Final Review

Slides based on “Database Management Systems” 3rd ed, Ramakrishnan and Gehrke
Grading

- Final exam (150 points) – open book
- Midterm (100 points) – open book
  - Open book does NOT include electronic devices!

- 6 homework assignments
  - 10 points each, drop lowest score.
  - Assignments for CS630 had a few additional questions
Course Overview

- Relational Data Model
- Relational Algebra (but not on final exam)
- Structured Query Language
  - The most important part of the course (still true!)
- Conceptual design – the ER model, views
- After midterm Exam:
  - Database application development
    - Java/JDBC, PL/SQL
    - Transactions
- Design Theory: FDs, Normalization
- Database Security: GRANT command, users
Lectures 1-13: see Midterm Review

- Final will skip relational algebra
- Final will have significant SQL, at least some ER
- SQL score can improve SQL score from midterm (separately recorded for this purpose)
- Note: after-midterm topics start at Lecture 14
Database Application Programming

- Embedded SQL
- Dynamic SQL

  Many host languages: C, Cobol, Pascal, etc.,
  but hardly in use any more: not on final exam

- JDBC: covered on final exam
- SQLJ (Embedded) – not in use, skip
- LINQ-to-SQL: for C#, newer, since ’07, in use with .NET: not on final exam

- Stored procedures, PL/SQL for Oracle: covered on final exam
Connections in JDBC

- Interaction with data source through sessions
  - A connection identifies a logical session
  - JDBC URL: `jdbc:<protocol>;<otherParameters>`

Example:
```
String url="jdbc:oracle:www.bookstore.com:3083";
Connection conn;
try{
    conn = DriverManager.getConnection(url,
        "user", "password");
} catch (SQLException e) {...}
```

For our programs, running on cs.umb.edu hosts:
- url="jdbc:oracle:thin:@dbs3.cs.umb.edu:1521:dbs3" or
- url="jdbc:mysql://topcat.cs.umb.edu:3306/joedb"
Retrieving Data: Statements, ResultSets

- `Statement.executeQuery` returns data
  - encapsulated in a ResultSet object (a cursor)
- `PreparedStatement` can also be used for this purpose
- Retrieval by attribute name or position

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT sname FROM Sailors WHERE rating = 10");
// rs lets us access the sequence of rows returned
while (rs.next()) {// process the data
    String name = rs.getString("sname"); // or
    // rs.getString(1);
}
```

Note: Similar code in Figure 6.4 on pg. 202 needs fixing
ResultSet

- ResultSet can be a very powerful cursor:
  - `next()`, `previous()`, `first()`, `last()`
  - `absolute(int num)`: moves to the row with the specified number
  - `relative (int num)`: moves forward or backward

- However, from the JDK docs:
  A default ResultSet object is not updatable and has a cursor that moves forward only.
  - So normally the only movement methods are `next()` and `last()`, and `last()` is rarely used.
  - When a result set is created, the cursor is positioned before the first row.
  - To move the cursor to the next row, call the `next` method. If the row is valid, this method moves the cursor to the next row and returns a true value.
The getXXX methods

- The getXXX methods can be used to return all eight primitive types. For example, the getInt method returns the int type and the getLong method returns the long type.

- The getXXX methods can also be used to return strings, dates, and times. For example, the getString method returns any object of the String class, and the getDate, getTime, and getTimestamp methods return objects of the Date, Time, and Timestamp classes of the java.sql package.
<table>
<thead>
<tr>
<th>SQL Type</th>
<th>Java class</th>
<th>ResultSet get method</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT</td>
<td>Boolean</td>
<td>getBoolean()</td>
</tr>
<tr>
<td>CHAR</td>
<td>String</td>
<td>getString()</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>String</td>
<td>getString()</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>Double</td>
<td>getDouble()</td>
</tr>
<tr>
<td>FLOAT</td>
<td>Double</td>
<td>getDouble()</td>
</tr>
<tr>
<td>INTEGER</td>
<td>Integer</td>
<td>getInt()</td>
</tr>
<tr>
<td>REAL</td>
<td>Double</td>
<td>getFloat()</td>
</tr>
<tr>
<td>DATE</td>
<td>java.sql.Date</td>
<td>getDate()</td>
</tr>
<tr>
<td>TIME</td>
<td>java.sql.Time</td>
<td>getTime()</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>java.sql.Timestamp</td>
<td>getTimestamp()</td>
</tr>
</tbody>
</table>

Fig. 6.5 Note: there is no standard “Double” SQL type. Should be DOUBLE PRECISION.
How to use the executeUpdate method to…

**Add a row**

String query =
   "INSERT INTO Product (ProductCode, ProductDescription, ProductPrice) " +
   "VALUES ('" + product.getCode() + ", " +
   "'" + product.getDescription() + ", " +
   "'" + product.getPrice() + "');"
Statement statement = connection.createStatement();
int rowCount = statement.executeUpdate(query);

**Update a row**

String query = "UPDATE Product SET " +
   "ProductCode = '" + product.getCode() + ", " +
   "ProductDescription = '" + product.getDescription() + ", " +
   "ProductPrice = '" + product.getPrice() + "'" +
   "WHERE ProductCode = '" + product.getCode() + "');"
Statement statement = connection.createStatement();
int rowCount = statement.executeUpdate(query);
First compile JdbcCheckup.java. No special classpath needs to be set up for this:

```
javac JdbcCheckup.java
```

This command assumes that JdbcCheckup.java is in the current dir.

Now we have JdbcCheckup.class in the current dir.

Use java to run it as follows. We need to add the driver jar file to the classpath to give the program access to the implementation of the JDBC API.

```
java -classpath driver.jar; JdbcCheckup  (Windows)
```

(change ‘;’ to ‘:’ for UNIX/Linux)

Driver.jar: ojdbc6.jar or mysql-connector-java-5.1.39-bin.jar
In the program, we find out what database the user wants to connect to, and their username and password (for Oracle or mysql)

For Oracle:
- the server host is "dbs3.cs.umb.edu"
- port 1521
- sid = "dbs3"

These are used in the "connection string" or "database url":
connStr = "jdbc:oracle:thin:@dbs3.cs.umb.edu:1521/dbs3".

The code ends up with strings connStr, user, and password.

Then get a connection from the driver:

```java
Connection conn =
    Driver.Manager.getConnection(connStr, user, password)
```
JDBCCheckup.java, continued

Create a statement using the Connection object:

```java
Statement stmt = conn.createStatement();
```

Do DB actions using Statement –

```java
stmt.execute("drop table welcome") ;
stmt.execute("create table welcome(msg char(20))");
stmt.execute("insert into welcome values ('Hello World!')");
ResultSet rest = stmt.executeQuery("select * from welcome") ;
```

Close connection (and its associated objects )

```java
conn.close() ← important to free up TCP/IP connection into the DB
```
static void tryWelcomeExperiment(Connection conn) throws SQLException {
    // Create a statement
    Statement stmt = conn.createStatement();
    ResultSet rset = null;
    try {
        <various DB actions>
    } finally { // Note: try without catch
        stmt.close(); // clean up statement resources
    }
}

- This method throws any exception occurring during the DB actions
- That’s good: in Java, a method is expected to report problems via an exception
How to use a prepared statement

To return a result set

String preparedSQL = "SELECT ProductCode, ProductDescription, " + "       ProductPrice " + "FROM Product WHERE ProductCode = ?";
PreparedStatement ps = connection.prepareStatement(preparedSQL);
ps.setString(1, productCode);
ResultSet product = ps.executeQuery();

To modify a row

String preparedSQL = "UPDATE Product SET " + "   ProductCode = ?, " + "   ProductDescription = ?, " + "   ProductPrice = ?" + "WHERE ProductCode = ?";
PreparedStatement ps = connection.prepareStatement(preparedSQL);
ps.setString(1, product.getCode());
ps.setString(2, product.getDescription());
ps.setDouble(3, product.getPrice());
ps.setString(4, product.getCode());
ps.executeUpdate();
How to use a prepared statement (continued)

To insert a row

```java
String preparedStatement = 
    "INSERT INTO Product " 
    + "(ProductCode, ProductDescription, ProductPrice) " 
    + "VALUES " 
    + "(?, ?, ?)";
PreparedStatement ps = connection.prepareStatement(preparedStatement);
ps.setString(1, product.getCode());
ps.setString(2, product.getDescription());
ps.setDouble(3, product.getPrice());
ps.executeUpdate();
```

To delete a row

