

---

# Introduction to SQL and JDBC

---



7400 E. Orchard Road, Suite 1450 N, Greenwood Village, CO 80111  
303-302-5234 | 800-292-3716  
[SkillDistillery.com](http://SkillDistillery.com)

# INTRODUCTION TO SQL AND JDBC

Student Workbook

### ***INTRODUCTION SQL AND JDBC***

**Author:** Rob Roselius

Published by ITCourseware, LLC, 7400 E. Orchard Road, Suite 1450, Greenwood Village, CO 80111

**Editor:** Jan Waleri

**Editorial Assistant:** Ginny Jaranowski

**Special thanks to:** Many instructors whose ideas and careful review have contributed to the quality of this workbook, including Elizabeth Boss, Denise Geller, Jennifer James, Julie Johnson, Roger Jones, John McCallister, Joe McGlynn, Jim McNally, Kevin Smith, Danielle Waleri and the many students who have offered comments, suggestions, criticisms, and insights.

Copyright © 2016 by ITCourseware, LLC. All rights reserved. No part of this book may be reproduced or utilized in any form or by any means, electronic or mechanical, including photo-copying, recording, or by an information storage retrieval system, without permission in writing from the publisher. Inquiries should be addressed to ITCourseware, LLC, 7400 E. Orchard Road, Suite 1450, Greenwood Village, Colorado, 80111. (303) 302-5280.

All brand names, product names, trademarks, and registered trademarks are the property of their respective owners.

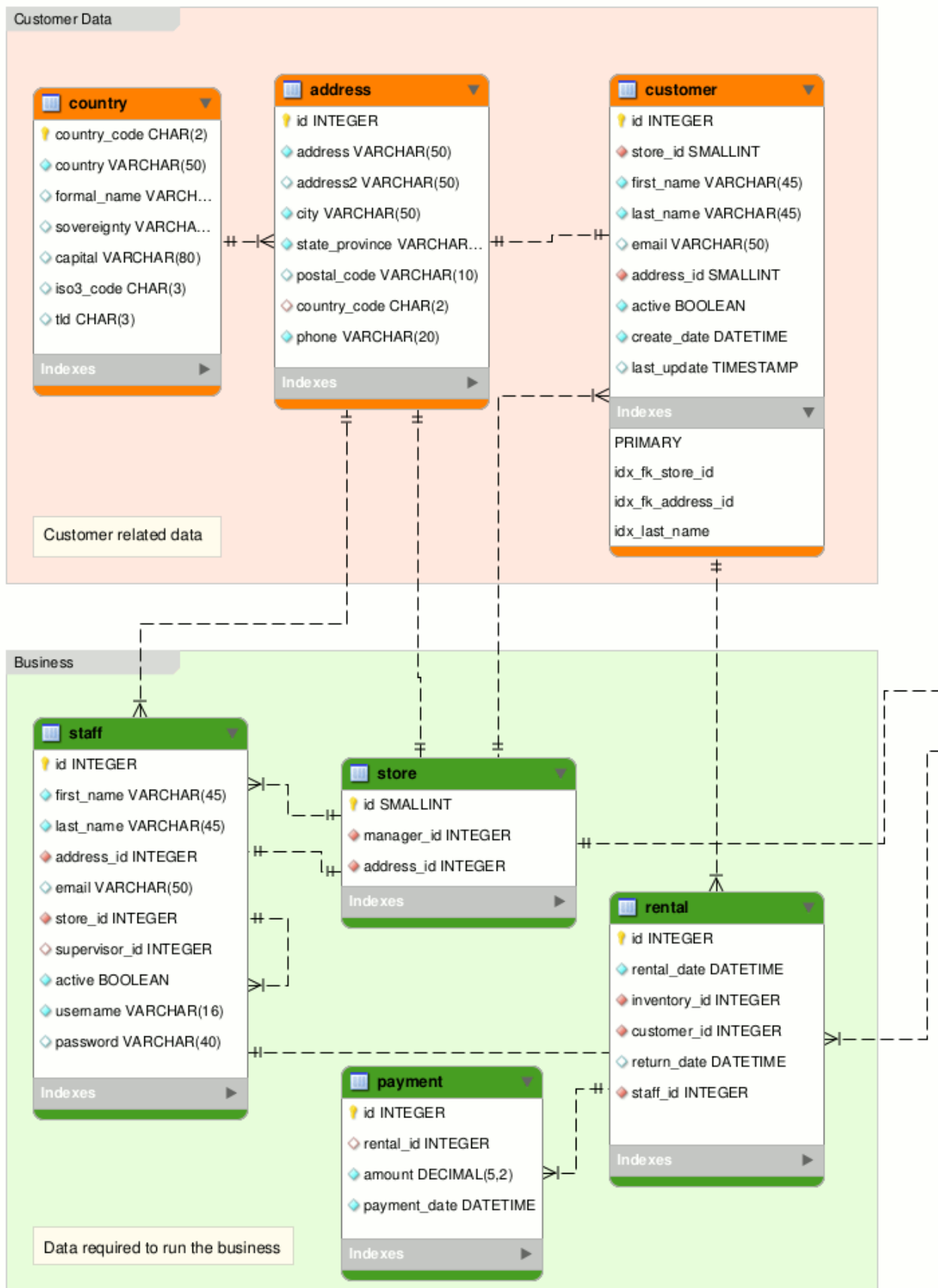
# CONTENTS

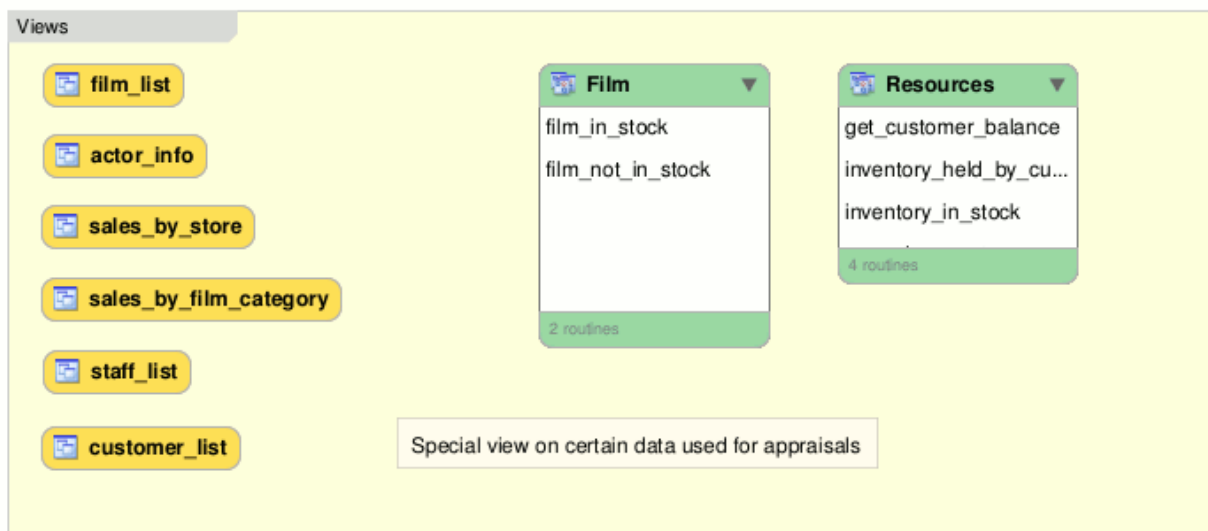
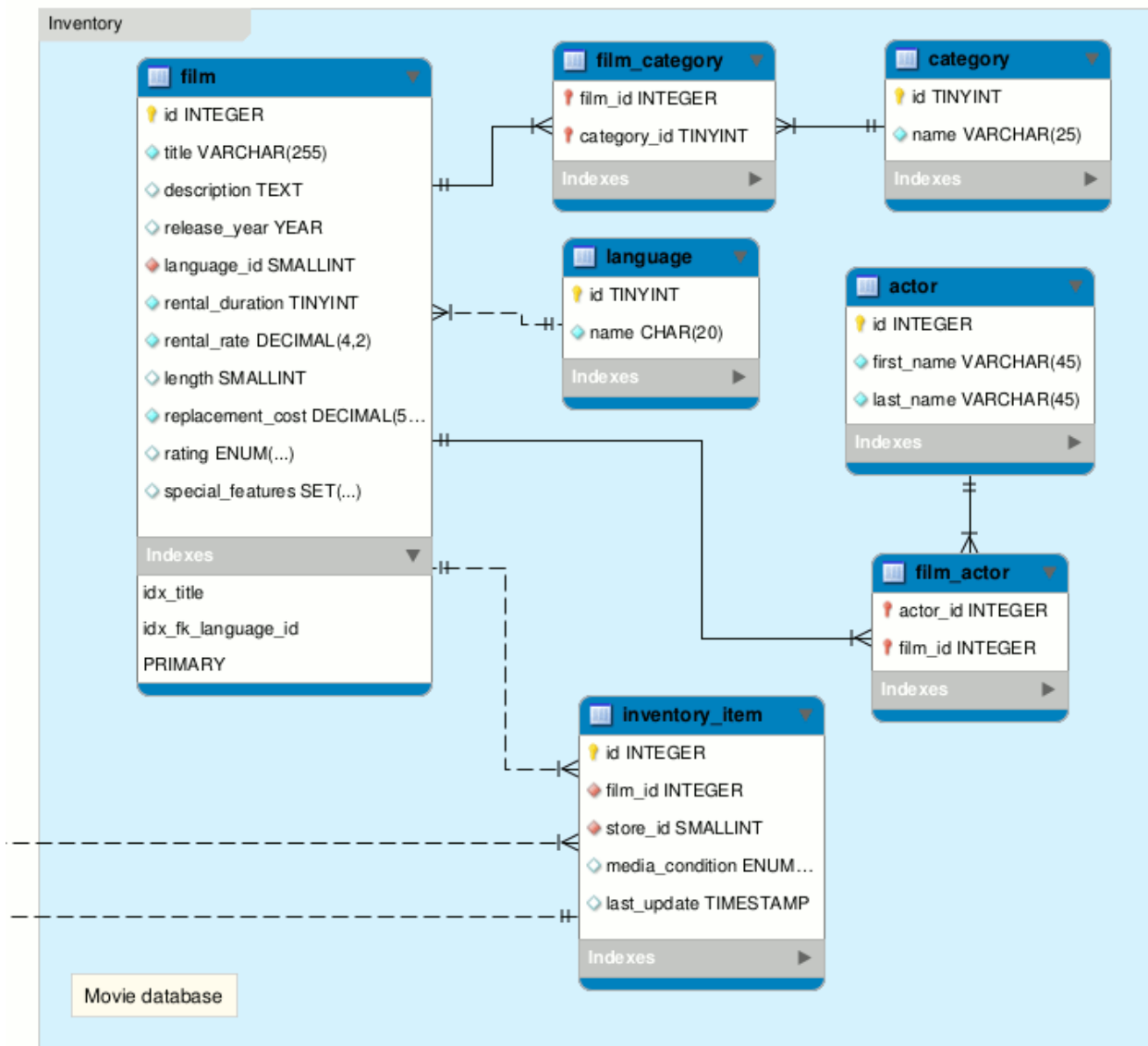
SQL Queries — The SELECT Statement .....	9
The SELECT Statement .....	10
Choosing Rows with the WHERE Clause .....	12
NULL Values .....	14
Compound Expressions .....	16
Creating Some Order .....	18
Labs .....	20
SQL Queries — Predicates and Functions .....	23
IN and BETWEEN .....	24
Pattern Matching: LIKE and REGEXP .....	26
SQL Functions .....	28
Labs .....	30
Introduction to JDBC .....	33
The JDBC Connectivity Model .....	34
Database Programming .....	36
Connecting to the Database .....	38
Creating a SQL Query .....	40
Getting the Results .....	42
JDBC Types .....	44
Finishing Up .....	46
Labs .....	48
Data Manipulation and Transactions .....	51
The INSERT Statement .....	52
The UPDATE Statement .....	54
The DELETE Statement .....	56
Transaction Management .....	58
Concurrency .....	60
Loading Tables From External Sources .....	62
Labs .....	64

JDBC SQL Programming .....	67
Error Checking and the SQLException Class .....	68
Executing SQL Updates .....	70
The execute() Method .....	72
ResultSetMetaData .....	74
Using a PreparedStatement .....	76
Parameterized Statements .....	78
Transaction Management .....	80
Labs .....	82
 SQL Queries — Joins .....	 85
Selecting from Multiple Tables .....	86
Join Conditions .....	88
Associative Tables .....	90
Labs .....	92
 SQL Queries — Outer and Self Joins .....	 95
Outer Joins .....	96
Self Joins .....	98
Equijoins, Non-equijoins, and Antijoins .....	100
Labs .....	102
 Aggregate Functions and Advanced Techniques .....	 105
Subqueries .....	106
Correlated Subqueries .....	108
The EXISTS Operator .....	110
The Aggregate Functions .....	112
Nulls and DISTINCT .....	114
Grouping Rows .....	116
Combining SELECT Statements .....	118
Labs .....	120
 Index .....	 123



## INTRODUCTION TO SQL AND JDBC









## SQL QUERIES – THE SELECT STATEMENT

### OBJECTIVES

- ✧ Retrieve data from a table using the SQL **SELECT** statement.
- ✧ Filter the returned rows with a **WHERE** clause.
- ✧ Locate fields which have not yet been populated.
- ✧ Combine multiple search criteria in a **WHERE** clause using logical operators.
- ✧ Sort the output of a **SELECT** statement.

