UiT The Arctic University of Norway

Faculty of Science and Technology
Department of Computer Science

A before B

Investigations into how best to perform Temporal Health Queries

Anders Søreide

Master's thesis in Computer Science INF-3990 May 2024

Supervisors

Main supervisor:	Edvard Pedersen	UiT The Arctic University of Norway, Faculty of Science and Technology, Department of Computer Science
Co-supervisor:	Hans Olav Melberg	UiT The Arctic University of Norway, Faculty of Health Sciences, Department of Community Medicine

Abstract

Querying and exploring health data can lead to the discovery of new relations between conditions, medications, hospital events, etc. For this purpose, temporal health queries are useful since the order in which events happen is important.

Many of the querying tools available do not address the unique needs of temporal health queries, making these queries difficult and time-consuming to perform. One tool made for this purpose, Snotra, enables temporal health queries with a syntax that is human-readable and easy to understand and write. Problems in the technical implementation and underlying architecture of Snotra currently prevent it from being used to query large datasets from health registers.

By implementing a subset of Snotra operations we can compare to design a new underlying engine for Snotra to handle larger datasets. This thesis explores possible avenues to fix the underlying architecture of Snotra, comparing a selection of approaches including SQL, Dataframes, and custom low-level querying functions. The most promising approach is further developed into a prototype supporting a small subset of Snotra operations.

This work shows how Polars extended with custom Rust query functions is a viable path for implementing performant and scalable temporal health queries.

Contents

Abstract **iii**

List of Figures **vii**

List of Tables **ix**

1 Introduction **1**
1.1 Research Questions 2
1.2 Contribution 3
1.3 Thesis statement 3

2 Background **5**
2.1 Health registers 5
2.2 Querying health data 5
2.3 Snotra 6
2.4 Same Language, New Engine 6

3 Querying approaches **9**
3.1 SQL 9
3.2 Dataframes 10
3.3 Graph databases 10
3.4 No-SQL databases 10
3.5 Building something new from the ground up 11

4 Methodology **13**
4.1 SQL or Rust 14
4.1.1 Form of the experiments 14
4.1.2 Performance requirements 15
4.1.3 Data pre-processing 15
4.1.4 Input Data 16
4.1.5 Datasets for checking query correctness 16
4.2 Hardware 17
4.2.1 A before B 17
4.2.2 Accumulated within 17

4.3 What subset of features represents Snotra 18
4.4 How does the new engine perform under expected workload? 18

5 Rust Vs SQL **19**
5.1 A before B 19
5.1.1 Rust 19
5.1.2 SQL 20
5.2 Accumulated within 20
5.2.1 Rust 20
5.2.2 SQL 21
5.3 Experiment Results 22

6 Design **29**
6.1 Queries 29
6.2 Combining statements 30
6.3 Memory requirements 30

7 Implementation **33**
7.1 Pandas and Python VS Polars and Rust 33
7.2 Extending Polars compared to Standalone Rust 34

8 Results **37**

9 Discussions **41**
9.1 Limitations with recusive functions 41
9.2 In-memory execution 41
9.2.1 Memory usage 42
9.3 As a standalone Rust program 43

10 Conclusion **45**

11 Future Work **47**

List of Figures

5.1 Using 'Accumulated Within' to find a patient with over **80** accumulated dosage within **20** days 21
5.2 Scaling in Rust 25
5.3 'A Before B' scaling: SQL vs Rust 26
5.4 'A Before B' Rust vs Polars 27

7.1 Polars plugin 34
7.2 Standalone Rust 35

8.1 'A Before B' in Polars, Rust, and as a Rust extention 38
8.2 Standalone Rust vs Polars extention 39

List of Tables

5.1 Scaling of ‘A Before B’ in Rust 22
5.2 ‘Accumulated within’ in Rust 23

9.1 light workload 42
9.2 medium workload 42

1

Introduction

A patient gets admitted to the hospital. The patient is recognized by a doctor, who just a week prior gave the patient a diabetes diagnosis. Said diagnosis should have prevented the hospital admission, and so the doctor concludes that something has gone wrong. To investigate what might have gone wrong, a natural first step might be to look at all other patients who have also been admitted to the hospital following a diabetes diagnosis. In this case, a temporal health query can be used to query health events and find patients who experience certain events in a certain order, like receiving a diabetes diagnosis before being admitted to the hospital. These are the types of queries Snotra is designed to handle.

Like the example above, there are many cases where we want to find relations between events in health care. Questions a physician or health care professional might want to ask might take the form of:

- 'Find patients with condition A before developing condition B'
- 'Find patients that received more than X units of medication Y in the last Z days'
- 'Find patients that are members of both groups J and K'

By answering these questions one can discover procedures where patients are

not given clear enough instructions, conditions and/or medications that affect some other condition, or specific patient groups that have been neglected.

In the Nordics, health data such as prescriptions and hospital procedures are commonly stored in health registers. To perform these temporal health queries on the various health registers custom code must be written and tested for each query, making exploratory queries time-consuming and expensive. Snotra allows users to express temporal health queries with a syntax very close to the example questions above, reducing the amount of code needed from a small program to a single sentence.

So far Snotra sounds great, however there are currently two limitations holding Snotra back. Snotra currently exists as a prototype, with no guarantees to the user that features work as intended. Another problem with Snotra currently is the underlying implementation that contains a significant amount of overhead for its queries, rendering it unviable for larger datasets. While the first issue can be solved with more development time, the second issue requires a more fundamental change to be addressed. The Snotra prototype has produced both a querying language to express, and several operations that enable the execution of, temporal health queries. Re-implementing these operations with an underlying architecture that can handle larger datasets will take Snotra one step closer to a complete querying tool that can be used in the field by users.

1.1 Research Questions

While Snotra has given us query operations and syntax for expressing temporal health queries, the implementation of Snotra is not equipped to handle large datasets like the ones we might expect to be working with when searching for relations in health data. We outline the following research questions:

- Should the new engine be created around already existing SQL, or should we create a new domain specific tool?
- What subset of Snotra functions can we implement to evaluate the new engine implementation?
- How does this new implementation perform under expected workloads?

1.2 Contribution

This thesis compares a handful of Snotra operations that have been re-implemented in SQL, Polars, and Rust. Through these evaluations we show how the different approaches perform, and how the different approaches scale when used on larger datasets. This work shows how the most promising approach can be used to handle large datasets through a small prototype supporting a subset of the original Snotra features.

1.3 Thesis statement

By evaluating a handful of re-implementations of select Snotra operations we show how the underlying engine of Snotra can be changed to enable fast temporal health queries on large datasets.

2 Background

2.1 Health registers

Health registers are commonly used in the Nordics, containing vast amounts of health data[1]. The Norwegian Prescription Database (NorPD)[2] is one such register, storing information about patients and their prescriptions. Other registers are used to store different health events like heart conditions, surgeries, etc. The large amounts of data stored in these registers can be combined, explored, and queried to find new relations, like linking certain conditions to side effects from medication, or discovering areas where patients are not given clear instructions from healthcare providers.

2.2 Querying health data

We have a wide array of tools available for querying data, from SQL and NoSQL databases to dataframes. Usually a query takes the form of defining some pattern or identifier and searching for data matching this pattern. This approach is great for finding independent pieces of data, but can struggle when the context surrounding the data is important. Individual data points in a health register can have different meanings depending on other data points. A patient with one condition might have an increased risk of another condition. As mentioned in the Snotra wiki[3], it can be a challenge to perform patient oriented queries on event level data.

When writing a query we have to translate the event oriented dataset into patient oriented data, consider the order in which events happen to respect the temporal aspect of some queries, handle potentially large datasets, and ideally return the result within a few minutes. Each new challenge multiply the difficulty of performing the required query. While any given problem can be solved with existing tools, the sum of all problems combined make temporal health queries difficult to perform with the tools currently available.