```java
String preparedStatement = "DELETE FROM Product " 
    + "WHERE ProductCode = ?";
PreparedStatement ps = connection.prepareStatement(preparedStatement);
ps.setString(1, productCode);
ps.executeUpdate();
```
How to work with prepared statements

- When you use *prepared statements* in your Java programs, the database server only has to check the syntax and prepare an execution plan once for each SQL statement. This improves the efficiency of the database operations and prevents most types of SQL injection attacks.

- To specify a parameter for a prepared statement, type a question mark (?) in the SQL statement.

- To supply values for the parameters in a prepared statement, use the set methods of the PreparedStatement interface.

- To execute a SELECT statement, use the executeQuery method. To execute an INSERT, UPDATE, or DELETE statement, use the executeUpdate method.
Transactions

The user/programmer can group a sequence of commands so that they are executed atomically and in a serializable fashion:

- **Transaction commit**: all the operations should be done and recorded.

- **Transaction abort**: none of the operations should be done.

Compare to file storage: abort leaves broken files commonly
Consider two transactions (in a really bad DB) where A = 100

<table>
<thead>
<tr>
<th>T1:</th>
<th>A = A + 100</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2:</td>
<td>A = A + 100</td>
</tr>
</tbody>
</table>

T1 & T2 are concurrent, running same transaction program
T1 & T2 both read old value, 100, add 100, store 200
One of the updates has been lost!

Consistency requirement: after execution, A should reflect all deposits (Money should not be created or destroyed)
No guarantee that T1 will execute before T2 or vice-versa…
… but the net effect must be equivalent to these two transactions running one-after-the-other in some order
Transaction Support in SQL

- A transaction is automatically started when user executes a statement or accesses the catalogs.
- Transaction is either committed (COMMIT) or aborted (ROLLBACK).
- Need to make sure to ROLLBACK on database error!
- We try to code so we can see the COMMIT and ROLLBACK in the same vicinity in the code.
Importance of Short Transactions

- Transactions hold locks in the database, preventing other transactions from finishing.
- The slowest activity involved in program execution is human response time, i.e. UI (user interaction) time.
- Thus there is a simple rule: no UI inside a transaction.
- Example:
  - Show user the product, its price (one transaction)
  - Accept order
  - Run a new transaction to record the sale, rollback if the object was bought meanwhile by someone else
Homework 4

- You should understand the solution to homework 4
- It uses a PreparedStatement to scan the flights table
- java.sql.Timestamp variables are used to hold Timestamps from the database
- The program FindRestaurants.java uses a complex query to find Yelp-listed restaurants near a particular point
  - JDBC can handle complex queries like this, no problem
  - But it's best to test the query outside the program first.
- In the EmailList.java program (provided), each email-list user had a row in a table, and the program used a User object to hold that information before the insert.
  - You wrote the complementary tool EditUser.java to find a particular user entry, expressed in a User object, and change the info.
  - The idea of rows<->objects is important in Java webapps.
PL/SQL: Why Stored Procedures?

- So far, all data processing is done at the client
  - Lots of data may have to be transferred
  - Functionality (code) replicated at each client
  - Lots of state (e.g., locks, transaction data) at the DBMS
    - While client processes the data
- Stored procedures execute in same process space as DBMS
  - Encapsulates application logic and is close to the data
  - Reuse of common functionality by different clients
- Vendors introduced their own procedural extensions
  - e.g., Oracle’s PL/SQL
PL/SQL Program Structure

DECLARE
    variable_declarations
BEGIN
    procedural_code
EXCEPTION
    error_handling
END;
Data Types

- It is possible to use ORACLE SQL types
  - NUMBER, VARCHAR, etc
- PL/SQL allows directly referring to a column type
  - `tablename.columnname%TYPE`
  - e.g., `SAILORS.SNAME%TYPE`
- Also possible to define a row type (e.g., tuple)
  - `tablename%ROWTYPE`
- Declaring a variable: `<var_name> <TYPE>;
  - `sailor_rec SAILORS%ROWTYPE;`
- Can later refer to individual fields using column names
  - `DBMS_OUTPUT.PUT_LINE('Name: ' || sailor_rec.name || 'Age:' || sailor_rec.age);`
  - `||` means string concatenation (like + in Java)
As before, there are two cases

1. Single-tuple result (the “easy” case)

   ```sql
   SELECT selectfields INTO declared_variables
   FROM table_list WHERE search_criteria;
   ```

   Example PL/SQL script:
   ```plsql
   DECLARE
     VAR_NAME Sailors.name%TYPE;
     VAR_AGE Sailors.age%TYPE;
   BEGIN
     SELECT name, age INTO VAR_NAME, VAR_AGE
     FROM Sailors WHERE SID = 10;
     DBMS_OUTPUT.PUT_LINE('Age of ' || VAR_NAME || ' is ' || VAR_AGE);
   END;
   ```
SQL Statements – retrieving data

2. Multiple-tuples result: **cursors** are needed
   
   ```sql
   CURSOR cursorname IS SELECT_statement;
   
