

Transaction Management and Concurrency Control

CS341



Transaction Management

CS341

Understanding Transaction

- A transaction is
  - A single unit of work
  - A sequence of operations performed as a single logical unit of work
- If a transaction is successful
  - All of the data modifications made during the transaction are committed and become a permanent part of the database.
- If a transaction is NOT successful (if it encounters errors, or canceled, or …)
  - must be rolled back
  - then all of the data modifications are undone

Understanding Transaction (cont.)

- A TRANSACTION is a logical unit of work that must be either entirely completed or aborted
- Successful transaction changes database from one consistent state to another
  - One in which all data integrity constraints are satisfied
- Most real-world database transactions are formed by two or more database requests (multiple SQL statements)
  - Each one of which yields I/O operations
- A single task may require MANY changes to the database
  - If ALL changes are NOT made database integrity will be lost

Transaction - Example

Steps to - Enroll a student - GENET to class DRB2019

1. START
2. Check to see if student "GENET" exists .. read only
3. Check to see if class "DRB2019" exists .. read only
4. Insert into ***Admission*** table .. make changes (INSERT)
5. Insert into ***Class*** table .. make changes(INSERT)
6. Commit transaction
8. END

IF an ERROR is encountered at any stage then whatever has been done should be UNDONE

ACID Properties

- The ACID property stands for
  - **A**tomicity
  - **C**onsistency
  - **I**solation
  - **D**urability
- Atomicity
  - A transaction must be an atomic unit of work
  - Either ALL or NONE of its data modifications are performed
  - All operations of a transaction must be completed
    - if not, then the transaction is aborted
  - Free from interference by operations of other transactions
    - A transaction is treated as an indivisible unit of work from the point of view of other transactions

ACID Properties (cont.)

- Consistency
  - A transaction must leave all data in a consistent state when it is completed
- Isolation
  - Modifications made by concurrent transactions must be isolated from the changes made by each other
  - A transaction either recognizes data in the state it was
    - Before another concurrent transaction modified it, or
    - After the second transaction has completed
      - IT does not recognize an intermediate state.
- Durability
  - After a transaction has completed,
    - Its effects are permanently in place in the system.
    - The modifications persist even in the event of a system failure

An Example

- Transfer BIRR 1,000 from checking account to savings account

```
UPDATE checking
SET Balance = Balance - 1000
WHERE AccountID = '213009'

UPDATE savings
SET Balance = Balance + 1000
WHERE AccountID = '456130443'
```

- What if the first statement executes and the second one doesn't?

Transactions with SQL

- Transaction support is provided by two SQL statements: COMMIT and ROLLBACK
- Transaction sequence must continue until one of the following is reached
  - COMMIT
  - ROLLBACK
  - End of program
  - Program is abnormally terminated

Transactions with SQL (cont.)

- BEGIN TRAN
  - Set the starting point
- COMMIT TRAN
  - The transaction is completed
  - All statements starting from the beginning of the transaction are made permanent
  - All changes are permanently recorded within the database
- ROLLBACK
  - Undo what has happened starting from the beginning of the transaction
  - All changes are aborted and the database is restored to a previous consistent state

Transaction – Example

```
DECLARE @errValue int

BEGIN TRANSACTION
        UPDATE    checking
        SET       Balance = Balance – 1000
        WHERE     AccID = ‘213009’

        SET  @errValue = @@ERROR

        IF     @errValue > 0
                  ROLLBACK TRANSACTION

        UPDATE    savings
        SET       Balance = Balance + 1000
        WHERE     AccID = ‘456130443

        SET  @errValue = @@ERROR

        IF     @errValue > 0
                  ROLLBACK TRANSACTION
COMMIT TRANSACTION
```

SQL Server - Transactions

- SQL Server operates in the following transaction modes:
- Auto-Commit Transaction Mode
  - Each individual statement is a transaction
- Implicit Transaction Mode
  - A new transaction is implicitly started when the prior transaction completes
  - BUT, each transaction is explicitly completed with a COMMIT or ROLLBACK statement.
- Explicit Transaction Mode
  - Each transaction is explicitly started with the BEGIN TRANSACTION statement and
  - explicitly ended with a COMMIT or ROLLBACK statement.

SQL Server – Transactions (cont.)

