SmartDataLake

DELIVERABLE D2.1

Query engine over virtualized data

PROJECT NUMBER: 825041
START DATE OF PROJECT: 01/01/2019
DURATION: 36 months

SmartDataLake is a Research and Innovation action funded by the Horizon 2020 Framework Programme of the European Union.

Horizon 2020

Dissemination Level	Public
Due Date of Deliverable	Month 16 (30/04/2020)
Actual Submission Date	30/04/2020
Work Package	WP2: Adaptive Data Virtualization and Storage Tiering
Tasks	Task 2.1: Query engine over virtualized data
Type	Other
Lead Beneficiary	EPFL
Approval Status	Submitted for approval
Version	1.0
Number of Pages	31
Filename	SmartDataLake-D2.1-Query_engine_over_virtualized_data.pdf

Abstract

This report presents the query engine of the SmartDataLake platform that allows running queries over virtualized data. First, we give an overview of the existing individual components that we have used in the query engine. We then list the features that we have developed on these components for the SmartDataLake platform. Next, we provide instructions for installation and present how the query engine can be used by other components of SmartDataLake as well as end users of SmartDataLake.

History

Version	Date	Reason	Revised by
0.1	10/02/2020	Table of Contents	Bikash Chandra
0.2	26/03/2020	First draft for internal review	Bikash Chandra
0.3	20/04/2020	Revised draft	Bikash Chandra
1.0	24/04/2020	Final version for submission	Bikash Chandra

Author List

Organization	Name	Contact information
RAW Labs	Benjamin Gaidioz	ben@raw-labs.com
EPFL	Bikash Chandra	bikash.chandra@epfl.ch
EPFL	Ioannis Mytilinis	ioannis.mytilinis@epfl.ch
EPFL	Dimitra Tsaoussis Melissargos	dimitra.tsaoussis@epfl.ch
EPFL	Anastasia Ailamaki	anastasia.ailamaki@epfl.ch
TU/e	Odysseas Papapetrou	o.papapetrou@tue.nl
TU/e	Hamid Shahrivari	h.shahrivari.joghan@tue.n

Executive Summary

The data virtualization layer allows SmartDataLake to run queries over heterogeneous data found in different storage locations while being agnostic to this fact, as well as to variations in data formats. In SmartDataLake, data is often accessed via queries written in SQL. This report presents our query engine, discussing its main components and how they interact with one another. It also provides instructions on the installation and usage of the query engine along with examples.

In Section 1, we motivate the need for the query engine of SmartDataLake and discuss why existing solutions are not sufficient. We also discuss how the query engine fits into SmartDataLake and how it is used.

The two main components of the query engine developed by the EPFL and RAW Labs teams are Proteus and RAW, respectively. We discuss these components of the query engine in Section 2. Proteus allows running queries on heterogeneous hardware and offers support for querying nested fields. It introduces the HetExchange operator that allows queries to exchange data between different hardware. It also uses JIT code generation to generate code at runtime to support execution of different operators on different hardware devices like CPUs and GPUs. RAW allows to process data from heterogeneous data sources and in heterogeneous data formats. It allows users to define and materialize views so that data from different sources and in different formats can be seamlessly accessed as virtualized tables. We discuss the features of each of the systems in the context of SmartDataLake. We present our contributions about the new features added in course of the SmartDataLake project.

Section 3 describes the integration of the two components to deal with the challenges of heterogeneous data sources and formats to process queries over heterogeneous hardware. We discuss how views and materialized views from RAW are made available to Proteus. We then explain how queries are executed in Proteus by accessing data from RAW. We also present the optimizations we implemented during the integration.

Section 4 provides a user manual for the end users of the query engine and other components of SmartDataLake. It includes the installation steps for RAW and Proteus. It also demonstrates how views and materialized views are created in RAW from different data sources and how different data formats can be handled. We provide steps for using both the web interface and the command line tool offered by RAW. We then show how a user can connect to Proteus and run queries. We use two example datasets to illustrate the usage, including a synthetic dataset and the H2020 projects dataset that is made publicly available by EU.

