CSCC43H: Introduction to Databases

Lecture 9

Wael Aboulsaadat

Acknowledgment: these slides are partially based on Prof. Garcia-Molina & Prof. Ullman slides accompanying the course's textbook.

Database Management System (DBMS)

- A collection of programs that enable:

  Defining (describing the structure),

  Populating by data (Constructing),

  Manipulating (querying, updating),

  → Preserving consistency,

  – Protecting from misuse,

  – Recovering from failure, and

  – Concurrent using

  of a database.

Normalization Example 2: salesperson and customer

Salesperson and Customer

- Unnormalized Relation

  Sales Report (Salesperson-No, Salesperson-Name, Sales-Area, {Customer-No, Customer-Name, Warehouse-No, Warehouse-Location, Sales-Amount})

  {} -- repeating group

- 1NF -- removes repeating groups

  Salesperson (Salesperson-No, Salesperson-Name, Sales-Area)

  Salesperson-Customer (Salesperson-No, Customer-No, Customer-Name, Warehouse-No, Warehouse-Location, Sales-Amount)

Salesperson and Customer

- 1NF

  Salesperson (Salesperson-No, Salesperson-Name, Sales-Area)

  Salesperson-Customer (Salesperson-No, Customer-No, Customer-Name, Warehouse-No, Warehouse-Location, Sales-Amount)

- 2NF -- removes non-full dependencies on primary key

  Salesperson (Salesperson-No, Salesperson-Name, Sales-Area)

  Sales (Salesperson-No, Customer-No, Sales-Amount)

  Customer-Warehouse (Customer-No, Customer-Name, Warehouse-No, Warehouse-Location)

Salesperson and Customer

- 2NF

  Salesperson (Salesperson-No, Salesperson-Name, Sales-Area)

  Sales (Salesperson-No, Customer-No, Sales-Amount)

  Customer-Warehouse (Customer-No, Customer-Name, Warehouse-No, Warehouse-Location)

- 3NF -- removes transitive dependencies

  Salesperson (Salesperson-No, Salesperson-Name, Sales-Area)

  Sales (Salesperson-No, Customer-No, Sales-Amount)

  Customer-Warehouse (Customer-No, Customer-Name, Warehouse-No)

  Warehouse (Warehouse-No, Warehouse-Location)

Normalization Example 3: salesperson and products

SALESPERSON/PRODUCT table

Salesperson Number	Product Number	Salesperson Name	Commission Percentage	Year of Hire	Department Number	Manager Name	Product Name	Unit Price	Quantity
137	19440	Baker	10	1995	73	Scott	Hammer	17.50	473
	24013						Saw	26.25	170
	26722						Pliers	11.50	688
186	16386	Adams	15	2001	59	Lopez	Wrench	12.95	1745
	19440						Hammer	17.50	2529
	21765						Drill	32.99	1962
	24013						Saw	26.25	3071
204	21765	Dickens	10	1998	73	Scott	Drill	32.99	809
	26722						Pliers	11.50	734
361	16386	Carlyle	20	2001	73	Scott	Wrench	12.95	3729
	21765						Drill	32.99	3110
	26722						Pliers	11.50	2738

First Normal Form

SALESPERSON/PRODUCT table

Salesperson Number	Product Number	Salesperson Name	Commission Percentage	Year of Hire	Department Number	Manager Name	Product Name	Unit Price	Quantity
137	19440	Baker	10	1995	73	Scott	Hammer	17.50	473
137	24013	Baker	10	1995	73	Scott	Saw	26.25	170
137	26722	Baker	10	1995	73	Scott	Pliers	11.50	688
186	16386	Adams	15	2001	59	Lopez	Wrench	12.95	1475
186	19440	Adams	15	2001	59	Lopez	Hammer	17.50	2529
186	21765	Adams	15	2001	59	Lopez	Drill	32.99	1962
186	24013	Adams	15	2001	59	Lopez	Saw	26.25	3071
204	21765	Dickens	10	1998	73	Scott	Drill	32.99	809
204	26722	Dickens	10	1998	73	Scott	Pliers	11.50	734
361	16386	Carlyle	20	2001	73	Scott	Wrench	12.95	3729
361	21765	Carlyle	20	2001	73	Scott	Drill	32.99	3110
361	26722	Carlyle	20	2001	73	Scott	Pliers	11.50	2738

