Global Scientific JOURNALS

GSJ: Volume 10, Issue 2, February 2022, Online: ISSN 2320-9186
www.globalscientificjournal.com

REVIEW OF CONCURRENCY ISSUES USING PESSIMISTIC CONTROL TECHNIQUE

ONUNGWE, H. O[1] ., EGBONO F[2]., and EKE, O. B[3].

Department of Computer Science

Faculty of Science

University of Port Harcourt

ABSTRACT

Concurrency is a situation where more than one processes are running at the same time. In such scenarios, there are tendencies that there will be conflict in reading and writing of data items Concurrency is associated with four major problems (P1 – P4): Unrepeated Read (P1), Inconsistent Analysis (P2), Lost Update (P3) and Phantom Read (P4). A Concurrency Control techniques need to be put in place to ensure that data items are serialize in a structured pattern to avoid an adverse compromise on data integrity and consistency. Basically, there are two major techniques of controlling Concurrency on Databases; Pessimistic and Optimistic Control techniques. This paper is aimed at handling the Pessimistic technique. Pessimistic Control Technique has two inherent problems; Frequent Lockouts and Deadlock Detection. This paper is limited to the Deadlock Detection aspect of the problem. We explore the various methods of handling Deadlock issues in Pessimistic Concurrency Control Technique, we paid particular attention to Time stamping as one of the methods of handling Deadlock issues when dealing with Unrepeated Read problem (P1) of Concurrency.

KEYWORDS: Concurrency, Pessimistic Control, Deadlock, Timestamping, Unrepeated Read.

1.0 INTRODUCTION

Multiple processes need to be serialize in a manner that there should not be conflict issues of data integrity. Concurrency is a property of a system representing the fact that multiple activities are executed at the same time. Concurrency and Parallelism are used interchangeably most times. Concurrency is a conceptual property of a program, while parallelism is a runtime state. Concurrency of a program depends on the programming language and the way it is coded, while parallelism depends on the actual runtime environment. Given two tasks to be executed concurrently, there are several possible execution orders. They may be performed sequentially (in any order), alternately, or even simultaneously. Only executing two different tasks simultaneously yields true parallelism. In terms of scheduling, parallelism can only be achieved if the hardware architecture supports parallel execution, like multi-core or multi-processor systems do. A single-core machine will also be able to execute multiple threads concurrently, however it can never provide true parallelism. According to Van Roy (2004), a program having "several independent activities, each of which executes at its own pace". In addition, the activities may perform some kind of interaction among them. The concurrent execution of activities can take place in different environments, such as single-core processors, multi-core processors, multi-processors or even on multiple machines as part of a distributed system. Yet, they all share the same underlying challenges: providing mechanisms to control the different flows of execution via coordination and synchronization, while ensuring consistency. Logically, Concurrency Control is an illusion that processes, transactions etc are running the same time without any form of conflict. Concurrency is a pertinent tool in modern programming, especially with the rise of multi-core architectures and the increasing prevalence of distributed systems. And like any other tool, it is important to understand how and when to use it to have efficiency. Morgan (2013) asserts that in common thread, concurrency, applied correctly, can improve the performance of a program but the correct application may not be readily apparent. He opined that Concurrency is conceptually difficult, and requires a different approach than sequential programming. The purpose of his case study is to investigate the advantages to executing multiple tasks in parallel. To this end, the concurrency in his case study was introduced at a higher level of the program than in previous case studies. Instead of using concurrency to calculate the image filters over multiple pixels within an image simultaneously, concurrency was used to apply the sequence of image filters to multiple images at once. One limitation of his work is that he did not explore the effects of varying the number of available processors. He presented results for the non concurrent (i. e: single core) case, as well as multiple concurrent implementations that execute on four-cores. Extending the case studies to utilize larger numbers of cores would be an interesting avenue of investigation, though would require additional hardware.

Problems of Concurrency:

Some problems may occur in multi-user environment when concurrent access to database is allowed. These problems may cause data stored in the multi-user DBMS to be damaged or destroyed. Transactions running concurrently may interfere with each other, causing various problems. The problems associated with concurrency are clearly depicted in Figure 1.1.

