# Relational Databases Management Issues (Part 1)

# Relational Database Management Issues

- Database transactions
- Lock-based concurrency control
  – This week

- Snapshot isolation
- Timestamping concurrency control
- Security
  – Next week

- The lecture slides are based on the ones provided with the course textbook:
  – Connolly,T. and Begg, C. "*Database Systems - A Practical Approach to Design, Implementation, and Management*.", 6th Ed., Pearson Education Ltd

CITY UNIVERSITY OF LONDON EST 1894

CSR Building confidence in a computerised world
www.csr.city.ac.uk

# DBMS Main Functions

Maintain the reliability and consistency of data(bases) in presence of failures of hardware and software, especially when multiple users are accessing the databases

**Transaction** support
**Concurrency** control
Database **recovery** services
**Authorisation** services

# Database Transaction

- Definition: A collection of operations that form a *single logical unit* of work on the database
- Application program is a series of transactions with (usually) non-database processing in between.
- Transforms database from one consistent state to another, (but, consistency may be violated).

E.g.: Transfer of money from ACC1 to ACC2

```
T1:    read(ACC1);
       ACC1 := ACC1 - 50;
       write(ACC1);
       read(ACC2);
       ACC2 := ACC2 + 50;
       write(ACC2);
```

# Transaction Processing

- Properly executed by the database system despite failures

- For example, “All or None”
  – Either the entire transaction is executed (all of its operations are executed), or none of them is executed

- A.C.I.D properties - to ensure consistency and integrity of the data

- **A**tomicity, **C**onsistency, **I**solation, **D**urability

# ACID Properties

- *Atomicity*: either all the operations in a transaction are executed against the database, or none is.
  - It implies *indivisibility* and *irreducibility*
- *Consistency*: the execution of a transaction must preserve the consistency of the database.
- *Isolation*: concurrent transactions are executed independently of one another.
  - In practice, *most frequently relaxed* out of all 4 properties!
- *Durability*: the effects of a successfully completed transaction are stored in the database permanently (not lost due to HW/SW failure).

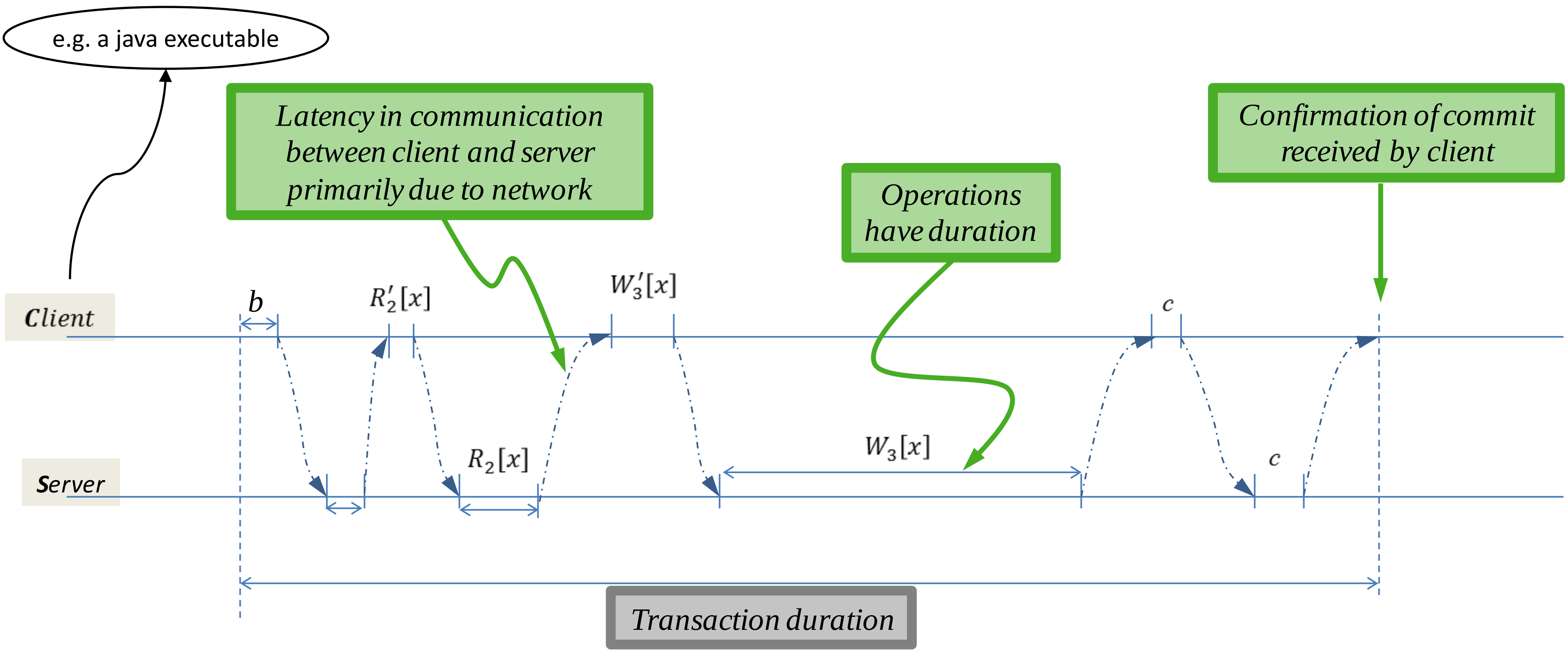
*ACID properties first defined in the paper *"Principles of transaction oriented database recovery"*, T Haerder, A Reuter, ACM Computing Survey, Volume 15, Issue 4, 1983

