

Research and Reflection reports Harley-Earl Burrow

by Harley-Earl Burrow

FILE	25365_HARLEY- EARL_BURROW_RESEARCH_AND_REFLECTION_REPORTS_HARLEY- EARL_BURROW_318857_1465820297.PDF (543.16K)		
TIME SUBMITTED	15-JAN-2021 02:33PM (UTC+0000)	WORD COUNT	5425
SUBMISSION ID	142269689	CHARACTER COUNT	28456



Submission: 15/1/2021

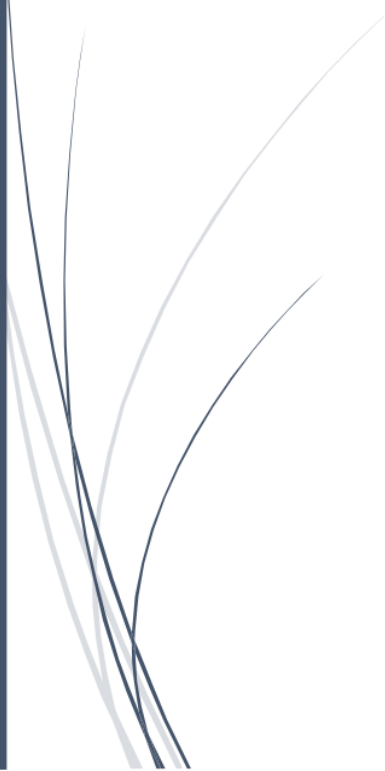
Software Testing Research and Reflection

Modules ACCA7000 & ACCA7002

[Harley-Earl Burrow](#)

STUDENT NUMBER: 2012811

Total Word Count: 4889



Programming Reflection Report

Write a reflective report which outlines your design decisions and any interesting features implemented throughout the programming task.

Upon reflection of this project, it is apparent that my design decisions were limited by my confidence in my ability to program in Java and my limited knowledge of databases. I went into this project extremely excited to have a real programming challenge in making a working program on my own, that not only has multiple functionalities, but also has real life applications. When first given the assignment, I began to draw things up in my head and attempt to "think like a programmer" in how to solve this problem.

I decided to create the database and its tables with a bit of dummy data first, so that when I began on the java program I could immediately see if these functions had worked or not. Then I ran into my first major challenge, how to enable the database and the java program to work together. I had initially created the database on Microsoft SQL server so I could look back on previous lecture notes on how to do this, however I simply could not get this working, changing driver versions, tweaking settings, everything I could find on stack overflow and YouTube, but I could not get it to work. I then tried to implement MySQL as I had previously used it within a team to create a java text-based RPG program that calls on MySQL database. However, as I was working on my personal windows PC originally, this did not go well, I could not get the server up and running let alone make my program work alongside it. I finally decided to move across to my Mac and use Postgresql, I found some information online on using a JDBC driver, I followed a guide on writing a function to check if the server is connected, and to print "Connected" if this was successful, which eventually after trying various different connection strings, I got working.

Upon looking at the core requirements, the first programming requirement (that would also require SQL queries) was that it must be possible to view and add drivers and vehicles through the program. So, I knew immediately this would require passing user input to be able to add drivers and vehicles, but that it also must be able to display data from the database in the console, which would require writing SQL READ functions within the java program. I had previously written similar functions within a team for our text-based RPG java program, so I had a rough idea how to move forward with this. I first managed to write a java function containing an SQL READ query to print everything from a table, completely unformatted with no titles or names, just random data. Then I figured out how to retrieve information by the column name and assigned them strings that I could reference and then use "System.out.print" to name them and format it nicely.

The other core criteria, view drivers whose road tax is close to expiry was an interesting one because of my database design. As I had already created my tables and changed them numerous times, I didn't want to change them again, my initial idea was to add a value to the tables with a set number of days and then write a function in the program that reads the database and displays all drivers with this value less than 30. However, as I had a large amount of dummy data in the system this would require going back and doing insert statements for each driver in the database which I tried to avoid.

Additionally, these would have to be static values, which would be unrealistic, a car cannot always be 27 days from having its tax expire. Eventually I figured I would have to write a function that takes today's date and adds 30 days to it, then search the table for expiry dates that fell in this range, which turned out working quite well and as I have tested it on separate days and the values change, it is quite future proofed which I am proud of.

