Databases

Lecture 2: SQL - Structured Query Language

Emily Lucia Antosch

HAW Hamburg

13.02.2025

1. Introduction

- 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. Introduction

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

- 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

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:

```sql
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 );
```
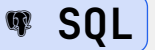
CREATE-Statement

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

```sql
1 _ID INTEGER PRIMARY KEY GENERATED ALWAYS AS IDENTITY
```
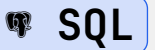
- `_ID`: Name of the column

> **? 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!

CREATE-Statement

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

```sql
1 _ID INTEGER PRIMARY KEY GENERATED ALWAYS AS IDENTITY
```
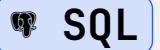
- `INTEGER`: The data type of the column. We'll explore data types a more in-depth very shortly

CREATE-Statement

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

```sql
1 _ID INTEGER PRIMARY KEY GENERATED ALWAYS AS IDENTITY
```
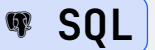
- `PRIMARY KEY`: This denotes that is column is used to unique identify each row of the table. This column has a unique value for each row. We'll look more closely at PKs (primary keys) in the near future.

CREATE-Statement

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

```sql
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.

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)

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 let empty.

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:

```sql
1  INSERT INTO EMP
2  (
3    _ID, NAME, DEPARTMENT, SALARY
4  )
5  VALUES
6  (
7    100, 'Max Power', 'HR', 3500
8  );
```
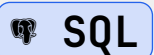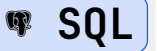
- Alternatively you also use a SELECT-Statement, but more on that later!
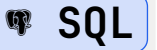
INSERT-Statement

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

INSERT-Statement

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

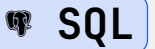**SELECT**-Statement

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

```sql
1  SELECT
2     _ID, NAME, DEPARTMENT, SALARY
3  FROM
4     EMP;
```

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:

```sql
1  SELECT
2      *
3  FROM
4      EMP;
```
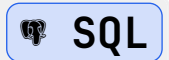
UPDATE-Statement

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

```sql
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

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.

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

ROW_NUM	_ID	NAME	DEPARTMENT	SALARY
2	101	Tim Maxwell	Engineering	5000

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:

```sql
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

CRUD

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

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.

ALTER-Statement

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

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

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.

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

3.2 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.

```SQL
1  ALTER TABLE EMP
2  DROP COLUMN DEPARTMENT;
```

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:

```SQL
1  DROP TABLE EMP;
```

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

4. DML: Data Manipulation Language

5. The `JOIN` statement

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**
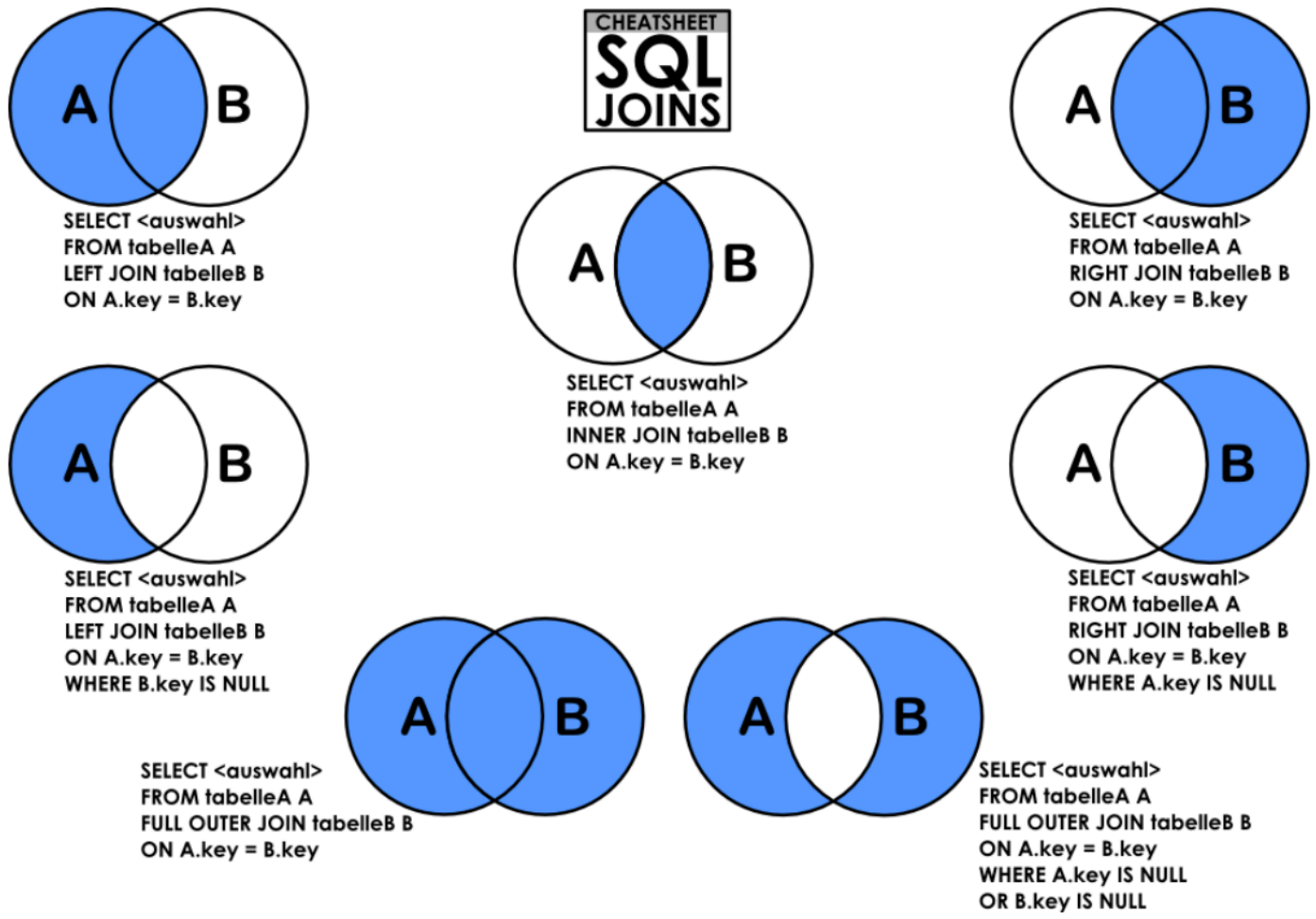>
> It can be difficult to wrap your head around, how different tables will end up looking after a join:
> 1. Don't fret, this is a difficult subject!
> 2. It is often very useful to try different types of joins and to look at the entire result of the unfiltered query.

Overview

5.1 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.

