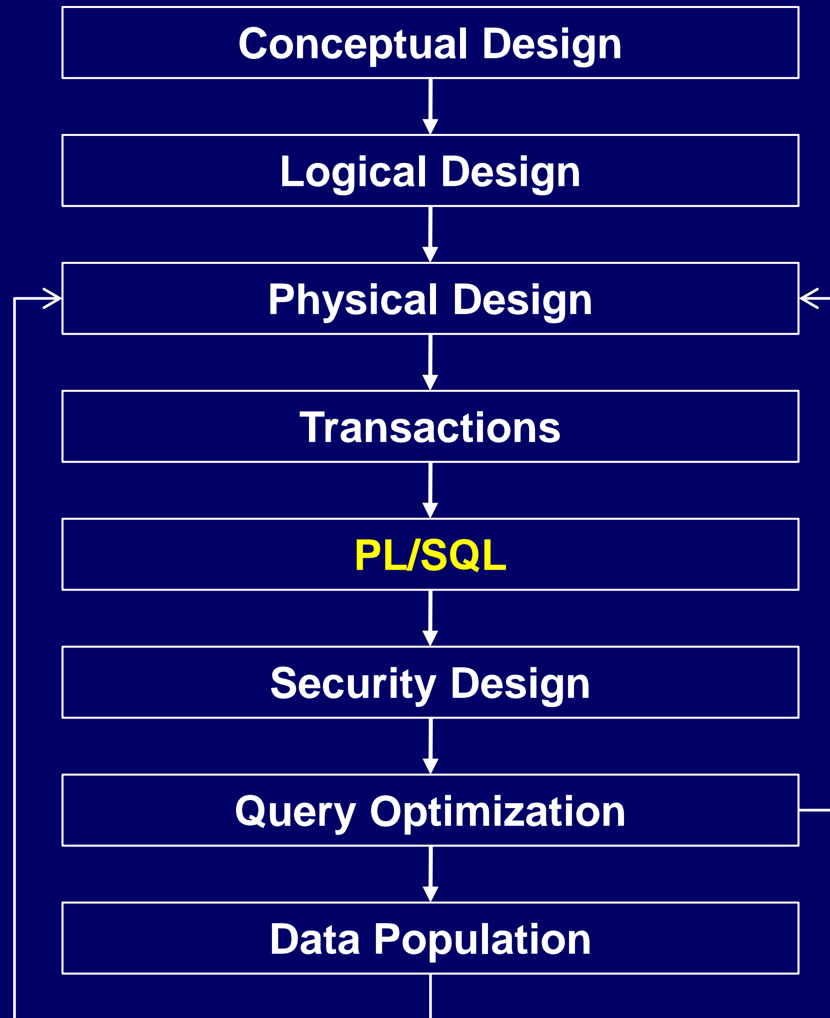
CSCD43: Database Systems Technology

Lecture 5

Wael Aboulsaadat

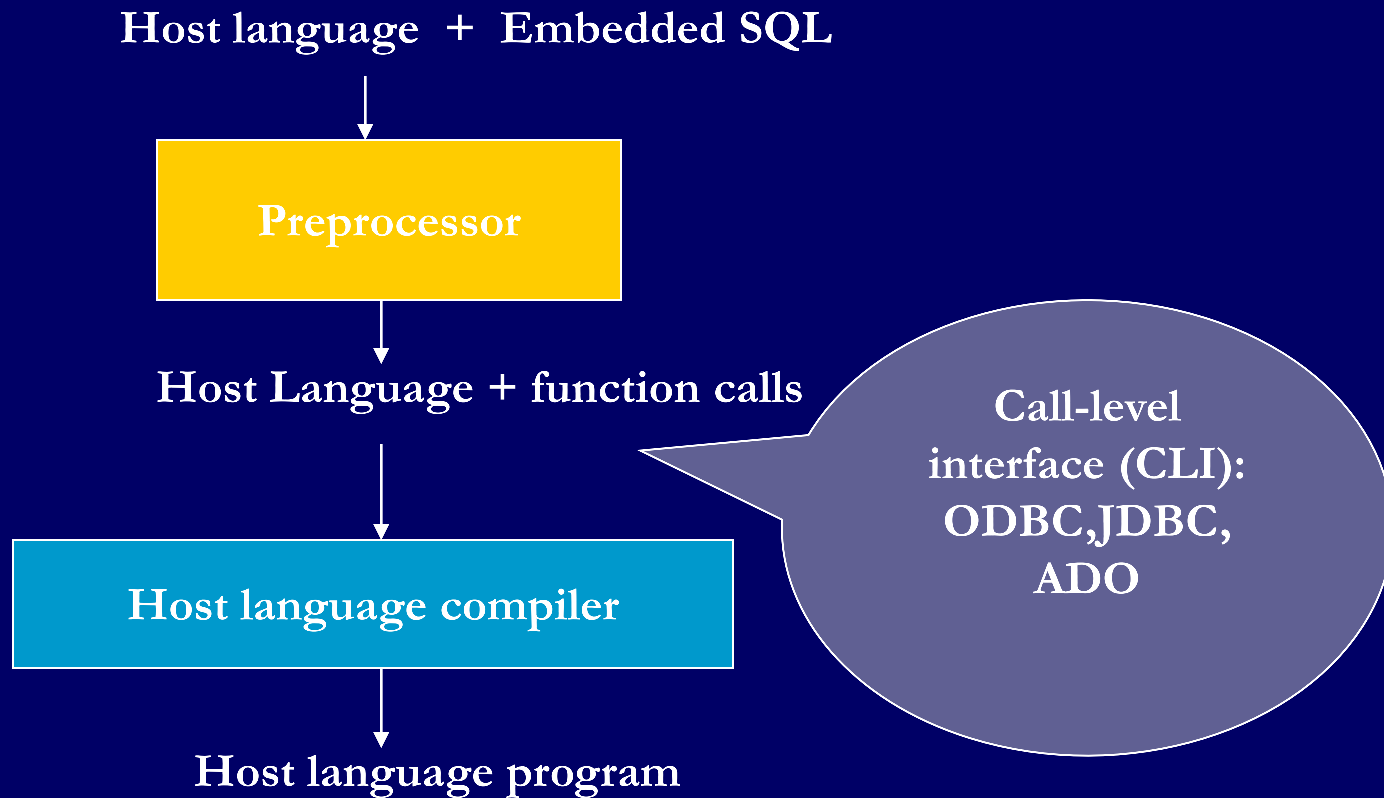
Acknowledgment: these slides are based on Prof. Garcia-Molina & Prof. Ullman slides accompanying the course's textbook.