# Transaction Outcomes & Boundaries

- Database transaction - a set of operations of these kinds:
  - Transaction boundary/edge: *begin*, *commit*, *abort (rollback)*
  - Reads (SELECT), writes (DELETE, INSERT, UPDATE)
- Committed: when it completes successfully;
- Aborted (Rolled-back): when it is not executed successfully;

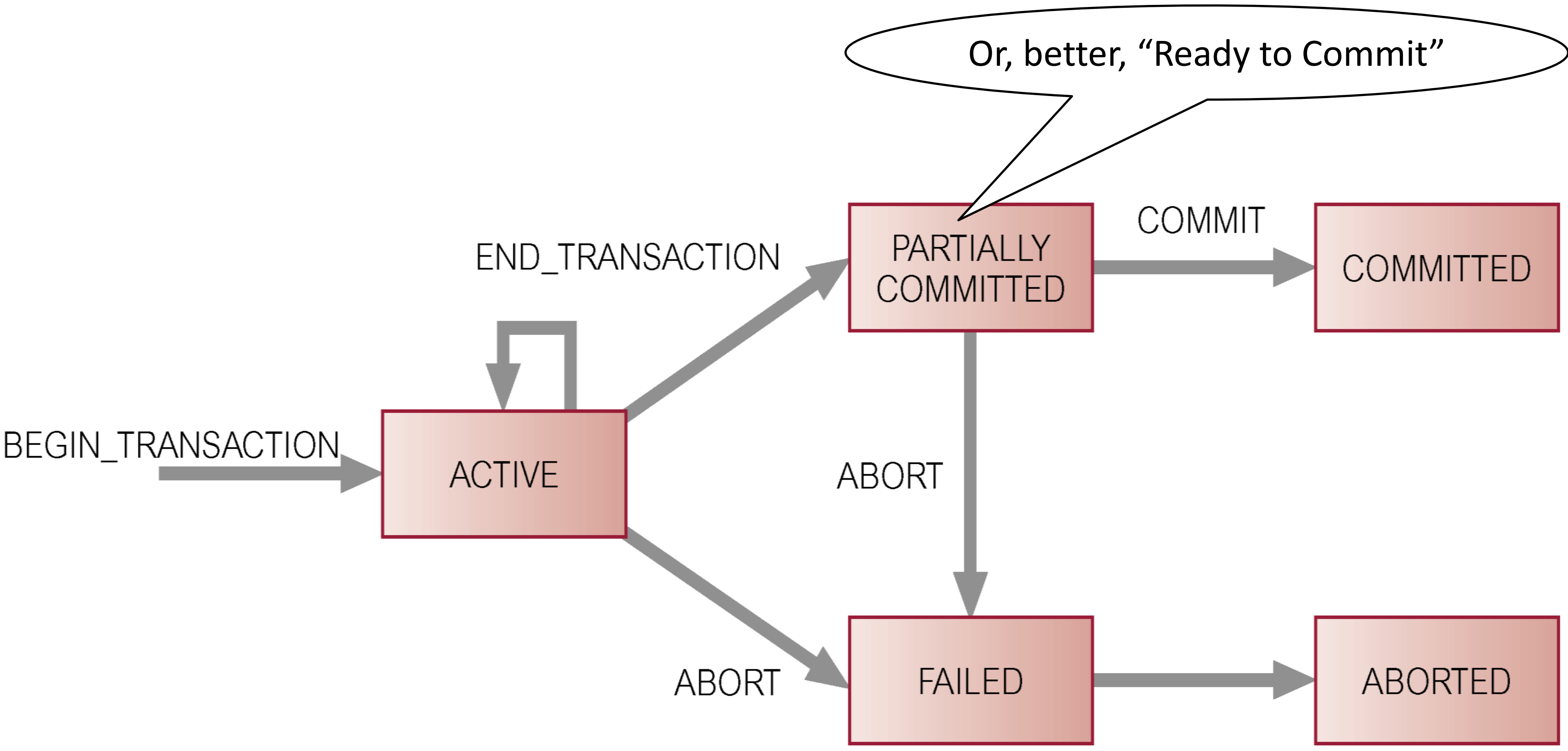
- SQL identifies start of transaction as BEGIN/START;
  - Please note some types of DB client connectivity, e.g. JDBC has not got explicit "start transaction"! This is consistent to (at least some versions of) the ISO/ANSI SQL standard.
- SQL identifies the end of a successful transaction as COMMIT;
- SQL identifies the abortion of an unsuccessful transaction as ROLLBACK (undo the transaction).

