THE SEQUEL TO SQL

The Sequel to SQL

Level 1 – Section 1

Common Aggregate Functions

THE SEQUEL TO SQL

Finding the Number of Rows

How do we find the total number of rows in the Movies table?

Movies

title	cost	duration
Gone With the Wind	390,000	220
Frankenstein	3,000,000	50
Creature From the Black Lagoon	500,000	79
NULL	100	10

```
SELECT *
FROM Movies;
```

We could pull a list and count them programmatically, but there's a better way.

THE SEQUEL TO SQL

Using the COUNT Function

Here's a way to get the total number of rows in the table.

Movies

title	cost	duration
Gone With the Wind	390,000	220
Frankenstein	3,000,000	50
Creature From the Black Lagoon	500,000	79
NULL	100	10

NULL rows still included in count.

```
SELECT count( *)
FROM Movies;
```

→

```
 count
-------
     4
(1 row)
```


THE SEQUEL TO SQL

Introducing the SQL Aggregate Functions

```
SELECT count(column_name)
FROM table_name;
```

Returns the **total number of rows** that match our search.

```
SELECT sum(column_name)
FROM table_name;
```

Returns the **added sum of values** for a group of rows.

```
SELECT avg(column_name)
FROM table_name;
```

Returns the **calculated average value** for a group of rows.

```
SELECT max(column_name)
FROM table_name;
```

Returns the **largest value** in a group of rows.

```
SELECT min(column_name)
FROM table_name;
```

Returns the **smallest value** in a group of rows.

These only work if columns contain numbers!

Using COUNT With a Column Name

Movies

title	cost	duration
Gone With the Wind	390,000	220
Frankenstein	3,000,000	50
Creature From the Black Lagoon	500,000	79
NULL	100	10

This row is ignored.

```
SELECT count( title)
FROM Movies;
```

→

```
 count
-------
     3
(1 row)
```


THE SEQUEL TO SQL

Using the SUM Function

How do we find the total cost of producing all our movies?

Movies

title	cost	duration
Gone With the Wind	390,000	220
Frankenstein	3,000,000	50
Creature From the Black Lagoon	500,000	79

```
SELECT sum(cost)
FROM Movies;
```

→

```
     sum
------------
 3890000.00
(1 row)
```


THE SEQUEL TO SQL

Using the AVG Function

How do we find the average number of tickets that were sold for all movies?

Movies

title	tickets
Gone With the Wind	1,500
Frankenstein	750
Creature From the Black Lagoon	801

```
SELECT avg(tickets)
FROM Movies;
```

→

```
          avg
----------------------
 1017.0000000000000000
(1 row)
```


THE SEQUEL TO SQL

Using the MAX or MIN Functions

How do we find the highest and lowest number of tickets sold?

Movies

title	tickets
Gone With the Wind	1,500
Frankenstein	750
Creature From the Black Lagoon	801

```
SELECT max(tickets)
FROM Movies;
```

→

```
 max
------
 1500
(1 row)
```

```
SELECT min(tickets)
FROM Movies;
```

→

```
 min
------
 750
(1 row)
```

THE SEQUEL TO SQL

Using the MAX and MIN Functions Together

Movies

title	tickets
Gone With the Wind	1,500
Frankenstein	750
Creature From the Black Lagoon	801

```
SELECT max(tickets), min(tickets)
FROM Movies;
```

→

```
   max    | min
----------+-----
     1500 |  750
(1 row)
```


THE SEQUEL TO SQL

The Sequel to SQL

Level 1 – Section 2

Aggregate Functions Within SQL Clauses

THE SEQUEL TO SQL

Using Aggregate Functions Within SQL Clauses

How can we find the cost of all the Romance and Horror movies, grouped by genre?

Movies

title	cost	genre
Gone With the Wind	390,000	Romance
Casablanca	1,039,000	Romance
Frankenstein	3,000,000	Horror
Creature From the Black Lagoon	500,000	Horror

```
SELECT sum(cost)
FROM Movies;
```

→

```
     sum
------------
 4929000.00
(1 row)
```

Not what we want.

THE SEQUEL TO SQL

Using the GROUP BY Clause

GROUP BY will group together the cost for all movies with similar genres.

Movies

title	cost	genre
Gone With the Wind	390,000	Romance
Casablanca	1,039,000	Romance
Frankenstein	3,000,000	Horror
Creature From the Black Lagoon	500,000	Horror

```
SELECT genre, sum(cost)
FROM Movies
GROUP BY genre;
```

→

```
  genre   |     sum
---------+------------
 Horror   | 3500000.00
 Romance  | 1429000.00
(2 rows)
```

THE SEQUEL TO SQL

Introducing the GROUP BY Clause

We can use the GROUP BY clause to condense a group of columns into a single row.

GROUP BY Recipe

```
SELECT column_name, aggregate_function(column_name)
FROM table_name
GROUP BY column_name;
```


THE SEQUEL TO SQL

Filtering Aggregate Functions

Movies

title	cost	genre
Gone With the Wind	390,000	Romance
Casablanca	1,039,000	Romance
Casanegra	1,000,000	Romance
Frankenstein	3,000,000	Horror
Creature From the Black Lagoon	500,000	Horror
Not So Funny Movie	100	Comedy
Some Scary Movie	200,000	Horror

*Only **1** Comedy film...*

```
SELECT genre, sum(cost)
FROM Movies
GROUP BY genre;
```

```
  genre   |     sum
---------+------------
 Horror   | 3700000.00
 Romance  | 2429000.00
 Comedy   |     100.00
(3 rows)
```

What if we only wanted genres that appear more than once to be part of our result?

Only groups that had 2 genres in it.

THE SEQUEL TO SQL

Using the HAVING Clause

We can use the HAVING clause to only include genres that have more than 1 movie.

Movies

title	cost	genre
Gone With the Wind	390,000	Romance
Casablanca	1,039,000	Romance
Casanegra	1,000,000	Romance
Frankenstein	3,000,000	Horror
Creature from the Black Lagoon	500,000	Horror
Not So Funny Movie	100	Comedy
Some Scary Movie	200,000	Horror

More than 1 row per group

```
SELECT genre, sum(cost)
FROM Movies
GROUP BY genre
HAVING COUNT(*) > 1;
```

```
  genre   |     sum
---------+------------
 Horror   | 3700000.00
 Romance  | 2429000.00
(2 rows)
```


THE SEQUEL TO SQL

Introducing the HAVING Clause

The HAVING clause restricts the groups of rows to only those who meet the specified condition.

HAVING Recipe

```
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;
```

*The **HAVING** clause*

THE SEQUEL TO SQL

Using WHERE With the HAVING Clause

We can use WHERE to filter individual rows before the application of GROUP BY.

Movies

title	cost	genre
Gone With the Wind	390,000	Romance
Casablanca	1,039,000	Romance
Casanegra	1,000,000	Romance
Frankenstein	3,000,000	Horror
Creature from the Black Lagoon	1,000,000	Horror
Not So Funny Movie	2,000,000	Comedy
Some Scary Movie	200,000	Horror

If we only wanted to count movies that grossed at least $1 million:

```
SELECT genre, sum(cost)
FROM Movies
WHERE cost >= 1000000
GROUP BY genre
HAVING COUNT(*) > 1;
```

```
  genre   |   sum
---------+---------
 Romance | 2039000
 Horror  | 4000000
(2 rows)
```


THE SEQUEL TO SQL

The Sequel to SQL

Level 2 – Section 1

Identifying Constraints

THE SEQUEL TO SQL

The Default Behavior of a Table

The default behavior of a table column is to allow insertion of NULL values.

```
CREATE TABLE Promotions
(
  id int,
  name varchar(50),
  category varchar(15)
);
```

Promotions

id	name	category
1	Half Off	Discount
2	Matinee	Non-cash
3	*NULL*	Merchandise

*Inserts row with a NULL value set for the **name** column*

```
INSERT INTO Promotions (id, category)
VALUES (3, 'Merchandise');
```

*When column name is omitted, the default value is **NULL**.*

Adding the NOT NULL Column Constraint

A NOT NULL column constraint ensures values **cannot** be NULL.

```
CREATE TABLE Promotions
(
  id int,
  name varchar(50) NOT NULL,
  category varchar(15)
);
```

Promotions

id	name	category
1	Half Off	Discount
2	Matinee	Non-cash

```
INSERT INTO Promotions (id, category)
VALUES (3, 'Merchandise');
```

```
ERROR:  null value in column "name" violates not-null constraint
DETAIL:  Failing row contains (3, null, Merchandise).
```

We want this error to happen so we don't have bad data.

Why Use Constraints?

The default behavior of database tables can be too permissive.

