# Table of Contents

# SQL Basics

### What This Book is About

This book will cover the basics of SQL, and should help introduce beginners to SQL concepts. It is also a good documentation source when forgetting how the basic SQL syntax works.

### Contributing

This book is open source, so please feel free to visit the github repository to help contribute! If you notice any typos or errors in this book, you can fork the repo and open a pull request with the new and improved changes, or simply just open a new issue. Any contribution is greatly appreciated and it will help make this book better for those reading it after you!

### License

### Here is the example database that will be used throughout the book:

### Movies

| mID | Title | Year | Director |
|---|---|---|---|
| 101 | Top Gun | 1986 | Tony Scott |
| 102 | Titanic | 1997 | James Cameron |
| 103 | The Lion King | 1994 | Rob Minkoff |
| 104 | Gravity | 2013 | Alfonso Cuaron |
| 105 | Harry Potter | 2001 | `<null>` |
| 106 | Cast Away | 2000 | Robert Zemeckis |
| 107 | Spider Man | 2002 | Sam Raimi |
| 108 | The Godfather | 1972 | Francis Coppola |

### User

| uID | Name |
|---|---|
| 201 | James Dean |
| 202 | Chris Anderson |
| 203 | Ashley Burley |
| 204 | Ralph Truman |
| 205 | Gordon Maximus |
| 206 | Sarah Rodgriguez |
| 207 | Darrel Sherman |
| 208 | Lisa Jackson |

### Review

| uID | mID | Rating | ratingDate |
|---|---|---|---|
| 201 | 101 | 2 | 2014-03-09 |
| 201 | 101 | 4 | 2014-03-02 |
| 202 | 104 | 4 | `<null>` |
| 203 | 107 | 2 | 2014-03-24 |
| 204 | 103 | 4 | 2014-03-17 |
| 204 | 104 | 2 | 2014-03-13 |
| 205 | 108 | 3 | 2014-03-24 |
| 206 | 102 | 3 | 2014-03-02 |
| 207 | 104 | 5 | `<null>` |
| 207 | 106 | 4 | 2014-03-07 |
| 207 | 102 | 5 | 2014-03-26 |
| 208 | 105 | 2 | 2014-03-13 |

# Basic Syntax

## What is Structured Query Language?

Structured Query Language, or SQL, is a special-purpose programming language used to manage data within a relational database mangement system (RDMS).

You will find that there are multiple ways to write the same query in SQL, and some ways are better than others.

## Two Parts of SQL

**Data Definition Language (DDL):** DDL includes commands to create a table, to drop a table, or create and drop other aspects of a database.

**Data Manipulation Language (DML):** DML includes commands that are used to query and modify a database. It includes the select statement for querying the database, and the insert, update, and delete statements, all for modifying the database.

# Basic Select Statements

## The Three Clauses:

The select statement has three clauses: the `FROM` clause, the `WHERE` clause, and the `SELECT` clause. The basic concept is that the `FROM` clause identifies the relation that you want to query over, the 'WHERE' condition is used to combine the relations and filter the relations, and the `SELECT` tells you what to return.

The syntax looks like this:

```
SELECT A1, A2, . . . , A(n)
FROM R1, R2, . . . , R(m)
WHERE <condition>;
```

Since relational query languages are compositional, when you run a query over relations, you get a relation as a result. Thus, the result of the above select statement is a relation, but it doesn't have a name. The schema of that relation is the set of attributes that are returned.

## A Basic Query:

Assume we have a database with three tables:

1. A `MOVIE` table that has four columns labeled `mID` (movie ID), `Title` , `Year` , and `Director` .
2. A `User` table that has two columns labeled `uID` (User ID), and `Name` .
3. Lastly, a `Review` table that has four columns labeled `uID` , `mID` , `Star` , and `ratingDate` .

*(This sample database can be viewed in the Introduction chapter of this book.)*

We’re going to do a basic query that finds the `Title` , and `Year` of movies that were created after the year 2000. The `SELECT` tells us what we want to get out of the query, the `FROM` tells us our table name, and the `WHERE` gives us the filtering condition.

The query would look like this:

```
SELECT Title, Year
FROM Movie
WHERE Year > 2000;
```

We would then get a table back that would have two columns labeled `Title` , and `Year` . It would then display all the movies that were created after the year 2000.

The resulting movies would include: *Gravity*, *Harry Potter*, *Cast Away*, and *Spiderman*.

## Combing Two Relations:

Now let’s create a query that combines two relations, such as finding movie titles, mID's and the rating that the movie recieved. We’re now involving the `Movie` table, and the `Review` table.

Combining relations looks like this:

```
SELECT Movie.mID, Title, Rating
FROM Movie, Review
WHERE  Movie.mID = Review.mID;
```

The condition above is called a **join** condition and is saying that we want to combine movies with review statistics that have the same `mID` . We would then get a table as a result with three columns labeled `mID` , `Title` , and `Rating` . It would then display all the movies with their `mID` and their `Rating` .

## Combing Two Relations w/ a Condition:

The next query is going to find the `Title` , `mID` and `Rating` of movies that were created before the year 2000, and `Rating` is greater than 2.

It would look like this:

```
SELECT Movie.mID, Title, Rating
FROM Movie, Review
WHERE Movie.mID = Review.mID
    and Rating > 2 and Year < 2000;
```

So in this case, we are looking for `mID`, `Title`, and `Rating`. We are looking inside the `Movie` and `Review` tables, and we have a join condition making sure that the query knows the `mID` in the `Movie` table is the same `mID` in the `Review` table. We are filtering the results based on the year the movie was created, and the rating it recieved. We would then get a table with the results of the query. The results would include all movies that were created before the year 2000, with a rating greater than 2.

The resulting movies would be: *Top Gun*, *Titanic*, *The Lion King*, and *The Godfather*.

