Compliments of MotherDuck

DuckDB IN ACTION

Mark Needham
Michael Hunger
Michael Simons

MEAP

MANNING

MotherDuck

SERVERLESS SQL ANALYTICS POWERED BY DUCKDB

USE IT FOR:

CLOUD DATA WAREHOUSE FOR THE REST OF US
DATA LAKE QUERY ENGINE
SERVERLESS BACKEND FOR DATA APPS

INCLUDES:

BEAUTIFUL WEB IDE AND NOTEBOOK FOR DUCKDB
CENTRALIZED DATA STORAGE AND SHARING FOR ORGS
SUPPORT FOR MANY TOOLS IN THE MDS ECOSYSTEM

HYBRID QUERYING: USE YOUR LAPTOP DUCKDB WITH THE CLOUD

MOTHERDUCK.COM

What is MEAP?

A book can take a year or more to write, so how do you learn that hot new technology today? The answer is MEAP, the Manning Early Access Program. In MEAP, you read a book chapter-by-chapter while it's being written and get the final eBook as soon as it's finished. In MEAP, you get the book before it's finished, and we commit to sending it to you immediately when it is published. The content you get is not finished and will evolve, sometimes dramatically, before it is good enough for us to publish. But you get the chapter drafts when you need the information. And you get a chance to let the author know what you think, which can really help us both make a better book.

MEAP offers several benefits over the traditional "wait to read" model.

- Get started now. You can read early versions of the chapters before the book is finished.
- Regular updates. We'll let you know when updates are available.
- Contribute to the writing process. Your feedback in the liveBook Discussion Forum makes the book better.

To learn more about MEAP, visit https://www.manning.com/meap-program.

MEAP Edition
Manning Early Access Program

DuckDB in Action

Version 2

For more information on this and other Manning titles go to manning.com.

Foreword

Welcome, dear reader, to this book about DuckDB. It feels somewhat surreal to write a foreword for a book about DuckDB because it feels like everything has happened so quickly. The world of data management systems moves slowly—software projects started in the ’70s are still in a strong position on the market.

It has only been a few short years since an evening in 2018 when we sat in the “Joost” bar in Amsterdam, and decided that we were going to build a new system. We had been toying with the concept before but had been hesitant as we knew it was a daft idea. The common wisdom is that it takes “ten million dollars” to render a new database system successful. But there was an equally daft plan. We would create a unique new data management system never built before: An in-process analytical system. Maybe the usual rules did not apply to this new kind of system. After some more beers, we more or less decided on the first rough draft of DuckDB’s architecture. The very next day, we started hacking.

Only a year later, in 2019, we opened our repository and started telling people. We showed a first demo of DuckDB at the 2019 SIGMOD conference, coincidentally in Amsterdam. Since we co-organized the conference, we snuck stickers in the goodie bags in an early attempt of somewhat viral marketing. At the same time, we also opened up the source code repository to the public. The duck was out of the bag, as the saying goes.

But there are thousands of open-source projects started every day, and the vast majority will—regrettably or not—never gain any traction. That was what we expected, too; most likely, nobody will care about our “DuckDB.” But an amazing thing happened: gradually, the stars on the GitHub repository started accumulating. We think it is because of another design goal of DuckDB: ease of use. We observed that the prevailing sentiment in data systems seemed to have been that the world should be grateful to be allowed to use the hard-won results of database systems research and the systems we build. However, we also observed a worrying effect. The results of decades of research were ignored, simply because they were hard to use. In somewhat of a paradigm shift for data systems, one design goal of DuckDB was to make it as easy to use as possible and to fix some of the biggest gripes we had heard from practitioners.

Somehow, people seem to have noticed. Considerable popularity gains came from activity on the social network formerly known as Twitter and, most notably, from regularly being featured on the Hacker News. Today, DuckDB has millions of downloads each month and is used

everywhere—from the largest companies to the smallest embedded devices. MotherDuck offers a hosted version but DuckDB style with a strong local component. Heck, people are even writing books about DuckDB.

We're glad that Mark and the two Michaels are the ones who bring this book to you. It's an honor for us that such an excellent team is writing this book. They are experts in explaining challenging data technology to developers in a fun, engaging, but still deeply competent way. We hope you will enjoy this book and, of course, also hope you will enjoy working with DuckDB.

—MARK RAASVELDT AND HANNES MÜHLEISEN
CREATORS OF DUCKDB

brief contents

1
An introduction to DuckDB

This chapter covers

- Why DuckDB, a single node in-memory database, emerged in the era of big data
- DuckDB's capabilities
- How DuckDB works and fits into your data pipeline

We're excited that you've picked up this book and are ready to learn about a technology that seems to go against the grain of everything that we've learned about big data systems over the last decade. We've had a lot of fun using DuckDB and we hope you will be as enthused as we are after reading this book. This book's approach to teaching is hands-on, concise, fast-paced, and will include lots of code examples.

After reading the book you should be able to use DuckDB to analyze tabular data in a variety of formats. You will also have a new handy tool in your toolbox for data transformation, cleanup and conversion. You can integrate it into your Python notebooks and processes to replace pandas DataFrames in situations where they are not performing. You will be able to build quick applications for data analysis using Streamlit with DuckDB.

Let's get started!

1.1 What is DuckDB?

DuckDB is a modern embedded analytics database that runs on your machine and lets you efficiently process and query gigabytes of data from different sources.

It was created in 2018 by Mark Raasveldt and Hannes Mühleisen who, at the time, were researchers in database systems at Centrum Wiskunde & Informatica (CWI) - the national research institute for mathematics and computer science in the Netherlands and their advisor Peter Boncz.

The founders and the CWI spun DuckDB Labs off as a startup to further develop DuckDB. Its engineering team focuses on making DuckDB more efficient, user-friendly, and better integrated.

The non-profit DuckDB Foundation governs the DuckDB Project by safeguarding the intellectual property and ensuring the continuity of the open-source project under the MIT license. The foundations operations and DuckDB's development are supported by commercial members, while association members can inform the development roadmap.

While DuckDB focuses on the local processing of data, another startup, MotherDuck, aims to extend DuckDB to a distributed, self-serve analytics system that can process data in the cloud and on the edge. It adds collaboration and sharing capabilities to DuckDB, and supports processing data from all kinds of cloud storage.

The DuckDB ecosystem is really broad, many people and organizations are excited about the possibilities and create integrations and generally useable applications.

The DuckDB community is very helpful and friendly, you can find them on Discord and GitHub. The documentation is comprehensive and detailed enough to answer most questions.

DuckDB lets you process and join local or remote files in different formats, including CSV, JSON, Parquet, and Arrow, as well as databases like MySQL, SQLite, and Postgres. You can even query pandas or Polars DataFrames from your Python scripts or Jupyter notebooks. A diagram showing how DuckDB is typically used is shown in figure 1.1.

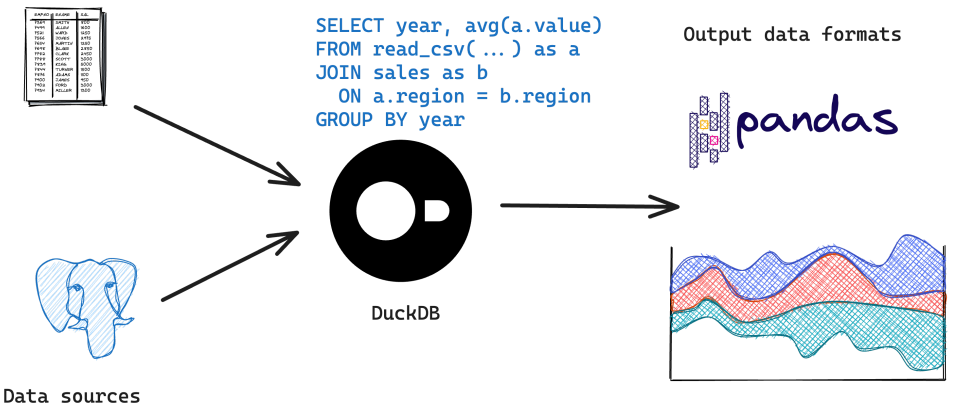


Figure 1.1 DuckDB and other tools in the ecosystem

Unlike the pandas and Polars libraries, DuckDB is a real analytics database, implementing efficient data processing mechanisms that can handle large volumes of data in seconds. With its SQL dialect, even complex queries can be expressed more succinctly. It allows you to handle more operations inside the database avoiding costly roundtrips to your client.

The architecture of the core database engine is the basis for efficient processing and memory management. You can see a diagram showing the way that a query is processed in figure 1.2.

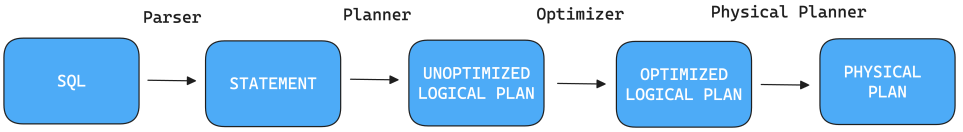


Figure 1.2 A high-level overview of DuckDB's architecture

We can see that DuckDB processes queries the same way as other databases, with a SQL parser, query execution planner, and query runtime. The query engine is vectorized, which means it processes chunks of data in parallel and benefits from modern multi-core CPU architectures. DuckDB supports several extensions, as well as user-defined functions, and has a variety of user interfaces, including a CLI, API, or lower-level integration into other systems.

1.2 Why should you care about DuckDB?

DuckDB makes data analytics fast and fun again, without the need to set up large Apache Spark clusters or run a cloud data warehouse just to process a few hundred gigabytes of data. Accessing data from many different sources directly, and running the processing where the data resides without copying it over the wire, makes your work faster, simpler and cheaper. This not only saves time but also a lot of money, as well as reducing frustration.

For example, we recently had to process AWS access log files residing in S3. Usually, we would run AWS Athena SQL queries against the compressed JSON files. This tends to get expensive as the biggest part of the cost is based on the amount of data scanned by the analytics service. Now we can instead deploy DuckDB to an EC2 VM and query the files in process for a fraction of the cost.

With DuckDB you can run lots of experiments and validate your ideas and hypotheses quickly and locally, and all of this using just SQL. As well as supporting the ANSI SQL standard, DuckDB's SQL dialect also includes innovations like:

- Simplifying `SELECT *` queries with `SELECT * EXCLUDE()` and `SELECT * REPLACE()`
- Ordering by and grouping results by `ALL` columns, e.g. `GROUP BY ALL` saves the user typing out all field names.
- Using `PIVOT` and `UNPIVOT` to transpose rows and columns

- The `STRUCT` data type and associated functions, which make it easy to work with complex nested data.

We are excited about DuckDB, because it helps to simplify data pipelines and data preparation, allowing more time for the actual analysis, exploration and experimentation.

In this book, we hope to convince you of the following:

- It is faster than SQLite for analytical workloads.
- It is easier to set up than a Spark cluster.
- It has lower resource requirements than pandas.
- It doesn't throw weird Rust errors like Polars.
- It is easier to set up and use than Postgres, Redshift and other relational databases.
- It is faster and more powerful for data transformations than Talend

1.3 When should you use DuckDB?

You can use DuckDB for all analytics tasks that can be expressed in SQL and work on structured data (i.e. tables or documents) as long as your data is already available (not streaming) and data volumes don't exceed a few hundred Gigabytes. DuckDB can process a variety of data formats as outlined before and can be extended to integrate with other systems.

As the data doesn't leave your system (local or privacy-guaranteed hosting), it's also great for analyzing privacy-related data like health information, home automation data, patient data, personal identifying information, financial statements and similar datasets.

Here are some examples of some common analysis tasks that DuckDB is well-placed to solve:

- Analyzing log files from where they are stored, without needing to copy them to new locations.
- Quantifying personal medical data about one's self, such as a runner might do when monitoring heart rates.
- Reporting on the power generation and consumption using data from smart meters.
- Optimizing ride data from modern transport operations for bikes and cars.
- Pre-processing and -cleaning of user-generated data for machine learning training.

A great use of DuckDB is for more efficiently processing data that is already available in pandas or Polars DataFrames because it can access the data in-process without having to copy the data from the DataFrame memory representation.

The same is true for outputs and tables generated by DuckDB. These can be used as DataFrames without additional memory usage or transfer.

1.4 When should you not use DuckDB?

As DuckDB is an analytics database, it has only minimal support for transactions and parallel write access. You therefore couldn't use it in applications and APIs that process and store input data arriving arbitrarily. Similarly when multiple concurrent processes read from a writeable database.

The data volumes that you can process with DuckDB are mostly limited by the main memory of your computer. While it supports spilling over memory (out-of-memory processing) to disk, that feature is more aimed at exceptional situations where the last few percent of processing don't fit into memory. In most cases, that means you'll have a limit of a few hundred gigabytes for processing, not all of which needs to be in memory at the same time, as DuckDB optimizes loading only what's needed.

DuckDB focuses on the long tail of data analytics use cases, so if you're in an enterprise environment with a complex setup, processing many Terabytes of data, DuckDB might not be the right choice for you.

DuckDB does not support processing live data streams that update continuously. Data updates should happen in bulk by loading new tables or large chunks of new data at once. DuckDB is not a streaming real-time database, you would have to implement a batching approach yourself by setting up a process to create mini-batches of data from the stream and store those mini-batches somewhere that could then be queried by DuckDB.

1.5 Use cases

There are many use cases for a tool like DuckDB. Of course, the most exciting is when it can be integrated with existing cloud, mobile, desktop and command line applications and do its job behind the scenes. In these cases, it would be the equivalent of the broad usage of SQLite today, only for analytical processing instead of transactional data storage. When analyzing data that shouldn't leave the user's device, such as health, training, financial or home automation data, an efficient local infrastructure comes in handy. The local analytics and pre-processing also reduces the volume of data that has to be transported from edge-devices like smart meters or sensors.

DuckDB is also useful for fast analysis of larger datasets, such as log files, where computation and reduction can be done where the data is stored, saving high data transfer time and costs. Currently, cloud vendors offer expensive analytics services like BigQuery, Redshift and Athena to process this kind of data. In the future, you can replace many of those uses with scheduled cloud functions processing the data with DuckDB. You can also chain those processing functions by writing out intermediate results to cloud storage which can then also be used for auditing.

For Data Scientists, data preparation, analysis, filtering and aggregation can be done more efficiently than with pandas or other DataFrame libraries, by leveraging DuckDB's state-of-the-art query engine. And all of this without leaving the comfortable environment of a notebook with Python or R APIs. This will put more advanced data analytics capabilities in the hands of data science users so that they can make better use of larger data volumes while being faster and more efficient. We will show several of these later in the book. Also, the complexity of setup can be greatly reduced, removing the need to involve a data operations group.

A final exciting use case will be the distributed analysis of data between cloud storage, edge network, and local device. This is for instance currently being worked on by MotherDuck which allows you to run DuckDB combined in the cloud and locally.

1.6 Where does DuckDB fit in?

This book assumes that you have some existing data that you want to analyze or transform. That data can reside in flat files like CSV, Parquet, or JSON, or another database system, like Postgres or SQLite.

Depending on your use case, you can use DuckDB transiently to transform, filter and pass the data through to another format. In most cases, though, you will create tables for your data to persist it for subsequent, high-performance analysis. When doing that, you can also transform and correct column names, data types and values. If your input data is nested documents, you can unnest and flatten the data to make relational data analysis easier and more efficient.

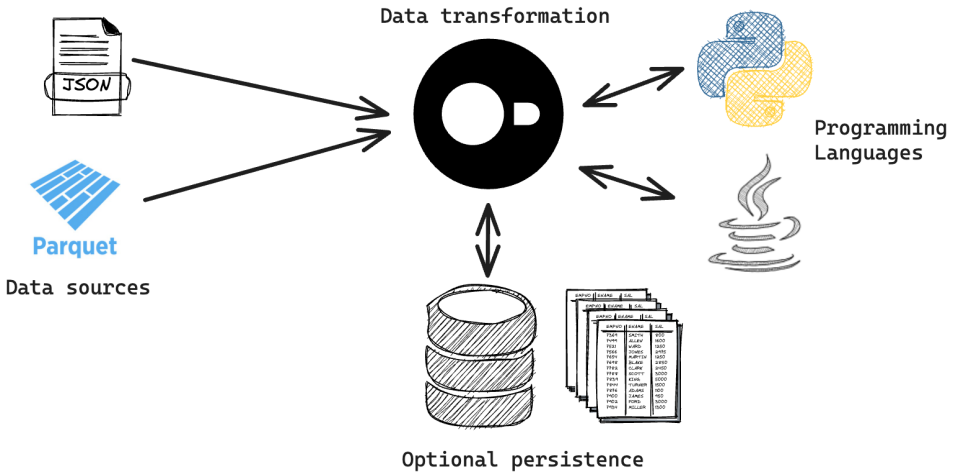


Figure 1.3 Using DuckDB in a data pipeline.

In the next step, you need to determine which SQL capabilities or DuckDB features can help you to perform that analysis or transformation. You can also do Exploratory Data Analysis (EDA) to quickly get an overview of the distribution, ranges and relationships in your data.

After getting acquainted with the data, you can proceed to the actual analytics tasks. Here you will build the relevant SQL statements incrementally, verifying at each step that the sample of the results produced matches your expectations. At this stage, you might create additional tables or views, before using advanced SQL features like window functions, common table expressions, and pivots. Finally, you need to decide which way the results are consumed, either by turning them into files or databases again or serving them to users through an application, API, or by visualizing them in a Jupyter notebook or dashboard.

1.7 Steps of the data processing flow

In the following sections, we will describe some specific aspects of DuckDB's architecture and feature set at a high level to give you an overall understanding and appreciation. We have ordered the sections in the sequence of how you would use DuckDB, from loading data to populating tables and writing SQL for analysis to visualizing those results, as shown in figure 1.4.

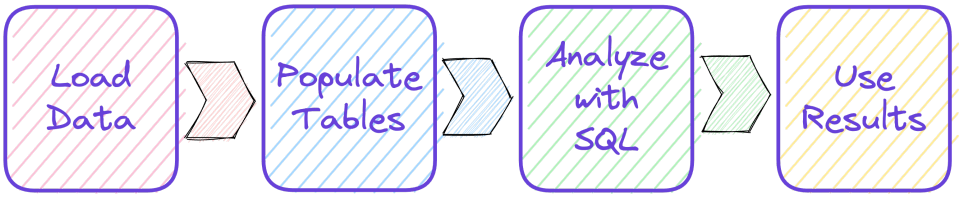


Figure 1.4 The data processing flow

1.7.1 Data Formats and Sources

DuckDB supports a large number of data formats and data sources and it lets you inspect and analyze their data with little ceremony. Unlike other data systems, like SQL Server, you don't need to first specify schema details upfront. When reading data the database uses sensible defaults and inherent schema information from the data, that you can override when needed.

> **NOTE** With DuckDB you can focus more on the data processing and analysis that you need to do and not so much on upfront data-engineering. Being an open-source project built by practitioners, there is a lot of emphasis on usability—if something is too hard to use, someone in the community will propose and submit a fix. And if the built-in functionality reaches not far enough, there's probably an extension that addresses your needs (e.g. geospatial data or full-text search).

DuckDB supports a variety of data formats:

- CSV files can be loaded in bulk and parallel and their columns are automatically mapped.
- DataFrames' memory can be handled directly by DuckDB inside the same Python process without the need to copy data
- JSON or JSONLines formats can be destructured, flattened, and transformed into relational tables. DuckDB also has a JSON type for storing this type of data.
- Parquet files along with their schema metadata can be queried. Predicates used in queries are pushed down and evaluated at the Parquet storage layer to reduce the amount of data loaded. This is the ideal columnar format to read and write for data lakes.
- Apache Arrow columnar shaped data via Arrow Database Connectivity (ADBC) without data copying and transformations
- Accessing data in cloud buckets like S3 or GCP reduces transfer and copy infrastructure and allows for cheap processing of large data volumes.

1.7.2 Data structures

DuckDB handles a variety of tables, views, and data types. For table columns, processing and results, there are more data types available than just the traditional data types like string (varchar), numeric (integer, float, decimal), dates, timestamps, intervals, boolean, and blobs (Binary Large Objects).

DuckDB also supports structured data types like enums, lists, maps (dictionaries) and structs.

- Enums are indexed, named elements of a set, that can be stored and processed efficiently.
- Lists or arrays hold multiple elements of the same type and there are a variety of functions for operating on these lists.
- Maps are efficient key-value pairs that can be used for keeping keyed data points. They are used during JSON processing and can be constructed and accessed in several ways.
- Structs are consistent key-value structures, where the same key always has values of the same data type. That allows for more efficient storage, reasoning and processing of structs.

DuckDB also allows you to create your own types and extensions can provide additional data types as well. DuckDB can also create virtual or derived columns that are created from other data via expressions.

1.7.3 Develop the SQL

When analyzing data, you usually start by gaining an understanding of the shape of the data. Then you work from simple queries to creating more and more complex ones from the basic building blocks. You can use `DESCRIBE` to learn about the columns and data types of your data sources, tables and views. Armed with that information you can get basic statistics and distributions of a dataset by running count-queries `count(*)` globally or grouped by interesting dimensions like time, location, or item type. That already gives you some good insights on what to expect from the data available.

DuckDB even has a `SUMMARIZE` clause that gives you statistics per column:

- count
- min, max, avg, std (deviation)
- approx_unique (estimated count of distinct values)
- percentiles (q25, q50, q75)
- null_percentage (part of the data being null)

To write your analytics query you can start working on a subset of the data by using `LIMIT` or only looking at a single input file. Start by outlining the result columns that you need (these may sometimes be converted, e.g. for dates using `strptime`). Those are the columns you would group by. Then apply aggregations and filters to your data as needed. There are many different aggregation functions available in DuckDB, from traditional ones like `min`, `avg`, `sum` to more advanced ones like `histogram`, `bitstring_agg`, `list` or approximations like `approx_count_distinct`. There are also advanced aggregations, including percentiles, entropy or regression computation, and skewness. For running totals and comparisons with previous and next rows, you would use window functions `aggregation OVER (PARTITION BY column ORDER BY column2 [RANGE …])`. Repeatedly used parts of your analytics statement can be extracted into named common table expressions (CTEs) or views. Often, it also helps for readability to move parts of the computation into subqueries and use their results to check for existence or do some nested data preparation.

While you're building up your analytical statement, you can check the results at any time to make sure they are still correct and you've not taken an incorrect detour. This takes us to our next and last section on using the results of your queries.

1.7.4 Use or process the results

You've written your statement and got the analytics results quickly from DuckDB. Now what?

It would be useful to keep your results around, e.g. by storing them in a file or a table. Creating a table from your results is straightforward with `CREATE TABLE <name> AS SELECT …`. DuckDB can output a variety of formats, including CSV, JSON, Parquet, Excel, and Arrow. It also supports other database formats like SQLite, Postgres and others via custom extensions. For smaller results sets, you can also use the DuckDB CLI to output the data as CSV or JSON.

But because a picture tells more than 1000 rows, often the preferred choice is data visualization. With the built-in `bar` function you can render inline bar charts of your data. You could also use command-line plotting tools like `youplot` for some quick results in your terminal.

In most cases though, you would use the large Python and Javascript ecosystem to visualize your results. For those purposes, you can turn your results into DataFrames, which then can be rendered into a variety of charts with `matplotlib`, `ggplot` in Python, `ggplot2` in R, or `d3`, `nivo`, `observable` in Javascript. A visual representation showing this is in figure 1.5.

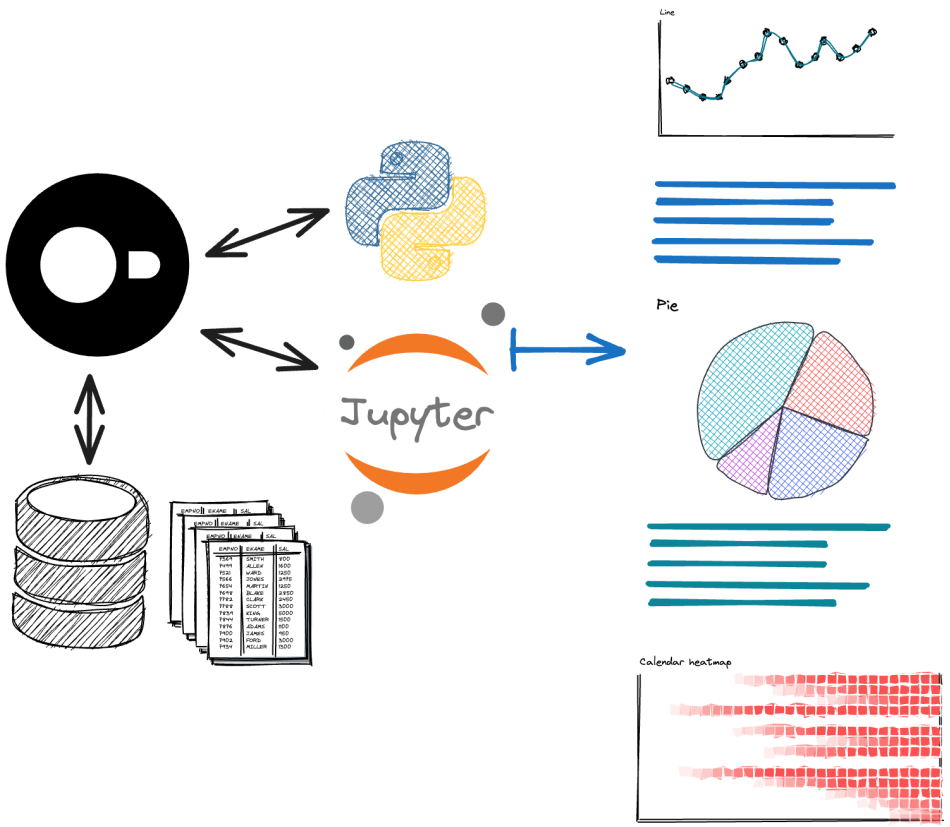


Figure 1.5 Visualizing data in a dashboard or Jupyter Notebook

As DuckDB is so fast, you can serve the results directly from your queries to an API endpoint that you or others can use downstream or integrate it into an app for processing like Streamlit. You only need a server setup, if your source data is too big to move around and your results comparatively small (much less than 1% of the volume). Otherwise, you can just embed DuckDB into your application and have it run on local raw data or a local DuckDB database.

