Lecture 06: SQL Joins

DATA 351: Data Management with SQL

Lucas P. Cordova, Ph.D.

2026-02-02

This lecture covers SQL joins using the Blockbuster Bend database. We explore inner joins, outer joins, cross joins, and self joins with real examples from film rental data.

Table of contents

1 Joining Tables

In the age of Netflix and other digital streaming platforms, Harding credits her store's success to both local support and Bend's budding tourism industry.
BLOCKBUSTER
BLOCKBUSTER

Blockbuster Bend is the final video rental store. Today we connect data across the store's database.

1.1 Blockbuster Bend Database

1.1.1 Load the Database

Run these commands to load the data into PostgreSQL:

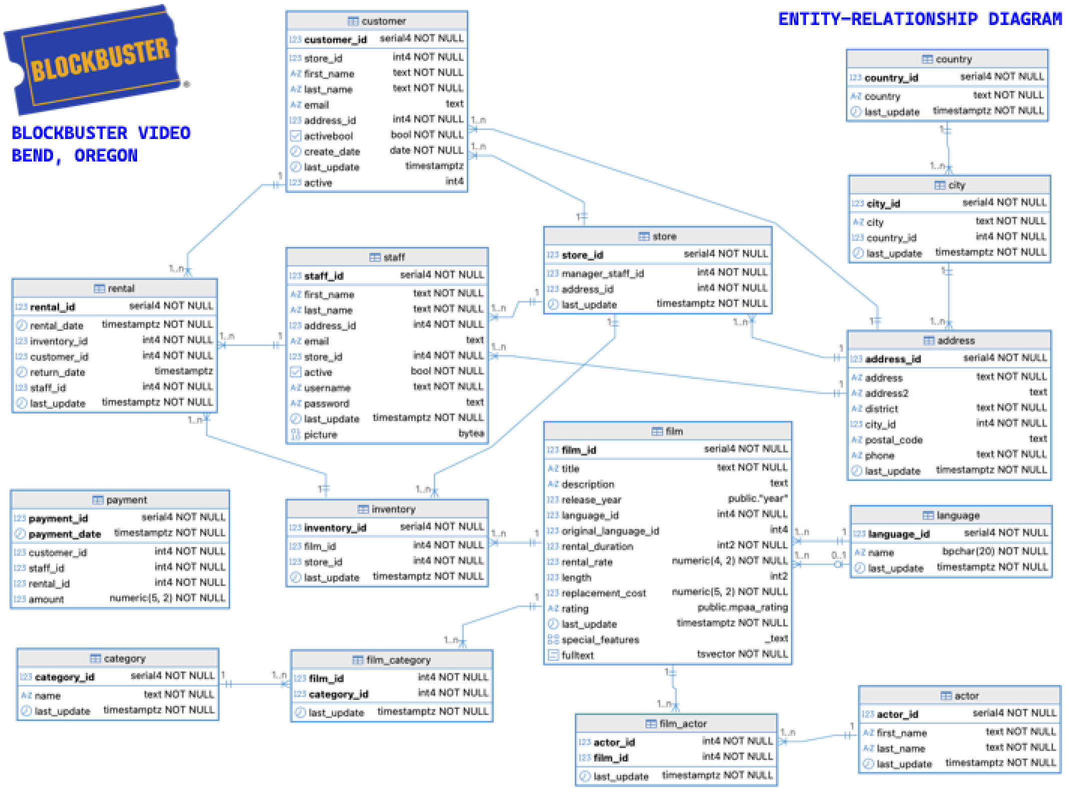
1.1.1.1 Commands

```
createdb blockbuster
psql -U postgres -d blockbuster -f blockbuster-schema.sql
psql -U postgres -d blockbuster -f blockbuster-data.sql
```

1.1.1.2 Expected Output

```
CREATE DATABASE
CREATE TABLE
...
INSERT 0 1
...
```

1.1.2 ERD Overview



customer	
customer_id	serial4 NOT NULL
store_id	int4 NOT NULL
first_name	text NOT NULL
last_name	text NOT NULL
email	text
address_id	int4 NOT NULL
activebool	bool NOT NULL
create_date	date NOT NULL
last_update	timestamptz
active	int4

country	
country_id	serial4 NOT NULL
country	text NOT NULL
last_update	timestamptz NOT NULL

city	
city_id	serial4 NOT NULL
city	text NOT NULL
country_id	int4 NOT NULL
last_update	timestamptz NOT NULL

rental	
rental_id	serial4 NOT NULL
rental_date	timestamptz NOT NULL
inventory_id	int4 NOT NULL
customer_id	int4 NOT NULL
return_date	timestamptz
staff_id	int4 NOT NULL
last_update	timestamptz NOT NULL

staff	
staff_id	serial4 NOT NULL
first_name	text NOT NULL
last_name	text NOT NULL
address_id	int4 NOT NULL
email	text
store_id	int4 NOT NULL
active	bool NOT NULL
username	text NOT NULL
password	text
last_update	timestamptz NOT NULL
picture	bytea

store	
store_id	serial4 NOT NULL
manager_staff_id	int4 NOT NULL
address_id	int4 NOT NULL
last_update	timestamptz NOT NULL

address	
address_id	serial4 NOT NULL
address	text NOT NULL
address2	text
district	text NOT NULL
city_id	int4 NOT NULL
postal_code	text
phone	text NOT NULL
last_update	timestamptz NOT NULL

film	
film_id	serial4 NOT NULL
title	text NOT NULL
description	text
release_year	public."year"
language_id	int4 NOT NULL
original_language_id	int4
rental_duration	int2 NOT NULL
rental_rate	numeric(4, 2) NOT NULL
length	int2
replacement_cost	numeric(5, 2) NOT NULL
rating	public.mpaa_rating
last_update	timestamptz NOT NULL
special_features	_text
fulltext	tsvector NOT NULL

