Databases

https://www.cst.cam.ac.uk/teaching/2627/Databases/

Dr. Martin Kleppmann
martin.kleppmann@cst.cam.ac.uk

University of Cambridge
Computer Science Tripos, Part IA

This work is published under a
Creative Commons BY-SA license.

Databases everywhere

Databases are almost everywhere, but often invisible:

- **Your phone**: messages, contacts, photo metadata (SQLite on Android/iOS: 10^{12} databases worldwide?)
- **Websites you log in to**: accounts, sessions, posts, orders
- **University**: student records, timetables, exam results
- **Banks**: accounts and every transaction ever made
- **Science**: genomes, particle collisions, sky surveys

Much modern software is getting data in and out of databases.

You have already used dozens of databases today.

This course

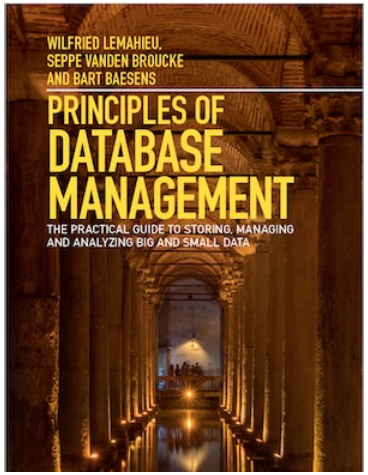
- How to **use** databases effectively
  $\rightarrow$ this course
- How databases work **internally**
  $\rightarrow$ Part IA Algorithms,
  Part IB Concurrent and Distributed Systems,
  Part II Cloud Computing, & others

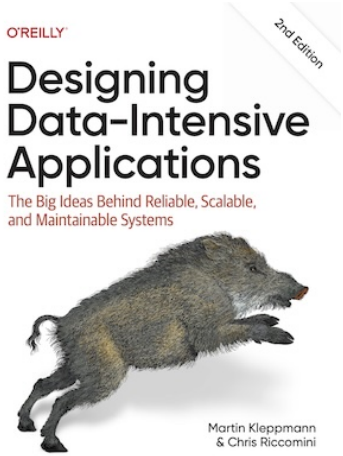
Query languages: SQL, relational algebra, Datalog, SQL/PGQ, JSONPath

Learning to **ask the right questions**:

- Trade-offs between different data representations
- Choosing between the many database systems that exist
- Thinking about database queries using different languages

Recommended reading





- Lemahieu, vanden Broucke, and Baesens. "**Principles of Database Management**". Cambridge University Press 2018
- Kleppmann. "**Designing Data-Intensive Applications**", second edition. O'Reilly 2026

Both are freely available online for Cambridge students. No need to buy!

Databases practicals

We provide example databases and practical exercises.

New in 2026/27: practicals are not formally assessed!
(No ticks!)

However:

- Please complete the exercises and discuss them with your supervisor
- Feel free to play around with the example databases and try queries of your own
- Getting hands-on experience with the practicals is the best exam preparation
- Letting AI do the work just harms yourself

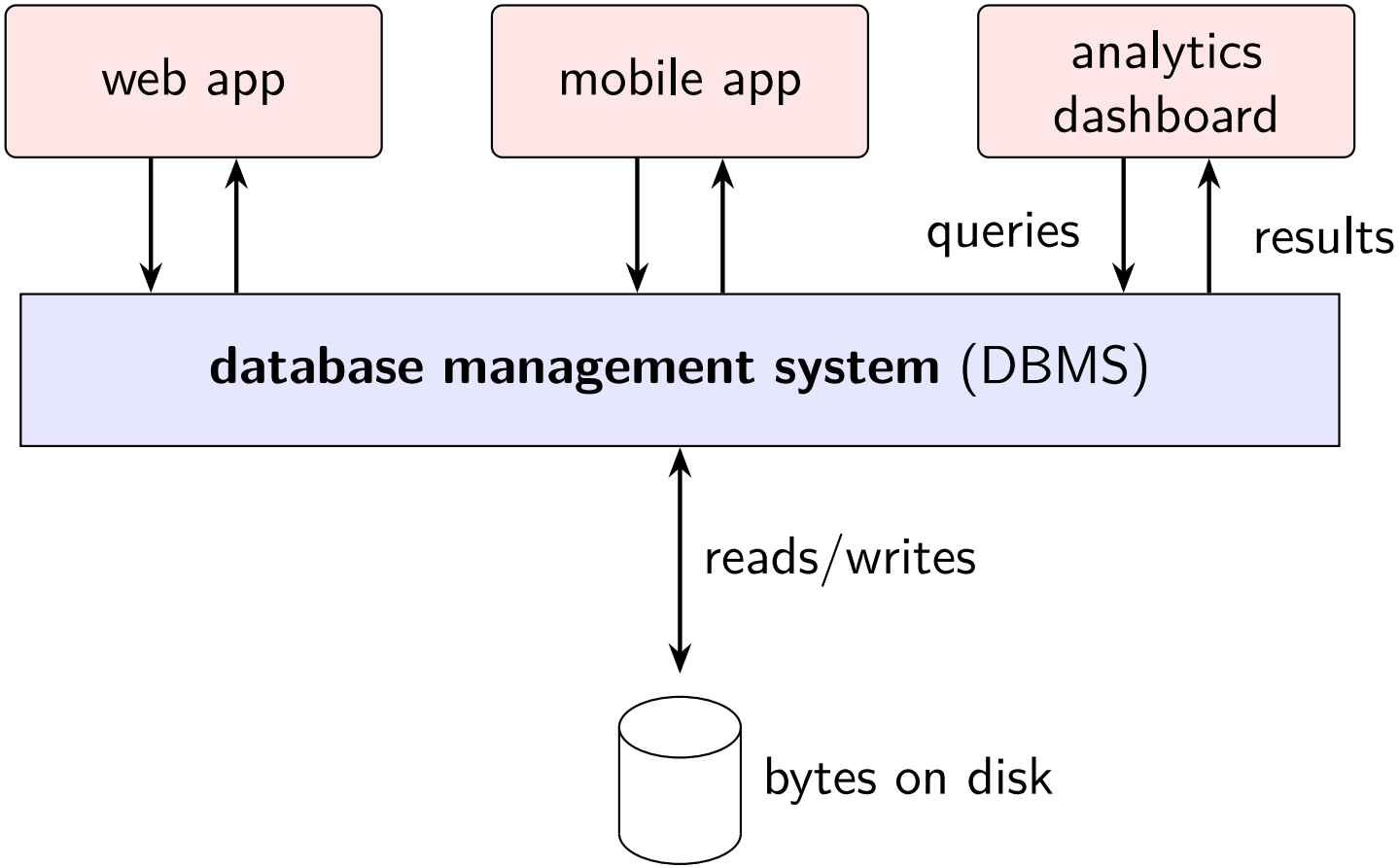
Is this a database?

```
movies = [
    {"title": "Alien", "year": 1979, "minutes": 117},
    {"title": "Se7en", "year": 1995, "minutes": 127},
    {"title": "Toy Story", "year": 1995, "minutes": 81},
]

# Which movies came out in 1995?
[movie for movie in movies if movie["year"] == 1995]
```

