

Databases
University of Cambridge
Computer Science Tripos, Part IA
Michaelmas term 2026/27

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

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

This set of notes is incomplete. Updates will be released on the course web page as the course progresses.

Contents

1	Introduction	2
1.1	What is a database?	4
1.2	Types of databases	8
2	Relational data modelling	10
3	SQL and relational algebra	17
3.1	Defining and manipulating data	18
3.2	Querying a single table	20
3.3	Grouping and aggregation	23
3.4	Joins	25
3.5	Comparing relational algebra and SQL	29
3.6	Query composition	32

Thank you to James Sharkey for preparing the practical materials.



This work is published under a [Creative Commons BY-SA](https://creativecommons.org/licenses/by-sa/4.0/) license.

1 Introduction

This course is about *databases*: systems whose job is to store data, keep it correct, and answer questions about it. Almost every application you use is built on one, though you rarely see it directly. It has been estimated that there are over one trillion databases in the world [[SQLite contributors](#)]. A surprisingly large amount of software development nowadays consists of deciding how to organise your data, putting it in a database, and getting it out again. A bank is, in some sense, just a very carefully guarded database of accounts and the transactions between them.

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.

Slide 1

In this course we will look at how data can be modelled, at the languages used to query it, and at the trade-offs between the different kinds of database that exist. By the end of it, if you need to build an application, you will hopefully be in a position to choose a sensible tool and use it well.

This course focuses on how to *use* databases, rather than how they work internally. The internals are interesting, and we will cover some of this in Part IB and Part II, but for now we will take the perspective of an author of an application that uses a database. You will learn a couple of languages that are used to query databases, but more importantly, you'll learn how to *think* about data, how it is modelled, and how it is accessed.

This course

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

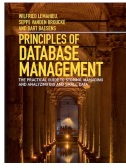
Query languages: SQL, relational algebra, Datalog, SQL/PGQ, JSONPath

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

Slide 2

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!

Slide 3

These notes and lectures should be self-contained, but if you would like to read up on further detail, the suggested textbooks are:

- Wilfried Lemahieu, Seppe vanden Broucke, and Bart Baesens. *Principles of Database Management*. Cambridge University Press, 2018. ISBN 978-1-107-18612-5

Free online access from the university network at <https://www.cambridge.org/highereducation/books/principles-of-database-management/2FAC1A38D7BF11C3BB1D330925571BE4>

- Martin Kleppmann. *Designing Data-Intensive Applications*, second edition. O'Reilly, 2026. ISBN 978-1-098-11906-5

For free online access, go to <https://www.oreilly.com/library/view/designing-data-intensive-applications/9781098119058/> and log in with your @cam.ac.uk email address.

Where appropriate, these lecture notes also contain references to research papers and other useful background reading (these are given in square brackets, and the details appear at the end of this document). However, only material covered in the lectures, practicals, and/or these notes is examinable.

As for other courses, past exam questions are available at <https://www.cl.cam.ac.uk/teaching/exams/pastpapers/t-Databases.html>. The syllabus, slides, and lecture notes for this course were substantially updated and revised in 2026/27. Because of syllabus changes, some past exam questions may no longer be applicable.

These notes also contain exercises, which are suggested material for discussion in supervisions. Solution notes for supervisors are available from the course web page, <https://www.cst.cam.ac.uk/teaching/2627/Databases/>.

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

Slide 4

In previous years, this course included formally assessed practicals (ticks). That is changing in 2026/27: since AI can now solve any question for you in an instant, we felt that this form of assessment was no longer meaningful. Instead, this course will be assessed only by the end-of-year exam, like most courses.

We still have practical exercises though, and they are more important than ever! But rather than formally assessing them, we ask you to complete the exercises in order to support your own learning, and to discuss your solutions with your supervisor. If you don't complete the practicals, you're missing out on the best possible preparation for the exam, not to mention for your future careers. You can use AI if you want, but you'd only be cheating yourself out of the education for which you're here. In the end, that education is your responsibility.

1.1 What is a database?

So, what is a database actually? You can write code that stores data and queries it – is that a database?

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?

Slide 5

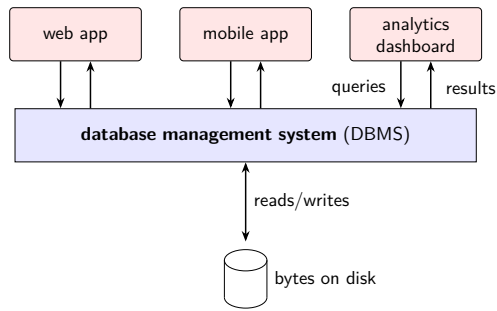
To answer the questions on [Slide 5](#):

- If you want to be able to make changes to the data and have them persist, even when you restart the program, you have to write them to a file on disk.
- If you want to allow several users to simultaneously read and modify the data, you will need some sort of concurrency control, such as locking (see Part IB Concurrent and Distributed Systems).
- If you want to handle a dataset that's too big to fit in memory, you will need some way of loading only a part of it at a time.
- If you want to allow users to query the data in arbitrary ways, you will need to either let those users edit your program, or have some way of running user code over your data.
- If you want to ensure the data doesn't get corrupted in case of a crash, you need careful handling of your writes to disk.

These are all things you could program yourself, but a database provides them for you in a convenient package. We say that the database is a *higher-level abstraction* that lets you store and access your data more conveniently, while handling all of the above issues for you.

Strictly speaking, a database is an organised collection of data; the program that manages it is a *database management system*, or DBMS. In practice the word “database” is used loosely for both, and we will do the same in this course. The arrangement is shown on [Slide 6](#): applications do not access the bytes on disk directly, but send requests to the DBMS, which manages access to the storage and provides a higher-level abstraction on top of it.

The database as abstraction layer



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

Slide 6

If your storage needs are simple – for example, the data is small and read-only (like a configuration file) or you only need to append to the end (like a log file) – using a file is often sufficient, and a database is unnecessary. But as the data or usage patterns become more complex, databases are indispensable.

The root of the difference is that a file is a sequence of bytes and nothing more. You can choose a file format, which defines how the file is broken down into values, records, and other structures, but the file system knows nothing about that structure. In contrast, with a database you generally specify the structure in advance (this is called a *schema*), so the database can ensure that the data conforms to this structure.