# DBMS transaction processing example - Single client




- If *client* and *server* on the same machine, then no network delay; the communication latency is due to Inter Process Communication (or Inter-Thread Communication).
- This is a simplified example - usually Databases are manipulated by multiple, concurrent users!

# State Transition Diagram for DB Transaction



# Database Transaction Log

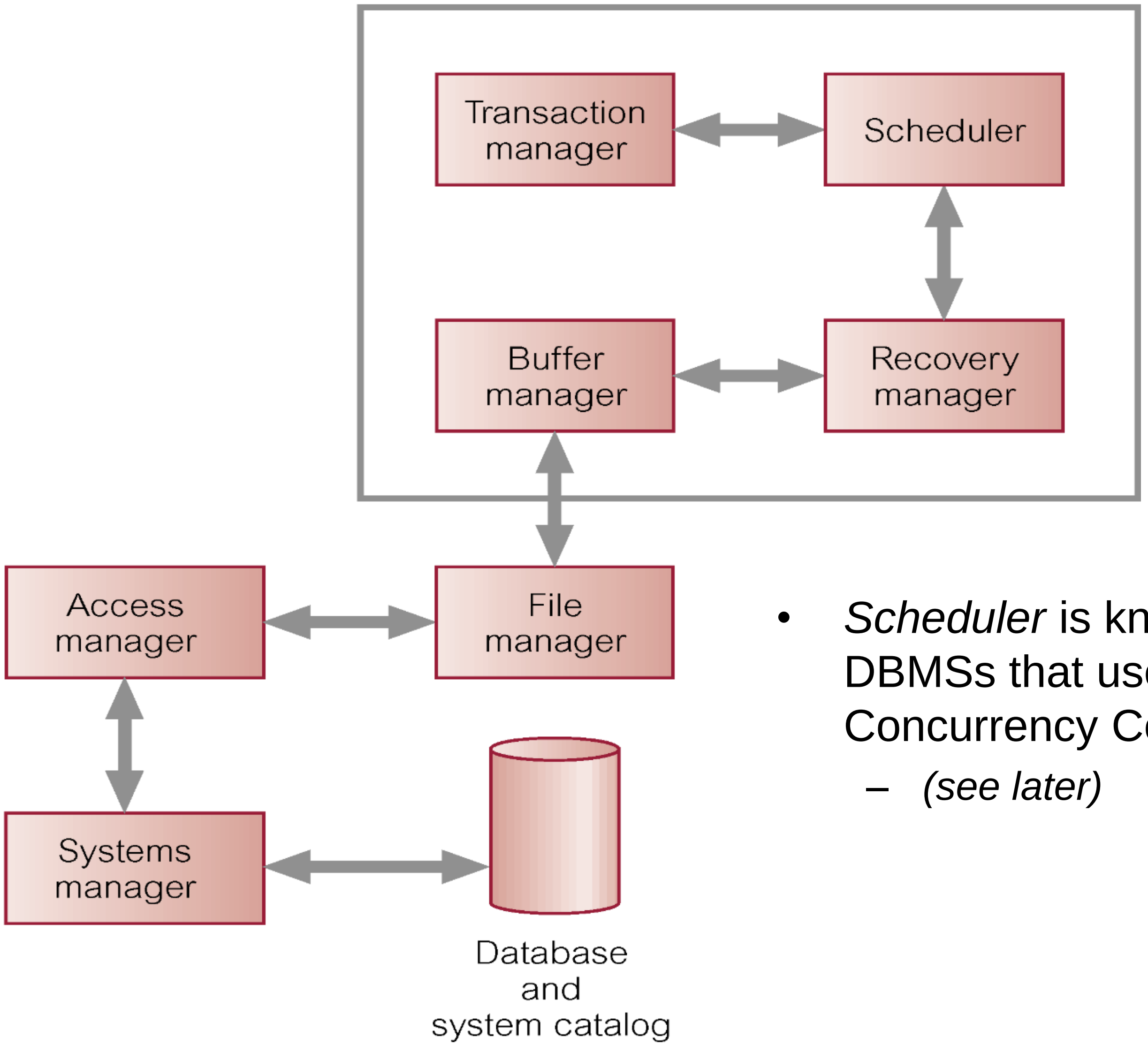
- In order to support atomicity/durability the system has to maintain a record of all the database modifications on a stable storage, before modifying the database (files)
- Physically, the log is a (set of) file(s) containing updates done to the database, stored in stable storage
- Redo a modification to ensure atomicity and durability, or undo a modification to ensure atomicity in case of a failure

