NOTE: This book has nonstandard 30x50 cm pages

MonetDB on Ubuntu

From installation to full power

by Bizkapish

MonetDB on Ubuntu

From Installation to Full Power

Bizkapish
Belgrade, Serbia
2026.

MonetDB on Ubuntu
From Installation to Full Power

Publisher: Bizkapish
Belgrade, Serbia

ISBN 978-86-908978-0-3

First edition 2026

Source code and additional materials:
https://github.com/Bizkapish/monetdb-on-ubuntu

YouTube:
youtube playlist

Odysee:
odysee playlist

Dear reader,

Each section of this book is accompanied by a YouTube video that demonstrates the practical steps described in the text. I recommend watching the video first and then reading the corresponding section of the book, for better understanding.

YouTube playlist:
youtube playlist

In addition to YouTube, all videos are also available on Odysee for those who prefer an alternative platform. Odysee provides the same content, with the added benefit that videos can be downloaded for offline viewing. This can be especially useful if you want to study without an internet connection or keep a local copy of the material for future reference.
odysee playlist

Some sections include downloadable materials such as scripts, configuration files, and example datasets. These resources are available in the GitHub repository for this book.

I may also publish additional chapters and updates there as the book evolves.

If you find any errors, unclear explanations, or technical issues, please report them using the Issues section of the repository.

GitHub repository:
https://github.com/Bizkapish/monetdb-on-ubuntu

Table of Contents

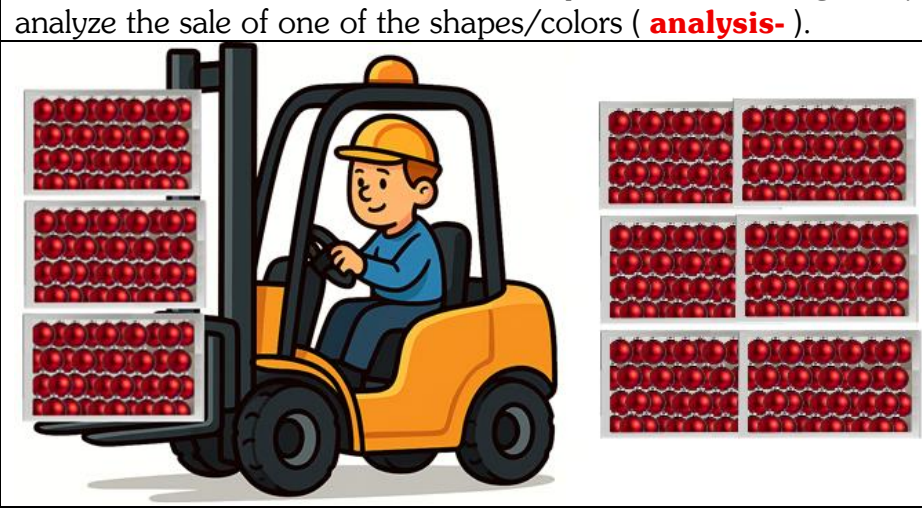
0000 MonetDB Introduction

Bundled Products vs Products in Bulk

Stores like to sell products as a bundle. For example, the store will sell a box with several decorations as one product. Store will benefit from this because:

- Stacking and scanning products is easier when the are bundled (**effort-**).
- The buyer will spent less time in the store choosing decorations (**speed+**).
- We can sell more decorations by bundling less popular decorations with the more popular ones (**sale+**).

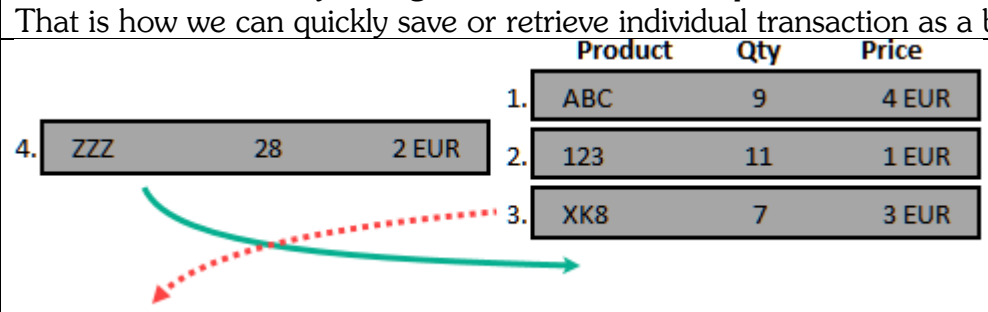
But bundles make production planning harder.
- We have to unbundle data about products to find out how much each individual shape/color was sold (**effort+**).
- We are usually looking at sale for a longer period of time. Because of quantity of data, unbundling will make our database real slow (**speed-**).
- We have to read data about all the shapes/colors, although we just want to analyze the sale of one of the shapes/colors (**analysis-**).



Data in Bundle or in Bulk

We can treat transactions as one bundle product. Product code, quantity and price are one bundle. We want to be able to quickly save this data bundle.

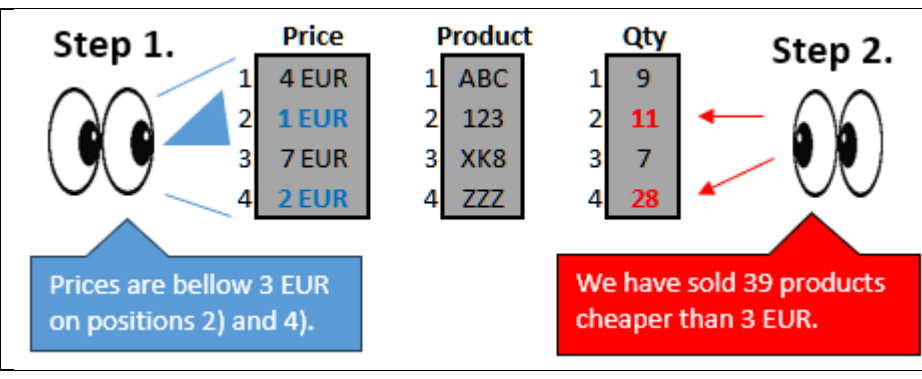
We can achieve this by saving our transactions as separate rows in a database. That is how we can quickly save or retrieve individual transaction as a bundle.



For production planning we will differently organize our data. We will organize it into columns. If we want to find out how many products, we have sold that have the price lower than 3 EUR, we just have to scan the column "Price", and then to find specific positions in the column "Qty".

Each column can be sorted and compressed for even faster retrieval.

We don't even need to read the column "Product".



When we organize data in rows, we use a row-oriented database. When we organize data in columns, we use a columnar database. Data is typically collected in a row-oriented database (Postgres, Oracle, MySQL) and then transferred to a columnar database for analytics (ClickHouse, DuckDB, Vertica).

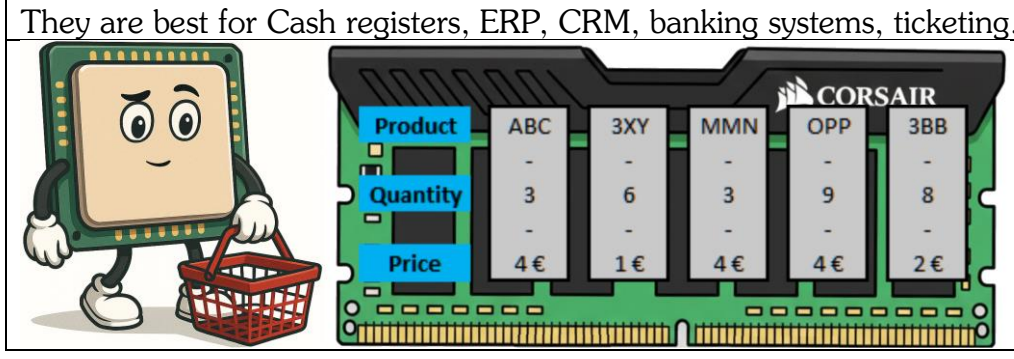
Row Oriented Databases vs Columnar Databases

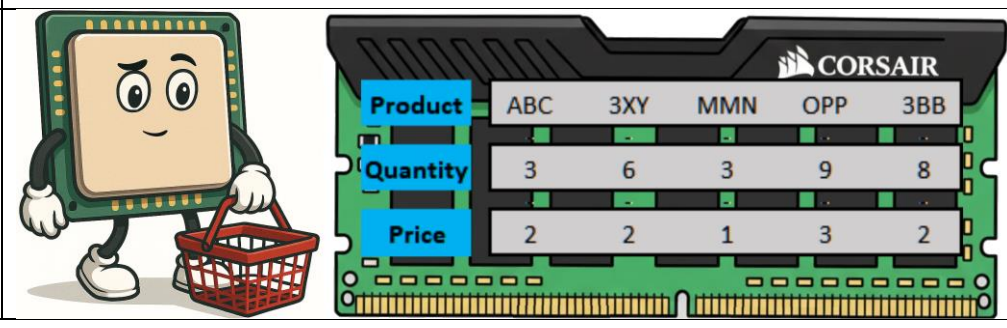
Row oriented databases:
1) They are great for small writes/reads. Perfect for "insert one order", "update this user", "delete this invoice line" etc.
2) Application code often works with whole objects/rows (User, Order, Invoice), which matches row layout.
3) Great for transaction isolation, replication, ACID complacence, constraints, triggers, foreign keys.
4) We have many users who send queries that touch only one row from a table and expect an immediate response from the database.

They are best for Cash registers, ERP, CRM, banking systems, ticketing.

Columnar databases:
1) We only read columns we need, and columns can be heavily compressed. That improves IO.
2) CPU can process several values of the same type at once. This is called vectorized execution. Queries with lot of data will work faster.
3) Column scans and aggregations are easy to parallelize across cores and nodes.
4) We have smaller number of queries, but they are much heavier, they touch a lot of data, with lot of joins, but the users are expecting results in 3-4 seconds.

They are best for sales reports, dashboards, time-series, data mining.

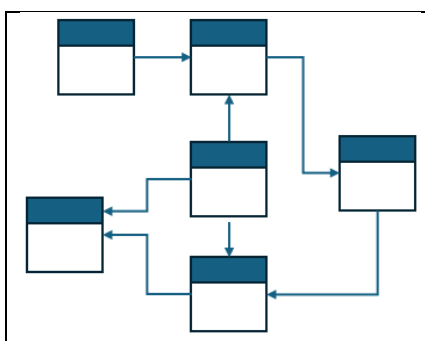




Row oriented databases are catching current state and providing transactional correctness. Columnar databases are for analysis of the historical data. We can also say that row-oriented databases are better in writing data and providing data consistency and integrity. Columnar databases are better in reading huge quantities of data.

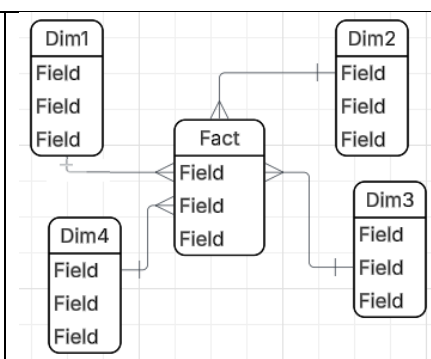
OLTP vs OLAP

If we are preparing our database for everyday operations and transactions, then we are making OLTP database. For analytics and BI, we would prepare OLAP database. OLTP is compatible with row-oriented databases, and OLAP with columnar databases.



In the OLTP databases, data is organized in a lot of small tables. There is no data duplication and every value is written only once.

Small tables make queries more complex so these databases usually don't grow significantly in size.



In the OLAP databases, we merge small tables to reduce their number. Columnar technology have no problem with tall tables that have a lot of columns.

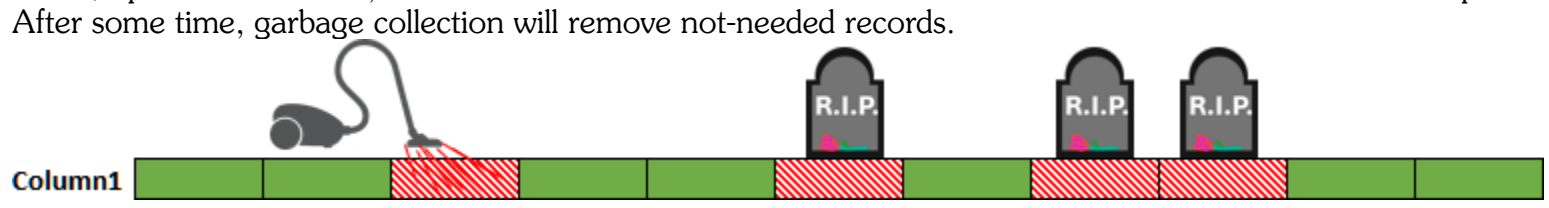
By reducing the number of joins, we can query our data with simpler (faster) queries. That is why we organize our tables in a star schema.

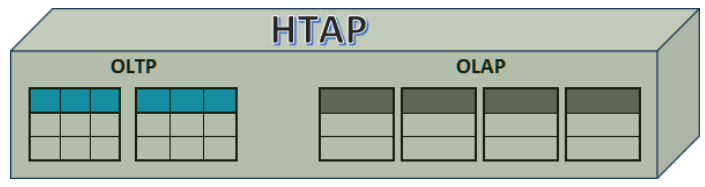
OLTP tables are filled with data by users and applications in small transactions. From time to time (usually nightly), data is transformed and transported into OLAP databases in batches. This makes data management different between OLTP and OLAP databases. OLTP databases receive data continuously, but OLAP databases receive data periodically and only after data is well-reformed.

OLTP is acronym for "Online Transaction Processing" and OLAP for "Online Analytical Processing". We saw that the difference is made by:
- Different purpose and usage.
- Different database technology.
- Different data organization inside the database.
- Different data management.

Data Immutability

Columnar (OLAP) databases are filled in batches. That doesn't mean that it is not possible to UPDATE or DELETE rows in tables. There are four strategies that OLAP databases use:

1) Data is immutable. We can only insert data in batches, and afterward we can not change individual rows. If data is immutable and presorted then we can achieve the biggest levels of compression. The example of this technology is the database used for Power BI.	2) MonetDB is an example of the OLAP database that is suited for light updates and deletes. When we delete/update some value, MonetDB will label old record as "dead". That record will become invisible to queries. After some time, garbage collection will remove not-needed records. 

3) We can divide our columns into small segments. Whenever we want to change some value, we modify the whole segment. This is good solution when we have a lot of hardware power, like cloud providers do. Google Snowflake is an example of such database. 	4) We can have two databases in one. One database is row-oriented, and the other one is columnar. Data is written into row-oriented database, but when we want to read data we read from both databases together. Historical data is read from the columnar database, and today's data is read from the row-oriented database. SAP Hana is using such approach.	This last strategy creates something called "Hybrid Transaction Analytical Processing", or HTAP. HTAP is when a single database can do fast real-time transactions and fast analytics on the same data at the same time. 

In recent years, MonetDB researchers have been gradually adding or experimenting with features that move it slightly toward the HTAP category:	• Better handling of frequent updates and small transactions • Improved concurrency and snapshot isolation • Faster incremental data loads • Optimizations for mixed workloads

MonetDB

If you have one server and install MonetDB on it, you will be able to:
- Enjoy blazing-fast query performance, even if you have a lot of data and your queries are more complex.
- Avoid thinking about indexing, compression, statistics tuning, partitioning decisions. MonetDB will do it all automatically.
- Apply the full power of SQL to all your tables, whether they are carefully modeled or just collected in one place.
- Spend your money on something else because MonetDB is an open source database under the Mozilla Public License.

MonetDB is a fast, complete SQL, easy-to-maintain, open source analytical database server that can process huge amounts of data on a single server machine.

Hardware and Speed Considerations for MonetDB

MonetDB doesn't have a requirement to keep the whole database in a RAM memory. Parts of data that are already processed will be deleted from memory. MonetDB have capacity to deal with data that is much bigger than the amount of memory on your server machine. Only data that is currently processed will have just-in-memory execution pipelines.

MonetDB, will benefit from the fast NVMe disk, that has sustained performance under write load. Such disk should be accompanied with a lot of RAM memory, so that usage of a disk is minimal.

