Problems Caused by Failures

- Update all account balances at a bank branch.

\[
\text{Accounts}(\text{Anum}, \text{CID}, \text{BranchId}, \text{Balance})
\]

\[
\text{update Accounts}
\]

\[
\text{set Balance} = \text{Balance} \times 1.05
\]

\[
\text{where BranchId} = 12345
\]

If the system crashes while processing this update, some, but not all, tuples with \text{BranchId} = 12345 \text{ may have been updated.}
Another Failure-Related Problem

- transfer money between accounts:
  
  \begin{verbatim}
  update Accounts
  set Balance = Balance - 100
  where Anum = 8888
  
  update Accounts
  set Balance = Balance + 100
  where Anum = 9999
  \end{verbatim}

If the system fails between these updates, money may be withdrawn but not redeposited
Problems Caused by Concurrency

• Application 1:

  ```sql
  update Accounts
  set Balance = Balance - 100
  where Anum = 8888
  ```

  ```sql
  update Accounts
  set Balance = Balance + 100
  where Anum = 9999
  ```

• Application 2:

  ```sql
  select Sum(Balance)
  from Accounts
  ```

If the applications run concurrently, the total balance returned to application 2 may be inaccurate.
Another Concurrency Problem

• Application 1:

```sql
select balance into :balance
from Accounts
where Anum = 8888
compute :newbalance using :balance
update Accounts
set Balance = :newbalance
where Anum = 8888
```

• Application 2: same as Application 1

---

If the applications run concurrently, one of the updates may be “lost”.

---
Transaction Properties

- Transactions are *durable, atomic* application-specified units of work.

 **Atomic**: indivisible, all-or-nothing.

 **Durable**: effects survive failures.

---

**A** tomic: a transaction occurs entirely, or not at all

**C** onsistent

**I** solated: a transaction’s unfinished changes are not visible to others

**D** urable: once it is complete, a transaction’s changes are permanent

---
Serializability (informal)

- Concurrent transactions must appear to have been executed sequentially, i.e., one at a time, in some order. If $T_i$ and $T_j$ are concurrent transactions, then either:
  - $T_i$ will appear to precede $T_j$, meaning that $T_j$ will “see” any 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.
Serializability: An Example

- An interleaved execution of two transactions, $T_1$ and $T_2$:
  \[ H_a = w_1[x] r_2[x] w_1[y] r_2[y] \]

- An equivalent serial execution of $T_1$ and $T_2$:
  \[ H_b = w_1[x] w_1[y] r_2[x] r_2[y] \]

- An interleaved execution of $T_1$ and $T_2$ with no equivalent serial execution:
  \[ H_c = w_1[x] r_2[x] r_2[y] w_1[y] \]

---

$H_a$ is serializable because it is equivalent to $H_b$, a serial schedule. $H_c$ is not serializable.
Transactions and Histories

• Two operations conflict if:
  – they belong to different transactions
  – they operate on the same object
  – at least one of the operations is a write

• A transaction is a sequence of read and write operations.

• An execution history over a set of transactions $T_1 \ldots T_n$ is an interleaving of the the operations of $T_1 \ldots T_n$ in which the operation ordering imposed by each transaction is preserved.
Serializability

- Two histories are *(conflict) equivalent* if
  - they are over the same set of transactions, and
  - the ordering of each pair of conflicting operations is the same in each history
- A history $H$ is said to be *(conflict) serializable* if there exists some *serial* history $H'$ that is *(conflict) equivalent* to $H$
Testing for Serializability

Is this history serializable?

A history is serializable iff its serialization graph is acyclic.
Serialization Graphs

\[ r_1[x] r_3[x] w_4[y] r_2[u] w_4[z] r_1[y] r_3[u] r_2[z] w_2[z] r_3[z] r_1[z] w_3[y] \]
Serialization Graphs (cont’d)

The history above is equivalent to

\[ r_1[x] r_3[x] w_4[y] r_2[u] w_4[z] r_1[y] r_3[u] r_2[z] w_2[z] r_3[z] r_1[z] w_3[y] \]

That is, it is equivalent to executing \( T_4 \) followed by \( T_2 \) followed by \( T_1 \) followed by \( T_3 \).
Abort and Commit

• A transaction may terminate in one of two ways:
  – 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 may have made are undone (erased), as if the transaction never ran at all. An abort is the “nothing” in “all-or-nothing” execution.