# THE SELECT STATEMENT

- \* SQL data retrievals are done using the **SELECT** statement.
  - Data retrievals are also called *queries*.
- \* A **SELECT** statement may make use of several *clauses*, but minimally must include the following two:

```
SELECT [DISTINCT] {item_list | *}  
FROM table;
```

- \* The **SELECT** clause specifies which data values will be retrieved, and will be followed by either an *item\_list* or a \*.
  - \* represents all the columns in *table*.
  - *item\_list* is a column, an expression, or a comma-separated list of any combination of these.
  - The query will return each item in *item\_list* under a column heading of the same name, unless an alias is specified.
  - Use **DISTINCT** to eliminate duplicate rows from the displayed results.
- \* The **FROM** clause specifies one or more sources, typically tables, from which to retrieve data.
- \* A heading may be overridden with a *column alias*, a word used to replace the default heading.

```
SELECT title, rental_rate,  
       replacement_cost / rental_rate AS break_even  
FROM film;
```

The **SELECT** statement may query one or many tables. The data that is retrieved may be thought of as a *virtual table* (i.e., one that exists in memory only until the query completes). This virtual table is also referred to as the query's *result set*.

The **SELECT** statement can be used to retrieve data from any or all columns from a table. For example, the following **SELECT** will display a list of everything about every employee in the staff table:

```
SELECT *  
FROM staff;
```

The query below will list just the name and store of each employee:

```
SELECT first_name, last_name, store_id  
FROM staff;
```

The query below will list each employee's store:

```
SELECT store_id  
FROM staff;
```

This query will retrieve each store that has employees.

```
SELECT DISTINCT store_id  
FROM staff;
```

### Note:

Table, column, or alias names containing spaces must be enclosed in quotes.

```
SELECT title, rental_rate,  
       replacement_cost / rental_rate AS "Break Even"  
FROM film;
```

## CHOOSING ROWS WITH THE WHERE CLAUSE

- ✴ Use the **WHERE** clause when only a subset of all the rows in a table is required.

```
SELECT {item_list | *}  
FROM table  
[WHERE conditions];
```

- ✴ The **WHERE** clause is evaluated once for each row in *table*.
- ✴ *conditions* will be one or more expressions that will evaluate to either true, false, or neither (null).
  - If the **WHERE** clause evaluates to true, then the current row is returned in the result set.
- ✴ Operators that may be used in the **WHERE** clause include:

=	equal to
!=, ^=, <>	not equal to
>, <	greater than, less than
>=, <=	greater than or equal to, less than or equal to
- ✴ Any character string values in conditions must be enclosed within single quotes.
  - Database data inside single quotes is case sensitive.

Retrieve all information for each employee at store 7:

```
SELECT *  
  FROM staff  
 WHERE store_id = 7;
```

Retrieve first and last name, salary, and hire date for employees in Massachusetts:

```
SELECT id, title, length, rental_rate, replacement_cost  
  FROM film  
 WHERE release_year = 1990;
```

Despite well-established standards, database products vary in certain low-level details of syntax. In ANSI standard SQL, doublequotes enclose identifiers - table, column, or alias names - *only*, and string literals must be enclosed in single-quotes. Oracle Database enforces this, but MySQL treats single, double, and back quotes interchangeably (unless a standards-compliance mode is enabled.)

Similarly, standard SQL comments have two forms:

\* C-style block comments:

```
SELECT id  
      , title  
      , length  
      /* , rental_rate  
        , replacement_cost */  
  FROM film WHERE release_year = 1990;
```

\* To-end-of-line comments:

```
SELECT last_name, supervisor_id, -- no supervisor_id for store mgrs  
      store_id  
  FROM staff;
```

However, MySQL requires at least one space to follow the --, and also supports # as a to-end-of-line comment starter.

Best practice is always to code to the established standards.

## NULL VALUES

✳ A *null* is the absence of a value.

➤ Null values are not equal to **0**, blanks, or empty strings.

✳ Compare null values with the **IS** operator.

```
SELECT *  
  FROM staff  
 WHERE supervisor_id IS NULL;
```

➤ The **IS** operator will result in either true or false.

✳ To search for any non-null values, use **IS NOT NULL**.

```
SELECT *  
  FROM staff  
 WHERE supervisor_id IS NOT NULL;
```

✳ **NULL** is not the same as a "false" value, but it isn't a "true" value, either.

➤ **NULL** means "unknown;" any operation with a **NULL** (other than **IS** or **IS NOT**) yields a **NULL**.

```
SELECT *  
  FROM staff  
 WHERE supervisor_id = NULL;
```

➤ This will never be true, even for rows with null values in the **supervisor\_id** column!

Retrieve all employees with a known email address:

```
SELECT last_name, first_name, username, email
FROM staff
WHERE email IS NOT NULL;
```

Retrieve all customers with without email addresses:

```
SELECT last_name, first_name, email
FROM customer
WHERE email IS NULL
      OR email = '';
```

### VARCHAR, VARCHAR2, and NULL

The SQL standards specify that for character-varying data an empty string, ' ', is a distinct, non-null value and should not be treated as a NULL. For example, if a customer declines to provide an email address, we would store empty string ( ' ') in their email column; but if we simply don't yet know what their email is, or whether they have one at all, we would store NULL. We should then get different results for the following two queries:

```
SELECT * FROM customer WHERE email = ' ';
SELECT * FROM customer WHERE email IS NULL;
```

Oracle's character-varying datatype does not comply with this, so Oracle named it VARCHAR2 instead of VARCHAR. Oracle VARCHAR2 treats empty string and NULL identically. Oracle reserves the word VARCHAR for a future implementation that complies with the SQL standards. However, it has never implemented this, so you use only VARCHAR2 with Oracle.



### COMPOUND EXPRESSIONS

- ✴ Use logical operators to group conditions in a **WHERE** clause.
  - **NOT** will negate the condition following it.
  - **AND** will result in true if both conditions are true for a row.
  - **OR** will result in true if either condition is true for a row.
- ✴ The logical operators are evaluated based on precedence: **NOT** has the highest precedence, then **AND**, and then **OR**.
  - If a **WHERE** clause contains both an **AND** and an **OR**, the **AND** will always be evaluated first.

```
SELECT id, first_name, last_name
FROM staff
WHERE first_name = 'Dave'
      OR first_name = 'Diane'
      AND last_name = 'Martindale';
```

- This query will find **Diane Martindale**, as well as anyone with the first name of **Dave**.

- ✴ Use parentheses to override the precedence of these operators.

```
SELECT id, first_name, last_name
FROM staff
WHERE ( first_name = 'Dave'
      OR first_name = 'Diane' )
      AND last_name = 'Martindale';
```

- This query will find people with a **last\_name** of **Martindale** and a **first\_name** of either **Dave** or **Diane**.

We don't delete their staff record when an employee leaves, since we record rentals and payments they've processed. Instead we mark their staff record as inactive.

```
SELECT *  
  FROM staff  
 WHERE active = 1  
       AND ( supervisor_id = 118  
             OR supervisor_id = 114 );
```

### CREATING SOME ORDER

- ✴ Data is not physically stored in a table in any specified order.
  - Unless the table is an Index-Organized Table (IOT), records are usually appended to the end of the table.
- ✴ The **SELECT** statement's **ORDER BY** clause is the only way to enforce sequencing on the result set.
- ✴ The **ORDER BY** clause may be included in the **SELECT** statement:

```
SELECT {item_list | *}
      FROM table
[ORDER BY column [ASC | DESC], ...];
```

  - *column* can either be the name or the numeric position of the column in the *item\_list*.
- ✴ The result set is sorted by the first column name in the **ORDER BY** clause.
  - If there are duplicate values in the first sort column, the sort is repeated on the second column in the list, and so on.
  - The rows may be sorted in ascending (the default) or descending order.
  - **ASC** and **DESC** may be applied independently to each column.
  - **NULL** values will be sorted after all non-**NULL** values in **ASC** order (and before all non-**NULL**s when you use **DESC**).

Retrieve a list of employees sorted by store from highest to lowest, and by last name within each store from lowest to highest:

```
SELECT id, first_name, last_name, store_id
FROM staff
ORDER BY store_id DESC, last_name ASC;
```

Sort using select item number instead of column name:

```
SELECT id, first_name, last_name, store_id
FROM staff
ORDER BY 4 DESC, 3 ASC;
```

### LABS

Write queries to:

- ❶ Retrieve a list of all departments and their location ids.
- ❷ List the ids of languages for which there are films, listing each language id only once.
- ❸ List the years in which our films have been released, listing each year only once. Sort these in ascending order.
- ❹ List the employees of store 4 whose supervisor's id is 89.
- ❺ Show which employees, if any, have no email address.
- ❻ Use at least two different ways to list the country code, name, and sovereignty of just those countries that are sovereign territories of another country (that is, that have a non-empty **sovereignty** column).





## SQL QUERIES – PREDICATES AND FUNCTIONS

### OBJECTIVES

- ✧ Use the **IN** and **BETWEEN** operators to match a column against multiple values.
- ✧ Use wildcards to search for a pattern in character data.
- ✧ Use SQL functions to transform data used in a query.



# IN AND BETWEEN

- \* Use the **IN** (or **NOT IN**) operator to compare a column against several possible values.

```
SELECT title, release_year, rating
FROM film
WHERE rating IN ('G', 'PG');
```

- **IN** is equivalent to **OR**ing several equality comparisons together.

```
SELECT title, release_year, rating
FROM film
WHERE rating = 'G'
      OR rating = 'PG';
```

- \* Use the **BETWEEN** operator to compare a column against a range of inclusive values.

```
SELECT title, release_year, rating
FROM film
WHERE release_year BETWEEN 1983 AND 1986;
```

- The **AND** is part of the **BETWEEN** operator.
- Make sure that lower value appears before the higher one:

```
SELECT id, last_name, first_name
FROM staff
WHERE first_name BETWEEN 'A' AND 'E';
```

Retrieve a list of Colorado addresses not in the 80202, 80205, or 80206 zip codes:

```
SELECT *  
  FROM address  
 WHERE postal_code NOT IN ('80202', '80205', '80206')  
    AND state_province = 'Colorado';
```

List payments since 2016 Q1 between \$5 and \$8::

```
SELECT *  
  FROM payment  
 WHERE payment_date > '2016-04-01' AND amount BETWEEN 5 AND 8;
```

# PATTERN MATCHING: LIKE AND REGEXP

✴ The **LIKE** operator provides pattern matching for character data with two simple *wildcards*:

- **%** Matches zero or more characters.
- **\_** Matches exactly one character and is position-dependent.

```
SELECT id, title FROM film
WHERE title LIKE 'PER%';

SELECT id, title FROM film
WHERE title LIKE '%PER';

SELECT id, title FROM film
WHERE title LIKE '%PER%';

SELECT first_name, last_name FROM staff
WHERE last_name LIKE 'S_____';

SELECT last_name FROM staff
WHERE last_name LIKE '%son';
```

✴ Many database vendors support Regular Expression pattern matching, though the syntax varies.

- **MySQL:**

```
SELECT first_name, last_name FROM customer
WHERE last_name REGEXP '[APR].*[ae]rt?son';
```

- **Oracle Database:**

```
SELECT last_name FROM customer
WHERE REGEXP_LIKE(last_name, '[APR].*[ae]rt?son');
```

A common use of **LIKE** is to wrap our search text between two percent-sign wildcards to search the entire column, finding the search string at the beginning, end, or anywhere in between.

```
SELECT title, description
FROM film
WHERE description LIKE '%thrilling%';
```

### Collation and Case Sensitivity

Ultimately, all text is stored as character codes whose meanings are defined in character sets such as the Unicode code tables, which specify which numeric code means which specific character symbol from some written language.

In addition to these unique character code numbers, a character set uses a corresponding *collation*. The collation determines which characters are considered to be equivalent in a given language. For example in the ASCII character set (also known as Unicode Basic Latin), A (capital A) is character code 65 but a (lower case A) is character code 97 - they are two distinct character codes. However, in the English language they are considered the same letter with the same meaning and pronunciation. If we use a case-sensitive ASCII collation, they would be considered different in terms of sorting and comparison. In a case-**ins**sensitive collation of ASCII they would be considered the same in terms of sorting and comparison.

MySQL supports a large number of character sets, with multiple collation options for each (for example, letters may sort differently in Spanish than in Swedish though they share the same character set). You can specify the character set and collation for each text column or for an entire table; the database can have defaults for new tables that don't specify their own.

MySQL by default uses a case-**ins**sensitive collation so string comparisons with =, !=, and LIKE are case-insensitive.

To get a case-sensitive search, specify the collation in the WHERE clause predicate:

```
SELECT title, description
FROM film
WHERE description LIKE '%thrilling%' COLLATE utf8_bin; -- not found

SELECT title, description
FROM film
WHERE description LIKE '%Thrilling%' COLLATE utf8_bin; -- found
```

Alternatively, qualify an operand with **BINARY** to force a binary (exact character code) comparison:

```
SELECT title, description
FROM film
WHERE BINARY description LIKE '%thrilling%'; -- not found
```

# SQL FUNCTIONS

✴ Use functions built into the server to manipulate data within your statement:

➤ In the SELECT list:

- Fetch only a portion of the film description.

```
SELECT title, SUBSTR(description, 3, 30)
FROM film;
```

➤ In the WHERE clause:

- Find rentals made on Fridays in the month of July.

```
SELECT *
FROM rental
WHERE MONTH(rental_date) = 7
AND WEEKDAY(rental_date) = 4;
```

✴ This kind of function is sometimes called a *scalar* or *single-row* function.

