SQL: The Structured Query Language

COMPSCI 2DB3: Databases

Jelle Hellings Holly Koponen

Department of Computing and Software
McMaster University

Winter 2024

What is SQL?

Simple answer

A *query language* for interacting with relational databases.

A *high-level programming language* for relational data in which you specify *what* you want, not *how to do it*.

What is SQL?

Simple answer

A *query language* for interacting with relational databases.
A *high-level programming language* for relational data in which you specify *what* you want, not *how to do it*.

Official answer: An ANSI and ISO standard

`Information technology–Database languages–SQL`
`Part 2: Foundation (SQL/Foundation)`

Source: https://www.iso.org/standard/63555.html.

What is SQL?

Simple answer

A *query language* for interacting with relational databases.
A *high-level programming language* for relational data in which you specify *what* you want, not *how to do it*.

Official answer: An ANSI and ISO standard

`Information technology–Database languages–SQL`
`Part 2: Foundation (SQL/Foundation)` ⟶ 1707 pages!

There are eight other parts.

Corrigendum in 2019 with one additional part.

Source: https://www.iso.org/standard/63555.html.

What is SQL?

Simple answer

A *query language* for interacting with relational databases.
A *high-level programming language* for relational data in which you specify *what* you want, not *how to do it*.

Official answer: An ANSI and ISO standard

`Information technology–Database languages–SQL`
`Part 2: Foundation (SQL/Foundation)` ⟶ 1707 pages!

There are eight other parts.

Corrigendum in 2019 with one additional part.

We will only look at a *very small* part of SQL.

Source: https://www.iso.org/standard/63555.html.

Classes of SQL Statements

DML *Data Manipulation Language*: SQL-data and SQL-data change statements.
Create, modify, delete, and inspect stored data (data in tables).

DDL *Data Definition Language*: SQL-schema statements.
Create, modify, delete, and inspect the schema (structure of tables).

SQL-data and SQL-data change statements

We will first look at *data manipulation*.

SQL-data and SQL-data change statements

We will first look at *data manipulation*.

Four main types of SQL-data and SQL-data change statements

INSERT	Insert row(s) into a table.
DELETE	Delete row(s) from a table.
UPDATE	Update row(s) in a table.
SELECT	Retrieve data from table(s).

SQL-data and SQL-data change statements

We will first look at *data manipulation*.

Four main types of SQL-data and SQL-data change statements

INSERT	Insert row(s) into a table.
DELETE	Delete row(s) from a table.
UPDATE	Update row(s) in a table.
SELECT	Retrieve data from table(s). ⟵ *Our main focus*

Single table: Retrieve a single table

courses

cid	title	lecturer
1	Programming	1
2	Discrete Mathematics	3
3	Databases	2
4	Advanced Databases	2

```
SELECT *
FROM courses;
```

Single table: Retrieve a single table

courses

cid	title	lecturer
1	Programming	1
2	Discrete Mathematics	3
3	Databases	2
4	Advanced Databases	2

```
SELECT *                -- SELECT specifies the columns: * is all columns.
FROM courses;
```

Single table: Retrieve a single table

courses

cid	title	lecturer
1	Programming	1
2	Discrete Mathematics	3
3	Databases	2
4	Advanced Databases	2

```
SELECT *                -- SELECT specifies the columns: * is all columns.
FROM courses;           -- FROM specifies the table(s): courses.
```

Single table: Retrieve a single table

courses

cid	title	lecturer
1	Programming	1
2	Discrete Mathematics	3
3	Databases	2
4	Advanced Databases	2

Query output

cid	title	lecturer
1	Programming	1
2	Discrete Mathematics	3
3	Databases	2
4	Advanced Databases	2

```
SELECT *
FROM courses;
```

-- **SELECT** specifies the columns: * is *all columns*.
-- **FROM** specifies the table(s): **courses**.

Returns a copy of the table as-is.

Single table: Retrieve specific columns

courses

cid	title	lecturer
1	Programming	1
2	Discrete Mathematics	3
3	Databases	2
4	Advanced Databases	2

```
SELECT lecturer
FROM courses;          -- FROM specifies the table(s): courses.
```

Single table: Retrieve specific columns

courses

cid	title	lecturer
1	Programming	1
2	Discrete Mathematics	3
3	Databases	2
4	Advanced Databases	2

```
SELECT lecturer
FROM courses;
```

```
-- SELECT specifies the columns: lecturer.
-- FROM specifies the table(s): courses.
```

Single table: Retrieve specific columns

courses

cid	title	lecturer
1	Programming	1
2	Discrete Mathematics	3
3	Databases	2
4	Advanced Databases	2

$\longrightarrow$

Query output

lecturer
1
3
2
2

```
SELECT lecturer
FROM courses;
```

```
-- SELECT specifies the columns: lecturer.
-- FROM specifies the table(s): courses.
```

Returns a table with a copy of the column lecturer from courses.

Single table: Retrieve specific columns

courses

cid	title	lecturer
1	Programming	1
2	Discrete Mathematics	3
3	Databases	2
4	Advanced Databases	2

⟶

Query output

lecturer
1
3
2
2

} *Multiset!*

```
SELECT lecturer          -- SELECT specifies the columns: lecturer.
FROM courses;            -- FROM specifies the table(s): courses.
```

Returns a table with a copy of the column lecturer from courses.

Single table: Retrieve specific columns

courses

cid	title	lecturer
1	Programming	1
2	Discrete Mathematics	3
3	Databases	2
4	Advanced Databases	2

Query output

lecturer
1
3
2

Set

```
SELECT DISTINCT lecturer   -- SELECT specifies the columns: lecturer.
FROM courses;              -- FROM specifies the table(s): courses.
```

Returns a table with a copy of the column lecturer from courses (unique values).

Single table: Retrieving specific rows

courses

cid	title	lecturer
1	Programming	1
2	Discrete Mathematics	3
3	Databases	2
4	Advanced Databases	2

```
SELECT *
FROM courses
WHERE lecturer = 2;
```

-- **SELECT** specifies the columns: * is all columns.
-- **FROM** specifies the table(s): **courses**.

Single table: Retrieving specific rows

courses

cid	title	lecturer
1	Programming	1
2	Discrete Mathematics	3
3	Databases	2
4	Advanced Databases	2

```
SELECT *                -- SELECT specifies the columns: * is all columns.
FROM courses            -- FROM specifies the table(s): courses.
WHERE lecturer = 2;     -- WHERE specifies conditions on each row.
```

Single table: Retrieving specific rows

courses

cid	title	lecturer
1	Programming	1
2	Discrete Mathematics	3
3	Databases	2
4	Advanced Databases	2

Query output

cid	title	lecturer
3	Databases	2
4	Advanced Databases	2

```
SELECT *
FROM courses
WHERE lecturer = 2;
```

-- **SELECT** specifies the columns: * is all columns.
-- **FROM** specifies the table(s): **courses**.
-- **WHERE** specifies conditions on each row.

Returns a copy of the rows in the table that meet the condition.

Single table: Retrieving specific rows

courses

cid	title	lecturer
1	Programming	1
2	Discrete Mathematics	3
3	Databases	2
4	Advanced Databases	2

Query output

cid	title	lecturer
2	Discrete Mathematics	3

```
SELECT *
FROM courses
WHERE cid < lecturer;
```

-- **SELECT** specifies the columns: * is all columns.
-- **FROM** specifies the table(s): **courses**.
-- **WHERE** specifies conditions on each row.
-- Conditions don't need to make sense!

Returns a copy of the rows in the table that meet the condition.

Single table: Retrieving specific rows (text)

courses

cid	title	lecturer
1	Programming	1
2	Discrete Mathematics	3
3	Databases	2
4	Advanced Databases	2

```
SELECT *                        -- SELECT specifies the columns: * is all columns.
FROM courses                    -- FROM specifies the table(s): courses.
WHERE title = 'Databases';      -- WHERE specifies conditions on each row.
```

Single table: Retrieving specific rows (text)

courses

cid	title	lecturer
1	Programming	1
2	Discrete Mathematics	3
3	Databases	2
4	Advanced Databases	2

```
SELECT *                          -- SELECT specifies the columns: * is all columns.
FROM courses                      -- FROM specifies the table(s): courses.
WHERE title = 'Databases';        -- WHERE specifies conditions on each row.
                                  -- 'Exact' text match.
```

Single table: Retrieving specific rows (text)

courses

cid	title	lecturer
1	Programming	1
2	Discrete Mathematics	3
3	Databases	2
4	Advanced Databases	2

⟶

Query output

cid	title	lecturer
3	Databases	2

```
SELECT *
FROM courses
WHERE title = 'Databases';
```

-- **SELECT** specifies the columns: * is all columns.
-- **FROM** specifies the table(s): **courses**.
-- **WHERE** specifies conditions on each row.
-- 'Exact' text match.

Returns a copy of the rows in the table that meet the text condition.

Single table: Retrieving specific rows (text)

courses

cid	title	lecturer
1	Programming	1
2	Discrete Mathematics	3
3	Databases	2
4	Advanced Databases	2

⟶

Query output

cid	title	lecturer
3	Databases	2

```
SELECT *
FROM courses
WHERE title = 'databases';
```

```
-- SELECT specifies the columns: * is all columns.
-- FROM specifies the table(s): courses.
-- WHERE specifies conditions on each row.
-- 'Exact' text match (influenced by the collation).
```

Returns a copy of the rows in the table that meet the text condition.

Single table: Retrieving specific rows (text)

courses

cid	title	lecturer
1	Programming	1
2	Discrete Mathematics	3
3	Databases	2
4	Advanced Databases	2

Query output

cid	title	lecturer
3	Databases	2
4	Advanced Databases	2

```
SELECT *                    -- SELECT specifies the columns: * is all columns.
FROM courses                -- FROM specifies the table(s): courses.
WHERE title LIKE            -- WHERE specifies conditions on each row.
        '%Databases%';      -- Text containing 'Databases': % matches any text.
```

Returns a copy of the rows in the table that meet the text condition.

Single table: Computations on returned columns

courses

cid	title	lecturer
1	Programming	1
2	Discrete Mathematics	3
3	Databases	2
4	Advanced Databases	2

```
SELECT *, cid + 5,
       cid * lecturer
FROM courses;
```

-- **SELECT** specifies the columns: `*` is all columns.

-- **FROM** specifies the table(s): **courses**.

Single table: Computations on returned columns

courses

cid	title	lecturer
1	Programming	1
2	Discrete Mathematics	3
3	Databases	2
4	Advanced Databases	2

```
SELECT *, cid + 5,          -- SELECT specifies the columns: * is all columns.
       cid * lecturer       -- New columns can be computed out of existing data.

FROM courses;               -- FROM specifies the table(s): courses.
```

Single table: Computations on returned columns

courses

cid	title	lecturer
1	Programming	1
2	Discrete Mathematics	3
3	Databases	2
4	Advanced Databases	2

⟶

Query output

cid	title	lecturer	*C1*	*C2*
1	Programming	1	6	1
2	Discrete Mathematics	3	7	6
3	Databases	2	8	6
4	Advanced Databases	2	9	8

```
SELECT *, cid + 5,          -- SELECT specifies the columns: * is all columns.
       cid * lecturer       -- New columns can be computed out of existing data.

FROM courses;               -- FROM specifies the table(s): courses.
```

Returns the table with two added columns computed from the data.

Single table: Computations on returned columns

courses

cid	title	lecturer
1	Programming	1
2	Discrete Mathematics	3
3	Databases	2
4	Advanced Databases	2

⟶

Query output

cid	title	lecturer	*one*	*two*
1	Programming	1	6	1
2	Discrete Mathematics	3	7	6
3	Databases	2	8	6
4	Advanced Databases	2	9	8

```
SELECT *, cid + 5 AS one,      -- SELECT specifies the columns: * is all columns.
       cid * lecturer          -- New columns can be computed out of existing data.
           AS two              -- AS will rename a column in the output table.
FROM courses;                  -- FROM specifies the table(s): courses.
```

Returns the table with two added columns computed from the data (named columns).

Selecting data from multiple tables

courses

cid	title	lecturer
2	Discrete Mathematics	3
3	Databases	2

faculty

fid	name	rank
2	Bo	Assistant
3	Celeste	Associate

Selecting data from multiple tables

courses

cid	title	lecturer
2	Discrete Mathematics	3
3	Databases	2

faculty

fid	name	rank
2	Bo	Assistant
3	Celeste	Associate

'Return courses together with the name of their lecturer.'

Selecting data from multiple tables

courses

cid	title	lecturer
2	Discrete Mathematics	3
3	Databases	2

faculty

fid	name	rank
2	Bo	Assistant
3	Celeste	Associate

'Return courses together with the name of their lecturer.'

```
SELECT *                -- SELECT specifies the columns: * is all columns.
FROM courses, faculty;  -- FROM specifies the table(s): courses, faculty.
```

Selecting data from multiple tables

courses

cid	title	lecturer
2	Discrete Mathematics	3
3	Databases	2

faculty

fid	name	rank
2	Bo	Assistant
3	Celeste	Associate

'Return courses together with the name of their lecturer.'

```
SELECT *                -- SELECT specifies the columns: * is all columns.
FROM courses, faculty;  -- FROM specifies the table(s): courses, faculty.
```

Return the table with all combinations of rows from courses and faculty.

