ADVANCED DATABASE

Transaction and Concurrency

Dr. NGUYEN Hoang Ha

Email: nguyen-hoang.ha@usth.edu.vn

Objectives

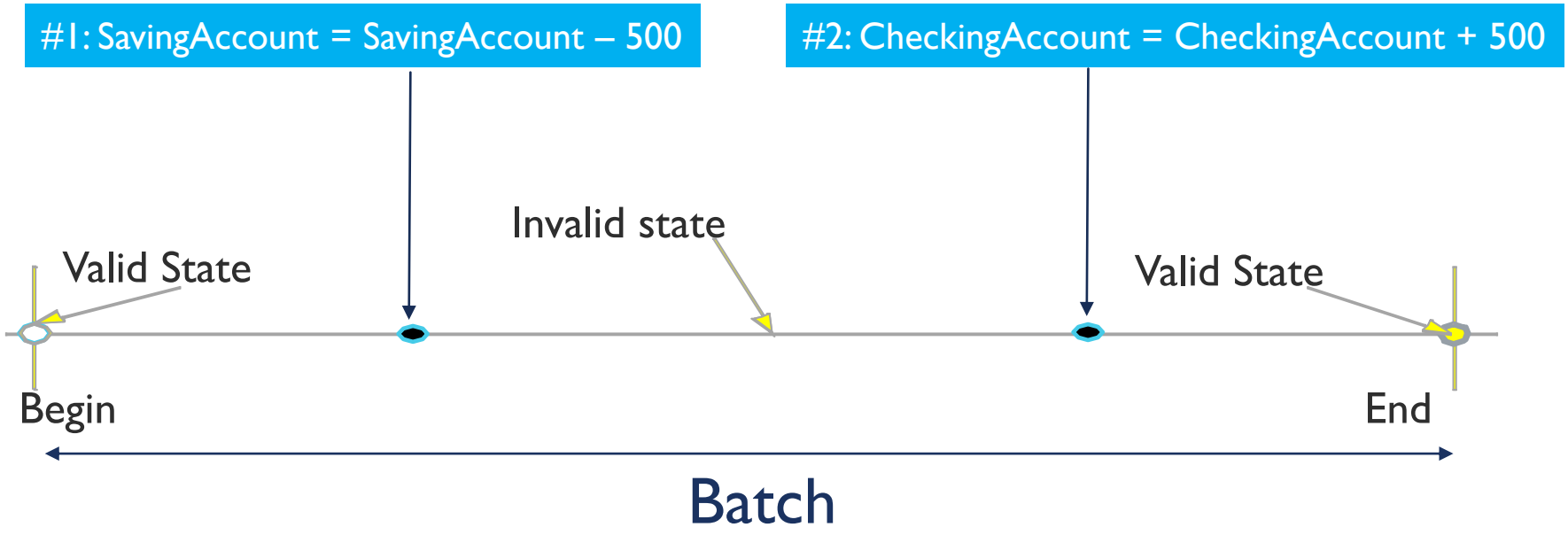
- Understand transaction
  - Concepts
  - Use isolation level to avoid concurrency problems
  - Redo and undo
- Know how to schedule tasks of multi transactions
- Able to implement Transaction in SQL Server

Agenda

- Transaction concepts & properties
- Transaction concurrency problems
- Isolation levels
- Lock
- Log and recovery
- Serializability checking

System failure in the batch middle

- Transfer 500$ between 2 accounts:



- The system crashes after step #1 → What happens?

Transaction

- Definition: a sequence of tasks executed as a whole, taking a consistent database state into another consistent database state
- Purpose: ensuring integrity and consistency of data
- Properties: ACID
  - Atomicity
    - All or none
  - Consistency
    - Leave data in a consistent state
    - Not accept tasks that's ½ done
  - Isolation
    - Like that transaction runs alone
  - Durability
    - Modifications are permanent

Atomicity and Durability: examples

ATOMICITY

Begin

SavingAcc= SavingAcc - 300

CheckingAcc= CheckingAcc + 300

Commit T1

Crash

➔ Stop all actions !!

DURABILITY

Begin

SavingAcc = ChekingAcc - 300

CheckingAcc = CheckingAcc + 300

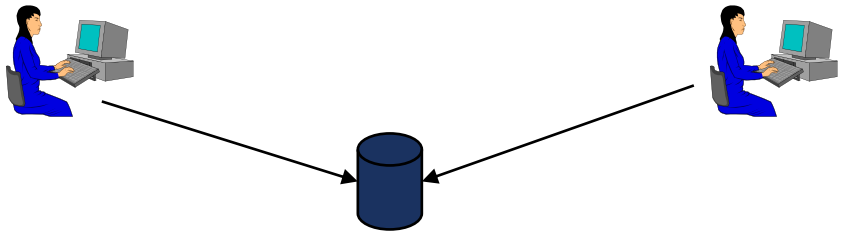
Commit T1

Crash

➔ be sure that the current account has been update !

Isolation

- User transactions are executed as if they were the only connected users to the database.
- Concurrency Control
  - Most (all?) DBMSs are multi-user systems.
  - The concurrent execution of many different transactions submitted by various users must be organized such that each transaction does not interfere with another transactions with one another in a way that produces incorrect results.
  - The concurrent execution of transactions must be such that each transaction appears to execute in isolation.



Agenda

- Transaction concepts & properties
- Transaction concurrency problems
- Isolation levels
- Lock
- Log and recovery
- Serializability checking

Example: Interacting Processes

- Assume the usual Sells(bar,beer,price) relation, and suppose that Joe's Bar sells only Bud for $2.50 and Miller for $3.00.
- Sally is querying Sells for the highest and lowest price Joe charges.
- Joe decides to stop selling Bud and Miller, but to sell only Heineken at $3.50.

Sally's Program

- Sally executes the following 02 SQL statements called (min) and (max) to help us remember what they do.

(max) SELECT MAX(price) FROM Sells

WHERE bar = 'Joe"s Bar';

(min) SELECT MIN(price) FROM Sells

WHERE bar = 'Joe"s Bar';

Joe's Program

- At about the same time, Joe executes the following steps: (del) and (ins).

(del) DELETE FROM Sells

WHERE bar = 'Joe"s Bar';

(ins) INSERT INTO Sells

VALUES('Joe"s Bar', 'Heineken', 3.50);

