

# Joins in SQLite

## What is a “join”?

If we want to get data from two or more tables we need a query that uses a join operation.

Examples:

- customer names are in the customer table, addresses are in the address table
- film titles are in the film table, actor names are in the actor table

Recall from intro that join ( $\bowtie$ ) forms the cross-product of two tables, creating a new table that has columns from both tables.

Today's topic: how to join tables in SQL queries

## Join Syntax

The syntax for selecting from multiple tables is simple -- just specify two or more table names after “FROM”, e.g.

```
SELECT col1, col2, ... FROM language, category;
```

But notice what happens: the result is the cross product of all rows from language with all rows from category:

```
sqlite> select language_id, category_id from language,  
category;
```

language_id	category_id
1	1
1	2
1	3
...	
2	1
2	2
2	3
...	
6	14
6	15
6	16

## Cross Joins

We can use the count function to verify the join created the cross product of the rows from each table:

```
sqlite> SELECT count(*) FROM language;
```

6

```
sqlite> SELECT count(*) FROM category;
```

16

```
sqlite> SELECT count(*) FROM language, category;
```

96

Bottom line: unless we really do want all combinations we need to “connect” rows in one table with rows in the other.

The languages/categories example is not very realistic — these tables have nothing in common, and it’s not likely they will be used together.

But it is a good example of how a join forms a cross product: all  $6 \times 16 = 96$  combinations are formed.

## Aside: Name Conflicts

If we want to see languages and categories as strings instead of their IDs we discover a problem: both tables have name as a column name:

```
sqlite> select * from language limit 2;
```

language_id	name	last_update
1	English	2011-09-14 18:05:00
2	Italian	2011-09-14 18:05:01

```
sqlite> select * from category limit 2;
```

category_id	name	last_update
1	Action	2006-02-15 04:46:27.000
2	Animation	2006-02-15 04:46:27.000

That means this statement is ambiguous:

```
sqlite> SELECT name FROM language, category;
```

```
Error: ambiguous column name: name
```

## Name Conflicts (cont'd)

When a column name appears in two tables use a “qualified name” syntax to distinguish them:

```
sqlite> SELECT language.name, category.name FROM language,  
category;
```

name	name
English	Action
English	Animation
English	Children
English	Classics
...	

But notice the column headers. If format is important (and there are cases where it does matter) we can rename the columns in the output table:

```
sqlite> SELECT language.name AS language, category.name AS  
category FROM language, category;
```

language	category
English	Action
English	Animation
English	Children
English	Classics
...	

## Primary Keys

Tables have at least one column that has a unique value for each row.

This column is called the ***primary key***.

The database system uses primary keys for efficient lookup using a process similar to binary search

It's common to use `id` or an identifier containing "id" as the column name

In Sakila primary keys have names that include the table name, e.g.

```
sqlite> SELECT * FROM category;
```

category_id	name	last_update
1	Action	2006-02-15 04:46:27.000
...		

```
sqlite> SELECT * FROM language;
```

language_id	name	last_update
1	English	2011-09-14 18:05:00
...		

## Foreign Keys

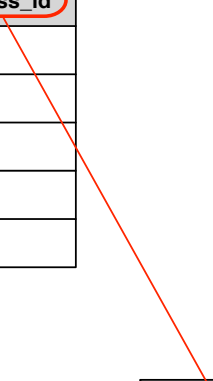
When tables have related information the data is “connected” using common data value.

The join operation that combines the tables uses these values to control the number of rows in the output

A common technique is to store the ID of a record from the other table. These ids are called **foreign keys**, and they are intended to be used when joining the two tables.