Steps in Database Design



JDBC

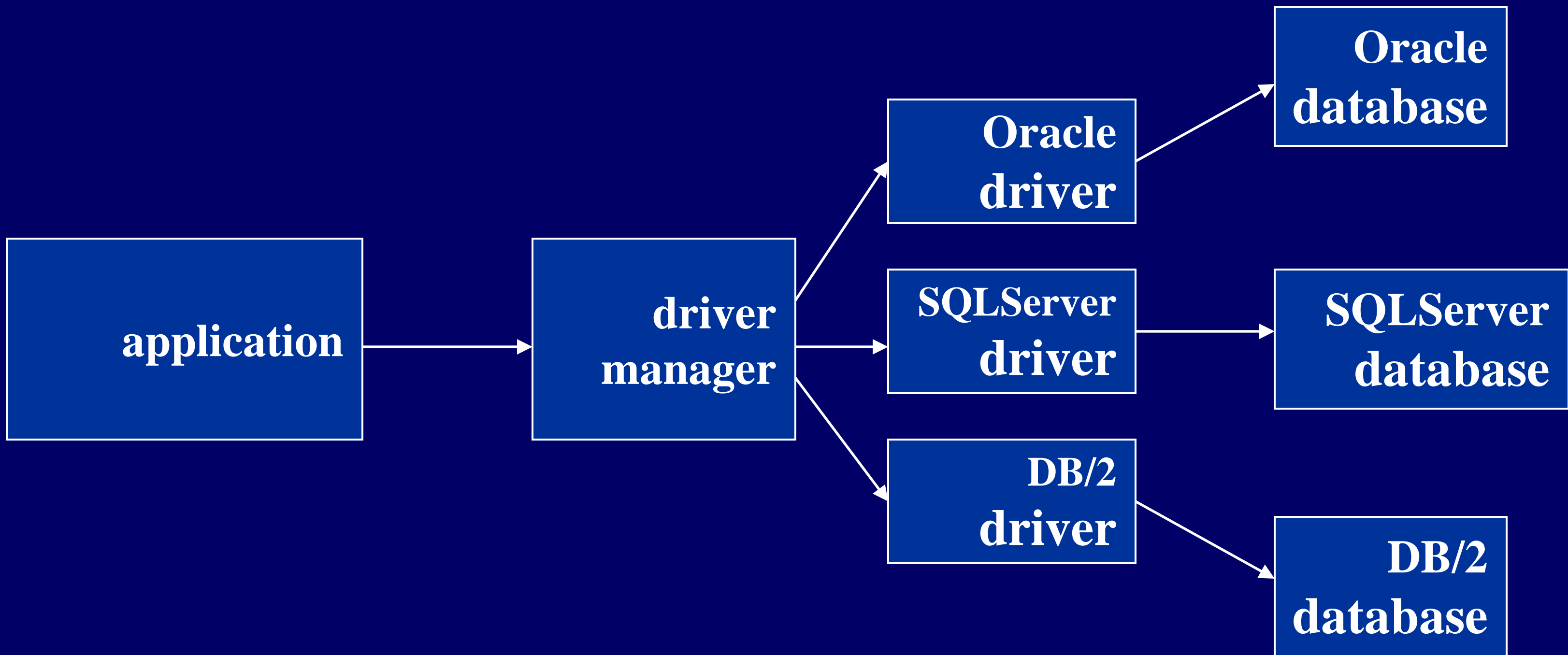
Programs with Embedded SQL



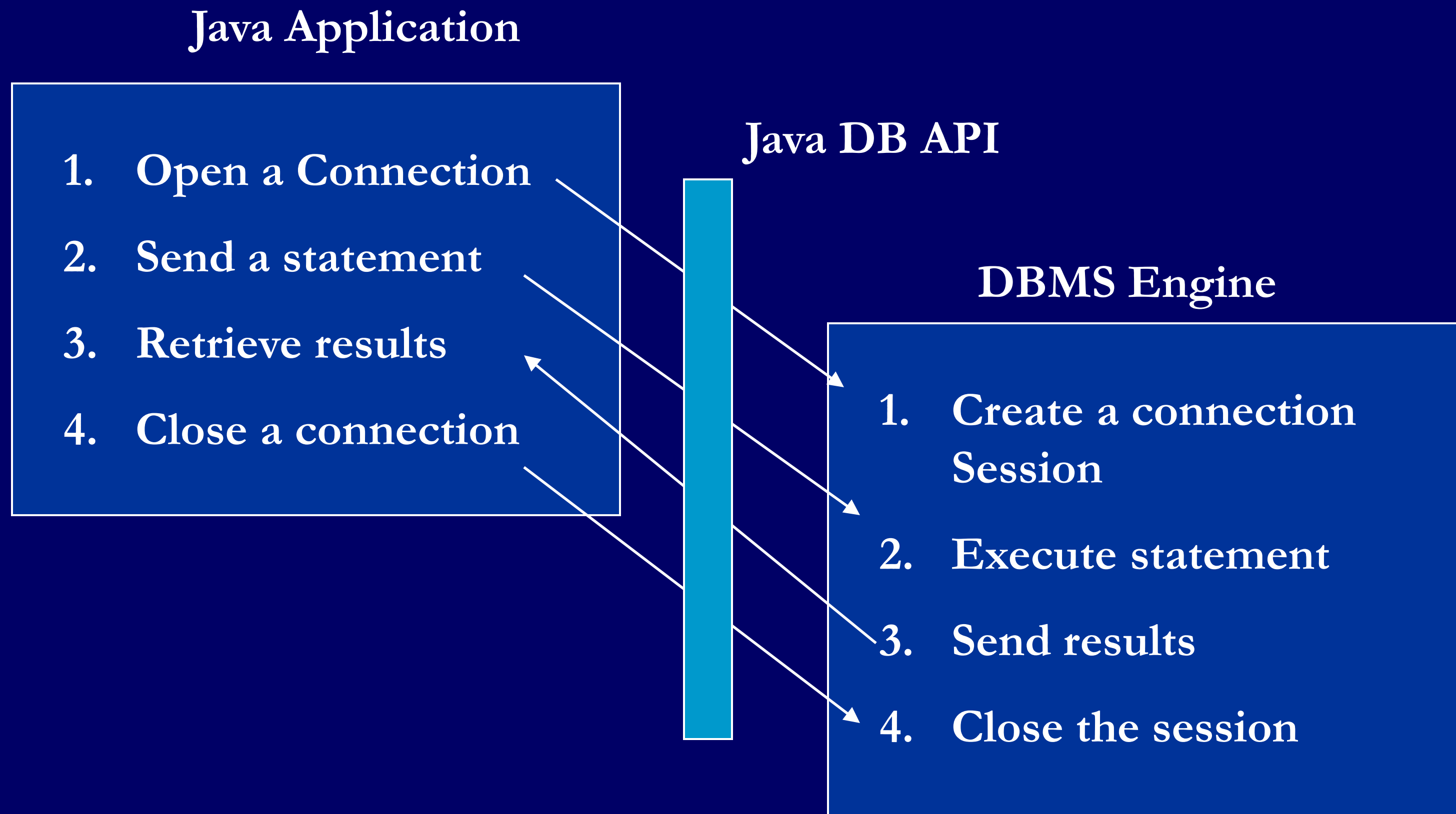
JDBC

- 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



What you expect in DB API



Steps to execute queries using JDBC

1. Register Oracle Driver

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

Or

```
Class.forName("org.postgresql.Driver" );
```

2. Establish connection to DB server

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

<url> identifies which 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, *fo 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 = ?";
```

```
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 in JDBC

- As with ODBC, each statement gets committed automatically in JDBC
- To turn off auto commit use
  conn.setAutoCommit(false);
- To commit or abort transactions use
  conn.commit()   or conn.rollback()
- To turn auto commit on again, use
  conn.setAutoCommit(false);

JDBC Data Types

Many of the non-primitive types are defined in the java.sql pacakge.