## Combining Three Relations:

This time we are going to combine all three relations, and we're going to get a table with the results of every `mID`, `Title`, `Year`, User `Name`, and `Rating`.

It would look like this:

```
SELECT Movie.mID, Title, Year, Name, User.uID, Rating
FROM Movie, User, Review
WHERE Movie.mID = Review.mID and User.uID = Review.uID;
```

Notice how in the `SELECT` and `WHERE` statements, we specify which table we want to pull some of the attributes out of. Since there is more than one table with `mID` and `uID`, we need to specify which table we want to pull it out of. It doesn't matter which table we specify, but if we don't specify we will get an error because the computer doesn't know which table to pull from.

## Sorting Table Results:

SQL by default does not order table results in any particular order. However, if we specify a specific order that we want, we can get results sorted by a specific attribute, or set of attributes. Say we want to sort all of our movies by descending `Rating`. In order to do this, we need to add an additional clause called the `ORDER BY` clause. If we want to get a descending order, we write what we want to search for and then use the keyword `DESC`.

It would look like this:

```
SELECT  Movie.mID, Title, Year, Name, User.uID, Rating
FROM Movie, User, Review
WHERE Movie.mID = Review.mID and User.uID = Review.uID
    ORDER BY Rating DESC;
```

If we wanted to have it sort by additional attributes, we would just put a comma after `DESC`, and add another attribute. However, SQL defaults to ascending order, so you need to specify which way you prefer for any additional attributes that you add.

## Doing Arithmetic within Select Statements:

While doing a `SELECT` statement, SQL allows for doing arithmetic operations. Say we want to find all the movie's attributes, but add to it a scaled `Rating`. Where we are going to scale the rating by 10 to get ratings that are in the teens.

The query would look like this:

```
SELECT Movie.mID, Title, Rating, Director, Rating + 10
FROM Movie, Review;
```

We would then get a table with all of the above attributes, and then an additional column that shows the movie's `Rating` after being scaled by 10. However, we will get a column labeled `Rating + 10`, but we want to change it to a different particular label.

We would just use the AS clause like so:

```
SELECT Movie.mID, Title, Rating, Director, Rating + 10 AS ScaledRating
FROM Movie, Review;
```

We would then have a column labeled `ScaledRating` instead of `Rating + 10`.

# Table Variables & Set Operators

## What are they?

Take a look at the code block below:

```
SELECT A1, A2, . . . , A(n)
FROM R1, R2, . . . , R(m)
WHERE <condition>;
```

All of the variables inside the `FROM` clause are called table variables. They help with making the query more readable, and they rename relations within the `FROM` clause when we have more than one instance of a relation.

The second construct is called a set operator. A few examples of them are: the `UNION` operator, the `INTERSECT` operator, and the `EXCEPT` operator.

## Adding Table Variables for Readability

Let's assume we want to make a query that outputs all the movies along with their movie ID, title, User ID, User name, and rating.

Our query would look like this:

```
SELECT Movie.mID, Title, User.uID, Name, Rating
FROM Movie, User, Review
WHERE Movie.mID = Review.mID and User.uID = Review.uID;
```

We would then get our expected table output. However, to make the query a little more readable, we can place variables inside the `FROM` clause, and replace all the table names with just the table variable.

Our query would look like this:

```
SELECT M.mID, Title, U.uID, Name, Rating
FROM Movie M, User U, Review R
WHERE M.mID = R.mID and U.uID = R.uID;
```

Notice how we added `M` after `Movie` inside the `FROM` clause. That is called adding a table variable, then where ever we used `Movie` in our entire select statement, we can just replace with `M`. We also did the same thing for `User U` and `Review R`.

## Adding Table Variables for Multiple Instances

In this next query, we want to find all movies that have the exact same rating. In order to do that, we need to have two instances of `Review`. We will call the first instance `R1` and the second instance `R2`. We also need to include `Movie` in the `FROM` clause to get the movie `Title`.

Our query would look like this:

```
SELECT DISTINCT Movie.mID, Title, R1.Rating
FROM Movie M, Review R1, Review R2
WHERE R1.Rating = R2.Rating and R1.mID = M.mID and R1.mID <> R2.mID and R1.uID <> R2.u
ID;
```

We first added the two table variables: `R1` and `R2` to separate each instance. We are looking for the `mID`'s, `Title`'s, and `Rating`'s of each movie. We specify `DISTINCT` in the `WHERE` clause to remove duplicates. We then place a condition in the `WHERE` clause to specify that we want to output movies with the same rating. We include the join condition `R1.mID = M.mID` to make sure the movies are the same in each relation. We then specify two final clauses: `and R1.mID <> R2.mID and R1.uID <> R2.uID;`. `R1.mID <> R2.mID` tells the computer that the `R1.mID` is different from `R2.mID`, or that movie 1 needs to be different from movie 2. If we didn't specify this clause, we would get an output of movies that equal themselves, because movie 1 would have the same `Rating` as itself, thus satisfying the condition. Likewise, `R1.uID <> R2.uID` makes sure we don't get back two instances of a review by the same reviewer.

## The Union Operator

The `UNION` operator allows us to create queries that will output a list of elements that come from multiple tables. Previously we could only separate these elements into different columns. However, by using the `UNION` operator, we can get elements from different tables listed into a single column together.

For example, if we wanted to get a single list of all the movie titles and reviewer names, we would create a query that looks like this:

```
SELECT Title FROM Movie
UNION
SELECT Name FROM User;
```

We would then get a table with only **one** column, and it would list each movie title and each reviewer name.

## Specifying a Column Name

By default, SQL would pick label the column either `Title` or `Name`. If you wanted to specify a label for the column, you would use the `AS` operator.

It would look like this:

```
SELECT Title AS list FROM Movie
UNION
SELECT Name AS list FROM User;
```

Notice how we placed `AS list` in each select clause, and this tells SQL to name the column `list`.

