replicaset [direct: secondary] Employee> db.EmpDetail.find()
[
  {
    _id: ObjectId("647778437e0714d84f821a48"),
    Emp_ID: 103,
    Emp_Name: 'Richard Franklin',
    Gender: 'Male',
    Designation: 'HR Manager'
  },
  {
    _id: ObjectId("647778437e0714d84f821a47"),
    Emp_ID: 101,
    Emp_Name: 'Oliver Smith',
    Gender: 'Male',
    Designation: 'Software Engineer'
  },
  {
    _id: ObjectId("647778437e0714d84f821a49"),
    Emp_ID: 105,
    Emp_Name: 'Linda Michael',
    Gender: 'Female',
    Designation: 'Accountant'
  }
]
```

Figure 9.3: Documents in `EmpDetail` Collection on Secondary Node

Similarly, the user verifies that these documents are visible on the secondary node connected at port `27020`.

9.3.1 Create a Transaction Within a Session and Commit the Transaction

Consider that the user wants to start a session in MongoDB. In this session, the user wants to execute a transaction to perform a single insert operation. Finally, the user wants to commit the transaction. To perform these operations:

1. To start a session named `session`, on the primary node, run the command as:

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

The session is created.

2. To start a transaction, run the command as:

```
session.startTransaction({"readConcern": { "level":
"snapshot" },"writeConcern": { "w": "majority" }})
```

The transaction starts.

3. To create a variable that represents the `EmpDetail` collection within the transaction, run the command as:

```
var empdetail =
session.getDatabase('Employee').getCollection('EmpDetail')
```

4. To insert a new document into `EmpDetail`, run the command as:

```
empdetail.insertOne({Emp_ID:102, Emp_Name:"John Edward",
    Gender:"Male", Designation:"Software Engineer"})
```

The command executes as shown in Figure 9.4.

```
replicaset [direct: primary] Employee> empdetail.insertOne({Emp_ID:102,
Emp_Name:"John Edward", Gender:"Male", Designation:"Software Engineer"})

{
  acknowledged: true,
  insertedId: ObjectId("64782d25e5414053d2f443a0")
}
```

Figure 9.4: Document Inserted in `EmpDetail` Collection

5. To view the newly inserted document in the primary node, on the primary node, run the command as:

```
empdetail.find()
```

The command executes as shown in Figure 9.5.

```
replicaset [direct: primary] Employee> db.EmpDetail.find()
[
  {
    _id: ObjectId("647778437e0714d84f821a48"),
    Emp_ID: 103,
    Emp_Name: 'Richard Franklin',
    Gender: 'Male',
    Designation: 'HR Manager'
  },
  {
    _id: ObjectId("647778437e0714d84f821a47"),
    Emp_ID: 101,
    Emp_Name: 'Oliver Smith',
    Gender: 'Male',
    Designation: 'Software Engineer'
  },
  {
    _id: ObjectId("647778437e0714d84f821a49"),
    Emp_ID: 105,
    Emp_Name: 'Linda Michael',
    Gender: 'Female',
    Designation: 'Accountant'
  },
  {
    _id: ObjectId("647834aee5414053d2f443a1"),
    Emp_ID: 102,
    Emp_Name: 'John Edward',
    Gender: 'Male',
    Designation: 'Software Engineer'
  }
]
```

Figure 9.5: Documents on the Primary Node

The newly inserted document with value in the `Emp_ID` field `102` is displayed.

6. To check if the newly inserted document is visible on the secondary node, on the secondary node, run the command as:

