# TRANSACTION PROCESSING

University of Waterloo

# List of Slides

# Setting and Goals

- Query (and update) processing converts requests for *sets of tuples* to requests for reads and writes of physical objects in the database.
- database objects (depending on granularity) can be
  - ⇒ individual attributes
  - ⇒ records
  - ⇒ physical pages
  - ⇒ files (only for concurrency control purposes)
- Goals:
  - ⇒ concurrent execution of queries and updates
  - ⇒ guarantee that data isn't lost

# ACID Requirements

Transactions are said to have the “ACID” properties:

- **A**tomicity: all-or-nothing execution
- **C**onsistency: execution preserves database integrity
- **I**solation: a transaction’s updates are not visible until it commits (finishes successfully)
- **D**urability: updates made by a committed transaction will not be destroyed by subsequent failures.

# ACID Requirements (cont.)

Implementation of transactions in a DBMS has two aspects:

- **Concurrency Control:** guarantees that committed transactions appear to execute sequentially
- **Recovery Management:** guarantees that committed transactions are durable, and that aborted transactions have no effect on the database

# Abort and Commit

A transaction may terminate in one of two ways: by aborting, or by committing.

- When a transaction **commits**, any updates it made become durable, and they become visible to other transactions. A commit is the “all” in “all-or-nothing” execution.
- When a transaction **aborts**, any updates it made have made are undone (erased), as if the transaction never ran at all. An abort is the “nothing” in “all-or-nothing” execution.

# Transactions

A database server will often be processing several transactions at the same time: generally much faster than processing transactions **serially**, one at a time.

If $T_i$ and $T_j$ are concurrent transactions, then it is always correct to schedule the operations in such a way that:

- $T_i$ will appear to precede $T_j$ meaning that $T_j$ will "see" all updates made by $T_i$, and $T_i$ will not see any updates made by $T_j$, or
- $T_i$ will appear to follow $T_j$, meaning that $T_i$ will see $T_j$'s updates and $T_j$ will not see $T_i$'s.

**Correctness:** it must appear as if the transactions have been executed sequentially (in some order).

# Concurrency Control

- we fix a database: a set of objects that can be read and written by transactions:

  ⇒ $r_i[x]$: transaction $T_i$ reads object $x$
  ⇒ $w_i[x]$: transaction $T_i$ writes (modifies) object $x$

- a transaction $T_i$ is a sequence of operations

$$T_i = r_i[x_1], r_i[x_2], w_i[x_1], \ldots, r_i[x_4], w_i[x_2], c_i$$

  where $c_i$ is the **commit request** of $T_i$.

- for a **set of transactions** $T_1, \ldots, T_k$ we want to produce a *schedule* $S$ of operations such that

  ⇒ every operation $o_i \in T_i$ appears also in $S$
  ⇒ operations in $S$ are ordered the same way as in $T_i$

- goal is to produce *correct* scheduled with maximal parallelism.

# Serializable Schedules

**Definition:**
An execution of is said to be **serializable** if it is equivalent to a serial execution of the same transactions.

**Example:**

- An interleaved execution of two transactions:

$$S_a = w_1[x]\, r_2[x]\, w_1[y]\, r_2[y]$$

- An equivalent serial execution ($T_1$ , $T_2$):

$$S_b = \ w_1[x]\, w_1[y]\, r_2[x]\, r_2[y]$$

- An interleaved execution with no equivalent serial execution:

$$S_c = \ w_1[x]\, r_2[x]\, r_2[y]\, w_1[y]$$

Here, $S_a$ is serializable because it is equivalent to $S_b$ , a serial schedule. $S_c$ is not serializable.

# Conflict Equivalence

How do we determine if two schedules are *equivalent*?

⇒ cannot be based on any particular database instance

⇒ idea of **conflict equivalence**:

- two operations **conflict** if they

  (1) belong to different transactions
  (2) access the same data item $x$
  (3) at least one of them is a write operation $w[x]$.

