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.

1.3.1 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).

Database Level - 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.

Transactions T1 and T2 cannot access the same database concurrently *even when they use different tables.*

Table Level - 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.

Page Level - In a **page-level lock**, the DBMS locks an entire **diskpage**. A diskpage, or page, is the equivalent of a *diskblock*, which can be described as a directly addressable section of a disk. A page has a fixed size, such as 4K, 8K, or 16K. For example, if you want to write only 73 bytes to a 4K page, the entire 4K page must be read from disk, updated in memory, and written back to 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.

Row Level - 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.

Field Level - The **field-level lock** allows concurrent transactions to access the same row as long as they require the use of different fields (attributes) within that row. Although field-level locking clearly yields the most flexible multiuser data access, it is rarely implemented in a DBMS because it requires an extremely high level of computer overhead and because the row-level lock is much more useful in practice.

1.3.2 Lock Types

Binary - 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.

Shared/Exclusive - 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. 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. Two transactions conflict only when at least one is a write transaction. Because the two read transactions can be safely executed at once, shared locks allow several read transactions to read the same data item concurrently. For example, if transaction T1 has a shared lock on data item X and transaction T2 wants to read data item X, T2 may also obtain a shared

lock on data item X. If transaction T2 updates data item X, an exclusive lock is required by T2 over data item X. The exclusive lock is granted if and only if no other locks are held on the data item (this condition is known as the **mutual exclusive rule**: only one transaction at a time can own an exclusive lock on an object.) Therefore, if a shared (or exclusive) lock is already held on data item X by transaction T1, an exclusive lock cannot be granted to transaction T2, and T2 must wait to begin until T1 commits. In other words, a shared lock will always block an exclusive (write) lock; hence, decreasing transaction concurrency.

Although the use of shared locks renders data access more efficient, a shared/exclusive lock schema increases the lock manager's overhead for several reasons:

- The type of lock held must be known before a lock can be granted.
- Three lock operations exist: `READ_LOCK` to check the type of lock, `WRITE_LOCK` to issue the lock, and `UNLOCK` to release the lock.
- The schema has been enhanced to allow a lock upgrade from shared to exclusive and a lock downgrade from exclusive to shared. Although locks prevent serious data inconsistencies, they can lead to two major problems:
  - The resulting transaction schedule might not be serializable.
  - The schedule might create deadlocks. 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.

  Fortunately, both problems can be managed: serializability is attained through a locking protocol known as two-phase locking, and deadlocks can be managed by using deadlock detection and prevention techniques.

1.3.3 Two-Phase Locking to Ensure Serializability

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.

