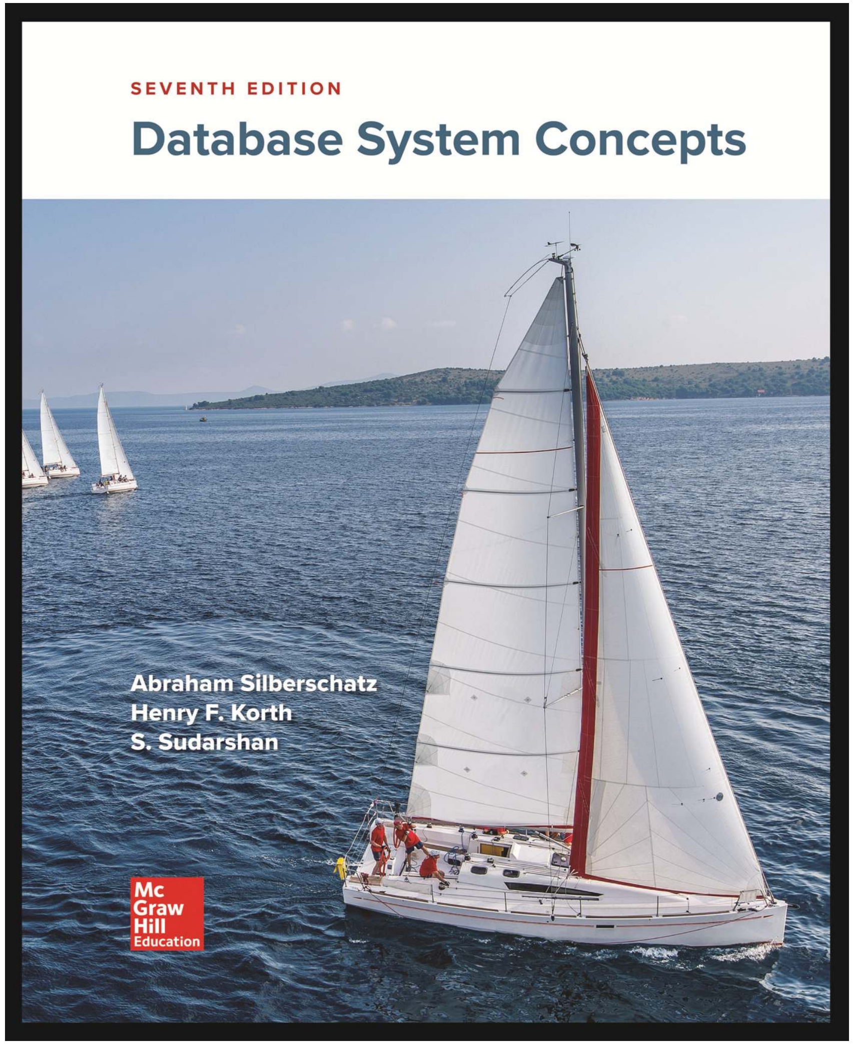
IF3140 – Sistem Basis Data
Transactions

SEMESTER I TAHUN AJARAN 2024/2025

KNOWLEDGE & SOFTWARE ENGINEERING



Sumber

Silberschatz, Korth, Sudarshan: "Database System Concepts", 7th Edition

- Chapter 17: Transactions

KNOWLEDGE & SOFTWARE ENGINEERING

Objectives

Students are able to:

- Explain the importance of transaction properties
- Explain serializable transactions
- Explain the concept of implicit commits
- Describe the issues specific to efficient transaction execution
- Explain at least two transaction protocols

Outline

- Transaction Concept
- Transaction State
- Concurrent Executions
- Serializability
- Recoverability
- Implementation of Isolation
- Transaction Definition in SQL
- Testing for Serializability

KNOWLEDGE & SOFTWARE ENGINEERING

Transaction Concept

A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.

Example

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

Main issues

- Failures of various kinds
- Concurrent execution of multiple transactions

KNOWLEDGE & SOFTWARE ENGINEERING

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.

KNOWLEDGE & SOFTWARE ENGINEERING

Example of Fund Transfer (Cont.)

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*)

- **Consistency requirement** in 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.)

T_1	T_2
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 requirement — if between steps 3 and 6, another transaction T_2 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).

- 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.