Example: the customer table has a column named `address_id`. Values in this column are ids of address strings in the address table.

customer_id	store_id	first_name	last_name	address_id
1	1	Mary	Smith	5
2	1	Patricia	Johnson	6
3	1	Linda	Williams	7
4	1	Barbara	Jones	8
5	1	Elizabeth	Brown	9



address_id	address	city_id
1	47 MySakila Dr	300
2	28 MySQL Blvd	576
...	...	...
5	1913 Hanoi Way	463
6	1121 Loja Ave	449

Important:

★ Queries that look up a customer's address need to join two tables

★ To avoid the full cross product, include

```
customer.address_id = address.address_id
```

in the the WHERE clause:

```
SELECT last_name, first_name, address
FROM customer, address
WHERE customer.address_id = address.address_id
```

To verify the join pairs up each customer with their address count the rows:

```
sqlite> SELECT count(*) FROM customer;
```

```
599
```

```
sqlite> SELECT count(*) FROM address;
```

```
603
```

The number of rows in the complete cross product:

```
sqlite> SELECT count(*) FROM customer, address;
```

```
361197
```

The number of rows in the query that compares IDs:

```
sqlite> SELECT count(*) FROM customer, address
WHERE customer.address_id = address.address_id;
```

```
599
```



## Join Examples

What are the names and addresses of the customers?

```
sqlite> SELECT last_name, first_name, address
        FROM customer, address
        WHERE customer.address_id = address.address_id;
```

last_name	first_name	address
Smith	Mary	1913 Hanoi Way
Johnson	Patricia	1121 Loja Aven
Williams	Linda	692 Joliet Str
...		

Which customers live on Hanoi Way? Use AND to specify additional constraints:

```
sqlite> SELECT last_name, first_name, address
        FROM customer, address
        WHERE customer.address_id = address.address_id
              AND address like "%Hanoi%";
```

last_name	first_name	address
Smith	Mary	1913 Hanoi Way
Richardson	Ashley	1214 Hanoi Way

## JOIN ... ON

The preferred way to write the previous queries is to use the JOIN operator:

```
SELECT last_name, first_name, address
FROM customer JOIN address ON customer.address_id =
address.address_id;
```

Compare the two versions of the query. They are the same except

(1) replace the comma separating table names with the JOIN keyword:

```
... FROM customer, address ...
... FROM customer JOIN address ...
```

(2) instead of WHERE use ON to specify the constraint:

```
... WHERE customer.address_id = address.address_id ...
... ON customer.address_id = address.address_id ...
```

## Why use the JOIN keyword?

Most queries will have additional constraints

Example: list customers who live on Salinas St:

```
SELECT last_name, first_name, address
      FROM customer JOIN address ON customer.address_id =
address.address_id
      WHERE address LIKE "%salinas%";
```

Using the original form would put two constraints after the word WHERE.

With longer queries the table-joining constraints will get lost among the other constraints.

Table joining constraints are IMPORTANT -- mess them up and you get a full cross product.

***Much*** better style to collect them after the word ON and before the word WHERE to separate the table joining constraints from other requirements.

## JOIN ... USING

Another special syntax for joins works when the connecting column has the same name in both tables:

```
SELECT last_name, first_name, address
FROM customer JOIN address USING (address_id)
WHERE address LIKE "%salinas%";
```

Note parentheses around column name after the word USING

I like this form -- shorter is better -- but use whichever form you prefer.

