TCL - Transaction Control Language

TCL

Transaction Control Language statements are utilized to manage transactions within a relational database.

- COMMIT, any DML operations that were executed before the statements will be persisted permanently.
- ROLLBACK, any DML operations between two COMMIT statements will be completely erased (something like Ctrl + Z that will stop only when it reaches last time you opened the specific file). Committed transactions cannot be rolled back.
- SAVEPOINT, utilized to ROLLBACK to a specific point in time.

The general flow of using TCL could be as follows:

```
[Many DML Operations]

SAVEPOINT A

[Many DML Operations]

ROLLBACK TO A
```

ACID Properties

Any amount of DML statements before a COMMIT statement is considered a **transaction**. After the COMMIT is done, the transaction should follow certain properties. Transactions are considered to be logical units of work - for example, transferring money from Person A's bank account to Person B's account. You should think about how to logically group your DML operations into transactions when writing code.

A transaction has 4 properties termed as ACID Properties:

- **A**tomicity - Atomicity means that either all of the transactions will execute successfully or none of them will.
- **C**onsistency - Consistency means that constraints are enforced for every committed transaction. That indicates that all Keys, Data types, Checks, and Triggers are successful and no constraint violation is triggered.
- **I**solation - If two transactions are executing concurrently and working on the same data, then one transaction should not disturb the other transaction. Isolation guarantees that concurrently running transactions should not affect each other.
- **D**urability - Durability means that once a transaction is complete, it guarantees that all of the changes are recorded in the database. If our system is suddenly affected by a system crash or a power outage, then all unfinished committed transactions may be replayed.

In other words,

The properties of transactions should be as follows:

- **ATOMIC** - "All or nothing", if any statement on the transaction fails, the whole transaction fails.
- **CONSISTENT** - If the database was in a consistent state before the transaction, it should be after it.
- **ISOLATED** - One transaction shouldn't affect other transactions. It can be applied in different levels.
- **DURABLE** - Persisted data should be saved permanently, even in the case of power loss or catastrophic software or hardware failure.
    - Some RDBMS's have different approaches, even to recover from catastrophes (Oracle has special logs [Bitacora]).

Isolation Levels

Isolation levels are applied in RDBMS to provide consistency and avoid certain read phenomena. The following isolation levels are presented from most to least isolated:

- Serializable
    - Allowed in Oracle
    - Read/Write locks
    - Applies range locks even in the WHERE clauses of a select statement
    - Phantom reads can't happen because of this
    - Table that is being read can't be modified until the reading is done (no INSERTS, no UPDATES, no DELETES)
- Repeatable Reads
    - Not used often
    - Read/Write locks
    - Doesn't provide range locks, that means phantom reads can happen
    - Doesn't lock the whole SELECT statement, nor INSERTS, nor UPDATES, nor DELETS
- Read Committed
    - Oracle default
    - Write only locks
    - Only data that is committed will be seen by other transactions
    - Dirty reads can't happen, but Phantom reads can
    - This is why it is recommended to not perform very long transactions
- Read Uncommitted
    - A disaster
    - Dirty reads are normal, any transaction can see any uncommitted data
    - Very inconsistent

Read Phenomena

- Dirty Read: reading data that is uncommitted

- Non-repeatable read: when a row is read twice in a transaction and the values are different
- Phantom Read: reading data that is being added or modified by a running transaction

Setting up the Database Driver

Database JDBC Drivers

Because JDBC is a Java language API, it is database agnostic. It uses database drivers which implement the interfaces defined in the JDBC API for the given database. For example, to connect with an Oracle database, you would use an OJDBC driver. Other database vendors have different drivers which implement the JDBC API.

Many JDBC drivers are available through Maven's central repository and can be added as a dependency in the `pom.xml` file. Oracle is a special exception due to license restrictions. You must accept the license agreement, download, and install it to your local Maven repository (tutorial here before you can add it to the `pom.xml` file.