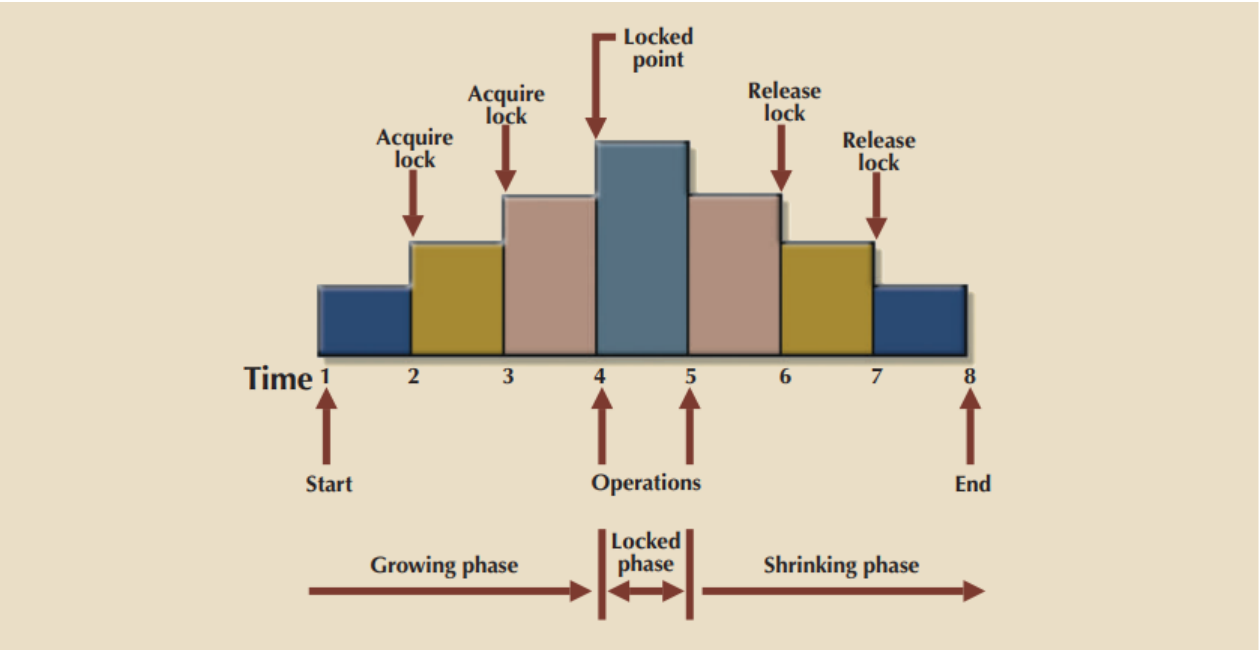

Figure 1.1: Two-phase locking process illustrated

1.3.4 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**. Note that deadlocks are possible only when one of the transactions wants to obtain an exclusive lock on a data item; no deadlock condition can exist among *shared* locks.

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.

1.4 Concurrency Control with Time Stamping Methods

The **time stamping** approach to scheduling concurrent transactions assigns a global, unique time stamp to each transaction. The time stamp value produces an explicit order in which transactions are submitted to the DBMS. Time stamps must have two properties: uniqueness and monotonicity. **Uniqueness** ensures that no equal time stamp values can exist, and **monotonicity**[1] ensures that time stamp values always increase.

1.4.1 Wait/Die and Wound/Wait Schemes

Using the wait/die scheme:

- If the transaction requesting the lock is the older of the two transactions, it will wait until the other transaction is completed and the locks are released.
- If the transaction requesting the lock is the younger of the two transactions, it will die (roll back) and is rescheduled using the same time stamp.

In short, in the **wait/die** scheme, the older transaction waits for the younger one to complete and release its locks. In the wound/wait scheme:

- If the transaction requesting the lock is the older of the two transactions, it will *preempt (wound)* the younger transaction by rolling it back. T1 preempts T2 when T1 rolls back T2. The younger, preempted transaction is rescheduled using the same time stamp.
- If the transaction requesting the lock is the younger of the two transactions, it will wait until the other transaction is completed and the locks are released.

In short, in the **wound/wait** scheme, the older transaction rolls back the younger transaction and reschedules it. In both schemes, one of the transactions waits for the other transaction to finish and release the locks.

1.5 Concurrency Control with Optimistic Methods

The **optimistic approach** is based on the assumption that the majority of database operations do not conflict. The optimistic approach requires neither locking nor time stamping techniques. Instead, a transaction is executed without restrictions until it is committed. Using an optimistic approach, each transaction moves through two or three phases, referred to as *read*, *validation*, and *write*.

- During the *read phase*, the transaction reads the database, executes the needed computations, and makes the updates to a private copy of the database values. All update operations of the transaction are recorded in a temporary update file, which is not accessed by the remaining transactions.
- During the *validation phase*, the transaction is validated to ensure that the changes made will not affect the integrity and consistency of the database. If the validation test is positive, the transaction goes to the write phase. If the validation test is negative, the transaction is restarted and the changes are discarded.
- During the *write phase*, the changes are permanently applied to the database.

[1] The term monotonicity is part of the standard concurrency control vocabulary.

1.6 Database Recovery Management

Database recovery restores a database from a given state (usually inconsistent) to a previously consistent state. Recovery techniques are based on the **atomic transaction property**: all portions of the transaction must be treated as a single, logical unit of work in which all operations are applied and completed to produce a consistent database.

