# sql-intro-4

February 27, 2018

Create the tables for this section.

```
In [1]: %load_ext sql
        # Connect to an empty SQLite database
        %sql sqlite://
```

```
Out[1]: 'Connected: None@None'
```

```
In [2]: %%sql
        DROP TABLE IF EXISTS Purchase;
        -- Create tables
        CREATE TABLE Purchase (
            Product  VARCHAR(255),
            Date     DATE,
            Price    FLOAT,
            Quantity INT
        );

        -- Insert tuples
        INSERT INTO Purchase VALUES ('Bagel', '10/21', 1, 20);
        INSERT INTO Purchase VALUES ('Bagel', '10/25', 1.5, 20);
        INSERT INTO Purchase VALUES ('Banana', '10/3', 0.5, 10);
        INSERT INTO Purchase VALUES ('Banana', '10/10', 1, 10);

        SELECT * FROM Purchase;
```

```
Done.
Done.
Done.
1 rows affected.
1 rows affected.
1 rows affected.
Done.
```

```
Out[2]: [('Bagel', '10/21', 1.0, 20),
         ('Bagel', '10/25', 1.5, 20),
         ('Banana', '10/3', 0.5, 10),
         ('Banana', '10/10', 1.0, 10)]
```

## 0.1 Aggregation Operations

SQL support several **aggregation** operations * SUM, COUNT, MIN, MAX, AVG * Except COUNT, all aggregations apply to a single attribute

### 0.1.1 COUNT

Syntax

```
SELECT COUNT(column_name)
FROM   table_name
WHERE  condition;
```

**Example:** Find the number of purchases

| Product | Date | Price | Quantity |
|---|---|---|---|
| Bagel | 10/21 | 1 | 20 |
| Bagel | 10/25 | 1.5 | 20 |
| Banana | 10/3 | 0.5 | 10 |
| Banana | 10/10 | 1 | 10 |

```
In [3]: %%sql
        SELECT COUNT(Product)
        FROM   Purchase;
```

```
Done.
```

```
Out[3]: [(4,)]
```

- Count applies to duplicates, unless otherwise stated
- Same as `COUNT(*)`. Why?

**Example:** Find the number of **different** product purchases

| Product | Date | Price | Quantity |
|---|---|---|---|
| Bagel | 10/21 | 1 | 20 |
| Bagel | 10/25 | 1.5 | 20 |
| Banana | 10/3 | 0.5 | 10 |
| Banana | 10/10 | 1 | 10 |

- Use DISTINCT

```
In [4]: %%sql
        SELECT COUNT(DISTINCT Product)
        FROM   Purchase;
```

```
Done.
```

```
Out[4]: [(2,)]
```

#### 0.1.2 SUM

Syntax

```
SELECT SUM(column_name)
FROM   table_name
WHERE  condition;
```

**Example:** How many units of all products have been purchased?

| Product | Date | Price | Quantity |
|---|---|---|---|
| Bagel | 10/21 | 1 | 20 |
| Bagel | 10/25 | 1.5 | 20 |
| Banana | 10/3 | 0.5 | 10 |
| Banana | 10/10 | 1 | 10 |

```
In [5]: %%sql
        SELECT SUM(Quantity)
        FROM   Purchase;
```

```
Done.
```

```
Out[5]: [(60,)]
```

**Example:** How many Bagels have been purchased?

| Product | Date | Price | Quantity |
|---|---|---|---|
| Bagel | 10/21 | 1 | 20 |
| Bagel | 10/25 | 1.5 | 20 |
| Banana | 10/3 | 0.5 | 10 |
| Banana | 10/10 | 1 | 10 |

```
In [6]: %%sql
        SELECT SUM(Quantity)
        FROM   Purchase
        WHERE  Product = 'Bagel';
```

```
Done.
```

```
Out[6]: [(40,)]
```

## 0.2 AVG

Syntax

```
SELECT AVG(column_name)
FROM   table_name
WHERE  condition;
```

**Example:** What is the average sell price of Bagels?

| Product | Date | Price | Quantity |
|---|---|---|---|
| Bagel | 10/21 | 1 | 20 |
| Bagel | 10/25 | 1.5 | 20 |
| Banana | 10/3 | 0.5 | 10 |
| Banana | 10/10 | 1 | 10 |

```
In [7]: %%sql
        SELECT AVG(Price)
        FROM   Purchase
        WHERE  Product = 'Bagel';
```

```
Done.
```

```
Out[7]: [(1.25,)]
```

