Advanced SQL Statements: Joining Data

**Session Objectives:**

- Access and manipulate the data from a Single table.

- Access and join the data from a two or more tables.

# PART 1: Reviewing the basic SQL commands

**Basic statements & clauses in an SQL Query**

appearance of an SQL Query:

SELECT \* <-Select all columns from all tables of interests

FROM student <- Specify which tables you would like to get the data from (for the SELECT statement)

WHERE student.student\_age > 19 <- clause used to filter records based on some condition. The condition is mentioned in the form of an equality

GROUP BY student <- Specify how to categorize the results. It groups data by the specified value and gives a summarized view

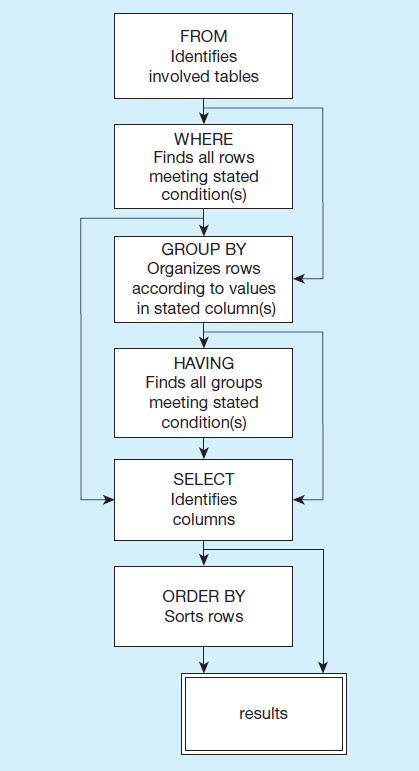
HAVING <- Specifies a filter for the groups.

ORDER BY <- Sorts the results by some specified criteria

; <-finish your queries with a semicolon. This may or may not be necessary depending on the DBMS, but its best that we get used to including it so that our queries are more reusable regardless of the DBMS

**NOTE:** The WHERE statement filters the rows of data you are interested in, and the HAVING statement filters which groups you want are interested in.

While we read and write queries in this order, the database management system (DBMS) will operate on them differently. Specifically, it will start by getting the data from the tables in the FROM statement followed by filtering the rows by the criteria in the WHERE clause. If you are using GROUP BY, the groups will be created in the third step and the undesired groups will be filtered out based on the specifications in the HAVING clause during the fourth step. Finally, the DBMS will select the columns of data you are interested in (5th step) and sort the rows (6th step) to display them to the user. This can be visualized here:



# PART 2: Visualizing Joins

Available tables:

student table:

| student\_ID | first\_name | last\_name | main\_campus | advisor |
| --- | --- | --- | --- | --- |
| 1 | ~ | ~ | ~ | ~ |
| 2 | ~ | ~ | ~ | ~ |
| 3 | ~ | ~ | ~ | ~ |
| 4 | ~ | ~ | ~ | ~ |

course table: [simplified model to maintain 2 tables only, we’ll review joining more tables in the next class]

| course\_ID | course\_name | classroom | student\_ID |
| --- | --- | --- | --- |
| 1 | ~ | ~ | 1 |
| 2 | ~ | ~ | 1 |
| 3 | ~ | ~ | 2 |
| 4 | ~ | ~ | 3 |

| course\_ID | course\_name | classroom | student\_ID | Student\_ID | first | last | main | advisor |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | ~ | ~ | 1 | 1 |  |  |  |  |
| 2 | ~ | ~ | 1 | 1 |  |  |  |  |
| 3 | ~ | ~ | 2 | 2 |  |  |  |  |
| 4 | ~ | ~ | 3 | 3 |  |  |  |  |
| NULL | NULL | NULL | NULL | 4 | ~ | ~ | ~ | ~ |

(most, would be in a class, but some would not. What do we do with them?)