KNOWLEDGE & SOFTWARE ENGINEERING

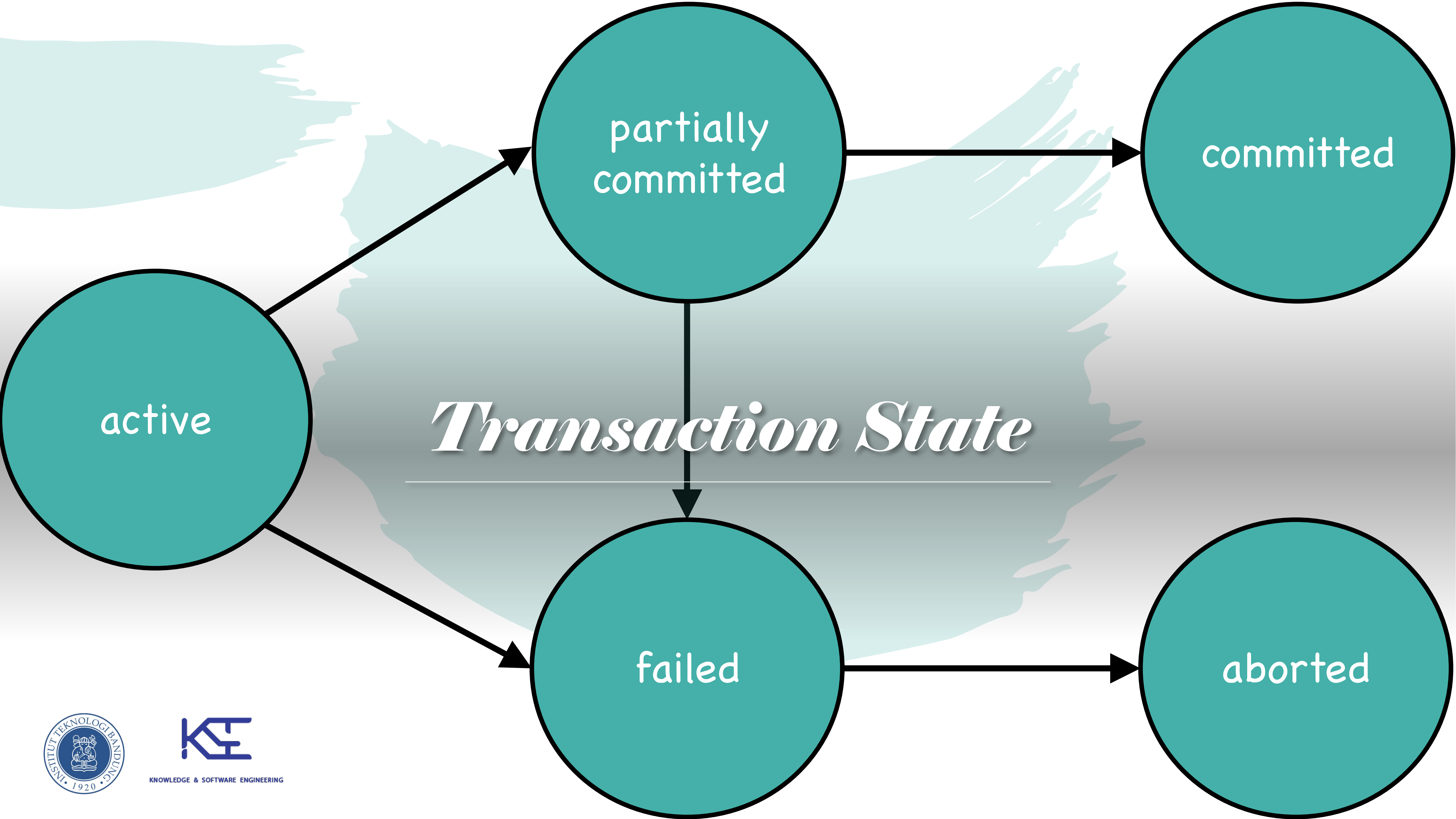
ACID Properties

A **transaction** is a unit of program execution that accesses and possibly updates various data items. 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.

KNOWLEDGE & SOFTWARE ENGINEERING

Transaction State



Concurrent Executions

- Advantages are
  - **increased processor and disk utilization**, leading to better transaction throughput
  - **reduced average response time for transactions**
- **Concurrency control schemes** – mechanisms to achieve isolation

KNOWLEDGE & SOFTWARE ENGINEERING

Schedules

Schedule – a sequences of instructions that specify the **chronological order** in which instructions of concurrent transactions are executed

- All instructions
- Preserve the order of instructions in each transaction

A transaction that successfully completes its execution will have a commit instructions as the last statement

A transaction that fails to successfully complete its execution will have an abort instruction as the last statement

KNOWLEDGE & SOFTWARE ENGINEERING

Schedule (Serial)

Let *T1* transfer $50 from A to B, and
T2 transfer 10% of the balance from A to B.