SET IMPLICIT_TRANSACTIONS ON | OFF

- When ON, SET IMPLICIT_TRANSACTIONS
  - The connection is set to implicit transaction mode
  - When OFF, it returns the connection to AUTO-COMMIT transaction mode.
- When a connection is in implicit transaction mode and the connection is not currently in a transaction
  - Any of the following statements starts a transaction

    ALTER TABLE, CREATE, DELETE, DROP, CREATE, FETCH, GRANT, INSERT, REVOKE, SELECT, TRUNCATE TABLE, UPDATE

SQL Server – Transactions (cont.)

- Example

```
SET IMPLICIT_TRANSACTIONS ON

INSERT Department (DepName)
Values( 'New Dep')

COMMIT TRAN
```

- If the transaction is not committed, then the INSERT will be rolled back

The Transaction Log

- A transaction log keeps records of all transactions and the database modifications made by each transaction
- Maintaining a log provides the possibility of redoing and undoing a modification
- The transaction log is used for database recovery
- Transaction log stores:
  - A record for the beginning of transaction
  - For each transaction component:
    - Type of operation being performed (update, delete, insert)
    - Names of objects affected by transaction
    - “Before” and “after” values for updated fields
    - Pointers to previous and next transaction log entries for the same transaction
  - Ending (COMMIT) of the transaction

The Transaction Log (cont.)

TABLE 10.1 A Transaction Log

TRL_ ID	TRX_ NUM	PREV PTR	NEXT PTR	OPERATION	TABLE	ROW ID	ATTRIBUTE	BEFORE VALUE	AFTER VALUE
341	101	Null	352	START	****Start Transaction				
352	101	341	363	UPDATE	PRODUCT	1558-QW1	PROD_QOH	25	23
363	101	352	365	UPDATE	CUSTOMER	10011	CUST_ BALANCE	525.75	615.73
365	101	363	Null	COMMIT	**** End of Transaction				



TRL_ID = Transaction log record ID
TRX_NUM = Transaction number
(*Note*: The transaction number is automatically assigned by the DBMS.)

PTR = Pointer to a transaction log record ID

Write-Ahead Transaction Log

- SQL Server uses a write-ahead log (WAL)
  - Guarantees that no data modifications are written to disk before the associated log record is written to disk.
- How does the write-ahead log works?
- SQL Server maintains a buffer cache into which it reads data pages when data must be retrieved.
  - Data modifications are not made directly to disk, but are made to the copy of the page in the buffer cache.
  - The modification is not written to disk until
    - a **checkpoint** occurs in the database, or
    - the modification NEED TO be written to disk so the buffer can be used to hold a new page.
  - Log records are written to disk when the transactions are committed.



Concurrency Control

Concurrency Control

- DBMS – Support multiple users to read and update data at the same time
  - Therefore, the DBMS must control the balance between Isolation and Concurrency
- **Concurrency** is the ability of a DBMS to allow multiple users to work together
- **Isolation** is the ability of one person to control a portion of data
- A Transaction can specify the degree to which it must be isolated from the data modifications made by other transactions
  - This is referred to as the ISOLLATION LEVEL for the transaction

The Isolation Level

- The ability to shield a transaction from the effects of other concurrent transactions can be set to different levels
  - This is called the ISOLATION LEVEL for the transaction
- The isolation level of a transaction controls:
  - What type of locks are requested for each action
  - How long the locks are to be held
  - Whether locks are taken when data is read
  - Whether an operation works with uncommitted data, etc

The Isolation Level (cont.)

- A lower isolation level
  - Allows many users to access data at the same time
  - but increases the number of concurrency problems (such as dirty reads or lost updates) users might encounter.
- A higher isolation level
  - reduces the concurrency problems that users may encounter
  - but increases the chances that one transaction will block another
- Choosing the appropriate isolation level depends on balancing the concurrency requirements of the application against the problems of each isolation level

SQL Server Isolation Levels

- SQL Server supports the following isolation levels
  - READ UNCOMMITTED
  - READ COMMITTED
  - REPEATABLE READ
  - SNAPSHOT
  - SERIALIZABLE

SQL Server Isolation Levels (cont.)

- READ UNCOMMITTED
  - Statements can read rows that have been modified by other transactions before the transaction is not committed
- READ COMMITTED
  - Statements cannot read data that has been modified but not committed by other transactions
  - Prevents dirty reads
  - May result in nonrepeatable reads or phantom data
  - This option is the SQL Server default

SQL Server Isolation Levels (cont.)

- REPEATABLE READ
  - Specifies that statements cannot read data that has been modified but not yet committed by other transactions
  - No other transactions can modify data that has been read by the current transaction until the current transaction completes
- SNAPSHOT
  - Specifies that data read by any statement in a transaction will be the version of the data that existed at the start of the transaction
  - The transaction can only recognize data modifications that were committed before the start of the transaction
  - The effect is as if the statements in a transaction get a snapshot of the committed data as it existed at the start of the transaction

SQL Server Isolation Levels (cont.)