Second Normal Form

SALESPERSON table

Salesperson Number	Salesperson Name	Commission Percentage	Year of Hire	Department Number	Manager Name

PRODUCT table

Product Number	Product Name	Unit Price

QUANTITY table

Salesperson Number	Product Number	Quantity

Third Normal Form

SALESPERSON table

Salesperson Number	Salesperson Name	Commission Percentage	Year of Hire	Department Number

DEPARTMENT table

Department Number	Manager Name

PRODUCT table

Product Number	Product Name	Unit Price

QUANTITY table

Salesperson Number	Product Number	Quantity

Normalization Example 4: patient and subscription

Wellmeadows Hospital
Patient Medication Form

Patient Number: ______________ Patient Name: ______________

Ward Number: ______________
Ward Name: ________

Drug Number	Name	Description	Dosage	Method of Admin	Units per Day	Start Date	Finish Date
10223	Morphine	Pain Killer	10 mg/ml	Oral	50	03/24/04	04/24/04
10334	Tetracyclene	Antibiotic	0.5 mg/ml	IV	10	03/24/04	04/17/04
10223	Morphine	Pain Killer	10 mg/ml	Oral	10	04/25/04	05/24/04

Functional Dependencies:

Patient No → Full Name

Ward No → Ward Name

Drug No → Name, Description, Dosage, Method of Admin

Patient No, Drug No, Start Date → Units per Day, Finish date

- **First Normal Form**

  Patient No, Drug No, Start Date, Full Name, Ward No, Ward Name, Bed No, Name, Description, Dosage, Method of Admin, Units per Day, Finish Date

- **Second Normal Form**

  Patient No, Drug No, Start Date, Ward No, Ward Name, Bed No, Units per Day, Finish Date

  Drug No, Name, Description, Dosage, Method of Admin

  Patient No, Full Name

- **Third Normal Form/BCNF**

  Patient No, Drug No, Start Date, Ward No, Bed No, Units per Day, Finish Date

  Drug No, Name, Description, Dosage, Method of Admin

  Patient No, Full Name

  Ward No, Ward Name

Normalization Example 5: car-owner and car

Car-owner and car

- Unnormalized Relation

  Owner-Car (Owner-ID, Owner-Name, Address, {Registration-No, Model, Manufacturer, No-Cylinders, Dealer, Dealer-Address, Sales-Amount})

  {} -- repeating group

- 1NF -- removes repeating groups

  Owner (Owner-ID, Owner-Name, Address)

  Owner-Registration (Owner-ID, Registration-No, Model, Manufacturer, No-Cylinders, Dealer, Dealer-Address, Sales-Amount)

Car-owner and car

- 1NF

  Owner (Owner-ID, Owner-Name, Address)

  Owner-Registration (Owner-ID, Registration-No, Model, Manufacturer, No-Cylinders, Dealer, Dealer-Address, Sales-Amount)

- 2NF -- removes non-full dependencies on primary key

  Owner (Owner-ID, Owner-Name, Address)

  Owner-Registration (Owner-ID, Registration-No)

  Registration (Registration-No, Model, Manufacturer, No-Cylinders, Dealer, Dealer-Address, Sales-Amount)

Car-owner and car

- 2NF

  Owner (<u>Owner-ID</u>, Owner-Name, Address)

  Owner-Registration (<u>Owner-ID, Registration-No</u>)

  Registration (<u>Registration-No</u>, Model, Manufacturer, No-Cylinders, Dealer, Dealer-Address, Sales-Amount)

- 3NF -- removes transitive dependencies

  Owner (<u>Owner-ID</u>, Owner-Name, Address)

  Owner-Registration (<u>Owner-ID, Registration-No</u>)

  Registration (<u>Registration-No</u>, Model, Manufacturer, No-Cylinders, Dealer, Sales-Amount)

