

Enterprise Computing

ASSIGNMENT 1

JACEK JANCZURA

MART. NR: 404975

Important info and credentials

Github repo: <https://github.com/jjanczur/ATMSystem>

AWS account ID: 701266085448

DB type: MySQL

DB credentials:

- **DB_URL**=jdbc:mysql://ec-atm-assignment1.cpmccfa2pqj.eu-central-1.rds.amazonaws.com:3306/innodb
- **USER**=Ecatm
- **PASS**=EnterpriceComputing1

(Note that the pass is with „c” not „s”)

To run ATM System execute jar by command: `java -jar ATMSystem-1.0-SNAPSHOT-shaded.jar`

System requirements

1. Relational database
2. ATM Java Client
 - a. Login using her account number and corresponding pin code
 - b. Deposit funds
 - c. Withdraw funds
 - d. Display account balance
3. Concurrent Access
 - a. Many ATM transactions happen in parallel
 - b. The account balance must not be negative

System requirements

1. Relational database
2. ATM Java Client
 - a. Login using her account number and corresponding pin code
 - b. Deposit funds
 - c. Withdraw funds
 - d. Display account balance
3. Concurrent Access
 - a. Many ATM transactions happen in parallel
 - b. The account balance must not be negative

1. Relational database

1. Create DB query:

- a. `CREATE TABLE IF NOT EXISTS account (account_id INT, pinCode INT, balance decimal(10,2) unsigned, UNIQUE (account_id))`
- b. `CHECK (balance > 0)` could be added to balance column. Unfortunately CHECK clause is parsed but ignored by all storage engines.¹ So there is no point in adding CHECK clause because it does nothing - as the manual states
- c. System requirement 3.b. - type of balance column is unsigned in combination with `sql_mode = 'STRICT_TRANS_TABLES'` it results in `SQLException`. When ATM tries to update balance with the negative value an exception is thrown with the message "Data truncation: Out of range value for column 'balance' at row n"

`SET @@GLOBAL.sql_mode = 'STRICT_TRANS_TABLES';` - this mode should not be set globally


`SET @@SESSION.sql_mode = 'STRICT_TRANS_TABLES';` - this command is executed for each session

2. Inserting values:

- a. `INSERT INTO account VALUES(1, 1111, 50), (2, 2222, 50), (3,3333,50)`

¹ <https://dev.mysql.com/doc/refman/8.0/en/create-table.html>

1. AWS management console

 Services ▾ Resource Groups ▾ 🔔

🔔 Jacek Janczura ▾ Frankfurt ▾ Support ▾

Amazon RDS ×

Dashboard

Instances

Clusters

Performance Insights

Snapshots

Reserved instances

Subnet groups

Parameter groups

Option groups

Events

Event subscriptions

Recommendations 1


RDS > Instances > ec-atm-assignment1

ec-atm-assignment1

Modify Delete Instance actions ▾


Summary

Engine MySQL 8.0.11	DB instance class db.t2.micro Info	DB instance status available	Pending maintenance none
------------------------	---	---------------------------------	-----------------------------


CloudWatch (17) 

