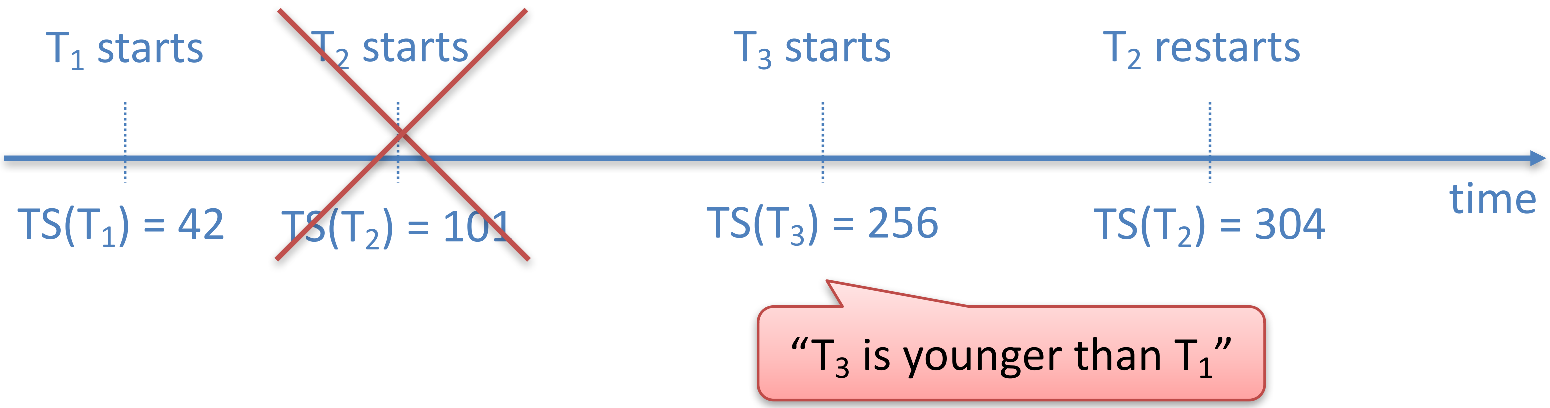
COMP207
Database Development

Lecture 12

Review of Chapter 2 and
the start of Chapter 3: Query Processing:
From SQL to Relational Algebra

Timestamp-based Scheduling

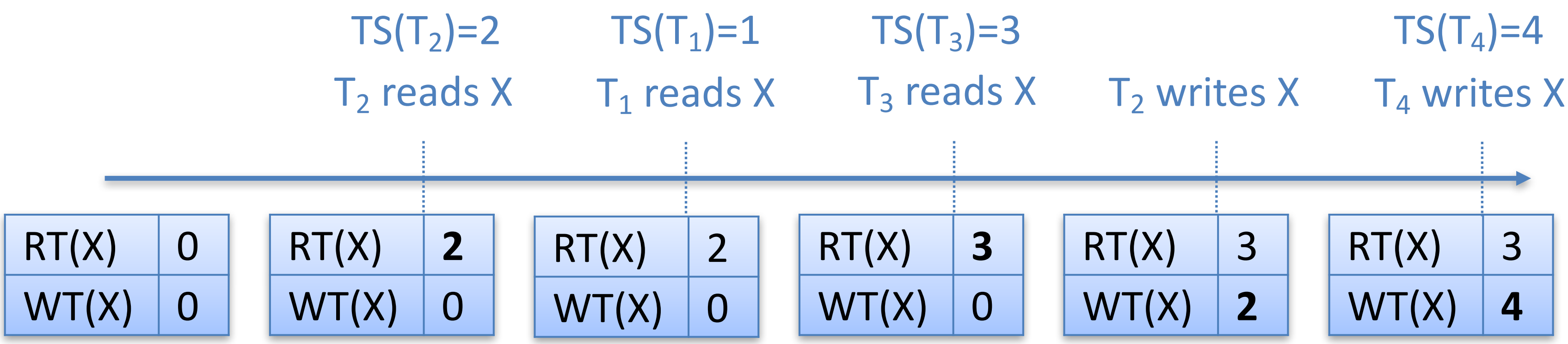
- Enforces a schedule that is *equivalent to the serial schedule* that contains all transactions ordered by start time.