Query output

cid	title	lecturer	fid	name	rank
2	Discrete Mathematics	3	2	Bo	Assistant
2	Discrete Mathematics	3	3	Celeste	Associate
3	Databases	2	2	Bo	Assistant
3	Databases	2	3	Celeste	Associate

Selecting data from multiple tables

courses

cid	title	lecturer
2	Discrete Mathematics	3
3	Databases	2

faculty

fid	name	rank
2	Bo	Assistant
3	Celeste	Associate

'Return courses together with the name of their lecturer.'

```
SELECT *                -- SELECT specifies the columns: * is all columns.
FROM courses, faculty;  -- FROM specifies the table(s): courses, faculty.
```

Return the table with all combinations of rows from courses and faculty.

Query output

cid	title	lecturer	fid	name	rank
~~2~~	~~Discrete Mathematics~~	~~3~~	~~2~~	~~Bo~~	~~Assistant~~
2	Discrete Mathematics	3	3	Celeste	Associate
3	Databases	2	2	Bo	Assistant
~~3~~	~~Databases~~	~~2~~	~~3~~	~~Celeste~~	~~Associate~~

Selecting data from multiple tables

courses

cid	title	lecturer
2	Discrete Mathematics	3
3	Databases	2

faculty

fid	name	rank
2	Bo	Assistant
3	Celeste	Associate

'Return courses together with the name of their lecturer.'

```
SELECT *                    -- SELECT specifies the columns: * is all columns.
FROM courses, faculty       -- FROM specifies the table(s): courses, faculty.
WHERE lecturer = fid;       -- WHERE specifies conditions on each row.
```

Selecting data from multiple tables

courses

cid	title	lecturer
2	Discrete Mathematics	3
3	Databases	2

faculty

fid	name	rank
2	Bo	Assistant
3	Celeste	Associate

'Return courses together with the name of their lecturer.'

```
SELECT *                    -- SELECT specifies the columns: * is all columns.
FROM courses, faculty       -- FROM specifies the table(s): courses, faculty.
WHERE lecturer = fid;       -- WHERE specifies conditions on each row.
```

Return the table with some combinations of rows from courses and faculty.

Query output

cid	title	lecturer	fid	name	rank
2	Discrete Mathematics	3	3	Celeste	Associate
3	Databases	2	2	Bo	Assistant

Selecting data from multiple tables

courses

cid	title	lecturer
2	Discrete Mathematics	3
3	Databases	2

faculty

fid	name	rank
2	Bo	Assistant
3	Celeste	Associate

'Return courses together with the name of their lecturer.'

```
SELECT title, name          -- SELECT specifies the columns: title and name only.
FROM courses, faculty       -- FROM specifies the table(s): courses, faculty.
WHERE lecturer = fid;       -- WHERE specifies conditions on each row.
```

Return the table with some columns from some combinations of rows from courses and faculty.

Query output

title	name
Discrete Mathematics	Celeste
Databases	Bo

SQL: You specify what you want, not how to do it

```
SELECT title, name
FROM courses, faculty
WHERE lecturer = fid;
```

Same query in a structured programming language

```
for i = 0 to |courses| do
  for j = 0 to |faculty| do
    if courses[i].lecturer = faculty[j].fid then
      output (courses[i].title = faculty[j].name)
```

SQL: You specify what you want, not how to do it

```
SELECT title, name
FROM courses, faculty
WHERE lecturer = fid;
```

Same query in a structured programming language

for $i = 0$ **to** $|courses|$ **do** — Assumption: *courses* is an in-memory array.
  for $j = 0$ **to** $|faculty|$ **do** — Assumption: *faculty* is an in-memory array.
    if $courses[i].lecturer = faculty[j].fid$ **then**
      output (*courses*[i].*title* = *faculty*[j].*name*)

SQL: You specify what you want, not how to do it

```
SELECT title, name
FROM courses, faculty
WHERE lecturer = fid;
```

Same query in a structured programming language

for $i = 0$ **to** $|courses|$ **do**
  for $j = 0$ **to** $|faculty|$ **do**
    if $courses[i].lecturer = faculty[j].fid$ **then**
      output ($courses$[i].$title$ = $faculty$[j].$name$)

} *Costly*: $O(|courses| \times |faculty|)$

SQL: You specify what you want, not how to do it

```
SELECT title, name
FROM courses, faculty
WHERE lecturer = fid;
```

Same query in a structured programming language

for $i = 0$ **to** $|courses|$ **do**
  for $j = 0$ **to** $|faculty|$ **do**
    if $courses[i].lecturer = faculty[j].fid$ **then**
      output ($courses$[i].$title$ = $faculty$[j].$name$)

} *Costly*: $O(|courses| \times |faculty|)$

Faster alternative: index structure (hash map, search tree) to look up *faculty*.

SQL: You specify what you want, not how to do it

```
SELECT title, name
FROM courses, faculty
WHERE lecturer = fid;
```

Same query in a structured programming language

for $i = 0$ **to** $|courses|$ **do**
  for $j = 0$ **to** $|faculty|$ **do**
    if $courses[i].lecturer = faculty[j].fid$ **then**
      output ($courses$[i].*title* = $faculty$[j].*name*)

} *Costly*: $O(|courses| \times |faculty|)$

Faster alternative: index structure (hash map, search tree) to look up *faculty*.

Completely different code!

SQL: You specify what you want, not how to do it

```
SELECT title, name
FROM courses, faculty
WHERE lecturer = fid;
```

Same query in a structured programming language

for $i = 0$ **to** $|courses|$ **do**
  for $j = 0$ **to** $|faculty|$ **do**
    if $courses[i].lecturer = faculty[j].fid$ **then**
      output ($courses$[i].*title* = $faculty$[j].*name*)

} *Costly*: $O(|courses| \times |faculty|)$

Faster alternative: index structure (hash map, search tree) to look up *faculty*.
Completely different code!

SQL query: complexity likely very low, e.g., $O(|result| + log\text{-}terms)$.

Sorting returned rows

courses

cid	title	lecturer
1	Programming	1
2	Discrete Mathematics	3
3	Databases	2
4	Advanced Databases	2

```
SELECT title, lecturer   -- Default: ordering of output is implementation defined.
FROM courses;            -- E.g., due to how data is stored and algorithms used.
```

Sorting returned rows

courses

cid	title	lecturer
1	Programming	1
2	Discrete Mathematics	3
3	Databases	2
4	Advanced Databases	2

$\longrightarrow$

Query output

title	lecturer
Advanced Databases	2
Databases	2
Discrete Mathematics	3
Programming	1

```
SELECT title, lecturer   -- Default: ordering of output is implementation defined.
FROM courses             -- E.g., due to how data is stored and algorithms used.
ORDER BY title;          -- You can specify an explicit ordering of the output.
```

Sorting returned rows

courses

cid	title	lecturer
1	Programming	1
2	Discrete Mathematics	3
3	Databases	2
4	Advanced Databases	2

Query output

title	lecturer
Discrete Mathematics	3
Advanced Databases	2
Databases	2
Programming	1

```
SELECT title, lecturer   -- Default: ordering of output is implementation defined.
FROM courses             -- E.g., due to how data is stored and algorithms used.
ORDER BY lecturer DESC, title ASC;
                         -- The ordering can be DESCending or ASCending.
                         -- ASCending is the default if not specified.
```

Intermission: Syntax for creating and deleting tables

CREATE TABLE *table name*
(
  column specifications & *optional constraint specifications*
);

Intermission: Syntax for creating and deleting tables

CREATE TABLE *table name*
(
  column specifications & *optional constraint specifications*
);

DROP TABLE *table name*;

Intermission: Creating a normal table

```
CREATE TABLE reviews
(
  pid INT NOT NULL,
  rid INT NOT NULL,
  quality INT NOT NULL,
  originality INT NOT NULL,
  description CLOB NOT NULL,

  PRIMARY KEY(pid, rid)
);
```

Intermission: Creating a normal table

```
CREATE TABLE reviews        -- Create a table named 'reviews'.
(
  pid INT NOT NULL,
  rid INT NOT NULL,
  quality INT NOT NULL,
  originality INT NOT NULL,
  description CLOB NOT NULL,

  PRIMARY KEY(pid, rid)
);
```

Intermission: Creating a normal table

```
CREATE TABLE reviews        -- Create a table named 'reviews'.
(
  pid INT NOT NULL,         -- Column 'pid' has type INT.
  rid INT NOT NULL,
  quality INT NOT NULL,
  originality INT NOT NULL,
  description CLOB NOT NULL,

  PRIMARY KEY(pid, rid)
);
```

Intermission: Creating a normal table

```
CREATE TABLE reviews        -- Create a table named 'reviews'.
(
  pid INT NOT NULL,         -- Column 'pid' has type INT.
  rid INT NOT NULL,
  quality INT NOT NULL,
  originality INT NOT NULL,
  description CLOB NOT NULL,
                            -- CHARACTER LARGE OBJECT: a lot of text.
  PRIMARY KEY(pid, rid)
);
```

Intermission: Creating a normal table

```
CREATE TABLE reviews        -- Create a table named 'reviews'.
(
  pid INT NOT NULL,         -- Column 'pid' has type INT.
  rid INT NOT NULL,         -- NOT NULL: must have a value.
  quality INT NOT NULL,
  originality INT NOT NULL,
  description CLOB NOT NULL,
                            -- CHARACTER LARGE OBJECT: a lot of text.
  PRIMARY KEY(pid, rid)
);
```

Intermission: Creating a normal table

```
CREATE TABLE reviews        -- Create a table named 'reviews'.
(
  pid INT NOT NULL,         -- Column 'pid' has type INT.
  rid INT NOT NULL,         -- NOT NULL: must have a value.
  quality INT NOT NULL,
  originality INT NOT NULL,
  description CLOB NOT NULL,
                            -- CHARACTER LARGE OBJECT: a lot of text.
  PRIMARY KEY(pid, rid)     -- The primary key consists of two columns.
);
```

Intermission: Creating a normal table

```
CREATE TABLE reviewswn      -- Create a table named 'reviewswn'.
(
  pid INT NOT NULL,         -- Column 'pid' has type INT.
  rid INT NOT NULL,         -- NOT NULL: must have a value.
  quality INT,              -- Might not have values!
  originality INT,          -- Might not have values!
  description CLOB NOT NULL,
                            -- CHARACTER LARGE OBJECT: a lot of text.
  PRIMARY KEY(pid, rid)     -- The primary key consists of two columns.
);
```

Intermission: Creating a table with default values

```
CREATE TABLE users          -- Create a table named 'users'.
(
  uid INT GENERATED ALWAYS AS IDENTITY,


  name VARCHAR(100) NOT NULL,

  level INT NOT NULL DEFAULT 13,

  mail VARCHAR(100)
);
```

Intermission: Creating a table with default values

```
CREATE TABLE users          -- Create a table named 'users'.
(
  uid INT GENERATED ALWAYS AS IDENTITY,
                            -- An identifier is generated automatically.

  name VARCHAR(100) NOT NULL,

  level INT NOT NULL DEFAULT 13,

  mail VARCHAR(100)
);
```

Intermission: Creating a table with default values

```
CREATE TABLE users          -- Create a table named 'users'.
(
  uid INT GENERATED ALWAYS AS IDENTITY,
                            -- An identifier is generated automatically.

  name VARCHAR(100) NOT NULL,
                            -- A small amount of text (up-to-100 characters).
  level INT NOT NULL DEFAULT 13,

  mail VARCHAR(100)
);
```

Intermission: Creating a table with default values

```
CREATE TABLE users          -- Create a table named 'users'.
(
  uid INT GENERATED ALWAYS AS IDENTITY,
                            -- An identifier is generated automatically.

  name VARCHAR(100) NOT NULL,
                            -- A small amount of text (up-to-100 characters).
  level INT NOT NULL DEFAULT 13,
                            -- A default value of 13.
  mail VARCHAR(100)
);
```

Intermission: Creating a table with default values

```
CREATE TABLE users          -- Create a table named 'users'.
(
  uid INT GENERATED ALWAYS AS IDENTITY,
                            -- An identifier is generated automatically.

  name VARCHAR(100) NOT NULL,
                            -- A small amount of text (up-to-100 characters).
  level INT NOT NULL DEFAULT 13,
                            -- A default value of 13.
  mail VARCHAR(100)         -- The default is the value NULL.
);
```

Intermission: Creating a table with default values

```
CREATE TABLE users          -- Create a table named 'users'.
(
  uid INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
                            -- An identifier is generated automatically.
                            -- The primary key consists of a single column.
  name VARCHAR(100) NOT NULL,
                            -- A small amount of text (up-to-100 characters).
  level INT NOT NULL DEFAULT 13,
                            -- A default value of 13.
  mail VARCHAR(100)         -- The default is the value NULL.
);
```

Intermission: NULL values

Meaning of NULL values

Applications give meaning to NULL values, not SQL. E.g.,

- optional value or value not applicable;
- missing value or corrupt value;
- something else entirely.

