Handout 2
Intro to SQL, Single Table Queries

SQL

SQL, originally called SEQueL, for Structured-English Query Language. A de-facto standard language of relational DB model.

It’s both a Data Definition Language (DDL) and a Data Manipulation Language (DML).

Basic commands are create, select, insert, delete, modify.

It’s a declarative language, i.e. the program specifies what the result of computation is rather than how to obtain it (as the procedural languages do).

Case-insensitive except with respect to string values.

CREATE TABLE and INSERT

CREATE operator is a part of DDL:

Example from the textbook:

```
CREATE TABLE CUSTOMER
    (CUSTOMER_NUMBER CHAR(3) PRIMARY KEY,
     LAST CHAR(10) NOT NULL,
     FIRST CHAR(8) NOT NULL,
     STREET CHAR(15),
     CITY CHAR(15),
     STATE CHAR(2),
     ZIP_CODE CHAR(5),
     BALANCE DECIMAL(7,2),
     CREDIT_LIMIT DECIMAL(6,2),
     SLSREP_NUMBER CHAR(2) )
;
```

Here’s how values are inserted into the CUSTOMER table:
INSERT INTO CUSTOMER
VALUES
INSERT INTO CUSTOMER
VALUES

SQL supports the following types:

- Integer.
- Smallint - like integer but covers a smaller range of values (and uses less memory).
- Decimal(p,q) - a decimal number p digits long with q digits after the decimal point.
- Char(n) - actually denotes a string of exactly n characters. Sometimes used to represent numbers that are never used in any arithmetic expression, for instance, Social Security Number.
- Date - dates in the form DD-Mon-YYYY or MM-DD-YYYY depending on the implementation.

Null represents no value; used when the value of the column is unknown or unavailable or nonexistent.

Specification NOT NULL in a column definition means null values cannot be entered in that column. Without that classification a column may be set to NULL unless it’s a primary key.


Does what the name suggests.

Examples:

- the shortest query: selects all (*) records from table CUSTOMER:

  SELECT *
  FROM CUSTOMER
  ;
• a more complex query: display the FIRST and the LAST names and the BALANCE of those CUSTOMERS whose credit limit is greater than $1500 and whose BALANCE is greater than 100 in increasing order of the balance:

```
SELECT First, Last, Balance
FROM CUSTOMER
WHERE (CREDIT_LIMIT > 1500) AND (BALANCE > 100 )
ORDER BY BALANCE ;
```

Summary of the Syntax :

```
SELECT <attribute or functions-applied-to-attributes-list >
FROM <table list>
WHERE <boolean condition>
GROUP BY <grouping attribute(s)>
HAVING <group condition>
ORDER BY <attribute list>
;
```

The SELECT and FROM clauses are mandatory. The others are optional. Now, further details of each of these clauses:

**SELECT in more detail**

SELECT can be used together with the **DISTINCT** operator that eliminates duplicates from the selected rows, e.g.

```
SELECT DISTINCT CITY
FROM CUSTOMER
;
```

**Computed columns** - columns that do not exist in the DB but can be computed using values from existing columns and mathematical operators +, −, /, *, or the string concatenation operator ||.

```
SELECT CUSTOMER_NUMBER, LAST || ',' || FIRST, (CREDIT_LIMIT - BALANCE)
FROM CUSTOMER ;
```

**Aggregate (Group) Functions** SUM, AVG, MAX, MIN, COUNT.

**SUM** returns the total sum of values in the specified <numeric column> from selected rows. Syntax:
• SUM (<numeric column>)
• SUM (DISTINCT <numeric column>)
• SUM (ALL <numeric column>)

Ignores NULL values.

**Example:** compute the total amount to be paid for order number 12498:

```sql
SELECT SUM(QUOTED_PRICE)
FROM ORDER_LINE
WHERE ORDER_NUMBER = 12498
;```

**AVG** like SUM, but returns the average of values in the specified <numeric column> from selected rows.

**MAX** returns the maximum (highest) of the values in the specified <column> from selected rows. The <column> parameter is not limited to numeric columns. **Syntax:**

• MAX (<column>)
• MAX (DISTINCT <column>)
• MAX (ALL <column>)

Ignores NULL values.

**Example:** find the most recent order

```sql
SELECT MAX(ORDER_DATE)
FROM ORDERS
;```

**COUNT** returns a number of rows that contain a value (i.e. are not NULL) in a specified <column>. Has a special form that counts all rows, including those that have NULLS. **Syntax:**

• COUNT (*) the special form that counts *all* rows
• COUNT (<column>). Ignores rows with NULL values in the <column>.
• COUNT (DISTINCT <column>). Ignores NULLS.
• COUNT (ALL <column>). Ignores NULLS.