Abbreviations and Acronyms

DL	Data Lake
ETL	Extract Transform Load
JIT	Just-in-time
RQL	Raw Query Language
SDL	SmartDataLake
SQL	Structured Query Language

Table of Contents

1. Introduction 8
   1.1. Overview 8
   1.2 Scope in the project 9
2. Query Engine Architecture 10
   2.1. Proteus 10
      2.1.1. Existing functionalities 13
      2.1.2. New functionalities 13
   2.2. RAW 14
      2.2.1. Views 15
      2.2.2. Credentials registration 15
      2.2.3. Functionalities added during SDL 15
3. RAW-Proteus integration 17
   3.1. Exposing RAW views to Proteus as tables 17
   3.2. Query transformations for using RAW in Proteus 17
   3.3. Query execution on Proteus and RAW 18
4. Query Engine User Manual 18
   4.1. Installation 18
   4.2. View definition: RAW language 19

4.2.1. RAW web frontend 19
4.2.2. Executing a RAW expression 20
4.2.3. Reading a dataset from an authenticated storage 21
4.2.4. Automatic inference 23
4.2.5. Defining a view 23
4.2.6. Reading a dataset from a public storage system 25
4.2.7. Defining a materialized view 26
4.2.8. RAW command-line tool 27
4.3. Running SQL queries on Proteus 28
4.3.1. Connecting to Proteus Query Engine 28
4.3.2. Listing tables and attributes 29
4.3.3. Running queries in Proteus 29
5. Conclusion 31
References 31

1. Introduction

1.1. Overview

Data lakes ingest and store data in different formats and from different locations. The data virtualization layer of SmartDataLake (SDL) provides an abstraction for accessing and querying different data sources in a unified manner. The query engine enables the data virtualization layer to run queries across different data formats and across different data sources.

Consider a use case where a company has collected some data on academic institutions to give recommendations to prospective students. The data may be stored in a database server on the premises of the company. The government provides some public information about these institutions that is periodically updated and is made available on their website in the form of Excel files. When making recommendations the company would like to consider the public data too. Without having a data virtualization layer, the company would need to download the public data locally, parse the files and extract the relevant information and then add it to their database before the public data can be used. This step would need to be repeated periodically as the public data may get updated.

In general, the input data may come in different formats such as JSON, CSV, Excel and binary (i.e. non textual format such as data stored in relational databases). Moreover, data could be available at different locations; some data could be stored locally, some could be stored in S3 storage, while others could be available from a website using an http URL. A single input query may require access to all of the data mentioned above. Existing solutions require assembling first the data into a central location and transforming the data to a desired format which the query engine supports before being able to run any query. This process of fetching data from sources, transforming it and then loading it to be made accessible for queries is known as the Extract, Transform and Load (ETL) pipeline. ETL is an expensive process both in terms of computation resources and manual effort. ETL is dependent on the specific application and the data formats being dealt with. The ETL step must be executed before any query can be processed on the underlying data. Moreover, the ETL process needs to be repeated regularly based on how frequently the data changes and the freshness requirement of the application.

Existing state of the art frameworks such as Hive and Spark provide support for some data formats and some storage locations. However, these frameworks do not support cross-format optimization. Several important features are restricted to specific well-established formats. End users still need to perform ETL for unsupported data formats since the data formats natively supported by these systems are limited. As discussed above, ETL is expensive, requires manual effort and takes time to make the data available for querying. These frameworks also do not facilitate good data organization which is essential for optimal data storage on data lakes.

The data virtualization layer of SDL allows applications to seamlessly access data stored in different locations and in different formats using a single query engine. The application running on SDL does not need to know the specific format of the stored data or the location of the specific files storing the data. The data location and format can be specified by the user. The data is accessed at its original format and from its original location. The data virtualization layer handles the complexities of parsing and handling different data formats, fetching data from different data sources and optimizing queries.

Traditionally, query processing has always been done on CPUs only. Modern computing hardware such as GPUs provides opportunities for efficient and parallel processing for several query operators. The query engine of the virtualization layer can also use different hardware for computation as it can execute queries using both CPUs and GPUs. The query engine uses just-in-time (JIT) code generation techniques to generate code optimized for the specific hardware.

In SDL, the data virtualization layer is made of two main components, the Proteus query engine and the commercial system RAW. RAW is used to provide data access to underlying hardware and provides access to different file formats and different file storage locations. Proteus provides the capability to run SQL queries on files present in RAW as well as local files. It supports queries on data that have nested file attributes, i.e. attributes that have nested fields themselves. Proteus allows queries to utilize both CPUs and GPUs. A query planner parses the query and generates the query plan. If the query requires data access to heterogeneous data sources, it generates an appropriate query plan using RAW to access the data.