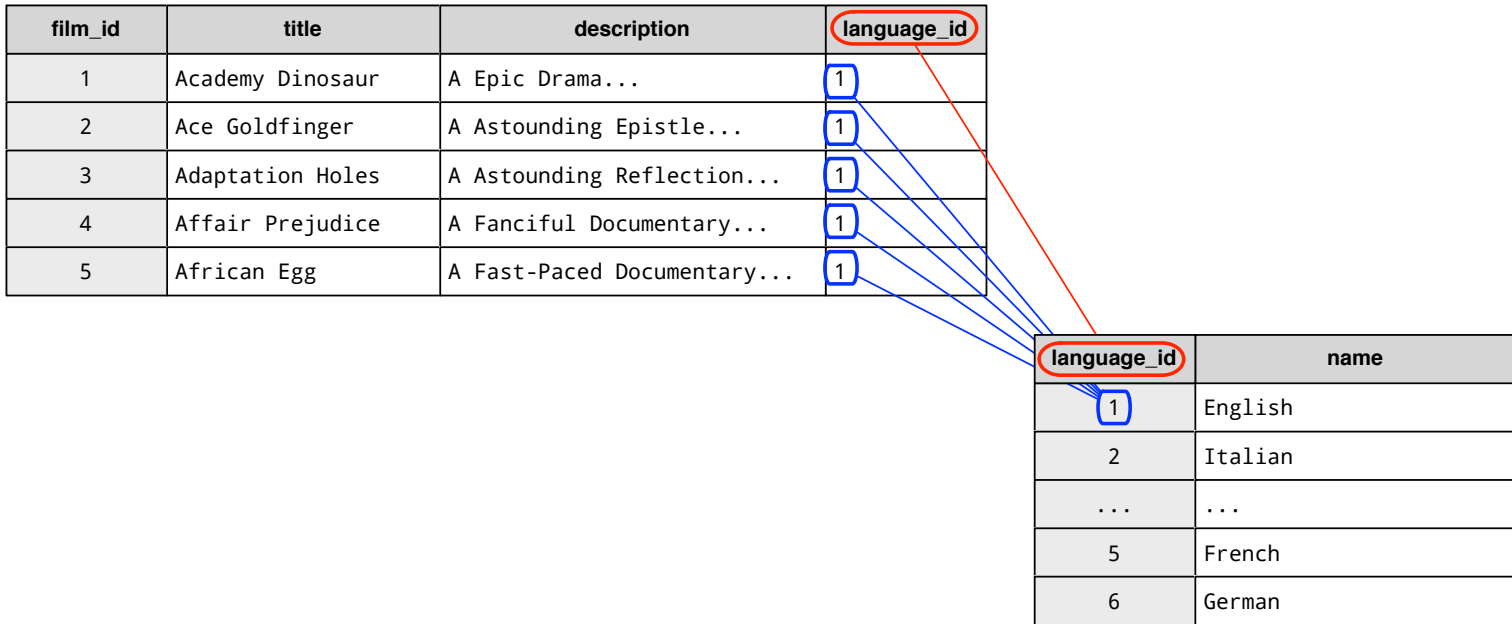
Note: this explains why the primary key column in the address table is `address_id` instead of just `id`.

The fully qualified name `address.address_id` seems redundant, but it allows the JOIN...USING syntax when `address_id` is a foreign key in other tables.

## One-to-Many Relationships

It is often the case that a record in one table can be linked to several records in another table

```
SELECT title, language.name AS language
FROM film JOIN language USING (language_id);
```