```
db.EmpDetail.find()
```

The command executes as shown in Figure 9.6.

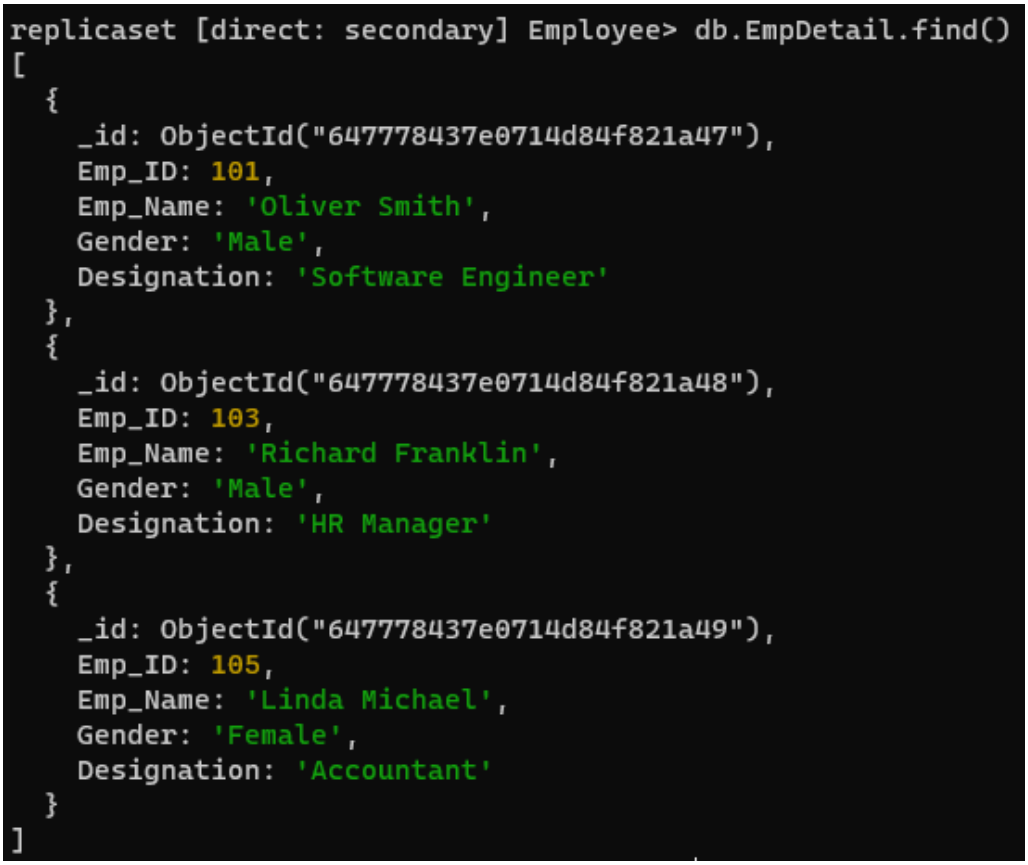


Figure 9.6: Documents on the Secondary Node

The newly inserted document with value in the `Emp_ID` field `102` is not displayed. The collection in the secondary node is not yet updated with the newly inserted document because the transaction has not been committed.

7. To commit the transaction, on the primary node, run the command as:

```
session.commitTransaction()
```

The command executes as shown in Figure 9.7.

```
replicaset [direct: primary] Employee> session.commitTransaction()
{
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1685600175, i: 1 }),
    signature: {
      hash: Binary(Buffer.from("0000000000000000000000000000000000000000", "hex"), 0),
      keyId: 0
    }
  },
  operationTime: Timestamp({ t: 1685600175, i: 1 })
}
```

Figure 9.7: Committing the Transaction

8. Now to check if the newly inserted document is visible in the secondary node, repeat Step 6.

The command executes as shown in Figure 9.8.