- *More about this in the Week 5 lecture!*

# Transaction Management System

- Must guarantee that:
  - Transaction atomicity is preserved
  - If a transaction commits, then the effects of all its (write) operations are reflected in the database
  - If a failure occurs during transaction execution (before commit), the updates are undone

# Database manager/ Transaction Subsystem




- *Scheduler* is known as *Lock Manager* in DBMSs that use Lock-based approach for Concurrency Control.
  - *(see later)*

# Concurrency Control

- A definition: The mechanism of managing simultaneous operations (executed by multiple transactions) on the database preventing them to interfere with one another.
- How can the system control the interaction among, and effects of, concurrent transactions?
- Different CC mechanisms exist (in different DBMSs)
- If control of execution of concurrent operations is left to the underlying OS, many possible schedules, i.e. orders of operation executions, are possible
  - Including the ones that would leave DB in an inconsistent state, and/or the ones that would send wrong results to DB users.

# Transaction isolation levels

- The Isolation levels control the degree to which the execution of one transaction is isolated from the execution of other concurrent transactions (accessing the same data).
- A *lower* isolation level increases concurrency (improves performance), but can impair data correctness. Conversely, a *higher* isolation level ensures that data remains correct, but can negatively affect concurrency.
- ISO SQL-92 defines the following isolation levels (from the strictest to the most relaxed):
  - *Serializable*, *Repeatable read*, *Read committed* and *Read uncommitted*
  - Based on the defined anomalies: *Dirty Read*, *Non-Repeatable* and *Phantom Reads*.

| Isolation Level | Transactions | Dirty Reads | Non-Repeatable Reads | Phantom Reads |
|---|---|---|---|---|
| TRANSACTION_NONE | Not supported | Not applicable | Not applicable | Not applicable |
| READ_UNCOMMITTED | Supported | Allowed | Allowed | Allowed |
| READ_COMMITTED | Supported | Prevented | Allowed | Allowed |
| REPEATABLE_READ | Supported | Prevented | Prevented | Allowed |
| SERIALIZABLE | Supported | Prevented | Prevented | Prevented |

# Read uncommitted level

- This is the *lowest* isolation level. In this isolation level, *even Dirty reads* anomaly is possible:
  - A *dirty read* occurs when a transaction is allowed to read **data modified** by another concurrent (**not-yet-committed**) transaction.

| Transaction 1 | Transaction 2 |
|---|---|
| SELECT * FROM customer WHERE id = 1; | |
| | UPDATE customer SET balance = balance + 10 WHERE id = 1; |
| SELECT * FROM customer WHERE id = 1; | |
| | **ROLLBACK** |

Time

T1 sees, incorrectly,
the value of the balance increased –
the effect of the subsequently rolled back T2

CITY UNIVERSITY OF LONDON EST 1894

CSR Building confidence in a computerised world
www.csr.city.ac.uk

# Read committed level

- *Dirty reads* are prevented, but *Non-repeatable reads* (and *Phantom Reads* – see next slide) are not
  - *Non-repeatable reads* happen when a query returns **data** that would be **different** if the query were repeated within the same transaction

T1 sees
the value of the balance increased –
the effect of the already committed transaction T2.
BUT, this value is different
than the value it initially read!

