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.
Concurrency in a DBMS

- 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.
  - Each transaction must leave the database in a consistent state if the DB is consistent when the transaction begins.
    - DBMS will enforce some ICs, depending on the ICs declared in CREATE TABLE statements.
    - Beyond this, the DBMS does not really understand the semantics of the data. (e.g., it does not understand how the interest on a bank account is computed).

**Issues:** Effect of *interleaving* transactions, and crashes.
Atomicity of Transactions

- A transaction might commit after completing all its actions, or it could abort (or be aborted by the DBMS) after executing some actions.
- A very important property guaranteed by the DBMS for all transactions is that they are atomic. That is, a user can think of a Xact as always executing all its actions in one step, or not executing any actions at all.
  - DBMS logs all actions so that it can undo the actions of aborted transactions.
Example

- Consider two transactions (*Xacts*):
  
<table>
<thead>
<tr>
<th>T1:</th>
<th>BEGIN A=A+100, B=B-100 END</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2:</td>
<td>BEGIN A=1.06<em>A, B=1.06</em>B END</td>
</tr>
</tbody>
</table>

- Intuitively, the first transaction is transferring $100 from B’s account to A’s account. The second is crediting both accounts with a 6% interest payment.

- There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together. However, the net effect *must* be equivalent to these two transactions running serially in some order.
Example (Contd.)

❖ Consider a possible interleaving (schedule):

| T1:  | A = A + 100, B = B - 100 |
| T2:  | A = 1.06 * A, B = 1.06 * B |

❖ This is OK. But what about:

| T1:  | A = A + 100, B = B - 100 |
| T2:  | A = 1.06 * A, B = 1.06 * B |

❖ The DBMS’s view of the second schedule:

| T1:  | R(A), W(A), R(B), W(B) |
| T2:  | R(A), W(A), R(B), W(B) |
Scheduling Transactions

- **Serial schedule:** Schedule that does not interleave the actions of different transactions.

- **Equivalent schedules:** For any database state, the effect (on the set of objects in the database) of executing the first schedule is identical to the effect of executing the second schedule.

- **Serializable schedule:** A schedule that is equivalent to some serial execution of the transactions.

(Note: If each transaction preserves consistency, every serializable schedule preserves consistency.)
Anomalies with Interleaved Execution

- **Reading Uncommitted Data (WR Conflicts, "dirty reads"):**

| T1: | R(A), W(A), R(B), W(B), Abort |
| T2: | R(A), W(A), C |

- **Unrepeatable Reads (RW Conflicts):**

| T1: | R(A), R(A), W(A), C |
| T2: | R(A), W(A), C |
Anomalies (Continued)

❖ Overwriting Uncommitted Data (WW Conflicts):

| T1: | W(A), W(B), C |
| T2: | W(A), W(B), C |
**Lock-Based Concurrency Control**

- **Strict Two-phase Locking (Strict 2PL) Protocol:**
  - Each Xact must obtain a \( S \) (shared) lock on object before reading, and an \( X \) (exclusive) lock on object before before writing.
  - All locks held by a transaction are released when the transaction completes
    - (Non-strict) 2PL Variant: Release locks anytime, but cannot acquire locks after releasing any lock.
  - If an Xact holds an \( X \) lock on an object, no other Xact can get a lock (\( S \) or \( X \)) on that object.

- **Strict 2PL allows only serializable schedules.**
  - Additionally, it simplifies transaction aborts
  - (Non-strict) 2PL also allows only serializable schedules, but involves more complex abort processing
Aborting a Transaction

- If a transaction $Ti$ is aborted, all its actions have to be undone. Not only that, if $Tj$ reads an object last written by $Ti$, $Tj$ must be aborted as well!

- Most systems avoid such cascading aborts by releasing a transaction’s locks only at commit time.
  - If $Ti$ writes an object, $Tj$ can read this only after $Ti$ commits.

- In order to undo the actions of an aborted transaction, the DBMS maintains a log in which every write is recorded. This mechanism is also used to recover from system crashes: all active Xacts at the time of the crash are aborted when the system comes back up.
The Log

- The following actions are recorded in the log:
  - *Ti writes an object:* the old value and the new value.
    - Log record must go to disk *before* the changed page!
  - *Ti commits/aborts:* a log record indicating this action.

- Log records are chained together by Xact id, so it’s easy to undo a specific Xact.

- Log is often *duplexed* and *archived* on stable storage.

- All log related activities (and in fact, all CC related activities such as lock/unlock, dealing with deadlocks etc.) are handled transparently by the DBMS.
Recovering From a Crash (Write-ahead Logging)

- There are 3 phases in the *Aries* recovery algorithm:
  - **Analysis:** Scan the log forward (from the most recent checkpoint) to identify all Xacts that were active, and all dirty pages in the buffer pool at the time of the crash.
  - **Redo:** Redoes all updates to dirty pages in the buffer pool, as needed, to ensure that all logged updates are in fact carried out and written to disk.
  - **Undo:** The writes of all Xacts that were active at the crash are undone (by restoring the *before value* of the update, which is in the log record for the update), working backwards in the log. (Some care must be taken to handle the case of a crash occurring during the recovery process!)
Summary

Concurrency control and recovery are among the most important functions provided by a DBMS.

- Users need not worry about concurrency.
  - System automatically inserts lock/unlock requests and schedules actions of different Xacts in such a way as to ensure that the resulting execution is equivalent to executing the Xacts one after the other in some order.

- Write-ahead logging (WAL) is used to undo the actions of aborted transactions and to restore the system to a consistent state after a crash.
  - Consistent state: Only the effects of committed Xacts seen.