

Report for Database System and Programming

by Nathan Christopher Sathanandan

FILE	25295_NATHAN_CHRISTOPHER_SATHANANDAN_REPORT_FOR_DATABASE_SYSTEM_AND_PROGRAMMING_318857_1772179745.PDF (692.94K)		
TIME SUBMITTED	15-JAN-2021 02:30PM (UTC+0000)	WORD COUNT	5892
SUBMISSION ID	142268986	CHARACTER COUNT	29362

COMBINED REPORTS
FOR PROGRAMMING
AND SOFTWARE
TESTING AND
DATABASE SYSTEMS

Nathan Sathanandan (2012819)

University of Wales Trinity Saint David

Programming and Software Testing Reflection

Introduction

For this report I will be reflecting upon my design decisions for the vehicle tax software that I implemented. The objective was to write java code that would interact with a database that I created, the code would then allow the user to see vehicles close to their expiry tax date, view drivers and add drivers.

Basic Design Description

My first decision was having to create a function called list options that used a switch statement to easily allow a user to choose what they wanted to do. This list function easily allowed me to map out how I wanted the main functions to interact with each other. Figure 1.2.1 the mind map of how I wanted my basic functions to work and interact throughout the whole software.



Figure 1.2.1: Design mind map

All of the main branch list functions connect to a check function to ensure that all the inputs from the user were correct and therefore greatly reducing the chances of any errors. The check function then connects to the SQL query functions which take in variables to try and optimising the reusability of code and therefore saving time having to rewrite out SQL queries. The SQL function then receives the query functions and pulls the data requested from the database and either prints the data straight away or returns a value to the main branch functions.

Features Implemented in Functions

The first main branch function implemented was the find a driver/vehicle function which allows a user to find the details for a driver/vehicle. The function works by asking what details they would like to use to search for the specific driver or vehicle, the options are to use the number plate, first and last name, driver number, address id or mot id. Then by using a SQL query function like the one in figure 1.3.1 combined with an SQL function, the results are printed to the console.

```

public static String viewAllInformationOfSpecificDriverWithNumberPlate(String numberPlate){
    String queryString = "SELECT vehicle.reg_num, driver.first_name, driver.last_name, driver.date_of_birth," +
        "vehicle.tax_expiry_date, mot_status, sorn_status, address.name_or_number, address.street, " +
        "address.city, address.postcode FROM vehicle INNER JOIN driver ON driver.driver_no = " +
        "vehicle.driver_no INNER JOIN address ON driver.address_id = address.address_id INNER JOIN mot " +
        "ON vehicle.mot_id = mot.mot_id INNER JOIN sorn ON vehicle.reg_num = sorn.reg_num WHERE vehicle.reg_num" +
        " = '" + numberPlate + "'";
    return queryString;
}

```

Figure 1.3.1: Example query string for view drivers

This function also uses a SQL function that takes in a query string and works out all the columns names for the table created with the query string and returns it as an array. By returning the function as a string it also makes it easily testable. Figure 1.3.2 shows what the function looks like.

```

public static String[] tableColumnNames(Connection conn, String queryString) throws SQLException{
    try {
        Statement statement = conn.createStatement();
        ResultSet results = statement.executeQuery(queryString);
        ResultSetMetaData resultsMetaData = results.getMetaData();
        int i = resultsMetaData.getColumnCount();
        String tableHeadings[] = new String[i];
        int j = 0;
        while(j < i){
            tableHeadings[j] = resultsMetaData.getColumnName(j+1);
            j++;
        }
        return tableHeadings;
    } catch (SQLException e) {
        e.printStackTrace();
        throw e;
    }
}

