DB Integrity & Transactions Part 1

IS240 – DBMS

Lecture #11 – 2010-04-05

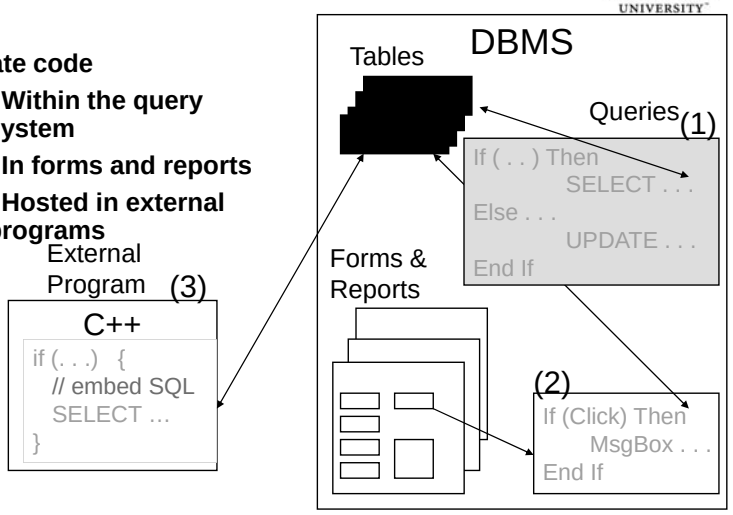
M. E. Kabay, PhD, CISSP-ISSMP
Assoc. Prof. Information Assurance
School of Business & Management, Norwich University
mailto:mkabay@norwich.edu **V: 802.479.7937**

Objectives

- Why would you need to use procedural code when SQL is so powerful?
- How do you use data triggers to make changes automatically?
- How does the DBMS ensure related changes are made together?
- How do you handle multiple users changing the same data at the same time?
- How are internal key values generated and used in updates?
- What is the purpose of database cursors?

Programming Environment

- **Create code**
  - **(1) Within the query system**
  - **(2) In forms and reports**
  - **(3) Hosted in external programs**



User-Defined Function

```
CREATE FUNCTION EstimateCosts
        (ListPrice Currency, ItemCategory VarChar)
RETURNS Currency
BEGIN
        IF (ItemCategory = 'Clothing') THEN
                RETURN ListPrice * 0.5
        ELSE
                RETURN ListPrice * 0.75
        END IF
END
```

Function to Perform Conditional Update

```
CREATE FUNCTION IncreaseSalary
        (EmpID INTEGER, Amt CURRENCY)
RETURNS CURRENCY
BEGIN
        IF (Amt > 50000) THEN
                RETURN -1               -- error flag
        END
        UPDATE Employee SET Salary = Salary + Amt
        WHERE EmployeeID = EmpID;
        RETURN Amt
END
```


Looking Up Data

```
CREATE FUNCTION IncreaseSalary
        (EmpID INTEGER, Amt CURRENCY)
RETURNS CURRENCY
DECLARE
        CURRENCY MaxAmount;
BEGIN
        SELECT MaxRaise INTO MaxAmount
        FROM CompanyLimits
        WHERE LimitName = 'Raise';

        IF (Amt > MaxAmount) THEN
                RETURN -1               -- error flag
        END
        UPDATE Employee SET Salary = Salary + Amt
        WHERE EmployeeID = EmpID;
        RETURN Amt;
END
```


Data Trigger Events

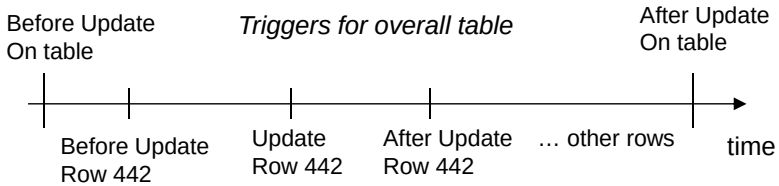
	INSERT	
BEFORE	DELETE	AFTER
	UPDATE	

- **Oracle additions:**
  - **Tables** **ALTER, CREATE, DROP**
  - **User** **LOGOFF, LOGON**
  - **Database** **SERVERERROR, SHUTDOWN, STARTUP**