- The function takes zero or more arguments and returns a single value
- The function is executed once for each row processed by the statement.
- The function doesn't modify the actual data stored in the database.

✴ Each database vendor has their own list of scalar functions.

- You must consult your specific database product documentation for names, parameters, behavior, and limitations of each function.

### SQL Functions

Every database product supports its own list of *scalar* or *single-row functions* - functions that can be used in SQL statements and which are evaluated for every row.

#### String Functions:

```
SELECT CONCAT(last_name, ' ', first_name) FROM staff;
SELECT last_name, LENGTH(last_name) FROM staff;
SELECT title, LOWER(title) FROM film;
SELECT SUBSTR(last_name, 1, 3) FROM staff;
SELECT title, INSTR(title, ' ') FROM film;
SELECT title, SUBSTR(title, 1, INSTR(title, ' ')) FROM film;
```

#### Numeric Functions:

```
SELECT COS(1);
SELECT 4 * ATAN(1);
SELECT SQRT(2);
SELECT FLOOR(RAND() * 52);
SELECT title, rental_rate, replacement_cost / rental_rate AS rawroi,
       ROUND(replacement_cost / rental_rate) AS roi FROM film;
```

#### Date and Time Functions:

```
SELECT CURDATE(), CURTIME(), NOW();
SELECT rental_date, return_date, DATEDIFF(return_date, rental_date)
       FROM rental;
SELECT amount, MONTH(payment_date) FROM payment;
SELECT DATE_ADD(CURDATE(), INTERVAL 30 DAY);
SELECT DATE_ADD(CURDATE(), INTERVAL 6 MONTH);
SELECT WEEKDAY(DATE_ADD(CURDATE(), INTERVAL 6 MONTH));
```

#### Comparison and Flow-Control Functions:

```
SELECT country_code, country, IFNULL(formal_name, 'N/A') FROM country;
SELECT country_code, COALESCE(formal_name, country) FROM country;
SELECT first_name, last_name, STRCMP(first_name, last_name) FROM staff;
```

#### System and Information Functions:

```
SELECT USER();
SELECT DATABASE();
SELECT VERSION();
```

These functions are not standardized, though most vendors support a similar rich set of functions in similar categories. You must refer to your particular database product's documentation for function names and usage.

### LABS

Write queries to:

- ❶ Retrieve the ids and names of employees from stores 2, 3, and 5.
- ❷ List all addresses with a country code of either GB or JA. Sort by state/province.
- ❸ List the title and replacement cost of films whose replacement cost is between \$10 and \$15.
- ❹ Show the names of employees whose lastnames are in the range of M-R.
- ❺ List the films whose titles end in "IDE".
- ❻ Show the title and description of each film, with the description displayed in all-uppercase.
- ❼ For all customers, show their name, the date they were created, and the date six months after the create date.
- ❽ Show each customer's name, as well as just the part of their email that's before the "@".







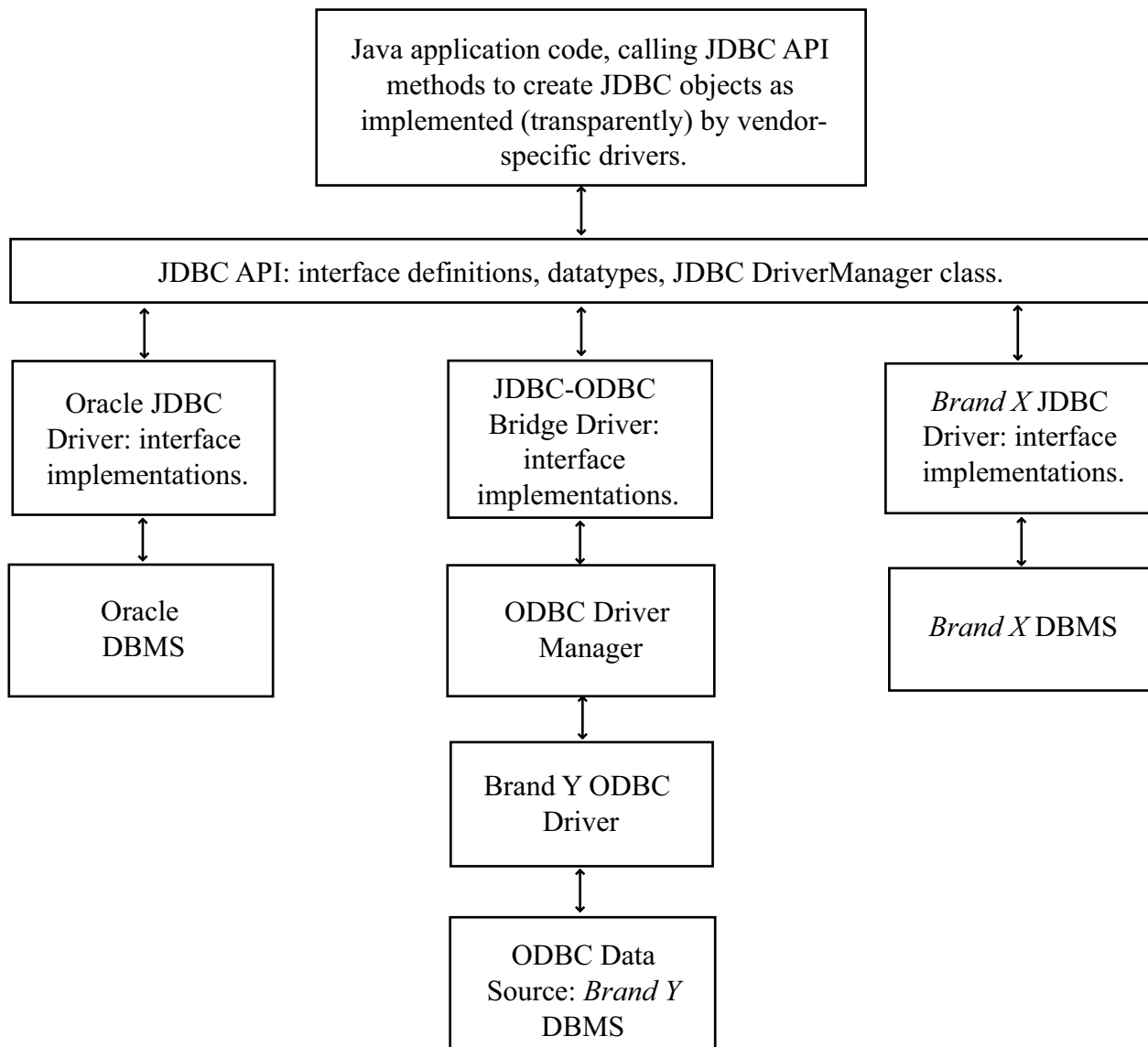
## INTRODUCTION TO JDBC

### OBJECTIVES

- \* Describe the Java Database Connectivity model.
- \* Write a simple Java program that uses JDBC.

### THE JDBC CONNECTIVITY MODEL

- ✳ JDBC provides a simple but complete programming interface for accessing Database Management Systems (DBMSs).
- ✳ Conceptually, JDBC resembles most other SQL database APIs.
  - Application programmers access a generic API, and a driver manager controls connections.
  - Vendor/product-specific drivers implement the connection and interaction with specific DBMSs.
- ✳ The **java.sql** and **javax.sql** packages define the JDBC interfaces and datatypes for dealing with database connections, statements, query results, etc.
  - Normally, you will work with just a few relatively simple interfaces and methods.
  - JDBC also provides features for tool builders and driver writers.
- ✳ Vendor-specific driver packages implement the **java.sql** interfaces.
  - The vendor must implement **java.sql.Driver** to provide the main entry point for JDBC.



You generally obtain a driver from the database vendor or from a third-party repository as a ZIP or JAR archive. The archive has compiled classes which implement the **java.sql** interfaces for the specific database product. You'll need to include this driver archive in your classpath and with your deployed application. You may also need to explicitly load the core **Driver** class in your application startup code.

### DATABASE PROGRAMMING

- ✱ Programmers writing applications that interact with a DBMS typically have four basic steps to complete:
  1. Create variables for database data.
  2. Connect to the DBMS.
  3. Do whatever it is you need to do with the database data.
  4. Disconnect from the DBMS.
  
- ✱ If what you need to do with the database data is to retrieve a set of records, the basic steps are:
  - 3.1. Declare a *cursor* — an object associated with a specific DBMS query statement.
  - 3.2. Open the cursor — this executes the query statement, and identifies its *result set*.
  - 3.3. Iterate through the records that make up the result set.
  - 3.4. Close the cursor.
  
- ✱ All of these standard database programming tasks are implemented simply in JDBC.

### Hands On:

Create a new Java program, *JDBCTest.java*. Import **java.sql.\*** so that you can use the **JDBC API**. Add a **main()** method.

JDBCTest.java

```
package examples;

import java.sql.*;

public class JDBCTest {
    public static void main(String args []) {
    }
}
```

# CONNECTING TO THE DATABASE

✳ To connect your Java program to the database invoke the **DriverManager.getConnection()** static method.

- **getConnection()** takes a **String** argument (the URL for a connection) and returns a **Connection** object.

```
String URL = "jdbc:mysql://localhost:3306/sdvid";  
Connection conn = DriverManager.getConnection(URL);
```

- Authentication information (username, password) may be embedded in the URL, or may be passed as separate arguments to **getConnection()**.

✳ **DriverManager** will load any available **Driver** implementations and ask each to connect to the URL.

- To make sure your desired **Driver** class is loaded use **Class.forName()**.

```
Class.forName("vendorpackage.Driver");
```

- **Class.forName()** throws **ClassNotFoundException**.
- When it loads, the driver will register itself with the **DriverManager**.

**Hands On:**

Use the constructor to make sure our database driver loads and registers with the **DriverManager**. We will use methods that throw **SQLException**. For now, rather than catching it, just declare **main()** as throwing it. Declare a **String** variable named **URL** and initialize it with the correct connection URL for your classroom database. Use the URL to obtain a connection from the **DriverManager**.

JDBCTest.java

```
package examples;

import java.sql.*;

public class JDBCTest {
    private static String url = "jdbc:mysql://localhost:3306/sdvid";

    public JDBCTest() throws ClassNotFoundException {
        Class.forName("com.mysql.jdbc.Driver");
    }

    public static void main(String args []) throws SQLException {
        String user = "student";
        String pass = "student";
        Connection conn = DriverManager.getConnection(url,user,pass);
        conn.close();
    }
}
```

**Note:**

This code uses the MySQL Connector Java driver. Other drivers will require different URL strings, and may require a username and password to get the connection. Your instructor will provide the specific connection URL for your environment as well as instructions for starting the database.



### CREATING A SQL QUERY

- ✴ To execute SQL statements, you must create a **Statement** object.
  - Use **Connection**'s **createStatement()** method to do so.
- ✴ The **Statement** interface provides methods for executing SQL queries and updates.
  - The methods we'll use for now take a single **String** argument: a SQL statement.
- ✴ You can reuse a **Statement** object to execute additional SQL statements.
  - SQL executed by a **Statement** object is parsed and compiled by the DBMS on each execution.
  - You can use a single **Statement** object to execute any number of SQL statements at different points in your program.

**Hands On:**

Declare a **String** variable. This will be the text of the SQL statement. Use **Connection.createStatement()** to create a **Statement** object.

JDBCTest.java

```
package examples;

import java.sql.*;

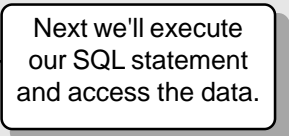
public class JDBCTest {
    private static String url = "jdbc:mysql://localhost:3306/sdvid";

    public JDBCTest() throws ClassNotFoundException {
        Class.forName("com.mysql.jdbc.Driver");
    }

    public static void main(String args []) throws SQLException {
        String user = "student";
        String pass = "student";
        Connection conn = DriverManager.getConnection(url,user,pass);

        String sqltxt;
        sqltxt = "SELECT id, first_name, last_name FROM staff";
        Statement stmt = conn.createStatement();

        conn.close();
    }
}
```



Next we'll execute  
our SQL statement  
and access the data.

### GETTING THE RESULTS

- ✴ Executing a SQL query (that is, a SQL **SELECT** statement) produces a *result set* — a "virtual table" consisting of the rows and column values retrieved by the **SELECT** statement.
- ✴ If you are making a SQL query, use **Statement's** **executeQuery()** method.
  - **executeQuery()** takes a **String** argument, the SQL **SELECT** statement.
  - **executeQuery()** returns a **ResultSet** object.
  - The **ResultSet** object represents the opened cursor for the SQL query.
- ✴ Use your **ResultSet** object's **next()** method to retrieve each row, in turn, of the result set.
  - **next()** returns **false** after the last row has been returned.
- ✴ **ResultSet's** **getString(*col*)** method returns the **String** value of the column whose name or number (left-to-right, starting with **1**, as listed in the **SELECT** clause) is *col*.
  - Other **ResultSet** **getXXX()** methods return DBMS values as different Java datatypes, performing any necessary conversion.
- ✴ The **close()** method of either the **ResultSet** or its **Statement** closes the cursor and frees resources.