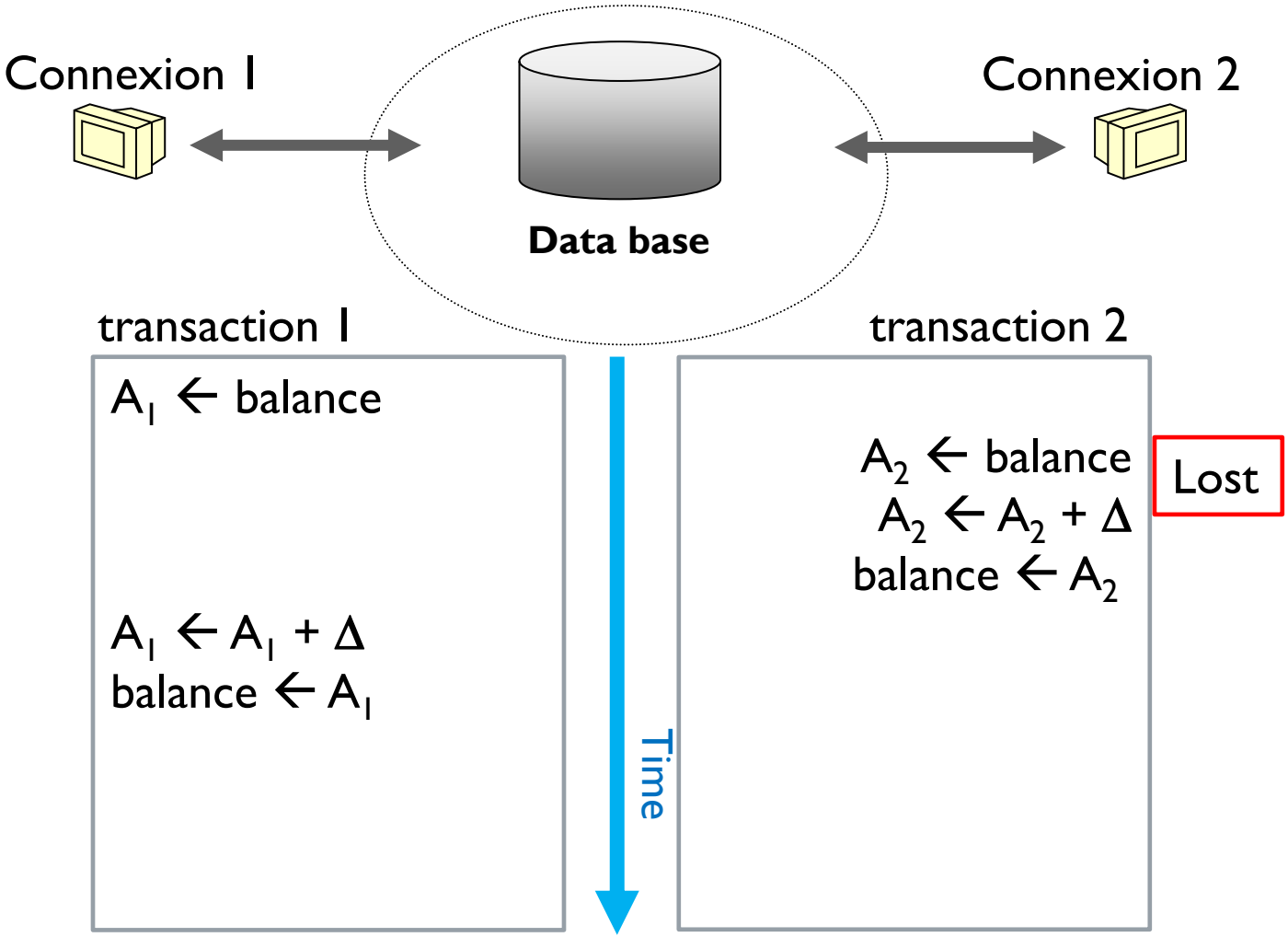
Strange Interleaving

- Although (max) must come before (min), and (del) must come before (ins), there are no other constraints on the order of these statements, unless we group statements into transactions.
- Suppose the steps execute in the order (max)(del)(ins)(min).

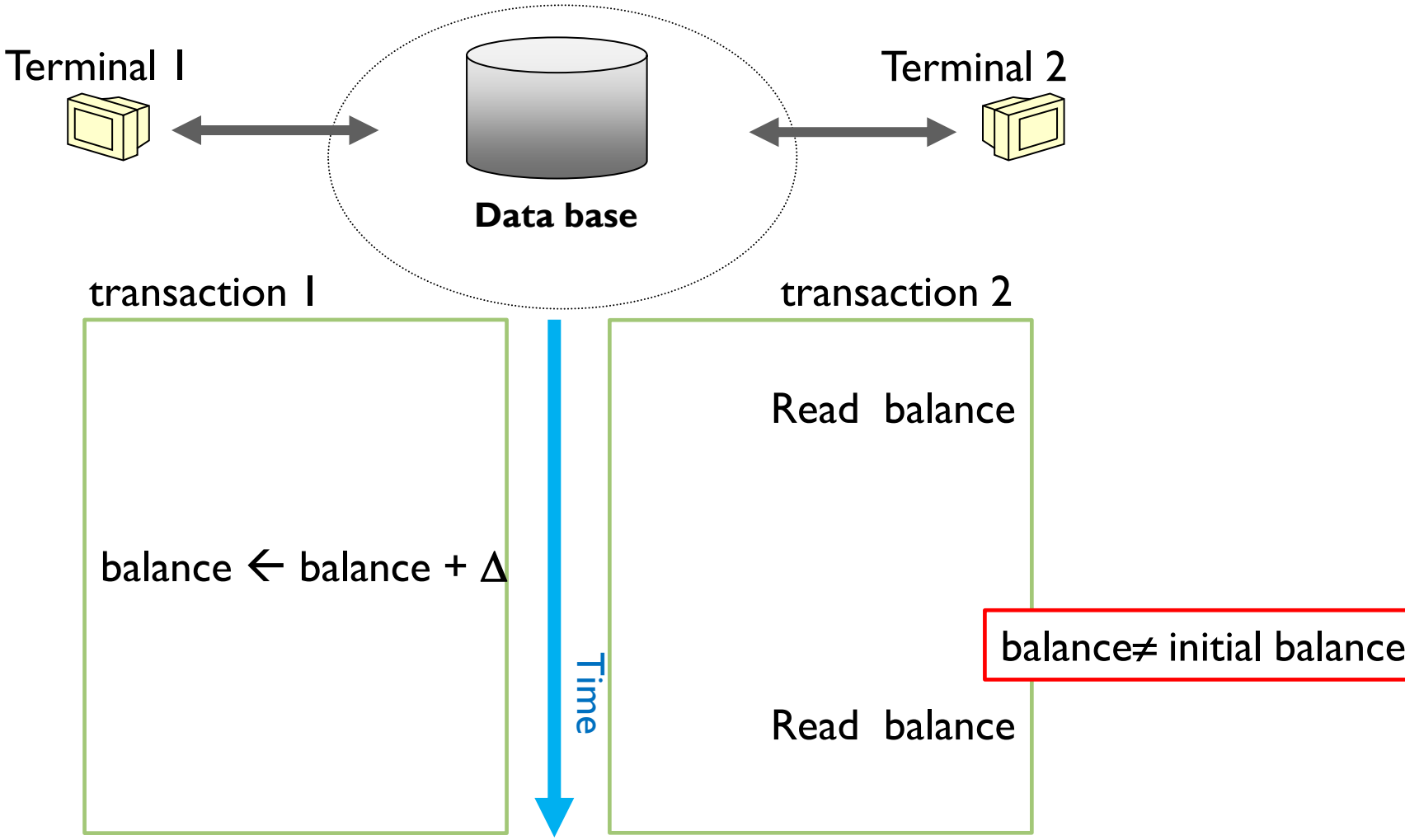
Joe's Prices:	2.50, 3.00		2.50, 3.00	3.50
Statement:				
Result:	(max)	(del)	(ins)	(min)
	3.00			3.50

- Sally sees MAX < MIN!