Questions:

- **Persistence**: what if you restart the program?
- **Sharing**: what if there are many users?
- **Scale**: what if there's more data than fits in memory?
- **Querying**: how do you answer unanticipated questions?
- **Integrity**: what if the computer crashes?

The database as abstraction layer



Every access goes through the DBMS, which can enforce rules, handle concurrent access, and recover from crashes.

Why not just use a file?

A file gives persistence; a database provides more structure.

File	**Database**
A sequence of bytes, you define the format	Predefined **schema** that data must conform to
Find a record: read the whole file	**Index**: efficient search for values
Change a record: rewrite everything after it	Efficient **insert, update, delete** of any record
Crash mid-write $\Rightarrow$ corrupted file	A **transaction** applies entirely, or not at all
Two writers at once $\Rightarrow$ corrupted file	**Concurrency control** stops them interfering
A new question: edit the program	Write a new **query**

Key-value stores

Two operations: `put(key, value)` and `get(key)`.
Similar to a Python `dict`:

```
movies_by_id = {}                          # the store

movies_by_id["tt0078748"] = {              # put(key, value)
    "title": "Alien", "year": 1979, "minutes": 117
}

movies_by_id["tt0078748"]                  # get(key)
```

...but it's persistent (on disk) and can be very large.

You can look up values by key, **and by nothing else**:

- To the store, keys and values are just bytes
- Can't search inside values
- Key is some kind of unique identifier (ID)

Searching for something that isn't the key

Our query from earlier:

```
# Which movies came out in 1995?
[movie for movie in movies if movie["year"] == 1995]

# ...which is the same as:
results = []
for movie in movies:
    if movie["year"] == 1995:
        results.append(movie)
```

Full scan over all movies.
Cost: $O(n)$ in a database of n movies

Fine for $n = 10^3$, not good for $n = 10^6$.

Key-value stores don't handle this well.

Indexes

Keep a separate structure containing, for each year, the IDs of movies released in that year:

```
movies_by_id = {                          # the data
    "tt0078748": {"title": "Alien", "year": 1979},
    "tt0114369": {"title": "Se7en", "year": 1995},
    "tt0114709": {"title": "Toy Story", "year": 1995},
}

movies_by_year = {                        # the index
    1979: ["tt0078748"],
    1995: ["tt0114369", "tt0114709"],
}

# Which movies came out in 1995?
[movies_by_id[key] for key in movies_by_year[1995]]
```

One lookup in the index, then one per movie ID.
Typically use B-tree for index: lookup cost $O(\log n)$

Keeping an index up to date

Now you have two separate data structures that you have to carefully keep consistent:

```
def add_movie(movie_id, movie):
    movies_by_id[movie_id] = movie
    year = movie["year"]
    movies_by_year.setdefault(year, []).append(movie_id)

add_movie("tt0113277", {"title": "Heat", "year": 1995})
# movies_by_year[1995] now lists three IDs, not two
```

Databases maintain indexes automatically!

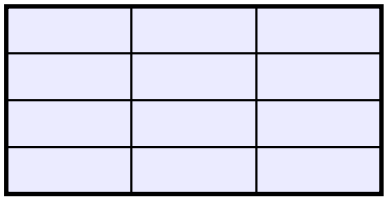
But:

- Index takes space and requires more work on writes
- Hence, databases create only explicitly requested indexes
- Choosing indexes requires knowledge of query patterns

Summary: four ways to keep data

	Data structure	**File**	**Key-value store**	**Database**
Persistent across restarts	no	yes	yes	yes
Shared between users	no	awkwardly	yes	yes
Find by key	you code it	you code it	yes	yes
Find by other fields (indexes)	you code it	you code it	no	yes
All-or-nothing updates	no	you code it	sometimes	yes
Checks data consistency	no	no	no	yes

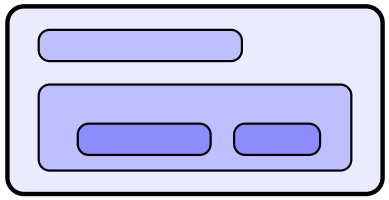
Data models



Relational

Tables with rows and columns

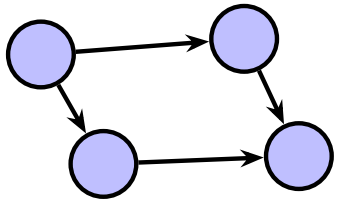
e.g. PostgreSQL, SQLite, DuckDB



Document

JSON: records nested inside one another

e.g. MongoDB, CouchDB



Graph

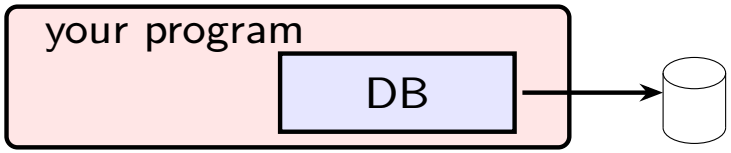
Vertices joined by edges

e.g. Neo4j, Memgraph

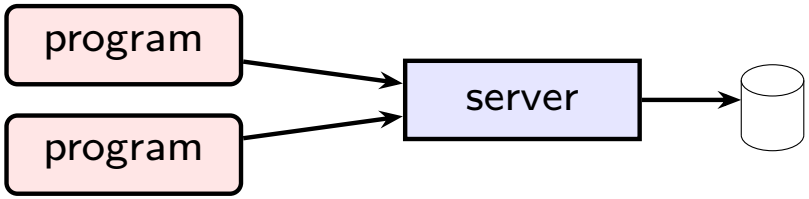
Also: time series, vectors, geospatial, scientific arrays...