**Hands On:**

Declare a **ResultSet** variable and use **executeQuery()** to get a **ResultSet** object, passing your SQL string to **executeQuery()**. Use a **while** loop to iterate through the **ResultSet**'s rows, calling **ResultSet.next()**. Get and print out the values of all columns retrieved.

JDBCTest.java

```
package examples;

import java.sql.*;

public class JDBCTest {
    private static String url = "jdbc:mysql://localhost:3306/sdvid";

    public JDBCTest() throws ClassNotFoundException {
        Class.forName("com.mysql.jdbc.Driver");
    }

    public static void main(String args []) throws SQLException {
        String user = "student";
        String pass = "student";
        Connection conn = DriverManager.getConnection(url,user,pass);

        String sqltxt;
        sqltxt = "SELECT id, first_name, last_name FROM staff";
        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery(sqltxt);
        while(rs.next()) {
            System.out.println(rs.getString(1) + " " +
                               rs.getString(2) + " " +
                               rs.getString(3));
        }
        rs.close();
        stmt.close();

        conn.close();
    }
}
```

### JDBC TYPES

- ✴ JDBC defines datatypes to separate the Java developer from the database implementation.
  - Database vendors often support different types of data or use different names for the datatypes.
  - The driver is responsible for converting DBMS types to JDBC types.
  - JDBC datatypes are based on standard SQL datatypes.
- ✴ When you use a **getXXX()** method on a **ResultSet**, the **XXX** refers to the Java type that is returned.

```
int id = rs.getInt(1);  
String lastName = rs.getString(3);
```

- There is a suggested **getXXX()** method for each JDBC datatype, but there is usually more than one choice.
  - For example, you can retrieve a DBMS INTEGER value as a JDBC **String**.
  - **getObject()** works for any JDBC type.
- Look in the mapping table to see which JDBC types are supported by which **getXXX()** methods.
- ✴ The **java.sql.Types** class defines a **static final int** for each JDBC datatype.
  - These constants are the way that JDBC references the type.

Mapping Table

	TINYINT	SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	BIT	CHAR	VARCHAR	LONGVARCHAR	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP	CLOB	BLOB	ARRAY	REF	STRUCT	JAVA OBJECT
getBytes()	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓												
getByte()	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓												
getShort()	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓												
getInt()	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓												
getLong()	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓												
getFloat()	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓												
getDouble()	✓	✓	✓	✓	✓	✗	✗	✓	✓	✓	✓	✓	✓												
getBigDecimal()	✓	✓	✓	✓	✓	✓	✓	✗	✗	✓	✓	✓	✓												
getBoolean()	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓												
getString()	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✓	✓	✓	✓	✓	✓	✓						
getBytes()														✗	✗	✓									
getDate()											✓	✓	✓				✗		✓						
getTime()											✓	✓	✓					✗	✓						
getTimestamp()											✓	✓	✓				✓	✓	✗						
getAsciiStream()											✓	✓	✗	✓	✓	✓									
getBinaryStream()														✓	✓	✗									
getCharacterStream()											✓	✓	✗	✓	✓	✓									
getClob()																				✗					
getBlob()																					✗				
getArray()																						✗			
getRef()																							✗		
getObject()	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗

✗ — Preferred

✓ — Optional (can implicitly convert)

### FINISHING UP

- ✴ Typically, the open cursor associated with a **ResultSet** object consumes DBMS resources (locks, shared memory, temporary table space, etc.).
- ✴ **Connection** and **Statement** objects also hold DBMS and local resources.
- ✴ When a **Connection**, **Statement**, or **ResultSet** object goes out of scope, it is subject to Garbage Collection.
  - Garbage Collection should free the resources associated with a Java database object, but . . .
  - Don't count on Garbage Collection: free cursor and other resources explicitly when your program is finished with them.
  - Each interface provides a **close()** method to do this.
- ✴ When a **Statement** is closed, any **ResultSets** associated with it are automatically closed.
- ✴ Closing a **Connection** normally disconnects you from the database.
- ✴ For simple database operations you can use try-with-resources syntax to automatically close your **Connection**, **Statement**, and/or **ResultSet**.
  - Embed the declarations of the **Closeable** resources within a set of parenthesis after the **try** keyword.

```
try (Connection c = DriverManager.getConnection(URL) ;  
    Statement stmt = c.createStatement() ;  
    ResultSet rs = stmt.executeQuery(sqltxt) ; ) {  
    ...  
}
```

- Omit the explicit calls to the **close()** methods on these resources.

EmployeeQuery1.java

```
...
    try {
        conn = DriverManager.getConnection(URL, user, pword);
        stmt = conn.createStatement();
        rs = stmt.executeQuery(sqltxt);
        while (rs.next()) {
            System.out.println(rs.getString(1) + " "
                               + rs.getString(2) + " " + rs.getString(3));
        }
        rs.close();
        stmt.close();
    }
    catch (SQLException e) {
        System.err.println(e);
    }
    finally {
        try {
            if (rs != null) { rs.close(); }
            if (stmt != null) { stmt.close(); }
            if (conn != null) { conn.close(); }
        }
        catch (SQLException sqle) {
            System.err.println(sqle);
        }
    }
...

```

Like most I/O and networking classes, these JDBC objects implement the **AutoClosable** interface which means they have a **void close()** method. In "try-with-resources" syntax you instantiate **AutoClosable** objects in the try statement; Java will automatically invoke their **close()** methods when the block completes

EmployeeQuery2.java

```
...
    try (Connection conn=DriverManager.getConnection(URL,user,pword);
         Statement stmt = conn.createStatement();
         ResultSet rs = stmt.executeQuery(sqltxt); ) {

        while (rs.next()) {
            System.out.println(rs.getString(1) + " "
                               + rs.getString(2) + " " + rs.getString(3));
        }
    }
    catch (SQLException e) {
        System.err.println(e);
    }
...

```



### LABS

- ❶ Write a program to retrieve and print the names, email addresses, and last update times of customers. Have your program exit cleanly after printing only the first 20 customers.  
(Solution: *Customers..java*)





## DATA MANIPULATION AND TRANSACTIONS

### OBJECTIVES

- \* Add new rows to database tables.
- \* Modify rows in tables.
- \* Delete rows from tables.
- \* Perform tasks consisting of more than one SQL statement.
- \* Make your changes permanent and visible to other users.
- \* Undo the effects of SQL statements.
- \* Access database tables concurrently with other users.
- \* Prevent other users from interfering with your changes.

## THE INSERT STATEMENT

- ✴ Use the **INSERT** statement to add new rows of data to a table.

```
INSERT INTO table_name [(column_name_list)]  
{VALUES (value_list) | subquery };
```

- Use ***column\_name\_list*** to specify the columns for which you will be providing values.
- You must own the table or have **INSERT** privilege on the table.

- ✴ To add a single row to the table, use the **VALUES** clause.

```
INSERT INTO actor (id, first_name, last_name)  
VALUES (201, 'Troy', 'McClure');
```

- The order and number of values must match the column list.

- ✴ To add zero or more rows at once by copying from existing data, use a subquery.

```
INSERT INTO film_actor (actor_id, film_id)  
SELECT 201, id FROM film WHERE title LIKE '%LOSE%';
```

- The subquery's **SELECT** list can include column values, expressions, and literals.
  - You can generate rows in which some column values come from existing records, while others are provided by the statement itself.

- ✴ Use the **NULL** or **DEFAULT** keywords in place of a value to insert a null (if the column permits nulls), or allow a column to be populated with its default value.

- Most databases support an integer column of type **IDENTITY**, **SERIAL**, **AUTOINCREMENT** etc. for generating primary key values automatically.

```
INSERT INTO address (address, city, state_province, phone)  
VALUES ('123 Fake St.', 'Roswell', 'NM', '5551234');
```

- Consult vendor documentation for how to find the generated key.

```
SELECT LAST_INSERT_ID();
```

You can enter a new project into the **projects** table with the following:

```
INSERT INTO address
VALUES (730, '123 Fake St.', NULL, 'Roswell', 'New Mexico',
       NULL, NULL, '5551234');
```

This format of the **INSERT** command requires that the number and exact order of all columns in the **address** table be known, and match the **VALUES** list in correct order. If the table definition should change, this statement would become invalid. A more reliable (and maintainable) format is to specify the order and names of the columns that are being populated:

```
INSERT INTO address (id, address, address2, city, state_province,
                    postal_code, country_code, phone)
VALUES (730, '123 Fake St.', NULL, 'Roswell', 'New Mexico',
       NULL, NULL, '5551234');
```

You can omit columns, which will be automatically assigned the specified default or (if allowed) null value.

```
INSERT INTO address (id, address, city, state_province, phone)
VALUES (730, '123 Fake St.', 'Roswell', 'New Mexico', '5551234');
```

Databases vary as to whether you can provide an explicit value for an automatically incremented primary key column; if you can, your value must not duplicate an existing one (which is true for any primary key column.) You can always leave such a column out of the **INSERT** statement:

```
INSERT INTO address (address, city, state_province, phone)
VALUES ('123 Fake St.', 'Roswell', 'New Mexico', '5551234');
```

Databases that allow auto-increment key columns usually provide a means to determine the most recently-generated value - for example, MySQL's `LAST_INSERT_ID()` function:

```
SELECT last_insert_id();
```

A subquery can be used to easily populate rows from other data in the database. To add another employee to the database with the same address and store information as an existing employee:

```
INSERT INTO staff (first_name, last_name, address_id, store_id, active,
                  username)
SELECT 'Suzi', 'Chiarelli', address_id, store_id, 1,
      'schiarelli')
FROM employees
WHERE id=68;
```

## THE UPDATE STATEMENT

- \* Use the **UPDATE** statement to change existing data in a single table.

```
UPDATE table_name
  SET column=expression [, column=expression, ...]
[WHERE condition];
```

- You must either own the table or have **UPDATE** privilege.

```
UPDATE film
  SET rental_duration = 7,
      rental_rate = 1.99
  WHERE id = 993;
```

- \* The *expression* may be a subquery that must return only one row.

```
UPDATE staff
  SET supervisor_id = (SELECT manager_staff_id
                      FROM store
                      WHERE id = 4)
  WHERE id = 71;
```

- Specify **DEFAULT** to set a value back to the default value specified for this column when the table was created.

```
UPDATE film
  SET rating = DEFAULT
  WHERE id = 132;
```

- A **NULL** will be issued if there is no default for this column.

- \* The **WHERE** clause identifies the rows to be updated.

- If no **WHERE** clause is used, all rows in *table\_name* are updated.





## THE DELETE STATEMENT

- \* The **DELETE** statement removes rows from a table.
- \* You must either own the table or have **DELETE** privilege.
- \* The format of the **DELETE** command is:

```
DELETE FROM table_name  
[WHERE condition];
```

- \* The **WHERE** clause identifies the rows to be deleted.

delete.sql

```
DELETE FROM payment  
WHERE id >= 56273;
```

- The *condition* can be the same as those found in the **SELECT** statement.
- If no **WHERE** clause is used, all rows in *table\_name* are deleted!

In addition, many databases provide the **TRUNCATE** statement. **TRUNCATE** deletes all of the rows in a table. Unlike the **DELETE** statement, you cannot specify any conditions on which rows are to be deleted. The example below will delete the contents of the **assignments** table:

```
TRUNCATE TABLE film_category;
```

1

**TRUNCATE** is used instead of the **DELETE** statement to free space allocated for the table; and, because each deletion is not logged, the **TRUNCATE** statement is usually faster than **DELETE**. The **TRUNCATE** statement cannot be undone (see **ROLLBACK**) and should, therefore, be used with caution.

## TRANSACTION MANAGEMENT

✴ A database *transaction* groups a series of DML statements into a single logical unit of work.

✴ At any point in a transaction, you can roll back all of your DML statements.

```
ROLLBACK;
```

➤ All of your changes are discarded and a new transaction may begin.

✴ None of your changes are visible in the database until you commit your transaction.

```
COMMIT;
```

➤ All your changes are made permanent and a new transaction may begin.

✴ **COMMIT** as soon as you have successfully completed the statements making up a logical unit of work.

➤ Transactions should have tightly-limited scope.

✴ Some databases operate by default in *autocommit* mode - each individual **INSERT**, **UPDATE**, or **DELETE** is immediately committed and can't be rolled back.

➤ Explicitly begin a multistatement transaction with the **SET TRANSACTION** or **START TRANSACTION** command.

➤ You can add savepoints to your transactions to give yourself a place to partially rollback to.

```
SAVEPOINT step1;
```

```
ROLLBACK TO step1;
```

Transactions help you manage tasks that must succeed as a logical group. To add a new employee the employee must first have an address:

1. Start a transaction:

```
START TRANSACTION;
```

2. Insert a new addresss and employee records, and set the employee's supervisor to his store manager:

```
INSERT INTO address( address, city, state_province, country_code, phone)
VALUES ('123 Fake St', 'Denver', 'CO', 'US', '5551234');

INSERT INTO staff ( first_name, last_name, address_id,
store_id, active, username)
VALUES ( 'Bob', 'Dobbs', LAST_INSERT_ID(), 2, 1, 'bdobbs');

UPDATE staff SET supervisor_id = (SELECT manager_staff_id FROM store
WHERE store.id = staff.store_id)
WHERE id = LAST_INSERT_ID();
```

4. If all three statements succeed, then commit the work:

```
COMMIT;
```

But, if the second **INSERT** or the **UPDATE** statements fail, and we haven't committed yet, then we can roll the work back:

```
ROLLBACK;
```

Note that if we had committed, then the **ROLLBACK** statement would have no effect.

