SQL Handout

SQL: Data Definition Language (DDL)

- CREATE TABLE: used to create a table.
- ALTER TABLE: modifies a table after it was created.
- DROP TABLE: removes a table from a database.

SQL: DML Commands

- INSERT: adds new rows to a table.
- UPDATE: modifies one or more attributes.
- DELETE: deletes one or more rows from a table.

Clauses of the SELECT statement:

SELECT

- List the columns (and expressions) that should be returned from the query

FROM

- Indicate the table(s) or view(s) from which data will be obtained

WHERE

- Indicate the conditions under which a row will be included in the result

GROUP BY

- Indicate columns to group the results

HAVING

- Indicate the conditions under which a group will be included

ORDER BY

- Sorts the result according to specified columns

SELECT: SINGLE TABLE QUERIES

SELECTING DATA FROM A TABLE

1) **Choosing all fields (columns)**

```
SELECT *
FROM table_name;
```

```
SELECT *
FROM Customer;
```

2) **Choosing a selected list of fields (columns)**

```
SELECT column_name [,column_name, …]
FROM table_name;

SELECT f_name, l_name, date_of_birth
FROM Customer;
```

- The order in which you list the columns affects the way in which they are presented in the resulting output.
- Items within [ ] are optional.

3) **Temporarily renaming columns in query results**

```
SELECT column_name AS column_heading [,column_name AS
column_heading]
FROM table_name;
```

Example:

```
SELECT f_name as "Name"
FROM Customer;
```

4) **Including calculated columns in the results**

```
SELECT date_due, rate, principal, rate * principal
FROM loan;
```

- If necessary, use parentheses to clarify order of precedence in a computation, as in a * (b+c)

5) **Eliminating duplicate query results with *distinct***

If you use the keyword *distinct* after the keyword SELECT, you will only get unique rows.

Example:

```
SELECT rate,
FROM Loan;
```

VS.

```
SELECT distinct rate
FROM Loan;
```

6) **Selecting rows: the *where* clause**

```
SELECT Select_list
FROM table
WHERE search_conditions;
```

Example:

```
SELECT *
FROM Customer
WHERE f_name = 'Carl';
```

- In SQL, string are delimited by single quotes, as in 'Carl'

Available Search Conditions Operators

- Comparison operators (=, <, >, !=. <>, <= ,>=)
```
SELECT * FROM loan
WHERE principal > 1000;
```

- Ranges (**between** and **not between**; inclusive of the end values)
```
SELECT * FROM loan
WHERE rate BETWEEN 7.5 AND 8.5;
```

- Lists (**in** and **not in**)
```
SELECT *
FROM Customer
WHERE city IN ('Cville', 'Roanoke', 'Lexington');
```

- Character matches (**like** and **not like**)
```
SELECT f_name, l_name
FROM Customer
WHERE l_name LIKE 'Fos%';

SELECT f_name, l_name
FROM Customer
WHERE l_name LIKE '_oster';
```

  - "%" (matches any string of zero or more characters) and "_" (matches any one character). In addition to those, brackets can be used to include either ranges or sets of characters.
- Combinations of previous options using logical operators **and**, **or**, and **not**
```
SELECT f_name, l_name
FROM Customer
WHERE l_name LIKE 'Fos%' AND City NOT IN ('Austin', 'Dallas');
```

SUMMARIZING, GROUPING, AND SORTING QUERY RESULTS

1) **Aggregate functions**

- Types of aggregate functions: **sum, avg, count, count(*), max, min**

```
SELECT SUM (principal) FROM loan;

SELECT AVG (rate) FROM loan;

SELECT MIN(rate), MAX(rate), COUNT(rate)
FROM loan;
```

- The **where** clause can be used to define the set of rows to which the aggregate functions apply

```
SELECT AVG (principal)
FROM loan
WHERE rate > 8.5;
```

- Difference between **count** and **count(*)**: **count** returns the number of non-null values in a specific column, whereas **count(*)** returns the number of rows.

```
SELECT COUNT(*) FROM customers;

SELECT COUNT(city) FROM customers;
```

- The keyword **distinct** can be used with **sum, avg,** and **count** to eliminate duplicate values before the calculations are made. Distinct appears inside the parenthesis and before the column name.

```
SELECT COUNT(DISTINCT city) FROM customers;
```

2) **Using aggregate functions with groupings**

- The **group by** clause can be used in select statements to divide a table into groups and get results (normally aggregates) separately for each group.

```
SELECT rate, AVG(principal)
FROM loan
GROUP BY rate;
```

- The **where** clause can be used in a statement with **group by**. Only those rows that satisfy the condition will be included in the grouping.

```
SELECT rate, AVG(principal)
FROM loan
WHERE principal > 5000
GROUP BY rate;
```

- The types of groups that will be included in the answer set can be limited with the **having** keyword. **Having** sets conditions for groups in the same way **where** sets conditions for individual rows. Aggregate functions can be used in a **having** clause.

```
SELECT rate, AVG(principal)
FROM loan
GROUP BY rate
HAVING AVG(principal) > 5000;
```

3) **Sorting query results with the order by clause**

- An **order by** clause is used to request the results of data retrieval in either ascending (**ASC**, which is the default) or descending (**DESC**) order by one or several (max 16) columns

```
SELECT *
FROM loan
ORDER BY rate;
```

- Multiple sorts are possible

```
SELECT *
FROM customer
ORDER BY l_name, f_name;
```

SELECT: MULTIPLE TABLE QUERIES

SELECTING DATA FROM MULTIPLE TABLES: RELATIONAL JOINS

- Relational joins are a tool for combining data from multiple tables
- They are the characteristic feature of the relational database management system
- A "join" correspond to the intuitive operation of using the values in one column in one table and matching them with the values of another column in another table.
- Joins implement the relations between tables. In the most common case, a join matches a foreign key in one table and the primary key in the other.
- Queries that include multiple joins are possible. These queries "hop" from one table to the next, to the next, to the next.