| Transaction 1 | Transaction 2 |
|---|---|
| SELECT * FROM customer WHERE id = 1; | |
| | UPDATE customer SET balance = balance + 10 WHERE id = 1; **COMMIT**; |
| SELECT * FROM customer WHERE id = 1; COMMIT; | |

Time

# Repeatable read level

- *Dirty reads* and *Non-repeatable reads* are prevented, but *Phantom reads* are not
  - A *phantom read* occurs when, in the course of a transaction, a SELECT is executed twice, and the **collection of rows** returned by the second execution is **different** from the first

For T1, the result set returned by the execution of the 1st SELECT is **DIFFERENT** than the result set returned by the 2nd SELECT. The latter includes the row created by T2.

| Transaction 1 | Transaction 2 |
|---|---|
| SELECT * FROM customer WHERE balance BETWEEN 10 AND 20; | |
| | INSERT INTO customer VALUES (3, 'Mick', **15**); **COMMIT**; |
| SELECT * FROM customer WHERE balance BETWEEN 10 AND 20; | |

Time

# Serializable level

Beware of the muddled terminology, e.g. Oracle calls SERIALIZABLE the level that actually implements SNAPSHOT ISOLATION (see next week lecture for SI)

- This is the highest level of isolation. It prevents *dirty reads*, *non-repeatable reads* and also *phantom reads*
- In Serializable isolation mode, the 1st SELECT in the previous slide would result in all records (current and *potential ones*!) with balance in the *range* 10 to 20 being locked (a *predicate lock*), thus the INSERT (executed by Transaction 2) would **block** until Transaction 1 was committed.
  - In Repeatable-read mode, however, the *range* would *not* be locked! This allows the record to be inserted and the second execution of the SELECT (by Transaction 1) to return the new row as part of its result set.
- Other types, or more correctly “flavours”, of the data correctness anomalies (shown in the previous slides) exist
  - e.g. Inconsistent Analysis Problem
- Serializable level is **independent** of the CC mechanisms used to implement it
  - *Locking* is the most common CC mechanism; others exist, however (see next week)
- NB: Not to be confused with the term **Serialization** in OO languages (“Marshalling”)

# Serializability

- Control of concurrent execution of transactions to guarantee consistency of the database
- *Transaction Schedule*: Sequence of operations by concurrent transactions that preserves the order of the operations in each individual transaction
- *Serial Schedule*: The transactions are performed in serial order (operations are not interleaved)
  - sequentially with no overlap in time
- *Non-serial Schedule*: The operation of the transactions are interleaved.

# Serializability (cont.)

- Goal: To find /enforce non-serial schedule(s) that allow transactions to be executed concurrently, but without interfering with one another, and therefore produce a database state that could be produced by a serial execution ("equivalence" to a serial schedule)
  - All this without sacrificing performance (significantly)
  - A trivial solution is to execute transactions *serially*. But no concurrency exists then – the performance is then trivialised and the system thus unusable in most situations!

- How can we guarantee serializability?

# Locking

- *An approach* for providing concurrency control and avoiding the problems presented before
- It controls data access by transactions to prevent incorrect results
- (Basic) lock types:
  - *Exclusive* lock (X-lock) - write lock
  - *Shared* lock (S-lock) - read lock
- Lock compatibility:

| $T_A$ has the lock<br><br>$T_B$ requests a lock | X | S | None |
|---|---|---|---|
| X | No | No | Yes |
| S | No | Yes | Yes |

CITY UNIVERSITY OF LONDON EST 1894

CSR Building confidence in a computerised world
www.csr.city.ac.uk

# Lock Granularity trade-offs

- Size of data items that are locked

| | | Concurrency/Performance | Data Correctness |
|---|---|---|---|
| Coarse | | low | high |
| ↓ | Entire DB | ↓ | ↓ |
| | File | | |
| | Page | | |
| | Record | | |
| | Field value | | |
| Fine | | high | low |

Using locking in transactions, does not guarantee serializability of schedules themselves! The logical time of when lock/unlock occur is an essential factor.

# Two-Phase Locking Protocol (2PL)

- Phase 1 (Growing/expanding phase)
  - A transaction T always acquires a lock on an object before reading (S-lock), or writing (X-lock), it.
- Phase 2 (Shrinking phase)
  - After releasing a lock, transaction T cannot acquire a new lock

  - *Strict 2PL (S2PL)* – 2PL + release its write (exclusive) locks only after it has ended, i.e., after being either committed or aborted
  - *Strong Strict 2PL (SS2PL)* – 2PL + release both its write (exclusive) and read (shared) locks only after it has ended
    - Referred to as *Rigorous 2PL* too.
  - But, S2PL and SS2PL frequently referred to as just 2PL

# Need for Concurrency Control

- 3 further examples of potential problems caused by concurrent database transactions:
  - *Lost update* problem.
  - *Uncommitted dependency* problem (similar to *dirty read*).
  - *Inconsistent analysis* problem (similar to *non-repeatable read*).