```

Figure 1.3.2: Table column name SQL function

The next main branch function created was the add a driver to the database. This function simply took the drivers details from the user which were then added to pre made SQL functions queries and then into a SQL function which would add all the data to the database. This section is where all the input checks and variable generators were created, they checked the date format of a variable; made sure a variable didn't contain numbers; checked the number plate was correctly formatted correctly; the email was formatted correctly; made sure that the postcode was formatted correctly out of 6 possible formats; checks a valid mot or tax date is used and checks the driver is at least 17. There is also a function that calculates the new tax end date based on the duration (6 or 12) that the user choses. The two functions used to calculate the new dates are shown below in figure 1.3.3. For easy I decided to use 365 days to represent 1 year and 183 days for 6 months.

```

public static String addYearToTheDate(String date) throws ParseException {
    SimpleDateFormat sdf = new SimpleDateFormat( pattern: "yyyy-MM-dd");
    Calendar c = Calendar.getInstance();
    c.setTime(sdf.parse(date));
    c.add(Calendar.DATE, amount: 365);
    date = sdf.format(c.getTime());
    return date;
}

public static String add183DaysToTheDate(String date) throws ParseException {
    SimpleDateFormat sdf = new SimpleDateFormat( pattern: "yyyy-MM-dd");
    Calendar c = Calendar.getInstance();
    c.setTime(sdf.parse(date));
    c.add(Calendar.DATE, amount: 183);
    date = sdf.format(c.getTime());
    return date;
}

```

Figure 1.3.3: Adding 1 year and 6 months functions.

Because of the way I created my primary keys using hard code I had to make sure that the SQL function that entered the address, driver, mot and vehicle details (in that order) were all in one rather than having separate functions.

The third main branch function that was created was to see vehicles that tax was close to their expiration date. Rather than having the feature only allow to see vehicles within a set period from the tax expiry date, I chose to allow the user to choose the range by entering the amount of days. This meant you could see all the vehicles that were in between 1 and however many days the user chose. This was implemented by using a SQL query string that takes days the user entered (shown in figure 1.3.4 below) and combined with an SQL prepared statement which then prints the data to the console.

```

public static String taxExpiryQuery(int days){
    String queryString = "SELECT vehicle.driver_no, vehicle.reg_num, driver.first_name, driver.last_name, " +
        "vehicle.tax_expiry_date, address.name_or_number, address.street, address.city, address.postcode" +
        " FROM vehicle INNER JOIN driver ON driver.driver_no = vehicle.driver_no INNER JOIN address " +
        "ON driver.address_id = address.address_id WHERE ((tax_expiry_date - CURRENT_DATE)) < "+days+" AND ((tax_expiry_date - CURRENT_DATE))>0";
    return queryString;
}

```

Figure 1.3.4: Tax expiry SQL query

A test SQL function was also created so that Junit tests could be written to check the table output was correct for different number of days. Figure 1.3.5 below shows the function, as you can see it populates 9 different single arrays with the column data and then converted into a 2d array format.

```

public static String[][] taxExpiryTest(Connection conn, int days) throws SQLException{
    String queryString = sqlQueryStrings.taxExpiryQuery(days);
    try{
        Statement statement = conn.createStatement();
        ResultSet results = statement.executeQuery(queryString);
        ResultSetMetaData resultsMetaData = results.getMetaData();
        int count = countRows(conn,sqlQueryStrings.taxExpiryQuery(days), columnLabel: "tax_expiry_date");
        String[][] table = new String[count][resultsMetaData.getColumnCount()];
        int i=0;
        while (results.next()){
            table[i][0] = results.getString( columnLabel: "driver_no").trim();
            table[i][1] = results.getString( columnLabel: "reg_num").trim();
            table[i][2] = results.getString( columnLabel: "first_name").trim();
            table[i][3] = results.getString( columnLabel: "last_name").trim();
            table[i][4] =results.getString( columnLabel: "tax_expiry_date").trim();
            table[i][5] =results.getString( columnLabel: "name_on_number").trim();
            table[i][6] =results.getString( columnLabel: "street").trim();
            table[i][7] =results.getString( columnLabel: "city").trim();
            table[i][8] =results.getString( columnLabel: "postcode").trim();
            i++;
        }
        return table;
    } catch (SQLException e){
        e.printStackTrace();
        throw e;
    }
}

```

Figure 1.3.5: The SQL tax expiry function

The next main branch implemented was the update sorn, this allowed the user to update the sorn details of a vehicle. A simple update statement was used for this function where a driver enters the number plate of the vehicle, they want to change the sorn details and gives them the option to make the car sorn or make the car taxable again.