1.8 Summary

- DuckDB is a newly developed analytical database that excels at in-memory processing.
- The database supports an extended dialect of SQL and gains new capabilities with extensions.
- DuckDB can read a variety of formats natively from local and remote sources.
- The integration in Python, R and other languages is seamless and efficient.
- As an in-process database it can process data efficiently without copying.
- In addition to the traditional data types, DuckDB also supports lists, maps, structs, and enums.
- DuckDB provides a lot of functions on data types and values, making data processing and shaping much easier.
- Building up your SQL queries step by step after learning about the shape of your data helps to stay in control.
- You can use the results of your query in a variety of ways, from generating reports and visualizing in charts to outputting in new formats.

2
Getting Started with DuckDB

This chapter covers

- Installing and learning how to use the DuckDB CLI
- Executing commands in the DuckDB CLI
- Querying remote files

Now that we've got an understanding of what DuckDB is and why it's come into prominence in the early 2020s, it's time to get familiar with it. This chapter will be centered around the DuckDB CLI. We'll learn how to install it on various environments, before learning about the in-built commands. We'll conclude by querying a remote CSV file.

2.1 Supported environments

DuckDB is available for a range of different programming languages and operating systems (Linux, Windows, macOS) both for Intel/AMD and ARM architectures. At the time of writing, there is support for the command line, Python, R, Java, Javascript, Go, Rust, Node.js, Julia, C/C++, ODBC, JDBC, WASM, and Swift. In this chapter, we will focus on the DuckDB command line exclusively, as we think that is the easiest way to get you up to speed.

The DuckDB CLI does not require a separate server installation as DuckDB is an embedded database and in the case of the CLI, it is embedded in exactly that.

The command line tool is published to GitHub releases and there are a variety of different packages for different operating systems and architectures. You can find the full list on the installation page.

2.2 Installing the DuckDB CLI

The installation is a "copy to" installation, no installers or libraries are needed. The CLI consists of a single binary named `duckdb`. Let's learn how to go about installing DuckDB.

2.2.1 macOS

On macOS the official recommendation is to use Homebrew:

Listing 2.1 Install DuckDB on macOS via Homebrew

```
# This is only necessary to install Homebrew itself,
➥ don't run if you already have it
# /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/
➥Homebrew/install/HEAD/install.sh)"

brew install duckdb
```

2.2.2 Linux and Windows

There are many different packages available for Linux and Windows, depending on the particular architecture and version that you're using. You can find a full listing on the GitHub releases page.

In listing 2.2, we learn how to do zero-process installation on Linux with an AMD64 architecture:

Listing 2.2 Zero-process installation on Linux

```
wget https://github.com/duckdb/duckdb/releases/download/v0.8.1/
➥duckdb_cli-linux-amd64.zip # #1
unzip duckdb_cli-linux-amd64.zip
./duckdb -version
```

#1 Don't forge to update this link to the latest version from the GitHub releases page.

2.3 Using the DuckDB CLI

The simplest way to launch the CLI is shown below, and yes, it's that little, and it's quick:

```
duckdb
```

This will launch DuckDB and the CLI. You should see something like the following output:

```
v0.8.1 6536a77232
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
```

The database will be transient, with all data held in memory. It will disappear when you quit the CLI, which you can do by typing `.quit` or `.exit`.

2.3.1 Dot commands

In addition to SQL statements and commands, the CLI has several special commands that are only available in the CLI, the special dot commands. To use one of these commands, begin the line with a period (.) immediately followed by the name of the command you wish to execute. Additional arguments to the command are entered, space separated, after the command. Dot commands must be entered on a single line, and no whitespace may occur before the period. No semicolon is required at the end of the line in contrast to a normal SQL statement or command.

Some of the most popular dot commands are described below:

- `.open` closes the current database file and opens a new one.
- `.read` allows reading SQL files to execute from within the CLI.
- `.tables` lists the currently available tables and views.
- `.timer on/off` toggles SQL timing output.
- `.mode` controls output formats.
- `.maxrows` controls the number of rows to show by default (for `duckbox` format).
- `.excel` shows the output of next command in spreadsheet.
- `.quit` or `ctrl-d` exit the CLI.

A full overview can be retrieved via `.help`.

2.3.2 CLI arguments

The CLI takes in arguments that can be used to adjust the database mode, control the output format, or decide whether the CLI is going to enter interactive mode. The usage is `duckdb [OPTIONS] FILENAME [SQL]`.

Some of the most popular CLI arguments are described below:

- `-readonly` opens the database in read-only mode.
- `-json` sets the output mode to `json`.
- `-line` sets the output mode to `line`.
- `-unsigned` allows for the loading of unsigned extensions.

- `-s COMMAND` or `-c COMMAND` runs the provided command and then exits. This is especially helpful when combined with the `.read` dot command, which reads input from the given file name.

To get a list of the available CLI arguments, call `duckdb -help`.

2.4 DuckDB's extension system

DuckDB has an extension system that is used to house functionality that isn't part of the core of the database. You can think of extensions as packages that you can install with DuckDB.

DuckDB comes pre-loaded with several extensions, which vary depending on the distribution that you're using. You can get a list of all the available extensions, whether installed or not, by calling the `duckdb_extensions` function. Let's start by checking the fields returned by this function:

```
DESCRIBE
SELECT *
FROM duckdb_extensions();
```

A truncated view of the output is shown in listing 2.3:

Listing 2.3 Fields returned by `duckdb_extensions`

```
┌─────────────────┬─────────────┐
│   column_name   │ column_type │
│     varchar     │   varchar   │
├─────────────────┼─────────────┤
│ extension_name  │ VARCHAR     │
│ loaded          │ BOOLEAN     │
│ installed       │ BOOLEAN     │
│ install_path    │ VARCHAR     │
│ description     │ VARCHAR     │
│ aliases         │ VARCHAR[]   │
└─────────────────┴─────────────┘
```

Let's check which extensions we have installed on our machine:

```
SELECT extension_name, loaded, installed
from duckdb_extensions()
ORDER BY installed DESC, loaded DESC;
```

The results of running the query are shown in listing 2.4:

Listing 2.4 A list of DuckDB's extensions

```
┌──────────────────┬─────────┬─────────────┐
│  extension_name  │ loaded  │ installed   │
│     varchar      │ boolean │  boolean    │
├──────────────────┼─────────┼─────────────┤
│ autocomplete     │ true    │ true        │
│ fts              │ true    │ true        │
│ icu              │ true    │ true        │
│ json             │ true    │ true        │
│ parquet          │ true    │ true        │
│ tpch             │ true    │ true        │
│ httpfs           │ false   │ false       │
│ inet             │ false   │ false       │
│ jemalloc         │ false   │ false       │
│ motherduck       │ false   │ false       │
│ postgres_scanner │ false   │ false       │
│ spatial          │ false   │ false       │
│ sqlite_scanner   │ false   │ false       │
│ tpcds            │ false   │ false       │
│ excel            │ true    │             │
├──────────────────┴─────────┴─────────────┤
│ 15 rows                    3 columns    │
└─────────────────────────────────────────┘
```

You can install any extension by typing the `INSTALL` command followed by the extension's name. The extension will then be installed in your database, but not loaded. To load an extension, type `LOAD` followed by the same name. The extension mechanism is idempotent, meaning you can issue both commands several times without running into errors.

> **NOTE** Since version 0.8 of DuckDB, the database will auto-load extensions that are installed, if it can determine that they are needed, so you might not need the `LOAD` command.

By default, DuckDB cannot query files that live elsewhere on the internet, but that capability is available via the official `httpfs` extension. If it is not already in your distribution, you can install and load the `httpfs` extension. This extension lets us directly query files hosted on an HTTP(S) server without having to download the files locally, it also supports S3 and some other cloud storage providers.

```
INSTALL httpfs;
LOAD httpfs;
```

We can then check where that's been installed by entering:

```
FROM duckdb_extensions()
SELECT loaded, installed, install_path
WHERE extension_name = 'httpfs';
```

You should see the following output:

```
┌─────────┬───────────┬──────────────────────────────────┐
│ loaded  │ installed │           install_path           │
│ boolean │  boolean  │              varchar             │
├─────────┼───────────┼──────────────────────────────────┤
│ true    │ true      │ /path/to/httpfs.duckdb_extension │
└─────────┴───────────┴──────────────────────────────────┘
```

We can see that this extension has now been loaded and installed, as well as the location where it's been installed.

2.5 Analyzing a CSV file with the DuckDB CLI

We're going to start with a demonstration of the CLI for a common task for any data engineer — making sense of the data in a CSV file! It doesn't matter where our data is stored, be it on a remote HTTP server or cloud storage (S3, GCP, HDFS), DuckDB can process it now directly without having to do a manual download and import process. As the ingestion of many supported file formats, such as CSV and Parquet, is parallelized by default, it should be super quick to get your data into DuckDB.

We went looking for CSV files on GitHub and came across a dataset that contains population numbers of countries. We can write the following query to count the number of records:

```
SELECT count(*)
FROM 'https://github.com/bnokoro/Data-Science/raw/master/
➥countries%20of%20the%20world.csv';
```

If we run this query, we should see the following output, indicating that we've got population data for over 200 countries:

```
┌──────────────┐
│ count_star() │
│    int64     │
├──────────────┤
│          227 │
└──────────────┘
```

If, as is the case here, our URL or file name ends in a specific extension (e.g. `.csv`), DuckDB will automatically process it. But what if we try to automatically process a short link of that same CSV file?

```
SELECT count(*)
FROM 'https://bit.ly/3KoiZR0';
```

Running this query results in the following error:

```
Error: Catalog Error: Table with name https://bit.ly/3KoiZR0 does not exist!
Did you mean "Player"?
LINE 1: select count(*) from 'https://bit.ly/3KoiZR0';
```

Although it's a CSV file, DuckDB doesn't know that because it doesn't have a `.csv` suffix. We can solve this problem by using the `read_csv_auto` function, which processes the provided URI as if it was a CSV file, despite its lack of `.csv` suffix. The updated query is shown in listing 2.5:

Listing 2.5 Specifying the format of a remote file

```
SELECT count(*)
FROM read_csv_auto("https://bit.ly/3KoiZR0");
```

This query will return the same result as the query that used the canonical link from which the format could be deduced.

2.5.1 Result modes

For displaying the results, you can choose between different modes using `.mode <name>`. You can see a list of available modes by typing `.help mode`.

Throughout this chapter, we've been using 'duckbox' mode, which returns a flexible table structure. DuckDB comes with a series of different modes, which broadly fit into a couple of categories:

- Table based, which work well with fewer columns — `duckbox`, `box`, `csv`, `ascii`, `table`, `list`, `column`

- Line based, which work well with more columns — `json`, `jsonline`, `line`

There are then some others that don't fit into those categories, including `html`, `insert`, and `trash` (no output).

Our first query counted the number of records in the CSV file, but it'd be interesting to know what columns it has. There are a lot of columns that will get truncated if we use the default mode, so we're going to change to the `line` mode before running the query:

```
.mode line -- #1
SELECT *
FROM read_csv_auto("https://bit.ly/3KoiZR0")
LIMIT 1;
```

#1 Changing to line mode

The results of running this query are shown in listing 2.6.

Listing 2.6 A result in `line` mode

```
                         Country = Afghanistan
                          Region = ASIA (EX. NEAR EAST)
                      Population = 31056997
                  Area (sq. mi.) = 647500
       Pop. Density (per sq. mi.) = 48,0
     Coastline (coast/area ratio) = 0,00
                    Net migration = 23,06
Infant mortality (per 1000 births) = 163,07
               GDP ($ per capita) = 700
                     Literacy (%) = 36,0
                Phones (per 1000) = 3,2
                       Arable (%) = 12,13
                        Crops (%) = 0,22
                        Other (%) = 87,65
                          Climate = 1
                        Birthrate = 46,6
                        Deathrate = 20,34
                      Agriculture = 0,38
                         Industry = 0,24
                          Service = 0,38
```

As you can see from the output, `line` mode takes up a lot more space than `duckbox`, but we've found it to be the best mode for doing initial exploration of datasets that have plenty of columns. You can always change back to another mode once you've decided a subset of columns that you'd like to use.

The dataset has lots of interesting information about various countries. Let's write a query to count the number of countries, and find the maximum population average area across all countries. This query only returns a few columns, so we'll switch back to `duckbox` mode before running the query:

```
.mode duckbox
SELECT count(*) AS countries,
       max(Population) AS max_population,
       round(avg(cast("Area (sq. mi.)" AS decimal))) AS avgArea
FROM read_csv_auto("https://bit.ly/3KoiZR0");
```

```
┌───────────┬────────────────┬──────────┐
│ countries │ max_population │ avgArea  │
│   int64   │     int64      │  double  │
├───────────┼────────────────┼──────────┤
│       227 │     1313973713 │ 598227.0 │
└───────────┴────────────────┴──────────┘
```

So far, no tables have been created in the process, and we've just touched the tip of the iceberg demonstrating what DuckDB actually can do. While the examples above have all been run in interactive mode, the DuckDB CLI can also run in a non-interactive fashion. It can read from standard input and write to standard output. This makes it possible to build all sorts of pipelines.

Let's conclude with a script that extracts the populate, birth rate, and death rate in countries in Western Europe and creates a new local CSV file containing that data. We can either `.exit` from the DuckDB CLI or open another tab before running the command below:

```
duckdb -csv \
 -s "SELECT Country, Population, Birthrate, Deathrate
     FROM read_csv_auto('https://bit.ly/3KoiZR0')
     WHERE trim(region) = 'WESTERN EUROPE'" \
 > western_europe.csv
```

The first few lines of `western_europe.csv` can be viewed with a command line tool or text editor. If we use the `head` tool, we could find the first 5 lines like this:

```
head -n5 western_europe.csv
```

And the output would look like table 2.1:

Table 2.1 First five lines of western_europe.csv showing population, birth rate, and death rate of some countries in Western Europe

Country	Population	Birthrate	Deathrate
Andorra	71201	8,71	6,25
Austria	8192880	8,74	9,76
Belgium	10379067	10,38	10,27
Denmark	5450661	11,13	10,36

We can also create Parquet files, but for that we can't pipe the output straight into a file with a Parquet extension. Instead, we can use the `COPY … TO` clause with stdout as the destination:

Listing 2.7 Writing explicitly to standard out so that options can be specified

```
duckdb \
-s "COPY (
      SELECT Country, Population, Birthrate, Deathrate
      FROM read_csv_auto('https://bit.ly/3KoiZR0')
      WHERE trim(region) = 'WESTERN EUROPE'
    ) TO '/dev/stdout' (FORMAT PARQUET)" \
> western_europe.parquet
```

You could then view the contents of the Parquet file using any Parquet reader, perhaps even DuckDB itself!

```
duckdb -s "FROM 'western_europe.parquet' LIMIT 5"
```

The results will be the same as seen in table 2.1.

TIP Repeated configuration and usage can be stored in a config file that lives at `$HOME/.duckdbrc`. This file is read during startup and all commands in it - both dot commands and SQL commands are executed via one `.read` command. This allows you to store both the configuration state of the CLI and anything you might want to initialize with SQL commands.

An example of something that might go in the duckdbrc file is a custom prompt and welcome message when you launch DuckDB:

```
-- Duck head prompt
.prompt '🦆 '
-- Example SQL statement
select 'Begin quacking!' as "Ready, Set, ...";
```

2.6 Summary

- The DuckDB Database with command-line interface (CLI) can be installed as CLI on Windows, Linux and OSX.
- DuckDB is available as a library for Python, R, Java, Javascript, Julia, C/C++, ODBC, WASM, and Swift.
- The CLI supports additional dot commands for controlling outputs, reading files, built-in help, and more.
- With `.mode` you can use several display modes, including `duckbox`, `line`, and `ascii`.
- You can query CSV files directly from an HTTP server by installing the `https` extension.
- You can use the CLI as a step in any data pipeline, without creating tables, by querying external datasets and writing results to standard out or other files.

3 Executing SQL Queries

This chapter covers

- The different categories of SQL statements and their fundamental structure
- Creating tables and structures for ingesting a real world dataset
- Laying the fundamentals for analyzing a huge dataset in detail
- Exploring DuckDB-specific extensions to SQL

Now that you've learned about the DuckDB CLI, it's time to tickle your SQL brain. We will be using the CLI version of DuckDB throughout this chapter. However, all the examples here can be fully applied from within any of the supported environments, such as the Python client, the Java JDBC driver, or any of the other supported language interfaces.

In this chapter, we will quickly go over some basic and necessary SQL statements and then move on to more advanced querying. In addition to explaining SQL basics, we'll also be covering more complex topics like common table expressions and window functions. DuckDB supports both of these and this chapter will teach you how to build queries for doing the best possible in-memory Online Analytical Processing (OLAP) with DuckDB.

To get the sample up and running you should have an idea about data ingestion with DuckDB as shown in Chapter 2, especially how to ingest CSV files, and deal with implicit (automatic) or explicit column detection. Knowledge of the data types presented in Chapter 1 will also be helpful. If you want to go straight to querying data, please jump to section 3.4.3, in which we discuss SQLs `SELECT` statement in detail. We think it's better to start by defining tables and structures first, populating them with data, and then querying them, rather than making up queries on generated or non-existent data.

3.1 A quick SQL recap

SQL queries are composed of several statements which are in turn composed of clauses. A command is a query submitted to the CLI or any other of the supported clients. Commands in the DuckDB CLI are terminated with a semicolon. Whitespaces can be used freely in SQL commands. Either align your commands beautifully or type them all in one line, it doesn't matter. SQL is case-insensitive about keywords and identifiers.

Most statements support several clauses that change their behavior, most prominently the `WHERE`, `GROUP BY` and `ORDER BY` clauses. `WHERE` adds conditions on which rows are included in the final result, `GROUP BY` aggregates many values into buckets defined by one or more keys and `ORDER BY` specifies the order of results returned.

You will learn all relevant statements and clauses for your analytical workloads based on a concrete example. We picked energy production from photovoltaics as a relevant, real-world example.

3.2 Analyzing energy production

Energy consumption and production has been the subject of OLAP related analysis for a while. Smart meters measuring consumption in 15 minute intervals have been available to many industries—such as metal processing and large production plants—for some time now, and have become quite standard. These measurements are used to price the consumed energy, forecast consumption, and more.

With the rise of smart monitoring systems, detailed energy readings are now available in private households as well, becoming more mainstream each year. Imagine you have a photovoltaic grid and smart meter installed at your house. You want to be able to plan your electricity usage a bit or forecast an amortization of your grid, the same way large industries can. To do so, you don't have to go into a full time-series database and a live dashboard to do that. DuckDB and the examples we use throughout this chapter might give you a good starting point for creating your own useful reports.

The dataset that we are going to use in the following examples is available from the US Department of Energy under the name Photovoltaic Data Acquisition (PVDAQ). The dataset is documented in here. The National Renewable Energy Laboratory from the Department of Energy also offers a nice, simple API for getting the partitioned CSV / Parquet files via PVDAQ. Access is free and requires little personal information. The dataset is published under the Creative Commons Attribution License. Parts of the dataset are redistributed unchanged for ease of access in this chapter with the sources of this book.

NOTE Why are we storing measurements in 15-minute or quarterly hour intervals when modern sensors produce much finer measurements? 15 minutes turned out to be fit enough for the aforementioned purposes such as pricing and buying smart intervals while at the same time are "small" enough to be handled with ease in most relational systems these days. The power output or consumption is measured in Watt (W) or Kilowatt (kW) and billed or sold usually as Kilowatt-hour (kWh). 15-minute intervals of Watts are easily converted to kWh, while still accurate enough for good production charts. In most cases you want to smooth the values at least on an hourly basis, peaks and dips due to clouds are often irrelevant. If you look at a chart for forecasting, daily measurements can be a good base interval, as they will cover weekends and bank holidays and smooth out small irregularities.

3.2.1 Downloading the dataset

We will use the `httpfs` extensions of DuckDB to load the data without going through CSV files. To install it, run `install httpfs; load httpfs;` in your DuckDB CLI. We'll be working with the following data files:

- https://oedi-data-lake.s3.amazonaws.com/pvdaq/csv/systems.csv containing the list of all PV systems the PVDAQ measures.
- Readings for the systems 10, 34 and 1200 in the years 2019 and 2020, the URLs all follow the schema below (please change the `system_id` and the `year` URL parameters accordingly). You'll need an API key to access them, we are using `DEMO_KEY`.

The URLs for getting the data are described as followed, with the API key, system id, and year all supplied via query string parameters:

```
https://developer.nrel.gov/api/pvdaq/v3/data_file?api_key=DEMO_KEY
➥&system_id=34&year=2019
```

If you can't access those URLs for any reason, the source code of this book contains a database export under the name `ch03_db` containing the complete dataset. You can import it into a fresh database by using the following commands:

```
duckdb my_ch03.db
import database 'ch03_db'
```

TIP Another option is using a remote database on MotherDuck.com via `ATTACH 'md:_share/duckdb_in_action_ch3_4/d0c08584-1d33-491c-8db7-cf9c6910eceb'` in your DuckDB cli. While the shared example is read-only, it contains all data we used, and you can follow all examples that don't deal with insertion and the like. Chapter 12 will cover the services offered by MotherDuck in detail.

We picked this dataset for specific reasons: its domain is easy to grasp yet complex enough to introduce many analytical concepts backed with actual real-world needs. As with any analytical process, you will eventually run into inconsistent data. This is the case in some series in this dataset, too.

If you don't use the ready-made database, don't worry about the necessary queries for ingesting the raw data yet, we will get there in a bit. In the next sections we will discuss and create the database schema first, before we download the readings for a several PV systems.

3.2.2 The target schema

DuckDB is a relational database management system (RDBMS). That means it is a system for managing data stored in relations. A relation is essentially a mathematical term for a table.

Each table is a named collection of rows. Each row of a given table has the same set of named columns, and each column is of a specific data type. Tables themselves are stored inside schemas, and a collection of schemas constitutes the entire database that you can access.

NOTE What is a surrogate key? To address rows in a table a column with a unique value or a combination of columns that is unique over all rows is required. Such a column is usually referred to as primary key. Not all data that you possibly can store in a database has attributes that are unique. One awful choice as a unique or primary key for a person would be their name for example. In such scenario database schema designer often introduce numerical columns based on a monotonous increasing sequence or columns containing UUIDs (Universally Unique Identifier) as so-called "surrogate keys".

The schema for our dataset consists of a handful of tables. These tables are normalized so that the supported joins can easily be demonstrated. The three tables that we'll be working with are as follows:

- **systems** Contains the systems for which production values are read.
- **readings** Contains the actual readings taken for the systems.

- **prices** Contains the prices for selling energy. Prices are in Ct/kWh (European Cent per Kilowatt-hour), but the examples work in "some unit" per Kilowatt-hour, too.

A diagram describing these tables and their relationships to each other is shown in figure 3.1.

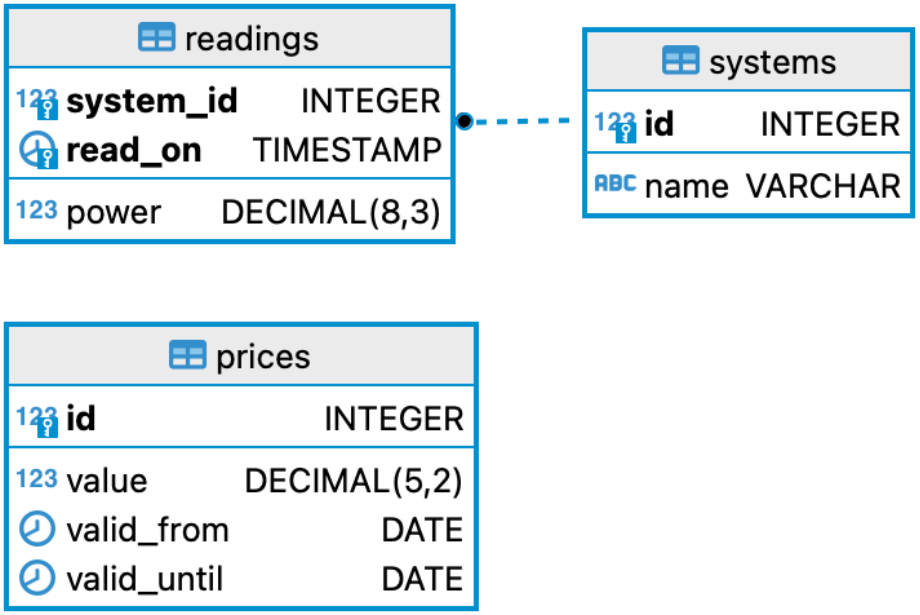


Figure 3.1 Energy consumption schema

Systems uses the id as defined in the CSV set. We treat it as an externally generated surrogate key. Prices uses a `SEQUENCE` and readings uses a concatenated, natural key (the id of the system they have been read from, plus the timestamp they have been read).

3.3 Data definition language (DDL) queries

We have already seen that you can query a lot of sources with DuckDB without creating a schema containing tables first. DuckDB however is a full-fledged RDBMS, and we will use DDL queries to create our target schema prior to ingesting our dataset. New tables are created with the `CREATE TABLE` statement, existing tables can be altered with the `ALTER TABLE` statement. If you don't need a table anymore, you will want to use `DROP TABLE`.

NOTE DuckDB supports the entire collection of data definition language clauses, but we only use a subset of them in this chapter for brevity's sake. Be sure to consult the statements documentation to see all the supported clauses.

3.3.1 The CREATE TABLE statement

Let's create the table for the systems we are going to monitor with the `CREATE TABLE` statement. You must specify the name of the table to create and the list of columns. Other options, such as modifiers to the whole statement, are optional. The column list is defined by the name of the column followed by a type and optional column constraints.

Listing 3.1 A basic `CREATE TABLE` statement

```
CREATE TABLE IF NOT EXISTS systems ( -- #1
    id          INTEGER PRIMARY KEY, -- #2
    name        VARCHAR(128) NOT NULL -- #3
);
```

#1 `IF NOT EXISTS` is an optional clause that makes the whole command idempotent, hence not failing if the table already exists

#2 `PRIMARY KEY` makes this column a non-optional column that serves as a primary and therefore unique key. An index will also be added.