• A transaction that has started but has not yet aborted or committed is said to be active.
Transactions in SQL

- A new transaction is begun when an application first executes an SQL command.
- Two SQL commands are available to terminate a transaction:
  - `commit work`: commits the transaction
  - `rollback work`: abort the transaction
- A new transaction begins with the application’s next SQL command after `commit work` or `rollback work`.
Implementing Transactions

- The implementation of transactions in a DBMS has two parts:
  
  **Concurrency Control:** guarantees that the execution history has the desired properties (such as serializability)

  **Recovery Management:** guarantees that committed transactions are durable (despite failures), and that aborted transactions have no effect on the database
Concurrency Control

• Serializability can be guaranteed by executing transactions serially, but in many environments this leads to poor performance.

• Typically, many transactions are in progress concurrently, and a concurrency control protocol is used to ensure that the resulting history is serializable.

• Many concurrency control protocols have been proposed, based on:
  – locking, or
  – timestamps, or
  – (optimistic) conflict detection, or . . .
Two-Phase Locking

- The rules
  1. 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
  2. Two or more transactions may not hold locks on the same object unless all hold shared locks.
  3. Once a transaction has released (unlocked) any object, it may not obtain any new locks.

If all transactions use two-phase locking, the execution history is guaranteed to be serializable.
**Strict Two-Phase Locking**

- Most systems implement a somewhat stronger protocol, called *strict two-phase locking*. It adds one more rule:
  - A transaction may not release any locks until it commits (or aborts)

If all transactions use strict two-phase locking, the execution history is guaranteed to be both serializable and strict.
Transaction Blocking

- Consider the following sequence of events:
  - $T_1$ acquires a shared lock on $x$ and reads $x$
  - $T_2$ attempts to acquire an exclusive lock on $x$ (so that it can write $x$)

- The two-phase locking rules prevent $T_2$ from acquiring its exclusive lock - this is called a lock conflict.

- Lock conflicts can be resolved in one of two ways:
  1. $T_2$ can be blocked - forced to wait until $T_1$ releases its lock
  2. $T_1$ can be pre-empted - forced to abort and give up its locks
Deadlocks

- transaction blocking can result in *deadlocks* For example:
  - $T_1$ 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)

---

A deadlock can be resolved only by forcing one of the transactions involved in the deadlock to abort.
Strict 2PL Example

requests : \( r_1[x] \) \( r_2[y] \)
schedule : \( r_1[x] \) \( r_2[y] \)

requests : \( r_1[x] \) \( r_2[y] \) \( w_3[x] \) \( w_2[y] \)
schedule : \( r_1[x] \) \( r_2[y] \) \( w_2[y] \)

requests : \( r_1[x] \) \( r_2[y] \) \( w_3[x] \) \( w_2[y] \) \( r_2[z] \) \( w_1[z] \) \( r_4[x] \)
schedule : \( r_1[x] \) \( r_2[y] \) \( w_2[y] \) \( r_2[z] \)

CS743 DB Management and Use Fall 2014
Strict 2PL Example (cont’d)

requests : \[ r_1[x] \; r_2[y] \; w_3[x] \; w_2[y] \; r_2[z] \; w_1[z] \; r_4[x] \; c_2 \]

schedule : \[ r_1[x] \; r_2[y] \; w_2[y] \; r_2[z] \; c_2 \; w_1[z] \]

requests : \[ r_1[x] \; r_2[y] \; w_3[x] \; w_2[y] \; r_2[z] \; w_1[z] \; r_4[x] \; c_2 \; c_1 \]

schedule : \[ r_1[x] \; r_2[y] \; w_2[y] \; r_2[z] \; c_2 \; w_1[z] \; c_1 \; w_3[x] \]

requests : \[ r_1[x] \; r_2[y] \; w_3[x] \; w_2[y] \; r_2[z] \; w_1[z] \; r_4[x] \; c_2 \; c_1 \; a_3 \; r_4[y] \; c_4 \]

schedule : \[ r_1[x] \; r_2[y] \; w_2[y] \; r_2[z] \; c_2 \; w_1[z] \; c_1 \; w_3[x] \; a_3 \; r_4[x] \; r_4[y] \; c_4 \]