### 0.2.1 Simple Aggregations

**Example:** Total earnings from Bagels sold?

| Product | Date | Price | Quantity |
|---|---|---|---|
| Bagel | 10/21 | 1 | 20 |
| Bagel | 10/25 | 1.5 | 20 |
| Banana | 10/3 | 0.5 | 10 |
| Banana | 10/10 | 1 | 10 |

```
In [8]: %%sql
        SELECT SUM(Price * Quantity)
        FROM   Purchase
        WHERE  Product = 'Bagel';
```

```
Done.
```

Out[8]: [(50.0,)]

## 0.3 GROUP BY

Used with aggregate functions (COUNT, MAX, MIN, SUM, AVG) to group the result-set by one or more columns.
Syntax

```
SELECT   column_name(s)
FROM     table_name
WHERE    condition
GROUP BY column_name(s)
[ORDER BY column_name(s)];
```

**Example:** Find total sales after October 1st. per product

| Product | Date | Price | Quantity |
|---|---|---|---|
| Bagel | 10/21 | 1 | 20 |
| Bagel | 10/25 | 1.5 | 20 |
| Banana | 10/3 | 0.5 | 10 |
| Banana | 10/10 | 1 | 10 |

```
In [9]: %%sql
        SELECT   Product, SUM(price * quantity) AS TotalSales
        FROM     Purchase
        WHERE    Date > '10/1'
        GROUP BY Product;
```

```
Done.
```

```
Out[9]: [('Bagel', 50.0), ('Banana', 15.0)]
```

### 0.3.1 Grouping and Aggregation: Semantics of the Query

**1.** Compute the FROM and WHERE clauses

```
In [10]: %%sql
         SELECT *
         FROM   Purchase
         WHERE  Date > '10/1'
```

```
Done.
```

```
Out[10]: [('Bagel', '10/21', 1.0, 20),
          ('Bagel', '10/25', 1.5, 20),
          ('Banana', '10/3', 0.5, 10),
          ('Banana', '10/10', 1.0, 10)]
```

**2.** Group attributes according to GROUP BY

| **Product** | **Date** | **Price** | **Quantity** |
|---|---|---|---|
| Bagel | 10/21/17 | 1 | 20 |
| | 10/25/17 | 1.5 | 20 |
| Banana | 10/03/17 | 0.5 | 10 |
| | 10/10/17 | 1 | 10 |

**Caution:** SQL *only* displays one row if no aggregation function is used

```
In [11]: %%sql
         SELECT   *
         FROM     Purchase
         WHERE    Date > '10/1'
         GROUP BY Product;

Done.


Out[11]: [('Bagel', '10/25', 1.5, 20), ('Banana', '10/10', 1.0, 10)]
```

```
In [12]: %%sql
         SELECT   Product, Count(Product)
         FROM     Purchase
         WHERE    Date > '10/1'
         GROUP BY Product;

Done.


Out[12]: [('Bagel', 2), ('Banana', 2)]
```

**3.** Compute the SELECT clause: grouped attributes and aggregates

| Product | Date | Price | Quantity |
|---|---|---|---|
| Bagel | 10/21 | 1 | 20 |
| Bagel | 10/25 | 1.5 | 20 |
| Banana | 10/3 | 0.5 | 10 |
| Banana | 10/10 | 1 | 10 |

```
In [13]: %%sql  -- Find total sales after '10/1' per product

         SELECT   Product, SUM(price * quantity) AS TotalSales
         FROM     Purchase
         WHERE    Date > '10/1'
         GROUP BY Product;
```

```
Done.
```

```
Out[13]: [('Bagel', 50.0), ('Banana', 15.0)]
```