Everything else follows from that. Looking up one record in a file means reading the file from the beginning until you come across it; a database can keep an *index* that allows it to more quickly find the record you're looking for. Changing a record in the middle of a file means rewriting everything that follows, since the new value rarely occupies exactly the same number of bytes as the old value; databases have more efficient ways of performing small updates to a large dataset. If the machine loses power halfway through rewriting a file you are left with a file that is neither the old version nor the new one, whereas a database applies a group of changes entirely or not at all, using a concept called a *transaction*. And two programs writing to one file at the same time will interleave their writes and corrupt the result, unless you build a locking scheme yourself.

None of this is impossible to do with files – it is, roughly, what a database does internally – but it is a great deal of work to get right, and it is work that has already been done.

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 ⇒ corrupted file	A transaction applies entirely, or not at all
Two writers at once ⇒ corrupted file	Concurrency control stops them interfering
A new question: edit the program	Write a new query

Slide 7

Between files and databases sits a widely-used family of systems called *key-value stores*. A key-value store offers a very simple interface: you can store a value under a key, and retrieve the value for a key. In return, key-value stores are simple, fast, and comparatively easy to spread across many machines, which is why they are popular for large amounts of data with simple access patterns.

A directory on a filesystem is similar to a key-value store: the filename is the key, the file contents are the value, and the directory gives you a way of listing keys. The difference is that filesystems are typically designed for larger values (each file is typically assumed to be at least a few kilobytes, if not megabytes or gigabytes in size), whereas in a key-value store, values are generally smaller (less than a megabyte, say, and often just a few bytes). Also, filesystems often store only a few thousand files per directory and might get slow with more than that, while key-value stores with millions of key-value pairs are common.

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)

Slide 8

Let's return to the example from [Slide 5](#). To find all the movies that came out in 1995, the list comprehension in Python has to scan over all of the movies in the list. In a database of n movies this has a cost of $O(n)$. If n is a few thousand this doesn't matter, but it does matter when n is in the millions.

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.

Slide 9

An *index* is an additional structure, maintained by the database alongside the main data store, that allows you to efficiently find the occurrences of some value – like the index at the end of a book, which allows you to find the occurrences of some concept without having to read the entire book.

[Slide 10](#) shows what an index could look like if you implemented it using a Python dictionary (which uses a hash table internally). In fact, most real databases use a sorted structure such as a *B-tree*, which allows lookup with $O(\log n)$ cost, and also supports efficient *range queries*, such as finding all the movies released between 1960 and 1980 [[Jones, 2019](#)]. See the Algorithms course later this year for more on hash tables and B-trees.

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 $O(\log n)$

Slide 10

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

Slide 11

A database management system is what you get when you combine all of the things we discussed: a schema to describe the structure of the data, a query language for asking questions about the data, transactions to avoid corruption due to concurrency or crashes, and indexes for efficient lookups. The price is that a DBMS is a larger, more complex system. For use cases that are well served by files or key-value stores, you are better off using those directly.

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

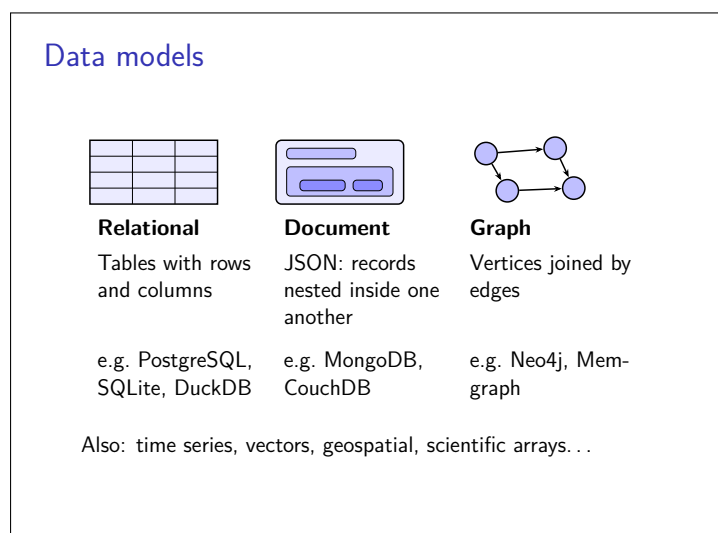
Slide 12

1.2 Types of databases

There are many different types of databases designed for different purposes. To choose the one that is most appropriate to the problem you are trying to solve, you need to know the ways they are commonly categorised.

The first way in which databases are distinguished is by their *data model*: the expected shape of your data. A *relational* database holds tables of rows and columns, with rows in one table referring to rows in another by key. The relational model was introduced by Codd [1970] and has been the most popular model for the last half century, so we focus most on it in this course. But there are also alternatives: a *document* database holds self-contained documents, usually encoded as JSON, in which related information is nested inside the document rather than being split across tables. A *graph* database holds vertices joined by edges, and is optimised for following those edges from one vertex to the next. (Graphs are introduced in more detail in the Algorithms course later this year.) The same information can generally be represented in any of these data models, but often one of them is more convenient than the others.

Beyond these three there are databases specialised for particular kinds of data: time series from sensors, vectors for AI applications, geospatial data such as geographic maps, and a variety of data formats used for scientific applications such as climate models, telescope images, and many others. They are outside the scope of this course, but it's worth knowing that they exist.



Slide 13

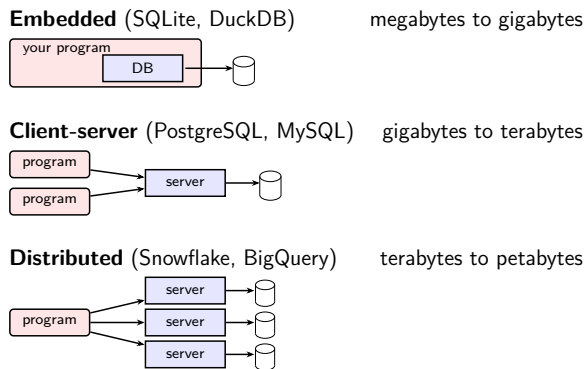
Another way in which we distinguish databases is in terms of where the database runs.

An *embedded* database is a library that runs inside your own program, in the same process, reading and writing a file on the same machine. This approach works well for apps that only need to store data on behalf of the local user. SQLite is a popular example (used in many mobile apps), and DuckDB (which we use in this course) is another.

A *client-server* database runs as a separate process, usually on a separate machine, and one or more programs connect to it over a network. This is convenient for services that are accessed from many different devices, such as a website. PostgreSQL and MySQL are common choices of client-server databases, but many others also exist.