2.3 Snotra

Snotra was developed as an attempt to make it easier to explore health register data. By creating a tool specialized for this workload, temporal health queries can be expressed easily with a human-readable language. Initially the idea was to benchmark the current implementation of Snotra to tetermine the cause of Snotra's perfromance issues. During early benchmarking it quickly became clear that Snotra had issues running several queries, even breaking when attempting to replicate example queries. This was not entirely unexpected, as Snotra at the time of writing does not have a stable release. A change in approach was needed to continue with the project. While the implementation might have technical limitations, the primary contribution of Snotra is its extensive language for *expressing* queries. The Snotra language as it's defined in the Snotra project[4] enables a large array of health queries using easy to understand scentences. If we want to select all patients with ICD[5] codes *K50, K51* excluding patients with ICD *K52*, we can express this in Snotra with the following query:

```
df.count_persons(codes=(K50 or K51) and not K52, cols='icd*')
```

ICD codes like *K50* are made up of a letter and two numbers, and can be used to consisely describe various health conditions. In this thesis these codes are chosen at random.

2.4 Same Language, New Engine

If we can keep the Snotra language, but change the implementation to something more robust, we would end up with a robust tool that enables expressive health queries that are easy to write and understand. One advantage we have over existing general purpose tools like Pandas or SQL is the ability to make assumptions about the data we are querying. By restricting our usecase to a specifc type of data and use, we can take shortcuts and make assumptions.

Creating the engine in a low level language like C or Rust might allow us to make use of these assumptions to create higly optimized queries. Alternatively we can leverage the optimizations that exist in already established tools like an SQL DBMS or Dataframe tools like Pandas/Polars. SQL might be complicated and difficult to use for temporal health queries, but this only matters if the user writes the SQL queries themselves. If instead SQL is used as an underlying engine to support the Snotra querying language, with Snotra queries being translated into complex SQL queries, we can address the primary dissadvantage to SQL for this workload without having to develop an enterily new tool from the ground up.

3

Querying approaches

There are a vast number of approaches, frameworks, and specific tools that can be used when implementing a query. Comparing every possible option available is outside the scope of this thesis, and so the set of querying tools must be narrowed down to 2 or 3 options that can be compared in greater detail.

3.1 SQL

SQL languages excel at querying tabular data, and so it's natural to ask if an SQL approach could be usefull for implementing Snotra. There are a number of SQL implementations, including temporal variants like QuestDB[6] and temporal extentions to existing SQL variants[7][8]. Many currently available SQL variants are created for the sake of business analytics and customer database uses. Even temporal variants often implement temporal features in the context of analyzing the last fiscal quarter, or finding trends in a 6-month time period. One attempt at creating an SQL variant specifically for health queries, DXtractor[9], does not rely on temporal SQL extentions, instead using standard SQL to support health queries.

By making use of MariaDB's *'GROUP CONCAT'* feature we are already able to implement an SQL variant of a Snotra query, making this our SQL variant of choice. MariaDB also support the majority of features present in MySQL,

another popular SQL variant[10]. It might be possible to make use of MariaDB's many available features to implement more Snotra opperations.

3.2 Dataframes

Dataframes have a row/column structure and support powerful set operations. This has made dataframe-centric tools like Pandas[11] and the more recent Polars[12] industry standard tools for filtering columnar data. Dataframe libraries do not require any connection to a database, since the data is stored in-memory in the dataframe structure. This, combined with both Pandas and Snotra being available as Python libraries has made the dataframe workflow highly accessible. The current implementation of Snotra uses Pandas for its queries. Dataframes enable set-centric query opperations that can be powerful, but can be difficult to use effectively for temporal health queries. Using the opperations found in Polars/Pandas to implement temporal health query opperations might be a viable approach, if it does not require unnecessary opperations to implement the query.

3.3 Graph databases

Given that a key motivation behind the Snotra project is exploring data for correlations, graph databases can look like a natural choice. Since graph databases are specifically created for modeling and querying relations, the logic follows that they should be able to find correlations in health register data. Graph databases excel in querying relations like patients that take the same medications or that have the same pre-existing conditions. These relations can be coded into the graph, resulting in fast retrieval times for the query. This could also be a downside for temporal health queries however, as relations must be expressed when creating the graph. If we are searching for relations that are currently not found, it would be tricky to create a graph containing said relations. Another factor to consider is Neo4j, a large industry standard graph database system, not recommending the use of graph databases to query static tabular data like the data we expect to find in health registers[13].

3.4 No-SQL databases

No-SQL database systems are great for storing and retrieving data of unknown types, however this is not a concern for us, as we are dealing with tabular

datasets. Another benefit of distributed no-sql databases are their ability to use sharding to parallelize query execution. Since we are querying a large group of individual patients, we already have a highly paralellized workload. A no-sql approach like MongoDB would likely be more of a hindrance to us, as our concern is not finding one patient as fast as possible, but quickly traversing the entire dataset.

3.5 Building something new from the ground up

Rather than looking at new ways to use existing tools, there is the option of creating something new from scratch. With the rather obvious downside of the immense programming effort required, a number of new possibilities present themselves when not burdened by design choices found in programs designed for other workloads. The benefits of these new possibilities must outweigh the steep upfront cost. Since the tool is used in a specific field, we can make assumptions about the input data that a more general tool can't. Instead of having to start queries with group by and sort options we can instead give this responsibility to the input data, and create query functions with the assumption that the data is sorted in the expected order. With SQL even if the input data was already sorted we have to perform the grouping to enable some operations.

The goal of making assumptions about the input data is to reduce the number of times we must traverse the input data. Since medical queries likely involve larger datasets, any reduction in data traversal could result in large time savings. The ideal query would involve traversing the input data linearly, going from the first to the last row once giving an $O(n)$ scaling to the query. When comparing different querying approaches the scalability of the approach is far more important than the raw performance. The datasets we are using for testing is likely smaller than many datasets where these temporal health queries are relevant, so a query having lower performance with better scaling will in many cases be preferable to another approach where the test might be completed faster.

As mentioned in the Dataframes section, creating a standalone tool build from custom query functions or creating custom query functions for extending Polars dataframes is essentially the same. The main difference is the dataformat used to load and store data, both being columnar.

4

Methodology

Originaly the plan was to first outline the performance of Snotra as it is currently. This would have formed a baseline performance for various opperations that could be compared to new implementations. When attempting to reproduce example queries from the Snotra wiki[1] the majority of queries tested would return an error, with exception to some functions like *count persons*. Since Snotra does not have a stable release, and provides no guarantees of functionality, new implementations of the different Snotra opperations will only be tested against each other and not their current implementation in Snotra.

SQL is of interest to us since we are working with static, tabular data, where SQL tends to perform well. We investigate how two different Snotra queries perform when implemented in SQL, represented by MariaDB, and when implemented in a low level language, represented by Rust.

Further experiments are performed with the most promising approach, with the goal of implementing a subset of Snotra using that approach.

1. `https://github.com/hmelberg/snotra/wiki/A-query-language-including-temporal-expressions:-X-before-Y`

4.1 SQL or Rust

There have been previous attempts at performing health queries in SQL.

We know Snotra is in need of a new engine, but the form of this engine is not clear. As discussed earlier there are many approaches to querying health queries, and we focus on comparing SQL with a custom Rust query.

While it is possible to extend tools like SQL and Pandas/Polars with custom agregate functions, this is effectively the same as developing a custom function. We therefore only care about comparing a query constructed from already existing features of MariaDB[14], and compare this to a custom query function developed from scrach in Rust.

MariaDB was chosen due to its wide array of querying tools available, including a group concattination feature that is very usefull for implementing one of the queries. Rust represents a modern low-level language that is steadily gaining popularity in low-level workloads. The Cargo package manager is also helpfull for integration of code between different Rust tools and projects.