- As SQL defines its own datatypes, it is imp[…] understand the mapping between SQL an[…] datatypes…

SQL	Java	SQL	Java
BIT	boolean	NUMERIC	BigDecimal
BIGINT	long	REAL	float
BINARY	byte[]	SMALLINT	short
CHAR	String	TIME	Time
DATE	Date	TIMESTAMP	Timestamp
DOUBLE	double	TINYINT	byte
FLOAT	float	VARBINARY	byte[]
INTEGER	int	VARCHAR	char[]
BLOB	Blob	REF	Ref
CLOB	Clob	STRUCT	Struct

Callable Statement

- Used for executing stored procedures
- Example

```
String createProcedure = “Create Procedure ShowGoodStudents” + “as
    Select Name from CS4400 where Marks > 90)”;
Stmt.executeUpdate(createProcedure);

CallableStatement cs = con.prepareCall(“(call ShowGoodStudents)”);
ResultSet rs = cs.executeQuery();
```

Callable Statement

- Passing IN parameters is done using the setXXX methods inherited from PreparedStatement
  - pstmt.setLong(1, 12345);
  - pstmt.setLong(2, 345);

- If OUT parameters are used, the JDBC type of each OUT parameter must be registered before execution
  - cstmt.registerOutParameter(1,java.sql.Types.TINYINT);
  - Then use getXXX methods to retrieve OUT parameter values
    - byte x = cstmt.getByte(1);

CallableStatement

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection con
        = DriverManager.getConnection("jdbc:odbc:uma","kworker","kworker");

//calling a stored procedure with no input/output param
CallableStatement cs1 = con.prepareCall("{call HELLOWORLD}");
 ResultSet rs1 = cs1.executeQuery();
 while(rs1.next())
 {
        String one = rs1.getString(“Column-name");
        System.out.println(one);
}
```

CallableStatement – cont'd

```
CallableStatement cs2 = con.prepareCall("{call ADDITION(?,?,?)}");
cs2.setInt(1,10);
cs2.setInt(2,25);
cs2.registerOutParameter(3,java.sql.Types.INTEGER);
cs2.execute();
int res = cs2.getInt(3);
System.out.println(res);

CallableStatement cs = con.prepareCall("{call ACCOUNTLOGIN(?,?,?)}");
cs.setString(1,theuser);
cs.setString(2,password);
cs.registerOutParameter(3,Types.DATE);
cs.executeQuery();
Date lastLogin = cs.getDate(3);
```

Views

Views

- A *view* is a relation defined in terms of stored tables (called *base tables*) and other views.
- Two kinds:
  1. *Virtual* = not stored in the database; just a query for constructing the relation.
  2. *Materialized* = actually constructed and stored.

Declaring Views

- Declare by:

  CREATE [MATERIALIZED] VIEW
  <name> AS <query>;

- Default is virtual.

Example: View Definition

- CanDrink(drinker, beer) is a view “containing” the drinker-beer pairs such that the drinker frequents at least one bar that serves the beer:

```
CREATE VIEW CanDrink AS
    SELECT drinker, beer
    FROM Frequents, Sells
    WHERE Frequents.bar = Sells.bar;
```

Example: Accessing a View

- Query a view as if it were a base table.
  - Also: a limited ability to modify views if it makes sense as a modification of one underlying base table.
- Example query:

```
SELECT beer FROM CanDrink
WHERE drinker = 'Sally';
```

Triggers on Views

- Generally, it is impossible to modify a virtual view, because it doesn't exist on disk.
- But an INSTEAD OF trigger lets us interpret view modifications in a way that makes sense.
- Example: View Synergy has (drinker, beer, bar) triples such that the bar serves the beer, the drinker frequents the bar and likes the beer.

Example: The View

Pick one copy of each attribute

```
CREATE VIEW Synergy AS
  SELECT Likes.drinker, Likes.beer, Sells.bar
  FROM Likes, Sells, Frequents
  WHERE Likes.drinker = Frequents.drinker
     AND Likes.beer = Sells.beer
     AND Sells.bar = Frequents.bar;
```

Natural join of Likes, Sells, and Frequents

Interpreting a View Insertion

- We cannot insert into Synergy --- it is a virtual view.
- But we can use an INSTEAD OF trigger to turn a (drinker, beer, bar) triple into three insertions of projected pairs, one for each of Likes, Sells, and Frequents.
  - Sells.price will have to be NULL.

The Trigger