Initially, the entire program was contained within one java class, as I did not have much experience writing separate classes with a main class to call on them. However, as I programmed more functions for the program, I realised this was going to be extremely difficult to test. Additionally, whilst writing the assignment about loose coupling and high cohesion and learning how much better programs with loose coupling and high cohesion are, I decided to work on breaking the program apart into separate classes. Upon reflection, I think if I would not have been writing this assignment at the same time, and researching the importance of having separate classes, the whole program may have been contained in one class. Also, I did not fully employ Test driven development during the programming process, which I now wished I had, as writing the tests at the end became an extremely arduous process. The classes for functions like adding a driver were in one large block, connecting to the server, reading from the server, requesting user input, then using WRITE statements to update the database. However, when I came to testing, I realised I would have to break these up into multiple functionalities so I could write unit tests. This added a lot of extra work for me, and if I had employed test driven development from the start, I could have written the tests as I went along, and would have had time for a lot more tests.

Word Count: 924

The importance to software development of high cohesion, loose coupling, data hiding and information hiding.

Introduction

With the ever-increasing growth of software products required in daily life, software development processes are continually in need of assessment and growth over time in order to keep up with the demand and to fulfil customers continually growing requirements and remove defects in a timely manner [1].

One of the main goals of software developers and designers has always been to produce high quality, reusable, maintainable and understandable software. There have been many challenges and features that have an influence on achieving this goal, amongst these challenges, research into the effects of cohesion, coupling, and data and information hiding has substantiated their importance. Research has shown that striving for high cohesion improves understandability, maintainability and reusability [2]. It is widely accepted that cohesion and coupling often go hand in hand and are regarded as the most fundamental features of the internal quality of software systems [3].

Coupling refers to the strength of the relationships between objects, the goal is to achieve loose coupling, so that many objects are not interdependent on each other, aiming for simple and stable relationships. One method of achieving loose coupling is through information hiding, where non-essential details of functions are inaccessible by other functions, thus reducing dependency on each other, methods and classes act like black boxes.

Increasing the use of one or more of these factors, or reducing them, has a large impact on the usability and efficiency of the code. It could mean the difference between a bad piece of code that requires constant updating, and every time it needs to be updated is extremely difficult as it keeps breaking, and a good simple, efficient piece of code that is easy to fault find, easy to update and easy to change or add new functionality. This report will aim to outline the importance of high cohesion, loose coupling, data hiding and information hiding, through various examples and current literature.

Cohesion

Cohesion is the degree to which parts of a module belong together. It is a measure of the strength of connection between the class data and methods. It is also similar to a measure of the strength of relationship between the class's method and data themselves. The higher the cohesiveness of the module, the better the design is.

High cohesion is desirable because it is associated with preferable traits such as:

- Robustness
- Reliability
- Understandability

Low cohesion is associated with undesirable traits such as:

- Being difficult to maintain
- Difficult to test
- Difficult to reuse
- Difficult to understand

High cohesion is often paired with low or loose coupling.

With high cohesion, functionalities embedded in the classes, which are accessed through the methods, have a lot in common. Also, methods will carry out a small number of related activities which helps avoid unrelated sets of data.

In low cohesion, one monolithic class may be tasked with executing many functions which may not have much in common, this will result in the code being difficult to maintain and reuse, as the whole body of code will be affected and have to be changed. Whereas in high cohesion, one class will do one small well defined job, and other jobs will be split up across other classes with much in common to each other. This can be seen in figure 1a, where the low cohesion is one large class with many jobs, and the high cohesion splits these jobs up into separate classes.

There are also special types of cohesion, such as logical, whereby parts of a module are grouped together logically as they perform similar jobs, however, are not similar in nature. Or sequential cohesion, where different parts of a module are grouped together as the output of one part is the input to another, similar to a production line. Logical cohesion is undesirable, whereas sequential cohesion is more desirable, the best type of cohesion is functional cohesion, where parts of the module are grouped together as they contribute to one well-defined task, however this is not always achievable.

Loose coupling

Coupling is the degree of interdependence between software modules; it is a measure of how closely connected two routines or modules are, like the strength of the relationships between modules.