4.1.1 Form of the experiments

The experiments take the form of selecting specific health queries, and implementing the queries in both SQL and rust. We can perform the same queries on datasets of varying size to test both the raw speed of the two approaches along with how they scale with an increasing dataset size. The queries are individually implemented to get an idea of how a tool implemented with SQL or Rust would perform.

For this experiment the following two queries where chosen:

- 'A before B'
- 'Accumulated Within'

'A before B' can be useful in cases like finding patients that first receive a diagnosis, but then later suffers a condition that should have been prevented with the diagnosis. 'Accumulated Within' looks for patients that have accumulated a certain dosage of medicine within a short time period. A patient might have been prescribed the same medication for different conditions, leading to the patient receiving a dosage higher than expected. These two queries represent two cases where a Snotra query can be useful, and tests how a query approach is able to handle the tricky temporal health queries we are attempting

to solve.

SQL and a custom query function should both be able to handle the first query without much issue. The second query is significantly more complex. Combined they should give a good estimation of how the underlying engine, be it SQL or a custom Rust implementation, performs during Snotra queries.

4.1.2 Performance requirements

A natural performance requirement for an improved query opperation is that it should be faster than the current version implemented in Snotra. Since we are unable to benchmark the current version of Snotra, a minimum performance "floor" is instead defined. This floor defines a minimum viable performance that must be reached by an implementation for it to not be considered a failure. If the end goal is to enable exploratory queries to find relations, it is implied that the number of queries that can be performed in one workday should be high. The time it takes a query to produce a result has a large effect on how we work; consider a query that runs for a week, versus a query that produces a result the same seccond it is started. Setting a maximum runtime of 15 minutes for a query to run on a 2 GB dataset would allow a user to run a query while taking a short coffee break, and have results ready when they return. If an implementation of a query takes any longer than this, the implementation will be considered a failure as the runtime is long enough that it interferes with the implied goal of performing as many queries a day as possible.

4.1.3 Data pre-processing

For the experiments we use a randomly generated dataset to approximate the expected input data. This is based on a random data generator used to test Snotra, consisting of the following 10 columns:

Pid	Gender	Date of birth	Admission Date	Discharge Date
Icd	Procedure	Dosage	Blood pressure	Atc

This dataset approximates what one might find in a health register, however different registers can store data using different columns. In the real world there would be a pre-processing step to ensure matching column names. Additionaly there are assumptions being made in our implementations of Snotra that rely on a specific ordering of the input data, so this pre-processing step would also include sorting the register data.

We assume that the data is grouped by the Patient ID (Pid) column, and that

each row for each patient is sorted in chronological order. This is what allows us to traverse the input data linearly, at the cost of having to sort the input data. This is beneficial for performing multiple queries on the same dataset as the sorting only needs to occur once, while any number of queries being performed each benefit from increased performance. If we know all rows belonging to a patient is grouped together we can traverse the array of rows linearly and quickly know when we have seen all rows belonging to one patient, as the next row contains a new ID number. When events are sorted chronological we also know that events occur in the order we see them when traversing the array. In the case of an 'A before B' query we can ignore B entirely until we have found A, saving us unnecessary comparisons.

4.1.4 Input Data

We create a dataset consisting of 5 million patients who each could have between 1 and 10 entries. In the dataset this translates to ~27 million entries total. The dataset is a columnar CSV file sorted by patient ID first and admission dates second. This means the entries for each patient is grouped together, and the entries for each patient is stored in chronological order. Smaller datasets are also created in the same way, giving us 4 datasets of increasing size from 1.9 MB to 2.0 GB that can be used when testing queries. The Rust implementation and MariaDB queries were benchmarked using Divan[15]. For MariaDB the benchmark results were double-checked with MariaDB's built-in timer to ensure no performance overhead was added during benchmarking. Each query was run multiple times on each of the 4 datasets, with as few background processes running as possible to prevent any variability between runs during benchmarking. This ensures accurate results, and the differently sized datasets help to visualize how the query performance scales as the input data increases in size.

4.1.5 Datasets for checking query correctness

To ensure a query returns the expected result, a small assertion dataset can be created. The dataset is manually counted, and each patient expected to match the query written down. When this dataset is fed to the query function, we check that the output matches with the expected output. The patients in this dataset also test for different edgecases. For example the 'A before B' dataset includes patients with multiple instances of A occuring before B, A before B where B is the last event for that patient, patients with only A or only B, and patients without either A or B.

4.2 Hardware

The experiments are all run on a ASUS Vivobook S14 (S435), with the following hardware:

CPU:
Intel® Core™ i5-1135G7 Processor 2.4 GHz
(8M Cache, up to 4.2 GHz, 4 cores)

Memory:
16 GB LPDDR4X @ 3200 MHz

Storage:
512GB M.2 NVMe™ PCIe® 3.0 SSD

4.2.1 A before B

Each entry also contains a random ICD code, which will be used as the A and B conditions.

The ICD code 'K52' is chosen for condition A, while 'B02' is chosen for condition B. Any code that appears in the dataset would also work, but these two are chosen to ensure the query gives the same result.

4.2.2 Accumulated within

In our test data 'Dosage' is just an abstraction meant to explain a certain amount of a medicine, and is represented as a random integer value. For this to be a useful query both the threshold dosage and the time range should be variables that can be set to different values. A patient can have between 1 and 10 entries/events, determined at random when the dataset was generated. In a more realistic scenario there would not be a limit to how many entries a patient can have, so the max number of entries for one patient could be expected to be significantly higher. Unlike the 'A before B' query it should be possible to evaluate a subset of events for each patient, since only the dosage has to be given to the patient within a 100-day window. While the length of the timespan is not changed, the dosage threshold is tested at different thresholds.

4.3 What subset of features represents Snotra

Implementing every feature is Snotra is not realistic. Instead, a select few features that represent Snotra are selected for implementation. From the set of all Snotra features, we select a subset of features including:

- 1 A before 1 B
- AND, OR, XOR
- Find 1 A
- Accumulated within

Combined these features make up a prototype Snotra implementation.

4.4 How does the new engine perform under expected workload?

We use four simulated datasets to approximate health register data of varying size. These datasets are queried to help answer the following questions:

Is the query able to run at all with the given dataset?
One way an implementation can fail when performing a query is by crashing or running out of memmory, however queries can also fail by taking too long. If our goal is to help explore large datasets, it's important that the user is able to quickly perform multiple queries.

How does the implementation scale as the size of input data increases?
When querying health data you can quickly end up with very large datasets. It's therefore important to check how a query implementation scales as the dataset increases in size.

How long does it take for the program to execute the query?
When comparing implementations that are both able to run and that both show a linear scaling, the absolute time to complete the query becomes the deciding factor when deciding what implementation to choose.

5

Rust Vs SQL

By implementing a minimum viable implementation of two queries, we can compare how MariaDB and Rust compare as directions for Snotra to take. We can quickly compare two workloads dealing with temporal health queries, and use the results to further develop the more promising approach. These experiments focus on 'A before B' and 'Accumulated within'.

5.1 A before B

5.1.1 Rust

The Rust program works by using a temporary list to store all the rows of a patient, along with flags that mark if a patient satisfies the conditions we are looking for. As patient entries are pushed to the temporary list, the rows are checked for the first condition. If the first condition is met, we mark that the patient satisfies condition A, and begin looking for condition B instead. Should condition B also be found, we mark that the patient matches our query. We know that the patient has developed at least one instance of B after at least one instance of A, since we only look for rows matching B after we have found A, and we never look at the same row twice. When all the rows for the current patient have been added to the temporary list, we can append the temporary list to a larger list that holds all entries for all patients that match the A before B query. The temporary list and all flags are cleared, and we move on to the

next patient. At the end of the query when the end of the CSV file is reached this larger list is returned.