```
CREATE TRIGGER ViewTrig
    INSTEAD OF INSERT ON Synergy
    REFERENCING NEW ROW AS n
    FOR EACH ROW
    BEGIN
        INSERT INTO LIKES VALUES(n.drinker, n.beer);
        INSERT INTO SELLS(bar, beer) VALUES(n.bar, n.beer);
        INSERT INTO FREQUENTS VALUES(n.drinker, n.bar);
    END;
```

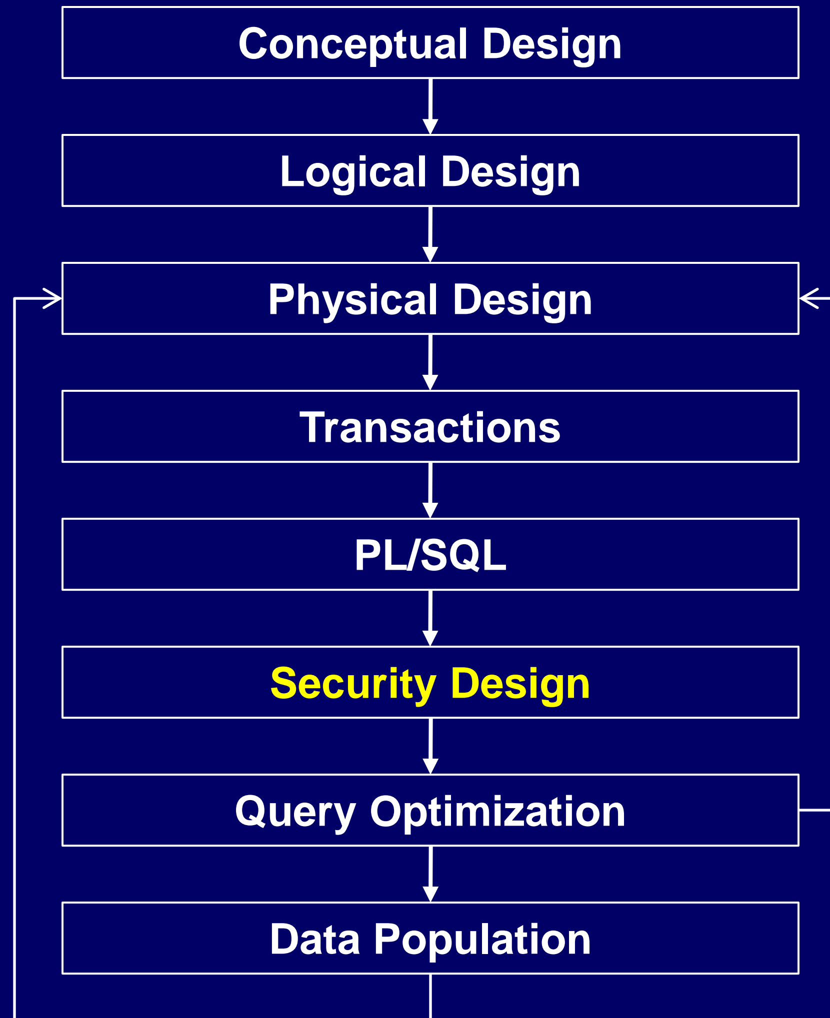
Materialized Views

- Problem: each time a base table changes, the materialized view may change.
  - Cannot afford to recompute the view with each change.
- Solution: Periodic reconstruction of the materialized view, which is otherwise “out of date.”

Example: A Data Warehouse

- Wal-Mart stores every sale at every store in a database.
- Overnight, the sales for the day are used to update a *data warehouse* = materialized views of the sales.
- The warehouse is used by analysts to predict trends and move goods to where they are selling best.

Steps in Database Design



SQL Authorization

Privileges

Grant and Revoke

Grant Diagrams

Authorization

- A file system identifies certain privileges on the objects (files) it manages.
  - Typically read, write, execute.
- A file system identifies certain participants to whom privileges may be granted.
  - Typically the owner, a group, all users.

Privileges --- 1

- SQL identifies a more detailed set of privileges on objects (relations) than the typical file system.
- Nine privileges in all, some of which can be restricted to one column of one relation.

Privileges --- 2

- Some important privileges on a relation:
  1. SELECT = right to query the relation.
  2. INSERT = right to insert tuples.
     - May apply to only one attribute.
  3. DELETE = right to delete tuples.
  4. UPDATE = right to update tuples.
     - May apply to only one attribute.

Example: Privileges

- For the statement below:

```
INSERT INTO Beers(name)
    SELECT beer FROM Sells
    WHERE NOT EXISTS
        (SELECT * FROM Beers
         WHERE name = beer);
```

beers that do not appear in Beers. We add them to Beers with a NULL manufacturer.

- We require privileges SELECT on Sells and Beers, and INSERT on Beers or Beers.name.

Authorization ID's

- A user is referred to by *authorization ID*, typically their name.
- There is an authorization ID PUBLIC.
  - Granting a privilege to PUBLIC makes it available to any authorization ID.

Granting Privileges

- You have all possible privileges on the objects, such as relations, that you create.
- You may grant privileges to other users (authorization ID's), including PUBLIC.
- You may also grant privileges WITH GRANT OPTION, which lets the grantee also grant this privilege.

The GRANT Statement

- To grant privileges, say:

  GRANT <list of privileges>

  ON <relation or other object>

  TO <list of authorization ID's>;

- If you want the recipient(s) to be able to pass the privilege(s) to others add:

  WITH GRANT OPTION

Example: GRANT

- Suppose you are the owner of Sells. You may say:

```
GRANT SELECT, UPDATE(price)
ON Sells
TO sally;
```

- Now Sally has the right to issue any query on Sells and can update the price component only.

Example: Grant Option

- Suppose we also grant:

```
GRANT UPDATE ON Sells TO sally
WITH GRANT OPTION;
```

- Now, Sally can not only update any attribute of Sells, but can grant to others the privilege UPDATE ON Sells.
  - Also, she can grant more specific privileges like UPDATE(price) ON Sells.

Revoking Privileges