```
replicaset [direct: secondary] Employee> db.EmpDetail.find()
[
  {
    _id: ObjectId("647778437e0714d84f821a47"),
    Emp_ID: 101,
    Emp_Name: 'Oliver Smith',
    Gender: 'Male',
    Designation: 'Software Engineer'
  },
  {
    _id: ObjectId("647778437e0714d84f821a48"),
    Emp_ID: 103,
    Emp_Name: 'Richard Franklin',
    Gender: 'Male',
    Designation: 'HR Manager'
  },
  {
    _id: ObjectId("647778437e0714d84f821a49"),
    Emp_ID: 105,
    Emp_Name: 'Linda Michael',
    Gender: 'Female',
    Designation: 'Accountant'
  },
  {
    _id: ObjectId("647834aee5414053d2f443a1"),
    Emp_ID: 102,
    Emp_Name: 'John Edward',
    Gender: 'Male',
    Designation: 'Software Engineer'
  }
]
```

Figure 9.8: Document on the Secondary Node After Commit

After the commit operation, the newly inserted document is visible in the secondary node.

9.3.2 Create a Transaction Within a Session and Abort the Transaction

Consider that the user wants to execute a few operations—update and delete—in a session. To do so, the user must perform these steps:

1. To start a session, a transaction, and create a variable that represents the `EmpDetail` collection, perform steps as learned earlier.
2. To update the `Designation` field of the employee with `Emp_ID` as `105` to `Project Lead`, on the primary node, run the command as:

```
empdetail.updateOne({Emp_ID:105},{$set:{Designation:
                    "Project Lead"}})
```

The command executes as shown in Figure 9.9.

```
replicaset [direct: secondary] test> var session = db.getMongo().startSession()

replicaset [direct: primary] test> session.startTransaction({"readConcern": { "level": "snapshot" },"writeConcern":
 { "w": "majority" }})

replicaset [direct: primary] test>  var empdetail = session.getDatabase('Employee').getCollection('EmpDetail')

replicaset [direct: primary] test> empdetail.updateOne({Emp_ID:105},{$set:{Designation:"Project Lead"}})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

Figure 9.9: Update a Document

3. To delete the document where `Emp_ID` is 103, on the primary node, run the command as:

```
empdetail.deleteOne({Emp_ID:103})
```

The command executes as shown in Figure 9.10.

```
replicaset [direct: primary] test> empdetail.deleteOne({Emp_ID:103})
{ acknowledged: true, deletedCount: 1 }
```

Figure 9.10: Delete a Document

4. To check if the update and delete operations reflect on the primary node and the secondary node, on both the nodes, run the command as:

```
db.EmpDetail.find()
```

The command executes as shown in Figures 9.11 and 9.12.