The fifth main branch function that was implemented was for statics, the user could choose from 6 different statics (how many cars within 28 days of expiring, how many cars are untaxed, how many cars are declared sorn, percentage people passed mot, how many drivers are in the database and how many vehicles are in the database). Majority of the statics use a simple select table query with a prepared statement that counted the rows in the table.

The last main branch Function that was created was the update driver details function. This function gave the user to update specific things within the database. For example, the user could choose to update the tax of a vehicle with the registration number once they provider the registration number. However, if a vehicle has a mot that has failed, the car is still taxed, or the vehicle is declared sorn you won't be able to update the vehicle tax (as you wouldn't be able in real life). For the address details a person wouldn't be able to update the address if that address was assigned to more than one person. The function would make you add a new address and then the function will calculate what the new address primary key is and update the driver table with the new address id. The other two updates that could be completed was to the personal details and the mot. For the personal details a user could only update the surname, contact number and email address as those were the only details that could possibly change (rarely do people ever change their first name). For the mot a user could change everything except the mot id and next mot date. The next mot date couldn't be changed as it was an auto generated date based on the date that was inputted for the mot date.

Conclusion

Overall, I think that a lot of interesting features were implemented in this programme, they mainly had to do with validation of input and interacting with the database. I think that I used the java code to work with the database in unique ways which allowed the programme to seem more realistic. One of the problems with the programme I didn't notice until the end was the tests, I created for my SQL functions wouldn't always be correct. This is because once a user interacts with the database and makes changes then all of the return SQL functions have different numbers. This means that you would have to go through the database each time to work out the new values and then input them into the expected section of the Junit test. Another flaw that I didn't think of when I planned the architecture of my programme and database was to allow for a user to delete a row. Whilst this seems like a simple task, I thought I would add my primary key using hard code rather than using the serial option within PostgreSQL. This meant any time you wanted to delete a column you would have to reset the simultaneously update all of the primary keys in the rest of the table so that they are in order. So, whilst I slightly planned my architecture for my programme it shows that I didn't plan it out as thoroughly as I should've. However, the features that were implemented I think work really well together and I think a lot of the code that I created was made so that It could be re-used for multiple different things.

Programming and Software Testing Research

Introduction

This report looks the importance of four major topics within software development. The first topic is cohesion and the importance of writing highly cohesive classes. The second topic is about loose coupling and why it needs to be implemented within applications. The third topic is why every programme should ensure they use data hiding techniques when creating applications and the fourth topic looks at the importance of information hiding.

High cohesion

When looking at high cohesion in the software engineering world you first have to understand what cohesion means. Cohesion is defined as the act or state of keeping together [1], when it comes to coding (Java) this term is used to describe the relationship between functions within a single task to perform a specific functionality [2]. In layman's terms it means if a group functions within a class belong together and whether they all work well together to accomplish a single well-defined task [3]. When a class contains functions that work well together to complete to complete a task it has a high cohesion and when functions within a class don't work well together, they have a low cohesion. Two cohesion examples are shown below.

```

class getDriverInformation {
    public void displayMessage(String message){...}
    public boolean fileExists(String filename){...}
    public String captureInput(){...}
    public String[] getClassMethods(Object o){...}
}

```

Figure 2.2.1: A class with low cohesion

```

class ObjectInformation {
    public String getObjectClassName(Object o){return ...}
    public String getObjectSuperclassName(Object o){return ...}
    public ArrayList<String> getObjectFields(Object o){return ...}
    public ArrayList<String> getObjectConstructors(Object o){return ...}
    public ArrayList<String> getObjectMethods(Object o){return ...}
}

```

Figure 2.2.2: A class with high cohesion

Figure 2.2.1 is perfect example of function with low cohesion, none of the methods within the class seem to function well together to complete the task of the class. When looking at figure 2.2.2 it's evident that the methods within the object information class have been well constructed so that they would have high cohesion [4].

All developers aim to create classes and methods with a high cohesion because it makes the classes much easier to maintain compared with classes with low cohesion as they don't need to be changed as frequently [4]. The other huge benefit to having classes with high cohesion which has a well-defined task is that it can easily be re-used within your code, this means that high cohesion classes lead to the user reducing the amount of code they need to write [4]. High cohesion classes also tend to be easier test and understand as they have been written so the code is readable [4].