```
REVOKE <list of privileges>
ON <relation or other object>
FROM <list of authorization ID's>;
```

- Your grant of these privileges can no longer be used by these users to justify their use of the privilege.
  - But they may still have the privilege because they obtained it independently from elsewhere.

REVOKE Options

- We must append to the REVOKE statement either:
  1. CASCADE. Now, any grants made by a revokee are also not in force, no matter how far the privilege was passed.
  2. RESTRICT. If the privilege has been passed to others, the REVOKE fails as a warning that something else must be done to "chase the privilege down."

Grant Diagrams

- Nodes = user/privilege/option/isOwner?
  - UPDATE ON R, UPDATE(a) on R, and UPDATE(b) ON R live in different nodes.
  - SELECT ON R and SELECT ON R WITH GRANT OPTION live in different nodes.
- Edge *X* -> *Y* means that node *X* was used to grant *Y*.

Notation for Nodes

- Use *AP* for the node representing authorization ID *A* having privilege *P*.
  - *P* * represents privilege *P* with grant option.
  - *P* ** represents the source of the privilege *P*. That is, *AP* ** means *A* is the owner of the object on which *P* is a privilege.
    - Note ** implies grant option.

Manipulating Edges --- 1

- When *A* grants *P* to *B*, We draw an edge from *AP* * or *AP* ** to *BP*.
  - Or to *BP* * if the grant is with grant option.
- If *A* grants a subprivilege *Q* of *P* (say UPDATE(a) on R when *P* is UPDATE ON R) then the edge goes to *BQ* or *BQ* *, instead.

Manipulating Edges --- 2

- Fundamental rule: user *C* has privilege *Q* as long as there is a path from *XQ* ** (the origin of privilege *Q*) to *CQ*, *CQ* *, or CQ**.
  - Remember that XQ** could *be* CQ**.

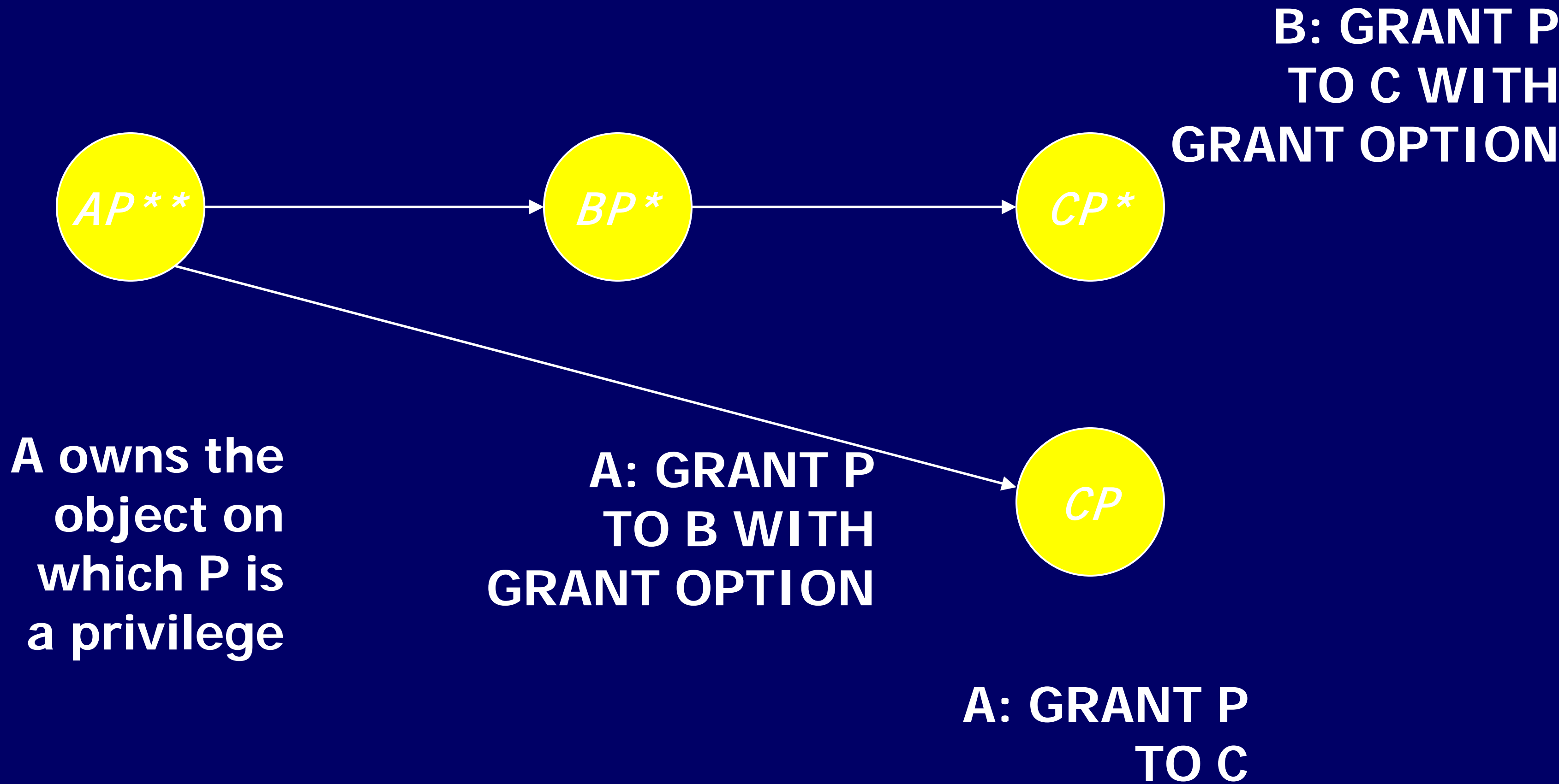
Manipulating Edges --- 3

- If *A* revokes *P* from *B* with the CASCADE option, delete the edge from *AP* to *BP*.
- If *A* uses RESTRICT, and there is an edge from *BP* to anywhere, then reject the revocation and make no change to the graph.