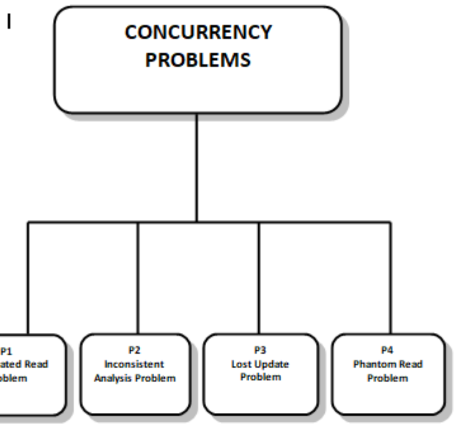

Figure 1.1: **Concurrency Problems in Transaction**

2.0 LITERATURE REVIEW

According to Gohil et al. (2013), Pessimistic control mechanism prevents the execution of concurrent transaction if any conflict is detected between the concurrent transaction in a distributed database system. This control mechanism blocks an operation of a transaction, if it may cause violation of the rules, until the possibility of violation disappears. Blocking operations is typically involved with performance reduction. It acquires locks when transactions start so conflicts are not possible. It is useful if there are a lot of updates and relatively high chances of users trying to update data at the same time. In pessimistic locking method data to be updated is locked in advance. Once the data to be updated has been locked, the application can make the required changes, and then commit or rollback during which the lock is automatically dropped. If anyone else attempts to acquire a lock of the same data during this process, they will be forced to wait until the first transaction has completed . This approach is called pessimistic because it assumes that another transaction might change the data between the read and the update. In order to prevent that change from happening and the data inconsistency that would result the read statement locks the data to prevent any other transaction from changing.

Dag Nystrom et.al (2004) in their paper presented a Concurrency Control Algorithm that allows co-existence of soft real-time, relational database transactions, and hard real-time database pointer transactions in real-time database management systems. Their algorithm used traditional Pessimistic Concurrency Control (i.e. locking) for soft transactions and versioning for hard transactions to allow them to execute regardless of any database lock. They provided a formal proof that the Algorithm is deadlock free and formally verify that transactions have atomic semantics. They also presented an evaluation that demonstrates significant benefits for both soft and hard transactions when their algorithm is used. Their proposed algorithm is suited for resource-constrained safety critical, real-time systems that have a mix of hard real-time control applications and soft real-time management, maintenance, or user-interface applications

According to Gohil et al. (2013), Pessimistic control mechanism prevents the execution of concurrent transaction if any conflict is detected between the concurrent transaction in a distributed database system. This control mechanism blocks an operation of a transaction, if it may cause violation of the rules, until the possibility of violation disappears. Blocking operations is typically involved with performance reduction. It acquires locks when transactions start so conflicts are not possible. It is useful if there are a lot of updates and relatively high chances of users trying to update data at the same time. In pessimistic locking method data to be updated is locked in advance. Once the data to be updated has been locked, the application can make the required changes, and then commit or rollback during which the lock is automatically dropped. If anyone else attempts to acquire a lock of the same data during this process, they will be forced to wait until the first transaction has completed . This approach is called pessimistic because it assumes that another transaction might change the data between the read and the update. In order to prevent that change from happening and the data inconsistency that would result the read statement locks the data to prevent any other transaction from changing.

The various techniques of Pessimistic Control and the problems associated with the various techniques are as shown in Figure 1.2 and 1.3 respectively.

