# Introduction to Transaction Management

**UVic C SC 370**

Dr. Daniel M. German

*Department of Computer Science*

University of Victoria

June 10, 2003 Version: 1.1.0

# Overview

- What is a transaction?
- What properties transactions have?
- Why do we want to interleave transactions?
- How does the DMBS deal with transactions?
- How do we use transactions from SQL?

# Transactions

- Concurrent execution of user programs is essential for good DBMS performance.
- Because disk accesses are frequent, and relatively slow, it is important to keep the cpu humming by working on several user programs concurrently.
- A user's program may carry out many operations on the data retrieved from the database, but the DBMS is only concerned about what data is read/written from/to the database.
- A **transaction** is the DBMS's abstract view of a user program: a sequence of reads and writes.

# ACID

- The DBMS must ensure 4 important properties of transactions:
  1. Transactions should be **atomic**. Either they happen or they don't happen at all.
  2. Each transaction, run by itself, alone, should preserve the **consistency** of the database. The DBMS assumes that consistency holds for each transaction.
  3. **Isolation**: Transactions are isolated from the effect of other transactions that might be executed concurrently
  4. **Durability**: Once the user is notified that the transaction was successful, its effects should persist even if the system crashes.

# Consistency

- Users are responsible for the consistency of their transactions
- Each transaction must leave the database in a consistent state if the database is consistent when the transaction begins.
- The DBMS will enforce ICs and other constraints
- Beyond this, the database does not really understand the semantics of the data. (e.g., it does not understand how the interest on a bank account is computed).
- **Database consistency** is the property that every transaction sees a consistent database instance

# Isolation

- Users submit transactions, and can think of each transaction as executing by itself.
- Concurrency is achieved by the DBMS, which interleaves actions (reads/writes of DB objects) of various transactions.
- The net effect of several transactions should be the same as if they are executed one after another

# Atomicity

- If a transaction ends, we say its **commits**, otherwise it **aborts**
- Transactions can be incomplete for three reasons:
  1. It can be **aborted** by the DBMS
  2. A system crash
  3. The transaction aborts itself
- When a transaction does not commit, its partial effects should be undone
- Users can then forget about dealing with incomplete transactions
- But if it is committed it should be durable
- The DBMS uses a **log** to ensure that incomplete transactions can be undone, if necessary

# Schedules

- A transaction is seen by the DBMS as a **series** (or list) of **actions**
- These actions are **reads** or **writes** of an object: $R_T(O), W_T(O)$
- In addition to reading and writing, a transaction should specify **commit** or **abort** at the end: $Commit_T, Abort_T$
- Assumptions:
  - Transactions only interact with each other through reads/writes
  - A database is a $fixed$ collection of $independent$ objects
- A **schedule** is a list of actions (read, write, abort, commit) for a set of transactions, and the order in which they happen in the schedule is the same as in the transaction

# Schedules...

- A schedule is a potential execution sequence of a set of transactions
- It describes actions as seen by the DBMS:

| $T_1$ | $T_2$ |
|---|---|
| R(A) | |
| W(A) | |
| R(C) | |
| W(C) | |
| Commit | |
| | R(B) |
| | W(B) |
| | Abort |

- If the actions are not interleaved, it is called a **serial schedule**.

# Serializable Schedules

- A **serializable schedule** of a set of S transactions is a schedule identical to a serial schedule of the same set of transactions.

| $T_1$ | $T_2$ |
|---|---|
| R(A) | |
| W(A) | |
| | R(A) |
| | W(A) |
| R(B) | |
| W(B) | |
| | R(B) |
| | W(B) |
| | Commit |
| Commit | |

| $T_1$ | $T_2$ |
|---|---|
| R(A) | |
| W(A) | |
| | R(A) |
| R(B) | |
| W(B) | |
| | W(A) |
| | R(B) |
| | W(B) |
| | Commit |
| Commit | |

| $T_1$ | $T_2$ |
|---|---|
| R(A) | |
| W(A) | |
| R(B) | |
| W(B) | |
| Commit | |
| | R(A) |
| | W(A) |
| | R(B) |
| | W(B) |
| | Commit |

- Note: SQL programmers can instruct the database to use non-serializable schedules.