Loose coupling

Loose coupling is tightly linked to high cohesion in that when a class has is highly cohesive then it is highly likely that the class will also be loosely coupled. When trying to understand what loose coupling is, there needs to be an understanding of what coupling is. Within the software world coupling means how dependent one method is on another method [5]. Classes or methods will either be loosely coupled or tightly coupled with each other (similar to how classes have high or low cohesion). When classes are tightly coupled it means that the two classes are highly reliant on each other meaning that when one class changes so those the other [6]. A good an analogy for tight coupling is if you want to change the skin, you would have to change the body as well as they are joined together [6]. An example of two classes with tight coupling is shown below.

```

class Subject {
    Topic t = new Topic();
    public void startReading()
    {
        t.understand();
    }
}
class Topic {
    public void understand()
    {
        System.out.println("Tight coupling concept");
    }
}

```

Figure 2.0.1.1: Two tightly coupled classes

Figure 2.3.1 classes are clearly tightly coupled because the subject class is heavily reliant on the topic class, if any changes were made to the topic class then changes would also need to be made to the subject class.

When class's or methods are loosely coupled it means that they aren't reliant on each other and that they work independently [6]. A good analogy for loose coupling is 'if you change your shirt, then you are not forced to change your body' - when this is possible then loose coupling has been achieved [6]. An example of loosely coupled functions are shown below.

```

class Volume
{
    public static void main(String args[])
    {
        Box b = new Box( length: 5, width: 5, height: 5);
        System.out.println(b.getVolume());
    }
}
final class Box
{
    private int volume;
    Box(int length, int width, int height)
    {
        this.volume = length * width * height;
    }
    public int getVolume()
    {
        return volume;
    }
}

```

Figure 2.3.2: Two loosely coupled classes

Figure 2.3.2 is a perfect example of loosely coupled classes as there are no dependency between the volume class and the box class. This means that if anything were to change in the box class then there wouldn't need to be any changes made within the volume class.

Keeping classes loosely coupled is very important when it comes to coding. This is because loosely coupled classes help improve the testability of an application when compared to tightly coupled classes [6]. Loose coupling makes it easier to change or swap pieces of code/modules/objects when compared to tight coupling [6]. When classes are written so that they are loosely coupled it makes it easier for the developers to change or grow their application in the future [6]. Loosely coupled classes also helps people follow the GOF (gang of four) [7] principal program interface unlike the tightly coupled functions [6]. Tightly coupled functions prevent code from being flexible and re-useable whilst loose coupling classes are the exact opposite, meaning they help reduce the amount of code that a develop writes.

Data hiding

Data hiding heavily used when it comes to object-oriented programming as it confirms the security of members of a class from unauthorized changes [8]. Using precautions like data hiding technique helps secure data, which is the most sensitive and volatile component of a program if the data was manipulated the program could provide an incorrect output [9]. Within java data hiding is achieved through the use of access modifiers which are public, private and protected [9]. When the access modifier on a class is public it is accessible from everywhere in the program meaning that there's no restriction on the scope public data members [10]. However, if you use the private access modifier on a class it means that private data is only accessible to data members within that class [10]. Therefore, any other class within the same package would not be able to access the class. If a class is declared protected, then all the data members are accessible within the same package or subclasses within different package [10].

For example, of data hiding. Imagine there's a class called check bank account which contained a function called bank balance [9]. This function allowed the user to check their bank account balance, this information would be considered highly sensitive [9]. Therefore, an outside application would be allowed to check the bank account balance, however, the application wouldn't be allowed to modify the account balance attribute [9]. Thus, by declaring the account balance attribute private the function would restrict the access to account balance from any outside applications. It's important that all programmers use data hiding techniques not only to increase the security of the programmes they write but to also help reduce the complexity of the system.

Information hiding

Information hiding is implemented to prevent system design change by hiding decisions so that specific code cannot not be modified or altered [11]. The idea is to hide the complexity of the system design from clients or end users [12]. Information hiding is commonly mistaken for encapsulation; however, the main difference is that encapsulation bundling of data with the methods that operate on that data and information hiding is specific implemented to hide design decisions rather than the data itself [13]. For example, in java you can have fully encapsulated data, but it's not actually hidden at all [13]. Information hiding is extremely important to implement as it allows you to have program that is easy to modify without worry about breaking the entire programme [13].

