

IT420: Database Management and Organization

Managing Multi-user Databases (Chapter 9)

1

PHP Miscellaneous

- `$db->insert_id`
 - **Retrieves** the ID generated for an AUTO_INCREMENT column by the previous INSERT query
 - Return value:
 - The ID generated for an AUTO_INCREMENT column by the previous INSERT query on success
 - 0 if the previous query does not generate an AUTO_INCREMENT value
 - **FALSE** if no MySQL connection was established.

Kroenke, Database Processing

2

Goals

- Database Administration
 - Concurrency Control

Kroenke, Database Processing

3

Database Administration

- All large and small databases need database administration
- Barber Shop database (small DB)

- Large, multi-user DB

Kroenke, Database Processing

4

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

Concurrency Control

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

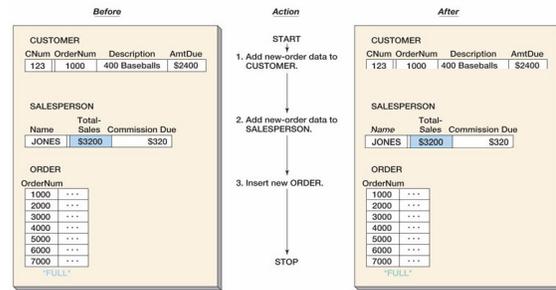
Atomic Transactions

- 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

Kroenke, Database Processing

9

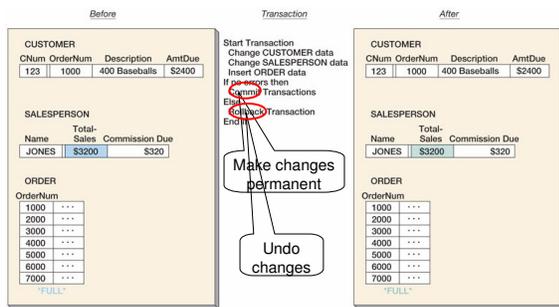
Errors Introduced Without Atomic Transaction



Kroenke, Database Processing

10

Errors Prevented With Atomic Transaction



Kroenke, Database Processing

11

Class Exercise

- Example of transaction in the Online Mids Store Application – submit order

Kroenke, Database Processing

12

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)
 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

T1: R(S) W(S) Commit
 T2: R(S) W(S) Commit

time

Kroenke, Database Processing

17

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

Kroenke, Database Processing

18

Class Exercise

- Transaction Steps
- Possible Schedule
- Possible Problems
- T1: Transfer money from savings to checking
- T2: Add interest for savings account

Kroenke, Database Processing

19

Inconsistent Read Example

Kroenke, Database Processing

20

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

User 1:

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

User 2:

Lock Snickers
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:

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)

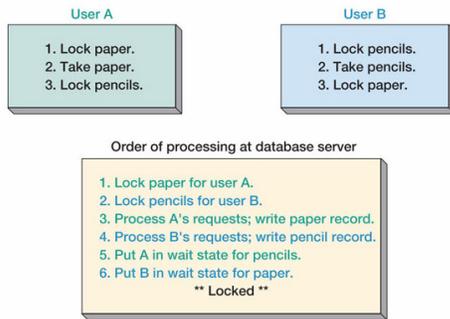
Strict 2PL Example

- | | |
|--------------|-----------|
| ▪ Strict 2PL | ▪ Not 2PL |
| ▪ X(A) | ▪ X(A) |
| ▪ R(A) | ▪ R(A) |
| ▪ W(A) | ▪ W(A) |
| ▪ X(B) | ▪ Rel(A) |
| ▪ R(B) | ▪ X(B) |
| ▪ W(B) | ▪ R(B) |
| ▪ Rel(B,A) | ▪ 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



Kroenke, Database Processing

29

Deadlock

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

Kroenke, Database Processing

30

Optimistic versus Pessimistic Locking

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

Kroenke, Database Processing

31

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}
```

Kroenke, Database Processing

32

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

```
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 exception 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:
 - **A**tomic - all or nothing
 - **C**onsistent
 - **I**solated
 - **D**urable – 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

- **A**tomic
- **C**onsistent
- **I**solated
- **D**urable

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

		Isolation Level			
		Read Uncommitted	Read Committed	Repeatable Read	Serializable
Problem Type	Dirty Read	Possible	Not Possible	Not Possible	Not Possible
	Nonrepeatable Read	Possible	Possible	Not Possible	Not Possible
	Phantom Read	Possible	Possible	Possible	Not Possible

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