Constraints can help with these shortcomings!

- Prevent **NULL** values
- Ensure column values are **unique**
- Provide additional **validations**

Preventing Unwanted Duplicates

The default behavior of a table column allows insertion of duplicate values.

```
CREATE TABLE Promotions
(
  id int,
  name varchar(50) NOT NULL,
  category varchar(15)
);
```

Promotions

id	name	category
1	Half Off	Discount
2	Matinee	Non-cash
3	Giveaways	Merchandise
4	Matinee	Discount

Inserts duplicate value on the ***name*** *column:*

```
INSERT INTO Promotions (id, name, category)
VALUES (4, 'Matinee', 'Discount');
```

THE SEQUEL TO SQL

Adding the UNIQUE Column Constraint

The UNIQUE constraint uniquely identifies each field in a table.

```
CREATE TABLE Promotions
(
  id int,
  name varchar(50) NOT NULL UNIQUE,
  category varchar(15)
);
```

Promotions

id	name	category
1	Half Off	Discount
2	Matinee	Non-cash
3	Giveaways	Merchandise

*Cannot insert duplicate values with **UNIQUE** constraint.*

More than 1 constraint can be used on a column!

```
INSERT INTO Promotions (id, name, category)
VALUES (4, 'Matinee', 'Discount');
```

```
ERROR:  duplicate key value violates unique constraint "promotions_name_key"
DETAIL:  Key (name)=(Matinee) already exists.
```

THE SEQUEL TO SQL

Assigning Constraint Names

The database will automatically assign computer-generated constraint names.

```
CREATE TABLE Promotions
(
  id int,
  name varchar(50) NOT NULL UNIQUE,
  category varchar(15)
);
```

Promotions

id	name	category
1	Half Off	Discount
2	Matinee	Non-cash
3	Giveaways	Merchandise

"promotions_name_key" is the constraint name automatically assigned by the database.

```
INSERT INTO Promotions (id, name, category)
VALUES (4, 'Matinee', 'Discount');
```

```
ERROR:  duplicate key value violates unique constraint "promotions_name_key"
DETAIL:  Key (name)=(Matinee) already exists.
```

THE SEQUEL TO SQL

Creating a UNIQUE Table Constraint

Assigning a name to a constraint can help you easily find it when you choose to alter your constraint.

```
CREATE TABLE Promotions
(
  id int ,
  name varchar(50)  NOT NULL ,
  category varchar(15) ,
  CONSTRAINT unique_name  UNIQUE (name)
);
```

"unique_name" is what we assigned for our custom constraint name.

THE SEQUEL TO SQL

Using Our Constraint Name

Assigning a constraint name we can remember helps if we have to troubleshoot any constraint errors.

```
CREATE TABLE Promotions
(
  id int ,
  name varchar(50)  NOT NULL ,
  category varchar(15) ,
  CONSTRAINT unique_name  UNIQUE (name)
);
```

Now we can see the constraint name in the error message.

```
INSERT INTO Promotions (id, name, category)
VALUES (4, 'Matinee', 'Discount');
```

```
ERROR:  duplicate key value violates unique constraint "unique_name"
DETAIL:  Key (name)=(Matinee) already exists.
```

THE SEQUEL TO SQL

Column vs. Table Constraint

Except for NOT NULL, every column constraint can also be written as a table constraint.

column constraint

```
CREATE TABLE Promotions
(
  id int,
  name varchar(50) NOT NULL UNIQUE,
  category varchar(15)
);
```

table constraint

```
CREATE TABLE Promotions
(
  id int ,
  name varchar(50)  NOT NULL ,
  category varchar(15) ,
  CONSTRAINT unique_name  UNIQUE (name)
);
```

Same behavior

THE SEQUEL TO SQL

Ensuring 2 Columns Are Unique

What if we didn't want to allow an insert that has the same name *and* category?

```
CREATE TABLE Promotions
(
  id int ,
  name varchar(50)  NOT NULL ,
  category varchar(15) ,
  CONSTRAINT unique_name  UNIQUE (name, category)
);
```

Promotions

id	name	category
1	Half Off	Discount
2	Matinee	Non-cash
3	Giveaways	Merchandise

```
INSERT INTO Promotions (id, name, category)
VALUES (4, 'Half Off', 'Discount');
```

We already have this unique combination.

```
ERROR:  duplicate key value violates unique constraint "unique_name"
DETAIL:  Key (name, category)=(Half Off, Discount) already exists.
```

Most Tables Should Have a Primary Key

As the primary key, the id column needs to *uniquely* identify every row in this table.

```
CREATE TABLE Promotions
(
  id int ,
  name varchar(50) ,
  category varchar(15)
);
```

Promotions

id	name	category
1	Half Off	Discount
2	Reward Points	Cash Back
3	Matinee	Non-cash
4	Giveaways	Merchandise
4	Buy-1-Get-1	Discount
NULL	Buy-5-Get-2	Discount

Shouldn't allow duplicates or NULL values

THE SEQUEL TO SQL

Making a Column a Primary Key

```sql
CREATE TABLE Promotions
(
  id int PRIMARY KEY,
  name varchar(50) ,
  category varchar(15)
);
```

Promotions

id	name	category
1	Half Off	Discount
2	Reward Points	Cash Back
3	Matinee	Non-cash
4	Giveaways	Merchandise

*Adding a PRIMARY KEY constraint means that column cannot be **NULL** and must be **UNIQUE**.*

THE SEQUEL TO SQL

A Primary Key Prevents Duplicate Entries

```
CREATE TABLE Promotions
(
  id int PRIMARY KEY,
  name varchar(50) ,
  category varchar(15)
);
```

Promotions

id	name	category
1	Half Off	Discount
2	Reward Points	Cash Back
3	Matinee	Non-cash
4	Giveaways	Merchandise

```
INSERT INTO Promotions (id, name, category)
VALUES (4, 'Free Shirt', 'Merchandise');
```

Cannot insert duplicates

```
ERROR:  duplicate key value violates unique constraint "promotions_pkey"
DETAIL:  Key (id)=(4) already exists.
```

A Primary Key Prevents NULL Values

```
CREATE TABLE Promotions
(
  id int PRIMARY KEY ,
  name varchar(50) ,
  category varchar(15)
);
```

Promotions

id	name	category
1	Half Off	Discount
2	Reward Points	Cash Back
3	Matinee	Non-cash
4	Giveaways	Merchandise

```
INSERT INTO Promotions (name, category)
VALUES ('Buy-1-Get-1', 'Merchandise');
```

Cannot insert NULL values

```
ERROR:  null value in column "id" violates not-null constraint
DETAIL:  Failing row contains (null, Buy-1-Get-1, Merchandise).
```


THE SEQUEL TO SQL

Difference Between PK and NOT NULL + UNIQUE

A PRIMARY KEY constraint automatically accomplishes the same goals of **both** the UNIQUE and the NOT NULL constraint. However, it's *not* the same thing.

PRIMARY KEY vs. **NOT NULL + UNIQUE**

Can only be defined once per table

```sql
CREATE TABLE Promotions
(
  id int PRIMARY KEY,
  name varchar(50) NOT NULL UNIQUE,
  category varchar(15)
);
```

Can be used multiple times per table

```sql
CREATE TABLE Promotions
(
  id int PRIMARY KEY,
  name varchar(50) NOT NULL UNIQUE,
  category varchar(15) NOT NULL UNIQUE,
);
```

The Sequel to SQL

Level 2 – Section 2

Value Constraints

THE SEQUEL TO SQL

Creating Valid References Between 2 Tables

How might we start associating particular promotions with specific movies?

Promotions

id	name	category
1	Half Off	Discount
2	Reward Points	Cash Back
3	Matinee	Non-cash

Movies

id	title
1	...
2	...
3	...

We could store the movie title inside the promotions table, but then we'd be repeating ourselves.

Referencing the Movies Primary Key

*The **movie_id** column references the **id** column in the **Movies** table.*

Promotions

id	movie_id	name	category
1	1	Half Off	Discount
2	2	Reward Points	Cash Back
3	3	Matinee	Non-cash

Movies

id	title
1	...
2	...
3	...

Common naming convention

movie_id

Singular version of the table you're referencing

An underscore followed by the column name

Introducing the Foreign Key

movie_id is a **foreign key**.

A foreign key is a column in 1 table that references the primary key column of another table.

*The **movie_id** column is a foreign key referencing the **id** primary key column in the **Movies** table.*

Promotions

id	movie_id	name	category
1	1	Half Off	Discount
2	2	Reward Points	Cash Back
3	3	Matinee	Non-cash

Movies

id	title
1	...
2	...
3	...