Note: You *must* (further) review those in *your own* time.

# Lost Update Problem

- Successfully completed update is overridden by another user.
- Initially, a bank account X has balance Bal of £100
- $T_1$ withdrawing £10 from an account with $bal_x$, initially £100.
- $T_2$ depositing £100 into same account.
- With serial execution, final balance would be £190.

| Time | $T_1$ | $T_2$ | $bal_x$ |
|---|---|---|---|
| $t_1$ | | begin_transaction | 100 |
| $t_2$ | begin_transaction | read($bal_x$) | 100 |
| $t_3$ | read($bal_x$) | $bal_x = bal_x + 100$ | 100 |
| $t_4$ | $bal_x = bal_x - 10$ | write($bal_x$) | 200 |
| $t_5$ | write($bal_x$) | commit | 90 |
| $t_6$ | commit | | 90 |

- *I leave it for you to work out this example. The book presents a solution – see Sect. 22.2.3, and the problems are described in 22.2.1*

# Uncommitted Dependency Problem

- Occurs when one transaction can see intermediate results of another transaction before it has committed (cf *dirty read* anomaly).
- Initially, a bank account X has balance Bal of £100
- $T_4$ updates $bal_x$ to £200 but it aborts, so $bal_x$ should be back at original value of £100.
- $T_3$ has read new value of $bal_x$ (£200) and uses value as basis of £10 reduction, giving a new balance of £190, instead of £90.

| Time | $T_3$ | $T_4$ | $bal_x$ |
|---|---|---|---|
| $t_1$ | | begin_transaction | 100 |
| $t_2$ | | read($bal_x$) | 100 |
| $t_3$ | | $bal_x = bal_x + 100$ | 100 |
| $t_4$ | begin_transaction | write($bal_x$) | 200 |
| $t_5$ | read($bal_x$) | ⋮ | 200 |
| $t_6$ | $bal_x = bal_x - 10$ | rollback | 100 |
| $t_7$ | write($bal_x$) | | 190 |
| $t_8$ | commit | | 190 |

- *I leave it for you to work out this example. The book presents a solution – see Sect. 22.2.3, , and the original problems are described in 22.2.1*

# Inconsistent Analysis Problem

- Occurs when the first transaction reads several values but the second transaction updates some of them during execution of the first.
- Sometimes referred to as *non-repeatable read*.
- $T_6$ is totaling balances of account x (£100), account y (£50), and account z (£25).
- Meantime, $T_5$ has transferred £10 from $bal_x$ to $bal_z$, so $T_6$ now has wrong result (£10 too high).

| Time | $T_5$ | $T_6$ | $bal_x$ | $bal_y$ | $bal_z$ | sum |
|---|---|---|---|---|---|---|
| $t_1$ | | begin_transaction | 100 | 50 | 25 | |
| $t_2$ | begin_transaction | sum = 0 | 100 | 50 | 25 | 0 |
| $t_3$ | read($bal_x$) | read($bal_x$) | 100 | 50 | 25 | 0 |
| $t_4$ | $bal_x = bal_x - 10$ | sum = sum + $bal_x$ | 100 | 50 | 25 | 100 |
| $t_5$ | write($bal_x$) | read($bal_y$) | 90 | 50 | 25 | 100 |
| $t_6$ | read($bal_z$) | sum = sum + $bal_y$ | 90 | 50 | 25 | 150 |
| $t_7$ | $bal_z = bal_z + 10$ | | 90 | 50 | 25 | 150 |
| $t_8$ | write($bal_z$) | | 90 | 50 | 35 | 150 |
| $t_9$ | commit | read($bal_z$) | 90 | 50 | 35 | 150 |
| $t_{10}$ | | sum = sum + $bal_z$ | 90 | 50 | 35 | 185 |
| $t_{11}$ | | commit | 90 | 50 | 35 | 185 |

- *I leave it for you to work out this example. The book presents a solution – see Sect. 22.2.3, and the problems are described in 22.2.1*