In the following sections, we discuss the architecture of the query engine and provide an overview of the Proteus and RAW systems. We then describe how the systems have been integrated and the required optimizations to achieve the integration. We provide instructions on how to install the integrated system and how to access data and run queries for SDL along with some examples.

1.2 Scope in the project

The query engine helps other components run queries over different data sources. For example, the Query Approximation Layer (WP2) can connect to the query engine via a JDBC interface for retrieving subsets of the available data, and summarizing it, in order to be able to approximate queries. The query engine can be used in a similar fashion for HIN exploration (WP3) and from the visualization components (WP4). Having a single point of data access enables efficient and location-transparent access to heterogeneous (and possibly remote) data. End users too can run queries using the query engine. Notice that this deliverable is limited to exact queries, since our work for approximate queries and the Query Approximation Layer is discussed in Deliverable 2.2. This deliverable provides the work package WP2 component of milestone MS2 due in month 18.

2. Query Engine Architecture

In this section, we discuss the architecture and main components of the query engine. This includes pre-existing as well as new components developed in the framework of the project.

The architecture of the data virtualization layer is depicted in Figure 1. A query planner parses the input SQL query and creates a query plan. Proteus executes the query by accessing data from the Storage Manager and RAW. Users can access RAW using a view management interface. Users can set views for different types of data sources in RAW along with their authentication keys if desired. RAW accesses data from multiple sources and makes it available to Proteus for querying. The Storage Manager is responsible for storing and optimizing data storage and access and will be described in Deliverable 2.3.

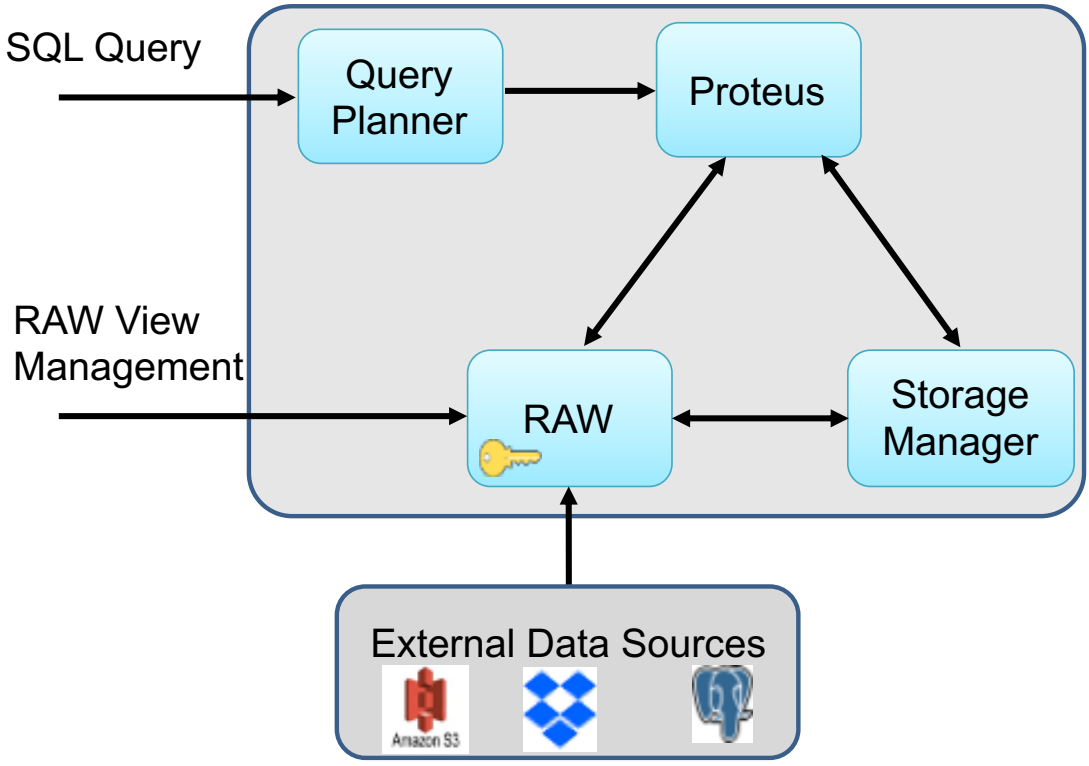


Figure 1: Query engine.

