

Databases

Lecture 2: SQL - Structured Query Language

Emily Lucia Antosch

HAW Hamburg

25.02.2025

1. Introduction

1.1 Where are we right now?

- Last time, we looked at the bare basics of databases and why we should use databases at all.
- Today, we'll be discussing
 - working with the main language of databases: SQL,
 - how to create a small relational database in PostgreSQL,
 - simple design pattern that we will build upon in the future!

1.1 Where are we right now?

1. Introduction
2. Basics
3. **SQL**
4. Entity-Relationship-Model
5. Relationships
6. Constraints
7. More SQL
8. Subqueries & Views
9. Transactions
10. Database Applications
11. Integrity, Trigger & Security

1.2 What is the goal of this chapter?

- At the end of this lesson, you should be able to
 - create a small database example using an installation of PostgreSQL,
 - create, update, remove and delete elements from your database (CRUD)
 - use simple design patterns to design a good database.

2. SQL: Structured Query Language

2.1 What is SQL?

- Standard language for managing relational databases
- Used for querying, updating, and managing data

2. SQL: Structured Query Language

2.1 What is SQL?

- SQL comes in different flavours, depending on the DBMS you use it in.
 - You'll find that some small things work differently in PostgreSQL, MySQL and SQLite.
 - However once you know one flavour, you can easily navigate writing code in another.

2.2 How can we use SQL right now?

- Depending on your installation of SQL and your OS, you have different ways to use SQL.
- In our case, we have installed PostgreSQL, which leaves us with multiple options:
 - Interacting with your database can be done using the CLI (command-line interface).
 - You can use Postgres' own database manager called pgAdmin.
 - In your editor of choice, you can most likely install a database interface to connect to your database (VSCode, JetBrains, NeoVim).

? Question

- Since passing this lecture requires you to install a DBMS on your system, I would like to ask you:
 - What OS are you using?
 - What Editor are you using?
 - Do you think you need help with installing a DBMS?



Idea

- CREATE: Create a database element, like a table or view
- SELECT: Retrieve data from an element
- INSERT: Add new records or rows into a table
- UPDATE: Modify existing records or rows in a table
- DELETE: Remove records or rows from a table

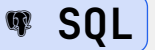
3. First Steps

3.1 SQL: Step-by-Step Walkthrough

CREATE-Statement

- To begin working with SQL, make sure to connect to your database in order to run your commands.
- Next, let's create a table, the most basic building block of our database:

```
1 CREATE TABLE EMP
2 (
3   _ID INTEGER PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
4   NAME TEXT NOT NULL,
5   DEPARTMENT TEXT,
6   SALARY INT DEFAULT 0
7 );
```

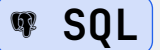


3.1 SQL: Step-by-Step Walkthrough

CREATE-Statement

Let's look at the `_ID` column a little more in detail:

```
1 _ID INTEGER PRIMARY KEY GENERATED ALWAYS AS IDENTITY
```



- `_ID`: Name of the column

? Question

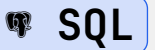
If you're wondering why I named the identifier of the table `_ID` and not `ID`: I do this in order to denote that the identifier is an internal value to the database and will not be shown to the user!

3.1 SQL: Step-by-Step Walkthrough

CREATE-Statement

Let's look at the `_ID` column a little more in detail:

```
1  _ID INTEGER PRIMARY KEY GENERATED ALWAYS AS IDENTITY
```



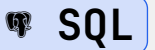
- **INTEGER:** The data type of the column. We'll explore data types a more in-depth very shortly

3.1 SQL: Step-by-Step Walkthrough

CREATE-Statement

Let's look at the `_ID` column a little more in detail:

```
1  _ID INTEGER PRIMARY KEY GENERATED ALWAYS AS IDENTITY
```



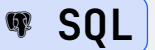
- **PRIMARY KEY:** This denotes that this column is used to uniquely identify each row of the table. This column has a unique value for each row. We'll look more closely at PKs (primary keys) in the near future.

3.1 SQL: Step-by-Step Walkthrough

CREATE-Statement

Let's look at the `_ID` column a little more in detail:

```
1  _ID INTEGER PRIMARY KEY GENERATED ALWAYS AS IDENTITY
```



- `GENERATED ALWAYS AS IDENTITY`: In order to achieve this, we'll let PostgreSQL auto generate the identifier for us.

3.1 SQL: Step-by-Step Walkthrough

CREATE-Statement

- You might have noticed, that each of the columns of a table has a data type.
- Data types, like in conventional programming, denote what type of data the column can contain.



Example

- INTEGER: Integer number data (`int` in C)
- REAL: Real number data (`double` in C)
- TEXT: String data (`char[]` in C)
- DATE: Dedicated date type (no direct equivalent in C)

3.1 SQL: Step-by-Step Walkthrough

CREATE-Statement

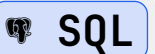
- Furthermore, columns can be defined as `NULL` or `NOT NULL`. This defines if the column can be left empty or not.
- Also, by using `DEFAULT` you can define the default value of a column if left empty.

3.1 SQL: Step-by-Step Walkthrough

INSERT-Statement

- Next, we need some form of data that our table can hold.
- Let's insert three employees. I'll show you the easiest method to do so:

```
1  INSERT INTO EMP
2  (
3    _ID, NAME, DEPARTMENT, SALARY
4  )
5  VALUES
6  (
7    100, 'Max Power', 'HR', 3500
8  );
```

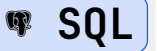


- Alternatively you also use a SELECT-Statement, but more on that later!