# Deadlock

- An *impasse* that may result when two (or more) transactions are each waiting for locks, held by the other(s), to be released.
  - E.g. $T_{17}$ and $T_{18}$ both attempt to get exclusive locks on x and y, but in different order.

| Time | $T_{17}$ | $T_{18}$ |
|---|---|---|
| $t_1$ | begin_transaction | |
| $t_2$ | write_lock($\mathbf{bal_x}$) | begin_transaction |
| $t_3$ | read($\mathbf{bal_x}$) | write_lock($\mathbf{bal_y}$) |
| $t_4$ | $\mathbf{bal_x} = \mathbf{bal_x} - 10$ | read($\mathbf{bal_y}$) |
| $t_5$ | write($\mathbf{bal_x}$) | $\mathbf{bal_y} = \mathbf{bal_y} + 100$ |
| $t_6$ | write_lock($\mathbf{bal_y}$) | write($\mathbf{bal_y}$) |
| $t_7$ | WAIT | write_lock($\mathbf{bal_x}$) |
| $t_8$ | WAIT | WAIT |
| $t_9$ | WAIT | WAIT |
| $t_{10}$ | ⋮ | WAIT |
| $t_{11}$ | ⋮ | ⋮ |

# Deadlock

- Only one way to break deadlock: abort one or more of the transactions.
- Deadlock should be transparent to user, so DBMS should restart transaction(s).
- However, in practice DBMS cannot automatically restart aborted transaction since it is unaware of transaction logic even if it was aware of the transaction history
  - Unless *there is no user input* in the transaction or *the input is not a function of the database state* – this is not true in general case!

# Deadlock

- Three general techniques for handling deadlock:
  - Deadlock prevention.
  - Timeouts.
  - Deadlock detection and recovery.

- NB: The following video explains some of the material covered today: https://www.youtube.com/watch?v=KvmiYidCAe4
  - Parts of the video refer to material we do not cover – you *must* be clear what parts are relevant to this module. Ask if in doubt.
  - E.g. (especially) relevant material - about **timestamp-based deadlock prevention** - starts from about 5 min 55 sec, and lasts until about 8 mins 35 sec.

# Deadlock Prevention

- DBMS looks ahead to see if transaction would cause deadlock and never allows deadlock to occur.
- Could order transactions using transaction *timestamps*:
  - Smaller timestamps represent older transactions, while larger timestamps represent younger, more recent ones.
  - <u>Wait-Die</u> - only an older (O) transaction can wait for younger (Y) one, otherwise transaction is aborted (*dies*) and restarted with the same timestamp - *non-preemptive* scheme.
  - <u>Wound-Wait</u> - only a younger transaction can wait for an older one. If older transaction requests lock held by a younger one, the **younger** one is aborted (*wounded*) - *preemptive* scheme.

| | Wait/Die | Wound/Wait |
|---|---|---|
| O needs a resource held by Y | O waits | Y dies<br>*(Y is pre-empted)* |
| Y needs a resource held by O | Y dies | Y waits |