MonetDB is designed to take the full advantage of your CPU. - When executing a query or loading data, MonetDB will spread the load across all processor cores, increasing CPU utilization to over 90%. There is a setting that can limit the number of cores used by MonetDB. - Vectorized query execution means that the processor can process thousands of values with a single instruction. The values must be of the same data type, which is perfect for columnar databases. - Data is processed in small chunks using simple instructions. This reduces RAM visits and uses the processor's fast L1/L2 memory. - Late materialization means that we avoid reading data that does not contribute to our query. We will only read the data needed for the final dataset, and skip everything else. - Partial query results are reused instead of being recomputed.	

Zero-Tuning Philosophy

MonetDB will not burden you with indexing strategies, partitioning decisions and vacuuming. MonetDB tries to eliminate complexity. Most workloads run at full speed without manual optimization. Indexes are automatic. Compression is automatic. Memory management is automatic.

Creating indexes manually is problematic because you need to know in advance where to create the indexes. Indexes can slow down updates and deletes because when data is modified, the database must also update the corresponding indexes to reflect these changes. This is something that we especially want to avoid in the columnar databases. If users change their behavior, that can make our indexes ineffective.

Technique that MonetDB use to automatically tune indexes is called database cracking. Cracking means that MonetDB will sort and group data, and create indexes during SELECT queries. If most of the queries on the OLAP databases are SELECT queries, and such queries touch a lot of rows, then it is best to create indexes and tune data during the SELECT queries execution. This is optimal because:
- The parts of a database that are heavily used will be tuned the best. We will not spend effort on data that no one reads.
- Instead of heaving one huge indexing job, we will have many tiny, incremental reorganizations.
- After some time, our data will be optimally indexed and sorted, but MonetDB can change strategy if users change their behavior.
- Cracking is especially powerful in columnar databases like MonetDB, where each column is stored separately.

Cracking is powerful but not ideal for everything:
- Heavy transactional workloads (many updates) disrupt adaptive patterns.
- Distributed cracking across many nodes is still complex.
- Highly unpredictable workloads (all queries unique) limit its benefits.

Comparison of MonetDB and Power BI

Power BI database (SSAS Tabular) is in-memory database, that is using compressed columnar tables, and is optimized for BI models (dimensions, measures, heirarchies). Power BI needs the whole model loaded in the memory, but columns are heavily compressed, so Power BI doesn't have huge memory footprint. For many calculations Power BI doesn't even need to decompress columns, so it can do its work while maintaining really fast IO.

Power BI likes table relationships and measures defined in advance, and is not optimized for arbitrary queries. It is not intended for complex joins outside the model. If we spend time creating optimal model, and we wait for our model to process/refresh during the load, we will get database that is optimized for:
- measure evaluation
- filters
- slice and dice

On the other side MonetDB is optimized for raw analytical SQL on large tables, without modeling overhead.

MonetDB Imperfections

As columnar server, MonetDB suffers from all of the shortcomings that columnar servers have. MonetDB is not good in transactional traffic, and lot of deletes and updates. MonetDB is not perfect for reports that return huge tables with lot of columns. MonetDB is best for short, aggregated analytical queries.

If you have massive amounts of data, you will need a computer cluster, and cloud scalability. This is something where MonetDB doesn't shine. MonetDB is best if you have one powerful server machine, and you want to run complex analytics on it. Distributed databases are better in scaling, but because their setup is more complex and they are spread over many computers, they usually have less rich SQL capabilities and other limitations.

MonetDB does not support replication, but it does support distributed operation across multiple machines via sharding. Sharding will speed up some queries, but not all. If we want to set up a cluster with MonetDB, partitioning and sharding must be done carefully and manually. If possible, it is better to have a single powerful server than to resort to sharding and expensive network equipment.

MonetDB Market Position

You should use MonetDB if you want to: - Self-hosting an analytical database on your server. - You run complex ad-hoc queries, window functions, joins, and aggregations. - You don't want to design cubes or DAX, you just want fast SQL. - Avoid vendor lock by using fully open source software.	MonetDB is best for: - Data engineers - BI developers - Researchers - Small/medium businesses - Python users

MonetDB supports SQL, ODBC, JDBC and many programming languages (python, java, R, ruby, PHP…). It can be easily integrated with different software, but the support of 3rd party software, clients and ORMs is limited. This is the consequence of the MonetDB origin which was research and science oriented. MonetDB was developed on the CWI institute in the Netherland. Development was driven by innovation and curiosity, and not by commercialization and marketing. This is why you can say that MonetDB is the fastest database you've never heard of. But that development lasted for 30 years and today MonetDB is complete and powerful database system.

0005 MonetDB Benchmark

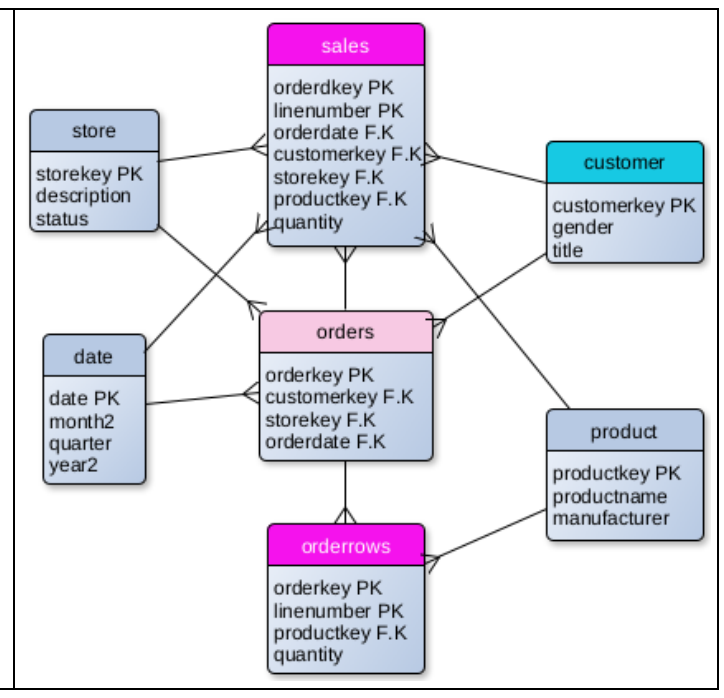
Benchmark Data

For this benchmark we will use sales data from the fictitious company "Contoso". We can download our database from the github. If we go to this address, we will find there compressed CSV files with the data. In total there are 8 files of 500 MB, and one smaller file with 250 MB (4250 MB in total).

https://github.com/sql-bi/Contoso-Data-Generator-V2-Data/releases/tag/ready-to-use-data

- csv-100m.7z.001
- csv-100m.7z.002
- csv-100m.7z.003
- csv-100m.7z.004
- csv-100m.7z.005
- csv-100m.7z.006
- csv-100m.7z.007
- csv-100m.7z.008
- csv-100m.7z.009

sales.csv 18,4 GB orderrows.csv 8,5 GB orders.csv 4,5 GB customer.csv 477,6 MB currencyexchange.csv 2,5 MB date.csv 377,4 kB product.csv 373,7 kB store.csv 6,4 kB	When we unzip these files, inside we will find 8 CSV files. This is sales cube with tables that can show us sales and orders per product, customer, date and store. Three big tables are "Orders" (88M), "**Sales**" (211M) and "**OrderRows**" (211M). Dimension table "**Customer**" has 2M rows, and all the other dimension tables are small. Not all of the columns are shown on the image. --- In CSV file "`date.csv`" I will change the names of columns `month=>month2` and `year=>year2,` because MonetDB will not accept original names. "`Month`" and "`Year`" are reserved words.



Machine Hardware

Model ASUS PRIME B550M-A Memory 64,0 GiB Processor AMD Ryzen™ 7 5700G with Radeon™ Graphics × 16 Graphics AMD Radeon™ Graphics Disk Capacity 4,0 TB Firmware Version 3002 OS Name Zorin OS 18 Core OS Type 64-bit Windowing System Wayland Kernel Version Linux 6.14.0-36-generic	For this benchmark we will use CPU with 8 cores and 64 GB of RAM. Our operational system is Zorin 18.

Creating Tables and Loading CSV Files

OrderRows Table

```
CREATE TABLE orderrows (
OrderKey BIGINT NOT NULL,
LineNumber SMALLINT NOT NULL,
ProductKey SMALLINT NOT NULL,
Quantity SMALLINT NOT NULL,
UnitPrice REAL NOT NULL,  --DECIMAL(8,4)
NetPrice REAL NOT NULL,   --DECIMAL(10,6)
UnitCost REAL NOT NULL    --DECIMAL(8,4)
);
```

First, I will create table for OrderRows in the MonetDB. I will not create primary and foreign key constraints this time.

I will fill this table from my CSV file with "COPY INTO" statement.

```
COPY OFFSET 2 INTO orderrows
FROM '/home/fff/Desktop/CSVs/orderrows.csv'
USING DELIMITERS ',', E'\n', '"';
```

Import of this 211M rows table will last only one minute (**56.241 sec**). Amazing.

OrderRows.csv

```
1 OrderKey,LineNumber,ProductKey,Quantity,UnitPrice,NetPrice,UnitCost¬
2 1000000,0,48,1,112.4625,98.967,57.3375¬
3 1000000,1,460,1,749.75,659.78,382.25¬
4 1000001,0,1704,1,4.893,4.893,2.492¬
```

```
sql>COPY OFFSET 2 INTO orderrows
FROM '/home/fff/Desktop/CSVs/orderrows.csv'
USING DELIMITERS ',', E'\n', '"';
211875108 affected rows
clk: 56.241 sec
```

Other Tables

I will leave a file for download from the repository listed at the beginning of the book. That file will have SQL for creation and import of all of the Contoso tables. Bellow we can see time for import for two other larger files. Smaller dimension tables are imported almost instantly.

Sales (211M) table will need almost **5** minutes to be imported.

sales.csv

```
1 OrderKey,LineNumber,OrderDate,DeliveryDate,CustomerKey,StoreKey,ProductKey,Quantity,UnitPrice,NetPrice,UnitCost,CurrencyCode,ExchangeRate¬
2 1000000,0,2015-01-01,2015-01-01,1169574,400,48,1,112.4625,98.967,57.3375,GBP,0.64155¬
3 1000000,1,2015-01-01,2015-01-01,1169574,400,460,1,749.75,659.78,382.25,GBP,0.64155¬
4 1000001,0,2015-01-01,2015-01-01,1244485,430,1704,1,4.893,4.893,2.492,USD,1.00000¬
```

```
sql>COPY OFFSET 2 INTO sales
FROM '/home/fff/Desktop/CSVs/sales.csv'
USING DELIMITERS ',', E'\n', '"';
211875108 affected rows
clk: 4:38 min
```

Orders (88M) CSV table will be imported in **62** seconds.

orders.csv

```
1 OrderKey,CustomerKey,StoreKey,OrderDate,DeliveryDate,CurrencyCode¬
2 1000000,1169574,400,2015-01-01,2015-01-01,GBP¬
3 1000001,1244485,430,2015-01-01,2015-01-01,USD¬
4 1000002,784385,290,2015-01-01,2015-01-01,EUR¬
```

```
sql>COPY OFFSET 2 INTO orders
FROM '/home/fff/Desktop/CSVs/orders.csv'
USING DELIMITERS ',', E'\n', '"';
88382161 affected rows
clk: 1:02 min
```

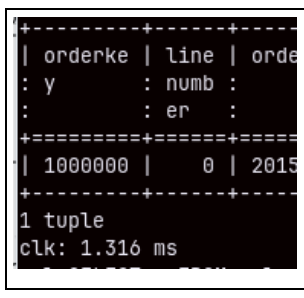
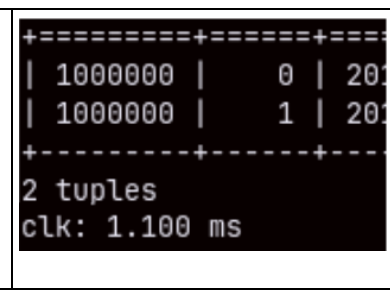
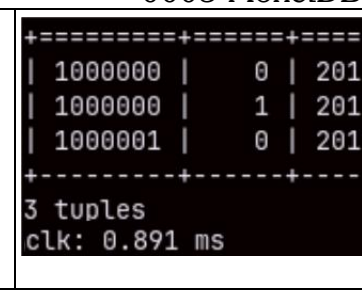
Query Benchmarking

Cold Start

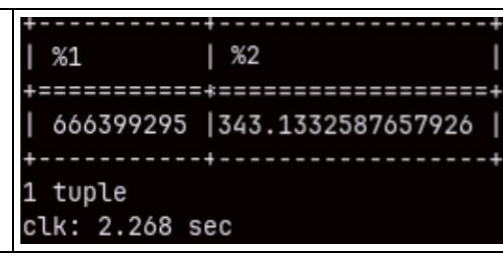
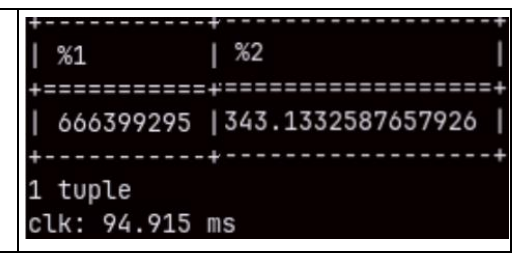
I will now restart my computer. I want to make sure to run a cold query. This simple query bellow will run for 10 seconds. MonetDB database becomes faster as more queries are executed. This ability of MonetDB is called "`cracking`". MonetDB will automatically sort, group and index columns during the SELECT queries. That will make subsequent queries faster.

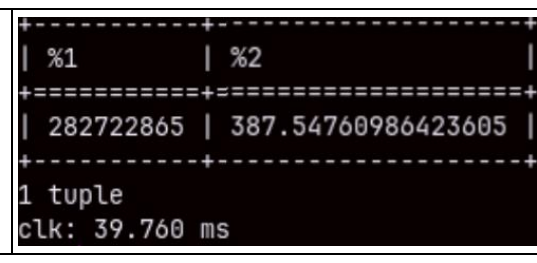
```
SELECT * FROM sales LIMIT 1;
```

```
+---------+------+--------
| orderke | line | orderda
: y       : numb :
:         : er   :
+=========+======+========
| 1000000 |    0 | 2015-01-
+---------+------+--------
1 tuple
clk: 9.458 sec
```

	If we run this query again, it will be executed in just 1.3 miliseconds. This is not the result of caching. If we make a query that takes 2 or 3 rows, we will again see these exceptional speeds.
	`SELECT * FROM sales LIMIT 2;`
	`SELECT * FROM sales LIMIT 3;`
			

Aggregated Queries in MonetDB

If we aggregate two columns in the "`sales`" table, that query will touch 211M rows. It will be fast (2.268 sec.), but we can repeat it to get only 94.915 ms.
 	
`SELECT SUM( quantity ), AVG( unitprice )`
`FROM sales;`
		

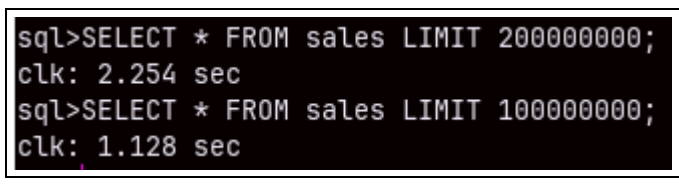
I will run the same query, but this time with a filter.
 	
`SELECT SUM( quantity ), AVG( unitprice )`
`FROM sales WHERE orderdate <= '2020-05-25';`
		We will get the result even faster. This proves that the result is not cached. MonetDB run the query again, but this time "`cracking`" made our query faster.

It's hard to make a benchmark when execution times are constantly changing. So, from now on I will focus on the fastest times.

How Database Reports Execution Time

`SELECT * FROM sales LIMIT 1000000;` **--13 ms** `SELECT * FROM sales LIMIT 2000000;` **--24 ms**	MonetDB is reporting that the second query is slower than the first one. That is something that we are expecting. The problem is that according to my computer clock the first query was finished after 7 seconds, and the second one after 15 seconds.