### 0.3.2 GROUP BY vs Nested Queries

```
SELECT   Product, SUM(price * quantity) AS TotalSales
FROM     Purchase
WHERE    Date > '10/1'
GROUP BY Product;
```

```
In [14]: %%sql
         SELECT DISTINCT x.Product, (SELECT Sum(y.price*y.quantity)
                                     FROM Purchase y
                                     WHERE x.product = y.product
                                           AND y.date > '10/1') AS TotalSales
         FROM  Purchase x
         WHERE x.date > '10/1';
```

```
Done.
```

```
Out[14]: [('Bagel', 50.0), ('Banana', 15.0)]
```

## 0.4 HAVING

- HAVING clauses contain conditions on **aggregates**
- WHERE clauses condition on **individual tuples**

Syntax

```
SELECT   column_name(s)
FROM     table_name
WHERE    condition
GROUP BY column_name(s)
HAVING   condition
[ORDER BY column_name(s)];
```

**Example:** Same query as before, except that we consider only products with more than 30 units sold

```
In [15]: %%sql
         SELECT   Product, SUM(price * quantity) AS TotalSales
         FROM     Purchase
         WHERE    Date > '10/1'
         GROUP BY Product
         HAVING SUM(Quantity) > 30;
```

```
Done.
```

```
Out[15]: [('Bagel', 50.0)]
```

### 0.4.1 Exercise II

An organism that sells tickets for football matches uses a database with the following relational schema:

```
Match(Match_ID, Date, Hour, Stadium_ID, Team_ID)
Team(Team_ID, Name, City)
Stadium(Stadium_ID, Name, Address, Capacity, Team_ID)
Ticket(Ticket_ID, Match_ID, Place_Number, Category, Price)
Sell(Sell_ID, Sell_Date, Ticket_ID, Payment_Method)
```

Write the following query in SQL: > What are the names of the stadiums with largest capacity?

---

# 1 Advanced* Topics

In this section * Relational Division is SQL * Nulls (revisited) * Outer Joins

## 1.1 Relational Division in SQL

- Not supported as a primitive operator, but useful for expressing queries like:

  *"Find suppliers who sell the x parts..."*

  *"Find buyers who bought all products from a given category..."*

- Let $A$ have 2 fields, $x$ and $y$, $B$ have only field $y$

  ```
  A(x, y)
  B(y)
  ```

- $A/B$ contains all $x$ tuples such that for every $y$ tuple in $B$, there is an $xy$ tuple in $A$
- Or: If the set of $y$ values associated with an $x$ value in $A$ contains all $y$ values in $B$, the $x$ value is in $A/B$.

**Classic Option 1**

```
SELECT T1.x
FROM A AS T1
WHERE NOT EXISTS( SELECT T2.y
                  FROM B AS T2
                  EXCEPT
                  SELECT T3.y
                  FROM A AS T3
                  WHERE T3.y=T1.y);
```

**Classic Option 2** (without EXCEPT)

```
SELECT DISTINCT T1.x
FROM A AS T1
WHERE NOT EXISTS(SELECT T2.y
                 FROM B AS T2
                 WHERE NOT EXISTS (SELECT T3.x
                                   FROM A AS T3
                                   WHERE T3.x=T1.x
                                   AND T3.y=T2.y
                                  )
                );
```

**Example:** Find Establishments which sell all products

```
Establishment(eid, ename)
Sells(eid, pname)
Products(pname)
```

Classic Option 2 (without EXCEPT)

```
SELECT DISTINCT E.ename
FROM Establishment AS E
WHERE NOT EXISTS (SELECT p.pname
                  FROM Products3 AS P
                  WHERE NOT EXISTS (SELECT S.eid
                                    FROM Sells AS S
                                    WHERE S.pname=P.pname
                                    AND S.eid=e.eid
                                   )
                );
```

Classic Option 2 (without EXCEPT)

```
SELECT DISTINCT E.ename
FROM Establishment AS E
WHERE NOT EXISTS (SELECT p.pname
                  FROM Products3 AS P
                  WHERE NOT EXISTS (SELECT S.eid
                                    FROM Sells AS S
                                    WHERE S.pname=P.pname
                                    AND S.eid=e.eid
                                   )
                );
```

- Semantics:
- *Establishment* E such that...

  ... there is no *Product* P...