## The Intersect Operator

The `INTERSECT` operator takes away the necessity to specify a joint relation. It automatically knows that each select statement in the query is for the same movie.

If we wanted to create a query that searched for movies that were created before the year 2000, and had a `mID` of less than 105, we could use the `INTERSECT` operator.

Our query would look like this:

```
SELECT Title FROM Movie WHERE Year < 2000
INTERSECT
SELECT Title FROM Movie WHERE mID < 105;
```

We would then get a table in return with all the movies that were created before the year 2000, and had a `mID` less than 105.

The resulting movies would be: *Top Gun*, *Titanic*, and *The Lion King*.

## The Except Operator

The `EXCEPT` operator does exactly the opposite of the `INTERSECT` operator.

Lets create a query that looks for movies that were created before the year 2000, but **do not** have a `mID` less than 105.

Our query would look like this:

```
SELECT Title FROM Movie WHERE Year < 2000
EXCEPT
SELECT Title FROM Movie WHERE mID < 105;
```

In this case, the `EXCEPT` operator tells SQL to look for movies that were made before the year 2000, and then take away the movies with a `mID` less than 105. This then leaves us with a result of movies which were created before 2000, but have a `mID` greater than or equal to 105.

The resulting movie would be: *The Godfather*.

# Subqueries in the Where Clause

## Basic Syntax

Just like before, our queries will contain a `SELECT` clause, a `FROM` clause, and a `WHERE` clause specifying a condition. However, we now are adding in the ability to nest a `SELECT` clause inside the `WHERE` clause, thus creating a subquery.

Subqueries can be very powerful when trying to eliminate duplicates, and is often more efficient than using joining relations.

## Creating a Basic Subquery

Let's create a query that looks for a movie's ID, title, and director, but only if it has a rating above 4.

We can create a sub-query like so:

```
SELECT DISTINCT Movie.mID, Title, Director
FROM Movie, Review
WHERE Movie.mID in (SELECT mID FROM Review WHERE Rating > 4);
```

We could easily do this query without implementing a subquery by joining the Movie relation with the Review relation. However, this is just to show how a subquery would be performed.

We would then get the movies: *Titanic*, and *Gravity*.

## Slightly More Complex Subqueries

Lets create a query that retrieves the `Title` of all movies which have a `Rating` less than 3, and have a `mID` greater than 103.

Our query would look like this:

```
SELECT Title
FROM Movie
WHERE mID in (SELECT mID FROM Review WHERE Rating < 3)
  AND mID NOT IN (SELECT mID FROM Review WHERE mID < 103);
```

Our outside query returns the `Title` of all movies whose `mID` is in the first subquery, but not in the second subquery. Our first subquery looks for all `mID` 's whose `Rating` is less than 3, and the second subquery looks for all `mID` s that are greater than 103.

The output movies would be: *Spiderman*, *Gravity*, and *Harry Potter*.

# Using Operators in a Subquery

## The Exist Operator & Correlated References

The `EXISTS` operator checks whether a subquery is empty or not, instead of checking whether values are in the subquery.

A correlated reference is where you use a value inside a subquery, that comes from *outside* that subquery.

Lets look at an example:

```
SELECT mID, Rating
FROM Review R1
WHERE EXISTS (SELECT * FROM Review R2
              WHERE R1.Rating = R2.Rating and R1.mID <> R2.mID);
```

This query will return all movies that have the same rating. First we're going to take the `mID` 's from `R1` . Then we're creating a new relation called `R2` . For each movie we're going to check if there is another `mID` , where the `Rating` in `R2` is the same as the `Rating` in `R1` . We then say that each `mID` should be different, and not equal to itself.

We use a correlated reference to use an outside variable *inside* a subquery.

## Looking for a Largest Value

Assume that you wanted to look for the largest value of some element. In this case, we want to find the movie that was most recently created. Thus, the movie's `Year` would be the largest.

We could write a query that looks like this:

```
SELECT Title, Year
FROM Movie M1
WHERE NOT EXISTS (SELECT * FROM Movie M2
                  WHERE M1.Year < M2.Year);
```

This query says that we are going to find all movies where there does **not exist** another movie whose `Year` is greater than the first movie. This would be a form of query that we could write whenever looking for the greatest value of some-sort.

The resulting movie would be: *Gravity*.

## The All Operator

The `ALL` keyword tells us that instead of checking whether a value is in or not in the result of a subquery, we're going to check if the value has a certain relationship with `ALL` the results of a subquery.

Lets create a query that checks to see if the `Rating` of a movie is greater than or equal to `ALL` elements of the subquery which returns all the `Ratings` of each movie.

It would look like this:

```
SELECT mID, Rating
FROM Review
WHERE Rating >= all (SELECT Rating FROM Review);
```

We would then get an output table of all the movie's with a `Rating` of 5, since there is no single movie with a greater `Rating` than every other movie.

The output table would include the movies: *Gravity*, and *Titanic*.

## The Any Operator

The `ANY` keyword performs very similar to the `ALL` keyword, except instead of having to satisfy a condition with `ALL` of the elements of a set, it only has to satisfy a condition with at least one element of a set.

Lets create a query that finds all movies that have a `Year` that is not the smallest `Year` value. In other words, we are looking for movies whose `Year` is greater than `ANY` other movie `Year`.

Our query would look like this:

```
SELECT Title, Year
FROM Movie
WHERE Year > ANY (SELECT Year FROM Movie);
```

In the above example query, a movie will be returned if there is some other movie whose `Year` is less than this movie. We then get a resulting table with all the movies that do not have the least `Year` value. Thus, we would get every movie except for *The Godfather*, because it has the smallest `Year` value.

## Conclusion on Operators

The `ANY` and `ALL` operators are very convienient when creating queries, however, they are not vital to creating a query. We can always write a query that would normally use the `ANY` or `ALL` keywords, by using the `EXISTS` or `NOT EXISTS` operators.