Finally, in your application code, you can register the driver using:

```
try {
   Class.forName("oracle.jdbc.driver.OracleDriver");
}
catch(ClassNotFoundException e) {
   System.out.println("Can't load driver class!");
}
```

This step is only necessary for drivers prior to JDBC 4.0 (released with Java SE 6). After JDBC 4.0, drivers will be auto loaded if they are included in the class path.

Connection Interface

`java.sql.Connection` interface represents a session between java application and database. All SQL statements are executed and results are returned within the context of a Connection object.

We can use the `DriverManager` class to get a `Connection` to the database, given that we have the JDBC URL, username, and password. Generally these parameters should be stored in an external configuration file that can be loaded dynamically and changed without affecting the application code.

```
try (Connection conn = DriverManager.getConnection(DB_URL,USERNAME,PASSWORD)) {
```

```
  // more code goes here
} catch (SQLException e) {}
```

Alternatively, the `DataSource` interface could be used to make connections and is covered extensively in this Oracle tutorial.

It's always a good idea to close your resources - here we've used the `try-with-resources` syntax to automatically close the `Connection` being created after the block ends.

Autocommit mode

By default, when a connection is created it is in auto-commit mode, so every SQL statement acts as a transaction and is committed immediately after execution. In order to manually group statements into a transaction, simply call:

```
Connection conn = DriverManager.getConnection(DB_URL,USERNAME,PASSWORD);
conn.setAutoCommit(false);
// execute some SQL statements...
con.commit();
```

JDBC String

The database URL is an address pointing to the database to be used, also known as the JDBC String. The format of this URL varies between database vendors, as shown in the table below:

RDBMS	JDBC driver	URL format
MySQL	`com.mysql.jdbc.Driver`	jdbc:mysql://hostname/databaseName
Oracle	`oracle.jdbc.driver.OracleDriver`	jdbc:oracle:thin:@hostname:portNumber:databas
SQLServer	`com.microsoft.sqlserver.jdbc.SQLServerDriver`	jdbc:sqlserver://serverName:portNumber;property
PostgreSQL	`org.postgresql.Driver`	jdbc:postgresql://hostname:port/databaseName

Interfaces of JDBC

JDBC Classes and Interfaces

JDBC stands for Java Database Connectivity. It is a relatively low-level API used to write Java code that interacts with relational databases via SQL.

The JDBC classes and interfaces are located in the `java.sql` and `javax.sql` packages. There are several key classes and interfaces that are commonly encountered when writing JDBC code:

- `DriverManager` class - to make a connection with a database driver

- `DataSource` interface - for retrieving connections, an alternative to `DriverManager`
- `Connection` interface - represents a physical connection with a database
- `SQLException` class - a general exception thrown when something goes wrong when accessing the database
- `Statement` interface - used for executing static SQL statements
- `PreparedStatement` interface - represents pre-compiled SQL statements
- `CallableStatement` interface - used to execute stored procedures
- `ResultSet` interface - represents data returned from the database

Creating a Database Connection

In order to interact with a database, we need to do several things:

1. Register the JDBC driver
2. Open a connection using:

- Database URL
- Username
- Password

3. Execute some SQL statement using either:

- `Statement`
- `PreparedStatement`
- `CallableStatement`

4. Retrieve the results that are returned in a `ResultSet` object

Simple & Prepared Statements

`Statement` and `PreparedStatement` Interfaces

`Statement` interface is used for executing static SQL statements. `PreparedStatement` interface is used for executing pre-compiled SQL statements.