  ...... without a *Sells* tuple showing that E sells P

**Example:** Find Establishments which sell all products

Establishment

| eid | ename |
|---|---|
| 1 | Carrefour |
| 2 | Franprix |
| 3 | Boulangerie |
| 4 | Biocoop |

Sells

| eid | pname |
|---|---|
| 1 | Wine |
| 1 | Bread |
| 1 | Meat |
| 1 | Cheese |
| 2 | Wine |
| 2 | Bread |
| 3 | Bread |
| 4 | Bread |
| 4 | Meat |

Products 1

| pname |
|---|
| Bread |

Products 2

| pname |
|---|
| Bread |
| Meat |

Products 3

| pname |
|---|
| Wine |
| Bread |
| Meat |

In [16]:
```
%%sql
DROP TABLE IF EXISTS Establishment;
-- Create tables
CREATE TABLE Establishment (
    eid INT,
    ename VARCHAR);

DROP TABLE IF EXISTS Sells;
-- Create tables
CREATE TABLE Sells (
    eid INT,
    pname VARCHAR);

DROP TABLE IF EXISTS Products1;
-- Create tables
CREATE TABLE Products1 (
    pname VARCHAR);

DROP TABLE IF EXISTS Products2;
-- Create tables
CREATE TABLE Products2 (
    pname VARCHAR);

DROP TABLE IF EXISTS Products3;
-- Create tables
CREATE TABLE Products3 (
    pname VARCHAR);

-- Insert tuples
INSERT INTO Establishment VALUES (1, 'Carrefour');
INSERT INTO Establishment VALUES (2, 'Franprix');
INSERT INTO Establishment VALUES (3, 'Boulangerie');
INSERT INTO Establishment VALUES (4, 'Biocoop');
INSERT INTO Sells VALUES (1, 'Wine');
INSERT INTO Sells VALUES (1, 'Bread');
INSERT INTO Sells VALUES (1, 'Cheese');
INSERT INTO Sells VALUES (1, 'Meat');
```

```
INSERT INTO Sells VALUES (2, 'Wine');
INSERT INTO Sells VALUES (2, 'Bread');
INSERT INTO Sells VALUES (3, 'Bread');
INSERT INTO Sells VALUES (4, 'Bread');
INSERT INTO Sells VALUES (4, 'Meat');
INSERT INTO Products1 VALUES ('Bread');
INSERT INTO Products2 VALUES ('Bread');
INSERT INTO Products2 VALUES ('Meat');
INSERT INTO Products3 VALUES ('Wine');
INSERT INTO Products3 VALUES ('Bread');
INSERT INTO Products3 VALUES ('Meat');
```

```
Done.
Done.
Done.
Done.
Done.
Done.
Done.
Done.
Done.
Done.
Done.
1 rows affected.
1 rows affected.
1 rows affected.
1 rows affected.
1 rows affected.
1 rows affected.
1 rows affected.
1 rows affected.
1 rows affected.
1 rows affected.
1 rows affected.
1 rows affected.
1 rows affected.
1 rows affected.
1 rows affected.
1 rows affected.
1 rows affected.
1 rows affected.
```

Out[16]: []

In [17]:
```
%%sql
SELECT * FROM Establishment;
```

```
Done.
```

```
Out[17]: [(1, 'Carrefour'), (2, 'Franprix'), (3, 'Boulangerie'), (4, 'Biocoop')]
```

```
In [18]: %%sql
         SELECT * FROM Sells;
```

```
Done.
```

```
Out[18]: [(1, 'Wine'),
          (1, 'Bread'),
          (1, 'Cheese'),
          (1, 'Meat'),
          (2, 'Wine'),
          (2, 'Bread'),
          (3, 'Bread'),
          (4, 'Bread'),
          (4, 'Meat')]
```

```
In [19]: %%sql
         SELECT * FROM Products1
```

```
Done.
```

```
Out[19]: [('Bread',)]
```

**Example:** Find Establishments which sell all products

**Establishment**

| eid | ename |
|---|---|
| 1 | Carrefour |
| 2 | Franprix |
| 3 | Boulangerie |
| 4 | Biocoop |

**Sells**

| eid | pname |
|---|---|
| 1 | Wine |
| 1 | Bread |
| 1 | Meat |
| 1 | Cheese |
| 2 | Wine |
| 2 | Bread |
| 3 | Bread |
| 4 | Bread |
| 4 | Meat |