Legend: ec-atm-assignment1


Add instance to compare Monitoring ▾ Last Hour ▾

< 1 2 3 > 

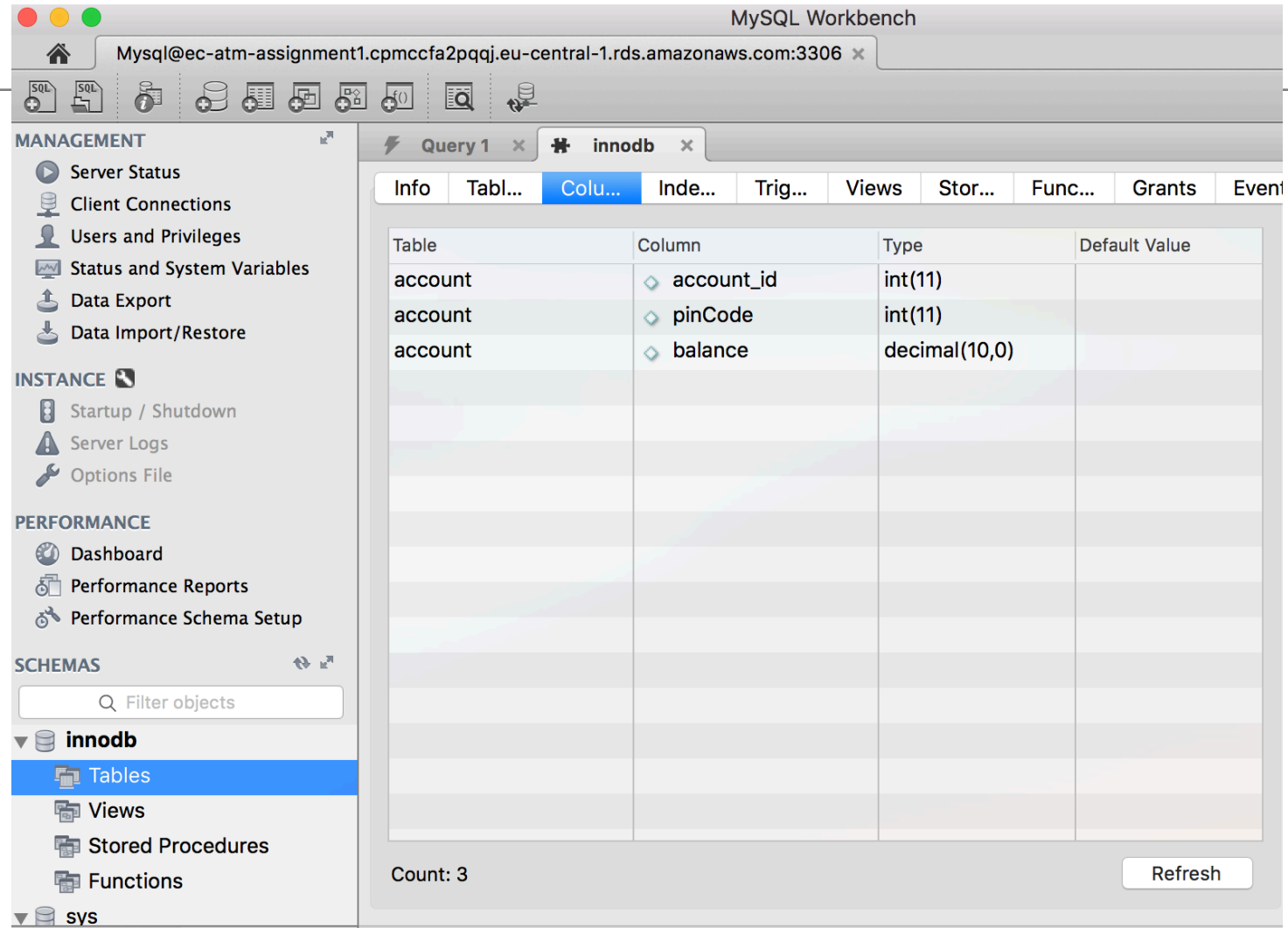
CPU Utilization (Percent)



DB Connections (Count)



1. MySQL Workbench



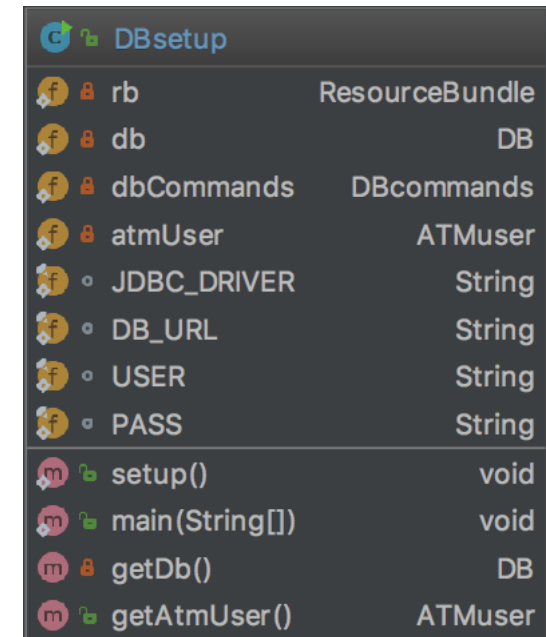
System requirements

1. Relational database
2. ATM Java Client
 - a. Login using her account number and corresponding pin code
 - b. Deposit funds
 - c. Withdraw funds
 - d. Display account balance
3. Concurrent Access
 - a. Many ATM transactions happen in parallel
 - b. The account balance must not be negative

2. Java Client

The Java Client consists a few classes:

- DBsetup – Class that is responsible for setting up the connection with DB.
 - It reads the DBconfig.properties - file where credentials are stored (file added to .gitignore)
 - Runs a main function where user menu display is called
 - Creates the connection to DB and passes the connection into object of the class DB



DBsetup		
f	rb	ResourceBundle
f	db	DB
f	dbCommands	DBcommands
f	atmUser	ATMuser
f	JDBC_DRIVER	String
f	DB_URL	String
f	USER	String
f	PASS	String
m	setup()	void
m	main(String[])	void
m	getDb()	DB
m	getAtmUser()	ATMuser

2. Java Client

The Java Client consists a few classes:

- DBcommands– Class that executes whole queries to DB. Methods from this class are used by ATMUser. All of the methodes are SQLInjection safe due to the correct usage of PreparedStatement
- atmLogin() – method that executes select query to find out if account id and pin are correct. If they are the method returns true otherwise false
- getBalance() – method that executes select query to find the balance of the user account.
- moneyWithdrawal() –
 - * Methode to perform withdrawal operation
 - * 1. Check if user is correctly logged in.
 - * 2. Perform sql update query - money withdrawal
 - * 3. Since we are in strickt mode and balance column is unsigned any update that may result in negative balance will throw an exception.
 - * So setting up negative balance is equal to trying update decimal column with string value – the exception will be thrown and the transaction will be rolled back. There is no option that the column balance will be negative because it's forbidden such as forbidden will be updating decimal with string. You can do that but the operation will throw an exception.
 - * Methode is SQLInjection safe due to usage of PreparedStatement
 - * **@param** atmUser Object that represents user
 - * **@param** amount Amount of money to withdrawal from users bank account
 - * **@throws** SQLException Problem with sql statement execution thrown by db
 - * **@throws** ATMaccountException Exception it inform about obligatory login
 - */
- Money deposit is withdrawal but with opposite value so the same method can be used.
- Demarcation – setting up auto commit off is redundant but since it was on the lecture I just want to show that I know how to use it

DBcommands		
f	db	DB
m	getDb()	DB
m	setDb(DB)	void
m	atmLogin(String, String)	boolean
m	getBalance(ATMUser)	double
m	moneyWithdrawal(ATMUser, Double)	void

¹ <https://dev.mysql.com/doc/refman/8.0/en/xa-states.html>

2. Java Client

The Java Client consists a few classes:

- ATMUsers - class that represents the user which is using the ATM. This class executes the commands from Dbcommands, stores the login and password and information if the user that is using the ATM is correctly logged in.
- At first I wanted it to extend Dbcommands but now I've decided to pass in a constructor Dbcommands object that will execute queries against one DB in one DB connection

ATMuser		
f	login	String
f	pass	String
f	isLogged	boolean
f	dbcommands	DBcommands
m	login()	void
m	getLogin()	String
m	setLogin(String)	void
m	setPass(String)	void
m	isLogged()	boolean
m	getBalance()	double
m	withdrawal(Double)	void

2. Java Client

The Java Client consists a few classes:

- DB – Class which objects represent a connection to DB
- Object of this class can:
 - Enable and disable demarcation mechanism which is a base for keeping atomicity of each update transaction. Only fully correct transaction is committed to DB.
 - Setup a connection with DB
 - To keep high isolation of transaction even though the default transaction isolation level for MySQL db is REPEATABLE_READ¹ was changed to SERIALIZABLE. Due to this isolation level anomalies such as dirty read, non-repeatable read and phantom read are prevented
 - To force db to throw *SQLException* when client tries to update balance with negative values (even though positive balance is checked during each withdrawal otherwise transaction is rolledback) `sql_mode = 'STRICT_TRANS_TABLES'` is eabled

DB		
f	JDBC_DRIVER	String
f	DB_URL	String
f	user	String
f	pass	String
f	conn	Connection
f	stmt	Statement
m	loadJdbcDriver()	void
m	login()	void
m	createStatement()	void
m	enableDemarcation()	void
m	disableDemarcation()	void
m	getJDBC_DRIVER()	String
m	getDB_URL()	String
m	getConn()	Connection
m	getStmt()	Statement

¹ <https://dev.mysql.com/doc/refman/5.7/en/innodb-transaction-isolation-levels.html>

2. Java Client

The Java Client consists a few classes:

- Menu – Class that is responsible to correctly display user menu and react for user choices.
 - Class show to the user options that can be used in ATM.
 - For testing purpose hidden option 5 has been added - execute withdrawal without demarcation and balance check
 - It calls the methods to withdrawal, deposit, login and perform the tests
- ATMaccountExceptio – This exception is thrown when the user is not logged in and tries to perform any operation on the account or user tries to login with wrong credentials
- BalanceException – This exception is thrown on every attempt setting up the balance below zero

Menu		
f	dbSetup	DBsetup
m	displayMenu()	void
m	executeChoice(int)	void
m	performWithdrawalDeposit(boolean)	void
m	getBalance()	void
m	performLogin()	void
m	atmLogin(String, String)	void
m	getDBsetup()	DBsetup
m	setDBsetup(DBsetup)	void
m	performSystemTest()	void

2. Java Client

DB testing – The basic test done by user in “one ATM”

1. Login to user 1 with pass 1111
 - withdrawal money,
 - deposit money,
 - withdrawal more than the balance
2. Repeat the same operation on the other user for example user 2 with pass 2222
3. trying to login with wrong credentials and repeat nr 1.

This type of easy testing checks if the basic methods and general architecture are working correctly but the real test is to login to different ATMs with the same credentials in the same time and trying to validate ACID principles in the system.

2. Java Client

DB testing – The real test done by user many ATMs at the same time

To talk about concurrent operations we need to create a few ATMs, run them in a different threads, and try to withdrawal some money

1. Class SystemTest which implements interface runnable was created. The class creates own db connection, setup DB, passes it to DBcommands and creates user object which can execute queries against this db. This setup is equivalent to one separate ATM with one user.
2. User tries to withdrawal amount lower than balance on his account its hardcoded to 5 and the user is hardcoded to id 1
3. Test is performed by creating and running 10 threads. To each thread is assigned processor time and we can assume that the threads are working “concurrently” since in most of the computers we have more than one core.
4. Results in a file *DBTest.log* and *DBTest2.log*

```
private void performSystemTest() {  
  
    int threadsNum = 10;  
    Runnable[] ATMs = new Runnable[threadsNum];  
    Thread[] threads = new Thread[threadsNum];  
    for (int i = 0; i < threadsNum; i++) {  
        ATMs[i] = new SystemTest(i);  
    }  
  
    for (int i = 0; i < threadsNum; i++) {  
        threads[i] = new Thread(ATMs[i]);  
    }  
    for (int i = 0; i < threadsNum; i++) {  
        threads[i].start();  
    }  
    displayMenu();  
}
```

System requirements

1. Relational database
2. ATM Java Client
 - a. Login using her account number and corresponding pin code
 - b. Deposit funds
 - c. Withdraw funds
 - d. Display account balance
3. Concurrent Access
 - a. Many ATM transactions happen in parallel
 - b. The account balance must not be negative

Concurrent access and positive balance

When multiple users are updating the same data simultaneously may result in inconsistent data.

To keep ACID principles, positive balance and assure correct concurrent access a few steps were taken:

1. Server side:
 - a. balance column is set to unsigned – as one of the requirements is to keep balance always positive
2. Client's side:
 - a. For any UPDATE queries autocommit was turned off. In case of failure db exception in any part of the transaction whole transaction was rolledback. Otherwise transaction is committed.
 - b. To ensure full isolation *TRANSACTION_SERIALIZABLE* was turned on. Due to this anomalies such as dirty read, non-repeatable read and phantom read will not occur.
 - c. To force DB to throw an error not just update balance column value with 0, SESSION.sql_mode = 'STRICT_TRANS_TABLES' is enabled as a session mode to each of the connections from each of the ATMs

Sum up

Protection on a Java clients side and DB side keeps the balance positive. It's important to note that within one atomic transaction I don't perform any balance check - update with commit or rollback is performed in just one atomic transaction so any dirty read won't occur. 2PC protocol is used² on purpose, but can be omitted and each of the transactions will still be atomic, balance will not be negative. Even if we turn off the power in ATM it will not change anything in db state.

DB is on AWS so I don't need to take care of the servers, backups and transaction logs¹ what makes the DB durable.

All transactions are isolated due to full isolation – SERIALIZABLE mode, DB is consistent between many ATMs.

Taking into consideration all previously mentioned facts I can assume that my DB system fulfills all the requirements.

¹ <https://dev.mysql.com/doc/refman/8.0/en/binary-log.html>

² <https://dev.mysql.com/doc/refman/8.0/en/xa.html>