Querying for Relationship Data

How would we find the promotions for the movie *Gone With the Wind*?

```
SELECT id
FROM Movies
WHERE title = 'Gone With the Wind';
```

Returns the value `2`

```
SELECT name, category
FROM Promotions
WHERE movie_id = 2;
```

Promotions

id	movie_id	name	category
1	1	Half Off	Discount
2	2	Reward Points	Cash Back
3	3	Matinee	Non-cash

Movies

id	title
1	...
2	Gone With the Wind
3	...

Querying for Relationship Data

Promotions

id	movie_id	name	category
1	1	Half Off	Discount
2	2	Reward Points	Cash Back
3	3	Matinee	Non-cash

Movies

id	title
1	...
2	Gone With the Wind
3	...

Inserting Invalid Data for movie_id

Promotions

id	movie_id	name	category
1	1	Half Off	Discount
2	2	Reward Points	Cash Back
3	3	Matinee	Non-cash
4	999	Matinee	Non-cash

Movies

id	title
1	...
2	...
3	...

*No record with id = **999***

Points to nonexistent primary key in the Movies table

```
INSERT INTO Promotions (id, movie_id, name, category)
VALUES (4, 999, 'Fake Promotion', ‘Hoax');
```

Creating a FOREIGN KEY Constraint

The REFERENCES keyword can be used to make a FOREIGN KEY constraint.

```
CREATE TABLE Movies
(
  id int PRIMARY KEY,
  title varchar(20) NOT NULL UNIQUE
);
```

*The table being referenced must be created **first**.*

*Notice we've added a **primary key**!*

```
CREATE TABLE Promotions
(
  id int PRIMARY KEY,
  movie_id int,
  name varchar(50),
  category varchar(15)
);
```

Adding constraint

```
CREATE TABLE Promotions
(
  id int PRIMARY KEY,
  movie_id int REFERENCES movies(id),
  name varchar(50),
  category varchar(15)
);
```

Preventing Inconsistent Relationships

The foreign key in the second table **must match** a primary key in the table being referenced.

Promotions

id	movie_id	name	category
1	1	Half Off	Discount
2	2	Reward Points	Cash Back
3	3	Matinee	Non-cash

Movies

id	title
1	...
2	...
3	...

```
INSERT INTO Promotions (id, movie_id, name, category)
VALUES (4, 999, 'Fake Promotion', 'Hoax');
```

*The FOREIGN KEY constraint will generate **errors** upon **invalid** data inserts.*

```
ERROR:  insert or update on table "promotions" violates foreign key constraint
"promotions_movie_id_fkey"
DETAIL:  Key (movie_id)=(999) is not present in table "movies".
```

Using a Shorter FOREIGN KEY Constraint Syntax

```
CREATE TABLE Promotions
(
  id int PRIMARY KEY,
  movie_id int REFERENCES movies(id),
  name varchar(50),
  category varchar(15),
);
```

Same thing

```
CREATE TABLE Promotions
(
  id int PRIMARY KEY,
  movie_id int REFERENCES movies ,
  name varchar(50),
  category varchar(15),
);
```

In absence of a column, the **primary key** of the referenced table is used.

THE SEQUEL TO SQL

Using Table Constraint Syntax

```sql
CREATE TABLE Promotions
(
  id int PRIMARY KEY,
  movie_id int REFERENCES movies
  name varchar(50),
  category varchar(15),
);
```

Same behavior

```sql
CREATE TABLE Promotions
(
  id int PRIMARY KEY,
  movie_id int,
  name varchar(50),
  category varchar(15),
  FOREIGN KEY (movie_id) REFERENCES movies
);
```

THE SEQUEL TO SQL

Orphan Records

Orphan records are child records with a foreign key to a parent record that has been **deleted**.

Movies

id	title
4	...
5	...
6	...

Promotions

id	movie_id	name	category
1	4	Half Off	Discount
2	5	Reward Points	Cash Back
3	6	Matinee	Non-cash

When this row is deleted...

*...this row becomes **orphan,** since there's no movie row with **id = 6**.*

Preventing Orphan Records

The FOREIGN KEY constraint helps avoid **orphan records**.

*Statements that would otherwise result in orphan records will now generate **errors**.*

```
DELETE FROM Movies WHERE id = 6;
```

```
ERROR:  update or delete on table "movies" violates foreign key constraint
"promotions_movie_id_fkey" on table "promotions"
DETAIL:  Key (id)=(6) is still referenced from table "promotions".
```

```
DELETE FROM Promotions WHERE movie_id = 6;
DELETE FROM Movies WHERE id = 6;
```

Preventing Orphan Records When Dropping Tables

Tables must be dropped in the correct order.

```
DROP TABLE Movies;
```

```
ERROR:  cannot drop table movies because other objects depend on it
DETAIL:  constraint promotions_movie_id_fkey on table promotions
depends on table movies
```

```
DROP TABLE Promotions;

DROP TABLE Movies;
```

Validating Data Insertion

We want to make sure no film has a duration of less than **10 minutes**.

Movies

id	title	genre	duration
1	Don Juan	Romance	110
2	Peter Pan	Adventure	120
3	The Lost World	Fantasy	105
4	Wolfman Lives	Horror	-10

*It's currently possible to set a **-10 minute** duration, which is clearly wrong.*

```
INSERT INTO Movies (id, title, genre, duration)
VALUES (4, 'Wolfman Lives', 'Horror', -10);
```

THE SEQUEL TO SQL

Adding CHECK Constraint

The CHECK constraint is used to validate the value that can be placed in a column.

```
CREATE TABLE Movies
(
  id int PRIMARY KEY,
  title varchar(20) NOT NULL UNIQUE,
  genre varchar(100),
  duration int CHECK (duration > 0)
);
```

```
INSERT INTO Movies (id, title, genre, duration)
VALUES (4, 'Wolfman Lives', 'Horror', -10);
```

Attempts to insert invalid data on the ***DURATION*** *column will now raise* ***errors****.*

```
ERROR:  new row for relation "movies" violates check constraint "movies_duration_check"
DETAIL:  Failing row contains (4, Wolfman Lives, Horror, -10).
```

The Sequel to SQL:
Level 3 – Section 1

Normalization

THE SEQUEL TO SQL

Create a New Movie

What is the movie's title?

What is the movie's genre?

- [ ] Romance
- [ ] Adventure
- [ ] Fantasy

What is the movie's duration?

Submit

Create a New Movie

What is the movie's title?

Peter Pan

What is the movie's genre?

- [ ] Romance
- [x] Adventure
- [x] Fantasy

What is the movie's duration?

120

Submit

Our Movies Table With Multiple Genres

This is the table we might generate.

id	title	genre	duration
1	Don Juan	Romance	110
2	Peter Pan	Adventure, Fantasy	120
3	The Lost World	Fantasy	105
4	Robin Hood	Adventure	143

We can't update these values separately.

What if we wanted all Adventure movies?

```
SELECT *
FROM movies
WHERE genre = "Adventure";
```

This would only return "Robin Hood".

THE SEQUEL TO SQL

We Need to Use Some Normalization

Normalization is the process of reducing duplication in database tables.

First Normal Form Rule:
Tables must not contain repeating groups of data in 1 column.

Second Normal Form Rule:
Tables must not contain redundancy (unnecessary repeating information).

id	title	genre	duration
1	Don Juan	Romance	110
2	Peter Pan	Adventure, Fantasy	120
3	The Lost World	Fantasy	105
4	Robin Hood	Adventure	143

THE SEQUEL TO SQL

First Normal Form: Flattening the Database

We now have no repeating groups of data.

Each record has a different ***GENRE***

id	title	genre	duration
1	Don Juan	Romance	110
2	Peter Pan	Adventure	120
3	Peter Pan	Fantasy	120
4	The Lost World	Fantasy	105
5	Robin Hood	Adventure	143

- When we update a movie's duration, we have to update every duplicate movie row.
- The title column has a unique constraint.

THE SEQUEL TO SQL

We Need the Second Normal Form

Tables must not contain redundancy (unnecessary repeating information).

id	title	genre	duration
1	Don Juan	Romance	110
2	Peter Pan	Adventure	120
3	Peter Pan	Fantasy	120
4	The Lost World	Fantasy	105
5	Robin Hood	Adventure	143

THE SEQUEL TO SQL

Going Back to a Movies Table With Unique Titles

We can reduce redundancy by eliminating repeating column values within our table.