payment	
payment_id	serial4 NOT NULL
payment_date	timestamptz NOT NULL
customer_id	int4 NOT NULL
staff_id	int4 NOT NULL
rental_id	int4 NOT NULL
amount	numeric(5, 2) NOT NULL

inventory	
inventory_id	serial4 NOT NULL
film_id	int4 NOT NULL
store_id	int4 NOT NULL
last_update	timestamptz NOT NULL

language	
language_id	serial4 NOT NULL
name	bpchar(20) NOT NULL
last_update	timestamptz NOT NULL

category	
category_id	serial4 NOT NULL
name	text NOT NULL
last_update	timestamptz NOT NULL

film_category	
film_id	int4 NOT NULL
category_id	int4 NOT NULL
last_update	timestamptz NOT NULL

film_actor	
actor_id	int4 NOT NULL
film_id	int4 NOT NULL
last_update	timestamptz NOT NULL

actor	
actor_id	serial4 NOT NULL
first_name	text NOT NULL
last_name	text NOT NULL
last_update	timestamptz NOT NULL

The ERD shows how tables connect through primary and foreign keys.

1.1.3 Crows-Foot Notation

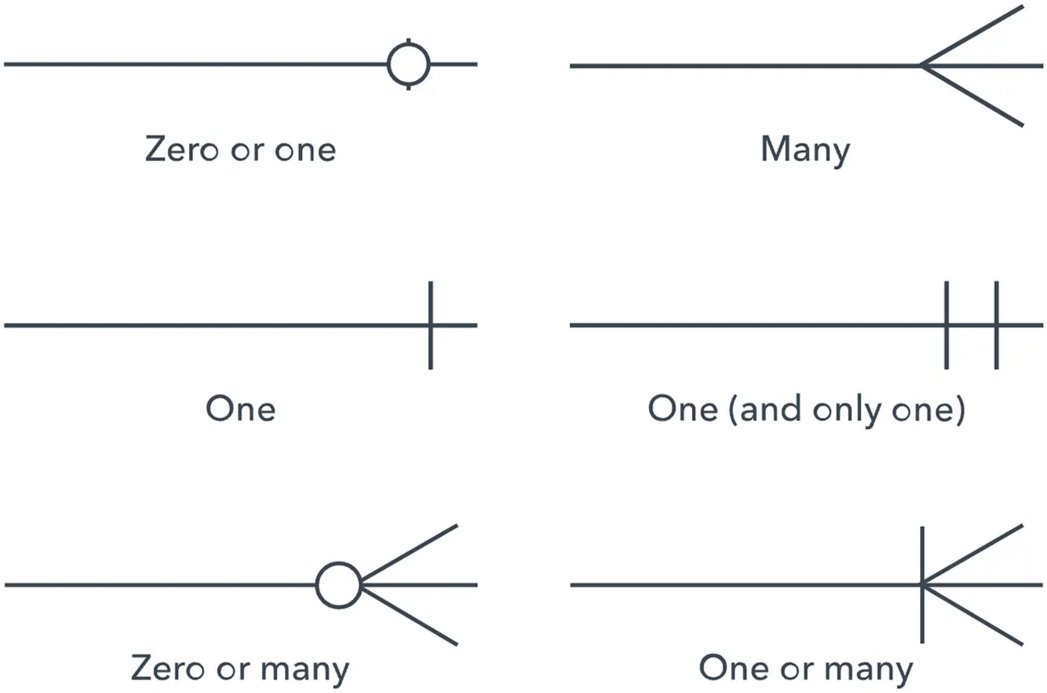


Figure 1: Crows-Foot Notation

1.1.4 Key Tables for Joins

The Blockbuster Bend database contains several interconnected table groups:

1.1.4.1 Film Data

- `film` - Movie titles and details
- `language` - Available languages
- `category` - Film genres (Action, Comedy, etc.)
- `film_category` - Links films to categories

1.1.4.2 People Data

- `actor` - Actor names
- `film_actor` - Links actors to films

- `customer` - Customer information
- `staff` - Employee records

1.1.4.3 Transaction Data

- `inventory` - Physical copies of films
- `rental` - Rental transactions
- `payment` - Payment records

1.1.4.4 Location Data

- `store` - Store locations
- `address` - Street addresses
- `city` - City names
- `country` - Country names

1.2 Why Joins Matter

1.2.1 Business Questions Require Multiple Tables

Most real questions span multiple tables:

- Which films were rented last month and by whom?
- Which customers have never rented a film?
- Which categories generate the most revenue at each store?
- Which actors appear in Action films?

Joins let us answer these questions by connecting tables.

1.2.2 The Problem with Separate Tables

Consider these two tables from our database:

1.2.2.1 film table (partial)

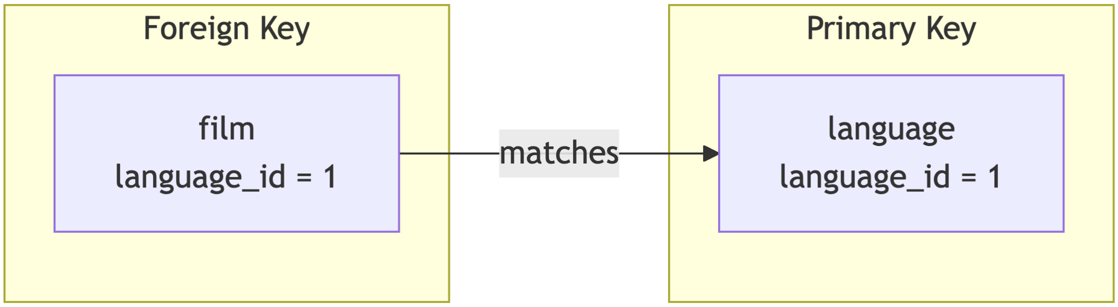
film_id	title	language_id
1	ACADEMY DINOSAUR	1
2	ACE GOLDFINGER	1
3	ADAPTATION HOLES	2

1.2.2.2 language table

language_id	name
1	English
2	Italian
3	Japanese
4	Mandarin
5	French
6	German

How do we see film titles with their language names in a single result?