# Subqueries in the From & Select Clause

## What They Are Used For

- When using subqueries in the `WHERE` clause, the subquery generates a set of elements that will be used in a comparison.
- When using subqueries in the `FROM` clause, the subquery generates a new table that will be used for the rest of the query.
- When using subqueries in the `SELECT` clause, the subquery produces a sub-select expression that returns a single value.

## Using a Subquery in the From Clause

Lets create a query that scales all the movie ratings depending on which year they were produced. For example, a movie produced in 2012 with a rating of 4 would be higher ranked than a movie produced in 1990 with a rating of 4. Lets assume that throughout the years in the movie industry, the CGI and film effects get better, thus producing a better movie. This technology wouldn't be available in the previous years, so older movie's ratings won't hold the same value as present day movies.

Our query would look like this:

```
SELECT *
FROM (SELECT M.mID, Title, Year, Rating, Rating*(Year/1000.0) as scaledRating
    FROM Movie M, Review
    WHERE M.mID = Review.mID) sR
WHERE abs(sR.scaledRating) > 8;
```

This query is saying that it will `SELECT` all attributes `FROM` a table produced by a subquery. This subquery is going to produced a table that has the `mID` , `Title` , `Year` , `Rating` , then a scaled rating with a column name as `scaledRating` . The results of this subquery has the table variable name of `sR` . Then in the main query, we only want to output the movies where the absolute value ( `abs()` ) of the `scaledRating` is greater than 8.

We would then get output the following movies: *Titanic*, and *Gravity*.

## Using Subqueries in the Select Clause

Lets create a query that lists all the users and pairs them with the highest rating that they have given.

The query would look like so:

```
SELECT uID, Name
(SELECT DISTINCT Rating
 FROM User, Review
 WHERE User.uID = Review.uID
    and Rating >= ALL
        (SELECT Rating
         FROM User, Review
         WHERE User.uID = Review.uID)) as hRating
FROM User;
```

This query says that we are going to find `uID` and that user's `Name` from the `User` table, and then it runs a subquery. The subquery looks for a `Rating` inside the table `Review`. We make sure to join the `uID` from the `User` table to the `uID` in the `Review` table. We then choose the largest `Rating` among `ALL` the `Rating`s that are associated with that user. Lastly, we give the subquery result's a variable name, which takes the name of `hRating`, which stands for 'highest rating'.

In other words, we would get a resutling table with each user's `uID` and `Name`. Then, it would have a column labeled `hRating` which has the highest rating that each individual user has ever done.

Our query would ouput these results:

| uID | Name | hRating |
|---|---|---|
| 201 | James Dean | 4 |
| 202 | Chris Anderson | 4 |
| 203 | Ashley Burley | 2 |
| 204 | Ralph Truman | 4 |
| 205 | Gordon Maximus | 3 |
| 206 | Sarah Rodgriguez | 3 |
| 207 | Darrel Sherman | 5 |
| 208 | Lisa Jackson | 2 |

## Important Note on Subqueries in the Select Clause

When implenting a subquery in the `SELECT` clause, it is crucial that the subquery only returns exactly one value, because the result of that subquery is being used to only fill in one cell of the parent query.

# The 'Join' Family Operators

## The Basics

In our select statements, we seperate tables in the `FROM` clause by commas, and that is implicitly a cross product of those tables.

Like so:

```
SELECT A1, A2, . . . , A(n)
FROM R1, R2, . . . , R(m)
WHERE <condition>;
```

However, there is a way to explicitly join two or more tables using one of the `JOIN` operators. There are a few different types, and they are listed below:

1. The `INNER JOIN` on *condition*
    - This kind of join operator takes the cross product of the tables and then applies a condition, and only taking the cross product elements that satisfy the condition given. It then eliminates all duplicate columns that are created.
2. The `NATURAL JOIN`
    - The `NATURAL JOIN` operator equates all columns with the same name in the tables that are being joined. It requires the values in the columns to be the same in order to keep the elements in the cross product. This type of join also eliminates any duplicate columns that are created.
3. The `INNER JOIN USING(attrs)`
    - This again is an `INNER JOIN`, however this type of join takes a special clause called `USING` and listing attributes. This is sort of like the `NATURAL JOIN`, except you specifically state the attributes that you want to be equated.
4. The `OUTER JOIN`
    - There are multiple forms of this kind of join operator. There is the `LEFT OUTER JOIN`, the `RIGHT OUTER JOIN`, and the `FULL OUTER JOIN`. These joins combine elements similar to the `INNER JOIN`, except when elements don't match the `INNER JOIN` condition, they're still added to the result and padded with `<null>` values.

All of these join operators don't add any specific power to SQL, they can all be described using different constructs, however they can be very helpful when creating queries. Especially the `OUTER JOIN`, for it is very difficult to express without the `OUTER JOIN` itself.

# The Inner Join Operator

## Basic `INNER JOIN` Query

Let's create a query that you should be familiar with, which outputs the `Title` and `Rating` of all the movies.

Like so:

```
SELECT Title, Rating
FROM Movie, Review
WHERE Movie.mID = Review.mID;
```

In the above query, we make a join relation making sure that the movie ID is the same across the `Movie` and `Review` table.

Now let's rewrite it using an `INNER JOIN`:

```
SELECT Title, RATING
FROM Movie INNER JOIN Review
ON Movie.mID = Review.mID;
```

This query does the `INNER JOIN`, or the combination of `Movie` and `Review`, **ON** a specific condition. So it does the cross product of the two tables, then after doing the cross product, it checks the condition and only returns the elements that satisfy the condition.

We would then get a table in return with every movie in the database and its `Rating`.

The `INNER JOIN` operator is the default operator in SQL, and even if you were to take out the `INNER` and just write: `<table> JOIN <table>`, it would default to an `INNER JOIN`.