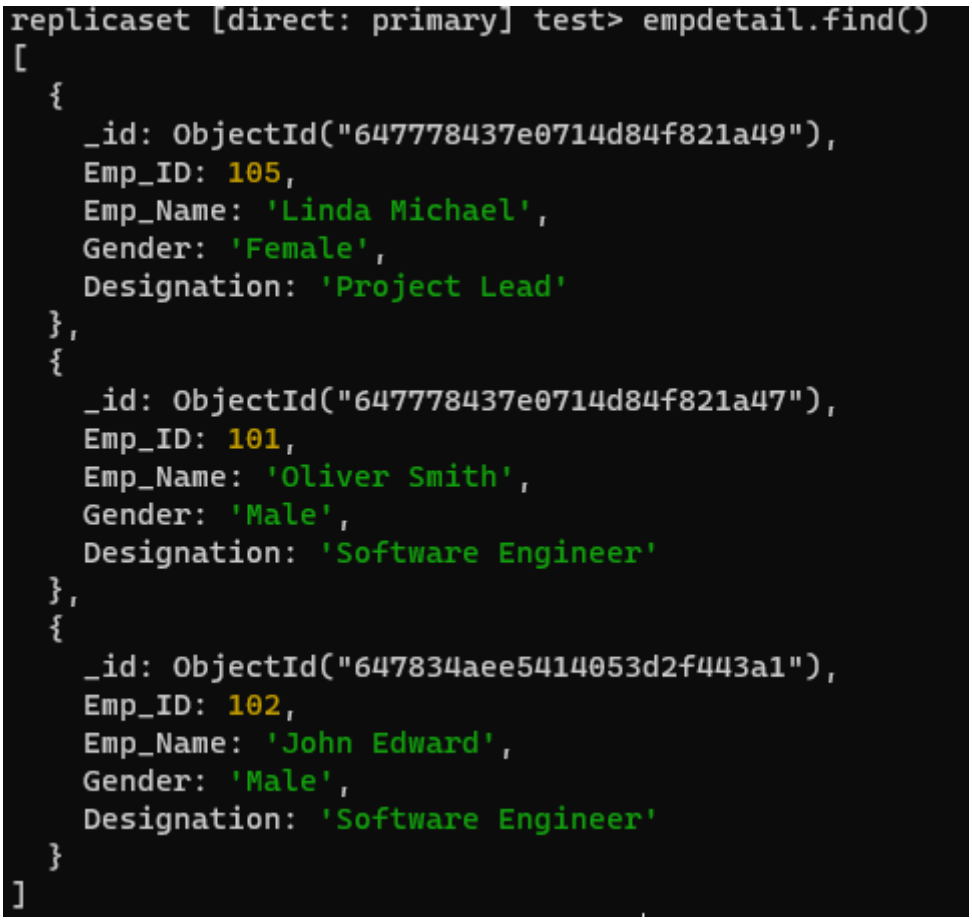


Figure 9.11: Documents on the Primary Node

```
replicaset [direct: secondary] Employee> db.EmpDetail.find()
[
  {
    _id: ObjectId("647778437e0714d84f821a47"),
    Emp_ID: 101,
    Emp_Name: 'Oliver Smith',
    Gender: 'Male',
    Designation: 'Software Engineer'
  },
  {
    _id: ObjectId("647778437e0714d84f821a48"),
    Emp_ID: 103,
    Emp_Name: 'Richard Franklin',
    Gender: 'Male',
    Designation: 'HR Manager'
  },
  {
    _id: ObjectId("647778437e0714d84f821a49"),
    Emp_ID: 105,
    Emp_Name: 'Linda Michael',
    Gender: 'Female',
    Designation: 'Accountant'
  },
  {
    _id: ObjectId("647834aee5414053d2f443a1"),
    Emp_ID: 102,
    Emp_Name: 'John Edward',
    Gender: 'Male',
    Designation: 'Software Engineer'
  }
]
```

Figure 9.12: Documents on the Secondary Node

Results of update and delete operations are seen on the primary node. However, the results of these operations are not reflected on the secondary node because the transaction is not yet committed.

Consider that the user has realized that document with `Emp_ID` as 102 must be deleted instead of document with `Emp_ID` as 103. Therefore, an abort action must be performed to cancel the delete operation. Since the transaction consists of two operations—update and delete, aborting the transaction will roll back both these operations.

To abort the transaction:

1. On the primary node, run the command as:

```
session.abortTransaction()
```

2. To verify if the changes are rolled back on the primary node, on the primary node, run the command as:

```
empdetail.find()
```

The command executes as shown in Figure 9.13.

Figure 9.13: Rolled Back Changes in the Collection

The changes made by the update and delete operations are rolled back.

By default, MongoDB aborts transactions that run for more than 60 seconds.

9.3.3 Create a Multi-Document Transaction Within a Session

Consider that the user has created a collection `DesignationDetail` with documents as shown in Figure 9.14.

```
[
  {
    _id: ObjectId("648ff438c48f7416b9f41ba0"),
    Designation_ID: 10003,
    Designation_Title: 'Accountant',
    Department: 'Finance'
  },
  {
    _id: ObjectId("648ff438c48f7416b9f41b9e"),
    Designation_ID: 10002,
    Designation_Title: 'HR Manager',
    Department: 'Human Resources'
  },
  {
    _id: ObjectId("648ff438c48f7416b9f41b9c"),
    Designation_ID: 10001,
    Designation_Title: 'Software Engineer',
    Department: 'Product Development'
  },
  {
    _id: ObjectId("648ff438c48f7416b9f41b9f"),
    Designation_ID: 10012,
    Designation_Title: 'HR Executive',
    Department: 'Human Resources'
  },
  {
    _id: ObjectId("648ff438c48f7416b9f41b9d"),
    Designation_ID: 10011,
    Designation_Title: 'Project Manager',
    Department: 'Product Development'
  },
  {
    _id: ObjectId("648ff438c48f7416b9f41ba1"),
    Designation_ID: 10013,
    Designation_Title: 'Finance Manager',
    Department: 'Finance'
  }
]
```

Figure 9.14: Documents in the `DesignationDetail` Collection

Now, the user wants to change the designation of `Accountant` to `Finance Executive`. This change must be made in both the `EmpDetail` as well as the `DesignationDetail` collections. To do this, the user can employ a multi-document transaction.