1.2.3 Keys Enable Joins



Primary Key: Uniquely identifies each row in a table (e.g., `language_id` in `language`)

Foreign Key: References a primary key in another table (e.g., `language_id` in `film`)

1.3 Inner Joins

1.3.1 Inner Join Concept

An **inner join** returns only rows where the join condition is satisfied in both tables.

film language Result

Only matching rows are returned

- Films with a valid `language_id`
- Languages that have films assigned
- Unmatched rows are excluded

1.3.2 Inner Join with Sample Data

Let's trace through an inner join step by step:

1.3.2.1 Source Tables

film_id	title	language_id
1	ACADEMY DINOSAUR	1
2	ACE GOLDFINGER	1
3	ADAPTATION HOLES	2
4	AFFAIR PREJUDICE	6
5	AFRICAN EGG	4

language_id	name
1	English
2	Italian
3	Japanese
4	Mandarin
5	French
6	German

1.3.2.2 Matching Process

The database compares each film's `language_id` to the language table:

- ACADEMY DINOSAUR (language_id=1) matches English
- ACE GOLDFINGER (language_id=1) matches English
- ADAPTATION HOLES (language_id=2) matches Italian
- AFFAIR PREJUDICE (language_id=6) matches German
- AFRICAN EGG (language_id=4) matches Mandarin

1.3.2.3 Result

title	name
ACADEMY DINOSAUR	English
ACE GOLDFINGER	English
ADAPTATION HOLES	Italian
AFFAIR PREJUDICE	German
AFRICAN EGG	Mandarin

title	name

Note: Japanese and French have no films, so they do not appear.

1.3.3 SQL Query Structure

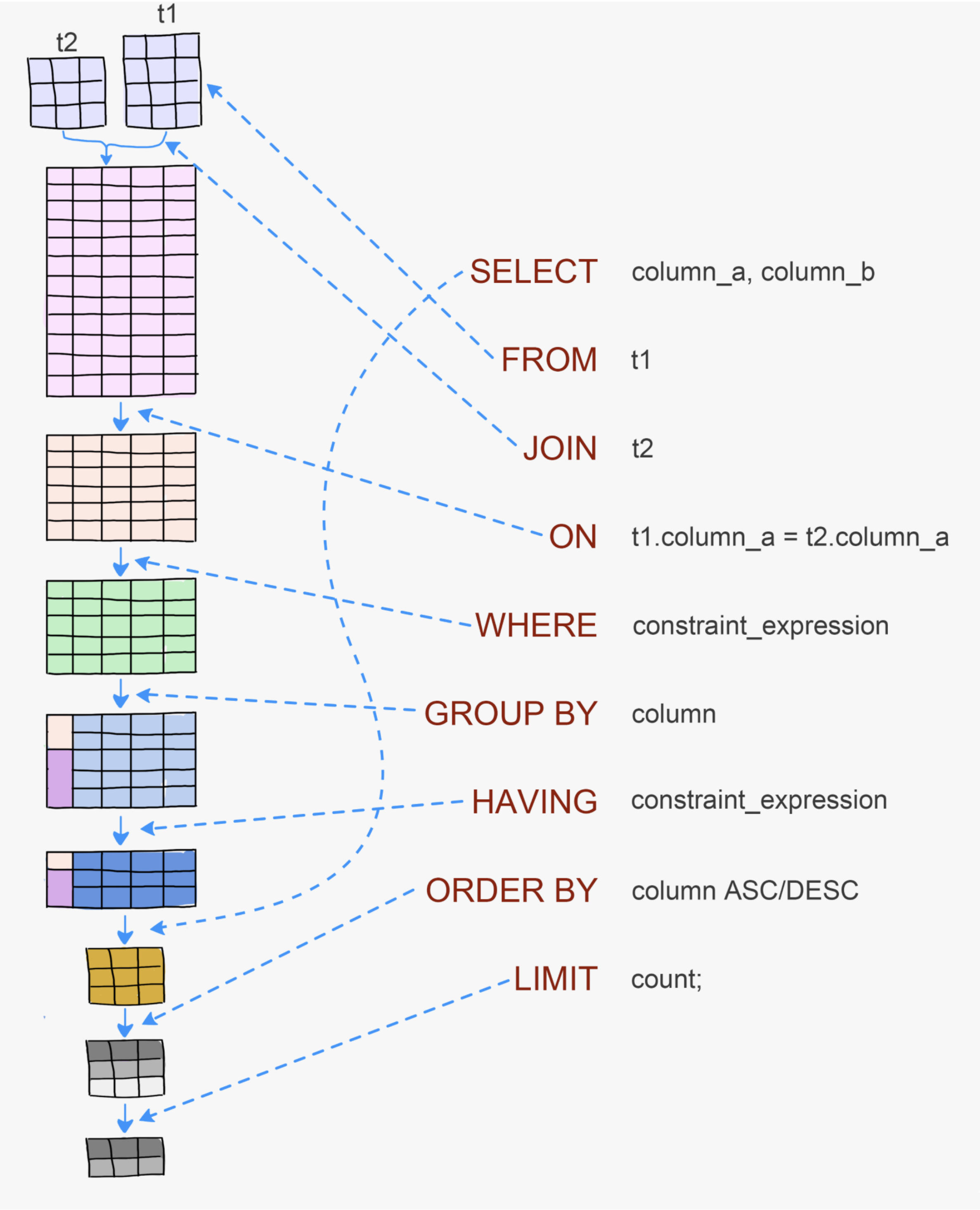

t1
t2
SELECT column_a, column_b
FROM t1
JOIN t2
ON t1.column_a = t2.column_a
WHERE constraint_expression
GROUP BY column
HAVING constraint_expression
ORDER BY column ASC/DESC
LIMIT count;

