* SELECT
  + A SQL keyword with a simple function: it retrieves information from a database
  + A SELECT statement always starts with the SELECT keyword
  + This is followed by the columns you’d like returned by the query:
    - SELECT column\_one, column\_two
      * This line is called a **clause**
        + A clause is made up of a keyword and a reference to the data
  + To see *all* of the columns in a table use the star (\*)
* FROM
  + Followed by the table name that contains our data
* DISTINCT
  + The DISTINCT keyword will return only the unique values in the field we specific
  + SELECT DISTINCT column\_one, column\_two will return only the unique combinations of column\_one and column\_two
* LIMIT
  + The LIMIT keyword is followed by the number of rows of data you would like returned
  + SELECT \*
  + FROM table\_one LIMIT 3;
* ORDER BY
  + Sorts your data output
  + When you ORDER BY numerical data, it will automatically arrange the values from smallest to largest
  + When you ORDER BY text, it will automatically sort from A-Z
  + You can ORDER BY multiple columns and just like sorting in Excel the sequence of the ORDER BY execution is determined by how the order you write them.
* DESC
  + The DESC keywod changes the output to decending order, from the largest to the smallest value (Z-A)
  + SELECT \*
  + FROM table\_one
  + ORDER BY price DESC
* WHERE
  + WHERE returns records from our data where the condition is true
    - SELECT column\_one, column\_two
    - FROM table\_name
    - WHERE column\_three = ‘Name’;
  + Wildcards
    - % sign is a wildcard and fills in for the letters we don’t know
    - \_ (underscore) is a wildcard character and fills in or one character, whether it’s a letter, number, or symbol
  + AND
    - AND is a logical operator, it is used to combine multiple conditions in a WHERE clause
  + OR
    - OR is a
  + ‘’
    - Text is wrapped with single quotes because it is a special type of data in SQL, known as strings.
* LIKE
* NOT LIKE
* JOIN Clauses
  + INNER JOIN
    - Returns only rows where there is a match in both tables
    - Excludes any records that don’t have a match on both sides

SELECT \*

FROM employees

INNER JOIN salaries

ON employees.id = salaries.id

* + LEFT JOIN
    - Returns **all** the values from the left table, plust matched values from the right table, or “null” in case of no match
  + RIGHT JOIN
    - Returns **all** the values from the right table, plus matched values from the left table, or “null” in case of no match
  + FULL OUTTER JOIN
    - Includes all records, regardless of whether or not there is a match
  + ON – maps the relationship between the two tables
* UNION
  + Combines data into new rows
  + Table UNION
  + Column UNION
  + Combination of the two
  + Couple things to know about the UNION query:
    - You must have the same number of columns of the same data type
    - UNION removes duplicates – UNION ALL allows duplicates

SELECT first\_name

FROM Employees1

UNION

SELECT last\_name

FROM Employees1

UNION

SELECT first\_name

FROM Employees2

UNION

SELECT last\_name

FROM Employees2

* Operators
  + 
* Delimiters
  + 
* AS
  + Creates an alias

SELECT first\_name + ' ' + last\_name AS full\_name,

job\_title,

FROM employees;

* ROUND
  + Takes 2 arguments:
    - The field to round
    - The number of decimals to be returned

SELECT first\_name + ' ' + last\_name AS full\_name,

job\_title,

ROUND(vacation\_days / 21, 2) AS vacation\_used

FROM employees

LIMIT 3;

