DTS207TC Database Development and Design

Lecture 11

Chap 17. Transactions

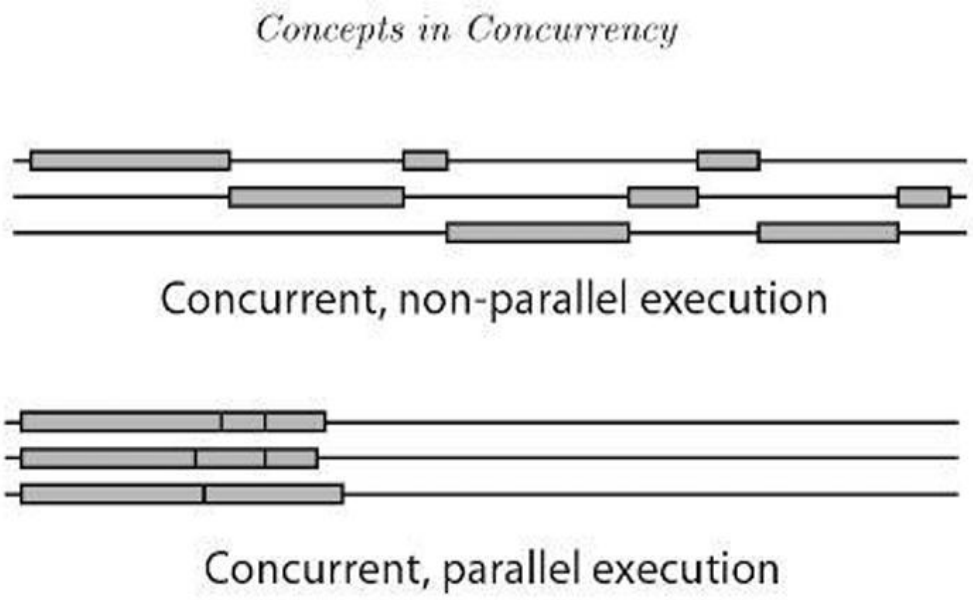
Di Zhang, Autumn 2025

Outline

- Review of OS&Parallel Computing
- Transaction Concept
- Serializability
- Recoverability
- Implementation of Isolation
- Transaction Definition in SQL

Concurrency vs. Parallelism

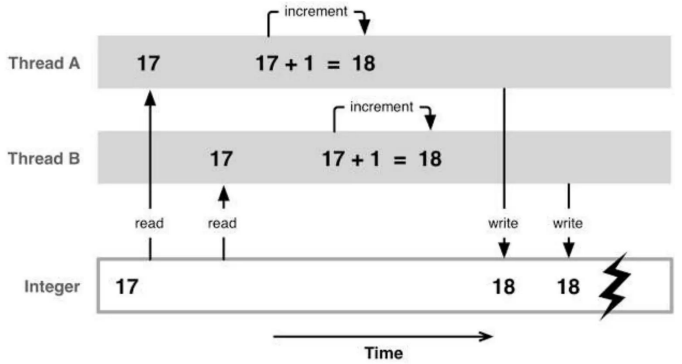
- Parallel Computing Perspective
  - Multiple cores execute tasks simultaneously
  - Goal: Speedup through parallel execution
- Operating System Perspective
  - Process/thread scheduling
  - Time-sharing creates the illusion of concurrent execution
- Database Connection
  - A database handles multiple transaction requests at once
  - This is also a form of concurrency and *possibly* parallelism