### CONCURRENCY

- ✴ Databases use *locks* of various types to coordinate data manipulation, and many other activities, among concurrent sessions.
- ✴ When concurrent sessions access the same rows and perform DML statements — **INSERTs**, **UPDATEs**, and **DELETEs** — the database isolates each session's work.
  - This assures one session can't update or delete a row another session has already updated, until the other session does a **COMMIT** (or **ROLLBACK**).
- ✴ When a transaction begins manipulating data, it creates an *Exclusive Lock*, and associates all inserted, modified, or deleted rows with that lock.
  - Another transaction trying to manipulate any of the rows will find the lock and wait.
  - When the first transaction commits or rolls back, its lock clears, and the next transaction in the queue can lock the rows it needs.
- ✴ A deadlock can occur if one transaction requires access to data locked by another in order to complete, while at the same time holding a lock that the other transaction is waiting for.
  - Most databases automatically detect this situation, rolling back statements or transactions to clear the deadlock.
- ✴ It's possible to explicitly lock data without performing DML:
  - **LOCK TABLES *tablename* WRITE;**
  - **SELECT \* FROM *tablename* FOR UPDATE;**

Locking is one of the more intricate and complicated features of a database, and the internal details vary from vendor to vendor.

# LOADING TABLES FROM EXTERNAL SOURCES

- ✴ Many systems provide statements or utilities that allow you to load bulk data from external files.
- ✴ The MySQL **LOAD DATA** command loads a file into a table according to the format and other options specified.
  - The **INFILE** parameter specifies the filename.
    - By default this refers to a file on the MySQL server host, with the path relative to MySQL's data directory.
    - Use **LOCAL INFILE** to specify a file relative to the location from which you're running the **mysql** client.
- ✴ The **mysqlimport** utility loads rows into a table from external data files.
  - The name of the datafile determines which table the data will load.
  - Command-line options specify the file's format, following the syntax of **LOAD DATA**.
- ✴ The **mysqldump** utility exports a database's table definitions and/or data, which you can redirect to a file.

```
mysqldump mydb > mydbdump.sql
```

- The dump contains standard SQL commands to create and insert data into the dumped tables; you can run this as a script using **mysql**.

```
mysql -u someuser -p < mydbdump.sql
```

MySQL provides a DML command, **LOAD DATA**, for loading rows into a table from an external source. **LOAD DATA** can load data files, parsing records (according to your specifications) into values of the correct types for your table's columns using whatever delimiters are used in the file.

empdata.txt

```
LNAME, FNAME, ADDRID, STORENUM, USERID
"Tsang", "Leila", 1, 2, ltsang
"Sierra", "Lisa", 1, 6, lsierra
"Happ", "Bill", 4, 1, bhapp
"Garia", "Pete", 1, 5, pgaria
"Ehrlich", "Donna", 1, 4, dehrlich
"Clemens", "Edith", 1, 8, eclemens
"Gonzales", "Pamela", 1, 1, pgonzales
"Merlick", "Mary", 1, 6, mmerlick
"Hausuarde von Ausfernschplendenschliptercrasscren", "Keith", 1, 6, khausuar
"Sierra", "Lance", 1, 4, lsierra
"Plank", "Scott", 1, 2, splank
"Minor", "Sammy", 1, 8, sminor
```

Use **LOAD DATA** to parse and load this into the **employees** table:

```
LOAD DATA
LOCAL INFILE 'empdata.txt'
IGNORE
INTO TABLE staff
FIELDS TERMINATED BY ','
OPTIONALLY ENCLOSED BY '"'
IGNORE 1 LINES
(last_name, first_name, address_id, store_id, username)
SET password=PASSWORD(username);
```

You may see messages about skipped records and warnings - run **SHOW WARNINGS** immediately after the **LOAD DATA** command to see the details (warnings are reset for each new SQL command).

```
show warnings;
```



### LABS

Through these labs, read carefully and be sure to consider whether and when to start and commit or roll back transactions.

- ❶ Insert your own address into the **address** table, then determine the address ID that was assigned.
- ❷ Using your new address ID, insert yourself into the **staff** table.
- ❸ With a single statement, set your **supervisor\_id** to be the manager of your store.
- ❹ Using an INSERT with a subquery, add a record for yourself in the **customer** table by retrieving appropriate fields from your **staff** record and including any other necessary customer items.
- ❺ You've convinced the company to hire your roommate. Add their new staff record using your same address and store information. Make yourself their supervisor.
- ❻ Delete all the payment records.
- ❼ What — are you nuts!? Get those payment records back!





## JDBC SQL PROGRAMMING

### OBJECTIVES

- ✧ Use **SQLExceptions** and **SQLWarnings** to detect and handle DBMS errors and warnings.
- ✧ Describe the most important JDBC interfaces, and use their methods.
- ✧ Use the JDBC datatypes to convert DBMS-specific data to Java data.
- ✧ Use **ResultSetMetaData** to handle dynamically-generated SQL.
- ✧ Use **PreparedStatement** objects to more efficiently run a SQL statement repeatedly.
- ✧ Invoke DBMS stored procedures and functions from Java.
- ✧ Manage DBMS transactions.

## ERROR CHECKING AND THE **SQLException** CLASS

- ✴ A runtime error from the DBMS triggers a **SQLException**.
- ✴ The **SQLException**'s methods provide the standard DBMS error information:
  - **getErrorCode()** returns the vendor-specific SQLCODE error code.
  - **getMessage()** is overridden by the **SQLException** class to return the vendor-specific error message.
  - **getSQLState()** returns the ANSI-standard SQLSTATE value (if your DBMS supports it).
- ✴ Most methods defined under **java.sql** throw **SQLException**; assume that your JDBC code needs to either catch or declare **SQLException**.

Database programmers rely on a simple mechanism for obtaining error information from the database engine: after every database operation, the DBMS sets an error value that the programmer can check. An error is typically identified by a numeric error code (SQLCODE) and a brief error message. By convention, DBMS error codes are negative numbers; a SQLCODE value of 0 means no error has occurred; and a special value, typically (but not always) +100, means there was no error, but no data was found to satisfy the operation.

SQLCODE values are vendor-specific, however, so programmers must learn the specific code values for each product they program against. To relieve this situation, standards organizations introduced the SQLSTATE, a standardized, 5-character string encoding the error condition.

Some DBMSs now implement SQLSTATE, though most still also return their own, vendor-specific, SQLCODE values and messages.

With JDBC, when you call certain methods, the driver automatically checks the SQLSTATE/SQLCODE and throws a **SQLException** when there's an error.

## EXECUTING SQL UPDATES

- ✴ Use **Statement's** `executeUpdate()` method to run SQL statements that do not generate result sets.

- Data Manipulation Language (DML): **INSERT, UPDATE, DELETE.**

```
stmt.executeUpdate( "DELETE FROM staff " +  
                    " WHERE id = 55");
```

- Data Definition Language (DDL): **CREATE, ALTER, DROP, GRANT, REVOKE.**

```
stmt.executeUpdate( "GRANT SELECT ON staff " +  
                    " TO '%@localhost'");
```

- ✴ **`executeUpdate()`** returns an **int**, the *update count*, which is one of two values:

```
int uc;  
uc = stmt.executeUpdate( "UPDATE film "  
                          + "   SET rental_rate = rental_rate + 1 "  
                          + " WHERE language_id = 3");
```

- The number of rows affected by a DML statement.
  - This may validly be **0** if no rows match an **UPDATE** or **DELETE** statement's **WHERE** clause or an **INSERT**'s subquery returns no rows.

- **0** for DDL or other statements.

- ✴ **`executeUpdate()`** throws an exception if you use it to run a **SELECT** or other statement that generates a result set.





## THE EXECUTE() METHOD

- ✴ **Statement's execute()** method is the most general way to execute SQL.
- ✴ **execute()** executes any arbitrary SQL string.
  - It may generate either a result set, an update count, a combination of any number of either, or even no results at all; for example:
    - A statement built dynamically at runtime, which you can't restrict to either a query or an update.
    - A stored procedure yielding multiple result sets or update counts (if the DBMS supports this).
- ✴ **execute()** returns a **boolean**.
  - **true** if the statement generated an initial result set.
  - **false** if the statement generated either an initial update count or no results at all.
- ✴ Use **getResultSet()** and **getUpdateCount()** to retrieve the actual results of the statement.
  - **getResultSet()** returns a **ResultSet** object if one is available, otherwise **null**.
  - **getUpdateCount()** returns an **int**: **0** or greater for DML or DDL, **-1** if no update count is available.

After an **execute()**, your **Statement** object has either a **ResultSet**, an update count, or neither. In addition, after you are finished with that result (either the **ResultSet** or the update count), there may be another result to retrieve and process. If you have no idea what to expect, you can still process all of your results:

ExecuteExample.java

```
...
public class ExecuteExample {
    public static void main(String args[]) {
        String url = "jdbc:mysql://localhost:3306/sdvid";
        String user = "student";
        String pword = "student";
        String sqlstring = readInSQLStatement();
        System.out.println("Executing SQL statement:\n" + sqlstring
            + "\n");
        try (Connection conn=DriverManager.getConnection(url,user,pword);
            Statement stmt = conn.createStatement();) {

            boolean haveResultSet = stmt.execute(sqlstring);
            if (haveResultSet) {
                ResultSet rs = stmt.getResultSet();
                ResultSetMetaData rsmd = rs.getMetaData();
                int cols = rsmd.getColumnCount();
                for (int col = 1; col <= cols; col++)
                    System.out.print(rsmd.getColumnName(col) + "\t");
                System.out.println("");
                while (rs.next()) {
                    for (int col = 1; col <= cols; col++) {
                        System.out.print(rs.getString(col) + "\t");
                    }
                    System.out.println("");
                }
                rs.close();
            }
            else { // No result set.
                int uc = stmt.getUpdateCount();
                System.out.println(uc + " row(s) updated.");
            }
        }
        ...
    }
    ...
}
```

### Try It:

Run *ExecuteExample.java*. Enter a **SELECT**, **INSERT**, **UPDATE**, or **DELETE** SQL statement when prompted.

## RESULTSETMETADATA

- \* For queries built at runtime for which you need to determine the number, types, names, etc. of columns of a **ResultSet**, get the **ResultSet**'s metadata.

```
ResultSetMetaData rsmd = rs.getMetaData();
```

- \* A **ResultSetMetaData** object's methods return information about the columns of its **ResultSet**.

```
int getColumnCount()
String getColumnName(int column)
String getColumnLabel(int column)
int getColumnType(int column)
String getColumnName(int column)
int getPrecision(int column)
int getScale(int column)
int getColumnDisplaySize(int column)
int isNullable(int column)
boolean isReadOnly(int column)
boolean isCurrency(int column)
etc...
```

- All of these methods may throw **SQLException**.

- \* **getColumnType()** returns an **int** corresponding to a constant defined in **java.sql.Types**.

- This can be useful in determining which **getXXX()** method to call.

- \* **ResultSetMetaData** is valuable when:

- **execute()** was used to run some arbitrary query.
- **executeQuery()** was used to run a query statement that was generated at runtime.

ExecuteFormat.java

```
...
public class ExecuteFormat {

    public static void main(String args[]) {
        ...
        ResultSetMetaData rsmd = rs.getMetaData();
        int cols = rsmd.getColumnCount();
        int colWidth[] = new int[cols];

        String resultLine = "";
        for (int col=1; col <= cols; col++) {
            colWidth[col-1] = getAppDisplayWidth(rsmd, col);
            resultLine += rPadTrunc(rsmd.getColumnName(col),
                                   colWidth[col-1], ' ');
            if (col < cols) {
                resultLine += " ";
            }
        }
        System.out.println(resultLine);
        ...
    }

    static int getAppDisplayWidth(ResultSetMetaData rsmd, int col)
        throws SQLException {

        switch (rsmd.getColumnType(col)) {
            case Types.NUMERIC:
            case Types.INTEGER:
                return 9;
            case Types.TIMESTAMP:
                return 22;
            case Types.DATE:
                return 10;
            case Types.TIME:
                return 8;
            default :
                return rsmd.getColumnDisplaySize(col);
        }
    }
}
```

**Try It:**

Run *ExecuteFormat.java* to see all **Department** data printed out in columns with headers.

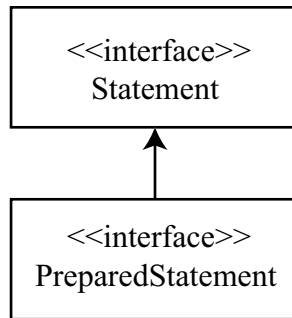
## USING A PREPAREDSTATEMENT

- \* If your program executes the same SQL statement repeatedly, perhaps just with different values in certain places, use a **PreparedStatement**.

- Use **Connection**'s **prepareStatement()** method to instantiate a **PreparedStatement** object.

```
PreparedStatement pst;  
pst=conn.prepareStatement("SELECT formal_name"  
    + " FROM country"  
    + " WHERE country_code = 'IS'");  
ResultSet rs = pst.executeQuery();
```

- You must supply the SQL syntax when you create the **PreparedStatement**, not when you execute it.
- \* A **Statement**'s *execute* methods - **execute(sqlstr)**, **executeQuery(sqlstr)**, **executeUpdate(sqlstr)** - cause the DBMS to parse, compile, and execute the SQL syntax each time.
  - A **PreparedStatement**'s SQL syntax is pre-parsed and compiled by the DBMS only once for repeated execution.
- \* **PreparedStatement** descends from **Statement**.
  - Note that a **PreparedStatement**'s *execute* methods don't take a SQL **String** argument — thus they don't override the **Statement**'s *execute(String sqlstr)* methods.
    - Do not use the **Statement**'s *execute(String sqlstr)* methods with a **PreparedStatement** object.