3.1 SQL: Step-by-Step Walkthrough

INSERT-Statement

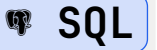
```
1  INSERT INTO EMP
2  (
3    _ID, NAME, DEPARTMENT, SALARY
4  )
5  VALUES
6  (
7    101, 'Tim Maxwell', 'Engineering', 5000
8  );
```



3.1 SQL: Step-by-Step Walkthrough

INSERT-Statement

```
1  INSERT INTO EMP
2  (
3    _ID, NAME, DEPARTMENT, SALARY
4  )
5  VALUES
6  (
7    102, 'Rachel Smith', 'IT', 5500
8  );
```



3.1 SQL: Step-by-Step Walkthrough

SELECT-Statement

- Now, we want to look at the data we just inserted into our database:

1 SELECT

2 _ID, NAME, DEPARTMENT, SALARY

3 FROM

4 EMP;

SQL

ROW_NUM	_ID	NAME	DEPARTMENT	SALARY
1	100	Max Power	HR	3500
2	101	Tim Maxwell	Engineering	5000
3	102	Rachel Smith	IT	5500

Tip

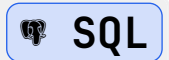
- Alternatively you can use * to select all columns of the table:

```
1 SELECT
```

```
2   *
```

```
3 FROM
```

```
4   EMP;
```



3.1 SQL: Step-by-Step Walkthrough

UPDATE-Statement

- Let's say we want to update the salary of an employee. Maybe they got a raise?

```
1 UPDATE emp
2 SET
3 SALARY = 6000
4 WHERE NAME = 'Max Power';
```




ROW_NUM	_ID	NAME	DEPARTMENT	SALARY
1	100	Max Power	HR	6000
2	101	Tim Maxwell	Engineering	5000
3	102	Rachel Smith	IT	5500

3.1 SQL: Step-by-Step Walkthrough

WHERE-Clause

- You may have noticed in the last example, that we used the keyword `WHERE`.
- It's one of the most important keywords, that you won't be able to live without.
- It defines conditions for the query to be executed.

```
1 SELECT
2   *
3 FROM
4   EMP
5 WHERE
6   SALARY <= 5000;
```

 SQL


ROW_NUM	_ID	NAME	DEPARTMENT	SALARY
2	101	Tim Maxwell	Engineering	5000

3.1 SQL: Step-by-Step Walkthrough

DELETE-Statement

- Now that we know about the `WHERE`-Clause, we can also use it to delete a record in the table.
- Let's say one of the employees has left the company:

```
1 DELETE FROM EMP
2 WHERE NAME = 'Rachel Smith';
```

 SQL

ROW_NUM	_ID	NAME	DEPARTMENT	SALARY
1	100	Max Power	HR	3500
2	101	Tim Maxwell	Engineering	5000

3.1 SQL: Step-by-Step Walkthrough

CRUD

- Using our knowledge, we are now able to design very simple databases!

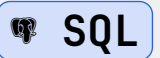
4. DDL: Data Definition Language

4.1 Sequences

Using Sequences

- Apart from tables, we can also use the CREATE-statement to create other database objects.
- One of those objects is the Sequence

```
1 CREATE SEQUENCE <seqname> [ INCREMENT BY < integer > ] [ START  
  WITH < integer > ] [...];
```



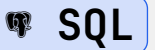
- Sequences allow the developer to define a special sequence. This is useful if your PRIMARY KEY follows a special sequence of values.

4.1 Sequences

Using Sequences

- These can be used in the DEFAULT definition of the column or later used in triggers.

```
1 CREATE SEQUENCE S_USERS INCREMENT BY 5 START WITH 100;  
2 CREATE TABLE USERS (  
3     _ID INTEGER DEFAULT nextval('S_USERS'),  
4     NAME TEXT  
5 )
```



i Info

I tend to prefix sequences with an S (I also do this for other special database objects, we'll learn about later like Views and Trigger!)

ALTER-Statement

? Question

- But what if our data changes? How can we adapt our database to suit our needs?
- Answer: The ALTER-Statement!

ALTER-Statement

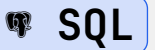
- Using the alter statement, we can add and remove columns, as well as change their data type.

4.2 Altering our database

ALTER-Statement

- Let's say we want to record, when our employees have joined the company.

```
1 ALTER TABLE EMP
2 ADD JOIN_DATE TEXT;
```




4.2 Altering our database

ALTER-Statement

- Oh, darn, we have assigned the wrong data type! The DATE data type is a better fit for this column.
- Let's correct our mistake.

```
1 ALTER TABLE EMP
2 MODIFY COLUMN JOIN_DATE DATE;
```

 SQL


4.2 Altering our database

ALTER-Statement

- And now, the DEPARTMENT of each employee is meant to be stored in a different table. We'll remove it for now.

```
1 ALTER TABLE EMP
```

```
2 DROP COLUMN DEPARTMENT;
```

 SQL

4.2 Altering our database

DROP-Statement

- In the last example, you saw how we can delete a column from a table, by using the DROP statement.
- We can also apply this to tables:

```
1 DROP TABLE EMP;
```

 SQL

! Memorize

If there is no problem with the deletion, this will delete the table and all of the records inside!

4.2 Altering our database

DROP-Statement

- Delete named schema elements, e.g., tables, constraints, schema, indexes, triggers
- DROP needs to observe referential integrity, in order to drop tables in correct order
- Two drop behavior options: → CASCADE → RESTRICT
- DROP deletes all data AND the data definition
 - if you want to delete only the data then use DELETE

4.2 Altering our database

DROP-Statement: Syntax

- Syntax:

```
1 DROP TABLE < relationname > [( CASCADE | RESTRICT )]
```

 SQL

Example

```
1 DROP TABLE Dependent RESTRICT;
```

 SQL

→ The table is dropped only if it is not referenced in any constraint or view or by another element

```
1 DROP TABLE Dependent CASCADE;
```

 SQL

→ The table is dropped even if there are references 436 S

4.2 Altering our database

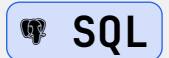
DROP-Statement: Columns

- Here, you can also define the drop behavior:
 - CASCADE
 - RESTRICT



Example

```
1 ALTER TABLE COMPANY.Employee DROP COLUMN Address CASCADE;
```

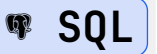


4.2 Altering our database

4. DDL: Data Definition Language

DROP-Statement: Columns

```
1  ALTER TABLE COMPANY.Department ALTER COLUMN Mgr_ssn DROP DEFAULT;
   -- Drop the default value
2
3  ALTER TABLE COMPANY.Department ALTER COLUMN Mgr_ssn SET DEFAULT
   '333445555'; -- Set a new default value
4
5  ALTER TABLE COMPANY.Employee DROP CONSTRAINT EMPSUPERFK CASCADE; -- Drop
   all constraints that depend on EMPSUPERFK
```