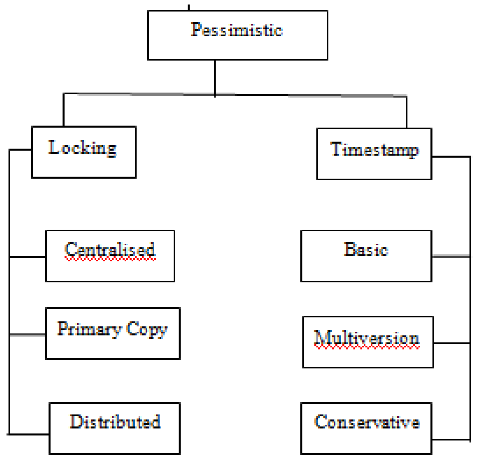

Figure 1.2 Pessimistic Control Techniques

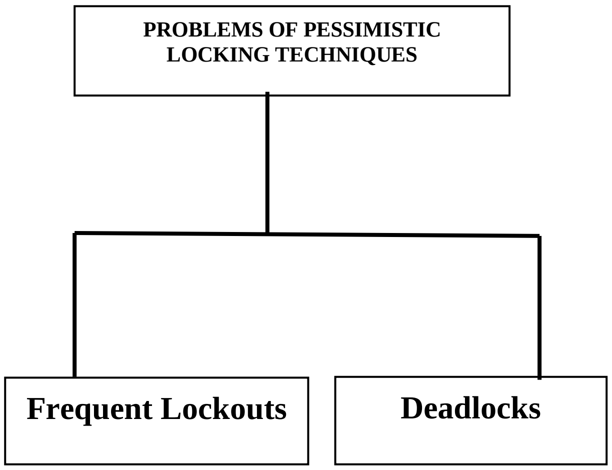

Figure 1.3 Problems of Pessimistic Locking Techniques

A pessimistic locking technique suffers from two major problems namely frequent lockouts and deadlocks. The problem of frequent lockout arises when a transaction invoked by a user selects a record for update, and executing the operations without finishing or aborting the transaction. All other users with their respective transaction that need to update that record are forced to wait until the user completes its transaction. The problem of deadlock arises when Transactions T1 and T2 are both updating the database at the same time. Now suppose transaction T1 locks a record and then attempt to acquire a lock held by transaction T2 who is waiting to obtain a lock held by transaction T1. In this situation both transactions goes in an infinite wait state which is called deadlock. The main thrust of this paper is to handle Deadlock issues in Concurrency.

Handling Deadlock issues in Pessimistic Locking Techniques:

Prerana Jain (2018) opined that in the multiprogramming operating system, there are a number of processing which fights for a finite number of resources and sometimes waiting process never gets a chance to change its state because the resources for which it is waiting are held by another waiting process. He said that a set of a process is called **Deadlock** when they are waiting for the happening of an event which is called by some another event in the same set. There are different approaches in handling Deadlock issues in Pessimistic Locking Techniques of Concurrency

Control. We are more concerned with the Time Stamping Approach.

Timestamp-based concurrency control algorithms use a transaction’s timestamp to coordinate concurrent access to a data item to ensure serializability. A timestamp is a unique identifier given by DBMS to a transaction that represents the transaction’s start time. Some of timestamp based concurrency control algorithms are: Basic timestamp ordering algorithm, Conservative timestamp ordering algorithm and Multiversion algorithm based upon timestamp ordering, these various algorithms are depicted in Figure 1.4. Timestamp of any transaction is determined by the physical clock reading. But, in a distributed system, any site’s local physical/logical clock readings cannot be used as global timestamps, since they are not globally unique. So, a timestamp comprises of a combination of site ID and that site’s clock reading, (Makundi et. al. 2017).