2.1. Proteus

Proteus [1,2] is a query execution engine developed at EPFL. It provides the implementation for various SQL operators that will be used in running queries. The query algebra of Proteus is based on monoid comprehension calculus [3]. Proteus provides support for hybrid query execution over CPUs and GPUs. It also supports queries over various data formats using a plugin architecture. The architecture of Proteus is shown in Figure 2.

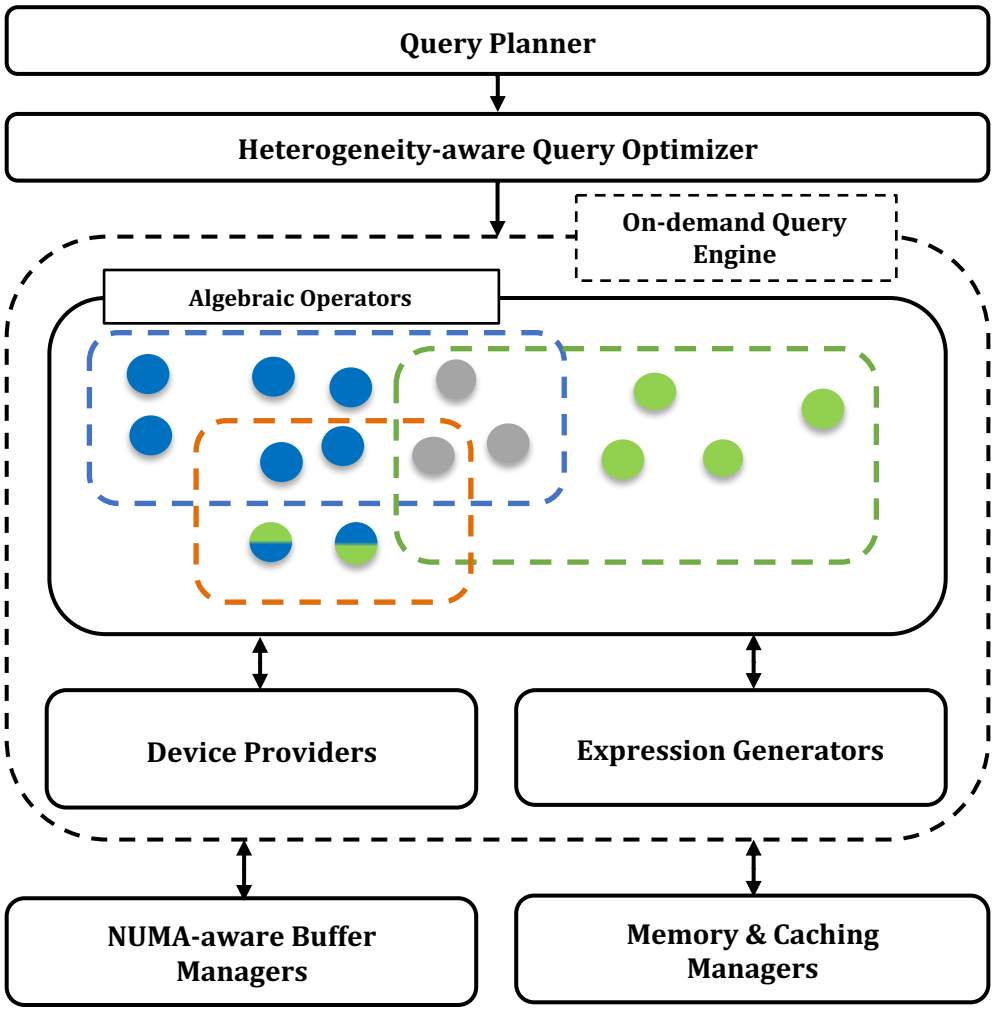


Figure 2: Proteus architecture.

Proteus supports common data formats like CSV and JSON using a plugin architecture. Each plugin implements some functions that enables Proteus to read the data format and run queries on it. Proteus generates an appropriate on-demand query engine which is customized and optimized for the given plugin and data format. Since Proteus uses an algebra based on monoid comprehension calculus, it also has support for nested fields like lists, sets and maps. Such nested fields need not be flattened when storing the data. A query can access nested attributes by using the "unnest" keyword. The nested fields are flattened/unnested during query execution if required.