A *distributed* database spreads the data over many machines, so that both the storage and the work of querying it are shared. This approach is needed when the amount of data is too big to hold on a single machine. Systems like Snowflake and BigQuery are used by large corporations to store large volumes of data for analytical purposes. For example, a supermarket chain might store every single purchase made by customers (with date, time, shop location, price, etc.), allowing them to analyse the effect of running a particular promotion. Such datasets can grow to be petabytes (10^{15} bytes) in size. A database designed for such business analytics is also known as a *data warehouse*.

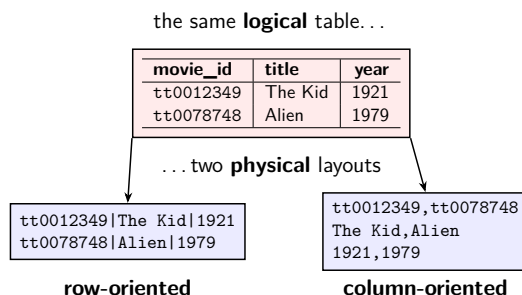
Where the database runs



Slide 14

Yet another way of distinguishing databases is in terms of their internal data layout. A database's data model presents a *logical* representation of the data, and this is the level at which you write queries. On the other hand, the *physical* representation is how the bytes are actually arranged on disk and in memory, and it has a big impact on the performance of the database. The separation between these two representations allows the physical representation to be changed without affecting the logical representation: the same data can be stored in different ways, and queries continue giving the same answers. This property is called *data independence*.

Row-oriented and column-oriented



Often update individual records $\Rightarrow$ row-oriented is better
Often summarise many records $\Rightarrow$ column-oriented is better

Slide 15

For databases with a relational model, there are two common physical representations. A *row-oriented* database stores each row as one contiguous run of bytes. A *column-oriented* database stores the values from each column together instead, so that reading one column of a million rows touches only the bytes of that column, and values of the same type sit next to each other and compress well [Abadi et al., 2013].

Which is better depends on the workload. As a rule of thumb, if you most often read or write one or a few rows at a time (known as *OnLine Transaction Processing* or OLTP), a row-oriented layout tends to be better. If you often read large numbers of rows at a time, and also load data in large batches (as is typical in *analytics*), a column-oriented database often performs better. For example, the database behind a website (where you can make a post, a purchase, etc.) is typically optimised for OLTP, whereas a data warehouse (as in the supermarket example) is designed for analytics.

Each data model comes with a query language, and this course covers several of them. *SQL* is the language of relational databases, and by far the most widely used; most of this course is about it. SQL is one of the world's most popular programming languages [Caballar, 2022]. But we will also look at several alternatives that are better in some circumstances.

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

Slide 16

All of the query languages we study in this course are *declarative* [Krishnaswami, 2013]. That is, the query describes only what is wanted, but it doesn't say what algorithm should be used to compute the query result. Thus, the DBMS is free to pick whatever algorithm it believes will be fastest; for example, it can automatically choose which indexes to use to answer the query. The *query optimiser* is the component within the DBMS that makes these choices; it produces a *query plan* for a query, which then tells the DBMS engine how to execute the query.

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.

Slide 17

Exercise 1. What are some advantages and disadvantages of a declarative query language compared to a general-purpose programming language such as Python?

2 Relational data modelling

Throughout this course we will use one running example: a database of movies. You can run every query in these notes for yourself, and experiment with different queries in order to get a feel for using different data models and query languages. Slide 30 explains how to get it running.

The data is a subset of the IMDb non-commercial datasets [IMDb, 2025]. The full dataset is too large to be convenient, so we have filtered it down to roughly 2,900 popular movies: enough that queries return interesting results, but small enough that the whole database runs comfortably on a laptop. For now we will look at a relational representation of the dataset, and we will consider other data models later.

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.

Slide 18

The database consists of seven tables, listed on [Slide 19](#). The tables fall into two groups. Three of them – **movies**, **people**, and **genres** – each describe a kind of *thing* in the world: a movie, a person, a genre. We will call these *entity tables*. The others describe *relationships* between those things: which movie has which genre, which person worked on which movie, and so on. This distinction is not part of SQL – to the database, all seven are simply tables – but it is a useful way to read a schema, and we will return to it when we look at how to design one.

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	} relationships
has_genre	movie paired with a genre	7,826	
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

Slide 19

Every table has a *primary key*: a column, or a combination of columns, whose value identifies a row uniquely. It is common for primary keys to simply be integers, but in our example database, the keys of **movies** and **people** are the string identifiers that IMDb itself uses: movies have keys like **tt0012349** and people have keys like **nm0000122**. A key like this, which already exists in the data and has a meaning outside the database, is called a *natural key*. The **genres** table is different: IMDb records genres only as names, but our database assigns an integer **genre_id** to each of the 26 genres. A key invented purely to identify rows is called a *surrogate key*.

[Slide 20](#) shows a few rows of two of the tables, with the primary key of each marked.

Primary keys

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

movies			genres	
movie_id	title	year	genre_id	name
tt0078748	Alien	1979	22	Sci-Fi
tt0120338	Titanic	1997	21	Romance
tt6751668	Parasite	2019	8	Drama

primary key 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**) $\implies$ values in all those columns
combined must be unique

Slide 20

Where one table needs to refer to a row of another, it stores that row's primary key in a column of its own, and declares it as a *foreign key*. For example, each row of **has_genre** holds a **movie_id** and a **genre_id**, which refer to rows of **movies** and **genres** respectively.

Slide 21 shows this happening: every value in the **has_genre** excerpt is the primary key of a row in one of the other two tables.

Foreign keys

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

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

Slide 21

A primary key does not have to be a single column. The primary key of **has_genre** is the pair (**movie_id**, **genre_id**), and it has to be: as Slide 22 shows, neither column on its own takes a different value in every row. A movie appears once for each of its genres, and a genre once for each movie that has it, so only the two columns *together* identify a row. A key made of several columns like this is called a *composite primary key*.

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.

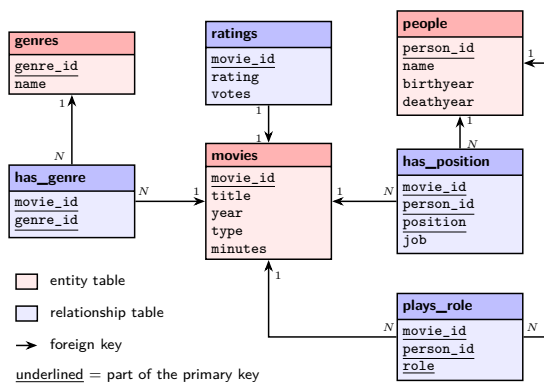
But (`movie_id`, `genre_id`) together are unique.