*New **Movies** table*

id	title	duration
1	Don Juan	110
2	Peter Pan	120
3	The Lost World	105
4	Robin Hood	143

id	title	genre	du
1	Don Juan	Romance	
2	Peter Pan	Adventure	
3	Peter Pan	Fantasy	
4	The Lost World	Fantasy	
5	Robin Hood	Adventure	

THE SEQUEL TO SQL

Creating a New Genres Table

id	title	genre	duration
1	Don Juan	Romance	110
2	Peter Pan	Adventure	120
3	Peter Pan	Fantasy	120
4	The Lost World	Fantasy	105
5	Robin Hood	Adventure	143

*New **Genres** table*

id	genre
1	Romance
2	Adventure
3	Fantasy

*Repeating values on 2 different rows can be combined to make 1 row in a new **Genres** table.*

THE SEQUEL TO SQL

Reviewing Our New Tables

Two new tables have been created without repeating values.

Movies

id	title	duration
1	Don Juan	110
2	Peter Pan	120
3	The Lost World	105
4	Robin Hood	143

Genres

id	genre
1	Romance
2	Adventure
3	Fantasy

Both new tables contain no repeating entries.

We Need a Link Between the Tables

Movies

id	title	duration
1	Don Juan	110
2	Peter Pan	120
3	The Lost World	105
4	Robin Hood	143

Genres

id	genre
1	Romance
2	Adventure
3	Fantasy

A new table

We need a way to link the data between these 2 tables.

Creating a Join Table

Movies

id	title	duration
1	Don Juan	110
2	Peter Pan	120
3	The Lost World	105
4	Robin Hood	143

Genres

id	genre
1	Romance
2	Adventure
3	Fantasy

Our join table

Join table naming convention

Movies_Genres

From the Movies table

From the Genres table

Building Our Join Table With Foreign Keys

Movies

id	title	duration
1	Don Juan	110
2	Peter Pan	120
3	The Lost World	105
4	Robin Hood	143

Genres

id	genre
1	Romance
2	Adventure
3	Fantasy

References the ***primary key*** *of the* ***Movies*** *table*

Movies_Genres

movie_id	genre_id

References the ***primary key*** *of the* ***Genres*** *table*

Both columns are ***foreign keys***

Mapping Movies to Genres

Movies

id	title	duration
1	Don Juan	110
2	Peter Pan	120
3	The Lost World	105
4	Robin Hood	143

Genres

id	genre
1	Romance
2	Adventure
3	Fantasy

Movies_Genres

movie_id	genre_id
1	1
2	2
2	3
3	3
4	2

*Now **Peter Pan** can have 2 genres without redundancy .*

We've Met 2 Normalization Form Rules

First Normal Form Rule:
Tables must not contain repeating groups of data in 1 column.

Second Normal Form Rule:
Tables must not contain redundancy (unnecessary repeating information).

Movies

id	title	duration
1	Don Juan	110
2	Peter Pan	120
3	The Lost World	105
4	Robin Hood	143

Movies_Genres

movie_id	genre_id
1	1
2	2
2	3
3	3
4	2

Genres

id	genre
1	Romance
2	Adventure
3	Fantasy

Updating Movie Information Is Easier

To update a movie duration:

```
UPDATE Movies
SET duration = 134
WHERE id = 2;
```

To add a genre to a movie:

```
INSERT INTO Movies_Genres (movie_id, genre_id)
VALUES (4, 3);
```

Movies

id	title	duration
1	Don Juan	110
2	Peter Pan	134
3	The Lost World	105
4	Robin Hood	143

Movies_Genres

movie_id	genre_id
1	1
2	2
2	3
3	3
4	2
4	3

Genres

id	genre
1	Romance
2	Adventure
3	Fantasy

Gathering Data Is a Little More Complex

How do we find the genres of *Peter Pan*?

```
SELECT id
FROM Movies
WHERE title = "Peter Pan";
```

*Fetches the **id***

```
SELECT genre_id
FROM Movies_Genres
WHERE movie_id = 2;
```

*Fetches the **genre_ids** for our movie*

```
SELECT name
FROM Genres
WHERE id = 2 or id = 3;
```

*Fetches the genre **names** for our movie*

```
WHERE id IN (2,3);
```

Movies

id	title	duration
1	Don Juan	110
2	Peter Pan	134
3	The Lost World	105
4	Robin Hood	143

Movies_Genres

movie_id	genre_id
1	1
2	2
2	3
3	3
4	2
4	3

Genres

id	name
1	Romance
2	Adventure
3	Fantasy

The Sequel to SQL:
Level 3 – Section 2

Relationships

THE SEQUEL TO SQL

3 Different Types of Table Relationships

One-to-One

One-to-Many

Many-to-Many

In this level, we'll take a closer look at each of these.

Previous Example From Level 2

*The **movie_id** column references the **id** column in the **Movies** table.*

Promotions

id	movie_id	name	category
1	1	Half Off	Discount
2	2	Reward Points	Cash Back
3	3	Matinee	Non-cash

Movies

id	title
1	...
2	...
3	...

*What is the name for this type of relationship, where **a promotion can have 1 movie** and **a movie can have many promotions**?*

Looking at One-to-Many Relationship Diagrams

...can have...

One movie... **Movies** —— 1 * **Promotions** *...many promotions.*

*This means it's a **ONE-TO-MANY** relationship.*

Table A —— 1 * **Table B**

One row on the left can refer to multiple rows on the right.

THE SEQUEL TO SQL

Naming Another Relationship

Movies

id	title	duration
1	Don Juan	110
2	Peter Pan	134
3	The Lost World	105
4	Robin Hood	143

Movies_Genres

movie_id	genre_id
1	1
2	2
2	3
3	3
4	2
4	3

Genres

id	name
1	Romance
2	Adventure
3	Fantasy

What is the name for this type of relationship, where ***a movie can have many genres*** *and* ***a genre can have many movies****?*

Looking at Many-to-Many Relationship Diagrams

Many movies...

...can have...

Movies * —— * **Genres**

...many genres.

*This means it's a **MANY-TO-MANY** relationship.*

Notice in this diagram we omit the join table.

Table A * —— * **Table B**

Multiple rows on the left can refer to multiple rows on the right.

THE SEQUEL TO SQL

Splitting a Table Into 2

Customers

id	name	street	city	state
1	Lou	21 Oak St	Miami	FL
2	Ricardo	17 Orange Ave	Orlando	FL
3	Ethel	222 Pine St	New York	NY
4	Carole	618 South St	Orlando	FL

Sometimes you want to split a single table item into 2 tables.
For example, address information.

THE SEQUEL TO SQL

Finding a Common Relationship Between Tables

Customers

id	name	address_id
1	Lou	2
2	Ricardo	3
3	Ethel	6
4	Carole	8

Addresses

id	street	city	state
2	21 Oak St	Miami	FL
3	17 Orange Ave	Orlando	FL
6	222 Pine St	New York	NY
8	618 South St	Orlando	FL

A customer has 1 address *and* ***an address has 1 customer.***

THE SEQUEL TO SQL

Explaining One-to-One Relationship Diagrams

...can have...

A customer... **Customers** 1 —— 1 **Addresses** *...only 1 address.*

This means it's a ***ONE-TO-ONE*** *relationship.*

Table A 1 —— 1 **Table B**

One row on the left can refer to 1 row on the right.

THE SEQUEL TO SQL

3 Different Types of Table Relationships

One-to-One

One-to-Many

Many-to-Many

Another Relationship Example

How should we relate these 2 tables?

Movies

id	title	genre	duration
1	Don Juan	Romance	110
2	Peter Pan	Adventure	120
3	The Lost World	Fantasy	105
4	Robin Hood	Adventure	143

Reviews

id	review	date
1	Loved it!	10/09/2013
2	A must-see!	08/01/2012
3	Hated it	08/01/2012
4	It was okay	02/21/2015
5	Do not see!	03/12/2014

One-to-Many

One movie... **Movies** —*...can have...*— **Reviews** *...many reviews.*

1 *

THE SEQUEL TO SQL

Adding a Linking Column

*The **movie_id** column is added as the foreign key, linking it back to **id** in the **Movies** table.*

Movies

id	title	genre	duration
1	Don Juan	Romance	110
2	Peter Pan	Adventure	120
3	The Lost World	Fantasy	105
4	Robin Hood	Adventure	143

Primary key

Reviews

Foreign key

id	review	date	movie_id
1	Loved it!	10/09/2013	1
2	A must-see!	08/01/2012	1
3	Hated it	08/01/2012	1
4	It was okay	02/21/2015	3
5	Do not see!	03/12/2014	4

THE SEQUEL TO SQL

Looking at How the Values Are Now Related

Don Juan *has many reviews through the* ***movie_id*** *foreign key column.*

Movies

id	title	genre	duration
1	Don Juan	Romance	110
2	Peter Pan	Adventure	120
3	The Lost World	Fantasy	105
4	Robin Hood	Adventure	143

Reviews

id	review	date	movie_id
1	Loved it!	10/09/2013	1
2	must-see!	08/01/2012	1
3	Hated it	08/01/2012	1
4	It was okay	02/21/2015	3
5	Do not see!	03/12/2014	4

THE SEQUEL TO SQL

Forming a Different Relationship

Movies can have many different promotions.

