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

---

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.
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

- 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 $\geq 0$

Example: Transaction in 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
```

Concurrency in a DBMS

- Concurrency is achieved by the DBMS, which **interleave 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
  ```

- 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?

  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

  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:

<table>
<thead>
<tr>
<th>T1:</th>
<th>T2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>A=A+100, B=B-100</td>
<td>A=1.06<em>A, B=1.06</em>B</td>
</tr>
</tbody>
</table>

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

- But what about:

<table>
<thead>
<tr>
<th>T1:</th>
<th>T2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>A=A+100, B=B-100</td>
<td>A=1.06<em>A, B=1.06</em>B</td>
</tr>
</tbody>
</table>

  **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>
</tr>
</thead>
<tbody>
<tr>
<td>R(A), W(A), R(B), W(B)</td>
<td>R(A), W(A), R(B), W(B)</td>
</tr>
</tbody>
</table>

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 Consistency

- **Consistency:** the data in the DBMS is accurate in modeling the 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 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 Durability (Recovering From a Crash)

- **Failure** scenarios
  - System crash
    - Data/updates in memory are lost, hard disk is 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

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., transactions run **serially** (one at a time)

• **Equivalent schedules**: Given two 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.

<table>
<thead>
<tr>
<th>S1:</th>
<th>T1: R(A), W(A)</th>
<th>S3:</th>
<th>T1: R(A), W(A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2: R(A), W(B)</td>
<td>T2: R(A), W(B)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Try Exercise 2

(a) yes, both T2, T1 and T1, T2
(b) yes, only T2, T1
(c) no

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.

<table>
<thead>
<tr>
<th>S1:</th>
<th>T1: R(A), W(A)</th>
<th>S3:</th>
<th>T1: R(A), W(A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2: R(A), W(B)</td>
<td>T2: R(A), W(B)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

  Order of conflicting actions matters!
  - If T2’s R(A) precedes T1’s W(A), then conceptually **T2 should precede T1**

• 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)*
Anomalies from Interleaved Execution

**Unrepeatable Reads (RW conflict):**

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

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

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

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

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

Example: Bank Concurrency Schedule

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

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

- The cycle in the graph reveals the problem: The output of T1 depends on T2, and vice-versa

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

Example: Bank Concurrency Schedule

- A schedule that IS conflict serializable:

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

- No cycle here!
Try Exercise 3

(a) 
T1 R(A), T2 W(A)
T2 R(A), T1 W(A)
T1 W(A) T2 W(A)
not conflict serializable

(b) 
T1 R(A) T3 W(A)
T2 R(B), T1 W(B)
T2 W(B), T1 R(B)
T2 W(B), T1 W(B)
is conflict serializable