Statement v. Row Triggers

SQL

```
UPDATE Employee
SET Salary = Salary + 10000
WHERE EmployeeID=442
OR EmployeeID=558
```



Triggers for each row

Data Trigger Example

```
CREATE TRIGGER LogSalaryChanges
AFTER UPDATE OF Salary ON Employee
REFERENCING  OLD ROW as oldrow
             NEW ROW AS newrow
FOR EACH ROW
    INSERT INTO SalaryChanges
    (EmpID, ChangeDate, User, OldValue, NewValue)
    VALUES
    (newrow.EmployeeID, CURRENT_TIMESTAMP,
    CURRENT_USER, oldrow.Salary, newrow.Salary);
```


Canceling Data Changes in Triggers

```
CREATE TRIGGER TestDeletePresident
BEFORE DELETE ON Employee
REFERENCING OLD ROW AS oldrow
FOR EACH ROW
        WHEN (oldrow.Title = 'President')
                SIGNAL _CANNOT_DELETE_PRES;
```


Cascading Triggers

Sale(SaleID, SaleDate, …)

SaleItem(SaleID, ItemID, Quantity, …)

AFTER INSERT
UPDATE Inventory
SET QOH = QOH – newrow.Quantity

Inventory(ItemID, QOH, …)

AFTER UPDATE
WHEN newrow.QOH < newrow.Reorder
INSERT {new order}
INSERT {new OrderItem}

Order(OrderID, OrderDate, …)

OrderItem(OrderID, ItemID, Quantity, …)

Trigger Loop

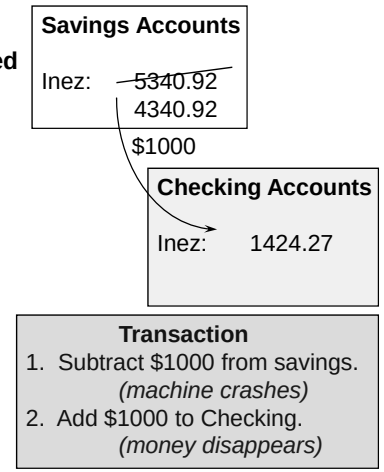
Employee(EID, Salary) → AFTER UPDATE
IF newrow.Salary > 100000 THEN
Add Bonus
END

BonusPaid(EID, BonusDate, Amount) → AFTER UPDATE Or INSERT
IF newrow.Bonus > 50000 THEN
Reduce Bonus
Add Options
END

StockOptions(EID, OptionDate, Amount, SalaryAdj) → AFTER UPDATE Or INSERT
IF newrow.Amount > 100000 THEN
Reduce Salary
END

Transactions

- **Some transactions result in multiple changes.**
  - **These changes must all be completed successfully, or the group must fail.**
  - **Protection for hardware and communication failures.**
  - **Example: bank customer transfers money from savings account to checking account.**
    - ✓ **Decrease savings balance**
    - ✓ **Increase checking balance**
    - ✓ **Problem if one transaction and machine crashes.**
- **Possibly: give users a chance to reverse/undo a transaction.**
- **Performance gain by executing transactions as a block.**



Transaction Steps

Steps	Savings Balance	Checking Balance
0. Start	**5,340.92**	**1,424.27**
1. Subtract 1,000	**4,340.92**	**1,424.27**
2. Add 1,000	**4,340.92**	**2,424.27**
Problem arises if transaction is not completed		
1. Subtract 1,000	**4,340.92**	**1,424.27**
2. Machine crashes		**1,000 is gone**