   OPEN cursorname;
   FETCH cursorname INTO variable_list;
   CLOSE cursorname;
   ```
DECLARE
    S Sailors%ROWTYPE;
CURSOR SAILORCURSOR IS
    SELECT * FROM Sailors;
BEGIN
    OPEN SAILORCURSOR;
    LOOP
        FETCH SAILORCURSOR INTO S;
        EXIT WHEN SAILORCURSOR %NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(‘AGE OF ’ || S.sname || ’ IS ’ || S.age);
    END LOOP;
    CLOSE SAILORCURSOR ;
END;
Declaring a Procedure

CREATE OR REPLACE PROCEDURE procedure_name (parameters) IS
    variable declarations
BEGIN
    procedure_body
END;

Parameters can be IN, OUT or INOUT, default is IN
CREATE OR REPLACE PROCEDURE SUM_AB (A INT, B INT, C OUT INT) IS
BEGIN
    C := A + B;
END;

Note no DECLARE needed here
Declaring a Procedure

CREATE OR REPLACE
PROCEDURE SUM_AB (A INT, B INT, C OUT INT) IS
BEGIN
    C := A + B;
END;

Calling this procedure from a PL/SQL script:
SET SERVEROUTPUT ON;
DECLARE
    SUM2 INT;  -- Don’t use a var named SUM!!
BEGIN
    SUM_AB(10, 20, SUM2);  -- fill in SUM2
    DBMS_OUTPUT.PUT_LINE('SUM2 = ' || SUM2);
END;

It is also possible to call it from Java (used in hw5) See article
A script that contains a transaction

BEGIN
  INSERT INTO invoices
  VALUES (115, 34, 'ZXA-080', '30-AUG-06',
    14092.59, 0, 0, 3, '30-SEP-06', NULL);

  INSERT INTO invoice_line_items
  VALUES (115, 1, 160, 4447.23, 'HW upgrade');

  INSERT INTO invoice_line_items
  VALUES (115, 2, 167, 9645.36, 'OS upgrade');

  COMMIT;
  DBMS_OUTPUT.PUT_LINE('The transaction was committed.');
EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK;
    DBMS_OUTPUT.PUT_LINE('The transaction was rolled back.');
END;
/

Note: If you have COMMIT, you need EXCEPTION handling and ROLLBACK.
A stored procedure that updates a table

```
CREATE OR REPLACE PROCEDURE update_invoices_credit_total
  (invoice_number_param VARCHAR2,
   credit_total_param NUMBER
  )
AS
BEGIN
  UPDATE invoices
  SET credit_total = credit_total_param
  WHERE invoice_number = invoice_number_param;

  COMMIT;
EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK;
END;
/
```

We are not required to code COMMIT and ROLLBACK in the PL/SQL stored procedure. They can be done in the caller
The syntax for declaring parameters

```
parameter_name_1 [IN|OUT|IN OUT] data_type
```

A stored procedure that uses parameters

```
CREATE OR REPLACE PROCEDURE update_invoices_credit_total
(
  invoice_number_param IN  VARCHAR2,
  credit_total_param   IN  NUMBER,
  update_count         OUT INTEGER
)
AS
BEGIN
  UPDATE invoices
  SET credit_total = credit_total_param
  WHERE invoice_number = invoice_number_param;
```

Naming parameters “something_param” is a very good idea, to avoid confusing them with same-named columns in SQL (no helpful warnings on SQL with “where invoice_number = invoice_number”, just execution and use of value true on all rows!)
The syntax for declaring parameters

\[
\text{parameter_name}_1 \ [\text{IN}|\text{OUT}|\text{IN OUT}] \ \text{data_type}
\]

A stored procedure that uses parameters

\[
\text{CREATE OR REPLACE PROCEDURE update_invoices_credit_total (}
\qquad \text{invoice_number_param} \ \text{IN} \ \text{VARCHAR2,}
\qquad \text{credit_total_param} \ \text{IN} \ \text{NUMBER,}
\qquad \text{update_count} \ \text{OUT} \ \text{INTEGER}
\qquad )}
\]

\[
\text{AS}
\]

\[
\begin{align*}
\text{BEGIN} \\
\quad \text{UPDATE invoices} \\
\quad \text{SET credit_total} = \text{credit_total_param} \\
\quad \text{WHERE invoice_number} = \text{invoice_number_param};
\end{align*}
\]

Naming parameters “something_param” is a very good idea, to avoid confusing them with same-named columns in SQL (no helpful warnings on SQL with “where invoice_number = invoice_number”, just execution and use of value true on all rows!)
A script that calls the stored procedure

