UST 과학기술연합대학원대학교
UNIVERSITY OF SCIENCE & TECHNOLOGY

Lecture 10: Transactions & Concurrency Control

Dr. Kyong-Ha Lee
(kyongha@kisti.re.kr)

KISTI 한국과학기술정보연구원
Korea Institute of Science and Technology Information
www.kisti.re.kr

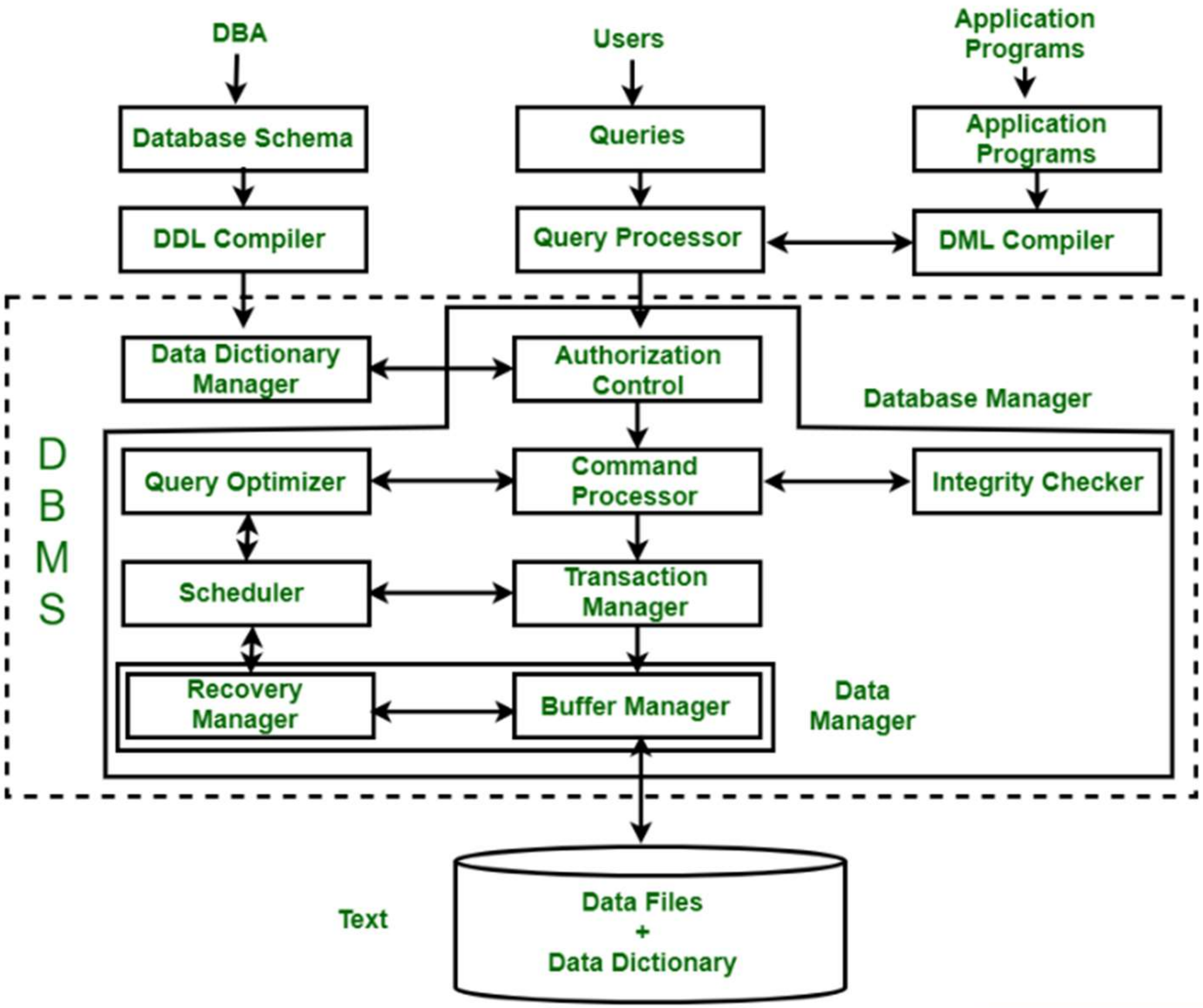
Contents

Brief overview of this lecture

- Basic theories and principles about Index mechanisms used in database systems
- Not much discussion on implementation or tools, but will be happy to discuss them if there are any questions

*Disclaimer: these slides are based on the slides created by the authors of Database System Concepts 7th ed. and modified by K.H. Lee.