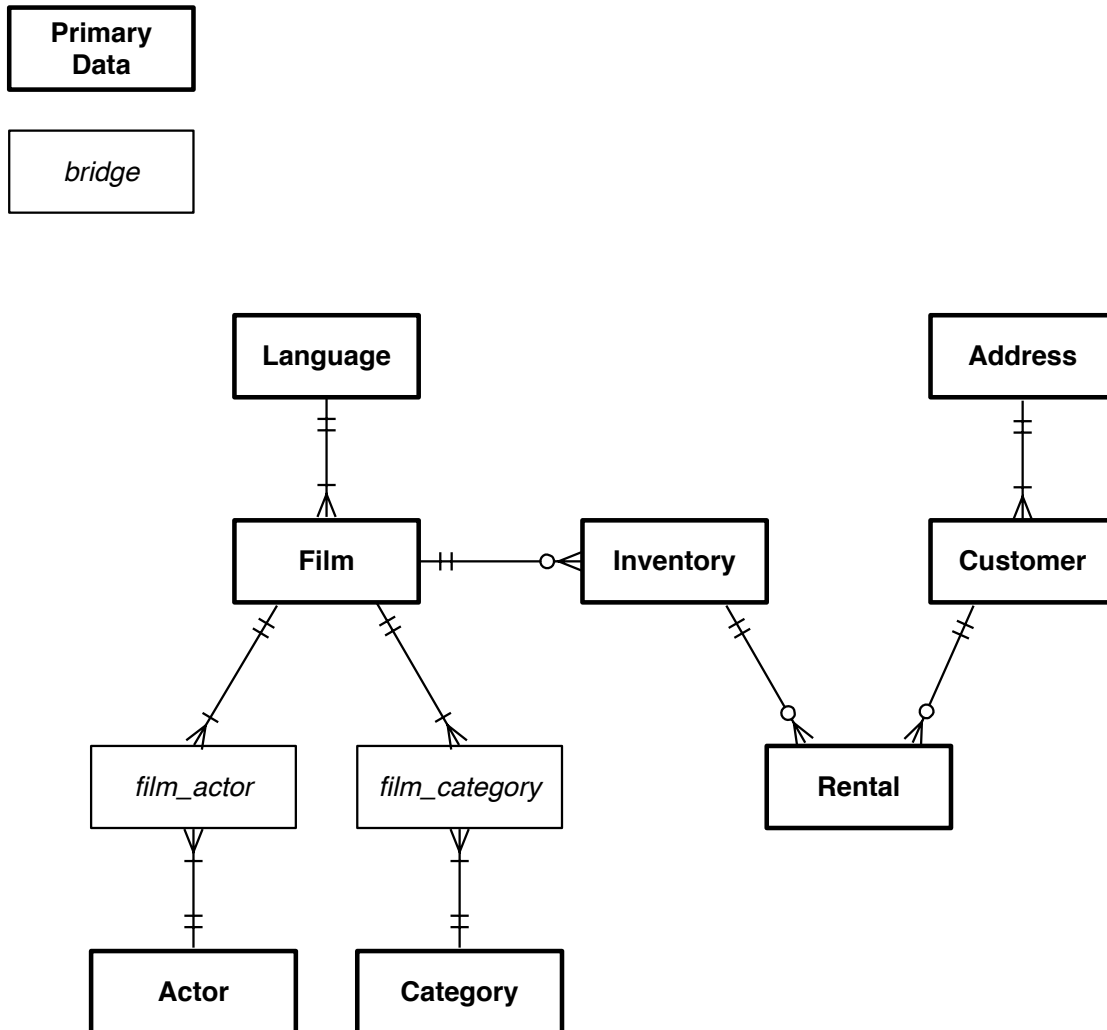


The fact that the foreign key (**language\_id** in the film table) is the primary key in the other table allows for a fast lookup of the language name

## Schema Diagrams

To write a query that joins two or more tables we need to know what the tables have in common

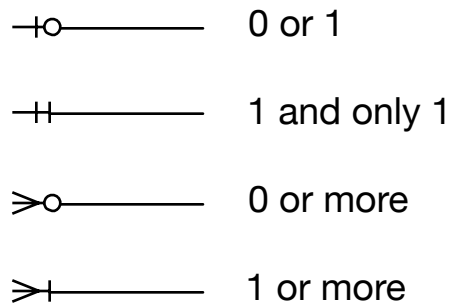
A useful aide: **entity-relationship diagram** (ERD)



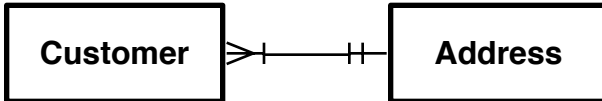
[Download Sakila.Schema.pdf from Canvas](#)

## “Crow’s Foot” Notation

The lines connecting boxes convey information about how many records are part of the relationship