```
SET SERVEROUTPUT ON;
DECLARE
  row_count INTEGER;
BEGIN
  update_invoices_credit_total('367447', 200, row_count);
  DBMS_OUTPUT.PUT_LINE('row_count: ' || row_count);
END;
/
```

Again we could move the commit/rollback and exception handling up to the caller.
Homework 5: more JDBC, plus PL/SQL

- Another JDBC program to move a row from table apartments to table rented_apartments.
  - Saw how to copy a table using SQL: create table T1 as (select * from T);
  - Copy a table, no rows: create table T1 as (select * from T where 1=0);
  - Copy a table, changing column names: use select a a1, b b1, … instead of *
- Using PL/SQL programming
  - We encapsulated the ridiculously complicated Oracle package function call needed for distance between (long1, lat1) and (long2, lat2) into an easy-to-use PL/SQL function geo_distance.
  - We can use geo_distance from SQL, PL/SQL, and Java.
  - FindRestaurants1.java uses geo_distance to simplify its complex query.
- FindRestaurants2.java uses a more complicated PL/SQL function that itself uses geo_distance, so we have Java calling PL/SQL, itself calling PL/SQL
- For a PL/SQL stored procedure, we wrote sp_delete_listing to move one row from table apartments to rented_apartments.
- Finally, cs630 students arranged to call sp_delete_listing from Java.
Schema Refinement and Normal Forms
Functional Dependencies (FDs)

- A functional dependency $X \rightarrow Y$ holds over relation $R$ if for every instance $r$ of $R$
  - $t_1, t_2 \in r$, $\pi_X (t_1) = \pi_X (t_2)$ implies $\pi_Y (t_1) = \pi_Y (t_2)$
  - given two tuples in $r$, if the $X$ values agree, $Y$ values must also agree
  - In $R \rightarrow W$ example, all emps with rating 8 have wage 10, etc.

- FD is a statement about all allowable relations.
  - Identified based on semantics of application (business logic)
  - Given an instance $r$ of $R$, we can check if it violates some FD $f$, but we cannot tell if $f$ holds over $R$!
Checking Functional Dependencies (FDs)

Check FD $AB \rightarrow C$ on this instance:

- Look for rows that are same on $A$ and $B$:
  - First two, and for these, $C$ values are the same
  - No more pairs of rows are the same on $A$ and $B,$

So $AB \rightarrow C$ holds for this instance

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$C$</th>
<th>$D$</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>b1</td>
<td>c1</td>
<td>d1</td>
</tr>
<tr>
<td>a1</td>
<td>b1</td>
<td>c1</td>
<td>d2</td>
</tr>
<tr>
<td>a1</td>
<td>b2</td>
<td>c2</td>
<td>d1</td>
</tr>
<tr>
<td>a2</td>
<td>b1</td>
<td>c3</td>
<td>d2</td>
</tr>
</tbody>
</table>

Figure 19.3 An Instance that Satisfies $AB \rightarrow C$
Sample Relation

Hourly_Emps (ssn, name, lot, rating, wage, hrs_worked)

- Denote relation schema by attribute initial: SNLRWH

- Constraints (functional dependencies, or FDs for short)
  - ssn is the key: \( S \rightarrow SNLRWH \)
  - rating determines wage: \( R \rightarrow W \)
    - E.g., worker with rating 10 receives 20$/hr
Anomalies

- Problems due to $R \rightarrow W$ being in Hourly_Emps
  - **Update anomaly**: Change value of $W$ only in a tuple, end up with a dependency violation
  - **Insertion anomaly**: How to insert employee if we don’t know hourly wage for that rating?
  - **Deletion anomaly**: If we delete all employees with rating 5, we lose the information about the wage for rating 5!

<table>
<thead>
<tr>
<th>S</th>
<th>N</th>
<th>L</th>
<th>R</th>
<th>W</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>123-22-3666</td>
<td>Attishoo</td>
<td>48</td>
<td>8</td>
<td>10</td>
<td>40</td>
</tr>
<tr>
<td>231-31-5368</td>
<td>Smiley</td>
<td>22</td>
<td>8</td>
<td>10</td>
<td>30</td>
</tr>
<tr>
<td>131-24-3650</td>
<td>Smethurst</td>
<td>35</td>
<td>5</td>