1. **Start with a simple query**:

SELECT \* <-Select all columns from all tables of interests

FROM student <-main table we want to get data from, also known as “Left Table”

INNER JOIN course <- table we want to get additional data on and connect it to the Left table, also known as “Right Table”

ON student.student\_ID = course.student\_ID <- how we want to connect the rows in these tables

; <- finishes the query (we can write another query right after this semicolon symbol)

1. **Alternatively**:

SELECT \*

FROM student

WHERE student.student\_ID = course.student\_ID <- alternatively, we can also use a where clause to identify how we want to join two tables

;

1. **Now let’s refine it**:

SELECT first\_name, last\_name, course\_name <- choose variables of interest

FROM student

INNER JOIN course

ON student.student\_ID = course.student\_ID

;

1. **Standard notation:**

SELECT colum1, column2, …

FROM [table\_of\_interest]

\_\_\_\_\_ [type\_of\_Join] JOIN [table\_we\_want\_to\_connect\_with]

ON firsttable.[variable\_used\_to\_connect\_in\_firsttable] = secondtable.[variable\_used\_to\_connect\_in\_secondtable]

WHERE condition(s)

ORDER BY vale\_to\_order\_by

;

1. Types of Join:
   1. **Inner Join**: Only returns connecting matching rows.

SELECT student.first\_name, student.last\_name, course.course\_name, course.course\_ID

FROM student

INNER JOIN course

ON student.student\_ID = course.student\_ID

;

In our previous example, we are picking information FROM the left table, which is the student table, and matching it to the right table which becomes the course table. This is also valid for the examples below.

Diagram, venn diagram

Description automatically generated

This will only give us all students taking a course with the corresponding course information.

* 1. **Left Join**: Return all connected rows + all unconnected rows from Left table (nulls in missing values).

Diagram, venn diagram

Description automatically generated

SELECT student.first\_name, student.last\_name, course.course\_name, course.course\_ID

FROM student LEFT JOIN course

ON student.student\_ID = course.student\_ID

;

This will give us all students, regardless of whether they are currently taking a course, AND all students taking a course with the corresponding course information.

* 1. **Right Join**: Return all connected rows + all unconnected rows from Right table (nulls in missing values).

Diagram, venn diagram

Description automatically generated

SELECT student.first\_name, student.last\_name, course.course\_name, course.course\_ID

FROM student

RIGHT JOIN course

ON student.student\_ID = course.student\_ID

;

This will give us all courses, regardless of whether they have students currently taking them, AND all courses with students in them and the corresponding information about those students.

* 1. **Full Join**: Return all connected rows + all unconnected rows from Both tables (nulls in missing values).

Diagram, venn diagram

Description automatically generated

SELECT student.first\_name, student.last\_name, course.course\_name, course.course\_ID

FROM student

FULL JOIN course

ON student.student\_ID = course.student\_ID

;

1. **Subqueries**:

We can also use the results of a query as the basis for another query. To exemplify this, we 1) first need to create a query to identify the data we are interested in, 2) we need to provide an alias for the results so we can refer to them in the same way we normally refer to a table, and 3) we need to write the query that will use this results.

Let us start with identifying the courses that student ID 1 has been taking (we will use this new table to match it with the student’s table and get more data on student 1).

SELECT \*

FROM course

WHERE course.student\_ID = 1

;

This query will give us a table with all the course table’s attributes but only those rows with student ID 1.

| course\_ID | course\_name | classroom | student\_ID |
| --- | --- | --- | --- |
| 1 | ~ | ~ | 1 |
| 2 | ~ | ~ | 1 |

Now we can give this new table an alias so that we can refer to it in other parts of the query:

(SELECT \*

FROM course

WHERE course.student\_ID = 1) AS c

;

Or

(SELECT \* FROM course WHERE course.student\_ID = 1) AS c

;

Finally, we can write another query that will use this table and match it with a different table (namely the student table) and join them so that we get all of the data for student 1 attached to the table above.

SELECT \*

FROM (SELECT \* FROM course WHERE course.student\_ID = 1) AS c <- we rename this table as c so we just need to write “c” to refer to it

RIGHT JOIN student AS s <- we rename the student table as s so we just need to write “s” to refer to it

ON c.student\_ID = s.student\_ID <- we match the student\_ID in the newly created table, with the student ID in the student table

ORDER BY c.course\_ID;

;

| course\_ID | course\_name | classroom | student\_ID | first | last | Campus | colleague |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | ~ | ~ | 1 | ~ | ~ | ~ | ~ |
| 2 | ~ | ~ | 1 | ~ | ~ | ~ | ~ |

1. **Self Join**:

SELECT \*

FROM student As s, student As c

INNER JOIN course AS c

ON s.student\_ID = c.collegue\_ID

ORDER BY s.student\_ID

;

1. **Identify those with Null values (IS NULL)**:

SELECT \*

FROM student As s

FULL JOIN course AS c

ON s.student\_ID = c.student\_ID

WHERE s.student\_ID IS NULL OR c.student\_ID IS NULL

;

The table showing from this query will have only results with a NULL student ID in either the student table or the course table.