- we require that in two *conflict-equivalent histories* all *conflicting* operations are ordered the same way.

  ⇒ guarantees to preserve computational effects
  ⇒ nonconflicting operations
  can be scheduled arbitrarily

- leads to **conflict-serializable** schedules

  ⇒ *conflict-equivalent* to a serial schedule

# Serialization Graph

How do we test if a schedule is conflict equivalent to a serial schedule?

- A **serialization graph** $SG(S)$ for a schedule $S$ is a directed graph with nodes labeled by transactions such that

  $$T_i \to T_j \in SG(S) \text{ iff } o_i[x] \text{ precedes } o_j[x] \text{ in } S$$

  where $o_i[x]$ and $o_j[x]$ are conflicting operations.

- **Theorem:**
  A schedule $S$ is serializable if and only if $SG(S)$ is acyclic graph.

# Other Properties of Schedules

Serializability guarantees correctness. However, we'd like to avoid other **unpleasant** situations:

**Recoverable Schedules:** Assume that a transaction $T_j$ *reads* a value $T_i$ has written, $T_j$ succeeds to **commit**, and $T_i$ tries to abort (in this order)

$\Rightarrow$ to abort $T_2$ we need to *undo* effects of a *committed* transaction $T_1$.

To avoid this problem:
$\Rightarrow$ commits in the order of the read-from dependency.

**Cascadeless Schedules:** if $T_j$ above didn't commit we can abort it: this may lead to *cascading aborts* of many transactions

To avoid this problem:
$\Rightarrow$ no reading of uncommitted data

# How to Get a Serializable Schedule?

So how do we build schedulers that produce serializable and cascadeless schedules?

The **scheduler** receives requests from the query processor(s). For each operation it chooses one of the following actions:

1. execute it (by sending to a lower module),
2. delay it (by inserting in some queue), or
3. reject it (thereby causing abort of the transaction)
4. ignore it (as it has no effect)

Two main kinds of schedulers:

⇒ conservative (favors delaying operations)
⇒ aggressive (favors rejecting operations)

# Locking

Before a transaction may read or write an object, it must have a lock on that object:

- a **shared lock** is required to read an object
- an **exclusive lock** is required to write an object

When a transaction cannot obtain a lock, it is **blocked** (made to wait) until the lock can be obtained.

There is no "lock" command in SQL. Instead, locks are acquired automatically by the database system.

# Locking Example

**Example:**

- $T_1$ reads object $x$
- $T_2$ attempts to write $x$

$T_2$ cannot be given the necessary lock on $x$ because of the rule prohibiting a shared and exclusive lock on the same object by different transactions.

In the example above, $T_2$ will have to wait until $T_1$ commits or aborts.

**WARNING:** obtaining a lock only for duration of a single operation in a transaction is **not sufficient**:

$\Rightarrow$ a **locking protocol** to guarantee serializability.

# Two Phase Locking (2PL)

The database system uses the following rules when acquiring locks for transactions:

- If two or more transactions hold locks on the same object, those locks must all be shared locks.
- A transaction has to **acquire** all locks before it **releases** any of them.

  $\Rightarrow$ often delayed till the transaction tries to commit or abort, i.e., until it is finished.

This algorithm is called **two-phase locking** (strict 2PL, if all locks are held till commit/abort).

**Theorem:**
Two-phase locking guarantees that the produced transaction schedules are serializable.

# Deadlocks and What to do

With 2PL we may end with a **deadlock**:

- $T_1$ reads object $x$
- $T_2$ reads object $y$
- $T_2$ attempts to write object $x$ (it is blocked)
- $T_1$ attempts to write object $y$ (it is blocked)

How do we deal with this:

1. deadlock prevention:

   $\Rightarrow$ locks granted only if they can't lead to a deadlock.
   $\Rightarrow$ ordered data items and locks granted in this order.