The Core Problem: Synchronization & Race Conditions

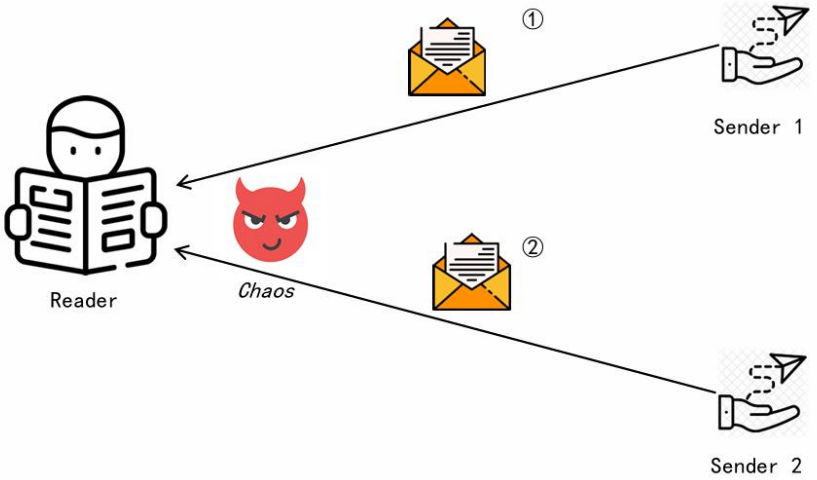
- Critical Section: A shared resource (printer, variable, file) that cannot be safely used by multiple threads at once.
- Race Condition: The outcome depends on the non-deterministic sequence of operations.
- Database Analogy:
  - The shared resource is a bank account balance.
  - Two concurrent transactions both read the balance, calculate a new one, and write it back.
  - Result: One transaction's update is lost!

Race Condition- Example



OS Solutions: Mutual Exclusion & Locks

- Mutual Exclusion (Mutex): Ensures only one thread executes a critical section at a time.
- Semaphores: A generalized counter for controlling access to a pool of N identical resources.
- These are the fundamental tools for preventing race conditions in OS and parallel programming.
- Database Preview:
  - Databases face the exact same problem with data items (rows, pages, tables).
  - They need their own version of locks and protocols. This is called Concurrency Control.



- Problems
  - Cross reading
  - Incomplete reading

The Bridge: From OS Concepts to Database Transactions

OS / Parallel Concept	Database Concept
Process / Thread	Transaction
Critical Section	Data Item (e.g., row)
Race Condition	Read-Write Conflict
Mutex / Lock	Data Lock
Synchronization Protocol	Concurrency Control

- In fact, database transactions are implemented using OS critical sections at the underlying level.
  - Hardware provides atomic instructions (such as CMPXCHG) -> The operating system uses them to implement low-level primitives (such as spinlocks) -> The operating system provides synchronization APIs to user space based on these low-level primitives (such as mutexes pthread_mutex, semaphores).
- You already understand the problem (race conditions) and the goal (correctness). In this lecture, we will learn the specific solutions and guarantees that databases provide. These guarantees are summarized by the ACID properties:
  - Atomicity, Consistency, Isolation, Durability.
  - We are now ready to dive into the world of database transactions!

Transaction Concept

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.
- 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 hardware failures and system crashes
  - Concurrent execution of multiple transactions

Example of Fund Transfer

- 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*)
- **Atomicity requirement**
  - If the transaction fails after step 3 and before step 6, money will be "lost" leading to an inconsistent database state
    - Failure could be due to software or hardware
  - The system should ensure that updates of a partially executed transaction are not reflected in the database
- **Durability requirement** — once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.

Example of Fund Transfer (Cont.)

- **Consistency requirement** in above example:
  - The sum of A and B is unchanged by the execution of the transaction
- In general, consistency requirements include
  - Explicitly specified integrity constraints such as primary keys and foreign keys
  - Implicit integrity constraints
    - e.g., sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand
  - A transaction must see a consistent database.
  - During transaction execution the database may be temporarily inconsistent.
  - When the transaction completes successfully the database must be consistent
    - Erroneous transaction logic can lead to inconsistency

Example of Fund Transfer (Cont.)

- **Isolation requirement** — if between steps 3 and 6, another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database (the sum $A + B$ will be less than it should be).

	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	

- Isolation can be ensured trivially by running transactions **serially**
  - That is, one after the other.
- However, executing multiple transactions concurrently has significant benefits, as we will see later.