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

Slide 22

Slide 23 shows the whole schema drawn as a diagram, with an arrow from each foreign key to the primary key it references. A diagram like this is the quickest way to get your bearings in an unfamiliar database. The colour-coding is not standard, but it's a common convention to underline the columns that form the primary key.

The movie database schema



Slide 23

The `movies` table has one row per movie:

- `movie_id`: a unique identifier (chosen by IMDb), which serves as the primary key.
- `title`: the movie's title; it is not unique, since different movies can have the same title.
- `year`: the year of release.
- `type`: one of `movie`, `tvMovie`, or `video`, distinguishing a cinema release from one made for television or released direct to video.
- `minutes`: the running time in minutes.

The `people` table has one row per member of a cast or crew:

- `person_id`: a unique identifier (chosen by IMDb), and the primary key.
- `name`: the person's name.
- `birthyear`: the year of birth (NULL where IMDb records none, which is the case for 38% of people).
- `deathyear`: the year of death, NULL for 88% of people. A NULL here means either that the person is still alive or that no date has been recorded, and the database does not distinguish between the two.

The **genres** table has one row per genre name:

- **genre_id**: a unique integer identifier, which serves as the primary key.
- **name**: the name of the genre, such as **Comedy** or **Documentary**.

The **ratings** table has one row per movie:

- **movie_id**: the ID of the movie being rated; both a foreign key to **movies** and the primary key of this table.
- **rating**: the average of the scores IMDb users have given the movie, out of ten.
- **votes**: the number of scores that average was calculated from.

The **has_genre** table has one row for each pairing of a movie with a genre. It has two columns that together form its primary key:

- **movie_id**: the ID of the movie being described; a foreign key to **movies**.
- **genre_id**: the ID of the genre attached to that movie; a foreign key to **genres**.

The **has_position** table has one row for each job that a person held on a movie. Its primary key consists of three of the four columns: **person_id**, **movie_id**, and **position**.

- **person_id**: the ID of the person; a foreign key to **people**.
- **movie_id**: the ID of the movie; a foreign key to **movies**.
- **position**: what the person did, one of **actor**, **producer**, **writer**, **composer**, or **director**.
- **job**: for some positions, extra details on the job they did, such as “written by” or “co-producer”. NULL for positions that do not have this extra detail, or where none was provided.

The **plays_role** table has one row for each character that an actor played, with all three columns forming the primary key:

- **person_id**: the ID of the person; a foreign key to **people**.
- **movie_id**: the ID of the movie; a foreign key to **movies**.
- **role**: the name of the character, as IMDb records it.

One caveat applies to all of these tables. IMDb’s data is incomplete, and this database is a filtered subset of it, so a cast member you expect to find may simply be absent, and two movies you expect to be connected through a shared collaborator may have no connection in the data.

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

Slide 24

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	<i>null</i>
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

Slide 25

Slide 24 and 25 show a sample of the actual data, following a single movie through the tables. *The Kid* is one row of **movies**, with the key **tt0012349**, and one row of **ratings** carrying its score. Its three genres appear as three rows of **has_genre**, each pairing the movie's key with a genre's key, so reading the genre *names* requires looking up those keys in **genres**. Charles Chaplin is one row of **people**, and his five positions on the movie are five rows of **has_position**, plus one row of **plays_role** for the character he played. Note how the movie's title appears exactly once in the whole database, and Chaplin's name exactly once, no matter how many rows refer to them. Storing each fact in exactly one place is a central concern of relational design, and we will look at it properly when we come to *normalisation*.

The tables are created with the SQL **CREATE TABLE** statements shown on Slide 26 and 27. This part of SQL, which defines the structure of the data rather than reading or changing it, is called the *data definition language* (DDL). Each column is given a name and a type, and the **PRIMARY KEY** and **REFERENCES** clauses declare the keys that we read off the diagram. Where the primary key consists of a single column it can be declared on that column; where it spans several columns it needs a separate **PRIMARY KEY (...)** clause.

Creating the entity tables in 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  
);
```

Slide 26

Creating the relationship tables in 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)  
);
```

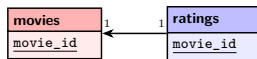
Slide 27

A foreign key declaration is not merely documentation: it is a constraint that the database can enforce. If `has_genre` contains a `movie_id` that does not appear in `movies`, the reference points at nothing, and the database is said to have lost *referential integrity*. A database system that enforces foreign keys will refuse any insert, update, or delete that would leave a dangling reference. DuckDB, which is what we are using, does enforce them, but SQLite, for example, ignores foreign key declarations unless `PRAGMA foreign_keys = ON` is specified. Thus, despite SQL being standardised, you can still find significant variations in behaviour between systems.

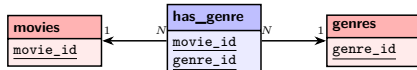
Primary and foreign keys are commonly used to construct three different types of relationship, as shown on Slide 28.

Three shapes of relationship

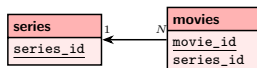
one-to-one: a movie has at most one rating



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

Slide 28

You can tell what kind of relationship it is by looking at the primary key of each table. The `ratings` table has `movie_id` as its entire primary key, and that column is also a foreign key to `movies`. Since a primary key value can appear only once, this means each movie has at most one row in the `ratings` table, and each rating belongs to exactly one movie: a *one-to-one* relationship.

Exercise 2. Instead of having a separate `ratings` table, another way of making a one-to-one relation would be to add `rating` and `votes` as columns of the `movies` table. What are the pros and cons of these approaches? Think about how the tables might be used by an application, how you might enforce constraints, and how future-proof each approach is.

In a *many-to-many* relationship like the `has_genre` table, both columns are foreign keys, and the primary key contains both columns. A table like this, whose only purpose is to connect rows of two other tables, is called an *associative table* or *join table*. Many-to-many relationships cannot be represented by putting a foreign key in either of the two entity tables, so a third table is the standard solution.

The `has_position` and `plays_role` tables represent many-to-many relationships with extra information about the relationship. In the `has_position` table, the primary key is (`movie_id`, `person_id`, `position`), which means that the same person may hold several positions on the same movie: Charles Chaplin has 5 rows for *The Kid*, since he acted in it, directed it, produced it, wrote it, and composed its music. Had the primary key been just (`movie_id`, `person_id`), that would have been impossible to record. The `job` column then holds IMDb's free-text description of the position ("written by", "co-producer"), and is null in 68% of rows. Similarly, `plays_role` has primary key (`movie_id`, `person_id`, `role`), so an actor can play more than one character in a movie; this occurs 1,007 times in the example database.