To do so, the user must perform the following steps:

1. To start a session, a transaction, and create a variable that represents the `EmpDetail` collection, perform steps as learned earlier.
2. To create a variable that represents the `DesignationDetail` collection within the transaction, run the command as:

```
var designation =
session.getDatabase('Employee').getCollection('Designation
Detail')
```

3. To update the `Designation` field of the employee with `Emp_ID` as `105` to `Finance Executive`, on the primary node, run the command as:

```
empdetail.updateOne({Emp_ID:105},{$set:{Designation:
"Finance Executive"}})
```

4. To update the `Designation_Title` field of the designation with `Designation_ID` as `10003` to `Finance Executive`, on the primary node, run the command as:

```
designation.updateOne({Designation_ID:10003},{$set:{
Designation_Title:"Finance Executive"}})
```

5. To commit the transaction, on the primary node, run the command as:

```
session.commitTransaction()
```

The command executes as shown in Figure 9.15.

```
replicaset [direct: primary] Employee> var session = db.getMongo().startSession()

replicaset [direct: primary] Employee> session.startTransaction({"readConcern": { "level": "snapshot" },"writeConcern": {
 "w": "majority" }})

replicaset [direct: primary] Employee> var empdetail = session.getDatabase('Employee').getCollection('EmpDetail')

replicaset [direct: primary] Employee> var designation = session.getDatabase('Employee').getCollection('DesignationDetail
')

replicaset [direct: primary] Employee> empdetail.updateOne({Emp_ID:105},{$set:{Designation:"Finance Executive"}})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
replicaset [direct: primary] Employee> designation.updateOne({Designation_ID:10003},{$set:{Designation_Title:"Finance Exe
cutive"}})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
replicaset [direct: primary] Employee> session.commitTransaction()
{
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1687162058, i: 2 }),
    signature: {
      hash: Binary(Buffer.from("0000000000000000000000000000000000000000", "hex"), 0),
      keyId: 0
    }
  },
  operationTime: Timestamp({ t: 1687162058, i: 1 })
}
```

Figure 9.15: Update Documents in Two Collections

6. To view the documents in the `EmpDetail` collection on the primary, run the command as:

```
db.EmpDetail.find()
```

Figure 9.16 shows the output of this query.

```
replicaset [direct: primary] Employee> db.EmpDetail.find()
[
  {
    _id: ObjectId("6490093070d9fd0f3fb43b57"),
    Emp_ID: 101,
    Emp_Name: 'Oliver Smith',
    Gender: 'Male',
    Designation: 'Software Engineer'
  },
  {
    _id: ObjectId("6490093070d9fd0f3fb43b58"),
    Emp_ID: 103,
    Emp_Name: 'Richard Franklin',
    Gender: 'Male',
    Designation: 'HR Manager'
  },
  {
    _id: ObjectId("6490093070d9fd0f3fb43b59"),
    Emp_ID: 105,
    Emp_Name: 'Linda Michael',
    Gender: 'Female',
    Designation: 'Finance Executive'
  },
  {
    _id: ObjectId("6490093070d9fd0f3fb43b5a"),
    Emp_ID: 102,
    Emp_Name: 'John Edward',
    Gender: 'Male',
    Designation: 'Software Engineer'
  }
]
```

Figure 9.16: Updated Document in the `EmpDetail` Collection

7. Similarly, to view the documents in the `DesignationDetail` collection on the primary, run the command as:

```
db.DesignationDetail.find()
```