ACID Properties

To preserve the integrity of data the database system must ensure:

- **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.
- **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
  - That is, for every pair of transactions T_i and T_j, it appears to T_i that either T_j, finished execution before T_i started, or T_j started execution after T_i finished.
- **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

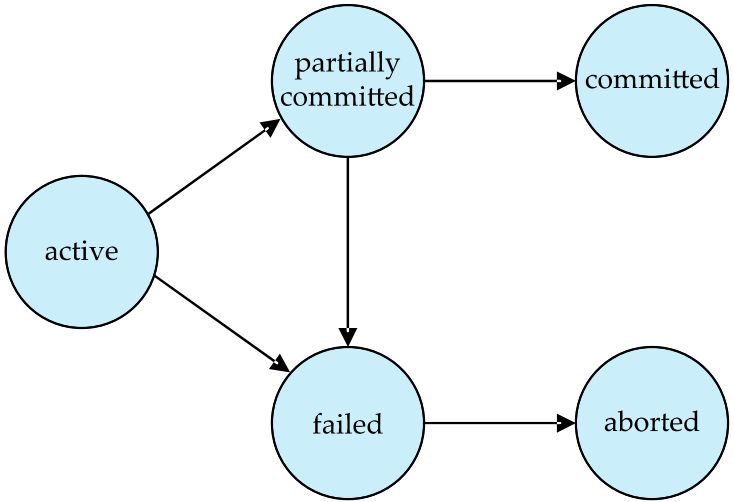
Comparison

Xi'an Jiaotong-Liverpool University 西交利物浦大学

Database Concept	Core Idea	OS / Parallel Computing Analogy	Key Insight
Transaction	A encapsulated unit of work.	A system call (e.g., fork()) or a critical section. It's a logical unit of execution.	It's the fundamental unit of work in the DBMS, grouping operations that must succeed or fail together.
Atomicity	"All or nothing": either all operations in the transaction happen, or none do.	A system call like fork(). It either fully succeeds (creates a new process) or fails completely (returns -1, no side effects).	Implemented via Write-Ahead Logging (WAL), similar to journaling in filesystems (e.g., ext4), where intent is logged before the actual change.
Isolation	Concurrent transactions must not interfere with each other; each should feel like it's running alone.	OS: Process isolation (separate memory spaces). A crash in one app doesn't corrupt another. Parallel Computing: Lock-free vs. lock-based data structures.	The key challenge: All transactions share the same "address space" (the database), requiring sophisticated concurrency control instead of simple memory separation.
Consistency	A transaction transitions the database from one valid state to another, preserving all defined rules (integrity constraints).	Parallel Computing: An invariant in a data structure (e.g., a linked list must remain connected after parallel updates).	The database must be consistent before and after a transaction. The transaction itself is responsible for maintaining consistency during execution.
Durability	Once a transaction is committed, its changes are permanent, even in case of system failures.	The OS call fsync(), which forces any buffered data to the permanent storage, ensuring data survives a power loss.	Implemented by forcing the transaction log (WAL) to stable storage upon commit. The actual data pages might be written later.

Transaction State

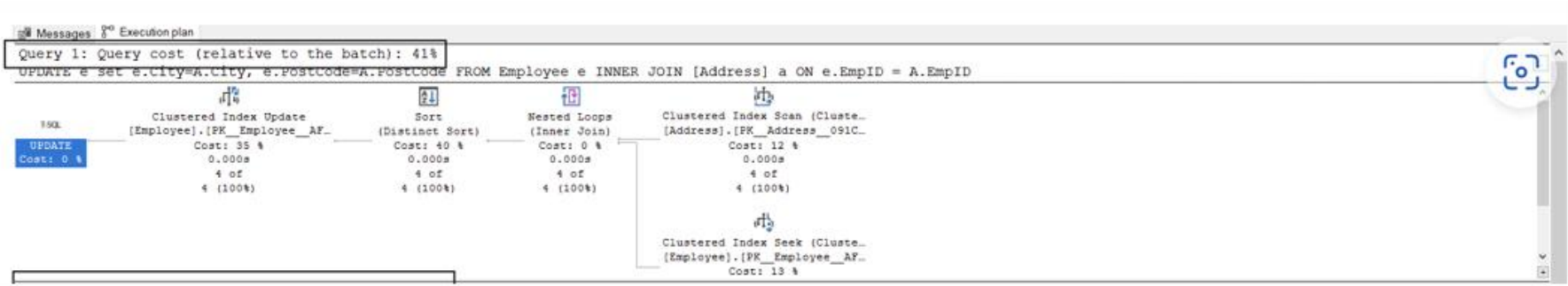
- **Active** – the initial state; the transaction stays in this state while it is executing
- **Partially committed** – after the final statement has been executed.
- **Failed** – after the discovery that normal execution can no longer proceed.
- **Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - Restart the transaction
    - Can be done only if no internal logical error
  - Kill the transaction
- **Committed** – after successful completion.



