Transaction Management | Meerza Ahmed

Tuesday, September 26, 2023 1:00 PM

Lecture 6: Link | Syllabus

*Class Week 6/15

Database Systems by Coronel & Morris

Chapter 10: Transaction Management and Concurrency Control

* 10.1 *What is a Transaction ?*

To understand the concept of a transaction, suppose that you sell a product to a customer. Furthermore, suppose that the customer may charge the purchase to his or her account.

Given that scenario, your sales transaction consists of at least the following parts :

- You must write a new customer invoice
- You must reduce the quantity on hand in the product's inventory.
- You must update the account transactions.
- You must update the customer balance

The preceding sales transaction must be reflected in the database. In database terms, a transaction is any action that reads from or writes to a database. A transaction may consist of the following :

- A simple SELECT statement to generate a list of table contents.
- A series of related UPDATE statements to change the values of attributes in various tables.
- A series of INSERT statements to add rows to one or more tables.
- A combination of SELECT, UPDATE, and INSERT statements.

* Given the preceding discussion, you can augment the definition of a transaction. A transaction is a logical unit of work that must be entirely completed or entirely aborted; no intermediate states are acceptable. In other words, a multicomponent transaction, such as the previously mentioned sale, must not be partially completed. Updating only the inventory or only the accounts receivable is not acceptable.

* A successful transaction changes the database from one consistent state to another. A consistent database state is one in which all data integrity constraints are satisfied. To ensure consistency of the database, every transaction must begin with the database in a known consistent state.

* Most real-world database transactions are formed by two or more database requests. A database request is the equivalent of a single SQL statement in an application program or transaction

* 10.1-b *Transaction Properties*

Each individual transaction must display atomicity, consistency, isolation, and durability. These four properties are sometimes referred to as the ACID test

- *Atomicity* requires that all operations (SQL requests) of a transaction be completed; if not, the transaction is aborted. If a transaction T1 has four SQL requests, all four requests must be successfully completed; otherwise, the entire transaction is aborted. In other words, a transaction is treated as a single, indivisible, logical unit of work.
- *Consistency* indicates the permanence of the database's consistent state. A transaction takes a database from one consistent state to another. When a transaction is completed, the database must be in a consistent state. If any of the transaction parts violates an integrity constraint, the entire transaction is aborted.
- *Isolation* means that the data used during the execution of a transaction cannot be used by a second transaction until the first one is completed. In other words, if trans action T1 is being executed and is using the data item X, that data item cannot be accessed by any other transaction (T2 … Tn) until T1 ends. This property is particularly useful in multiuser database environments because several users can access and update the database at the same time.
- *Durability* ensures that once transaction changes are done and committed, they cannot be undone or lost, even in the event of a system failure

* In addition to the individual transaction properties indicated above, there is another important property that applies when executing multiple transactions concurrently. each individual transaction must comply with the ACID properties and, at the same time, the schedule of such multiple transaction operations must exhibit the property of serializability. Serializability ensures that the schedule for the concurrent execution of the transactions yields consistent results.

* 10.1-c *Transaction Management with SQL*

The American National Standards Institute (ANSI) has defined standards that govern SQL database transactions. The ANSI standards require that when a transaction sequence is initiated by a user or an application program, the sequence must continue through all succeeding SQL statements until one of the following four events occurs :

- A COMMIT statement is reached, in which case all changes are permanently recorded within the database. The COMMIT statement automatically ends the SQL transaction.
- A ROLLBACK statement is reached, in which case all changes are aborted and the database is rolled back to its previous consistent state.
- The end of a program is successfully reached, in which case all changes are permanently recorded within the database. This action is equivalent to COMMIT.
- The program is abnormally terminated, in which case the database changes are aborted and the database is rolled back to its previous consistent state. This action is equivalent to ROLLBACK

* Not all SQL implementations follow the ANSI standard; some (such as SQL Server) use transaction management statements such as the following to indicate the beginning of a new transaction : ---> BEGIN TRANSACTION;

*** 10.1-d** *The Transaction Log*

A DBMS uses a transaction log to keep track of all transactions that update the data base. The DBMS uses the information stored in this log for a recovery requirement triggered by a ROLLBACK statement, a program's abnormal termination, or a system failure such as a network discrepancy or a disk crash.

The transaction log stores the following :

- A record for the beginning of the transaction.
- For each transaction component (SQL statement) :
  - The type of operation being performed (INSERT, UPDATE, DELETE).
  - The names of the objects affected by the transaction (the name of the table).
  - The "before" and "after" values for the fields being updated.
  - Pointers to the previous and next transaction log entries for the same transaction.
- The ending (COMMIT) of the transaction

*** 10.2** *Concurrency Control*

Coordinating the simultaneous execution of transactions in a multiuser database system is known as concurrency control. The objective of concurrency control is to ensure the serializability of transactions in a multiuser database environment. To achieve this goal, most concurrency control techniques are oriented toward preserving the isolation property of concurrently executing transactions. Concurrency control is important because the simultaneous execution of transactions over a shared database can create several data integrity and consistency problems. The three main problems are lost updates, uncommitted data, and inconsistent retrievals

*** 10.2-a** *Lost Updates*

The lost update problem occurs when two concurrent transactions, T1 and T2, are updating the same data element and one of the updates is lost (overwritten by the other transaction).

