Transactions and Concurrency Control

(Kroenke, Chapter 9, pg 321-335)

Goals

- Database Administration
  - Concurrency Control
Database Administration

- All large and small databases need database administration
- Barber Shop database (small DB)
- Large, multi-user DB

DBA Tasks

- Managing database structure
- Controlling concurrent processing
- Managing processing rights and responsibilities
- Developing database security
- Providing for database recovery
- Managing the DBMS
- Maintaining the data repository

Who do people blame if something goes wrong?
Managing Database Structure

- Participate in database and application development
- Facilitate changes to database structure
- Maintain documentation

DBA Tasks

- Managing database structure
- Controlling concurrent processing
- Managing processing rights and responsibilities
- Developing database security
- Providing for database recovery
- Managing the DBMS
- Maintaining the data repository
Concurrent Control

- **Concurrency control**: ensure that one user’s work does not inappropriately influence another user’s work

Atomic Transactions

- A *transaction*, or *logical unit of work (LUW)*, is a series of actions taken against the database that occurs as an atomic unit
  - Either all actions in a transaction occur - COMMIT
  - Or none of them do – ABORT / ROLLBACK
Errors Introduced Without Atomic Transaction

Before
CUSTOMER
- Code: 100
- Name: JOE

SALESPERSON
- Name: JOE
- Total: 1000
- Commission: 5000

ORDER
- Order: 1000
- Quantity: 2000
- Price: .99
- Total: 2000

Action
START
1. Add new order data to CUSTOMER.
2. Add new order data to SALESPERSON.
3. Insert new ORDER.
STOP

After
CUSTOMER
- Code: 100
- Name: JOE

SALESPERSON
- Name: JOE
- Total: 1600
- Commission: 8000

ORDER
- Order: 1000
- Quantity: 2000
- Price: .99
- Total: 2000

Errors Prevented With Atomic Transaction

Before
CUSTOMER
- Code: 100
- Name: JOE

SALESPERSON
- Name: JOE
- Total: 1000
- Commission: 5000

ORDER
- Order: 1000
- Quantity: 2000
- Price: .99
- Total: 2000

Transaction
Start Transaction
Change CUSTOMER data
Change SALESPERSON data
Insert ORDER data
Commit

After
CUSTOMER
- Code: 100
- Name: JOE

SALESPERSON
- Name: JOE
- Total: 1600
- Commission: 8000

ORDER
- Order: 1000
- Quantity: 2000
- Price: .99
- Total: 2000

Make changes permanent
Undo changes
Class Exercise

- Example of transaction in the Online Store Application

Other Transaction Examples?
Concurrent Transaction

- **Concurrent transactions**: transactions that appear to users as they are being processed at the same time
- In reality, CPU can execute only one instruction at a time
  - Transactions are interleaved
- Concurrency problems
  - Lost updates
  - Inconsistent reads

Concurrent Transaction Processing

User 1: Buy 10 Snicker bars
User 2: Buy 2 Gatorade bottles

User 1:
- Read nb Snickers (ns=500)
- Reduce count Snickers by 10 (ns=490)
- Write new nb Snickers back (ns=490)

User 2:
- Read nb Gatorades (ng=200)
- Reduce count Gatorades by 2 (ng=198)
- Write new nb Gatorades back (ng=198)

Possible order of processing at DB server:
- Read nb Snickers (ns=500)
- Read nb Gatorades (ng=200)
- Reduce count Snickers by 10 (ns=490)
- Write new nb Snickers back (ns=490)
- Reduce count Gatorades by 2 (ng=198)
- Write new nb Gatorades back (ng=198)
Lost Update Problem

User 1: Buy 10 Snicker bars
User 2: Buy 2 Snicker bars

User 1:
Read nb Snickers (ns=500)
Reduce count Snickers by 10 (ns=490)
Write new nb Snickers back (ns=490)

User 2:
Read nb Snickers (ns2=500)
Reduce count Snickers by 2 (ns2=498)
Write new nb Snickers back (ns2=498)

Order of processing at DB server:

U1: Read nb Snickers (ns=500)
U2: Read nb Snickers (ns2=500)
U1: Reduce count Snickers by 10 (ns=490)
U1: Write new nb Snickers back (ns=490)
U2: Reduce count Snickers by 2 (ns2=498)
U2: Write new nb Snickers back (ns2=498)

DBMS’s View

U1: Read nb Snickers (ns=500)          | T1: R(Snickers)                  
U2: Read nb Snickers (ns2=500)         | T2: R(Snickers)                  
U1: Reduce count Snickers by 10 (ns=490)| T1: W(Snickers)                  
U1: Write new nb Snickers back (ns=490) | T1: COMMIT                      
U2: Reduce count Snickers by 2 (ns2=498)| T2: W(Snickers)                  
U2: Write new nb Snickers back (ns2=498)| T2: COMMIT                      

T1: R(S) W(S) Commit
T2: R(S) W(S) Commit
Inconsistent-Read Problem

- Dirty reads – read uncommitted data
  - T1: R(A), W(A), R(B), W(B), Abort
  - T2: R(A), W(A), Commit