SQL has specific rules on how it deals with NULL values in all situations.

NULL values complicate *everything*!
Use them sparingly and prefer NOT NULL columns.

Intermission: Syntax for inserting rows of data

INSERT INTO *table name*
(*column names*)$_{optional}$
VALUES (*values for the row; one value per column*)
, (*more rows*) $\ldots_{optional}$;

Intermission: Inserting rows of data

reviews				
pid	rid	quality	originality	description

```
INSERT INTO reviews
VALUES(1, 1, 6, 7, 'Great!');
```

Intermission: Inserting rows of data

reviews				
pid	rid	quality	originality	description

```
INSERT INTO reviews                -- Add a row to the table named 'review'.
VALUES(1, 1, 6, 7, 'Great!');      -- The values for the new row.
```

Intermission: Inserting rows of data

reviews				
pid	rid	quality	originality	description

```
INSERT INTO reviews                  -- Add a row to the table named 'review'.
VALUES(1, 1, 6, 7, 'Great!'),        -- The values for the new row.
      (1, 2, 9, 3, 'Amazing!'),      -- You can also add several rows at once.
      (2, 1, 4, 9, 'Perfect!'),
      (2, 3, 10, 10, 'Superb!');
```

Intermission: Inserting rows of data with default values

users uid	name	level	mail
(generated)	(no default)	(13)	(NULL)

```
INSERT INTO users(name)
VALUES('Bo');
```

Intermission: Inserting rows of data with default values

users uid	name	level	mail
(generated)	(no default)	(13)	(NULL)

```
INSERT INTO users(name)          -- Add a row to the table named 'users'.
VALUES('Bo');                    -- Only provide values for the column 'name'.
```

Intermission: Inserting rows of data with default values

users uid	name	level	mail
(generated)	(no default)	(13)	(NULL)

```
INSERT INTO users(name)          -- Add a row to the table named 'users'.
VALUES('Bo');                    -- Only provide values for the column 'name'.

INSERT INTO users(name, level) VALUES('Celeste', 15);
```

Intermission: Inserting rows of data with default values

users uid	name	level	mail
(generated)	(no default)	(13)	(NULL)

```
INSERT INTO users(name)                    -- Add a row to the table named 'users'.
VALUES('Bo');                              -- Only provide values for the column 'name'.

INSERT INTO users(name, level) VALUES('Celeste', 15);
INSERT INTO users(name, level, mail) VALUES('Alicia', 9, 'a@b.com');
```

Intermission: Inserting rows of data with default values

users uid	name	level	mail
(generated)	(no default)	(13)	(NULL)

```
INSERT INTO users(name)                    -- Add a row to the table named 'users'.
VALUES('Bo');                              -- Only provide values for the column 'name'.

INSERT INTO users(name, level) VALUES('Celeste', 15);
INSERT INTO users(name, level, mail) VALUES('Alicia', 9, 'a@b.com');
INSERT INTO users(level, name) VALUES(20, 'Dafni');
```

Intermission: Inserting rows of data with default values

users uid	name	level	mail
(generated)	(no default)	(13)	(NULL)

```
INSERT INTO users(name)          -- Add a row to the table named 'users'.
VALUES('Bo');                    -- Only provide values for the column 'name'.

INSERT INTO users VALUES(12, 'Eva', 13, 'e@b.com');
```

Intermission: Inserting rows of data with default values

users uid	name	level	mail
(generated)	(no default)	(13)	(NULL)

```
INSERT INTO users(name)                -- Add a row to the table named 'users'.
VALUES('Bo');                          -- Only provide values for the column 'name'.

INSERT INTO users VALUES(12, 'Eva', 13, 'e@b.com');
                                       -- Will not work, 'uid' must be generated!
```

Intermission: Inserting rows of data with default values

users uid	name	level	mail
(generated)	(no default)	(13)	(NULL)

```
INSERT INTO users(name)                    -- Add a row to the table named 'users'.
VALUES('Bo');                              -- Only provide values for the column 'name'.

INSERT INTO users(level, mail) VALUES(17, 'f@b.com');
```

Intermission: Inserting rows of data with default values

users uid	name	level	mail
(generated)	(no default)	(13)	(NULL)

```
INSERT INTO users(name)                    -- Add a row to the table named 'users'.
VALUES('Bo');                              -- Only provide values for the column 'name'.

INSERT INTO users(level, mail) VALUES(17, 'f@b.com');
                                           -- Will not work, 'name' has no default!
```

Intermission: Syntax for deleting rows of data

DELETE FROM *table name*
WHERE *conditions on the rows to delete*$_{optional}$;

Intermission: Deleting rows of data

reviews

pid	rid	quality	originality	description
1	1	6	7	Great!
1	2	9	3	Amazing!
2	1	4	9	Perfect!
2	3	10	10	Superb!

Intermission: Deleting rows of data

reviews

pid	rid	quality	originality	description
1	1	6	7	Great!
1	2	9	3	Amazing!
2	1	4	9	Perfect!
~~2~~	~~3~~	~~10~~	~~10~~	~~Superb!~~

```
DELETE FROM reviews WHERE rid = 3;
```

Intermission: Deleting rows of data

reviews

pid	rid	quality	originality	description
~~1~~	~~1~~	~~6~~	~~7~~	~~Great!~~
1	2	9	3	Amazing!
2	1	4	9	Perfect!
2	3	10	10	Superb!

```
DELETE FROM reviews WHERE pid = rid;
```

Intermission: Deleting rows of data

reviews

pid	rid	quality	originality	description
~~1~~	~~1~~	~~6~~	~~7~~	~~Great!~~
~~1~~	~~2~~	~~9~~	~~3~~	~~Amazing!~~
~~2~~	~~1~~	~~4~~	~~9~~	~~Perfect!~~
~~2~~	~~3~~	~~10~~	~~10~~	~~Superb!~~

```
DELETE FROM reviews;
```

Intermission: Syntax for updating rows of data

UPDATE *table name*
SET *column name* = *new value*
    more columns to update $\ldots_{optional}$
WHERE *conditions on the rows to update*$_{optional}$;

Intermission: Updating rows of data

reviews

pid	rid	quality	originality	description
1	1	6	7	Great!
1	2	9	3	Amazing!
2	1	4	9	Perfect!
2	3	10	10	Superb!

Intermission: Updating rows of data

reviews

pid	rid	quality	originality	description
1	1	6	7	Great!
1	2	9	3	Amazing!
2	1	4	9	Perfect!
2	3	0	10	Superb!

```
UPDATE reviews SET quality = 0 WHERE rid = 3;
```

Intermission: Updating rows of data

reviews

pid	rid	quality	originality	description
1	1	6	7	Great!
1	2	10	3	Amazing!
2	1	4	9	Perfect!
2	3	10	10	Superb!

```
UPDATE reviews SET quality = quality + 1 WHERE rid = 2;
```

Intermission: Updating rows of data

reviews

pid	rid	quality	originality	description
1	1	7	6	Great!
1	2	3	9	Amazing!
2	1	9	4	Perfect!
2	3	10	10	Superb!

```
UPDATE reviews SET quality = originality, originality = quality;
```

This is well-defined in the standard—some DBMSs (MySQL) are broken and do not support this.

A closer look at comparisons in **WHERE**

reviews

pid	rid	quality	originality	description
1	1	6	7	Great!
1	2	9	3	Amazing!
2	1	4	9	Perfect!
2	3	10	10	Superb!

$\longrightarrow$

Query output

pid	rid
1	2
2	3

```
SELECT pid, rid
FROM reviews
WHERE quality > 8;
```

Return review-rows with a high ($\geq$ 8) quality.

A closer look at comparisons in **WHERE**

reviews

pid	rid	quality	originality	description
1	1	6	7	Great!
1	2	9	3	Amazing!
2	1	4	9	Perfect!
2	3	10	10	Superb!

Query output

pid	rid
1	2
2	3

```
SELECT pid, rid
FROM reviews
WHERE quality > 8;
-- Alternatives: quality >= 9, NOT(quality <= 8), and NOT quality < 9.
```

Return review-rows with a high (≥ 8) quality.

A closer look at comparisons in **WHERE**

reviews

pid	rid	quality	originality	description
1	1	6	7	Great!
1	2	9	3	Amazing!
2	1	4	9	Perfect!
2	3	10	10	Superb!

Query output

pid	rid
2	3

```
SELECT pid, rid
FROM reviews
WHERE quality > 8 AND originality > 8;
```

Return review-rows with a high (≥ 8) quality and originality.

A closer look at comparisons in **WHERE**

reviews

pid	rid	quality	originality	description
1	1	6	7	Great!
1	2	9	3	Amazing!
2	1	4	9	Perfect!
2	3	10	10	Superb!

⟶

Query output

pid	rid
1	2
2	1
2	3

```
SELECT pid, rid
FROM reviews
WHERE quality > 8 OR originality > 8;
```

Return review-rows with a high (≥ 8) quality or originality.

A closer look at comparisons in **WHERE**

reviews

pid	rid	quality	originality	description
1	1	6	7	Great!
1	2	9	3	Amazing!
2	1	4	9	Perfect!
2	3	10	10	Superb!

⟶

Query output

pid	rid
1	1

```
SELECT pid, rid
FROM reviews
WHERE NOT(quality > 8 OR originality > 8);
```

Return review-rows without a high (≥ 8) quality or originality.

A closer look at comparisons in **WHERE**

reviews

pid	rid	quality	originality	description
1	1	6	7	Great!
1	2	9	3	Amazing!
2	1	4	9	Perfect!
2	3	10	10	Superb!

$\longrightarrow$

Query output

pid	rid
1	1

```
SELECT pid, rid
FROM reviews
WHERE NOT(quality > 8 OR originality > 8);
-- Alternative: NOT(quality > 8) AND NOT(originality > 8).
```

Return review-rows without a high (≥ 8) quality or originality.

A closer look at comparisons in **WHERE**

reviews

pid	rid	quality	originality	description
1	1	6	7	Great!
1	2	9	3	Amazing!
2	1	4	9	Perfect!
2	3	10	10	Superb!

Query output

pid	rid
1	1
1	2
2	1

```
SELECT pid, rid
FROM reviews
WHERE quality <> 10;
```

Return review-rows without the highest quality.

Queries and NULL values

reviewswn

pid	rid	quality	originality	description
1	1	6	7	Great!
1	2	NULL	NULL	Amazing!
2	1	NULL	9	Perfect!
2	3	10	NULL	Superb!

```
SELECT pid, rid, quality + originality AS x
FROM reviewswn
WHERE quality > 8 OR originality > 8;
```

Question: Which pairs (`pid, rid`) do you expect in the output?

Vote at https://strawpoll.com/pdaw6rza6.
Or: go to https://strawpoll.live and use the code **268762**.

Comparisons with NULL values

Question: What should the result of `NULL = NULL` and of `NULL <> NULL` be?

Vote at https://strawpoll.com/w3hycpxzq.
Or: go to https://strawpoll.live and use the code **733519**.

Comparisons with NULL values

Question: What should the result of `NULL = NULL` and of `NULL <> NULL` be?

Vote at https://strawpoll.com/w3hycpxzq.
Or: go to https://strawpoll.live and use the code **733519**.

- SQL does not know what a `NULL` value means.
- Comparing with a `NULL` value has an *unknown* result.
- SQL does not know whether that *unknown* is `true` or `false`.

All comparisons with `NULL` result in unknown!

Comparisons with NULL values

Question: What should the result of `NULL = NULL` and of `NULL <> NULL` be?

Vote at https://strawpoll.com/w3hycpxzq.
Or: go to https://strawpoll.live and use the code **733519**.

- SQL does not know what a `NULL` value means.
- Comparing with a `NULL` value has an *unknown* result.
- SQL does not know whether that *unknown* is `true` or `false`.

All comparisons with `NULL` result in unknown!
DISTINCT *does* consider different `NULL` values to be equal.

The three-valued (ternary) logic of SQL

A logic with three values: `true`, `false`, and unknown.

The three-valued (ternary) logic of SQL

A logic with three values: `true`, `false`, and unknown.

Rules for **AND**

- A **AND** `true` $\equiv A$.
- A **AND** `false` $\equiv$ `false`.
- unknown **AND** unknown $\equiv$ unknown.

The three-valued (ternary) logic of SQL

A logic with three values: `true`, `false`, and unknown.

Rules for **OR**

- A **OR** `true` $\equiv$ `true`.
- A **OR** `false` $\equiv A$.
- unknown **OR** unknown $\equiv$ unknown.

The three-valued (ternary) logic of SQL

A logic with three values: `true`, `false`, and unknown.

Rules for **NOT**

- **NOT** `true` = `false`.
- **NOT** `false` = `true`.
- **NOT** unknown = unknown.

The three-valued (ternary) logic of SQL

A logic with three values: `true`, `false`, and unknown.

