Goals for Today

- Understand the challenges that **concurrent access** to a DBMS pose for data consistency
- Reason about which actions on data can **conflict** and the possible implications
- Learn strategies for dealing with concurrent access

**Halloween Problem!**

```
UPDATE Employees
SET salary = salary * 1.1
WHERE salary < 10000;
```

**Simplified RDBMS Architecture**
Querying a DBMS (Example Architectures)

- **Client-Server Arch**: SQL → SQL
- **Three-Tier Arch**: SQL → HTTP → SQL

Transactions

- **Transaction (xact)**: an atomic sequence of read/write actions on the database
- Moves the database from one **consistent** state to another
- Final action is **commit** or **abort**

Example: Transferring Money

- **Checking**: $200 → **Transaction** → $300
- **Savings**: $1000 → $900
- In this example, **consistency** is based on knowledge of banking semantics
- In general, up to the writer of the transaction to ensure a transaction preserves consistency
  - DBMS provides (limited) automatic enforcement, via **integrity constraints**
  - e.g., account balances must be $\geq 0$

Concurrent execution

- **Q. Why not just run queries one at a time?**
- Concurrent execution is essential for good DBMS performance

**Important**: keep the CPU busy while slow disk accesses are occurring

These two can happen at the same time!

Reading a page from disk

Evaluating WHERE condition(s) on page in buffer pool

Note: 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.
Example: Transaction SQL

```
BEGIN;  --BEGIN TRANSACTION

UPDATE accounts
SET balance = balance - 100.00
WHERE account_num = SavingsAccountNum
  AND user_id = 18;

UPDATE accounts
SET balance = balance + 100.00
WHERE account_num = CheckingAccountNum
  AND user_id = 18;

COMMIT;  --COMMIT WORK
```

Concurrence in a DBMS

- Concurrency is achieved by the DBMS, which *interleaves actions* (reads/writes of DB "objects") of multiple transactions

- **Issues:**
  - Effect of *interleaving* transactions
  - System *crashes*

Example: Concurrency Outcomes

- Consider two transactions (*xacts*):
  - **T1:**
    - BEGIN
    - A = A + 100
    - B = B - 100
    - END
  - **T2:**
    - BEGIN
    - A = 1.06 * A
    - B = 1.06 * B
    - END

  - **We’ll often use letters like A and B to refer to database “objects”**
  - T1 transfers $100 from account B to A
  - T2 credits both accounts with 6% interest

- Assume at first accounts A and B each contain $1000
  - Q. What is a legal outcome for A and B after running T1 and T2?
    - A + B should add up to $2000 * 1.06 = $2120

- If T1 and T2 submitted at the same time, there is **no guarantee that T1 will execute before T2 or vice-versa.**

  *Consistency: the net effect must be equivalent to* these two transactions running *serially in some order.*

Example: Concurrency Outcomes

- Consider a possible *interleaved schedule*:
  - **T1:** A = A + 100,  B = B - 100
  - **T2:** A = 1.06 * A,  B = 1.06 * B

  - **This is OK** (result same as T1;T2)

- **But what about:**
  - **T1:** A = A + 100,  B = B - 100
  - **T2:** A = 1.06 * A,  B = 1.06 * B

  - **Result:** A = 1166,  B = 960;  A + B = 2126 → Bank loses $6!

- **The DBMS’ s view of the second schedule:**

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A),</td>
<td>R(B),</td>
<td></td>
</tr>
<tr>
<td>W(A),</td>
<td>W(B)</td>
<td></td>
</tr>
<tr>
<td>R(A),</td>
<td>W(A),</td>
<td>R(B),</td>
</tr>
<tr>
<td>W(B)</td>
<td></td>
<td>W(B)</td>
</tr>
</tbody>
</table>

  - What went wrong??
ACID: Transaction Atomicity

- A transaction ends in one of two ways:
  - It **commits** after completing all its actions
  - or it could **abort** (self-inflicted or by the DBMS) after executing some actions

- User expectation: **atomic transactions**
  - a transaction must either execute all its actions, or not execute any actions at all

Wait, what?! What if it already started making changes to the database?

Later: logging and recovery

ACID: Transaction Isolation

- Transactions must be protected from concurrent access

  - **Isolation**: each xact executes as if it was running by itself
    - Concurrency is achieved by DBMS, which interleaves actions (reads/writes of DB objects) of multiple transactions

- Many techniques for isolation, two basic categories:
  - **Pessimistic** – don’t let problems arise in the first place
  - **Optimistic** – assume conflicts are rare, deal with them after they happen

ACID: Transaction Consistency

- **Consistency**: the data in the DBMS is accurate in modeling real world, follows appropriate integrity constraints

  - The user must ensure a transaction maintains consistency!

- **DBMS Guarantee**: if DBMS is consistent before transaction, it will still be consistent after the transaction completes

  - DBMS checks integrity constraints and if they fail, the transaction rolls back (i.e., is **aborted**)

ACID: Transaction Durability

(Recovering From a Crash)

- **Failure** scenarios
  - System crash
    - Data/updates in memory are lost, hard disk okay
    - This is the case we will look at when we cover recovery
  - Hard Disk crash
    - Need backups, RAID and data replication can help

- **Durability**: all updates from committed transactions and only those updates will be reflected in the database

Image: [http://www.clker.com/cliparts/b/w/I/b/F/8/half-full-half-empty-md.png](http://www.clker.com/cliparts/b/w/I/b/F/8/half-full-half-empty-md.png)
A.C.I.D. Properties of Transactions

Atomicity:
All actions in the transaction happen, or none happen.