5.1.2 SQL

A challenge with SQL is the increased difficulty in querying rows conditionally like we did with the flags in the Rust query. To check for condition A before condition B, a workaround is used. We can take all the ICD codes belonging to a patient, and append them into a string. Since the data is assumed to be sorted, we can assume that the ICD codes appear in the string in chronological order.

For our testing, we used MariaDB which features a group concatenate function that can be used to concatenate all column entries into a string for each group. We group by the patient ID, and create a new column to store the concatenated ICD string. The strings can then be checked to see if 'K52' appears before 'B02'. This gives the following SQL query:

```
SELECT *, GROUP_CONCAT(Icd) con FROM test
GROUP BY Pid
HAVING con LIKE '%K52%B02%'
```

Before running the query the CSV file is loaded into a database file, and a SQL index is created for the Patient ID and ICD columns.

5.2 Accumulated within

Accumulated within searches for patients that have received an accumulated dosage greater than a set threshold value, within a specific time span. We look at a time span of 100 days, however this value can be set to any number of days. Similarly, the threshold value can be set to any integer value.

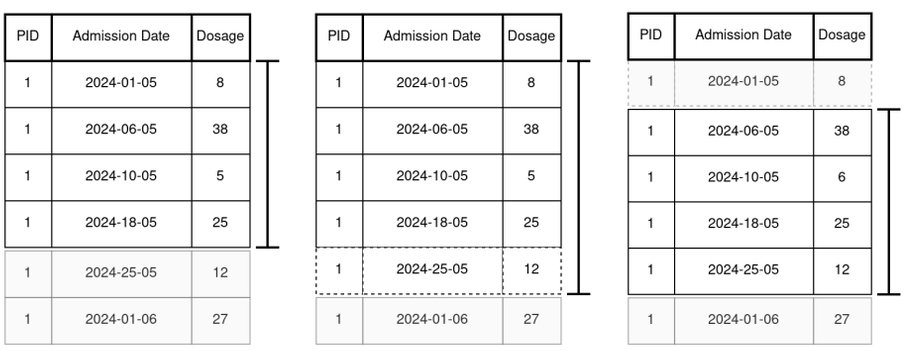
5.2.1 Rust

We use a sliding window to calculate patients accumulated dosage over the length of the window. The window begins at the first row, where we check if the dosage is above the threshold. If not, we increase the end of the window to include one more row. Whenever the length of the window is increased, we check the dates of all the rows in the window to ensure all events in the window occur within a span of 100 days. As long as the total dosage is below

the threshold we increase the window size until the events in the window span more than 100 days.

When the span is longer than 100 days, we start removing rows from the start of the window. Each time we remove a row, we check to see if the window spans less than 100 days, and if the accumulated dosage is over the threshold. We have to check the dosage since the last row added may have both increased the span over 100 days, and increased the dosage enough that the patient is over the threshold even when one row is removed. This process of pushing and popping rows to and from the window continues until we reach the end of the patient, or until the query is satisfied.

We make use of a temporary list to store the current window and a temporary array of all the patient's rows. On a hit, the temporary array is filled with all the rows belonging to the patient and returned. The window list always gets cleared when we look at the next patient.



(a) Extend sub-window until time span is more than 20 days, or dosage over 80

(b) Extended sub-window reaches threshold, but spans more than 20 days

(c) Expell earlier rows in the sub-window until time span is within 20 days

Figure 5.1: Using 'Accumulated Within' to find a patient with over **80** accumulated dosage within **20** days

5.2.2 SQL

There are a few problems keeping us from easily implementing this query in SQL. Firstly each patient has a varying number of rows in the data. Secondly, we need to both sum the values of a subset of the patient's rows and check that these rows happen within the set time period. Window functions, lead, and lag functions enable queries that evaluate multiple rows. If we could define two variables to represent the start and end row of a window we could use a lag/lead function to sum the dosage and compare the first and last row to get the time range. This first step is fairly manageable in SQL. What makes the SQL query

more complex is that each patient does not just have one window to evaluate, but a large array of potential windows that need to be evaluated.

A brute-force technique could be implemented where each potential window for each patient is evaluated, but given that patients have N^N possible windows for N rows per patient this approach would not scale well for datasets of any reasonable size. As with the Rust implementation, we should only evaluate potential windows where the first and last row appear within *Threshold* days of each other. In order to implement this query in SQL we would either need to run multiple queries to support the changing window size, or have a query capable of evaluating and changing the window size and accumulating the dosage within the window until all windows within a *Threshold* timespan have been evaluated for each patient. While this should be possible to implement in SQL, it would have taken more time than we were willing to spend.

Implementing a query like 'Accumulated Within' in MariaDB proved tricky. Unlike 'A before B', we need to conditionally include and exclude rows depending on the timespan length. This conditionality is a weak spot for SQL, where it's more common to search for rows matching a set condition. Even in time-series databases like QuestDB[6], the temporal features are developed to find trends and specific events over a known time period like a week or 6 months. In the case of 'Accumulated within', the timespan itself is not known until we check the admission dates when a patient received a dosage of medication.

5.3 Experiment Results

File size		Best time		Median time	
1.9	MB	5.298	ms	5.432	ms
196.9	MB	646.2	ms	751.9	ms
1.2	GB	3.822	s	3.863	s
2.0	GB	5.894	s	6.767	s

Table 5.1: Scaling of 'A Before B' in Rust

The tables show the runtime for each Rust query. For 'Accumulated Within' different threshold values were also tested, showing that the threshold value has some impact on performance. This threshold value is how high the accumulated dosage value must be for a patient to be included in the result. Plotting the data gives a better picture of how the queries scale as the amount of data queried changes.

For MariaDB, the largest dataset takes a significant hit to performance. In an

File size		Threshold	Best time		Median time	
1.9	MB	300	15.04	ms	15.09	ms
196.9	MB	300	1.564	s	1.542	s
1.2	GB	300	9.425	s	9.31	s
2.0	GB	300	15.7	s	15.62	s

File size		Threshold	Best time		Median time	
1.9	MB	90	16.17	ms	17.17	ms
196.9	MB	90	1.77	s	1.788	s
1.2	GB	90	10.58	s	10.69	s
2.0	GB	90	17.14	s	17.18	s

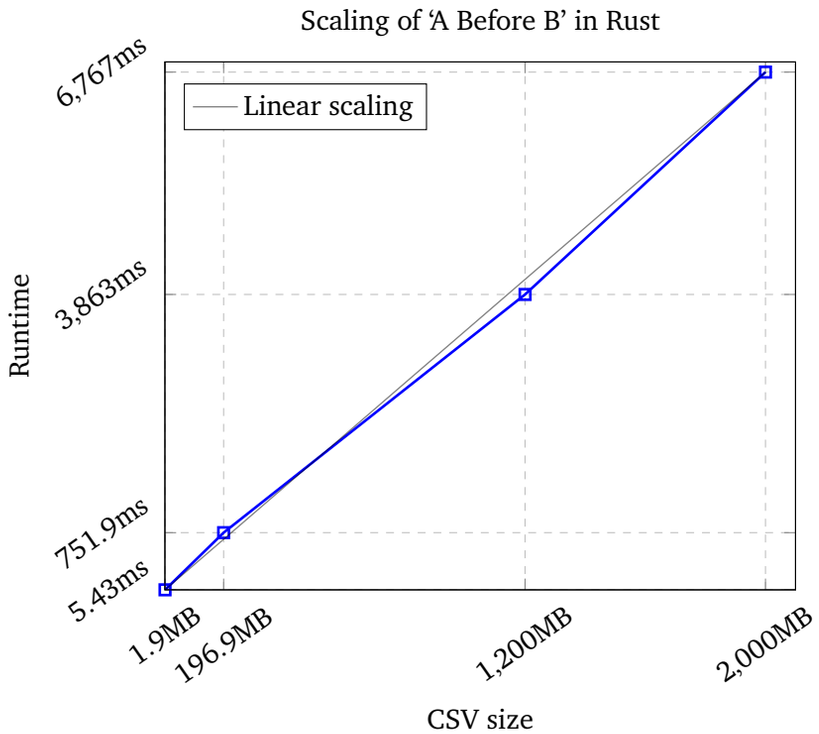
File size		Threshold	Best time		Median time	
1.9	MB	30	17.16	ms	17.57	ms
196.9	MB	30	1.821	s	1.841	s
1.2	GB	30	10.91	s	11.07	s
2.0	GB	30	19.2	s	19.29	s

