CSE 344

AUGUST 8TH

TRANSACTIONS
ADMINISTRIVIA

• HW7 due today

• HW8 out already
  • writing a DB application
  • puts together some different pieces
    • DB design, queries, transactions
  • start early (has many parts)

• WQ7 due Monday
TECH INTERVIEWS

- Guest lecture in CSE 331
  - Friday at 1:10pm in GUG 220

- Topic is tech interviews
CLASS OVERVIEW

Unit 1: Intro
Unit 2: Relational Data Models and Query Languages
Unit 3: Non-relational data
Unit 4: RDMBS internals and query optimization
Unit 5: Parallel query processing
Unit 6: DBMS usability, conceptual design
Unit 7: Transactions
  • Locking and schedules (internals), SQL transactions
  • Important for writing DB applications
APPLICATION

Examples

• Bank $$ transfers
• Online shopping
• Signing up for classes

Work consists of a large set of tasks to complete for users

• Most tasks are independent of the others
• But some are not...
  • two users trying to withdraw from the same account at once
    • (if insufficient funds are available for both, only one is allowed)
  • users trying to buy the last copy of an item for sale
  • students trying to get the last spot in a class
CHALLENGES

Want to execute many apps concurrently
  • All these apps read and write data to the same DB

Simple solution: only execute one task at a time
  • What’s the problem?

Want: multiple operations run simultaneously on the same DBMS
**TRANSACTIONS**

We use database transactions everyday

- Bank $$$ transfers
- Online shopping
- Signing up for classes

For this class, a transaction is a series of DB queries & updates

- Read / Write / Update / Delete / Insert
- Unit of work issued by a user that is independent from others
- (Note: we won’t talk about rows much here... transactions are a broader concept than databases)
WHAT CAN GO WRONG?

Manager: balance budgets among projects
- Remove $10k from project A
- Add $7k to project B
- Add $3k to project C

CEO: check company’s total balance
- SELECT SUM(money) FROM budget;

This is called a dirty / inconsistent read aka a WRITE-READ conflict
WHAT CAN GO WRONG?

App 1:
SELECT inventory FROM products WHERE pid = 1

App 2:
UPDATE products SET inventory = 0 WHERE pid = 1

App 1:
SELECT inventory * price FROM products WHERE pid = 1

This is known as an unrepeateable read aka READ-WRITE conflict
WHAT CAN GO WRONG?

Account 1 = $100
Account 2 = $100
Total = $200

• App 1:
  – Set Account 1 = $200
  – Set Account 2 = $0

• App 2:
  – Set Account 2 = $200
  – Set Account 1 = $0

• At the end:
  – Total = $200

• App 1: Set Account 1 = $200

• App 2: Set Account 2 = $200

• App 1: Set Account 2 = $0

• App 2: Set Account 1 = $0

• At the end:
  – Total = $0

This is called the lost update aka WRITE-WRITE conflict
WHAT CAN GO WRONG?

Paying for Tuition...

- Fill up form with your mailing address
- Put in debit card number (because you don’t trust the gov’t)
- Click submit
- Screen shows money deducted from your account
- [Your browser crashes]

Lesson:

Changes to the database should be ALL or NOTHING
TRANSACTIONS

Collection of statements that are executed atomically (logically speaking)

BEGIN TRANSACTION
  [SQL statements]
COMMIT or ROLLBACK (=ABORT)

If BEGIN… missing, then TXN consists of a single instruction
KNOW YOUR TRANSACTIONS: ACID

Atomic
• State shows either all the effects of txn, or none of them

Consistent
• Txn moves from a DBMS state where integrity holds, to another where integrity holds
  • remember integrity constraints?

Isolated
• Effect of txns is the same as txns running one after another (i.e., looks like batch mode)

Durable
• Once a txn has committed, its effects remain in the database
**ATOMIC**

Definition: A transaction is ATOMIC if all its updates must happen or not at all.