Movies

id	title	date
1	Don Juan	14-JUN-2014
2	The Wolf Man	04-JUL-2014
3	Frankenstein	20-JUL-2014
4	Peter Pan	18-JUN-2014
5	The Lost World	20-JUL-2014
6	Robin Hood	23-JUL-2014

Primary key

Promotions

id	name	category
1	Half Off	Discount
2	Reward Points	Cash Back
3	Matinee	Non-cash
4	Giveaways	Merchandise
5	Free Pass	Non-cash

Primary key

THE SEQUEL TO SQL

This Is a Many-to-Many Relationship

Many movies...

...can have...

Movies * —— * **Promotions**

...many promotions.

THE SEQUEL TO SQL

Creating a Join Table

In this case, we need a join table in order to form a relationship between 2 tables.

Movies_Promotions

Join Table

movie_id	promo_id
1	2
2	3
3	1
4	2
5	2
6	4

Created with only the primary keys from both tables

THE SEQUEL TO SQL

The Sequel to SQL:
Level 4 – Section 1

Inner Joins

THE SEQUEL TO SQL

Fetching Data From Multiple Tables

How do we get a list of reviews and all the associated movie titles?

Movies

id	title	genre	duration
1	Don Juan	Romance	110
2	Peter Pan	Adventure	105
3	The Lost World	Fantasy	106
4	Robin Hood	Adventure	143

Reviews

id	review	movie_id
1	Loved it!	1
2	A must-see!	1
3	Hated it	1
4	It was okay	3
5	Do not see!	4

```
SELECT review, movie_id
FROM Reviews;
```

Fetches all the reviews

```
SELECT title
FROM Movies
WHERE id in (1,3,4);
```

Fetches all the associated movie titles

But we can return this data using 1 query instead of 2.

THE SEQUEL TO SQL

Looking at How to Join 2 Tables

We want to list only the **Reviews** that have **Movies** associated with them.

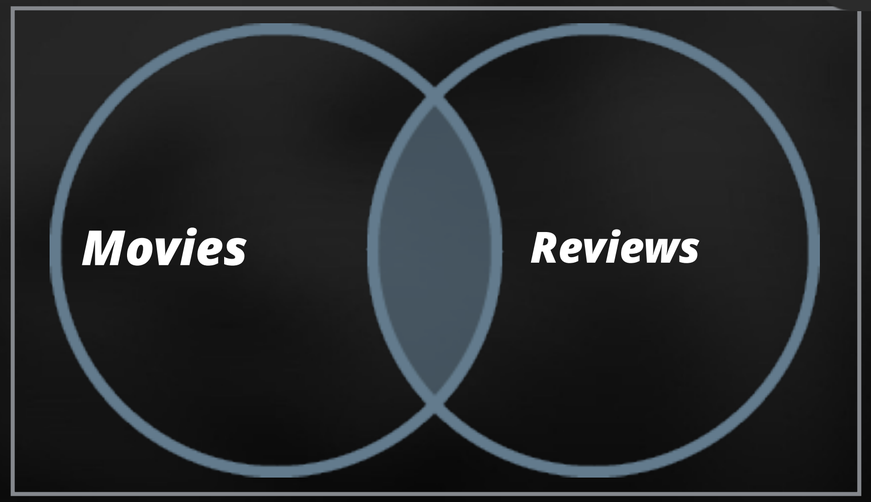
Movies

id	title	genre	duration
1	Don Juan	Romance	110
2	Peter Pan	Adventure	105
3	The Lost World	Fantasy	106
4	Robin Hood	Adventure	143

Reviews

id	review	movie_id
1	Loved it!	1
2	A must-see!	1
3	Hated it	1
4	It was okay	3
5	Do not see!	4

We are looking for all records that only match for both tables.



This inner part is all movies and their matched reviews.

Using the Primary and Foreign Key to Join

Two tables can be joined by the primary key and a foreign key.

Movies

id	title	genre	duration
1	Don Juan	Romance	110
2	Peter Pan	Adventure	105
3	The Lost World	Fantasy	106
4	Robin Hood	Adventure	143

Reviews

id	review	movie_id
1	Loved it!	1
2	A must-see!	1
3	Hated it	1
4	It was okay	3
5	Do not see!	4

*The **movie_id** column in our **Reviews** table is the foreign key, which relates back to the **id** column in the **Movies** table.*

THE SEQUEL TO SQL

Using the INNER JOIN Clause

By using INNER JOIN to create this query, we can show where both tables have matching values.

Movies

id	title	genre	duration
1	Don Juan	Romance	110
2	Peter Pan	Adventure	105
3	The Lost World	Fantasy	106
4	Robin Hood	Adventure	143

Reviews

id	review	movie_id
1	Loved it!	1
2	A must-see!	1
3	Hated it	1
4	It was okay	3
5	Do not see!	4

```
SELECT *
FROM Movies
INNER JOIN Reviews
ON Movies.id=Reviews.movie_id
```

*The **INNER JOIN** clause, followed by the table that will be matched*

We must specify how to match the 2 tables together.

The Result of an Inner Join

```
SELECT *
FROM Movies
INNER JOIN Reviews
ON Movies.id=Reviews.movie_id
```

Where both of our tables have matching values

```
     title      |   genre   | duration | id | movie_id |   review     | id
----------------+-----------+----------+----+----------+--------------+----
 Don Juan       | Romance   |      110 |  1 |        1 | Loved it!    |  1
 Don Juan       | Romance   |      110 |  1 |        1 | A must-see!  |  2
 Don Juan       | Romance   |      110 |  1 |        1 | Hated it     |  3
 The Lost World | Fantasy   |      106 |  3 |        3 | It was okay  |  4
 Robin Hood     | Adventure |      143 |  4 |        4 | Do not see!  |  5
(5 rows)
```

Notice that movie #2 is not here, since it does not have a matching review.

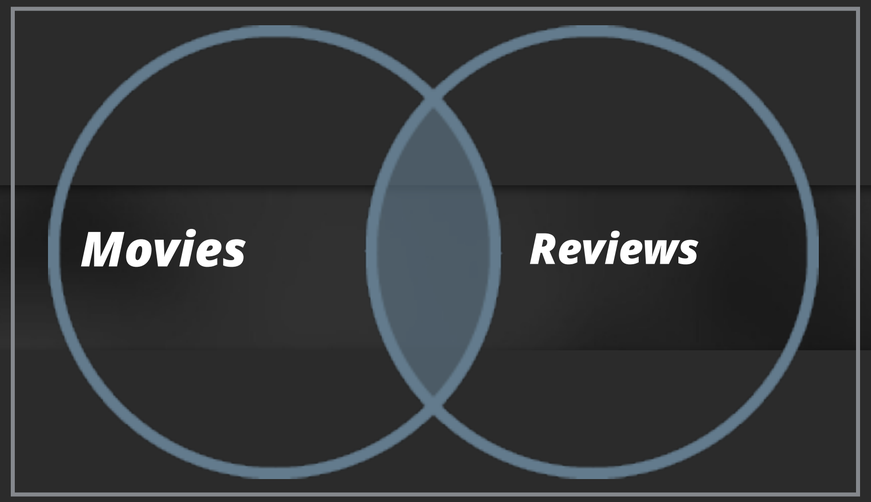
Reversing the INNER JOIN

Since we're always finding the records that match, this query can be written two ways.

```
SELECT *
FROM Movies
INNER JOIN Reviews
ON Movies.id=Reviews.movie_id
```

Same result

```
SELECT *
FROM Reviews
INNER JOIN Movies
ON Reviews.movie_id=Movies.id
```



Using an INNER JOIN to Create Specific Matches

What if we only wanted the movie title and review text to be returned?

Movies

id	title	genre	duration
1	Don Juan	Romance	110
2	Peter Pan	Adventure	105
3	The Lost World	Fantasy	106
4	Robin Hood	Adventure	143

Reviews

id	review	movie_id
1	Loved it!	1
2	A must-see!	1
3	Hated it	1
4	It was okay	3
5	Do not see!	4

Since we have 2 tables, we need to specify which table each column is coming from.

```
SELECT Movies.title, Reviews.review
FROM Movies
INNER JOIN Reviews
ON Movies.id=Reviews.movie_id
```

```
     title       |   review
-----------------+-----------
 Don Juan        | Loved it!
 Don Juan        | A must-see!
 Don Juan        | Hated it
 Robin Hood      | It was okay
 The Lost World  | Do not see!
(5 rows)
```

Remember This Slide?