Where the database runs

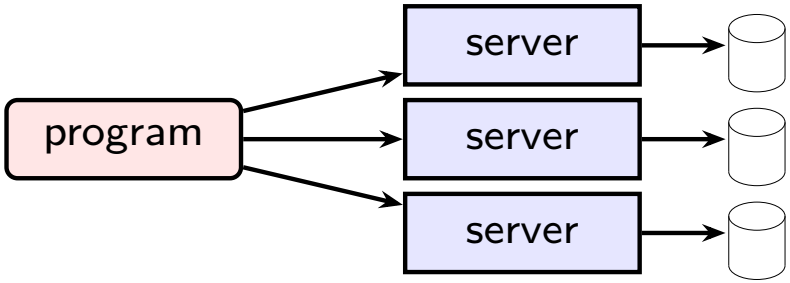
Embedded (SQLite, DuckDB) megabytes to gigabytes



Client-server (PostgreSQL, MySQL) gigabytes to terabytes



Distributed (Snowflake, BigQuery) terabytes to petabytes



Row-oriented and column-oriented

the same **logical** table...

movie_id	title	year
tt0012349	The Kid	1921
tt0078748	Alien	1979

...two **physical** layouts

```
tt0012349|The Kid|1921
tt0078748|Alien|1979
```

row-oriented

```
tt0012349,tt0078748
The Kid,Alien
1921,1979
```

column-oriented

Often update individual records ⇒ row-oriented is better
Often summarise many records ⇒ column-oriented is better

Query languages covered in this course

SQL (Structured Query Language)

- Very popular, you have to know it
- Originally relational, but extensions for JSON, XML, graphs, etc. have been added
- Corresponds roughly to **relational algebra**

Datalog

- Also relational, much less well known
- Learn it to understand recursive queries

SQL/PGQ (graphs)

- $\approx$ Cypher, absorbed into SQL

JSON (documents)

- Not a query language per se, but a useful data model

Declarative languages

SQL and the other query languages in this course are **declarative**:

```
SELECT * FROM movies WHERE year = 1995;
```

Describe the properties of the data you want; not how to compute it. Database automatically chooses indexes to use.

In Python you write an **algorithm** to compute the data:

```
# Full scan over all movies, no index
[movie for movie in movies if movie["year"] == 1995]

# Look up using the movies_by_year index
[movies_by_id[key] for key in movies_by_year[1995]]
```

To start using an index you have to rewrite the algorithm.

Relational data modelling

Dr. Martin Kleppmann
martin.kleppmann@cst.cam.ac.uk

University of Cambridge
Computer Science Tripos, Part IA

Our running example: a movies database

A subset of the IMDb public datasets:

- ≈2,900 movies, ≈23,000 cast and crew members
- Small enough to run on a laptop, big enough to be interesting
- Incomplete (e.g. missing people), real-world messiness

Download it yourself; query and modify it as you like!
(Instructions in the notes)

We will use the same dataset with different data models and query languages to compare those approaches.

Tables in the movies database

A relational database stores everything in tables:

table	one row per...	rows	
`movies`	movie	2,935	entities
`people`	member of cast or crew	23,179	
`genres`	genre name	26	
`ratings`	movie's average rating	2,935	
`has_genre`	movie paired with a genre	7,826	relationships
`has_position`	job someone held on a movie	49,811	
`plays_role`	character an actor played	30,303	

Entities: "things" in the world
Relationships: how one entity relates to another

Both entities and relationships are represented as tables

Primary keys

A **primary key** identifies a row uniquely:
no two rows of a table may hold the same value in it.

movies

movie_id	title	year
tt0078748	Alien	1979
tt0120338	Titanic	1997
tt6751668	Parasite	2019

primary key

genres

genre_id	name
22	Sci-Fi
21	Romance
8	Drama

primary key

`movie_id` is a string; `genre_id` is an integer.
Either is fine, as long as it is unique.

A primary key may consist of several columns
(**composite primary key**) $\Longrightarrow$ values in all those columns
combined must be unique

Foreign keys

A **foreign key** holds the primary key of a row in another table.

movies

movie_id	title	year
tt0078748	Alien	1979
tt0120338	Titanic	1997
tt6751668	Parasite	2019

has_genre

movie_id	genre_id
tt6751668	8
tt0120338	8
tt0120338	21
tt0078748	22

genres

genre_id	name
22	Sci-Fi
21	Romance
8	Drama

"Alien" is Sci-Fi; "Titanic" is Drama and Romance; "Parasite" is Drama

Composite primary keys

In `has_genre`, neither column is unique on its own.