The goal of loose coupling is achieved by a relationship where the internal implementation of a module does not matter when one module accesses another, the relationship should be simple and stable, and not require accessing internal parts of one another.

Ways to increase coupling include, one class having an attribute that refers to the other class, a class making a call on the services of another class, one class having a method that references the other class, and one class being a subclass of another class.

In coupling there are several properties to consider to achieve low or loose coupling, the degree, this refers to the number of connections between one module and others, the aim is to keep this low and only require connecting to another module through a few parameters.

The flexibility, which is how easy it is to replace modules in the future without affecting other modules. And ease, which refers to how easy it is to connect the modules together, for low coupling the aim is to make simple connections between the modules.

Tightly coupling modules is avoided as it comes with disadvantages such as, making a small change in one module would create a ripple effect and result in changes of other modules.

Additionally, tightly coupled modules might make the module harder to test and reuse.

Data Hiding

Data hiding was a term coined by David Lorge Parnas in the 1970s, it refers to each module having design decisions within it, that it hides from the rest of the system, particularly design decisions that would have a ripple effect across the code if they were to be changed. The idea is that there should be a clear distinction between the interface and the implementation of software components. In order to leave room for evolution of the software without

invalidating the interface, the interface specification should be weaker than the implementation, additionally this allows for multiple interfaces at the same time [4].

Information Hiding

Information hiding refers to the interdependency of two modules in a system, this goal is to create objects with private members in order to shield the implementation of each member from other objects, this in turn promotes loose coupling between the objects. Information hiding is achieved via a method known as encapsulation, whereby internal data representation and implementation details of an object are hidden and can only be accessed by using defined operations. One question that often arises during information hiding is which information should be hidden and which information should be exposed, Parnas suggested a heuristic approach of what to hide is to think what is likely to change. [5] Information hiding is important to programming as it helps promote loose coupling and can result in classes not being dependant on each other, which allows the classes to work without needing to change other classes.

Summary

It is widely recognised that in order to improve the reusability, maintainability and understandability of a class, high cohesion should be used [6]. Research has shown that software engineering experts agree that using designs with loose coupling and high cohesion, the result of the product is that it is more reliable and more maintainable [7]. Reliability and maintainability are of central importance to software design, no customer wants a piece of software that is not reliable and keeps breaking, they also do not want a program that cannot be updated or maintained should their preferences change, or they require new operations to be performed. It is worth noting however, that explicitly controlling coupling is more important to object oriented programming as opposed to structured programming. It is noted that controlling coupling is not as important in structured (especially top-down) design [7]. As previously identified, data hiding is important in order to achieve loose coupling, but it is also important to developers during updating or maintenance, as they are able to change internal logic of methods and variables, without changing dependent code. This is important as changing the logic of public variables used by other classes, will result in more of the code being changed, which will increase the need for new tests and potentially creating bugs. The importance of loose coupling and data hiding go hand in hand, but with loose coupling being identified as closely linked and important to high cohesion, there appears to be a central link to the importance of all these features.

In summary, in order to have maintainable, high quality, reusable code that meets all the customer's needs, and to future proof the code to make the developers life easier in the future, it is paramount that information hiding, data hiding, loose coupling and high cohesion are implemented. As it is difficult to achieve high cohesion without loose coupling, and loose coupling without data and information hiding, is it evident that they are all as important as each other to achieve the best software or code or program possible and should always be kept in mind throughout the development process.

Word Count: 1533
References

- [1] Rathee, A., & Chhabra, J. K. (2018). Improving cohesion of a software system by performing usage pattern-based clustering. *Procedia Computer Science*, 125, 740-746.
- [2] Izadkhah, H., & Hooshyar, M. (2017). Class cohesion metrics for software engineering: A critical review. *Computer Science Journal of Moldova*, 73(1), 44-74.
- [3] Izadkhah, H., & Hooshyar, M. (2017). Class cohesion metrics for software engineering: A critical review. *Computer Science Journal of Moldova*, 73(1), 44-74.
- [4] Ostermann, K., Giarrusso, P. G., Kästner, C., & Rendel, T. (2011, July). Revisiting information hiding: Reflections on classical and nonclassical modularity. In *European Conference on Object-Oriented Programming* (pp. 155-178). Springer, Berlin, Heidelberg.
- [5] Harmes, R., & Diaz, D. (2008). Encapsulation and Information Hiding. *Pro JavaScript Design Patterns*, 25-40.
- [6] Izadkhah, H., & Hooshyar, M. (2017). Class cohesion metrics for software engineering: A critical review. *Computer Science Journal of Moldova*, 73(1), 44-74.
- [7] Hitz, M., & Montazeri, B. (1995). *Measuring coupling and cohesion in object-oriented systems* (pp. 25-27).