  Dealer (<u>Dealer</u>, Dealer-Address)

Intuitive Normalization

1NF	Tables represent entities
2NF	Each table represents only one entity
3NF	Tables do not contain attributes from embedded entities

JDBC

Programs with Embedded SQL

Host language + Embedded SQL

Preprocessor

Host Language + function calls

Host language compiler

Host language program

Call-level interface (CLI): ODBC,JDBC, ADO

JDBC

- Call-level interface (CLI) for executing SQL from a Java program
- SQL statement is constructed at run time as the value of a Java variable (as in dynamic SQL)
- JDBC passes SQL statements to the underlying DBMS. Can be interfaced to any DBMS that has a JDBC driver
- Part of SQL:2003

JDBC Run-Time Architecture

Steps to execute queries using JDBC

1. Register Oracle Driver

   DriverManager.registerDriver(new oracle.jdbc.OracleDriver())

2. Establish connection to DB server

   Connection con =
   DriverManager.getConnection(<url>,<username>,<password>);

   <url> identifies which Oracle Driver to use, connect to which database, on which port and what is the service name.

3. Create Statement

   Statement sta = con.createStatement();

Steps to execute queries using JDBC (contd..)

4. Execute Query

```
ResultSet query = sta.executeQuery(<Query>);
```

5. Display/Process Result

```
while(query.next()) {
//process data from tuples.
}
```

6. Close connection

```
query.close();
sta.close();
con.close();
```

Executing a Query

import java.sql.*; -- *import all classes in package* java.sql

Class.forName (*driver name*); // *static method of class* Class
// *loads specified driver*

Connection con = DriverManager.getConnection(*Url, Id, Passwd*);

- *Static method of class* DriverManager; *attempts to connect to DBMS*
- *If successful, creates a connection object,* con, *for managing the connection*

Statement stat = con.createStatement ();

- *Creates a statement object* stat
- *Statements have* executeQuery() *method*

Executing a Query (cont’d)

```
String query = “SELECT   T.StudId  FROM  Transcript T” +
                  “WHERE   T.CrsCode = ‘cse305’ ” +
                  “AND  T.Semester = ‘S2000’ ”;
ResultSet  res = stat.executeQuery (query);
```

- *Creates a result set object,* res*.*
- *Prepares and executes the query.*
- *Stores the result set produced by execution in* res *(analogous to opening a cursor).*
- *The query string can be constructed at run time (as above).*
- *The input parameters are plugged into the query when the string is formed (as above)*

Preparing and Executing a Query

String query = "SELECT T.*StudId* FROM Transcript T" +
"WHERE T.*CrsCode* = ? AND T.*Semester* = ?";

placeholders

PreparedStatement ps = con.prepareStatement (query);

- *Prepares the statement*
- *Creates a prepared statement object,* ps, *containing the prepared statement*
- **Placeholders** (?) *mark positions of* **in** *parameters; special API is provided to plug the actual values in positions indicated by the* ?*'s*

Preparing and Executing a Query (cont’d)

```
String  crs_code, semester;
………
ps.setString(1, crs_code);     // set value of first in parameter
ps.setString(2, semester);     // set value of second in parameter

ResultSet  res = ps.executeQuery ( );
```

- *Creates a result set object,* res
- *Executes the query*
- *Stores the result set produced by execution in* res

```
while  ( res.next ( ) )  {                    // advance the cursor
      j = res.getInt (“StudId”);               // fetch output int-value
      …process output value…
}
```

Result Sets and Cursors

- Three types of result sets in JDBC:
  - *Forward-only*: not scrollable
  - *Scroll-insensitive*: scrollable; changes made to underlying tables after the creation of the result set are not visible through that result set
  - *Scroll-sensitive*: scrollable; updates and deletes made to tuples in the underlying tables after the creation of the result set are visible through the set

Result Set

```
Statement stat = con.createStatement (
    ResultSet.TYPE_SCROLL_SENSITIVE,
    ResultSet.CONCUR_UPDATABLE );
```