Figure 2: SQL Query Structure

1.3.4 Basic Inner Join Syntax

1.3.4.1 Query

```sql
SELECT
    f.title,
    l.name AS language
FROM film AS f
JOIN language AS l
    ON f.language_id = l.language_id
ORDER BY f.title
LIMIT 5;
```

1.3.4.2 Result

```
title                 | language
----------------------+----------
ACADEMY DINOSAUR      | English
ACE GOLDFINGER        | English
ADAPTATION HOLES      | Italian
AFFAIR PREJUDICE      | German
AFRICAN EGG           | Mandarin
```

1.3.4.3 Explanation

- `JOIN` is shorthand for `INNER JOIN`
- `AS f` and `AS l` create table aliases
- `ON` specifies the join condition
- We can reference columns from both tables

1.3.5 Table Aliases Keep Joins Readable

Without aliases, queries become verbose and harder to read:

1.3.5.1 With Aliases (Preferred)

```sql
SELECT
    f.title,
    f.release_year,
    l.name AS language
FROM film AS f
```

```sql
JOIN language AS l
    ON f.language_id = l.language_id
WHERE f.rating = 'PG'
ORDER BY f.title
LIMIT 3;
```

1.3.5.2 Without Aliases (Verbose)

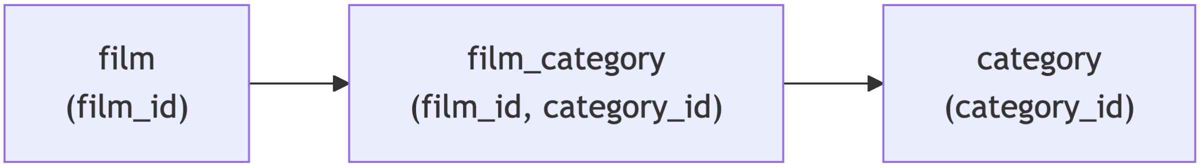
```sql
SELECT
    film.title,
    film.release_year,
    language.name AS language
FROM film
JOIN language
    ON film.language_id = language.language_id
WHERE film.rating = 'PG'
ORDER BY film.title
LIMIT 3;
```

1.3.5.3 Result

```
title                | release_year | language
-------------------+--------------+----------
ACADEMY DINOSAUR   | 2012         | English
AGENT TRUMAN       | 2010         | English
ALASKA PHANTOM     | 2016         | English
```

1.3.6 Multi-Table Joins: Film to Category

Films connect to categories through the `film_category` bridge table:



1.3.6.1 Query

```sql
SELECT
    f.title,
    c.name AS category
FROM film AS f
JOIN film_category AS fc
    ON f.film_id = fc.film_id
JOIN category AS c
    ON fc.category_id = c.category_id
WHERE c.name = 'Action'
ORDER BY f.title
LIMIT 5;
```

1.3.6.2 Result

```
| title             | category |
| ----------------- | -------- |
| ACE GOLDFINGER    | Action   |
| ADAPTATION HOLES  | Action   |
| AIRPLANE SIERRA   | Action   |
| ALASKA PHANTOM    | Action   |
| ANGELS LIFE       | Action   |
```

1.3.6.3 Explanation

- First join connects `film` to `film_category`
- Second join connects `film_category` to `category`
- The bridge table handles the many-to-many relationship

1.3.7 Film to Actor Join

The `film_actor` bridge table connects films and actors:

1.3.7.1 Query

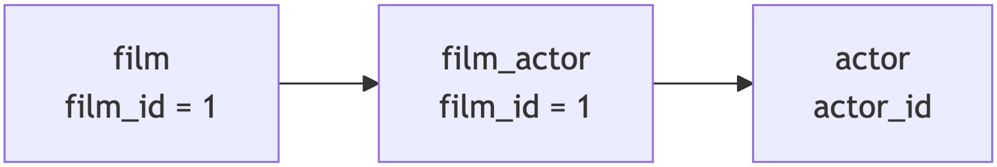
```sql
SELECT
    f.title,
    a.first_name,
    a.last_name
FROM film AS f
```

```
JOIN film_actor AS fa
    ON f.film_id = fa.film_id
JOIN actor AS a
    ON fa.actor_id = a.actor_id
WHERE f.title = 'ACADEMY DINOSAUR'
ORDER BY a.last_name, a.first_name;
```

1.3.7.2 Result

```
| title            | first_name | last_name |
| ---------------- | ---------- | --------- |
| ACADEMY DINOSAUR | JOHNNY     | CAGE      |
| ACADEMY DINOSAUR | ROCK       | DUKAKIS   |
| ACADEMY DINOSAUR | CHRISTIAN  | GABLE     |
| ACADEMY DINOSAUR | PENELOPE   | GUINESS   |
| ACADEMY DINOSAUR | MARY       | KEITEL    |
| ACADEMY DINOSAUR | OPRAH      | KILMER    |
| ACADEMY DINOSAUR | WARREN     | NOLTE     |
| ACADEMY DINOSAUR | SANDRA     | PECK      |
| ACADEMY DINOSAUR | MENA       | TEMPLE    |
| ACADEMY DINOSAUR | LUCILLE    | TRACY     |
```

1.3.7.3 Data Flow



1.3.8 Practice: Customer Location Join

1.3.8.1 Challenge

Write a query that returns:

- `customer_id`
- `first_name`
- `last_name`
- `city`

Join `customer` to `address` to `city`. Order by `last_name`, then `first_name`. Limit to 5 rows.