#3 This modifier makes the column a mandatory column (you cannot insert literal `NULL` values.)

NOTE DuckDB also offers a `CREATE OR REPLACE TABLE` statement. This will drop an existing table and replace it with the new definition. We prefer the `IF NOT EXISTS` clause though as we think it's safer than unconditionally dropping a table: Any potential data will be gone afterwards.

The definition of the `readings` table looks slightly different. The table uses a composite primary key. This is a key composed of the reference column `system_id`, which points back to the systems table and the timestamp column containing the date and time the value was read. Such a primary key constraint cannot be directly defined with one of the columns but goes outside the column list.

Listing 3.2 Creating the `readings` table with an idempotent statement

```
CREATE TABLE IF NOT EXISTS readings (
    system_id   INTEGER NOT NULL,
    read_on     TIMESTAMP NOT NULL,
    power       DECIMAL(10,3) NOT NULL
            DEFAULT 0 CHECK(power >= 0), -- #1
    PRIMARY KEY (system_id, read_on), -- #2
    FOREIGN KEY (system_id)
            REFERENCES systems(id) -- #3
);
```

#1 Here several clauses are used to ensure data quality: A default value of `0` is assumed for the power readings, and as an additional column check constraint is used that makes sure no negative values are inserted

#2 This is how a composite primary key is defined after the list of columns

#3 Foreign key constraints are also table constraints and go after the column definitions

Finally, the `prices` table. The script for it actually contains two commands, as we are going to use an incrementing numeric value as the surrogate primary key. We do this by using a `DEFAULT` declaration with a function call to `nextval()`. This function takes the name of sequence as input. Sequences are numeric values stored in the database outside table definitions

Listing 3.3 Creating the `prices` table with a primary key based on a sequence

```
CREATE SEQUENCE IF NOT EXISTS prices_id
    INCREMENT BY 1 MINVALUE 10; -- #1

CREATE TABLE IF NOT EXISTS prices (
    id          INTEGER PRIMARY KEY
                  DEFAULT(nextval('prices_id')), -- #2
    value       DECIMAL(5,2) NOT NULL,
    valid_from  DATE NOT NULL,
    CONSTRAINT prices_uk UNIQUE (valid_from) -- #3
);
```

#1 This is a monotonous incrementing sequence, starting with 10

#2 This uses the `nextval()` function as a default value for the `id` column

#3 This adds a unique table constraint for the `valid_from` column

Why do we not use `valid_from` as the primary key? In the initial application we might be only dealing with selling prices, but in the future we might be dealing with buying prices too. There are several ways to model that: with an additional table or introducing a `type` column in the `prices` table, that specifies whether a certain value is a selling or buying price. If `valid_from` would be a primary key, you could not have two prices with different types be valid from the same date. Therefore, you would need to change a simple primary key to a composite one. While other database might allow dropping and recreating primary and unique, DuckDB does not, so in this case you would need to go through a bigger migration.

Also, updating the values of primary keys can be costly on its own, not only from an index perspective but also from an organizational one: it might be that the column has already been used as a reference column for a foreign key. Every constraint is backed by an index and changing values requires often a reorganization of that index, which might be slow and costly. Updating several tables in one transaction can be error-prone and might lead to inconsistencies mostly due to human errors. We don't have that danger in the `readings` table where we used the timestamp column as the primary key because the readings are essentially immutable.

3.3.2 The ALTER TABLE statement

Defining a schema is a complex task and organizations usually put a lot of effort into it. However, there is hardly ever a case you will cover all eventualities and get a schema completely right from the start. Requirements change all the time. Such a requirement can be the capturing the validity of a price, for which we need an additional column. In that case, use the `ALTER TABLE` statement:

```
ALTER TABLE prices ADD COLUMN valid_until DATE;
```

Other clauses that can be used with `ALTER TABLE` are `DROP` and `RENAME` column. We can also `RENAME` the table. Some column options such as default values can be changed, however adding, dropping or changing constraints is not supported at the time of writing. If you want to do that, you'll need to recreate the table.

There are further ways to create tables, including "Create table as select" (CTAS). This is a shortcut that duplicates the shape of a table and its content in one go. For example, we could create a duplicate of the `prices` table like this:

```
CREATE TABLE prices_duplicate AS
SELECT * FROM prices;
```

We could also add a `LIMIT 0` clause to copy the schema of a table without data or a `WHERE` clause with conditions to copy the shape together with some data.

3.3.3 The CREATE VIEW statement

The `CREATE VIEW` statement defines a view of a query. It essentially stores the statement that represents the query, including all conditions and transformations. The view will behave as any other table or relation when being queried and additional conditions and transformations can be applied. Some databases materialize view, others don't. DuckDB will run the underlying statements of a view if you query that view. In case you are running into performance issues, you might want to materialize the data of a view through a CTAS statement yourself into a temporary table. Any additional predicates that you might use when querying a view inside the `WHERE` clause are oftentimes used as "push down predicates". That means they will be added to the underlying query defining the view and are not used as filters after the data has been loaded.

A view that is helpful in our scenario is a view that gives us the amount of energy produced per system and per day in kWh. This view will encapsulate the logic to compute that value together with the necessary grouping statements for us. Views are a great way to create an API inside your database. That API can serve adhoc queries and applications alike. When the underlying computation changes, the view can be recreated with the same structure without affecting any outside application.

The `GROUP BY` clause is one of those clauses you hardly can go without in the relational world, and we will explain this in detail later in this chapter. For the example here, it is enough to understand that the `GROUP BY` clause computes the total power produced by system and day. The `sum` function used in the select list is a so-called aggregate function, aggregating the values belonging to a group.

Listing 3.4 Create a view for power production by system and day

```
CREATE OR REPLACE VIEW v_power_per_day AS
SELECT system_id,
       date_trunc('day', read_on)        AS day,
       round(sum(power)  / 4 / 1000, 2)  AS kWh,
FROM readings
GROUP BY system_id, day;
```

It does not matter whether the underlying tables are empty or not for a view to be created, as long as they exist. While we did create the `readings` table, we didn't insert any data, yet, so querying the view with `SELECT * FROM v_power_per_day` will return an empty result for now. We will go back to this view in section 3.4.1 and use it after that in several examples throughout chapters 3 and 4.

3.3.4 The DESCRIBE statement

Probably all relational databases support the `DESCRIBE` statement to query the database schema. In its most basic implementation it works usually with tables and views.

TIP Relational databases are based on the relational model and eventually relational algebra. The relational model had first been described by Edgar F. Codd in 1970. In essence all data is stored as sets of tuples grouped together in relations. A tuple is an ordered list of attributes. Think of them as the column list of a table. A table then is the relation of a set of tuples. A view is a relation of tuples, too and so is the result of a query.
Graph databases in contrast to relational databases store actual relations between entities. In this book however we use the term as defined in the relational model.

The `DESCRIBE` statement in DuckDB works not only with tables, but with everything else being a relation, too: views, queries, sets, and more. You might want to describe the `readings` table with `DESCRIBE readings;`. Your result should be similar:

```
┌─────────────┬──────────────┬─────────┬─────────┬─────────┬───────┐
│ column_name │ column_type  │  null   │   key   │ default │ extra │
│   varchar   │   varchar    │ varchar │ varchar │ varchar │ int32 │
├─────────────┼──────────────┼─────────┼─────────┼─────────┼───────┤
│ system_id   │ INTEGER      │ NO      │ PRI     │         │       │
│ read_on     │ TIMESTAMP    │ NO      │ PRI     │         │       │
│ power       │ DECIMAL(8,3) │ NO      │         │ 0       │       │
└─────────────┴──────────────┴─────────┴─────────┴─────────┴───────┘
```

Describing a specific subset of columns (a new tuple) selected from any table such as `DESCRIBE SELECT read_on, power FROM readings;` yields:

```
┌─────────────┬──────────────┬─────────┬─────────┬─────────┬─────────┐
│ column_name │ column_type  │  null   │   key   │ default │  extra  │
│   varchar   │   varchar    │ varchar │ varchar │ varchar │ varchar │
├─────────────┼──────────────┼─────────┼─────────┼─────────┼─────────┤
│ read_on     │ TIMESTAMP    │ YES     │         │         │         │
│ power       │ DECIMAL(8,3) │ YES     │         │         │         │
└─────────────┴──────────────┴─────────┴─────────┴─────────┴─────────┘
```

Last but not least, describing any constructed tuple such as `DESCRIBE VALUES (4711, '2023-05-28 11:00'::timestamp, 42);` works the same:

```
┌─────────────┬─────────────┬─────────┬─────────┬─────────┬─────────┐
│ column_name │ column_type │  null   │   key   │ default │  extra  │
│   varchar   │   varchar   │ varchar │ varchar │ varchar │ varchar │
├─────────────┼─────────────┼─────────┼─────────┼─────────┼─────────┤
│ col0        │ INTEGER     │ YES     │         │         │         │
│ col1        │ TIMESTAMP   │ YES     │         │         │         │
│ col2        │ INTEGER     │ YES     │         │         │         │
└─────────────┴─────────────┴─────────┴─────────┴─────────┴─────────┘
```

TIP Use the `DESCRIBE` statement in all scenarios in which you are unsure about the shape of the data. It works for all kind of relations, local and remote files. Depending on the type of the file DuckDB can optimize the `DESCRIBE` statement. Even describing remote files (such as files in Parquet format) is very fast. Files in CSV format can be slower to describe, as they don't carry a schema with them and the engine needs to sample their content.

3.4 Data manipulation language (DML) queries

In the context of databases, all statements that insert, delete, modify *and* read data are called "data manipulation language" or DML in short. This section will first cover the `INSERT` and `DELETE` statements before going into querying data. We won't go into much detail of the `UPDATE` statement. The beauty of SQL queries is that they compose very naturally, so everything you'll learn for example about the `WHERE` clause does apply to the clause being used in `INSERT`, `DELETE`, `UPDATE` and `SELECT` statements.

3.4.1 The INSERT statement

For creating data, the `INSERT` statement is used. Inserting data is a task ranging from simple "fire and forget" statements to complex statements mitigating conflicts and ensuring good data quality. We start simple and naive by populating the price table we created in listing 3.3. An `INSERT` statement first specifies where you want to insert and then what you want to insert. The where is a table name, here it is the prices table. The what can be a list of column values, but they must match the column types and order of the table. In our case we're inserting one row with four values, two numeric and two strings, the latter will be automatically be cast to a `DATE`:

```
INSERT INTO prices
VALUES (1, 11.59, '2018-12-01', '2019-01-01');
```

The above query is fragile in a couple of ways: relying on the order of columns will break your statement as soon as the target table change. Also, we explicitly use the `1` as a unique key. If you were to execute the query a second time, it would rightfully fail, as the table contains already a row with the given key. The second row does violate the constraint that a primary key must be unique:

```
D INSERT INTO prices
> VALUES (1, 11.59, '2018-12-01', '2019-01-01');
Error: Constraint Error: Duplicate key "id: 1" violates primary key
➥ constraint. If this is an unexpected constraint violation please
➥ double check with the known index limitations section in our
➥ documentation (docs - sql - indexes).
```

While the conflict could not have been prevented given the schema, we can mitigate it by using the non-standard `ON CONFLICT` clause and just do nothing. The `DO NOTHING` clause targets the primary index by default (the `id` column in this case). While still being fragile, this statement is at least now idempotent.

```
INSERT INTO prices
VALUES (1, 11.59, '2018-12-01', '2019-01-01')
ON CONFLICT DO NOTHING;
```

In this case idempotency might be less useful than you think: you don't get an error, but you most likely won't get the expected result either. A better solution in our example would be specifying all columns we want to insert and not using an explicit value for the id. Overall, we already defined a sequence and a default value for the column that generates ids for us:

```
INSERT INTO prices(value, valid_from, valid_until)
VALUES (11.47, '2019-01-01', '2019-02-01'),
       (11.35, '2019-02-01', '2019-03-01'),
       (11.23, '2019-03-01', '2019-04-01'),
       (11.11, '2019-04-01', '2019-05-01'),
       (10.95, '2019-05-01', '2019-06-01');
```

There's another possible cause of failure: we defined a unique key for the validity date. For that we can actually react in way that can make sense from a business perspective. We can insert or replace (merge) the value when a conflict on that key arises:

```
INSERT INTO prices(value, valid_from, valid_until)
VALUES (11.47, '2019-01-01', '2019-02-01')
ON CONFLICT (valid_from) -- #1
  DO UPDATE SET value = excluded.value;
```

#1 As the table has multiple constraints (primary and unique keys), we must specify on which key the conflict mitigation shall happen

We will revisit that topic in section "Merging data".

Of course, it is possible to use the outcome of a `SELECT` statement as input for the `INSERT` statement. We will have a look at the anatomy of a `SELECT` statement shortly, but to complete the example please use it as follows. Think of this statement as a pipeline to the `INSERT` clause. It selects all the data from the file named `prices.csv` and inserts them in order of appearance (you find that file inside the `ch03` folder in the repository of this book on GitHub: https://github.com/duckdb-in-action/examples).

Listing 3.5 Inserting data from other relations

```
INSERT INTO prices(value, valid_from, valid_until)
SELECT * FROM 'prices.csv' src;
```

Let's also fill the `systems` table and load the first bunch of readings before we go over to the `SELECT` statement in detail. To be able to write the `INSERT` statement properly, we must understand what the CSV data looks like. We will make use of the fact that we can use `DESCRIBE` with any relation. In this case, a relation that is defined by reading the CSV file:

```
INSTALL 'httpfs'; -- #1
LOAD 'httpfs';
DESCRIBE SELECT * FROM
    'https://oedi-data-lake.s3.amazonaws.com/pvdaq/csv/systems.csv';
```

#1 Install the `httpfs` extension and load it so that we can access the url

Without specifying any type hints, `systems.csv` looks like this for DuckDB:

```
┌──────────────────┬──────────────┬─────────┬─────────┬─────────┬─────────┐
│   column_name    │ column_type  │  null   │   key   │ default │  extra  │
│     varchar      │   varchar    │ varchar │ varchar │ varchar │ varchar │
├──────────────────┼──────────────┼─────────┼─────────┼─────────┼─────────┤
│ system_id        │ BIGINT       │ YES     │         │         │         │
│ system_public_name │ VARCHAR    │ YES     │         │         │         │
│ site_id          │ BIGINT       │ YES     │         │         │         │
│ site_public_name │ VARCHAR      │ YES     │         │         │         │
│ site_location    │ VARCHAR      │ YES     │         │         │         │
│ site_latitude    │ DOUBLE       │ YES     │         │         │         │
│ site_longitude   │ DOUBLE       │ YES     │         │         │         │
│ site_elevation   │ DOUBLE       │ YES     │         │         │         │
└──────────────────┴──────────────┴─────────┴─────────┴─────────┴─────────┘
```

Using the `system_id` and `system_public_name` will do just nicely for us. However, it turns out that there are duplicates in the file which will cause our insertion to fail. The easiest way to filter out duplicates is applying the `DISTINCT` keyword in the columns clause of the `SELECT` statement. This ensures a unique set over all the columns we select.

Listing 3.6 Inserting a distinct set of rows from another table

```
INSTALL 'httpfs';
LOAD 'httpfs';

INSERT INTO systems(id, name)
SELECT DISTINCT system_id, system_public_name
FROM 'https://oedi-data-lake.s3.amazonaws.com/pvdaq/csv/systems.csv'
ORDER BY system_id ASC;
```

The systems in section 3.2.1 have been picked for specific reasons. We start with the dataset for system 34 as it suits our requirements to begin with (having readings in a 15 minutes interval). It does have some inconsistencies to deal with: the power output is sometimes `NULL` (not present) or negative. We will use a `CASE` expression to default missing values to 0.

As the URL does not give a clear indication for DuckDB which type of file or structure is behind it (like spotting an extension such as `.csv` or `.parquet`), we must use the `read_csv_auto` function, as the database won't be able to infer the correct file type.

Listing 3.7 Download and ingest the first set of readings

```
INSERT INTO readings(system_id, read_on, power)
SELECT SiteId, "Date-Time",
       CASE
           WHEN ac_power < 0 OR ac_power IS NULL THEN 0
           ELSE ac_power END
FROM read_csv_auto(
       'https://developer.nrel.gov/api/pvdaq/v3/data_file?' ||
       'api_key=DEMO_KEY&system_id=34&year=2019'
     );
```

A sample of the data for system 34 in 2019 we just ingested can be achieved with `SELECT * FROM readings WHERE date_trunc('day', read_on) = '2019-08-26' AND power <> 0;`:

```
┌─────────────┬─────────────────────┬─────────────────┐
│ system_id   │       read_on       │     power       │
│   int32     │      timestamp      │ decimal(10,3)   │
├─────────────┼─────────────────────┼─────────────────┤
│          34 │ 2019-08-26 05:30:00 │       1700.000 │
│          34 │ 2019-08-26 05:45:00 │       3900.000 │
│          34 │ 2019-08-26 06:00:00 │       8300.000 │
│           · │          ·          │           ·     │
│           · │          ·          │           ·     │
│           · │          ·          │           ·     │
│          34 │ 2019-08-26 17:30:00 │       5200.000 │
│          34 │ 2019-08-26 17:45:00 │       2200.000 │
│          34 │ 2019-08-26 18:00:00 │        600.000 │
├─────────────┴─────────────────────┴─────────────────┤
│ 51 rows (6 shown)                     3 columns     │
└─────────────────────────────────────────────────────┘
```

Now that we finally ingested some data into the readings table, the view `v_power_per_day` created in listing 3.4 also returns data. Remember, `v_power_per_day` creates daily groups and sums up their power values as shown with the output of `SELECT * FROM v_power_per_day WHERE day = '2019-08-26'`.

```
┌───────────┬────────────┬────────┐
│ system_id │    day     │  kWh   │
│   int32   │    date    │ double │
├───────────┼────────────┼────────┤
│        34 │ 2019-08-26 │  716.9 │
└───────────┴────────────┴────────┘
```

If you don't remember the definition of the view, be sure to check it again. A view is a great way to encapsulate logic such as truncating the date to a day and aggregating the total value of readings on that day such as in our example.

NOTE The query is essentially the same for 2020, apart from the URL parameter. Why don't we generate a list of file names using the `range` function that acts as an inline table like this?

```
SELECT *
FROM (
    SELECT 'https://' || years.range || '.csv' AS v
    FROM range(2019,2021) years
) urls, read_csv_auto(urls.v);
```

While this query is theoretically correct, it does not (yet) work due to restrictions in how so-called table functions (see chapter 4.9) are implemented in DuckDB. At the time of writing they only accept constant parameters. Furthermore, `read_csv` or `read_parquet` learn about their schema by looking at the input parameters and reading the given files, so there's a chicken-and-egg problem to be solved.

MERGING DATA

Often times you find yourself with a dataset that contains duplicates or entries that already exists within your database. While you can certainly ignore conflicts as shown in section 3.4.1 when your only task is to refine and clean new data, you sometimes want to merge new data into existing data. For this purpose DuckDB offers the `ON CONFLICT DO UPDATE` clause, known as `MERGE INTO` in other databases. In our example you might have multiple readings from different meters for the same system, and you want to compute the average reading. Instead of doing nothing on conflict we use a `DO UPDATE` now.

In listing 3.8 a random reading is inserted first and then an attempt is made to insert a reading on the same time for the same device. The second attempt will cause a conflict, not on a primary key, but on the composed key of `system_id` and `read_on`. With the `DO UPDATE` clause we specify the action to take when a conflict arises. The update clause can update as many columns as necessary, essentially doing a merge / upsert; complex expressions such as a `CASE` statement are allowed, too.

Listing 3.8 Compute new values on conflict

```
INSERT INTO readings(system_id, read_on, power)
  VALUES (10, '2023-06-05 13:00:00', 4000);

INSERT INTO readings(system_id, read_on, power)
  VALUES (10, '2023-06-05 13:00:00', 3000)
ON CONFLICT(system_id, read_on) DO UPDATE -- #1
SET power = CASE
  WHEN power = 0 THEN excluded.power -- #2
  ELSE (power + excluded.power) / 2 END;
```

#1 Here, the action is specified

#2 Columns from the original dataset can be referred to by the alias `excluded`

> **NOTE DuckDB also offers `INSERT OR REPLACE` and `INSERT OR IGNORE` as shorthand alternatives for `ON CONFLICT DO UPDATE` respectively `ON CONFLICT DO NOTHING`. `INSERT OR REPLACE` however does not have the ability to combine existing values as in the example above nor does it allow to define the conflict target.**

3.4.2 The DELETE statement

There are some outliers in the data sources we're using. We imported a bunch of readings that are measured on different minutes of the hour, and we just don't want them in our dataset. The easiest way to deal with them is to apply the `DELETE` statement and get rid of them. The following `DELETE` statement filters the rows to be deleted through a condition based on a negated `IN` operator. That operator checks the containment of the left expression inside the set of expressions on the right-hand side. `date_part` is just one of the many built-in functions of DuckDB dealing with dates and timestamps. This one extracts a part from a timestamp, in this case, the minutes from the `read_on` column:

Listing 3.9 Cleaning the ingested data

```
DELETE FROM readings
WHERE date_part('minute', read_on) NOT IN (0,15,30,45);
```

Sometimes, you will know about quirks and inconsistencies such as these upfront, and you don't have to deal with them after you ingested the data. With time-based data such in our example, you could have written the ingesting statement utilizing the `time_bucket` function. We noticed that inconsistency only after importing and think it's worthwhile to point this out.

3.4.3 The SELECT statement

This section is all about the `SELECT` statement and querying the ingested data. This statement retrieves data as rows from the database or, if used in a nested fashion, creates ephemeral relations. Those relations can be queried again or used to insert data as we have already seen.

The essential clauses of a `SELECT` statement and their canonical order is as follows:

Listing 3.10 The structure of a `SELECT` statement

```
SELECT select_list
FROM tables
WHERE condition
GROUP BY groups
HAVING group_filter
ORDER BY order_expr
LIMIT n
```

There are more clauses, in both the standard and the DuckDB specific SQL dialect, and we will discuss a couple of them in the next chapter as well. The official DuckDB documentation has a dedicated page to the `SELECT` statement: https://duckdb.org/docs/sql/statements/select which we recommend using as a reference on how each clause of the `SELECT` statement is supposed to be constructed.

We think that the following clauses are the most important to understand:

- `FROM` in conjunction with `JOIN`
- `WHERE`
- `GROUP BY`

They define the sources of your queries, they filter not only reading queries, but also writing queries and eventually, reshape them. They are used in many contexts, not only in querying data. Many other clauses are easier to understand, such as `ORDER` for example, which, as you'd expect from its name, puts things in order.

THE SELECT AND FROM CLAUSES

Every standard SQL statement that reads data starts with the `SELECT` clause. The `SELECT` clause defines the columns or expressions that will eventually be returned as rows. If you want to get everything from the source tables of your statement, you can use the `*`.

> **NOTE** Sometimes the `SELECT` clause is called a projection, choosing which columns to be returned. Ironically, the selection of rows happens in the `WHERE` clause.

For us, the `SELECT` and `FROM` clauses go together while explaining them, and we could pick either one to explain first or explain them together: The `FROM` clause specifies the source of the data on which the remainder of the query should operate and for a majority of queries, that will be one or more tables. In case there is more than one table list in the `FROM` clause or the additional `JOIN` clause is used, we speak about joining tables together.

The following statement will return two rows from the prices table. The `LIMIT` clause we are introducing here as well limits the number of returned rows. It's often wise to limit the amount of data you get back in case you don't know the underlying dataset so that you don't cause huge amount of network traffic or end up with an unresponsive client.

```
SELECT *
FROM prices
LIMIT 2;
```

It will return the first two rows. Without an `ORDER` clause, the order is actually undefined and might differ in your instance:

```
┌─────────┬──────────────┬────────────┬────────────┐
│   id    │    value     │ valid_from │ valid_until │
│  int32  │ decimal(5,2) │    date    │    date    │
├─────────┼──────────────┼────────────┼────────────┤
│       1 │        11.59 │ 2018-12-01 │ 2019-01-01 │
│      10 │        11.47 │ 2019-01-01 │ 2019-02-01 │
└─────────┴──────────────┴────────────┴────────────┘
```

The SQL dialect of DuckDB allows us to cut the above down to just `FROM prices;` (without the limit it will return all rows, but that's ok, we know the content of that table from section 3.4.1).

THE WHERE CLAUSE

The WHERE clause allows filtering your data by adding conditions to a query. Those conditions are built of one or more expressions. Data that is selected through a SELECT, DELETE or UPDATE statement must match those predicates to be included in the operations This allows you to select only a subset of the data in which you are interested. Logically the WHERE clause is applied immediately after the FROM clause or the preceding DELETE or UPDATE statement.

In our example we can replace that arbitrary LIMIT with a proper condition that will include only the prices for a specific year (2000) by adding the following WHERE clause:

```
FROM prices
WHERE valid_from BETWEEN -- #1
  '2020-01-01' AND '2020-12-31';
```

#1 The BETWEEN keyword is a shorthand for `x >= v AND v <= x`.

Based on our example data, the query will return 11 rows:

```
┌──────┬──────────────┬────────────┬──────────────┐
│  id  │    value     │ valid_from │ valid_until  │
│ int32 │ decimal(5,2) │    date    │     date     │
├──────┼──────────────┼────────────┼──────────────┤
│   15 │         8.60 │ 2020-11-01 │ 2023-01-01   │
│   17 │         8.64 │ 2020-10-01 │ 2020-11-01   │
│    · │            · │      ·     │      ·       │
│    · │            · │      ·     │      ·       │
│    · │            · │      ·     │      ·       │
│   25 │         9.72 │ 2020-02-01 │ 2020-03-01   │
│   26 │         9.87 │ 2020-01-01 │ 2020-02-01   │
├──────┴──────────────┴────────────┴──────────────┤
│ 11 rows (4 shown)                  4 columns │
└──────────────────────────────────────────────┘
```

THE GROUP BY CLAUSE

Grouping by one or more columns generates one row of output per unique value of those columns; it lets you group all rows that match those field together. Then the grouped values get aggregated via an aggregation function, like `count`, `sum`, `avg`, `min` or `max` so that one single value for that group is produced. This can be useful when you want to do things like computing the average number of readings per day, or the sum of the customers in each state. If the `GROUP BY` clause is specified, the query is always an aggregate query, even if no aggregations are present in the select list. DuckDB has a handy extension that lets you group your query by all columns that are not part of an aggregate function, `GROUP BY ALL`. Figure 3.2 demonstrates how a selection of rows is grouped by the column `year` and the results of applying the aggregates `count`, `avg` and `min` and `max` to it.