Schedule 1

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

KNOWLEDGE & SOFTWARE ENGINEERING

Schedule (Serial)

Let *T1* transfer $50 from A to B, and
T2 transfer 10% of the balance from A to B.

Schedule 1

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

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

The following schedule is not a serial schedule, but it is equivalent to Schedule 1.

Let *T1* transfer $50 from A to B, and
T2 transfer 10% of the balance from A to B.

Schedule 1

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 3

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

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

Let *T1* transfer $50 from A to B, and
T2 transfer 10% of the balance from A to B.

Schedule 4

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:
  - **conflict serializability**
  - **view serializability**

KNOWLEDGE & SOFTWARE ENGINEERING

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 l_i and l_j of transactions T_i and T_j respectively, **conflict** if and only if there exists some item Q accessed by both l_i and l_j, and at least one of these instructions wrote Q.
- Intuitively, a conflict between l_i and l_j forces a (logical) temporal order between them.

T1	T2
read (Q)	
	read (Q)

≡

T1	T2
	read (Q)
read (Q)	

T1	T2
read (Q)	
	write (Q)

≢

T1	T2
	write (Q)
read (Q)	

T1	T2
write (Q)	
	read (Q)

≢

T1	T2
	read (Q)
write (Q)	

T1	T2
write (Q)	
	write (Q)

≢

T1	T2
	write (Q)
write (Q)	

Conflict Serializability

conflict equivalent schedules	conflict serializable schedule
schedule *S* can be transformed into schedule *S´* by a series of swaps of non-conflicting instructions	a schedule *S* that is conflict equivalent to a serial schedule

KNOWLEDGE & SOFTWARE ENGINEERING

Conflict Serializability – Example I

[conflict] serializable

Schedule 3

T1	T2
read (A)	
write (A)	
	read (A)
	write (A)
read (B)	
write (B)	
	read (B)
	write (B)

→

Schedule 1

T1	T2
read (A)	
write (A)	
read (B)	
write (B)	
	read (A)
	write (A)
	read (B)
	write (B)

KNOWLEDGE & SOFTWARE ENGINEERING

Conflict Serializability – Example 2

Not conflict serializable

Schedule 13

T1	T2
read (A)	
	write (A)
write (A)	

≠

Schedule 11

T1	T2
read (A)	
write (A)	
	write (A)

≠

Schedule 12

T1	T2
	write (A)
read (A)	
write (A)	

KNOWLEDGE & SOFTWARE ENGINEERING

View Serializability

Let S and $S´$ be two schedules with the same set of transactions.

S and $S´$ are **view equivalent** if the following three conditions are met, for each data item Q,

- If in schedule S transaction T_i reads the initial value of Q, then in schedule S' transaction T_i must also read the initial value of Q.
- If in schedule S transaction T_i reads the value of Q that was produced by transaction T_j, then in schedule S' transaction T_i must also read the value of Q that was produced by the same **write**(Q) operation of transaction T_j.
- 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 serializable schedule: a schedule S that is view equivalent to a serial schedule.

View Serializability – Example

view serializable

Schedule 22

T1	T2	T3
read (Q)		
	write (Q)	
write (Q)		
		write (Q)

≡

Schedule 21

T1	T2	T3
read (Q)		
write (Q)		
	write (Q)	
		write (Q)

KNOWLEDGE & SOFTWARE ENGINEERING

T_1	T_5
read (A) $A := A - 50$ write (A)	
	read (B) $B := B - 10$ write (B)
read (B) $B := B + 50$ write (B)	
	read (A) $A := A + 10$ write (A)

Other Notions of Serializability

Determining such equivalence requires analysis of operations other than read and write

Testing for Serializability

Precedence Graph

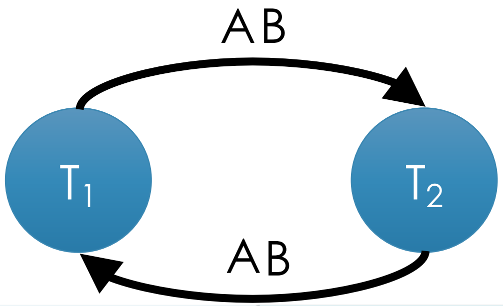
- A directed graph
- The vertices are the transactions (names).
- An arc from T_i to T_j = the two transactions conflict, and T_i accessed earlier.
- May label the arc by the item that was accessed.