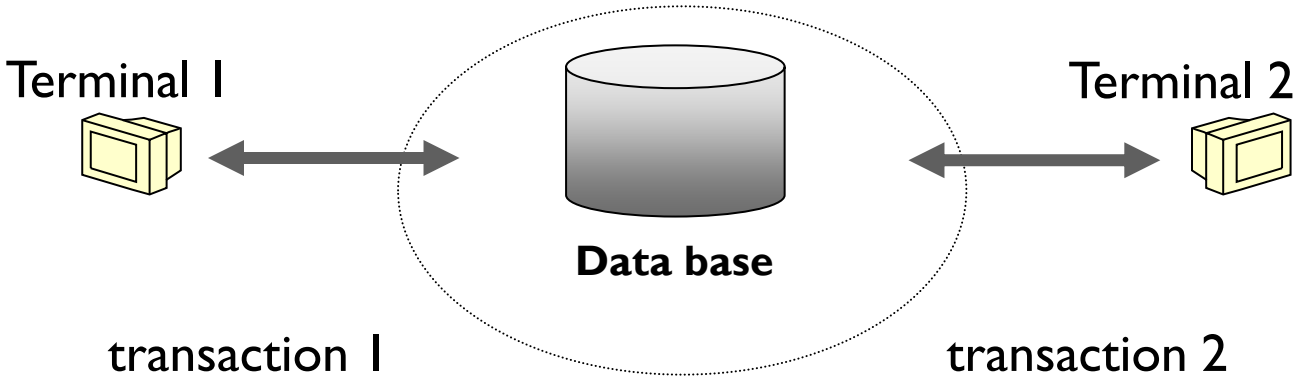
Problem 1: lost update



Problem 2: non-repeatable read



Problem 3: Phantom read



transaction 1

balance = 300

UPDATE Account
SET BALANCE = 400
WHERE …

balance = 400

Time

transaction 2

SELECT COUNT (*)
FROM Account
WHERE balance= 400

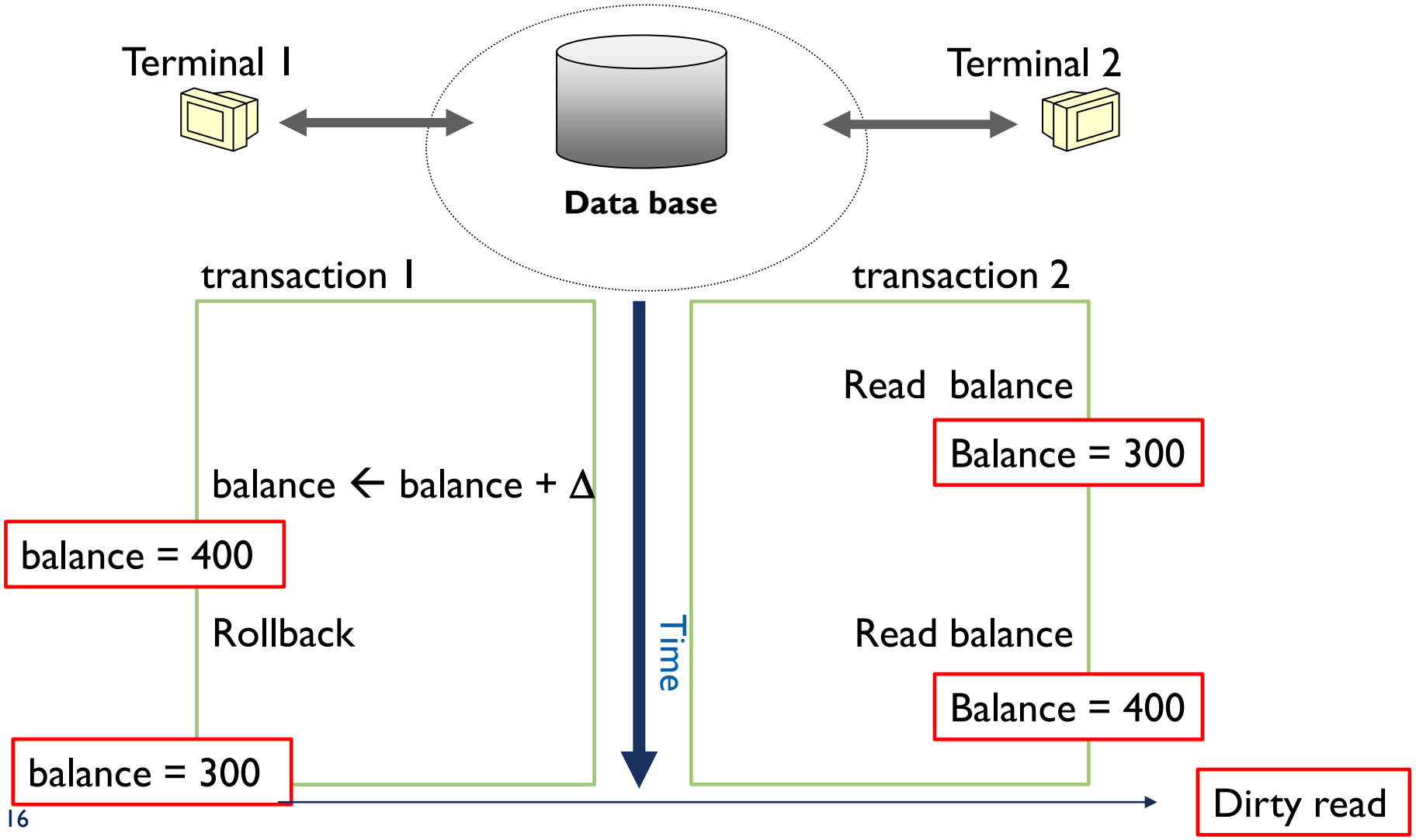
Results = N

SELECT COUNT (*)
FROM Account
WHERE balance= 400

Results = N+ Δ

Phantom Read

Problem 4: dirty read



Agenda

- Transaction concepts & properties
- Transaction concurrency problems
- Isolation levels
- Lock
- Serializability checking
- Log and recovery

Isolation levels

- The level of isolation, or the height of the fence between transaction, can be adjusted to control which transactional faults are permitted.
- Isolation level is personal choice
  - Example:
    - Joe Runs serializable, but Sally doesn't → Sally might see no prices for Joe's Bar.
    - i.e., it looks to Sally as if she ran in the middle of Joe's transaction.

Isolation levels

- **Read uncommitted**
  - This is the lowest isolation level. In this level, dirty reads are allowed (see below), so one transaction may see not-yet-committed changes made by other transactions
- **Read committed**
  - In this isolation level, a lock-based concurrency control DBMS implementation keeps write locks (acquired on selected data) until the end of the transaction, but read locks are released as soon as the SELECT operation is performed
- **Repeatable reads**
  - In this isolation level, a lock-based concurrency control DBMS implementation keeps read and write locks (acquired on selected data) until the end of the transaction.
- **Serializable**
  - This is the *highest* isolation level.
  - With a lock-based concurrency control DBMS implementation, serializability requires read and write locks (acquired on selected data) to be released at the end of the transaction, including range locks..

Isolation levels with concurrency problems

Isolation Level	*Dirty Read* Seeing another transaction's non-committed changes	*Non-Repeatable Read* Seeing another transaction's committed changes	*Phantom Row* Seeing rows selected by `where` clause change as a result of another transaction
Read Uncommitted ***(least restrictive)***	Possible	Possible	Possible
Read Committed ***(SQL Server default; moderately restrictive)***	Prevented	Possible	Possible
Repeatable Read	Prevented	Prevented	Possible
Serializable (most restrictive)	Prevented	Prevented	Prevented

With SQL Server

```
SET TRANSACTION ISOLATION LEVEL
    { READ UNCOMMITTED
    | READ COMMITTED
    | REPEATABLE READ
    | SNAPSHOT
    | SERIALIZABLE
    }
```

E.g.:

```
USE AdventureWorks2012;
GO
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
GO
```

Agenda

- Transaction concepts & properties
- Transaction concurrency problems
- Isolation levels
- Lock
- Log and recovery
- Serializability checking

Locks

- DBMSs implement the isolation property with locks, that protect a transaction's rows from being affected by another transaction.
- Every lock has the following 3-properties:
  - Granularity – the size of the lock
  - Mode – the type of the lock
  - Duration – the isolation mode of the lock

Lock Granularity

- Indicates the level/scope of lock use

Resource	Description
RID	A row identifier used to lock a single row within a heap.
KEY	A row lock within an index used to protect key ranges in serializable transactions.
PAGE	An 8-kilobyte (KB) page in a database, such as data or index pages.
EXTENT	A contiguous group of eight pages, such as data or index pages.
HoBT	A heap or B-tree. A lock protecting a B-tree (index) or the heap data pages in a table that does not have a clustered index.
TABLE	The entire table, including all data and indexes.
FILE	A database file.
APPLICATION	An application-specified resource.
METADATA	Metadata locks.
ALLOCATION_UNIT	An allocation unit.
DATABASE	The entire database.

A Database-Level Locking Sequence

- Good for batch processing
- T1 and T2 can not access the same database concurrently even if they use different tables
- Unsuitable for online multi-user DBMSs