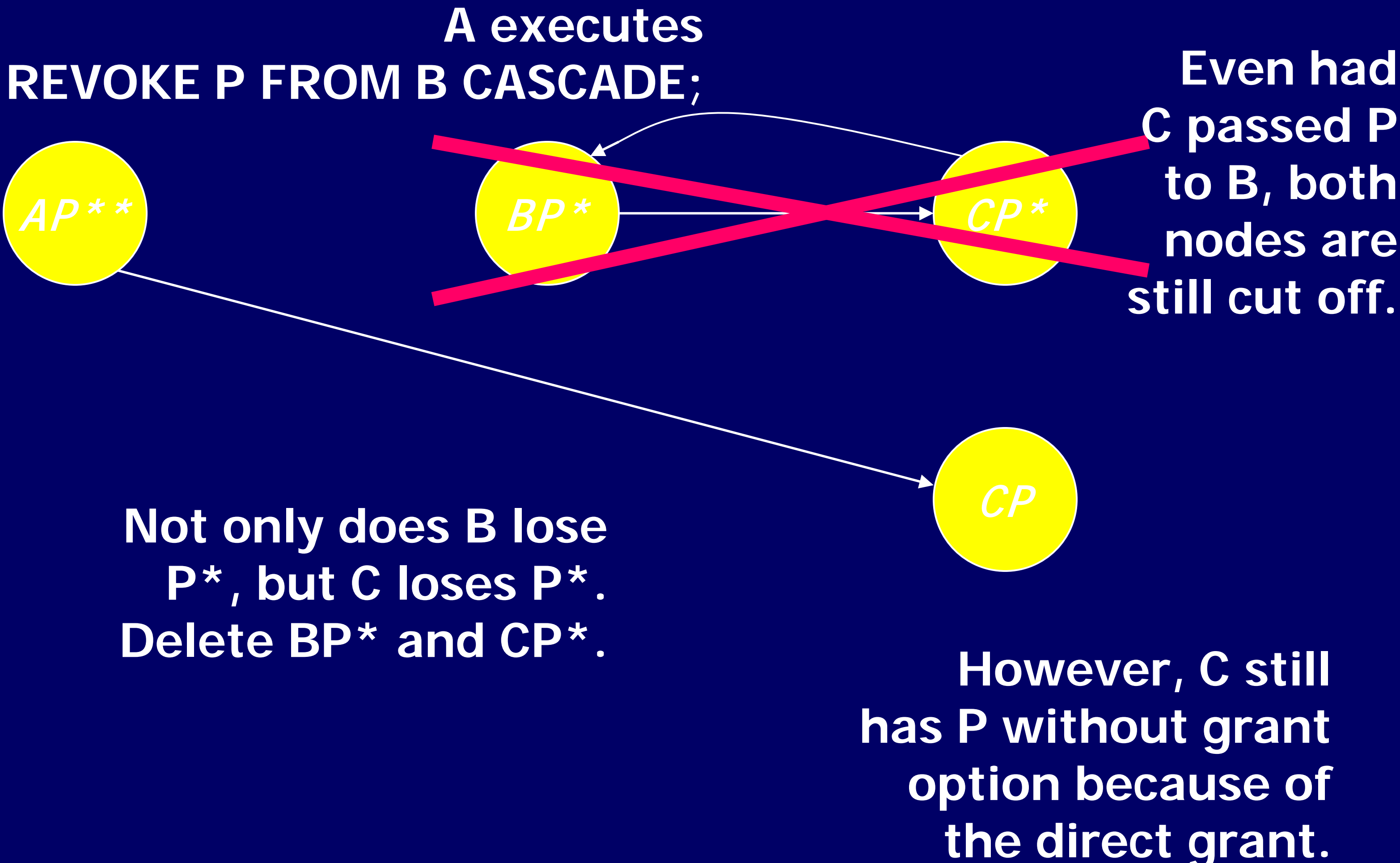
Manipulating Edges --- 4

- Having revised the edges, we must check that each node has a path from some ** node, representing ownership.
- Any node with no such path represents a revoked privilege and is deleted from the diagram.

Example: Grant Diagram



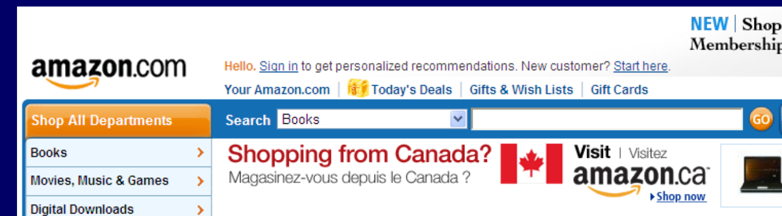
Example: Grant Diagram



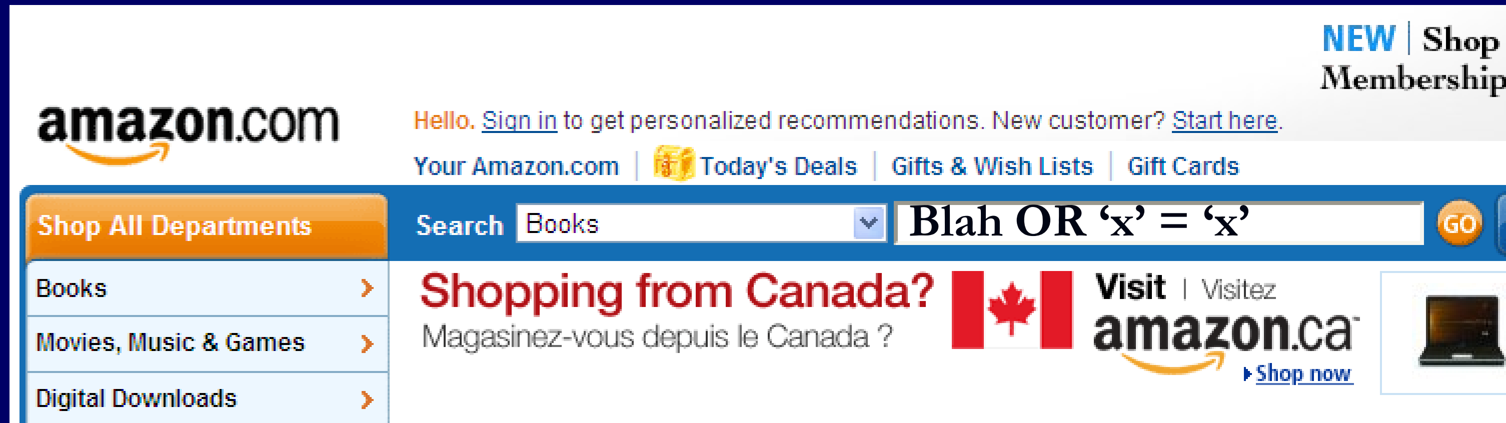
SQL Injection

What is a SQL Injection Attack?

- Many web applications take user input from a form
- Often this user input is used literally in the construction of a SQL query submitted to a database. For example:
  - SELECT productdata FROM table WHERE productname = '***user input product name***';
- A SQL injection attack involves placing SQL statements in the user input



An Example SQL Injection Attack



- This input is put directly into the SQL statement within the Web application:
  - $query = "SELECT prodinfo FROM prodtable WHERE prodname = '" . $_POST['prod_search'] . "'";
- Creates the following SQL:
  - SELECT prodinfo FROM prodtable WHERE prodname = '**blah' OR 'x' = 'x'**
  - Attacker has now successfully caused the entire database to be returned.

A More Malicious Example

- What if the attacker had instead entered:
  - **blah‘; DROP TABLE prodinfo; --**
- Results in the following SQL:
  - SELECT prodinfo FROM prodtable WHERE prodname = ‘**blah’; DROP TABLE prodinfo; --’**
  - Note how comment (--) consumes the final quote
- Causes the entire database to be deleted
  - Depends on knowledge of table name
  - This is sometimes exposed to the user in debug code called during a database error
  - Use non-obvious table names, and never expose them to user
- Usually data destruction is not your worst fear, as there is low economic motivation

Other injection possibilities

- Using SQL injections, attackers can:
  - Add new data to the database