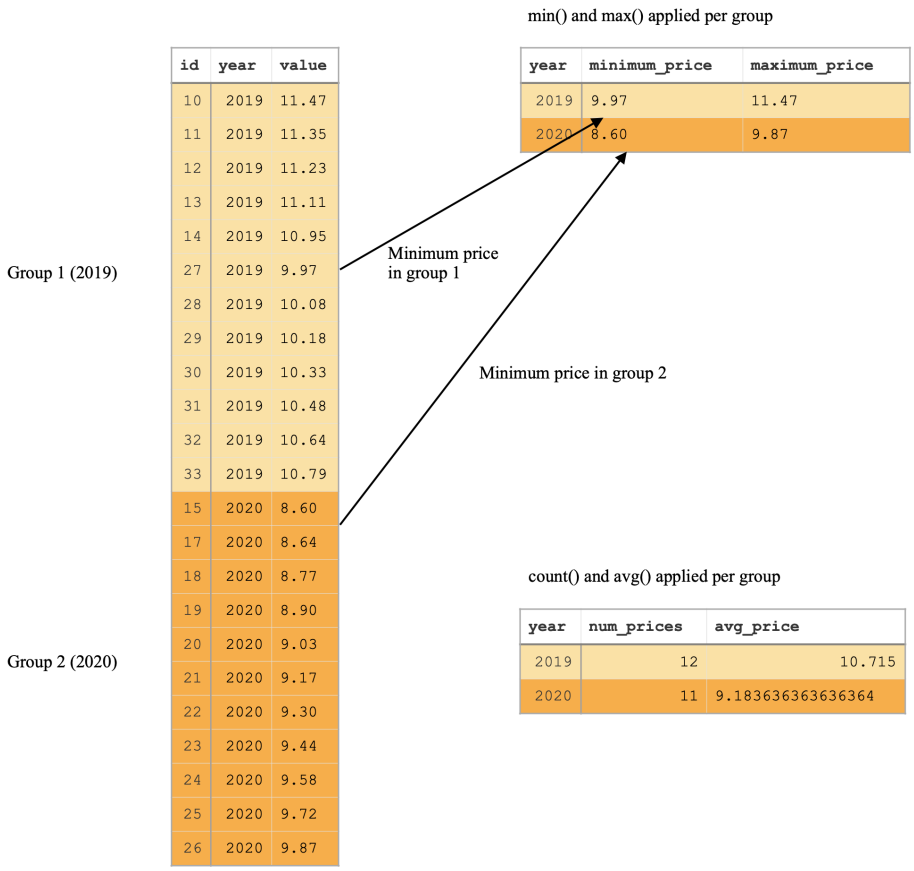


Figure 3.2 Grouping a dataset by year

There are many aggregate functions to choose from. In addition to the above, which are pretty standard, here are some that we think are often helpful:

- `list` to aggregate all values of each group into a list structure
- `any_value` picks any value from a non-grouping column
- `first` or `last` to pick the first or the last value from a non-grouping column if the result is ordered
- `arg_max` and `arg_min` solve the common task of finding the value of an expression in the row having a maximum or minimum value
- Bit operations working on sets, such as `bit_and`, `bit_or` and `bit_xor`
- Plus an exhaustive set of statistical aggregates ranging from `median`, quantile computation to computing covariance and general regressions

The full list is here https://duckdb.org/docs/sql/aggregates With that knowledge, let's see what we can do with our dataset.

Let's pick up the prices example we started to use in section The `SELECT` and `FROM` clauses. First, we added the `WHERE` clause to find prices in a year. While that was interesting, how about finding out the minimum and maximum prices per year?

We will use the `min` and `max` aggregates grouped by the year in which the prices have been valid to find out the highest and lowest prices in the years from 2019 to 2020. The `valid_from` column is a date, we are only interested in the year. The `date_part` function can extract that. If used without an alias, the resulting column would be named `date_part('year', valid_from)`. This does not read nicely, and it is also cumbersome to refer to. Therefore, the `AS` keyword is used to introduce the alias `year`. DuckDB allows us to refer to such an alias in the `GROUP BY` clause, which is different to the SQL standard and very helpful. The `year` becomes the grouping key by specifying it in the `GROUP BY` clause and its distinct values will define the buckets for which the minimum and maximum values of the column we chose should be computed.

Listing 3.11 Grouped aggregates

```
SELECT date_part('year', valid_from) AS year,
       min(value) AS minimum_price, -- #1
       max(value) AS maximum_price
FROM prices
WHERE year BETWEEN 2019 AND 2020
GROUP BY year -- #2
ORDER BY year;
```

#1 You can have as many aggregate functions in the `SELECT` clause as you want

#2 Note how we can reuse the alias we gave in the `SELECT` clause in the `GROUP BY` clause.

The result of this query is shown below:

```
┌───────┬─────────────┬─────────────┐
│ year  │ minimum_price │ maximum_price │
│ int64 │ decimal(5,2)  │ decimal(5,2)  │
├───────┼─────────────┼─────────────┤
│  2019 │        9.97 │       11.47 │
│  2020 │        8.60 │        9.87 │
└───────┴─────────────┴─────────────┘
```

TIP DuckDB offers choice when dealing with date parts. You can use the generic `date_part` function like we did and specify the part as parameter. There are identifiers for all relevant parts such as `'day'`, `'hour'`,`'minute'` and many more. All of them exist also as dedicated functions, so in listing 3.11 we could have used `year(valid_from)`, too. The generic function is helpful when the part is derived from other expressions in the statement or when you try to write portable SQL. The dedicated functions are easier to read.

THE `VALUES` CLAUSE

The `VALUES` clause is used to specify a fixed number of rows. We have seen it already while inserting data, which is a quite common use case. It is however much more versatile in DuckDB than in some other databases, as it can be used as a stand-alone statement and as part of the FROM clause too, with any number of rows and columns. There are a couple of scenarios in which this is handy, for example providing seed data for conditions.

Here's how to define a single row with two columns like with a simple `VALUES (1,2)`:

```
┌───────┬───────┐
│ col0  │ col1  │
│ int32 │ int32 │
├───────┼───────┤
│     1 │     2 │
└───────┴───────┘
```

Take note that multiple rows can be generated by just enumerating multiple tuples: `VALUES (1,2), (3,4)`, you don't need to wrap them in additional parenthesis.

```
┌─────────┬─────────┐
│  col0   │  col1   │
│  int32  │  int32  │
├─────────┼─────────┤
│       1 │       2 │
│       3 │       4 │
└─────────┴─────────┘
```

If you do however, such as in `VALUES ((1,2), (3,4));` you will create a single row with two columns each containing a structured type:

```
┌──────────────────────────────────┬──────────────────────────────────┐
│               col0               │               col1               │
│ struct(v1 integer, v2 integer)   │ struct(v1 integer, v2 integer)   │
├──────────────────────────────────┼──────────────────────────────────┤
│ {'v1': 1, 'v2': 2}               │ {'v1': 3, 'v2': 4}               │
└──────────────────────────────────┴──────────────────────────────────┘
```

When used in a from clause the resulting types can be named, together with their columns. We will make use of that in the next section while discussing joining logic. The following snippet defines two rows with 3 columns within the `VALUES` clause and creates a named type that holds the column names. The name of the type is arbitrary, we just picked `t`.

```
SELECT *
FROM (VALUES
    (1, 'Row 1', now()),
    (2, 'Row 2', now())
) t(id, name, arbitrary_column_name);
```

The resulting virtual table looks like this:

```
┌─────────┬──────────┬──────────────────────────────┐
│   id    │   name   │    arbitrary_column_name     │
│  int32  │  varchar │  timestamp with time zone    │
├─────────┼──────────┼──────────────────────────────┤
│       1 │ Row 1    │ 2023-06-02 13:44:30.309+02   │
│       2 │ Row 2    │ 2023-06-02 13:44:30.309+02   │
└─────────┴──────────┴──────────────────────────────┘
```

THE JOIN CLAUSE

While you can get away without using a JOIN clause when analyzing single Parquet or CSV files, you should not skip this section: Joins are a fundamental relational operation used to connect two tables or relations. The relations are referred to as the left and right sides of the join with the left side of the join being the table listed first. This connection represents a new relation combining previously unconnected information, thus providing new insights.

In essence a join creates matching pairs of rows from both sides of the join. The matching is usually based on a key-column in the left table being equal to a column in the right table. Foreign key constraints are not required for joining tables together. We prefer the SQL standard definition of joins based on the JOIN .. USING over JOIN .. ON clauses as you'll see in the following examples and throughout the rest of the book. Nevertheless, joins can be expressed by just enumerating the tables in the FROM clause and comparing the key columns in the WHERE clause.

> **NOTE** We are not using Venn diagrams for explaining joins because join-operations are not pure set operations, for which Venn diagrams would be a great choice. SQL does know set operations such as UNION, INTERSECT and EXCEPT—and DuckDB supports all of them. Joins on the other hand are all based on a cartesian product in relational algebra, or in simple terms: They are all based on joining everything with everything else, and then filter things out. In essence, all different joins can be derived from the CROSS JOIN. The inner join filters then on some condition, and a left or right outer join adds a union to it, but that's all there is than to set-based operations in joins.

In the following examples, we will use the VALUES clause to define virtual tables with a fixed number of rows with a given set of values. These sets are helpful to understand the joining logic as you will see both the sources and the resulting rows in the example. Usually you will find yourself joining different tables together, such as the power readings and the prices in our example.

The simplest way of joining is an INNER JOIN, which also happens to be the default. An inner join matches all rows from the left-hand side to rows from the right-hand side that have a column with the same value.

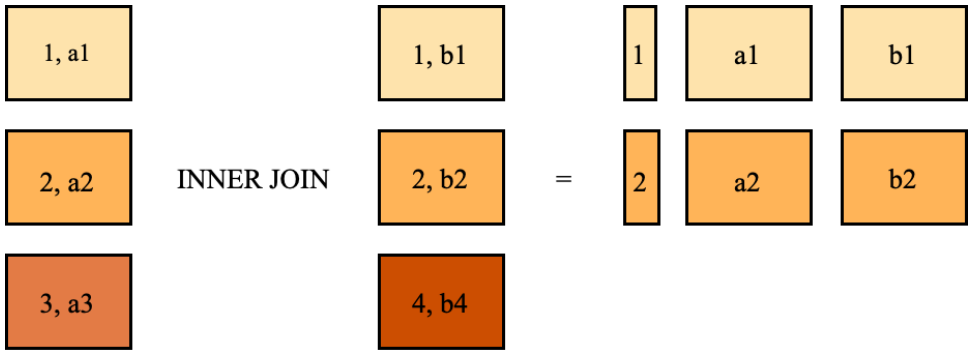


Figure 3.3 The inner join only matching pairs with equal keys

If both relations have a column with the same name, the `USING` clause can be used to specify that. The `USING` clause will look up the specified columns in both relations and work the same way as specifying them yourself via the `ON` clause (`ON tab1.col = tab2.col`).

Listing 3.12 Using an inner join

```
SELECT *
FROM
    (VALUES (1, 'a1'),
            (2, 'a2'),
            (3, 'a3')) l(id, nameA)
JOIN
    (VALUES (1, 'b1'),
            (2, 'b2'),
            (4, 'b4')) r(id, nameB)
USING (id); -- #1
```

#1 This is equivalent to `ON r1.id = r2.id`

The result will look like this:

```
┌───────┬─────────┬─────────┐
│  id   │  nameA  │  nameB  │
│ int32 │ varchar │ varchar │
├───────┼─────────┼─────────┤
│     1 │ a1      │ b1      │
│     2 │ a2      │ b2      │
└───────┴─────────┴─────────┘
```

An outer join on the other hand supplements `NULL` values for rows on the specified side of the relation that have no matching entry on the other. Think of several power producing systems in your database: for some you might have stored additional vendor information in another table, for some you don't. You would use an outer join when tasked with "give me a list of all systems with the optional vendor or an empty column if there's no such vendor." Listing 3.13 uses a `LEFT OUTER JOIN` so that all rows of the left relations are included and supplemented with `NULL` values for rows that don't have a match.

Listing 3.13 Using a left outer join

```
SELECT *
FROM
    (VALUES (1, 'a1'),
            (2, 'a2'),
            (3, 'a3')) l(id, nameA)
LEFT OUTER JOIN
    (VALUES (1, 'b1'),
            (2, 'b2'),
            (4, 'b4')) r(id, nameB)
USING (id)
ORDER BY id;
```

Joining the virtual tables from listing 3.13 with a `LEFT OUTER JOIN` results in:

```
┌───────┬─────────┬─────────┐
│  id   │  nameA  │  nameB  │
│ int32 │ varchar │ varchar │
├───────┼─────────┼─────────┤
│     1 │ a1      │ b1      │
│     2 │ a2      │ b2      │
│     3 │ a3      │         │
└───────┴─────────┴─────────┘
```

All rows from the left-hand side have been included and for `a3` a `NULL` value has been joined. Try changing the outer join from `LEFT` to `RIGHT` and observe which values are now included. Both `LEFT` and `RIGHT` outer join will return 3 rows in total. For getting back 4 rows you must use a full outer join as shown below:

Listing 3.14 Using a full outer join

```
SELECT *
FROM
    (VALUES (1, 'a1'),
            (2, 'a2'),
            (3, 'a3')) l(id, nameA)
FULL OUTER JOIN
    (VALUES (1, 'b1'),
            (2, 'b2'),
            (4, 'b4')) r(id, nameB)
USING (id)
ORDER BY id;
```

Four rows will be returned, with two `NULL` values, once for `nameA` and once for `nameB`:

```
┌───────┬─────────┬─────────┐
│  id   │  nameA  │  nameB  │
│ int32 │ varchar │ varchar │
├───────┼─────────┼─────────┤
│     1 │ a1      │ b1      │
│     2 │ a2      │ b2      │
│     3 │ a3      │         │
│     4 │         │ b4      │
└───────┴─────────┴─────────┘
```

Figure 3.4 represents both the left outer join, the full outer join that we had in code before plus the right outer join for comparison, too:

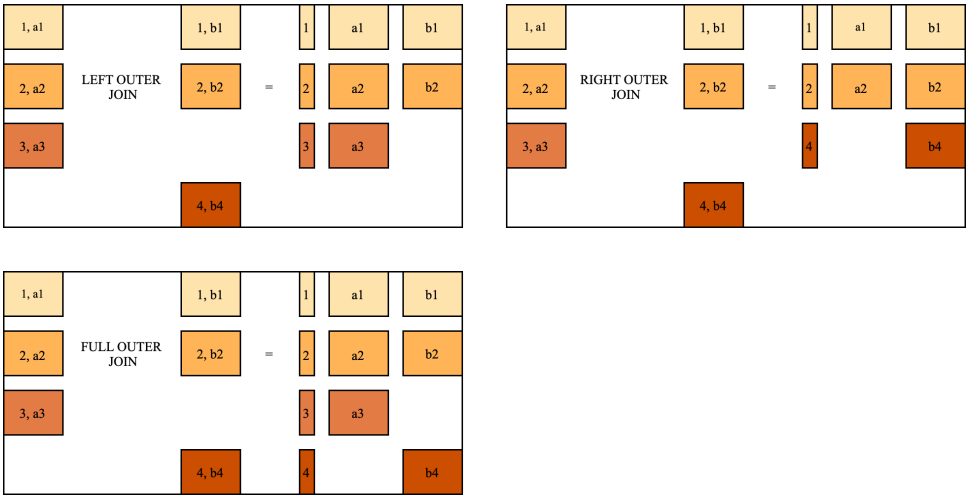


Figure 3.4 Types of outer joins

While an outer join gives you always the rows an inner join would give, it would be wrong to suggest always using outer joins. An inner join will filter out rows that have no matching data in the other table, which is often a requirement. An outer join will usually be appropriate when you want to enrich required data with optional data.

The above example applies the `USING` clause for the join conditions as both tables have an `id` column. In our example we defined an `id` column in the `systems` table and the foreign key column in `readings` as `system_id`. We therefore must use the `ON` clause. When joined on that column, the join will always produce a matching row, as the join column (`id`) is the column referenced by the foreign key we defined on `system_id`. That means there can't be any row in the `readings` table without a matching entry in the `systems` table:

```
SELECT name, count(*) as number_of_readings
FROM readings JOIN systems ON id = system_id
GROUP BY name;
```

NOTE A cartesian product is a mathematical term. It describes the list of all ordered pairs that you can produce from two sets of elements by combining each element from the first set with each element of the second set. The size of a cartesian product is equal to the product of the sizes of each set.

There are more join types, such as the `CROSS JOIN`, which creates a Cartesian product of all tuples, and the `ASOF` ("as of"), that will come in handy when dealing with the prices having a restricted validity for example: the `ASOF` join allows to match rows from one table with rows from another table based on temporal validity (or as a matter of fact with anything that has an inequality condition (`<=`, `<`, `>` or `>=`)). You will read about the `ASOF` join in chapter 4.8 in detail.

TIP **You are building data pipelines around CSV files and often have data split across several files with one common column per file. What if you wanted to reduce these files to exactly one file without duplicating the common column? That's easy to achieve with an inner join and the `COPY TO` command. The latter takes any relation and copies it to a file using the specified format:**

```
console
duckdb -c "COPY (SELECT * FROM 'production.csv' JOIN 'consumption.csv'
➥ USING (ts) JOIN 'export.csv' USING (ts) JOIN 'import.csv' USING (ts) )
➥ TO '/dev/stdout' (HEADER)"
```

This command will join 4 csv files on a shared column `ts`, keep only one copy of the shared column in the `SELECT *` statement and copy the result to standard out.

We'd like to end this section with some sort of warning. In our examples for inner and outer joins we only discussed what happens when a value of a key-column is not found in one of the other tables. But what happens when one of the join columns contains the same value multiple times, either in one of the join tables or in both? Let's find out. The value 2 for the `id` column appears twice in the left table and the value 3 twice in the right table.

Listing 3.15 An inner join between tables with duplicate key columns

```
SELECT *
FROM
    (VALUES (1, 'a1'),
            (2, 'a2'),
            (2, 'a2'),
            (3, 'a3')) l(id, nameA)
JOIN
    (VALUES (1, 'b1'),
            (2, 'b2'),
            (3, 'b3'),
            (3, 'b3')) r(id, nameB)
USING (id)
ORDER BY id;
```

The result of this statement won't be 4 rows as before, but 6 rows:

```
┌───────┬─────────┬─────────┐
│  id   │  nameA  │  nameB  │
│ int32 │ varchar │ varchar │
├───────┼─────────┼─────────┤
│     1 │ a1      │ b1      │
│     2 │ a2      │ b2      │
│     2 │ a2      │ b2      │
│     3 │ a3      │ b3      │
│     3 │ a3      │ b3      │
└───────┴─────────┴─────────┘
```

This is something that you can prepare for when defining your schema. Usually joins will happen on columns that are known upfront. In our example that would be the pair of `id` of the `systems` table that is referred to as `system_id` in the `readings` table. In the `systems` table that column is defined as primary key and as such, it will always be a unique value, hence it can only appear once in that table. On the `readings` table it is defined as foreign key, meaning it must exist in the other one. The foreign key usually creates a so-called index in the database that allows quick lookups, without going through all rows, making the join perform well. The foreign key is not unique and does not need to be in most models. In our example having the system appear multiple times in the readings table (the right-hand side) is expected, unless you want your system to produce power only once.

THE WITH CLAUSE

The WITH clause is also known as a common table expression (CTE). CTEs are essentially views that are limited in scope to a particular query. Like a view, you might want to use them to encapsulate parts of the logic of your query into a standalone statement or at least, into an isolated part of a bigger query. While it would be perfectly ok to create a view, you might not want that because you would only need its result in the specific context of the bigger query. In addition, CTEs have one special trait that views don't have: views can reference other views, but they cannot be nested. A CTE can reference other CTEs that are defined in the same WITH clause. With that, you can build your query logic in an incremental fashion.

WITH clauses prevent the anti-pattern of having sub-queries defined in the FROM clause. A sub-query as a source relation in a FROM clause is syntactically and semantically valid as its result is a relation on its own, but it is often hard to read. In addition, nested sub-queries are not allowed to reference themselves.

Finding the row containing the maximum value of a specific column in that row is often computed by using a sub-query in the FROM clause like this:

```
SELECT max_power.v, read_on
FROM (
  SELECT max(power) AS v FROM readings
)  max_power
JOIN readings ON power = max_power.v;
```

The sub-query is a pretty simple query and rewriting it as a CTE doesn't seem to make a big difference at first glance. We take the same query, move it out of the FROM clause and give a name within the WITH clause. The join statement stays the same:

Listing 3.16 Replacing a sub-query with a CTE

```
WITH max_power AS (
  SELECT max(power) AS v FROM readings
)
SELECT max_power.v, read_on
FROM max_power
JOIN readings ON power = max_power.v;
```

For single and rather basic queries like this it does not make much of a difference whether to use a sub-query or a CTE. But what if we ask for something like the maximum average production of power per system and hour? Aggregate functions like max and avg cannot be nested—i.e., you cannot do avg(max(v))—so you need to individual aggregates.

The question of which row contains the minimum or maximum value of a column is such a common task, that DuckDB has two built-in functions performing it: `arg_max` and `arg_min`. These functions compute an expression defined by their first parameter on the columns in the row for which the minimum or maximum value of the second parameter occurs the first time. The following query will produce one row from the dataset at which the highest amount of power was generated (not the 5 times, the query in listing 3.16 will return). This is because `arg_max` stops at the first value it finds that matches the maximum value, while the join will include all rows.

```
SELECT max(power), arg_max(read_on, power) AS read_on
FROM readings;
```

The next query in listing 3.17 makes use of the `arg_max` aggregate. It first encapsulates the complex logic of grouping the readings into average production by system and hour —creating the first aggregate—in a CTE that we name `per_hour` and then takes that CTE and computes a second aggregate over it.

Listing 3.17 Creating multiple groups

```
WITH per_hour AS ( -- #1
    SELECT system_id,
           date_trunc('hour', read_on) AS read_on,
           avg(power) / 1000 AS kWh -- #2
    FROM readings
    GROUP BY ALL
)
SELECT name,
       max(kWh), -- #3
       arg_max(read_on, kWh) AS 'Read on'
FROM per_hour -- #4
   JOIN systems s ON s.id = per_hour.system_id
WHERE system_id = 34
GROUP by s.name;
```

#1 Using a proper name for the CTE

#2 The average value per hour and day is the first aggregate we need; `GROUP BY ALL` is a DuckDB extension creating a group from all columns not part of an aggregate

#3 The nested aggregate we look for

#4 Using the CTE as the driving table in the `FROM` clause

The result shows the "Andre Agassi Preparatory Academy" having the system with the highest production in our dataset

```
┌──────────────────────────────────────┬──────────┬─────────────────────┐
│                name                  │ max(kWh) │       Read on       │
│               varchar                │  double  │      timestamp      │
├──────────────────────────────────────┼──────────┼─────────────────────┤
│ [34] Andre Agassi Preparatory Academy │   123.75 │ 2020-04-09 11:00:00 │
└──────────────────────────────────────┴──────────┴─────────────────────┘
```

NOTE We looked this building up and the readings and values add up. From https://www.bombardre.com/wp-content/uploads/2017/10/Andre-Agassi-Academy.pdf: "Between April 2010 and July 2011, Bombard installed 2,249 Sharp 240 watt solar modules on the roofs of five buildings and three solar support structures at the Agassi Academy in Las Vegas."

CTEs can do one more cool thing that views and sub-queries cannot: The `WITH` clause has the additional keyword `RECURSIVE` that makes it possible to reference a CTE not only from other succeeding CTEs and the `FROM` clause, but from within itself. Such a recursive CTE essentially will follow this pattern shown below in listing 3.18. To make this work, we need to have some kind of initial seed for the recursion. This is easy for a tree structure: we take the row that has no parent row and use this as one leaf of a `UNION` clause.

Listing 3.18 Selecting a graph-shaped structure with recursive SQL

```
CREATE TABLE IF NOT EXISTS src (
    id INT PRIMARY KEY,
    parent_id INT, name VARCHAR(8)
);

INSERT INTO src (VALUES
    (1, null, 'root1'),
    (2,    1, 'ch1a'),
    (3,    1, 'ch2a'),
    (4,    3, 'ch3a'),
    (5, null, 'root2'),
    (6,    5, 'ch1b')
);

WITH RECURSIVE tree AS (
    SELECT id,
           id AS root_id,
           [name] AS path -- #1
    FROM src WHERE parent_id IS NULL -- #2
    UNION ALL
    SELECT src.id,
           root_id,
           list_append(tree.path, src.name) AS path
    FROM src
      JOIN tree ON (src.parent_id = tree.id) -- #3
)
SELECT path FROM tree;
```

#1 Initialize a new list with a list literal

#2 This is the recursive initial seed

#3 Recursive join until there are no more entries from the `src` table with the given parent id

The results are several paths, all starting at the root, making up their way to the corresponding leaves:

```
┌──────────────────────┐
│         path         │
│       varchar[]      │
├──────────────────────┤
│ [root1]              │
│ [root2]              │
│ [root1, ch1a]        │
│ [root1, ch2a]        │
│ [root2, ch1b]        │
│ [root1, ch2a, ch3a]  │
└──────────────────────┘
```

The example aggregates names into a path from the root of a tree to the leaf by using `list_append`. You could use `list_prepend` and inverse the parameter to build up paths from the leafs to the root nodes.