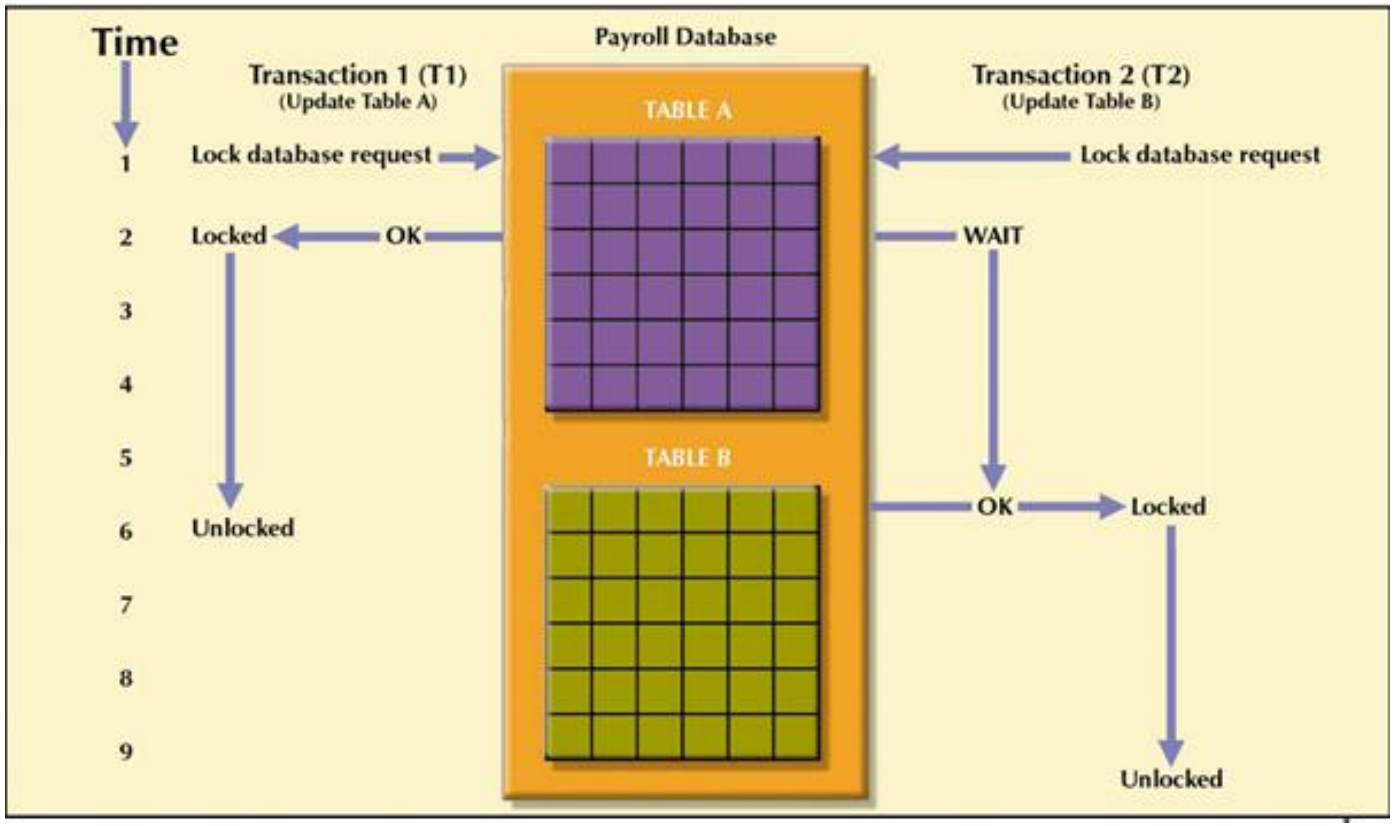
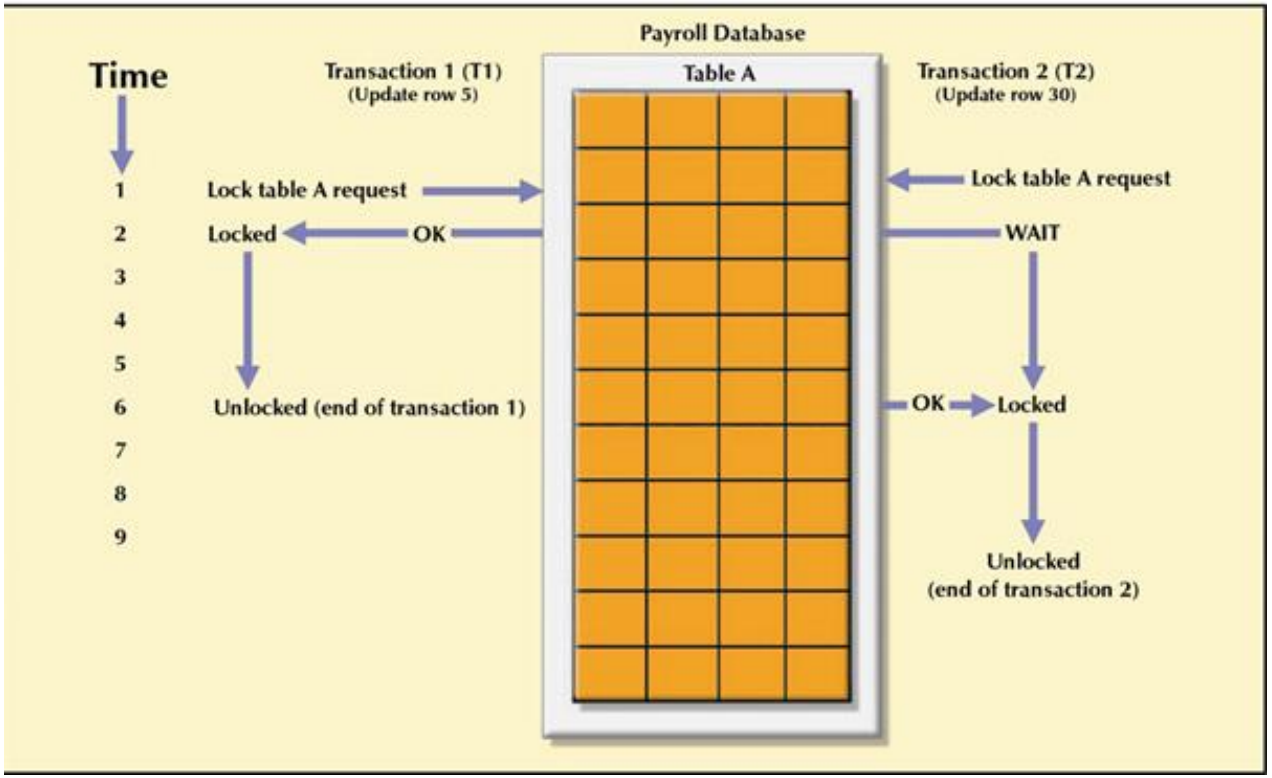


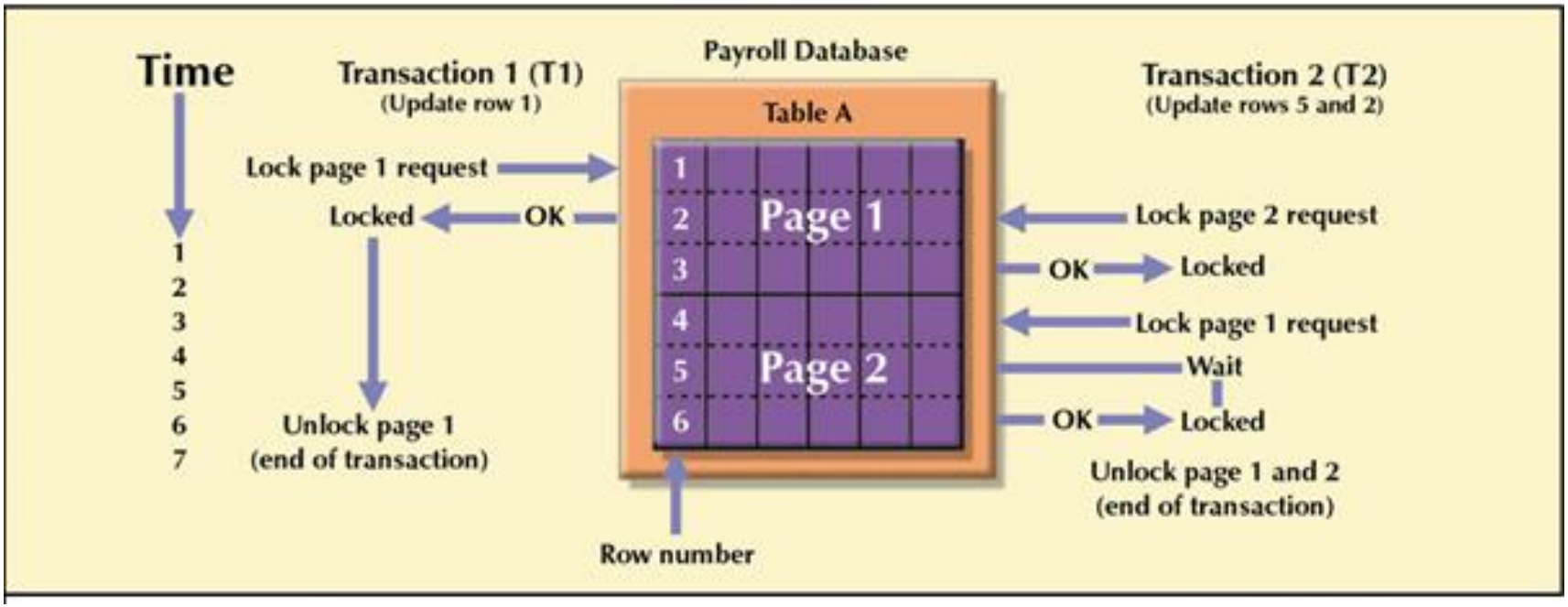
Table-Level Lock

- T1 and T2 can access the same database concurrently as long as they use different tables
- Can cause bottlenecks when many transactions are trying to access the same table
- Not suitable for multi-user DBMSs