*** 10.2-b** *Uncommitted Data*

The phenomenon of uncommitted data occurs when two transactions, T1 and T2, are executed concurrently and the first transaction (T1) is rolled back after the second transaction (T2) has already accessed the uncommitted data—thus violating the isolation property of transactions.

*** 10.2-c** *Inconsistent Retrievals*

Inconsistent retrievals occur when a transaction accesses data before and after one or more other transactions finish working with such data. For example, an inconsistent retrieval would occur if transaction T1 calculated some summary (aggregate) function over a set of data while another transaction (T2) was updating the same data.

*** 10.2-c** *The Scheduler*

The scheduler is a special DBMS process that establishes the order in which the operations are executed within concurrent transactions. The scheduler interleaves the execution of database operations to ensure serializability and isolation of transactions. To determine the appropriate order, the scheduler bases its actions on concurrency control algorithms. However not all transactions are serializable. Generally, transactions that are not serializable are executed on a first-come, first-served basis by the DBMS. The scheduler's main job is to create a serializable schedule of a transaction's operations, in which the interleaved execution yields the same results as if the transactions were in serial order

*** 10.3** *Concurrency Control with Locking Methods*

Locking methods are one of the most common techniques used in concurrency control because they facilitate the isolation of data items used in concurrently executing transactions. A lock guarantees exclusive use of a data item to a current transaction. In other words, transaction T2 does not have access to a data item that is currently being used by transaction T1.

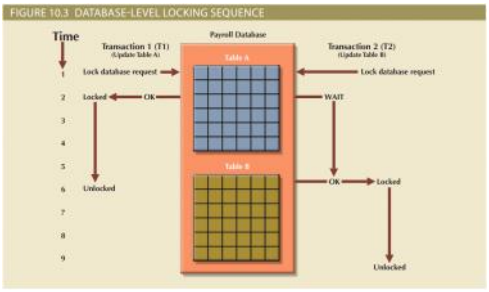
* The use of locks based on the assumption that conflict between transactions is likely is usually referred to as pessimistic locking.

* Most multiuser DBMSs automatically initiate and enforce locking procedures. All lock information is handled by a lock manager, which is responsible for assigning and policing the locks used by the transactions

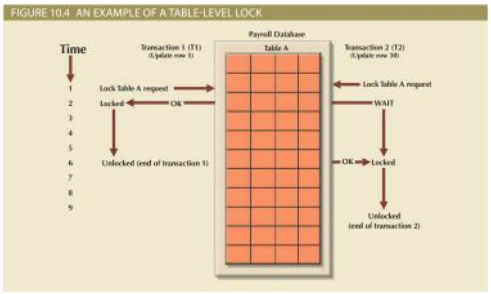
*** 10.3-a** *Lock Granularity*

Lock granularity indicates the level of lock use. Locking can take place at the following levels: database, table, page, row, or even field (attribute)

FIGURE 10.3 DATABASE-LEVEL LOCKING SEQUENCE

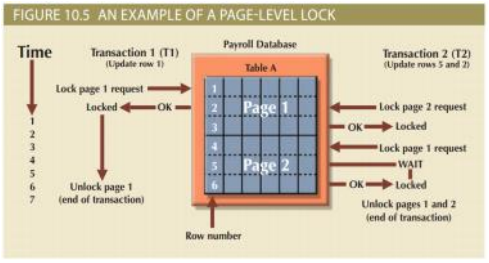


* In a database-level lock, the entire database is locked, thus preventing the use of any tables in the database by transaction T2 while transaction T1 is being executed. This level of locking is good for batch processes, but it is unsuitable for multiuser DBMSs. You can imagine how s-l-o-w data access would be if thousands of transactions had to wait for the previous transaction to be completed before the next one could reserve the entire database. Figure 10.3 illustrates the database-level lock; because of it, transactions T1 and T2 cannot access the same database concurrently even when they use different tables.

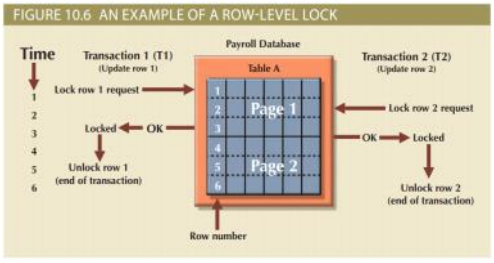


* In a table-level lock, the entire table is locked, preventing access to any row by transaction T2 while transaction T1 is using the table. If a transaction requires access to several tables, each table may be locked. However, two transactions can access the same database as long as they access different tables.