## Inner Join with Multiple Conditions

Let's create another query that gets the `Title` and `Rating` of all movies whose `Rating` is greater than 3, and was produced after the year 1990.

Like so:

```
SELECT Title, Rating
FROM Movie, Review
WHERE Movie.mID = Review.mID
    and Rating > 3 and Year > 1990;
```

Let's now rewrite this query to use the `INNER JOIN` operator. Our query would look like this:

```
SELECT Title, Rating
FROM Movie JOIN Rating
ON Movie.mID = Review.mID
    and Rating > 3 and Year > 1990;
```

Our query selects all movies whose `Rating` is greater than 3, and whose `Year` is greater than 1990. It joins the `Movie` and `Review` tables, and the join relation is again combining the `Movie` and `Review` records where the `mID` matches. It then checks the condition and returns the tuples that satisfy the condition.

We would then get the following movies in return: *Gravity*, *The Lion King*, *Titanic*, and *Cast Away*.

The `ON` condition can also be ran using the `WHERE` clause, but it's more efficient to use the the `ON` operator for reasons I will not get into here.

## Running a Query with Three Relations

We will create a query that just gets all the general information on each individual movie, and also return all the user's names.

Our query will look like this:

```
SELECT Movie.mID, Title, Year, Director, Rating, User.uID Name
FROM Movie, User, Review
WHERE Movie.mID = Review.mID
    and User.uID = Review.uID;
```

Now lets rewrite it using a join operator:

```
SELECT Movie.mID, Title, Year, Director, Rating, User.uID, Name
FROM Movie JOIN User JOIN Review
ON Movie.mID = Review.mID
    and User.uID = Review.uID;
```

In this particular query, we would **possibly** get an error depending on the type of system that you are using. A few SQL sytems are: SQLite, MySQL, and Postrisk.

If working in the Postrisk system, we would get an error when running the above query because Postrisk does not support multiple join operators. It requires all join operations to be binary, meaning it can only join two relations. If running on the Postrisk system, you could rewrite the query to look like this:

```
SELECT Movie.mID, Title, Year, Director, Rating, User.uID, Name
FROM (Movie JOIN Review ON Movie.mID = Review.mID) JOIN User
ON User.uID = Review.uID;
```

Notice in the above query we joined the two relations `Movie` and `Review` and then wrapped them in parenthesis. This allows for it to satisfy the Postrisk's requirement for all join relations to be binary. We are saying "First join the `Movie` and `Review` table, then join that result with the `User` table. Lastly, we moved the `ON` condition inside the parenthesis for that particular join operator.

# The Natural Join Operator

## Basic `NATURAL JOIN` Query

Let's use one of our previous querys where we used the `INNER JOIN` to combine the `Movie` and `Review` in order to find the `Rating` that the each movie had.

It looked like this:

```sql
SELECT Title, Rating
FROM Movie INNER JOIN Review
ON Movie.mID = Review.mID;
```

As a reminder, the `NATURAL JOIN` operator takes two relations that have column names in common, and then it performs a cross-product that only keeps the tuples where the tuples have the same value in those common attribute names. For example, `Movie` and `Review` have the `mID` column in common. If I were to change the `INNER JOIN` to a `NATURAL JOIN` , they system will automatically apply this equality between the `mID` in the `Movie` relation and the `Review` relation.

It would look like this:

```sql
SELECT Title, Rating
FROM Movie NATURAL JOIN Review;
```

## `NATRUAL JOIN` with Additional Conditions

Let's go back to our query using the `INNER JOIN` that finds the movies whose `Rating` is greater than 3, and was produced after the year 1990.

```sql
SELECT Title, Rating
FROM Movie JOIN Rating
ON Movie.mID = Review.mID
    and Rating > 3 and Year > 1990;
```

Now if we changed this to using a `NATURAL JOIN`, it would look like this:

```
SELECT Title, Rating
FROM Movie NATURAL JOIN Rating
WHERE Rating > 3 and Year > 1990;
```

We changed `JOIN` to `NATURAL JOIN`, then deleted the `ON` condition and changed it to a `WHERE` clause. Lastly, we deleted join relation since `NATURAL JOIN` automatically equates columns with the same name.

We would then get the same movies we got before: *Gravity*, *The Lion King*, *Titanic*, and *Cast Away*.

## The `USING` Clause

There is a feature in SQL that goes with the `NATURAL JOIN` operator that is often regarded as better practice than just using `NATURAL JOIN`. That feature is the `USING` clause, and it explicitly lists the attributes that should be equated when joining two relations.

Using the previous `INNER JOIN` query, it would look like this:

```
SELECT Title, Rating
FROM Movie JOIN Rating
ON Movie.mID = Review.mID
    and Rating > 3 and Year > 1990;
```

Now if we changed this to using a `NATURAL JOIN` with `USING`, it would look like this:

```
SELECT Title, Rating
FROM Movie JOIN Rating USING(mID)
WHERE Rating > 3 and Year > 1990;
```

Notice that we deleted `NATURAL` and only kept the word `JOIN`. We then specify that the `mID` is the attribute that should be equated across `Movie` and `Review`.

We can only put in the `USING` clause any attributes that appear in *both* tables. If we tried to put an attribute that was only in one table, the query would not run and would return an error.

The reason this is considered better practice is because the `NATURAL JOIN` implicitly joins all columns that have the same name, when this may not be favored. For instance, it's possible to not realize that two relations have the same column name, and then the system will sort-of, under the covers, equate those values. Compared to specificaly stating which relations to join which will prevent the query from equating values that don't need to be equated. Also, in real applications there can often be upwards of 100 attributes in a relationship. Thus, making it more likely that you have attributes with the same name but aren't meant to get equated.

# The Outer Join Operator

## Basic Query

Let's start with a query that joins using the `INNER JOIN` operator which will join `Movie` and `Review` on the matching movie IDs, and will return a few attributes.