```sql
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.

5.1 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.

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

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

CUSTOMER

int	_ID
string	name
int	age

borrows

BOOKS

int	BORROWER_ID
string	title
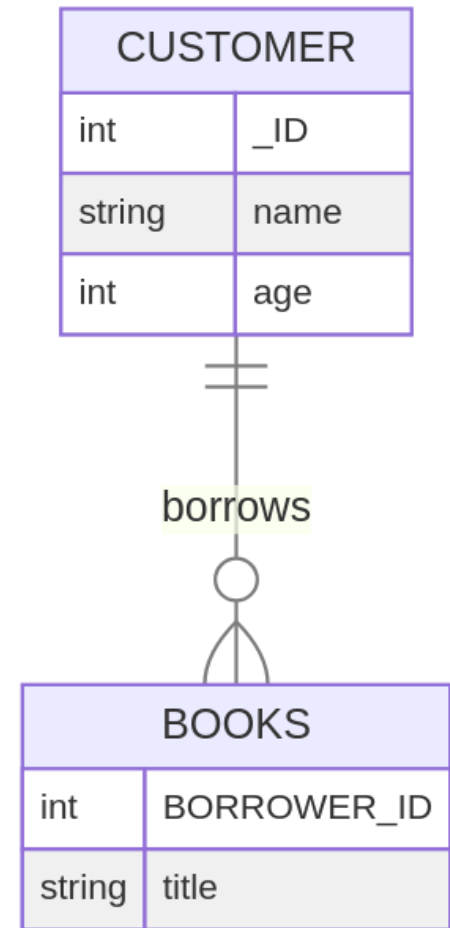
5.1 Joining tables

RIGHT (OUTER) JOIN

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

```sql
1  SELECT * FROM CUSTOMERS c
2  RIGHT JOIN BOOKS b ON c._ID = b.BORROWER_ID;
```
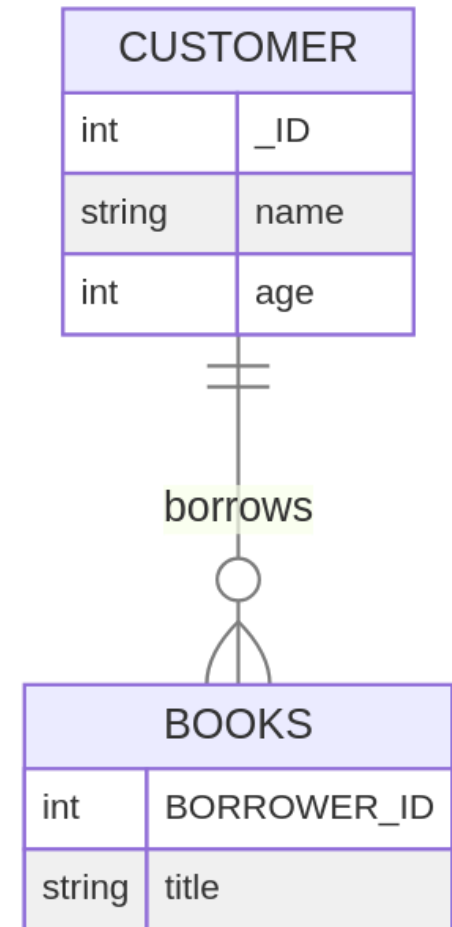
> **i Info**
>
> You will, similarly to the previous example, find the term RIGHT OUTER JOIN.