**Examples:**
1. Count number of rows in table PARTS

   ```sql
   SELECT COUNT (*)
   FROM PARTS;
   ```

2. Count number of distinct warehouses

   ```sql
   SELECT COUNT(DISTINCT WAREHOUSE_NUMBER)
   FROM PART;
   ```

Aggregate functions can be used over the *computed columns* as well as the real ones.

**Practice problem:** Write a query that

1. displays the total amount that can be derived from selling all parts for the price specified in the PART table;

2. displays the average of the credit still available to customers.

**WHERE**

WHERE clause is used to restrict the set of selected rows by specifying a condition that must be true for a row to be selected.

**Syntax:** WHERE `<condition>`,
where the `<condition>` is a boolean expression that must be true for the record to be included in the selected set.

The following operators can be used in the condition:

1. Comparison operators (=, != (or <>), <, >, <=, >=)
2. Comparison to NULL: IS NULL
3. Logical operators AND, OR, NOT
4. Set operators: IN, EXISTS
5. Pattern matching: LIKE

**Comparison Operators:** and IS NULL not limited to comparison of numeric values:

Examples:
• SELECT First, Last, Balance
  FROM CUSTOMER
  WHERE (BALANCE > 150)
  ;

• SELECT First, Last
  FROM CUSTOMER
  WHERE BALANCE IS NULL
  ;

• SELECT PART_DESCRIPTION, UNITS_ON_HAND
  FROM PART
  WHERE PART_DESCRIPTION = 'Bike'
  ;

Logical Operators: AND, OR, NOT - combine the results of simple conditions into compound conditions.

Set membership: IN

Syntax: <column> IN (<setOfValues>).

Examples:

• the set of values can be given as a list, e.g.

  SELECT PART_DESCRIPTION, UNITS_ON_HAND
  FROM PART
  WHERE WAREHOUSE_NUMBER IN ('1','2')
  ;

• or as the result of another query. The example below demonstrates that. It lists descriptions of all parts that are included in the order number 12491.

  SELECT PART_DESCRIPTION
  FROM PART
  WHERE PART_NUMBER IN (SELECT PART_NUMBER
                         FROM ORDER_LINE
                         WHERE ORDER_NUMBER = '12491')
  ;

Pattern matching: LIKE

Syntax: <column> LIKE (<pattern string>), where <pattern string> may use the following wildcard symbols:
• % represents any collection of characters,
• _ represents any single character.

The value in the column is checked against the pattern and if the value matches the pattern, the row is selected, otherwise it is excluded.

Example: list all customers whose address includes the word 'Pine':

```sql
SELECT First, Last, Street
FROM CUSTOMER
WHERE (Street LIKE '%Pine%');
```

**Part 2: Grouping**

1. How to formulate the following query with the operators defined in Part 1?

   *For each warehouse display the total price of parts stored in it.*

   **Syntax:** GROUP BY <grouping columns>

   groups all rows that have the same value in each of the <grouping columns> together. The aggregate function in the SELECT clause is then applied to members of each separate group.

   Example: for each warehouse display the total price of parts stored.

   ```sql
   SELECT WAREHOUSE_NUMBER, SUM(UNIT_PRICE*UNITS_ON_HAND)
   FROM PART
   GROUP BY WAREHOUSE_NUMBER ;
   ```

2. How to formulate the following query:

   *For those warehouses that store more than 2 different parts, display the total price of parts stored in each warehouse.*

   Here, the **HAVING** clause would work. It restricts the selected groups according to a specified condition. **Syntax:** HAVING <group condition>, where the <group condition> is similar to the condition defined for the **WHERE** clause, but can also use aggregate functions that apply to members of each individual group.

   Example:
SELECT WAREHOUSE_NUMBER, SUM UNIT_PRICE*UNITS_ON_HAND FROM PART GROUP BY WAREHOUSE_NUMBER HAVING COUNT(PART_NUMBER) > 2 ;

The HAVING clause for groups serves the same role as the WHERE clause for individual columns: it selects only those groups that satisfy a certain group condition. Since the WHERE clause cannot use aggregate functions, it cannot be used to filter out groups in the way HAVING clause does.

ORDER BY

sorts the rows in increasing or decreasing order of the values in specified columns.