Figure 1a: An example of Low and High cohesion

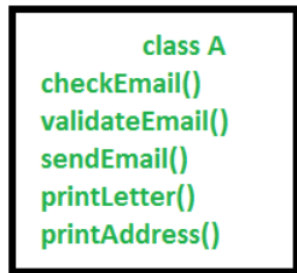


Fig: Low cohesion

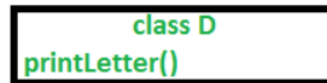
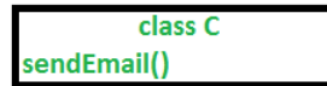
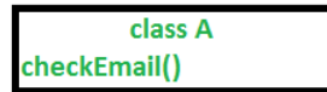


Fig: High cohesion

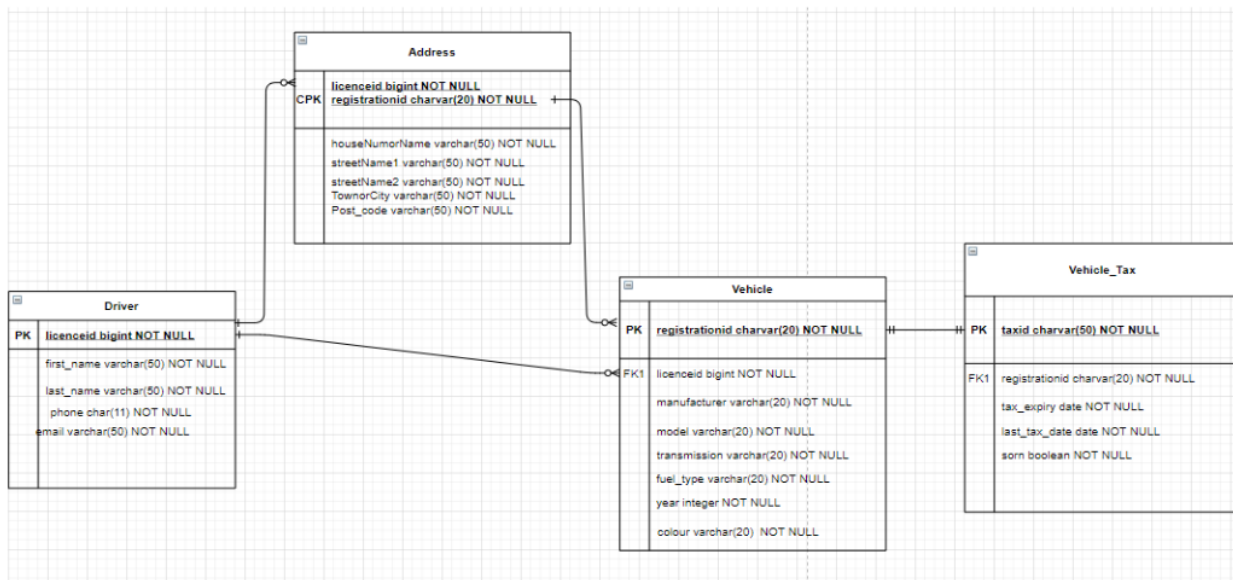
Ref: <https://ducmanhphan.github.io/2019-03-23-Coupling-and-Cohension-in-OOP/>

Database Reflection Report

Write a reflective report which outlines your database design. Also comment on the most important database transaction you feel are necessary for the monitoring systems in order to improve query performance.