    - Could be embarrassing to find yourself selling politically incorrect items on an eCommerce site
    - Perform an INSERT in the injected SQL
  - Modify data currently in the database
    - Could be very costly to have an expensive item suddenly be deeply ‘discounted’
    - Perform an UPDATE in the injected SQL
  - Often can gain access to other user’s system capabilities by obtaining their password

Defenses

- Check syntax of input for validity
  - Many classes of input have fixed languages
    - Email addresses, dates, part numbers, etc.
    - Verify that the input is a valid string in the language
    - Sometime languages allow problematic characters (e.g., ‘*’ in email addresses); may decide to not allow these
    - If you can exclude quotes and semicolons that’s good
  - Not always possible: consider the name Bill O’Reilly
    - Want to allow the use of single quotes in names
- Have length limits on input
  - Many SQL injection attacks depend on entering long strings

Even More Defenses

- Scan query string for undesirable word combinations that indicate SQL statements
  - INSERT, DROP, etc.
  - If you see these, can check against SQL syntax to see if they represent a statement or valid user input
- Limit database permissions and segregate users
  - If you're only reading the database, connect to database as a user that only has read permissions
  - Never connect as a database administrator in your web application

More Defenses

- Configure database error reporting
  - Default error reporting often gives away information that is valuable for attackers (table name, field name, etc.)
  - Configure so that this information is never exposed to a user
- If possible, use bound variables
  - Some libraries allow you to bind inputs to variables inside a SQL statement
  - PERL example (from http://www.unixwiz.net/techtips/sql-injection.html)

```
$sth = $dbh->prepare("SELECT email, userid FROM members WHERE
    email = ?;");
$sth->execute($email);
```