Figure 9.17 shows the output of this query.

```
replicaset [direct: primary] Employee> db.DesignationDetail.find()
[
  {
    _id: ObjectId("64900bbf70d9fd0f3fb43b61"),
    Designation_ID: 10001,
    Designation_Title: 'Software Engineer',
    Department: 'Product Development'
  },
  {
    _id: ObjectId("64900bbf70d9fd0f3fb43b62"),
    Designation_ID: 10011,
    Designation_Title: 'Project Manager',
    Department: 'Product Development'
  },
  {
    _id: ObjectId("64900bbf70d9fd0f3fb43b63"),
    Designation_ID: 10002,
    Designation_Title: 'HR Manager',
    Department: 'Human Resources'
  },
  {
    _id: ObjectId("64900bbf70d9fd0f3fb43b64"),
    Designation_ID: 10012,
    Designation_Title: 'HR Executive',
    Department: 'Human Resources'
  },
  {
    _id: ObjectId("64900bbf70d9fd0f3fb43b65"),
    Designation_ID: 10003,
    Designation_Title: 'Finance Executive',
    Department: 'Finance'
  },
  {
    _id: ObjectId("64900bbf70d9fd0f3fb43b66"),
    Designation_ID: 10013,
    Designation_Title: 'Finance Manager',
    Department: 'Finance'
  }
]
```

Figure 9.17: Updated Document in the `DesignationDetail` Collection

8. To verify that the changes have been reflected on the secondary also, on the secondary node, run the command as:

```
db.EmpDetail.find()
```

Figure 9.18 shows the output of this query.

```
replicaset [direct: secondary] Employee> db.EmpDetail.find()
[
  {
    _id: ObjectId("64900bd570d9fd0f3fb43b69"),
    Emp_ID: 105,
    Emp_Name: 'Linda Michael',
    Gender: 'Female',
    Designation: 'Finance Executive'
  },
  {
    _id: ObjectId("64900bd570d9fd0f3fb43b68"),
    Emp_ID: 103,
    Emp_Name: 'Richard Franklin',
    Gender: 'Male',
    Designation: 'HR Manager'
  },
  {
    _id: ObjectId("64900bd570d9fd0f3fb43b6a"),
    Emp_ID: 102,
    Emp_Name: 'John Edward',
    Gender: 'Male',
    Designation: 'Software Engineer'
  },
  {
    _id: ObjectId("64900bd570d9fd0f3fb43b67"),
    Emp_ID: 101,
    Emp_Name: 'Oliver Smith',
    Gender: 'Male',
    Designation: 'Software Engineer'
  }
]
```

Figure 9.18: Updated Document in the `EmpDetail` Collection

9. Similarly, to view the documents in the `DesignationDetail` collection on the secondary, run the command as:

```
db.DesignationDetail.find()
```

Figure 9.19 shows the output of this query.

```
replicaset [direct: secondary] Employee> db.DesignationDetail.find()
[
  {
    _id: ObjectId("64900bbf70d9fd0f3fb43b66"),
    Designation_ID: 10013,
    Designation_Title: 'Finance Manager',
    Department: 'Finance'
  },
  {
    _id: ObjectId("64900bbf70d9fd0f3fb43b64"),
    Designation_ID: 10012,
    Designation_Title: 'HR Executive',
    Department: 'Human Resources'
  },
  {
    _id: ObjectId("64900bbf70d9fd0f3fb43b61"),
    Designation_ID: 10001,
    Designation_Title: 'Software Engineer',
    Department: 'Product Development'
  },
  {
    _id: ObjectId("64900bbf70d9fd0f3fb43b65"),
    Designation_ID: 10003,
    Designation_Title: 'Finance Executive',
    Department: 'Finance'
  },
  {
    _id: ObjectId("64900bbf70d9fd0f3fb43b62"),
    Designation_ID: 10011,
    Designation_Title: 'Project Manager',
    Department: 'Product Development'
  },
  {
    _id: ObjectId("64900bbf70d9fd0f3fb43b63"),
    Designation_ID: 10002,
    Designation_Title: 'HR Manager',
    Department: 'Human Resources'
  }
]
```

Figure 9.19: Updated Document in the `DesignationDetail` Collection

9.4 Summary

- Transactions in MongoDB are a single logical unit of work that involves single or multiple operations.
- The integrity of data in the database is protected by the four properties of transactions—Atomicity, Consistency, Isolation, and Durability (ACID).
- MongoDB provides two transaction APIs—Callback API and Core API.
- MongoDB provides methods to create, commit, and abort transactions in a session.
- Two major error labels in MongoDB are `TransientTransactionError` and `UnknownTransactionCommitResult`.
- MongoDB sessions can contain multiple transactions. Only one of these transactions remains active at a time.