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 $\implies$ 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.

Slide 29

There is no example of a *one-to-many* relationship in the example database, but we could add one. Suppose we wanted to record which series or franchise a movie belongs to, so that, say, the *Star Wars* films form one series and the *Lord of the Rings* trilogy another. A series contains many movies, and a movie belongs to at most one series, which gives us a one-to-many relationship between series and movies.

Exercise 3. Give the SQL statements that you would use to change the example database to add the concept of a series of movies. You can add a column to an existing table using the `ALTER TABLE` statement; see the [DuckDB documentation](#) for details. How would you record a movie's position within its series (e.g. being part 1, 2, or 3 of a trilogy)? Can you prevent two movies from being given the same position within their series?

Exercise 4. Every type of relationship that a person may have with a movie – acting, directing, producing, writing, composing – is recorded in one table, `has_position`, with a `position` column saying which type each row is. An alternative design would be to create a separate table for each type of relationship: `directs`, `produces`, `composes`, and so on. How would you decide which of these approaches to use?

3 SQL and relational algebra

We've seen some examples of SQL already, but now let's explore it a bit more systematically. SQL is a large and complex language, and we will not be able to cover all of it in this course. However, knowing the core ideas of SQL is essential for working with relational databases.

The best way of learning SQL is not just to read these notes, but also to try everything yourself. We will use DuckDB for SQL, so please get the example database of movies from the course web page, <https://www.cst.cam.ac.uk/teaching/2627/Databases/>.

See <https://duckdb.org/install/?environment=cli> for instructions on installing the DuckDB command line interface on your own machine. You should then be able to simply run `duckdb movies.duckdb` where `movies.duckdb` is the name of the database file you downloaded from the course web page. Alternatively, you can run DuckDB inside your web browser without installing anything at all.

Either way, you have your own copy of the database that is not shared with any other student. Therefore, feel free to do anything you want with the database, including deleting all of the data. If you break the database file, you can always re-download it from the course web page. The browser version does not save changes at all: if you reload the tab, the database is reset to its original state.

These notes only give an outline of the SQL language; for full details please refer to the [DuckDB documentation](#).

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

Slide 30

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)

Slide 31

Alongside SQL we will introduce *relational algebra*, which is a small mathematical notation that corresponds roughly to the core concepts in SQL. Although we don't normally use relational algebra to write queries in practice, it is a useful tool for defining what relational queries actually *mean*, and query optimisers typically use a form of relational algebra internally when planning how to execute a query.

A *relation*, the fundamental building block of relational algebra, is a set of tuples. For now you can think of it as the set of rows in a table, and we will refine this idea later. The Part IA Discrete Maths course offers a more formal introduction to relations.

3.1 Defining and manipulating data

Let's start with DDL. We have already seen examples of it: the CREATE TABLE statements on [Slide 26](#). A CREATE TABLE statement names the table, then lists its columns, each with a name and a type, and any constraints.

The types in our example database are simple: INT for integers, TEXT for strings, NUMERIC (a fixed-point decimal number) for the ratings. Many more types exist: dates, times, booleans, floating-point numbers, JSON, arrays, and more – though their support varies considerably between different DBMSs.

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, ...

Slide 32

Changing and removing tables

```
-- 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

Slide 33

DDL also covers indexes. On [Slide 10](#) we saw an index `movies_by_year` built from a second Python dictionary, which you needed to manually keep consistent with `movies_by_id` ([Slide 11](#)). In SQL, indexes are much easier to work with: you only need to once declare what index you want using `CREATE INDEX`, and then the database automatically keeps the index up-to-date for you.

Creating and deleting indexes

```
-- 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.)