movie_id	genre_id
tt6751668	8
tt0120338	8
tt0120338	21
tt0078748	22

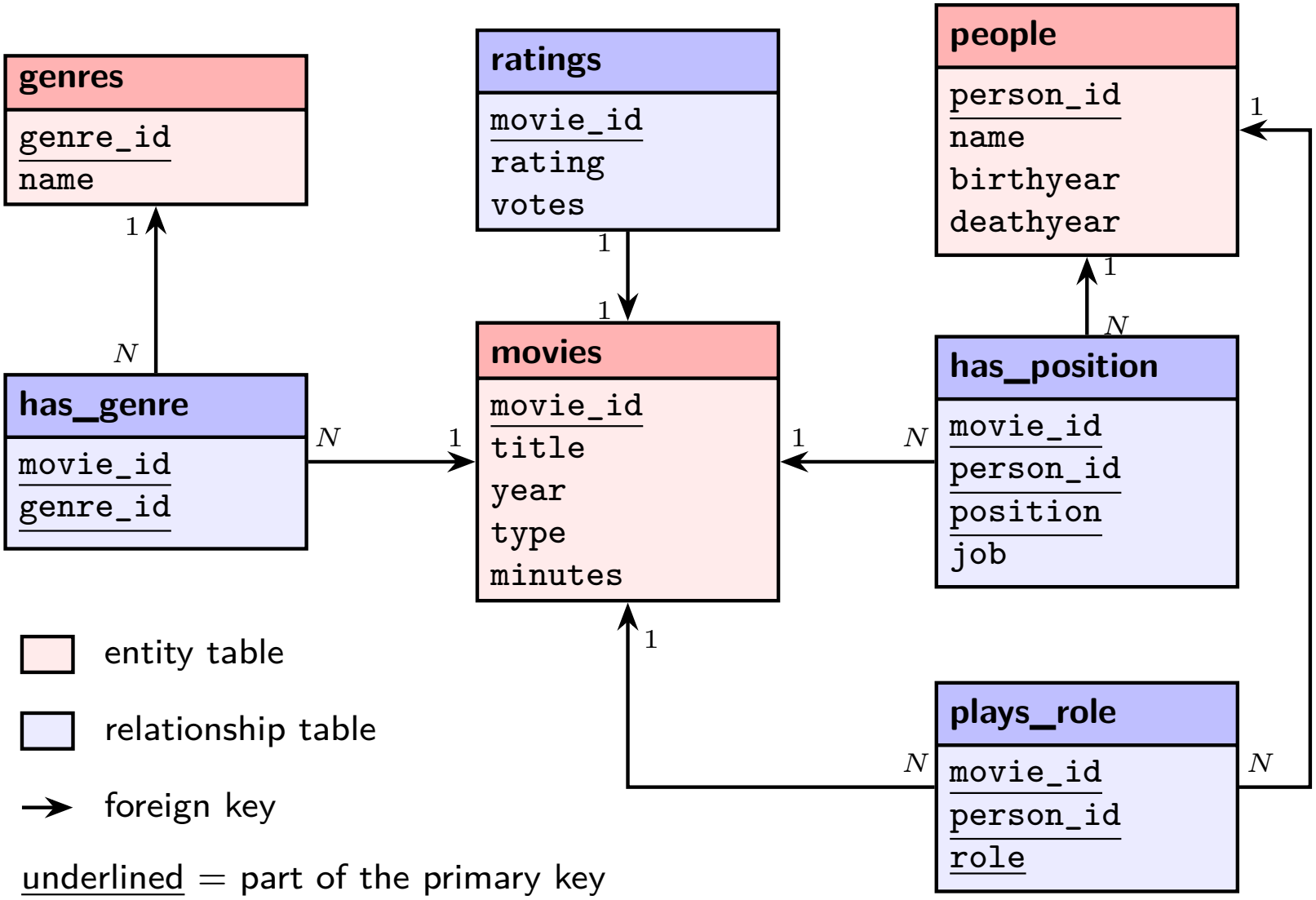
primary key

`movie_id` repeats: one movie can have several genres.
`genre_id` repeats: one genre applies to several movies.

But (`movie_id`, `genre_id`) together are unique.

It's also okay for a table not to have any primary key.
Then it may contain duplicate rows.

The movie database schema



One movie across the tables (1/2)

movies

movie_id	title	year	type	minutes
tt0012349	The Kid	1921	movie	68

ratings

movie_id	rating	votes
tt0012349	8.2	142,991

has_genre

movie_id	genre_id
tt0012349	5
tt0012349	8
tt0012349	11

genres

genre_id	name
5	Comedy
8	Drama
11	Family

One movie across the tables (2/2)

people

person_id	name	birthyear	deathyear
nm0000122	Charles Chaplin	1889	1977

has_position

person_id	movie_id	position	job
nm0000122	tt0012349	actor	*null*
nm0000122	tt0012349	composer	composer
nm0000122	tt0012349	director	director
nm0000122	tt0012349	producer	producer
nm0000122	tt0012349	writer	written by

plays_role

person_id	movie_id	role
nm0000122	tt0012349	A Tramp

Creating the entity tables in SQL

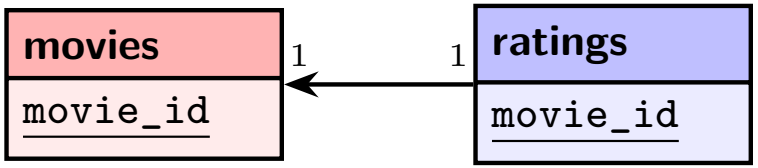
```sql
CREATE TABLE movies(
  movie_id TEXT PRIMARY KEY,
  title    TEXT,
  year     INT,
  type     TEXT,
  minutes  INT
);

CREATE TABLE people(
  person_id TEXT PRIMARY KEY,
  name      TEXT,
  birthyear INT,
  deathyear INT
);

CREATE TABLE genres(
  genre_id INT PRIMARY KEY,
  name     TEXT
);
```

Creating the relationship tables in SQL

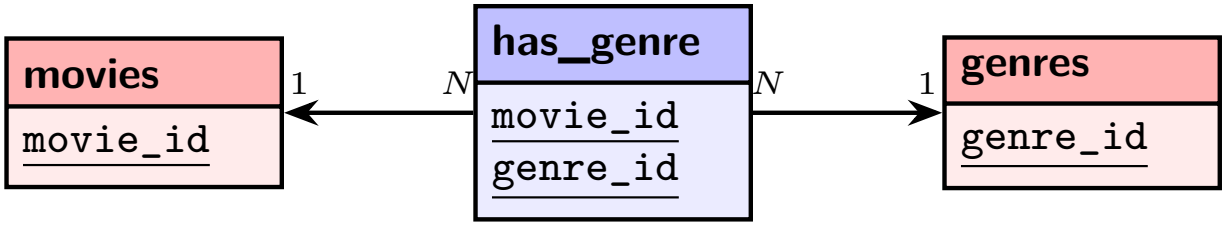
```sql
CREATE TABLE ratings(
  movie_id TEXT PRIMARY KEY REFERENCES movies(movie_id),
  rating   NUMERIC,
  votes    INT
);
CREATE TABLE has_genre(
  movie_id TEXT REFERENCES movies(movie_id),
  genre_id INT  REFERENCES genres(genre_id),
  PRIMARY KEY (movie_id, genre_id)
);
CREATE TABLE has_position(
  person_id TEXT REFERENCES people(person_id),
  movie_id  TEXT REFERENCES movies(movie_id),
  position  TEXT,
  job       TEXT,
  PRIMARY KEY (movie_id, person_id, position)
);
CREATE TABLE plays_role(
  person_id TEXT REFERENCES people(person_id),
  movie_id  TEXT REFERENCES movies(movie_id),
  role      TEXT,
  PRIMARY KEY (movie_id, person_id, role)
);
```

Three shapes of relationship

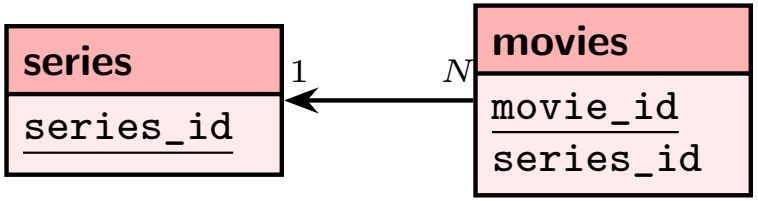
one-to-one: a movie has at most one rating



many-to-many: a movie has many genres, a genre many movies



one-to-many: a series contains many movies



Adding `series` to the example database is an exercise

Reading the shape off the primary key

table	primary key	relationship
`ratings`	`movie_id` (one foreign key)	one-to-one
`has_genre`	`(movie_id, genre_id)` (two foreign keys)	many-to-many
`has_position`	`(movie_id,` `person_id, position)`	many-to-many, with attributes
`plays_role`	`(movie_id,` `person_id, role)`	many-to-many, with attributes

`position` is in the key $\Longrightarrow$ one person can hold **several** positions on the same movie.