It is apparent that my database design choices were limited by my confidence in my ability to create databases. When provided with the problem, my first thought was that I probably have the ability to write a program in java that takes user input and displays based on certain values, but implementing this with a database, and designing the database myself would make this much more challenging. Upon reading the brief, the very first thing I decided to do was decide what information I would need in order to normalize the database and eventually draw an ERD. A mistake I definitely made was not taking the time to actually normalize the tables, and just jumped straight into creating an ERD. My first draft of my ERD was extremely basic and only contained 3 tables, it wasn't until I got halfway through writing the SQL queries that I discovered it required a composite primary key, which I was missing. Additionally, I had addresses inside the driver table, which would have meant it would be rather difficult for multiple drivers to live at the same address.

After getting back to drawing up an ERD I came up with the design below, having a separate address table would make it much easier for multiple drivers to be at the same address through a many to many relationship. This way, many drivers can be at one address, and also a driver can have more than one address, which is more realistic to real world applications.



Halfway through designing this database I started researching and writing the report on the importance to database systems of transactions processing and access concurrency with reference to ACID. I started to apply ACID techniques to my database design, my database

transactions are atomic as they are not dependent on previous or following transactions, also the consistency only allows the data to be changed in a certain way, and isolation ensures the transactions are isolated from each other and cannot affect each other. All of the test data and transactions I have put through the database have remained despite various issues and changes, I have not met with any crashes though, so cannot be entirely sure of the durability. Also, I did not manage to implement log ins and generate different views for people with different access levels, I feel this would have been the next step I would take if this were to be applied to a real-world application, as concurrency plays an important role in database design, and there is no way I could really test this.

The most important database transaction in my database design is probably adding a new driver, as with my design it is not possible to add data to any other tables without first adding an existing driver. You cannot add an address without a driver with a driving license id number or add a vehicle without a driving license id number contained in the system and you also cannot add vehicle tax information without a vehicle which again, relies on a licence id. Though I feel this is the most important transaction, I am aware this is a flaw in my design, as in the real world you may need to add an address without having to add a driving license id.

This results in my database query performance having both strengths and weaknesses, the strength in the transaction is that it is simple, relevant and does exactly what it needs to without multiple joins. However, this could also be viewed as a weakness as it could be more specific and less ambiguous, which in a real-life situation with stakeholders and customers, would probably not put them at ease in the integrity of the system.

In the future, if I am to design a database again, I will definitely stress the importance of normalizing all the data before designing the tables, as this was something that hindered me. Additionally, this project was created using Postgresql as opposed to Microsoft SQL server which the lecture material covers, I had also tried to use MySQL at one point. This design decision was taken as I struggled to get Microsoft SQL and MySQL to work correctly and with the java program, in the future, if given more time I will not limit myself to just whatever database program I can get to work.

Additionally, I think more tables would be required for a better functional database, and some of the constraints could be better, as they still allow data that would not work in real

life, initially I had no constraints on my vehicles table, which allowed the user to input any year, which could result in some potentially strange data (such as a ford focus from the year 4000), however this was overcome by setting a data range for the year field, where cars could only be from the year range of 1940-2021. In hindsight, I will not leave adding the constraints until the end next time, as each time I added a new constraint I had to check the data integrity and rework the program around this, which cost me extra time I could have used on going for the higher mark criteria. I did however manage to make it possible to update and delete vehicles and drivers through the program and database, which I was very happy with.

Word Count: 931

Tables:

```
assignmentvehicletax=# \dt
          List of relations
 Schema |   Name   | Type | Owner
-----+-----+-----+-----
 public | address  | table | burrowh
 public | driver   | table | burrowh
 public | vehicle  | table | burrowh
 public | vehicle_tax | table | burrowh
(4 rows)
```

```
assignmentvehicletax=# \d address
          Table "public.address"
   Column   |      Type      | Collation | Nullable |      Default
-----+-----+-----+-----+-----
 housenumorname | character varying(50) |           | not null |
 streetname1   | character varying(50) |           | not null |
 streetname2   | character varying(50) |           | not null |
 townorcity    | character varying(50) |           | not null |
 post_code     | character varying(50) |           | not null |
 licenceid     | bigint          |           | not null | nextval('address_licenceid_seq'::regclass)
 registrationid | character varying(50) |           | not null |
Indexes:
    "address_pkey" PRIMARY KEY, btree (licenceid, registrationid)
Foreign-key constraints:
    "address_licenceid_fkey" FOREIGN KEY (licenceid) REFERENCES driver(licenceid)
    "address_registrationid_fkey" FOREIGN KEY (registrationid) REFERENCES vehicle(registrationid)

assignmentvehicletax=#
```

```
assignmentvehicletax=# \d vehicle_tax
```