Table 5.2: 'Accumulated within' in Rust, 100 days

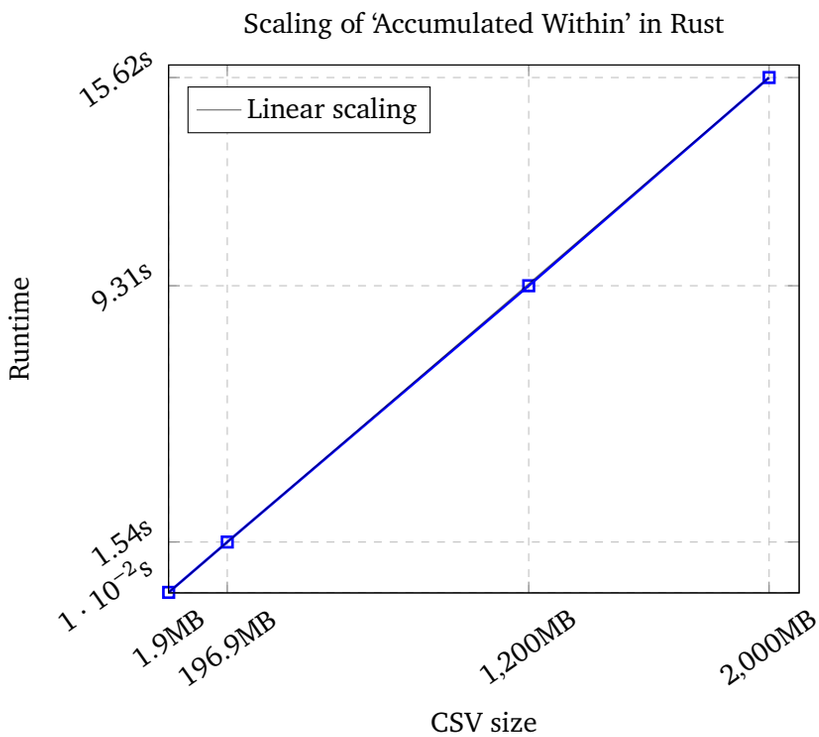
earlier benchmark, the date column was incorrectly set as a string type instead of a date column. With the incorrect column type, we got a linear performance across all dataset sizes. When the column type was fixed the performance of the query increased for the 3 smaller datasets, while the largest dataset slowed down. This is the graph included in the thesis. In the SQL query, we have to sort the patient groups by the admission date to ensure events happen in the correct order, so it is not surprising that performance slows down with larger datasets. This slowdown hints that this might not be the best approach for larger datasets.

For the Rust implementation we see a noticeable increase in performance with the larger datasets. The Rust implementation is not responsible for sorting the input data and will return an incorrect result if the input data is in the wrong order. The increase in performance should therefore be taken with a grain of salt. More important than the absolute performance, the scaling seen in the graph is clearly linear. An additional experiment was performed to compare the Rust implementation of 'A before B' with a Polars variant of the same query. By using just the operations found in Polars, we are able to match the performance of the Rust implementation. The Polars variant has the benefit of not requiring the input data to be sorted, however unlike the SQL query, not having to sort the data either. In the Polars query, we compare patients' first occurrence of 'A' with their first occurrence of 'B'. We can then compare the admission dates of

the two events to check if the first ‘A’ happened before the first ‘B’.