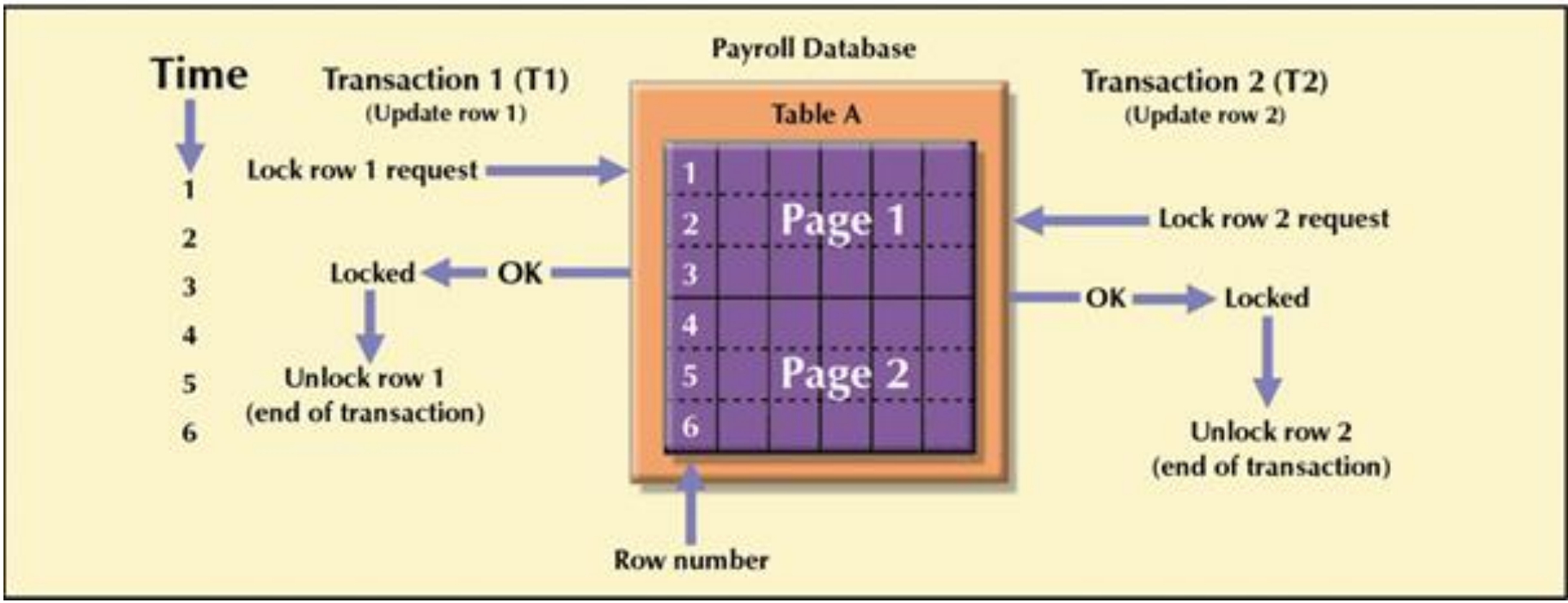
Page-Level Lock

- A table spans several pages. Each page contains several rows of one or more tables
- An entire disk page is locked
- Most frequently used multi-user DBMS locking method



Row-Level Lock

- Concurrent transactions can access different rows of the same table even if the rows are located on the same page
- Improves data availability but with high overhead (each row has a lock that must be read and written to)



Lock modes: Shared/Exclusive

- Exclusive lock
  - Access is specifically reserved for the transaction that locked the object
  - Must be used when the potential for conflict exists – when a transaction wants to update a data item and no locks are currently held on that data item by another transaction
  - *Granted if and only if no other locks are held on the data item*
- Shared lock
  - Concurrent transactions are granted Read access on the basis of a common lock
  - Issued when a transaction wants to read data and no exclusive lock is held on that data item
  - Multiple transactions can each have a shared lock on the same data item if they are all just reading it

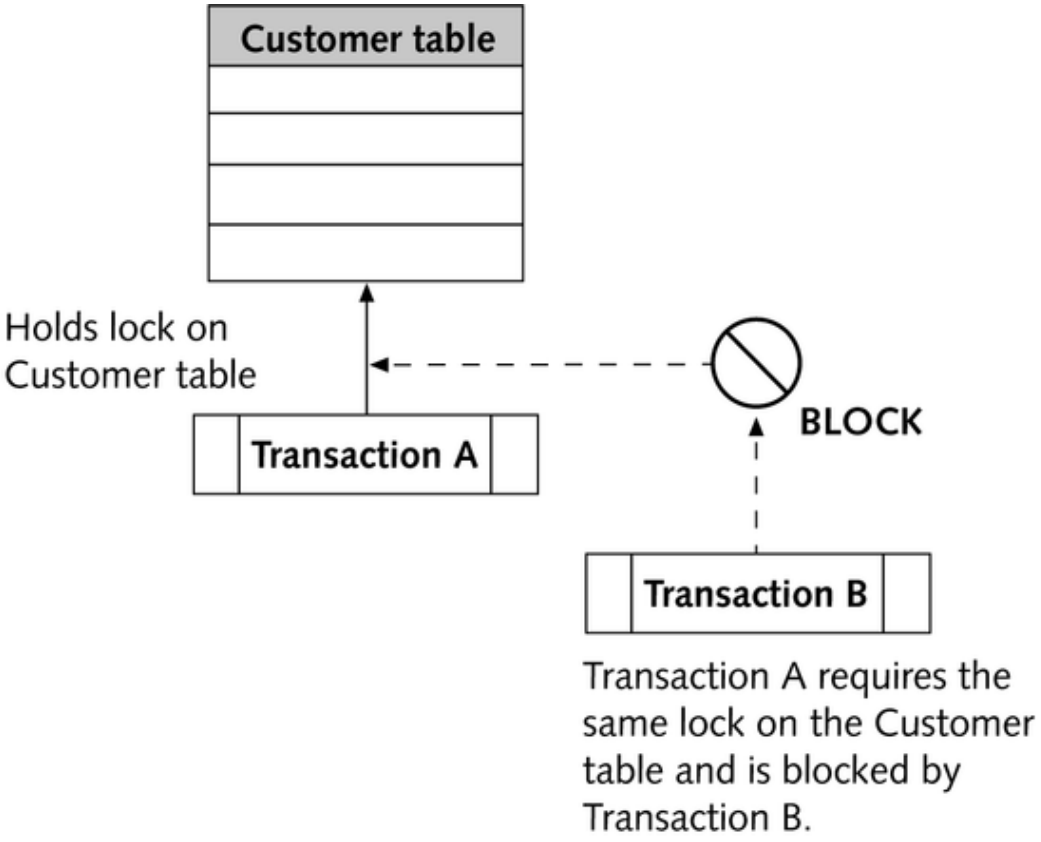
Lock Compatibility

- Locks may or may not be compatible with other Locks

	LS	LX
LS	true	false
LX	false	false

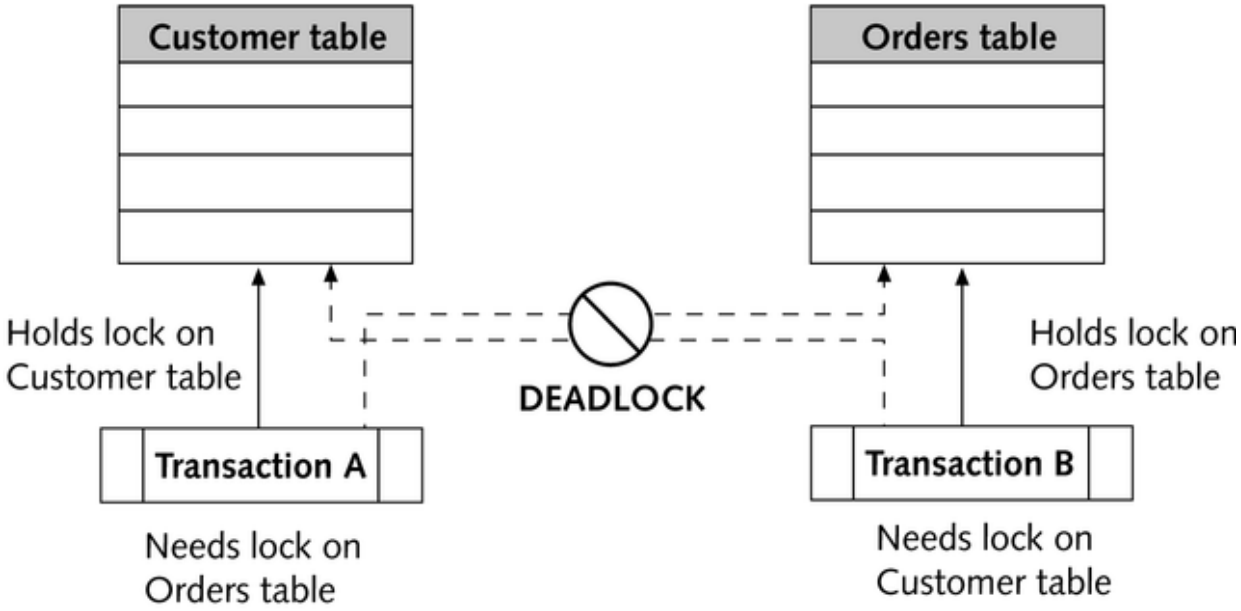
- Examples
  - Shared locks are compatible with all locks except exclusive
  - Exclusive locks are not compatible with any other locks

Blocks …


Customer table
Holds lock on
Customer table
BLOCK
Transaction A
Transaction B
Transaction A requires the same lock on the Customer table and is blocked by Transaction B.