Why is Distinguishing These States Important?

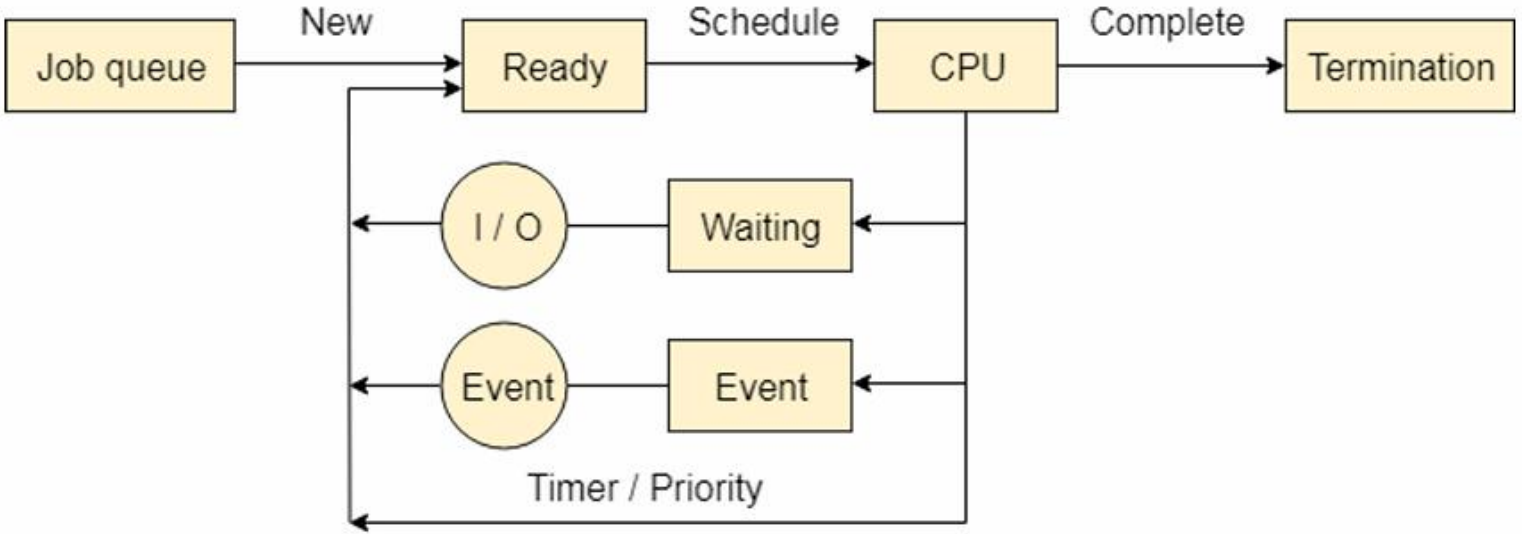
- To Ensure Atomicity:
  - The distinction between Active/Partially Committed and Committed is vital. The system must not commit a transaction if it is in a Failed state. The rollback process during the transition to Aborted ensures that a failed transaction leaves no partial effects on the database.
- To Enable Recovery:
  - After a system crash, the recovery manager examines the log. It identifies:
    - Transactions that were Committed but whose changes may not have reached the disk: It redoes them.
    - Transactions that were Active or Failed at the time of the crash: It undoes them.
  - Without clear state tracking, the recovery manager would not know which transactions to redo and which to undo, leading to database inconsistency.
- To Manage Concurrency:
  - The state determines when a transaction's locks can be released. Locks are typically held until the transaction is either Committed or Aborted. This prevents other transactions from reading uncommitted or inconsistent data, thereby enforcing Isolation.
- To Provide User Feedback:
  - The system can inform the user or application whether the transaction was successful (Committed) or not (Aborted). This is crucial for application logic.
- To Handle Transaction Termination:
  - The Aborted state provides two clear options, which is a crucial design point:
    - Restart the Transaction: The transaction can be restarted automatically by the system if the failure was not due to an internal logical error (e.g., it was aborted due to a deadlock). The system can ensure the new run is serializable with other transactions.
    - Kill the Transaction: If the failure was due to a logical error in the transaction's code, it must be killed and the error reported to the user/application for correction.