Like so:

```
SELECT Movie.mID, Title, Rating, ratingDate
FROM Movie INNER JOIN Review USING(mID);
```

Now let's assume that we want results with movies that *do not* have a rating date. That is, there is a `<null>` value for that particular element.

Our query would look like this:

```
SELECT Movie.mID, Title, Rating, ratingDate
FROM Movie RIGHT OUTER JOIN Review USING(mID);
```

Notice that we changed the `INNER JOIN` to a `RIGHT OUTER JOIN` operator. We would get our exact same results as before, which would be table with all the movies and some of their data, such as `Rating` and `ratingDate`. What the `RIGHT OUTER JOIN` operator does is that it takes the tuples on the right side of the relation, and if they don't have a matching tuple on the left then it's still added to the results and padded with `null` values. When there is a tuple on the right with no matching tuple on the left side of the relation, that is called a dangling tuple, and the `RIGHT OUTER JOIN` includes all dangling tuples in its results.

You can also abbreviate the `RIGHT OUTER JOIN` to just `<LEFT/RIGHT> JOIN` and it will produce the same results. On top of that, the `OUTER` join can also be combined with the `NATURAL JOIN` by joining relations like this: `NATURAL <LEFT/RIGHT> OUTER JOIN`.

In our previous query, we checked to see if a tuple in the `Review` table matched the tuple in the `Movie` table. Now what if we wanted to do it the other way arround? For example, check to see if a tuple in the `Movie` table matches a tuple in the `Review` table. You might assume that we could just switch the relations around and put `Movie` on the right, and `Review` on the left. However, SQL actually has a counterpart to the `RIGHT OUTER JOIN` operator, and it is the `LEFT OUTER JOIN`.

It would look like this:

```
SELECT Movie.mID, Title, Director, Rating
FROM Movie LEFT OUTER JOIN Review USING(mID);
```

In the above query we are checking to see if the tuples in the `Movie` table match any tuples in the `Review` table. Instead of checking it the other way arround like in our previoius query. We are also including the `Director` names in this query, and since the `Director` tuples won't match any tuples in the `Review` table, they will be considered dangling tuples. However, recall that the `OUTER JOIN` operator includes dangling tuples in the result, so no need to worry!

## The `FULL OUTER JOIN` Operator

So we know how to include dangling tuples from one relation into our results, but what if we want to include unmatched results from both the left *and* the right side relation? In this case, we use the `FULL OUTER JOIN` operator.

Let's create another query that includes `null` values from both the `ratingDate` and `Director` columns.

Our query would look like this:

```
SELECT Movie.mID, Title, Director, Rating, ratingDate
FROM Movie FULL OUTER JOIN Review USING(mID);
```

We would then get a table of results in return will all the values for each column even the ones that are `null`.

# Aggregation

## Aggregation Functions

The aggregate, or aggregation functions, initially appear inside the `SELECT` clause of a query, and they perform computations over sets of values in multiple rows of our relations. The basic aggregation functions suuported by every SQL systems are: `MIN` , `MAX` , `SUM` , `AVG` , and `COUNT` .

## Two New Clauses

Now that the aggregation functions have been introduced, two new clauses can be added to the SQL select statements. These are the: `GROUP BY` , and the `HAVING` clauses.

- The `GROUP BY` allows us to partition our relations into groups, and then compute aggregated aggregate functions over each group independently.
- The `HAVING` condition allows us to test filters on the results of aggregate values.
- The difference between the `WHERE` and `HAVING` conditions is the `HAVING` applies to all the groups generated from the `GROUP BY` clause. While the `WHERE` condition applies to single rows at a time.

The syntax looks like this:

```
SELECT A1, A2, . . . , A(n)
FROM R1, R2, . . . , R(m)
WHERE <condition>
GROUP BY <columns>
HAVING <condition>;
```

# Aggregation Functions

## Basic Aggregation Query

Our first aggregation query is going to compute the average movie `Rating` of the movies in the database.

Like so:

```
SELECT AVG(Rating)
FROM Review;
```

We would then get a single cell back with the result of: `3.333333333`. All the ratings add up to 40, and there are twelve ratings, so 40/12 = 3.33 repeating.

## Aggregation and Joins

Our second query is going to be a little more complicated. It finds the minimum `Rating` of movies that were produced before the year `2000`.

Our query would look like this:

```
SELECT MIN(Rating)
FROM Movie, Review
WHERE Movie.mID = Review.mID and Year < 2000
```

The above query is saying that the aggregation is going to look at the `Rating` column and it's going to find the lowest value. It is going to look inside the `Movie` and `Review` tables, it will join the `mID` across both relations, and will filter for movies that were produced before the year 2000.

Our resulting movie would be *Top Gun*.

Now lets go back to our `AVG` aggregation query again. We will again compute the average `Rating` of all the movies that were produced after the year 1995. However, we have to change up the query because some movies were rated more than once, and we don't want to include the duplicate ratings in our average rating computation. We only want to count the `Rating` one time for each movie. In order to do that, we need to use a subquery from where we select from `Review`, and the we just want to check for each movie whether their ID is among those whose year is greater than 1995.

Our query would look like this:

```
SELECT AVG(Rating)
FROM Review
WHERE mID IN (SELECT mID FROM Movie WHERE Year > 1995);
```

So now we would get a resulting table where the `Rating` for each movie is only counted once in the computation. Our average `Rating` for all movies produced after the year 1995 would be: 3.

## The `COUNT` Function

The `COUNT` function, not suprisingly, counts the number of tuples in the result that meet the `WHERE` condition.

Like so:

```
SELECT COUNT(*)
FROM Movie
WHERE Year > 1990;
```

We are `SELECT`ing all attributes inside the `Movie` table, and we are going to `COUNT` the number of movies whose year is greater than 1990.