1.3.8.2 Solution

```sql
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    ci.city
FROM customer AS c
JOIN address AS a
    ON c.address_id = a.address_id
JOIN city AS ci
    ON a.city_id = ci.city_id
ORDER BY c.last_name, c.first_name
LIMIT 5;
```

1.3.8.3 Result

```
| customer_id | first_name | last_name | city                    |
| ----------- | ---------- | --------- | ----------------------- |
| 505         | RAFAEL     | ABNEY     | Talavera                |
| 504         | NATHANIEL  | ADAM      | Joliet                  |
| 36          | KATHLEEN   | ADAMS     | Arak                    |
| 96          | DIANA      | ALEXANDER | Augusta-Richmond County |
| 470         | GORDON     | ALLARD    | Hodeida                 |
```

1.3.9 Filtering: ON vs WHERE

Use `ON` for join conditions and `WHERE` for row filters:

1.3.9.1 Correct Approach

```sql
SELECT
    f.title,
    f.rating,
    c.name AS category
FROM film AS f
JOIN film_category AS fc
    ON f.film_id = fc.film_id
JOIN category AS c
    ON fc.category_id = c.category_id
WHERE f.rating = 'PG'
```

```sql
    AND c.name = 'Comedy'
ORDER BY f.title
LIMIT 5;
```

1.3.9.2 Result

```
| title            | rating | category |
| ---------------- | ------ | -------- |
| ALI FOREVER      | PG     | Comedy   |
| BLACKOUT PRIVATE | PG     | Comedy   |
| CAROL TEXAS      | PG     | Comedy   |
| CHARADE DUFFEL   | PG     | Comedy   |
| DISCIPLE MOTHER  | PG     | Comedy   |
```

1.3.9.3 Why This Matters

- `ON` defines how tables relate
- `WHERE` filters the joined result
- Putting filters in `ON` can produce unexpected results with outer joins

1.4 Outer Joins

1.4.1 When Inner Joins Are Not Enough

Inner joins exclude rows without matches. Sometimes we need to see unmatched rows:

- Which languages have no films?
- Which customers have never rented?
- Which inventory items have never been rented?

Outer joins preserve unmatched rows.

1.4.2 Left Join Concept

A **left join** keeps all rows from the left table, even without matches.

All A B LEFT JOIN

All left table rows returned

- Matching rows show data from both tables
- Non-matching rows show `NULL` for right table columns

1.4.3 Left Join with Film and Inventory Data

Let's say that we wish to list all films that we do not have a copy of in our inventory. In other words, we want to find all films that are not in the inventory table.

1.4.3.1 Source Tables

film (left table)

film_id	title
1	ACADEMY DINOSAUR
2	ACE GOLDFINGER
3	ADAPTATION HOLES

... (1000 rows)

inventory (right table)

inventory_id	film_id
1	1
2	2
3	3
4	4

... (4581 rows)

1.4.3.2 Query

```sql
SELECT
  f.film_id,
  f.title,
  i.inventory_id
FROM film f
LEFT JOIN inventory i
  ON i.film_id = f.film_id
WHERE i.inventory_id IS NULL
ORDER BY f.title;
```

1.4.3.3 Left Join Result

film_id	title	inventory_id
14	ALICE FANTASIA	NULL
33	APOLLO TEEN	NULL
36	ARGONAUTS TOWN	NULL
38	ARK RIDGEMONT	NULL

… (42 rows)

1.4.3.4 Explanation

- `LEFT JOIN` keeps all films
- Films without a matching inventory get `NULL` values
- `WHERE i.inventory_id IS NULL` filters to only unmatched rows

1.4.4 Right Join Concept

A **right join** keeps all rows from the right table, even without matches.

A All B RIGHT JOIN

All right table rows returned

- Equivalent to a left join with tables swapped
- Less common in practice

1.4.5 Full Outer Join Concept

A **full outer join** keeps all rows from both tables.

All A All B FULL OUTER JOIN

All rows from both tables

- Unmatched left rows show NULL for right columns
- Unmatched right rows show NULL for left columns
- Useful for finding all mismatches

1.4.6 What Would Left Join and Right Join Look Like for these Tables?

Table A

id	value_a
1	Apple
2	Banana
3	Cherry

Table B

id	value_b
2	Two
3	Three
4	Four

1.4.7 Full Outer Join Example

1.4.7.1 Sample Data

Table A

id	value_a
1	Apple
2	Banana
3	Cherry

Table B

id	value_b
2	Two
3	Three
4	Four

1.4.7.2 Full Outer Join Result

a.id	value_a	b.id	value_b
1	Apple	NULL	NULL
2	Banana	2	Two
3	Cherry	3	Three
NULL	NULL	4	Four

1.4.7.3 Identifying Unmatched Rows

```
-- Rows only in A
WHERE b.id IS NULL

-- Rows only in B
WHERE a.id IS NULL

-- Rows only in one table (not both)
WHERE a.id IS NULL OR b.id IS NULL
```

1.4.8 Outer Join Comparison Summary

Join Type	Left Table	Right Table	Use Case
INNER JOIN	Only matched	Only matched	Standard queries
LEFT JOIN	All rows	Only matched	Find unmatched in right
RIGHT JOIN	Only matched	All rows	Find unmatched in left
FULL OUTER JOIN	All rows	All rows	Find all unmatched

1.4.9 Practice: Unrented Inventory

1.4.9.1 Challenge

Find inventory items that have never been rented.

Return:

- `inventory_id`
- `film_id`
- `title`
- `store_id`

Order by `store_id`, then `inventory_id`. Limit to 5 rows.

