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

<table>
<thead>
<tr>
<th>Before</th>
<th>Action</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSTOMER</td>
<td>START</td>
<td></td>
</tr>
<tr>
<td>CNum</td>
<td>OrderNum</td>
<td>Description</td>
</tr>
<tr>
<td>123</td>
<td>1000</td>
<td>400 Baseballs</td>
</tr>
<tr>
<td>SALESPERSON</td>
<td>1. Add new-order data to CUSTOMER.</td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Total-Sales</td>
<td>Commission Due</td>
</tr>
<tr>
<td>JONES</td>
<td></td>
<td>$320</td>
</tr>
<tr>
<td>ORDER</td>
<td>3. Insert new ORDER.</td>
<td></td>
</tr>
<tr>
<td>OrderNum</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SALESPERSON</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Total-Sales</td>
<td>Commission Due</td>
</tr>
<tr>
<td>JONES</td>
<td></td>
<td>$320</td>
</tr>
<tr>
<td>ORDER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OrderNum</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Errors Prevented With Atomic Transaction

<table>
<thead>
<tr>
<th>Before</th>
<th>Transaction</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSTOMER</td>
<td>Start Transaction</td>
<td></td>
</tr>
<tr>
<td>CNum</td>
<td>OrderNum</td>
<td>Description</td>
</tr>
<tr>
<td>123</td>
<td>1000</td>
<td>400 Baseballs</td>
</tr>
<tr>
<td>SALESPERSON</td>
<td>Change CUSTOMER data</td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Total-Sales</td>
<td>Commission Due</td>
</tr>
<tr>
<td>JONES</td>
<td></td>
<td>$320</td>
</tr>
<tr>
<td>ORDER</td>
<td>Make changes permanent</td>
<td></td>
</tr>
<tr>
<td>OrderNum</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SALESPERSON</td>
<td>Undo changes</td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Total-Sales</td>
<td>Commission Due</td>
</tr>
<tr>
<td>JONES</td>
<td></td>
<td>$320</td>
</tr>
<tr>
<td>ORDER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OrderNum</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Class Exercise

- Example of transaction in the Online Store Application

Other Transaction Examples?
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

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)
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)

T1: R(Snickers)
T2: R(Snickers)

T1: W(Snickers)
T1: COMMIT
T2: W(Snickers)
T2: 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

- **Phantom reads** – similar to unrepeatable reads, but set of values is different
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</td>
<td>Reduce count Snickers by 2</td>
</tr>
<tr>
<td>(ns=490)</td>
<td>(ns2=498)</td>
</tr>
<tr>
<td>Write new nb Snickers back</td>
<td>Write new nb Snickers back</td>
</tr>
<tr>
<td>(ns=490)</td>
<td>(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

<table>
<thead>
<tr>
<th>Strict 2PL</th>
<th>Not 2PL</th>
</tr>
</thead>
<tbody>
<tr>
<td>X(A)</td>
<td>X(A)</td>
</tr>
<tr>
<td>R(A)</td>
<td>R(A)</td>
</tr>
<tr>
<td>W(A)</td>
<td>W(A)</td>
</tr>
<tr>
<td>X(B)</td>
<td>Rel(A)</td>
</tr>
<tr>
<td>R(B)</td>
<td>X(B)</td>
</tr>
<tr>
<td>W(B)</td>
<td>R(B)</td>
</tr>
<tr>
<td>Rel(B,A)</td>
<td>W(B)</td>
</tr>
<tr>
<td></td>
<td>Rel(B)</td>
</tr>
</tbody>
</table>

### Class Exercise – Place Locks

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

```sql
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

```sql
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

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

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