As an exercise you try to compute the longest path in the tree. The recursive CTE would stay the same, but you will want to apply the `arg_max` function you already learned about in the `SELECT` statement together with the `length` aggregate on a list.

3.5 DuckDB-specific SQL extensions

One of the goals of the authors of DuckDB is to make SQL more accessible and user-friendly. One way that they've done this is by adding additions to their implementation of SQL that make it easy to do common tasks. In this section, we'll introduce those additions.

3.5.1 Dealing with SELECT *

`SELECT *` is a two-edged sword: it is easy to write down and the resulting tuples most likely will contain what you actually need.

Some problems that go along with selecting all columns of a relation are:

- Instability of the resulting tuples as a table definition might change (adding or removing columns)
- Putting more memory pressure on the database server or process
- While DuckDB is an embedded database and won't involve network traffic, star-selects will cause more traffic on non-embedded databases
- A star-select might prevent an index-only-scan. An index-only-scan will occur when your query can use an index, and you only return columns from that index so that any other IO can be avoided. An index-only-scan is a desired behaviour in most cases

While it's best to avoid doing too many `SELECT *` queries, sometimes they are necessary, and DuckDB actually makes them safer to use with the addition of two keywords, `EXCLUDE` and `REPLACE`.

If you are sure that you really want all columns, DuckDB offers a simplified version of the `SELECT` statement, omitting the `SELECT` clause altogether, starting with the `FROM` clause, so that you can do for example a `FROM prices`. We have another example coming up in listing 3.22.

EXCLUDING SOME COLUMNS WITH `EXCLUDE`

`EXCLUDE` excludes one or more columns from a star query. This is helpful when you have a table or relation with a lot of columns and nearly all of them are needed, apart from some that are irrelevant to your specific use case. Normally you would have to enumerate all the columns you are interested in and exclude the ones you don't care about. For example, you want only the relevant data from prices:

```
SELECT value, valid_from, valid_until FROM prices;
```

This gets tedious and error-prone real quickly, especially with more than a handful of columns With the `EXCLUDE` clause, you only have to enumerate the columns you are not interested in:

```
SELECT * EXCLUDE (id)
FROM prices;
```

You can exclude as many columns as you want. You will achieve most of the flexibility of a pure `SELECT *`, keep the readability of the star, and make sure you don't access something you don't need.

RESHAPING RESULTS WITH `REPLACE`

Think about the view `v_power_per_day`. It computes the kWh in fractions. For some users you may only want to return the integer values. You could create a copy of that view that recomputes the values rounding them, but you can also reuse the existing logic and do the rounding where needed and keeping the structure the same:

```
SELECT * REPLACE (round(kWh)::int AS kWh)
FROM v_power_per_day;
```

The `REPLACE` clause takes in one or more pairs of `x AS y` constructs with `x` being an expression that can refer to columns of the original select list, applying functions and other transformations to them and `y` being a name that has been used in the original select list.

The structure of the result is the same, but the kWh column now is an integer column:

```
┌─────────────┬──────────────┬────────┐
│ system_id   │    day       │  kWh   │
│   int32     │    date      │ int32  │
├─────────────┼──────────────┼────────┤
│        1200 │ 2019-08-29   │    289 │
│           · │      ·       │      · │
│           · │      ·       │      · │
│           · │      ·       │      · │
│          10 │ 2020-03-19   │      0 │
├─────────────┴──────────────┴────────┤
│ 1587 rows (2 shown)  3 columns      │
└─────────────────────────────────────┘
```

DYNAMICALLY PROJECTING AND FILTERING ON COLUMNS

Let's recap the `prices` table, it has two columns containing information about the validity of a price:

```
┌──────────────┬──────────────┬──────────┬─┬──────────────────────┬─────────┐
│ column_name  │ column_type  │   null   │…│       default        │ extra   │
│   varchar    │   varchar    │ varchar  │ │       varchar        │ int32   │
├──────────────┼──────────────┼──────────┼─┼──────────────────────┼─────────┤
│ id           │ INTEGER      │ NO       │…│ nextval('prices_id') │         │
│ value        │ DECIMAL(5,2) │ NO       │…│                      │         │
│ valid_from   │ DATE         │ NO       │…│                      │         │
│ valid_until  │ DATE         │ YES      │…│                      │         │
├──────────────┴──────────────┴──────────┴─┴──────────────────────┴─────────┤
│ 4 rows                                                 6 columns (5 shown) │
└────────────────────────────────────────────────────────────────────────────┘
```

The `COLUMNS` expression can be used to project, filter and aggregate one or more columns based on a regular expression. To select only columns that contain information about validity you can query the table like this:

```
SELECT COLUMNS('valid.*') FROM prices LIMIT 3;
```

Returns all the relevant columns:

```
┌────────────┬─────────────┐
│ valid_from │ valid_until │
│    date    │    date     │
├────────────┼─────────────┤
│ 2018-12-01 │ 2019-01-01  │
│ 2019-01-01 │ 2019-02-01  │
│ 2019-02-01 │ 2019-03-01  │
└────────────┴─────────────┘
```

You want to use that technique if you have a table with lots of columns that have similar names. That could be the case with a readings or measurement table. For example, think of an IoT-sensor that produces many different readings per measurement. For that use case another feature is interesting: You can apply any function over a dynamic selection of columns that will produce as many computed columns. Here we compute several maximum values at once for all columns in the price table that contain the word "valid":

```
SELECT max(COLUMNS('valid.*')) FROM prices;
```

Which results in the maximum values for `valid_from` and `valid_until`:

```
┌──────────────────────────┬──────────────────────────┐
│ max(prices.valid_from)   │ max(prices.valid_until)  │
│          date            │          date            │
├──────────────────────────┼──────────────────────────┤
│ 2023-01-01               │ 2024-02-01               │
└──────────────────────────┴──────────────────────────┘
```

If you find yourself writing long conditions in the `WHERE` clause combining many predicates with `AND`, you can simplify that with the `COLUMNS` expression as well. To find all the prices that have been valid in 2020 alone you would want every row for which both the `valid_from` and `valid_until` columns are between January 1st 2020 and 2021 which is exactly what the following query expresses:

```
FROM prices WHERE COLUMNS('valid.*') BETWEEN '2020-01-01' AND '2021-01-01';
```

You might have noticed that the regular expression `.*` sticks out a bit. Many people are more familiar with the `%` and `_` wildcards that are used with the `LIKE` operator. `%` represents zero, one, or multiple characters while the underscore sign represents one, single character. Luckily, `COLUMNS` supports lambda functions.

TIP A Lambda Function is a self-contained block of functionality that can be passed around and used in your code. Lambda functions have different names in different programming languages, such as Lambda expressions in Java, Kotlin and Python, Closures in Swift and blocks in C.

The query above selecting a range of prices can also be written like this:

```
FROM prices
WHERE COLUMNS(col -> col LIKE 'valid%') -- #1
BETWEEN '2020-01-01' AND '2021-01-01';
```

#1 The expression inside the `COLUMNS` expression is a Lambda function evaluating to `true` when the column name is like the given text.

Last but not least, you can combine the `COLUMNS` expression with the `REPLACE` or `EXCLUDE` conditions, too. Let's say you want to compute the maximum value over all the columns in the `prices` table except the generated `id` value, you can get them like this:

```
SELECT max(COLUMNS(* EXCLUDE id)) FROM prices;
```

3.5.2 Inserting by name

Remember listing 3.6? In that listing we used a statement in the form of `INSERT INTO target(col1, col2) SELECT a, b FROM src` to populate our systems table. This works but can be fragile to maintain, as the `INSERT` statement requires either the selected columns to be in the same order as they are defined by the target table or that you repeat the column names. Once in the `INTO` clause, once in the select list.

DuckDB offers a `BY NAME` clause to solve that issue and listing 3.6 can be rewritten as follows, keeping the mapping from the column names in the source to the column names for the target together in one place. The `BY NAME` keyword in the following listing indicates that the columns in the select clause that follows shall be matched by name onto columns of the target table.

Listing 3.19 Insertion by name

```
INSERT INTO systems BY NAME
SELECT DISTINCT
       system_id AS id,
       system_public_name AS NAME
FROM 'https://oedi-data-lake.s3.amazonaws.com/pvdaq/csv/systems.csv';
```

Whether you add new columns or remove columns from the insertion you only need to change the query now in one place. Any constraints however, such as non-null columns, must be still full-filled.

3.5.3 Accessing aliases everywhere

You probably haven't noticed it, but several of our examples benefit from something that should be the standard, but isn't. The moment you introduce an alias to a column, you can access them in succeeding clauses. In the following listing we access the non-aggregate alias `is_not_system10`—as defined in the select list—in the `WHERE` and the `GROUP BY` clauses without repeating the column definition. The latter is not possible in many other relational databases. The same applies to the alias `power_per_month` we gave to the `sum` aggregate: we can access it in the `HAVING` clause, too.

Listing 3.20 Access aliases in `WHERE`, `GROUP BY` and `HAVING` clauses

```
SELECT system_id > 10 AS is_not_system10,
       date_trunc('month', read_on) AS month,
       sum(power) / 1000 / 1000 AS power_per_month
FROM readings
WHERE is_not_system10 = TRUE -- #1
GROUP BY is_not_system10, month
HAVING power_per_month > 100; -- #2
```

#1 Accessing an alias that refers to a non-aggregate
#2 Accessing an alias that refers to an aggregate

3.5.4 Grouping and ordering by all relevant columns

As learned in the section about the `GROUP BY` clause, all non-aggregate columns need to be enumerated in a `GROUP BY` clause. If you have lots of non-aggregate columns this can be a painful experience and one that DuckDB alleviates by allowing you to use `GROUP BY ALL`. We can rewrite `v_power_per_day` like this:

Listing 3.21 Creating grouping-sets by grouping by all non-aggregate values

```
CREATE OR REPLACE VIEW v_power_per_day AS
SELECT system_id,
       date_trunc('day', read_on)        AS day,
       round(sum(power)  / 4 / 1000, 2)  AS kWh,
FROM readings
GROUP BY ALL;
```

A similar concept exists for ordering. An `ORDER BY ALL` will sort the result by the included columns from left to right. Querying the freshly created view with `SELECT system_id, day FROM v_power_per_day ORDER BY ALL` will sort the result first by `system_id`, then `day`. In case of a select-start the order of the columns is defined by the table or view definition, of course. The following statement is valid SQL in DuckDB and it returns the power produced in kWH per day sorted by systems first, then days and then kWH:

Listing 3.22 Omitting the `SELECT` clause and simplify ordering

```
FROM v_power_per_day ORDER BY ALL;
```

3.5.5 Sampling data

When working with large datasets, we often want to get a sample of the data rather than having to look through everything. Assuming you have imported the readings for at least system 34, you will have more than 50000 records in your database. This can be confirmed with a `SELECT count(*) FROM readings`. We can get an overview of the non-zero power readings by asking for a sample of n percent or n number of rows:

Listing 3.23 Sample a relation

```
SELECT power
FROM readings
WHERE power <> 0
USING SAMPLE 10% -- #1
  (bernoulli); -- #2
```

#1 Retrieve a sample that has roughly 10% the size of the data
#2 Specify the sampling method to use (see below)

This is much easier and more flexible than dealing with arbitrary limits as it gives a better and more reliable overview. The sampling itself uses probabilistic sampling methods however, unless a seed is specified with the additional `REPEATABLE` clause. The sampling rate in percent is not meant to be an exact hit. In our example, it varies around 2000 rows from somewhat more than 20000 which have `power` column not equal to zero.

If you instruct DuckDB to use a specific rate for sampling, it applies `system` sampling, including each vector by an equal chance. Sampling on vectors instead of working on tuples (which is done by the alternative `bernoulli` method) is very effective and has no extra overhead. As one vector is roughly about 1000 tuples in size it is not suited for smaller datasets, as all data will be included or filtered out. Even for the total of ~100000 readings that have a power value greater than zero we recommend `bernoulli` for a more evenly distributed sampling.

For a fixed sampling size a method called `reservoir` is used. The reservoir is filled up first with as many elements as requested and then streaming the rest, randomly swapping elements in the reservoir.

Find more about this interesting technique in the samples documentation.

3.5.6 Functions with optional parameters

A couple of functions in DuckDB, such as `read_json_auto` for example, have some required parameters and one or more parameters with sensible defaults that are optional. Aforementioned example has 17 parameters, you can get a list of them with

```
SELECT DISTINCT unnest(parameters)
FROM duckdb_functions()
WHERE function_name = 'read_json_auto';
```

We are using a distinct here because there's a couple of overloads with different types, too. Luckily, DuckDB supports named optional arguments. Assume you want to specify the `dateformat` only, you would use the `name=value` syntax:

Listing 3.24 Using named parameters

```
echo '{"foo": "21.9.1979"}' > 'my.json'
duckdb -s \
"SELECT * FROM read_json_auto(
  'my.json',
  dateformat='%d.%M.%Y' -- #1
)"
```

#1 This is using the named parameter `dateformat`

DuckDB is able to parse the non-iso-formatted string into a proper date as we used the `dateformat` parameter:

```
┌────────────┐
│    foo     │
│    date    │
├────────────┤
│ 1979-01-21 │
└────────────┘
```

3.6 Summary

- SQL queries are composed of several statements which are in turn composed of clauses. Queries are categorized as *Data Definition Language* (DDL) or *Data Manipulation Language* (DML).
- DML queries covers creating, reading, updating and deleting rows.
- Manipulation of data is not only about changing persistent state, but transforming existing relations into new ones, hence reading data falls also under DML.
- DDL queries such as `CREATE TABLE` and `CREATE VIEW` are used in DuckDB to create a persistent schema. This is in line with any other relational database and is independent whether DuckDB is started with database stored on disk or in-memory.
- A rigid schema makes data inconsistencies more visible: blindly ingesting data with inconsistencies will fail due to constraint errors.
- Constraint errors can be mitigated with appropriate actions defined `ON CONFLICT` when creating or updating rows.
- DuckDB makes SQL even easier to write with innovations like `SELECT * EXCLUDE()` and `SELECT * REPLACE()` and more intuitive alias usage.

4 Advanced aggregation and analysis of data

This chapter covers

- Preparing, cleaning and aggregating data while ingesting
- Using window functions to create new aggregates over different partitions of any dataset
- Understanding the different types of sub-queries
- Using Common Table Expressions (CTEs)
- Applying filters to any aggregate

The goal of this chapter is to give you some ideas on how an analytical database such as DuckDB can be used to provide reports that would take a considerably larger amount of code written in an imperative programming language. While we will build upon the foundation laid in chapter 3, we will leave a simple `SELECT xzy FROM abc` behind quickly. Investing your time in learning modern SQL won't be wasted. The constructs presented here can be used everywhere where DuckDB can be run or embedded and therefore enrich your application.

4.1 Pre-aggregate data while ingesting

Let's move forward with our example scenario. In section 3.4.1 we worked with the data for a photovoltaic grid that—while having some consistency issues—was a good fit for our schema and idea. Remember, the goal is to store measurements in intervals of 15 minutes. If you look at the other datasets you have downloaded throughout section 3.2.1 you will notice that some come in intervals other than 15 minutes. One quick way to peek into the files is the `tail` command, returning the last n-lines of a file (`head` would work as well). Using it on `2020_10.csv` shows that this file contains measurements in 1-minute intervals:

```
> duckdb -s ".maxwidth 40" -s "FROM read_csv_auto('2020_10.csv') LIMIT 3"
┌─────────┬─────────────────────┬───┬──────────────────┬──────────────────┐
│ SiteID  │      Date-Time      │ … │ module_temp_3    │ poa_irradiance   │
│ int64   │      timestamp      │   │    double        │     double       │
├─────────┼─────────────────────┼───┼──────────────────┼──────────────────┤
│      10 │ 2020-01-23 11:20:00 │ … │           14.971 │           748.36 │
│      10 │ 2020-01-23 11:21:00 │ … │           14.921 │           638.23 │
│      10 │ 2020-01-23 11:22:00 │ … │           14.895 │           467.67 │
├─────────┴─────────────────────┴───┴──────────────────┴──────────────────┤
│ 3 rows                                        16 columns (4 shown)     │
└────────────────────────────────────────────────────────────────────────┘
```

And of course, `2020_1200.csv` has another interval, this time 5 minutes but also the overall structure looks different:

```
> duckdb -s ".maxwidth 40" -s "FROM read_csv_auto('2020_1200.csv') LIMIT 3"
┌─────────┬─────────────────────┬───┬──────────────────┬──────────────┐
│ SiteID  │      Date-Time      │ … │ ac_power_metered │ power_factor │
│ int64   │      timestamp      │   │      int64       │    double    │
├─────────┼─────────────────────┼───┼──────────────────┼──────────────┤
│    1200 │ 2020-01-01 00:00:00 │ … │               20 │        0.029 │
│    1200 │ 2020-01-01 00:05:00 │ … │               20 │        0.029 │
│    1200 │ 2020-01-01 00:10:00 │ … │               20 │        0.029 │
├─────────┴─────────────────────┴───┴──────────────────┴──────────────┤
│ 3 rows                                      6 columns (4 shown)    │
└────────────────────────────────────────────────────────────────────┘
```

Remember, those are datafiles from the same pool. Even those are inconsistent between different sources. Data analytics is quite often about dealing with those exact same problems. Let's use one of the many functions DuckDB offers to deal with dates, times and timestamps, in this case `time_bucket()`. `time_bucket()` truncates timestamps to a given interval and aligns them to an optional offset, creating a time bucket. Time buckets are a powerful mechanism for aggregating sensor readings and friends. Together with `GROUP BY` and `avg` as aggregate functions, we can prepare and eventually ingest the data according to our requirements: We create time buckets in a 15-minute interval and compute the average power produced of all readings that fall into a specific bucket.

When you look at the query you'll notice a `CASE WHEN THEN ELSE END` construct, a `CASE`-Statement, which works like an `if/else` construct. What it does here is turn readings with a value lower than zero, or with no value at all into zero before computing the average. That's one of the oddities of this dataset: maybe the sensor had issues, maybe the network had issues, you'll never know, but you have to deal with the data. Here we decided that it is ok to treat `null` values like negative values and cap them to zero. In cases that throws off your calculation, you might consider a `FILTER` for the aggregate. We will discuss this in the section Section 46.3.

Listing 4.1 Cleaning and transforming data during ingestion

```
INSERT INTO readings(system_id, read_on, power)
SELECT any_value(SiteId), -- #1
       time_bucket(
           INTERVAL '15 Minutes',
           CAST("Date-Time" AS timestamp)
       ) AS read_on, -- #2
       avg(
           CASE
               WHEN ac_power < 0 OR ac_power IS NULL THEN 0
               ELSE ac_power END) -- #3
FROM
    read_csv_auto(
        'https://developer.nrel.gov/api/pvdaq/v3/' ||
        'data_file?api_key=DEMO_KEY&system_id=10&year=2019'
    )
GROUP BY read_on
ORDER BY read_on;
```

#1 This picks any value of the column `SiteId` from the CSV-File. The files are per system, which means that this column is the same in each row, so picking any one of them is correct. Applying `any_value()` is necessary as we compute an aggregate (`avg`)

#2 This truncates the timestamp to a quarter-hour, notice how we explicitly cast the column to a timestamp with standard SQL syntax; in addition, the transformed value gets an alias (`read_on`)

#3 Here, `avg` computes the average of all readings in the bucket created above because we group the result by this bucket.

The imports for the remaining dataset are identical, you will want to change the filename in the `FROM` clause accordingly.

> **TIP** There are many more date and time based functions in DuckDB, when in doubt, have a look at the reference documentation: https://duckdb.org/docs/sql/functions/timestamp. You will be able to parse nearly any string into a proper date or timestamp.

Whether you don't ingest at all and do all kind of analytics in-memory based on external files, want to aggregate to some extent during ingestion, or only aggregate during analysis is usually a trade-off, depending among other things on the size of your dataset, goals for long-time storage, further processing needs. Trying to make a general applicable solution here is therefore bound to fail. In the scenario here we decided to both ingest and aggregate for educational purposes as well as keep the dataset small enough to be shareable.

4.2 Summarizing data

You usually want to know some characteristics of a new dataset before going into an in-depth analysis of it, such as the number of values (in our example how many readings), the distribution and magnitude of numerical values (without knowing if we are dealing in Watt or Kilowatt our reports would be blatantly wrong) and the interval size of time series.

DuckDB has the unique `SUMMARIZE` command that quickly gives you this information about any dataset. Run `SUMMARIZE readings;` in your database. Your result should be similar to this:

```
┌─────────────┬──────────────┬─────────────────────┬───┬─────────┬─────────┐
│ column_name │ column_type  │         max         │ … │   q75   │ count   │
│   varchar   │   varchar    │       varchar       │   │ varchar │ int64   │
├─────────────┼──────────────┼─────────────────────┼───┼─────────┼─────────┤
│ system_id   │ INTEGER      │ 1200                │ … │ 1200    │ 151879  │
│ read_on     │ TIMESTAMP    │ 2020-06-26 11:00:00 │ … │         │ 151879  │
│ power       │ DECIMAL(10,3)│ 133900.000          │ … │ 5125    │ 151879  │
└─────────────┴──────────────┴─────────────────────┴───┴─────────┴─────────┘
```

There are many more columns, but we abbreviated the list for readability. Switch your CLI to line mode by running `.mode line` and then summarizing a subset of the readings with `SUMMARIZE SELECT read_on, power FROM readings WHERE system_id = 1200;`:

```
    column_name = read_on
    column_type = TIMESTAMP
            min = 2019-01-01 00:00:00
            max = 2020-06-26 11:00:00
  approx_unique = 50833
            avg =
            std =
            q25 =
            q50 =
            q75 =
          count = 52072
null_percentage = 0.0%

    column_name = power
    column_type = DECIMAL(10,3)
            min = 0.000
            max = 47873.333
  approx_unique = 6438
            avg = 7122.5597121293595
            std = 11760.089219586542
            q25 = 20
            q50 = 27
            q75 = 9532
          count = 52072
null_percentage = 0.0%
```

`SUMMARIZE` works directly on tables, but as shown above, on query results, too. You don't even have to ingest data at all before applying `SUMMARIZE`, it can be run against a CSV- or Parquet-file as well.

4.3 On sub-queries

Imagine you want to compute the average of the total power produced by the systems you manage. For that, you would need to apply two aggregate functions, `avg` and `sum`. It turns out, you cannot nest them. Aa naive approach like `SELECT avg(sum(kWh)) FROM v_power_per_day GROUP BY system_id` fails with `Error: Binder Error: aggregate function calls cannot be nested`. You need to stage that computation and a sub-query is one way to achieve this:

Listing 4.2 A sub-query being used to compute nested aggregates

```
SELECT avg(sum_per_system)
FROM (
    SELECT sum(kWh) AS sum_per_system
    FROM v_power_per_day
    GROUP BY system_id
);
```

This statement now dutifully returns `avg(sum_per_system) = 133908.087`. The inner query in this statement has two characteristics:

- It returns several rows
- It does not depend on values from the outer query

This query is called an **uncorrelated sub-query**. An uncorrelated subquery is just a query nested inside another one, and operates as if the outer query executed on the results of the inner query.

Now on to the next task you might have: on which day and for which system was the highest amount of power produced? One way to solve this issue is using a subquery as the right-hand-side of a comparison in the ´WHERE` clause.

Listing 4.3 A sub-query being used inside a condition selecting the row containing the maximum value

```
SELECT read_on, power
FROM readings
WHERE power = (SELECT max(power) FROM readings);
```

This sub-query is different from the first one in that it only returns a single, scalar value. It is called **scalar, uncorrelated sub-query**.

> **NOTE** `arg_min` and `arg_max` are aggregate functions that compute an expression of the row in which the minimum or maximum value appears. In case you are interested in only one expression, they are the preferable solution compared to any sub-query for tasks like above. If you are interested in more than one expression or evaluating other values than minimum or maximum values you won't get around sub-queries in conditions.

The result essentially reads: "The maximum output of 133900W has been produced at 5 different times":

```
┌─────────────────────┬─────────────────┐
│       read_on       │      power      │
│      timestamp      │ decimal(10,3)   │
├─────────────────────┼─────────────────┤
│ 2019-05-08 12:15:00 │    133900.000   │
│ 2019-05-23 10:00:00 │    133900.000   │
│ 2019-05-23 11:30:00 │    133900.000   │
│ 2019-05-28 11:45:00 │    133900.000   │
│ 2020-04-02 11:30:00 │    133900.000   │
└─────────────────────┴─────────────────┘
```

What if we wanted to determine the maximum power and reading time on a per-system basis? This will be tricky to do with the original subquery, because that only shows us the values for the overall max power production. We would need the sub-query to return different values for different rows; to do this we can use a correlated sub-query, which uses the fields from the outer query inside the inner one like this:

Listing 4.4 Using correlated, scalar sub-queries

```
SELECT system_id, read_on, power
FROM readings r1
WHERE power = (
    SELECT max(power)
    FROM readings r2
    WHERE r2.system_id = r1.system_id -- #1
)
ORDER BY ALL;
```

#1 This is the condition that correlates the sub-query to the outer query, not the comparison of the `power` value

Now this sub-query is a **scalar, correlated sub-query**. The inner query is related to the outer query in that way that the database must evaluate it for every row of the outer query.