**Products 1**

| pname |
|---|
| Bread |

**Products 2**

| pname |
|---|
| Bread |
| Meat |

**Products 3**

| pname |
|---|
| Wine |
| Bread |
| Meat |

```
In [20]: %%sql  -- Change bellow to query Products[1,2,3]
         SELECT DISTINCT E.ename
         FROM Establishment AS E
         WHERE NOT EXISTS (SELECT P.pname
                           FROM Products1 AS P
                           WHERE NOT EXISTS (SELECT S.eid
                                             FROM Sells AS S
                                             WHERE S.pname=P.pname
                                             AND S.eid=E.eid
                                            )
                          );
```

```
Done.
```

```
Out[20]: [('Carrefour',), ('Franprix',), ('Boulangerie',), ('Biocoop',)]
```

**Exercise:** Write the same query with EXCEPT (Classic Option 1)

```
Establishment(eid, ename)
Sells(eid, pname)
Products(pname)
```

```
SELECT T1.x
FROM A AS T1
WHERE NOT EXISTS( SELECT T2.y
                  FROM B AS T2
                  EXCEPT
                  SELECT T3.y
                  FROM A AS T3
                  WHERE T3.y=T1.y);
```

**Example:** Find Establishments which sell all products

**Establishment**

| eid | ename |
|---|---|
| 1 | Carrefour |
| 2 | Franprix |
| 3 | Boulangerie |
| 4 | Biocoop |

**Sells**

| eid | pname |
|---|---|
| 1 | Wine |
| 1 | Bread |
| 1 | Meat |
| 1 | Cheese |
| 2 | Wine |
| 2 | Bread |
| 3 | Bread |
| 4 | Bread |
| 4 | Meat |

**Products 1**

| pname |
|---|
| Bread |

**Products 2**

| pname |
|---|
| Bread |
| Meat |

**Products 3**

| pname |
|---|
| Wine |
| Bread |
| Meat |

```
In [21]: %%sql
         -- Write the same query with EXCEPT (Classic Option 1)
```

```
Done.
```

```
Out[21]: []
```

### 1.1.1 Exercise III

An organism that sells tickets for football matches uses a database with the following relational schema:

```
Match(Match_ID, Date, Hour, Stadium_ID, Team_ID)
Team(Team_ID, Name, City)
Stadium(Stadium_ID, Name, Address, Capacity, Team_ID)
Ticket(Ticket_ID, Match_ID, Place_Number, Category, Price)
Sell(Sell_ID, Sell_Date, Ticket_ID, Payment_Method)
```

Write the following query in SQL: > What are the teams that will play at least once in all the stadiums?

### 1.1.2 Exercise IV

An organism that sells tickets for football matches uses a database with the following relational schema:

```
Match(Match_ID, Date, Hour, Stadium_ID, Team_ID)
Team(Team_ID, Name, City)
Stadium(Stadium_ID, Name, Address, Capacity, Team_ID)
Ticket(Ticket_ID, Match_ID, Place_Number, Category, Price)
Sell(Sell_ID, Sell_Date, Ticket_ID, Payment_Method)
```

Write the following query in SQL: > What are the dates and identifiers of matches for which there are no more tickets to sell?

### 1.1.3 Yet another option

"A Simpler (and Better) SQL Approach to Relational Division"
Journal of Information Systems Education, Vol. 13(2)

## 1.2 Null Values

- For *numerical operations*, NULL -> NULL:
- If x is NULL then `4*(3-x)/7` is still NULL
- For *boolean operations*, in SQL there are three values:

```
FALSE   = 0
UNKNOWN = 0.5
TRUE    = 1
```

- If x is NULL then `x = 'Joe'` is UNKNOWN

```
C1 AND C2    = min(C1, C2)
C1 OR C2     = max(C1, C2)
NOT C1       = 1  C1
```

**Example:**

```
SELECT *
FROM   Person
WHERE  (age < 25)
  AND  (height > 6 AND weight > 190);
```

Won't return: - age=20 - height=NULL <-- - weight=200
**Rule in SQL:** include only tuples that yield TRUE (1.0)