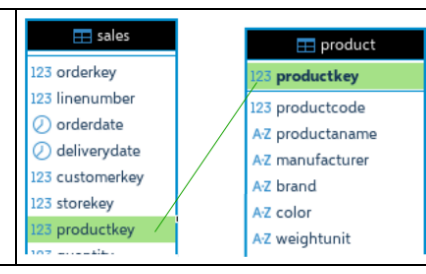
Databases only report the time spent to produce the results in the memory. It will not include the time needed to print the result in the shell or any other client. That is why MonetDB is reporting 13 ms, but I can see the result only after 7 seconds. MonetDB has command to suppress printing of the result in the shell. I will use that command (command explained in 0580) next, to test reading the whole tables.

	This is how long it takes MonetDB to read a large number of rows. Columnar databases are better suited for aggregate queries, but we can see that MonetDB is capable of performing OLTP types of queries quite well.

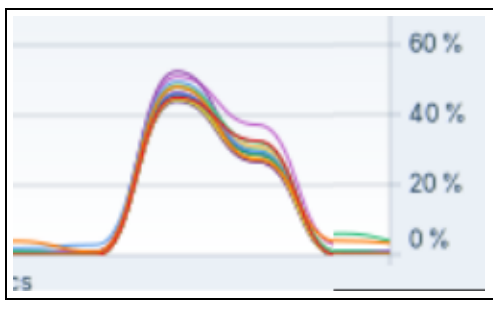
I will disable my command, so we can again see the results of our queries.

Joins

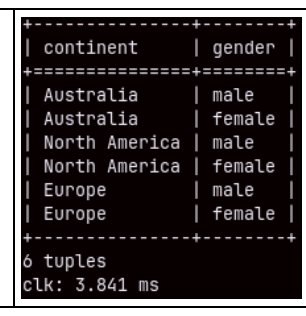
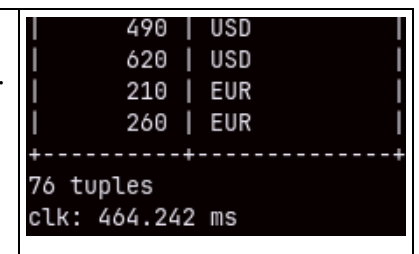
`SELECT productkey, SUM( quantity ), AVG( netprice )` `FROM sales GROUP BY productkey;`	This query will execute for 340 ms. If we want to see brands then we have to make a join between "`product`" and "`sales`" tables.

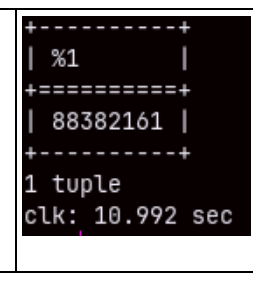
`SELECT brand, SUM( quantity ), AVG( netprice )`
`FROM sales INNER JOIN product`
`  ON sales.productkey = product.productkey`
`GROUP BY brand;`
	The query with a join will last more than 1 second. We can speed it up if we create a foreign key constraint.	

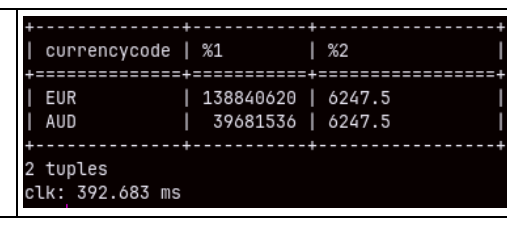
`ALTER TABLE product ADD CONSTRAINT product_pk PRIMARY KEY ( productkey );`
`ALTER TABLE sales ADD CONSTRAINT FKfromProduct FOREIGN KEY ( productkey )`
`REFERENCES product ( productkey );`
	Now that we have foreign key constraint, the query from before will become 200 ms faster. That is 20% faster.

	Query from before is a traditional analytical query. If we look at system monitor, we will see that during the execution of this query the load will be equally distributed between CPU cores. MonetDB is capable to significantly parallelize query execution. That means that our individual queries can be speed up with a CPU that has even more cores.

DISTINCT, LIKE, ROLLUP

From the table "`customer`" we can list distinct continents and genders in 3.841 ms.
 	
`SELECT DISTINCT continent, gender`
`FROM customer;`
		We can get distinct combinations of storekey and currency code from "`sales`" table in half of the second.
		
		`SELECT DISTINCT storekey, currencycode`
		`FROM sales;`
			

If we want to count unique combinations of the orderkey and currencycode, then our query will be slow, it will last almost 11 seconds. But this query has to count 88 million rows.
 	
`SELECT COUNT( * ) FROM`
`  ( SELECT DISTINCT orderkey, currencycode`
`  FROM sales );`
		Query like this will also last 11 seconds.
		
		`SELECT COUNT( * ) FROM`
		`  ( SELECT orderkey, currencycode`
		`  FROM sales`
		`  GROUP BY orderkey, currencycode );`

LIKE operator allows usage of the wild cards. Sign "_" will replace one letter.
 	
`SELECT currencycode, SUM( quantity ), MAX( unitprice )`
`FROM sales`
`WHERE currencycode LIKE '_U_' GROUP BY currencycode;`
	

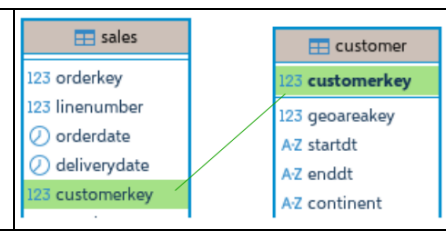
If we use the sign "%" that replaces several characters, then the speed will drop, almost double.

```
SELECT currencycode, SUM( quantity ), MAX( unitprice )
FROM sales
WHERE currencycode LIKE '%U_' GROUP BY currencycode;
```

```
+--------------+----------+----------------
| currencycode | %1       | %2
+==============+==========+================
| EUR          | 138840620 | 6247.5
| AUD          |  39681536 | 6247.5
+--------------+----------+----------------
2 tuples
clk: 777.460 ms
```

Before we test ROLLUP, I will create foreign key constraint between "customer" and "sales" tables.

```
ALTER TABLE customer ADD CONSTRAINT customer_pk PRIMARY KEY ( customerkey );
ALTER TABLE sales ADD CONSTRAINT FKfromCustomer FOREIGN KEY ( customerkey )
REFERENCES customer ( customerkey );
```



This time we have unusually slow query. It will take full 10 seconds.

```
SELECT continent, title, SUM( quantity )
FROM customer INNER JOIN sales
   ON customer.customerkey = sales.customerkey
GROUP BY ROLLUP( continent, title );
```

Query with union would be much better choice for this. Only **1.837** seconds.

```
SELECT continent, title, SUM( quantity )
FROM sales INNER JOIN customer ON sales.customerkey = customer.customerkey
GROUP BY continent, title
UNION
SELECT continent, null, SUM( quantity )
FROM sales INNER JOIN customer ON sales.customerkey = customer.customerkey
GROUP BY continent
UNION
SELECT null, null, SUM( quantity ) FROM sales;
```

```
+---------------+-------+-----------+
| continent     | title | %1        |
+===============+=======+===========+
| Europe        | Mrs.  |  48051308 |
| North America | Mr.   | 203567434 |
| Europe        | Mr.   | 103911818 |
| Australia     | Mr.   |  19457712 |
| North America | Mrs.  |  94577665 |
| Europe        | Ms.   |  53519789 |
| North America | Ms.   | 104219984 |
| Australia     | Ms.   |  10068474 |
| Australia     | Mrs.  |   8946620 |
| North America | Dr.   |  12612223 |
| Europe        | Dr.   |   6257538 |
| Australia     | Dr.   |   1208730 |
| Europe        | null  | 211740453 |
| North America | null  | 414977306 |
| Australia     | null  |  39681536 |
| null          | null  | 666399295 |
+---------------+-------+-----------+
16 tuples
clk: 10.118 sec
```

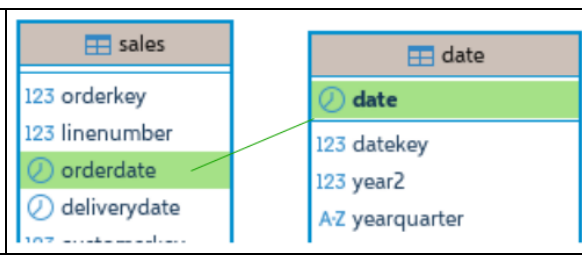
```
+---------------+-------+-----------+
| continent     | title | %1        |
+===============+=======+===========+
| Europe        | Mrs.  |  48051308 |
| North America | Mr.   | 203567434 |
| Europe        | Mr.   | 103911818 |
| Australia     | Mr.   |  19457712 |
| North America | Mrs.  |  94577665 |
| Europe        | Ms.   |  53519789 |
| North America | Ms.   | 104219984 |
| Australia     | Ms.   |  10068474 |
| Australia     | Mrs.  |   8946620 |
| North America | Dr.   |  12612223 |
| Europe        | Dr.   |   6257538 |
| Australia     | Dr.   |   1208730 |
| Europe        | null  | 211740453 |
| North America | null  | 414977306 |
| Australia     | null  |  39681536 |
| null          | null  | 666399295 |
+---------------+-------+-----------+
16 tuples
clk: 1.837 sec
```

This show us that there is still room for improvements in MonetDB optimizer.

Window Functions

I will first add foreign key constraint between "sales" and "date" tables.

```
ALTER TABLE date ADD CONSTRAINT date_pk PRIMARY KEY ( date );
ALTER TABLE sales ADD CONSTRAINT FKfromDate FOREIGN KEY ( orderdate )
REFERENCES date ( date );
```



We can use LAG function to compare sales for the current and the previous row. Each value from quantity column will have a pair in the BeforeQty column, except in the first row (no previous). Quantity in that first row will make a difference. This query will execute in 2 seconds.

```
SELECT ( SUM( Qty ) - SUM( BeforeQty ) ) AS Difference FROM
( SELECT Quantity AS Qty,
  LAG( Quantity, -1 ) OVER ( ORDER BY date ) AS BeforeQty
  FROM date INNER JOIN sales ON date.date = sales.orderdate );
```

```
+------------+
| difference |
+============+
|          1 |
+------------+
1 tuple
clk: 1.959 sec
```

For each row in the table, we will calculate the average quantity of the previous 6 rows and the current row. At the end will see sums of the quantity and these rolling averages. This query will last 29 seconds.

```
SELECT date, SUM( Quantity ),
    AVG( SUM( Quantity ) ) OVER
        ( ORDER BY date ROWS BETWEEN 7 PRECEDING AND 1 PRECEDING )
            AS SevenDaysAVG
FROM date INNER JOIN sales ON date.date = sales.orderdate
GROUP BY date
ORDER BY date;
```

```
+-----------+--------------------------+
| SumQty    | SumAvgBefore7Qty         |
+===========+==========================+
| 666399295 |       666399288.1857142 |
+-----------+--------------------------+
1 tuple
clk: 29.418 sec
```

Updates

MonetDB should be slow for updates, but it managed to update 44 milion rows for just 4.19 seconds.

```
UPDATE sales SET currencycode = 'EU' WHERE currencycode = 'EUR';
```

```
sql>UPDATE sales SET currencycode = 'EU' WHERE currencycode = 'EUR';
44143743 affected rows
clk: 4.190 sec
```

We can confirm that all of the values 'EUR' are updated to 'EU'.

```
SELECT DISTINCT currencycode FROM sales;
```

```
+--------------+
| currencycode |
+==============+
| GBP          |
| USD          |
| EU           |
| AUD          |
| CAD          |
+--------------+
```

Problematic Queries

Double Grouping

Double grouping is when we first group our data, and then we group that result. For example, we will total sales quantity per customerkey, and then we will count customers per total quantity. We will count how many customers have the same total quantity.

This query is problematic because while the first grouping can be fast, the second one could be much longer. The result of the first grouping will have 2M rows, because we have so much customers. In the second stage, we have to group these 2M rows, and that is when I expect the performance to become bad.

```
SELECT customer.customerkey, SUM( quantity ) AS TotQty
FROM customer INNER JOIN sales ON customer.customerkey = sales.customerkey
GROUP BY customer.customerkey;
```

In the first phase I will measure how much time is needed to group by customer. It is 5 seconds because there are 2 million customers.

Second phase:

```
SELECT TotQty, COUNT( customerkey ) FROM
( SELECT customer.customerkey, SUM( quantity ) AS TotQty
  FROM customer INNER JOIN sales ON customer.customerkey = sales.customerkey
  GROUP BY customer.customerkey ) as FirstPhase
GROUP BY TotQty;
```

We can see on the image that we have **2,663** customers with total quantity of **333** items, and only **two** with **1,130** items. The time for execution is again 5 seconds. This is something I didn't expect. I am pleasantly surprised. I can tell you that these kinds of queries are problematic for Power BI database (SSAS).

```
+--------+------+
| totqty | %2   |
+========+======+
|    333 | 2663 |
|    466 | 3867 |
|    272 | 2871 |
|   1250 |    1 |
|   1260 |    1 |
|   1130 |    2 |
+--------+------+
1194 tuples
clk: 4.764 sec
```

Aggregated Query from Two Fact Tables (Stitch Query)

This time I will create foreign key constraint on the "OrderRows" (211M) table. I want to aggregate sales and orderrows per product brend.

```
ALTER TABLE orderrows ADD CONSTRAINT FK_Product FOREIGN KEY ( productkey ) REFERENCES product ( productkey );
```

"Stitch" query is when we aggregate two fact tables per the same dimension and we get two data sets as a result. Then we join those two data sets in the final result. This is how we aggregate values from two fact tables.

Sales Grouped			Orders Grouped			Stitched Query		
Brand	Sales		Brand	Orders		Continent	Sales	Orders
Contoso	50		Contoso	40		Contoso	50	40
Fabrikam	100		Fabrikam	80		Fabrikam	100	80
Litware	30	JOIN	Litware	20	==>	Litware	30	20
Proseware	40		Proseware	30		Proseware	40	30

Query bellow will last 7.5 seconds. This is longer than I expected. If we ran subqueries separately the time would be just 900 ms each. Because we only have 15 brands, it is surprising that it will take 5 seconds just to join two small tables.

```
SELECT S.brand, Sq, Oq FROM
( SELECT Brand, SUM( quantity ) Sq FROM Product INNER JOIN Sales ON Product.Productkey = Sales.ProductKey GROUP BY Brand ) S
INNER JOIN
( SELECT Brand, SUM( quantity ) Oq FROM Product INNER JOIN Orderrows ON Product.Productkey = OrderRows.ProductKey GROUP BY Brand ) O
ON S.Brand = O.Brand;
```

If we "UNION ALL" our subqueries, the execution will last 7.5 seconds, too.

```
SELECT Brand, SUM( quantity ) Sq FROM Product INNER JOIN Sales ON Product.Productkey = Sales.ProductKey GROUP BY Brand
UNION ALL
SELECT Brand, SUM( quantity ) Oq FROM Product INNER JOIN Orderrows ON Product.Productkey = OrderRows.ProductKey GROUP BY Brand;
```

I have tried to read two small subqueries into python, and then to join them with pandas. Python reported execution time of just 1.2 seconds. It is strange that we can get the final result faster by combining MonetDB and Pandas, then just by using MonetDB.

```
WITH
S AS
( SELECT productkey, SUM( quantity ) AS Sq FROM Sales GROUP BY productkey ),
O AS
( SELECT productkey, SUM( quantity ) AS Oq FROM Orderrows GROUP BY productkey ),
PS AS
( SELECT brand, SUM( Sq ) AS SQty FROM Product INNER JOIN S ON Product.productkey = S.productkey GROUP BY brand ),
PO AS
( SELECT brand, SUM( Oq ) AS OQty FROM Product INNER JOIN O ON Product.productkey = O.productkey GROUP BY brand )
SELECT PS.brand, Sqty, OQty FROM PS INNER JOIN PO ON PS.brand = PO.brand;
```

We can reduce our fact tables by grouping them by productkey and then following the same logic. This approach would speed up our query to 5.5 seconds.

```
WITH S AS ( SELECT productkey, SUM(quantity) AS Sq
            FROM sales
            GROUP BY productkey ),
O AS ( SELECT productkey, SUM(quantlty) AS Oq
       FROM orderrows
       GROUP BY productkey )
SELECT P.brand, SUM(S.Sq) AS Sq, SUM(O.Oq) AS Oq
FROM product P
 LEFT JOIN S ON S.productkey = P.productkey
 LEFT JOIN O ON O.productkey = P.productkey
GROUP BY P.brand;
```

This would be the fastest version of this query. It would execute in only 3 seconds.