4.2 Altering our database

]

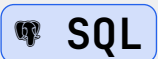
4. DDL: Data Definition Language

4.3 Index

Using INDEX

- Internal structure to increase speed of queries
- Speed up the search for and retrieval of records (access paths) → But slow down inserts and updates → Memory consumption!
- Earlier versions of SQL had commands for creating indexes, but these were removed because they were not at the conceptual schema level
- Many systems still have the CREATE INDEX commands.

```
1 CREATE [ UNIQUE ] INDEX <name> ON < table > ( < column >  
[ , . . . ] )
```



4.3 Index

When to use **INDEX**

- Column is used often for searches or sorting
- Many different values, NULL seldom
- Many rows in table
- More reads than writes on data
- Might be used as join condition
- RDBMS must check value for referential integrity
- Column is an FK
- Referenced column (PK) usually already has an index

4.4 More Objects

More Database Objects

- DB Objects can be CREATED, ALTERed, DROPped
- USER, ROLE
 - DB users and groups

```
1 CREATE USER user [ IDENTIFIED BY [PASSWORD] ' passwd ' ] [ , user  
  [ IDENTIFIED BY [PASSWORD] ' passwd ' ] ];
```

 SQL

- VIEW
 - User view on table (external layer)

```
1 CREATE OR REPLACE VIEW view [SELECT ... FROM ...];
```

 SQL

! Memorize

While on first glance views might not seem all that powerful, they are one of the most used database objects, offering a wide variety of uses in application development.

4.4 More Objects

USER and ROLE

- Example: User
 - Either owner of a relation or the DBA can grant (or revoke) selected users the privileges to use a SQL statement (e.g., SELECT, INSERT, DELETE)

```
1 CREATE USER 'student' IDENTIFIED BY '123';
2 GRANT ALL PRIVILEGES ON COMPANY.Employee TO 'student';
3
4 REVOKE DROP ON COMPANY.Employee FROM 'student';
5 SHOW GRANTS FOR student;
```



4.4 More Objects

More examples

- TABLESPACE → Grouping of tables based on physical storage
- SYNONYM → Alias name for tables, views, sequences
- FUNCTION, PROCEDURE → Stored Procedure, Persistent Stored Module (PSM)
- TRIGGER → Active rule for certain events

5. DML: Data Manipulation Language

5.1 Simple Data Manipulation

INSERT, UPDATE, DELETE

- INSERT, UPDATE, DELETE
 - All operations work on a set of tuples
 - Special case(!): work on one tuple
 - Example for modifications of sets of tuples:
 - Increase the wage of all employees by 10
 - Delete stock with price below 1€
 - Set the academic title of some students to 'BSc'

5.2 Transactions in a nutshell

What is a transaction?

- Start a transaction → While some DBMS (e.g. PostgreSQL) need to explicitly start a transaction, some others (e.g. Oracle) do not
- After starting a transaction, you can make changes to the db that stay local to your session.
- Whenever you feel ready, you can use `COMMIT;` to commit those changes to the database.
- If you make a mistake, you can use `ROLLBACK;` while in a transaction to undo your changes.

! Memorize

If you forget to explicitly commit your changes, you will lose those changes when quitting the transaction.

5.3 Inserting into tables

INSERT-statement

```
1  INSERT INTO < table > [ ( < column > [ , ... ] ) ] VALUES ( <  
expression > [ , ... ] );
```



- The column list is optional
 - If omitted, values list must match table's attributes
 - If given, we don't have to specify values for all columns. Other columns will get the DEFAULT value (or NULL).

5.3 Inserting into tables

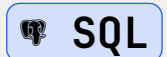
INSERT-statement

- In general, you have two possibilities for inserting into a table using the INSERT-statement
 - **value list**
 - tuples returned by a query



Example

```
1  INSERT INTO EMPLOYEE VALUES ( 'Arthur' , 'C' , 'Brown' ,  
    323232323 , '1970-12-31' , 'London' , 'm' , 45000 , 333445555 ,  
    5 );
```



```
2  INSERT INTO EMPLOYEE ( first_name, last_name, emp_num, superior_num,  
    dept_num) VALUES ( 'Andi' , 'Red' , 343434343 , 333445555 , 5 );
```

i Info

You may define what columns to fill. You must fill all **NOT NULL** columns.

5.3 Inserting into tables

INSERT-statement



Example

```
1  INSERT INTO Book
   VALUES ( 1 , 1 , 4712 ,
   'DBS' );
2  INSERT INTO Book VALUES ( 2 ,
   2 , 9991 , 'DB1' );
3  COMMIT;
4  INSERT INTO Book VALUES ( 3 ,
   NULL , 4242 , 'Hitch' );
5  ROLLBACK;
6  INSERT INTO Book ( BNr, ISBN ,
   Title ) VALUES ( 3 , 4242,
   'Hitchhiker');
```

Book	<u>BNr</u>	PNr	ISBN	Title
	1	1	4212	DBS
	2	2	9991	DB1
	3	NULL	4242	Hitchhiker

5.3 Inserting into tables

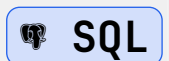
INSERT-statement

- In general, you have two possibilities for inserting into a table using the INSERT-statement
 - value list
 - **tuples returned by a query**



Example

```
1 INSERT INTO USERS ( last_name, first_name ) SELECT last_name,  
first_name FROM EMP WHERE IS_USER = 1;
```



5.3 Inserting into tables

Date formats

- Inserting dates into the DB requires to use the format that depends on the flavor of DBMS

```
1 UPDATE Person SET birthdate = '2008-12-31'
```

 SQL

- A date-function can also be used:

```
1 INSERT INTO Person (name, birthdate) VALUES ('Anna',  
  '02-FEB-1955');
```

 SQL

```
2 INSERT INTO Person (name, birthdate) VALUES ('Anna',  
  TO_DATE('02.02.1955'));
```

```
3 INSERT INTO Person (name, birthdate) VALUES ('Anna',  
  TO_DATE('02-02-1955', 'DD-MM-YYYY'));
```

5.3 Inserting into tables

Constraints

- All modifications need to observe constraints:
 - Domain Constraints meaning the data types must match
 - If type conversion are implicit or are required to be explicit is vendor-specific
 - Entity Integrity, so whether or not the Primary Key value is not null and unique
 - Referential Integrity (Foreign Key), so if the master table has a row that matches the referenced data
 - Semantical Integrity (check constraints)

5.4 Updating table data

UPDATE-statement

```
1  UPDATE < table > SET < column > = < expression > [ WHERE < condition >];
```



- Used to modify attribute values of one or more selected tuples
- Can modify only tuples of one table at a time
- WHERE-clause is optional and if left out updates all tuples of the table

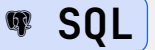
i Info

Updating a primary key value may propagate to the foreign key values of tuples in other relations if such a referential triggered action is specified in the referential integrity constraints of the DDL.