**Example:** Unexpected behavior

```
SELECT *
FROM   Person
WHERE  age < 25 OR age >= 25;
```

Some tuples from *Person* are not included
Test for NULL explicitly: * x IS NULL * x IS NOT NULL

```
SELECT *
FROM   Person
WHERE  age < 25 OR age >= 25 OR age IS NULL;
```

Now it includes all tuples in *Person*

### 1.3 Inner Joins + NULLS = Lost data?

- By default, joins in SQL are **inner joins**

  **Example:** Find Products (Name) and the Stores where they are sold.

```
Product(name, category)
Purchase(prodName, store)
```

**Example:** Find Products (Name) and the Stores where they are sold.

```
Product(name, category)
Purchase(prodName, store)
```

Syntax 1

```
SELECT Product.name, Purchase.store
FROM   Product
JOIN   Purchase ON Product.name = Purchase.prodName;
```

Syntax 2

```
SELECT Product.name, Purchase.store
FROM   Product, Purchase
WHERE  Product.name = Purchase.prodName;
```

- Both equivalent, both *inner joins*
- **However:** Products that never sold (with no Purchase tuple) will be lost!

### 1.4 Outer Joins

- An **outer join** returns tuples from the joined relations that don't have a corresponding tuple in the other relations
- i.e. If we join relations A and B on a.X = b.X, and there is an entry in A with X=5, but none in B with X=5 LEFT [OUTER] JOIN will return a tuple **(a, NULL)**

Syntax

```
SELECT column_name(s)
FROM   table1
LEFT OUTER JOIN table2 ON table1.column_name = table2.column_name;
```

In [22]: %%sql

```
-- Create tables
DROP TABLE IF EXISTS Product;
CREATE TABLE Product (
    name VARCHAR(255) PRIMARY KEY,
    category VARCHAR(255)
);

DROP TABLE IF EXISTS Purchase;
CREATE TABLE Purchase(
    prodName varchar(255),
    store varchar(255)
);

-- Insert tuples
INSERT INTO Product VALUES ('Gizmo', 'Gadget');
INSERT INTO Product VALUES ('Camera', 'Photo');
INSERT INTO Product VALUES ('OneClick', 'Photo');

INSERT INTO Purchase VALUES ('Gizmo', 'Wiz');
INSERT INTO Purchase VALUES ('Camera', 'Ritz');
INSERT INTO Purchase VALUES ('Camera', 'Wiz');
```

```
Done.
Done.
Done.
Done.
Done.
1 rows affected.
1 rows affected.
1 rows affected.
1 rows affected.
1 rows affected.
```

Out[22]: []

In [23]: %%sql

```
SELECT *
FROM   Product;
```

```
Done.
```

```
Out[23]: [('Gizmo', 'Gadget'), ('Camera', 'Photo'), ('OneClick', 'Photo')]
```

```
In [24]: %%sql
         SELECT *
         FROM   Purchase;
```

```
Done.
```

```
Out[24]: [('Gizmo', 'Wiz'), ('Camera', 'Ritz'), ('Camera', 'Wiz')]
```

```
In [25]: %%sql
         SELECT Product.name, Purchase.store
         FROM   Product
         LEFT OUTER JOIN Purchase
         ON Product.name = Purchase.prodName;
```

```
Done.
```

```
Out[25]: [('Camera', 'Ritz'), ('Camera', 'Wiz'), ('Gizmo', 'Wiz'), ('OneClick', None)]
```

### 1.5 Outer Joins

- **Left outer join**
- Include the left tuple even if there is no match
- **Right outer join**
- Include the right tuple even if there is no match
- **Full outer join**
- Include both left and right tuples even if there is no match

---

## 2 Summary

- The relational model has rigorously defined query languages that are simple and powerful.
- Several ways of expressing a given query
- A query optimizer should choose the most efficient version.
- SQL is the lingua franca (common language) for accessing relational database systems.
- SQL is a rich language that handles the way data is processed ***declaratively***
- Expresses the logic of a computation without describing its control flow

---

```
In [26]: # Modify the css style
         # from IPython.core.display import HTML
         # def css_styling():
         #     styles = open("./style/custom.css").read()
         #     return HTML(styles)
         # css_styling()
```