To make a **one-to-many** relationship:
A foreign key column that is not part of the primary key, and no join table. See exercise in the notes.

SQL and relational algebra

Dr. Martin Kleppmann
martin.kleppmann@cst.cam.ac.uk

University of Cambridge
Computer Science Tripos, Part IA

Running SQL yourself

You can use DuckDB in two ways:

- **On your own machine**:
  - Install DuckDB from duckdb.org
  - Download the database file from the course web page
  - If you break it, just re-download it
- **In your web browser**
  - Nothing to install, go to course web page
  - Nothing is saved when you reload the tab

You have your own copy: feel free to break it!
Nobody else is affected.

Please check the **DuckDB documentation** for language details

Overview of SQL

- **Data Definition Language** (DDL)
  - `CREATE TABLE`, `ALTER TABLE`, `DROP TABLE`
  - `CREATE INDEX`, ...
  - Describes the **schema** of the database
- **Data Manipulation Language** (DML)
  - `INSERT`, `UPDATE`, `DELETE`
  - `BEGIN TRANSACTION`, `COMMIT`, `ROLLBACK`
  - Changes the data in the database
- **Queries**
  - `SELECT`: Asks questions about the data

We will also cover **relational algebra**

- Mathematical notation approximately equivalent to SQL queries (`SELECT`)

Data definition language: CREATE TABLE

```
CREATE TABLE genres(
  genre_id INT  PRIMARY KEY,    -- unique, identifies a row
  name     TEXT NOT NULL UNIQUE
);

CREATE TABLE has_genre(
  movie_id TEXT REFERENCES movies(movie_id),
  genre_id INT  REFERENCES genres(genre_id),
  PRIMARY KEY (movie_id, genre_id)
);
```

A column has a **name** and a **type**. A table may also have **constraints** that the database will not let you violate:

- `PRIMARY KEY` – identifies a row; at most one row per value
- `REFERENCES` – a foreign key; must name a row that exists
- `NOT NULL`, `UNIQUE`, `CHECK`, ...

Changing and removing tables

```sql
-- Adding a column to an existing table
ALTER TABLE movies ADD COLUMN budget INT;

-- Renaming an existing column
ALTER TABLE ratings RENAME COLUMN votes TO num_votes;

-- Deleting a table entirely
DROP TABLE plays_role;
```

Schema migration: changing the schema of an existing database

Problems: can require rewriting table (slow); queries written against the old schema may stop working

Creating and deleting indexes

```sql
-- Make an index to efficiently find movies by year
CREATE INDEX movies_by_year ON movies (year);

-- Changed your mind, no longer want the index?
DROP INDEX movies_by_year;
```

Declare it once; the database automatically maintains and uses the index thereafter.

An index does not change query **results**, only **performance**.

A `PRIMARY KEY` or `UNIQUE` constraint usually creates an index implicitly. (The index is needed to check uniqueness.)

Inserting, updating, and deleting rows

```sql
-- Insert a new row into a table
INSERT INTO genres (genre_id, name)
VALUES (27, 'Superhero');

-- Modify existing rows in a table
UPDATE movies
SET    minutes = 117
WHERE  movie_id = 'tt0078748';

-- Delete rows from a table
DELETE FROM movies
WHERE year < 1900;
```

⚠️ `UPDATE` and `DELETE` apply to every row matching the `WHERE` clause. If you leave out the `WHERE`, they apply to **every row** in the table!

Fetching all rows from one table

```
SELECT * FROM movies;
```

movie_id	title	year	type	minutes
tt0012349	The Kid	1921	movie	68
tt0015324	Sherlock Jr.	1924	movie	45
tt0015864	The Gold Rush	1925	movie	95
tt0017136	Metropolis	1927	movie	153
⋮	⋮	⋮	⋮	⋮

All 2,935 rows in arbitrary order, all five columns.

Use `LIMIT` if you want fewer rows:

```
SELECT * FROM movies LIMIT 3;
```

movie_id	title	year	type	minutes
tt0012349	The Kid	1921	movie	68
tt0015324	Sherlock Jr.	1924	movie	45
tt0015864	The Gold Rush	1925	movie	95

Note: Since the order of rows is not defined, this might give you **any three** rows, not three specific ones

Ordering rows: ORDER BY

```
SELECT * FROM movies ORDER BY title LIMIT 4;
```

movie_id	title	year	type	minutes
tt10620868	#Alive	2020	movie	98
tt2614684	'71	2014	movie	99
tt1179933	10 Cloverfield Lane	2016	movie	103
tt0147800	10 Things I Hate About You	1999	movie	97

If several rows have the same value in the column you're sorting by, you can specify a secondary column to sort by:

```
SELECT * FROM movies ORDER BY year DESC, title LIMIT 4;
```

movie_id	title	year	type	minutes
tt10548174	28 Years Later	2025	movie	115
tt7181546	Ballerina	2025	movie	124
tt30988739	Black Bag	2025	movie	93
tt32246771	Bring Her Back	2025	movie	104

Ascending by default; DESC reverses one column.