5.4 Updating table data

UPDATE-statement: examples

```
1  UPDATE Person SET lname= 'Brown', married = TRUE WHERE id = 45;  
2  UPDATE Employee SET salary = salary * 1.1;  
3  UPDATE Person SET email = NULL WHERE email IS NOT NULL;
```



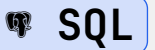
]

5.5 Deleting table data

DELETE-statement

- Removes tuples from a relation, the relation stays in the database

```
1  DELETE FROM < table > [WHERE < condition >]
```



- Again, the WHERE-clause is optional and, if left out, will delete all tuples!

! Memorize

It is imperative to observe the referential integrity!

5.5 Deleting table data

DELETE-statement

5. DML: Data Manipulation Language

5.5 Deleting table data

- Insert a new student <‘Johnson’, 25, 1, ‘Math’> into the database.
- Change the class of STUDENT ‘Smith’ to 2.
- Insert a new course: <‘Knowledge Engineering’, ‘CS4390’, 3, ‘CS’>.
- Delete the record for the student whose name is ‘Smith’ and whose student number is 17.

STUDENT

Name	Student_number	Class	Major
Smith	17	1	CS
Brown	8	2	CS

COURSE

Course_name	Course_number	Credit_hours	Department
Intro to Computer Science	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Database	CS3380	3	CS

SECTION

Section_identifier	Course_number	Semester	Year	Instructor
85	MATH2410	Fall	07	King
92	CS1310	Fall	07	Anderson
102	CS3320	Spring	08	Knuth
112	MATH2410	Fall	08	Chang
119	CS1310	Fall	08	Anderson
135	CS3380	Fall	08	Stone

GRADE REPORT

Student_number	Section_identifier	Grade
17	112	B
17	119	C
8	85	A
8	92	A
8	102	B
8	135	A

PREREQUISITE

Course_number	Prerequisite_number
CS3380	CS3320
CS3380	MATH2410
CS3320	CS1310

6. Queries: An in-depth look

6.1 What are queries?

- Ad-hoc-formula: No programs, requests!
- Descriptiveness: “What do I want”, not “How do I get it”
- Set-orientation: Much data at once, not only a single tuple
- Seclusion: All results are relations again and can be queried again
- Adequate: All data model constructs are supported
- Orthogonal: There are view independent commands that can be combined

6.2 What is a good Query Language?

- Performant: The language is transformable, so that the user may use simple queries that are substituted into fast ones (with the same result!).
- Efficient: Each operation can be executed efficiently.
- Secure: All queries lead to finite result sets in finite time.
- Complete: Everything that is requestable, can be formulated by a query.
- Definitive: The same request on the same data always yields the same result.

6.3 Queries in SQL

DML: Data Manipulation Language

```
1  INSERT INTO (...) VALUES  
   (...);
```

 SQL

```
2  DELETE FROM ...;
```

```
3  UPDATE ... SET ...;
```

DQL: Data Query Language

```
1  SELECT ... FROM ...;
```

 SQL

! Memorize

- DQL is only used to retrieve data that is already there.
- It does not modify data.
- It only uses one basic keyword: SELECT

Relational Model vs. SQL

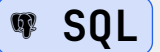
! Memorize

SQL allows, in contrast to the Relational Model, a table to have two or more tuples to be identical in all of their attributes. This makes SQL tables multi sets by design.

- There are ways to enforce tables to follow certain rules using SQL constraints.
- We'll check these out later in the lecture.

Query Syntax

```
1 SELECT [ DISTINCT | ALL ] < attribute_list >
2 FROM < table list >
3 [ WHERE < condition > ]
4 [ <group by clause > ] [ <having clause > ]
5 [ UNION [ ALL ] < query specification> ]
6 [ < order by clause > ]
```



Basic Syntax

```
1  SELECT <attribute list>
2  FROM <table list>
3  WHERE <condition>
```



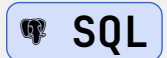
- <attribute list> is a list of attribute names (columns) whose values are to be retrieved by the query
- <table list> is a list of the relation names (e.g., tables) required to process the query
- <condition> is an optional conditional (Boolean) expression that identifies the tuples to be retrieved by the query

Examples



Example

```
1 SELECT Bdate, Address
2 FROM Employee
3 WHERE Fname = 'John' AND Minit = 'B' AND Lname = 'Smith';
4
5 SELECT Fname, Lname, Address
6 FROM Employee, Department
7 WHERE Dname = 'Research' AND Dnumber = Dno;
```



? Question

What do these queries mean?

Attribute List

- The <attribute list> represents the selected columns, whose values are supposed to be retrieved.

```
1 SELECT NAME, SALARY
2 FROM EMP
```

 SQL

- Using the *, you can select all columns at once.

Info

You can also use aggregate functions, arithmetic expressions as well as constants in your queries:


```
1 SELECT 'Hello World!' as HELLO_WORLD FROM EMP;
```

 SQL

Attribute List

- When the column names are unambiguous, you can just write their name
- If the same column could be attributed to multiple columns, you need to write the table or table alias in front of the name:

```
1 SELECT d._ID, e.NAME, e.SALARY as sal
2 FROM EMP e
3 LEFT JOIN DEPARTMENT d ON d._ID = e.DEPARTMENT_ID
```

 SQL


Info

As seen in the previous example, both tables and attributes can be given an alias either by using AS or by simply writing the alias after the table name.