Example: move $100 from A to B

- UPDATE accounts SET bal = bal - 100
  WHERE acct = A;

- UPDATE accounts SET bal = bal + 100
  WHERE acct = B;

- BEGIN TRANSACTION;
  UPDATE accounts SET bal = bal - 100 WHERE acct = A;
  UPDATE accounts SET bal = bal + 100 WHERE acct = B;
  COMMIT;
ISOLATED

Definition:

- An execution ensures that transactions are isolated, if the effect of each transaction is as if it were the only transaction running on the system.
CONSISTENT

Recall: integrity constraints govern how values in tables are related to each other

- Can be enforced by the DBMS, or ensured by the app

How consistency is achieved by the app:

- App programmer ensures that txns only takes a consistent DB state to another consistent state
- DB makes sure that txns are executed atomically

Can defer checking the validity of constraints until the end of a transaction
DURABLE

A transaction is durable if its effects continue to exist after the transaction and even after the program has terminated.

How?

• By writing to disk
  • or multiple disks
• By writing to memory of multiple servers
  • geographically separated
ROLLBACK TRANSACTIONS

If the app gets to a state where it cannot complete the transaction successfully, execute ROLLBACK

The DB returns to the state prior to the transaction
  • remove the effects of any WRITEs that occurred
Again: by default each statement is its own txn

- Unless auto-commit is off then each statement starts a new txn
A schedule is a sequence of interleaved actions from all transactions.
A *serial schedule* is one in which transactions are executed one after the other, in some sequential order.

Fact: nothing can go wrong if the system executes txns *serially*.
- (rather, whatever does go wrong is the app’s fault)
- But DBMS don’t do that because we want better overall system performance.
### Example

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A, t)</td>
<td>READ(A, s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(A, t)</td>
<td>WRITE(A, s)</td>
</tr>
<tr>
<td>READ(B, t)</td>
<td>READ(B, s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(B, t)</td>
<td>WRITE(B, s)</td>
</tr>
</tbody>
</table>

A and B are elements in the database, t and s are variables in txn source code.
### EXAMPLE OF A (SERIAL) SCHEDULE

<table>
<thead>
<tr>
<th>Time</th>
<th>T1</th>
<th></th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>READ(A, t)</td>
<td></td>
<td>WRITE(A, t)</td>
</tr>
<tr>
<td></td>
<td>t := t+100</td>
<td></td>
<td>READ(B, t)</td>
</tr>
<tr>
<td></td>
<td>WRITE(B, t)</td>
<td></td>
<td>s := s*2</td>
</tr>
<tr>
<td></td>
<td>READ(A, s)</td>
<td></td>
<td>WRITE(A, s)</td>
</tr>
<tr>
<td></td>
<td>s := s*2</td>
<td></td>
<td>READ(B, s)</td>
</tr>
<tr>
<td></td>
<td>WRITE(B, s)</td>
<td></td>
<td>End with A=200 B=200</td>
</tr>
</tbody>
</table>

Starting with A=0 B=0

Starting with A=200 B=200
ANOTHER SERIAL SCHEDULE

Starting with $A=0 \ B=0$

End with $A=100 \ B=100$
A schedule is **serializable** if it is equivalent to a serial schedule

Not necessarily serial

BUT equally good from app’s perspective
### A SERIALIZABLE SCHEDULE

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A, t)</td>
<td>READ(A, s)</td>
</tr>
<tr>
<td>t := t + 100</td>
<td>s := s * 2</td>
</tr>
<tr>
<td>WRITE(A, t)</td>
<td>WRITE(A, s)</td>
</tr>
<tr>
<td>READ(B, t)</td>
<td>READ(B, s)</td>
</tr>
<tr>
<td>t := t + 100</td>
<td>s := s * 2</td>
</tr>
<tr>
<td>WRITE(B, t)</td>
<td>WRITE(B, s)</td>
</tr>
</tbody>
</table>

This is a **serializable** schedule.