In this query, we would use the the last subquery to join reduced sales and orderRows tables, with the product table.

```
+----------------------+-----------+-----------+
| brand                | sq        | oq        |
+======================+===========+===========+
| Wide World Importers |  73401968 |  73401968 |
| Southridge Video     |  93572541 |  93572541 |
| A. Datum             |  14959469 |  14959469 |
| Fabrikam             |  23883270 |  23883270 |
| Tailspin Toys        |  50282184 |  50282184 |
| Contoso              | 160054266 | 160054266 |
| Litware              |  14560225 |  14560225 |
| Contoso              |  16875425 |  16875425 |
| Adventure Works      |  67361187 |  67361187 |
| Fabrikam             |   7669532 |   7669532 |
| Northwind Traders    |  16638966 |  16638966 |
| Proseware            |  38335616 |  38335616 |
| The Phone Company    |  86222513 |  86222513 |
| Adventure Works      |    446262 |    446262 |
| Litware              |   2135871 |   2135871 |
+----------------------+-----------+-----------+
15 tuples
clk: 3.187 sec
```

INSERT INTO SELECT

I will use "INSERT INTO SELECT" to make "Sales" table bigger.
Before doing that, I will remove FK constraints. I will also change optimizer.

```
ALTER TABLE sales DROP CONSTRAINT fkfromproduct;
ALTER TABLE sales DROP CONSTRAINT fkfromcustomer;
ALTER TABLE sales DROP CONSTRAINT fkfromdate;
SET sys.optimizer = 'minimal_pipe';
```

I will now run this statement to make my sales table twice bigger.

```
INSERT INTO sales SELECT * FROM sales;
```

MonetDB needed 5 minutes to do this.

```
sql>INSERT INTO sales SELECT * FROM sales;
211875108 affected rows
clk: 5:24 min
```

I will do this 1 more time. That will double the number of rows to 844M. That was done in 9:48 minutes.

Then, I will again read from the CSV file into this table. That will add another 211M rows, so in total "sales" table will now have one billion rows.

```
sql>SELECT COUNT( * ) FROM sales;
+------------+
| %1         |
+============+
| 1059375540 |
+------------+
```

I will recreate foreign key constraint toward "product" table.

```
ALTER TABLE sales ADD CONSTRAINT FKfromProduct FOREIGN KEY ( productkey ) REFERENCES product ( productkey );
```

I will run now this query twice. The first time it will end after 28 seconds, and the second time after 5,5 seconds.

```
SELECT brand, SUM( quantity ), AVG( netprice )
FROM sales INNER JOIN product
   ON sales.productkey = product.productkey
GROUP BY brand;
```

```
| Adventure Works      |   2231310 |
| Litware              |  10679355 |
+----------------------+-----------+
15 tuples
clk: 28.286 sec
```

```
| Adventure Works      |   223 |
| Litware              |  1067 |
+----------------------+-------
15 tuples
clk: 5.405 sec
```

```
SELECT color, SUM( quantity ), AVG( netprice )
FROM product INNER JOIN sales
   ON sales.productkey = product.productkey
GROUP BY color;
```

Immediately after, I run the same query, by it was grouped by color. The time was again 5 seconds.

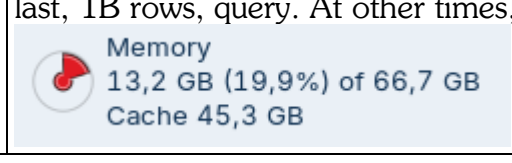
We can see that performance is good, even with 1B rows.

```
| blue         |   1592
| Transparent  |   3716
+--------------+-------
17 tuples
clk: 5.399 sec
```

Conclusions

We can conclude some things:
- MonetDB is usually very fast.
- Initially, until the database worm up, queries can be slow.
- We should always set foreign key constraints to achieve speed boost.
- Some kinds of queries are better optimized than others.

- MonetDB does not use much memory. During the import of the "sales" table, the RAM usage increased from 4 to 13 GB. It was the same during this last, 1B rows, query. At other times, the usage was much less.



0010 Install MonetDB Server on Ubuntu Linux

Notice: In this section, we will create a VOC database. This database will be used as the main database for most of the subsequent sections.

Getting the Codename of our Ubuntu Version

First, we need to know the code name of our Ubuntu version. We can find that by reading from the file "`os-release`". From this file we can read only the line that has words "`VERSION_CODENAME`" inside of it.

`cat /etc/os-release | grep VERSION_CODENAME`	`fffovde@FffOvdeKomp:~$ cat /etc/os-release | grep VERSION_CODENAME` `VERSION_CODENAME=focal`

Our Ubuntu codename is "`focal`". It is also possible to use command:

`lsb_release -cs`	`fffovde@FffOvdeKomp:~$ lsb_release -cs` `focal`

We can see from the command line above that our user account is "`fffovde`". "`FffOvdeKomp`" is the name of our computer.

Installing GPG key

GPG key is a key that Ubuntu will use to verify MonetDB packages before installing them.

First, we will create folder `/etc/apt/keyrings`, if we don't have it. We will use permissions 755 on this folder. Administrator will have full access to this folder, but other users will have read and execute rights. This is necessary because "`apt`" application will access this GPG key, not only with administrator rights, but also as a special low-privilege user.

```
sudo mkdir -p /etc/apt/keyrings          #option -p means that we will create parent directories, if they do not exist.
sudo chmod 0755 /etc/apt/keyrings
```

We will use wget to download GPG key from the internet. Option "**-O**" means output.

```
sudo wget -O /etc/apt/keyrings/monetdb.gpg https://dev.monetdb.org/downloads/MonetDB-GPG-KEY.gpg
```

We will also change permissions on this binary ".gpg" file.

```
sudo chmod 0644 /etc/apt/keyrings/monetdb.gpg
```

```
fffovde@FffOvdeKomp:~$ wget -qO- https://dev.monetdb.org/downloads/MonetDB-GPG-KEY.gpg | sudo gpg --dearmor -o /etc/apt/keyrings/monetdb.gpg
fffovde@FffOvdeKomp:~$ sudo chmod 0644 /etc/apt/keyrings/monetdb.gpg
fffovde@FffOvdeKomp:~$
```

This *monetd.gpg* file is a binary file. We can read its content with the command:

```
gpg --no-default-keyring --keyring /etc/apt/keyrings/monetdb.gpg --fingerprint
```

```
--no-default-keyring                    #This means that we will not read other GPG keys.
--keyring /etc/apt/keyrings/monetdb.gpg #We will read only this GPG key.
--fingerprint                           #Because GPG key is binary file, we will only read its textual presentation, its "fingerprint".
                                        #This fingerprint is a long hexadecimal number.
```

If result of this command is equal to " DBCE 5625 94D7 1959 7B54 CE85 3F1A D47F 5521 A603" for MonetDB, then that means that we have installed the correct key.
There is also a key "8289 A5F5 75C4 9F50 22F8 EE20 F654 63E2 DF0E 54F3", but that key is for versions of MonetDB older than 11.49.

```
fffovde@FffOvdeKomp:~$ gpg --no-default-keyring --keyring /etc/apt/keyrings/monetdb.gpg --fingerprint
          /etc/apt/keyrings/monetdb.gpg
          ----------------------------
          pub   rsa2048 2017-08-02 [SC]
                8289 A5F5 75C4 9F50 22F8  EE20 F654 63E2 DF0E 54F3
          uid           [ unknown] MonetDB Database System Packager <info@monetdb.org>
          sub   rsa2048 2017-08-02 [E]

          pub   rsa3072 2023-09-26 [SC]
                DBCE 5625 94D7 1959 7B54  CE85 3F1A D47F 5521 A603
          uid           [ unknown] MonetDB Database System Packager <info@monetdb.org>
```

Adding a Repository Where MonetDB is Stored

Next, in folder "`/etc/apt/sources.list.d`" we will create a file with the name "`monetdb.list`".


```
#jump to that folder
cd /etc/apt/sources.list.d
#create new file, you will be asked to provide password
sudo touch monetdb.list
```

Inside of this file we must place this text. These are addresses to MonetdDB repository.

```
deb [signed-by=/etc/apt/keyrings/monetdb.gpg] https://dev.monetdb.org/downloads/deb/ focal monetdb
deb-src [signed-by=/etc/apt/keyrings/monetdb.gpg] https://dev.monetdb.org/downloads/deb/ focal monetdb
```

We can add this text by running these two lines in our terminal:

```
sudo sh -c 'echo "deb [signed-by=/etc/apt/keyrings/monetdb.gpg] https://dev.monetdb.org/downloads/deb/ focal monetdb" >> monetdb.list'
sudo sh -c 'echo "deb-src [signed-by=/etc/apt/keyrings/monetdb.gpg] https://dev.monetdb.org/downloads/deb/ focal monetdb" >> monetdb.list'
```

Now our file looks like this:

MonetDB Installation

Now we can install MonetDB. First, we will update our list of available software with the command:

```
sudo apt update
```

Then we can install MonetDB server and client: `sudo apt install monetdb5-sql monetdb-client`	`fffovde@FffOvdeKomp:~$ sudo apt install monetdb5-sql monetdb-client` `Reading package lists... Done`

Creating a DBfarm

	In MonetDB, databases are located inside a folder usually called "`DBfarm`". In addition to the databases, in this folder we will find configuration files with settings for the DBfarm.		Monetdb**d**, linux **d**aemon, is used to initialize DBfarm by placing configuration files inside it. We use this daemon for changing DBfarm settings and for communication with databases. Monetdbd manages DBfarm and its databases.

`monetdbd create /home/fffovde/DBfarm1`	We will use "`monetdbd`" to create a DBfarm on our disk.
`fffovde@FffOvdeKomp:~$ ls -A  /home/fffovde/DBfarm1` `.merovingian_properties`	Inside of this folder, a new file "`.merovingian_properties`" will appear. `ls -A  /home/fffovde/DBfarm1`

The Merovingian dynasty was the ruling family of the Franks from the mid-5th century until 751. This dynasty ruled the Netherlands, the country from which MonetDB originates. MonetDB is using this term for some of its internal files and commands.

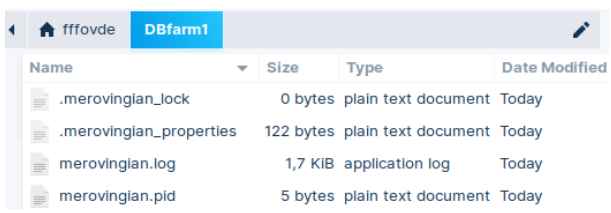
`fffovde@FffOvdeKomp:~$ cat /home/fffovde/DBfarm1/.merovingian_properties`
`# DO NOT EDIT THIS FILE - use monetdb(1) and monetdbd(1) to set properties`
`# This file is used by monetdbd`
 	
`control=false`
	If we look inside of this file, we will find only **one property**. All other properties are using default values. We can read those default values by command:
	`monetdbd get all /home/fffovde/DBfarm1`

```
fffovde@FffOvdeKomp:~$ monetdbd get all /home/fffovde/DBfarm1
   property            value
hostname          FffOvdeKomp
dbfarm            /home/fffovde/DBfarm1
status            no monetdbd is serving this dbfarm
mserver           unknown (monetdbd not running)
logfile           /home/fffovde/DBfarm1/merovingian.log
pidfile           /home/fffovde/DBfarm1/merovingian.pid
loglevel          information
sockdir           /tmp
listenaddr        localhost
port              50000
exittimeout       60
forward           proxy
discovery         true
discoveryttl      600
control           no
passphrase        <unset>
snapshotdir       <unset>
snapshotcompression  .tar.lz4
keepalive         60
mapisock          /tmp/.s.monetdb.50000
controlsock       /tmp/.s.merovingian.50000
```

property	value
hostname	FffOvdeKomp
dbfarm	/home/fffovde/DBfarm1
status	no monetdbd is serving this dbfarm
mserver	unknown (monetdbd not running)
logfile	/home/fffovde/DBfarml/merovingian.log
pidfile	/home/fffovde/DBfarml/merovingian.pid
loglevel	information
sockdir	/tmp
listenaddr	localhost
port	/50000
exittimeout	60
forward	proxy
discovery	true
discoveryttl	600
control	no
passphrase	<unset>
snapshotdir	<unset>
snapshotcompression	.tar.lz4
mapisock	/tmp/.s.monetdb.50000
controlsock	/tmp/.s.merovingian.50000

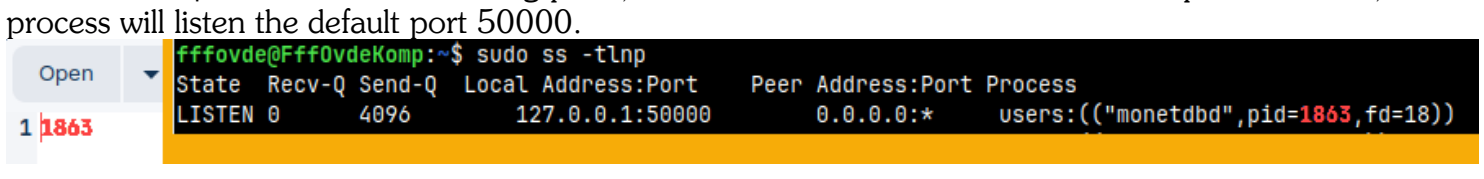
Now, that DBfarm is created, I will start the daemon. We can use daemon to control DBfarm.	`monetdbd start /home/fffovde/DBfarm1`

`fffovde@FffOvdeKomp:~$ ls -A /home/fffovde/DBfarm1` `.merovingian_lock  merovingian.log  merovingian.pid  .merovingian_properties`	Inside of the DBfarm1, we now have 4 files: `ls -A /home/fffovde/DBfarm1`



File "*.merovingian_lock*" is empty. This file probably just signalizes that this is DBfarm.

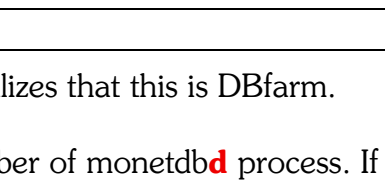
File "*merovingian.pid*" has the number 1863. This is the number of monetdb**d** process. If we use command "`sudo ss -tlnp`" to show us all listening ports, we will see the name monetdb**d** beside process 1863, and this process will listen the default port 50000.



`fffovde@FffOvdeKomp:~$ cat /home/fffovde/DBfarm1/merovingian.log`
`2025-09-26 19:09:58 MSG merovingian[1852]: Merovingian 11.53.3 (Mar2025) starting`
`2025-09-26 19:09:58 MSG merovingian[1852]: monitoring dbfarm /home/fffovde/DBfarm1`
`2025-09-26 19:09:58 MSG merovingian[1852]: accepting connections on TCP socket ipó-localhost:50000`
	We can also read content of the log file.
	
	`cat /home/fffovde/DBfarm1/merovingian.log`

Creation of a Database

While monetdb**d** is used to manage DBfarm, "`monetdb`" console application is used to manage individual databases. In the background, "`monetdb`" will send our commands to monetdb**d**, and monetdb**d** will be the one exercising direct control over databases. So, we control databases from "`monetdb`", but through the power of monetdb**d**.

This is how we create a database with the name "`voc`": `monetdb create voc`	`fffovde@FffOvdeKomp:~$ monetdb create voc` `created database in maintenance mode: voc`	A folder with the name "`voc`" will appear inside of the DBfarm1 directory.	

When database is created, it will only have one default user. That user is administrator "`monetdb`", and he has default password "`monetdb`". Database will be created in the maintenance mode. That means that only administrator will be able to start the database, and only on the local computer.

Administrator should start the database with this command. Then he should log into database, and he should change his default password to some other secret and complex password. We will not do that this time, we will continue using the default password "monetdb".

I will change the mode of the database, and I will take it out of the maintenance mode. We can do this to make database available to all the users (although we currently don't have other users).
`monetdb release voc`

```
fffovde@FffOvdeKomp:~$ monetdb release voc
taken database out of maintenance mode: voc
```

We can start our database. After the database start, it will be ready for users to log in.
`monetdb start voc`
We can check the status of the database with the command:
`monetdb status`

```
fffovde@FffOvdeKomp:~$ monetdb status
name  state   health                remarks
voc   S                 mapi:monetdb://FffOvdeKomp:50000/voc
```

The process with our database server is called "mserver5". This process will run when the database is opened:
`pgrep mserver5`

This command will return process ID of our database server process.