A truth table for the three-valued logic of SQL

A	*B*	*A* **AND** *B*	*A* **OR** *B*	**NOT** *A*
`false`	`false`	`false`	`false`	`true`
`true`	`false`	`false`	`true`	`false`
`false`	`true`	`false`	`true`	`true`
`true`	`true`	`true`	`true`	`false`

The three-valued (ternary) logic of SQL

A logic with three values: `true`, `false`, and unknown.

A truth table for the three-valued logic of SQL

A	*B*	*A* **AND** *B*	*A* **OR** *B*	**NOT** *A*
`false`	`false`	`false`	`false`	`true`
`true`	`false`	`false`	`true`	`false`
`false`	`true`	`false`	`true`	`true`
`true`	`true`	`true`	`true`	`false`
`false`	`unknown`	`false`	`unknown`	`true`
`true`	`unknown`	`unknown`	`true`	`false`

The three-valued (ternary) logic of SQL

A logic with three values: `true`, `false`, and unknown.

A truth table for the three-valued logic of SQL

A	*B*	*A* **AND** *B*	*A* **OR** *B*	**NOT** *A*
`false`	`false`	`false`	`false`	`true`
`true`	`false`	`false`	`true`	`false`
`false`	`true`	`false`	`true`	`true`
`true`	`true`	`true`	`true`	`false`
`false`	`unknown`	`false`	`unknown`	`true`
`true`	`unknown`	`unknown`	`true`	`false`
`unknown`	`false`	`false`	`unknown`	`unknown`
`unknown`	`true`	`unknown`	`true`	`unknown`
`unknown`	`unknown`	`unknown`	`unknown`	`unknown`

Computations with NULL

- Simple computations: if the input has NULL, then the output is NULL.
  E.g., $5 + \texttt{NULL} = \texttt{NULL}$, $\texttt{NULL} * 8 = \texttt{NULL}$, ...
- Aggregate operations (which we cover later): NULL is *mostly* ignored.

Answering queries with NULL values

reviewswn

pid	rid	quality	originality	description
1	1	6	7	Great!
1	2	NULL	NULL	Amazing!
2	1	NULL	9	Perfect!
2	3	10	NULL	Superb!

```
SELECT pid, rid, quality + originality AS x
FROM reviewswn
WHERE quality > 8 OR originality > 8;
```

Answering queries with NULL values

reviewswn

pid	rid	quality	originality	description
1	1	6	7	Great!
1	2	NULL	NULL	Amazing!
2	1	NULL	9	Perfect!
2	3	10	NULL	Superb!

```
SELECT pid, rid, quality + originality AS x
FROM reviewswn
WHERE quality > 8 OR originality > 8;
```

Answering queries with NULL values

reviewswn

pid	rid	quality	originality	description
1	1	6	7	Great!
1	2	NULL	NULL	Amazing!
2	1	NULL	9	Perfect!
2	3	10	NULL	Superb!

```
SELECT pid, rid, quality + originality AS x
FROM reviewswn
WHERE quality > 8 OR originality > 8;
```

Answering queries with NULL values

reviewswn

pid	rid	quality	originality	description
1	1	6	7	Great!
1	2	NULL	NULL	Amazing!
2	1	NULL	9	Perfect!
2	3	10	NULL	Superb!

→

Query output

pid	rid	x
2	1	NULL
2	3	NULL

```
SELECT pid, rid, quality + originality AS x
FROM reviewswn
WHERE quality > 8 OR originality > 8;
```

Working with NULL values

reviewswn

pid	rid	quality	originality	description
1	1	6	7	Great!
1	2	NULL	NULL	Amazing!
2	1	NULL	9	Perfect!
2	3	10	NULL	Superb!

```
SELECT pid, rid
FROM reviewswn
WHERE quality IS NULL AND
      originality IS NOT NULL;
```

Working with NULL values

reviewswn

pid	rid	quality	originality	description
1	1	6	7	Great!
1	2	NULL	NULL	Amazing!
2	1	NULL	9	Perfect!
2	3	10	NULL	Superb!

```
SELECT pid, rid
FROM reviewswn
WHERE quality IS NULL AND  -- Check whether 'quality' is a NULL value.
      originality IS NOT NULL;
```

Working with NULL values

reviewswn

pid	rid	quality	originality	description
1	1	6	7	Great!
1	2	NULL	NULL	Amazing!
2	1	NULL	9	Perfect!
2	3	10	NULL	Superb!

```
SELECT pid, rid
FROM reviewswn
WHERE quality IS NULL AND  -- Check whether 'quality' is a NULL value.
      originality IS NOT NULL;
                           -- Check whether 'originality' is not a NULL value.
```

Working with NULL values

reviewswn

pid	rid	quality	originality	description
1	1	6	7	Great!
1	2	NULL	NULL	Amazing!
2	1	NULL	9	Perfect!
2	3	10	NULL	Superb!

⟶

Query output

pid	rid
2	1

```
SELECT pid, rid
FROM reviewswn
WHERE quality IS NULL AND  -- Check whether 'quality' is a NULL value.
      originality IS NOT NULL;
                           -- Check whether 'originality' is not a NULL value.
```

Set-wise combining **SELECT** queries

students

sid	name	year
1	Alicia	2020
3	Celeste	2018
4	Dafni	2019

faculty

fid	name	rank
2	Bo	Assistant
3	Celeste	Associate
5	Eva	Full

```
SELECT sid AS id FROM students;
```

Return the identifier of all students.

Set-wise combining **SELECT** queries

students

sid	name	year
1	Alicia	2020
3	Celeste	2018
4	Dafni	2019

faculty

fid	name	rank
2	Bo	Assistant
3	Celeste	Associate
5	Eva	Full

```
SELECT sid AS id FROM students
UNION
SELECT fid AS id FROM faculty;
```

Return the identifier of all students and all faculty.

Set-wise combining **SELECT** queries

students

sid	name	year
1	Alicia	2020
3	Celeste	2018
4	Dafni	2019

faculty

fid	name	rank
2	Bo	Assistant
3	Celeste	Associate
5	Eva	Full

⟶

Query output

id
1
2
3
4
5

```
SELECT sid AS id FROM students
UNION                                  -- UNION combines results as a set.
SELECT fid AS id FROM faculty;
```

Return the identifier of all students and all faculty.

Set-wise combining **SELECT** queries

students

sid	name	year
1	Alicia	2020
3	Celeste	2018
4	Dafni	2019

faculty

fid	name	rank
2	Bo	Assistant
3	Celeste	Associate
5	Eva	Full

⟶

Query output

id
1
2
3
3
4
5

```
SELECT sid AS id FROM students
UNION ALL                          -- UNION ALL combines results as a multiset.
SELECT fid AS id FROM faculty;
```

Return the identifier of all students and all faculty.

Set-wise combining **SELECT** queries

students

sid	name	year
1	Alicia	2020
3	Celeste	2018
4	Dafni	2019

faculty

fid	name	rank
2	Bo	Assistant
3	Celeste	Associate
5	Eva	Full

⟶

Query output

id
3

```
SELECT sid AS id FROM students
INTERSECT                          -- INTERSECT combines results as a set.
SELECT fid AS id FROM faculty;
```

Return the identifier of all students that are faculty.

Set-wise combining **SELECT** queries

students

sid	name	year
1	Alicia	2020
3	Celeste	2018
4	Dafni	2019

faculty

fid	name	rank
2	Bo	Assistant
3	Celeste	Associate
5	Eva	Full

⟶

Query output

id
1
4

```
SELECT sid AS id FROM students
EXCEPT                          -- EXCEPT combines results as a set.
SELECT fid AS id FROM faculty;
```

Return the identifier of all students that are not faculty.

The set and multiset semantics of SQL

A
v
1
2
2
3
3
3
4
4
4

B
v
1
2
2
3
3
3
3
4
4

Consider a value W

- Let m be the count of W in ‘A’.
- Let n be the count of W in ‘B’.

The set and multiset semantics of SQL

A
v
1
2
2
3
3
3
4
4
4

UNION

B
v
1
2
2
3
3
3
3
4
4

$\longrightarrow$

Query output
v
1
2
3
4

Consider a value W

- Let m be the count of W in 'A'.
- Let n be the count of W in 'B'.

- **UNION**: W will be once in the output if W is in the input (if $m + n > 0$).

The set and multiset semantics of SQL

A
v
1
2
2
3
3
3
4
4
4

UNION ALL

B
v
1
2
2
3
3
3
3
4
4

⟶

Query output
v
1
1
2
2
2
2
3
3
3
3
3
3
3
…

Consider a value W

- Let m be the count of W in 'A'.
- Let n be the count of W in 'B'.

- **UNION ALL**: W will be once in the output for every copy of W in the input ($m + n$ output copies).

The set and multiset semantics of SQL

A
v
1
2
2
3
3
3
4
4
4

INTERSECT

B
v
1
2
2
3
3
3
3
4
4

$\longrightarrow$

Query output
v
1
2
3
4

Consider a value W

- Let m be the count of W in 'A'.
- Let n be the count of W in 'B'.

- **INTERSECT**: W will be once in the output if it is in both inputs (if $min(m, n) > 0$).

The set and multiset semantics of SQL

A
v
1
2
2
3
3
3
4
4
4

INTERSECT ALL

B
v
1
2
2
3
3
3
3
4
4

$\longrightarrow$

Query output
v
1
2
2
3
3
3
4
4

Consider a value W

- Let m be the count of W in 'A'.
- Let n be the count of W in 'B'.

- **INTERSECT ALL**: W will be once in the output for every copy of W in both inputs ($min(m, n)$ output copies).

The set and multiset semantics of SQL

A
v
1
2
2
3
3
3
4
4
4

EXCEPT

B
v
1
2
2
3
3
3
3
4
4

$\longrightarrow$

Query output
v

Consider a value W

- Let m be the count of W in 'A'.
- Let n be the count of W in 'B'.

- **EXCEPT**: W will be once in the output if it is only in the left input (if $m > 0 = n$).

The set and multiset semantics of SQL

A
v
1
2
2
3
3
3
4
4
4

EXCEPT ALL

B
v
1
2
2
3
3
3
3
4
4

$\longrightarrow$

Query output
v
4

Consider a value W

- Let m be the count of W in 'A'.
- Let n be the count of W in 'B'.

- **EXCEPT ALL**: W will be once in the output for every extra copy in the left input ($m - n$ copies if $m > n$).

The set and multiset semantics of SQL

A
v
1
2
2
3
3
3
4
4
4

B
v
1
2
2
3
3
3
3
4
4

What about `NULL` values?

The set and multiset semantics of SQL

A
v
1
2
2
3
3
3
4
4
4

B
v
1
2
2
3
3
3
3
4
4

What about `NULL` values?

Are you doing the right thing?

The set and multiset semantics of SQL

A
v
1
2
2
3
3
3
4
4
4

B
v
1
2
2
3
3
3
3
4
4

What about `NULL` values?

Are you doing the right thing?

`NULL`s are treated as normal values.

Nested queries: Another option for intersections and differences

students

sid	name	year
1	Alicia	2020
3	Celeste	2018
4	Dafni	2019

faculty

fid	name	rank
2	Bo	Assistant
3	Celeste	Associate
5	Eva	Full

```
SELECT sid
FROM students
WHERE sid IN (                    -- IN checks whether 'sid' is in a list
    SELECT fid FROM faculty);     -- obtained from column 'fid' in table 'faculty'.
```

Nested queries: Another option for intersections and differences

students

sid	name	year
1	Alicia	2020
3	Celeste	2018
4	Dafni	2019

faculty

fid	name	rank
2	Bo	Assistant
3	Celeste	Associate
5	Eva	Full

⟶

Query output

id
3

```
SELECT sid
FROM students
WHERE sid IN (                    -- IN checks whether 'sid' is in a list
    SELECT fid FROM faculty);     -- obtained from column 'fid' in table 'faculty'.
```

Return the identifier of all students that are faculty.

Nested queries: Another option for intersections and differences

students

sid	name	year
1	Alicia	2020
3	Celeste	2018
4	Dafni	2019

faculty

fid	name	rank
2	Bo	Assistant
3	Celeste	Associate
5	Eva	Full

$\longrightarrow$

Query output

id
1
4

```
SELECT sid
FROM students
WHERE sid NOT IN (                  -- NOT IN checks whether 'sid' is not in a list
    SELECT fid FROM faculty);       -- obtained from column 'fid' in table 'faculty'.
```

Return the identifier of all students that are not faculty.

Nested queries: Another option for intersections and differences

students

sid	name	year
1	Alicia	2020
3	Celeste	2018
4	Dafni	2019

faculty

fid	name	rank
2	Bo	Assistant
3	Celeste	Associate
5	Eva	Full

→

Query output

id
1
4

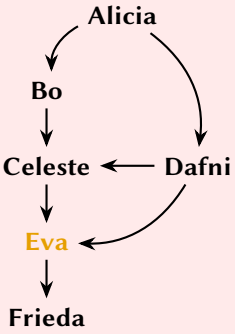
```
SELECT sid
FROM students
WHERE sid NOT IN (                  -- NOT IN checks whether 'sid' is not in a list
    SELECT fid FROM faculty);       -- obtained from column 'fid' in table 'faculty'.
```

Return the identifier of all students that are not faculty.