Database systems use a CPU-only query execution model. Modern processing hardware like GPUs provide a high degree of parallelism. Queries running analytics over large datasets, such as the queries used in data lakes, benefit from the use of such high degree of parallelism. However, it may not be beneficial to run all operators on GPUs. Proteus allows execution of queries in a hybrid mode using both CPUs and GPUs. The users of the system do not need to be aware of the processing heterogeneity. Proteus has an optimizer that takes into account the underlying

hardware heterogeneity to optimize query plans for the given hardware. In order to execute query operators across different devices, Proteus uses just-in-time (JIT) compiled engines based on LLVM [4] to generate compiled query execution code at runtime. The generated code is based on the specific hardware on which the operator will be executed. The JIT code specifies the different algebraic operators for the query execution as shown in Figure 2. The device providers and expression generators are used to generate the specific code for the hardware. Since the query execution is done across CPUs and GPUs, which do not have a shared memory, data needs to be transferred between devices.

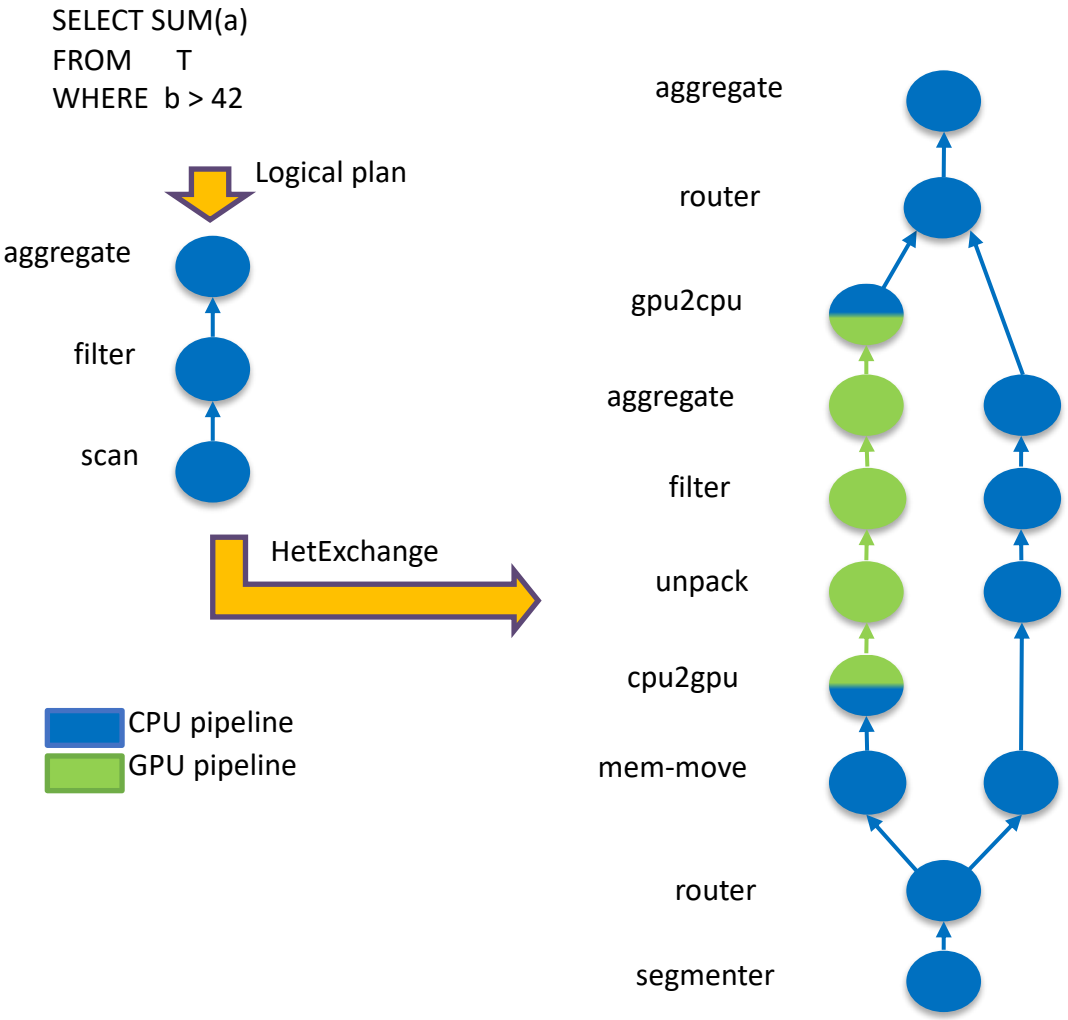


Figure 3: Example of HetExchange operator.