DBMS Architecture OVerview



Transaction

- A **unit** of execution that accesses and possibily update various data items
  - From the DBMS's point of view, a transaction is a series of the following actions
    - **READs:** DB object is read from disk into buffer page.
    - **WRITEs:** DB object is written from buffer to disk
    - **ABORT:** Last action of a Xact that fails
    - **COMMIT:** Last action of a Xact that succeeds
- e.g., transaction to transfer $50 from account A to account B:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B*)

Two main issues to deal with :

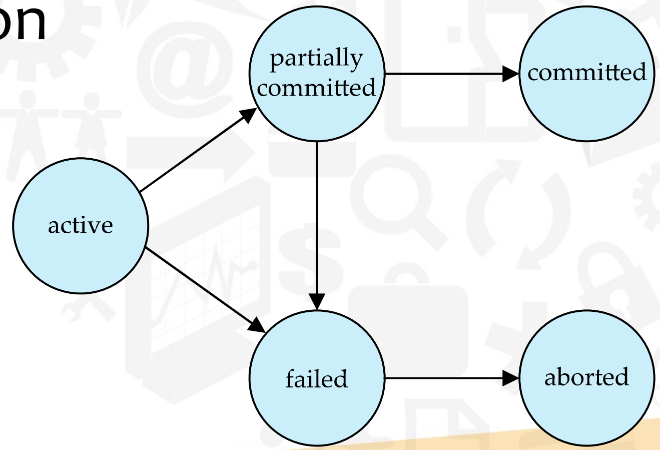
- Failures of various kinds, such as HW failures and system crashes
- Concurrent execution of multiple transactions

Properties of Transactions (ACID)

- **Atomicity** : all or nothing
  - If system crashes, partially done transactions must be undone or "rolled back“
- **Consistency**: if each transaction is consistent and the DB starts consistents, it must be ended consistently
- **Isolation**: transactions are protected from the effects of all other concurrent transactions
  - Concurrency for performance must not affect each transaction.
- **Durability**: once the transaction has completed, updates by the transaction must persist even if SW or HW failures come

Transaction State

- **Active** – Initial state; TX stays in this stae while it is executing
- **Partially committed** – after final statement has been executed
- **Failed** – after that normal execution can no longer proceed
- **Aborted** – after TX was rolled back and DB restored to its state prior to the start of the TX
  - Restart the TX
  - Kill the transaction
- **Committed** – after successful completion



Isolatioin requirement

- Isolation requirement
  - Isolation can be ensured trivially by running transactions **serially.** That is, one after the other.

T1		**T2**
1.	**read**(*A*)	
2.	*A* := *A* – 50	
3.	**write**(*A*)	
		read(A), read(B), print(A+B)
4.	**read**(*B*)	
5.	*B* := *B* + 50	
6.	**write**(*B*	

- However, executing multiple transactions concurrently has significant benefits

Concurrent Execution

- Users submit transactions and can think of each transaction as executing by itself(Give users such illusion)
  - Concurrency is achieved by the DBMS that i**nterleaves actions(reads/writes of DB objects) of various transactions**
  - Advantages
  - **Increased processor and disk utilization,** leading to better transaction *throughput*
    - E.g., one transaction can be using the CPU while another is reading from or writing to the disk
  - **Reduced average response time** for transactions: short transactions need not wait behind long ones.

Concurrent Execution

- Each transaction must leave the database in a consistent state if the DB is consistent when the transaction begins
  - DBMS will enforce some Integrity Constraints(IC), depending on the ICs declared in CREATE TABLE statements
  - Beyond this, the DBMS does not really understand the semantics of the data (e.g., it does not understand how the interest on a bank account is computed)
- <u>Issues:</u> effect of *interleaving* transactions and *system crashes*
- **Concurrency control schemes** – mechanisms to achieve isolation

Schedule

- **Schedule**
  - a sequences of instructions that specify the **chronological order** in which instructions of concurrent transactions are executed
  - A schedule for a set of transactions **must consist of all instructions** of those transactions
  - Must preserve the order in which the instructions appear in each transaction.