## PARAMETERIZED STATEMENTS

- \* You can execute a prepared statement multiple times with different values for where clause predicates or DML **SET** or **VALUES** clause parameters.
- \* When creating the statement, use a question mark, **?**, as a placeholder.

```
PreparedStatement pst;  
pst = conn.prepareStatement( "SELECT formal_name"  
    + " FROM country"  
    + " WHERE country_code = ?");
```

- \* You must set values for all placeholders before executing the statement.
  - Use a **PreparedStatement**'s **setXXX()** methods, where **XXX** is the Java datatype (the JDBC driver implicitly converts to the appropriate JDBC datatype).

```
pst.setString(1, "IS");  
ResultSet rs = pst.executeQuery();
```

- The **setXXX()** methods take the index of the placeholder, from left-to-right starting at **1**, as the first argument.

```
int storeId = 5;  
int mgrId = 75;  
pst = conn.prepareStatement("UPDATE store "  
    + " SET manager_staff_id = ? "  
    + " WHERE id = ?");  
pst.setInt(1, mgrId);  
pst.setInt(2, storeId);  
int uc = pst.executeUpdate();  
System.out.println(uc + " row(s) updated.");
```

This example uses a prepared statement to retrieve the department name using the department code.

#### PrepParamExample.java

```
...
public class PrepParamExample {
    public PrepParamExample() {
        String url = "jdbc:mysql://localhost:3306/sdvid";
        String user = "student";
        String pword = "student";
        String sql = "SELECT formal_name FROM country "
            + "WHERE country_code = ?";

        try (Connection conn = DriverManager.getConnection(url,user,pword);
            PreparedStatement pst = conn.prepareStatement(sql);) {

            System.out.println(getCountryName(pst, "IS"));
            System.out.println(getCountryName(pst, "NA"));
            System.out.println(getCountryName(pst, "CX"));
        }
        catch (SQLException e) {
            System.err.println(e);
        }
    }

    private String getCountryName(PreparedStatement pst, String ctyCode)
        throws SQLException {
        pst.setString(1, ctyCode);
        ResultSet rs = pst.executeQuery();
        String name = "";
        while (rs.next()) {
            name += rs.getString(1) + "\n";
        }
        return name;
    }

    public static void main(String args[]) {
        new PrepParamExample();
    }
}
```



#### Try It:

Run this program.



## TRANSACTION MANAGEMENT

- ✴ A DBMS can log all DML operations in a session as part of a *transaction*.
  - At your request, the DBMS will either:
    - *Commit* the entire transaction, making your operations permanent and visible to other users.
    - *Rollback* the entire transaction, completely undoing all DML operations.
  - When you commit or rollback a transaction, a new transaction can begin.
- ✴ The default behavior of a **Connection** object is to automatically commit after executing each statement.
  - In this case, each statement is a single, independent transaction.
- ✴ To turn off autocommit mode, use **setAutoCommit(false)**.
  - You can then use the **commit()** or **rollback()** methods of your **Connection** object to manage your transactions.
  - The **getAutoCommit()** method of your **Connection** object returns **true** when autocommit mode is on.

In most cases, autocommit is not desirable — you don't have the ability to check errors and warnings and program appropriately, or to treat multiple individual statements as parts of a single transaction, or to set transaction savepoints (if your DBMS supports them).

### **Transaction Side Effects**

- Some DBMSs, notably Oracle, automatically commit the current transaction any time you execute a DDL statement.
- Committing or rolling back the current transaction frees all locks.
- Committing the current transaction frees the update locks associated with an updatable cursor, thus forcing the cursor closed.

### LABS

- ❶ Write a program to print a listing of all film titles and replacement costs, and the total of all replacement costs at the end.  
(Solution: *FilmRpt.java*)
- ❷ Write a program that uses a **PreparedStatement** to retrieve the id, name, and email of all employees of a store. The statement should take the store ID as its only parameter. Prompt the user for a store ID and use the response to run the query and print the results.  
(Solution: *StoreEmps.java*)
- ❸ Write a program that prompts the user to enter a SQL statement (using **System.in** or a simple GUI, as you prefer). The program should then determine if the statement is a query or not, and execute it using the appropriate method. If it is a query, just print out the number of rows found. If it is a DML or other statement, print the number of rows affected.  
(Solution: *SQLExec.java*)
- ❹ Modify your solution to *SQLExec.java*: if the statement is executed as a query, use **ResultSetMetaData** to determine the number of columns retrieved by the query. For each row, print out all columns.  
(Solution: *SQLExec2.java*)
- ❺ Modify your solution to *SQLExec.java*: if the statement is executed as a query, use **ResultSetMetaData** to determine the number of columns retrieved by the query. For each row, print out all columns.  
(Solution: *SQLExec2.java*)





## SQL QUERIES – JOINS

### OBJECTIVES

- ✧ Collect data from multiple tables with a single query.
- ✧ Use the relational aspects of your database in queries.
- ✧ Differentiate between filter conditions and join conditions.

## SELECTING FROM MULTIPLE TABLES

✳ To retrieve data from more than one table in the same **SELECT** statement, the tables are *joined* together.

✳ Specify the tables in the **FROM** clause using the **INNER JOIN** keywords.

- Use the **ON** clause with a join condition to specify how the rows in the second table should match the original rows.

```
SELECT address, city, country
FROM address INNER JOIN country
      ON address.country_code = country.country_code;
```

- The word **INNER** is optional.

✳ If you reference a column whose name appears in more than one table, you must precede the column name with the table name, called a *qualified reference*.

```
SELECT country.country_code, city, ...
```

- Using a *table alias* can save you some typing.

```
SELECT c.country_code, a.address, a.city, c.country
FROM address a JOIN country c
      ON a.country_code = c.country_code;
```

✳ If the join clause involves only columns with the same name, you may use the **USING** clause, omitting the explicit join condition:

```
SELECT country_code, a.address, a.city, c.country
FROM address a JOIN country c USING (country_code);
```

- With **USING**, the table name or alias cannot be used to qualify the column name, as the join columns are rolled together into a single column.

When you join two tables, the result set consists of all of the rows that matched the join condition (and any other conditions). All rows that do not have a corresponding row in the other table are thrown out. This is called an *inner join*.

The following query will return information only for stores with managers - no other stores from the **stores** table will have a matching **id** in the **staff** table.

```
SELECT s.id, s.address_id, e.last_name
FROM store s JOIN staff e ON e.id = s.manager_id;
```

Joins often illustrate one-to-many relationships in your database schema. For example, one store might have many employees:

```
SELECT s.id, s.address_id, e.first_name, e.last_name
FROM store s JOIN staff e ON s.id = e.store_id;
```

**CROSS JOIN** — A join done without any join conditions. Every row in the first table is matched with every row in the second table. This is also called a *Cartesian product*.

```
SELECT s.id, s.address_id, e.first_name, e.last_name
FROM store s CROSS JOIN staff e;
```

This cross join will show the combination of every employee with every store - this is seldom useful



### JOIN CONDITIONS

- \* A *join condition* is a predicate (in the ON or WHERE clause) that relates columns from two tables.

- A predicate that references only one table is a *filter condition*.

- \* Though you can join tables using any columns with compatible data types, parent/child relationships are often used in the join criteria.

```
SELECT e.last_name, e.first_name, a.city, a.state_province
FROM staff e JOIN address a ON e.address_id = a.id
WHERE e.id = 59;
```

- This relationship is implemented with a *foreign key*.

- \* The join condition can also be complex, involving multiple columns.

```
SELECT e.id, e.first_name, e.last_name, c.first_name
FROM staff e JOIN customer c ON e.last_name = c.last_name
AND e.address_id = c.address_id;
```

- \* All fields from a table can be selected.

```
SELECT e.id, e.first_name, e.last_name, c.*
FROM staff e JOIN customer c ON e.last_name = c.last_name
AND e.address_id = c.address_id;
```

Joining the same tables using different join conditions will retrieve very different result sets.

This query retrieves the names of the store managers.

```
SELECT s.id, a.city, e.id, e.first_name, e.last_name
  FROM staff e JOIN store s ON e.id = s.manager_id
                        JOIN address a ON s.address_id = a.id
 ORDER BY city;
```

This query retrieves the names and store locations of all employees.

```
SELECT s.id, a.city, e.id, e.first_name, e.last_name
  FROM staff e JOIN store s ON e.store_id = s.id
                        JOIN address a ON s.address_id = a.id
 ORDER BY city;
```

### Note:

The examples in this chapter use the ANSI standard syntax for joins, in which you list join conditions in the **FROM** clause and filter predicates in the **WHERE** clause:

```
SELECT c.country_code, a.address, a.city, c.country
  FROM address a JOIN country c ON a.country_code = c.country_code
 WHERE c.country LIKE 'U%';
```

However, you might see older join syntax, which is still valid SQL, in which the join conditions are mingled with filter predicates in the **WHERE** clause:

```
SELECT c.country_code, a.address, a.city, c.country
  FROM address a, country c
 WHERE c.country LIKE 'U%'
    AND a.country_code = c.country_code;
```

## ASSOCIATIVE TABLES

- \* Joins often use tables whose columns aren't included in the results.
  - Use joins to link rows from tables that don't have a direct connection.
    - Use **address** to connect **customer** to **country**:

```
SELECT c.first_name, c.last_name, co.country
FROM customer c JOIN address a
              ON c.address_id = a.id
              JOIN country co
              ON a.country_code = co.country_code
ORDER BY c.last_name;
```

- \* An *associative table* is often used to resolve many-to-many relationships.
  - A **film** has many **actors**, and an **actor** appears in many **films**; **film\_actor** associates these two entities.

```
SELECT a.first_name, a.last_name, f.title
FROM actor a JOIN film_actor fa ON a.id = fa.actor_id
              JOIN film f ON fa.film_id = f.id
ORDER BY f.title;
```

- \* There is no limit to the number of tables that can be joined together.
  - Each table joined into the query will have its own join criteria, matching its data to data from any of the previously listed tables.

```
SELECT a.first_name, a.last_name, f.title, l.name
FROM actor a JOIN film_actor fa ON a.id = fa.actor_id
              JOIN film f      ON fa.film_id = f.id
              JOIN language l ON f.language_id = l.id
ORDER BY f.title;
```

You will need to carefully decide which tables are joined to which other tables in your query. In this first query, the **address** table is joined with the **staff** table and the **staff** table to the **store** table. We are retrieving a list of all managers with their *home* phone numbers.

```
SELECT e.first_name, e.last_name, a.phone
FROM staff e JOIN store s ON e.id = s.manager_id
      JOIN address a ON e.address_id = a.id
ORDER BY last_name;
```

This next query involves the same three tables, but they are joined together differently. We are joining the **address** table to the **store** table and the **store** table to the **staff** table. We are retrieving a list of managers with their *store's* phone numbers.

```
SELECT e.first_name, e.last_name, a.phone
FROM staff e JOIN store s ON e.id = s.manager_id
      JOIN address a ON s.address_id = a.id
ORDER BY last_name;
```

These two queries retrieve very different result sets.

### LABS

- ❶ Using joins, write queries to:
  - a. List all stores and their manager's first and last name.
  - b. List all stores and their street addresses .
  - c. List the names and street addresses of all employees who live in Denver.
  - d. List the name and city of every employee who doesn't have a username.
- ❷ Using joins, write queries to:
  - a. List the id and full name of each employee, with their full address and their supervisor's id.
  - b. List the id and full name of each employee, with their store's city and state.
  - c. List the full name and address of each customer, with their full country name.
  - d. List all stores along with their full addresses and their managers' full names.
- ❸ Write a query to list employee assignments. It should retrieve employee id and full name, their store location, and their supervisor's id.
- ❹ Create a list of film rentals, sorted by rental date. Include the rental\_id, customer name, rental date, and the name of the employee who processed the rental.
- ❺ Modify the query from ❹ to also list the title of the film..
- ❻ Modify the query from ❺ so that:
  - \* It includes the city of the store the film belongs to.
  - \* Customer and employee names are returned as a single "Firstname Lastname" string.
  - \* Column headers are unambiguous.





## SQL QUERIES – OUTER AND SELF JOINS

### OBJECTIVES

- ✧ Use a self-join to relate records within the same table.
- ✧ Use **LEFT**, **RIGHT**, and **FULL** outer joins.
- ✧ Explain the difference between equijoins and non-equijoins.
- ✧ Explain what an anti-join returns.