Our result would be the number 6, because the following movies all were produced after the year 1990: *The Lion King*, *Titanic*, *Gravity*, *Harry Potter*, *Cast Away*, and *Spiderman*.

Now lets create another query that counts the number of movies with a `Rating` greater than 3.

It would look like this:

```
SELECT COUNT(*)
FROM Review
WHERE Rating > 3;
```

We would then get a result of 6. However, there are some movies that were rated more than once and who have multiple ratings greater than 3, so we want to eliminate the duplicates and only count each movie once.

Our new query would look like this:

```
SELECT COUNT(DISTINCT mID)
FROM Review
WHERE Rating > 3;
```

SQL includes a nice keyword for us ot use in this particular query. In the `COUNT` function we put the `DISTINCT` keyword and then the name of the attribute that `COUNT` will look for. In this case, `COUNT` will look at the result, and then it will count the distinct values for the particular attribute.

Our result would be the number 5, because the movie *Gravity* was rated twice with a `Rating` greater than 3. So now we eliminate the duplicate and get our correct result by only counting each movie once in the computation.

## Aggregation in Subqueries

We're going to write up a fairly complicated query this time. This query computes the difference of the average `Rating` between movies produced after the year 2000, and movies produced before the year 2000.

Our query will look like this:

```
SELECT Post.avgRating - Pre.avgRating
FROM
    (SELECT AVG(Rating) as avgRating
    FROM Review
    WHERE mID IN
        (SELECT mID FROM Movie WHERE Year >= 2000)) as Post
    (SELECT AVG(Rating) as avgRating
    FROM Review
    WHERE mID NOT IN
        (SELECT mID FROM Movie WHERE Year >= 2000)) as Pre;
```

In the above query we are using subqueries in the `FROM` clause. Recall from earlier chapters that a subquery in the `FROM` clause allows you to write a select statement, and then use the result as if it were an actual relation in the database. So we are going to compute two subqueries in the `FROM` clause, one of them computing the average `Rating` of movies that were produced on or after the year 2000, and the second one computing the average `Rating` of movies that were NOT produced on or after the year 2000.

So lets walk through the query:

- The first subquery says, let's find the movies whose `Year` is greater than or equal to 2000, let's compute their average `Rating`, and we will call it `avgRating`. We will take the whole result of this query and then name it `Post`, as in post-2000.
- Similarly the second relation that we are computing in the `FROM` clause computes the average `Rating` of movies whose `Year` is not greater than or equal to 2000, so their `mID` is `NOT IN` the set of movies whose `Year` is greater than 2000. We then name the result of this query `Pre`, as in pre-2000.
- To conclude, in the `FROM` clause we now have a relation called `Post` with an attribute called `avgRating`, and a second relation called `Pre` with an attribute called `avgRating`. Then, in the `SELECT` clause of the main query, we subtract the `avgRating` of movies from `Pre` from the `avgRating` of movies from `Post`.

If we were to run the query, we would get the result of: 0. Thus, that means the average `Rating` of movies produced before the year 2000 is exactly the same as the average `Rating` of movies produced on or after the yeary 2000.

# Aggregation Clauses

## The `GROUP BY` Clause

The `GROUP BY` clause is only used in conjuction with aggregation. Our first query is going to find the number of movies that were produced in each year, and it's going to do so by using grouping. Essentially what grouping does is it takes a relation and it partitions it by values of a given attribute or set of attributes.

```
SELECT Year, COUNT(*)
FROM Movie
GROUP BY Year;
```

Specifically in this query we're taking the `Movie` relation and we're breaking it into multiple groups. Each group is represented by each individual year. Then for each group we return one tuple in the result containing the `Year` for the group and the number of tuples in the group.

We would then get the number 1 for each year, because there is no two movies in our database that were produced in the same year.

## The `HAVING` Clause

The `HAVING` clause is another clause that is only used with aggregation. The `HAVING` clause allows us to apply conditions to the results of the aggregate functions. The `HAVING` clause is placed after the `GROUP BY` clause and it allows us to check conditions that involve the entire group. In contrast, the `WHERE` clause applies only to one tuple at a time.

Let's create a query that finds directors which have produced more than one movie.

Our query will look like this:

```
SELECT Director
FROM Movie
GROUP BY Director
HAVING COUNT(DISTINCT mID) > 1;
```

The above query is going to get each `Director` from the `Movie` table, and then put them into their own groups. For each group, or `Director`, it is going to check to see if there are more than one `mID`'s that are associated with that `Director`. If there is, the `Director` will be returned in the results. If not, then the `Director` will not be returned.

For this particular query, we would either get an error or a `null` value, because there is no director in our database that has produced more than one movie.

# Data Modification Statements

## Inserting Data

In SQL, there are statements for inserting data, deleting data, and modifying data in a databse. For inserting data, there are two methods:

```
INSERT INTO <table>
    VALUES(A1, A2, . . ., An);
```

The first method allows us to insert one tuple into the database by specifying its actual value. The command above is saying to `INSERT INTO` a table, then specify the value of that tuple, and the result of the command will be to insert one new tuple into that table with the specified value.

The other method of inserting data into a table is to run a query over the database as a select statement. That select statement will produce a set of tuples, and as long as that set of tuples has the same schema as the table, we could insert all of the tuples into the table.

Like this:

```
INSERT INTO <table>
    SELECT STATEMENT;
```

## Deleting Data

Deleting data is fairly simple, and looks like this:

```
DELETE FROM <table>
WHERE <condition>;
```

The above query is saying to `DELETE FROM` a table, `WHERE` a certain condition is true, so this condition is similar to the conditions that we see in the select statement. Then every tuple in the table that satisfies the given condition will be deleted.

This condition can sometimes get fairly complicated, because it can include subqueries, and aggregation over other tables.

## Updating Data

Updating data is done through a command similar to the `DELETE FROM` command. It similarily operates on a single table, it then evaluates a condition over each tuple of the table, and when the condition is true, it will modify the tuple.

