Joins in SQLite

Reading: Sec 12.2

**What is a “join”?**

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

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

Example (using Saklila): suppose we want to find Japanese animations. Info is likely in the language table and the category table so we need to join those two tables.

**Cross Joins**

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

```sql
SELECT * FROM language, category;
```

But notice what happened: the result is the cross product of all rows from language with all rows from category, which you can verify using the count function:

```sql
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.
**Foreign Keys**

People who design databases often include columns in one table that have ids of records in another 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.

**Queries that look up a customer’s address should do a join of the two tables, but 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 this works as expected, count the rows:

```
sqlite> SELECT count(*) FROM customer;
599
sqlite> SELECT count(*) FROM address;
603
```

Complete cross product:

```
sqlite> SELECT count(*) FROM customer, address;
361197
```

Join restricted by customer.address_id = address.address_id:

```
sqlite> SELECT count(*) FROM customer, address WHERE customer.address_id = address.address_id;
599
```

Conclusion: every customer has exactly one address.
JOIN ... ON

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

```sql
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:

```sql
... FROM customer, address ...;
... customer JOIN address ...
```

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

```sql
... WHERE customer.address_id = address.address_id ...
```

Why use JOIN?

Most queries will have additional constraints.

Example: list customers who live on Salinas St:

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

Using the first 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 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:

```sql
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 id column in the address table is address_id instead of just id. The full name (address.address_id) seems redundant, but it allows the JOIN...USING syntax since address_id will be a foreign key in other tables.
Many to Many Relationships

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

```sql
SELECT * FROM film_category;
```

<table>
<thead>
<tr>
<th>film_id</th>
<th>category_id</th>
<th>last_update</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>2011-09-14 18:10:02</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>2011-09-14 18:10:02</td>
</tr>
</tbody>
</table>

These sorts of tables implement “many to many” relationships. In this case, it’s possible for a film to have several categories (animated horror musical?) or for one category to describe several films.

This query lists the category names of all the films (which are listed by their IDs):

```sql
SELECT film_id, category.name
FROM film_category JOIN category USING (category_id);
```

(but it turns out that in Sakila DB each film has just one category)

This query lists film names that belong to category 5 (comedy):

```sql
SELECT category_id, film.title
FROM film_category JOIN film USING (film_id) WHERE category_id = 5
```

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

We can do this by joining all three, e.g. if a, b, and c are table names we can write

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

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

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

Joins are associative, just like addition. In algebra

\[ a + (b + c) = (a + b) + c \]

and in relational algebra

\[ a \bowtie (b \bowtie c) = (a \bowtie b) \bowtie c \]

BUT: for efficiency sake the first join should be the one that produces the smallest table. Without parens SQL does joins from left to right.
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”?

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

Note: the result is kind of boring -- it turns out all films were made in English. How can you verify that assertion? (It also means we’re out of luck if we want to find Japanese animations)

The inventory table links films and stores in a many-to-many relationship: one store can have multiple copies of a film, and a film can be in stock at several stores.

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

```sql
SELECT title, store_id, inventory_id
FROM film JOIN inventory USING (film_id)
WHERE title = "CITIZEN SHREK";
```

We can use group by to find out how many copies of the film are in stock (in all stores):

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

This one’s a little trickier: we can group by both film and store id to get a table to show how many copies are at each store:

```sql
SELECT title, store_id, count(*)
FROM film JOIN inventory USING (film_id) GROUP BY film_id, store_id;
```
The next examples show how to use two joins to connect actors to film categories. First, connect an actress to IDs of films she starred in:

**What are the film ids of films starring Meryl Gibson?**

```sql
SELECT film_id
FROM film_actor JOIN 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):

**What are the names and ratings of films starring Meryl Gibson?**

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

If you want to count how many films are in each category group by category and replace the film name with the count:

```sql
SELECT rating, count(*)
FROM film_actor JOIN actor USING (actor_id) JOIN film USING (film_id)
WHERE actor.first_name = "MERYL" and actor.last_name = "GIBSON"
GROUP BY rating;
```