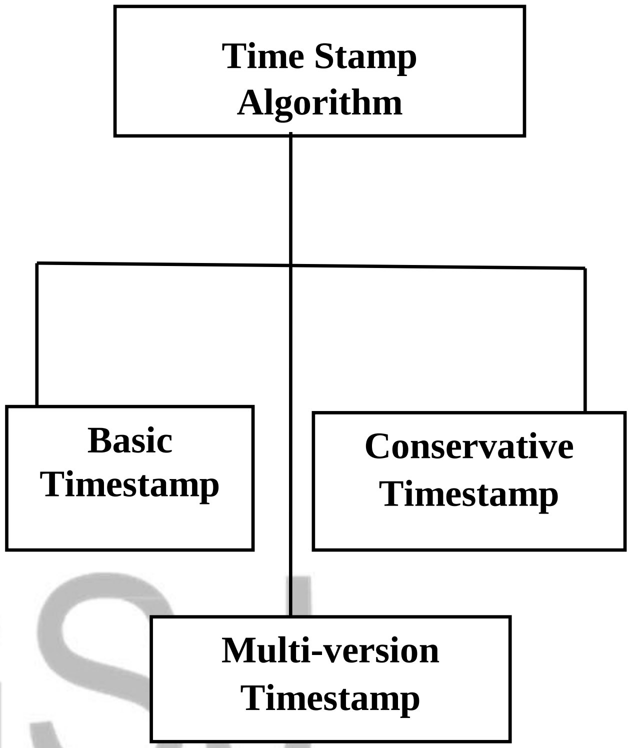

Figure 1.3 Types of Time Stamp Algorithms

Using Time Stamping to solve Unrepeated Read Problem (P1) of Concurrency

Unrepeated Read problem (P1) occurs when a transaction gets to read unrepeatedly i.e. different values of the same variable in its different read operations even when it has not updated its value. In certain simple cases it is possible to ensure that an operation can be repeated without causing any errors or inconsistencies. When deciding how certain operations should be specified, such as the form their arguments should take, it may be possible to choose an idempotent implementation. For example, the repeatable method of writing a sequence of bytes to the end of a file is to specify the precise byte position at which the new bytes should be written and make this an argument of an 'append' operation. If an append operation is relative to a system-maintained pointer, the bytes could be written more than once if the operation was repeated. It is not always possible to achieve repeatable operations. This point becomes more important in distributed systems, when a congested network may cause a reply message saying 'operation done' to be greatly delayed or lost. The invoker does not know this and repeats the request when the operation requested has already been carried out. *Unrepeatable Read* occurs when T1 transaction wants to read Q data again, but Q data just for once after reading is altered by T2 transaction, therefore T1 transaction cannot read the same amount of Q data again and it is a critical problem.

Manuel (2000), worked on solving the Phantom Problem by Predicative Optimistic Concurrency Control. His Predicative Optimistic Concurrency Control is used to attack problems inherent in Predicate Locking for detecting conflicts that actually occurred between transactions. He divided each transaction into three phases; Read, Validation and Write phase. For each transaction, two sets are maintained; a Read and Write set. He compared the Predicative Concurrency with other control method; the Locking method. Our work shall attempt to solve the Unrepeated Read (P1) problem using the Hybrid Concurrency Control approach unlike that of Manuel (2000) that used only the Optimistic approach.

Related Work:

Timestamping after commit of time-varying data in local and distributed environments was studied by Salzberg (2010). In his paper, he extends and refines that study in several respects. While Salzberg is concerned with timestamping the transaction-time dimension, this paper also considers time stamping in transaction time of Inconsistent Analysis problem of Concurrency. In Salzberg's study, timeslice queries are considered; this paper proceeds to consider general queries in Inconsistent Analysis. Finally, Salzberg assumes an integrated DBMS architecture, which may be extended to incorporate a new recovery algorithm and as well as multidimensional temporal indexes; in contrast, this paper describes how timestamping implementation techniques after commit may be achieved, without necessitating any changes to the underlying DBMS.

Finger et.el (2011) studied timestamping, including the use of the validtime variable. They took into consideration that the actual execution of a transaction has a duration in time, and they argue that the value should remain constant within a transaction. However, they rule out using the commit time for timestamping the valid-time dimension and instead suggest using the start time or the time of the first update. They showed that using the start time can lead to appearing to be moving backwards in time and in the case of using the time of the first update that the serialization of transactions can be violated. They suggested ignoring the problem of time moving backwards or making transactions serializable on their start-times. This paper takes the opposite approach, ruling out using any value for now other than the commit time. We show first that the problem of moving backwards cannot be ignored because it may also violate the isolation principle. Second, we argue that transaction executions cannot be serializable in the order of their start times, if concurrency is allowed. Finally, we show that using the commit time, can solve the two problems identified by Finger and McBrien. eight extra attributes to each table.