Proteus uses the HetExchange operator [2] to transfer data across device boundaries. The HetExchange operators allows parallel execution of query operators across different devices. An example of the HetExchange operator is shown in Figure 3. For the given query, the query planner generates a logical query plan which has 3 nodes. The heterogeneity aware optimizer decides to run filter and partial aggregation in hybrid CPU-GPU mode and generates code in time for both

the CPU and GPU execution. HetExhange allows the exchange of data from CPU to GPU and vice versa. The router operator allows Proteus to route data for CPU and GPU execution pipelines. The mem-move operator and cpu2gpu operators are used to move the data from CPU to GPU. Each pipeline in the CPU and GPU execution unpack the data, execute the filter and compute partial aggregates. Data from GPU is then transferred back to CPU to compute the final aggregate.

To run queries on Proteus, users can connect to it using either a SQL command line interface or using standard database connectors like JDBC. We give a detailed explanation along with examples in Section 4.

2.1.1. Existing functionalities

Below, we list the features provided by Proteus before the SDL project started.

Query execution over heterogeneous hardware. Proteus supports query execution over both CPUs and GPUs. GPUs provide a high degree of data parallelism and help accelerate processing of analytical queries. Query execution for a given query can be split across CPUs and GPUs to get better performance on analytical workloads. Proteus uses code generation to generate LLVM code specific to the hardware on which a given part of the query has to be executed. It uses a HetExchange [2] operator to transfer data between CPU memory and GPUs.

Query execution over heterogeneous data using a plugin architecture. Proteus can run queries over heterogeneous data formats like CSV and JSON. It uses a plugin architecture and plugins need to be defined for each supported data format. The plugins define how data in a given format needs to be parsed to access the records in the data and the fields in each record. Proteus supports nested data formats; a field may be a list of values or a map containing different attributes.

2.1.2. New functionalities

In the following, we present the features that are added to Proteus as part of the work in SDL. Data access from RAW is already completed, while the other two parts are ongoing work.

Data access from RAW. RAW provides efficient access to different data formats and different data storage locations. In order to run queries over these data formats and storage locations, we have added support for accessing RAW views and materialized views through Proteus. Details of the implementation are provided In Section 3. In a nutshell:

- In SmartDataLake, we have added support for fetching schema information of RAW views and materialized views. The schema information contains the table names, the column names and the data type of the column. The schema information helps the query planner appropriately parse and compile the input queries.
- We have added support for new query operators that treat RAW views as special tables during query execution in SmartDataLake. When a query based on RAW views is processed, we create appropriate RAW query calls to fetch data from RAW. We also add

optimization to push selections and projections over RAW views to the appropriate RAW query.

- Using REST APIs provided by RAW, we issue queries to RAW. Proteus then uses the data returned by RAW to run the remainder of the query plan.

Storage manager. The input data could be present at different levels of hierarchy. In particular, data may need to be accessed from cloud storage services. We will add support for optimally accessing data stored at different storage hierarchies including cloud storage.

Scale out query processing. Proteus currently runs on a single machine only. For processing queries across multiple nodes, we plan on using the Resource Manager to allocate resources to run the query. We plan to add support for distributing input data and intermediate results across nodes. We also plan on implementing support for running operators in parallel across multiple nodes, as well as for synchronization to check when an operator finishes and releasing resources accordingly.

2.2. RAW

RAW is a query execution engine, which is being developed by RAW-Labs since 2015. It connects to storage systems and enables users to read and combine datasets together to make enhanced sets of data in real time, regardless of data format and complexity. These datasets can be used to build data-driven applications without having to replicate the data.

