* Common Aggregate Functions
  + COUNT function: returns the total number of rows that match our search
    - SELECT count(\*) FROM Movies;
  + SUM function: returns the added sum of values for a group of rows
    - SELECT sum(column\_name) FROM table\_name;
  + AVERAGE function: returns the calculated average value for a group of rows
    - SELECT avg(column\_name) FROM table\_name;
  + MAX function: returns the largest value in the group of rows
    - SELECT max(column\_name) FROM table\_name;
  + MIN function: returns the smallest value in a group of rows
    - SELECT min(column\_name) FROM table\_name;
  + sum, avg, max, min only works if columns contains numbers
* Aggregates Within Clauses
  + GROUP BY: group together by column
    - SELECT genre, sum(cost) FROM Movies GROUP BY genre;
  + can use GROUP BY to condense a group of columns into a single row
    - SELECT column\_name, aggregate\_function(column\_name)
    - FROM table\_name
    - GROUP BY column\_name;
  + can filter aggregate functions
  + using the HAVING clause
    - SELECT genre, sum(cost)
    - FROM Movies
    - GROUP BY genre
    - HAVING COUNT(\*) > 1;
  + HAVING clause restricts the group of rows to only those who meet the specified conditions
    - SELECT column\_name, aggregate\_function(column\_name)
    - FROM table\_name
    - WHERE column\_name operator value (optional)
    - GROUP BY column\_name
    - HAVING aggregate\_function(column\_name) operator value;
  + SELECT genre, sum(cost)
  + FROM Movies
  + WHERE cost >= 1000000
  + GROUP BY genre
  + HAVING COUNT(\*) > 1;
  + Use NOT NULL to ensure values cannot be null
    - Column constraint example
    - CREATE TABLE Promotions
    - (
    - id int,
    - name varchar(50) NOT NULL UNIQUE,
    - category varchar(15)
    - );
  + UNIQUE constraint uniquely identifies each field in a table
  + Assigning a name to a constraint can help you easily find it when you choose to alter your constraint
    - Table constraint example
    - CREATE TABLE Promotions
    - (
    - id int,
    - name varchar(50) NOT NULL,
    - category varchar(15),
    - CONSTRAINT unique\_name UNIQUE(name)
    - );
    - unique\_name is the assigned name, name is the column
  + except for NOT NULL, every column constraint can also be written as a table constraint
  + can also not allow insert if it has the same name and category
    - ….
    - CONSTRAINT unique\_name UNIQUE (name, category)
    - …
  + As the primary key, the id column needs to uniquely identify every row in this table
    - Adding a primary key constraint means that column cannot be null and must be unqiue
  + Primary key vs NOT NULL + UNIQUE
    - Primary key can only be defined once per table
    - NOT NULL + UNIQUE can be used multiple times per table
  + Foreign Key: references the primary key column of another table
    - Naming convention
    - Singular version of the table you’re referencing + an underscore + column name
  + The REFERENCES keyword can be used to make a FOREIGN KEY constraint
    - CREATE TABLE Promotions
    - (
    - id int PRIMARY KEY,
    - movie\_id int REFERENCES movies(id),
    - name varchar(50),
    - category varchar(50)
    - );
  + The table being referenced must be created first
  + Can also use the table constraint syntax
    - ….
    - FOREIGN KEY (movie\_id) REFERENCES movies
    - …
  + Orphan records
    - Child records with a foreign key to a parent record that has been deleted
  + Use foreign key constraint helps avoid orphan records
  + Tables must be dropped in the correct order to avoid orphan records
  + Can use CHECK constraint to validate the value that can be placed in a column
    - ….
    - duration int CHECK (duration > 0)
  + Normalization is the process of reducing duplication in database tables
    - First Rule: table must not contain repeating groups of data in 1 column
    - Second Rule: tables must not contain redundancy(unnecessary repeating information)
  + Create a join table
    - Name convention: table1\_table2
    - Use foreign keys
    - Join table don’t follow normalization
  + INNER JOIN: show where both tables have matching values
    - SELECT \* FROM Movies INNER JOIN Reviews
    - ON Movies.id = Reviews.movie\_id
  + Can select specific columns
    - SELECT Movies.title, Reviews.review
    - FROM Movies
    - INNER JOIN Reviews
    - ON Movies.id = Reviews.movie\_id
  + Aliases give the columns new temporary names
    - SELECT Movies.titles AS films, Reviews.review AS reviews
    - …
    - Or you can just remove as
    - SELECT Movies.title films, Reviews.review reviews
  + Use “ “ if you want alias with more than 1 word
  + LEFT OUTER JOIN
    - Display all rows from table 1
    - And join them with matching rows in table 2
    - All firm information and only the reviews associated with them
    - SELECT \* FROM MOVIES
    - LEFT OUTER JOIN Review
    - On Movies.id=Reviews.movie\_id
  + RIGHT OUTER JOIN
    - Display all rows from table 2
    - And join them with matching rows in table 2
    - List all reviews and only firms that are associated
  + Using subqueries
    - SELECT SUM(sales) FROM Movies
    - WHERE id IN
    - (
    - SELECT movie\_id FROM Promotions
    - WHERE category = ‘Non-cash’);
  + Can sometimes find the same results using a JOIN
    - SELECT SUM(m.sales) FROM Movies m
    - INNER JOIN Promotions p
    - ON m.id = p.movie\_id
    - WHERE p.category = ‘Non-cash’;
  + Subquery syntax
    - WHERE <field> IN(<subquery>) to filter rows that have a matching id
    - WHERE <field> NOT IN(<subquery>) to filter rows that don’t have a matching id
  + Aggregate functions are not allowed in WHERE
    - Use subqueries instead
    - SELECT \* FROM Movies WHERE durations >
    - (SELECT AVG(duration) FROM Movies);