Example: Explain of a simple UPDATE



- It may contain multiple internal steps

Process Scheduling in OS



- In some aspects, simpler than DB

Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - **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.
- **Concurrency control schemes** – mechanisms to achieve isolation
  - That is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database
    - Will study in Chapter 15, after studying notion of correctness of concurrent executions.

Schedules

- **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 individual transaction.
- A transaction that successfully completes its execution will have a commit instructions as the last statement
  - By default transaction assumed to execute commit instruction as its last step
- A transaction that fails to successfully complete its execution will have an abort instruction as the last statement

Schedule 1

- 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 2

- A serial schedule where T_2 is followed by T_1

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

Schedule 3

- 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)	
$A := A - 50$	
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 4

- The following concurrent schedule does not preserve the value of ($A + B$).

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

Serializability

- **Basic Assumption** – Each transaction preserves database consistency.
- Thus, 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**

Simplified view of transactions

- We ignore operations other than **read** and **write** instructions
- We assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes.
- Our simplified schedules consist of only **read** and **write** instructions.

Conflicting Instructions

- 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

- Intuitively, a 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.

Conflict Serializability

- 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

Conflict Serializability (Cont.)

- Schedule 3 can be transformed into Schedule 6, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions. Therefore Schedule 3 is conflict serializable.

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

Schedule 3

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

Schedule 6

Conflict Serializability (Cont.)

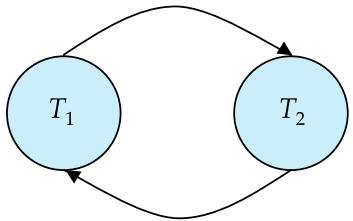
- Example of a schedule that is not conflict serializable:

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

- We are unable to swap instructions in the above schedule to obtain either the serial schedule < T_3, T_4 >, or the serial schedule < T_4, T_3 >.

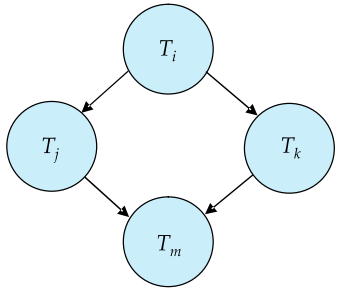
Testing for Serializability

- Consider some schedule of a set of transactions T_1, T_2, ..., T_n
- **Precedence graph** — a direct graph where the vertices are the transactions (names).
- 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.
- Example of a precedence graph

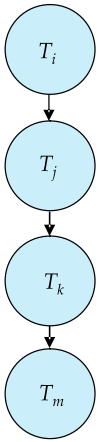


Test for Conflict Serializability

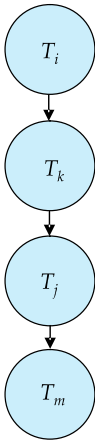
- A schedule is conflict serializable if and only if its precedence graph is acyclic.
- Cycle-detection algorithms exist which take order n^2 time, where n is the number of vertices in the graph.
  - (Better algorithms take order $n + e$ where e is the number of edges.)
- If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph.
  - This is a linear order consistent with the partial order of the graph.
  - For example, a serializability order for Schedule A would be $T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$
    - Are there others?



(a)



(b)



(c)

Recoverable Schedules

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)	

- If T_8 should abort, T_9 would have read (and possibly shown to the user) an inconsistent database state. Hence, database must ensure that schedules are recoverable.