- SERIALIZABLE
  - Statements cannot read data that has been modified but not yet committed by other transactions
  - No other transactions can modify data that has been read by the current transaction until the current transaction completes
  - Other transactions cannot insert new rows with key values that would fall in the range of keys read by the current transaction

SQL Server Isolation Levels (cont.)

- Example: Setting the Isolation Level

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
GO

BEGIN TRANSACTION;
SELECT   *
FROM     EmployeePayHistory

UPDATE  Department
SET       Location = 'Addis Ababa'

COMMIT TRANSACTION
GO
```

Isolation levels: Problems

- Simultaneous execution of transactions over a shared database can create several data integrity and consistency problems
- Different isolation levels may have different kinds of side-effects or problems
- Problems of concurrency control:
  - Lost Updates
  - Dirty Reads
  - Unrepeatable Reads
  - Phantom Reads

The Lost Update Problem

- Lost Update
  - Occurs when two different transactions are trying to update the same data at the same time.
  - Update made by one transaction is lost
- Example: Both T1and T2 are updating ‘acc1’

Transactions		**State**
T1:	**T2:**	**acc1**
read(acc1)		20
acc1 := acc1 + 10	**read**(acc1)	20
	acc1 := acc1 + 20	
	write(acc1)	40
	commit	
write(acc1)		30
commit		

- Result: Changes made by T2 are lost

The Dirty Read Problem

- Dirty Read
  - A transaction is reading data that has not been committed by another transaction
- Example

Transactions		**State**	
T1:	**T2:**	**acc1**	**sum**
read(acc1)		20	0
acc1 := acc1 + 10			
write(acc1)	...		
	read(acc1)	30	
	sum := sum + acc1		
	write(sum)		30
acc1 := acc1 + 10	**commit**		
write(acc1)		40	
commit			

- Result: T2 sees dirty data (uncommitted data) of T1

Unrepeatable Read Problem

- Unrepeatable Read
  - Multiple reads of a transaction are different because of updates made by another transaction
- Example:

Transactions		State
T1:	**T2:**	**acc1**
read(acc1)		20
	read(acc1)	20
	acc1 := acc1 + 20	
	write(acc1)	40
	commit	
read(acc1)		
sum := sum + acc1		
write(sum)		40
commit		

- Result: Multiple reads of Acc1 by T1 are different

Phantom Read

- A **Phantom Read** occurs when new rows are added or removed by another transaction to the records being read.

Concurrency side effects enabled by the different isolation levels

Isolation level	Dirty read	Non-Repeatable read	Phantom
Read uncommitted	Yes	Yes	Yes
Read committed	No	Yes	Yes
Repeatable read	No	No	Yes
Snapshot	No	No	No
Serializable	No	No	No

Concurrency Control with Locking

- A Lock guarantees exclusive use of a data item to the current transaction
- Locks are required to prevent another transaction from reading inconsistent data
- The Lock Manager is responsible for assigning and policing the locks used by transactions
- Locking can take place at following levels
  - Database-level lock
  - Table-level lock
  - Page-level lock
  - Row-level lock
  - Field-level lock

Main Types of Locks

- Exclusive lock
  - Access is specifically reserved for transaction that locked an object
  - Must be used when there is a potential for conflicts
- Shared lock
  - Concurrent transactions are granted read access on basis of a common lock

Deadlocks

- A DEADLOCK is a condition that occurs when two transactions wait for each other to unlock data
- Example
  - T1 needs data items X and Y
  - T2 needs Y and X
  - Each gets access to the first piece of data
  - But they have to waits to get the second
    - which the other transaction has locked exclusively
- Possible only if one of the transactions wants to obtain an exclusive lock on a data item
  - No deadlock condition can exist among shared locks

Deadlocks (cont.)

- When the DBMS detects a deadlock
  - One f the transactions must be aborted (rolled back) and the other transaction continues
- When a deadlock occurs, by default, in SQL Server the deadlock "victim“ is selected by
  - Identifying the process that uses the least resources to roll back
  - Transactions with a lower DEADLOCK_PRIORITY is rolled back
- Tips on avoiding deadlocks
  - Ensure the database design is properly normalized
  - Design applications so that access to database objects is in the same order each time
  - Do not allow any user input during transactions
  - Keep transactions as short as possible
  - Avoid using object that are resources intensive, eg. cursors