- Examle
  - Let T_1 transfer $50 from *A* to *B*, and T_2 transfer 10% of the balance from *A* to *B*.
  - A serial schedule in which T_1 is followed by T_2 :

T_1	T_2
read (A) $A := A - 50$ write (A) read (B) $B := B + 50$ write (B) commit	
	read (A) $temp := A * 0.1$ $A := A - temp$ write (A) read (B) $B := B + temp$ write (B) commit

Schedule 1

- Let T_1 and T_2 be the transactions defined previously. The following schedule is not a serial schedule, but it is *equivalent* to Schedule 1

T_1	T_2
read (A) $A := A - 50$	
	read (A) $temp := A * 0.1$ $A := A - temp$ write (A) read (B)
write (A) read (B) $B := B + 50$ write (B) commit	
	$B := B + temp$ write (B) commit

Schedule 2

T_1	T_2
read (A) $A := A - 50$ write (A)	
	read (A) $temp := A * 0.1$ $A := A - temp$ write (A)
read (B) $B := B + 50$ write (B) commit	
	read (B) $B := B + temp$ write (B) commit

Schedule 3

Serializability

- **Basic Assumption** – Each transaction preserves database consistency.
  - serial execution of a set of transactions preserves database consistency.
- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule
  - Different forms of schedule equivalence give rise to the notions of:

  1. **Conflict serializability**
  2. **View serializability**

Note: we ignore operations other than read and write instructions

Conflicting instruction

- Instructions I_i and I_j of transactions T_i and T_j respectively, **conflict** if and only if there exists some item Q accessed by both I_i and I_j, and at least one of these instructions wrote Q.
  1. I_i = **read**(Q), I_j = **read**(Q). I_i and I_j don't conflict.
  2. I_i = **read**(Q), I_j = **write**(Q). They conflict.
  3. I_i = **write**(Q), I_j = **read**(Q). They conflict
  4. I_i = **write**(Q), I_j = **write**(Q). They conflict
- Conflict between I_i and I_j forces a (logical) temporal order between them.
- If I_i and I_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

- If a schedule *S* can be transformed into a schedule *S'* by a series of swaps of non-conflicting instructions, we say that *S* and *S'* are **conflict equivalent**.
- We say that a schedule *S* is **conflict serializable** if it is conflict equivalent to a serial schedule
- Schedule 3 is conflict serializable and schedule 4 is not. Why?
  - Unable to swan instructions in schedule 4 to obtain either the serial schedule < T_3, T_4 >, or the serial schedule < T_4, T_3 >.

T_1	T_2
read (A)	
write (A)	
	read (A)
	write (A)
read (B)	
write (B)	
	read (B)
	write (B)

Schedule 3

T_3	T_4
read (Q)	
	write (Q)
write (Q)	

Schedule 4

View Serializability

- *Schedule* S and *S'* are **view equivalent** if the 3 conditions are met, for each data item *Q*
  1. If in schedule S, transaction T_i reads the initial value of Q, then in schedule *S'* also transaction T_i must read the initial value of *Q*
  2. If in schedule S transaction T_i executes **read**(*Q*), and that value was produced by transaction T_j (if any), then in schedule *S'* also transaction T_i must read the value of *Q* that was produced by the same **write**(Q) operation of transaction T_j .
  3. The transaction (if any) that performs the final **write**(*Q*) operation in schedule *S* must also perform the final **write**(*Q*) operation in schedule *S'*.

View Serializability(Cont.)

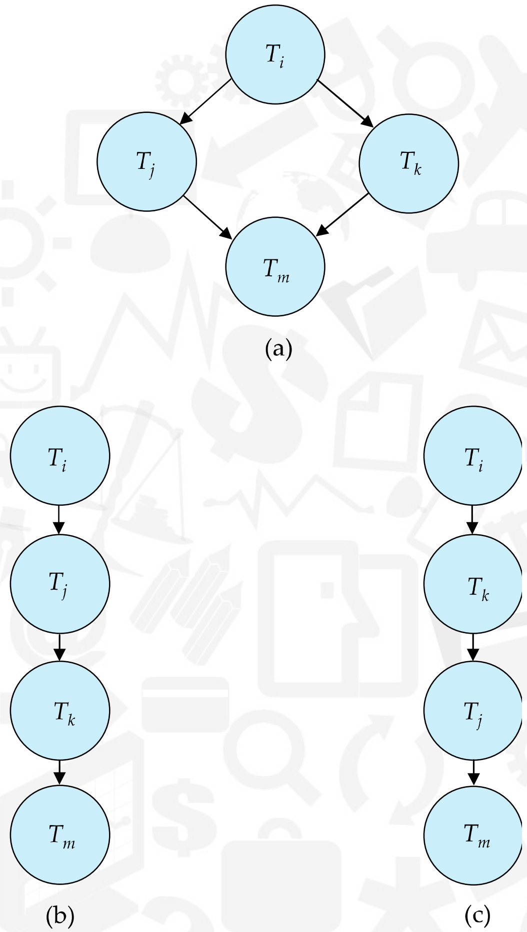
- A schedule S is **view serializable** if it is view equivalent to a serial schedule.
- Every conflict serializable schedule is also view serializable.
- Below is a schedule which is view-serializable but *not* conflict serializable.