Table "public.vehicle_tax"					
Column	Type	Collation	Nullable	Default	
tax_expiry	date		not null		
last_tax_date	date		not null		
sorn	boolean		not null		
taxid	bigint		not null	nextval('vehicle_tax_taxid_seq'::regclass)	
registrationid	character varying(50)				

Indexes:

"vehicle_tax_pkey" PRIMARY KEY, btree (taxid)

"not_null" UNIQUE CONSTRAINT, btree (registrationid)

Check constraints:

"taxid_constraint" CHECK (taxid > 0)

Foreign-key constraints:

"vehicle_tax_registrationid_fkey" FOREIGN KEY (registrationid) REFERENCES vehicle(registrationid)

```
assignmentvehicletax=#
```

```
assignmentvehicletax=# \d driver
```

Table "public.driver"					
Column	Type	Collation	Nullable	Default	
licenceid	bigint		not null	nextval('driver_licenceid_seq'::regclass)	
first_name	character varying(50)		not null		
last_name	character varying(50)		not null		
email	character varying(50)		not null		
phone	character varying(50)				

Indexes:

"driver_pkey" PRIMARY KEY, btree (licenceid)

Check constraints:

"licenceid_constraint" CHECK (licenceid > 0)

Referenced by:

TABLE "address" CONSTRAINT "address_licenceid_fkey" FOREIGN KEY (licenceid) REFERENCES driver(licenceid)

TABLE "vehicle" CONSTRAINT "vehicle_licenceid_fkey" FOREIGN KEY (licenceid) REFERENCES driver(licenceid)

```
assignmentvehicletax=#
```

```
assignmentvehicletax=# \d vehicle
```

Table "public.vehicle"					
Column	Type	Collation	Nullable	Default	
registrationid	character varying(20)		not null	nextval('vehicle_registrationid_seq'::regclass)	
manufacturer	character varying(20)		not null		
model	character varying(20)		not null		
transmission	character varying(10)		not null		
fuel_type	character varying(10)		not null		
year	integer		not null		
colour	character varying(20)		not null		
licenceid	bigint		not null	nextval('vehicle_licenceid_seq'::regclass)	

Indexes:

"vehicle_pkey" PRIMARY KEY, btree (registrationid)

Check constraints:

"year_constraint" CHECK (year > 1940)

"year_constraint2" CHECK (year < 2022)

Foreign-key constraints:

"vehicle_licenceid_fkey" FOREIGN KEY (licenceid) REFERENCES driver(licenceid)

Referenced by:

TABLE "address" CONSTRAINT "address_registrationid_fkey" FOREIGN KEY (registrationid) REFERENCES vehicle(registrationid)

TABLE "vehicle_tax" CONSTRAINT "vehicle_tax_registrationid_fkey" FOREIGN KEY (registrationid) REFERENCES vehicle(registrationid)

```
assignmentvehicletax=#
```

The importance to database systems of transaction processing (refer to ACID) with a focus on access concurrency of database management systems

Introduction

Today's web applications face the challenge of serving millions of users committing millions of transactions distributed all over the world, users rely on these services being always available, reliable and with a high degree of consistency [1].

The exponential growth of web applications, and in turn, database systems has increased the requirement for transactional processing, if applications are broadly adopted, then often, the storage of data becomes the bottleneck [2].

Therefore, it is reasonable to surmise that the way in which this data is stored and accessed through transactional processes, is of great importance. A transaction is a basic indivisible unit of work performed by the database, whether something minor like updating the name of a single user, or large like updating the prices of many items in the database due to higher demand for those items. Transaction processing refers to the real time business transactions, such as withdrawing money from an ATM, or purchasing an item online, and are performed by transaction processing systems. In order to fulfil a transaction, a transaction processing system (TPS) must access a database and fulfil and pass the ACID criteria.