## OUTER JOINS

- \* An *outer join* retrieves *all* rows from one table, plus the matching rows from the second table.
  - The column values for the non-matched rows (which would have come from the second table) will be NULL.
- \* Standard join syntax specifies which table, the **LEFT** or **RIGHT**, will have all of its rows returned.

```
SELECT s.id, s.address_id, e.last_name
FROM store s LEFT OUTER JOIN staff e
            ON e.id = s.manager_id;
```

- All records from the **LEFT** table will be returned, with **NULLs** in columns where information from the **RIGHT** table was not available.
- The **OUTER** keyword is optional.
- \* **FULL [OUTER] JOIN** — All matching rows from the two tables, plus rows from each table that have no matching rows in the other.
- \* If you join another table to an outer joined table, you will also need to outer join the new table.

```
SELECT s.id, s.address_id, e.last_name
FROM store s LEFT OUTER JOIN staff e
            ON e.id = s.manager_id
LEFT OUTER JOIN address a
            ON e.address_id = a.id;
```

- Without the second outer join, you will not be able to match against **NULL** values in the result set, and the rows will be lost.

```
SELECT s.id, s.address_id, e.last_name
FROM store s LEFT OUTER JOIN staff e
            ON e.id = s.manager_id
JOIN address a
            ON e.address_id = a.id;
```

When a row in one joined table does not have a matching row in the other table, the row is left out of the result set of an ordinary (**INNER**) join.

```
SELECT emp.first_name, emp.last_name, sup.last_name
FROM staff emp JOIN staff sup ON emp.supervisor_id = sup.id
ORDER BY emp.last_name;
```

With an **OUTER** join, when a row in one joined table does not have a match in the other table, the unmatched row remains in the result set. **NULL** values are supplied for the columns that would have come from the other table.

```
SELECT emp.first_name, emp.last_name, sup.last_name
FROM staff emp LEFT JOIN staff sup ON emp.supervisor_id = sup.id
ORDER BY emp.last_name;
```

**LEFT** and **RIGHT** refer to the position of the table name in the **FROM** clause, relative to the **JOIN** keyword. The **LEFT OUTER JOIN** above can just as easily be written as a **RIGHT OUTER JOIN**:

```
SELECT emp.first_name, emp.last_name, sup.last_name
FROM staff sup RIGHT JOIN staff emp ON emp.supervisor_id = sup.id
ORDER BY emp.last_name;
```

### SELF JOINS

- \* A *self join* queries a single table as though it were two separate tables.
  - Self joins are most frequently performed on tables that model hierarchical data, having foreign keys referencing back into the same table.
- \* To perform a self join just use two different aliases for the same table.

```
SELECT emp.first_name, emp.last_name, sup.last_name
FROM staff emp JOIN staff sup
      ON emp.supervisor_id = sup.id
ORDER BY sup.last_name, emp.last_name;
```

- SQL treats the table as though it were two separate tables.
- \* You can self-join a table any number of times.

```
SELECT emp.first_name, emp.last_name, sup.last_name,
      sup2.last_name
FROM staff emp JOIN staff sup
      ON emp.supervisor_id = sup.id
JOIN staff sup2
      ON sup.supervisor_id = sup2.id
ORDER BY sup2.last_name, sup.last_name, emp.last_name;
```



## EQUIJOINS, NON-EQUIJOINS, AND ANTIJOINS

- \* Most joins are *equijoins* - the join conditions compare for equality.

```
SELECT DATE(rental_date), DATE(payment_date), amount
FROM rental r JOIN payment p
      ON r.id = p.rental_id
      AND DATE(rental_date) = DATE(payment_date);
```

- A *non-equijoin* uses a condition other than equality, such as <, >, **BETWEEN**, or **LIKE** to compare the join columns.

```
SELECT DATE(rental_date), DATE(payment_date), amount
FROM rental r JOIN payment p
      ON r.id = p.rental_id
      AND DATE(rental_date) < DATE(payment_date);
```

```
SELECT f.title, c.name
FROM film f JOIN category c
      ON f.title LIKE CONCAT(SUBSTR(c.name,1), '%');
```

- \* An *antijoin* is an outer join with the matched rows omitted.

```
SELECT r.id, r.rental_date, p.id, p.amount
FROM rental r LEFT JOIN payment p
      ON r.id = p.rental_id
WHERE p.id IS NULL;
```

joins.sql

A	
<i>id</i>	<i>animal</i>
2	cat
3	dog
4	frog
5	giraffe

B	
<i>id</i>	<i>sound</i>
1	tweet
2	meow
3	woof
4	ribbet

**Cartesian Join:**

```
SELECT a.id, a.animal, b.id, b.sound
FROM a CROSS JOIN b;
```

```
SELECT a.id, a.animal, b.id, b.sound
FROM a, b;
```

**Inner Join:**

```
SELECT a.id, a.animal, b.id, b.sound
FROM a INNER JOIN b ON a.id = b.id;
```

```
SELECT a.id, a.animal, b.id, b.sound
FROM a INNER JOIN b USING (id);
```

```
SELECT a.id, a.animal, b.id, b.sound
FROM a NATURAL JOIN b;
```

```
SELECT a.id, a.animal, b.id, b.sound
FROM a, b
WHERE a.id = b.id;
```

**Left Outer Join:**

```
SELECT a.id, a.animal, b.id, b.sound
FROM a LEFT OUTER JOIN b ON a.id = b.id;
```

```
SELECT a.id, a.animal, b.id, b.sound
FROM a, b
WHERE a.id = b.id(+);
```

**Right Outer Join:**

```
SELECT a.id, a.animal, b.id, b.sound
FROM a RIGHT OUTER JOIN b ON a.id = b.id;
```

```
SELECT a.id, a.animal, b.id, b.sound
FROM a, b
WHERE a.id(+) = b.id;
```

**Full Outer Join:**

```
SELECT a.id, a.animal, b.id, b.sound
FROM a FULL OUTER JOIN b ON a.id = b.id;
```

```
SELECT a.id, a.animal, b.id, b.sound
FROM a, b WHERE a.id = b.id(+)
UNION
SELECT a.id, a.animal, b.id, b.sound
FROM a, b WHERE a.id(+) = b.id;
```

**Antijoin:**

```
SELECT a.id, a.animal, b.id, b.sound
FROM a LEFT OUTER JOIN b ON a.id = b.id
WHERE b.id IS NULL;
```

```
SELECT id, animal FROM a
WHERE id NOT IN (SELECT id FROM b);
```

```
SELECT id, animal FROM a
WHERE NOT EXISTS (SELECT 1 FROM b WHERE b.id = a.id);
```

**Cartesian Join:**

<i>a.id</i>	<i>animal</i>	<i>b.id</i>	<i>sound</i>
2	cat	1	tweet
2	cat	2	meow
2	cat	3	woof
2	cat	4	ribbet
3	dog	1	tweet
3	dog	2	meow
3	dog	3	woof
3	dog	4	ribbet
4	frog	1	tweet
4	frog	2	meow
4	frog	3	woof
4	frog	4	ribbet
5	giraffe	1	tweet
5	giraffe	2	meow
5	giraffe	3	woof
5	giraffe	4	ribbet

**Inner Join:**

<i>a.id</i>	<i>animal</i>	<i>b.id</i>	<i>sound</i>
2	cat	2	meow
3	dog	3	woof
4	frog	4	ribbet

**Left Outer Join:**

<i>a.id</i>	<i>animal</i>	<i>b.id</i>	<i>sound</i>
2	cat	2	meow
3	dog	3	woof
4	frog	4	ribbet
5	giraffe	null	null

**Right Outer Join:**

<i>a.id</i>	<i>animal</i>	<i>b.id</i>	<i>sound</i>
null	null	1	tweet
2	cat	2	meow
3	dog	3	woof
4	frog	4	ribbet

**Full Outer Join:**

<i>a.id</i>	<i>animal</i>	<i>b.id</i>	<i>sound</i>
null	null	1	tweet
2	cat	2	meow
3	dog	3	woof
4	frog	4	ribbet
5	giraffe	null	null

**Antijoin:**

<i>a.id</i>	<i>animal</i>	<i>b.id</i>	<i>sound</i>
5	giraffe	null	null

<i>a.id</i>	<i>animal</i>
5	giraffe

### LABS

- ❶** Using joins, write queries to:
  - a. List all stores, their city and state, and their manager's names; include stores with no manager.
  - b. List all stores, their city and state, and their manager's names, and the manager's street address; include stores with no manager.
  - c. List the name and city of every employee who doesn't have a username.
- ❷** Using joins, write queries to:
  - a. List the rental date, return date, and payment amount (where applicable) of all rentals including those that have no payments.
  - b. List only the rentals that have no payments.
- ❸** Write a query to list employee assignments. It should retrieve employee id and full name, their store location, and their supervisor's id.
- ❹** Modify the query from **❸** to include the last name of their supervisor.
- ❺** Modify the query from **❹** to also list all employees, even those without supervisors.
- ❻** Modify the query from **❺** so that:
  - \* Include the name of each employee's store manager.
  - \* The results are sorted by store location and supervisor names.
  - \* Employee name is returned as a single "Firstname Lastname" string.
  - \* Column headers are unambiguous.







## AGGREGATE FUNCTIONS AND ADVANCED TECHNIQUES

### OBJECTIVES

- ✧ Describe and use regular and correlated subqueries.
- ✧ Use the **EXISTS** operator.
- ✧ Use aggregate functions to generate a summary row.
- ✧ Group summary rows by key values.
- ✧ Combine multiple queries using set operators.

### SUBQUERIES

- \* A *subquery* is a **SELECT** that appears as part of another SQL statement.
  - In the **WHERE** clause of another **SELECT** statement - this is called a *nested subquery*.
  - In the **FROM** clause of another **SELECT** statement - this is called an *inline view*.
    - The result of the subquery is treated as though it were another table; you can even give the subquery a table alias.
  - As an expression in a **SELECT**, **VALUES**, or **SET** clause.
- \* A subquery must return the correct number and type of columns and rows.
  - A single-row subquery returns exactly one record, while a multi-row subquery might return more than one record.
  - Operators such as = and != must be given a *scalar subquery*: a single-row subquery with just one column.

```
SELECT first_name, last_name, username
FROM staff
WHERE id = (SELECT manager_id
            FROM store
            WHERE id = 4);
```

- Operators such as **IN** allow a multi-row subquery, one that can return any number of rows (but still with a single column).

```
SELECT first_name, last_name, username
FROM staff
WHERE id IN (SELECT manager_id FROM store);
```

Show the address of the manager of employee #111:

```
SELECT address, city, state_province, postal_code, country_code
FROM address
WHERE id = (SELECT supervisor_id
            FROM staff
            WHERE id = 111);
```

Show the address of the employees supervised by the manager of store #2:

```
SELECT address, city, state_province, postal_code, country_code
FROM address a JOIN staff e ON a.id = e.address_id
WHERE e.supervisor_id = (SELECT manager_id
                        FROM store
                        WHERE id = 2);
```

When do you use a subquery versus a **JOIN** in a **SELECT** statement?

When the query is only returning column values from a single table, you *can* use a subquery in the **WHERE** clause to help limit the result set.

```
SELECT first_name, last_name
FROM staff
WHERE id IN (SELECT customer_id FROM rental);
```

When the query returns column values from multiple tables, you will have to use a **JOIN** so that all of the table data is available in the main query.

```
SELECT first_name, last_name, rental_date
FROM staff c JOIN rental r
ON c.id = r.customer_id;
```

### CORRELATED SUBQUERIES

- ✴ A subquery whose **WHERE** clause refers to a table in the **FROM** clause of a parent query is a *correlated subquery*.
  - The subquery is correlated to - re-executed with - specific values in each row processed by the parent statement.

```
SELECT first_name, last_name
  FROM staff e JOIN rental r
                ON e.id = r.customer_id
 WHERE e.store_id = (SELECT store_id
                    FROM inventory_item ii
                    WHERE r.inventory_id = ii.id);
```

- ✴ Subqueries often can be rewritten as joins in **SELECT** statements.

```
SELECT first_name, last_name
  FROM staff e JOIN rental r ON e.id = r.customer_id
                JOIN inventory_item ii ON r.inventory_id = ii.id
 WHERE e.store_id = ii.store_id;
```

- However, subqueries can help you factor a very complex query into manageable pieces.



# THE EXISTS OPERATOR

- \* The **EXISTS** operator is used in the **WHERE** clause of a correlated subquery to test for the existence of data; it does not return data.
  - The outer query relies on the subquery's boolean result to determine whether to include its current row in the resultset.

```
WHERE [NOT] EXISTS (subquery);
```

- \* An **EXISTS** condition is true if its subquery returns at least one row.
- \* The item in the subquery's **SELECT** list is often the primary key or a \*.

```
SELECT first_name, last_name
  FROM staff e
 WHERE EXISTS (SELECT *
                FROM rental r JOIN inventory_item ii
                        ON r.inventory_id = ii.id
                WHERE r.customer_id = e.id);
```

- It doesn't matter what you put in the subquery's **SELECT** list since no values from the subquery are used in the parent statement.
- \* The **EXISTS** operator results in a *semi-join*.
  - A matched row from the containing query appears only once, no matter how many matching rows would be found by the subquery.

There is always more than one way to do it in SQL:

```
SELECT rental_date, return_date
  FROM rental
 WHERE NOT EXISTS (SELECT *
                   FROM payment
                   WHERE payment.rental_id = rental.id);
```

```
SELECT rental_date, return_date
  FROM rental r
 WHERE r.id NOT IN (SELECT rental_id
                   FROM payment);
```

```
SELECT r.rental_date, r.return_date
  FROM rental r LEFT OUTER JOIN payment p
                                     ON r.id = p.rental_id
 WHERE p.id IS NULL;
```

These are examples of *antijoins* — queries that return rows from one table for which there are no matching rows in the other table.

