CMPG321 - Advanced Databases

Additional Notes

Semester 2

Contents

Study Unit 1

Transaction Management and Concurrency Control

1.1 What Is a Transaction?

A **transaction** is a *logical* unit of work that must be entirely completed or entirely aborted; no intermediate states are acceptable.
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.

1.1.1 Evaluating Transaction Results

Not all transactions update the database. The DBMS cannot guarantee that the semantic meaning of the transaction truly represents the real-world event.

1.1.2 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.
- *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.
- *Durability* ensures that once transaction changes are done and committed, they cannot be undone or lost, even in the event of a system failure.

1.1.3 Transaction Management with SQL

- When a `COMMIT` statement is reached, all changes are permanently recorded within the database. The `COMMIT` statement automatically ends the SQL transaction.
- When a `ROLLBACK` statement is reached, 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`.

1.1.4 The Transaction Log

A DBMS uses a **transaction log** to keep track of all transactions that update the database. 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. Some RDBMSs use the transaction log to recover a database forward to a currently consistent state.

While the DBMS executes transactions that modify the database, it also automatically updates the transaction log. 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.

1.2 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.

1.2.1 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).

1.2.2 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.

1.2.3 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. The problem is that the transaction might read some data before it is changed and other data after it is changed, thereby yielding inconsistent results.

1.2.4 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, such as locking or time stamping methods.

The scheduler's main job is to create a serializable schedule of a transaction's operations, in which the interleaved execution of the transactions (T1, T2, T3, etc.) yields the same results as if the transactions were executed in serial order (one after another).

The scheduler also makes sure that the computer's central processing unit (CPU) and storage systems are used efficiently.

1.3 Concurrency Control with Locking Methods

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. A transaction acquires a lock prior to data access; the lock is released (unlocked) when the transaction is completed.

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.