DB Integrity & Transactions Part 2

IS240 – DBMS

Lecture #12 – 2010-04-12

M. E. Kabay, PhD, CISSP-ISSMP

Assoc. Prof. Information Assurance
School of Business & Cyber Studies, Norwich University

mailto:mkabay@norwich.edu V: 802.479.7937

Objectives

- Define elements of *ACID* transactions
  - ❑Atomicity
  - ❑Consistency
  - ❑Isolation
  - ❑Durability
- Define SQL Isolation Levels
- What are phantom rows and how do we avoid them?
- How are internal key values generated and used in updates in the face of concurrency?
- What is the purpose of database cursors?

Topics

- ACID Transactions
- SQL 99/2003 Isolation Levels
- Phantom Rows
- Generated Keys
- Database Cursors
- Sally’s Pet Store Inventory

ACID Transactions

- **A**tomicity: all changes succeed or fail together.
- **C**onsistency: all data remain internally consistent (when committed) and can be validated by application checks.
- **I**solation: The system gives each transaction the perception that it is running in isolation. There are no concurrent access issues.
- **D**urability: When a transaction is committed, all changes are permanently saved even if there is a hardware or system failure.

SQL 99/2003 Isolation Levels

- READ UNCOMMITTED
  - ❑ Problem: might read dirty data that is rolled back
  - ❑ Restriction: not allowed to save any data
- READ COMMITTED
  - ❑ Problem: Second transaction might change or delete data
  - ❑ Restriction: Need optimistic concurrency handling
- REPEATABLE READ
  - ❑ Problem: Phantom rows caused by concurrent access
- SERIALIZABLE
  - ❑ Provides same level of control as if all transactions were run sequentially.
  - ❑ But, still might encounter locks and deadlocks
    - ✓ Remember to LOCK in SAME ORDER and UNLOCK in REVERSE ORDER!

Phantom Rows

ALICE

SELECT SUM(QOH)
FROM Inventory
WHERE Price BETWEEN 10 and 20
Result: 5 + 4 + 8 = 17

BOB

INSERT INTO Inventory
VALUES (121, 7, 16)
INSERT INTO Inventory
VALUES (122, 3, 14)

Included in first query

ItemID	QOH	Price
111	5	15
113	**6**	**7**
117	**12**	**30**
118	4	12
119	**7**	**22**
120	8	17
121	*7*	*16*
122	*3*	*14*

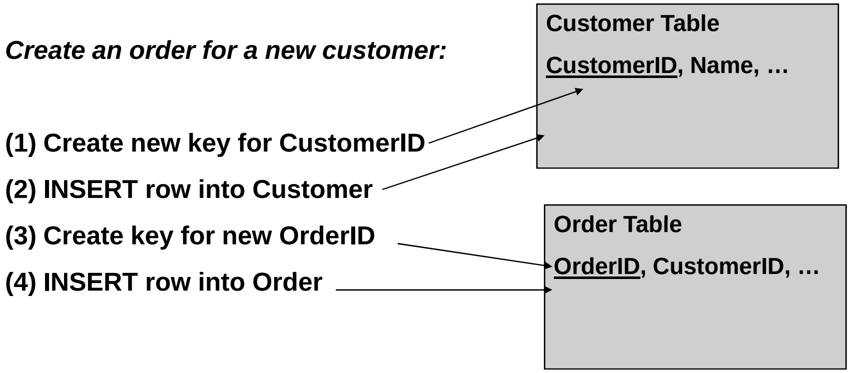
Additional or changed rows will be included in the second query, which may cause contradictions in results

ALICE

SELECT SUM(QOH)
FROM Inventory
WHERE Price BETWEEN 10 and 20
Result: 5 + 4 + 8 + 7 + 3 = 27

Generated Keys

Create an order for a new customer:



(1) Create new key for CustomerID

(2) INSERT row into Customer

(3) Create key for new OrderID

(4) INSERT row into Order

Customer Table

<u>CustomerID</u>, Name, …

Order Table

<u>OrderID</u>, CustomerID, …

Problem: What if someone concurrently generates another autokey just as you are trying to use the one you created?

Generally the DBMS remembers only the latest autokey!

Methods to Generate Keys

1. The DBMS generates key values automatically whenever a row is inserted into a table.
   - ❑ Drawback: it is tricky to get the generated value to use it in a second table.
2. A separate key generator is called by a programmer to create a new key for a specified table.
   - ❑ Drawback: programmers have to write code to generate a key for every table and each row insertion.
   - ❑ Overall drawbacks: neither method is likely to be transportable. If you change the DBMS, you will have to rewrite the procedures to generate keys.