- Unrepeatable reads
  - T1: R(A), R(A), W(A), Commit
  - T2: R(A), W(A), Commit

Class Exercise

- Transaction Steps
- Possible Schedule
- Possible Problems

- T1: Transfer money from savings to checking
- T2: Add interest for savings account
Inconsistent Read Example

Resource Locking

- **Locking**: prevents multiple applications from obtaining copies of the same resource when the resource is about to be changed
Lock Terminology

- **Implicit locks** - placed by the DBMS
- **Explicit locks** - issued by the application program
- **Lock granularity** - size of a locked resource
  - Rows, page, table, and database level
- **Types of lock**
  - **Exclusive lock (X)** - prohibits other users from reading the locked resource
  - **Shared lock (S)** - allows other users to read the locked resource, but they cannot update it

Explicit Locks

User 1: Buy 10 Snicker bars
User 2: Buy 2 Snicker bars

<table>
<thead>
<tr>
<th>User 1:</th>
<th>User 2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock Snickers</td>
<td>Lock Snickers</td>
</tr>
<tr>
<td>Read nb Snickers (ns=500)</td>
<td>Read nb Snickers (ns2=500)</td>
</tr>
<tr>
<td>Reduce count Snickers by 10 (ns=490)</td>
<td>Reduce count Snickers by 2 (ns2=498)</td>
</tr>
<tr>
<td>Write new nb Snickers back (ns=490)</td>
<td>Write new nb Snickers back (ns2=498)</td>
</tr>
</tbody>
</table>

Order of processing at DB server:
Class Exercise – Place Locks

- T1: R(Sa), W(Sa), R(Ch), W(Ch), Abort
- T2: R(Sa), W(Sa), C

Serializable Transactions

- **Serializable transactions:**
  - Run concurrently
  - Results like when they run separately

- **Strict two-phase locking** – locking technique to achieve serializability
Strict Two-Phase Locking

- Strict two-phase locking
  - Locks are obtained throughout the transaction
  - All locks are released at the end of transaction (COMMIT or ROLLBACK)

Strict 2PL Example

- Strict 2PL
  - X(A)
  - R(A)
  - W(A)
  - X(B)
  - R(B)
  - W(B)
  - Rel(B,A)

- Not 2PL
  - X(A)
  - R(A)
  - W(A)
  - Rel(A)
  - X(B)
  - R(B)
  - W(B)
  - Rel(B)
Class Exercise – Place Locks

- T1: R(Sa), W(Sa), R(Ch), W(Ch)
- T2: R(Ch), W(Ch), R(Sa), W(Sa)

Deadlock

User A
1. Lock paper.
2. Take paper.
3. Lock pencils.

User B
1. Lock pencils.
2. Take pencils.
3. Lock paper.

Order of processing at database server
1. Lock paper for user A.
2. Lock pencils for user B.
3. Process A’s requests; write paper record.
4. Process B’s requests; write pencil record.
5. Put A in wait state for pencils.

**Locked**
Deadlock

- **Deadlock**: two transactions are each waiting on a resource that the other transaction holds

- Prevent deadlocks

- Break deadlocks

Optimistic versus Pessimistic Locking

- **Optimistic locking** assumes that no transaction conflict will occur

- **Pessimistic locking** assumes that conflict will occur
Optimistic Locking

```
SELECT  PRODUCT.Name, PRODUCT.Quantity
FROM    PRODUCT
WHERE   PRODUCT.Name = 'Pencil'

Set NewQuantity = PRODUCT.Quantity - 5

[process transaction – take exception action if NewQuantity < 0, etc.
Assuming all is OK:]

LOCK PRODUCT

UPDATE PRODUCT
SET   PRODUCT.Quantity = NewQuantity
WHERE  PRODUCT.Name = 'Pencil'
       AND   PRODUCT.Quantity = OldQuantity

UNLOCK PRODUCT

[check to see if update was successful; if not, repeat transaction]
```

Pessimistic Locking

```
LOCK PRODUCT

SELECT  PRODUCT.Name, PRODUCT.Quantity
FROM    PRODUCT
WHERE   PRODUCT.Name = 'Pencil'

Set NewQuantity = PRODUCT.Quantity - 5

[process transaction – take exception action if NewQuantity < 0, etc.
Assuming all is OK:]

UPDATE PRODUCT
SET   PRODUCT.Quantity = NewQuantity
WHERE  PRODUCT.Name = 'Pencil'

UNLOCK PRODUCT

[no need to check if update was successful]
```
Declaring Lock Characteristics

- Most application programs **do not explicitly declare locks** due to its complication
- Mark **transaction boundaries** and declare locking behavior they want the DBMS to use
  - Transaction boundary markers: BEGIN, COMMIT, and ROLLBACK TRANSACTION
- Advantage
  - If the locking behavior needs to be changed, only the lock declaration need be changed, not the application program