It looks like this:

```
UPDATE <table>
SET <attr> = <expression>
WHERE <condition>;
```

The above query takes an attribute that is specified and reassigns it to have the value that is the result of the expression. The condition can also get fairly complicated, for it can have subqueries and so on. As well as the expression, for it can involve queries over other tables or the same table in the database.

You can also update multiple attributes in a tuple. You can update any number of attributes simultaneously by evaluating an expression for each, and assigning the result of that expression to the attribute.

Like so:

```
UPDATE <table>
SET A1=<expr>, A2=<expr>, ...,  An=<expr>
WHERE <condition>;
```

# Insert Commands

Let's assume that we want to insert a new movie into our database. We do this by saying we want to `INSERT INTO Movie` , we use the keyword values and we siply list the values we want to insert.

Like so:

```
INSERT INTO Movie VALUES(109, 'Cinderella', 1950);
```

In this query we are inserting values for the attributes of the `Movie` table. We included a `mID` , `Title` , and `Year` . We did not include a `Director` , so thus there will be a `null` value in the database for that attribute of Cinderella.

Now let's do a little more complicated insert command. Now that we have *Cinderella* in our database, lets add a few users into the `Review` table and have them rate the new movie. We're going to start by inserting two new users into the `User` table.

Like so:

```
INSERT INTO Review VALUES(209, 'Jonathan Murleau')
INSERT INTO Review VALUES(210, 'Barabara Vance');
```

We now have two new users but they have not yet rated on any movies, so let's have them rate on our new movie *Cinderella*.

We now want to return all users where there `mID` does not appear in the `Review` table. Like so:

```
SELECT *
FROM User
WHERE uID NOT IN (SELECT uID FROM Review);
```

Our query would the result the two users that we just inserted: Jonathan and Barbara. Well now that we have singled out the users who have not rated yet, lets insert those users into the `Review` table with the correct schema.

Like this:

```
INSERT INTO Review
SELECT uID, 109, 3, null
FROM User
WHERE uID NOT IN (SELECT uID FROM Review);
```

This query is saying to select all users who do not already appear in the `Review` table. We're going to take these users and we're going to select their `uID` , then we are going to assign them the correct values for all of the `Review` attributes. As a reminder, the `Review` table has the attributes: `uID` , `mID` , `Rating` , and `ratingDate` . So in our query we are selecting their `uID` , then having the rate the `mID` of 109 (which is the movie *Cinderella*), they both are going to rate the movie with a 3, and we inserted a `null` value for the `ratingDate` . Lastly, we are going to take this end result, and `INSERT INTO` the `Review` table.

# Delete Commands

Let's create a query that finds all users who rated more than 2 movies. Let's assume that users are only allowed to rate 2 movies, and no more.

Our query would look like this:

```
SELECT uID, COUNT(DISTINCT Rating)
FROM Review
GROUP BY uID
HAVING COUNT(DISTINCT Rating) > 2;
```

This query says to go into the `Review` relation, then form groups or paritions by each user ID. This allows us to evaluate the set of ratings for each individual `User`. We're going to count how many `DISTINCT` `Rating`s there are in each group (or for each user). Then it's going to check to see if that number is greater than two, and if it is, it's going to return the `uID` of that user, and also the number of `Rating`s that user has given.

If we were to run the query we would return the user: "Darrel Sherman", who has a `uID` of 207. He is the only person to have gone over his rating limit, and has rated 3 movies. Since he has broken the hypothetical contract, we are going to have to delete him from the database.

Our query would look like this:

```
DELETE FROM Review
WHERE uID in
(SELECT uID, COUNT(DISTINCT Rating)
FROM Review
GROUP BY uID
HAVING COUNT(DISTINCT Rating) > 2);
```

So all we did was add the `DELETE FROM <table> WHERE <condition>` statement, and turned our previous query into a subquery in the `WHERE` clause. The query is saying to return all the users who have rated more than 2 movies (just like before), then `DELETE` those users from the `Review` relation.

If we were to run the query, we would delete all instances of the `uID` 207, or Darrel Sherman, from the `Review` relation. However, we would NOT delete him from the `User` relation. In order to do that we would just have to change the relation to `DELETE FROM`.

Like so:

```
DELETE FROM User
WHERE uID in
(SELECT uID, COUNT(DISTINCT Rating)
FROM Review
GROUP BY uID
HAVING COUNT(DISTINCT Rating) > 2);
```

We did the exact same query before, except change the table from `Review` to `User`. Now some SQL database systems don't allow you to delete data if the subquery includes the same relation that you are deleting from, so it can get a little tricky depending on the database you are using.

# Update Commands

Let's find all users who do not currently have a `ratingDate` assigned to one of their rating's, and let's assign it a value.

Our query will look like this:

```
SELECT *
FROM User
WHERE uID in (SELECT uID FROM Review WHERE ISNULL ratingDate);
```

The query is going to look in the `User` relation and for each individual user, it will check for their `uID` in the `Review` table, and if the `ratingDate` returns true for `ISNULL`, then it will retrn that user in the end result.

If we ran the query, our results would include the users "Darrel Sherman", and "Chris Anderson", with a `uID` of 207 and 202 respectively.

Well no we want to update the `ratingDate` for these two individuals in the `Review` table. We would have to edit our query a little bit and have it look like this:

```
UPDATE Review
SET ratingDate = '2014'
WHERE uID in (SELECT uID FROM Review WHERE ISNULL ratingDate) and ISNULL ratingDate;
```

We added the `UPDATE` statement, and we're telling SQL to `UPDATE` the `Review` table. Find all the users with a `uID` that satisfy the subquery in the `WHERE` clause, but only consider reviews that currently have no `ratingDate`. Then take the `ratingDate` for those reviews and give it a value of `'2014'`. We'll assume we don't know the exact date that they rated the movie, but we know it was in 2014.