Slide 34

Inserting, updating, and deleting rows

```
-- 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!

Slide 35

3.2 Querying a single table

The **SELECT** statement in SQL is used to query the database. We will start off with simple **SELECT** statements and work our way up to more complex ones. Let's start by reading a whole table: **SELECT * FROM movies** fetches every column of every row. The ***** means "all columns"; **movies** is the table to read.

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

Slide 36

Nothing in that query says anything about the order of the rows. A relation is a *set* of rows (strictly speaking, a multiset – more on this in [Section 3.5](#)), and a set has no order, so the database is free to return the rows in whatever order it finds most convenient. It may return them in one order today, and a different order tomorrow. If you want a particular order you have to ask for it using an **ORDER BY** clause in the **SELECT** statement. If you combine **ORDER BY** with **LIMIT n**, the database will first sort the rows and then return the first *n* of the sorted result.

You can choose to sort either in ascending order, from smallest to largest value (the default), or in descending order from largest to smallest by putting the **DESC** keyword after the column name. You can sort by more than one column, as on [Slide 37](#): the second column is consulted only when the value in the first is equal, the third only where the first two are equal, and so on.

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.

Slide 37

The **WHERE** clause filters the result to keep only rows satisfying a condition. You can express the condition using comparison operators `=`, `<`, `<=`, `>`, `>=`, and `<>`. The `<>` operator is the standard SQL way of writing “not equal”; DuckDB also accepts `!=` to express the same thing. You can combine conditions with **AND**, **OR**, and **NOT** as usual.

This is the first of our relational algebra operators. *Selection*, written σ , takes a relation and a condition, and returns the relation containing just the rows that satisfy it. The condition goes in the subscript and the relation in brackets, so the first query on Slide 38 is $\sigma_{\text{type}='video'}(\text{movies})$ and the second, which narrows it further, is $\sigma_{\text{type}='video' \wedge \text{year} \geq 2012}(\text{movies})$. Because the result is itself a relation, a selection can be applied to the result of another relational algebra operation.

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 :

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

Slide 38

Besides those comparison operators there are additional operators you can use in a **WHERE** clause. The **IN** operator allows you to check whether the value in a column is one of a given list of values, and the **LIKE** operator allows you to do pattern-matching on strings. Most relational databases include a wide variety of additional operators beyond these, for example to support regular expressions or to manipulate various datatypes.

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

Slide 39

WHERE chooses a subset of rows to return; to choose a subset of *columns*, you list them after **SELECT** instead of the *. The corresponding relational algebra operator is *projection*, written π , which takes a relation and a list of column names and returns the relation with only those columns.

Here the names are an unfortunate accident of history. In relational algebra, selection is the operation that filters rows and projection is the one that leaves out some of the columns. SQL's **SELECT** keyword does *projection*, *not selection*; it's the **WHERE** clause that performs selection.

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!

Slide 40

Exercise 5. Write each of the following as a relational algebra expression over the example database, and then as a SQL query.

1. The titles of the movies released in 1995.
2. The titles of the movies that are at least three hours long, together with their running times.

Exercise 6. Both of the following are valid SQL. Which one corresponds to $\sigma_{\text{year}=1995}(\text{movies})$?

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

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

What does the other query return, and why?

Renaming, written ρ , gives a relation or one of its columns a new name. It corresponds to the **AS**

keyword in SQL. It may look trivial, but renaming is important in two situations: when a column of the result is computed (e.g. with an arithmetic expression), and so it has no name of its own; and when a query mentions the same table twice, in which case the two copies of the table need different names so you can tell them apart.

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 :

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

Renaming tables:

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

Needed when a query mentions the same table twice.

Slide 41

3.3 Grouping and aggregation

Every query so far has returned one output row for each input row that satisfied the WHERE clause. Sometimes we don't actually want the individual rows, but rather a summary of them. An *aggregate function* computes a single value from a set of rows, and the GROUP BY clause in SQL says which rows belong to the same set. Rows that have the same value in the grouping columns form a group, and the query returns one row per group.

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 :

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

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

Slide 42

Everything mentioned in the SELECT clause must be either a column in the GROUP BY clause, or an aggregate function. For example, “SELECT year, type, COUNT(*) FROM movies GROUP BY year” does not make sense, because the movies of 2023 may have different values in the type column. (SQLite is unusual in allowing that query and just picking one of the type values arbitrarily, but most other relational databases reject that query.) However, if you also name the type in the GROUP BY clause, the query makes sense again, as shown on [Slide 43](#).

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.

Slide 43

Grouping by several columns gives you a way of breaking down data in a finer-grained way: notice that the rows for 2023 in [Slide 43](#) add up to the total figure for that year on [Slide 42](#).

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

Slide 44

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

Slide 45

Exercise 7. Two different movies may have the same title – for example, a remake. Write a query that returns the titles that are shared by two or more movies in the example database.

3.4 Joins

The *cartesian product*, written $\times$, takes two relations and forms every possible pairing of a row from one with a row from the other. In SQL you get this by naming two tables in the **FROM** clause with nothing to connect them. The cartesian product is something we hardly ever want to actually compute in practice – for example, computing the product of 2,935 movies and 26 genres would return 76,310 rows – but it is useful as a prerequisite for understanding joins.

Cartesian product between two tables

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

`movies  $\times$  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.

Slide 46

In our example database, the rating for a movie is stored in a separate table from the movie itself. If we want to return both a movie's title and its rating in the query result, we need to combine the information across these tables. A *join*, written $\bowtie$, is the relational way of doing this.

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 $\bowtie$:

`movies  $\bowtie_{\text{movie\_id}}$  ratings`
 `$= \sigma_{\text{movies.movie\_id} = \text{ratings.movie\_id}}(\text{movies} \times \text{ratings})$`

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

Slide 47

One way you can think of a join is that it's a cartesian product followed immediately by a selection: form all the pairings of movies and ratings (most of which will be ratings paired with the wrong movie!), then keep only those where the two rows are about the same movie. That's what is done by the *nested loop join* on Slide 48: the two nested loops compute the cartesian product, and the **if** condition performs the selection. However, this algorithm would be unusably slow in many situations.

In practice, various optimised join algorithms are used, which compute the same result as the nested loop join, but which do so much more efficiently. The *index nested loop join* on Slide 48 is one of them: given an index on `ratings.movie_id`, the engine scans the `movies` table once and looks up each movie's

rating directly, instead of ever forming the pairings it would immediately throw away. Other approaches also exist, and choosing between them is the job of the query optimiser.

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

Slide 48

A query often has to follow more than one relationship at a time. To list the cast of a movie we need three tables: **movies** for the title, **people** for the actor's name, and **plays_role** in between to say which person played which part in which movie. Each JOIN adds one more table, and each needs its own ON condition saying which columns must match.

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

movies ⋈_{movie_id} plays_role ⋈_{person_id} people

Joins are associative: the DBMS chooses the execution order.

Slide 49

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.

Slide 50

Writing out a join condition such as `ON movies.movie_id = ratings.movie_id` is tedious when the two columns have the same name. SQL therefore offers `USING`, which names the column once and means the same thing. It has an additional effect: whereas `ON` leaves both columns in the result, `USING` merges them into a single column, which is what you usually want.

Exercise 8. Write a relational algebra expression, and then a SQL query, that returns the title of every movie paired with the name of every genre it belongs to. That query will return several rows for the same movie if it belongs to multiple genres. Next, modify the query so that it returns one row per movie, with the movie's genres gathered into a single comma-separated string. (Hint: use `GROUP BY` together with DuckDB's `string_agg()` aggregate function.)

A `JOIN` produces an output row only when a matching row exists on both of its inputs. This is called an *inner join*, and it is usually what you want. But sometimes it is not quite right: for example, if a movie has no actors (e.g. because it is dialogue-free animation), the query on [Slide 49](#) will never return any rows for that movie. A *left outer join*, or simply *left join*, handles this case by keeping all rows on the left-hand side of the join, and filling in `NULL` values in cases where there is no matching row on the right-hand side of the join.

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	Tintrínai Thikhasuk
Flow	null	null

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

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

Slide 51

`m.title IN ('Arctic', 'Flow')` is the same as `m.title = 'Arctic' OR m.title = 'Flow'`. For any movie `m` where there is no row `r` in `plays_role` such that `m.movie_id = r.movie_id`, the DBMS fills in all columns of `r` with `NULL` values, including `r.role` and `r.person_id`. Next, when joining with the `people` table, there is no person `p` such that `r.person_id = p.person_id` if `r.person_id` is `NULL`. Since the second join is also a left join, the columns of `p` are also filled with `NULL`, including `p.name`.

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.

Slide 52

Slide 52 shows pseudocode of how a left join might be executed. As with inner joins, the details are decided by the query optimiser.

Exercise 9. Write a SQL query that returns the ID and title of all the movies in the example database that have no actors, using `LEFT JOIN`. (Hint: to test whether the value in some column `x` is null you have to use the operator `x IS NULL`, not `x = NULL` as you might expect. The query should return 2 rows on the example database.)

(We will look at different ways of searching for movies that have no actors on [Slide 64](#) and [65](#), but for now please answer the question using `LEFT JOIN`.)

Outer joins

JOIN	⋈	matching rows only
LEFT JOIN	⋈ _L	all left rows, null on unmatched right
RIGHT JOIN	⋈ _R	all right rows, null on unmatched left
FULL JOIN	⋈ _⋈	all rows of both, null on either side

Example of full outer join:

Slide 53

A right outer join, $\bowtie$ or `RIGHT JOIN`, is the same as a left join with the two sides swapped. A full outer join, $\Join$ or `FULL JOIN`, keeps the unmatched rows of both sides. You can also equivalently write `LEFT OUTER JOIN`, `RIGHT OUTER JOIN`, or `FULL OUTER JOIN` if you prefer.

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.

Slide 54

A join condition is not restricted to a single equality: it is an arbitrary expression, and may compare several columns or apply a restriction to one side of the join. With an inner join, it makes no difference to the query result whether you put the condition in the `ON` clause or the `WHERE` clause. However, with a left/right/full outer join, it does matter where you put the condition: if the condition in the `ON` clause is false, a row with some null values is returned; if the condition in the `WHERE` clause is false, the row is not returned at all.

3.5 Comparing relational algebra and SQL

There is a discrepancy between relational algebra and SQL that we have so far glossed over. Mathematically, a relation is a *set* of tuples: a tuple either is a member or it is not, but it is not possible for a relation to contain the same tuple more than once. In contrast, SQL tables and query results are *multisets*, also called bags: the same row may occur any number of times.

The difference shows up as soon as a projection discards the column that distinguished two rows. In relational algebra, $\pi_{\text{type}}(\text{movies})$ is a set, and so it has one element for each distinct value of `type`, regardless of however many movies have it. In SQL, `SELECT type FROM movies` returns one row per movie, with the same few values repeated thousands of times. If you want to remove duplicates in SQL, you have to use `SELECT DISTINCT`.

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

Slide 55

`SELECT DISTINCT` thus turns a multiset back into a set. Note that `DISTINCT` applies to the whole row of the result, not to one column: `SELECT DISTINCT type, year FROM movies` returns the distinct *pairs*. Performing deduplication has a computational cost, which is one reason why it is not performed by default. Another reason is that when performing aggregations, removing duplicates before aggregation may change the result in undesirable ways. The standard set operations *union* ($\cup$), *intersection* ($\cap$), and

difference $(-)$ apply to multisets as well.

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;
```