Conclusion

Overall, each of the topics discussed within the report should always be followed by any person who wishes to develop an application to make their easier and also to increase the security of their users. This is because by ensuring that the classes written are highly cohesive and loosely coupled the code produced will be easier to test, the code produced is much more flexible and re-useable, it's easier to swap pieces of code/modules/objects, applications can grow with easy as functions and classes don't have to be constantly edited as new features are added, helps people follow the GOF (gang of four) principal program interface, this all helps to reduce the amount of code that is written and repeated. Programmers using data and information hiding techniques are also ensuring that the application they create is safe from outside programs altering the data within their application, whilst at the same time reducing the complexity of

the system. Upholding all these standards and techniques would ensure that an application written by a developer would be perfectly organised and secure for anyone to use.

References

- [1] Anon 2021 COHESION I meaning in the Cambridge English Dictionary *Dictionary.cambridge.org*
- [2] Anon Definition: High cohesion *Educative: Interactive Courses for Software Developers*
- [3] Ross D 2016 *Learn Java Programming - Cohesion Tutorial* (youtube: Daniel Ross)
- [4] Anon 2017 Cohesion in Java - GeeksforGeeks *GeeksforGeeks*
- [5] Montiel I 2018 Low Coupling, High Cohesion *Medium*
- [6] Anon 2020 Coupling in Java - GeeksforGeeks *GeeksforGeeks*
- [7] Anon 2014 GOF / Gang of Four Design Patterns *JavaBrahman*
- [8] Anon 2014 What is Data Hiding? - Definition from *Techopedia.com*
- [9] Anon Difference Between Data Hiding and Encapsulation (with Comparison Chart) - Tech Differences *Tech Differences*
- [10] Anon 2020 Access Modifiers in Java - GeeksforGeeks *GeeksforGeeks*
- [11] Anon 2012 What is Information Hiding? - Definition from Techopedia *Techopedia.com*
- [12] Anon 2013 *Wiki.c2.com*
- [13] Rogers W 2021 Encapsulation is not information hiding *InfoWorld*

Database Systems Reflection

Introduction

For this report I'll be looking at the database that I created using PostgreSQL. The database was created to host data related to vehicle tax and using a java programme a user could then interact with the database. The main aim was to create a database that allowed drivers to own multiple cars, allow multiple drivers to live in the same address, have appropriate attributes, each table to have a primary key, one table to use a composite key, foreign keys to be used and have at least three check constraints

Database design

When designing the database, I thought of 5 different entities I wanted to add to the database design architecture. These were vehicle, driver, address, sorn and mot. Then by using all of these 5 entities I created an entity relationship diagram so I could determine the relationship between each entity. Figure 3.2.1 below shows what the final design that I implemented.