1) **Joining tables using a foreign key/primary key combination**

```
SELECT l_id, principal, date_due, loan_officer.lo_id,l_name
FROM loan, loan_officer
WHERE loan.lo_id = loan_officer.lo_id;
```

- Table name qualifiers (customer and product in the example above) are used when a column name is not unique. Their format is *tableName.attributeName*

- If the **where** clause is (accidentally) omitted, SQL returns a result that contains the "Cartesian product" of the tables, i.e., all possible combinations of the rows from each of the tables. Thus, if the customer table contained 3 entries and the product table contained 18 entries, the Cartesian product consists of 54 entries. This is very rarely what you intended. Bottom line: remember to include the **where** clause!
- The **where** clause restricts the entries to those where the join condition is true.
- The column set to be displayed can come from either one of the tables, or from both.

2) **Adding elements to the *where* clause**

```
SELECT l_id, principal, date_due, loan_officer.lo_id, l_name
FROM loan, loan_officer
WHERE loan.lo_id = loan_officer.lo_id
AND principal > 10000;
```

- Any combination of logical operators can be used to combine conditions in the **where** clause

3) **Joining three or more tables**

- Joins are not limited to two tables; however, you will seldom see queries with more than 6 or 7 tables joined together. "Normal" is 2-4 tables. Here is an example with 3 tables.

```
SELECT customer.f_name, customer.l_name
FROM loan_officer, loan, customer_in_loan, customer
WHERE loan_officer.l_name = 'Romani'
AND loan_officer.lo_id = loan.lo_id
AND loan.l_id = customer_in_loan.l_id
AND customer_in_loan.c_ssn = customer.c_ssn;
```

- The columns used to join the tables (order number and product number above) may be included in the ***select*** statement but do not have to be.

EXAMPLE 2

```
SELECT   employeeid, last_name, first_name
FROM     employee
WHERE    last_name = 'Smith'
ORDER BY first_name DESC

SELECT employeeid, last_name, first_name
FROM employee
WHERE salary > 41000
ORDER BY last_name, first_name DESC

SELECT *
FROM employee
ORDER BY 2;

SELECT last_name, first_name, salary
FROM employee
WHERE departmentid = 3
ORDER BY salary DESC
```

Relational Operators

```
SELECT employeeid, last_name, first_name
        FROM   employee
        WHERE  salary > 40000


SELECT AVG(salary)
        FROM   employee
        WHERE  departmentid = 3


SELECT *
        FROM   employee
        WHERE  last_name = 'Smith' AND departmentid = 3
```

EXAMPLE OF SUB-QUERY

Give the name of the employee with the highest salary in the company

```
SELECT first_name, last_name, salary
FROM   employee
```

```
WHERE  salary =
  (  SELECT MAX(salary) FROM employee
);
```

Show the employees with the highest salaries in each department:

```
SELECT    first_name, last_name, departmentid, salary
FROM      employee e1
WHERE     salary =
   (
     SELECT max(salary)
     FROM   employee e2
     WHERE  e1.departmentid = e2.departmentid
   )
ORDER BY salary DESC;
```

Show a COUNT of the number of employees in each department

```
SELECT    departmentid, COUNT(employeeid) AS EmployeeCount
FROM      employee
GROUP BY  departmentid
```

Show the current salary and a proposed new salary for each employee:

```
  SELECT first_name, salary AS CurrentSalary,
         (salary * 1.04) AS SalaryWithRaise
FROM   employee
```

Show the Date of Birth and the age of each employee in Department 3. This uses the Now() function which returns the current date and time.

```
SELECT first_name, date_of_birth,
           (Now() - date_of_birth) AS Age
  FROM   employee
WHERE  departmentid = 3
```

```
SELECT first_name, date_of_birth,
           ((Now() - date_of_birth) / 365) AS Age
  FROM   employee
WHERE  departmentid = 3
```

```
SELECT first_name, date_of_birth,
           FORMAT(((Now() - date_of_birth) / 365), "0.0") AS Age
  FROM   employee
WHERE  departmentid = 3
```

Show all employees who have a birthday in August. this uses the MONTH function. Given a date, MONTH(date) returns the month as a number. Similar functions include DAY and YEAR.

List all of the employees working in New York:

```
SELECT  employee.first_name, employee.last_name
FROM    employee, department
WHERE   employee.departmentid = department.departmentid
AND     department.department_location = 'NY';
```

List each employee name and what state (location) they work in. List them in order of location and name:

```
SELECT    employee.last_name, department.department_location
FROM      employee, department
WHERE     employee.departmentid = department.departmentid
ORDER BY  department.department_location, employee.last_name;
```

List each department and all employees that work there. Show the department and location even if no employees work there.

```
SELECT    department.departmentid, department.department_location,
          employee.last_name
FROM      employee RIGHT JOIN department
ON        employee.departmentid = department.departmentid
```

What is the highest paid salary in New York ?

```
SELECT MAX(employee.salary)
FROM   employee, department
WHERE  employee.departmentid = department.departmentid
AND    department.department_location = 'NY';
```

Cartesian Product of the two tables:

```
SELECT *
FROM    employee, department;
```

Using DISTINCT to eliminate duplicates in the result. For example: In which states do our employees work ?

```
SELECT    DISTINCT department_location
FROM      department;
```

Here is a combination of a function and a column alias (the **AS** part of the statement):

```
SELECT first_name, last_name, departmentid,
       salary  AS CurrentSalary,
       (salary * 1.03)  AS ProposedRaise
FROM   employee;
```