Using **DISTINCT**

- When you want to retrieve all unique values for a data set, you can use the keyword **DISTINCT**.
- This will deduplicate the results, depending on the set of attributes you selected in your **SELECT** statement.

```
1 SELECT DISTINCT e.NAME
2 FROM EMP e
```

 SQL

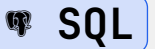
- This query will only retrieve all unique names of all employees. If an employee is listed more than once due to some reason, their name will only appear once in the result of the query.

Table List

- The `<table list>` is a list of relation names (tables) required to process the query.

```
1 SELECT * FROM EMP;
```

```
2 SELECT * FROM EMP, DEPT;
```



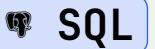
! Memorize

- If there is more than one table, then the resulting relation is the Cartesian Product of all of these tables.
 - This may lead to very huge result list if used without a `<condition>`!

Condition

- The <condition> is a boolean expression that identifies the tuples to be retrieved by the query.

```
1 SELECT * FROM EMP WHERE _ID = 100;  
2 SELECT * FROM EMP WHERE NAME is not null;
```



i Info

The WHERE-clause is **optional**:

- If left out → retrieves all tuples
- If there is two or more relations in the <table list> → SQL selects the Cartesian Product

WHERE: Logic Basics

- The WHERE-clause allows you to compare two expressions
 - Comparison is done using: <, >, <=, >=, =, <>
 - Both literals and columns are allowed

```
1 SELECT * FROM USERS WHERE age >= 18;  
2 SELECT * FROM USERS WHERE NAME <> 'Miller';
```

 SQL

- Checking for NULL: IS (NOT) NULL
- Logical Operators like AND, OR and NOT can add more nuance to the WHERE-clause.

```
1 SELECT * FROM USERS WHERE age >= 18 AND NAME <> 'Miller';
```

 SQL

WHERE: NULL

? Question

- What is $42 < \text{NULL}$?
- Comparisons with NULL are never true
- Therefore, we need a third boolean state \rightarrow TRUE, FALSE and NULL

1 NOT NULL \rightarrow NULL

2

3 TRUE AND NULL \rightarrow NULL

4 FALSE AND NULL \rightarrow FALSE

5

6 TRUE OR NULL \rightarrow TRUE

7 FALSE OR NULL \rightarrow NULL

SQL

6.3 Queries in SQL

WHERE: NULL

6. Queries: An in-depth look

a	b	a AND b	a OR b	NOT a
0	0	0	0	1
0	1	0	1	1
0	NULL	0	NULL	1
1	0	0	1	0
1	1	1	1	0
1	NULL	NULL	1	0
NULL	0	0	NULL	NULL
NULL	1	NULL	1	NULL
NULL	NULL	NULL	NULL	NULL

6.4 Set Operations

Set Operations

- SQL allows for some of the basic set operations:
 - Union (UNION)
 - Set Difference (EXCEPT)
 - Intersection (INTERSECT)

i Info


Set Operations apply only to union-compatible relations:

- the two relations need to have the same number of attributes
- each corresponding set of attributes has the same domain

6.4 Set Operations

Set Operations: Example

```
1  (SELECT DISTINCT USER_NAME FROM USERS WHERE age >= 18)
2  UNION
3  (SELECT DISTINCT USER_NAME FROM USERS WHERE LAST_NAME = 'Miller');
```

 SQL

? Question

What does the above statement do?


Multiset Operations

- SQL allows for certain multiset operations using the keyword ALL.

1 **UNION ALL**

2 EXCEPT ALL

3 INTERSECT ALL

 SQL

6.4 Set Operations

Assignment

6. Queries: An in-depth look

6.4 Set Operations

- Retrieve the names of all students with Class 2 majoring in “CS” (computer science).
- Retrieve the names of all courses taught by Professor King in 2007 and 2008.
- For each section taught by Professor King, retrieve the course number, semester, year, and name of students who took the section.
- Retrieve the name and transcript of each student with Class 2 majoring in CS. A transcript includes course name, course number, credit hours, semester, year, and grade for each course completed by the student.

6. Queries: An in-depth look

STUDENT

Name	Student_number	Class	Major
Smith	17	1	CS
Brown	8	2	CS

COURSE

Course_name	Course_number	Credit_hours	Department
Intro to Computer Science	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Database	CS3380	3	CS

SECTION

Section_identifier	Course_number	Semester	Year	Instructor
85	MATH2410	Fall	07	King
92	CS1310	Fall	07	Anderson
102	CS3320	Spring	08	Knuth
112	MATH2410	Fall	08	Chang
119	CS1310	Fall	08	Anderson
135	CS3380	Fall	08	Stone

GRADE_REPORT

Student_number	Section_identifier	Grade
17	112	B
17	119	C
8	85	A
8	92	A
8	102	B
8	135	A

PREREQUISITE

Course_number	Prerequisite_number
CS3380	CS3320
CS3380	MATH2410
CS3320	CS1310

7. The JOIN statement

7.1 Joining tables

What is joining two tables?

- More often than not, the magic of SQL happens when you join two tables together.
- This is done using the SQL statement JOIN, which combines two tables either by building the cartesian product or by connecting them based on certain criteria.
- There are multiple types of joins available, which we will explore in the following slides.