- Uses timestamps
  - If a transaction T starts, assigns a new timestamp TS(T) to it
  - "Younger" transactions are those with higher timestamps
  - Timestamps are used to decide whether to grant read/write requests

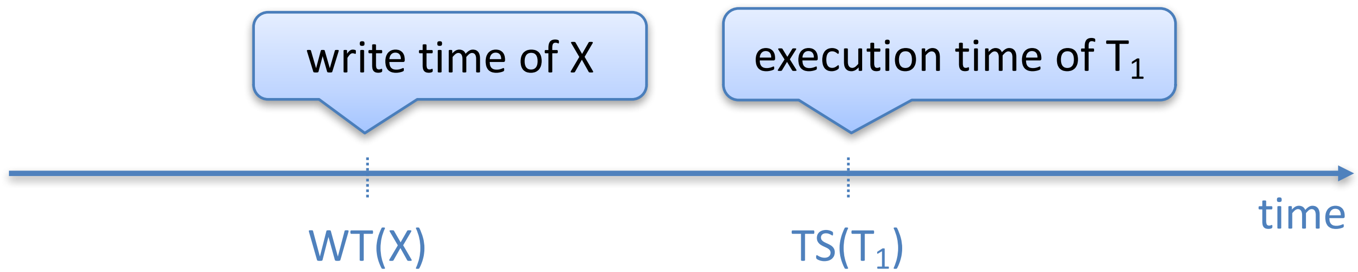
Additional Bookkeeping

- For each database item X, maintain:
  - **Read Time of X: RT(X)**
    Timestamp of youngest transaction that read X
  - **Write Time of X: WT(X)**
    Timestamp of youngest transaction that wrote X



Read Requests

T_1 requests to **read** X ...

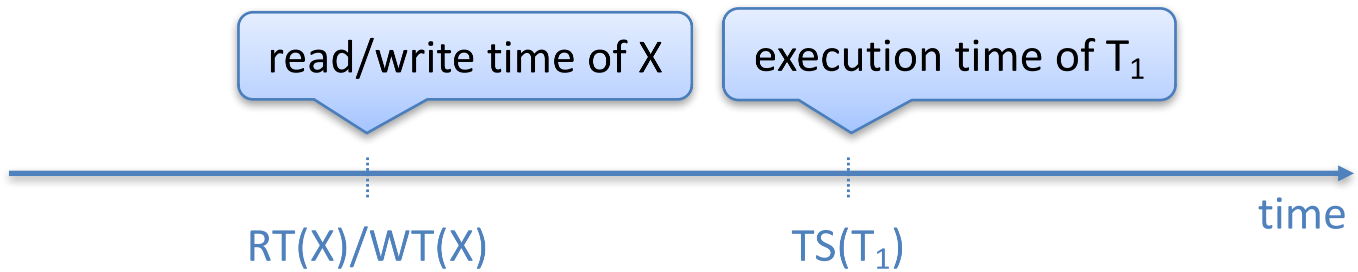


"X was last written before T_1 started"

- **Grant request** if $WT(X) \leq TS(T_1)$
- **Abort & restart T_1** otherwise

Write Requests

T_1 requests to **write** X …



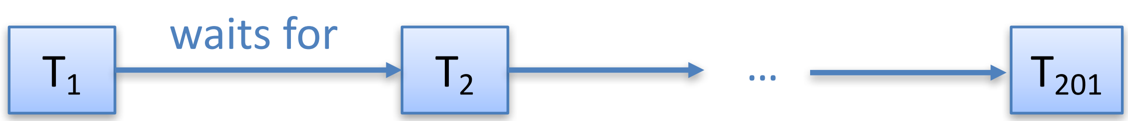
"X was last read or written before T_1 started"