Once we have the `Connection` object, we can write our SQL and execute it:

```
Statement stmt = conn.createStatement();

String sql = "SELECT * FROM employees";

ResultSet rs = stmt.executeQuery(sql);
```

Alternatively, a `PreparedStatement` can be used. This interface gives us the flexibility of specifying parameters with the `?` symbol. It also protects against SQL injection when user input is used by pre-compiling the SQL statement.

```
PreparedStatement ps = conn.prepareStatement();
String sql = "SELECT * FROM employees WHERE age > ? AND location = ?";
ps.setInt(1, 40);
ps.setString(2, "New York");
ResultSet rs = ps.executeQuery(sql);
```

The `Statement` and `PreparedStatement` also have additional methods for sending SQL, including:

- `.execute()` - for any kind of SQL statement, returns a `boolean`
- `.executeUpdate()` - for DML statements, returns an `int` which is the number of rows affected

Retrieving Results

Results from an SQL query are returned as a `ResultSet`, which can be iterated over to extract the data:

```
List<Employee> empList = new ArrayList<>();
while (rs.next()) {
  int id = rs.getInt("id");
  String name = rs.getString("first_name");
  empList.add(new Employee(id, name));
}
```

Callable Statement

`CallableStatment` Interface

`CallableStatement` interface is used to execute stored procedures. You create an instance of a `CallableStatement` by calling the `prepareCall()` method on a connection object.
A `CallableStatement` can return one `ResultSet` object or multiple `ResultSet` objects.

```
//con is the Connection object
//It calls the procedure 'myprocedure' that receives 2 arguments.
CallableStatement stmt=con.prepareCall("{call myprocedure(?,?)}");
```

Example:

In this example, we are going to call the stored procedure `INSERTROWS` that receives `id` and `name` as the parameter and inserts it into the table `STUDENT`.

The `STUDENT` table structure is given below:

```
create table STUDENT(id number(10), name varchar2(200));
```

And `INSERTROWS` stored procedure looks like:

```
create or replace procedure "INSERTROWS"
(id IN NUMBER,
name IN VARCHAR2)
is
begin
insert into user420 values(id,name);
end;
/
```

Accessing Stored Procedures

Following code inserts new records into the `STUDENT` table by calling the above created stored procedure `INSERTROWS`, using a callable statement.

```
public static void main(String[] args) throws Exception{


//Registering the Driver
Class.forName("oracle.jdbc.driver.OracleDriver");
//Getting the connection
Connection con=DriverManager.getConnection( "jdbc:oracle:thin:@localhost:1521:xe","sy
stem","oracle");


//Preparing a CallableStateement
CallableStatement stmt=con.prepareCall("{call INSERTROWS(?,?)}");


stmt.setInt(1,1011);
stmt.setString(2,"John");
stmt.execute();


System.out.println("success");
```

```
}
}
```

JDBC Properties Files

References

- [Properties and Environment Oracle tutorial]

JDBC Connection

When making a JDBC connection to a database you may have noticed the code typically looks like this.

```
 DriverManager.getConnection(URL, CONNECTION_USERNAME, CONNECTION_PASSWORD);
```

But, where are we getting those values? Well up until now you have probably seen them as plain text Strings. As you can imagine this is not secure and requires you to change the code anytime the password etc. is updated.

To solve this problem developers began to collect all of this information into properties files. A properties file stores information as key value pairs, each on their own line; the file has a .properties extension. For instance you might have,

```
URL=jdbc:postgresql://localhost:5432/PubHub
CONNECTION_USERNAME=user
CONNECTION_PASSWORD=password
```

Then you can use the Properties class and FileInputStream to use these properties in your class.

```
FileInputStream fileStream = new FileInputStream("pathtopropertiesfile");
Properties properties = new Properties();
properties.load(fileStream);
URL = properties.getProperty("URL");
CONNECTION_PASSWORD = properties.getProperty("CONNECTION_PASSWORD");
CONNECTION_USERNAME = properties.getProperty("CONNECTION_USERNAME");
```

Great! So this solves some of our problem, but we are still saving our information as plain text. Now it's just in a different kind of file.

Aside: If you are using this method to read in properties for a web application that will be deployed on a server, you may need to use a different approach. When reading in a file, if you use a relative path it will be relative to the working directory. This can be unpredictable when it comes to servers, so instead we can use this method in those instances.

```
Properties prop = new Properties();

try {

   InputStream dbProps = ConnectionUtil.class.getClassLoader().getResourceAsStream("d
atabase.properties");

   prop.load(dbProps);

} catch (Exception e) {

   LogUtil.logException(e, ConnectionUtil.class);

}
```

So the next step is to include the information as System Environment Variables instead of just plain text.