Nested queries are power tools that, with proper usage, are highly efficient!

Nesting nested queries

friendof	
from	to



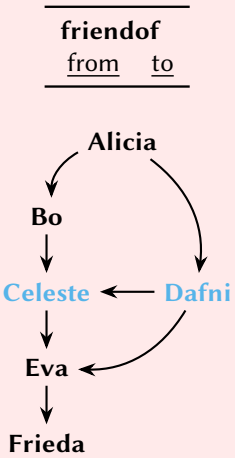

```
SELECT "FROM" FROM friendof
WHERE to = 'Frieda';
```

-- "FROM" is the column name ‘from’ and is not
-- interpreted as the **FROM** keyword.

-- Double-quoted names are *case sensitive*!

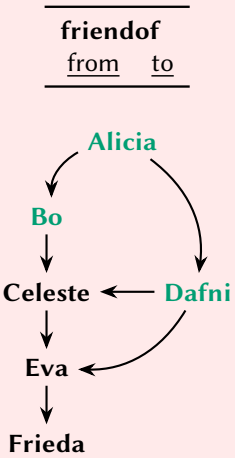
-- DB2 stores names upper-case internally.

Nesting nested queries




```
SELECT "FROM" FROM friendof
WHERE to IN (
  SELECT "FROM" FROM friendof
  WHERE to = 'Frieda');
```

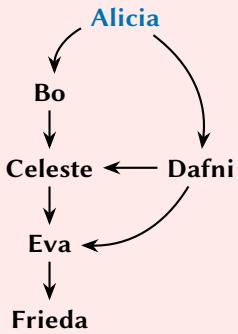
Nesting nested queries




```
SELECT "FROM" FROM friendof
WHERE to IN (
  SELECT "FROM" FROM friendof
  WHERE to IN (
    SELECT "FROM" FROM friendof
    WHERE to = 'Frieda'));
```

Nesting nested queries

friendof	
from	to




```
SELECT "FROM" FROM friendof
WHERE to IN (
  SELECT "FROM" FROM friendof
  WHERE to IN (
    SELECT "FROM" FROM friendof
    WHERE to IN (
      SELECT "FROM" FROM friendof
      WHERE to = 'Frieda')));
```

Nested queries and **ALL**

reviews

pid	rid	quality	originality	description
1	1	6	7	Great!
1	2	9	3	Amazing!
2	1	4	9	Perfect!
2	3	10	10	Superb!

```
SELECT pid, rid
FROM reviews
WHERE originality >= ALL (
    SELECT originality        -- The nested query gives a list L of
    FROM reviews);            -- all 'originality' scores.
```

Nested queries and **ALL**

reviews

pid	rid	quality	originality	description
1	1	6	7	Great!
1	2	9	3	Amazing!
2	1	4	9	Perfect!
2	3	10	10	Superb!

```
SELECT pid, rid
FROM reviews
WHERE originality >= ALL ( -- >= ALL: 'originality' must be max(L).
    SELECT originality     -- The nested query gives a list L of
    FROM reviews);         -- all 'originality' scores.
```

Nested queries and **ALL**

reviews

pid	rid	quality	originality	description
1	1	6	7	Great!
1	2	9	3	Amazing!
2	1	4	9	Perfect!
2	3	10	10	Superb!

⟶

Query output

pid	rid
2	3

```
SELECT pid, rid
FROM reviews
WHERE originality >= ALL ( -- >= ALL: 'originality' must be max(L).
    SELECT originality     -- The nested query gives a list L of
    FROM reviews);         -- all 'originality' scores.
```

Return the primary keys of review(s) with the highest originality score.

Nested queries and **ALL**

reviews

pid	rid	quality	originality	description
1	1	6	7	Great!
1	2	9	3	Amazing!
2	1	4	9	Perfect!
2	3	10	10	Superb!

⟶

Query output

pid	rid
1	2

```
SELECT pid, rid
FROM reviews
WHERE originality < ALL (    -- < ALL: 'originality' must be less than min(L).
    SELECT quality           -- The nested query gives a list L of
    FROM reviews);           -- all 'quality' scores.
```

Return the primary keys of review(s) with an originality score lower than any quality score.

Nested queries and **ALL**

reviews

pid	rid	quality	originality	description
1	1	6	7	Great!
1	2	9	3	Amazing!
2	1	4	9	Perfect!
2	3	10	10	Superb!

→

Query output

pid	rid
1	1
1	2

```
SELECT pid, rid
FROM reviews
WHERE originality <> ALL ( -- <> ALL: 'originality' must be NOT IN L.
    SELECT quality          -- The nested query gives a list L of
    FROM reviews);          -- all 'quality' scores.
```

Return the primary keys of review(s) with an originality score that is not a quality score.

Correlated nested queries

reviews

pid	rid	quality	originality	description
1	1	6	7	Great!
1	2	9	3	Amazing!
2	1	4	9	Perfect!
2	3	10	10	Superb!

```
SELECT pid, rid
FROM reviews
WHERE originality >= ALL (
    SELECT originality
    FROM reviews);
```

Return, for each reviewer, the primary keys of its review(s) with the highest originality score.

Correlated nested queries

reviews

pid	rid	quality	originality	description
1	1	6	7	Great!
1	2	9	3	Amazing!
2	1	4	9	Perfect!
2	3	10	10	Superb!

⟶

Query output

pid	rid
2	3

```
SELECT pid, rid
FROM reviews
WHERE originality >= ALL (
    SELECT originality
    FROM reviews);
```

Return~~, for each reviewer,~~ the primary keys of its review(s) with the highest originality score.

Correlated nested queries

reviews

pid	rid	quality	originality	description
1	1	6	7	Great!
1	2	9	3	Amazing!
2	1	4	9	Perfect!
2	3	10	10	Superb!

⟶

Query output

pid	rid
2	3

```
SELECT pid, rid
FROM reviews AS ra              -- Give the two 'reviews' different names.
WHERE originality >= ALL (
    SELECT originality
    FROM reviews AS rb);        -- Give the two 'reviews' different names.
```

Return~~, for each reviewer,~~ the primary keys of its review(s) with the highest originality score.

Correlated nested queries

reviews

pid	rid	quality	originality	description
1	1	6	7	Great!
1	2	9	3	Amazing!
2	1	4	9	Perfect!
2	3	10	10	Superb!

$\longrightarrow$

Query output

pid	rid
2	3

```
SELECT pid, rid
FROM reviews ra                  -- Give the two 'reviews' different names.
WHERE originality >= ALL ( -- AS is always optional.
    SELECT originality
    FROM reviews rb);            -- Give the two 'reviews' different names.
```

Return~~, for each reviewer,~~ the primary keys of its review(s) with the highest originality score.

Correlated nested queries

reviews

pid	rid	quality	originality	description
1	1	6	7	Great!
1	2	9	3	Amazing!
2	1	4	9	Perfect!
2	3	10	10	Superb!

⟶

Query output

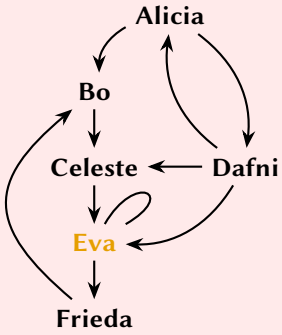
pid	rid
1	2
2	1
2	3

```
SELECT pid, rid
FROM reviews AS ra              -- Give the two 'reviews' different names.
WHERE originality >= ALL ( -- AS is always optional.
    SELECT originality
    FROM reviews AS rb          -- Give the two 'reviews' different names.
    WHERE ra.rid = rb.rid); -- Only 'originality' scores of this reviewer.
```

Return, for each reviewer, the primary keys of its review(s) with the highest originality score.

Nesting correlated nested queries

friendof	
from	to

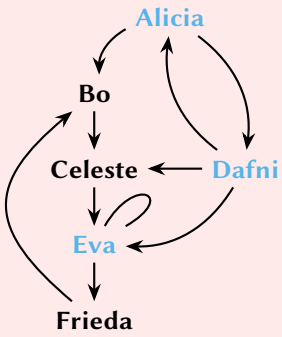



```
SELECT "FROM" FROM friendof F
WHERE to = "FROM";
```

Nesting correlated nested queries

friendof

from	to

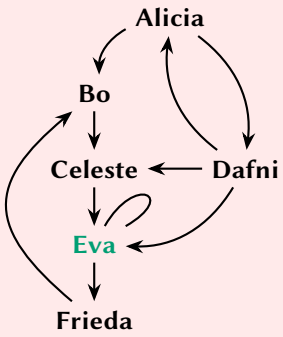



```
SELECT "FROM" FROM friendof F
WHERE to IN (
  SELECT "FROM" FROM friendof
  WHERE to = F."FROM");
```

Nesting correlated nested queries

friendof

from	to

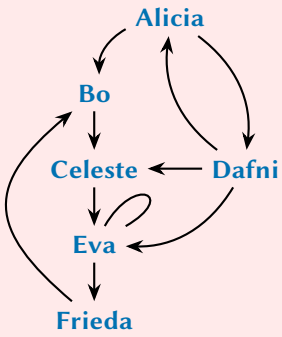



```
SELECT "FROM" FROM friendof F
WHERE to IN (
  SELECT "FROM" FROM friendof
  WHERE to IN (
    SELECT "FROM" FROM friendof
    WHERE to = F."FROM"));
```

Nesting correlated nested queries

friendof

from	to

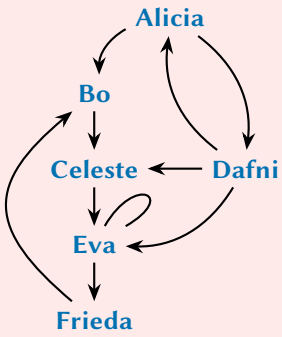



```
SELECT "FROM" FROM friendof F
WHERE to IN (
  SELECT "FROM" FROM friendof
  WHERE to IN (
    SELECT "FROM" FROM friendof
    WHERE to IN (
      SELECT "FROM" FROM friendof
      WHERE to = F."FROM")));
```

Nesting correlated nested queries

friendof

from	to



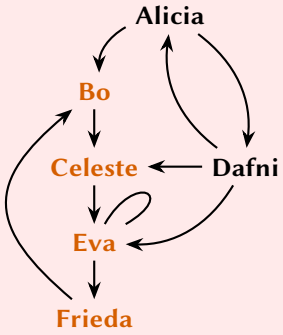

```
SELECT "FROM" FROM friendof F
WHERE to IN (
  SELECT "FROM" FROM friendof
  WHERE to IN (
    SELECT "FROM" FROM friendof
    WHERE to IN (
      SELECT "FROM" FROM friendof
      WHERE to = F."FROM")));
```

How to do a 4-length Hamiltonian cycle?

Nesting correlated nested queries

friendof

from	to

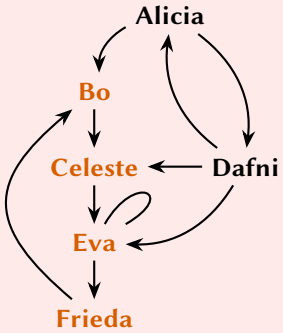



```
SELECT "FROM" FROM friendof F
WHERE to IN (
  SELECT "FROM" FROM friendof F2
  WHERE to IN (
    SELECT "FROM" FROM friendof F3
    WHERE to IN (
      SELECT "FROM" FROM friendof F4
      WHERE to = F."FROM"AND
        F."FROM" <> F2."FROM" AND
        F."FROM" <> F3."FROM" AND
        F."FROM" <> F4."FROM" AND
        F2."FROM" <> F3."FROM" AND
        F2."FROM" <> F4."FROM" AND
        F3."FROM" <> F4."FROM")));
```

Nesting correlated nested queries

friendof

from	to



How to do a 4-length Hamiltonian cycle?

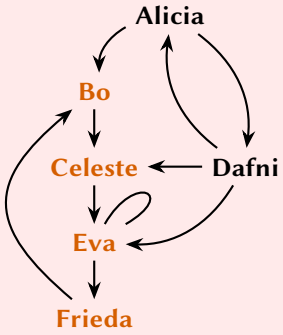
We also still have normal joins!

```
SELECT F."FROM", F2."FROM",
       F3."FROM", F4."FROM"
FROM friendof F, friendof F2,
     friendof F3, friendof F4
WHERE F.to = F2."FROM" AND F2.to = F3."FROM"
  AND F3.to = F4."FROM" AND F4.to = F."FROM"
  AND F."FROM" <> F2."FROM" AND F."FROM" <> F3."FROM"
  AND F."FROM" <> F4."FROM" AND F2."FROM" <> F3."FROM"
  AND F2."FROM" <> F4."FROM" AND F3."FROM" <> F4."FROM";
```

Return the nodes on a 4-length Hamiltonian cycle .

Nesting correlated nested queries

friendof	
from	to



How to do a 4-length Hamiltonian cycle?