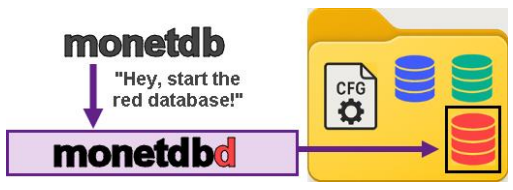
```
fffovde@FffOvdeKomp:~$ pgrep mserver5
2128
```

Logging into Database

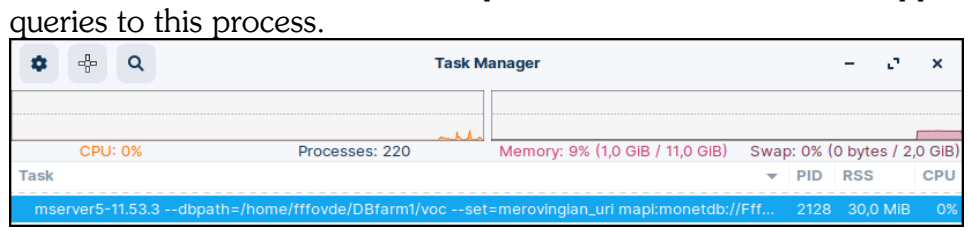
Now that our database is working and is listening on port 50000, we can try to use it. We will now use another application, with the name "mclient". Let's first recapitulate three console applications used by MonetDB:

"monetdbd" is managing DBfarm.

"monetdb" is managing individual database, with the help of "monetdbd".



Database will run as a "mserver5" process. We'll use "mclient" application to send queries to this process.



We will login to the "voc" database as a "monetdb" user. Password is the default password "monetdb".
`mclient -u monetdb -d voc`

```
fffovde@FffOvdeKomp:~$ mclient -u monetdb -d voc
password:
```

This will be our welcome screen. We will get "sql>" prompt. There we can type our queries.

```
fffovde@FffOvdeKomp:~$ mclient -u monetdb -d voc
password:
Welcome to mclient, the MonetDB/SQL interactive terminal (Mar2025)
Database: MonetDB v11.53.3 (Mar2025), 'mapi:monetdb://FffOvdeKomp:50000/voc'
FOLLOW US on https://github.com/MonetDB/MonetDB
Type \q to quit, \? for a list of available commands
auto commit mode: on
sql>
```

I can run this query in my database:

`SELECT 'columnValues' as columnName;`

```
sql>SELECT 'columnValues' as columnName;
+--------------+
| columnname   |
+==============+
| columnValues |
+--------------+
```

```
fffovde@FffOvdeKomp:~$ mclient -u monetdb -d voc
password:
```

We can exit the "mclient" program by typing the word "quit".

How to Stop our Server?

This is how we can stop "mserver5" process, the process of the Monetdb database.
`monetdb stop voc`

```
fffovde@FffOvdeKomp:~$ monetdb stop voc
stopping database 'voc'... done
```

```
fffovde@FffOvdeKomp:~$ tail -1 /home/fffovde/DBfarm1/merovingian.log
2025-09-26 21:10:14 MSG merovingian[1863]: database 'voc' (2128) has exited with exit status 0
```

Log file will tell us what happened.
`tail -1 /home/fffovde/DBfarm1/merovingian.log`

We can put our database in maintenance mode at any time. It doesn't matter if database is opened or closed. We use "lock" command.
`monetdb lock voc`

```
fffovde@FffOvdeKomp:~$ monetdb lock voc
put database under maintenance: voc
```

Next time, only administrator, on the local computer, can start the database with "`monetdb start voc`" command. He can start the database in exclusive mode, so that he can run some maintenance operations on the database (he can do backup, or he can make changes in the schema). We saw previously that database can be taken out of maintenance mode with "`monetdb release voc`" command. After that, any user can login to a database.

We can also stop "monetdbd" daemon.

`monetdbd stop /home/fffovde/DBfarm1`

```
fffovde@FffOvdeKomp:~$ tail -1 /home/fffovde/DBfarm1/merovingian.log
2025-09-26 21:13:53 MSG merovingian[1863]: Merovingian 11.53.3 stopped
```

Log file will show us that daemon has stopped.

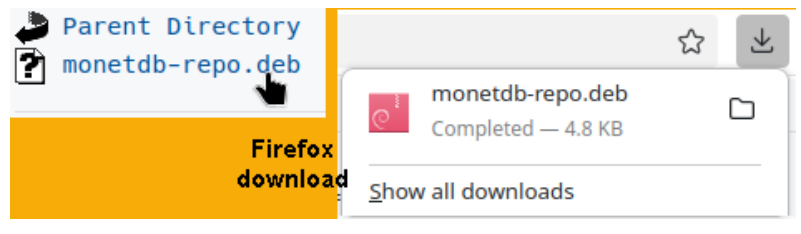
Install MonetDB in Alternative Way

```
fff@fffkomp:~$ cat /etc/os-release | grep VERSION_CODENAME
VERSION_CODENAME=noble
```

This time I will jump to the newer version of the Ubuntu. It is "noble".
`cat /etc/os-release | grep VERSION_CODENAME`

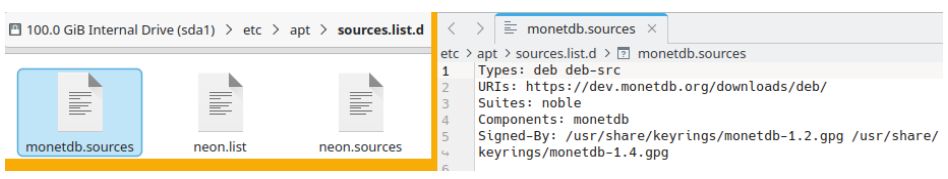
```
Parent Directory
bookworm/        2025-08-18 08:39
bullseye/        2025-08-18 08:39
jammy/           2025-08-18 08:39
noble/           2025-08-18 08:39
plucky/          2025-08-18 08:39
trixie/          2025-08-18 08:39
```

Then, I will go the web page https://www.monetdb.org/downloads/deb/repo/. On this web page we have a list of the newest versions of the Ubuntu and Debian. One of the versions is "noble". Enter that folder, click on "monetdb-repo.deb". Firefox will download this file.

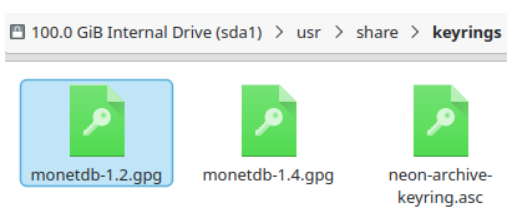


```
fff@fffkomp:~$ sudo apt install /home/fff/Downloads/monetdb-repo.deb
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
```

We will install this "`monetdb-repo.deb`" package.
`sudo apt install /home/fff/Downloads/monetdb-repo.deb`



This package will add the file "`monetdb.sources`" inside of the "`/etc/apt/sources.list.d`". This file has the same content as the file "`monetdb.list`" that we have created by hand during the original installation.



This "`monetdb-repo.deb`" package will also provide GPG keys that can be found on this location "`/usr/share/keyrings/monetdb-1.4.gpg`". We now understand that we can prepare our computer for monetdb installation by using this package, instead of doing everything by hand.

```
sudo apt update
sudo apt install monetdb-sql monetdb-client
```

We can now continue installing MonetDB in the standard way.
The process is the same as above.

Uninstalling Of the MonetDB

```
monetdbd create /home/fff/DBfarm1
monetdbd start /home/fff/DBfarm1
```

I will uninstall MonetDB on the "`noble`" server.
We will first create and start a DBfarm, because I want to show you the whole process.

For uninstallation process, if monetdb**d** is running, we must stop it.

```
monetdbd stop /home/fff/DBfarm1
fff@fffkomp:~$ monetdbd stop /home/fff/DBfarm1
```

We will now list all the packages that have "`monetdb`" in their name.

`dpkg -l | grep monetdb`

```
fff@fffkomp:~$ dpkg -l | grep monetdb
ii  libmonetdb-client28    11.53.13      amd64        MonetDB client/server interface library
ii  libmonetdb-mutils      11.53.13      amd64        MonetDB mutils library
ii  libmonetdb-stream28    11.53.13      amd64        MonetDB stream library
ii  libmonetdb30           11.53.13      amd64        MonetDB core library
ii  monetdb-client         11.53.13      amd64        MonetDB database client
ii  monetdb-repo           0.1-2         all          MonetDB Debian/Ubuntu Repository
ii  monetdb-server         11.53.13      amd64        MonetDB database server
ii  monetdb-sql            11.53.13      amd64        MonetDB SQL support
ii  monetdb5-sql           11.53.13      all          transitional package
```

We will remove all those packages. When we delete "`monetdb-repo`", files with repositories and gpg keys will also be deleted.

`sudo apt purge libmonetdb-client28 libmonetdb-mutils libmonetdb-stream28 libmonetdb30 monetdb-client monetdb-repo monetdb-server monetdb-sql monetdb5-sql`

```
fff@fffkomp:~$ sudo apt purge libmonetdb-client28 libmonetdb-mutils libmonetdb-stream28 libmonetdb30 monetdb-client monetdb-repo monetdb-server monetdb-sql monetdb5-sql
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
Starting pkgProblemResolver with broken count: 0
```

We will clean all the possible remains:
`sudo apt autoremove`
`sudo apt autoclean`

```
fff@fffkomp:~$ sudo apt autoremove
Reading package lists... Done
Building dependency tree       Done
fff@fffkomp:~$ sudo apt autoclean
Reading package lists... Done
Building dependency tree       Done
```

We still have "`monetdb`" group and user.
`cat /etc/passwd | grep monetdb`
`getent group monetdb`

```
fff@fffkomp:~$ cat /etc/passwd | grep monetdb
monetdb:x:120:125::/var/lib/monetdb:/usr/sbin/nologin
fff@fffkomp:~$ getent group monetdb
monetdb:x:125:fff
```

We will delete the group and the user.
`sudo deluser monetdb`
`sudo delgroup monetdb`
Actually, when we delete the user, the group will also be deleted, so we don't have to delete it separately.

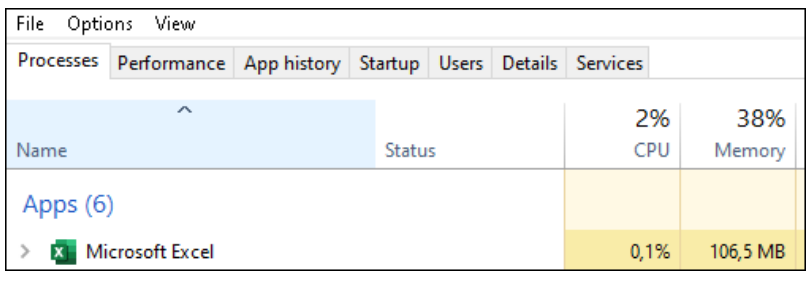
```
fff@fffkomp:~$ sudo deluser monetdb
info: Removing crontab ...
info: Removing user `monetdb' ...
fff@fffkomp:~$ sudo delgroup monetdb
warn: The group `monetdb' does not exist.
```

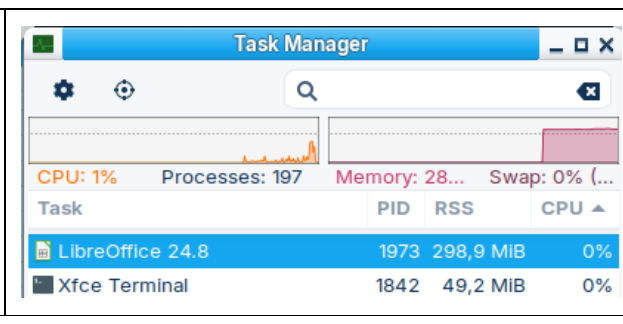
This will not delete DBfarms, only MonetDB application.

0020 SystemD, Monetdbd, Mserver5: Clarification

Computer Process

When we start one program (e.g., Excel), we'll start one process.





Process occupies part of a CPU, part of memory and can only see files that are opened by that process. In that way, process is like a small virtual machine. Excel process is working inside of such small virtual machine and can only access workbooks that we have opened inside of that Excel (e.g., Book1.xlsx and Book2.xlsx). The purpose of a process is to isolate one program from everything else.

Processes, Daemons and Services

We can divide processes into 4 groups:

- **User process** is a process with a friendly interface toward human users (like Notepad.exe).
- **Daemon process** is a process with a friendly interface toward other programs ("Windows Time" (w32time) service).
- **Service is a daemon** which provides interface to some essential functionality. For example, Apache server is a service because through it we get our web pages. In the background, Apache server will pull needed data from the MySQL database which is another daemon. In this setup, Apache server is a service daemon, and MySQL database is a normal daemon.
- Some processes are in between. Microsoft Outlook is a user process dominantly, but it also has an VBA api, so it is partially a daemon.

We can see that the major difference between these processes is stemming from their indented usage. Usually, daemons will start when the OS starts and they will work in the background without users' interaction. On the other side, daemons can also be started manually and users can interact with them. We have already saw an example of such, manually started, daemon:

```
monetdbd start /home/fffovde/DBfarm1
monetdbd get all /home/fffovde/DBfarm1
```

Official Processes

In the "real world" other definitions are used. Something is a daemon or service only if it is officially registered with the operating system. Only processes registered in "Systemd" on Linux systems are called Daemons. In Windows, only processes registered in SCM (Service Control Manager) are called Services. Systemd and SCM are operating system components used to automatically start official daemons and services in the correct order, control resource usage, access rights, log, and restart failed services. From an OS perspective, this is the only way to distinguish between ordinary and special processes. Since Windows and Linux use such definitions, most people will also use those definitions.

Two Ways to Manually Start Monetdbd Daemon

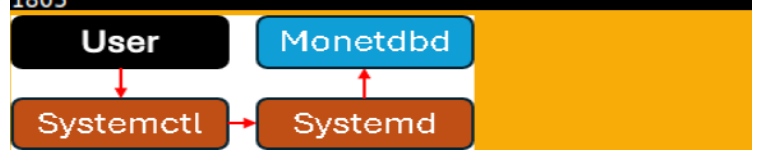
We will first check that Monetdbd daemon is not working. "`pgrep`" command is "`process grep`". We are searching the process by its name. Pgrep should return ID of a monetdbd process, but our process is not active.

```
pgrep monetdbd
```

```
fffovde@FffOvdeKomp:~$ pgrep monetdbd
fffovde@FffOvdeKomp:~$
```

We will then start our daemon using systemctl. "`Systemctl`" is a console program used to control "`systemd`". We already saw that systemd is OS component used to manage daemons.
`sudo systemctl start monetdbd`
"`Pgrep`" command will confirm that our MonetDB daemon is now working.
`pgrep monetdbd`
If we start daemon in this way, then the control of the "`monetdbd`" daemon is under systemd, and we have to use "`systemctl`" auxiliary program to send commands to systemd, and systemd will control "`monetdbd`".

```
fffovde@FffOvdeKomp:~$ sudo systemctl start monetdbd
[sudo] password for fffovde:
fffovde@FffOvdeKomp:~$ pgrep monetdbd
1865
```



We can also start monetdbd daemon with the "`start`" command. This will fail because the daemon is already working and is using the port 50.000.
`monetdbd start /home/fffovde/DBfarm1`

```
fffovde@FffOvdeKomp:~$ monetdbd start /home/fffovde/DBfarm1
cannot remove socket files: Operation not permitted
```

The explanation is that when we open monetdbd with systemctl, two things will happen:

1. Systemctl will start the default dbfarm "`/var/monetdb5/dbfarm`".
2. "`/var/monetdb5/dbfarm`" will will occupy the port 50.000. The dbfarm "`/home/fffovde/Dbfarm1`" is also trying to use the port 50.000. That is why we are getting an error "`cannot remove socket files`".