In the result we see the 5 days for the highest value overall again and the highest values produced for systems 10 and 1200 now too:

```
┌─────────────┬─────────────────────┬─────────────────┐
│ system_id   │       read_on       │     power       │
│   int32     │      timestamp      │ decimal(10,3)   │
├─────────────┼─────────────────────┼─────────────────┤
│          10 │ 2019-02-23 12:45:00 │       1109.293  │
│          34 │ 2019-05-08 12:15:00 │     133900.000  │
│          34 │ 2019-05-23 10:00:00 │     133900.000  │
│          34 │ 2019-05-23 11:30:00 │     133900.000  │
│          34 │ 2019-05-28 11:45:00 │     133900.000  │
│          34 │ 2020-04-02 11:30:00 │     133900.000  │
│        1200 │ 2020-04-16 12:15:00 │      47873.333  │
└─────────────┴─────────────────────┴─────────────────┘
```

When used as expression, sub-queries may be rewritten as joins—with the computation of nested aggregates being the exception. For the last example it would look like this:

Listing 4.5 An uncorrelated sub-query join with the outer table

```
SELECT r1.system_id, read_on, power
FROM readings r1
JOIN (
    SELECT r2.system_id, max(power) AS value
    FROM readings r2
    GROUP BY ALL
) AS max_power ON (
    max_power.system_id = r1.system_id AND
    max_power.value = r1.power
  )
ORDER BY ALL;
```

It’s up to the reader to judge whether this adds to readability or not. In other relational databases people often do this, as the evaluation of a correlated sub-query for every row in a large table might be slow. DuckDB, on the other hand, uses a sub-query decorrelation optimizer that always makes sub-queries independent of outer queries, thus allowing users to freely use sub-queries to create expressive queries without having to worry about manually rewriting sub-queries into joins. It is not always possible to manually decorrelate certain sub-queries by rewriting the SQL. Internally DuckDB uses special types of joins that will decorrelate all sub-queries. In fact, DuckDB does not have support for executing sub-queries that are not decorrelated.

The good thing for you is the fact that you can focus on the readability and expressiveness of your queries, and the business problem you are trying to solve and don’t need to worry about what type of sub-query you use.

4.3.1 Sub-queries as expressions

All forms of sub-queries, both correlated and uncorrelated, that are not used as a relation in a `JOIN`, are expressions. As such, there are many more operators that might be used with them. The `=` operator and the inequality operators `<`, `<=`, `>=` and `>` require the sub-query to be a scalar sub-query, returning exactly one row. Working with both scalar and non-scalar sub-queries, additional operators exists. These are `IN`, `EXISTS`, `ANY` and `ALL` and they work by doing set comparisons.

Sub-queries can also be used in set-comparisons answering questions like "give me all rows that compare successfully to all or any of the rows returned by another query".

The artificial examples in this section will all return `v = 7`.

'EXISTS'

You might want to select all the rows of a table that have a value that might exist inside one row of another table. For this, you can use the `EXISTS` expression:

Listing 4.6 Sub-query used with the `EXISTS` expression

```
.mode line
SELECT * FROM VALUES (7), (11) s(v)
WHERE EXISTS (SELECT * FROM range(10) WHERE range = v);
```

'IN'

`EXISTS` can usually be rewritten as an uncorrelated sub-query using the `IN` operator: When the outer value is contained at least once in the results of the sub-query, this operator evaluates to true.

Listing 4.7 Sub-query used with the `IN` expression

```
.mode line
SELECT * FROM VALUES (7), (11) s(v)
WHERE v IN (SELECT * FROM range(10));
```

This is useful to know when you work with other relational databases than DuckDB that might not do all kinds of optimizations on sub-queries.

'ANY'

The `IN` operator works with an equal comparison of each value. You might find yourself in a situation in which you want to answer whether any value does satisfy an inequality condition. Here, you need to use the `ANY` operator together with the desired comparison. When the comparison of the outer value with any of the inner values evaluates to true, the whole expression evaluates to true.

Listing 4.8 Sub-query used with the ANY expression

```
.mode line
SELECT * FROM VALUES (7), (11) s(v)
WHERE v <= ANY (SELECT * FROM range(10)); -- #1
```

#1 Please take note of the additional comparison prior to ANY

'ALL'

And last but not least, the `ALL` operator, which evaluates to true when the comparison of the outer value with all of the inner values evaluates to true. It helps you find rows in which a value satisfies a comparison between all values of a sub-query. While you can replace `= ANY()` with `IN()` there is no such simplification for the `ALL` operator.

Listing 4.9 Sub-query used with the ALL expression

```
.mode line
SELECT * FROM VALUES (7), (11) s(v)
WHERE v = ALL (SELECT 7);
```

4.4 Grouping sets

In listing 3.1 we created a table named `readings` which contains the date, the time, and the actual value of power produced at that time. We also suggested several example datasets from the National Renewable Energy Laboratory to import. When looking at such a dataset it is always helpful getting an overview about the minimum and maximum values of an attribute, or maybe the average. Sometimes you might have outliers in there that you want to delete, or maybe you made a mistake with the units. The easiest way to compute that is just using them in one query, without any `GROUP BY` clause, so that the aggregation happens in one bucket: the whole table.

Listing 4.10 Using various aggregates to check if the imports make sense

```
SELECT count(*),
       min(power) AS min_W, max(power) AS max_W,
       round(sum(power) / 4 / 1000, 2) AS kWh -- #1
FROM readings;
```

#1 Going from power produced in Watt and read in a 15-minute interval to power produced per hour in kWh is expressed by summing the values up, dividing them by 4 to go to Watts per hour and then by 1000 for Kilowatt per hour.

If you followed the suggestion, your `readings` table should have key figures like the following, which is the result of the query in above:

```
┌──────────────┬───────┬────────────┬────────────┐
│ count_star() │ min_W │   max_W    │    kWh     │
├──────────────┼───────┼────────────┼────────────┤
│ 151879       │ 0.000 │ 133900.000 │ 401723.22  │
└──────────────┴───────┴────────────┴────────────┘
```

The readings seem to be reasonable, even the minimum value of zero: there is just no production during nighttime. As we already learned about the `GROUP BY` clause in image 3.2 we could go further and have a look at the production of kilowatts per hour and system. We will also select the number of readings per system. We imported several years, truncated the readings to 15-minute intervals, so we should find roughly 35040 readings per year. A `GROUP BY system_id, year` confirms this assumption as shown with the next statement:

Listing 4.11 A plain `GROUP BY` with essentially one set of grouping keys ()

```
SELECT year(read_on) AS year,
       system_id,
       count(*),
       round(sum(power) / 4 / 1000, 2) AS kWh
FROM readings
GROUP BY year, system_id
ORDER BY year, system_id;
```

The result adds up. We did have a bunch of invalid values and the second year ends halfway in 2020:

```
┌──────┬───────────┬──────────────┬────────────┐
│ year │ system_id │ count_star() │    kWh     │
├──────┼───────────┼──────────────┼────────────┤
│ 2019 │ 10        │ 33544        │ 1549.34    │
│ 2019 │ 34        │ 35040        │ 205741.9   │
│ 2019 │ 1200      │ 35037        │ 62012.15   │
│ 2020 │ 10        │ 14206        │ 677.14     │
│ 2020 │ 34        │ 17017        │ 101033.35  │
│ 2020 │ 1200      │ 17035        │ 30709.34   │
└──────┴───────────┴──────────────┴────────────┘
```

Now what about the totals, i.e., the total number of readings as well as the total power production per year and system and overall? Or in other words: can we create a drill-down report, showing different levels of detail per group? While we could now enter the numbers into a calculator one by one and sum them manually, or write an additional count-query without a grouping key like the initial one, there's a better option called "grouping sets":

Listing 4.12 Explicitly using `GROUPING SETS`

```
SELECT year(read_on) AS year,
       system_id,
       count(*),
       round(sum(power) / 4 / 1000, 2) AS kWh
FROM readings
GROUP BY GROUPING SETS ((year, system_id), year, ())
ORDER BY year NULLS FIRST, system_id NULLS FIRST;
```

Before we dissect `GROUP BY GROUPING SETS ((system_id, year), year, ())` let's have a look at the result first:

```
┌──────┬───────────┬──────────────┬────────────┐
│ year │ system_id │ count_star() │    kWh     │
├──────┼───────────┼──────────────┼────────────┤
│      │           │ 151879       │ 401723.22  │
│ 2019 │           │ 103621       │ 269303.39  │
│ 2019 │ 10        │ 33544        │ 1549.34    │
│ 2019 │ 34        │ 35040        │ 205741.9   │
│ 2019 │ 1200      │ 35037        │ 62012.15   │
│ 2020 │           │ 48258        │ 132419.83  │
│ 2020 │ 10        │ 14206        │ 677.14     │
│ 2020 │ 34        │ 17017        │ 101033.35  │
│ 2020 │ 1200      │ 17035        │ 30709.34   │
└──────┴───────────┴──────────────┴────────────┘
```

The grouping sets created several buckets to compute the aggregates over:

- A bucket defined by the combined values of `system_id` and `year` (6 different combinations in our example, thus leading to 6 rows)
- A bucket defined by the `year` alone. For keys not included in this but in other sets, `null` values are provided (here for the `system_id`)
- The last one (`()`) can be described as the empty bucket or group: `null` values are provided for all other keys

The result contains everything that listing 4.11 returned plus the number of readings per year (grouping by `year` alone) plus the overall count (grouping by nothing).

The same result can be achieved by using the shorthand clause `ROLLUP`. The `ROLLUP` clauses produces the sets we discussed above automically for you as n+1 grouping sets where n is the amount of terms in the `ROLLUP` clause:

Listing 4.13 Using `GROUP BY ROLLUP`

```
SELECT year(read_on) AS year,
       system_id,
       count(*),
       round(sum(power) / 4 / 1000, 2) AS kWh
FROM readings
GROUP BY ROLLUP (year, system_id)
ORDER BY year NULLS FIRST, system_id NULLS FIRST;
```

If we want to see the totals by system in all years this is quite possible, too. Instead of falling back from `ROLLUP` to `GROUP BY GROUPING SETS` and manually adding, you can use `GROUP BY CUBE`. `GROUP BY CUBE` will not produce subgroups but actual combinations (2^n grouping sets). In our example `(year, system_id)`, `(year)`, `(system)` and `()`:

Listing 4.14 Using `GROUP BY CUBE`

```
SELECT year(read_on) AS year,
       system_id,
       count(*),
       round(sum(power) / 4 / 1000, 2) AS kWh
FROM readings
GROUP BY CUBE (year, system_id)
ORDER BY year NULLS FIRST, system_id NULLS FIRST;
```

produces now:

```
┌──────┬───────────┬──────────────┬────────────┐
│ year │ system_id │ count_star() │    kWh     │
├──────┼───────────┼──────────────┼────────────┤
│      │           │ 151879       │ 401723.22  │
│      │ 10        │ 47750        │ 2226.48    │
│      │ 34        │ 52057        │ 306775.25  │
│      │ 1200      │ 52072        │ 92721.48   │
│ 2019 │           │ 103621       │ 269303.39  │
│ 2019 │ 10        │ 33544        │ 1549.34    │
│ 2019 │ 34        │ 35040        │ 205741.9   │
│ 2019 │ 1200      │ 35037        │ 62012.15   │
│ 2020 │           │ 48258        │ 132419.83  │
│ 2020 │ 10        │ 14206        │ 677.14     │
│ 2020 │ 34        │ 17017        │ 101033.35  │
│ 2020 │ 1200      │ 17035        │ 30709.34   │
└──────┴───────────┴──────────────┴────────────┘
```

We have now a complete overview of our power production readings, in a single, compact query instead of several queries. All the additional drill-downs that we added on the way can be expressed with grouping sets. The minimum and maximum values have only been omitted to keep the listing readable.

4.5 Window functions

Windows and functions applied over windows are an essential part of modern SQL and analytics. Window functions in general let you look at other rows. Normally, an SQL function can only see the current row at a time, unless you’re aggregating. In that case however you reduce the number of rows.

Unlike a regular aggregate function, the use of a window function does not cause rows to become grouped into a single output row—the rows retain their separate identities. If you want to peek at other rows, you would use a window function. A window is introduced by the `OVER()` clause, following the function you want to apply to the data inside that window. The window itself is the definition of the rows that are worked on, and you can think about it as a window that moves along a defined order with a defined size of rows over your dataset. Windowing works by breaking a relation up into independent partitions, optionally ordering those partitions, and then computing a new column for each row as a function of the nearby values.

For looking at all rows, you can use an empty window `OVER ()`. If you want to look at all rows that have the same value matching another field, use a partition for that field. And last but not least, if you want to look at nearby rows, you can use a frame.

The size of a window is not equal to the size of a partition, both can be defined independently. Eventually, the contents of a window are fed to a function to compute new values. While there are a couple of dedicated functions that work only in the context of windows, all regular aggregate functions might be used as window functions.

This allows use cases such as

- Ranking
- Computing independent aggregates per window
- Computing running totals per window
- Computing changes by accessing preceding or following rows via `lag` or `lead`

Let's have a look at a concrete example. Imagine you want to retrieve the system and every time at which the top 3 amounts of power were produced at a quarter-hour. One naive approach would be ordering the results by power produced and limit to 3: `SELECT * FROM readings ORDER BY power DESC LIMIT 3;`.

```
┌─────────────┬──────────────────────┬─────────────────┐
│ system_id │       read_on        │     power     │
│   int32   │      timestamp       │ decimal(10,3) │
├─────────────┼──────────────────────┼─────────────────┤
│        34 │ 2019-05-08 12:15:00 │    133900.000 │
│        34 │ 2019-05-23 10:00:00 │    133900.000 │
│        34 │ 2019-05-23 11:30:00 │    133900.000 │
└─────────────┴──────────────────────┴─────────────────┘
```

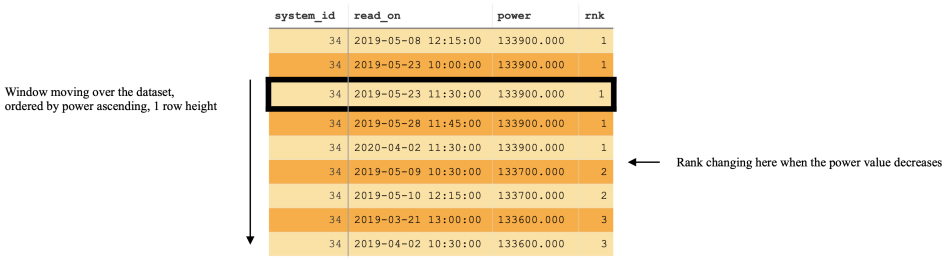
While the above results present readings for system 34 at different dates, you notice that they have the same value in the `power` column. This might be good enough but is not necessarily what we have been asked for. In regard to the raw value of power produced, it is only the top one ranking, not all readings for the top 3 power values. For computing a proper top 3, we will use the window function `dense_rank()`. This function computes the rank for a row without skipping ranks for equal rankings. `dense_rank` returns the rank of the current row without gaps. That means that after 5 rows having the rank 1, the next rank will be 2, not 6. If you need the latter, you would use `rank` instead.

Listing 4.15 A proper top-n query

```
WITH ranked_readings AS (
    SELECT *,
           dense_rank()
             OVER (ORDER BY power DESC) AS rnk  -- #1
    FROM readings
)
SELECT *
FROM ranked_readings
WHERE rnk <= 3;
```

#1 Here the window is opened over one row each, ordered by the amount of power in descending order

The result looks very different now, with 3 different, decreasing values for `power` as shown in figure 4.1.



system_id	read_on	power	rnk
34	2019-05-08 12:15:00	133900.000	1
34	2019-05-23 10:00:00	133900.000	1
34	2019-05-23 11:30:00	133900.000	1
34	2019-05-28 11:45:00	133900.000	1
34	2020-04-02 11:30:00	133900.000	1
34	2019-05-09 10:30:00	133700.000	2
34	2019-05-10 12:15:00	133700.000	2
34	2019-03-21 13:00:00	133600.000	3
34	2019-04-02 10:30:00	133600.000	3

Figure 4.1 The most simple window possible over the power readings

We will revisit the statement above when we learn about the `QUALIFY` clause in section Section 46.2, avoiding the somewhat odd condition in the `WHERE` clause that filters on the rank.

The `ORDER` clause as part of the window definition inside the `OVER()` clause is optional and unlike the `ORDER BY` clause at the end of a statement, it does not sort the query result. When used as part of the `OVER()` clause, `ORDER BY` defines in which order window functions are executed. If `ORDER BY` is omitted, the window function is executed in an arbitrary order. In our example above it would make no sense to omit it, as an unordered, dense rank would always be one. We will see an example in the next section where we can safely omit `ORDER BY`.

4.5.1 Defining partitions

The ranked power values above are better, but they are not yet particularly helpful, as the systems have production values that are orders of magnitudes different. Computing ranks without differentiating the systems might not be what are after. What we actually need in this case is having the top 3 readings per system, each system making up its own partition of the data. Partitioning breaks the relation up into independent, unrelated pieces in which the window function is applied. If we don't define how a partition is made up by using the `PARTITION BY` clause the entire relation is treated as a single partition. Window functions cannot access values outside the partition containing the row they are being evaluated at.

Requesting the top n measurements per system would be a partitioning task. For the sake of readability of the results we requested only the top-two power-production values per system.

Listing 4.16 Applying a partition to a window

```
WITH ranked_readings AS (
    SELECT *,
           dense_rank()
             OVER ( -- #1
               PARTITION BY system_id -- #2
               ORDER BY power DESC
             ) AS rnk
    FROM readings
)
SELECT * FROM ranked_readings WHERE rnk <= 2
ORDER BY system_id, rnk ASC;
```

#1 Starting the definition of the moving window
#2 Defining a partition: the window

Look closely at how the number of ranks repeat now in the result in figure 4.2. They have now been computed individually inside the respective partitions which makes a pretty different statement about the dataset.

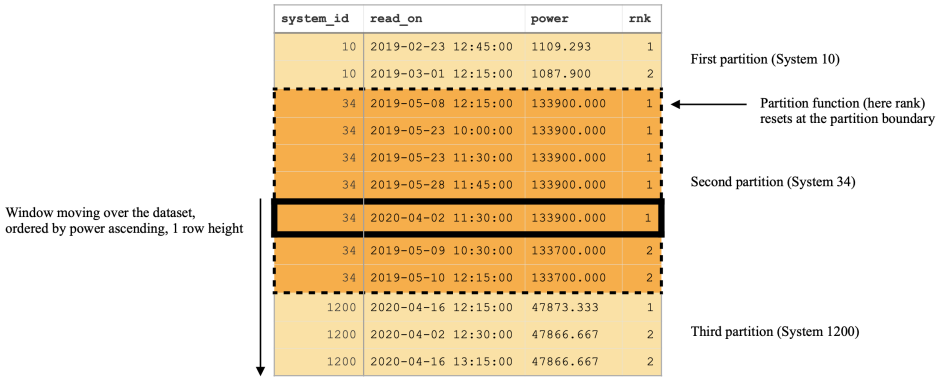


Figure 4.2 Partitioning the data before applying a window

We see ranks 1 and 2 for all systems, system 34 has 5 times a top production of 133900W and the second place twice. System 1200 has only one first place, but two seconds. The window was partitioned by the systems and then ordered by the value of power produced.

Of course ranking tasks are not the only things that can be applied within partitions. Aggregate functions like `avg`, `sum`, `max` and `min` are other great candidates to be used in a windowing context. The difference of using aggregates within a window context is the fact that they don't change the number of rows being produced. Let's say you want to select both the production on each system each day and in an additional column, the average overall production of system. You might think of using `GROUP BY ROLLUP` and you would not be wrong. That grouping set would however be quite big (`GROUP BY ROLLUP (system_id, day, kwh)`) and not produce the average value in an additional column, but produce additional rows. The value that you would be looking for (the overall production per system) would be found in the rows that have a value for the system and no value for the day.

One way to avoid dealing with additional rows would be a self-join, in which you select the desired aggregate grouping by a key to join the same table again. While it does produce the results we want, it will be hard to read and does most likely not perform well as the whole table would be scanned twice. Using `avg` in a partitioned window context is much easier to read and will perform well. The aggregate, in this case `avg(kWh)`, is computed over the window that follows, it does not change the number of rows and will be present in each row. It will be computed for every system as defined by the partition:

```
SELECT *,
       avg(kWh) -- #1
         OVER (
           PARTITION BY system_id
         ) AS average_per_system
FROM v_power_per_day;
```

#1 Computing an aggregate over a partition

And you will find the requested value in an additional column:

```
┌───────────┬────────────┬────────┬────────────────────┐
│ system_id │    day     │  kWh   │ average_per_system │
│   int32   │    date    │ double │       double       │
├───────────┼────────────┼────────┼────────────────────┤
│        10 │ 2019-01-01 │   2.19 │  4.444051896207586 │
│        10 │ 2019-01-04 │   5.37 │  4.444051896207586 │
│         · │     ·      │     ·  │          ·         │
│         · │     ·      │     ·  │          ·         │
│         · │     ·      │     ·  │          ·         │
│      1200 │ 2019-07-25 │ 232.37 │ 170.75771639042347 │
│      1200 │ 2019-04-29 │ 210.97 │ 170.75771639042347 │
├───────────┴────────────┴────────┴────────────────────┤
│ 1587 rows (4 shown)                      4 columns │
└────────────────────────────────────────────────────┘
```

Note that we omitted the `ORDER BY` inside the window definition, as it is irrelevant for the average value in which order values are fed to the aggregate.

As a rule of the thumb you probably want to use a window function every time you consider writing a self-join like the one above for adding aggregates to your query without changing the row count.

4.5.2 Framing

Top-N queries are useful, for example, if you happen to have a streaming service and want to present the Top-N charts. A more interesting question in our example is this: "What is the moving 7-day average of energy produced system-wide?" To answer this question we must

- Aggregate the readings per 15-minutes interval into days (grouping and summing)
- Partition by day and systems
- Create frames of 7 days

This is where framing comes into play. Framing specifies a set of rows relative to each row where the function is evaluated. The distance from the current row is given as an expression either preceding or following the current row. This distance can either be specified as an integral number of rows or as a range delta expression from the value of the ordering expression.

For the purpose of readability and to keep the following examples focused on the window definitions, we will use the view `v_power_per_day` defined in chapter 3, that returns the amount of energy produced in kWh per system and day. We could equally express `v_power_per_day` as a CTE.

The following statement computes the average power over a window per system that moves along the days and is 7 days wide (3 days before, the actual day and 3 days ahead). The statement utilizes all options for defining a window.

Listing 4.17 Using a range partition for applying a window-function

```
SELECT system_id,
       day,
       kWh,
       avg(kWh) OVER (
            PARTITION BY system_id -- #1
            ORDER BY day ASC -- #2
            RANGE BETWEEN INTERVAL 3 Days PRECEDING -- #3
                      AND INTERVAL 3 Days FOLLOWING
       ) AS "kWh 7-day moving average"
FROM v_power_per_day
ORDER BY system_id, day;
```

#1 The window should move over partitions defined by the system id
#2 Ordered by day
#3 With a size of 7 days in total

The result will have as many rows as there are full days in the source readings, so we can only show a subset as an example. Figure 4.3 demonstrates the size of the window and how rows are included.

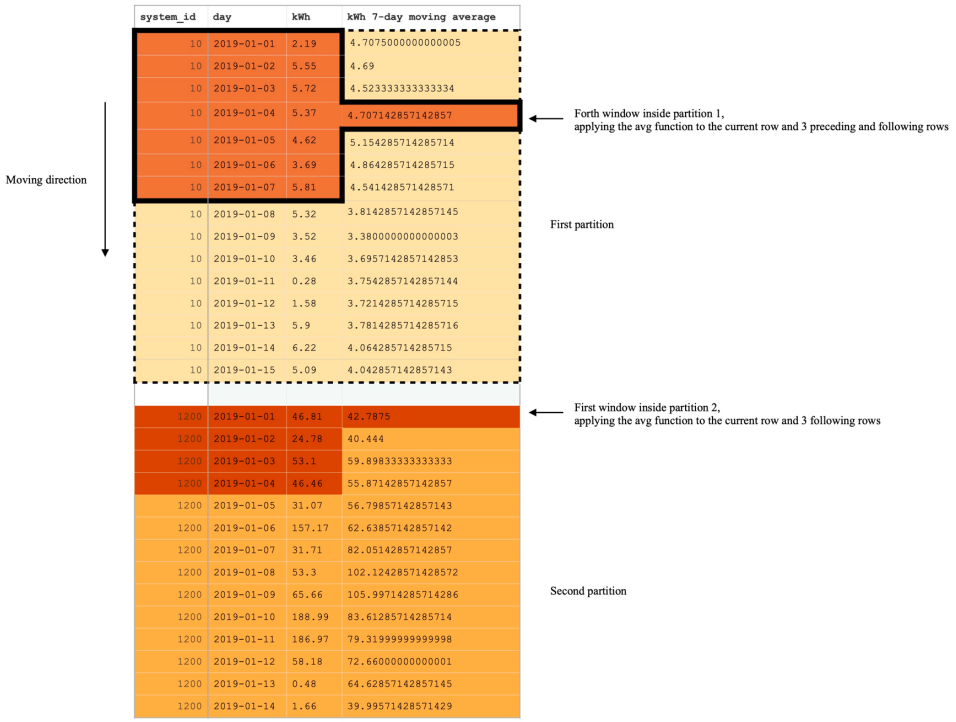


Figure 4.3 The result of framing a window

4.5.3 Named windows

Window definitions can be pretty complex as we have just learned while discussing windows with ranges. They can include the definition of the partition, the order and the actual range of a window. Sometimes you are interested in more than just one aggregate over a given window. It would be a tedious task, repeating the window definition over and over again.

For our domain—measuring power production from photovoltaic systems—we could use quantiles to create a report that essentially takes in both the seasons and the weather by computing the quantiles over a 7-day-window per month. Sometimes a broad monthly average might be enough, but a chart would represent only a relatively smooth curve changing with the months. The fluctuation of the amount of power produced is higher throughout the changing weather in a week. Outliers and runaway values would be better caught and represented by quantiles. The result can easily be used to create a moving box-and-whisker plot for example.