Selecting rows satisfying a condition: WHERE

```
SELECT * FROM movies WHERE type = 'video';
```

movie_id	title	year	type	minutes
tt0328832	The Animatrix	2003	video	100
tt1166827	Zeitgeist	2007	video	118
tt1569923	Batman: Under the Red Hood	2010	video	75
tt2313197	Batman: The Dark Knight Returns, Part 1	2012	video	76
tt2166834	Batman: The Dark Knight Returns, Part 2	2013	video	76
tt2820466	Justice League: The Flashpoint Paradox	2013	video	75

Can use =, <, <=, >, >=, <>, AND, OR, NOT:

```
SELECT * FROM movies
WHERE type = 'video' AND year >= 2012;
```

movie_id	title	year	type	minutes
tt2313197	Batman: The Dark Knight Returns, Part 1	2012	video	76
tt2166834	Batman: The Dark Knight Returns, Part 2	2013	video	76
tt2820466	Justice League: The Flashpoint Paradox	2013	video	75

Relational algebra **selection** operator σ:

$$\sigma_{\text{type}=\text{'video'} \wedge \text{year} \geq 2012}(\text{movies})$$

More operators: IN and LIKE

IN tests membership of a list of values:

```
SELECT * FROM movies
WHERE  year IN (1927, 1931);
```

movie_id	title	year	type	minutes
tt0017136	Metropolis	1927	movie	153
tt0021749	City Lights	1931	movie	87
tt0022100	M	1931	movie	99

LIKE matches a pattern:

% is any sequence of characters, _ any single character.

```
SELECT * FROM movies
WHERE  title LIKE 'The Matrix%';
```

movie_id	title	year	type	minutes
tt0133093	The Matrix	1999	movie	136
tt0234215	The Matrix Reloaded	2003	movie	138
tt0242653	The Matrix Revolutions	2003	movie	129
tt10838180	The Matrix Resurrections	2021	movie	148

Projecting columns: SELECT

```
SELECT title, year FROM movies
WHERE type = 'video' AND year >= 2012;
```

title	year
Batman: The Dark Knight Returns, Part 1	2012
Batman: The Dark Knight Returns, Part 2	2013
Justice League: The Flashpoint Paradox	2013

Relational algebra **projection** operator π:

$$\pi_{\text{title, year}}(\sigma_{\text{type='video'} \wedge \text{year} \geq 2012}(\text{movies}))$$

⚠️ SQL's SELECT is projection; the WHERE clause is selection!

Renaming columns and tables: AS

```
SELECT title, year AS release_year FROM movies
WHERE type = 'video' AND year >= 2012;
```

title	release_year
Batman: The Dark Knight Returns, Part 1	2012
Batman: The Dark Knight Returns, Part 2	2013
Justice League: The Flashpoint Paradox	2013

Relational algebra **renaming** operator ρ:

$$\rho_{\text{year}\rightarrow\text{release_year}}(\pi_{\text{title, year}}(\sigma_{\text{type}=\text{'video'} \wedge \text{year}\geq 2012}(\text{movies})))$$

Renaming tables:

```
SELECT * FROM movies AS m
```

Needed when a query mentions the same table twice.

Grouping rows: GROUP BY

How many movies came out in each year since 2021?

```
SELECT year, COUNT(*) AS n_movies
FROM   movies
WHERE  year >= 2021
GROUP BY year
ORDER BY year;
```

year	n_movies
2021	89
2022	98
2023	96
2024	73
2025	27

Relational algebra **grouping** operator γ:

$$\gamma_{\text{year; count}(*)\rightarrow\text{n_movies}}(\sigma_{\text{year}\geq 2021}(\text{movies}))$$

One output row per distinct `year`, and `COUNT(*)` counts the rows that fell into that group.

Grouping by several columns

You can name several columns in the `GROUP BY` clause:

```
SELECT year, type, COUNT(*) AS n_movies
FROM   movies
WHERE  year >= 2023
GROUP BY year, type
ORDER BY year, type;
```

year	type	n_movies
2023	movie	92
2023	tvMovie	4
2024	movie	73
2025	movie	27

Relational algebra:

$$\gamma_{\text{year, type; count}(*)\rightarrow\text{n_movies}}(\sigma_{\text{year}\geq 2023}(\text{movies}))$$

One row per `(year, type)` combination that actually occurs: no row for combinations that don't occur.

Aggregate functions

After grouping by the `GROUP BY` columns, values within each group can be combined with **aggregate functions** (`COUNT`, `MIN`, `MAX`, `SUM`, `AVG`, ...)

Leave out `GROUP BY`: aggregate whole table into one row

```
SELECT COUNT(*) AS n_movies, MIN(rating) AS worst,
       MAX(rating) AS best, AVG(rating) AS mean
FROM   ratings;
```

n_movies	worst	best	mean
2935	2.1	9.3	7.17

Variations of `COUNT`:

- `COUNT(*)` counts rows
- `COUNT(c)` counts rows where column c is not null
- `COUNT(DISTINCT c)` counts how many different non-null values the group contains in column c

Filtering by the result of aggregates: HAVING

Which years had more than 140 movies?

```
SELECT year, COUNT(*) AS n_movies
FROM   movies
GROUP BY year
HAVING n_movies > 140
ORDER BY year;
```

year	n_movies
2013	153
2014	146
2016	147

"`WHERE n_movies > 140`" doesn't work: selection happens before grouping. `HAVING` filters **after** grouping.

Execution order: `FROM` → `WHERE` → `GROUP BY` → `HAVING` → `SELECT` → `ORDER BY` → `LIMIT`

Cartesian product between two tables

Cartesian product $\times$: every combination of a row from one relation with a row from the other

$$\text{movies} \times \text{genres}$$

contains $2{,}935 \cdot 26 = 76{,}310$ rows

In SQL: `SELECT * FROM movies, genres`

movie_id	title	year	type	minutes	genre_id	name
tt0012349	The Kid	1921	movie	68	1	Action
tt0012349	The Kid	1921	movie	68	2	Adventure
tt0012349	The Kid	1921	movie	68	3	Animation
tt0012349	The Kid	1921	movie	68	4	Biography
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Rarely useful by itself – but good to help understand joins.