1.4.9.2 Solution

```sql
SELECT
    i.inventory_id,
    i.film_id,
    f.title,
    i.store_id
FROM inventory AS i
LEFT JOIN rental AS r
    ON i.inventory_id = r.inventory_id
JOIN film AS f
    ON i.film_id = f.film_id
WHERE r.rental_id IS NULL
ORDER BY i.store_id, i.inventory_id
LIMIT 5;
```

1.4.9.3 Result

```
inventory_id | film_id | title              | store_id
-------------+---------+--------------------+----------
1            | 1       | ACADEMY DINOSAUR   | 1
2            | 1       | ACADEMY DINOSAUR   | 1
...
```

1.5 Cross Joins

1.5.1 Cross Join Concept

A **cross join** (Cartesian product) returns every combination of rows from both tables.

3 rows 4 rows Result: 3 x 4 = 12 rows

No join condition

- Every row in A pairs with every row in B
- Result size = rows(A) x rows(B)
- Can produce very large results

1.5.2 Cross Join with Sample Data

1.5.2.1 Source Tables

store

store_id
1
2

category (partial)

category_id	name
1	Action
5	Comedy
7	Drama

1.5.2.2 Cross Join Result

store_id	name
1	Action
1	Comedy
1	Drama
2	Action
2	Comedy
2	Drama

Every store paired with every category (2 x 3 = 6 rows).

1.5.3 Cross Join Use Case: Store-Category Grid

Generate a planning grid for all store-category combinations:

1.5.3.1 Query

```sql
SELECT
    s.store_id,
    c.name AS category,
    0 AS planned_inventory
FROM store AS s
CROSS JOIN category AS c
WHERE s.store_id IN (1, 2)
ORDER BY s.store_id, c.name
LIMIT 10;
```

1.5.3.2 Result

```
store_id | category    | planned_inventory
---------+-------------+------------------
1        | Action      | 0
1        | Animation   | 0
1        | Children    | 0
1        | Classics    | 0
1        | Comedy      | 0
1        | Documentary| 0
1        | Drama       | 0
1        | Family      | 0
1        | Foreign     | 0
1        | Games       | 0
```

1.5.3.3 Use Cases

- Inventory planning templates
- Report scaffolding
- Generating test data
- Date/category combinations for analysis

1.5.4 Cross Join Caution

Cross joins can create enormous result sets:

Table A Rows	Table B Rows	Result Rows
100	100	10,000

Table A Rows	Table B Rows	Result Rows
1,000	1,000	1,000,000
10,000	10,000	100,000,000

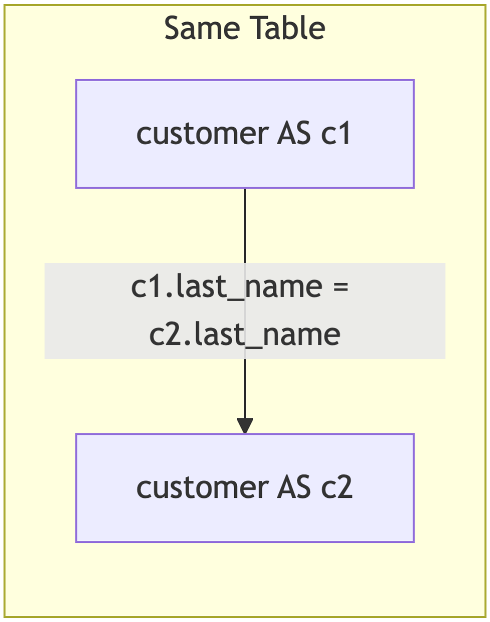
Always use `WHERE` or `LIMIT` when exploring cross joins.

1.6 Self Joins

1.6.1 Self Join Concept

A **self join** joins a table to itself. This is useful when:

- Comparing rows within the same table
- Finding hierarchical relationships
- Detecting duplicates or related records



1.6.2 Finding Customers with Same Last Name

1.6.2.1 Query

```sql
SELECT
    c1.customer_id AS customer_1,
    c1.first_name AS first_1,
    c1.last_name,
    c2.customer_id AS customer_2,
    c2.first_name AS first_2
FROM customer AS c1
JOIN customer AS c2
    ON c1.last_name = c2.last_name
    AND c1.customer_id < c2.customer_id
ORDER BY c1.last_name, c1.customer_id
LIMIT 5;
```

1.6.2.2 Result

```
customer_1 | first_1   | last_name | customer_2 | first_2
-----------+-----------+-----------+------------+---------
318        | BRIAN     | WYMAN     | 412        | JOHN
...
```

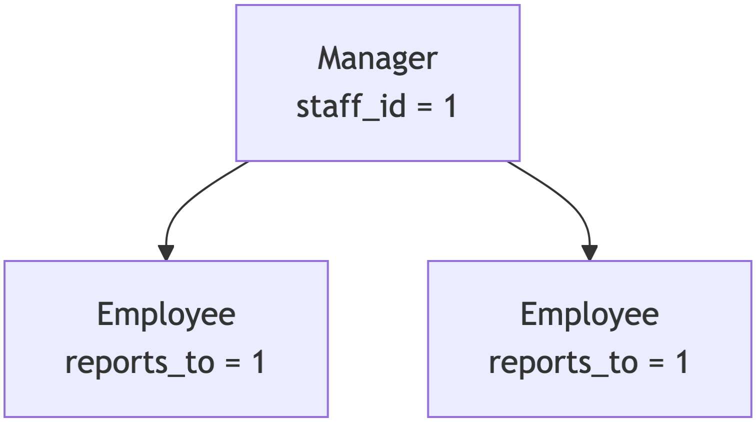
1.6.2.3 Explanation

- Table aliased as both `c1` and `c2`
- `c1.customer_id < c2.customer_id` prevents duplicate pairs
- Without this condition, we would get (A,B) and (B,A)

1.6.3 Self Join for Hierarchical Data

Self joins work well for parent-child relationships:

1.6.3.1 Concept



1.6.3.2 Query Pattern

```sql
-- If staff had a reports_to column:
SELECT
    e.first_name AS employee,
    m.first_name AS manager
FROM staff AS e
JOIN staff AS m
    ON e.reports_to = m.staff_id;
```

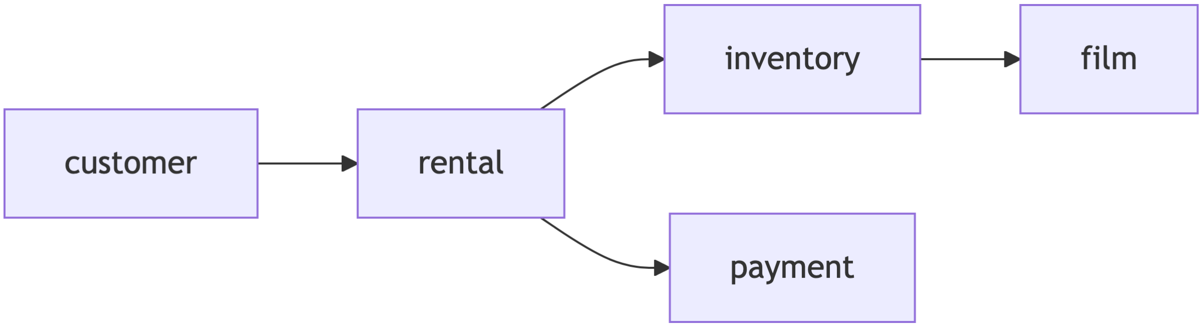
1.6.3.3 Applications

- Organization charts
- Category hierarchies
- Reply threads in forums

1.7 Multi-Table Join Patterns

1.7.1 The Rental Transaction Chain

Tracking a rental requires joining multiple tables:



1.7.2 Complete Rental Query

1.7.2.1 Query

```sql
SELECT
    c.first_name || ' ' || c.last_name AS customer,
    f.title,
    r.rental_date::date AS rented,
    r.return_date::date AS returned,
    p.amount
FROM customer AS c
JOIN rental AS r
    ON c.customer_id = r.customer_id
JOIN inventory AS i
    ON r.inventory_id = i.inventory_id
JOIN film AS f
    ON i.film_id = f.film_id
JOIN payment AS p
    ON r.rental_id = p.rental_id
ORDER BY r.rental_date DESC
LIMIT 5;
```

1.7.2.2 Result

```
customer          | title              | rented     | returned   | amount
------------------+--------------------+------------+------------+-------
AUSTIN CINTRON    | SOMETHING DUCK     | 2022-07-27 | 2022-08-02 | 4.99
AUSTIN CINTRON    | TITANS JERK        | 2022-07-27 | 2022-08-01 | 4.99
AUSTIN CINTRON    | SUNRISE LEAGUE     | 2022-07-27 | 2022-07-28 | 2.99
...
```

1.7.2.3 Join Path

1. customer to rental via customer_id
2. rental to inventory via inventory_id
3. inventory to film via film_id
4. rental to payment via rental_id

1.7.3 Actor Filmography Query

1.7.3.1 Query

```sql
SELECT
    a.first_name,
    a.last_name,
    f.title,
    f.release_year,
    c.name AS category
FROM actor AS a
JOIN film_actor AS fa
    ON a.actor_id = fa.actor_id
JOIN film AS f
    ON fa.film_id = f.film_id
JOIN film_category AS fc
    ON f.film_id = fc.film_id
JOIN category AS c
    ON fc.category_id = c.category_id
WHERE a.last_name = 'GUINESS'
ORDER BY f.release_year, f.title;
```

1.7.3.2 Result

```
first_name | last_name | title              | release_year | category
-----------+-----------+--------------------+--------------+----------
PENELOPE   | GUINESS   | ACADEMY DINOSAUR   | 2012         | Documentary
PENELOPE   | GUINESS   | ANACONDA CONFESSIONS| 2020        | Animation
...
```

1.7.3.3 Query Structure

Five tables joined through their foreign key relationships.

1.7.4 Join on Multiple Columns

Sometimes joins need multiple columns to match correctly:

1.7.4.1 Query

```sql
SELECT
    p.payment_id,
    p.customer_id,
    p.rental_id,
    p.amount,
    r.rental_date::date
FROM payment AS p
JOIN rental AS r
    ON p.rental_id = r.rental_id
    AND p.customer_id = r.customer_id
WHERE p.customer_id = 1
ORDER BY r.rental_date
LIMIT 5;
```

1.7.4.2 Result

```
payment_id | customer_id | rental_id | amount | rental_date
-----------+-------------+-----------+--------+------------
17503      | 1           | 76        | 2.99   | 2022-05-25
17504      | 1           | 573       | 0.99   | 2022-05-28
17505      | 1           | 1185      | 5.99   | 2022-06-15
...
```

1.7.4.3 When to Use Multiple Columns

- Composite keys
- Data validation
- Ensuring correct matches in denormalized data

1.8 Common Pitfalls

1.8.1 Ambiguous Column Names

When two tables have the same column name:

1.8.1.1 Error

```sql
SELECT
    customer_id,  -- Ambiguous!
    first_name,
    last_name
FROM customer
JOIN rental
    ON customer.customer_id = rental.customer_id;
```

```
ERROR: column reference "customer_id" is ambiguous
```

1.8.1.2 Fixed

```sql
SELECT
    c.customer_id,  -- Qualified with alias
    c.first_name,
    c.last_name
FROM customer AS c
JOIN rental AS r
    ON c.customer_id = r.customer_id;
```

1.8.2 Missing Join Conditions

Forgetting the `ON` clause creates a cross join:

1.8.2.1 Problem

```sql
-- This creates a cross join!
SELECT f.title, c.name
FROM film AS f, category AS c
LIMIT 5;
```

Every film paired with every category (1000 x 16 = 16,000 rows).