3.0 MATERIALS AND METHODS

In this section, we investigate how to prevent global deadlocks using Time Stamp approach, which happens more often than local deadlocks. Existing technologies for global deadlock prevention are generally based on sequential resource access. It is a pessimistic static resource allocation scheme that needs to exploit prior knowledge of transaction access patterns.

Pre-check based approach for preventing global deadlocks In service-oriented environments, each business transaction knows what resources it will request. So, it is appropriate to make sure whether resources needed by a transaction are available or not before starting the transaction.

In the Pre-Check stage, the coordinator delivers all the sub-transactions to participants, and then these participants communicate with resource managers to check the state of resources. If these resources are available, the participant will hold them and at the same time return OK to the coordinator. Otherwise, it will return a failed message. After receiving OK messages from all participants, the coordinator will start the standard two-phase commit. Our pre-check phase includes the following three steps.

Step 1. Transaction delivery. After receiving a transaction request, transaction manager (TM) produces a unique root transaction ID, which can be a function of current time to distinguish starting time of transactions. Next, TM divides the task into sub-transactions and distributes them to different sites which host specified services.

Step 2. Resource pre-check. Resource allocation and local deadlock prevention.. The basic idea is that each global transaction has to check and then hold all the necessary resources if they are available at the beginning of the transaction execution.

On receiving the pre-check instruction, each participant begins to check all the needed resources through their resource managers.

We used Time Stamp to lock the entire table by ensuring that each transaction receives a Unique TimeStamp (UTS). When the system is locked, a unique counter is incremented using a Scheduler.

4.0 CONCLUSION

Haven reviewed several literatures on various approaches of resolving Concurrency issues, we observed that Timestamp approached is the best approach of resolving the Unrepeated Read problem (P1) of Concurrency issues. Our Time Stamp approach used, was able to resolve conflict in serialization and recoverability. One can also explore using Timestamp to resolve the Inconsistent Analysis problem (P2) of Concurrency issues.

REFERENCES

Chase, D., and Lev, Y. (2005) Dynamic circular work-stealing deque. In Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures (New York, NY, USA, 2005), SPAA ’05, ACM, pp. 21–28

Dag Nystrom, Mikael Nolin, Aleksandra Tesanovic, Christer Norstrom and Jorgen Hansson (2004) Pessimistic Concurrency Control and Versioning to support Database Pointers in Real-time Databases. Proceedings of the

12th 16th Euromicro Conference on Real-Time Systems (ECRTS'04) 1068-3070/04 $20.00 © 2004 IEEE

Felber, P., KORLAND, G., AND SHAVIT, N. Deuce (2010): Noninvasive concurrency with a Java STM. In Electronic Proceedings of the workshop on Programmability Issues for Multi-Core Computers (MULTIPROG) (2010), p. 10 pages.

Karmani, R. K., SHALI, A., AND AGHA, G. (2009). Actor frameworks for the JVM platform: a comparative analysis. In Proceedings of the 7th International Conference on Principles and Practice of Programming in Java (New York, NY, USA, 2009), PPPJ '09, ACM, pp. 11–20.

M. Finger and P. McBrien. (2011). The Semantics of 'Current-Time' in Temporal Databases. In 11th Brazilian Symposium on Databases, pp. 324–337,

Morgan Atkins (2013). Modern Concurrency Technique: An Exploration

Prerana Jain (2018). Deadlock and Method for Handling Dealock

Salzberg B. (2010) Timestamping After Commit. In Proceedings of the Conference on Parallel and Distributed Information Systems, pp. 160–167.

Sonal Kanungo, Patel Z.S and Ruston D.M. (2016). Comparison of Concurrency Control and Deadlock Handling in Different OODBMS. *An International Journal of Engineering Research and Technology (IJERT) ISSN:2278-0181 Vol 5, Issue 05, May 2016*

Stonebraker M. The Design of the Postgres Storage System. In Proceedings of VLDB Conference, pp. 289–300, 1987.

Stonebraker M., L. A. Rowe, and M. Hirohama. The Implementation of Postgres. IEEE Transaction on Knowledge and Data Engineering, 2(1):125–142, March 1990