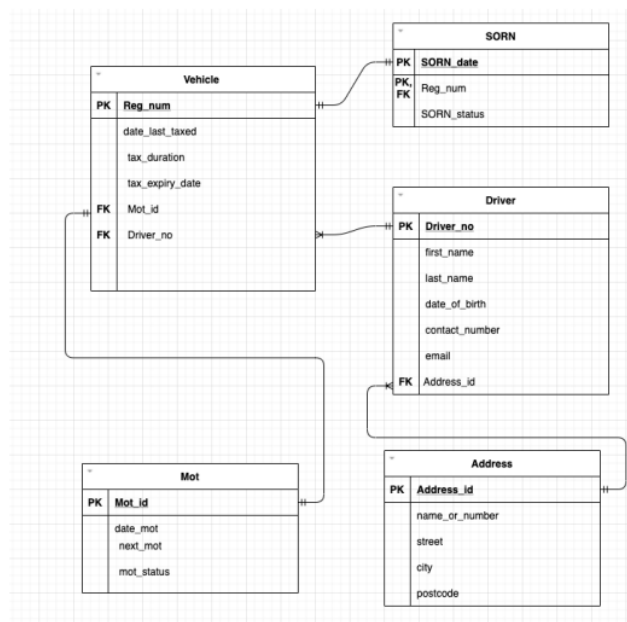


Figure 3.2.1: Entity relationship diagram

The structure of the database fits third normal form as I have reduced the any duplication to the least possible amount (can't stop people from having the same names, surnames, dates of birth etc) and removed any functional dependencies. The database was created so that there were no many to many relationships and therefore the eliminating the need for any bridging tables and reducing the complexity of the database. Within the database there are two one to one relationships and these are between the mot and vehicle tables and the sorn and vehicle tables. They both have a one-to-one relationship because a vehicle can only have one mot and one sorn status and the sorn status and mot can only have one vehicle to be related to. The second type of relationship within the database design a one-to-many relationship, which is between the vehicle and driver table and the address and driver tables. The driver and address table have this kind of relationship because many drivers can live in the same address, however, a driver can only be registered to one address. Therefore, the same logic can be applied to the vehicle and driver tables, a driver can have many cars, however, a vehicle can't have more than one registered owner. In total the database contains 4 primary keys and one composite key, the 4 primary keys are the reg_num, mot_id, driver_no and address_id. The composite key is made up of the sorn date and reg_num because they both can be used together to uniquely identify a row in the sorn table.

When inserting the SQL code, I made sure that I created the address table first, the driver table second, the mot table third, the vehicle table fourth and the sorn table last. The tables were created in this order to due to the foreign key constraints, as a table can't have a foreign key when the original primary key doesn't exist. Below I have given an example of the SQL code that I used to create the driver table.

Figure 0.1: SQL code for the vehicle table

Within the database there are a lot of possible transactions that could be used, however, from a real-life perspective I think the tax expiry update, the view driver details, the add driver and update mot details would be the most important transactions. I believe this because these are transactions that would be mostly commonly used in real life. For example, every time someone called the DVLA I'm sure they would have to do some sort of security, and the only way they could complete this security check is if they were viewing the driver details on the database. Whilst this might sound trivial at first but the DVLA mostly likely get 100's if not 1000's of calls a day relating to something like road tax. Therefore, by monitoring the transactions you could discover what the most common information given by customers to find them on the database and make more specific queries and also index them to improve the speed of the search. I think tax expiry updates are an incredibly crucial thing to monitor because the

database is designed to allow people to update their tax, if more efficient queries can be made then the tax update one should be the priority. Compared to the real vehicle tax database mine is meniscal that's why the add transaction should be highly monitored as many drivers and vehicles need to be added to the database and its essential that these transactions are completed as quickly because they are the longest possible transaction for the database currently. In the United Kingdom you can't update your vehicle tax if you the vehicle has failed its mot or if the mot is out of date. Therefore, it's high priority that the mot update transactions are as quick as possible so members of the public can update their vehicle tax.

Conclusions

Overall, I think I have created a well-designed database for this small-scale programme. However, I do believe that if I had more time, I would possibly add a few more authentication steps to allow users to update their vehicle tax. For example, I could have added a driver license that would've allowed me to authenticate a user more securely and also make sure that they are still allowed to drive. I also realised that I should have used the serial data types for the primary keys in the database rather than manually inputting all of them into the columns. This wouldn't have only saved me time but also would've mean that my java programme wouldn't have to calculate the new primary key for each insert.

Database System Research

Introduction

Many people around the world complete their daily task by interacting with a database. We use databases to complete our weekly shops, ordering items online, reading articles on the internet etc. However, most people never consider the importance of managing the transactions that are created when completing those tasks and how much goes behind maintaining the database because of these transactions. This report explores what transactions and concurrent transactions and the techniques used to maintain the integrity of the database during after these transactions.

Transactions

Within the topic of databases transactions are a logical unit that is independently executed for data retrieval or updates [1]. Database transactions are often known as a unit of work that is achieved within a database environment [1]. An example if someone were updating a record or creating a record or even deleting a record from the database table, then that person would be performing a transaction on that table [1].