- **Grant request** if $RT(X) \leq TS(T_1)$ and $WT(X) \leq TS(T_1)$
- **Abort & restart T_1** otherwise

MySQL (with InnoDB)

Also uses a timestamp based approach to ensure that reads do not interfer with writes

- Uses Wait-For graphs
- Except: If transaction has line c it is rolled back
- E.g. T_1 is rolled back in this case:



- If cycle: rollback the smallest transaction
- Deadlock detection can be switched off, in which case time-out is used (on locks)

Other DBMS

PostgreSQL

- Uses timeout on locks followed by Wait-For graphs
- Like MySQL: Uses a timestamp based approach for ensuring that reads do not interfere with writes

Oracle DB

- Uses timeout directly or timeout followed by Wait-For graphs
- Does not use locks on read

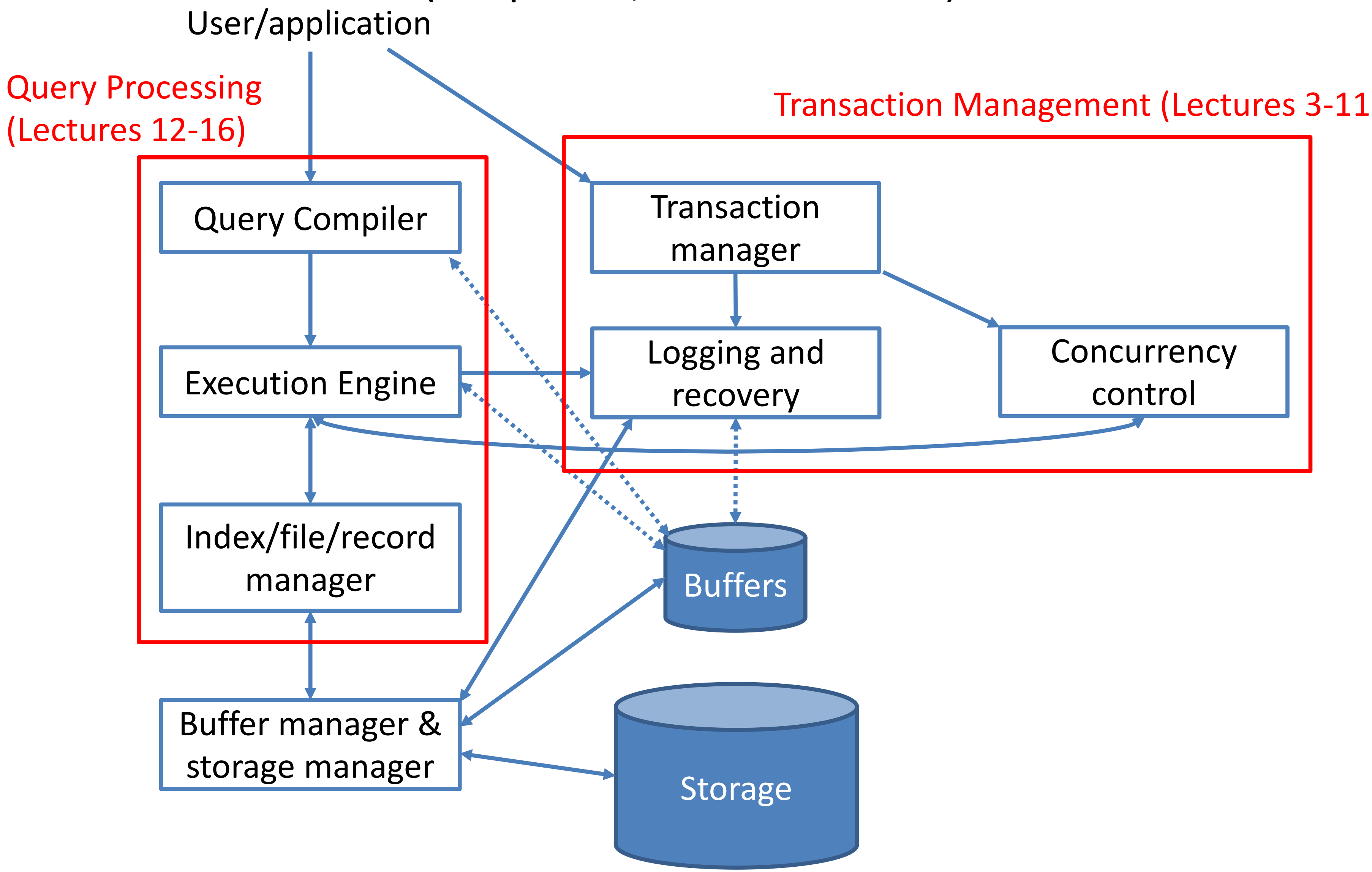
IBM DB2

- Uses lock timeout or global time-out followed by Wait-For graphs
- Uses update-locks

Concluding Remarks

Relational DBMS Components

(Simplified, from Lecture 1)



Transaction Management Review

- Dealing with **transactions** is a core task of DBMS
  - Many things can go wrong when processing transactions, even when executing single SQL statements.
  - Need to ensure **ACID properties**
- Requires careful **scheduling** of transactions and **logging** of relevant information
  - Schedules should be **conflict-serialisable**
  - Schedules should be **strict**
- Methods for enforcing conflict-serialisability & strictness:
  - **Strict two-phase locking** & **deadlock prevention** methods
  - **Timestamping**

ACID

- **A**tomicity
  - Transactions are fully executed or not at all
  - Ensured by Undo logging, Undo/Redo logging or Force
- **C**onsistency
  - Schedule executes transactions equivalent to a serial schedule
  - (needs two assumptions for this: non-database operations can be ignored and if a schedule is serial, then it is consistent)
  - Ensured by Serializability, Conflict-Serializability, 2PL and Timestamp-based Scheduling (also Strict versions of the last two)