* Additional SQL mathematical functions can be used with aliases (AS) to modify returning values when the field has a numerical data type. Here’s a list of some of the most common:
  + ABS(): Find the absolute value
  + SIGN(): Return the sign for the numeric expression
    - -1 for negative expressions
    - 0 for zero expressions
    - +1 for positive expressions
    - NULL then the value is also NULL
  + MOD(numerator, divisor): Return the remainder from dividing two numbers (only the integer component)
    - Numerator: the number for which you want to find the remainder after the division is performed
    - Divisor: the number by which you want to divide
  + FLOOR(): Drop the decimal, just keep the integer component
  + CEILING(): Round the number up to the next-closest integer
  + SQRT(): Take the square root of the number
* SUM
  + This function adds together the value for each row of a specified column. SUM is extremely useful for adding together the values of a numberic column
  + Syntax: SELECT SUM(<column\_name>) FROM table\_a
* COUNT
  + This function counts the number of rows returned by a query.
  + COUNT is useful for counting the number of rows returned by your query. This is especially useful when combined with a WHERE clause that gives specific conditions. Then, you are getting a COUNT of occurrences of that specific condition
  + Syntax options:
    - SELECT COUNT (<column\_name>) FROM table\_a
    - SELECT COUNT (\*) FROM table\_a
* COUNT DISTINCT
  + This function counts the number of DISTINCT rows returned by a query, not considering duplicate rows
  + Syntax: SELECT COUNT (DISTINCT <column\_name>) FROM table\_a
* COUNT DISTINCT vs COUNT
  + COUNT DISTINCT is particularly useful for determining the number of unique occurrences of something. Like COUNT, it is especially useful when combined with a WHERE clause
* MIN
  + This function returns the minimum value in a given column. It’s often useful to know the minimum value of a particular column in your data set. It gives you an idea of the lower limit of your data set.
  + Syntax: MIN(<column\_name>)
* MAX
  + This function returns the maximum value in a given column. It’s often useful to know the maximum value of a particular column in your data set. It gives you an idea of the upper limit of your data set
  + Syntax: MAX(<column\_name>)
* AVG
  + This function returns the average of all the values in a given column. It’s often useful to know the average of a particular column in your data set. It gives you a good value for comparison of the rest of the values in your data set
  + Syntax: AVG(<column\_name>)
* NULL

SELECT \*

FROM top\_donors

WHERE first IS NULL;

* + - Which means, "SQL, what are the full records for which first names are missing or unknown?"
  + You can use the NOT operator with NULL to identify non-null rows, like so:

SELECT \*

FROM top\_donors

WHERE first IS NULL;

* + - Which means, "SQL, what are the full records that have first names associated with them?"
  + NULLIF
    - Will take any zeros and change them to null
  + IFNULL
    - The opposite of NULLIF. This function will take a null value and turn it into a zero or a null
    - This can be helpful when adding and subtracting values, as SQL will not allow you to add or subtract null, but will allow you to add and subtract numbers.
* COALESE
  + Similar to IFNULL and NVL, the COALESCE function returns the first non-null expression among the arguments in its parentheses.
  + Syntax: COALESCE ("expression1", "expression2", "another\_expression", ...)
* GROUP BY
  + GROUP BY gathers alike rows together into one summary row
* HAVING
  + HAVING works a lot like WHERE with some key differences.
  + WHERE filters out records based on a condition before the data is acted on by the GROUP BY clause.
  + When working with more advanced SQL it can be unclear when it makes sense to use a WHERE versus a HAVING clause.
  + Though it appears that both clauses do the same thing, they do it in different ways. In fact, their functions complement each other.
    - A WHERE clause is used to filter records from a result. The filter occurs before any groupings are made.
    - A HAVING clause is used to filter values from a group.
  + Or in other words:
    - WHERE is used to filter records before any groupings take place.
    - HAVING is used to filter values after they have been grouped. Only columns or expression in the group can be included in the HAVING clause conditions.
* CASE
  + The IF function is a popular function in Excel — it’s commonly used for creating a column that categorizes another column.
  + You can do something similar in SQL using the CASE statement. It uses the same IF…THEN…ELSE, and ELSEIF logic.
  + The syntax in SQL is slightly different, but the outcome is the same.
  + Using CASE statements allows you to to map decision trees into SQL, and then you can use them for other functions or grouping.
  + Syntax:

SELECT dimensions

CASE

WHEN condition THEN result

WHEN condition THEN result

ELSE

END

AS output name

FROM table;