We also still have normal joins!

```
SELECT F."FROM", F2."FROM",
       F3."FROM", F4."FROM"
FROM friendof F, friendof F2,
     friendof F3, friendof F4
WHERE F.to = F2."FROM" AND F2.to = F3."FROM"
  AND F3.to = F4."FROM" AND F4.to = F."FROM"
  AND F."FROM" <> F2."FROM" AND F."FROM" <> F3."FROM"
  AND F."FROM" <> F4."FROM" AND F2."FROM" <> F3."FROM"
  AND F2."FROM" <> F4."FROM" AND F3."FROM" <> F4."FROM"
  AND F."FROM" < F2."FROM" AND F."FROM" < F3."FROM"
  AND F."FROM" < F4."FROM";
```

Return the nodes on a 4-length Hamiltonian cycle once.

Nested queries and **ANY**

reviews

pid	rid	quality	originality	description
1	1	6	7	Great!
1	2	9	3	Amazing!
2	1	4	9	Perfect!
2	3	10	10	Superb!

⟶

Query output

pid	rid
2	1
2	3

```
SELECT pid, rid
FROM reviews
WHERE originality = ANY (  -- = ANY: 'originality' must be IN L.
    SELECT quality         -- The nested query gives a list L of
    FROM reviews);         -- all 'quality' scores.
```

Return the review(s) with an originality score equal to a quality score.

Nested queries and **ANY**

reviews

pid	rid	quality	originality	description
1	1	6	7	Great!
1	2	9	3	Amazing!
2	1	4	9	Perfect!
2	3	10	10	Superb!

⟶

Query output

pid	rid
1	1
2	1
2	3

```
SELECT pid, rid
FROM reviews
WHERE originality > ANY (  --> ANY: 'originality' must be more than min(L).
    SELECT quality         -- The nested query gives a list L of
    FROM reviews);         -- all 'quality' scores.
```

Return the review(s) with an originality score larger than the minimum quality score.

Nested queries and **ANY**

reviews

pid	rid	quality	originality	description
1	1	6	7	Great!
1	2	9	3	Amazing!
2	1	4	9	Perfect!
2	3	10	10	Superb!

$\longrightarrow$

Query output

pid	rid
1	1
1	2
2	1
2	3

```
SELECT pid, rid
FROM reviews
WHERE originality <> ANY ( -- <> ANY: a different value must be in L.
    SELECT quality          -- The nested query gives a list L of
    FROM reviews);          -- all 'quality' scores.
```

Return the review(s) with an originality score different from a quality score.

Nested queries and **EXIST**

students

sid	name	year
1	Alicia	2020
3	Celeste	2018
4	Dafni	2019

faculty

fid	name	rank
2	Bo	Assistant
3	Celeste	Associate
5	Eva	Full

```
SELECT * FROM students
WHERE EXISTS (
  SELECT *                    -- The nested query gives a list L with
  FROM faculty);              -- all rows in 'faculty'.
```

Nested queries and **EXIST**

students

sid	name	year
1	Alicia	2020
3	Celeste	2018
4	Dafni	2019

faculty

fid	name	rank
2	Bo	Assistant
3	Celeste	Associate
5	Eva	Full

$\longrightarrow$

Query output

sid	name	year
1	Alicia	2020
3	Celeste	2018
4	Dafni	2019

```
SELECT * FROM students
WHERE EXISTS (                  -- EXISTS: true if L ≠ ∅.
  SELECT *                      -- The nested query gives a list L with
  FROM faculty);                -- all rows in 'faculty'.
```

Return the students if there are faculty members.

Nested queries and **EXIST**

students

sid	name	year
1	Alicia	2020
3	Celeste	2018
4	Dafni	2019

faculty

fid	name	rank
2	Bo	Assistant
3	Celeste	Associate
5	Eva	Full

$\longrightarrow$

Query output

sid	name	year
1	Alicia	2020
4	Dafni	2019

```
SELECT * FROM students
WHERE NOT EXISTS (          -- NOT EXISTS: true if L = ∅.
  SELECT *                  -- The nested query gives a list L with
  FROM faculty              -- all rows in 'faculty' that are the same
  WHERE sid = fid);         -- person as the student.
```

Return those students that are not also faculty members.

BETWEEN: A feature that is easily forgotten

reviews

pid	rid	quality	originality	description
1	1	6	7	Great!
1	2	9	3	Amazing!
2	1	4	9	Perfect!
2	3	10	10	Superb!

```
SELECT pid, rid
FROM review
WHERE originality BETWEEN 7 AND 9;
```

BETWEEN: A feature that is easily forgotten

reviews

pid	rid	quality	originality	description
1	1	6	7	Great!
1	2	9	3	Amazing!
2	1	4	9	Perfect!
2	3	10	10	Superb!

→

Query output

pid	rid
1	1
2	1

```
SELECT pid, rid
FROM review
WHERE originality BETWEEN 7 AND 9;
```

Returns those reviews with an originality score between 7 and 9.

BETWEEN: A feature that is easily forgotten

reviews

pid	rid	quality	originality	description
1	1	6	7	Great!
1	2	9	3	Amazing!
2	1	4	9	Perfect!
2	3	10	10	Superb!

→

Query output

pid	rid
1	1
2	1

```
SELECT pid, rid
FROM review
WHERE originality BETWEEN 7 AND 9;
-- Equivalent to 7 <= originality AND originality <= 9.
```

Returns those reviews with an originality score between 7 and 9.

Selecting data from multiple tables: Revisited

courses

cid	title
1	Programming
2	D. Mathematics
3	Databases

instructors

cid	name
2	Eva
3	Alicia
4	Bo

```
SELECT *
FROM courses C, instructors I
WHERE C.cid = I.cid;
```

Selecting data from multiple tables: Revisited

courses

cid	title
1	Programming
2	D. Mathematics
3	Databases

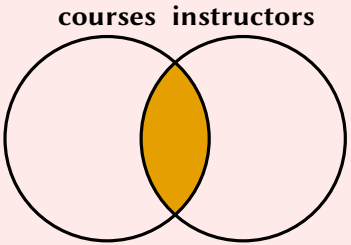
instructors

cid	name
2	Eva
3	Alicia
4	Bo

→

Query output

cid	title	cid	name
2	D. Mathematics	2	Eva
3	Databases	3	Alicia




```
SELECT *
FROM courses C, instructors I
WHERE C.cid = I.cid;
```

-- Output only contains those courses and lectures
-- that are connected to each other.

Selecting data from multiple tables: Revisited

courses

cid	title
1	Programming
2	D. Mathematics
3	Databases

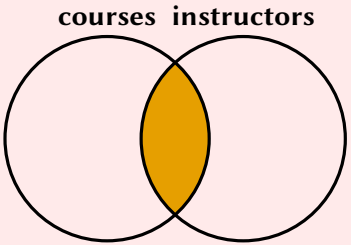
instructors

cid	name
2	Eva
3	Alicia
4	Bo

→

Query output

cid	title	cid	name
2	D. Mathematics	2	Eva
3	Databases	3	Alicia




```
SELECT *
FROM courses C CROSS JOIN
       instructors I
WHERE C.cid = I.cid;

-- Output only contains those courses and lectures
-- that are connected to each other.
```

Selecting data from multiple tables: Revisited

courses

cid	title
1	Programming
2	D. Mathematics
3	Databases

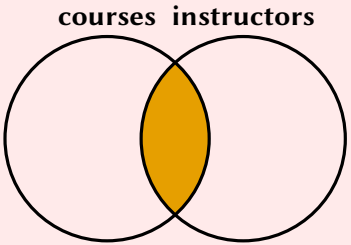
instructors

cid	name
2	Eva
3	Alicia
4	Bo

→

Query output

cid	title	cid	name
2	D. Mathematics	2	Eva
3	Databases	3	Alicia




```
SELECT *
FROM courses C JOIN
     instructors I ON C.cid = I.cid;
```

-- Output only contains those courses and lectures
-- that are connected to each other.

Selecting data from multiple tables: Revisited

courses

cid	title
1	Programming
2	D. Mathematics
3	Databases

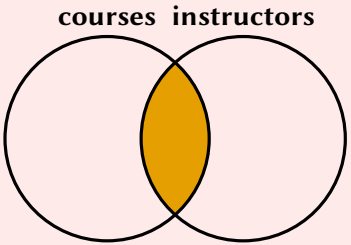
instructors

cid	name
2	Eva
3	Alicia
4	Bo

→

Query output

cid	title	cid	name
2	D. Mathematics	2	Eva
3	Databases	3	Alicia




```
SELECT *
FROM courses C INNER JOIN
        instructors I ON C.cid = I.cid;
```

-- Output only contains those courses and lectures
-- that are connected to each other.

Selecting data from multiple tables: Revisited

courses

cid	title
1	Programming
2	D. Mathematics
3	Databases

instructors

cid	name
2	Eva
3	Alicia
4	Bo

```
SELECT *
FROM courses C LEFT JOIN
        instructors I ON C.cid = I.cid;
```

Selecting data from multiple tables: Revisited

courses

cid	title
1	Programming
2	D. Mathematics
3	Databases

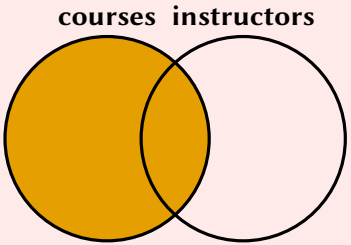
instructors

cid	name
2	Eva
3	Alicia
4	Bo

$\longrightarrow$

Query output

cid	title	cid	name
1	Programming	NULL	NULL
2	D. Mathematics	2	Eva
3	Databases	3	Alicia




```
SELECT *
FROM courses C LEFT JOIN
        instructors I ON C.cid = I.cid;
```

-- Output contains *all* courses.
-- Courses with instructors include instructor details.

Selecting data from multiple tables: Revisited

courses

cid	title
1	Programming
2	D. Mathematics
3	Databases

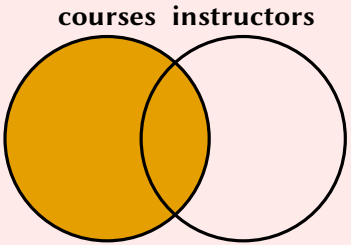
instructors

cid	name
2	Eva
3	Alicia
4	Bo

$\longrightarrow$

Query output

cid	title	cid	name
1	Programming	NULL	NULL
2	D. Mathematics	2	Eva
3	Databases	3	Alicia




```
SELECT *
FROM courses C LEFT OUTER JOIN
        instructors I ON C.cid = I.cid;
```

-- Output contains *all* courses.
-- Courses with instructors include instructor details.

Selecting data from multiple tables: Revisited

courses

cid	title
1	Programming
2	D. Mathematics
3	Databases

instructors

cid	name
2	Eva
3	Alicia
4	Bo

```
SELECT *
FROM courses C RIGHT JOIN
       instructors I ON C.cid = I.cid;
```

-- Output contains *all* instructors.
-- Instructors with courses include course details.

Selecting data from multiple tables: Revisited

courses

cid	title
1	Programming
2	D. Mathematics
3	Databases

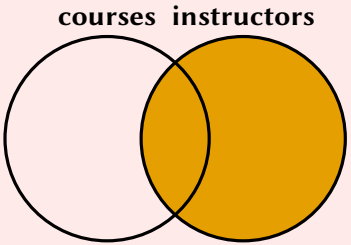
instructors

cid	name
2	Eva
3	Alicia
4	Bo

$\longrightarrow$

Query output

cid	title	cid	name
2	D. Mathematics	2	Eva
3	Databases	3	Alicia
NULL	NULL	4	Bo




```
SELECT *
FROM courses C RIGHT JOIN
       instructors I ON C.cid = I.cid;
```

-- Output contains *all* instructors.
-- Instructors with courses include course details.

Selecting data from multiple tables: Revisited

courses

cid	title
1	Programming
2	D. Mathematics
3	Databases

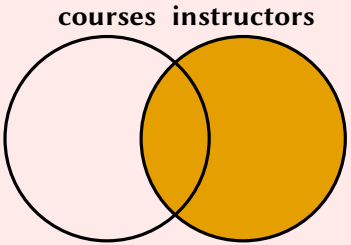
instructors

cid	name
2	Eva
3	Alicia
4	Bo

$\longrightarrow$

Query output

cid	title	cid	name
2	D. Mathematics	2	Eva
3	Databases	3	Alicia
NULL	NULL	4	Bo




```
SELECT *
FROM courses C RIGHT OUTER JOIN
       instructors I ON C.cid = I.cid;
```

-- Output contains *all* instructors.
-- Instructors with courses include course details.

Selecting data from multiple tables: Revisited

courses

cid	title
1	Programming
2	D. Mathematics
3	Databases

instructors

cid	name
2	Eva
3	Alicia
4	Bo

```
SELECT *
FROM courses C FULL JOIN
       instructors I ON C.cid = I.cid;
```

-- Output contains *all* courses and instructors.
-- Courses and instructors are connected if possible.