Inner join between two tables: JOIN

Get the three top-rated movies, returning their title and rating:

```
SELECT title, rating
FROM   movies
JOIN   ratings ON movies.movie_id = ratings.movie_id
ORDER BY rating DESC LIMIT 3;
```

title	rating
The Shawshank Redemption	9.3
The Godfather	9.2
The Dark Knight	9.1

Relational algebra **join** operator $\bowtie$:

$$\begin{aligned} &\text{movies} \bowtie_{\text{movie_id}} \text{ratings} \\ &= \sigma_{\text{movies.movie_id} = \text{ratings.movie_id}}(\text{movies} \times \text{ratings}) \end{aligned}$$

(No relational algebra operators for `ORDER BY` and `LIMIT`)

How a DBMS might execute a join

Nested loop join: for each movie, scan all of `ratings`

```
results = []
for movie in movies:                    # for each movie...
    for rating in ratings:              # ...scan all of ratings
        if rating["movie_id"] == movie["movie_id"]:
            result = (movie["title"], rating["rating"])
            results.append(result)
```

8,614,225 comparisons for 2,935 result rows

Index nested loop join: look up each movie's rating directly

```
results = []
for movie in movies:                    # scan the movies
    rating = ratings_by_movie_id.get(movie["movie_id"])
    if rating is not None:              # no rating, no row
        results.append((movie["title"], rating))
```

One lookup per movie, given an index on `ratings.movie_id`

Joining more than two tables

```
SELECT   m.title, r.role, p.name
FROM     movies AS m
JOIN     plays_role AS r ON m.movie_id = r.movie_id
JOIN     people AS p ON r.person_id = p.person_id
WHERE    m.title = 'The Kid'
ORDER BY r.role
LIMIT    4;
```

title	role	name
The Kid	A Tramp	Charles Chaplin
The Kid	Bride	Beulah Bains
The Kid	Car Thief	Albert Austin
The Kid	His Assistant	B.F. Blinn

$$\text{movies} \bowtie_{\text{movie_id}} \text{plays_role} \bowtie_{\text{person_id}} \text{people}$$

Joins are associative: the DBMS chooses the execution order.

USING: a shorthand for join conditions

An explicit join condition:

```
SELECT title, rating
FROM   movies
JOIN   ratings ON movies.movie_id = ratings.movie_id;
```

When the columns have the same name on both sides, you can also write:

```
SELECT title, rating
FROM   movies
JOIN   ratings USING (movie_id);
```

USING also merges the two movie_id columns into one, where ON leaves both in the result.

Left outer join: LEFT JOIN

What if a movie has no actors, but we still want to return it?

```
SELECT    m.title, r.role, p.name
FROM      movies AS m
LEFT JOIN plays_role AS r ON m.movie_id = r.movie_id
LEFT JOIN people AS p ON r.person_id = p.person_id
WHERE     m.title IN ('Arctic', 'Flow')
ORDER BY  m.title, p.name;
```

title	role	name
Arctic	Overgård	Mads Mikkelsen
Arctic	Young Woman	Maria Thelma Smáradóttir
Arctic	Helicopter Pilot	Tintrinai Thikhasuk
Flow	*null*	*null*

$$\text{movies} \mathbin{⟕}_{\text{movie_id}} \text{plays_role} \mathbin{⟕}_{\text{person_id}} \text{people}$$

Return all rows from **left** side of join (i.e. `movies`), filling in `NULL` if there is no match on right side.

How a DBMS might execute a left join

Left join: every row of the left table reaches the output

```
results = []
for movie in movies:                        # every movie appears...
    matches = roles_by_movie_id.get(movie["movie_id"], [])
    for row in matches:
        results.append((movie["title"], row["role"]))
    if not matches:                         # ...even with no match
        results.append((movie["title"], None))
```

A movie with several matches produces several result rows; one with none still gives a row, padded with `None`.

An inner join is the same loop without the last two lines.

Outer joins

`JOIN`	$\bowtie$	matching rows only
`LEFT JOIN`	$⟕$	all left rows, null on unmatched right
`RIGHT JOIN`	$⟖$	all right rows, null on unmatched left
`FULL JOIN`	$⟗$	all rows of both, null on either side

Example of full outer join:

R

a	**b**
1	x
2	y

$⟗_{\mathrm{b}}$

S

b	**c**
x	p
z	q

$=$

$\mathrm{R} ⟗_{\mathrm{b}} \mathrm{S}$

a	**b**	**c**
1	x	p
2	y	*null*
null	z	q

More complex join conditions

Get the composer for each movie, or *null* if there is none:

```
SELECT    m.title, p.name AS composer_name
FROM      movies AS m
LEFT JOIN has_position AS hp ON m.movie_id = hp.movie_id
                              AND hp.position = 'composer'
LEFT JOIN people AS p USING (person_id)
WHERE     m.year = 1977
ORDER BY  m.title;
```

title	composer_name
Annie Hall	*null*
Close Encounters of the Third Kind	John Williams
Star Wars: Episode IV - A New Hope	John Williams

Moving `AND hp.position = 'composer'` into the `WHERE` clause would drop *Annie Hall* from the result.

Deduplicating rows: SELECT DISTINCT

Relational algebra is usually defined on **sets**, but in SQL rows can be duplicated (**multiset**):

```
SELECT type FROM movies;
```

type
movie
movie
movie
⋮

2,935 rows, one per movie

To keep one copy of each distinct value:

```
SELECT DISTINCT type FROM movies;
```

type
movie
tvMovie
video

Set operations

Q_1 UNION Q_2	$\cup$	returns rows in either
Q_1 INTERSECT Q_2	$\cap$	returns rows in both
Q_1 EXCEPT Q_2	$-$	returns rows in the first but not the second

These operators implicitly remove duplicate rows.
Both sides must have the same columns.

```
-- Return the IDs of all movies that are both in the
-- Crime (genre_id = 6) and Drama (genre_id = 8) genres
SELECT movie_id FROM has_genre WHERE genre_id = 6
INTERSECT
SELECT movie_id FROM has_genre WHERE genre_id = 8;
```

$$\pi_{\text{movie_id}}(\sigma_{\text{genre_id}=6}(\text{has_genre})) \cap \pi_{\text{movie_id}}(\sigma_{\text{genre_id}=8}(\text{has_genre}))$$

Preserving duplicates: `UNION` vs. `UNION ALL`

IDs of movies that are Crime **or** Drama:

```
SELECT movie_id FROM has_genre WHERE genre_id = 6
UNION
SELECT movie_id FROM has_genre WHERE genre_id = 8;
```

1,915 rows

`UNION ALL` = relational algebra **multiset union** operator $\uplus$:

```
SELECT movie_id FROM has_genre WHERE genre_id = 6
UNION ALL
SELECT movie_id FROM has_genre WHERE genre_id = 8;
```

2,312 rows

621 Crime movies + 1,691 Drama movies = 2,312 total;
the 397 movies that belong to both genres are counted twice.

Nulls in SQL

`NULL` can mean various things:

- Value unknown
- Value not applicable (e.g. death year when person is alive)
- Outer join did not match a row

`NOT NULL` constraint on a column disallows nulls.
Primary keys are automatically `NOT NULL`.