… Deadlocks

- Occurs when two transactions are blocking each other



- How to detects
  - In Wait-for graph, there exist loops

Deadlock solution

- If a set of transactions is deadlocked
  - Choose the victim
  - Rollback victim transaction and redo
- How to customize the lock time-out setting
  - `SET LOCK_TIMEOUT` function
- How to minimize deadlocks
  - Access objects in same order
  - Keep transactions short
  - Use low isolation level
  - Use bound connections

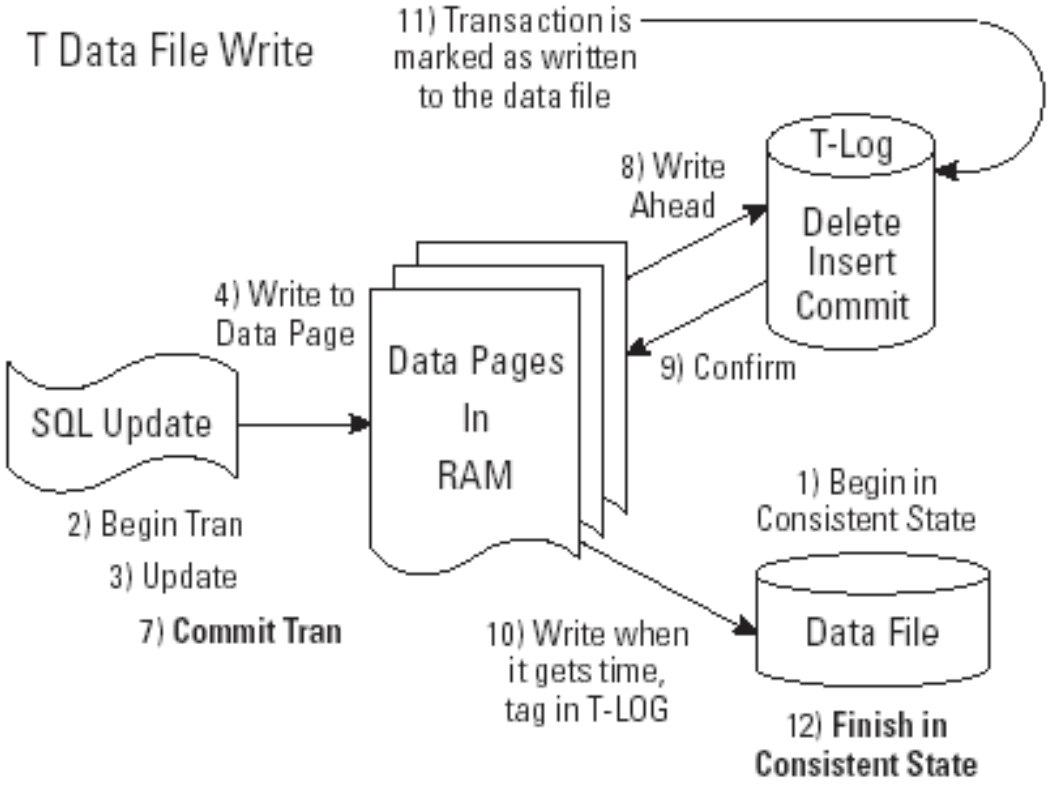
Remarks

- Keep transactions short
- Design transactions to avoid/minimize deadlocks
- In most cases: use SQL Server defaults for locking
- Be careful when use locking options

Agenda

- Transaction concepts & properties
- Transaction concurrency problems
- Isolation levels
- Lock
- Log and recovery
- Serializability checking

How RDBMS works with transactions



As one of the last steps, the data modification is written to the data file.

36

Transaction Recovery

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

Transaction Recovery

- Deferred write
  - Transaction operations do not immediately update the physical database
  - Only the transaction log is updated
  - Database is physically updated only after the transaction reaches its commit point using the transaction log information
  - If the transaction aborts before it reaches its commit point, no ROLLBACK is needed because the DB was never updated
  - A transaction that performed a COMMIT after the last checkpoint is redone using the "after" values of the transaction log

Transaction Recovery

- Write-through
  - 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 is done to restore the database to a consistent state
    - A transaction that committed after the last checkpoint is redone using the "after" values of the log
    - A transaction with a ROLLBACK after the last checkpoint is rolled back using the "before" values in the log

The Transaction Log

- Increases processing overhead but the ability to restore a corrupted database is worth the price
- Log contains:
  - A record for the beginning of transaction
  - For each transaction component (SQL statement)
    - Type of operation being performed (update, delete, insert)
    - Names of objects affected by the transaction (the name of the table)
    - "Before" and "after" values for updated fields
    - Pointers to previous and next transaction log entries for the same transaction
  - The ending (COMMIT) of the transaction
- If a system failure occurs, the DBMS will examine the log for all uncommitted or incomplete transactions and it will restore the database to a previous state

Transaction Log Architecture

- Write-Ahead transaction log
  - Flushing the pages
- Use of checkpoints
  - Minimize what must be processed to recover
- Shrinking the transaction log
  - `DBCC SHRINKDATABASE (UserDB, 10);`
  - `DBCC SHRINKFILE (AirlineReservation, 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

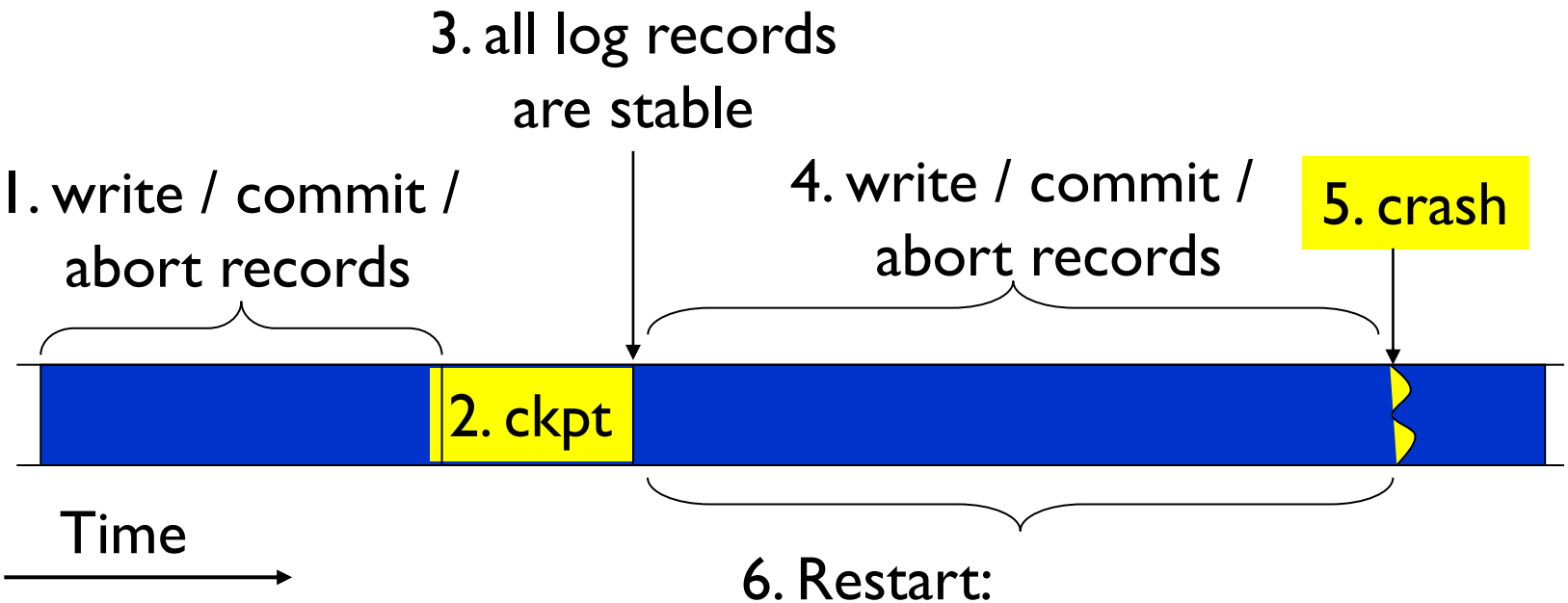
PTR = Pointer to a transaction log record ID

TRX_NUM = Transaction number

(Note: The transaction number is automatically assigned by the DBMS.)