# Anomalies

- Concurrency can leave to an inconsistent state
- Two actions in the same object conflict it at least one is a write
- 3 types of anomalies (assume transactions $T_1, T_2$)
  - **Write-Read WR conflict**: $T_2$ reads data previously written by $T_1$
  - **Read-Write RW conflict**: $T_2$ writes data to something previously read by $T_1$
  - **Write-Write WW conflict**: $T_2$ writes data to something previously written by $T_1$

# WR Conflict

- $T_2$ reads data that has not been committed yet

| $T_1$ | $T_2$ |
|---|---|
| R(A) | |
| W(A) | |
| | R(A) |
| | W(A) |
| | R(B) |
| | W(B) |
| | Commit |
| R(B) | |
| W(B) | |
| Commit | |

- This situation is called a **dirty read**.

# RW Conflicts: Unrepeatable Reads

- $T_2$ changes the value of an object already read by $T_1$
- If $T_1$ tries to read it again, then it will be different
- Called **unrepeatable read**

# WW Conflicts: Overwriting Uncommitted Data

- $T_2$ overwrites the value of an object A, already modified by $T_1$, while $T_1$ is still in progress

| $T_1$ | $T_2$ |
|---|---|
| W(A) | |
| | W(A) |
| | W(B) |
| W(B) | |
| Commit | |
| | Commit |

- Writes that don't read the object are called **blind writes**

# What about aborted transactions?

- A **serializable schedule** over a set S of transactions is a schedule whose effect on any **consistent database instance** is guaranteed to be identical to that of some complete serial schedule over the set of **committed** transactions.
- This means we might have to **undo** aborted transactions
- But this is not always possible: **unrecoverable schedule**

| $T_1$ | $T_2$ |
|---|---|
| R(A) | |
| W(A) | |
| | R(A) |
| | W(A) |
| | R(B) |
| | W(B) |
| | Commit |
| Abort | |

# Recoverable Schedules

- In a **recoverable schedule** transactions **can only read** data that has been **already committed**
- There is still the situation of a **blind write**

| $T_1$ | $T_2$ |
|---|---|
| R(A) | |
| W(A) | |
| | W(A) |
| | Commit |
| Abort | |

- What should the value of A be after the abort?

# Lock Based Concurrency Control

- We use locks to guarantee recoverable schedules
- A **locking protocol** is a set of rules to be followed by each transaction (enforced by the DBMS) to ensure that, even though actions of several transactions might be **interleaved**, the net effect is executing those transactions in **some** serial order.
- We will use shared and exclusive locks

# Strict 2PL: Strict Two-Phase Locking

- A simple locking protocol with 2 rules:
  1. If a transaction T wants to read (modify) an object, it first requests a **shared** (**exclusive**) lock on that object.
  2. All locks held by a transaction are released when the transaction is completed.
- Requests to acquire and release the locks can be automatically inserted into transactions by the DBMS, the user does not have to worry
- This allows **safe** interleaving of operations
- When two transactions want to use the same object, they are serialized by the database.

# Example

- Using locks to avoid WR conflicts:

| $T_1$ | $T_2$ |
|---|---|
| X(A) | |
| R(A) | |
| W(A) | |
| | X(A) |
| X(B) | |
| R(B) | |
| W(B) | |
| Commit | |
| | R(A) |
| | W(A) |
| | X(B) |
| | R(B) |
| | W(B) |
| | Commit |

# Deadlocks

- And, of course, when we have locking, we run the risk of **deadlocks**
- The DBMS **must** either prevent or detect **deadlocks**
- The common solution: **detect**, and **resolve**
- A simple way to detect them is by using **timeouts**
- If a transaction **timeouts** then the DBMS **aborts** it

# Performance of Locking

- The more locking the lower performance in concurrent systems
- And furthermore, there is trashing
- How can we increase throughput?
  1. Lock the smallest sized objects possible
  2. Reduce the time you lock objects
  3. Reduce **hot spots** (objects that are frequently access and modified)

# Transaction Support in SQL

- A transaction is automatically created with the first statement that accesses the database or the catalogs
- Subsequent statements are considered part of the transaction until it is terminated with `COMMIT` or `ROLLBACK`.

# Transactions Characteristics

- Transactions have three special characteristics:
  1. Access mode: What type of read/write access the transaction has
  2. Isolation level: How isolated should it run?
  3. Diagnostics Size (we will not discuss this)

# Access Modes

- If the transaction is `READ ONLY`, it cannot modify the database
- Otherwise it can

# Isolation Levels

- The programmer can obtain greater concurrency at the cost of increasing the exposure to other transactions' uncommitted changes

| Level | Dirty Read | Unrepeatable Read | Phantom |
|---|---|---|---|
| `READ UNCOMMITED` | Maybe | Maybe | Maybe |
| `READ COMMITTED` | No | Maybe | Maybe |
| `REPEATABLE READ` | No | No | Maybe |
| `SERIALIZABLE` | No | No | No |