Selecting data from multiple tables: Revisited

courses

cid	title
1	Programming
2	D. Mathematics
3	Databases

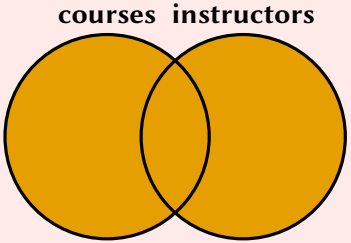
instructors

cid	name
2	Eva
3	Alicia
4	Bo

⟶

Query output

cid	title	cid	name
1	Programming	NULL	NULL
2	D. Mathematics	2	Eva
3	Databases	3	Alicia
NULL	NULL	4	Bo




```
SELECT *
FROM courses C FULL JOIN
     instructors I ON C.cid = I.cid;
```

-- Output contains *all* courses and instructors.
-- Courses and instructors are connected if possible.

Selecting data from multiple tables: Revisited

courses

cid	title
1	Programming
2	D. Mathematics
3	Databases

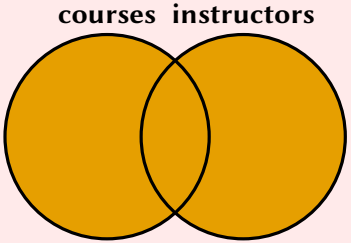
instructors

cid	name
2	Eva
3	Alicia
4	Bo

⟶

Query output

cid	title	cid	name
1	Programming	NULL	NULL
2	D. Mathematics	2	Eva
3	Databases	3	Alicia
NULL	NULL	4	Bo




```
SELECT *
FROM courses C FULL OUTER JOIN
     instructors I ON C.cid = I.cid;
```

-- Output contains *all* courses and instructors.
-- Courses and instructors are connected if possible.

Selecting data from multiple tables: Revisited

courses

cid	title
1	Programming
2	D. Mathematics
3	Databases

instructors

cid	name
2	Eva
3	Alicia
4	Bo

Reminder: Queries don't have to make sense!

```
SELECT *
FROM courses C FULL JOIN
     instructors I ON C.cid > I.cid;
```

Selecting data from multiple tables: Revisited

courses

cid	title
1	Programming
2	D. Mathematics
3	Databases

instructors

cid	name
2	Eva
3	Alicia
4	Bo

$\longrightarrow$

Query output

cid	title	cid	name
1	Programming	NULL	NULL
2	D. Mathematics	NULL	NULL
3	Databases	2	Eva
NULL	NULL	3	Alicia
NULL	NULL	4	Bo

Reminder: Queries don't have to make sense!

```
SELECT *
FROM courses C FULL JOIN
     instructors I ON C.cid > I.cid;
```

Aggregation with nested queries

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	9
Eva	shoe	8
Bo	phone	3
Bo	shoe	5
Celeste	cheese	7

Consider the following two queries

- Per product: get the review with highest rating.
- Per user: get the product they rated lowest.

Aggregation with nested queries

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	9
Eva	shoe	8
Bo	phone	3
Bo	shoe	5
Celeste	cheese	7

Consider the following two queries

- Per product: get the review with highest rating.
- Per user: get the product they rated lowest.

Using nested queries

```
SELECT *
FROM productreview P
WHERE rating >= ALL (
  SELECT rating
  FROM productreview Q
  WHERE P.product = Q.product);
```

Aggregation with nested queries

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	9
Eva	shoe	8
Bo	phone	3
Bo	shoe	5
Celeste	cheese	7

Consider the following two queries

- Per product: get the review with highest rating.
- Per user: get the product they rated lowest.

Using nested queries

```
SELECT *
FROM productreview P
WHERE rating <= ALL (
  SELECT rating
  FROM productreview Q
  WHERE P.user = Q.user);
```

Aggregation with nested queries

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	9
Eva	shoe	8
Bo	phone	3
Bo	shoe	5
Celeste	cheese	7

Consider the following two queries

- Per product: get the review with highest rating.
- Per user: get the product they rated lowest.

Using nested queries

Works—but limited and finicky.

The basics of aggregation

```
SELECT aggregate
FROM productreview;
```

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	9
Eva	shoe	8
Bo	phone	3
Bo	shoe	5
Celeste	cheese	7

The basics of aggregation

```
SELECT MIN(rating)
FROM productreview;
```

-- Minimum value in column ‘rating’.

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	9
Eva	shoe	8
Bo	phone	3
Bo	shoe	5
Celeste	cheese	7

Minimum of ratings: 3.

The basics of aggregation

```
SELECT MIN(rating)
FROM productreview;
```

-- Minimum value in column ‘rating’.
-- Aggregates operate on *groups* of rows.

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	9
Eva	shoe	8
Bo	phone	3
Bo	shoe	5
Celeste	cheese	7

Minimum of ratings: 3.

The basics of aggregation

```
SELECT MIN(rating), product
FROM productreview;
```

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	9
Eva	shoe	8
Bo	phone	3
Bo	shoe	5
Celeste	cheese	7

-- Minimum value in column ‘rating’.
-- Aggregates operate on *groups* of rows.
-- No longer access to individual unaggregated columns!

Invalid query.

The basics of aggregation

```
SELECT MAX(rating)
FROM productreview;
```

```
-- Maximum value in column 'rating'.
```

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	9
Eva	shoe	8
Bo	phone	3
Bo	shoe	5
Celeste	cheese	7

Maximum of ratings: 10.

The basics of aggregation

```
SELECT COUNT(rating)
FROM productreview;
```

-- Number of rows.

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	9
Eva	shoe	8
Bo	phone	3
Bo	shoe	5
Celeste	cheese	7

Ratings: 7.

The basics of aggregation

```
SELECT COUNT(*)
FROM productreview;
```

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	9
Eva	shoe	8
Bo	phone	3
Bo	shoe	5
Celeste	cheese	7

Rows: 7.

```
-- Number of rows.
-- For COUNTing rows: the columns don't matter.
```

The basics of aggregation

```
SELECT COUNT(DISTINCT rating)
FROM productreview;

-- Number of distinct values in column 'rating'.
```

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	9
Eva	shoe	8
Bo	phone	3
Bo	shoe	5
Celeste	cheese	7

Distinct ratings: 6.

The basics of aggregation

```
SELECT SUM(rating)
FROM productreview;
```

-- Sum of the values in column ‘rating’.

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	9
Eva	shoe	8
Bo	phone	3
Bo	shoe	5
Celeste	cheese	7

Sum of ratings: 47.

The basics of aggregation

```
SELECT SUM(DISTINCT rating)
FROM productreview;
```

```
-- Sum of the distinct values in column 'rating'.
```

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	9
Eva	shoe	8
Bo	phone	3
Bo	shoe	5
Celeste	cheese	7

Sum of distinct ratings: 42.

The basics of aggregation

```
SELECT AVG(rating)
FROM productreview;
```

-- Average of the values in column 'rating'.

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	9
Eva	shoe	8
Bo	phone	3
Bo	shoe	5
Celeste	cheese	7

Average of ratings: 6.

The basics of aggregation

```
SELECT AVG(rating)
FROM productreview;
```

```
-- Average of the values in column 'rating'.
-- The average can be an INT!
```

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	9
Eva	shoe	8
Bo	phone	3
Bo	shoe	5
Celeste	cheese	7

Average of ratings: 6 (**INT**).

The basics of aggregation

```
SELECT AVG(CAST(rating AS DOUBLE))
FROM productreview;
```

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	9
Eva	shoe	8
Bo	phone	3
Bo	shoe	5
Celeste	cheese	7

-- Average of the values in column ‘rating’.
-- The average can be an **INT**!
-- Solution: make column ‘rating’ a **DOUBLE**.

Average of ratings: 6.7 (**DOUBLE**).

The basics of aggregation

```
SELECT AVG(CAST(rating AS DECIMAL))
FROM productreview;
```

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	9
Eva	shoe	8
Bo	phone	3
Bo	shoe	5
Celeste	cheese	7

-- Average of the values in column ‘rating’.
-- The average can be an **INT**!
-- Floating point numbers (**DOUBLE**) cause issues.
-- Solution: make column ‘rating’ a **DECIMAL**.

Average of ratings: 6.7 (**DECIMAL**).

The basics of aggregation

```
SELECT AVG(DISTINCT rating)
FROM productreview;
```

```
-- Average of the distinct values in column 'rating'.
```

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	9
Eva	shoe	8
Bo	phone	3
Bo	shoe	5
Celeste	cheese	7

Average of distinct ratings: 7.

The basics of aggregation

```
SELECT AVG(DISTINCT CAST(rating AS DECIMAL))
FROM productreview;
```

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	9
Eva	shoe	8
Bo	phone	3
Bo	shoe	5
Celeste	cheese	7

-- Average of the distinct values in column ‘rating’.

Average of distinct ratings: 7.0 (**DECIMAL**).

Using basic aggregations in queries

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	9
Eva	shoe	8
Bo	phone	3
Bo	shoe	5
Celeste	cheese	7

Consider again the following query

Per product: get the review with highest rating.

Using nested queries

```
SELECT *
FROM productreview P
WHERE rating >= ALL (
  SELECT rating
  FROM productreview Q
  WHERE P.product = Q.product);
```

Using basic aggregations in queries

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	9
Eva	shoe	8
Bo	phone	3
Bo	shoe	5
Celeste	cheese	7

Consider again the following query

Per product: get the review with highest rating.

Using aggregation and nested queries

```
SELECT *
FROM productreview P
WHERE rating = (
  SELECT MAX(rating)
  FROM productreview Q
  WHERE P.product = Q.product);
```

Using basic aggregations in queries

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	9
Eva	shoe	8
Bo	phone	3
Bo	shoe	5
Celeste	cheese	7

Consider again the following query

Per product: get the review with highest rating.

Using aggregation and nested queries

```
SELECT *
FROM productreview P
WHERE rating = (
  SELECT MAX(rating)
  FROM productreview Q
  WHERE P.product = Q.product);
```

Still a bit underwhelming!

GROUP BY: Putting aggregation to good use

Reminder

Aggregates operate on *groups* of rows.

You can no longer access individual unaggregated columns!

GROUP BY: Putting aggregation to good use

Reminder

Aggregates operate on *groups* of rows ⟵ *by default: entire table is one group*.
You can no longer access individual unaggregated columns!

GROUP BY: Putting aggregation to good use

Reminder

Aggregates operate on *groups* of rows ⟵ *by default: entire table is one group*.
You can no longer access individual unaggregated columns!

We can specify our own row grouping

SELECT *aggregated output columns*
FROM *sources*
WHERE *row conditions*
GROUP BY *columns to group rows on*;

Using **GROUP BY** in queries

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	9
Eva	shoe	8
Bo	phone	3
Bo	shoe	5
Celeste	cheese	7

Consider the following query

Get the highest and lowest rating per product.

```
SELECT *
FROM productreview;
```

Using **GROUP BY** in queries

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	9
Eva	shoe	8
Bo	phone	3
Bo	shoe	5
Celeste	cheese	7

Consider the following query

Get the highest and lowest rating per product.

```
SELECT *
FROM productreview
GROUP BY product;

-- We want aggregated information per product.
-- Hence, GROUP BY product!
```

Using **GROUP BY** in queries

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	9
Eva	shoe	8
Bo	phone	3
Bo	shoe	5
Celeste	cheese	7

Consider the following query

Get the highest and lowest rating per product.

```
SELECT product, MAX(rating), MIN(rating)
FROM productreview
GROUP BY product;

-- We want aggregated information per product.
-- Hence, GROUP BY product!

-- The column 'product' can be used:
-- every row in a single aggregated group is the same product.
```

Using **GROUP BY** in queries

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	9
Eva	shoe	8
Bo	phone	3
Bo	shoe	5
Celeste	cheese	7

Consider the following query

Get the highest and lowest rating per product.

```
SELECT product, MAX(rating), MIN(rating)
FROM productreview
GROUP BY product;
```

Query output

product	*C1*	*C2*
cheese	10	7
phone	5	3
shoe	8	5

Using **GROUP BY** in queries

productreview

user	product	rating
Alicia	cheese	10
~~Alicia~~	~~phone~~	~~5~~
Eva	cheese	9
Eva	shoe	8
Bo	phone	3
~~Bo~~	~~shoe~~	~~5~~
Celeste	cheese	7

Consider the following query

Get the highest and lowest rating per product.

```
SELECT product, MAX(rating), MIN(rating)
FROM productreview
WHERE rating <> 5  -- Filters rows before grouping!
GROUP BY product;
```

Query output

product	*C1*	*C2*
cheese	10	7
phone	*3*	3
shoe	8	*8*

HAVING: Filtering on groups

Structure of aggregated queries

```
SELECT aggregated output columns
FROM sources
WHERE row conditions                    -- Filters rows before grouping!
GROUP BY columns to group rows on;
```

HAVING: Filtering on groups

Structure of aggregated queries

SELECT *aggregated output columns*
FROM *sources*
WHERE *row conditions* -- Filters rows *before* grouping!
GROUP BY *columns to group rows on*
HAVING *conditions on aggregated groups*;

HAVING: Filtering on groups

Structure of aggregated queries

```
SELECT aggregated output columns
FROM sources
WHERE row conditions                          -- Filters rows before grouping!
GROUP BY columns to group rows on
HAVING conditions on aggregated groups;       -- Filters aggregated rows
                                              -- (after grouping)!
```

Using **HAVING** in queries

```
SELECT product, MAX(rating), MIN(rating)
FROM productreview
GROUP BY product;
```

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	9
Eva	shoe	8
Bo	phone	3
Bo	shoe	5
Celeste	cheese	7

Query output

product	*C1*	*C2*
cheese	10	7
phone	5	3
shoe	8	5

Using **HAVING** in queries

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	9
Eva	shoe	8
Bo	phone	3
Bo	shoe	5
Celeste	cheese	7

```
SELECT product, MAX(rating), MIN(rating)
FROM productreview
GROUP BY product
HAVING COUNT(*) > 2;
```

Using **HAVING** in queries

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	9
Eva	shoe	8
Bo	phone	3
Bo	shoe	5
Celeste	cheese	7

```
SELECT product, MAX(rating), MIN(rating)
FROM productreview
GROUP BY product
HAVING COUNT(*) > 2;
```

-- Only look at products with *many* reviews!

