CHICAGO STATE UNIVERSITY

**MATHEMATICS AND COMPUTER SCIENCE DEPARTMENT**

CPTR 5600-01

**ADVANCED DATA BASE DESIGN AND IMPLEMENTATION**

Topic –TRANSACTION MANAGEMENT AND CONCURRENCY CONTROL

**Submitted By:**

DRONAMRAJU VENKATA SESHA SURYA SAI MANI PREETHI

900726470

Instructor:

DR. LUIS VIDAL

Fall-2016

ABSTARCT

Later now, Many real-world applications taking time consuming to access data from database, For example, consider a navigation systems, control systems, command systems, stock trading, networking management and telephonic switching system. These applications needed gathering data from sources, processing information in the context of information obtained in the past, and contributing response in timely manner. Hence, these applications need a database system where transactions are associated with deadlines on their completion times.

In order to exploit parallel transactions, database management systems must achieve a high level of concurrency when executing transactions. Many real-time concurrency control methods are based on two-phase locking (2PL). For every success full transaction, it must maintain Atomicity, Consistency, Isolation, and Durability commonly known as ACID properties.

In this paper, we focus on transaction management and how transaction management maintains ACID properties in concurrency controlling system. We also discuss transaction management for web based applications, and cover topics such as spring transaction management. We then discuss current challenges for database parallel transactions and some preliminary approaches that address some of these challenges.

TABLES OF CONTENTS

1. Introduction

1.1 Introduction to Transaction Management

1.2 Concurrency Control in Database

2. Database Transaction Management

2.1 Transaction properties (ACID)

2.2 Isolation of Transactions

2.3 Transaction states

2.4 Rollback/Aborting a Transaction

3. Concurrency Control

3.1 Concurrency Control mechanisms

3.2 Concurrency Control Mechanisms and Protocols

3.2.1 Lock-Based Concurrency Control Protocols

3.2.2 Strict Two Phase locking (2PL) protocol

3.3 Concurrency control in Real-time web applications

4. Deadlock and How to prevent deadlocks

5. Challenges in Transaction Management and Concurrency Control

6. Conclusion

7. References

INTRODUCTION

As organizations increase their adoption of database systems as the key data management technology for day-to-day operations and decision making, processing of transactions in parallel improves the application performance and productivity. In Advanced data base, there are following two primary processes.

1. Transaction management (TM) handles all transactions properly in DBMS. Database transactions are the events or activities such as series of data read/write operations on data object(s) stored in database system
2. Concurrency control (CC) is a process to ensure that data is updated correctly and appropriately when multiple transactions are concurrently executed in DBMS.

In general, concurrency control is an essential part of Transaction Management. It is a mechanism for correctness when two or more database transactions that access the same data or data set are executed concurrently with time overlap. If multiple transactions are executed serially or sequentially, data is consistent in a database. However, if concurrent transactions with interleaving operations are executed, some unexpected data and inconsistent result may occur. Data interference is usually caused by a write operation among transactions on the same set of data in DBMS. For example, the lost update problem may occur when a second transaction writes a second value of a data content on top of the first value written by a first concurrent transaction.

Concurrency control is one of the primary mechanisms in transaction management to provide integrity of data and safety in DBMS. Today, with hundred thousand or more transactions in a few minutes, transaction management and concurrency control become much more complex and sophisticated. Two pessimistic and optimistic mechanisms are still popular, but other techniques such as semi-optimistic are also applied in DBMS for higher performance, better throughput, more accurate results, and faster run time.

**Database Transaction Management**

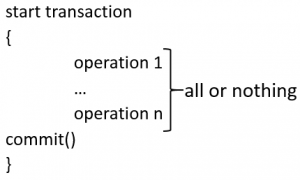
**2.1 Transaction properties (ACID)**

A transaction is a very small unit of a task. It may contain several lower level sub tasks. In order to data accuracy, integrity and completeness, a transaction in database system must maintain Atomicity, Consistency, Durability and Isolation.

**Atomicity**: The database transaction should be atomic, that is, either all operations should execute or none. Any transaction should be left with partially completed. States should be defined either before execution of the transaction or after the execution of the transaction.

Typically, systems implement Atomicity by providing some mechanism to indicate which transactions have started and which finished; or by keeping a copy of the data before any changes occurred. Several file systems have developed methods for avoiding the need to keep multiple copies of data, using journaling. Databases usually implement this using some form of logging/journaling to track changes. The system synchronizes the logs as necessary once the actual changes have successfully taken place. Afterwards, crash recovery simply ignores incomplete entries. Although implementations vary depending on factors such as concurrency issues, the principle of atomicity, that is, complete success or complete failure remain.

An example of an atomic transaction is a monetary transfer from bank account A to account B. It consists of two operations, withdrawing the money from account A and saving it to account B. Performing these operations in an atomic transaction ensures that the database remains in a consistent state, that is, money is not lost nor created if either of those two operations fail.



If for any reason an error occurs and the transaction is unable to complete all of its steps, then the system is returned to the state it was in before the transaction was started.

**Consistency:** The database must remain in a consistent state after any transaction. No transaction should have any adverse effect on the data residing in the database. If the database was in a consistent state before the execution of a transaction, it must remain consistent after the execution of the transaction as well.