Unfortunately, Java on its own will not parse our System environment variables into the application.properties file. When you begin using certain frameworks like Spring you may be able to change your .properties file to look like this.

```
URL=${URL}

CONNECTION_USERNAME=${CONNECTION_USERNAME}

CONNECTION_PASSWORD=${CONNECTION_PASSWORD}
```

However for now we can simply use our properties to read in values from the System through the System library.

```
URL=url

CONNECTION_USERNAME=connectionUsername

CONNECTION_PASSWORD=connectionPassword
```

Then on the system we would need to make sure to create url, connectionUsername, and connectionPassword respectively. You can do this in Eclipse if you don't want to set them system wide. Simply go to Run Configurations and tab over to the Environment tab.

Then in our Java class we'll use our values to read in from the System variables.

```
    FileInputStream fileStream = new FileInputStream("pathtopropertiesfile");

    Properties properties = new Properties();

    properties.load(fileStream);

        String url = properties.getProperty("URL");

        String password = properties.getProperty("CONNECTION_PASSWORD");

        String username = properties.getProperty("CONNECTION_USERNAME");
```

```
        URL = System.getenv(url);
        CONNECTION_PASSWORD = System.getenv(password);
        CONNECTION_USERNAME = System.getenv(username);
```

And then we should be good to go!

Design Pattern: Data Access Object (DAO)

Data Access Objects

When writing JDBC code, the application business logic may get mixed in with JDBC boilerplate code for querying the database, resulting in hard to read spaghetti code. One way to address this problem is to logically separate the code that accesses the database into Data Access Objects - this is referred to as the **DAO design pattern**.

To use the DAO design pattern, define an interface which declares methods through which the database will be queried. Then, concrete implementation classes can implement the interface and contain the data access logic to return the required data.

For example, if we have an Employee table in our database we'd like to query, we would create a `EmployeeDAO` interface:

```
public interface EmployeeDAO {
  // define some CRUD operations here
  public List<Employee> getAllEmployees();
  public List<Employee> getEmployeesByLocation(String location);
  public void updateEmployeeById(int id);
  public void deleteEmployeeById(int id);
  public void addEmployee(Employee e);
}
```

This interface would be implemented for a specific database - e.g. Oracle:

```
public class EmployeeDAOImplOracle implements EmployeeDAO {
  public List<Employee> getAllEmployees() {
    List<Employee> list = new ArrayList<>();
    // JDBC code here...
        return list;
  };
  public List<Employee> getEmployeesByLocation(String location) {
    List<Employee> list = new ArrayList<>();
```

```
    // JDBC code here...
         return list;
  };
  public void updateEmployeeById(int id) {
    // JDBC code here...
  };
  public void deleteEmployeeById(int id) {
    // JDBC code here...
  };
  public void addEmployee(Employee e) {
    // JDBC code here...
  };
}
```

Now whenever we need to query the Employee table in the database, we have a simple, clean interface which abstracts the data access logic:

```
EmployeeDAO dao = new EmployeeDAOImplOracle();
List<Employee> allEmpls = dao.getAllEmployees();
allEmpls.forEach( e -> System.out.println(e));

List<Employee> NYEmpls = dao.getEmployeesByLocation("New York");
NYEmpls.forEach( e -> System.out.println(e));
```

Also, we can simply swap out the concrete class `EmployeeDAOImplOracle` for another database-specific class if we need to at some point in the future, since we rely only on the `EmployeeDAO` interface. The implementation doesn't even need to be talking to a database - it could be reading and writing to files for all we know! We don't care **how** the data is being read or written, we just care **what operations** are defined for the object. That is the benefit the DAO design pattern brings to the table.