We need three aggregates (`min`, `max`, and `quantiles`) for caching outliers and computing the quantiles, and we don't want to define the window each time. We basically take the definition from listing 4.17 and add the month of the reading to the partition. Otherwise, the window definition is the same. We move the definition after the `FROM` clause and name it `seven_days`. It can be referenced from as many aggregates than as necessary:

Listing 4.18 Using a named window with a complex order and partition

```
SELECT system_id,
       day,
       min(kWh) OVER seven_days AS "7-day min", -- #1
       quantile(kWh, [0.25, 0.5, 0.75]) -- #2
         OVER seven_days AS "kWh 7-day quartile",
       max(kWh) OVER seven_days AS "7-day max",
FROM v_power_per_day
WINDOW -- #3
    seven_days AS (
        PARTITION BY system_id, month(day)
        ORDER BY day ASC
        RANGE BETWEEN INTERVAL 3 Days PRECEDING
                  AND INTERVAL 3 Days FOLLOWING
    )
ORDER BY system_id, day;
```

#1 Referencing the window defined after the `FROM` clause

#2 The `quantile` function takes in the value for which the quantiles should be computed and a list of the desired quantiles

#3 The window clause has to be specified after the `FROM` clause and the definition of window itself follows inline windows

The result now showcases a structured column type, "kWh 7-day quartile":

```
┌─────────────┬─────────────┬─────────────┬──────────────────────────┬─────────────┐
│system_id    │    day      │ 7-day min   │    kWh 7-day quartile    │ 7-day max   │
│  int32      │    date     │  double     │         double[]         │  double     │
├─────────────┼─────────────┼─────────────┼──────────────────────────┼─────────────┤
│          10 │ 2019-01-01  │        2.19 │ [2.19, 5.37, 5.55]       │        5.72 │
│          10 │ 2019-01-02  │        2.19 │ [4.62, 5.37, 5.55]       │        5.72 │
│          10 │ 2019-01-03  │        2.19 │ [3.69, 4.62, 5.55]       │        5.72 │
│          10 │ 2019-01-04  │        2.19 │ [3.69, 5.37, 5.72]       │        5.81 │
│          10 │ 2019-01-05  │        3.69 │ [4.62, 5.37, 5.72]       │        5.81 │
│           · │      ·      │           · │          ·               │           · │
│           · │      ·      │           · │          ·               │           · │
│           · │      ·      │           · │          ·               │           · │
│        1200 │ 2020-06-22  │      107.68 │ [149.11, 191.61, 214.68] │       279.8 │
│        1200 │ 2020-06-23  │         0.0 │ [107.68, 191.61, 214.68] │       279.8 │
│        1200 │ 2020-06-24  │         0.0 │ [190.91, 191.61, 214.68] │       279.8 │
│        1200 │ 2020-06-25  │         0.0 │ [191.61, 203.06, 214.68] │       279.8 │
│        1200 │ 2020-06-26  │         0.0 │ [0.0, 203.06, 214.68]    │       279.8 │
├─────────────┴─────────────┴─────────────┴──────────────────────────┴─────────────┤
│1587 rows (10 shown)                                                 5 columns   │
└─────────────────────────────────────────────────────────────────────────────────┘
```

All aggregates can be used as windowing functions as we already learned. That includes complex statistical functions, such as computing the exact quantiles in a group (`quantile` and `quantile_disc`) or the interpolated ones (`quantile_cont`) as shown above. The implementations of these functions have been optimized for windowing, and we can use them without worrying about performance. Use a named window when you're querying for several aggregates.

4.5.4 Accessing preceding or following rows in a partition

We already discussed ranking and will see an example of computing running totals later in section Section 48, "Using the `ASOF` Join", but we haven't used the ability to jump back and forth between rows inside a partition. So let's have a look at computing changes and what could be a better example these days than prices?

In chapter 3 we created a table named `prices` that stores the prices in ct/kWH that are paid in Germany for feeding back energy to the grid. Those selling prices for renewable energy have been decreased as the promotion of renewables is getting cut back. You now want to know how much the compensation for renewable energy changed over time. For computing a difference you need the price value of row `n` and compare it with the value in row `n-1`. This is not possible without windows as the rows of a table are processed in isolation, essentially row by row. If you however span a window over any orderable column you can use `lag()` and `lead()` to access rows outside the current window. This allows you to pick the price from yesterday that you want to compare with today's price.

The `lag` function will give you the value of the expression in the row preceding the current one within the partition or `NULL` if there is none. This is the case for the first row in a partition. `lead` behaves the other way around (return `NULL` for the last row in a partition). Both functions have several overloads in DuckDB that allow not only to specify the offset of how many rows to lag or lead, but also a default window. Otherwise, working with `coalesce` would be an option when `NULL` values are not practicable.

TIP **The `coalesce` function will return its first non `NULL` argument.**

The query in 4.19 will compute the difference of the prices when new regulations have been introduced, using `lag()`:

Listing 4.19 Using a window function to compute values based on lagging and leading values

```
SELECT valid_from,
       value,
       lag(value) -- #1
         OVER validity AS "Previous value",
       value - lag(value, 1, value)
         OVER validity AS Change -- #2
FROM prices
WHERE date_part('year', valid_from) = 2019
WINDOW validity AS (ORDER BY valid_from)
ORDER BY valid_from;
```

#1 Jumps back a row and picks out the value column
#2 The change is computed as difference of the price in the current row and the price in the row before that, or the same value if there is no row before

As you see, in each new period the price decreased considerably in 2019:

```
┌────────────┬──────────────┬──────────────────┬──────────────┐
│ valid_from │    value     │ Previous value   │    Change    │
│    date    │ decimal(5,2) │  decimal(5,2)    │ decimal(6,2) │
├────────────┼──────────────┼──────────────────┼──────────────┤
│ 2019-01-01 │        11.47 │                  │         0.00 │
│ 2019-02-01 │        11.35 │            11.47 │        -0.12 │
│ 2019-03-01 │        11.23 │            11.35 │        -0.12 │
│ 2019-04-01 │        11.11 │            11.23 │        -0.12 │
│     ·      │           ·  │               ·  │           ·  │
│     ·      │           ·  │               ·  │           ·  │
│     ·      │           ·  │               ·  │           ·  │
│ 2019-09-01 │        10.33 │            10.48 │        -0.15 │
│ 2019-10-01 │        10.18 │            10.33 │        -0.15 │
│ 2019-11-01 │        10.08 │            10.18 │        -0.10 │
│ 2019-12-01 │         9.97 │            10.08 │        -0.11 │
├────────────┴──────────────┴──────────────────┴──────────────┤
│ 12 rows (8 shown)                             4 columns     │
└─────────────────────────────────────────────────────────────┘
```

If we are interested in computing the total change in prices in 2019, we must use a CTE, as we cannot nest window function calls inside aggregate functions. One possible solution looks like this:

Listing 4.20 Computing the aggregate over a window

```
WITH changes AS (
    SELECT value - lag(value, 1, value) OVER (ORDER BY valid_from) AS v
    FROM prices
    WHERE date_part('year', valid_from) = 2019
    ORDER BY valid_from
)
SELECT sum(changes.v) AS total_change
FROM changes;
```

The compensation for privately produced, renewable energy has been cut back by 1.50 ct/kWh in 2019 in Germany.

4.6 Conditions and filtering outside the WHERE clause

Filtering of computed aggregates or the result of a window function cannot be done via the standard `WHERE` clause. Such filtering is necessary to answer questions like

- Selection of groups that have an aggregated value that exceeds value x. For this you would have to use the `HAVING` clause.

- Selection of data that exceeds a certain value in a range of days. Here the `QUALIFY` clause must be used.

In addition, you might need to filter out values from entering an aggregate function at all, using the `FILTER` clause.

Table 4.1 Filtering clauses and where to use them

	Where to use it	Effect
`HAVING`	After `GROUP BY`	Filters rows based on aggregates computed for a group
`QUALIFY`	After the `FROM` clause referring to any window expression	Filters rows based on anything that is computed in that window
`FILTER`	After any aggregate function	Filters the values that are passed to the aggregate

4.6.1 Using the HAVING clause

"Please give me all the days on which more than 900kWh have been produced!" In chapter 3 you learned about both the `WHERE` clause and how `GROUP BY` works, and you just try to combine them like this:

```
SELECT system_id,
       date_trunc('day', read_on)        AS day,
       round(sum(power)  / 4 / 1000, 2)  AS kWh,
FROM readings
WHERE kWh >= 900
GROUP BY ALL;
```

In DuckDB 0.8.1 it gives you an error like this: "Error: Binder Error: Referenced column "kWh" not found in FROM clause!", other versions or databases might be clearer here in the wording. What it means is this: The computed column "kWh" is not yet known when the `WHERE` clause will be applied, and it can't be known at that point (in contrast to `day`, which is a computed column as well). Selecting rows in the `WHERE` clause, or filtering rows in other words, modifies what rows get aggregated in the first place. Therefore, you need another clause that gets applied after aggregation: The `HAVING` clause. It is used after the `GROUP BY` clause to provide a filter criteria after the aggregation of all selected rows has been completed.

Going back to the initial task: all you have todo is move the condition out of the `WHERE` clause into `HAVING` clause, that follows after the `GROUP BY`:

Listing 4.21 Using the `HAVING` clause to filter rows based on aggregated values

```
SELECT system_id,
       date_trunc('day', read_on)        AS day,
       round(sum(power)  / 4 / 1000, 2)  AS kWh,
FROM readings
GROUP BY ALL
HAVING kWh >= 900
ORDER BY kWh DESC;
```

The results are now filtered after they have been grouped together by the `sum` aggregate:

```
┌───────────┬────────────┬────────┐
│ system_id │    day     │  kWh   │
│   int32   │    date    │ double │
├───────────┼────────────┼────────┤
│        34 │ 2020-05-12 │ 960.03 │
│        34 │ 2020-06-08 │ 935.33 │
│        34 │ 2020-05-23 │ 924.08 │
│        34 │ 2019-06-09 │  915.4 │
│        34 │ 2020-06-06 │ 914.98 │
│        34 │ 2020-05-20 │ 912.65 │
│        34 │ 2019-05-01 │  912.6 │
│        34 │ 2020-06-16 │ 911.93 │
│        34 │ 2020-06-07 │ 911.73 │
│        34 │ 2020-05-18 │ 907.98 │
│        34 │ 2019-04-10 │ 907.63 │
│        34 │ 2019-06-22 │ 906.78 │
│        34 │ 2020-05-19 │  906.4 │
├───────────┼────────────┼────────┤
```

4.6.2 Using the QUALIFY clause

Let's say you want to only return rows where the result of a window function matches some filter. You can't add that filter in the `WHERE` clause, because that would filter out rows that get included in the window, and you need to use the results of the window function. However, you also can't use `HAVING`, because window functions get evaluated before an aggregation. So `QUALIFY` lets you filter on the results of a window function.

When we introduced window functions, we had to use a CTE to filter the results. We can rewrite the query much more simply and clearly by using `QUALIFY`, still getting the 3 highest ranked values.

Listing 4.22 Using the `QUALIFY` clause to filter rows based on aggregated values in a window

```
SELECT dense_rank() OVER (ORDER BY power DESC) AS rnk, *
FROM readings
QUALIFY rnk <= 3;
```

Going back to our example that uses a moving window over 7 days in 4.17. The 7-day average production value is a good indicator of the efficiency of a photovoltaic grid, and we might ask for the days at which a certain threshold was reached. We only want results for which the average in a 7-day window was higher than 875kWh, so that goes into the `QUALIFY` clause. The `QUALIFIY` clause can refer to the window function by name.

Listing 4.23 Finding out days with a power production over a certain threshold in a window of 7 days

```
SELECT system_id,
       day,
       avg(kWh) OVER (
            PARTITION BY system_id
            ORDER BY day ASC
            RANGE BETWEEN INTERVAL 3 Days PRECEDING
                      AND INTERVAL 3 Days FOLLOWING
       ) AS "kWh 7-day moving average"
FROM v_power_per_day
QUALIFY "kWh 7-day moving average" > 875  -- #1
ORDER BY system_id, day;
```

#1 Here's where we set the threshold

With the example data, we find three dates that represent a typical "good day" of power production in the western hemisphere for photovoltaics:

```
┌─────────────┬─────────────┬──────────────────────────┐
│ system_id   │    day      │ kWh 7-day moving average │
│   int32     │    date     │          double          │
├─────────────┼─────────────┼──────────────────────────┤
│          34 │ 2020-05-21  │        887.4628571428572 │
│          34 │ 2020-05-22  │        884.7342857142858 │
│          34 │ 2020-06-09  │        882.4628571428572 │
└─────────────┴─────────────┴──────────────────────────┘
```

4.6.3 Using the FILTER clause

Sometimes you want to compute an aggregate, an average, or a count of values, and you realize that there are some rows you don't want to include. You could add to the filter clause, but in a complex query, you might need to keep those rows to compute other fields. For example, let's say you sometimes got bad readings which show up as negative values. You want to compute the total number of readings and the average reading of the sensor. If you filtered out the bad readings in the `WHERE` clause, you wouldn't be able to compute the total number of readings. But if you just do an average over all the readings, then you'll be including some of the bad, negative values.

To solve this type of problem, you can use FILTER expressions as part of the aggregation.

Going back to section Section 41, in which we had to deal with inconsistent sensor readings, we are actually presented with the very problem of pulling `null` values into the average, which is most likely not what we want. Instead of capping `null` values to zero, we can filter them out altogether from the average value like this

Listing 4.24 Keeping nonsensical data out of the aggregates

```
INSERT INTO readings(system_id, read_on, power)
SELECT any_value(SiteId),
       time_bucket(
           INTERVAL '15 Minutes',
           CAST("Date-Time" AS timestamp)
       ) AS read_on,
       coalesce(avg(ac_power)
         FILTER (
           ac_power IS NOT NULL AND
           ac_power >= 0
         ),0 ) -- #1
FROM
    read_csv_auto(
        'https://developer.nrel.gov/api/pvdaq/v3/' ||
        'data_file?api_key=DEMO_KEY&system_id=10&year=2019'
    )
GROUP BY read_on
ORDER BY read_on;
```

#1 Values that are `NULL` or less than zero are not included in the average anymore

You might wonder why we the `coalesce` function: if all data is filtered out, nothing goes into the aggregate and the whole expression turns to `null`. That means if you filter out all the input from the aggregate, the value turns to `NULL` and that would violate the constraint on our reading table. As usual, there is no one right way here whether you prefer the solution in listing 4.1 or 4.24. In this case we slightly tend towards the `FILTER` based solution combined with `coalesce` because the intention here is slightly clearer.

4.7 The PIVOT statement

You can have many aggregates in one query and all of them can be filtered individually. This can help you to answer a task like this: "I want a report of the energy production per system and year, and the years should be columns!" Aggregating the production per system is easy, and so is aggregating the production per year. Grouping by both keys isn't hard either, a statement like this `SELECT system_id, year(day), sum(kWh) FROM v_power_per_day GROUP BY ALL ORDER BY system_id;` will do just fine and returns:

```
┌────────────┬──────────────┬────────────────────┐
│ system_id  │ year("day")  │      sum(kWh)      │
│   int32    │    int64     │       double       │
├────────────┼──────────────┼────────────────────┤
│         10 │         2019 │  1549.280000000001 │
│         10 │         2020 │  677.1900000000003 │
│         34 │         2019 │ 205742.59999999992 │
│         34 │         2020 │ 101033.75000000001 │
│       1200 │         2019 │ 62012.109999999986 │
│       1200 │         2020 │ 30709.329999999998 │
└────────────┴──────────────┴────────────────────┘
```

While we did group the data by system and year, the years per system appear in rows, not in columns. We want 3 rows with 2 columns, 2019 and 2020, containing the values, pretty much as you would find the above data in a spreadsheet program. The process of reorganizing such a table is called pivoting and DuckDB offers a couple of possibilities to do this and one of them is using multiple, filtered aggregates. Instead of having only one `sum` aggregate we define several and filter out each value that we don't want to have for a specific column and end up with the following statement:

Listing 4.25 Statically pivoting a result by applying a filter to all aggregates selected

```
SELECT system_id,
       sum(kWh) FILTER (WHERE year(day) = 2019)
            AS 'kWh in 2019',
       sum(kWh) FILTER (WHERE year(day) = 2020)
            AS 'kWh in 2020'
FROM v_power_per_day
GROUP BY system_id;
```

The values for the sum are equal, but the years are now columns, no more individual groups and your fictive boss can now view that data like they are used todo in their spreadsheet program:

```
┌───────────┬────────────────────┬────────────────────┐
│ system_id │    kWh in 2019     │    kWh in 2020     │
│   int32   │       double       │       double       │
├───────────┼────────────────────┼────────────────────┤
│        10 │  1549.280000000001 │  677.1900000000003 │
│        34 │ 205742.59999999992 │ 101033.75000000001 │
│      1200 │ 62012.109999999986 │ 30709.329999999998 │
└───────────┴────────────────────┴────────────────────┘
```

There's one downside to it: the columns are essentially hardcoded, and you need to revisit that query every time a year gets added. If you are sure that your desired set of columns is constant, or you find yourself targeting other databases that might not support any other form of pivoting, the static approach might be the right solution for you.

To solve this problem with DuckDB, use the `PIVOT` clause instead. The `PIVOT` clause in DuckDB allows dynamically pivoting tables on arbitrary expressions.

Listing 4.26 Using DuckDBs `PIVOT` statement

```
PIVOT (FROM v_power_per_day) -- #1
ON year(day) -- #2
USING sum(kWh); -- #3
```

#1 You can omit the `FROM` if you want to select all columns, but we included it to demonstrate that this can actually be a full `SELECT`.

#2 All distinct values from this expression are turned into columns

#3 The aggregate to be computed for the columns

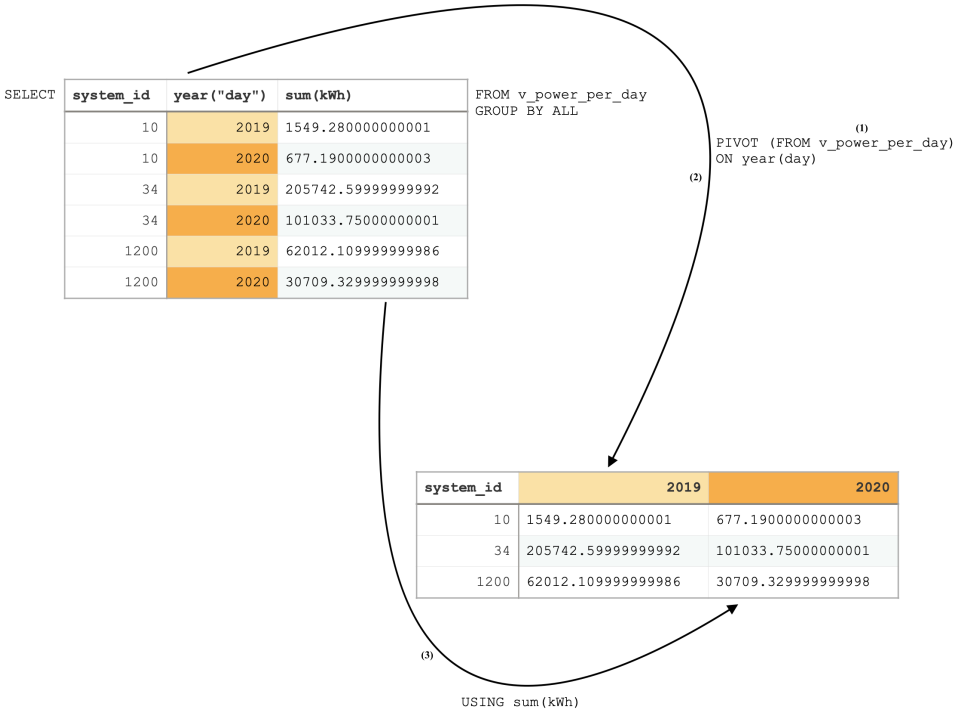


Figure 4.4 Pivoting the power values on the year

The result matches exactly what we statically constructed in 4.25: Years as columns and systems as rows with the sum of the power produced by a system and year as intersection of row and column.

```
┌─────────────┬─────────────────────┬─────────────────────┐
│ system_id   │        2019         │        2020         │
│   int32     │       double        │       double        │
├─────────────┼─────────────────────┼─────────────────────┤
│          10 │  1549.280000000001  │  677.1900000000003  │
│          34 │ 205742.59999999992  │ 101033.75000000001  │
│        1200 │ 62012.109999999986  │ 30709.329999999998  │
└─────────────┴─────────────────────┴─────────────────────┘
```

In case you are using an aggregate for the "cell" values, all columns that are not part of the `ON` clause will be used as a grouping key for the aggregate. However, you do not need to use an aggregate. `PIVOT v_power_per_day ON day` will produce a result of 1382 rows and 545(!) columns. Why is that? `v_power_per_day` contains 1382 distinct values of `(system_id, kWh)`, which make up the rows. The system has been asked to create a column using the `day`, not `year(day)` and there are 543 different days recorded. The two additional columns are the `system_id` and the `kWh` column. What's in the cells? Many, many zeros and a couple of ones. Without the `USING` clause DuckDB will fill the cells with zeros for days that didn't have the specific value, and ones for days that had. So in case you are actually interested in a tabular view of all days, you might want to use the `first` aggregate like this in such a case:

```
PIVOT (
    FROM v_power_per_day WHERE day BETWEEN '2020-05-30' AND '2020-06-02'
)
ON DAY USING first(kWh);
```

Note that we deliberately chose to select only a couple of days instead of trying to print several hundred columns. The above query pivots this result on the day:

```
┌───────────┬────────────┬────────┐
│ system_id │    day     │  kWh   │
│   int32   │    date    │ double │
├───────────┼────────────┼────────┤
│      1200 │ 2020-05-30 │  280.4 │
│      1200 │ 2020-05-31 │ 282.25 │
│      1200 │ 2020-06-01 │ 288.29 │
│      1200 │ 2020-06-02 │ 152.83 │
│         · │     ·      │      · │
│         · │     ·      │      · │
│         · │     ·      │      · │
│        10 │ 2020-05-30 │   4.24 │
│        10 │ 2020-05-31 │   3.78 │
│        10 │ 2020-06-01 │   4.47 │
│        10 │ 2020-06-02 │   5.09 │
├───────────┴────────────┴────────┤
│ 12 rows (8 shown)     3 columns │
└─────────────────────────────────┘
```

into a tabular view that would make any spreadsheet artist happy:

```
┌────────────┬────────────┬────────────┬────────────┬────────────┐
│ system_id  │ 2020-05-30 │ 2020-05-31 │ 2020-06-01 │ 2020-06-02 │
│   int32    │   double   │   double   │   double   │   double   │
├────────────┼────────────┼────────────┼────────────┼────────────┤
│         10 │       4.24 │       3.78 │       4.47 │       5.09 │
│         34 │      732.5 │     790.33 │     796.55 │     629.17 │
│       1200 │      280.4 │     282.25 │     288.29 │     152.83 │
└────────────┴────────────┴────────────┴────────────┴────────────┘
```

All queries above are using the proprietary DuckDB variant of `PIVOT`. DuckDB's syntax makes writing pivot statements much easier and less error-prone as it completely eliminates any static enumeration of the rows on which the table should be pivoted. DuckDB also supports a more standard SQL form of `PIVOT`. However, the support for the `PIVOT` clause wildly differs across different databases systems, and it is unlikely that other possible target databases have the exact same flavour of the standard. Therefore, we would rather use the proprietary syntax in this case, which is easier to read than hoping for more portable SQL.

In DuckDB it is perfectly possible to compute multiple aggregates in the `USING` clause as well as using multiple columns for pivoting. We could use this to not only compute the total production per year (which is the `sum` of all days) but also add two additional columns that highlight the best day:

```
PIVOT v_power_per_day
ON year(day)
USING round(sum(kWh)) AS total, max(kWh) AS best_day;
```

We rounded the totals so that the result is more readable:

```
┌────────────┬─────────────┬──────────────┬─────────────┬──────────────┐
│ system_id  │ 2019_total  │ 2019_best_day │ 2020_total  │ 2020_best_day │
│   int32    │   double    │    double     │   double    │    double     │
├────────────┼─────────────┼──────────────┼─────────────┼──────────────┤
│         10 │      1549.0 │          7.47 │       677.0 │          6.97 │
│         34 │    205743.0 │         915.4 │    101034.0 │        960.03 │
│       1200 │     62012.0 │        337.29 │     30709.0 │        343.43 │
└────────────┴─────────────┴──────────────┴─────────────┴──────────────┘
```

4.8 Using the ASOF JOIN ("as of")

Imagine you are selling a volatile product at arbitrary times of the day. You are able to predict prices at an interval, let's say 15 minutes, but that's as precise as you can go. Yet people demand your product all the time. This might lead to the following fictive situation. The query in listing 4.27 generates two CTEs: a fictive price table with 4 entries for an hour of a random day, as well as a sales table with 12 entries. It then joins them naively together and instead of the prices of 12 sales, you find only 4 results:

Listing 4.27 Using an inner join for timestamps

```
WITH prices AS (
  SELECT range AS valid_at,
         random()*10 AS price
  FROM range(
       '2023-01-01 01:00:00'::timestamp,
       '2023-01-01 02:00:00'::timestamp, INTERVAL '15 minutes')
),
sales AS (
  SELECT range AS sold_at,
         random()*10 AS num
  FROM range(
      '2023-01-01 01:00:00'::timestamp,
      '2023-01-01 02:00:00'::timestamp, INTERVAL '5 minutes')
)
SELECT sold_at, valid_at AS 'with_price_at', round(num * price,2) as price
FROM sales
JOIN prices ON prices.valid_at = sales.sold_at;
```

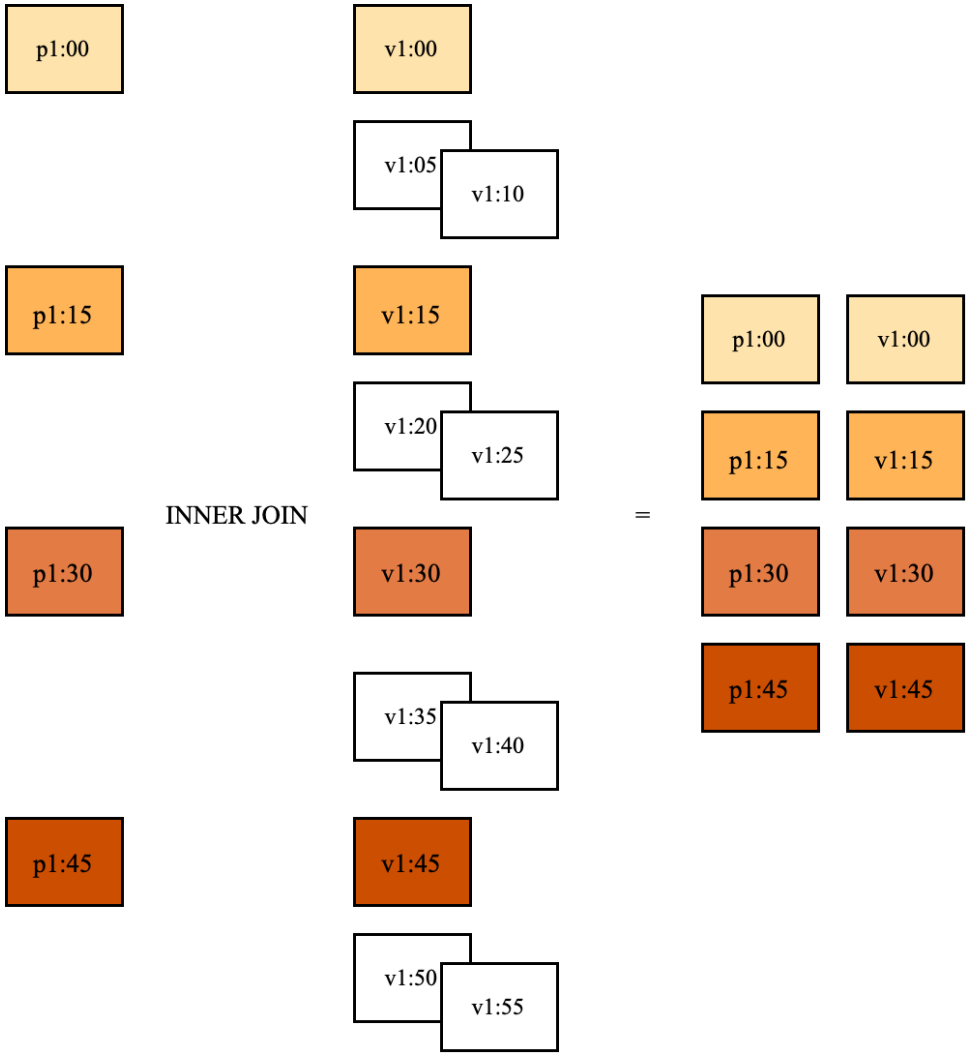


Figure 4.5 Inner join of time series data gone wrong

Sales are sad, as clearly indicated by this result and represented in figure 4.5:

```
┌─────────────────────┬─────────────────────┬─────────┐
│       sold_at       │    with_price_at    │ price   │
│      timestamp      │      timestamp      │ double  │
├─────────────────────┼─────────────────────┼─────────┤
│ 2023-01-01 01:00:00 │ 2023-01-01 01:00:00 │  21.17  │
│ 2023-01-01 01:15:00 │ 2023-01-01 01:15:00 │  12.97  │
│ 2023-01-01 01:30:00 │ 2023-01-01 01:30:00 │  44.61  │
│ 2023-01-01 01:45:00 │ 2023-01-01 01:45:00 │   9.45  │
└─────────────────────┴─────────────────────┴─────────┘
```

Enter the `ASOF JOIN`: The `ASOF JOIN` (as in "as of") is a join clause that joins on inequality, picking a "good enough" value for the gaps where the join columns are not exactly equal. Going back to listing 4.27 we must change two things: replacing the `JOIN` keyword with `ASOF JOIN`, and provide an inequality operator. The inequality condition `prices.valid_at <= sales.sold_at` below means all prices that have been valid before or at the point of sales can be used to compute the total price.