Table-level locks, while less restrictive than database-level locks, cause traffic jams when many transactions are waiting to access the same table. Consequently, table-level locks are not suitable for multiuser DBMSs. Figure 10.4 illustrates the effect of a table-level lock. Note that transactions T1 and T2 cannot access the same table even when they try to use different rows; T2 must wait until T1 unlocks the table



* In a page-level lock, the DBMS locks an entire disk page. A disk page, or page, is the equivalent of a disk block, which can be described as a directly addressable section of a disk. A table can span several pages, and a page can contain several rows of one or more tables. Page-level locks are currently the most frequently used locking method for multiuser DBMSs. An example of a page-level lock is shown in Figure 10.5. Note that T1 and T2 access the same table while locking different disk pages. If T2 requires the use of a row located on a page that is locked by T1, T2 must wait until T1 unlocks the page.



* A row-level lock is much less restrictive than the locks discussed earlier. The DBMS allows concurrent transactions to access different rows of the same table even when the rows are located on the same page. Although the row-level locking approach improves the availability of data, its management requires high overhead because a lock exists for each row in a table of the database involved in a conflicting transaction. Modern DBMSs automatically escalate a lock from a row level to a page level when the application session requests multiple locks on the same page. Figure 10.6 illustrates the use of a row-level lock.

* 10.3-b *Lock Types*

Regardless of the level of granularity of the lock, the DBMS may use different lock types or modes: binary or shared/exclusive.

* A binary lock has only two states: locked (1) or unlocked (0). If an object such as a database, table, page, or row is locked by a transaction, no other transaction can use that object. If an object is unlocked, any transaction can lock the object for its use. Every database operation requires that the affected object be locked. As a rule, a transaction must unlock the object after its termination. Therefore, every transaction requires a lock and unlock operation for each accessed data item. Such operations are automatically managed and scheduled by the DBMS; the user does not lock or unlock data items.

* An exclusive lock exists when access is reserved specifically for the transaction that locked the object. The exclusive lock must be used when the potential for conflict exists (see Table 10.11). A shared lock exists when concurrent transactions are granted read access on the basis of a common lock. A shared lock produces no conflict as long as all the concurrent transactions are read-only.

* A shared lock is issued when a transaction wants to read data from the database and no exclusive lock is held on that data item. An exclusive lock is issued when a transaction wants to update (write) a data item and no locks are currently held on that data item by any other transaction. Using the shared/exclusive locking concept, a lock can have three states: unlocked, shared (read), and exclusive (write).

* A deadlock occurs when two transactions wait indefinitely for each other to unlock data. A database deadlock, which is similar to traffic gridlock in a big city, is caused when two or more transactions wait for each other to unlock data.

* 10.3-c *Two Phase Locking to Ensure Serializability Types*

Two-phase locking (2PL) defines how transactions acquire and relinquish locks. Two-phase locking guarantees serializability, but it does not prevent deadlocks.

The two phases are :

1. A growing phase, in which a transaction acquires all required locks without unlocking any data. Once all locks have been acquired, the transaction is in its locked point.
2. A shrinking phase, in which a transaction releases all locks and cannot obtain a new lock.

The two-phase locking protocol is governed by the following rules:

- Two transactions cannot have conflicting locks.
- No unlock operation can precede a lock operation in the same transaction.
- No data is affected until all locks are obtained—that is, until the transaction is in its locked point.

* 10.3-d *Deadlocks*

A deadlock occurs when two transactions wait indefinitely for each other to unlock data.
For example, a deadlock occurs when two transactions, T1 and T2, exist in the following mode :

T1 = access data items X and Y
T2 = access data items Y and X

If T1 has not unlocked data item Y, T2 cannot begin; if T2 has not unlocked data item X, T1 cannot continue. Consequently, T1 and T2 each wait for the other to unlock the required data item. Such a deadlock is also known as a deadly embrace.

The three basic techniques to control deadlocks are:

- *Deadlock prevention*. A transaction requesting a new lock is aborted when there is the possibility that a deadlock can occur. If the transaction is aborted, all changes made by this transaction are rolled back and all locks obtained by the transaction are released. The transaction is then rescheduled for execution. Deadlock prevention works because it avoids the conditions that lead to deadlocking.

- *Deadlock detection*. The DBMS periodically tests the database for deadlocks. If a deadlock is found, the "victim" transaction is aborted (rolled back and restarted) and the other transaction continues.

- *Deadlock avoidance*. The transaction must obtain all of the locks it needs before it can be executed. This technique avoids the rolling back of conflicting transactions by requiring that locks be obtained in succession. However, the serial lock assignment required in deadlock avoidance increases action response times

* The choice of which deadlock control method to use depends on the database environment. For example, if the probability of deadlocks is low, deadlock detection is recommended. However, if the probability of deadlocks is high, deadlock prevention is recommended.