<td>7</td>
<td>30</td>
</tr>
<tr>
<td>434-26-3751</td>
<td>Guldu</td>
<td>35</td>
<td>5</td>
<td>7</td>
<td>32</td>
</tr>
<tr>
<td>612-67-4134</td>
<td>Madayan</td>
<td>35</td>
<td>8</td>
<td>10</td>
<td>40</td>
</tr>
</tbody>
</table>

- Note redundancy: multiple $R=8, W=10$ rows, also $R=5, W=7$ rows
Removing Anomalies

### Hourly_Emps2

<table>
<thead>
<tr>
<th>S</th>
<th>N</th>
<th>L</th>
<th>R</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>123-22-3666</td>
<td>Attishoo</td>
<td>48</td>
<td>8</td>
<td>40</td>
</tr>
<tr>
<td>231-31-5368</td>
<td>Smiley</td>
<td>22</td>
<td>8</td>
<td>30</td>
</tr>
<tr>
<td>131-24-3650</td>
<td>Smethurst</td>
<td>35</td>
<td>5</td>
<td>30</td>
</tr>
<tr>
<td>434-26-3751</td>
<td>Guldu</td>
<td>35</td>
<td>5</td>
<td>32</td>
</tr>
<tr>
<td>612-67-4134</td>
<td>Madayan</td>
<td>35</td>
<td>8</td>
<td>40</td>
</tr>
</tbody>
</table>

### Wages

<table>
<thead>
<tr>
<th>R</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

Create 2 smaller tables!

- Updating rating of employee will result in the wage “changing” accordingly
  - Note that there is no physical change of W, just a “pointer change”
- Deleting employee does not affect rating-wages data
- This process is called **decomposition**.
Reasoning About FDs

- Given FD set $F$, we can usually infer additional FDs:
  - $F^+ = \text{closure of } F$ is the set of all FDs that are implied by $F$

- Armstrong’s Axioms ($X, Y, Z$ are sets of attributes):
  - **Reflexivity**: If $Y \subseteq_X X$, then $X \rightarrow Y$
  - **Augmentation**: If $X \rightarrow Y$, then $XZ \rightarrow YZ$ for any $Z$
  - **Transitivity**: If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$

- Theorems: Union and decomposition (splitting)
  - $X \rightarrow Y$ and $X \rightarrow Z \implies X \rightarrow YZ$
  - $X \rightarrow YZ \implies X \rightarrow Y$ and $X \rightarrow Z$

- You need to know these, but not how to do multiline proofs using them. They are used in attribute closure...
Attribute Closure

- **Attribute closure** of $X$ (denoted $X^+$) wrt FD set $F$:
  - Set of all attributes $A$ such that $X \rightarrow A$ is in $F^+$
  - Set of all attributes that can be determined starting from attributes in $X$ and using FDs in $F$

- **Algorithm**:
  
  $X^+ = X$
  
  Repeat
  
  $Y = X^+$ (remember starting $X^+$ value for this pass)
  
  Search all FDs in $F$ with LHS completely included in $X^+$
  
  Add RHS of each such FD to $X^+$
  
  Until $Y = X^+$ (same as at start of this pass)
Attribute Closure

\[ X^+ = X \]
Repeat
\[ Y = X^+ \quad \text{(remember starting } X^+ \text{ value)} \]
Find FDs in F with LHSs completely included in \( X^+ \)
Add RHS of each such FD to \( X^+ \)
Until \( Y = X^+ \) (same as at start of this pass)

- \( C \) is the key: \( C \rightarrow \text{CSJDPQV} \) (i.e., \( C \rightarrow R \))
- Project purchases each part using single contract: \( JPD \rightarrow C \)
- Dept purchases at most one part from a supplier: \( SD \rightarrow P \)
  
  Example: Compute \( JPD^+ \)
  \[ JPD^+ = JPD, \text{ look for FDs with LHS in } JPD, \text{ find } JPD \rightarrow C \]
  \[ JPD^+ = JPD, \text{ look again for FDs with LHS in } JPD, \text{ find } C \rightarrow R \]
  \[ JPD^+ = \text{CSJDPQV}, \text{ all attributes, so done} \]

  with abbreviated notation, after numbering the FDs:
  \[ JPD^+ = JPD, \text{ = JPDC by (2), = JPDCR by (1), = R} \]
Verifying if attribute set is a key

- Key verification can also be done with attribute closure

- To verify if $X$ is a key, two conditions needed:
  - $X^+ = R$
  - $X$ is minimal

- How to test minimality
  - Removing an attribute from $X$ results in $X'$: check if $X'$ is a key
Lossless Join Decompositions