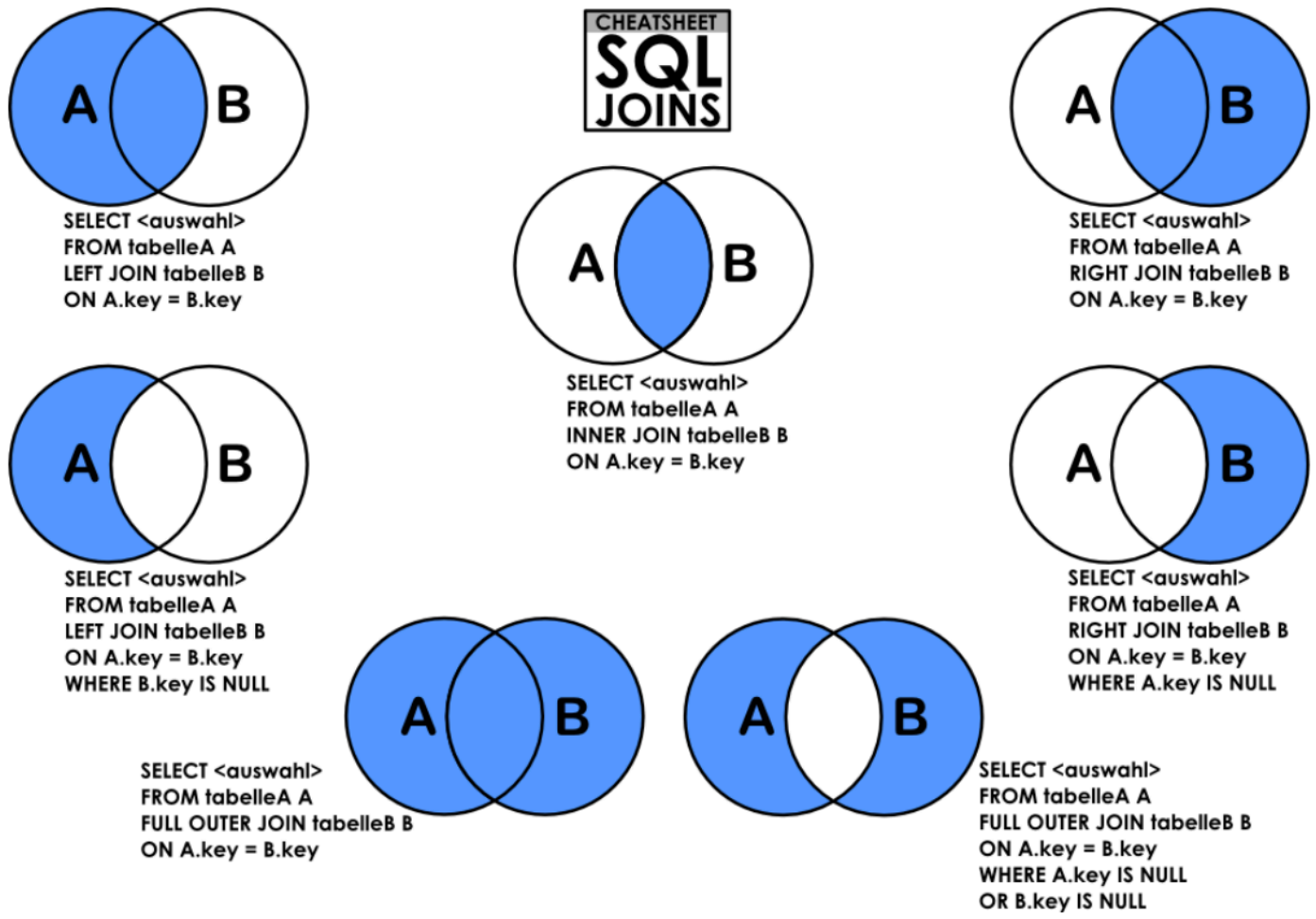
! Memorize

It can be difficult to wrap your head around, how different tables will end up looking after a join:

1. Don't fret, this is a difficult subject!
2. It is often very useful to try different types of joins and to look at the entire result of the unfiltered query.

7.1 Joining tables

Overview



Source: <https://stackoverflow.com/questions/59590346/trying-to-do-a-left-join-but-ending-up-getting-empty-result>

7.1 Joining tables

(INNER) JOIN

- The simplest form of joining two tables is the INNER JOIN.
- Only rows that match the condition defined in the join are returned.

```
1  SELECT *
2  FROM EMP e
3  INNER JOIN DEPARTMENT d ON e.DEPARTMENT_ID = d._ID;
```



! Memorize

While you also use just a JOIN and also just a comma to also execute an inner join, I ask you to always write INNER JOIN as it is by far easier to read what you are actually doing.

7.1 Joining tables

LEFT (OUTER) JOIN

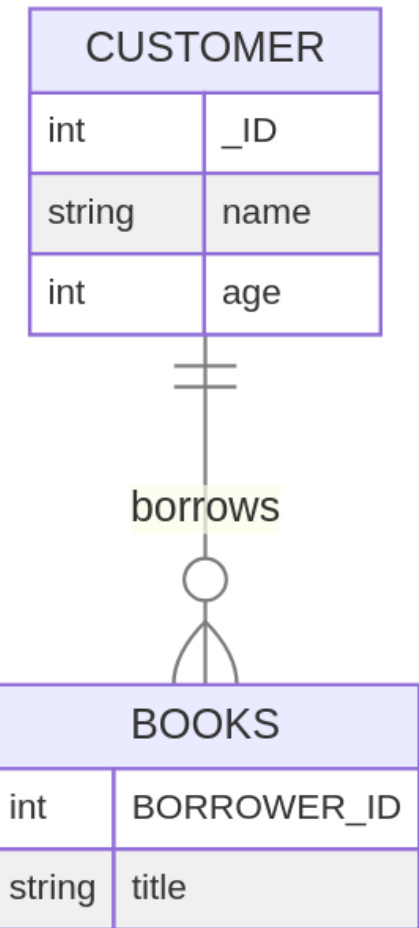
- In practice, this is one of the most used join types. This join type will retain all rows of the left (main) table, while appending the rows of the right to that.
- If a row on the left table cannot be matched to a row in the right table, then the values of the right table will be left empty with the columns still attached.

```
1 SELECT * FROM CUSTOMERS c
2 LEFT JOIN BOOKS b ON c._ID = b.BORROWER_ID;
```

[SQL](#)

i Info

- You will also find the name LEFT OUTER JOIN, but I'll keep using LEFT JOIN.