We will now just stop the monetdbd daemon using systemctl to release the port 50.000.
`sudo systemctl stop monetdbd`
Only after that we will be able to start "`DBfarm1`".
`monetdbd start /home/fffovde/DBfarm1`
`pgrep monetdbd`

```
fffovde@FffOvdeKomp:~$ sudo systemctl stop monetdbd
fffovde@FffOvdeKomp:~$ monetdbd start /home/fffovde/DBfarm1
fffovde@FffOvdeKomp:~$ pgrep monetdbd
2028
```

Mserver5

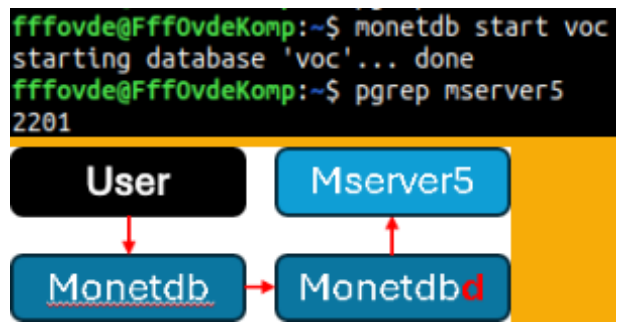
When our database is opened, then the process Mserver5 is working. Currently our database is not opened. We can check that with: `pgrep mserver5`.

```
fffovde@FffOvdeKomp:~$ pgrep mserver5
fffovde@FffOvdeKomp:~$
```

There are two ways how to start database. One is to use monetdb console program. Just like systemctl console program is used to control system daemon systemd, in the same way monetdb console program is used to control MonetDB daemon monetdbd. Monetdb will use monetdbd power to start a database.

```
monetdb start voc
pgrep mserver5
```

```
fffovde@FffOvdeKomp:~$ monetdb start voc
starting database 'voc'... done
fffovde@FffOvdeKomp:~$ pgrep mserver5
2201
```



The other way is to just call database from mclient. I will first stop the database with "`monetdb stop voc`", and mserver5 will close.
`pgrep mserver5` **#we can see that mserver5 is stopped**
I'm closing the database just to show you that we can also open the database with the mclient.
`mclient -u monetdb -d voc`

```
fffovde@FffOvdeKomp:~$ monetdb stop voc
stopping database 'voc'... done
fffovde@FffOvdeKomp:~$ pgrep mserver5
fffovde@FffOvdeKomp:~$ mclient -u monetdb -d voc
password:
Welcome to mclient, the MonetDB/SQL interactive terminal (Aug2024-SP2)
Database: MonetDB v11.51.7 (Aug2024-SP2), 'mapi:monetdb://FffOvdeKomp:50000/voc'
```

We can exit mclient applicaton with "`quit`". We will only close client application, but the database will remain open. We can check that with pgrep command and we will see that our database is still opened:

`pgrep mserver5`

```
Type \q to quit, \? for a list of avai
auto commit mode: on
sql>quit
fffovde@FffOvdeKomp:~$ pgrep mserver5
2239
```

When we login to the voc database with mclient, at that moment the database will be opened if it is not already open. When our database is opened, process Mserver5 is also working. This is because Mserver5 process is OUR DATABASE. This process will perform all processing on request of clients for a database voc. Mclient console application is used to send our queries to this Mserver5 process.

Adding User to "monetdb" Group

If we want to use "`systemd`", it is wise to add your linux user to the "`monetdb`" user group, before creating a database.

"`Monetdb`" user group was created when we installed MonetDB.

```
getent group monetdb              # we have group without users
sudo adduser fffovde monetdb      # we add user "fffovde" to "monetdb" group
getent group monetdb              # our user is now in the group
```

After that, you should log out/in, so that change becomes active.

```
fffovde@FffOvdeKomp:~$ getent group monetdb
monetdb:x:135:
fffovde@FffOvdeKomp:~$ sudo adduser fffovde monetdb
[sudo] password for fffovde:
Adding user `fffovde' to group `monetdb' ...
Adding user fffovde to group monetdb
Done.
fffovde@FffOvdeKomp:~$ getent group monetdb
monetdb:x:135:fffovde
```

I must say that this is not always requirement. I didn't have to do this on "`Zorin`" linux distro. But on the "`Linux Lite`" distribution, and while using "`systemd`", it was a necessity.

I could start "`systemd`" with "`systemctl`" **(1)**, but I was not able to create a database **(2)**. Solution was to add my user to "`monetdb`" group **(3)**. After that I forgot to log out/in, so creation of the database was still impossible **(4)**.

Instead of logging in/out, we can just run the command "`newgrp montedb`" **(5)**, and it is the same as if were logged out. Now creation of the database succeeded without problems **(6)**. So, it is wise to always add your user to "`monetdb`" group.

```
liteclean ~ sudo systemctl start monetdbd 1
liteclean ~ monetdb create newDB
monetdb: cannot connect: no permission to access control socket 2
liteclean ~ 1 sudo adduser liteclean monetdb 3
info: Adding user `liteclean' to group `monetdb' ...
liteclean ~ monetdb create newDB
monetdb: cannot connect: no permission to access control socket 4
liteclean ~ 1 newgrp monetdb
Welcome to Linux Lite 7.6 liteclean 5

Sunday 26 October 2025, 13:31:49
Memory Usage: 1020/12738MB (8.01%)
Disk Usage: 14/98GB (15%)
Support - https://www.linuxliteos.com/forums/ (Right click, Open Link)

liteclean ~ monetdb create newDB 6
created database in maintenance mode: newDB
liteclean ~
```

Starting the Monetdbd Daemon When the Computer Starts up

We can not start "`/home/fffovde/DBfarm1`" dbfarm automatically when the computer boots up. We can only do that with the default dbfarm "`/var/monetdb5/dbfarm`". First we will stop "`/home/fffovde/DBfarm1`" to release the port 50.000:

`monetdbd stop /home/fffovde/DBfarm1`

```
fffovde@FffOvdeKomp:~$ monetdbd stop /home/fffovde/DBfarm1
fffovde@FffOvdeKomp:~$
```

Then we will start "`/var/monetdb5/dbfarm`". We'll use systemctl, because we want the start of our daemon to be controlled by systemd.
`systemctl start monetdbd`

```
fffovde@FffOvdeKomp:~$ systemctl start monetdbd
fffovde@FffOvdeKomp:~$
```

Next, we will create a new database, and we will open mclient application to test it:

```
monetdb create newDB
monetdb release newDB #make database accessible
mclient -u monetdb -d newDB
```

(**password is monetdb for the administrator monetdb**)

```
fffovde@FffOvdeKomp:~$ monetdb create newDB
created database in maintenance mode: newDB
fffovde@FffOvdeKomp:~$ monetdb release newDB
taken database out of maintenance mode: newDB
fffovde@FffOvdeKomp:~$ mclient -u monetdb -d newDB
password:
Welcome to mclient, the MonetDB/SQL interactive terminal (Aug2024-SP2)
Database: MonetDB v11.51.7 (Aug2024-SP2), 'mapi:monetdb://FffOvdeKomp:50000/newDB'
FOLLOW US on https://github.com/MonetDB/MonetDB
Type \q to quit, \? for a list of available commands
```

Now that we have confirmed that our database is working, we will make our "`/var/monetdb5/dbfarm`" to a full fledged systemctl controled daemon. This will make our daemon to open automatically after each system boot.

`systemctl enable monetdbd`

```
fffovde@FffOvdeKomp:~$ systemctl enable monetdbd
Created symlink /etc/systemd/system/multi-user.target.wants/monetdbd.service → /lib/systemd/system/monetdbd.service.
```

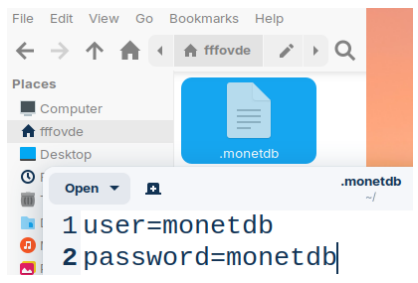
Now we can exit everything and we can reboot our computer.

After restarting, our daemon will open automagically:

```
pgrep monetdbd
mclient -u monetdb -d newDB
```

```
fffovde@FffOvdeKomp:~$ pgrep monetdbd
696
fffovde@FffOvdeKomp:~$ mclient -u monetdb -d newDB
password:
Welcome to mclient, the MonetDB/SQL interactive terminal (Aug2024-SP2)
Database: MonetDB v11.51.7 (Aug2024-SP2), 'mapi:monetdb://FffOvdeKomp:50000/newDB'
FOLLOW US on https://github.com/MonetDB/MonetDB
```

Avoid Entering Credentials Every Time



Inside of our home folder "`/home/fffovde`", we can create textual file "`.monetdb`". Inside of this file we can type our password and username:
`user=monetdb`
`password=monetdb`
After that we don' have to type our credentials any more. We just have to type this, and we are in.
`mclient -d newDB`

```
fffovde@FffOvdeKomp:~$ mclient -d newDB
Welcome to mclient, the MonetDB/SQL interactive term
Database: MonetDB v11.51.7 (Aug2024-SP2), 'mapi:mone
B'
FOLLOW US on https://github.com/MonetDB/MonetDB
```

Running Two DBfarms at the Same Time

```
fffovde@FffOvdeKomp:~$ monetdbd start /home/fffovde/DBfarm1
cannot remove socket files: Operation not permitted
```

Dbfarm "`/var/monetdb5/dbfarm`" is blocking the port 50.000. While this dbfarm is opened, we will not be able to use dbfarm "`/home/fffovde/Dbfarm1`" because that other dbfarm is also using the port 50.000.

```
fffovde@FffOvdeKomp:~$ monetdbd set port=50001 /home/fffovde/DBfarm1
fffovde@FffOvdeKomp:~$
```

First, we will change the port of the "`DBfarm1`". This is how we change properties of a dbfarm.
`monetdbd set port=50001 /home/fffovde/DBfarm1`

```
fffovde@FffOvdeKomp:~$ monetdbd start /home/fffovde/DBfarm1
fffovde@FffOvdeKomp:~$ mclient -p50001 -u monetdb -d voc
password:
Welcome to mclient, the MonetDB/SQL interactive terminal (Mar2025)
Database: MonetDB v11.53.3 (Mar2025)  'mapi:monetdb://FffOvdeKomp:50001/voc'
```

Now, we are able to start manually our "`DBfarm1`":
`monetdbd start /home/fffovde/DBfarm1`
For mclient application we have to provide our **new port number**:
mclient **-p50001** -u monetdb -d voc

Cleaning up

I will quit my session with "`quit`".
I will close and disable "`/var/monetdb5/dbfarm`" dbfarm. Disabling means that the dbfarm will no longer run automatically after the boot.
`systemctl stop monetdbd`
`systemctl disable monetdbd`
I can now delete newDB database. I delete it just like any other folder:
`sudo rm -rf /var/monetdb5/dbfarm/newDB`
I will change port number of "`/home/fffovde/DBfarm1`" back to 50.000.
`monetdbd set port=50000 /home/fffovde/DBfarm1`
Then, I will close "`/home/fffovde/DBfarm1`", too. Both farms are closed now.
`monetdbd stop /home/fffovde/DBfarm1`

```
sql>quit
fffovde@FffOvdeKomp:~$ systemctl stop monetdbd
fffovde@FffOvdeKomp:~$ systemctl disable monetdbd
Removed /etc/systemd/system/multi-user.target.wants/monetdbd.service.
fffovde@FffOvdeKomp:~$ monetdbd set port=50000 /home/fffovde/DBfarm1
fffovde@FffOvdeKomp:~$ monetdbd stop /home/fffovde/DBfarm1
```

After this I will continue to use "`/home/fffovde/DBfarm1`" and "`voc`" database. I will always use commands:
`monetdbd start /home/fffovde/DBfarm1`
`mclient -u monetdb -d voc`

Summary

We can open a dbfarm manually or automatically. To open it manually, we use "`monetdbd start`". For automatic startup we use "`systemctl`". We can only automatically open dbfarm on location "`/var/monetdb5/dbfarm`". If we use two dbfarms at the same time, they must have different port numbers.

In MonetDB, the databases are quite independent. In Microsoft SQL server things are different. SQL server can have many databases that are sharing many resources. They are sharing logins, tempdb, resource pools, memory settings, collation. Backups, replication, and monitoring tools (like SQL Agent jobs) are often configured at the server level.

Because of that it is correct to say that Mserver5 process is not just a database, it is a WHOLE SERVER. Monetdbd is not really a server. That daemon is only a managing tool for Mserver5 processes. Almost the only thing databases within the same dbfarm folder share is their port number. So, "`voc`" and "`newDB`" are monetdb servers. "`/home/fffovde/DBfarm1`" and "`/var/monetdb5/dbfarm`" are just folders for those servers.

Monetdb is console application used by a user to interact with montdb**d** daemon. Monetdb**d** daemon is managing dbfarm and its databases. Mserver5 is a database/server process. Mclient is a console application used to send SQL statements to Mserver5.

0030 Systemd Unit File, MonetDB Sample Database

Systemd Unit File

Systemd Unit File is a file with settings that Systemd will use when starting and controlling some daemon. From version 11.53.13, MonetDB is using a new systemd file that allows us to change the default directory (DBfarm) for databases that are controlled by Systemd.

Old Systemd Unit File

```
fffovde@FffOvdeKomp:~$ systemctl status monetdbd
● monetdbd.service - MonetDB database server daemon
     Loaded: loaded (/lib/systemd/system/monetdbd.service; d
     Active: inactive (dead)
```

`systemctl status monetdbd`
This command will show us where is systemd unit file for Monetdbd daemon.
`/lib/systemd/system/monetdbd.service`

`cat /lib/systemd/system/monetdbd.service`

The old system unit file was hard coded to always use the directory "`/var/monetdbd5/dbfarm`". Only databases located within this directory could be controlled by systemd. Only such databases could be started automatically after the computer boots.

```
1 [Unit]
2 Description=MonetDB database server daemon
3 Documentation=man:monetdbd https://www.monetdb.org/-
  documentation/admin-guide/manpages/monetdbd/
4 After=network.target
5
6 [Service]
7 Type=forking
8 User=monetdb
9 Group=monetdb
0 ExecStart=/usr/bin/monetdbd-11.53.3 start /var/monetdb5/dbfarm
1 ExecStop=/usr/bin/monetdbd-11.53.3 stop /var/monetdb5/dbfarm
2 Restart=on-failure
3 PIDFile=/run/monetdb/merovingian.pid
4 PrivateDevices=no
5 ProtectSystem=full
6 ProtectHome=read-only
7
8 [Install]
9 WantedBy=multi-user.target
```

```
[Unit]
Description=MonetDB database server daemon
Documentation=man:monetdbd https://www.monetdb.org/documentation/admin-
guide/manpages/monetdbd/
After=network.target

[Service]
Type=forking
User=monetdb
Group=monetdb
ExecStart=/usr/bin/monetdbd-11.53.3 start /var/monetdb5/dbfarm
ExecStop=/usr/bin/monetdbd-11.53.3 stop /var/monetdb5/dbfarm
Restart=on-failure
PIDFile=/run/monetdb/merovingian.pid
PrivateDevices=no
ProtectSystem=full
ProtectHome=read-only

[Install]
WantedBy=multi-user.target
```

New Systemd Unit File

From version 11.53.13, we have a new system file that allows us to change DBfarm folder, controlled by Systemd.

```
fff@fffkomp:~$ systemctl status monetdbd
○ monetdbd.service - MonetDB database server daemon
     Loaded: loaded (/usr/lib/systemd/system/monetdbd.service;
     Active: inactive (dead)
```

`systemctl status monetdbd`
This command will show us where is systemd unit file for Monetdb**d** daemon.
`/usr/lib/systemd/system/monetdbd.service`

`cat /usr/lib/systemd/system/monetdbd.service`

This is the new look of the Systemd unit file. We can notice that now we have environ variable DBFARM that is directed to directory "`DBFARM=/var/monetdb5/dbfarm`" by default. This file is now more complex because it allows us to take some other directory for the DBfarm.