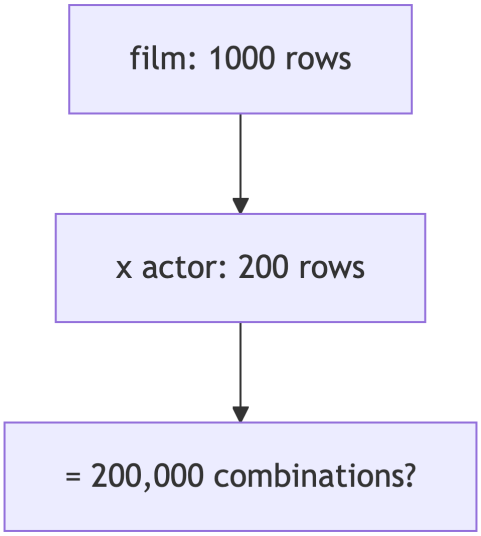
1.8.2.2 Solution

Always use explicit `JOIN ... ON` syntax:

```sql
SELECT f.title, c.name
FROM film AS f
JOIN film_category AS fc ON f.film_id = fc.film_id
JOIN category AS c ON fc.category_id = c.category_id
LIMIT 5;
```

1.8.3 Cartesian Explosion

Adding more tables can multiply result sizes:



Prevention:

- Check join conditions carefully
- Use `COUNT(*)` before `SELECT *`
- Add `LIMIT` during development

1.8.4 Join Verification Checklist

Before running a complex join:

1. Are all join conditions specified?
2. Are column references qualified with aliases?
3. Is this an inner or outer join?

4. Could any join create a Cartesian product?
5. Have I tested with `LIMIT` first?

1.9 Practice Problems

1.9.1 Practice 1: Store Revenue by Category

1.9.1.1 Challenge

For each store, find total revenue by film category.

Return:

- `store_id`
- `category`
- `total_revenue`

Order by `store_id`, then `total_revenue` descending.

1.9.1.2 Solution

```sql
SELECT
    i.store_id,
    c.name AS category,
    SUM(p.amount) AS total_revenue
FROM payment AS p
JOIN rental AS r ON p.rental_id = r.rental_id
JOIN inventory AS i ON r.inventory_id = i.inventory_id
JOIN film AS f ON i.film_id = f.film_id
JOIN film_category AS fc ON f.film_id = fc.film_id
JOIN category AS c ON fc.category_id = c.category_id
GROUP BY i.store_id, c.name
ORDER BY i.store_id, total_revenue DESC;
```

1.9.1.3 Result

```
store_id | category     | total_revenue
---------+--------------+--------------
1        | Sports       | 4892.19
1        | Sci-Fi       | 4756.98
1        | Animation    | 4656.30
...
```

1.9.2 Practice 2: Actors Without Films

1.9.2.1 Challenge

Find any actors who have no films in the database.

Return:

- actor_id
- first_name
- last_name

Order by last_name, first_name.

1.9.2.2 Solution

```sql
SELECT
    a.actor_id,
    a.first_name,
    a.last_name
FROM actor AS a
LEFT JOIN film_actor AS fa
    ON a.actor_id = fa.actor_id
WHERE fa.film_id IS NULL
ORDER BY a.last_name, a.first_name;
```

1.9.2.3 Result

```
actor_id | first_name | last_name
---------+------------+----------
(0 rows - all actors have films in this database)
```

1.9.3 Practice 3: Customer Rental History

1.9.3.1 Challenge

Create a rental history for customer MARY SMITH (customer_id = 1).

Return:

- rental_date
- title
- category

- amount

Order by rental_date descending. Limit to 10 rows.

1.9.3.2 Solution

```
SELECT
    r.rental_date::date,
    f.title,
    c.name AS category,
    p.amount
FROM customer AS cu
JOIN rental AS r ON cu.customer_id = r.customer_id
JOIN payment AS p ON r.rental_id = p.rental_id
JOIN inventory AS i ON r.inventory_id = i.inventory_id
JOIN film AS f ON i.film_id = f.film_id
JOIN film_category AS fc ON f.film_id = fc.film_id
JOIN category AS c ON fc.category_id = c.category_id
WHERE cu.customer_id = 1
ORDER BY r.rental_date DESC
LIMIT 10;
```

1.10 Key Takeaways

1.10.1 Join Type Summary

Join Type	Returns	NULL Handling
INNER JOIN	Only matching rows	No NULLs from join
LEFT JOIN	All left + matched right	NULLs for unmatched right
RIGHT JOIN	All right + matched left	NULLs for unmatched left
FULL OUTER JOIN	All rows from both	NULLs for unmatched on both sides
CROSS JOIN	All combinations	No join condition
SELF JOIN	Table joined to itself	Depends on join type used

1.10.2 Best Practices

1. **Always use table aliases** for readability
2. **Qualify all column references** to avoid ambiguity

3. **Use explicit JOIN syntax** instead of comma-separated tables
4. **Put join conditions in ON**, filters in WHERE
5. **Test with LIMIT** before running full queries
6. **Verify row counts** to catch Cartesian products

1.10.3 Exit Ticket

Write a query that answers:

Which films were rented in 2022 by customers from store 1?

Return the customer name, film title, and rental date.

Be ready to share your join path and key columns.

1.11 References

1.11.1 References

1. Forta, B. (2024). *SQL in 10 Minutes a Day* (6th ed.). Addison-Wesley.
2. PostgreSQL Documentation. *SELECT - Joins.* https://www.postgresql.org/docs/current/queries-table-expressions.html
3. Silberschatz, A., Korth, H., & Sudarshan, S. (2019). *Database System Concepts* (7th ed.). McGraw-Hill.