Auto-Generated Keys

- *Create an order for a new customer:*
  1. INSERT row into Customer
  2. Get the key value that was generated
  3. Verify the key value is correct. How?
  4. INSERT row into Order
- Major problem:
  - ❑ Step 2 requires that the DBMS return the key value that was most recently generated.
  - ❑ How do you know it is the right value?
  - ❑ What happens if two transactions generate keys at almost the same time on the same table?

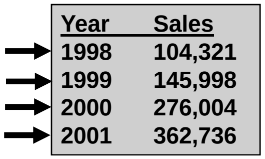
Key-Generation Routine

- Create an order for a new customer:
  - ❑ Generate a key for CustomerID
  - ❑ INSERT row into Customer
  - ❑ Generate a key for OrderID
  - ❑ INSERT row into Order
- This method ensures that unique keys are generated
  - ❑ You can use the keys in multiple tables because you know the value
  - ❑ But none of it is automatic
  - ❑ Always requires procedures and sometimes data triggers

Topics

- ACID Transactions
- SQL 99/2003 Isolation Levels
- Phantom Rows
- Generated Keys
- Database Cursors
- Sally’s Pet Store Inventory

Database Cursors



Year	Sales
1998	104,321
1999	145,998
2000	276,004
2001	362,736

- Purpose
  - ❑ Track through table or query one row at a time.
  - ❑ Data cursor is a pointer to active row.
- Why?
  - ❑ Performance.
  - ❑ SQL cannot do everything.
    - ✓ Complex calculations.
    - ✓ Compare multiple rows.

Database Cursor Program Structure

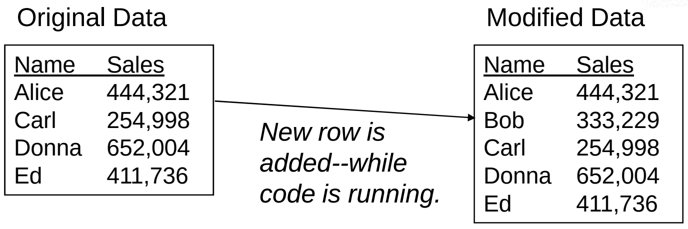
```
DECLARE cursor1 CURSOR FOR
    SELECT AccountBalance
    FROM Customer;
sumAccount, balance Currency;
SQLSTATE Char(5);
BEGIN
    sumAccount = 0;
    OPEN cursor1;
    WHILE (SQLSTATE = ‘00000’)
    BEGIN
        FETCH cursor1 INTO balance;
        IF (SQLSTATE = ‘00000’) THEN
            sumAccount = sumAccount + balance;
        END IF
    END
    CLOSE cursor1;
    -- display the sumAccount or do a calculation
END
```

Cursor Positioning with FETCH

```
DECLARE cursor2 SCROLL CURSOR FOR
SELECT …
OPEN cursor2;
FETCH LAST FROM cursor2 INTO …
Loop…
    FETCH PRIOR FROM cursor2 INTO …
End loop
CLOSE cursor2;
```

FETCH positioning options:	
FETCH NEXT	next row
FETCH PRIOR	prior row
FETCH FIRST	first row
FETCH LAST	last row
FETCH ABSOLUTE 5	fifth row
FETCH RELATIVE -3	back 3 rows

Problems with Multiple Users



Original Data

Name	Sales
Alice	444,321
Carl	254,998
Donna	652,004
Ed	411,736

New row is added--while code is running.

Modified Data

Name	Sales
Alice	444,321
Bob	333,229
Carl	254,998
Donna	652,004
Ed	411,736

The SQL standard can prevent this problem with the *INSENSITIVE* option:

DECLARE cursor3 INSENSITIVE CURSOR FOR …

But this is an expensive approach because the DBMS usually makes a <u>copy</u> of the data. Instead, avoid moving backwards.

Changing Data with Cursors

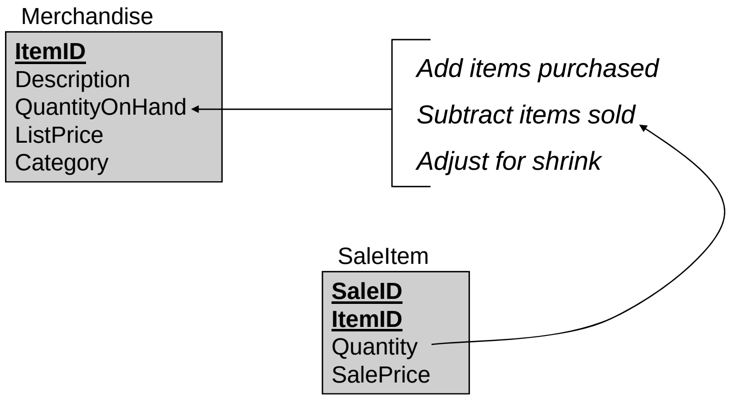
```
DECLARE cursor1 CURSOR FOR
SELECT Year, Sales, Gain
FROM SalesTotal
ORDER BY Year
FOR UPDATE OF Gain;
priorSales, curYear, curSales, curGain
BEGIN
    priorSales = 0;
    OPEN cursor1;
    Loop:
        FETCH cursor1 INTO curYear, curSales, curGain
        UPDATE SalesTotal
        SET Gain = Sales – priorSales
        WHERE CURRENT OF cursor1;
        priorSales = curSales;
    Until end of rows
    CLOSE cursor1;
    COMMIT;
END
```