7.1 Joining tables

RIGHT (OUTER) JOIN

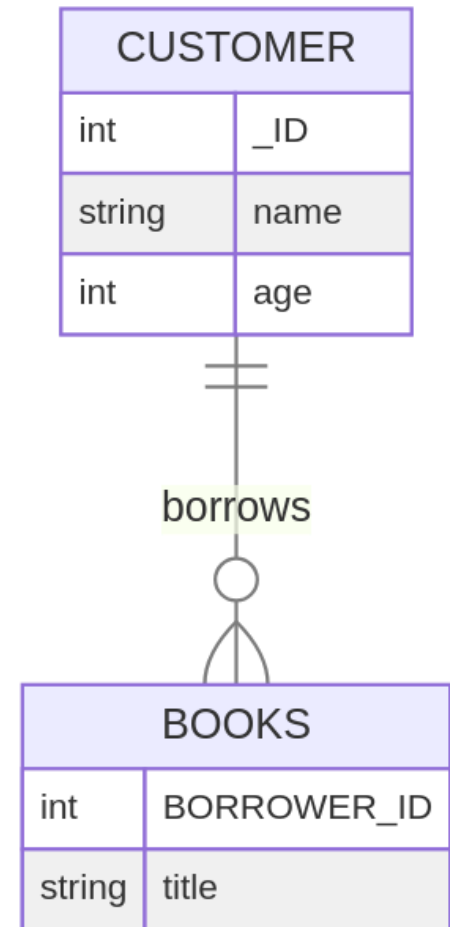
- This join type does the same as the LEFT JOIN but for the right-hand table instead.

```
1 SELECT * FROM CUSTOMERS c
2 RIGHT JOIN BOOKS b ON c._ID = b.BORROWER_ID;
```



i Info

You will, similarly to the previous example, find the term RIGHT OUTER JOIN.

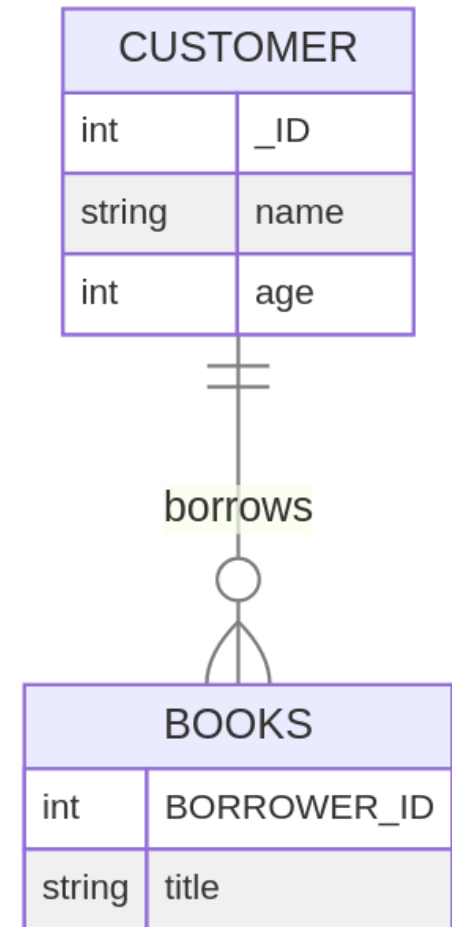


7.1 Joining tables

FULL (OUTER) JOIN

- This join type combines the LEFT JOIN and RIGHT JOIN and selects all records from both tables, even if there are rows that do not match.

```
1 SELECT * FROM CUSTOMERS c
2 FULL JOIN BOOKS b ON c._ID = b.BORROWER_ID;
```

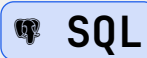
[SQL](#)

7.1 Joining tables

Info

You can also achieve the same behaviour by using UNION on two queries, one with a LEFT JOIN and the other with a RIGHT JOIN.

```
1 SELECT * FROM CUSTOMERS c
2 LEFT JOIN BOOKS b ON c._ID = b.BORROWER_ID;
3 UNION
4 SELECT * FROM CUSTOMERS c
5 RIGHT JOIN BOOKS b ON c._ID = b.BORROWER_ID;
```



7.1 Joining tables

CROSS JOIN

- Using CROSS JOIN allows the developer to build the Cartesian Product of the two tables, so that each column is matched to each column of the right-hand table once.

```
1  SELECT * FROM CUSTOMERS c
2  CROSS JOIN BOOKS b;
```

[SQL](#)

i Info

In this example, this does not make a lot of sense, but in certain cases this can be a life-saver!

7.2 Why use JOIN?

Using JOIN to combine tables

- The power of relational database comes from its ability to connect different tables using well-defined connections
- Any column can be used to combine tables, but there are also FOREIGN KEYS that allow these connections to be more complex.
- This helps normalizing databases and to remove duplicate data sets.

i Info

We'll look at these concepts more in-depth in the one of the next lectures, so stay tuned for that!

8. SQL: Advanced features

8.1 SELECT without a table

- Sometimes, you might want to select data, such as the current date, without accessing a table.
- In PostgreSQL you can just write:

```
1 SELECT NOW( );
```

 SQL

```
1 OUTPUT: 2023-04-15 13:56:51.120277+02
```

! Memorize

- In some other flavours, there is a dummy table called `dual`. They serve the same purpose!


8.2 WHERE clause

Info

- We discussed the WHERE-clause earlier:
 - It contains logical operators to filter the query.
 - The WHERE-clause is optional.

Querying NULL values

```
1 WHERE a IS NULL;  
2 WHERE a IS NOT NULL;
```

 SQL

8.2 WHERE clause

Using BETWEEN



Example

Let's say you want to query for all your employees between the ages of 18 and 21.

```
1 WHERE age >= 18 AND age <= 21;
```



```
2 WHERE age BETWEEN 18 AND 21;
```

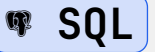
Using IN



Example

Let's say you want to check if the person entering data into the form is one of three special employees.

```
1 WHERE _ID = 102 OR _ID = 304 OR _ID = 201;  
2 WHERE _ID IN (102, 304, 201);
```



- The IN-clause checks whether a value is part of a set.
- Improves the readability of the code!