How do we find the genres of *Peter Pan*?

```
SELECT id
FROM Movies
WHERE title = "Peter Pan";
```

*Fetches the **id***

```
SELECT genre_id
FROM Movies_Genres
WHERE movie_id = 2;
```

*Fetches the **genre_ids** for our movie*

```
SELECT name
FROM Genres
WHERE id = 2 or id = 3;
```

*Fetches the genre **names** for our movie*

```
WHERE id IN (2,3);
```

Movies

id	title	duration
1	Don Juan	110
2	Peter Pan	134
3	The Lost World	105
4	Robin Hood	143

Movies_Genres

movie_id	genre_id
1	1
2	2
2	3
3	3
4	2
4	3

Genres

id	name
1	Romance
2	Adventure
3	Fantasy

SEQUEL TO SQL

INNER JOIN on Multiple Tables

How do we find the genres of *Peter Pan*?

First join →

Second join →

```
SELECT Movies.title, Genres.name
FROM Movies
INNER JOIN Movies_Genres
ON Movies.id = Movies_Genres.movie_id
INNER JOIN Genres
ON Movies_Genres.genre_id = Genres.id
WHERE Movies.title = "Peter Pan";
```

→

```
     title      |    name
----------------+-----------
 Peter Pan      | Adventure
 Peter Pan      | Fantasy

(2 rows)
```

Movies

id	title	duration
1	Don Juan	110
2	Peter Pan	134
3	The Lost World	105
4	Robin Hood	143

Movies_Genres

movie_id	genre_id
1	1
2	2
2	3
3	3
4	2

Genres

id	name
1	Romance
2	Adventure
3	Fantasy

Inn

UEL SQL

The Sequel to SQL:
Level 4 – Section 2

Aliases

THE SEQUEL TO SQL

Giving Our Results More Meaningful Names

Can we give our columns a name that has a more accurate meaning?

The Current Query

```
SELECT Movies.title , Reviews.review
FROM Movies
INNER JOIN Reviews
ON Movies.id=Reviews.movie_id;
```

```
     title       |   review
-----------------+-----------
 Don Juan        | Loved it!
 Don Juan        | A must-see!
 Don Juan        | Hated it
 Robin Hood      | It was okay
 The Lost World  | Do not see!
(5 rows)
```


THE SEQUEL TO SQL

Using Column Aliases

Give the columns new temporary names.

```
SELECT Movies.title AS films, Reviews.review AS reviews
FROM Movies
INNER JOIN Reviews
ON Movies.id=Reviews.movie_id;
```

These are temporary table names that will only affect this query.

```
     films       |   reviews
-----------------+-----------
 Don Juan        | Loved it!
 Don Juan        | A must-see!
 Don Juan        | Hated it
 Robin Hood      | It was okay
 The Lost World  | Do not see!
(5 rows)
```


THE SEQUEL TO SQL

Using Column Aliases

The AS can be dropped from the query.

```
SELECT Movies.title films, Reviews.review reviews
FROM Movies
INNER JOIN Reviews
ON Movies.id=Reviews.movie_id;
```

```
     films      |   reviews
----------------+-----------
 Don Juan       | Loved it!
 Don Juan       | A must-see!
 Don Juan       | Hated it
 Robin Hood     | It was okay
 The Lost World | Do not see!
(5 rows)
```


THE SEQUEL TO SQL

Using More Than 1 Word for a Column Alias

```
SELECT Movies.title "Weekly Movies",
Reviews.review "Weekly Reviews"
FROM Movies
INNER JOIN Reviews
ON Movies.id=Reviews.movie_id;
```

```
     Weekly Films      |   Weekly Reviews
-----------------------+--------------------
 Don Juan              |  Loved it!
 Don Juan              |  A must-see!
 Don Juan              |  Hated it
 Robin Hood            |  It was okay
 The Lost World        |  Do not see!
(5 rows)
```

When using aliases with more than 2 words, you must use quotation marks.

Quotes are also needed if you want capitalization.

THE SEQUEL TO SQL

Our Queries Can Get Verbose

Every time we need to reference the Movies or Promotions table, we have to type the whole word.

Our Original Query

```
SELECT Movies.title, Reviews.review
FROM Movies
INNER JOIN Reviews
ON Movies.id=Reviews.movie_id
ORDER BY Movies.title;
```

*By using **Table Aliases**, we can shorten this query by substituting the Table Name.*

THE SEQUEL TO SQL

Using Table Aliases

By shortening our queries, we can save time when producing longer queries.

```
SELECT m.title, Reviews.review
FROM Movies m
INNER JOIN Reviews
ON m.id=Reviews.movie_id
ORDER BY m.title;
```

*Allows us to refer to the **Movies** table as **m***

THE SEQUEL TO SQL

Using Table Aliases

Table aliases can also be created on any or all tables both within and outside the FROM clause.

```
SELECT m.title, r.review
FROM Movies m
INNER JOIN Reviews r
ON m.id=r.movie_id
ORDER BY m.title;
```

We can create table aliases within JOIN statements.

```
     title       |   review
----------------+-----------
 Don Juan        | Loved it!
 Don Juan        | A must-see!
 Don Juan        | Hated it
 Robin Hood      | It was okay
 The Lost World  | Do not see!
(5 rows)
```

Same result!

THE SEQUEL TO SQL

Remember This INNER JOIN?

How do we find the genres of *Peter Pan*?

First join →

Second join →

```
SELECT Movies.title, Genres.name
FROM Movies
INNER JOIN Movies_Genres
ON Movies.id = Movies_Genres.movie_id
INNER JOIN Genres
ON Movies_Genres.genre_id = Genres.id
WHERE Movies.title = "Peter Pan";
```

THE SEQUEL TO SQL

INNER JOIN on Multiple Tables With Aliases

How do we find the genres of *Peter Pan*?

First join →

Second join →

```
SELECT m.title, g.name
FROM Movies m
INNER JOIN Movies_Genres mg
ON m.id = mg.movie_id
INNER JOIN Genres g
ON mg.genre_id = g.id
WHERE m.title = "Peter Pan";
```

```
     title       |   name
-----------------+-----------
 Peter Pan       | Adventure
 Peter Pan       | Fantasy

(2 rows)
```

THE SEQUEL TO SQL

The Sequel to SQL:
Level 4 – Section 3

Outer Joins

THE SEQUEL TO SQL

Listing All Movies With Optional Reviews

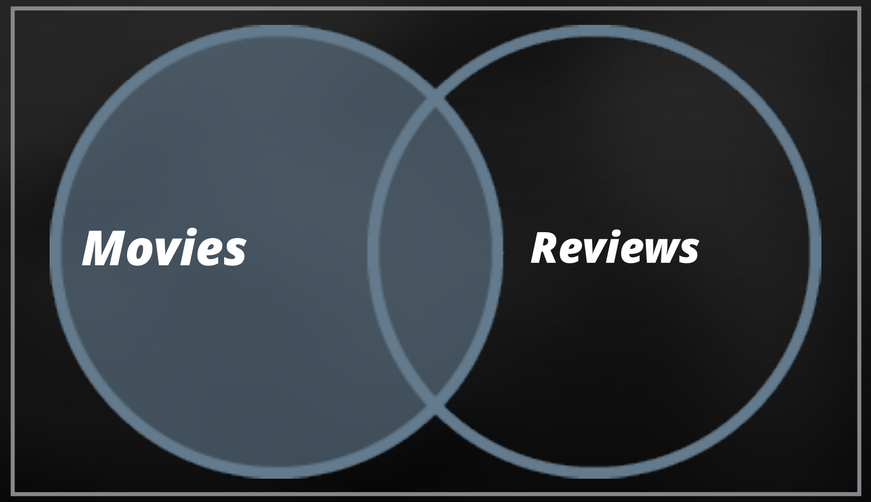
Movies

id	title	genre	duration
1	Don Juan	Romance	110
2	Peter Pan	Adventure	105
3	The Lost World	Fantasy	106
4	Robin Hood	Adventure	143

Reviews

id	review	movie_id
1	Loved it!	1
2	A must-see!	1
3	Hated it	1
4	It was okay	3
5	Do not see!	4

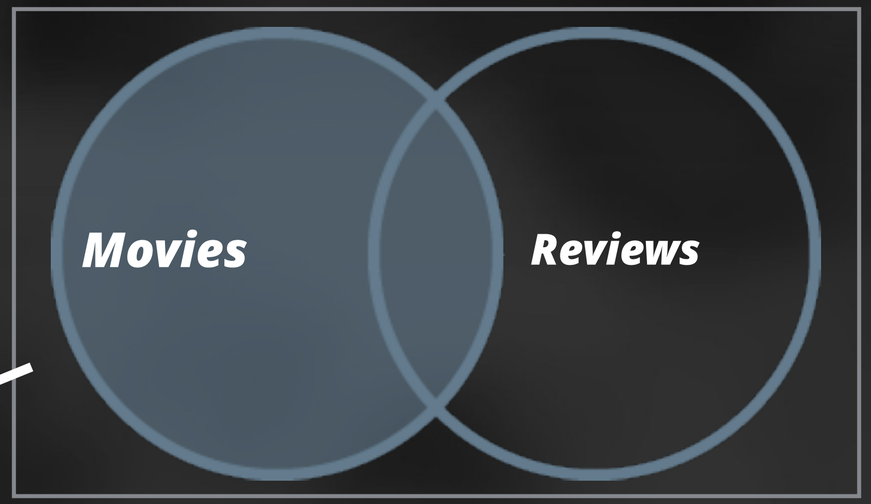
*We are looking to display **all** records in the left table...*



...and display matching records from the table on the right.