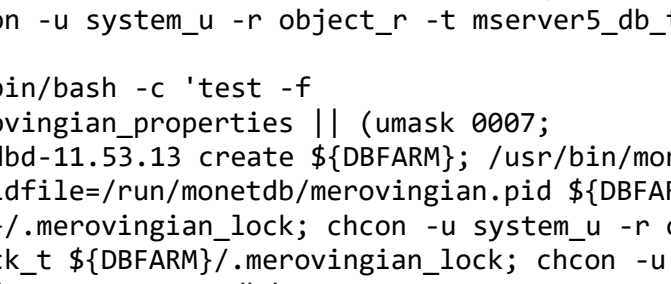
```
usr lib > systemd > system > monetdbd.service
[Unit]
Description=MonetDB database server daemon
Documentation=man:monetdbd https://www.monetdb.org/documentation/admin-guide/
manpages/monetdbd/
After=network.target

[Service]
Type=forking
User=monetdb
Group=monetdb
UMask=0007
# DBFARM is configurable by creating an override file in
# /etc/systemd/system/monetdbd.service.d (to be created) with content
# [Service]
# Environment=DBFARM=/some/other/directory
Environment=DBFARM=/var/monetdb5/dbfarm
# create the dbfarm directory if it doesn't exist
ExecStartPre=/bin/bash -c 'test -d ${DBFARM} || (mkdir -m 770 ${DBFARM}; chcon
-u system_u -r object_r -t mserver5_db_t ${DBFARM})'
# if no properties file, create one with correct pidfile, and set contexts
ExecStartPre=/bin/bash -c 'test -f ${DBFARM}/.merovingian_properties || (umask
0007; /usr/bin/monetdbd-11.53.13 create ${DBFARM}; /usr/bin/monetdbd-11.53.13
set pidfile=/run/monetdb/merovingian.pid ${DBFARM}; touch $
{DBFARM}/.merovingian_lock; chcon -u system_u -r object_r -t monetdbd_lock_t $
{DBFARM}/.merovingian_lock; chcon -u system_u -r object_r -t monetdbd_etc_t $
{DBFARM}/.merovingian_properties)'
# make sure pidfile property is set correctly since systemd depends on it
ExecStartPre=/usr/bin/grep -q pidfile=/run/monetdb/merovingian.pid $
{DBFARM}/.merovingian_properties
# start and stop the server
ExecStart=/usr/bin/monetdbd-11.53.13 start ${DBFARM}
ExecStop=/usr/bin/monetdbd-11.53.13 stop ${DBFARM}
Restart=on-failure
PIDFile=/run/monetdb/merovingian.pid
PrivateDevices=no
ProtectSystem=full
ProtectHome=read-only

[Install]
WantedBy=multi-user.target
```

```
[Unit]
Description=MonetDB database server daemon
Documentation=man:monetdbd
https://www.monetdb.org/documentation/admin-
guide/manpages/monetdbd/
After=network.target

[Service]
Type=forking
User=monetdb
Group=monetdb
UMask=0007
Environment=DBFARM=/var/monetdb5/dbfarm
ExecStartPre=/bin/bash -c 'test -d ${DBFARM} || (mkdir -m 770
${DBFARM}; chcon -u system_u -r object_r -t mserver5_db_t
${DBFARM})'
ExecStartPre=/bin/bash -c 'test -f
${DBFARM}/.merovingian_properties || (umask 0007;
/usr/bin/monetdbd-11.53.13 create ${DBFARM}; /usr/bin/monetdbd-
11.53.13 set pidfile=/run/monetdb/merovingian.pid ${DBFARM};
touch ${DBFARM}/.merovingian_lock; chcon -u system_u -r object_r
-t monetdbd_lock_t ${DBFARM}/.merovingian_lock; chcon -u
system_u -r object_r -t monetdbd_etc_t
${DBFARM}/.merovingian_properties)'
ExecStartPre=/usr/bin/grep -q
pidfile=/run/monetdb/merovingian.pid
${DBFARM}/.merovingian_properties
ExecStart=/usr/bin/monetdbd-11.53.13 start ${DBFARM}
ExecStop=/usr/bin/monetdbd-11.53.13 stop ${DBFARM}
Restart=on-failure
PIDFile=/run/monetdb/merovingian.pid
PrivateDevices=no
ProtectSystem=full
ProtectHome=read-only

[Install]
WantedBy=multi-user.target
```

Changing The Default Directory

We will not change the original Systemd unit file. Instead of that we will create a "`drop-in`". That means that we will create another file that will be "amendment" to the original file. "`Drop-in`" file augments or overrides parts of a unit file without touching the original unit file.

We create this "`drop-in`" file by running the command:
`sudo systemctl edit monetdbd`



An instance of Nano text editor will open. Inside of it we will se the whole original Systemd unit file, but all the lines will be commented out (every line will start with #).

Inside of this file we have to add these lines.

```
[Service]
Environment=DBFARM=/var/monetdb5/dbfarm2
```

This location will override the original "`/var/monetdb5/dbfarm`" location.

I am using linux distribution "KDE Neon". This distribution doesn't use "SELinux". "SELinux" is security feature of the "Red Hat" distribution. The original Systemd unit file provided by the MonetDB developer team is using the command "**chcon**" that is only useful for the distributions that are using SELinux. I will exclude this code from my Systemd unit file.

```
ExecStartPre=/bin/bash -c 'test -d ${DBFARM} || (mkdir -m 770 ${DBFARM}; chcon -u system_u -r object_r -t mserver5_db_t ${DBFARM} )'

ExecStartPre=/bin/bash -c 'test -f ${DBFARM}/.merovingian_properties || (umask 0007; /usr/bin/monetdbd-11.53.13 create ${DBFARM};
/usr/bin/monetdbd-11.53.13 set pidfile=/run/monetdb/merovingian.pid ${DBFARM}; touch ${DBFARM}/.merovingian_lock; chcon -u system_u -
r object_r -t monetdbd_lock_t ${DBFARM}/.merovingian_lock; chcon -u system_u -r object_r -t monetdbd_etc_t
${DBFARM}/.merovingian_properties)'
```

Extraordinary, because I am using distro without SELinux, I will also add these corrected lines to my "`drop-in`" Systemd unit file.

```
# First, we will delete all of the "ExecStartPre=".
ExecStartPre=
# This line will add the first part of the "ExecStartPre=". The line "ExecStartPre" is split into three parts,
# just to make it more readable. These three lines can be considered as one script.
ExecStartPre=/bin/bash -c 'test -d ${DBFARM} || (mkdir -m 770 ${DBFARM})'
# This line will add the second part of the "ExecStartPre=".
ExecStartPre=/bin/bash -c 'test -f ${DBFARM}/.merovingian_properties || (umask 0007; /usr/bin/monetdbd-11.53.13 create ${DBFARM};
/usr/bin/monetdbd-11.53.13 set pidfile=/run/monetdb/merovingian.pid ${DBFARM}; touch ${DBFARM}/.merovingian_lock)'
# Third line is unchanged.
ExecStartPre=/usr/bin/grep -q pidfile=/run/monetdb/merovingian.pid ${DBFARM}/.merovingian_properties
```

```
### Anything between here and the comment below will become the contents of the drop-in file

[Service]
Environment=DBFARM=/var/monetdb5/dbfarm2

ExecStartPre=
ExecStartPre=/bin/bash -c 'test -d ${DBFARM} || (mkdir -m 770 ${DBFARM})'
ExecStartPre=/bin/bash -c 'test -f ${DBFARM}/.merovingian_properties || (umask 0007; /usr/bin/monetdbd-11.53.13 create ${DBFARM}; /usr/bin/monetdbd-11.53.13 set pidfile=/run/monetdb/merovingian.pid ${DBFARM}; touch ${DBFARM}/.merovingian_lock)'
ExecStartPre=/usr/bin/grep -q pidfile=/run/monetdb/merovingian.pid ${DBFARM}/.merovingian_properties

### Edits below this comment will be discarded
```

This config snippet will be "amendment" to original Systemd unit file. This snippet must be added above the comment "**Edits below this comment will be discarded**".

We will save this change with "Ctrl+O" **ENTER,** and then we will exit with the "Ctrl+X".
After that we must reload all the systemd unit files.

```
sudo systemctl daemon-reload
```

The Changes We Made

```
fff@fffkomp:~$ systemctl show monetdbd | grep ^Environment=
Environment=DBFARM=/var/monetdb5/dbfarm2
```

This command will show us the new value of the DBFARM environ:
`systemctl show monetdbd | grep ^Environment=`

```
fff@fffkomp:/etc/systemd/system/monetdbd.service.d$ ls -alh
total 36K
drwxr-xr-x  2 root root 4.0K Oct 11 15:41 .
drwxr-xr-x 30 root root 4.0K Oct 11 13:44 ..
-rw-r--r--  1 root root  460 Oct 11 15:41 override.conf
-rw-r--r--  1 root root 2.4K Oct 11 15:25 .#override.conf3737b4ec6a86b434
-rw-r--r--  1 root root 1.9K Oct 11 14:00 .#override.conff299325b80c0e4b2
fff@fffkomp:/etc/systemd/system/monetdbd.service.d$ cat override.conf
[Service]
Environment=DBFARM=/var/monetdb5/dbfarm2

ExecStartPre=
ExecStartPre=/bin/bash -c 'test -d ${DBFARM} || (mkdir -m 770 ${DBFARM}; chcon -u s
ExecStartPre=/bin/bash -c 'test -f "${DBFARM}/.merovingian_properties" || (umask 00
M}"; touch "${DBFARM}/.merovingian_lock")'
```

"`drop-in`" file is written inside of the:

`/etc/systemd/system/monetdbd.service.d/override.conf`

On the image we use these commands to check `override.conf`:

```
cd /etc/systemd/system/monetdbd.service.d
ls -alh
cat override.conf
```

`systemctl cat monetdbd`	This command is also useful because it will show us the original file, and it overrides, and their locations, all together.

Testing The Changes

Because we are going to use Systemd, we should add our user to "`monetdb`" group.

`sudo usermod -aG monetdb "$USER"`	After that, we should log out and then log in.

`newgrp monetdb`	Sometimes log out/in will not be enough. In that case, try to open a new session (a tab) in a terminal, or run this command in the existing session. This is happening because OS will try to recycle the old session, so changes are not applied.

We will now create one database and after that we will restart our computer to see if it will be started automatically.

`systemctl start monetdbd`
`monetdb create DBdesktop`
`monetdb release DBdesktop`
`systemctl enable monetdbd`
`reboot`
	-- Start monetdbd daemon controlled by systemd.
	-- Create a database.
	-- Make database available.
	-- Make monetdbd to start automatically after computer reboot.
	-- Restart our computer.

```
fff@fffkomp:~$ mclient -u monetdb -d DBdesktop
password:
Welcome to mclient, the MonetDB/SQL interactive terminal (Mar2025-SP2)
Database: MonetDB v11.53.13 (Mar2025-SP2), 'mapi:monetdb://fffkomp:50000/DBdesktop'
FOLLOW US on https://github.com/MonetDB/MonetDB
Type \q to quit, \? for a list of available commands
auto commit mode: on
sql>
```

After the reboot, I will open the terminal, and I will try to log in:

`mclient -u monetdb -d DBdesktop`

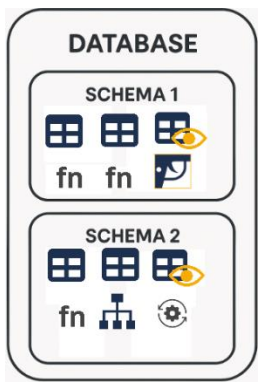
It will work. We don't have to start the daemon manually.

The last step is to check if DBdesktop is really inside of the dbfarm**2** directory.

```
cd /var/monetdb5/dbfarm2
ls -alh
```

```
fff@fffkomp:~$ cd /var/monetdb5/dbfarm2
fff@fffkomp:/var/monetdb5/dbfarm2$ ls -alh
total 48K
drwxrws--- 3 monetdb monetdb 4.0K Oct 27 21:31 .
drwxrws--- 4 monetdb monetdb 4.0K Oct 27 18:01 ..
drwx--S--- 4 monetdb monetdb 4.0K Oct 27 21:55 DBdesktop
-rw-rw---- 1 monetdb monetdb    0 Oct 11 15:42 merovingian.lock
```

What is Schema



Database is made of tables, views, indices and so on. Inside of Database, each of these objects belongs to some schema. Database is organizationally divided into schemas. There is no overlap between schemas. Some special objects are outside of a schema, like roles.

Usage and modification of schema elements is strictly done by the user or the role that owns that schema. During creation of a schema, we should decide who will be the owner, because later it will not be possible to change ownership. If we want several people to maintain one schema then we should set a role as an owner of a schema.

Only '`monetdb`' user and '`sysadmin`' role are allowed to create new schemas.

Schema as a File

We can create a file that will contain all the instructions the server needs to create database objects inside of the schema. This file would tell the database which tables, relations, indexes, views to create. In this way, we can create everything that makes up one schema. That is why we call such a file a "`schema file`". Although we have not learned all the SQL commands needed to create such a file, we can use the "`schema file`" that the MonetDB development team has prepared for us.

First, we will download sample database from this location:

https://dev.monetdb.org/Assets/VOC/voc_dump.zip

This will give us ZIP file. Inside of it, there is SQL script with the schema for our new database.

Creation of a New User

For our schema we will create a new user. First, we will enter *mclient* with 'monetdb' privileges.

```
> monetdbd start /home/fffovde/DBfarm1
> mclient -u monetdb -d voc --password monetdb
```

```
fffovde@FffOvdeKomp:~$ mclient -u monetdb -d voc
password:
```

For creation of a user, we need username (USER), password (WITH PASSWORD), user's full name (NAME), and default schema for that user (SCHEMA). The default schema is schema that MonetDB will use as a current schema when the user log in. For tables in the current schema, the user can type "`SELECT * FROM Table1`", but for tables in NON current schemas, the user must type "`SELECT * FROM schema.Table1`".

"`sys`" schema is a built-in schema in MonetDB. We will use it temporarily so we can create a new user.

```
sql> CREATE USER "voc" WITH PASSWORD 'voc' NAME 'VOC Explorer' SCHEMA "sys";
```

```
sql>CREATE USER "voc" WITH PASSWORD 'voc' NAME 'VOC Explorer' SCHEMA "sys";
operation successful
```

As a '*monetdb*' administrator we can create a new schema. We will say that previously created "*voc*" user is the owner of that schema.

```
sql> CREATE SCHEMA "voc" AUTHORIZATION "voc";
```

```
sql>CREATE SCHEMA "voc" AUTHORIZATION "voc";
operation successful
```

Don't get confused. The name of our database is "`voc`", but the name of the new schema is also "`voc`", and the name of the user is "`voc`".

We will set the new schema as the default schema for our user.

```
sql> ALTER USER "voc" SET SCHEMA "voc";
sql> \q  -- we can exit mclient with "quit" or "\q"
```

```
sql>ALTER USER "voc" SET SCHEMA "voc";
operation successful
```

Since the "`voc`" schema is the default schema for the "`voc`" user, this schema will be active when this user logs in to MonetDB. Everything the user does will be reflected in this schema, unless the user explicitly mentions that they want to work in a different schema.