- A decomposition of R replaces it by two or more relations
  - Each new relation schema contains a subset of the attributes of R
  - Every attribute of R is in some relation, in some cases in two.
  - E.g., SNLRWH decomposed into SNLRH and RW

- Decomposition of R into X and Y is lossless-join if:
  - \( \pi_X(r) \bowtie \pi_Y(r) = r \)
  - If not lossless, then join has extra rows

- It is essential that all decompositions used to deal with redundancy be lossless!
Lossless Join Decompositions

- Decomposition of R into X and Y is **lossless-join** if:
  \[ \pi_X (r) \bowtie \pi_Y (r) = r \]

- It is always true that \[ r \subseteq \pi_X (r) \bowtie \pi_Y (r) \]
  
  - In general, the other direction does not hold!
    - i.e., the join has “extra” rows, not in r
  
  - If the other direction holds, the decomposition is lossless-join.

- **It is essential that all decompositions used to deal with redundancy be lossless!**
Condition for Lossless-join

- The decomposition of R into X and Y is lossless-join wrt F if and only if the closure of F contains:
  - $X \cap Y \rightarrow X$, or
  - $X \cap Y \rightarrow Y$

- Note that $X \cap Y$ constitutes the join columns, so the join columns need to be a superkey of X or Y.

- In particular, the decomposition of R into UV and R - V is lossless-join if $U \rightarrow V$ holds over R.
  - Example: SNLRWH, R -> W decomposed into SNLRH and RW
Dependency Preserving Decomposition

- Consider CSJDPQV, C is key, JP → C and SD → P.
  - Consider decomposition: CSJDQV and SDP
  - Problem: Checking JP → C requires a join!
  - We want J, P, and C all in one table of the decomposition
  - Then we'll be able to say we've "preserved" this dependency

- **Dependency preserving decomposition (Intuitive):**
  - If R is decomposed into X and Y, and we enforce the FDs that hold on X, Y then all FDs that were given to hold on R must also hold

- We get a dependency preserving decomposition if all the FDs lie in different tables of the decomposition, and in some additional cases that require a lot of machinery (math-like) that we don't have time to study.
  - E.g., SNLRWH decomposed into SNLRH and RW: S→NLRH in first, R→W in second, S→W by transitivity
Boyce-Codd Normal Form (BCNF)

- Relation R with FDs F is in BCNF if, for all $X \rightarrow A$ in $F^+$
  - $A \subseteq X$ (called a trivial FD), or
  - $X$ is a superkey of R.
- The only non-trivial FDs allowed are key constraints
- Every non-key column must depend on the whole key (2NF)
  - Because if non-key B depended on part of the key, say $X'$, we would have FD $X' \rightarrow B$, where $B$ is not in $X'$ and $X'$ is not a superkey, so the table is not BCNF.
- No redundancy (provable by FDs), no update, insertion, or deletion anomalies are possible
- Note that BCNF is the same as 3NF unless there are multiple keys and at least one multicolumn key, so very close to our goal.
Consider relation R with FDs F. If X → Y violates BCNF*, decompose R into R - Y and XY.

Repeated application of this idea will give us a collection of relations that are in BCNF; lossless join decomposition, and guaranteed to terminate.

e.g., CSJDPQV, key C, JP → C, SD → P, J → S

To deal with SD → P, decompose into SDP, CSJDPQV.

To deal with J → S, decompose CSJDPQV into JS and CJDQV

*or 3NF if doing decomposition to 3NF, defined soon.
In general, there may not be a dependency preserving decomposition into BCNF.

- e.g., ABC, AB → C, C → A
- Can’t decompose while preserving first FD; not in BCNF because LHS of C → A is not a superkey.
- We will see that ABC is in 3NF.
Third Normal Form (3NF)

- Relation R with FDs F is in 3NF if, for all $X \rightarrow A$ in $F^+$
  - $A \in X$ (called a trivial FD), or
  - $X$ contains a key for R, or