Query output

product	*C1*	*C2*
cheese	10	7

Using **HAVING** in queries

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	9
Eva	shoe	8
Bo	phone	3
Bo	shoe	5
Celeste	cheese	7

```
SELECT product, MAX(rating), MIN(rating)
FROM productreview
GROUP BY product
HAVING AVG(rating) >= 5;
```

-- Only look at products with *high* average ratings!

Query output

product	*C1*	*C2*
cheese	10	7
shoe	8	5

Using **HAVING** in queries

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	9
Eva	shoe	8
Bo	phone	3
Bo	shoe	5
Celeste	cheese	7

```
SELECT product, MAX(rating), MIN(rating)
FROM productreview
WHERE rating > 4
GROUP BY product
HAVING AVG(rating) >= 5;

-- First ignore the bad ratings (WHERE).
-- Then look at products with high ratings (HAVING).
```

Query output

product	*C1*	*C2*
cheese	10	7
phone	5	5
shoe	8	5

The basics of aggregation with NULL

What about NULL values?

Are you doing the right thing?

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	NULL
Eva	shoe	8
Bo	phone	NULL
Bo	shoe	5
Celeste	cheese	7

The basics of aggregation with NULL

What about NULL values?

```
SELECT MIN(rating)
FROM productreview;
```

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	NULL
Eva	shoe	8
Bo	phone	NULL
Bo	shoe	5
Celeste	cheese	7

-- General rule: NULL values are discarded.

Minimum of ratings: 5.

The basics of aggregation with NULL

What about NULL values?

```
SELECT MAX(rating)
FROM productreview;
```

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	NULL
Eva	shoe	8
Bo	phone	NULL
Bo	shoe	5
Celeste	cheese	7

-- General rule: NULL values are discarded.

Maximum of ratings: 10.

The basics of aggregation with NULL

What about NULL values?

```
SELECT COUNT(rating)
FROM productreview;
```

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	NULL
Eva	shoe	8
Bo	phone	NULL
Bo	shoe	5
Celeste	cheese	7

Ratings: 5.

-- General rule: NULL values are discarded.

The basics of aggregation with NULL

What about NULL values?

```
SELECT COUNT(*)
FROM productreview;
```

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	NULL
Eva	shoe	8
Bo	phone	NULL
Bo	shoe	5
Celeste	cheese	7

Rows: 7.

-- General rule: NULL values are discarded.

The basics of aggregation with NULL

What about NULL values?

```
SELECT COUNT(DISTINCT rating)
FROM productreview;
```

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	NULL
Eva	shoe	8
Bo	phone	NULL
Bo	shoe	5
Celeste	cheese	7

-- General rule: NULL values are discarded.

Distinct ratings: 4.

The basics of aggregation with NULL

What about NULL values?

```
SELECT SUM(rating)
FROM productreview;
```

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	NULL
Eva	shoe	8
Bo	phone	NULL
Bo	shoe	5
Celeste	cheese	7

-- General rule: NULL values are discarded.

Sum of ratings: 35.

The basics of aggregation with NULL

What about NULL values?

```
SELECT SUM(DISTINCT rating)
FROM productreview;
```

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	NULL
Eva	shoe	8
Bo	phone	NULL
Bo	shoe	5
Celeste	cheese	7

-- General rule: NULL values are discarded.

Sum of distinct ratings: 30.

The basics of aggregation with NULL

What about NULL values?

```
SELECT AVG(rating)
FROM productreview;
```

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	NULL
Eva	shoe	8
Bo	phone	NULL
Bo	shoe	5
Celeste	cheese	7

-- General rule: NULL values are discarded.

Average of ratings: 7.

The basics of aggregation with NULL

What about NULL values?

```
SELECT AVG(DISTINCT rating)
FROM productreview;
```

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	NULL
Eva	shoe	8
Bo	phone	NULL
Bo	shoe	5
Celeste	cheese	7

-- General rule: NULL values are discarded.

Average of distinct ratings: 7.

The basics of aggregation with NULL

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	NULL
Eva	shoe	8
Bo	phone	NULL
Bo	shoe	5
Celeste	cheese	7

Minimum of ratings: NULL.

What about NULL values?

```
SELECT MIN(rating)
FROM productreview
WHERE rating IS NULL;

-- General rule: NULL values are discarded.
-- Unless all values are NULL!
```

The basics of aggregation with NULL

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	NULL
Eva	shoe	8
Bo	phone	NULL
Bo	shoe	5
Celeste	cheese	7

Maximum of ratings: NULL.

What about NULL values?

```
SELECT MAX(rating)
FROM productreview
WHERE rating IS NULL;

-- General rule: NULL values are discarded.
-- Unless all values are NULL!
```

The basics of aggregation with NULL

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	NULL
Eva	shoe	8
Bo	phone	NULL
Bo	shoe	5
Celeste	cheese	7

Ratings: 0.

What about NULL values?

```
SELECT COUNT(rating)
FROM productreview
WHERE rating IS NULL;
```

-- General rule: NULL values are discarded.

-- Unless all values are NULL!

The basics of aggregation with NULL

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	NULL
Eva	shoe	8
Bo	phone	NULL
Bo	shoe	5
Celeste	cheese	7

Rows: 2.

What about NULL values?

```
SELECT COUNT(*)
FROM productreview
WHERE rating IS NULL;

-- General rule: NULL values are discarded.
-- Unless all values are NULL!
```

The basics of aggregation with NULL

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	NULL
Eva	shoe	8
Bo	phone	NULL
Bo	shoe	5
Celeste	cheese	7

Distinct ratings: 0.

What about NULL values?

```
SELECT COUNT(DISTINCT rating)
FROM productreview
WHERE rating IS NULL;

-- General rule: NULL values are discarded.
-- Unless all values are NULL!
```

The basics of aggregation with NULL

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	NULL
Eva	shoe	8
Bo	phone	NULL
Bo	shoe	5
Celeste	cheese	7

Sum of ratings: NULL.

What about NULL values?

```
SELECT SUM(rating)
FROM productreview
WHERE rating IS NULL;

-- General rule: NULL values are discarded.
-- Unless all values are NULL!
```

The basics of aggregation with NULL

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	NULL
Eva	shoe	8
Bo	phone	NULL
Bo	shoe	5
Celeste	cheese	7

Sum of distinct ratings: NULL.

What about NULL values?

```
SELECT SUM(DISTINCT rating)
FROM productreview
WHERE rating IS NULL;

-- General rule: NULL values are discarded.
-- Unless all values are NULL!
```

The basics of aggregation with NULL

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	NULL
Eva	shoe	8
Bo	phone	NULL
Bo	shoe	5
Celeste	cheese	7

Average of ratings: NULL.

What about NULL values?

```
SELECT AVG(rating)
FROM productreview
WHERE rating IS NULL;

-- General rule: NULL values are discarded.
-- Unless all values are NULL!
```

The basics of aggregation with NULL

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	NULL
Eva	shoe	8
Bo	phone	NULL
Bo	shoe	5
Celeste	cheese	7

What about NULL values?

```
SELECT AVG(DISTINCT rating)
FROM productreview
WHERE rating IS NULL;
```

-- General rule: NULL values are discarded.
-- Unless all values are NULL!

Average of distinct ratings: NULL.

The **FROM** clause

The basic **SELECT-FROM-WHERE** structure

SELECT *output columns*
FROM *(joined) source tables*
WHERE *row conditions*;

The **FROM** clause

The basic **SELECT-FROM-WHERE** structure

SELECT *output columns*
FROM *(joined) source tables* -- Query results can be used as tables!
WHERE *row conditions*;

The **FROM** clause

The basic **SELECT-FROM-WHERE** structure

SELECT *output columns*
FROM *(joined) source tables*
WHERE *row conditions*;

Example

```
SELECT *
FROM (SELECT * FROM courses
      WHERE cid IN (SELECT cid FROM instructors))
WHERE title LIKE '%Databases%';
```

The **FROM** clause

The basic **SELECT-FROM-WHERE** structure

SELECT *output columns*
FROM *(joined) source tables*
WHERE *row conditions*;

Example: Very useful for complex queries with aggregation!

```
SELECT *
FROM (SELECT product, COUNT(*) AS c, SUM(rating) AS s
      FROM productreview
      GROUP BY product) P, sellers S
WHERE P.product = S.product;
```

The **FROM** clause

The basic **SELECT-FROM-WHERE** structure

SELECT *output columns*
FROM *(joined) source tables*
WHERE *row conditions*;

Example: Very useful for complex queries with aggregation!

```
SELECT *
FROM (SELECT product, COUNT(*) AS c, SUM(rating) AS s
      FROM productreview
      GROUP BY product) P, sellers S
WHERE P.product = S.product;
```

-- Aggregation *before* the join, otherwise the computed aggregates don't make any sense!

Dealing with complex queries

Question: How do I write *complex* queries?

SQL might look weird, but is still a *programming language*!

Dealing with complex queries

Question: How do I write *complex* queries?

SQL might look weird, but is still a *programming language*!

You write complex programs by *first* breaking them up into subproblems and *then* solving these subproblems independently (e.g., functions).

Dealing with complex queries

Question: How do I write *complex* queries?

SQL might look weird, but is still a *programming language*!

You write complex programs by *first* breaking them up into subproblems and *then* solving these subproblems independently (e.g., functions).

WITH *name* (*column names*)$_{optional}$ **AS** (
  `query`
)
SELECT * **FROM** *name*;

Dealing with complex queries

Question: How do I write *complex* queries?

SQL might look weird, but is still a *programming language*!

You write complex programs by *first* breaking them up into subproblems and *then* solving these subproblems independently (e.g., functions).

WITH *name*$_1$ (*column names*)$_{optional}$ **AS** (
  query
), *name*$_2$ (*column names*)$_{optional}$ **AS** (
  query
)
SELECT * **FROM** *name*$_1$, *name*$_2$;

Dealing with complex queries

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	9
Eva	shoe	8
Bo	phone	3
Bo	shoe	5
Celeste	cheese	7

```
WITH sumresult AS (
  SELECT product, SUM(rating) AS srating
  FROM productreview GROUP BY product)
SELECT * FROM sumresult
WHERE srating >= ALL (SELECT srating FROM sumresult);
```

Dealing with complex queries

productreview

user	product	rating
Alicia	cheese	10
Alicia	phone	5
Eva	cheese	9
Eva	shoe	8
Bo	phone	3
Bo	shoe	5
Celeste	cheese	7

⟶

Query output

product	srating
cheese	26

```
WITH sumresult AS (
  SELECT product, SUM(rating) AS srating
  FROM productreview GROUP BY product)
SELECT * FROM sumresult
WHERE srating >= ALL (SELECT srating FROM sumresult);
```

Addendum: Retrieving specific rows based on text patterns

Standard: **SIMILAR TO**

```
SELECT *
FROM courses
WHERE title LIKE '%Databases%';
```

Addendum: Retrieving specific rows based on text patterns

Standard: **SIMILAR TO**

```
SELECT *
FROM courses
WHERE title SIMILAR TO '%(data|algo)%';
```

Addendum: Retrieving specific rows based on text patterns

Standard: **SIMILAR TO**

```
SELECT *
FROM courses
WHERE title SIMILAR TO '%(data|algo)%';
```

- Standards-compliant support seems to be mostly absent, except for PostgreSQL.
- Other vendors have vendor-specific options.
  (e.g., extensions to **LIKE**, **REGEXP_LIKE**, **REGEXP**, **RLIKE**).

Addendum: Retrieving specific rows based on text patterns

Standard: **SIMILAR TO**

```
SELECT *
FROM courses
WHERE title SIMILAR TO '%(data|algo)%';
```

- In general: relational databases are not optimized for *text search*:
  searching patterns in unstructured text is the opposite of structured (tabular) data!
- Searching a few small fields: probably fine (options for index support).
- Anything beyond: dedicated full-text search and indexing engines are a safer bet.

- Regular expressions are not a full replacement of proper input checking!
  Expressions for names, e-mail, phone numbers, … are often oversimplified or wrong.