Test Your Knowledge

1. Which of the following statements are true about Callback API in MongoDB?

    a. It starts a transaction without an explicit call.
    b. It does not incorporate error handling logic for `TransientTransactionError` and `UnknownTransactionCommitResult`.
    c. It starts a transaction only upon an explicit call.
    d. Automatically incorporates error handling logic for `TransientTransactionError` and `UnknownTransactionCommitResult`.

2. Which of the following options is not true about aborting transactions in MongoDB?

    a. By default, a transaction is aborted if it runs for more than 30 seconds.
    b. The `Session.abortTransaction` method is used to roll back the changes made by the operations in a transaction.
    c. If an error occurs during aborting a transaction, MongoDB drivers retry the abort method even when the `retryWrites` parameter in the connection string is set to `false`.
    d. When a transaction is rolled back, all the changes made by the operations in the transaction are discarded.

3. Which of the following is the correct syntax to start a session and execute a transaction?

    a. `var session = db.startSession()`
    b. `var session = session.startSession()`
    c. `var session = db.getMongo().startSession()`
    d. `var session = db.getMongo().startTransaction()`

4. Which of the following is the correct syntax to create a variable `Inventdetail` that represents the `Invdetail` collection in the `Inventory` database within the session?

   a.
```
var Inventdetail =
db.getDatabase('Inventory').getCollection('Invdetail
')
```
   b.
```
var Inventdetail =
Startsession.getDatabase('Inventory').getCollection(
'Invdetail')
```
   c.
```
var Inventdetail = db.Inventory.Invdetail
```
   d.
```
var Inventdetail =
session.getDatabase('Inventory').getCollection('Invd
etail')
```

5. Which of the following method is used to save the changes made by the operations in a transaction and make the changes visible outside the transaction?

   a. `db.commitTransaction`
   b. `Session.commitTransaction`
   c. `Session.transactionCommit`
   d. `db.transactionCommit`

Answers to Test Your Knowledge

1	a, d
2	a
3	c
4	d
5	b

Try it Yourself

1. Implement a three-member replica set with the P-S-S architecture.
2. Create a database named `Inventory` which includes a collection named `sales_invent`.
3. In the primary server, insert the given three documents into the `sales_invent` collection.

```
[
   {
     cust_id:1002,
     customername: "Richard",
     gender:"M",
     purchased_product:"cereals",
     quantity:6,
     price:60
     },
   {
     cust_id:1005,
     customername: "Williams",
     gender:"M",
     purchased_product:"Vegetables",
     quantity:10,
     price:150
  },
   {
     cust_id:1007,
     customername: "John",
     gender:"M",
     purchased_product:"Baby Food",
     quantity:3,
     price:300
  }
 ]
```

4. In the primary server, create an index for the field `cust_id` and view the documents in the `sales_invent` collection.
5. Start a session and start a transaction to perform the two given operations:
   a. Update the `purchased_product` field of the customer with `cust_id` as `1007` to "`Dairy product`".
   b. Delete the document where `cust_id` is `1002`.

6. In the primary server, commit the transaction given in question 5 and check whether the two operations that are executed in the transaction are visible in the secondary servers.

7. In the primary server, start another new session and start a transaction to insert one document into the `sales_invent` collection as:

```
{
    cust_id:1009,
    customername: "Smith",
    gender:"M",
    purchased_product:"Fruits",
    quantity:5,
    price:180
}
```

8. In the primary server, abort the transaction given in question 7 and check whether the insert operation that is performed in the transaction is not reflected in the secondary servers.