Listing 4.28 Using an "as of" join for timestamps

```
WITH prices AS (
  SELECT range AS valid_at,
         random()*10 AS price
  FROM range(
       '2023-01-01 01:00:00'::timestamp,
       '2023-01-01 02:00:00'::timestamp, INTERVAL '15 minutes')
),
sales AS (
  SELECT range AS sold_at,
         random()*10 AS num
  FROM range(
      '2023-01-01 01:00:00'::timestamp,
      '2023-01-01 02:00:00'::timestamp, INTERVAL '5 minutes')
)
SELECT sold_at, valid_at AS 'with_price_at', round(num * price,2) as price
FROM sales
ASOF JOIN prices -- #1
    ON prices.valid_at <= sales.sold_at; -- #2
```

#1 Specify the join to be ASOF

#2 Note the <= in contrast to the = in listing 4.27

Note how DuckDB picks the price that is closest to the point in time of the sales as shown below. Also, we do get the 12 expected rows now:

```
┌─────────────────────┬─────────────────────┬─────────┐
│       sold_at       │    with_price_at    │ price   │
│      timestamp      │      timestamp      │ double  │
├─────────────────────┼─────────────────────┼─────────┤
│ 2023-01-01 01:00:00 │ 2023-01-01 01:00:00 │   1.59  │
│ 2023-01-01 01:05:00 │ 2023-01-01 01:00:00 │   3.56  │
│ 2023-01-01 01:10:00 │ 2023-01-01 01:00:00 │   2.71  │
│ 2023-01-01 01:15:00 │ 2023-01-01 01:15:00 │  29.12  │
│ 2023-01-01 01:20:00 │ 2023-01-01 01:15:00 │  14.92  │
│ 2023-01-01 01:25:00 │ 2023-01-01 01:15:00 │   4.83  │
│ 2023-01-01 01:30:00 │ 2023-01-01 01:30:00 │   2.84  │
│ 2023-01-01 01:35:00 │ 2023-01-01 01:30:00 │   3.84  │
│ 2023-01-01 01:40:00 │ 2023-01-01 01:30:00 │   4.95  │
│ 2023-01-01 01:45:00 │ 2023-01-01 01:45:00 │   23.1  │
│ 2023-01-01 01:50:00 │ 2023-01-01 01:45:00 │  30.07  │
│ 2023-01-01 01:55:00 │ 2023-01-01 01:45:00 │   11.6  │
├─────────────────────┴─────────────────────┴─────────┤
│ 12 rows                                 3 columns   │
└─────────────────────────────────────────────────────┘
```

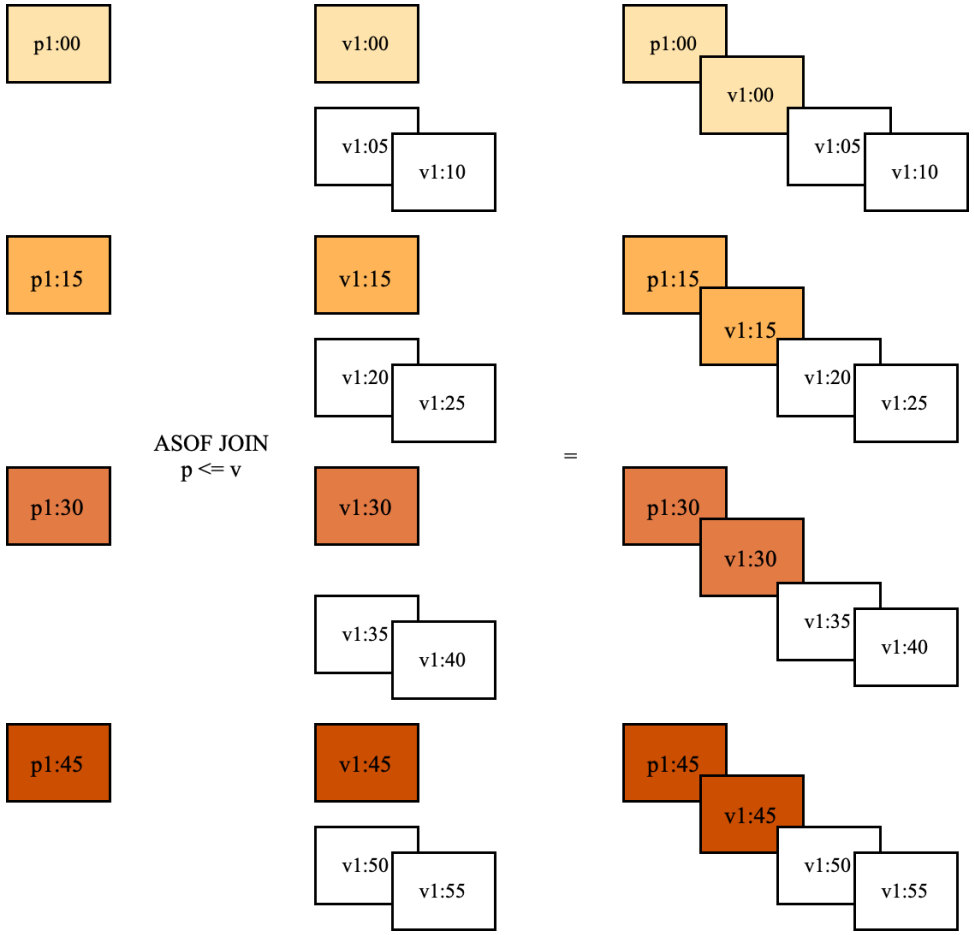


Figure 4.6 Using `ASOF JOIN` to join all timestamps together that don't have an exact match, too

The `ASOF JOIN` is often used to work with time series data such as stock quotes, prices, or IoT sensors. In our example, it can be used to join the changing selling prices with the readings from the systems to compute the prices at any given point in time. The last example below uses our photovoltaic example data again, applying the same logic to pick a valid price. It then demonstrates that the `ASOF JOIN` can be used with other constructs we learned in this chapter, such as using a window to accumulate the running total earnings in a sales period with different prices:

Listing 4.29 Using an `ASOF JOIN` together with a window function to compute earnings over different price periods

```
SELECT power.day,
       power.kWh,
       prices.value as 'ct/kWh',
       round(sum(prices.value * power.kWh)
           OVER (ORDER BY power.day ASC) / 100, 2)
           AS 'Accumulated earnings in EUR'
FROM v_power_per_day power
    ASOF JOIN prices
    ON prices.valid_from <= power.day
WHERE system_id = 34
ORDER BY day;
```

The result shows the day, the amount of kWH produced, the price in ct per kWH on that day, and the accumulated sum of the product of power produced and price:

```
┌────────────┬─────────┬──────────────┬──────────────────────────┐
│    day     │   kWh   │    ct/kWh    │ Accumulated earnings in EUR │
│    date    │ double  │ decimal(5,2) │          double          │
├────────────┼─────────┼──────────────┼──────────────────────────┤
│ 2019-01-01 │  471.4  │        11.47 │                    54.07 │
│ 2019-01-02 │ 458.58  │        11.47 │                   106.67 │
│ 2019-01-03 │ 443.65  │        11.47 │                   157.56 │
│ 2019-01-04 │ 445.03  │        11.47 │                    208.6 │
│     ·      │    ·    │           ·  │                      ·   │
│     ·      │    ·    │           ·  │                      ·   │
│     ·      │    ·    │           ·  │                      ·   │
│ 2020-06-23 │ 798.85  │         9.17 │                  31371.86 │
│ 2020-06-24 │ 741.15  │         9.17 │                  31439.83 │
│ 2020-06-25 │  762.6  │         9.17 │                  31509.76 │
│ 2020-06-26 │  11.98  │         9.17 │                  31510.86 │
├────────────┴─────────┴──────────────┴──────────────────────────┤
│ 543 rows (8 shown)                                 4 columns │
└──────────────────────────────────────────────────────────────┘
```

DuckDB is positioned as an OLAP database with a broad range of use cases. Dealing with time-series data is certainly one of them and the `ASOF` join is part of that. Regardless of the domain, which be anything from the sensor readings in our example to patient heart rate monitoring and fluctuations in the stock market, values recorded at a certain time are often enriched by joining them with specific key values that have been valid for a time. Having support for `ASOF` enables all scenarios in which timestamps are not aligned perfectly well.

4.9 Using Table functions

Most functions in SQL take parameters and return a single value. Table functions, on the other hand, don't just return a single value, they return a collection of rows. As such, they can appear everywhere where a table can appear. Depending on the function, they can access external resources such as files or URLs and turn them into relations that are part of standard SQL statements. DuckDB is not the only relational database supporting the concept of table-producing functions, but it comes with an impressive set of table functions catering to many use cases. A list of all table functions in your DuckDB installation can be retrieved via the following statement which uses a table function named `duckdb_functions()`

Listing 4.30 Getting a list of all available table functions

```
SELECT DISTINCT ON(function_name) function_name
FROM duckdb_functions() -- #1
WHERE function_type = 'table'
ORDER BY function_name;
```

#1 The `FROM` clause is the most common place to call a table-producing function.

In the examples in this chapter and during the ingestion of data we have made already extensive use of `read_csv*`, `read_parquet` and others. Additional extensions, such as the spatial extension, add to the list of table functions that read external resources and produce relational data.

`range(start, stop)` and `generate_series(start, stop)` are some really cool table functions. Both functions create a list of values in the range between `start` and `stop`. The `start` parameter is inclusive. For the `range` function, the `stop` parameter is exclusive, while it is inclusive for `generate_series`. Both functions provide overloads with an additional third parameter `step` defining the step-size which defaults to 1. Variants that only take the `stop` parameter and default `start` to 0 exists, too. While used as normal functions, they provide useful constructs, but are much more powerful when queried like a table.

If you need a list of numbers between 1 and 5 and don't want to hardcode them, you can use `SELECT generate_series(1, 5);` Numbers are helpful, but those functions also work with temporal data. When using temporal data be aware though that you need to specify both `start` and `end` parameter, as there is no sensible default for either. Let's put this to practical use. The readings in our example data end in the middle of 2020. Reports based on that would prematurely end if they are for a whole year as shown below:

```
SELECT strftime(day, '%Y-%m') AS month, avg(kwh)
FROM v_power_per_day WHERE year(day) = 2020
GROUP BY ALL ORDER BY month;
```

The result will look like the output below:

```
┌─────────┬────────────────────┐
│  month  │      avg(kwh)      │
│ varchar │       double       │
├─────────┼────────────────────┤
│ 2020-01 │ 222.13169014084497 │
│ 2020-02 │ 133.52356321839076 │
│ 2020-03 │ 207.86670454545438 │
│ 2020-04 │  309.7838888888888 │
│ 2020-05 │  349.5753763440861 │
│ 2020-06 │ 337.80820512820515 │
└─────────┴────────────────────┘
```

If you are tasked to create a chart you might find yourself in a situation in which you need to think about how to represent the future months. Here's one way to use the `range()` function to cover a whole year and indicate missing values as 0:

Listing 4.31 Using a range of dates as driving table

```
WITH full_year AS (
    SELECT generate_series AS day
    FROM generate_series( -- #1
        '2020-01-01'::date,
        '2020-12-31'::date, INTERVAL '1 day')
)
SELECT strftime(full_year.day, '%Y-%m') AS month,
       avg(kWh) FILTER (kWh IS NOT NULL) AS actual
FROM full_year -- #2
LEFT OUTER JOIN v_power_per_day per_day -- #3
  ON per_day.day = full_year.day
GROUP BY ALL ORDER BY month;
```

#1 A range defined from the first up to the last day of the year, in an interval of 1 day
#2 Use the output of the table function in the `FROM` clause as driving table
#3 Outer join the values of interest

The result is now a report for a full year, which sadly lacks values after June 2020:

```
┌─────────┬────────────────────┐
│  month  │       actual       │
│ varchar │       double       │
├─────────┼────────────────────┤
│ 2020-01 │ 222.13169014084508 │
│ 2020-02 │ 133.52356321839076 │
│ 2020-03 │ 207.86670454545455 │
│ 2020-04 │  309.7838888888888 │
│ 2020-05 │ 349.57537634408607 │
│ 2020-06 │ 337.80820512820515 │
│ 2020-07 │                    │
│ 2020-08 │                    │
│ 2020-09 │                    │
│ 2020-10 │                    │
│ 2020-11 │                    │
│ 2020-12 │                    │
├─────────┴────────────────────┤
│ 12 rows            2 columns │
└──────────────────────────────┘
```

Taking this idea one step further would be using the value of the same month in the previous year to forecast the production value. For that you would have to join `v_power_per_day` a second time using an offset of one year:

Listing 4.32 Projecting past data into the future

```
WITH full_year AS (
    SELECT generate_series AS day
    FROM generate_series(
        '2020-01-01'::date,
        '2020-12-31'::date, INTERVAL '1 day')
)
SELECT strftime(full_year.day, '%Y-%m') AS month,
       avg(present.kWh) FILTER (present.kWh IS NOT NULL) AS actual,
       avg(past.kWh) FILTER (past.kWh IS NOT NULL) AS forecast,
FROM full_year
LEFT OUTER JOIN v_power_per_day present
  ON present.day = full_year.day
LEFT OUTER JOIN v_power_per_day past -- #1
  ON past.day = full_year.day - INTERVAL '1 year'
GROUP BY ALL ORDER BY month;
```

#1 Joining power per day a second time but subtracting a year from the values of the generated series

The result is much more pleasant and happens to give a comparison between this year and last year essentially for free:

```
┌─────────┬────────────────────┬────────────────────┐
│  month  │       actual       │      forecast      │
│ varchar │       double       │       double       │
├─────────┼────────────────────┼────────────────────┤
│ 2020-01 │ 222.13169014084505 │ 161.59319248826304 │
│ 2020-02 │  133.5235632183909 │ 111.07298850574716 │
│ 2020-03 │  207.8667045454546 │ 150.65231060606064 │
│ 2020-04 │ 309.78388888888895 │  316.1782222222224 │
│ 2020-05 │ 349.57537634408595 │ 325.36881720430125 │
│ 2020-06 │  337.8082051282051 │ 351.60691056910514 │
│ 2020-07 │                    │ 334.32311827956994 │
│ 2020-08 │                    │   314.928817204301 │
│ 2020-09 │                    │  289.6049999999999 │
│ 2020-10 │                    │ 253.82935483870958 │
│ 2020-11 │                    │  191.3843333333334 │
│ 2020-12 │                    │ 164.88628205128202 │
├─────────┴────────────────────┴────────────────────┤
│ 12 rows                                3 columns │
└──────────────────────────────────────────────────┘
```

4.10 Using LATERAL joins

In section Section 43 we learned about correlated and uncorrelated sub-queries. Listing 4.5 demonstrated how an uncorrelated sub-query can be joined once with the outer query. From a performance point of view that might be beneficial, as the sub-query needs to be only evaluated once and the join is then performed for each row of the other table against the memorized values.

Sometimes, however, you precisely want to evaluate the inner query for each value of an outer query. This is where the `LATERAL JOIN` comes into play. You can think of it as the inner block of a for-loop and the outer query being the control structure.

Unnesting arrays, fanning out data and similar tasks can be dealt by using `LATERAL`. Assume you are interested in the intensity of the sun, how much of its energy reaches your place at certain hours of the day, past or future. Open Meteo offers a free API that provides a broad range of weather data, including the so-called Global Horizontal Irradiance (GHI). That is the total amount of short-wave radiation received from above by a surface horizontal to the ground. This value is of particular interest to photovoltaic installations and is measured in W/m². Their API generates a JSON object that contains two individual arrays, one with the timestamps, one with the selected values. The latter array is the array of interest, we want to retrieve specific values for some given facts.

Listing 4.33 `ch04/ghi_past_and_future.json`**, a JSON structure with past and forecast data for GHI, from Open Meteo**


```
{
    "latitude": 50.78,
    "longitude": 6.0799994,
    "utc_offset_seconds": 7200,
    "timezone": "Europe/Berlin",
    "timezone_abbreviation": "CEST",
    "elevation": 178.0,
    "hourly_units": {
       "time": "iso8601",
       "shortwave_radiation_instant": "W/m\u00b2"
    },
    "hourly": {
        "time": [
            "2023-08-26T00:00",
            "2023-08-26T01:00",
            "2023-08-26T02:00",
            "2023-08-26T03:00",
            "2023-08-26T04:00",
            "2023-08-26T05:00"
        ],
        "shortwave_radiation_instant": [
            0.0,
            0.0,
            0.0,
            0.0,
            0.0,
            9.1
       ]
    }
}
```


The above JSON is in the code repository of the book under `ch04/ghi_past_and_future.json`. Alternatively you can get fresh data using this URL: https://api.open-meteo.com/v1/forecast?latitude=52.52&longitude=13.41&hourly=shortwave_radiation_instant&past_days=7

At first sight it might be a daunting task, using SQL to pick out the morning hours, noon and the evening hours from that array. Let's see how `LATERAL` can solve this task. We already read in chapter 1 that DuckDB is able to process JSON and will go into more details in chapter 5. For now, it shall be good enough to know you can select from a JSON file like from any other table in the `FROM` clause. The below query generates a series of 7 days, joins those with the hours 8, 13 and 19 (7pm) to create indexes. Those indexes are the day number * 24 plus the desired hour of the day to find the value in the JSON array. That index is the lateral driver for the sub-query:

```
INSTALL json;
LOAD json;

WITH days AS (
  SELECT generate_series AS value FROM generate_series(7)
), hours AS (
  SELECT unnest([8, 13, 18]) AS value
), indexes AS (
  SELECT days.value * 24 + hours.value AS i
  FROM days, hours
)
SELECT date_trunc('day', now()) - INTERVAL '7 days' +
         INTERVAL (indexes.i || ' hours') AS ts, -- #1
       ghi.v AS 'GHI in W/m^2'
FROM indexes,
LATERAL (
  SELECT hourly.shortwave_radiation_instant[i+1] -- #2
          AS v
  FROM 'code/ch04/ghi_past_and_future.json' -- #3
) AS ghi
ORDER BY ts;
```

#1 Recreates a data again from the hourly index
#2 Arrays are 1 based in DuckDB (and SQL in general)
#3 DuckDB automatically detects that this string refers to a JSON files, loads and parses it

The end of August in Aachen did look like this in 2023, it was not a great month for photovoltaics:

```
┌──────────────────────────┬──────────────┐
│           ts             │ GHI in W/m^2 │
│ timestamp with time zone │    double    │
├──────────────────────────┼──────────────┤
│ 2023-08-26 08:00:00+02   │         36.0 │
│ 2023-08-26 13:00:00+02   │        490.7 │
│ 2023-08-26 18:00:00+02   │          2.3 │
│ 2023-08-27 08:00:00+02   │        243.4 │
│ 2023-08-27 13:00:00+02   │        124.3 │
│           .              │          .   │
│           .              │          .   │
│           .              │          .   │
│ 2023-09-01 13:00:00+02   │        392.0 │
│ 2023-09-01 18:00:00+02   │          0.0 │
│ 2023-09-02 08:00:00+02   │        451.0 │
│ 2023-09-02 13:00:00+02   │        265.0 │
│ 2023-09-02 18:00:00+02   │          0.0 │
├──────────────────────────┴──────────────┤
│ 24 rows (10 shown)            2 columns │
└─────────────────────────────────────────┘
```

The sub-query can produce zero, one or more rows for each row of the driving, outer table. In the example above, it produced one row for each outer row. In case the sub-query produces more rows, the values of the outer row will be repeated, similar to what a `CROSS` join does. In case the sub-query does not produce any value, the join won't produce a value either. We must apply an `OUTER JOIN` in this case as well. At this point the `LATERAL` keyword alone is not enough, and we must use the full `JOIN` syntax like this. The following query is artificial and of little value except demonstrating the syntax. Both queries produce a series of values from 1 to 4, the outer in a step size of one, the inner in a step size of two. We compare both values in the `ON` clause.

```
SELECT i, j
FROM generate_series(1, 4) t(i)
  LEFT OUTER JOIN LATERAL (
    SELECT * FROM generate_series(1, 4, 2) t(j)
  ) sq ON sq.j = i -- #1
ORDER BY i;
```

#1 While the condition is now on the outside and cannot be formulated otherwise, it is still a correlated sub-query

The result of this query looks like this:

```
┌───────┬───────┐
│   i   │   j   │
│ int64 │ int64 │
├───────┼───────┤
│     1 │     1 │
│     2 │       │
│     3 │     3 │
│     4 │       │
└───────┴───────┘
```

The problem with the prices in section Section 48 can be solved with a sub-query and `LATERAL JOIN`, too. In essence, the subquery must return a row from the price table that has a validity that is as close to the date in time of the sale as possible. For that to work we cannot use a normal join, as the sub-query must produce different values for each incoming date. Therefore, the date-column that would normally be part of the join must move inside the sub-query. Thus, the joined sub-query now becomes correlated, or laterally joined to the outer query. The correlation in the example below is the validity of the price compared to the day the power production was recorded.

Listing 4.34 Comparing the `ASOF` join from 4.29 to a `LATERAL JOIN`

```
SELECT power.day, power.kWh,
       prices.value as 'EUR/kWh'
FROM v_power_per_day power,
    LATERAL ( -- #1
        SELECT *
        FROM prices
        WHERE prices.valid_from <= power.day -- #2
        ORDER BY valid_from DESC limit 1 -- #3
    ) AS prices
WHERE system_id = 34
ORDER BY day;
```

#1 Mark the sub-query as lateral thus allowing correlation

#2 Correlate by inequality

#3 While the `ASOF JOIN` would automatically pick the closest value for us, we must order the values ourselves when using `LATERAL`

For time-series related computations with DuckDB we would most certainly use the `ASOF JOIN`. `LATERAL` is attractive when you need to think about portability, and you probably find more databases supporting `LATERAL` then `ASOF JOIN`. Use `LATERAL` in scenarios in which you want to fan-out of a dataset to produce more rows.

4.11 Summary

- The SQL standard has evolved a lot since the last major revision in 1992 (SQL-92) and DuckDB supports a broad range of modern SQL, including CTEs (SQL:1999), window functions (SQL:2003), list aggregations (SQL:2016) and more.
- Grouping sets allow the computation of aggregates over multiple groups, doing a drill-down into different levels of detail; `ROLLUP` and `CUBE` can be used to generate subgroups or combinations of grouping keys
- DuckDB fully supports window functions, including named windows and ranges, enabling use cases such as computing running totals, ranks and more.
- All aggregate functions, including statistic computations and interpolations, are optimized for usage in a windowed-context.
- `HAVING` and `QUALIFY` can be used to select aggregates and windows after they have been computed, `FILTER` prevents unwanted data from going into aggregates.
- DuckDB includes `ASOF` join which is needed in use cases involving time-series data.
- DuckDB also supports `LATERAL` joins that help fanning out data and can emulate loops to a certain extent.
- Results can be pivoted, either with a simplified, DuckDB specific `PIVOT` statement or with a more static, standard SQL approach.