Marking Transaction Boundaries

```sql
BEGIN TRANSACTION
SELECT PRODUCT.Name, PRODUCT.Quantity
FROM PRODUCT
WHERE PRODUCT.Name = 'Pencil'
Old Quantity = PRODUCT.Quantity
Set NewQuantity = PRODUCT.Quantity + 5
(process transaction - take an action if NewQuantity < 0, etc.)
UPDATE PRODUCT
SET PRODUCT.Quantity = NewQuantity
WHERE PRODUCT.Name = 'Pencil'
(continue processing transaction) . . .

IF transaction has completed normally THEN
  COMMIT TRANSACTION
ELSE
  ROLLBACK TRANSACTION
END IF

(continue processing other actions not part of this transaction . . .)
```
ACID Transactions

- Transaction properties:
  - **Atomic** - all or nothing
  - **Consistent**
  - **Isolated**
  - **Durable** – changes made by committed transactions are permanent

Consistency

- **Consistency** means either statement level or transaction level consistency
  - **Statement level consistency**: each statement independently processes rows consistently
  - **Transaction level consistency**: all rows impacted by either of the SQL statements are protected from changes during the entire transaction
    - With transaction level consistency, a transaction may not see its own changes
Statement Level Consistency

UPDATE CUSTOMER
SET AreaCode = '410'
WHERE ZipCode = '21218'

- All qualifying rows updated
- No concurrent updates allowed

Transaction Level Consistency

Start transaction
UPDATE CUSTOMER
SET AreaCode = '425'
WHERE ZipCode = '21666'
…..other transaction work
UPDATE CUSTOMER
SET Discount = 0.25
WHERE AreaCode = '425'
End Transaction

The second Update might not see the changes it made on the first Update
ACID Transactions

- Atomic
- Consistent
- Isolated
- Durable

Inconsistent-Read Problem

- Dirty reads – read uncommitted data
  - T1: R(A), W(A), R(B), W(B), Abort
  - T2: R(A), W(A), Commit
- Unrepeatable reads
  - T1: R(A), R(A), W(A), Commit
  - T2: R(A), W(A), Commit
- Phantom reads
  - Re-read data and find new rows
Isolation

- SQL-92 defines four transaction isolation levels:
  - Read uncommitted
  - Read committed
  - Repeatable read
  - Serializable

Transaction Isolation Level

<table>
<thead>
<tr>
<th>Problem Type</th>
<th>Isolation Level</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Read Uncommitted</td>
</tr>
<tr>
<td></td>
<td>Read Committed</td>
</tr>
<tr>
<td></td>
<td>Repeatable Read</td>
</tr>
<tr>
<td></td>
<td>Serializable</td>
</tr>
<tr>
<td>Dirty Read</td>
<td>Possible</td>
</tr>
<tr>
<td>Repeatable Read</td>
<td>Not Possible</td>
</tr>
<tr>
<td>Phantom Read</td>
<td>Not Possible</td>
</tr>
<tr>
<td></td>
<td>Not Possible</td>
</tr>
<tr>
<td>Not Repeatable</td>
<td>Possible</td>
</tr>
<tr>
<td></td>
<td>Not Possible</td>
</tr>
<tr>
<td></td>
<td>Not Possible</td>
</tr>
<tr>
<td>Supports Consistency</td>
<td>Possible</td>
</tr>
<tr>
<td></td>
<td>Possible</td>
</tr>
<tr>
<td></td>
<td>Not Possible</td>
</tr>
<tr>
<td></td>
<td>Not Possible</td>
</tr>
</tbody>
</table>
### Class Exercise

- **T1**: insert product
- **T2**: add sale (checkout)

What transaction isolation level would you use for each of the procedures above, and why?

### Cursor Type

- A **cursor** is a pointer into a set of records
- It can be defined using SELECT statements
- Four cursor types
  - **Forward only**: the application can only move forward through the recordset
  - Scrollable cursors can be scrolled forward and backward through the recordset
    - **Static**: processes a snapshot of the relation that was taken when the cursor was opened
    - **Keyset**: combines some features of static cursors with some features of dynamic cursors
  - **Dynamic**: a fully featured cursor
- Choosing appropriate isolation levels and cursor types is critical to database design
## Cursor Summary

<table>
<thead>
<tr>
<th>Cursor Type</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward only</td>
<td>Application can only move forward through the recordset. Changes made by other cursors in this transaction or in other transactions will be visible only if they occur on rows ahead of the cursor.</td>
<td></td>
</tr>
<tr>
<td>Static</td>
<td>Application uses the data as they were at the time the cursor was opened. Changes made by this cursor are visible. Changes from other sources are not visible. Insert and forward sending are allowed.</td>
<td></td>
</tr>
<tr>
<td>Keyset</td>
<td>When the cursor is opened, a primary key value is saved for each row in the recordset. When the application accesses a row, the key is used to fetch the current values for the row. Updates from any source are visible. Inserts from sources outside this cursor are not visible. Changes in row order are not visible. If the isolation level is dirty read, then committed updates and deletions are visible; otherwise only committed updates and deletions are visible.</td>
<td></td>
</tr>
<tr>
<td>Dynamic</td>
<td>Changes of any type and from any source are visible. All inserts, updates, deletions, and changes in recordset order are visible. If the isolation level is dirty read, then uncommitted changes are visible. Otherwise, only committed changes are visible.</td>
<td></td>
</tr>
</tbody>
</table>