(a) Scaling of 'A before B' in Rust



(b) Scaling of 'Accumulated within' in Rust

Figure 5.2: Scaling in Rust

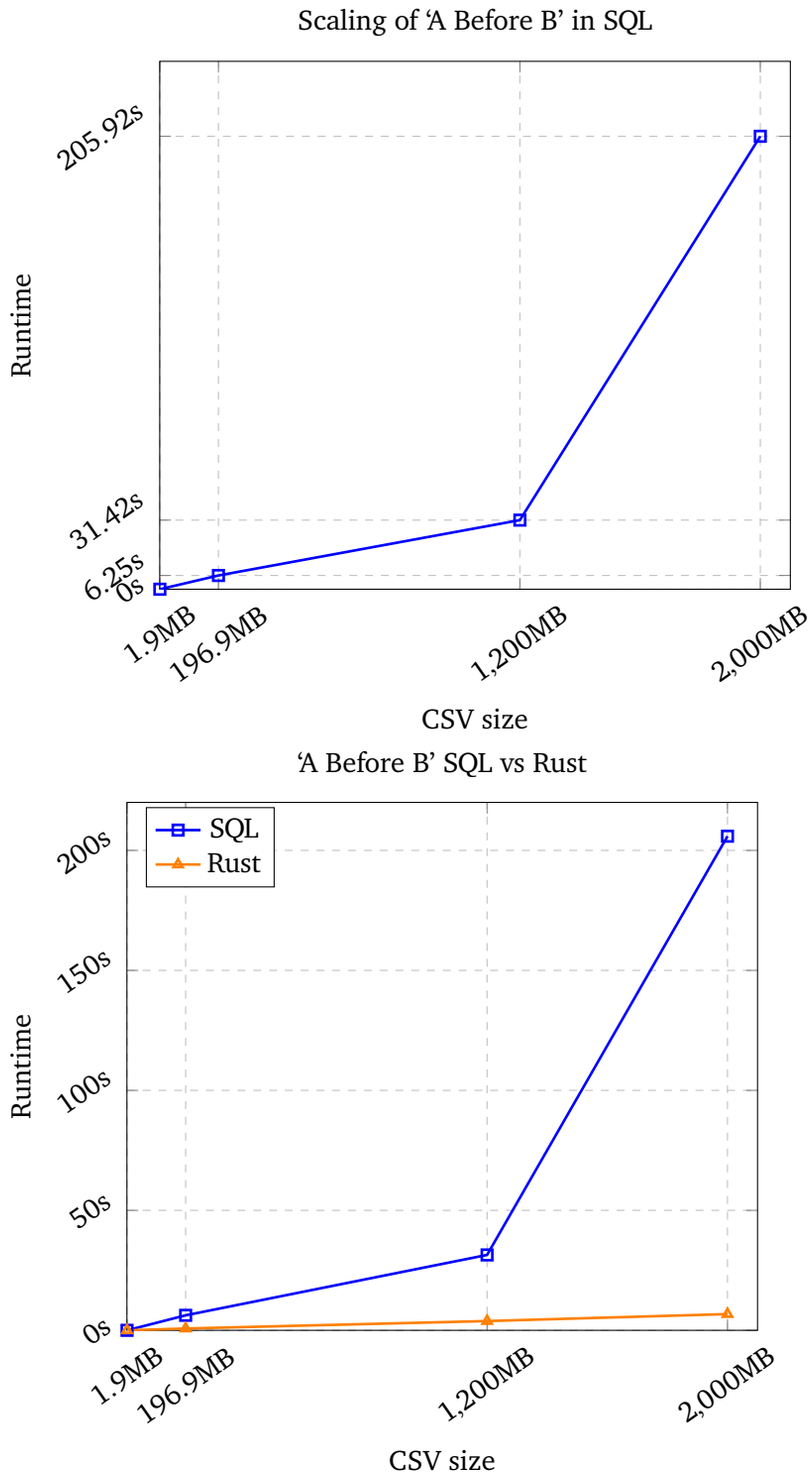


Figure 5.3: 'A Before B' scaling: SQL vs Rust

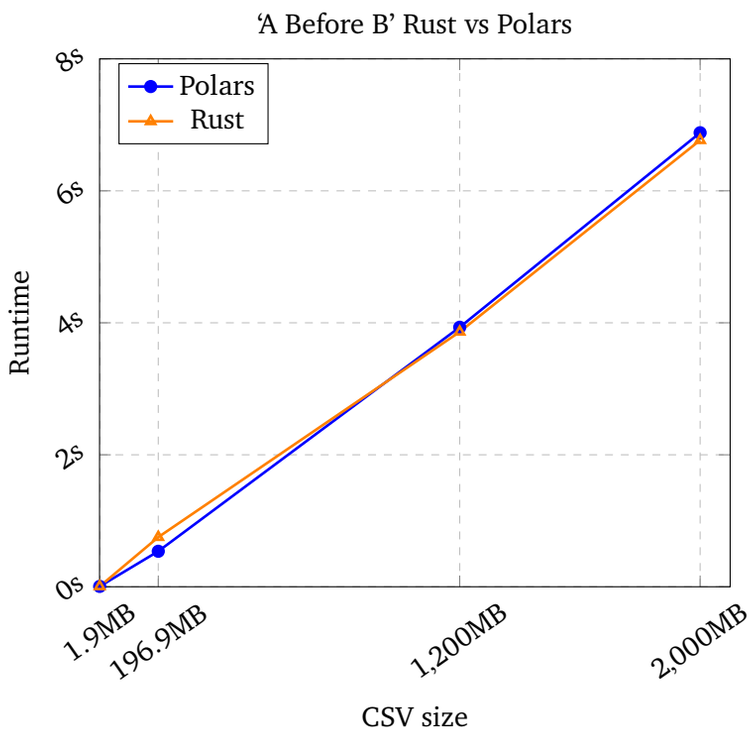


Figure 5.4: 'A Before B' Rust vs Polars

6 Design

When attempting to run the 'A before B' and 'Accumulated within' queries in SQL, it became clear that SQL is not practical for this type of work. In the original Snotra implementation Snotra query functions are implemented as collections of Pandas operations that combine to perform the different queries. We have already created the individual query functions in Rust, and so to make use of the implemented functions in a larger program we can implement the queries as Polars expression plugins[16]. Like the original Snotra implementation, the query functions can easily be accessed by users.

6.1 Queries

A set of assumptions or rules are shared between the different querying functions.

1. All rows belonging to the same patient are grouped together

2. Events are sorted chronologically for each patient

If we always follow these rules we can be sure that events happen in the same order that they are presented in the data. Each query function takes a dataframe as its input, returning a list of patient IDs that match the query. This ID list can be used to filter a larger dataframe, leaving behind a dataframe with only

data pertaining to the relevant patients. This new dataframe can be further filtered by future queries to implement complex compound queries.

6.2 Combining statements

Being able to AND, OR, and XOR different query results allows for more complex queries to be performed. We can for example ask for patients where either:
Condition A occurs before condition B
OR
Condition A occurs before condition C

AND operations between queries can be performed by using the output of the first query as input to the second query. AND operations between queries can be achieved by performing the two operations after each other, filtering out any patients not found in the first query. The second query is only performed on the remaining patients. Any resulting patients from the second query will have to have been also found in the first query.

For OR operations we can perform two queries, getting two lists of patient id's. Combining and removing duplicates from both lists, we end up with a list of unique IDs that are present in at least one of the OR queries. The input data can then be filtered and include only patients found in the OR list, and we have the full output data containing all patient data for all patients matching the query. XOR can be similarly implemented by performing an XOR of the two lists, keeping only IDs present in at most one list.

6.3 Memory requirements

We are querying a very large dataset divided into patients that are each represented by only a small amount of data. The queries evaluate each patient individually, determining if each patient is relevant or not depending on the query. If this evaluation is CPU-bound, adding more patient data to memory will not affect query performance. In-memory execution is therefore only a benefit if the evaluation of each patient is quick enough that I/O becomes the bottleneck. As the scale of the dataset increases in-memory execution becomes more of a hindrance than a benefit, as each additional patient held in memory increases the memory load of the program without providing any benefit. Data-streaming is a good fit for our use case, as we can store the large dataset in a "infinite" virtual array that is easily read linearly. As we process each patient the next is read into memory, allowing us to begin evaluating the next patient

immediately without waiting for I/O, and without unnecessarily high memory usage.

7

Implementation

7.1 Pandas and Python VS Polars and Rust

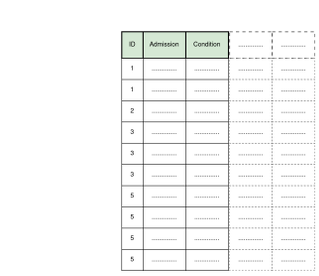
Originally Snotra was developed as a number of expression extensions in Pandas. By combining Python code with collections of Pandas dataframe operations new Snotra operations were implemented, allowing users to easily make use of the new temporal health queries provided by the Snotra language. To implement the more complex features of Snotra, complex Python code is written to check for what features a query is using, sort data, check types, etc. The data is then found by using collections of Pandas operations that create and combine different sets from the input data until the resulting set contains the queried data. The extensive logic in Python combined with the complex set operations in Pandas result in complex query implementations with significant performance overhead.

By moving away from a set-centric workflow into a more linear approach we can simplify the implementation of the various queries. This simplification incurs less overhead and enables us to query data linearly, enabled by the use of custom Rust querying functions. We can compare an 'A before B' query implemented using pre-existing Polars operations, and another implemented with a custom Rust 'A before B' function to see the difference in overhead when implementing queries with custom code compared to just using the available operations. The change from Pandas to Polars complements the custom Rust operations, however it would be possible to achieve something similar by using C extensions for Pandas. A benefit of Rust is its native support for the Cargo

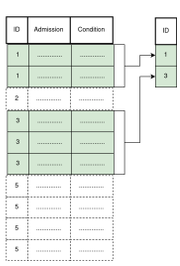
package manager and strict compiler, however these are more benefits for development than the end product. It's natural to change Pandas to Polars since we can make use of its support of Rust while keeping many features from Pandas. Polars focus on speed over flexibility can also be beneficial to our use case in querying since Polars will in many cases outperform Pandas.

7.2 Extending Polars compared to Standalone Rust

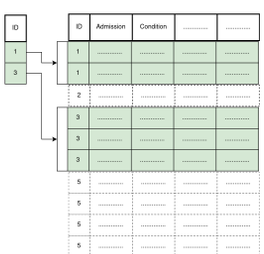
Two prototypes were developed for the new Snotra implementation, one being implemented entirely in Rust and the other porting the Rust query functions into Polars plugins. For the most part, the standalone Rust, and Polars Rust extensions implement the same algorithms, however they differ in how they return the results of the query. The standalone Rust implementation takes as input the entire dataset, returning a new dataset containing only the rows that match the query. In the Polars extensions, only the relevant columns are provided as input, and a list of IDs matching the queries is returned. This list can then be used to filter a dataframe, resulting in a dataframe containing only rows relevant to the query.