As database transactions have requirements to ensure the validity of data, ACID helps enforce this through its four properties, namely, atomicity, consistency, isolation and durability. Atomicity ensures transactional database operations are irreducible and invisible, Consistency ensures that database transactions can only change data in valid ways, according to all of the defined rules. Isolation ensures all transactions are isolated from each other and cannot affect each other, one transaction cannot read data from another transaction. Durability ensures that once transactions are completed, they will remain in the system.

Large scale databases need to be able to accurately handle hundreds of thousands of transactions per minute, therefore the access concurrency of the databases play a large role and are too governed by ACID. There are different levels of access required, which different views must be generated for, for example a customer should only be able to see their own transactions and information, concurrently another user should only be able to see their won transactions and information. Additionally, data must be handled in a non-conflicting way, for example two separate customers may try to buy the last item in stock at the same time, the database must ensure only one person can purchase this item through concurrency controls and ACID.

This report will aim to investigate the importance of transaction processing in database systems, referring to ACID and access concurrency whilst drawing from research in this field.

Main

In recent years, the support for transactions in web applications has become indispensable, and it has also been suggested that the biggest mistake to be made in highly available datastores is not implementing transaction processing support systems [3].

Database transaction processes aim to protect the database and application data from integrity violations and corruption by using the ACID framework [4]. It has been

suggested that the key to many database applications is efficient online transaction processing [5].

Then need for efficient transaction processing is growing at an exponential rate as memory size and CPU core counts within a single users machine increase, high end modern servers require dozens of terabytes of main memory and hundreds of cores as most datasets are processed by a single server and stored entirely in memory [6]. Relational databases rely heavily on transaction processing, in fact, one of the reasons for the popularity of relational databases is transaction processing as it helps preserve data consistency [7]

The control and access of Database concurrency has been considered as a core element of any commercial database management system, as it allows for better utilization of underlying hardware, which in turn permits quicker user response. This is achieved by overlapping execution of transactions concurrently [8].

An issued faced with large scale databases, is two or more users trying to simultaneously access or edit the same piece of data, concurrency control helps to maintain the integrity of the system.

Concurrency would be easy to enforce in a database where users can only read data, however most functional and popular web applications require users to make READ and WRITE operations, which makes concurrency more difficult.

A system which helps prevent collisions or loss of data by multi-user access is known as a concurrency control system. This provides access to shared information in a safe way, avoiding integrity problems and loss of data. There are two main ways to achieve this, via optimistic locking and pessimistic locking, which are both types of preventative locking. Preventative locking prevents or limits a second user from accessing data whilst it is being used by another user, this is also achieved by detecting collisions, where collisions that would cause issues are refused. Optimistic locking checks if the record was updated by another user before committing the transaction, whereas pessimistic locking locks the record so that another user cannot modify it.

Other types of lock protocols include, shared locks, also known as read-only locks, which allows the data items to be shared between transactions, but data cannot be updated. Exclusive locks, where data can be read and written, but is exclusive and cannot be held concurrently by more than one user. Simplistic locks, which allow a lock on all transactions before the beginning of the operation and then data is unlocked after completing the write operation. Finally, a pre-claiming lock protocol, which evaluates which operations are required and creates a list of required items in order for the process to be executed, the transaction will be executed when all locks are granted and when the operations are complete. The locks will release. There are also many other locking mechanisms such as strict two-phase locking, centralized 2PL and distributed 2PL. Different locks also lock different parts of the database, it is possible to achieve a lock either by locking a row from a table in a database or locking an entire table, both methods have upsides and downsides, such as table locks being faster than row locks, but row locks being more flexible.

Another access concurrency protocol is a timestamp-based protocol which uses the system time to serialize and then execute operations concurrently, this ensures conflicting read and write operations will be executed in the order of their timestamp. Access concurrency of transaction processing enforced through the ACID framework is important as it helps resolve read-write and write-write conflicts. It also preserves the database consistency and helps ensure serializability. Many different errors can

be avoided with efficient access concurrency implemented, such as lost updates, incorrect summary, dirty read and non-repeatable read issues.