2. deadlock detection:

   $\Rightarrow$ wait for graphs and cycle detection.
   $\Rightarrow$ resolution: the system **aborts** one of
   the offending transactions (involuntary abort).

# Variations on Locking

- Multi-granularity Locking

  ⇒ not all locked objects have the same size
  ⇒ advantageous in presence of bulk vs. tiny updates

- Predicate Locking

  ⇒ locks based on selection predicate rather than on a value

- Tree Locking

  ⇒ tries to avoid congestion in roots of (B-)trees
  ⇒ allows relaxation of 2PL due to tree structure of data

- Lock Upgrade protocols
- . . .

# Inserts and Deletes

We have been assuming a **fixed set** of data items.

⇒ what if we try to *insert* or *delete* an item?

- does plain 2PL handle this situation? NO:

  ⇒ one transaction tries to count records in a table
  ⇒ second transactions adds/ deletes a record

- the **phantom problem**. Solutions:

  ⇒ operations that ask for “all records” have to lock against insertion/deletion of a qualifying record
  ⇒ locks on *control information*
  ⇒ index locking and other techniques

# Timestamps

What if we don't want transaction to wait (ever)?

**Idea:**

1. give each object $x$ a **read timestamp** $RTS(x)$ and a **write timestamp** $WTS(x)$.
2. give each transaction $T$ a timestamp $TS(T)$.

Timestamps must be assigned to transactions in an **increasing order** (e.g., time of arrival).

If two transactions conflict on an object then the operation from the transaction with lower timestamp must be first.

$\Rightarrow$ guarantees all conflicting op's ordered the same way
$\Rightarrow$ conflict-serializable

# Timestamps (read)

$T$ wants to read object $x$. Two possibilities:

- $TS(T) < WTS(x)$: this violates the timestamp ordering (w.r.t. the transaction that wrote $x$).

  $\Rightarrow$ abort $T$.

- $TS(T) > WTS(x)$:

  $\Rightarrow$ read $x$

  $\Rightarrow$ set $RTS(x) := \max(TS(T), RTS(x))$.

# Timestamps (write)

$T$ wants to write object $x$. Two possibilities:

- $TS(T) < RTS(x)$: this violates the timestamp ordering (w.r.t. the transaction that wrote $x$).

  $\Rightarrow$ abort $T$.

- $TS(T) < WTS(x)$: violates timestamp order

  $\Rightarrow$ the value to be written was already overwritten
  
  $\Rightarrow$ we just ignore the request

- else:

  $\Rightarrow$ write $x$
  
  $\Rightarrow$ set $WTS(x) := \max(TS(T), WTS(x))$.

# Isolation Levels in SQL

For some applications, the guarantee of serializable executions may carry a heavy price. Performance may be poor because of blocked transactions and deadlocks.

Four **isolation levels** are supported, with the highest being serializability:

- Level 3: (Serializability)
- Level 2: (Repeatable Read)
  $\Rightarrow$ Serializable except for insertions and deletions
  $\Rightarrow$ "phantom tuples" may occur

# Isolation Levels (cont.)

- Level 1: (Cursor Stability)
  - ⇒ exclusive locks held until the end of the transaction
  - ⇒ shared locks not held until the end
  - ⇒ non-repeatable reads are possible: reading the same object twice may give in different values
- Level 0:
  - ⇒ neither read nor write locks are acquired
  - ⇒ transaction may read uncommitted updates
  - ⇒ no updates, insertions, or deletions are permitted

# Summary

- ACID properties of transactions guarantee correctness of concurrent access to the database and of data storage.
- correctness of concurrent access is based on the notion of **serializability**

  ⇒ leads to definition of correct **schedulers**
- additional properties of schedules:

  ⇒ recoverability and cascadelessness
- many ways to implement a correct scheduler:

  ⇒ conservative: locking (2PL)
  with deadlock prevention
  with deadlock detection

  ⇒ aggressive: timestamps
- schedulers that may *abort* transactions rely on the **recovery manager**