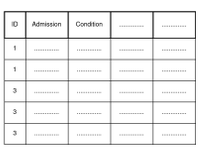
(a) Read relevant columns from dataframe



(b) Find and return patient ID's matching query

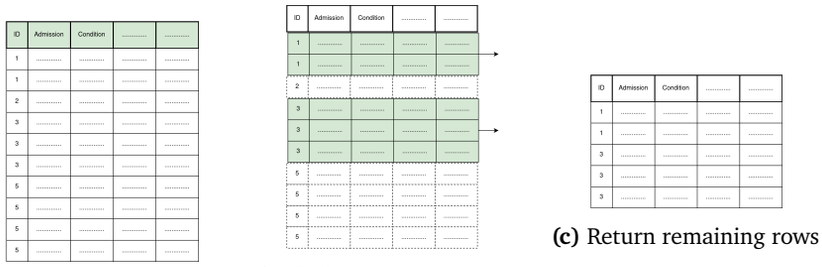


(c) Use ID's to filter dataframe



(d) Dataframe contains all data matching query

Figure 7.1: Polars plugin



(a) Read all input data

(b) Collect all rows where patient matches query

(c) Return remaining rows

Figure 7.2: Standalone Rust

8

Results

Despite having to perform an additional filter step compared to the standalone Rust implementation, Polars shows better performance across the board. When executing an extension in Polars it is the responsibility of Polars to distribute the workload. Polars automatically parallelize the workload, allowing us to benefit from parallel execution without needing to write any multithreaded code. Since we are working with lazy dataframes there could also be optimizations to the filter step that reduce the performance penalty of performing this additional step.

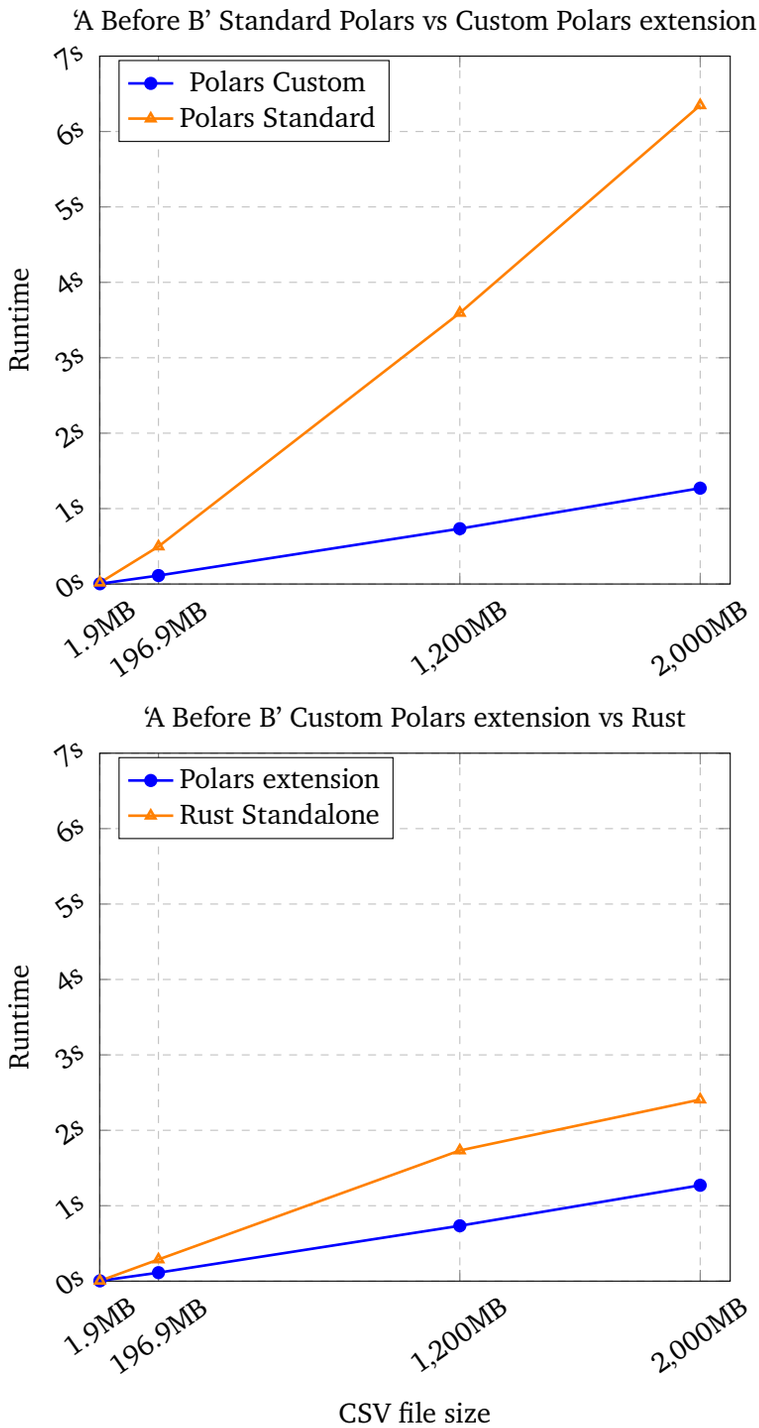


Figure 8.1: 'A Before B' in Polars, Rust, and as a Rust extention

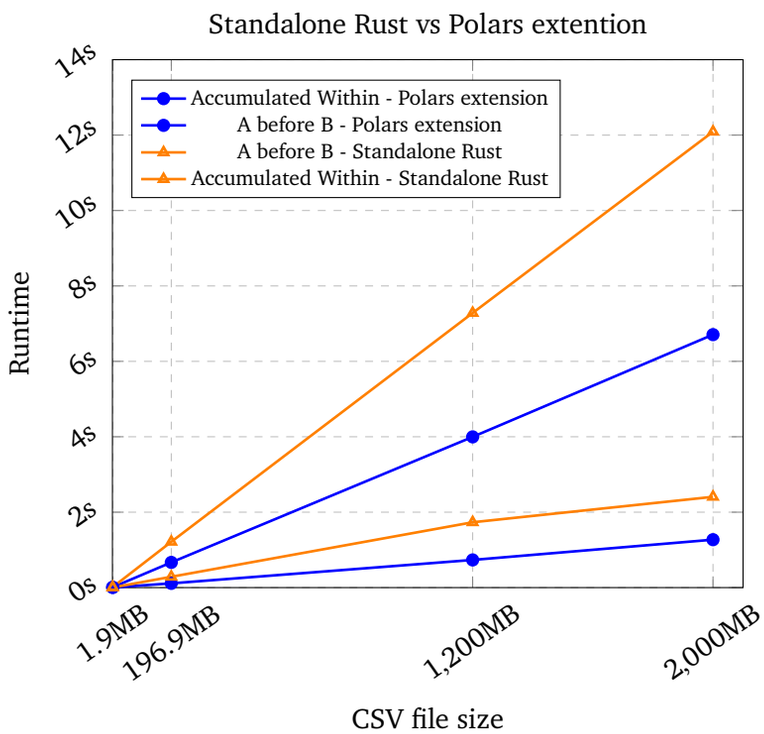


Figure 8.2: Standalone Rust vs Polars extention

9

Discussions

9.1 Limitations with recusive functions

One feature of Snotra not implemented is the ability for an ‘A before B’ query to use a query function in place of ‘A’ or ‘B’. Rust can support passing a function as an input to another function, however there are some limitations with Polars. When implementing expression extensions for Polars, the input for the expression is deserialized. The deserializer does not allow deserializing unknown functions, making it very difficult to pass generic functions as inputs to expression extensions. In this case, if we want to provide a function as an input to another query function, we would have to define the inputs to that function. This makes it difficult to implement expressions like ‘A before (Accumulated Within())’ or ‘A before (A before (A before B))’, since the ‘Accumulated Within’ and ‘A before B’ functions have different inputs. That said, limiting the types of ‘A’ and ‘B’ makes the query easier to execute, which could lead to an advantage in terms of performance. The increase in performance might be more valuable than the increased expressiveness gained from more general inputs, however this gain is difficult to quantify.

9.2 In-memory execution

There was an attempt at performing in-memory query execution to improve performance. This in-memory design conflicted with complex compound queries.