KNOWLEDGE & SOFTWARE ENGINEERING

Testing for Serializability

Precedence Graph

- A directed graph
- The vertices are the transactions (names).
- An arc from T_i to T_j = the two transactions conflict, and T_i accessed earlier.
- May label the arc by the item that was accessed.

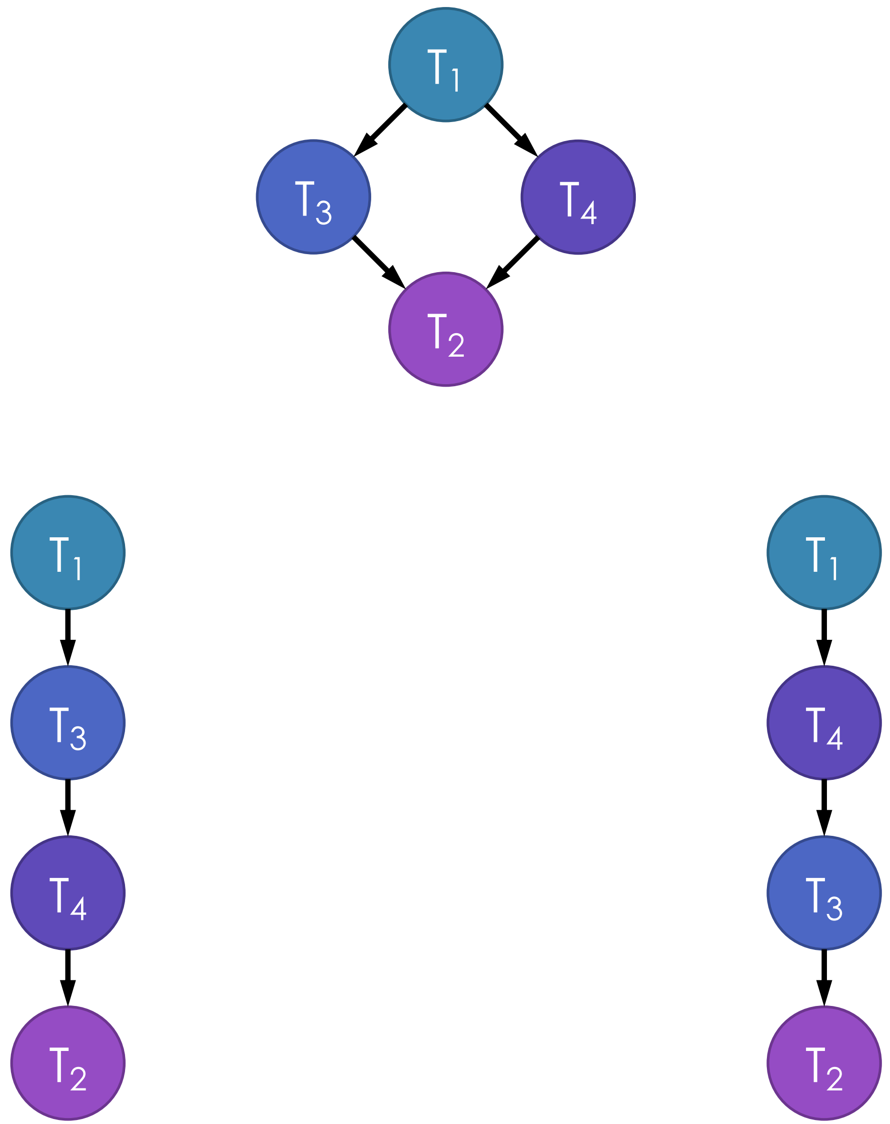


T_1	T_2
read (A)	
$A := A - 50$	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	$B := B + temp$
	write (B)
	commit

Test for Conflict Serializability

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.



Test for View Serializability?

- The precedence graph test for conflict serializability cannot be used directly to test for view serializability.
  - Extension to test for view serializability has cost exponential in the size of the precedence graph.

KNOWLEDGE & SOFTWARE ENGINEERING

Recoverable Schedules

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.

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

Cascadeless Schedules

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

- **Cascading rollback**
  a single transaction failure leads to a series of transaction rollbacks.