This is **NOT** a serial schedule.

Starting with A=0 B=0
End with A=200 B=200
A NON-SERIALIZABLE SCHEDULE

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A, t)</td>
<td>READ(A,s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(A, t)</td>
<td>WRITE(A,s)</td>
</tr>
<tr>
<td>READ(B, t)</td>
<td>READ(B,s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(B,t)</td>
<td>WRITE(B,s)</td>
</tr>
</tbody>
</table>

Starting with A=0 B=0
End with A=200 B=100
HOW DO WE KNOW IF A SCHEDULE IS SERIALIZABLE?

Notation:

\[ T_1: r_1(A); w_1(A); r_1(B); w_1(B) \]
\[ T_2: r_2(A); w_2(A); r_2(B); w_2(B) \]

Key Idea: Focus on conflicting operations (i.e., where changing order can change result)
CONFLICTS

Write-Read – WR
Read-Write – RW
Write-Write – WW
Read-Read?
CONFLICT SERIALIZABILITY

Conflicts: (i.e., swapping will change program behavior)

Two actions by same transaction $T_i$:
- $r_i(X); w_i(Y)$

Two writes by $T_i$, $T_j$ to same element:
- $w_i(X); w_j(X)$
- $w_i(X); r_j(X)$
- $r_i(X); w_j(X)$

Read/write by $T_i$, $T_j$ to same element:
- $r_i(X); w_j(X)$
A schedule is *conflict serializable* if it can be transformed into a serial schedule by a series of swaps of adjacent non-conflicting actions.

Every conflict-serializable schedule is serializable.

The converse is not true (why?)
CONFLICT SERIALIZABILITY

Example:

\[ r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B) \]
CONFLICT
SERIALIZABILITY

Example:

\[ r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B) \]
CONFLICT SERIALIZABILITY

Example:

\[ r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B) \]
CONFLICT SERIALIZABILITY

Example:

\[
\begin{align*}
&\text{r}_1(\text{A}); \text{w}_1(\text{A}); \text{r}_2(\text{A}); \text{w}_2(\text{A}); \text{r}_1(\text{B}); \text{w}_1(\text{B}); \text{r}_2(\text{B}); \text{w}_2(\text{B}) \\
&\text{r}_1(\text{A}); \text{w}_1(\text{A}); \text{r}_2(\text{A}); \text{r}_1(\text{B}); \text{w}_2(\text{A}); \text{w}_1(\text{B}); \text{r}_2(\text{B}); \text{w}_2(\text{B}) \\
&\text{r}_1(\text{A}); \text{w}_1(\text{A}); \text{r}_1(\text{B}); \text{w}_1(\text{B}); \text{r}_2(\text{A}); \text{w}_2(\text{A}); \text{r}_2(\text{B}); \text{w}_2(\text{B})
\end{align*}
\]
CONFLICT
SERIALIZABILITY

Example:

\[ r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B) \]

\[ r_1(A); w_1(A); r_2(A); r_1(B); w_2(A); w_1(B); r_2(B); w_2(B) \]

\[ r_1(A); w_1(A); r_1(B); r_2(A); w_2(A); w_1(B); r_2(B); w_2(B) \]

\[ r_1(A); w_1(A); r_1(B); r_2(A); w_2(A); w_1(B); r_2(B); w_2(B) \]

\[ \ldots \]

\[ r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B) \]
TESTING FOR CONFLICT-SERIALIZABILITY

**Precedence graph:**

- A node for each transaction $T_i$,
- An edge from $T_i$ to $T_j$ whenever an action in $T_i$ conflicts with, and comes before an action in $T_j$

The schedule is conflict-serializable iff the precedence graph is acyclic
EXAMPLE 1

\[ r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B) \]
EXAMPLE 1

This schedule is conflict-serializable
EXAMPLE 2

\[ r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B) \]
EXAMPLE 2

\( r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B) \)

This schedule is NOT conflict-serializable