Summary

There are many issues faced by modern database systems and database management systems, with the ever-growing need for faster, more efficient and always available web applications relying on databases. The issues are avoided and resolved by controlling access concurrency for transaction processing using the ACID framework. ACID helps control the atomicity, consistency, isolation and durability of the system via access concurrency methods such as optimistic and pessimistic locking, shared locks and exclusive locks.

Access concurrency is important to limit concurrent users submitting the same transaction by performing similar write functions simultaneously, such as purchasing the same item when there is only one in stock.

Without stable ACID methods and access concurrency applied to database management systems, many conflicts would occur such as lost updates and conflicting data. These methods are vital to database management systems and prevent many errors from occurring, though it is recognised they are not without their weaknesses.

High availability on large scale databases is required which is not guaranteed by using ACID, this can be instead provided by supporting BASE (Basically Availability, Soft state and Eventual consistency) [3] It is also recognised that ACID is not always the ideal solution for large scale databases as they lack scalability and flexibility, and large data systems require 3 V's, volume, Velocity and Variety [1].

Additionally, using ACID comes at a significant cost to performance and availability, as enforcing ACID requires running a distributed commit protocol for transactions whilst simultaneously holding a lock on records modified during execution, however it is recognised that if instead BASE is embraced it is up to developers to explicitly code logic necessary to ensure consistency of the database [9].

Finally, as it is evident that ACID is an efficient process to use during transaction processing, it is not the only viable option and depends on the requirements of the database. A good transaction processing system can simultaneously process thousands of transactions, and as the world steps further into the realm of online shopping, this is vital for business owners and customers alike.

It is vital to database systems that transaction processing is implemented to efficiently control information processing by breaking operations up into individual, indivisible operations that can either complete or fail. As transactions cannot remain in an intermediate or incomplete state, the integrity of the database is protected with thanks to transaction processing, making it an extremely important process to database management systems.

Word Count: 1501

References

- [1] Lotfy, A. E., Saleh, A. I., El-Ghareeb, H. A., & Ali, H. A. (2016). A middle layer solution to support ACID properties for NoSQL databases. *Journal of King Saud University-Computer and Information Sciences*, 28(1), 133-145.].
- [2] Pritchett, D. (2008). Base: An acid alternative. *Queue*, 6(3), 48-55.
- [3] Zhu, Y., Liu, J., Guo, M., Ma, W., Yi, G., & Bao, Y. (2017). ACIA, not ACID: conditions, properties and challenges. *arXiv preprint arXiv:1701.07512*.].
- [4] Warszawski, T., & Bailis, P. (2017, May). Acidrain: Concurrency-related attacks on database-backed web applications. In *Proceedings of the 2017 ACM International Conference on Management of Data* (pp. 5-20).].
- [5] Lin, Q., Chen, G., & Zhang, M. (2018, April). On the design of adaptive and speculative concurrency control in distributed databases. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)* (pp. 1376-1379). IEEE.].
- [6] Tang, D., Jiang, H., & Elmore, A. J. (2017). Adaptive Concurrency Control: Despite the Looking Glass, One Concurrency Control Does Not Fit All. In *CIDR* (Vol. 2, p. 1).].
- [7] Smolinski, M. (2018, September). Impact of storage space configuration on transaction processing performance for relational database in PostgreSQL. In *International Conference: Beyond Databases, Architectures and Structures* (pp. 157-167). Springer, Cham.
- [8] Liarokapis, D., O'Neil, E., & O'Neil, P. (2020, September). Testing Concurrency in Databases still Matters. In *2020 International Conference on Information Technologies (InfoTech)* (pp. 1-6). IEEE.].
- [9] Liarokapis, D., O'Neil, E., & O'Neil, P. (2020, September). Testing Concurrency in Databases still Matters. In *2020 International Conference on Information Technologies (InfoTech)* (pp. 1-6). IEEE.].

Research and Reflection reports Harley-Earl Burrow

GRADEMARK REPORT

FINAL GRADE

/100

GENERAL COMMENTS

Instructor

Programming and Software Testing mark: 72

Database Systems mark: 70

Implementation (50): 32

Programming and Software Testing Report (50): 40

Database Systems Report (50): 38

A fair effort made - the implementation is of a reasonably good quality, if a little hard coded in places.

The reports are fairly well written, with an appropriate level of detail. Good work overall.

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