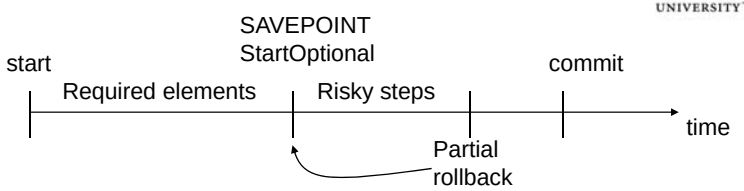
Defining Transactions

- **The computer needs to be told which changes must be grouped into a transaction.**
  - **Turn on transaction processing.**
  - **Signify a transaction start.**
  - **Signify the end.**
    - ✓ **Success: save all changes**
    - ✓ **Failure: cancel all changes**
- **Must be set in module code**
  - **Commit**
  - **Rollback**

SQL Transaction Code

```
CREATE FUNCTION TransferMoney(Amount Currency, AccountFrom Number,
                AccountTo Number) RETURNS NUMBER
curBalance Currency;
BEGIN
        DECLARE HANDLER FOR SQLEXCEPTION
        BEGIN
                ROLLBACK;
                Return -2;          -- flag for completion error
        END;
        START TRANSACTION;      -- optional
        SELECT CurrentBalance INTO curBalance
        FROM Accounts WHERE (AccountID = AccountFrom);
        IF (curBalance < Amount) THEN
                RETURN -1;          -- flag for insufficient funds
        END IF
        UPDATE Accounts
        SET CurrentBalance = CurrentBalance – Amount
        WHERE AccountID = AccountFrom;
        UPDATE Accounts
        SET CurrentBalance = CurrentBalance + Amount
        WHERE AccountID = AccountTo;
        COMMIT;
        RETURN 0;                   -- flag for success
END;
```

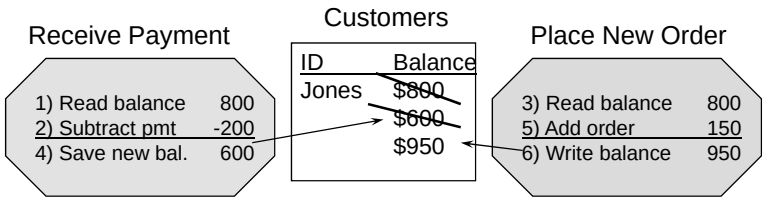
SAVEPOINT




```
START TRANSACTION;
SELECT …
UPDATE …
SAVEPOINT StartOptional;
UPDATE …
UPDATE …
If error THEN
        ROLLBACK TO SAVEPOINT StartOptional;
END IF
COMMIT;
```


Concurrent Access

- **Concurrent Access**
  - **Multiple users or processes changing the same data at the same time.**
  - **Final data will be wrong!**
- **Force sequential**
  - **Locking**
  - **Delayed, batch updates**
- **Two processes**
  - **Receive payment ($200)**
  - **Place new order ($150)**
- **Initial balance $800**
  - **Result should be $800 - 200 + 150 = $750**
  - **Interference result is either $600 or $950**

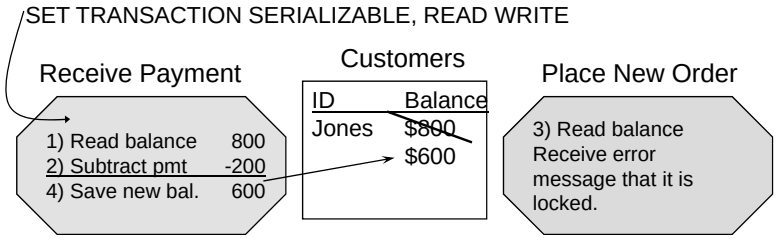


Concurrent Access Steps

Receive Payment	Balance	Place New Order
1. Read balance 800	**800**	
2. Subtract Pmt. -200		**3. Read balance 800**
	600	
4. Save balance 600		**5. Add order 150**
	950	**6. Write balance 950**

Pessimistic Locks: Serialization

- **One answer to concurrent access is to prevent it.**
- **When a transaction needs to alter data, it places a SERIALIZABLE lock on the data used, so no other transactions can even read the data until the first transaction is completed.**