Although there is always more than one way to do it, not everything you try will actually work. Consider:

```
SELECT r.rental_date, r.return_date
  FROM rental r JOIN payment p
                 ON r.id != p.rental_id;
```

Exactly which rows will be selected by this join condition? (**Hint**: do NOT try running it!)

This results not in an antijoin, but in a Cartesian or **CROSS** join of **rental** with **payment**, with just the matching rows left out.



## THE AGGREGATE FUNCTIONS

- \* The *aggregate* functions take a group of rows generated by a **SELECT** statement and calculate a single value.
- \* By default, aggregate functions will generate a single summary row for all values retrieved by the query.
- \* The most commonly used aggregate functions are:
  - **COUNT(\*)** — Total count of all rows selected.
  - **COUNT(*column*)** — Total count of rows selected in which *column* is not **NULL**.
  - **SUM(*column*)** — Sum of values of *column* for selected rows.
  - **AVG(*column*)** — Average value of *column* for selected rows.
  - **MAX(*column*)** — Maximum value of *column* for selected rows.
  - **MIN(*column*)** — Minimum value of *column* for selected rows.

- \* The *column* may also be an expression that yields a value.

```
SELECT SUM(amount) FROM payment
WHERE payment_date
      BETWEEN '2015-01-01' AND '2015-02-01';
```

- \* Aggregate functions are sometimes called *grouping*, *column*, or *set* functions.
- \* Aggregate functions cannot be used in a **WHERE** clause.

- However, you can place the aggregate in a subquery's **SELECT** list.

```
SELECT c.id, first_name, last_name
FROM customer c JOIN rental r
      ON c.id = r.customer_id
WHERE r.id = (SELECT MAX(rental_id) FROM payment);
```

## AGGREGATE FUNCTIONS AND ADVANCED TECHNIQUES

---

```
SELECT COUNT(*) FROM payment;
```

```
SELECT AVG(amount) FROM payment;
```

```
SELECT MIN(replacement_cost) FROM film  
WHERE department_id = 3;
```

```
SELECT MAX(last_name) FROM actor;
```

Multiple aggregates can be used in a single query.

```
SELECT MIN(amount), AVG(amount), MAX(amount)  
FROM payment  
WHERE payment_date > '2016-01-01';
```

## NULLS AND DISTINCT

- \* When you aggregate the values of a specific column, rows with null values for that column are skipped.

```
SELECT COUNT(supervisor_id)
FROM staff;
```

- When all rows have null values for the rows selected, the aggregation will result in null.

- \* When you include the **DISTINCT** keyword with the column to be aggregated, duplicate row values for the column are removed, so that only unique values are aggregated.

```
SELECT COUNT(DISTINCT supervisor_id)
FROM staff;
```

The following query show the sovereign country of protectorate states.

```
SELECT sovereignty FROM country;
```

This query will return a list of sovereign countries that have protectorates.

```
SELECT DISTINCT sovereignty FROM country;
```

This query will return the total number of countries.

```
SELECT COUNT(*) FROM country;
```

This query will return the number of countries that have sovereign countries, excluding NULLs.

```
SELECT COUNT(sovereignty) FROM country;
```

This last query will return the number of unique sovereign countrys that have protectorates, excluding NULLS.

```
SELECT COUNT(DISTINCT sovereignty) FROM country;
```

## GROUPING ROWS

- ✴ Use **GROUP BY** to evaluate an aggregate function over groups of rows.

```
SELECT s.id, a.city, COUNT(*)  
  FROM staff e JOIN store s ON e.store_id = s.id  
                JOIN address a ON s.address_id = a.id  
 GROUP BY s.id, a.city;
```

- ✴ The rows are organized into groups in which the values of the grouping column are equal.

- The aggregate function is then evaluated once for each group.
- The **SELECT** statement returns a single summary row for each group.
- Only aggregate functions and the column(s) used in the **GROUP BY** clause are permitted in the **SELECT** clause.

- ✴ **HAVING** can be used to eliminate group summary rows based on the value of their aggregations.

```
SELECT s.id, a.city, COUNT(*)  
  FROM staff e JOIN store s ON e.store_id = s.id  
                JOIN address a ON s.address_id = a.id  
 GROUP BY s.id, a.city  
HAVING COUNT(*) > 12;
```

```
SELECT e.first_name, e.last_name, COUNT(*)
  FROM staff e JOIN rental r ON e.id = r.staff_id
 GROUP BY e.first_name, e.last_name;
```

The **GROUP BY** clause comes after any **WHERE** clause:

```
SELECT e.first_name, e.last_name, COUNT(*)
  FROM staff e JOIN rental r ON e.id = r.staff_id
 WHERE e.store_id = 2
 GROUP BY e.first_name, e.last_name;
```

**ORDER BY** is always the last clause in a query.

```
SELECT e.first_name, e.last_name, COUNT(*)
  FROM staff e JOIN rental r ON e.id = r.staff_id
 WHERE e.store_id = 2
 GROUP BY e.first_name, e.last_name
 ORDER BY COUNT(*);
```

The database software can do this kind of analysis much more efficiently, over potentially very large amounts of data, than you would be able to do by retrieving the raw data and crunching numbers in a spreadsheet or with program code.

```
SELECT e.first_name, e.last_name, SUM(amount), AVG(amount)
  FROM staff e JOIN rental r ON e.id = r.staff_id
                JOIN payment p ON r.id = p.rental_id
 WHERE e.store_id = 7 AND r.rental_date > '2015-01-01'
 GROUP BY e.first_name, e.last_name
 ORDER BY AVG(amount);
```

```
SELECT s.id, a.city, SUM(amount), AVG(amount)
  FROM payment p JOIN rental r ON p.rental_id = r.id
                JOIN inventory_item ii ON r.inventory_id = ii.id
                JOIN store s ON ii.store_id = s.id
                JOIN address a ON s.address_id = a.id
 GROUP BY s.id, a.city;
```

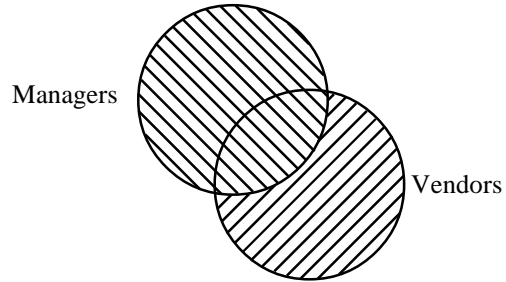
### COMBINING SELECT STATEMENTS

- ✴ Set operators combine the results of two or more queries into one.
  - The **SELECT** lists of all queries must have the same number and data type of expressions in the select list, in the same order.
  - An **ORDER BY** clause can appear after the final query.
- ✴ The **UNION** (*OR*) operator combines the results of both queries into a single resultset, but returns only distinct records.

```
SELECT last_name
  FROM customer
 WHERE store_id = 4
 UNION
SELECT last_name
  FROM staff;
```

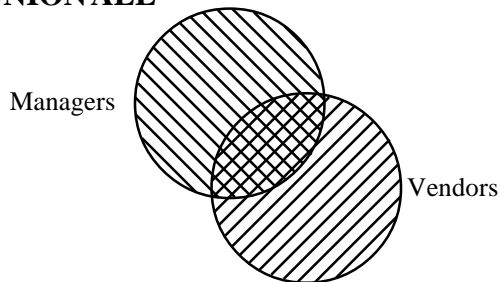
- The **UNION ALL** operator will retain duplicates.
- ✴ Many database products also support the **INTERSECT** (*AND*) operator that returns rows common to both, and the **EXCEPT** or **MINUS** (*AND NOT*) operator returning all rows in the first query not returned by the second (another way of producing an antijoin.)
  - MySQL does not support these operations.

### UNION



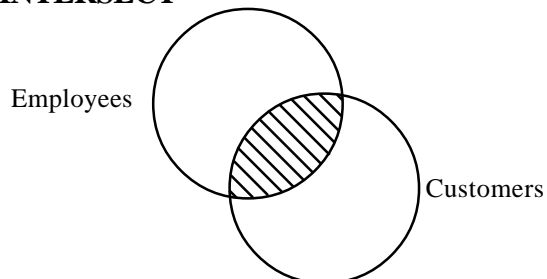
```
SELECT manager_id
  FROM store
UNION
SELECT supervisor_id
  FROM staff;
```

### UNION ALL



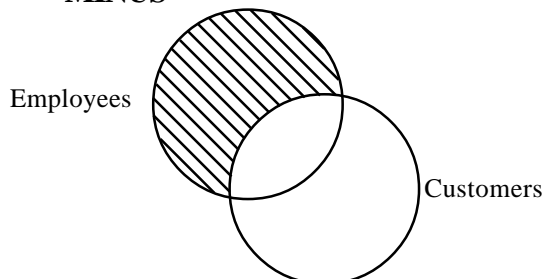
```
SELECT manager_id
  FROM store
UNION ALL
SELECT supervisor_id
  FROM staff;
```

### INTERSECT



```
SELECT id
  FROM staff
INTERSECT
SELECT customer_id
  FROM rental;
```

### MINUS



```
SELECT id
  FROM staff
MINUS
SELECT customer_id
  FROM rental;
```



### LABS

**❶** Write single queries that will:

- a. Retrieve the total number of employees of store #6.
- b. Retrieve the total number of employees of store #6 who live in Los Angeles.
- c. Retrieve the total number of employee usernames.
- d. Report the average length of films.
- e. Report the lowest, average, and highest rental rates of films.
- f. Report the titles and lengths of the films with the highest rental rate.
- g. Report the titles and replacement costs of the films with above average replacement cost.
- h. Report the total amount of payments for rentals made by employee #114.
- i. Report the total amount of payments by customers living in California.

**❷** Write single queries that will:

- a. Report the number of employees for each state.
- b. Report the number of rentals and total payments received for each store.
- c. Report the number of payments and total amount paid for each store that has more than 10000 rentals.
- d. Report the number of employees based on their store's state and whether or not the employee is active.
- e. Report the number of rentals based on film rating, sort by rental count from highest to lowest.





# INDEX

## SYMBOLS

? 78

## A

aggregate function 112

AVG 112

COUNT 112

MAX 112

MIN 112

SUM 112

AND operator 16, 24

antijoin 100, 111, 118

ASC 18

autocommit 80

## B

BETWEEN operator 24

## C

Cartesian product 87

class

DriverManager 34

clause

FROM 10, 108

GROUP BY 116

HAVING 116

ORDER BY 18, 118

USING 86

WHERE 12, 54, 56, 108, 110

close 42, 46

column

alias 10

commit 80

Connection 46

Connection interface 40, 46

createStatement 40

CROSS JOIN 87

cursor 36, 46

## D

Database Management Systems (DBMS) 34, 36

DEFAULT 54

DELETE 56

DESC 18

DISTINCT 10

driver 34

DriverManager class 34

DriverManager.getConnection 38

## E

equijoin 100

execute 72

executeQuery 42

executeUpdate 70

EXISTS operator 110

## F

FROM clause 10, 108

function

aggregate 112

## G

Garbage Collection 46

getResultSet 72

getString 42

getUpdateCount 72

GROUP BY clause 116

## H

HAVING clause 116

## I

IN operator 24, 106

inline view 106

inner join 87

INSERT 52

interface

Connection 40, 46

ResultSet 42, 46

Statement 40, 42, 46

INTERSECT operator 118

IS operator 14

### J

Java Database Connectivity (JDBC) 34, 36  
java.sql 34  
java.sql.Types 44  
JDBC type 44  
join  
    CROSS 87  
    inner 87  
    NATURAL 87  
    outer 96  
    self 98

### L

LIKE operator 26

### M

method  
    execute 72  
    executeUpdate 70  
    getResultSet 72  
    getUpdateCount 72  
    prepareStatement 76  
MINUS operator 118

### N

nested subquery. 106  
next 42  
non-equijoin 100  
NOT operator 16  
NULL value 14, 54

### O

operator  
    AND 16, 24  
    BETWEEN 24  
    EXISTS 110  
    IN 24  
    INTERSECT 118  
    IS 14  
    LIKE 26  
    MINUS 118  
    NOT 16  
    OR 16  
    UNION 118  
    UNION ALL 118  
OR operator 16  
ORDER BY clause 18, 118  
outer join 96

### P

placeholder 78  
PreparedStatement 76, 78  
prepareStatement 76

### Q

query 10

### R

result set 36, 42  
ResultSet 74  
ResultSet interface 42  
    object 42, 46  
ResultSetMetaData 74  
rollback 80

### S

savepoint 58  
scalar subquery 106  
SELECT 42  
SELECT statement 10, 18, 86, 106, 108, 112, 118  
self join 98  
semi-join 110  
SQL 40  
    query 42  
    SELECT 42  
    statements 40  
SQLException 68  
Statement 70, 72  
statement  
    DELETE 56  
    INSERT 52  
    SELECT 10, 18, 86, 106, 108, 112, 118  
    UPDATE 54  
Statement interface 42, 46  
    object 40  
subquery 106

### T

table  
    alias 86  
transaction 80  
try-with-resources 46

**U**

Uniform Resource Locator (URL) 39  
    JDBC connection 38  
UNION operator 118  
UPDATE 54  
USING clause 86

**W**

WHERE clause 12, 54, 56, 108, 110  
wildcard 26









# ***Skill Distillery***

7400 E. Orchard Road, Suite 1450 N  
Greenwood Village, Colorado 80111  
303-302-5234  
[www.SkillDistillery.com](http://www.SkillDistillery.com)