  - $A$ is part of some key for R ($A$ here is a single attribute)
- Minimality of a key is crucial in third condition above!
- If R is in BCNF, it is also in 3NF.
- If R is in 3NF, some redundancy is possible
  - compromise used when BCNF not achievable
  - Lossless-join, dependency-preserving decomposition of R into a collection of 3NF relations always possible.
- 2NF: replace "X contains a key" with "X contains a key or is determined by a key" to allow transitive dependencies
Summary of Schema Refinement

- **BCNF**: relation is free of FD redundancies
  - Having only BCNF relations is desirable
  - If relation is not in BCNF, it can be decomposed to BCNF
    - Lossless join property guaranteed
    - But some FD may be lost

- **3NF** is a relaxation of BCNF
  - Guarantees both lossless join and FD preservation, but allows a small amount of redundancy (only if multiple keys and a multicolumn key)

- Decompositions may lead to performance loss
  - *performance requirements* must be considered when using decomposition
Finding FDs and keys in given tables

Given a set of FDs for a table, determine keys, then BCNF status, 3NF status, also 2NF status

- BCNF: see if LHS of each FD is a superkey of the table
- Note than any BCNF table is also 2NF and 3NF
- Any 3NF table is also 2NF.

BCNF Decomposition: driven by a non-key FD (FD with LHS not a superkey)

- 3NF Decomposition: driven by non-key FD that doesn't determine part of a key (these FDs are allowed by 3NF)
Security and Authorization

CS430/630
Lectures 21, 21a

Slides based on “Database Management Systems” 3rd ed, Ramakrishnan and Gehrke
GRANT Command: Entry SQL92

GRANT privilege_list ON object TO user_list [WITH GRANT OPTION]

- The following privileges can be specified:
  - SELECT
    - can read all columns
    - including those added later via ALTER TABLE command
  - INSERT(col-name)
    - can insert tuples with non-null or non-default values in this column
    - INSERT means same right with respect to all columns
  - DELETE
    - can delete tuples
  - REFERENCES (col-name)
    - can define foreign keys (in other tables) that refer to this column
Examples

GRANT INSERT, SELECT ON Sailors TO Horatio
- Horatio can query Sailors or insert tuples into it.
- If the granting user has schema "sally", Horatio can "select * from sally.sailors;"

GRANT DELETE ON Sailors TO Yuppy WITH GRANT OPTION
- Yuppy can delete tuples, and also authorize others to do so
- Yuppy can "grant delete on sally.sailors to horatio"

GRANT UPDATE (rating) ON Sailors TO Dustin
- Dustin can update (only) the rating field of Sailors tuples

GRANT select on sally.Sailors to PUBLIC (executed by sally or a DBA)
- Like yelp_db tables, any user can read them.
Grant on views: Example

- User joe, with table Sailors in his schema, does:
  ```sql
  create view bestsailors (name, age, rating) as
  select s.sname, s.age, s.rating from sailors where rating = 10;
  grant select on bestsailors to sue;
  ```

- Now user sue can access bestsailors but not sailors:
  ```sql
  select * from joe.sailors;
  Error: Either no such table or no privileges on table
  select * from joe.bestsailors;
  <data displayed>
  ```

Note: this could be Oracle, with schemas based on username, or mysql, with schema/database joe, also user joe.
How to use a view (empinfo) to provide only a subset of the table's data (table employees) to a certain user (username clerk)

In schema hr: table employees, view empinfo

Employees(eid, ename, age, salary) as in createdb.sql

View empinfo(eid, ename)

Answer: User hr can execute:

create view empinfo as select eid, ename from employees;
grant select on empinfo to clerk;

After this, user clerk can access empinfo, like this:

Select * from hr.empinfo; -- see eid, ename