CSV file	Filesize	Peak memory usage	Runtime
test large	2.0 GB	8.5 GB	31.49 S
test medium large	1.2 GB	4.5 GB	18.04 S
test medium	196.9 MB	1.0 GB	3.08 S
test small	1.9 MB	8.4 MB	40.62 MS

Table 9.1: Peak memory usage with file size

CSV file	Filesize	Peak memory usage	Runtime
test large	2.0 GB	DNF	DNF
test medium large	1.2 GB	DNF	DNF
test medium	196.9 MB	3.6 GB	12.27 S
test small	1.9 MB	8.4 MB	86.71 MS

Table 9.2: Peak memory usage with file size

For an AND query, we require at least three queries, two sub-queries followed by the final AND query between the sub-queries. For both sub-queries, we have to keep not only the entire input data in memory but also the output. Depending on the query performed, the output could be similar in size to the input data.

We can only expel the input data from memory when all sub-queries have been completed, however the outputs still have to remain in memory until the final AND has finished execution. Expelling the input data from memory also means we don't get the performance benefits of in-memory data when running multiple queries. We end up with slower query execution since we have to wait for all data to be loaded before we can begin executing queries.

9.2.1 Memory usage

When loading all data into memory before executing queries.

On average the operation above takes ~4.4 times as much space in RAM as the input CSV file. A heavier workload could be taking the XOR of two A before B queries. Here the memory usage is closer to an 11 times increase compared to the input file.

With 16 GB of memory available, compound queries on large datasets end up getting killed as we run out of memory. Systems with large memory capacity can keep more data in memory. With patient-oriented loading of data, it is easy to increase the number of patients loaded into memory at once to allow high RAM systems to benefit from in-memory execution. Patient-oriented loading

of data also enables easy parallel query execution as each node can be given N patients each to evaluate and collect the found patients in a shared list.

9.3 As a standalone Rust program

A benefit with having the querying opperations available as a plugin for another program like Polars is the ease-of-use for users. In cases where for example a data scientist is performing temporal health queries there is a high chance that other types of queries are being performed. Having Snotra as a plugin therefore allows them to use the same tools for all of their work. This can be offset if the standalone program is published as a python package. Significant amounts of developmental effort would be needed to implement the performance optimizations present in Polars. The main benefit of developing a standalone program as opposed to a plugin is the additional flexibility gained by not having to match another api, however it is difficult to quantify how much this would benefit development, if at all.

10

Conclusion

When performing patient-oriented queries, the event-oriented health registers combined with the temporal aspects of the query can be tricky to work with utilizing existing tools. By creating tools to more easily perform these types of queries the barrier to exploring health register data is lowered, enabling the discovery of new relations between health events like medications, procedures, and pre-existing conditions. Querying tools currently available, including temporal SQL variants, are not designed with health queries and therefore make performing these temporal health queries difficult and time-consuming. While Snotra attempts to handle these types of queries, the underlying architecture is not able to scale with increasingly large datasets.

Two typical usecases for Snotra were selected as expample queries that could be implemented to find a better approach for the underlying implementation. 'A before B' being useful for finding cases like a patient that first recieves a diagnosis, before later being admitted to the hospital. 'Accumulated Within' helps find patients that have recieved too much accumulated medicine from different procedures within a set time-span.

The experiments comparing these queries in SQL to Rust found Rust to be the better approach. In the larger datasets, the SQL queries were not able to keep up with the Rust counterpart. An SQL query for 'Accumulated Within' was not found with reasonable performance.

While it is possible to develop an entirely standalone Rust app to handle tem-

poral health queries, there are several benefits to instead develop Rust plugins for Polars. A user is then able to use Polars for data collection, processing, and the temporal health queries. Performance wise Polars is able to distribute the workload effectively on the machine running the query, without the need to write any parallel code. We are also able to utilize Polars opperations to enable AND, OR and XOR opperations between queries, allowing for more query expressiveness.

By extending Polars with custom querying functions written in Rust we are able to implement temporal health queries in a manner that both scales well and is easy to use. The performance of the prototype exceeded expectations, able to perform queries on a 2 GB input file in a matter of secconds. This approach can be used to improve the implementation of Snotra to enable its use with large datasets. Our experiments did not result in the findings of any benefit of using standard SQL, however development of custom agregate functions might yield better results. Similar results to the ones found in Polars could likely be achieved using Pandas and custom extentions written in C.

11

Future Work

Several features of Snotra are missing from the new prototype. A key feature missing is the ability to use arbitrary values, including other queries, as arguments in queries like “A before B”. This enables queries such as

“1 (A) before 3 (B)”

“(A) before (Accumulated within(...))”

“Find 3 (Accumulated within(...))”

More flexibile data input is another feature missing from the prototype. The current prototype is currently designed to work for data in a very specific structure. Features to make the prototype more flexibile in the type of input data would not only help with the usability of the program, but might enable uses even outside of medicine.

Bibliography

[1] Morten Schmidt Mika Gissler Unnur Anna Valdimarsdottir Astrid Lunde Kristina Laugesen Jonas F Ludvigsson and Henrik Toft Sørensen. "Nordic Health Registry-Based Research: A Review of Health Care Systems and Key Registries". In: *Clinical Epidemiology* 13 (2021). PMID: 34321928, pp. 533–554. DOI: 10.2147/CLEP.S314959. eprint: https://www.tandfonline.com/doi/pdf/10.2147/CLEP.S314959. URL: https://www.tandfonline.com/doi/abs/10.2147/CLEP.S314959.

[2] Kari Furu. "Establishment of the nationwide Norwegian Prescription Database (NorPD) – new opportunities for research in pharmacoepidemiology in Norway". In: *Norsk Epidemiologi* 18.2 (2009). DOI: 10.5324/nje.v18i2.23. URL: https://www.ntnu.no/ojs/index.php/norepid/article/view/23.

[3] *From event level data to person level answers*. 2019. URL: https://github.com/hmelberg/snotra/wiki/From-event-level-data-to-person-level-answers (visited on 26/02/2024).

[4] *Snotra - Health registry research using Pandas and Python*. 2019. URL: https://github.com/hmelberg/snotra (visited on 17/04/2024).

[5] *ICD-10 Version:2019*. 2019. URL: https://icd.who.int/browse10/2019/en (visited on 23/04/2024).

[6] .

[7] In: ().

[8] David Toman. "SQL/TP: A Temporal Extension of SQL". In: *Constraint Databases*. Ed. by Gabriel Kuper, Leonid Libkin and Jan Paredaens. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 391–399. ISBN: 978-3-662-04031-7. DOI: 10.1007/978-3-662-04031-7_19. URL: https://doi.org/10.1007/978-3-662-04031-7_19.

[9] In: ().

[10] *Incompatibilities and Feature Differences Between MariaDB 10.4 and MySQL 8.0*. URL: https://mariadb.com/kb/en/incompatibilities-and-feature-differences-between-mariadb-10-4-and-mysql-8-/ (visited on 23/04/2024).

[11] *About pandas*. 2024. URL: https://pandas.pydata.org/about/index.html (visited on 21/02/2024).

[12] *DataFrames for the new era*. 2024. URL: https://pola.rs/ (visited on 21/02/2024).
[13] *How Do You Know If a Graph Database Solves the Problem?* 2024. URL: https://neo4j.com/developer-blog/how-do-you-know-if-a-graph-database-solves-the-problem/ (visited on 01/03/2024).
[14] *MariaDB*. URL: https://mariadb.com/ (visited on 05/05/2024).
[15] *Divan*. 2024. URL: https://github.com/nvzqz/divan (visited on 20/02/2024).
[16] *Expression plugins*. URL: https://docs.pola.rs/user-guide/expressions/plugins/ (visited on 23/04/2024).