- Any result set type can be declared *read-only* or *updatable* – CONCUR_UPDATABLE (assuming SQL query satisfies the conditions for updatable views)
- *Updatable*: Current row of an updatable result set can be changed or deleted, or a new row can be inserted. Any such change causes changes to the underlying database table

```
res.updateString ("Name", "John" ); // change the attribute "Name" of
                                    // current row in the row buffer.
res.updateRow ( ); // install changes to the current row buffer
                   // in the underlying database table
```

Handling Exceptions

```
try {
    ...Java/JDBC code...
}  catch ( SQLException  ex ) {
    ...exception handling code...
}
```

- try/catch is the basic structure within which an SQL statement should be embedded
- If an exception is thrown, an exception object, *ex*, is created and the catch clause is executed
- The exception object has methods to print an error message, return SQLSTATE, etc.

Transactions

Example: Bad Interaction

- You and your domestic partner each take $100 from different ATM's at about the same time.
  - The DBMS better make sure one account deduction doesn't get lost.

- Compare: An OS allows two people to edit a document at the same time. If both write, one's changes get lost!

An Example: Interacting Processes

- Assume the usual Sells(bar,beer,price) relation, and suppose that Joe's Bar sells only Bud for $2.50 and Miller for $3.00.
- Sally is querying Sells for the highest and lowest price Joe charges.
- Joe decides to stop selling Bud and Miller, but to sell only Heineken at $3.50.

Sally's Program

- Sally executes the following two SQL statements, which we call (min) and (max), to help remember what they do.

(max)SELECT MAX(price) FROM Sells
WHERE bar = 'Joe''s Bar';

(min) SELECT MIN(price) FROM Sells
WHERE bar = 'Joe''s Bar';

Joe's Program

- At about the same time, Joe executes the following steps, which have the mnemonic names (del) and (ins).

```
(del)   DELETE FROM Sells
        WHERE bar = 'Joe''s Bar';
(ins)   INSERT INTO Sells
        VALUES('Joe''s Bar', 'Heineken', 3.50);
```

Interleaving of Statements

- Although (max) must come before (min), and (del) must come before (ins), there are no other constraints on the order of these statements, unless we group Sally's and/or Joe's statements into transactions.

Example: Strange Interleaving

- Suppose the steps execute in the order (max)(del)(ins)(min).

Joe's Prices:

Statement:

Result:

Joe's Prices:	2.50, 3.00	2.50, 3.00		3.50
Statement:	(max)	(del)	(ins)	(min)
Result:	3.00			3.50

- Sally sees MAX < MIN!

Transaction Concept

- A ***transaction*** is a *unit* of program execution that accesses and possibly updates various data items
- A transaction must see a consistent database
- During transaction execution the database may be inconsistent
- When the transaction is committed, the database must be consistent
- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

ACID Properties

- **To preserve integrity of data,**
- **Atomicity**
  - Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency**
  - Execution of a transaction in isolation preserves the consistency of the database.
- **Isolation**
  - Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions.
  - Intermediate transaction results must be hidden from other concurrently executed transactions.
  - That is, for every pair of transactions T_i and T_j, it appears to T_i that either T_j, finished execution before T_i started, or T_j started execution after T_i finished.
- **Durability**
  - After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Example of Fund Transfer

- Transaction to transfer $50 from account *A* to account *B*:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B*)