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

Slide 56

While UNION, INTERSECT and EXCEPT remove duplicates, there are also operators UNION ALL, INTERSECT ALL, and EXCEPT ALL that preserve duplicates.

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.

Slide 57

Exercise 10. The following query returns the IDs of the movies that belong to both the Crime genre and the Drama genre:

```
-- 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;
```

Write a query that returns the same result using a join instead of INTERSECT.

Exercise 11. Write a SQL query that returns the ID and title of all the movies in the example database that have no actors, like in [Exercise 9](#), but this time using `EXCEPT` instead of `LEFT JOIN`. Why would it be a mistake to write this query returning the title alone?

Another peculiarity of SQL that doesn't appear in the relational algebra concerns the handling of NULL. By default, SQL allows nulls in columns of any type, unless the column is declared with a NOT NULL or primary key constraint. For example, 88% of the `deathyear` entries in `people` are null, since

most of the people in the database are alive (though it could also be the case that the person is dead and the year of their death is unknown). Nulls are also produced by outer joins when no matching row is found.

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

Slide 58

= or <> comparisons with NULL return neither TRUE, nor FALSE, but NULL. As a result, SQL's logic is not binary, but three-valued, where the inputs and return values of the logical operators AND, OR and NOT can be either TRUE, FALSE, or 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.

Slide 59

Watch out when performing comparisons on columns that might be null. For example:

- “**SELECT COUNT(*) FROM people WHERE deathyear < 2000**” returns 959;
- “**SELECT COUNT(*) FROM people WHERE NOT (deathyear < 2000)**” returns 1,726;
- together that is far short of the 23,179 rows in `people` because the 20,494 rows with a null `deathyear` are not counted in either query.

Aggregate functions ignore nulls: COUNT(c), SUM(c) and AVG(c) are computed over the rows where column c is not null. GROUP BY, DISTINCT and the set operations treat nulls as equal; thus, grouping by a nullable column collects all the null rows into a single group. This is slightly illogical, since according to the = operator, nulls are not equal. There is an operator called IS NOT DISTINCT FROM which behaves the same as =, except that it treats nulls as equal and always returns TRUE or FALSE. Its negation is called IS DISTINCT FROM, which behaves similarly to <> except for nulls. (Yes, this is all very confusing.)

Finally, a trap worth knowing about: NOT IN over a list containing a null is never true, as explained on Slide 64. For example, the expression 3 NOT IN (1, 2, NULL) evaluates to NULL, even though 3 plainly is not in the list. The positive form is safe: 1 IN (1, 2, NULL) finds the occurrence of 1 and returns

true regardless of the NULL. You can use the NOT EXISTS operator (Slide 65) instead of NOT IN to avoid this problem.

Exercise 12. In boolean logic, the expression `not (a and b)` is equivalent to `(not a) or (not b)`, and the expression `not (a or b)` is equivalent to `(not a) and (not b)`. Is the same true in 3-valued logic?

Is the expression `a or (not a)` always true in 3-valued logic? If not, is there a similar expression that is always true regardless of the value of `a`?

3.6 Query composition

Relational algebra composes nicely: every operator takes relations and returns a relation, so operators can be nested arbitrarily. For example, you can select rows, group/aggregate them, and then select again. In contrast, a SQL query has a fixed sequence of clauses – **SELECT**, **FROM**, **WHERE**, **GROUP BY**, **HAVING**, **ORDER BY** – each of which may appear only once, in that order. Where relational algebra may apply σ twice within the same expression, SQL has two different keywords: **WHERE** for the selection that happens before grouping, and **HAVING** for the one that happens after. It's somewhat clunky that the same concept needs two different names in SQL.

Composing queries

Relational algebra composes

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

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

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.

Slide 60

You don't have to use the **HAVING** clause though: you can get the same result by using a *subquery* in SQL, which allows you to use one query's result as the input to another query. A subquery written in the **FROM** clause, in place of a table name, is called a derived table: the inner query is evaluated and its result used as though it were a stored table.