Checkpoints

- Problem - Prevent Restart from scanning back to the start of the log
- A checkpoint is a procedure to limit the amount of work for Restart



Transaction log example

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	54778-2T	PROD_QOH	45	43
363	101	352	365	UPDATE	CUSTOMER	10011	CUST_BALANCE	615.73	675.62
365	101	363	Null	COMMIT	**** End of Transaction				
397	106	Null	405	START	****Start Transaction				
405	106	397	415	INSERT	INVOICE	1009			1009,10016, ...
415	106	405	419	INSERT	LINE	1009,1			1009,1, 89-WRE-Q,1, ...
419	106	415	427	UPDATE	PRODUCT	89-WRE-Q	PROD_QOH	12	11
423				CHECKPOINT					
427	106	419	431	UPDATE	CUSTOMER	10016	CUST_BALANCE	0.00	277.55
431	106	427	457	INSERT	ACCT_TRANSACTION	10007			1007,18-JAN-2004, ...
457	106	431	Null	COMMIT	**** End of Transaction				
521	155	Null	525	START	****Start Transaction				
525	155	521	528	UPDATE	PRODUCT	2232/QWE	PROD_QOH	6	26
528	155	525	Null	COMMIT	**** End of Transaction				

Implementing Restart (rev 1)

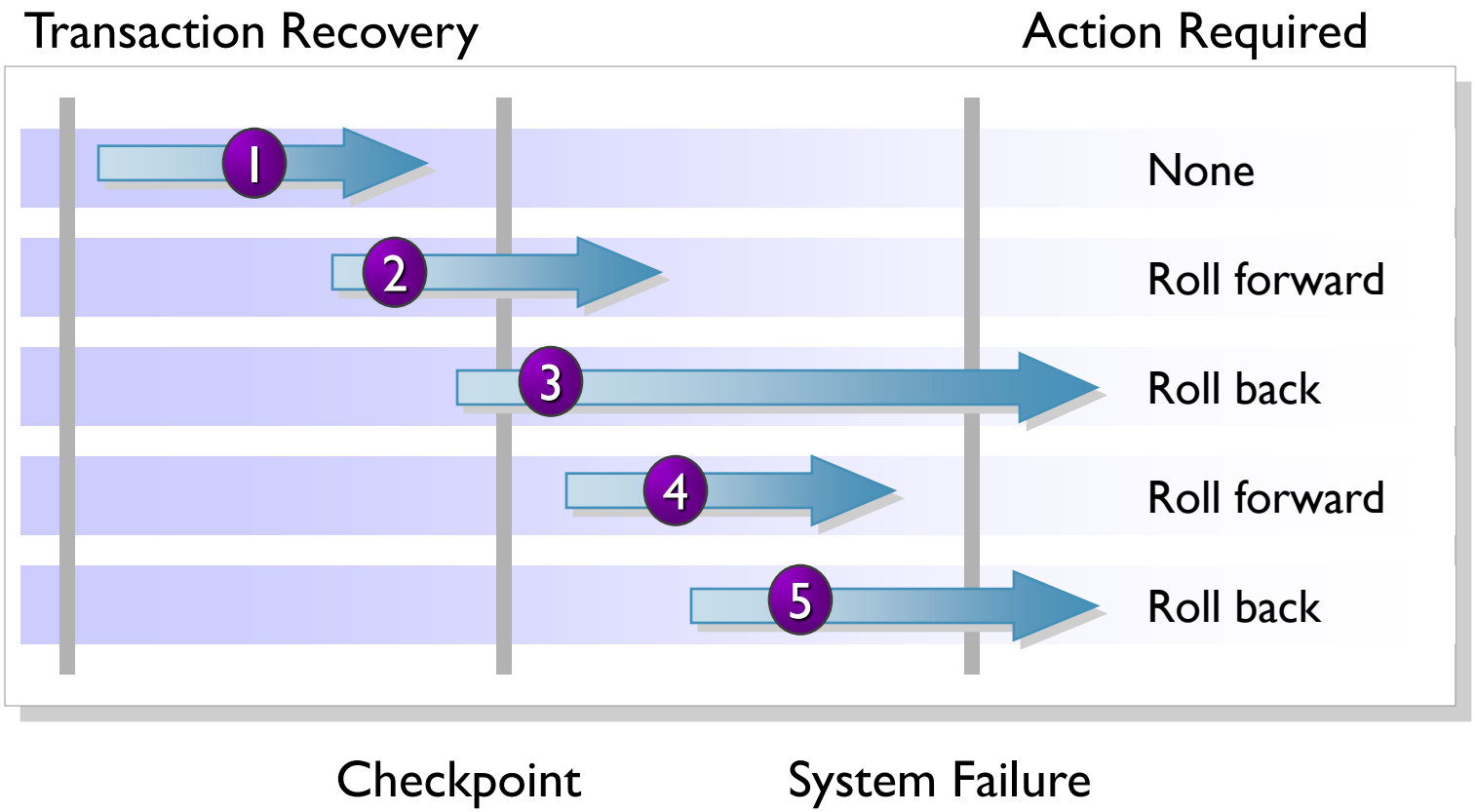
- Scan the log backwards from the end.
- Construct a commit list and page list during the scan (assuming page level logging)
- Commit (T) record => add T to commit list
- Update record for P by T
  - if P is not in the page list then
    - add P to the page list
    - if T is in the commit list, then redo the update, else undo the update

Restart Algorithm (rev 2)

- No need to redo records before last checkpoint,
  - Starting with the last checkpoint, scan forward in the log.
  - Redo all update records. Process all aborts.
    Maintain list of active transactions (initialized to content of checkpoint record).

→ Reduce restart time by checkpointing frequently.

Transaction Recovery and Checkpoints



Agenda

- Transaction concepts & properties
- Transaction concurrency problems
- Isolation levels
- Lock
- Log and recovery
- Serializability checking

Serial and non-serial Schedules

- Serial schedule: transactions are ordered one after the other
- Nonserial schedules: transactions are interleaved. There are many possible orders or schedules → good performance
- *Serializability* theory attempts to determine the 'correctness' of the schedules.
  - *The Objective of serializability is to find nonserial schedules that allow transactions to execute concurrently without interfering with one another.*
- A schedule S of n transactions is serializable if it is equivalent to some serial schedules of the same n transactions.

The Scheduler

- Special DBMS program: establishes order of operations within which concurrent transactions are executed
- 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 and time stamping
- Ensures computer's central processing unit (CPU) is used efficiently
  - Default would be FIFO without preemption – idle CPU (during I/O) is inefficient use of the resource and result in unacceptable response times in within the multiuser DBMS environment
- Facilitates data isolation to ensure that two transactions do not update the same data element at the same time

Conflicting operations

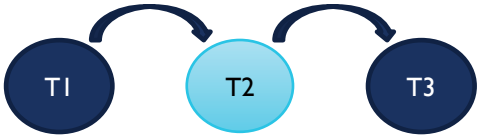
- Transactions are executing concurrently over the same data
  - **R(x)** reads the x value of the object
  - **W(x)** assigns the value **v** to **x**
- When they both access the data and at least one is executing a WRITE, a conflict can occur

	R(x)	**W(x)**
R(x)	ok	Conflict
W(x)	Conflict	Conflict

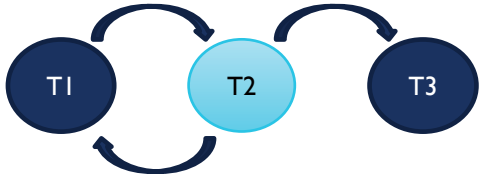
Operations of T1 and T2	compatible	permutable
T1 writes on A and T2 writes on A	no	no
T1 reads A and T2 writes on A	no	no
T1 reads or writes A and T2 reads or writes B	yes	yes

Precedence graph of a schedule

- Definition: there is a precedence from a transaction T1 on a transaction T2 if there exists at least one no permutable operation O1 from T1 with an operation O2 from T2 in the serialisable schedule
- Serialisable condition: precedence graph with no loop



