CS 133: Databases

Spring 2017
Lec 16 – 3/21
Transactions

Prof. Beth Trushkowsky

Warm-up Exercise
(See exercise sheet. You can start before class.)

Data corruption; Good performance from user's perspective

Administrivia

• Lab 3:
  – Exercises 1-3 due this Wednesday
  – Remaining exercises due following Wednesday

• Problem set 7 due this Thursday

• Office hour on Fridays 2:45pm instead of Mondays

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
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

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 >= 0

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
```

Example: Concurrency Outcomes

- Consider two transactions (**xacts**):
  - 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**:

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1: A=A+100, B=B-100</td>
<td>This is OK (result same as T1;T2)</td>
</tr>
<tr>
<td>T2: A=1.06<em>A, B=1.06</em>B</td>
<td></td>
</tr>
</tbody>
</table>

• But what about:

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1: A=A+100, B=B-100</td>
<td></td>
</tr>
<tr>
<td>T2: A=1.06<em>A, B=1.06</em>B</td>
<td></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>Schedule</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1: R(A), W(A), R(B), W(B)</td>
<td></td>
</tr>
<tr>
<td>T2: R(A), W(A), R(B), W(B)</td>
<td></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

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 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 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

**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)

**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.

**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.

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

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

  - Swap T1’s W(A) and T2’s W(B)?
  - Swap T1’s W(A) and T2’s R(A)?

**What about “C”??**

If the system can achieve guarantees for A, I, and D, then we get C for free!
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.

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

Swap T1's R(A) and T2's R(A)?
```

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

Swap T1's R(A) and T2's W(B)?
```

Try Exercise 2

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

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):**

```
T1: R(A), W(A), Commit  R(A), W(A), Commit
T2: R(A), W(B), Commit  R(A), W(A), Commit
```

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

```
T1: R(A), W(A), Commit  R(B), W(B), Abort
T2: R(A), W(A), Commit
```

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

```
T1: W(A), W(B), Commit  W(A), W(B), Commit
T2: W(A), W(B), Commit
```


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 **not** conflict serializable (earlier banking example):

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

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

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
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.

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.

Basic Locking: Example

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

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

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.

Lock Compatibility Matrix

<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>

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, Output =?

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