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)

Slide 61

The query on [Slide 61](#) is equivalent to the one on [Slide 45](#), and which one you prefer is perhaps a matter of taste. Note that the derived table is given a name, `per_year`; the SQL standard requires that you give a derived table a name even when that name is never used, although DuckDB doesn't require one.

Nested subqueries enable composition, but they can be hard to read – essentially, you need to read the query “inside out” to understand what it's doing. A *common table expression*, introduced by `WITH`, is functionally equivalent but often more readable, as it can be read top to bottom.

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.

Slide 62

A subquery may also appear in a `WHERE` clause. If it returns a single row with a single column (this is called a *scalar* subquery) it can be used anywhere a value can. A subquery makes it possible to compare a row to an aggregate of the whole table, since an aggregate function may not appear directly in a `WHERE` clause.

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.

Slide 63

The subquery on [Slide 63](#) is evaluated once and turned into a single number before running the outer query. This works because the subquery `SELECT AVG(rating) FROM ratings` does not refer to anything in the outer query. It is also possible for the subquery to return a single column, but multiple rows. In that case you can't use it with the `>` operator or other operators that compare individual values, but you can use it with the `IN` operator that we saw on [Slide 39](#), or its negation `NOT IN`, which work with a set of values.

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:

- ▶ $c \text{ NOT IN } (v_1, v_2, \dots, v_n)$ expands to $(c \neq v_1) \text{ AND } (c \neq v_2) \text{ AND } \dots \text{ AND } (c \neq v_n)$
- ▶ If v_i is NULL, $(c \neq v_i)$ evaluates to NULL
- ▶ TRUE AND NULL evaluates to NULL
- ▶ So $c \text{ NOT IN } (v_1, v_2, \dots, v_n)$ evaluates to NULL
- ▶ So the whole query returns no rows

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

Slide 64

The problem with NOT IN and NULL does not apply in the particular example on [Slide 64](#), since plays_role.movie_id can never be NULL. But it can happen – here's a (slightly contrived) example:

```
-- Which movies came out in a year in which none
-- of the people in the database died?
SELECT title FROM movies
WHERE year NOT IN (SELECT deathyear FROM people)
```

The intended query result should contain 4 rows, but in fact this query returns 0 rows due to the nulls in the deathyear column. If you restrict the subquery with WHERE deathyear IS NOT NULL the whole query returns 4 rows as expected.

Exercise 13. The following pairs of queries are over two tables, *r* with columns *x* and *y*, and *s* with columns *y* and *z*. For each pair, say whether the two queries always return the same result. If they do not, give an example database on which the results differ.

1.

```
SELECT r.x FROM r WHERE r.y IN (SELECT s.y FROM s);
```



```
SELECT r.x FROM r JOIN s ON r.y = s.y;
```
2.

```
SELECT r.x FROM r JOIN s ON r.y = s.y AND s.z = 99;
```



```
SELECT r.x FROM r JOIN s ON r.y = s.y WHERE s.z = 99;
```
3.

```
SELECT r.x FROM r LEFT JOIN s ON r.y = s.y AND s.z = 99;
```



```
SELECT r.x FROM r LEFT JOIN s ON r.y = s.y WHERE s.z = 99;
```
4.

```
SELECT r.x FROM r LEFT JOIN s ON r.y = s.y WHERE s.y IS NULL;
```



```
SELECT r.x FROM r WHERE r.y NOT IN (SELECT s.y FROM s);
```

The danger with nulls and NOT IN can also be avoided by instead using the NOT EXISTS operator, which is not vulnerable to this problem. However, it requires a more complex type of subquery where the subquery refers to columns in the outer query (this is known as a *correlated* subquery). For example, on [Slide 65](#), the subquery refers to m.movie_id, a column of the table *m* defined in the outer query. You can imagine a correlated subquery being separately evaluated for each candidate row of the outer query (although the actual query plan may do it more efficiently). The NOT EXISTS operator is known in relational algebra as an *antijoin*.

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 $\triangleright$:

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

Slide 65

Slide 65 shows another way of computing the same result as in Exercise 9. Which one to prefer is mostly a matter of readability; a good query optimiser will execute them in the same way.

The positive form, **EXISTS**, has a name of its own too: it is known in relational algebra as a *semijoin*, written $\bowtie$. This operator returns the rows of its left relation that have at least one match on the right, but (unlike an inner join) it does not include any values from the right relation in the query result. If a row on the left has multiple matches on the right, an inner join would output multiple rows (one per match, with the values from the left row duplicated), but the semijoin only outputs at most one row for each row on the left.

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 $\bowtie$:

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

Slide 66

There are 3,281 composer credits in **has_position** but only 1,086 people holding them, because a composer may write the score for many movies; Hans Zimmer alone accounts for dozens. Joining **people** to **has_position** would therefore return a name once per credit, where the **EXISTS** form returns each person once and expresses the intention of the query more directly.

Exercise 14. Write a query that returns the names of people who are credited as a composer on at least one movie, like on the slide, but using an inner join instead of the **EXISTS** or **IN** operator. Be careful that it still gives the right answer if two different people in the database happen to have the same name.

The pseudocode on Slide 67 is one simple way in which a DBMS might execute these queries (although it might well use better algorithms in practice). The **break** in the semijoin is what distinguishes it from an ordinary join: having found one match, there is no reason to look for another. The antijoin runs the same search but outputs the row only when there is no match.

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)
```

Slide 67

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	$\bowtie\!\!\!\!\!\lrcorner$	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**.

Slide 68

References

- Daniel J Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. The design and implementation of modern column-oriented database systems. *Foundations and Trends in Databases*, 5(3):197–280, 2013. doi:[10.1561/19000000024](https://doi.org/10.1561/19000000024). URL <https://www.cs.umd.edu/~abadi/papers/abadi-column-stores.pdf>.
- Rina Diane Caballar. The rise of SQL. *IEEE Spectrum*, August 2022. URL <https://spectrum.ieee.org/the-rise-of-sql>.
- E.F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970. doi:[10.1145/362384.362685](https://doi.org/10.1145/362384.362685).
- IMDb. IMDb non-commercial datasets. <https://developer.imdb.com/non-commercial-datasets/>, 2025.
- Evan Jones. Why databases use ordered indexes but programming uses hash tables, December 2019. URL <https://www.evanjones.ca/ordered-vs-unordered-indexes.html>.
- Neel Krishnaswami. What declarative languages are, July 2013. URL <https://semantic-domain.blogspot.com/2013/07/what-declarative-languages-are.html>.
- SQLite contributors. Most widely deployed and used database engine. <https://sqlite.org/mostdeployed.html>.