Transactions are made up of 6 different states, which are the active state partially committed state, failed state, committed state, aborted state and finally the terminated state [2]. The active state is when the instructions of a transaction is running and if all the operations are completed without an error the transaction goes to the partially committed state [2]. Although if any of the operations do fail in the active state then the transaction moves into the failed state [2]. During the partially committed state changes are made to within the main memory or local buffer and then once the saving process is complete the transaction moves to the committed state [2]. However, if for some reason the transaction fails to save then the transaction is moved

to the failed state. The failed state is when a transaction has any type of failure, by moving the transaction type to the failed state the system can eliminate a transaction from making permanent incorrect changes to the database [2]. After the failed state a transaction moves into the aborted state and in the before states the changes are made to the local buffer or main memory, meaning these changes are deleted or rolled back [2]. After the partially committed state a transaction moves into the committed stage where changes are made permanent to the database and the transaction is complete [2]. After the committed or aborted state, the transaction moves into the final state known as the terminated state [2]. The terminated state this is where the old transaction is terminated and ready to deal with a new transaction [2]. Figure 4.2.1 below shows a diagram of the 6 transactional states.

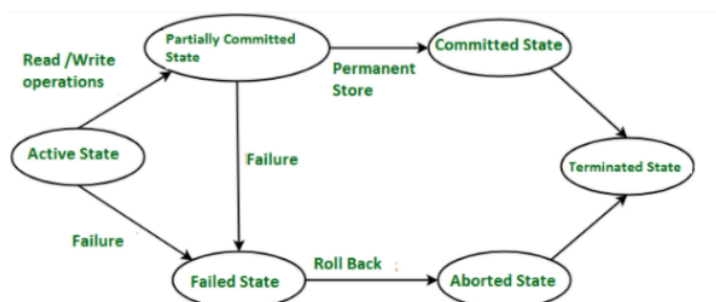


Figure 4.2.1: Diagram explaining the transaction state path

Concurrency

When multiple users have the ability to affect multiple transactions within a database then the database is using concurrency [3]. Concurrency is one of the major differences when database is compared to other alternative data storage methods such as spreadsheets [3]. This is why people store the majority of their data within in databases because it means that multiple users have the capability to view and work on different pieces of data within the same file [3].

Concurrent transactions can be a massive advantage when using databases, however, these advantages can be easily over shadowed when it comes to facing the problems of implementing concurrency. There are 5 major problems with concurrency which are known as temporary update (or dirty read), unrepeatable read, incorrect summary, lost update and phantom read [4]. A temporary update is when data is read from a partially committed transaction, it's known as a temporary update because the partially committed transaction could be moved to the failed state, leading to it being rolled back [4]. Therefore, these partially committed transactions could read the temporary update with values that don't actually exist within the database, which will lead to inconsistency within the database [4]. Unrepeatable read problems are when two or more read operations of the same transaction read different values for the same variable [4]. This can be a big problem as you could receive incorrect data if two transactions were running at the same time [4]. Incorrect summaries commonly occur when there is an aggregate function transaction being applied on some records whilst at the same time a transaction is updating the exact same records [4]. The aggregate function could possibly calculate use a mic of values from before and after the update transaction was completed resulting in an incorrect summary [4]. Lost update problems are cause when there are two update transactions which are run one after the other [4]. The data from the first transaction is overwritten by the second transaction leading to data being permanently lost [4]. The final big problem when it comes to concurrent

transaction is phantom reading, this is when a transaction reads a variable once but when it tries to read that same variable again, an error occurs because the variable no longer exists [4].

ACID

With the many problems that concurrent transactions have on the consistency of a database, people have started use what's known as ACID to help control concurrent transactions. ACID is an acronym where the A stands for atomicity, the C stands for consistency, the I stand's for isolation and the D stands for durability [5].

Atomicity means that an entire transaction must be completed, meaning that if the transaction is not completed the transaction is moved to the aborted state [5]. By doing this it is ensured that the database can never be left with partially filled completed transactions that would lead to poor data integrity [5]. A good example to why atomicity is essential is if someone tried to send money to someone else and the send request was successful but the receive request failed, you would want to cancel the entire transaction [5]. This is because money would leave your account but not actually end up going into the intended recipients account leaving it in a limbo state [5].