Critical events can cause a database to stop working and compromise the integrity of the data. Examples of critical events are:

- *Hardware/software failures.* A failure of this type could be a hard disk media failure, a bad capacitor on a motherboard, or a failing memory bank. Other causes of errors under this category include application program or operating system errors that cause data to be overwritten, deleted, or lost.
- *Human-caused incidents.* This type of event can be categorized as unintentional or intentional.
  - An unintentional failure is caused by a careless end user. Such errors include deleting the wrong rows from a table, pressing the wrong key on the keyboard, or shutting down the main database server by accident.
  - Intentional events are of a more severe nature and normally indicate that the company data is at serious risk. Under this category are security threats caused by hackers trying to gain unauthorized access to data resources and virus attacks caused by disgruntled employees trying to compromise the database operation and damage the company.
- *Natural disasters.* This category includes fires, earthquakes, floods, and power failures.

1.6.1 Transaction Recovery

Database transaction recovery uses data in the transaction log to recover a database from an inconsistent state to a consistent state.There are four important concepts that affect the recovery process:

- The **write-ahead-log protocol** ensures that transaction logs are always written before any database data is actually updated.
- **Redundant transaction logs** ensure that a physical disk failure will not impair the DBMS's ability to recover data.
- Database **buffers** are temporary storage areas in primary memory used to speed up disk operations. To improve processing time, the DBMS software reads the data from the physical disk and stores a copy of it on a "buffer" in primary memory.
- Database **checkpoints** are operations in which the DBMS writes all of its updated buffers in memory (also known as *dirty buffers*) to disk. While this is happening, the DBMS does not execute any other requests. A checkpoint operation is also registered in the transaction log. As a result of this operation, the physical database and the transaction log will be in sync.

The database recovery process involves bringing the database to a consistent state after a failure. Transaction recovery procedures generally make use of deferred-write and write-through techniques.

When the recovery procedure uses a **deferred-write technique** (also called a **deferred update**), the transaction operations do not immediately update the physical database. Instead, only the transaction log is updated. The database is physically updated only with data from committed transactions, using information from the transaction log. If the transaction aborts before it reaches its commit point, no changes (no `ROLLBACK` or undo) need to be made to the database because it was never updated. The recovery process for all started and committed transactions (before the failure) follows these steps:

1. Identify the last checkpoint in the transaction log. This is the last time transaction data was physically saved to disk.

2. For a transaction that started and was committed before the last checkpoint, nothing needs to be done because the data is already saved.

3. For a transaction that performed a commit operation after the last checkpoint, the DBMS uses the transaction log records to redo the transaction and update the database, using the "after" values in the transaction log. The changes are made in ascending order, from oldest to newest.

4. For any transaction that had a `ROLLBACK` operation after the last checkpoint or that was left active (with neither a `COMMIT` nor a `ROLLBACK`) before the failure occurred, nothing needs to be done because the database was never updated.

When the recovery procedure uses a **write-through technique** (also called an **immediate update**), the database is immediately updated by transaction operations during the transaction's execution, even before the transaction reaches its commit point. If the transaction aborts before it reaches its commit point, a `ROLLBACK` or undo operation needs to be done to restore the database to a consistent state. In that case, the `ROLLBACK` operation will use the transaction log "before" values. The recovery process follows these steps:

1. Identify the last checkpoint in the transaction log. This is the last time transaction data was physically saved to disk.

2. For a transaction that started and was committed before the last checkpoint, nothing needs to be done because the data is already saved.

3. For a transaction that was committed after the last checkpoint, the DBMS re-does the transaction, using the "after" values of the transaction log. Changes are applied in ascending order, from oldest to newest.

4. For any transaction that had a `ROLLBACK` operation after the last checkpoint or that was left active (with neither a `COMMIT` nor a `ROLLBACK`) before the failure occurred, the DBMS uses the transaction log records to `ROLLBACK` or undo the operations, using the "before" values in the transaction log. Changes are applied in reverse order, from newest to oldest.

Study Unit 2

Database Performance Tuning and Query Optimization

Study Unit 3

Distributed Database Management Systems

Study Unit 4

Business Intelligence and Data Warehouses

Study Unit 5

Big Data and NoSQL