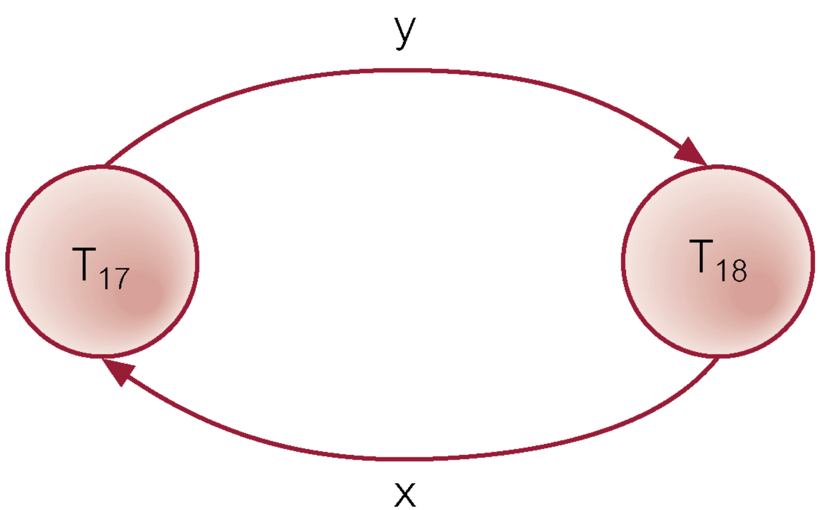
# (Dead)lock Timeouts

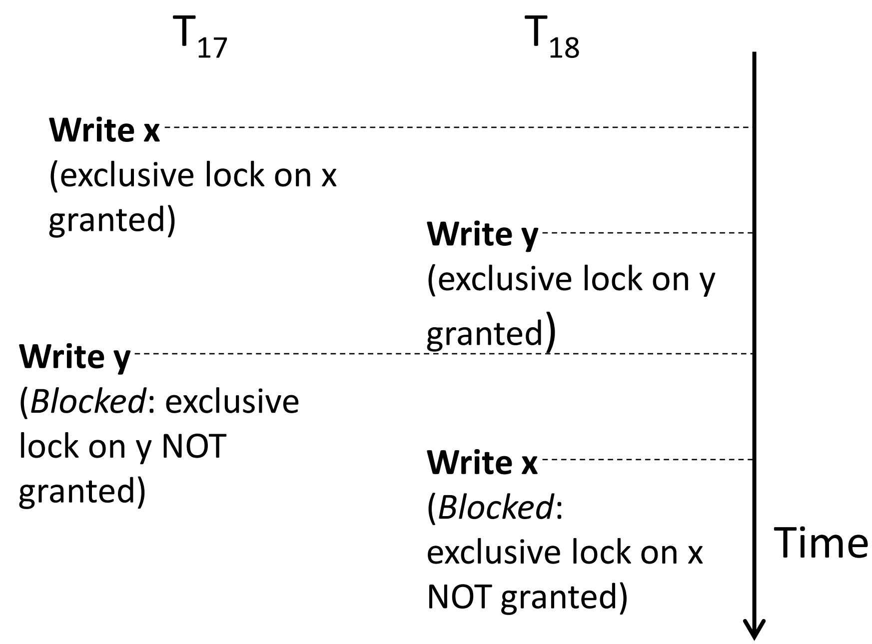
- Transaction that requests a lock will only wait for a period of time (which is usually system-defined).
  - An extreme setting: “NOWAIT” feature
    - in Oracle, PostgreSQL, Firebird, and other DBMSs!
    - Specified on different “levels”: SQL operation, DB connection, etc.
      - `SELECT * FROM dept WHERE deptno=10 FOR UPDATE` **`NOWAIT;`**
      - `int[]tpb={org.firebirdsql.jdbc.FirebirdConnection.`**`TPB_NOWAIT`**`};`
- If the requested lock has *not* been granted within this period, the lock request times out.
- In this case, DBMS assumes a transaction is deadlocked, even though it may not be
  - DBMS aborts, and (automatically) restarts the transaction.

# Deadlock Detection and Recovery

- DBMS allows deadlock to occur but detects it and breaks it.
- Usually handled by construction of **wait-for graph (WFG)** showing transaction dependencies:
  - Create a node for each transaction.
  - Create edge $T_i \rightarrow T_j$, **if** $T_i$ waiting to lock an item already locked by $T_j$.
- Deadlock exists if and only if WFG contains cycle.
- WFG is created at regular intervals
  - WFG creation initiation – usually a configuration parameter
    - e.g. if lock not granted after 1 sec – the default in PostgreSQL
  - Trade-off with performance
    - e.g. if deadlock detection runs too frequently, performance might be impacted

# Example - Wait-For-Graph (WFG), with a cycle




**There is a cycle in the graph – deadlock exists!**

# Recovery from Deadlock Detection

- Several issues:
  - Choice of deadlock victim;
  - How far to roll a transaction back;
    - It might be possible to rollback only a part of a txn, not all the operations/changes (“savepoints”)
  - Avoiding starvation.
- Selection of the victim based on:
  - The thx duration
  - How many data items the txn used
  - How many more data items needed until completion
    - But, this likely not known!
  - How many other txns will be involved in the rollback, etc.
  - Note: Store number of times a txn has been aborted, and if upper limit reached, make a different selection.

# Required Reading

- Connolly,T. and Begg, C. "*Database Systems - A Practical Approach to Design, Implementation, and Management*.", 6th Ed., Pearson Education Ltd
  - Transactions: Section 7.5, Section 22.1
  - Concurrency Control: 22.2 (but NOT "View Serializability", 22.2.6 "Multiversion timestamp ordering" and 22.2.7, "Optimistic techniques")

- This list is only indicative, and the text in these sections should be of course looked at as "additional", but required, reading to the lecture slides.
- This is certainly not an exhaustive list.
- The content of the chapters referred to *might not* be fully relevant to the module. You need to make the final decision about what is relevant and what is not yourselves, given the material that we covered in the lecture.