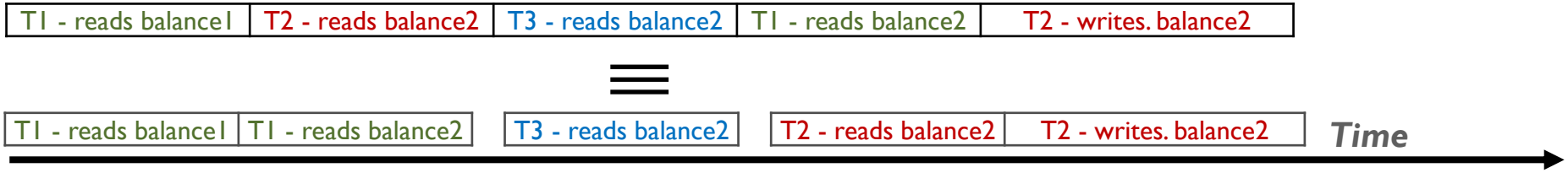
Serializable



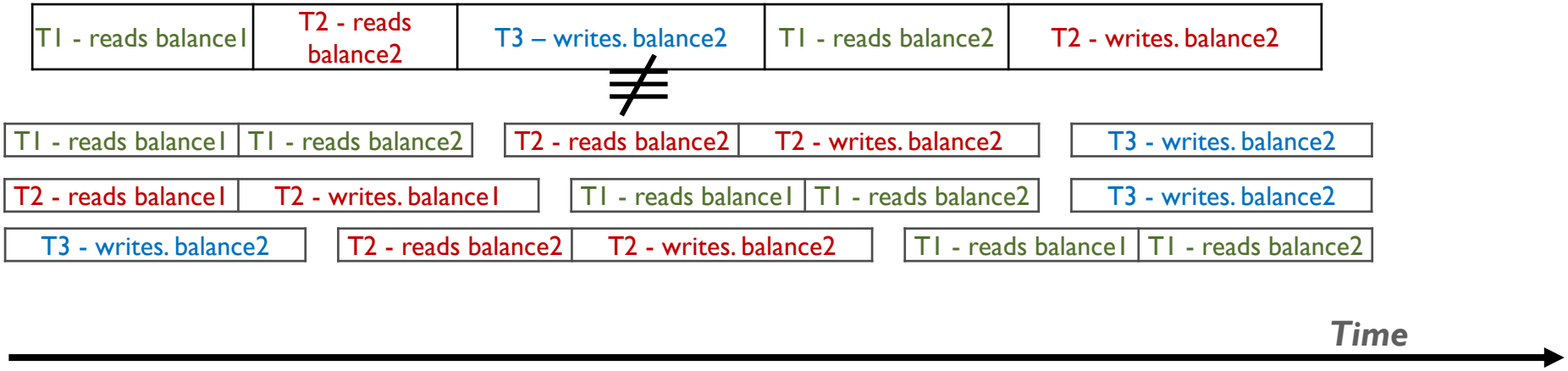
Non-serializable

Serialisable or not?

- Example of Serializable schedule



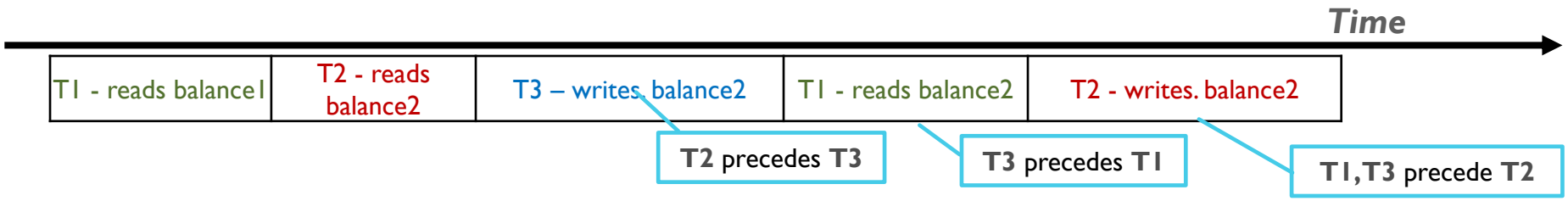
- Example of non-serialisable schedule

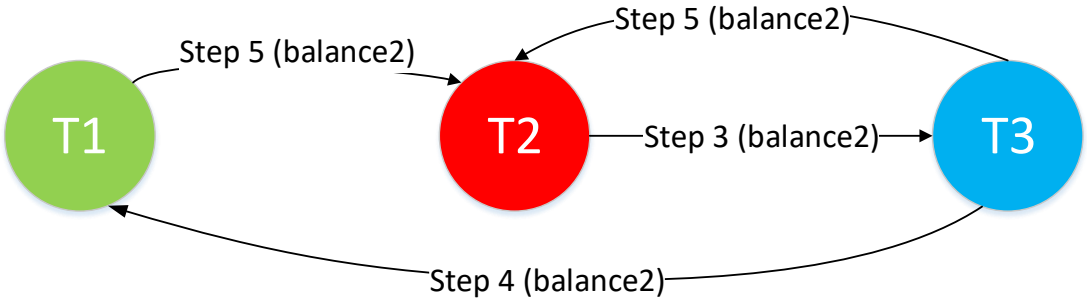


→ How to detect is it not serializable?

Example (previous schedule)

- Serializable?

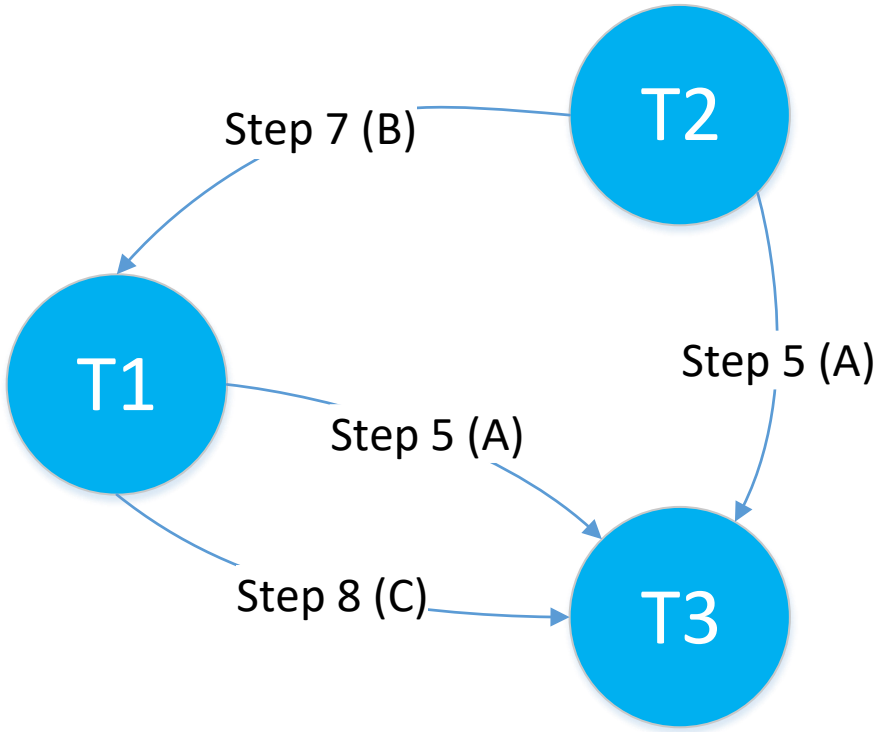




→ Is it possible to make this schedule serialisable? No

Exercise: draw precedence graph

Step	T 1	T 2	T 3
1	Read A		
2	Read B		
3		Read A	
4		Read B	
5			Write A
6	Write C		
7	Write B		
8			Write C



Excercise

- Consider schedule: r1 (x) r2 (y) w1 (y) w3 (x) w1 (t) w5 (x) r4 (z) r2 (z) w4 (z) w5 (z) r3 (t) r5 (t)
  - Transactions: 1,2 … 5
  - Resources: x,y,z,r,t
- Question:
  - Draw precedence graph
  - Is this execution serializable? If yes, give a serial execution that is equivalent to this one, otherwise explain why it is not

- Transaction is a sequence of statements that runs as a unit to ensure ACID properties
- Choose appropriate TRAN ISOLATION levels to avoid concurrency problems while obtaining good performance
- Avoid and be careful with Deadlock

TAKE-HOME MESSAGES

APPENDIX: TRANSACTIONS WITH SQL SERVER

Considerations for Using Transactions

- Transaction guidelines
  - Keep transactions as short as possible
  - Use caution with certain Transact-SQL statements
  - Avoid transactions that require user interaction

- Issues in nesting transactions
  - Allowed, but not recommended
  - Use `@@trancount` to determine nesting level

Transaction Types

- Explicit Transaction
  - Explicitly define start and end
- Autocommit Transactions
  - Every statement is committed or rolled back
- Implicit Transactions
  - Statement after "Commit Transaction" or Rollback Transaction" starts a new transaction
  - SET IMPLICIT_TRANSACTIONS ON

Explicit Transaction

- Defining

BEGIN TRANSACTION ROLLBACK TRANSACTION

COMMIT TRANSACTION

```
BEGIN TRANSACTION
    INSERT INTO Extensions (PrimaryExt)
       VALUES ('555-1212')
    INSERT INTO Extensions (VoiceMailExt)
       VALUES ('5551212')
ROLLBACK TRANSACTION
```

Implicit Transaction

- Using

```
SET IMPLICIT_TRANSACTIONS ON
```

- An implicit transaction starts when one of the following statements is executed

- ALTER DATABASE
- CREATE
- DELETE
- DROP
- FETCH
- GRANT
- INSERT
- OPEN
- REVOKE
- SELECT
- TRUNCATE TABLE
- UPDATE

➢ Transaction must be explicitly completed with `COMMIT` or `ROLLBACK TRANSACTION`

Savepoints

- For long transactions that contain many SQL statements, intermediate markers, or **savepoints**, can be declared. Savepoints can be used to divide a transaction into smaller parts.

- By using savepoints, you can arbitrarily mark your work at any point within a long transaction. This gives you the option of later rolling back all work performed from the current point in the transaction to a declared savepoint within the transaction.

- For example, you can use savepoints throughout a long complex series of updates, so if you make an error, you do not need to resubmit every statement.

63