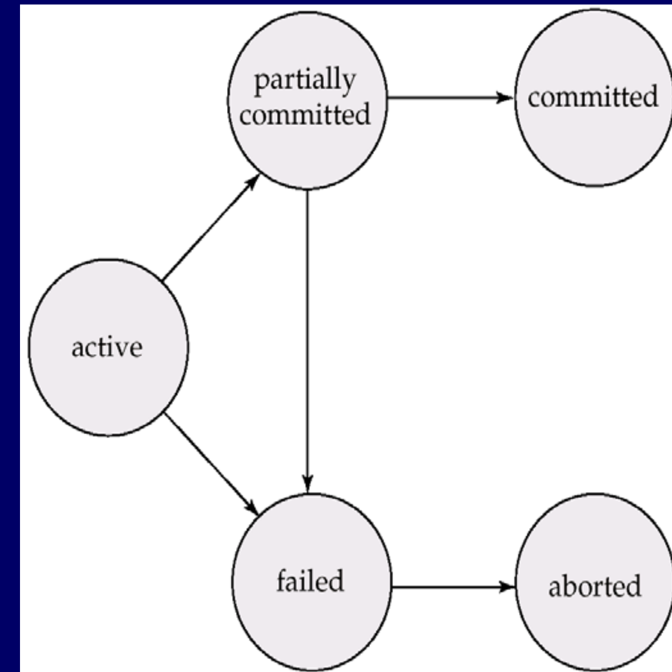
- Consistency requirement
  – the sum of *A* and *B* is unchanged by the execution of the transaction.

- Atomicity requirement
  – if the transaction fails after step 3 and before step 6, the system should ensure that its updates are not reflected in the database, else an inconsistency will result.

Example of Fund Transfer (Cont.)

- Durability requirement
  - once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist despite failures.
- Isolation requirement
  - if between steps 3 and 6, another transaction is allowed to access the partially updated database, it will see an inconsistent database (the sum $A + B$ will be less than it should be).
  - Can be ensured trivially by running transactions ***serially***, that is one after the other. However, executing multiple transactions concurrently has significant benefits, as we will see.

Transaction State



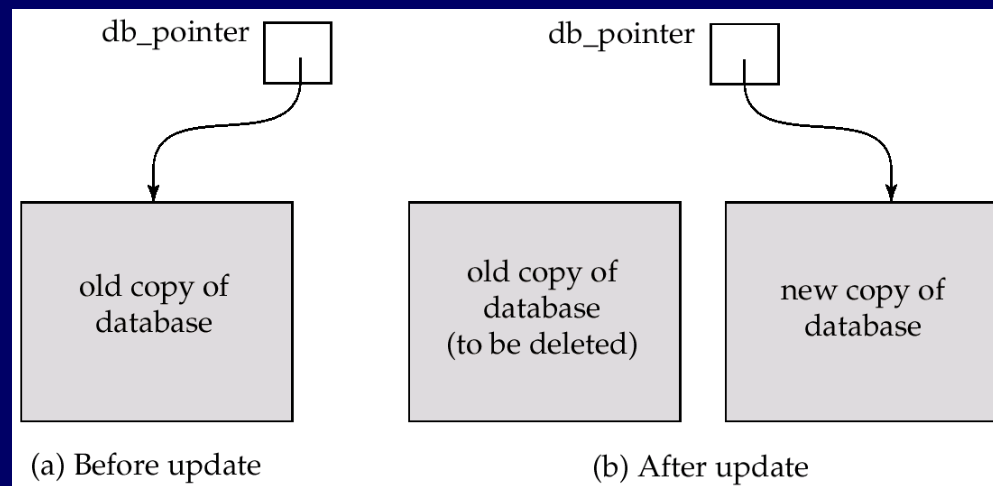
- **Active**
  - the initial state; the transaction stays in this state while it is executing
- **Partially committed**
  - after the final statement has been executed.
- **Failed**
  - after the discovery that normal execution can no longer proceed.
- **Aborted**
  - after the transaction has been rolled back and the database restored to its state prior to the start of the transaction.
  - Two options after it has been aborted:
    - restart the transaction – only if no internal logical error
    - kill the transaction
- **Committed:** after *successful completion*.

Implementation of Atomicity and Durability

- The recovery-management component of a database system implements the support for atomicity and durability.
- Recovery Schemes: Log-based approach vs Shadowing approach
- The *shadow-database* scheme:
  - assume that only one transaction is active at a time.
  - a pointer called db_pointer always points to the current consistent copy of the database.
  - all updates are made on a *shadow copy* of the database, and **db_pointer** is made to point to the updated shadow copy only after the transaction reaches partial commit and all updated pages have been flushed to disk.
  - in case transaction fails, old consistent copy pointed to by **db_pointer** can be used, and the shadow copy can be deleted.