This is intuition.
Exact def. is different

ACID continued

- Isolation
  - Transactions never reads uncommitted data
  - Ensured by Cascadeless and Strict schedules (incl. Strict 2PL and Strict Timestamp-based schedules)
- **D**urability
  - If a transaction is committed, it does not disappear
  - Ensured by Redo logging, Undo/Redo logging or No Steal
  - Recoverable schedules are also required

Transaction Support in DBMS

- Part of SQL:
  - Begin/end transactions, isolation levels, auto commit, …
  - Need to understand the consequences of these commands to make effective use of DBMS
    - When to combine different SQL statements into a transaction?
    - When do we need (conflict) serialisability? When is a weaker isolation level fine?
- Widespread ACID support in major DBMSs
  - Fully ACID compliant: PostgreSQL, Oracle DB, IBM DB2, …
  - Partly ACID compliant: MySQL (full compliance requires additional engines like InnoDB)

Transactions Beyond DBMS

- The techniques covered in this chapter are not confined to DBMS
- Similar issues whenever systems share resources
- Some example scenarios:
  – Processes in an operating system that access the same files, network resources, etc.
  – Users editing the same document online
  – Document versioning systems like subversion, git, etc.

Try it out…

- CREATE TABLE Student (id INT NOT NULL, name …);
- INSERT INTO Student VALUES (1, 'Anna', …);
  SELECT * FROM Student;
- **START TRANSACTION**;
  INSERT INTO Student VALUES (2, 'Ben', …);
  INSERT INTO Student VALUES (3, 'Chloe', …);
  ROLLBACK;
  SELECT * FROM Student;

Which tuples are returned?

Try out with any DBMS: MySQL, PostgreSQL, …

MySQL and PostgreSQL are easy to set up.

Lots of tutorials online.

- Try out reads, writes, different isolation levels, dirty reads, look up the documentation, …
- Experiment with more complex scenarios…