- **Cascadeless Schedules**: no possibility of cascading rollback
  - 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

Concurrency Control

- A database must provide a mechanism that will ensure
  - either conflict or view serializable
  - recoverable and preferably cascadeless
- Only one transaction can be executed at a time → serial schedules → a poor degree of concurrency
- Testing a schedule for serializability <u>*after*</u> it has executed is <u>a little too late</u>!
- **Goal** – to develop concurrency control protocols that will assure serializability.

KNOWLEDGE & SOFTWARE ENGINEERING

Weak Levels of Consistency

TRADEOFF ACCURACY FOR PERFORMANCE

Levels of Consistency in SQL-92

- **Serializable** — default
- **Repeatable read** — repeated reads must return same value.
- **Read committed** — only committed records can be read.
- **Read uncommitted** — even uncommitted records may be read.

KNOWLEDGE & SOFTWARE ENGINEERING

Read Phenomena

- The ANSI/ISO standard SQL 92 refers to three different *read phenomena* when Transaction 1 reads data that Transaction 2 might have changed:
  - Dirty Reads
  - Non-repeatable reads
  - Phantom Reads
- Suppose we have the following data:

users

id	name	age
1	Joe	20
2	Jill	25

KNOWLEDGE & SOFTWARE ENGINEERING

Dirty Reads

- aka. Uncommitted dependency: transaction is allowed to read data from a row that has been modified by another running transaction and not yet committed.
- Level of consistency **read uncommitted** allows dirty reads

Transaction 1	Transaction 2
`/* Query 1 */` `SELECT age FROM users WHERE id = 1;` `/* will read 20 */`	
	`/* Query 2 */` `UPDATE users SET age = 21 WHERE id = 1;` `/* No commit here */`
`/* Query 1 */` `SELECT age FROM users WHERE id = 1;` `/* will read 21 */`	
	`ROLLBACK; /* lock-based DIRTY READ */`

Non-Repeatable Reads

- During the course of a transaction, a row is retrieved twice and the values within the row differ between reads
- Level of consistency **read committed** allows non-repeatable reads

Transaction 1	Transaction 2
`/* Query 1 */` `SELECT * FROM users WHERE id = 1;`	

	`/* Query 2 */`
	`UPDATE users SET age = 21 WHERE id = 1;`
	`COMMIT; /* in multiversion concurrency control, or lock-based READ COMMITTED */`

`/* Query 1 */`
`SELECT * FROM users WHERE id = 1;`
`COMMIT; /* lock-based REPEATABLE READ */`	

KNOWLEDGE & SOFTWARE ENGINEERING

Phantom Reads

- In the course of a transaction, new rows are added by another transaction to the records being read
- Level of consistency **repeatable-read** allows phantom reads

Transaction 1

```
/* Query 1 */
SELECT * FROM users
WHERE age BETWEEN 10 AND 30;
```

Transaction 2

```
/* Query 2 */
INSERT INTO users(id,name,age) VALUES ( 3, 'Bob', 27 );
COMMIT;
```

Transaction 1

```
/* Query 1 */
SELECT * FROM users
WHERE age BETWEEN 10 AND 30;
COMMIT;
```

KNOWLEDGE & SOFTWARE ENGINEERING

Transaction Definition in SQL

- A transaction begins implicitly
- A transaction ends by:
    - **Commit work**
    - **Rollback work**
- Every SQL statement commits implicitly
- Isolation level can be set at database level
- Isolation level can be changed at start of transaction

KNOWLEDGE & SOFTWARE ENGINEERING

Implementation of Isolation Levels

- **Locking**
  - Granularity?
  - How long?
  - Shared vs exclusive locks
- **Timestamps**
  - Transaction timestamp assigned
  - Read timestamp and Write timestamp
  - Timestamps: detect out of order accesses
- **Multiple versions**
  - Several snapshots of data item
  - Read from a snapshot

Transactions as SQL Statements

Transaction 1	**select** *ID, name* **from** *instructor* **where** *salary* > 90000
Transaction 2	**insert into** *instructor* **values** ('11111', 'James', 'Marketing', 100000)
Transaction 3 (with Wu's salary = 90000)	**update** *instructor* **set** *salary* = *salary* * 1.1 **where** *name* = 'Wu'

- Key idea: Detect "**predicate**" conflicts → "**predicate locking**"

KNOWLEDGE & SOFTWARE ENGINEERING

End of Chapter

KNOWLEDGE & SOFTWARE ENGINEERING