Consistency is one of the four guarantees that define ACID transactions. However, significant ambiguity exists about the nature of this guarantee. It is defined variously as:

* The guarantee that any transactions started in the future necessarily see the effects of other transactions committed in the past
* The guarantee that database constraints are not violated, particularly once a transaction commits
* The guarantee that operations in transactions are performed accurately, correctly, and with validity, with respect to application semantics

Consistency is a very general term, which demands that the data must meet all validation rules. In the previous example, the validation is a requirement that A + B = 100. Also, it may be inferred that both A and B must be integers. A valid range for A and B may also be inferred. All validation rules must be checked to ensure consistency. Assume that a transaction attempts to subtract 10 from A without altering B. Because consistency is checked after each transaction, it is known that A + B = 100 before the transaction begins. If the transaction removes 10 from A successfully, atomicity will be achieved. However, a validation check will show that A + B = 90, which is inconsistent with the rules of the database. The entire transaction must be cancelled and the affected rows rolled back to their pre-transaction state. If there had been other constraints, triggers, or cascades, every single change operation would have been checked in the same way as above before the transaction was committed.

**Durability**: The database should be durable enough to hold all its latest updates even if the system fails or restarts. If a transaction updates a chunk of data in a database and commits, then the database will hold the modified data. If a transaction commits but the system fails before the data could be written on to the disk, then that data will be updated once the system springs back into action.

The durability property ensures that once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors. In a relational database, for instance, once a group of SQL statements execute, the results need to be stored permanently (even if the database crashes immediately thereafter). To defend against power loss, transactions must be recorded in a non-volatile memory.

Consider a transaction that transfers 10 from A to B. First it removes 10 from A. then it adds 10 to B. At this point, the user is told the transaction was a success. however the changes are still queued in the disk buffer waiting to be committed to disk. Power fails and the changes are lost. The user assumes that the changes have been persisted.

**Isolation:** In a database system where more than one transaction are being executed simultaneously and in parallel, the property of isolation states that all the transactions will be carried out and executed as if it is the only transaction in the system. No transaction will affect the existence of any other transaction.

The isolation property ensures that the concurrent execution of transactions results in a system state that would be obtained if transactions were executed serially, that is, one after the other. Providing isolation is the main goal of concurrency control. Depending on the concurrency control method, the effects of an incomplete transaction might not even be visible to another transaction.

To demonstrate isolation, we assume two transactions execute at the same time, each attempting to modify the same data. One of the two must wait until the other completes in order to maintain isolation. Consider two transactions. T1 transfers 10 from A to B. T2 transfers 10 from B to A. Combined, there are four actions:

T1 subtracts 10 from A.

T1 adds 10 to B.

T2 subtracts 10 from B.

T2 adds 10 to A.

If these operations are performed in order, isolation is maintained, although T2 must wait. Consider what happens if T1 fails halfway through. The database eliminates T1's effects, and T2 sees only valid data.

By interleaving the transactions, the actual order of actions might be:

T1 subtracts 10 from A.

T2 subtracts 10 from B.

T2 adds 10 to A.

T1 adds 10 to B.

Again, consider what happens if T1 fails halfway through. By the time T1 fails, T2 has already modified A; it cannot be restored to the value it had before T1 without leaving an invalid database. This is known as a write-write failure,[citation needed] because two transactions attempted to write to the same data field. In a typical system, the problem would be resolved by reverting to the last known good state, canceling the failed transaction T1, and restarting the interrupted transaction T2 from the good state.

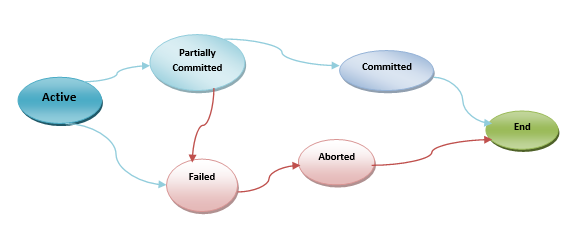
**2.2 Transaction states:** A transaction in a database can be one of the following 5 stages.

* **Active**
* **Partially committed**
* **Failed**
* **Aborted**
* **Committed**

**Active:** This is the first state of transaction and here the transaction is being executed. For example, updating or inserting or deleting a record is done here. But it is still not saved to the database. When we say transaction it will have set of small steps, and those steps will be executed here.

**Partially Committed:** This is also an execution phase where last step in the transaction is executed. But data is still not saved to the database. In our example of calculating total marks, final display the total marks step is executed in this state.

**Committed**: In this state, all the transactions are permanently saved to the database. This step is the last step of a transaction, if it executes without fail.



**Failed**: If a transaction cannot proceed to the execution state because of the failure of the system or database, then the transaction is said to be in failed state. In the total mark calculation example, if the database is not able fire a query to fetch the marks, i.e.; very first step of transaction, then the transaction will fail to execute.

**Aborted**: If a transaction is failed to execute, then the database recovery system will make sure that the database is in its previous consistent state. If not, it brings the database to consistent state by aborting or rolling back the transaction. If the transaction fails in the middle of the transaction, all the executed transactions are rolled back to it consistent state before executing the transaction. Once the transaction is aborted it is either restarted to execute again or fully killed by the DBMS.

**References**

Connolly, T. M., & Begg, C. E. (2014). Database Systems: A Practical Approach to Design, Implementation and Management. New Jersey, NJ: Pearson.

Larson, P. Å., Blanas, S., Diaconu, C., Freedman, C., Patel, J. M., & Zwilling, M. (2011). High-performance concurrency control mechanisms for main-memory databases. Proceedings of the VLDB Endowment, 5(4), 298-309.

Kung, H. T., & Robinson, J. (1981). Carnegie-Mellon University, Retrieved from http://www.csd.uoc.gr/~hy460/pdf/kung.pdf.

Vallejo, E., Sanyal, S., Harris, T., Vallejo, F., Beivide, R., Unsal, O., & ... Valero, M. (2011). Hybrid Transactional Memory with Pessimistic Concurrency Control. International Journal of Parallel Programming, 39(3), 375-396. doi:10.1007/s10766-010-0158-x

Wikipedia (2014). Optimistic Concurrency Control. Retrieved from http://en.wikipedia.org/wiki/Optimistic\_concurrency\_control.