* CONCAT
  + The CONCAT function combines two fields or exressions together.
  + Syntax: CONCAT(field1, field2, field3…)
  + In SQLite which we will be learning about in the next Unit, || is used instead of CONCAT(). So in SQLite the syntax is field1 || field2 || field3 …
* LEFT / RIGHT
  + A different way of combining the characters in a field is using the left and right functions
  + LEFT: Selects a given number of characters from left side.
    - Syntax: LEFT(field1, length)
  + RIGHT: Selects a given number of characters from right side.
    - Syntax: RIGHT(field1, length)
* SUBSTRING
  + This function allows you to isolate a section of characters within a field to retrieve
  + Syntax: SUBSTRING(field1, starting position, number of characters to retrieve from starting positions)
* LENGTH
  + The length function counts the length of the characters in a field
  + Syntax: LENGTH(field1)
* Left / Right Trim
  + These functions trim blanks from the given side
  + LTRIM: trims all blanks from the left side
    - Syntax: LTRIM(field1)
  + RTRIM: trims all blanks from the right side
    - Syntax: RTRIM(field1)
* TRIM
  + This functions removes specified characters from start of field (leading characters), end of field (trailing characters), or both
  + Syntax:
    - TRIM(leading ‘characters’, from field1)
    - TRIM(trailing ‘characters’, from field1)
    - TRIM(both ‘characters’, from field1)
* REPLACE
  + This function is similar to the Excel function substitute; it allows you to replace a value in a field with another value
  + Syntax: REPLACE(field\_to\_change, content\_to\_replace, new\_content)
* CURRENT\_DATE
  + CURRENT\_DATE brings back the current date from the system
  + Syntax: CURRENT\_DATE
* AGE or DATEDIFF (in MySQL)
  + AGE or DATEDIFF (in MySQL) returns the difference between two dates in days.
  + Syntax:
    - AGE( date1, date2)
    - DATEDIFF( date1, date2) in MySQL
  + DATEDIFF in SQL Server calculates the difference between two dates in units specified in the query.
    - Syntax: DATEDIFF (datepart, expression1, expression2)
* IN
  + Allows you to filer the results of one query by the results of another using the IN operator

SELECT \*

FROM table1

WHERE one\_val IN

(SELECT other\_val

FROM table2);

* WITH AS
  + Syntax: WITH [temp\_table\_name] AS ([subquery]) [regular query]
  + Example: We could use the following query to return all of the rows in “historical\_sales\_table” in which the “sales” column is greater than the average sales from “current\_sales\_table.”

WITH current\_avg\_sales AS (

SELECT AVG(sales) AS avg\_sales FROM current\_sales\_table

)

SELECT \*

FROM historical\_sales\_table WHERE sales > (

SELECT avg\_sales FROM current\_avg\_sales )

* VIEWs
  + A VIEW, in general, is just a shortcut for a SELECT statement.
  + It does not imply that the results are ever run and processed.
  + If you use a VIEW, the results will need to be regenerated each time it is used
  + A temporary table actually stores these results

CREATE VIEW view\_name AS

SELECT column1, column2.....

FROM table\_name

WHERE [condition];

* + You can update a VIEW using the following syntax:

CREATE OR REPLACE VIEW view\_name AS

SELECT column1, column2, ...

FROM table\_name

WHERE condition;

Obviously, where you have a VIEW, you need a way to drop it if it is no longer needed. The syntax for this is very simple:

DROP VIEW view\_name;

* CREATE INDEX
  + The CREATE INDEX command creates a new index. The formal syntax ofr an index on a table looks like this:

CREATE INDEX index\_name ON table\_name;

* + The formal syntax for an index on a column in a table looks like this:

CREATE INDEX index\_name

ON table\_name (column\_name);

* + You can also create a UNIQUE INDEX that doesn’t allow duplicate values:

CREATE UNIQUE INDEX index\_name

ON table\_name (column1, column2, ...);