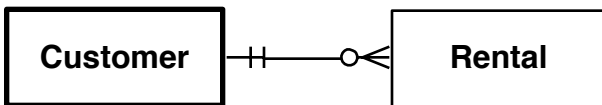


### Examples



*Every customer has one address*

*More than one customer might live at an address*



*A customer can have 0 or more rental transactions*

*Each transaction has exactly one customer*

## Many-to-Many Relationships

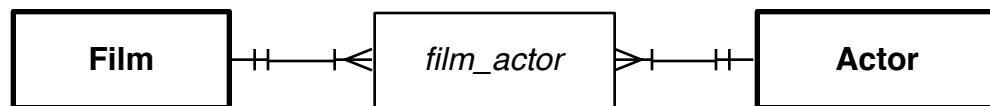
Several tables in the Sakila DB consist of nothing but foreign keys, e.g.

```
sqlite> select * from film_actor;
```

actor_id	film_id
1	1
1	23
1	25
...	
2	31
2	47
...	
15	31
...	

These sorts of tables implement “many to many” relationships.

- an actor can appear in many different films
- films can have many different actors



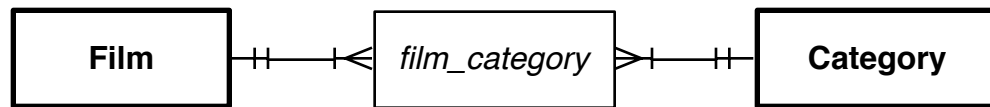
*These sorts of tables are sometimes called “bridge” tables*

- *they don't contain data*
- *records serve as connections between data tables*



## Queries with Bridge Tables

The film\_category table links films and categories (because films might have several categories, e.g. “animated horror romance”).



To get both film attributes (e.g. titles) and category attributes (e.g. names) we need to join all three tables

If a, b, and c are table names the JOIN clause looks like this:

```
... a JOIN b USING (x) JOIN c USING (y) ...
```

This query lists the names and ratings of all the horror movies:

```
SELECT title, rating, name
FROM film JOIN film_category USING (film_id)
      JOIN category USING (category_id)
WHERE name = "Horror"
```

Note: joins are associative, just like addition in regular algebra. In algebra

$$a + b + c = a + (b + c) = (a + b) + c$$

and in relational algebra

$$a \bowtie b \bowtie c = a \bowtie (b \bowtie c) = (a \bowtie b) \bowtie c$$

## More Join Examples

The film table has a language\_id column with foreign keys that link it to the language table.

*What language was used in films described as “epic”?*

```
SELECT title, description, language.name
FROM film JOIN language USING (language_id)
WHERE description LIKE "%epic%";
```

The inventory table links films and stores.

*Which stores have copies of the film named “Citizen Shrek”?*

```
SELECT title, store_id, inventory_id
FROM film JOIN inventory USING (film_id)
WHERE title = "Citizen Shrek";
```

We can add a GROUP BY clause to find out how many copies of the film are in stock (in all stores):

```
SELECT title, count(title) AS n
FROM film JOIN inventory USING (film_id) GROUP BY film_id;
```

The next examples show how to use two joins to connect actors to films.

Suppose we want to know the names and ratings of films starring Meryl Gibson.

First, write a query that finds the film IDs of Meryl Gibson movies:

```
SELECT film_id FROM actor JOIN film_actor USING (actor_id)
WHERE actor.first_name = "Meryl" and actor.last_name =
"Gibson";
```

Next join with the film table to get the rating (and we don't need to see film\_id any more):

```
SELECT title, rating FROM actor JOIN film_actor USING
(actor_id) JOIN film USING (film_id) WHERE actor.first_name =
"Meryl" and actor.last_name = "Gibson";
```

Now suppose we want to know how many films are in each category. Add a "group by" clause and replace the film name with the count:

```
SELECT count(title) as n, rating FROM actor JOIN film_actor
USING (actor_id) JOIN film USING (film_id) WHERE
actor.first_name = "Meryl" and actor.last_name = "Gibson"
GROUP BY rating;
```

Challenge: can you write a query that prints the names of the Japanese animations?

***Writing a query is a lot like writing a small program***

*Start with a very basic and simple query*

*Gradually add more tables and/or constraints*

*Include "scaffolding" (columns that can be used to validate the query but won't necessarily be part of the final answer)*

*Test each version in an interactive session*