# PART 3: Getting to Practice

Using the “classic models” database in MySQL, we will create queries to interact with each table individually and with multiple tables at the same time. Please review the following model/schema as a reference for your queries:

Diagram

Description automatically generated

TIPS:

To run queries on the classicmodels database use the following code at the beginning:

* USE classicmodels;

Typical **aggregate functions** include:

| COUNT() | MAX() | MIN() | SUM() | AVG() |
| --- | --- | --- | --- | --- |

**Activity:**

**Q1.** Create a Query to explore the products available. The output should show all attributes from the products table.

| SELECT \*  FROM PRODUCTS; |
| --- |

**Q2.** Expanding on the query you created above, limit your output to show the product code, product name, product line, quantity in stock, buy price, and MSRP.

| SELECT PRODUCTCODE,  PRODUCTNAME,  PRODUCTLINE,  QUANTITYINSTOCK,  BUYPRICE,  MSRP  FROM PRODUCTS; |
| --- |

**Q3.** Now let’s rename attributes to make it more understandable. Give an alias of your choice tot the columns in the output from Q2. For example: SKU, Product\_Name, Line, Available, Cost, and Suggested\_Price.

| SELECT PRODUCTCODE AS pc,  PRODUCTNAME AS p\_name,  PRODUCTLINE AS p\_line,  QUANTITYINSTOCK as available,  BUYPRICE AS cost,  MSRP AS "sale price"  FROM PRODUCTS; |
| --- |

**Q4.** Let’s explore the data through new queries.

1. Write a query that shows what the average MSRP is.

| SELECT AVG(MSRP) AS "average price"  FROM PRODUCTS p; |
| --- |

1. Write a query that shows what the average MSRP is by product Line.

| SELECT p.PRODUCTLINE,  FLOOR(AVG(p.MSRP)) AS "avg msrp"  FROM PRODUCTS p  GROUP BY p.PRODUCTLINE; |
| --- |

1. Write a query that shows what the average buyPrice is per product Line. Add aliases to the columns to better present the output.

| SELECT p.PRODUCTLINE AVG(BUYPRICE) AS bp  FROM PRODUCTS p  GROUP BY PRODUCTLINE |
| --- |

Let’s explore the data through new queries.

**Q5.** Let’s explore the data through new queries.

1. Write a query that shows the product name, product buy price, and MSRP and give each column an appropriate alias.

| SELECT p.PRODUCTNAME,  p.MSRP AS "REVENUE==",  p.BUYPRICE AS "COST",  FROM products p |
| --- |

1. We will now create a calculated attribute. Using the last code (Q5a) and add a new column with the markup for each product (see the formula below). Give it an appropriate alias.



**NOTE:** You can create calculated columns by using parenthesis and typing your formula. Refer to the columns as if they were variables in your formula (that is, selling price would be the MSRP, Cost would be the buyPrice).

| SELECT p.PRODUCTNAME,  p.MSRP AS "REVENUE==",  p.BUYPRICE AS "COST",  FLOOR(((p.msrp - p.BUYPRICE)/p.buyprice)\*100) as "markup%"  FROM products p |
| --- |

Now we will create filters based on the calculated fields.

1. Please add a filter so that we only see products (rows) with an MSRP higher than 100, and a markup higher than 100%.

**Note:** the alias used for the Markup in the SELECT statement cannot be referenced in the WHERE clause since the WHERE clause runs before the SELECT statement (See page 2 of this file to see the order of operations). Therefore, you will need to repeat the formula for MARKUP in the where clause).

| SELECT p.PRODUCTNAME,  p.MSRP AS "REVENUE==",  p.BUYPRICE AS "COST",  FLOOR(((p.msrp - p.BUYPRICE)/p.buyprice)\*100) as "markup%"  FROM products p  WHERE p.MSRP > 100  AND FLOOR(((p.msrp - p.BUYPRICE)/p.buyprice)\*100) > 100 |
| --- |

**Working with Data in Multiple Tables**

For the following questions, we will start using different forms of joins. All of the operations and functions we ran in the first 5 questions can be directly applied to queries composed of multiple tables.