Where the Left Outer Join Takes Place

We want to take all rows in the ***Movies*** *table...*



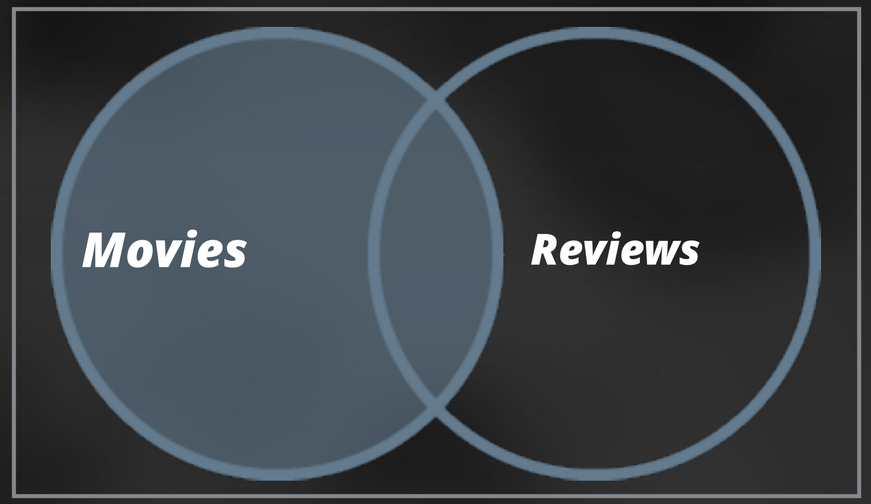
Movies

id	title	genre	duration
1	Don Juan	Romance	110
2	Peter Pan	Adventure	105
3	The Lost World	Fantasy	106
4	Robin Hood	Adventure	143

Reviews

id	review	movie_id
1	Loved it!	1
2	A must-see!	1
3	Hated it	1
4	It was okay	3
5	Do not see!	4

The Left Outer Join



*...and join them with matching **movie_id**s in our **Reviews** table, which is our table on the right.*

Movies

id	title	genre	duration
1	Don Juan	Romance	110
2	Peter Pan	Adventure	105
3	The Lost World	Fantasy	106
4	Robin Hood	Adventure	143

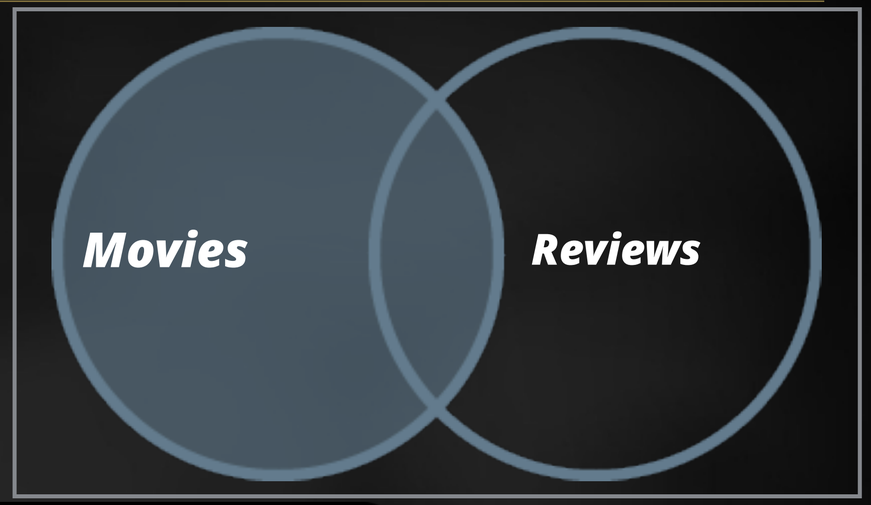
Reviews

id	review	movie_id
1	Loved it!	1
2	A must-see!	1
3	Hated it	1
4	It was okay	3
5	Do not see!	4

Matching id and movie_id Values

All film information and only the **Reviews** associated with them

```
SELECT *
FROM Movies
LEFT OUTER JOIN Reviews
ON Movies.id=Reviews.movie_id;
```



title	genre	duration	id	movie_id	review	id
Don Juan	Romance	110	1	1	Loved it!	1
Don Juan	Romance	110	1	1	A must-see!	2
Don Juan	Romance	110	1	1	Hated it	3
Peter Pan	Adventure	105	2			
The Lost World	Fantasy	106	3	3	It was okay	4
Robin Hood	Adventure	143	4	4	Do not see!	5

(6 rows)

*This row is empty because there is no matching value for this row in the **Reviews** table.*

THE SEQUEL TO SQL

Finding Matching Columns With LEFT OUTER JOIN

Movies

id	title	genre	duration
1	Don Juan	Romance	110
2	Peter Pan	Adventure	105
3	The Lost World	Fantasy	106
4	Robin Hood	Adventure	143

Reviews

id	review	movie_id
1	Loved it!	1
2	A must-see!	1
3	Hated it	1
4	It was okay	3
5	Do not see!	4

This time with columns and table aliases

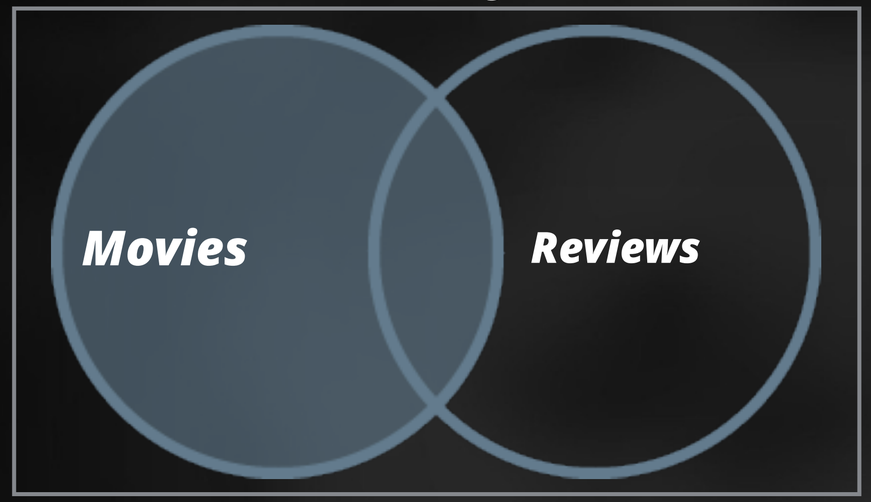
```
SELECT m.title, r.review
FROM Movies m
LEFT OUTER JOIN Reviews r
ON m.id=r.movie_id
ORDER BY r.id;
```

```
     title       |   review
----------------+-----------
 Don Juan        | Loved it!
 Don Juan        | A must-see!
 Don Juan        | Hated it
 Robin Hood      | It was okay
 The Lost World  | Do not see!
 Peter Pan       |
(6 rows)
```


THE SEQUEL TO SQL

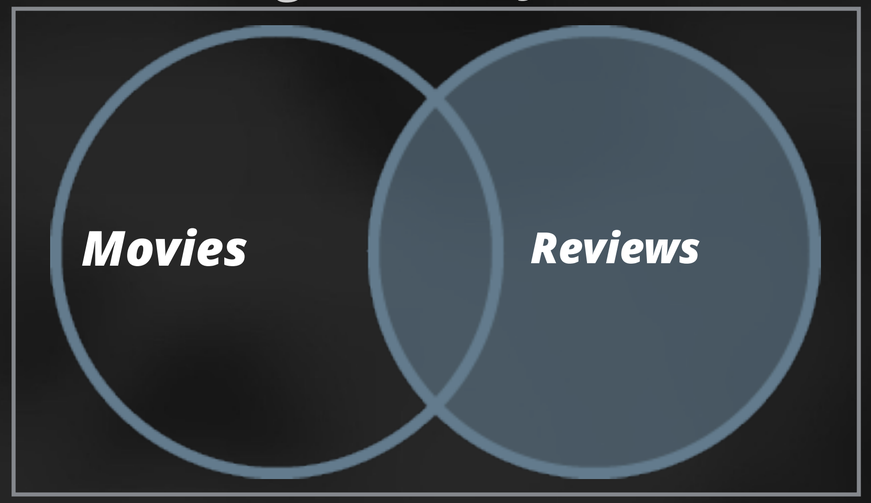
Left Outer Join vs. Right Outer Join

Left Outer Join



*Display all the **Movies** and matching **Reviews** if they exist.*

Right Outer Join



*Display all the **Reviews** and matching **Movies** if they exist.*

Using the RIGHT OUTER JOIN Clause

We will use the RIGHT OUTER JOIN clause to list all **Reviews** and only films that are associated.