T_{27}	T_{28}	T_{29}
read (Q)		
	write (Q)	
write (Q)		
		write (Q)

- What serial schedule is above equivalent to?
- Every view serializable schedule that is not conflict serializable has **blind writes.**

Testing for serializaibility

- **Precedence graph** — a direct graph where the vertices are the transactions (names)
  – Consider some schedule of a set of transactions $T_1, T_2, \ldots, T_n$
  – We draw an arc from T_i to T_j if the two transaction conflict, and T_i accessed the data item on which the conflict arose earlier.
  – We may label the arc by the item that was accessed.
- A schedule is conflict serializable if and only if its precedence graph is acyclic
- If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph.



Recoverable Schedule

- We need to address the effect of transaction failures on concurrently running transactions
- **Recoverable schedule**
  – if a transaction T_j reads a data item previously written by a transaction T_i, then the commit operation of T_i appears before the commit operation of T_j.
- The following schedule (Schedule 11) is not recoverable

T_8	T_9
read (A)	
write (A)	
	read (A)
	commit
read (B)	

- DB system must ensure recoverable schedules
  – If T_8 should abort, T_9 would have read (and possibly shown to the user) an inconsistent database state

Cascading Rollbacks

- **Cascading rollback**
  - a single TX failure leads to a series of transaction rollbacks.
  - Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)

T_{10}	T_{11}	T_{12}
read (A)		
read (B)		
write (A)		
	read (A)	
	write (A)	
		read (A)
abort		

  If T_{10} fails, T_{11} and T_{12} must also be rolled back.
    - Can lead to the undoing of a significant amount of work
- **Cascadeless schedules**
  - For each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j.
  - Every Cascadeless schedule is also recoverable
- It is desirable to restrict the schedules to those that are cascadeless

Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are
  - either conflict or view serializable, and
  - are recoverable and preferably cascadeless
- Serial Schedule provides a poor degree of concurrency
- Concurrency-control schemes tradeoff between the amount of concurrency they allow and the amount of overhead that they incur.
- Some schemes allow only conflict-serializable schedules to be generated, while others allow view-serializable schedules that are not conflict-serializable.

Weak Levels of Consistency

- Some applications are willing to live with **<u>weak levels of consistency</u>**, allowing schedules that are not serializable
  - E.g., a read-only transaction that wants to get an approximate total balance of all accounts
  - E.g., database statistics computed for query optimization can be approximate (why?)
  - Such transactions need not be serializable with respect to other transactions
- Tradeoff accuracy for performance

Levels of Consistency in SQL-92

- **Serializable** — default
- **Repeatable read** — only committed records to be read.
  - Repeated reads of same record must return same value.
  - However, a transaction may not be serializable – it may find some records inserted by a transaction but not find others.
- **Read committed** — only committed records can be read.
  - Successive reads of record may return different (but committed) values.
- **Read uncommitted** — even uncommitted records may be read.

Levels of Consistency

- Lower degrees of consistency useful for gathering approximate information about the database
- Warning: some database systems do not ensure serializable schedules by default
- E.g., Oracle (and PostgreSQL prior to version 9) by default support a level of consistency called snapshot isolation (not part of the SQL standard)
- MariaDB offer all four consistency levels
  - Default isolation level for InnoDB is REPEATABLE READ

Transaction Definition in SQL

- In SQL, a transaction begins implicitly.
- A transaction in SQL ends by:
  - **Commit work** commits current transaction and begins a new one.
  - **Rollback work** causes current transaction to abort.
- In almost all database systems, by default, every SQL statement also commits implicitly if it executes successfully
  - Implicit commit can be turned off by a database directive
    - E.g., in JDBC -- connection.setAutoCommit(false);
- Isolation level can be set at database level
- Isolation level can be changed at start of transaction
    - E.g. In SQL **set transaction isolation level serializable**
    - E.g. in JDBC -- connection.setTransactionIsolation(
      Connection.TRANSACTION_SERIALIZABLE)

Implementation of Isolation Levels

- Locking
  - Lock on whole database vs lock on items
  - How long to hold lock?
  - Shared vs exclusive locks
- Timestamps
  - Transaction timestamp assigned e.g. when a transaction begins
  - Data items store two timestamps
    - Read timestamp
    - Write timestamp
  - Timestamps are used to detect out of order accesses
- Multiple versions of each data item
  - Allow transactions to read from a "snapshot" of the database

Question?

–source: https://www.fox.com/the-simpsons