Consistency:
If each transaction is consistent, and the DB starts consistent, it ends up consistent.

Isolation:
Execution of one transaction is isolated from that of all others.

Durability:
If a transaction commits, its effects persist.

Concurrency Control

• Now: focus on the “I” (isolation) part

• Later: when we talk about recovery, we’ll get to the “A” (atomicity) and “D” (durability)

What about “C” ??
If the system can achieve guarantees for A, I, and D, then we get C for free!

Serial and Equivalent Schedules

• Serial schedule: A schedule that does not interleave the actions of different transactions.
  – i.e., you run the transactions serially (one at a time)

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

Serializable Schedules

• Serializable schedule: A schedule that is equivalent to some serial execution of the transactions.
  – Intuition: with a serializable schedule you only see things that could happen in situations where you were running transactions one-at-a-time.
All About Conflict

- **Conflicting actions**
  - Two actions from different transactions on the same data objects conflict if at least one of the actions is a write

- Two schedules are conflict equivalent iff:
  - They involve the same actions of the same transactions
  - Every pair of conflicting actions is ordered the same way

- Schedule $S$ is conflict serializable if $S$ is conflict equivalent to some serial schedule

Note: a pair of conflicting actions does not always mean a “problem” (or that we care)

Precedence Graph

- Node = transaction
- Directed edges:
  - Edge from $T_i$ to $T_j$ if an action in $T_i$ precedes and conflicts with an action in $T_j$

- **Theorem**: Schedule is conflict serializable if and only if its precedence graph is acyclic

Anomalies from Interleaved Execution

**Unrepeatable Reads (RW conflict):**

- $T_1$: $R(A)$, $W(A)$, Commit
- $T_2$: $R(A)$, $W(A)$, Commit

**Reading Uncommitted Data (‘dirty reads’, WR conflict):**

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

**Overwriting Uncommitted Data: (“lost update”, WW conflict)**

- $T_1$: $W(A)$, $W(B)$, Commit
- $T_2$: $W(A)$, $W(B)$, Commit

Example: Bank Concurrency Schedule

- A schedule that is not conflict serializable (earlier banking example):

- The cycle in the graph reveals the problem: The output of $T_1$ depends on $T_2$, and vice-versa
Example: Bank Concurrency Schedule

• A schedule that IS conflict serializable:

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

T1:
R(A), W(A), R(B), W(B)

T2:
R(B), R(A), R(B), W(B)

No cycle here!

Notes on Conflict Serializability

• Conflict Serializability does not allow all schedules that you would consider correct
  – This is because it is strictly syntactic; it doesn’t consider the meanings of the operations or the data.

| T1:  | R(A), A=A-50, W(A) | R(B), B=B+50, W(B) |
| T2:  | R(B), B=B-10, W(B) | R(A), A=A+10, W(A) |

Same result as the serial schedule T1, T2 (addition commutative)

• In practice, conflict serializability is what gets used, because it can be done efficiently
  – Note: in order to allow more concurrency, some special cases do get implemented, such as for travel reservations, etc.

Locks

• We use locks to control access to objects

• Shared (S) locks – multiple transactions can hold these on a particular object at the same time.

• Exclusive (X) locks – only one of these and no other locks, can be held on a particular object at a time.

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>X</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

Lock Compatibility Matrix

Basic Locking: Example

A= 1000, B=2000, Output from T2’s print =?

T1

<table>
<thead>
<tr>
<th>Lock_X(A) &lt;granted&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(A)</td>
</tr>
<tr>
<td>A = A-50</td>
</tr>
<tr>
<td>Write(A)</td>
</tr>
<tr>
<td>Unlock(A)</td>
</tr>
</tbody>
</table>

T2

| Read(B)             |
|                     |
|                     |
| Unlock(B)           |
| PRINT(A+B)          |
Two-Phase Locking (2PL)

1) Each transaction must obtain:
   - a S (shared) or an X (exclusive) lock on object before reading
   - an X (exclusive) lock on object before writing

2) All lock requests precede all unlock requests!
   I.e., a transaction can not request additional locks once it releases any locks

Each transaction has a “growing phase” followed by a “shrinking phase”

---

Basic Locking: Example (Take 2)

\[ A = 1000, B = 2000, \text{Output} =? \]

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock_X(A)</td>
<td>Lock_S(A)</td>
</tr>
<tr>
<td>Read(A)</td>
<td></td>
</tr>
<tr>
<td>( A = A - 50 )</td>
<td></td>
</tr>
<tr>
<td>Write(A)</td>
<td></td>
</tr>
<tr>
<td>Lock_X(B)</td>
<td>&lt;granted&gt;</td>
</tr>
<tr>
<td>Unlock(A)</td>
<td>&lt;granted&gt;</td>
</tr>
<tr>
<td>Read(A)</td>
<td></td>
</tr>
<tr>
<td>Lock_S(B)</td>
<td></td>
</tr>
<tr>
<td>Read(B)</td>
<td></td>
</tr>
<tr>
<td>( B = B + 50 )</td>
<td></td>
</tr>
<tr>
<td>Write(B)</td>
<td></td>
</tr>
<tr>
<td>Unlock(B)</td>
<td>&lt;granted&gt;</td>
</tr>
<tr>
<td>Unlock(A)</td>
<td></td>
</tr>
<tr>
<td>Read(B)</td>
<td></td>
</tr>
<tr>
<td>Unlock(B)</td>
<td></td>
</tr>
<tr>
<td>PRINT(A + B)</td>
<td></td>
</tr>
</tbody>
</table>