It is not a value, so **comparing with it returns `NULL`**:

- `NULL = NULL` returns `NULL`
- `NULL <> NULL` returns `NULL`
- `x = NULL` is `NULL` whatever `x` is
- Special operators that return true or false:
  `x IS NULL`, `x IS NOT NULL`

Three-valued logic

a	b	a AND b	a OR b	NOT a
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE
TRUE	NULL	NULL	TRUE	FALSE
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	FALSE	FALSE	FALSE	TRUE
FALSE	NULL	FALSE	NULL	TRUE
NULL	TRUE	NULL	TRUE	NULL
NULL	FALSE	FALSE	NULL	NULL
NULL	NULL	NULL	NULL	NULL

WHERE keeps a row only when the condition is TRUE; rows evaluating to NULL or FALSE are filtered out.

Composing queries

Relational algebra composes

Every operator takes relations and returns a relation, so expressions can be nested:

$$\sigma_{\ldots}(\gamma_{\ldots}(\sigma_{\ldots}(\text{movies})))$$

In SQL, a `SELECT` is a fixed sequence of clauses:

```
SELECT ... FROM ... WHERE ... GROUP BY ... HAVING ...
```

each appearing at most once, in that order.

Selecting rows *before* grouping is `WHERE`; selecting *after* grouping needs a separate keyword, `HAVING`.

`UNION`, `INTERSECT`, `EXCEPT` combine the results of multiple queries, but don't let you use one query's result as another query's input.

Subquery in FROM: a derived table

Using query A's result as query B's input:

- Put A in parentheses in B's FROM clause
- Use AS to give a name to this **derived table**

```
-- Which years have more than 140 movie releases?
SELECT year, n_movies
FROM  (SELECT year, COUNT(*) AS n_movies
       FROM movies GROUP BY year) AS per_year
WHERE n_movies > 140;
```

year	n_movies
2013	153
2014	146
2016	147

Same result as the HAVING query, executed in the same way (relational algebra in the DBMS's query plan is the same)

Common table expressions: WITH

Like **defining a variable**: run query A, give its result a name, then refer to that name in query B as if it were a table.

```
-- Define per_year as a name you can refer to later
WITH per_year AS (
    SELECT year, COUNT(*) AS n_movies
    FROM movies GROUP BY year
)
SELECT year, n_movies FROM per_year  -- Now use it
WHERE n_movies > 140;
```

year	n_movies
2013	153
2014	146
2016	147

The same query again, read top to bottom. A name can be used more than once, and you can define several names.

Scalar subquery in WHERE

Which 1976 movies are rated above the overall average?

```
SELECT title, rating
FROM   movies JOIN ratings USING (movie_id)
WHERE  year = 1976
  AND  rating > (SELECT AVG(rating) FROM ratings);
```

title	rating
Carrie	7.4
Network	8.1
Rocky	8.1
Taxi Driver	8.2

`WHERE rating > AVG(rating)` is an error: aggregates not allowed in the `WHERE` clause.

The subquery computes the average separately, as a value.

Subquery in the NOT IN operator

Which movies have no actors?

```
SELECT movie_id, title FROM movies
WHERE movie_id NOT IN (SELECT movie_id FROM plays_role);
```

movie_id	title
tt4772188	Flow
tt27719898	Kaiva

⚠ NOT IN is **surprising** if the subquery may return NULL:

- c NOT IN $(v_1, v_2, \ldots, v_n)$ expands to $(c$ <> $v_1)$ AND $(c$ <> $v_2)$ AND ... AND $(c$ <> $v_n)$
- If v_i is NULL, $(c$ <> $v_i)$ evaluates to NULL
- TRUE AND NULL evaluates to NULL
- So c NOT IN $(v_1, v_2, \ldots, v_n)$ evaluates to NULL
- So the whole query returns no rows

plays_role.movie_id cannot be null (part of primary key).

Antijoin: NOT EXISTS

Which movies have no actors?

```
SELECT movie_id, title
FROM   movies AS m
WHERE NOT EXISTS (
    SELECT 1 FROM plays_role AS r
    WHERE r.movie_id = m.movie_id
);
```

movie_id	title
tt4772188	Flow
tt27719898	Kaiva

Correlated subquery: refers to `m` from the outer one.

`EXISTS` is true if subquery returns ≥ 1 rows;
`NOT EXISTS` is true if subquery returns 0 rows.

Row content doesn't matter; `SELECT 1` just returns the constant value 1 in a single unnamed column

Relational algebra **antijoin** operator $\triangleright$:

$$\text{movies} \triangleright_{\text{movie_id}} \text{plays_role}$$

Semijoin: EXISTS

Who is credited as a composer on at least one movie?

```
SELECT name FROM people AS p
WHERE EXISTS (
    SELECT 1 FROM has_position AS hp
    WHERE hp.person_id = p.person_id
      AND hp.position = 'composer'
);
```

name
A.C. Newman
A.R. Rahman
Aaron May
Aaron Zigman
⋮

1,086 rows, one per person.
A `JOIN` would return 3,281 rows, one per credit.

Relational algebra **semijoin** operator $\ltimes$:

$$\text{people} \ltimes_{\text{person_id}} (\sigma_{\text{position}=\text{'composer'}}(\text{has_position}))$$

Semijoin and antijoin as pseudocode

Semijoin: stop at the first match

```
results = []
for person in people:
    for credit in composer_credits:
        if credit["person_id"] == person["person_id"]:
            results.append(person)
            break          # at most one row per person
```

Antijoin: output only if there was no match

```
results = []
for movie in movies:
    matched = False
    for role in plays_role:
        if role["movie_id"] == movie["movie_id"]:
            matched = True
            break
    if not matched:        # no match, so output it
        results.append(movie)
```

Relational algebra summary

SQL	Relational algebra	
`WHERE ...`	σ	selection: rows
`SELECT a, b`	π	projection: columns
`AS`	ρ	renaming
`GROUP BY`	γ	grouping & aggregation
`FROM a, b`	$\times$	cartesian product
`JOIN ... ON`	$\bowtie$	inner join
`LEFT JOIN`	$⟕$	left outer join
`WHERE EXISTS ...`	$\ltimes$	semijoin
`WHERE NOT EXISTS ...`	$\triangleright$	antijoin
`UNION`	$\cup$	union
`INTERSECT`	$\cap$	intersection
`EXCEPT`	$-$	difference

Relational algebra works on **sets**, SQL on **multisets**.