8.2 WHERE clause

String Patterns using LIKE



Example

Imagine you only save the full name of each employee. Now you want to query of all employees, whose last name is Smith.

```
1 WHERE NAME LIKE '%SMITH';
```

[SQL](#)

- %: Replaces an arbitrary number of letters and numbers
- _: Replaces a single character
- \: Escapes one of the wildcard characters (AB_CD → AB_CD)

```
1 `
2 'abc' LIKE 'abc' -> TRUE
3 'abc' LIKE 'a%' -> TRUE
```

[SQL](#)

```
4 'abc' LIKE '_b_' -> TRUE
```

```
5 'abc' LIKE 'b' -> FALSE
```


8.2 WHERE clause

Comparison with DATE



Example

Let's say you want to congratulate all employees who have started in the founding year of your company. Let's take 2018 as an example.

```
1 WHERE JOIN_DATE >= '2018-01-01' AND JOIN_DATE <= '2018-12-31';  
2 WHERE JOIN_DATE BETWEEN '2018-01-01' AND '2018-12-31';  
3 WHERE JOIN_DATE::TEXT LIKE '2018%';
```

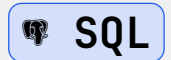


SQL

! Memorize

The example of date comparisons are DBMS dependant. Example for Oracle:

```
1 WHERE JOIN_DATE >= TO_DATE('2018-01-01', 'yyyy.mm.dd') AND  
   JOIN_DATE <= TO_DATE('2018-12-31', 'yyyy.mm.dd')
```



8.3 Sorting query results

Using **ORDER BY**

- The results of queries are sets, meaning they have no order applied to them
- Using **ORDER BY**, you can impose an order on the result of a query
- You can order by more than one column.

```
1 SELECT
```

```
2     NAME
```

```
3 FROM
```

```
4     EMP
```

```
5 ORDER BY
```

```
6     _ID;
```

 SQL

! Memorize

You can change the direction of the sort by using ASC (ascending) and DESC (descending).

```
1 ORDER BY _ID ASC;
```

```
2 ORDER BY _ID DESC;
```

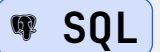


8.4 Aggregate Functions

Using Aggregate Functions

- Using aggregate functions, you can analyze your data and create summaries of the shape of your data.
- For instance, you can count the number of rows that match your condition or simply return the maximum value of a set of values.

```
1 SELECT
2   COUNT(NAME)
3 FROM
4   EMP
5 WHERE
6   NAME LIKE '%SMITH'
7 ORDER BY
8   _ID;
```

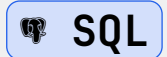


! Memorize

You can change the direction of the sort by using ASC (ascending) and DESC (descending).

```
1 ORDER BY _ID ASC;
```

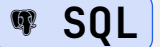
```
2 ORDER BY _ID DESC;
```



8.4 Aggregate Functions

Using Aggregate Functions

```
1 SELECT COUNT(*) FROM Book; -> 4
2 SELECT COUNT(PNr) FROM Book; -> 3
3 SELECT COUNT(DISTINCT PNr) FROM Book; -> 2
4 SELECT MIN(Price), MAX(Price) FROM Book; -> 9.99
5 SELECT SUM(Price) FROM Book; -> 64.87
6 SELECT AVG(Price) FROM Book; -> 16.22
```



8.5 GROUP BY & HAVING

Using GROUP BY

- Grouping is used to create subgroups of tuples before summarization
 - partition the relation into nonoverlapping subsets (or groups) of tuples
 - Using a grouping attribute
 - Grouping attribute should appear in the SELECT clause
 - If NULLs exist in the grouping attribute, then a separate group is created for all tuples with a NULL value



Example

For each department, we want to retrieve the department number, the number of employees in the department, and their average salary.

```
1 SELECT
2   DEPARTMENT, COUNT(*), AVG(SALARY)
```




```
3 FROM EMP
```

```
4 GROUP BY DEPARTMENT;
```

8.5 GROUP BY & HAVING

Using HAVING

- HAVING provides a condition on the summary information regarding the group of tuples associated with each value of the grouping attributes
- Only the groups that satisfy the condition are retrieved in the result of the query
- HAVING clause appears in conjunction with GROUP BY clause

i Info

- Selection conditions in WHERE clause limit the tuples
- HAVING clause serves to choose whole groups

8.5 GROUP BY & HAVING

Using HAVING

- Imagine you have a whole lot of employees in your company and you want to find the name and amount of employees per department, but only take into account those department that have 10 or more employees.

```
1 SELECT NAME, count(*) as emp_amount
2 FROM EMP
3 GROUP BY DEPARTMENT
4 HAVING count(*) >= 10;
```

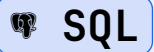


8.5 GROUP BY & HAVING

Using HAVING

- Let's say you wanna retrieve all projects in your company that have more than 2 employees working on it. The PROJECTS table contains all projects and the EMP_PROJECT_RESPONSIBILITY matches employees to projects.

```
1 SELECT p.PROJECT_NUM, p.PROJECT_NAME, count(*) as emp_amount
2 FROM PROJECTS p
3 LEFT JOIN EMP_PROJECT_RESPONSIBILITY e ON e.project_id = p._id
4 GROUP BY p.PROJECT_NUM, p.PROJECT_NAME
5 HAVING count(*) > 2;
```



8.5 GROUP BY & HAVING

Using HAVING

i Info

- When using groups, only two types of things are allowed in a SELECT-clause:
 - Aggregate Functions
 - Columns contained in a GROUP BY-statement
- And while HAVING allows aggregate functions, WHERE does not.

8.6 Special Features

Special Features not covered

- ANY and SOME
- ALL in comparisons of a WHERE-clause
- EXISTS in a WHERE-clause
- UNIQUE in a WHERE-clause
- Nested queries

8.6 Special Features

Assignment

- How many students are studying CS?
- List all course names and how often they have been taught.
- For each section taught by Professor Anderson, retrieve the course number, semester, year and number of students who took the section.

8.7 Query: Summary

```
1  SELECT <attribute and function list>
2  FROM <table list>
3  [ WHERE <condition> ]
4  [ GROUP BY <grouping attribute(s)> ]
5  [ HAVING <group condition> ]
6  [ ORDER BY <attribute list> ];
```

 SQL

8.8 Query: Execution Order

- Order of Execution:
 - FROM (cartesian product, JOIN)
 - WHERE (selection)
 - GROUP BY (grouping)
 - HAVING (condition on group)
 - ORDER BY (sorting)
 - SELECT (projection)

i Info

- The optimizer might build a different execution order if that would speed up the query.

9. License Notice

9.1 Attribution

- This work is shared under the CC BY-NC-SA 4.0 License and the respective Public License.
- <https://creativecommons.org/licenses/by-nc-sa/4.0/>
- This work is based off of the work by Prof. Dr. Ulrike Herster.
- Some of the images and texts, as well as the layout were changed.
- The base material was supplied in private, therefore the link to the source cannot be shared with the audience.