Year	Sales	Gain
2000	151,039	
2001	179,332	
2002	195,453	
2003	221,883	
2004	223,748	

Sally’s Pet Store Inventory

- Inventory method 1: calculate the current quantity on hand by totaling all purchases and sales every time the total is needed.
  - ❑ Drawback: performance
- Inventory method 2: keep a running balance in the inventory table and update it when an item is purchased or sold.
  - ❑ Drawback: tricky code
- Also, you need an adjustment process for “inventory shrink”
  - ❑ Corrections of mistakes

Inventory QuantityOnHand



Merchandise

ItemID
Description
QuantityOnHand
ListPrice
Category

Add items purchased

Subtract items sold

Adjust for shrink

SaleItem

SaleID
ItemID
Quantity
SalePrice

Inventory Events

SaleItem

SaleID
ItemID
Quantity
SalePrice

A USER MAY

- Add a row.
- Delete a row.
- Update Quantity.
- Update ItemID.

- For a new sale, a row is added to the SaleItem table.
- A sale or an item could be removed because of a clerical error or the customer changes his or her mind. A SaleItem row will be deleted.
- An item could be returned, or the quantity could be adjusted because of a counting error. The Quantity is updated in the SaleItem table.
- An item is entered incorrectly. ItemID is updated in the SaleItem table.

New Sale: Insert SaleItem Row

```
CREATE TRIGGER NewSaleItem
AFTER INSERT ON SaleItem
REFERENCING  NEW ROW AS newrow
FOR EACH ROW
        UPDATE Merchandise
        SET QuantityOnHand = QuantityOnHand – newrow.Quantity
        WHERE ItemID = newrow.ItemID;
```

Delete SaleItem Row

```
CREATE TRIGGER DeleteSaleItem
AFTER DELETE ON SaleItem
REFERENCING      OLD ROW AS oldrow
FOR EACH ROW
        UPDATE Merchandise
        SET QuantityOnHand =
                QuantityOnHand + oldrow.Quantity
        WHERE ItemID = oldrow.ItemID;
```

Inventory Update Sequence

SaleItem	Clerk	Event Code	Merchandise
SaleID 101 ItemID 15 Quantity 10 Quantity 8	1. Enter new sale item, enter Quantity of 10. 3. Change Quantity to 8.	2. Subtract Quantity 10 from QOH. 4. Subtract Quantity 8 from QOH.	ItemID 15 QOH 50 QOH 40 QOH 32
Solution that corrects for change			
SaleID 101 ItemID 15 Quantity 10 Quantity 8	1. Enter new sale item, enter Quantity of 10. 3. Change Quantity to 8.	2. Subtract Quantity 10 from QOH. 4. Add original Quantity 10 back and subtract Quantity 8 from QOH.	ItemID 15 QOH 50 QOH 40 QOH 42

OOPS

Quantity Changed Event

```
CREATE TRIGGER UpdateSaleItem
AFTER UPDATE ON SaleItem
REFERENCING      OLD ROW AS oldrow
                 NEW ROW AS newrow
FOR EACH ROW
    UPDATE Merchandise
    SET QuantityOnHand =
        QuantityOnHand
        + oldrow.Quantity
        – newrow.Quantity
    WHERE ItemID = oldrow.ItemID;
```

ItemID or Quantity Changed Event

```
CREATE TRIGGER UpdateSaleItem
AFTER UPDATE ON SaleItem
REFERENCING   OLD ROW AS oldrow
              NEW ROW AS newrow
FOR EACH ROW
BEGIN
        UPDATE Merchandise
        SET QuantityOnHand = QuantityOnHand + oldRow.Quantity
        WHERE ItemID = oldrow.ItemID;
        UPDATE Merchandise
        SET QuantityOnHand = QuantityOnHand – newRow.Quantity
        WHERE ItemID = newrow.ItemID;
        COMMIT;
END
```