Movies

id	title	genre	duration
1	Don Juan	Romance	110
2	Peter Pan	Adventure	105
3	The Lost World	Fantasy	106
4	Robin Hood	Adventure	143

```
SELECT *
FROM Movies
RIGHT OUTER JOIN Reviews
ON Movies.id=Reviews.movie_id;
```

Reviews

id	review	movie_id
1	Loved it!	1
2	A must-see!	1
3	Hated it	*NULL*
4	It was okay	*NULL*
5	Do not see!	4

Notice we modified a few reviews to have a null ***movie_id***

Result of RIGHT OUTER JOIN

Movies

id	title	genre	duration
1	Don Juan	Romance	110
2	Peter Pan	Adventure	105
3	The Lost World	Fantasy	106
4	Robin Hood	Adventure	143

Reviews

id	review	movie_id
1	Loved it!	1
2	A must-see!	1
3	Hated it	NULL
4	It was okay	NULL
5	Do not see!	4

```
     title       |   genre    | duration | id | movie_id |    review     | id
-----------------+------------+----------+----+----------+---------------+-----
 Don Juan        | Romance    |      110 |  1 |        1 | Loved it!     |  1
 Don Juan        | Romance    |      110 |  1 |        1 | A must-see!   |  2
                 |            |          |    |          | Hated it      |  3
                 |            |          |    |          | It was okay   |  4
 Robin Hood      | Adventure  |      143 |  4 |        4 | Do not see!   |  5
(5 rows)
```

*These rows are empty because there are no matching films for these rows in the **Movies** table.*

SEQUEL TO SQL

Finding Matching Columns With RIGHT OUTER JOIN

And this time with table aliases

Movies

id	title	genre	duration
1	Don Juan	Romance	110
2	Peter Pan	Adventure	105
3	The Lost World	Fantasy	106
4	Robin Hood	Adventure	143

Reviews

id	review	movie_id
1	Loved it!	1
2	A must-see!	1
3	Hated it	NULL
4	It was okay	NULL
5	Do not see!	4

```
SELECT m.title, r.review
FROM Movies m
RIGHT OUTER JOIN Reviews r
ON m.id = r.movie_id
ORDER BY r.id;
```

```
     title       |   review
-----------------+-----------
 Don Juan        | Loved it!
 Don Juan        | A must-see!
                 | Hated it
                 | It was okay
 The Lost World  | Do not see!

(6 rows)
```


THE SEQUEL TO SQL

The Sequel to SQL:
Level 5

Subqueries

THE SEQUEL TO SQL

Performing Simple Subqueries

Let's find the **sum of all sales** for **Movies** that were showing as a **Non-cash** promotion.

Movies

id	title	sales	duration
1	Don Juan	45000	110
2	Peter Pan	30000	105
3	The Lost World	30000	106
4	Robin Hood	50000	143

Promotions

id	name	category	movie id
1	Half Off	Discount	1
2	Rewards	Cash Back	1
3	Matinee	Non-cash	1
4	Giveaways	Merchandise	3
5	Free Pass	Non-cash	4

*Let's use data from our **Promotions** table...*

*...and use it to complete a query on our **Movies** table.*

THE SEQUEL TO SQL

Creating Queries to Break Down Our Problem

We can create our inner query, which helps us find which **movie_id**s are Non-cash promotions.

Movies

id	title	sales	duration
1	Don Juan	45000	110
2	Peter Pan	30000	105
3	The Lost World	30000	106
4	Robin Hood	50000	143

Promotions

id	name	category	movie_id
1	Half Off	Discount	1
2	Rewards	Cash Back	1
3	Matinee	Non-cash	1
4	Giveaways	Merchandise	3
5	Free Pass	Non-cash	4

We will call this our "inner query"

```
SELECT movie_id
FROM Promotions
WHERE category = 'Non-cash' ;
```

```
 movie_id
----------
        1
        4
(2 rows)
```


THE SEQUEL TO SQL

Using Our Inner Query on Our Outer Query

We can take the results from the inner query and add them to an outer query to find the SUM of sales.

Movies

id	title	sales	duration
1	Don Juan	45000	110
2	Peter Pan	30000	105
3	The Lost World	30000	106
4	Robin Hood	50000	143

Promotions

id	name	category	movie_id
1	Half Off	Discount	1
2	Rewards	Cash Back	1
3	Matinee	Non-cash	1
4	Giveaways	Merchandise	3
5	Free Pass	Non-cash	4

```
SELECT SUM(sales)
FROM Movies
WHERE id IN
 (SELECT movie_id
 FROM Promotions
 WHERE category = 'Non-cash' );
```

This is a "query in a query."

*The inner query or **subquery** is surrounded by parentheses.*

THE SEQUEL TO SQL

Finding the Solution to Our Combined Queries

We can take the results from the inner query and add them to an outer query to find the SUM of sales.

Movies

id	title	sales	duration
1	Don Juan	45000	110
2	Peter Pan	30000	105
3	The Lost World	30000	106
4	Robin Hood	50000	143

Promotions

id	name	category	movie id
1	Half Off	Discount	1
2	Rewards	Cash Back	1
3	Matinee	Non-cash	1
4	Giveaways	Merchandise	3
5	Free Pass	Non-cash	4

```
SELECT SUM(sales)
FROM Movies
WHERE id IN
 (SELECT movie_id
 FROM Promotions
 WHERE category = 'Non-cash' );
```

→

```
SELECT SUM(sales)
FROM Movies
WHERE id IN
 (1,4);
```

→

This is the sum of 45000 and 50000.

```
  sum
-------
 95000
(1 row)
```


SEQUEL TO SQL

Comparing Subqueries vs. JOINs

We can also find the same results by using a JOIN.

Subquery — easier to read

```
SELECT SUM(sales)
FROM Movies
WHERE id IN
 (SELECT movie_id
 FROM Promotions
 WHERE category = 'Non-cash' );
```

JOIN query — better for performance

```
SELECT SUM(m.sales)
FROM Movies m
INNER JOIN Promotions p
ON m.id = p.movie_id
WHERE p.category = 'Non-cash';
```

Same result

```
  sum
-------
 95000
(1 row)
```


THE SEQUEL TO SQL

Subquery Syntax

```
WHERE <field> IN(<subquery>)
```

Filters rows that have a matching id

```
WHERE <field> NOT IN(<subquery>)
```

Filters rows that don't have a matching id

Using Correlated Subqueries

We want to find only films that have a duration greater than the average duration of all films.

Movies

id	title	genre	duration
1	Don Juan	Romance	110
2	Peter Pan	Adventure	105
3	The Lost World	Fantasy	106
4	Robin Hood	Adventure	143

```
SELECT * FROM Movies WHERE duration > AVG(duration);
```

```
ERROR:  aggregate functions are not allowed in WHERE
```

We need a correlated subquery!

THE SEQUEL TO SQL

Using Correlated Subqueries

Supplying data from the main query back into the subquery is called a correlated subquery.

Movies

id	title	genre	duration
1	Don Juan	Romance	110
2	Peter Pan	Adventure	105
3	The Lost World	Fantasy	106
4	Robin Hood	Adventure	143

```
SELECT * FROM Movies WHERE duration >
(SELECT AVG(duration) FROM Movies);
```

Both queries depend on each other's values to return a correct result.

```
 id |    title     |   genre    | duration
----+--------------+------------+----------
  4 | Robin Hood | Adventure |      143
(1 row)
```