Populating our Schema with Database Objects

Our "`voc`" schema is currently empty, but we have definitions of all the tables, view, indices … inside of our downloaded SQL script. We will use that script to populate our schema. We type:

```
> mclient -u voc -d voc ~/Desktop/voc_dump.sql
```

This code will log "`voc`" user into "`voc`" server. It will also execute all the SQL commands from the "`voc_dump.sql`" file. When the user log in, and his default schema is "`voc`", that means that he will log in into "`voc`" schema. All the SQL commands will be executed inside of this schema.

```
fffovde@FffOvdeKomp:~$ mclient -u voc -d voc ~/Desktop/voc_dump.sql
1 affected row
1 affected row
1 affected row
1 affected row
```

We will log in again:
`mclient -u voc -d voc` **#password is voc**

We can use mclient command "`\d`" to list all the tables and views inside of our database.

```
sql> \d
TABLE voc.craftsmen
TABLE voc.impotenten
TABLE voc.invoices
TABLE voc.passengers
TABLE voc.seafarers
TABLE voc.soldiers
TABLE voc.total
TABLE voc.voyages
```

This is what we would get:

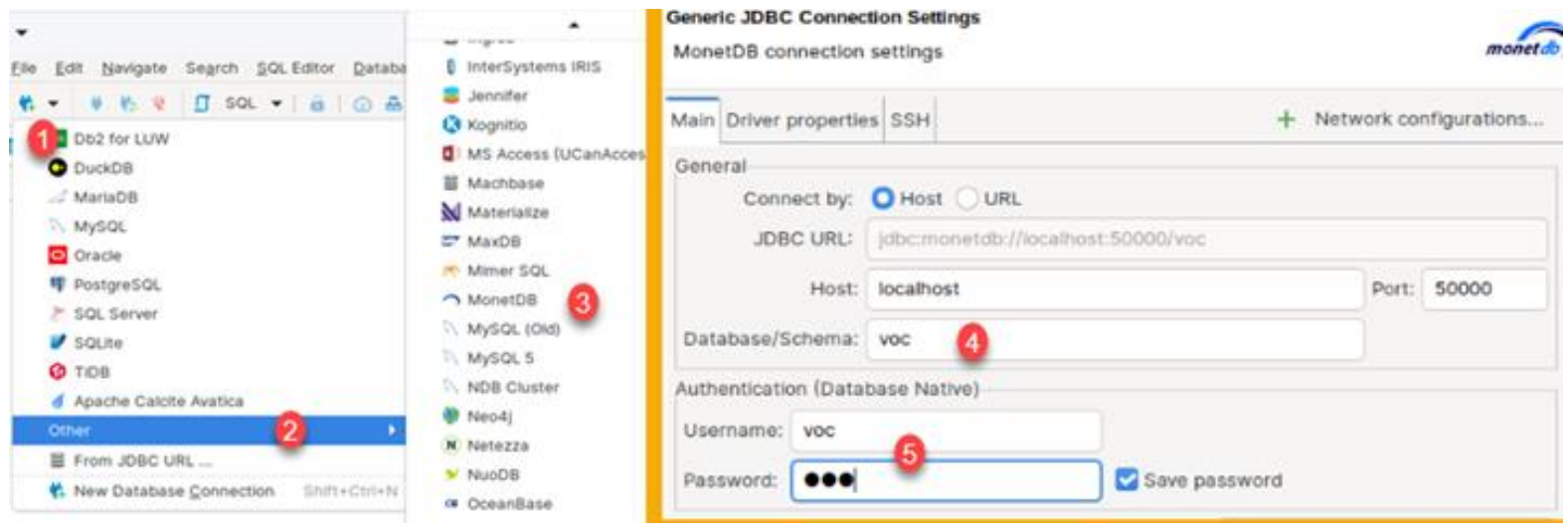
```
sql>\d
TABLE  voc.craftsmen
TABLE  voc.impotenten
TABLE  voc.invoices
TABLE  voc.passengers
TABLE  voc.seafarers
TABLE  voc.soldiers
TABLE  voc.total
TABLE  voc.voyages
```

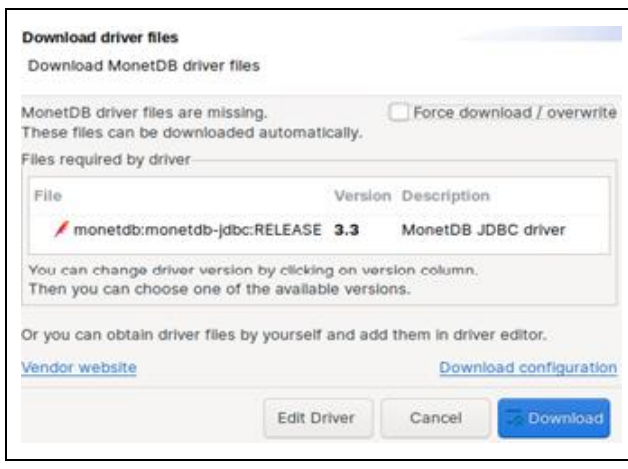
DBeaver Database Manager Program

We will install DBeaver database manager program to peruse our database.
`> sudo snap install dbeaver-ce`

```
fffovde@FffOvdeKomp:~$ sudo snap install dbeaver-ce
dbeaver-ce 23.2.0.202309040911 from DBeaver (dbeaver-corp) installed
```

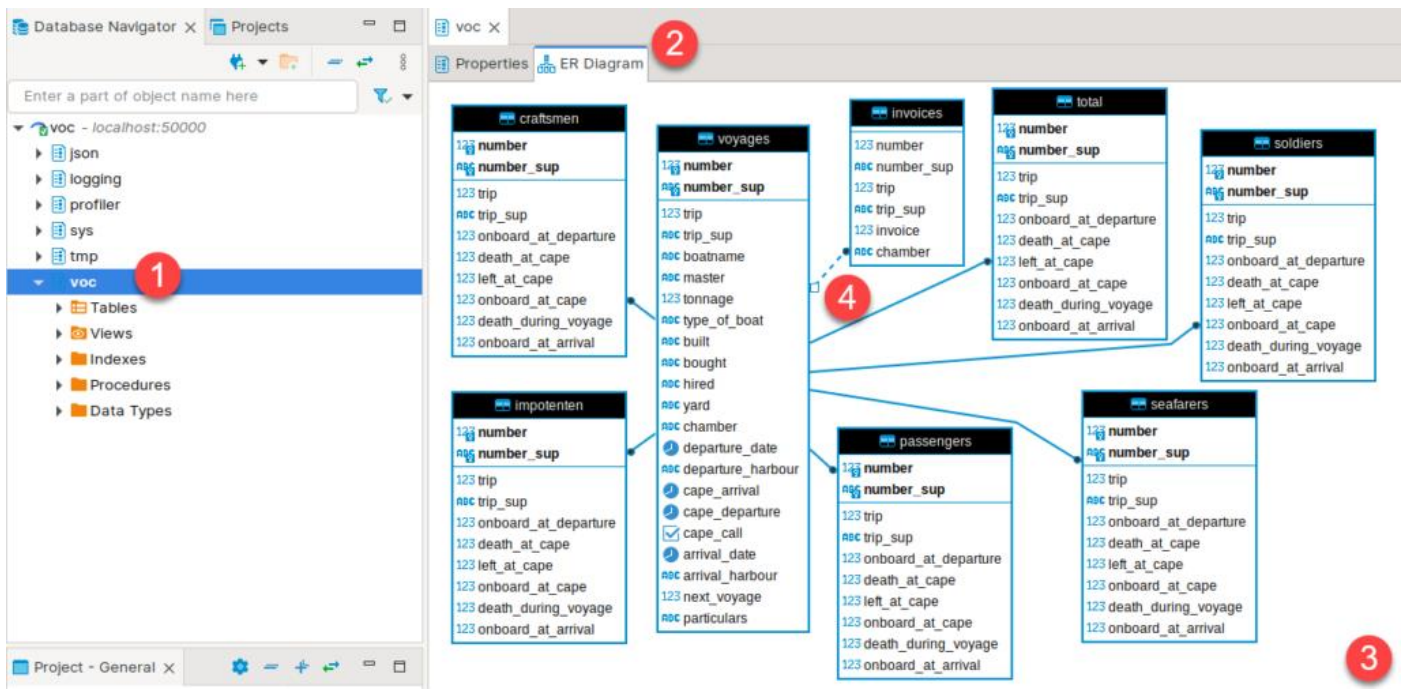
This program is GUI program, so we will open it in our desktop environment. In the program we will click on the triangle **(1)**, and then we will select "`Other`" **(2)** to open other servers. There we will find MonetDB. We will click on it **(3)**, and a new dialog will open. In this dialog, host and port are already entered. We just need to enter Database/Schema **(4)**, Username and Password **(5)** (which is "**voc**").





DBeaver will not have driver for MonetDB database installed, so it will offer you to download it. Just accept that offer.

At the end, objects of our database will appear inside of pane on the left side of a program. There, we should double click on schema name **(1)**. After that we can select tab "`ER Diagram`" **(2)**. There, after some rearrangement, we will see ER diagram of our database **(3)**. As we can see, tables are organized in star schema with "`voyages`" table as the only fact table. All tables are connected with foreign key constraints, where foreign key is also primary key inside of dimension tables. The only exception is Invoice table where foreign key columns are not primary key columns, and that is why that relationship is shown with dashed line **(4)**.



"voc_dump.zip" file is available for download from the repository listed at the beginning of the book.

DBeaver is an excellent program. If you want to learn more about it, you can try this linkedin tutorial.

DBeaver Tutorial

0040 Connect to MonetDB from Python

Installation of Pymonetdb Module on Ubuntu 24.04

Python comes preinstalled on most of the Linux distributions. We can check version of our python with a command:

```
python3 --version
```

```
fff@fffkomp:~$ python3 --version
Python 3.12.3
```

For connection to MonetDB, python is using Pymonetdb module. Ubuntu's default Python (in `/usr/bin/python3`) is **owned and maintained by APT**. Ubuntu doesn't want users to use python installers, like "`pip`", because doing so could corrupt packages that Ubuntu's package manager relies on. This is done to avoid conflicts to packages installed by OS package manager.

For installation of the python packages that are not available in the Ubuntu's repository, we should use virtual environments. We will see bellow that Pymonetdb is provided by Ubuntu, so we don't have to use virtual environment.	`fff@fffkomp:~$ pip install pymonetdb`
	`error: externally-managed-environment`
	
	`× This environment is externally managed`
	`╰─> To install Python packages system-wide, try apt install`
	`python3-xyz, where xyz is the package you are trying to`
	`install.`
	
	`If you wish to install a non-Debian-packaged Python package,`
	`create a virtual environment using python3 -m venv path/to/venv.`
	`Then use path/to/venv/bin/python and path/to/venv/bin/pip. Make`
	`sure you have python3-full installed.`

For that reason, we will not install Pymonetdb using "`pip`" as we would normally do, but we will use apt package manager. In this way we can only install packages that are available in the Ubuntu's repository. First, I will update the index of the available Ubuntu's repository packages, and I will search for Pymonetdb module.

```
sudo apt update
apt search pymonetdb
```

Modern Ubuntu only has Pymonetdb module that is meant to be used with the Python3.

```
fff@fffkomp:~$ apt search pymonetdb
Sorting... Done
Full Text Search... Done
python3-pymonetdb/noble 1.8.5-1 all
  Pure Python database driver for MonetDB/SQL
```

We will install this module from the Ubuntu repository:
 	
`sudo apt install python3-pymonetdb`
	`fff@fffkomp:~$ sudo apt install python3-pymonetdb`
	`[sudo] password for fff:`
	`Reading package lists... Done`
	`Building dependency tree... Done`
	`Reading state information... Done`

Pymonetdb module is now installed.

Installation of Pymonetdb Module in a Legacy Way

On the older versions of Ubuntu (older than 23.04) we could install Pymonetdb with the "`pip`" installer. We already have python installed:

```
python3 --version
```

```
fffovde@FffOvdeKomp:~$ python3 --version
Python 3.8.10
```

Pip is a console program used for installing python modules. So, first we need to install pip, if we don't have it.

```
sudo apt install python3-pip
```

```
fffovde@FffOvdeKomp:~$ sudo apt install python3-pip
Reading package lists... Done
Building dependency tree
```

After installing pip, we will use it to install pymonetdb module:

```
pip install pymonetdb
```

```
fffovde@FffOvdeKomp:~$ pip install pymonetdb
Collecting pymonetdb
  Downloading pymonetdb-1.7.1-py2.py3-none-any.whl (93 kB)
     |████████████████████████████████| 93 kB 1.2 MB/s
Installing collected packages: pymonetdb
Successfully installed pymonetdb-1.7.1
```

Pymonetdb module is installed.

Starting a Database

Before we try to connect to MonetDB, we must start our database:

```
monetdbd start /home/fffovde/DBfarm1
monetdb start voc
```

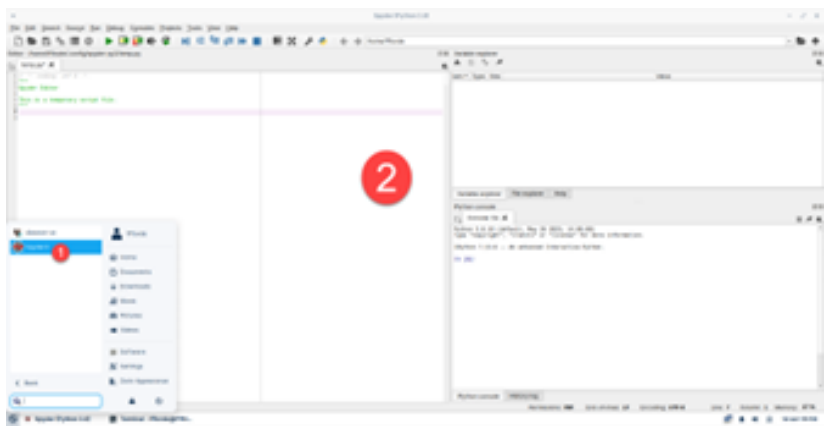
Installing of Spyder IDE on Ubuntu

Now we can try to connect to MonetDB from python. For that, I will type python commands into Spyder IDE. We have to first install Spyder IDE on Ubuntu.

```
sudo apt install spyder
```

```
p:~$ sudo apt install spyder
```

We can then start Spyder from the graphical interface **(1)**. This is how spyder looks like **(2)**:



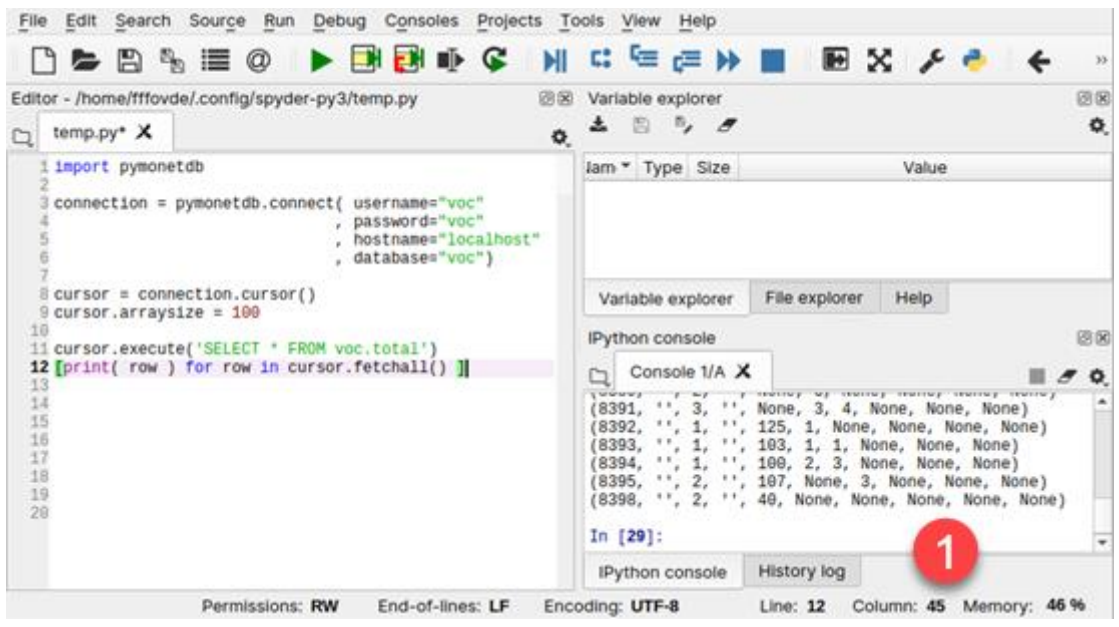
Spyder is a free and open source scientific environment for Python.

Python Script to Connect to MonetDB

Inside of Spyder IDE, I will add this script. This script will first create connection object. Using that connection object, we will create cursor object. Then we can use cursor object to execute our query.

```
import pymonetdb
connection = pymonetdb.connect(username="voc", password="voc", hostname="localhost", database="voc")
cursor = connection.cursor()
cursor.execute('SELECT * FROM voc.total')
[print( row ) for row in cursor.fetchall() ]
```

Result of our query will be list of tuples (like [(a,b,c),(1,2,3)]), where each tuple is one row of a table. We will use list comprehension to print those rows one by one. At the end, Spyder console **(1)** will show us result.



Pymonetdb Help

If you want to learn more about pymonetdb, you can go to official documentation on this address:

https://pymonetdb.readthedocs.io/en/latest/index.html