Cascading Rollbacks

- **Cascading rollback** – a single transaction 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

- **Cascadeless schedules** — cascading rollbacks cannot occur;
  - 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
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
  - Are serial schedules recoverable/cascadeless?
- Testing a schedule for serializability *after* it has executed is a little too late!
- **Goal** – to develop concurrency control protocols that will assure serializability.

Concurrency Control (Cont.)

- Schedules must be conflict serializable, and recoverable, for the sake of database consistency, and preferably cascadeless.
- A policy in which only one transaction can execute at a time generates serial schedules, but 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.

Concurrency Control vs. Serializability Tests

- Concurrency-control protocols allow concurrent schedules, but ensure that the schedules are conflict/view serializable, and are recoverable and cascadeless .
- Concurrency control protocols (generally) do not examine the precedence graph as it is being created
  - Instead a protocol imposes a discipline that avoids non-serializable schedules.
  - We study such protocols in Chapter 16.
- Different concurrency control protocols provide different tradeoffs between the amount of concurrency they allow and the amount of overhead that they incur.
- Tests for serializability help us understand why a concurrency control protocol is correct.

Question

- What is the relationship between Recoverable Schedules and Collision Serializability?
  - Independent but need to coexist

Property	The "Evil" it Prevents	Core Focus
Conflict Serializability	Incorrect Final Outcome	Even if all transactions commit successfully, the final database state could be wrong.
Recoverability	Unrecoverability after Commit	The failure of one transaction can cause already-committed transactions to be based on invalid data.

Question

- A schedule can be Conflict-Serializable but NOT Recoverable.

```
T1          T2
---         ---
write(A)
            read(A)  // T2 read A written by T1
            commit   // **Danger! T2 commits before T1**
commit
```

- Analysis:
  - Conflict-Serializable? Yes. This schedule is conflict-equivalent to the serial schedule <T1, T2>. If the system runs normally, the final state is correct.
  - Recoverable? No! If the system crashes after T2 commits but before T1 commits, T1 must be rolled back. However, T2 has already committed, and it read a value of A from T1 (a value that subsequently vanished with T1's rollback). This results in a committed transaction (T2) being based on a state that never officially existed, permanently corrupting the database's consistency.

Question

- A schedule can be Recoverable but NOT Conflict-Serializable.

```
T1          T2
---         ---
read(A)
            write(A)
write(A)
commit
            commit
```

- Analysis:
  - Conflict-Serializable? No. The precedence graph has a cycle (T1 -> T2 -> T1), so it is not equivalent to any serial schedule. The final outcome might be incorrect.
  - Recoverable? Yes. Although the result might be wrong, from a recovery perspective, this schedule does not violate the commit-order rule for read-after-write dependencies. Here, T2 writes A, but T1 did not read the A written by T2 (it read the initial A). Thus, there is no "read-dependency" to enforce. Therefore, the abort of either transaction would not cause the other committed transaction to have read its dirty data.

Weak Levels of Consistency

- Some applications are willing to live with weak levels of consistency, 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)

Question

- If it's not Serializable, does that break conflict serializability?
  - If the isolation level is set to a level lower than Serializable (such as Repeatable Read, Read Committed, or Read Uncommitted), then the system allows for non-conflicting serializable scheduling, thus "breaking" conflict-serializable.

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

Transactions as SQL Statements

- E.g., Transaction 1:
  select *ID, name* **from** *instructor* **where** *salary* > 90000
- E.g., Transaction 2:
  insert into *instructor* **values** ('11111', 'James', 'Marketing', 100000)
- Suppose
  - T1 starts, finds tuples salary > 90000 using index and locks them
  - And then T2 executes.
  - Do T1 and T2 conflict? Does tuple level locking detect the conflict?
  - Instance of the **phantom phenomenon**
- Also consider T3 below, with Wu's salary = 90000
  update *instructor*
  set *salary* = *salary* * 1.1
  where *name* = 'Wu'
- Key idea: Detect "**predicate**" conflicts, and use some form of "**predicate locking**"

Reading&Demo

- https://www.thenile.dev/blog/transaction-isolation-postgres

1-3