Implementation of Atomicity and Durability (Cont.)

- The shadow-database scheme:
  - Assumes disks to not fail
  - Simple & Useful for text editors, but extremely inefficient for large databases: executing a single transaction requires copying the *entire* database.



Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system.
- Advantages are:
  - **increased processor and disk utilization**, leading to better transaction *throughput:*
    - one transaction can be using the CPU while another is reading from or writing to the disk
  - **reduced average response time** for transactions:
    - short transactions need not wait behind long ones.
- *Concurrency control schemes*
  - mechanisms to achieve isolation, i.e., to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database

Schedules

- *Schedules*
  - sequences that indicate the chronological order in which instructions of concurrent transactions are executed
  - a schedule for a set of transactions must consist of all instructions of those transactions
  - must preserve the order in which the instructions appear in each individual transaction.
- Serial Schedule
  - instruction sequences from one by one transactions

Example Schedule: Schedule 1

- Let T_1 transfer $50 from *A* to *B*, and T_2 transfer 10% of the balance from *A* to *B.*
- The following is <u>a serial schedule</u> in which T_1 is followed by T_2.

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

Schedule 2 -- Another Serial Schedule

- The following is a serial schedule in which T2 is followed by T1.

T_1	T_2
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	

Read-Only vs. Read-Write Transactions

- We can tell the DBMS that we won’t be performing any updates:

```
SET TRANSACTION READ ONLY;
SELECT * FROM Accounts
WHERE account#='1234';
```

- If we are going to modify the DBMS, we need:

```
SET TRANSACTION READ WRITE;
UPDATE Accounts
SET balance = balance - $100
WHERE account#= '1234'; ...
```

Isolation

- General rules of thumb w.r.t. isolation:
  - Fully serializable isolation is expensive
    - We can't do as many things concurrently (or we have to undo them frequently)
  - For performance, the DBMS lets you relax the *isolation level* if your application can tolerate it, e.g:

```
SET TRANSACTION READ WRITE
ISOLATION LEVEL READ UNCOMMITTED;
```

Fixing Sally's Problem by Using Transactions

- If we group Sally's statements (max)(min) into one transaction, then she cannot see this inconsistency.
- She sees Joe's prices at some fixed time.
  - Either before or after he changes prices, or in the middle, but the MAX and MIN are computed from the same prices.

Another Problem: Rollback

- Suppose Joe executes (del)(ins), not as a transaction, but after executing these statements, thinks better of it and issues a ROLLBACK statement.
- If Sally executes her statements after (ins) but before the rollback, she sees a value, 3.50, that never existed in the database.

Solution

- If Joe executes (del)(ins) as a transaction, its effect cannot be seen by others until the transaction executes COMMIT.
  - If the transaction executes ROLLBACK instead, then its effects can *never* be seen.

Transactions in JDBC

- Default for a connection is
  - Transaction boundaries
    - *Autocommit mode*: each SQL statement is a transactio
    - To group several statements into a transaction use con.setAutoCommit (false)
  - Isolation
    - default isolation level of the underlying DBMS
    - To change isolation level use con.setTransactionIsolationLevel (TRANSACTION_SERIALIZABLE)
- With autocommit off:
  - transaction is committed using con.commit().
  - next transaction is automatically initiated (chaining)
- Transactions on each connection committed separate

Transactions in JDBC - example

```
con.setAutoCommit(false);
PreparedStatement updateSales =
    con.prepareStatement( "UPDATE COFFEES SET SALES = ?
            WHERE       COF_NAME LIKE ?");
updateSales.setInt(1, 50);
updateSales.setString(2, "Colombian");
updateSales.executeUpdate();
PreparedStatement updateTotal =
    con.prepareStatement( "UPDATE COFFEES SET TOTAL =
            TOTAL + ?     WHERE COF_NAME LIKE ?");
updateTotal.setInt(1, 50);
updateTotal.setString(2, "Colombian");
updateTotal.executeUpdate();
con.commit();
con.setAutoCommit(true);
```