CUSTOMER

int	_ID
string	name
int	age

borrows

BOOKS

int	BORROWER_ID
string	title

5.1 Joining tables

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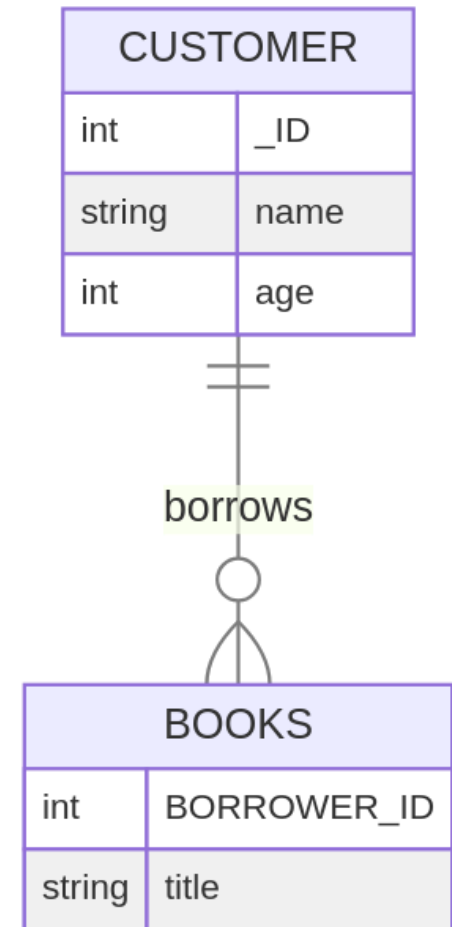
- This join type combines the LEFT JOIN and RIGHT JOIN and selects all records from both tables, even if there are rows that do not match.

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

CUSTOMER

int	_ID
string	name
int	age

borrows

BOOKS

int	BORROWER_ID
string	title

> **_i_ 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.
>
> ```sql
> 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;
> ```

6. SQL: Advanced features

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

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

```
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!

> **ℹ Info**
>
> - We discussed the `WHERE`-clause earlier:
>   - ‣ It contains logical operators to filter the query.
>   - ‣ The `WHERE`-clause is optional.

Querying `NULL` values

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

Using **BETWEEN**

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

```sql
1  WHERE age >= 18 AND age <= 21;
2  WHERE age BETWEEN 18 AND 21;
```
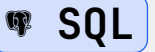
Using **IN**

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

```sql
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!

6.2 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.

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

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

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

```
4  'abc' LIKE '_b_' -> TRUE
5  'abc' LIKE 'b' -> FALSE
```

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.

```SQL
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%';
```

> ❗ **Memorize**
>
> The example of date comparisons are DBMS dependant. Example for Oracle:
>
> ```sql
> WHERE JOIN_DATE >= TO_DATE('2018-01-01', 'yyyy.mm.dd') AND
> JOIN_DATE <= TO_DATE('2018-12-31', 'yyyy.mm.dd')
> ```
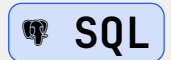
Using ORDER BY

- 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.

```sql
1  SELECT
2      NAME
3  FROM
4      EMP
5  ORDER BY
6      _ID;
```

> ❗ **Memorize**
>
> You can change the direction of the sort by using `ASC` (ascending) and `DESC` (descending).
>
> ```sql
> 1 ORDER BY _ID ASC;
> 2 ORDER BY _ID DESC;
> ```

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.

```sql
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).
>
> ```sql
> 1 ORDER BY _ID ASC;
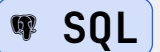> 2 ORDER BY _ID DESC;
> ```

6.4 Aggregate Functions

Using Aggregate Functions

```SQL
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
```

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.

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

```
3  FROM EMP
4  GROUP BY DEPARTMENT;
```

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

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.

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

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.

```sql
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;
```

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.

6.6 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

6.6 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.

```sql
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

- 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.