SQL Pessimistic Lock

```
CREATE FUNCTION ReceivePayment (
        AccountID NUMBER, Amount Currency) RETURNS NUMBER
BEGIN
        DECLARE HANDLER FOR SQLEXCEPTION
        BEGIN
                ROLLBACK;
                RETURN -2;
        END
        SET TRANSACTION SERIALIZABLE, READ WRITE;
        UPDATE Accounts
        SET AccountBalance = AccountBalance - Amount
        WHERE AccountNumber = AccountID;
        COMMIT;
        RETURN 0;
END
```


Serialization Effects

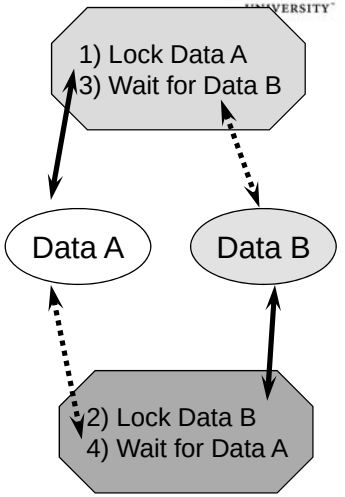
Receive Payment	Balance	Place New Order
1. Read balance 800	**800**	
2. Subtract Pmt. -200		**3. Read balance**
	600	**Error: Blocked**
4. Save balance 600		**3. Read balance 600**
	750	**4. Add order 150**
		5. Write balance 750

Transaction to Transfer Money

```
CREATE FUNCTION ReceivePayment (
        AccountID NUMBER, Amount Currency) RETURNS
NUMBER
BEGIN
        DECLARE HANDLER FOR SQLEXCEPTION
        BEGIN
                ROLLBACK;
                RETURN -2;
        END
        SET TRANSACTION SERIALIZABLE, READ WRITE;
        UPDATE Accounts
        SET AccountBalance = AccountBalance - Amount
        WHERE AccountNumber = AccountID;
        COMMIT;
        RETURN 0;
END
```


Deadlock

- **Deadlock**
  - **Two (or more) processes have placed locks on data and are waiting for the other's data.**
- **Many solutions**
  - **Random wait time**
  - **Global lock manager**
  - **Two-phase commit - messages**



Deadlock Sequence

Process 1	Data A	Data B	Process2
1. Lock Data A **3. Wait for Data B**	**Locked by 1**	**Locked by 2**	**2. Lock Data B** **4. Wait for Data A**

Optimistic Locks

- **Assume that collisions are rare**
- **Improved performance, fewer resources**
- **Allow all code to read any data (no locks)**
- **When code tries to write a new value**
  - **Lock the records**
  - **Check to see if the existing value is different from the one you were given earlier**
  - **If it is different, someone changed the database before you finished, so it is a collision--raise an error and unlock**
  - **Try again**

Optimistic Locks for Simple Update

- **(1) Read the balance and save it (e.g., to *initial-value_buffer)***
- **(2) Prepare the new computed value**
- **(3) Write the new value to a buffer (e.g., *new-value_buffer*)**
- **(4) LOCK the resources involved**
- **(5) Check for errors by comparing the *initial-value_buffer* to the *currentvalue* in the database**
  - **(5a) If i*nitial-value_buffer* <> *currentvalue*, UNLOCK and go back to step (1).**
  - **(5b) If *initial-value_buffer = currentvalue* then write *new-value_buffer* into *currentvalue and UNLOCK***

Optimistic Locks with SQL

```
CREATE FUNCTION ReceivePayment (
        AccountID NUMBER, Amount Currency) RETURNS NUMBER
oldAmount Currency;
testEnd Boolean = FALSE;
BEGIN
        DO UNTIL testEnd = TRUE
        BEGIN
                SELECT Amount INTO oldAmount
                WHERE AccountNumber = AccountID;
                …
                UPDATE Accounts
                SET AccountBalance = AccountBalance - Amount
                WHERE AccountNumber = AccountID
                AND Amount = oldAmount;
                COMMIT;
                IF SQLCODE = 0 and nrows > 0 THEN
                        testEnd = TRUE;
                        RETURN 0;
                END IF
                -- keep a counter to avoid infinite loops
        END
END
```

This is the comparison of current value vs old value