**Q6.** Let’s compare 2 types of joins (Equi-Join vs Inner Join):

1. The simplest type of join is an equi-join. Use an equi-join (joining with equality in WHERE clause) to get shared data between the “products” and “productlines” tables. Give an alias to each table so that we can differentiate between them with ease, and keep all columns.

| SELECT \*  FROM products T1, productlines T2  Where T1.productLine = T2.productLine; |
| --- |

**NOTE:** These 2 tables are connected through the productLine. Note that your output will return 2 copies of the productLine column as a result of the equijoin (1 from each table).

1. Replicate the join in Q6a using an INNER Join instead of the WHERE clause.

| SELECT \*  FROM products T1 INNER JOIN productlines T2  ON T1.productLine = T2.productLine; |
| --- |

**NOTE:** These 2 tables are connected through the productLine. Note that your output will return 1 copy of the productLine column since we used an INNER Join rather than an equi-join.

**Q7.** Joining 3 or more Tables: It is possible to use these same joins repeatedly to join 3 or more tables. Just keep adding AND functions after the WHERE clause if using an equi-join, or use extra lines with INNER JOIN [next\_table] ON [equality]. See example below:

|  |
| --- |

1. Use an equi-join to join the data from the tables productLines, products and order details. Display the “textDescription” from the productlines table, the product’s name and MSRP from the products table, and the order Number and quantity ordered from the orderdetails table.

| SELECT T2.textDescription, T1.productname, T1.MSRP,  T3.orderNumber, T3.QuantityOrdered  FROM products T1, productlines T2, orderdetails T3  Where T1.productLine = T2.productLine  AND T1.productCode = T3.productCode; |
| --- |

1. Use INNER JOINs to join the data from the tables productLines, products and order details. Display the “textDescription” from the productlines table, the product’s name and MSRP from the products table, and the order Number and quantity ordered from the orderdetails table.

| SELECT T2.textDescription, T1.productname, T1.MSRP,  T3.orderNumber, T3.QuantityOrdered  FROM products T1 INNER JOIN productlines T2  ON T1.productLine = T2.productLine  INNER JOIN orderdetails T3  ON T1.productcode = T3.productcode; |
| --- |

**Q8.** Outer Joins. Now let’s practicing including all data from one of the tables included in our queries. We will repeat the same query for each situation just altering the type of outer query.

For each of the following lettered subsections, join the customer table with the orders table. Remember to:

* display all attributes
* use aliases for the tables.
* Record from which table each attribute is coming from when specifying how the tables will be joined.

1. LEFT OUTER Join.

| SELECT \*  FROM CUSTOMERS T1 LEFT OUTER JOIN ORDERS T2  ON T1.customerNumber = T2.customerNumber; |
| --- |

1. RIGHT OUTER Join.

| SELECT \*  FROM CUSTOMERS T1 RIGHT OUTER JOIN ORDERS T2  ON T1.customerNumber = T2.customerNumber; |
| --- |

**Q9.** Now we will create a self-join. In our model some employees report to other employees. Create a query that will allow you to see the first and last names of the employees, the employee’s office code, the code of their manager, their manager’s first and last name, and job title.

| SELECT  em.firstname AS "Employee FirstName"  ,em.lastname AS "Employee LastName"  ,em.officecode AS "Employee OfficeCode"  ,ma.officecode AS "Manager OfficeCode"  ,ma.firstname AS "Manager FirstName"  ,ma.lastname AS "Manager LastName"  ,ma.jobtitle AS "Manager JobTitle"  FROM employees em INNER JOIN employees ma  ON em.Reportsto = ma.employeenumber; |
| --- |

**Q10.** Finally, create a view of the query in Q9 so that we can reuse it later on.

| CREATE VIEW EmployeeManagerView AS  SELECT  em.firstname AS "Employee FirstName"  ,em.lastname AS "Employee LastName"  ,em.officecode AS "Employee OfficeCode"  ,ma.officecode AS "Manager OfficeCode"  ,ma.firstname AS "Manager FirstName"  ,ma.lastname AS "Manager LastName"  ,ma.jobtitle AS "Manager JobTitle"  FROM employees em INNER JOIN employees ma  ON em.Reportsto = ma.employeenumber; |
| --- |