Consistency within the ACID acronym is in place to ensure the state of data when certain conditions are met [5]. For example, your database may have rule where an invoice has to be linked to a customer within a customer table [5]. However, these rules can be broken during a transaction, for example if an invoice was added to the database without having a related customer but was then added later during the same transaction [5]. Therefore, the temporary violations can only be seen within the transaction itself and not outside, plus all these violations are always resolved before a transaction is complete [5].

Isolation is one of the main parts of ACID that helps control concurrency transactions as it helps data being used during the processing of one transaction to not be used by another transaction until the first transaction is complete [5]. For example, if two people were trying to interact with a bank account with a balance of £1000 and the first person was trying to deposit £100 and the second person was charging £300 [5]. The transactions should read as £100 added to £1000 and then £300 subtracted from £1100 [5]. However, if the transactions ran at the same time and the 2nd transaction was completed before the first then both transactions would seem to have been completed but £100 would have gone missing [5]. Transaction isolation can be measured using four levels which are known as read uncommitted, read committed, repeatable read and serializable [6]. The lowest level of transaction isolation is read uncommitted, at this level one transaction can read changes that have not yet been committed by another transaction [7]. This isolation level is vulnerable to temporary updates, unrepeatable data and phantom reads [7]. With the read committed level data that is read is always committed at the moment it's read. Therefore, this eliminates the problems that arise when transactions are partially committed and are then rolled back (removes problems of temporary updates) [7]. This is achieved by the transaction holding a read or write lock on the current row and therefore preventing other transactions from manipulating the row [7]. Repeatable read level is higher than the read committed level because it holds read locks on all rows it references and writes locks on all rows that are inserted, updated or deleted [7]. This means other transactions cannot read, update or delete any of these rows which eliminates the problems of temporary updates and unrepeatable reads [7]. Finally, the highest level of transaction isolation is serializable execution [8]. This execution means that if you run transactions simultaneously you would have the same result as running the transactions serially (one after the other) [7]. Using this

level of isolation, you can prevent all the major problems with concurrency transactions, however, the transactions will take longer [7].

Durability within the ACID acronym represents how data should remain within a database even after a system failure as long as the data has been committed allowing the data's effects to remain [5]. During transactions data shouldn't be persistent therefore when databases crash, the restored version of the database will only contain the data prior to the incomplete transaction [5].

Conclusion

Overall, it's clear how important it's to consider implementing processes like ACID to ensure the integrity of database when you have concurrent transactions (which are present in all databases). The importance of maintain high levels of isolation levels within all the database across the world is imperative as you wouldn't places like banks sounding losing money between transactions because they poorly managed their database concurrency system. Therefore, by implementing ACID by companies to their database it can be ensured that the database can maintain its integrity.

References

- [1] Anon 2020 What is a Transaction (in a Database)? - Definition from Techopedia *Techopedia.com*
- [2] Anon 2020 Transaction States in DBMS - GeeksforGeeks *GeeksforGeeks*
- [3] Anon 2020 What is Database Concurrency? - Definition from Techopedia *Techopedia.com*
- [4] Anon 2019 Concurrency problems in DBMS Transactions - GeeksforGeeks *GeeksforGeeks*
- [5] Anon ACID: Concurrency Control with Transactions *MariaDB KnowledgeBase*
- [6] Engel D, Pen M, Stein S and Rabeler C 2017 Transaction Isolation Levels (ODBC) - SQL Server *Docs.microsoft.com*
- [7] Anon 2019 Transaction Isolation Levels in DBMS - GeeksforGeeks *GeeksforGeeks*
- [8] Anon 2017 Serializability - SQL Server *Docs.microsoft.com*

Report for Database System and Programming

GRADEMARK REPORT

FINAL GRADE

/100

GENERAL COMMENTS

Instructor

Programming and Software Testing mark: 77

Database Systems mark: 81

Implementation (50): 42

Programming and Software Testing Report (50): 35

Database Systems Report (50): 39

Some good work, Nathan. A good implementation, with two well-written reports. Your use of sources could use another look, however - they're largely pretty weak. I know it can be tough to track down academically credible material for these more technically oriented subjects, but it is possible.

I'll be putting on a couple of sessions on report writing during semester 2 which you might like to attend.

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16