RAW's main features include:

- It queries the data in-place, without requiring data loading or schema creation.
- It supports complex structured data, including hierarchical data and multidimensional arrays.
- It provides multiple extensions to SQL to support additional operations ranging from data cleaning to log parsing.
- It supports multiple input locations, including HDFS, HTTP, Amazon S3, Dropbox and relational database systems.
- It supports multiple input formats, including CSV, JSON, HJSON, XML, Microsoft Excel, log files.
- It supports multiple output formats, including JSON, HJSON, Parquet among others.
- It autonomously caches and optimizes data and queries, based on usage patterns and without DBA intervention.

RAW's expression language –RQL– supports all functions necessary to create workable data. While it supports by itself multiple analysis use cases, in SmartDataLake, RAW has the role of the first access layer to the data: it exposes the datasets to the SmartDataLake query engine (Proteus) through RAW *views* which encapsulate the RAW processing (credential management to access storage systems as well as data parsing, cleansing, formatting).

While Proteus supports the CSV and JSON data formats, it is not sufficient to use it to handle heterogeneous data in SDL for the following reasons:

- Proteus does not support data access from different cloud services. In SDL, since datasets could reside on different locations, Proteus alone would not be sufficient to run queries on them without downloading the data to some local storage.
- Proteus does not handle files like Excel documents and XML files which would be useful in SDL. Even in case of CSV and json files that are used in SDL, some files require special parsing mechanisms which are not supported in Proteus.

Hence, we use RAW for handling different data formats in SDL.

2.2.1. Views

In RAW, views are a way for a user to encapsulate a specific expression under a logical name so that it can be reused elsewhere easily. In SmartDataLake, this is also how Proteus is able to access RAW. Proteus populates its catalog with the RAW views so that a user can run Proteus queries on these views as if they were regular tables. Since Proteus is the query tool for exact queries in SmartDataLake, this is how users of SmartDataLake eventually query the physical datasets. Integration between RAW and Proteus for that purpose is described in this document.

2.2.2. Credentials registration

Access to a storage system is in general protected and a user is granted access to it with credentials. The management of a user's credentials is supported in RAW and is reused in SmartDataLake.

When authenticated in RAW, a user can register credentials to storage locations using the RAW user interface, the RAW command-line tool or a REST API.

2.2.3. Functionalities added during SDL

RAW existed before SmartDataLake and could already read the various file formats we work with. Still, SmartDataLake came with use cases which were challenging to RAW, and required developing the following new features:

2.2.3.1. Full inference

RAW includes a practical inference feature: a dataset can be loaded by a RAW expression without specifying its parsing details because they are inferred during compilation, using a sample of the dataset. During SmartDataLake we had to process large files whose parsing details could not be inferred using a sample and it felt reasonable to trigger inference on the full dataset content.

2.2.3.2. Materialized views

Some datasets stored in raw text format are particularly expensive to read and parse: their size is large due to simple encoding (e.g., boolean values stored in strings "true", "false") which leads to high disk IO, and turning them into binary for processing is expensive.

Internally RAW does sometimes save intermediate results in an efficient binary format which significantly improves the cost of reading a dataset. A user could not control this feature. We have implemented materialized views so that a user can instruct RAW to cache a view in binary format. Part of this work also included important improvements to the generation of the code in charge of reading these binary files.

2.2.3.3. Performance of cache reading/writing

Materialized views are highly used in SmartDataLake and performance of writing and then reading the cached data is critical. We carried investigations and developments to ensure that component was performing as efficiently as possible.

2.2.3.4. Input file format support

Prior to the project, RAW supported common file formats which are followed by most of the datasets used in SmartDataLake. Nevertheless, while many of the datasets could be read straight away, some of them came with challenges to both format inference and reading, e.g. CSV files with an unsupported convention which we had to support, or JSON-encoded very large records which required optimization of the reader code (e.g., several of the SpazioDati Elasticsearch exports are containing very large records).

2.2.3.5. Programmatic API (REST and C)

In SmartDataLake, RAW is used both interactively to define views, and programmatically when executed by Proteus. Prior to the project, RAW provided a REST API as well as a Python API. New code was developed in order for Proteus queries to be able to trigger RAW from C code, using the existing REST API. Initially it was an ad hoc component implemented for the project. During the project, the preferred communication layer became a new one based on web sockets supported by RAW as well. Therefore, it was eventually decided that RAW will support a C API providing the functionalities needed by Proteus, using web sockets. This is under development and will be finalized during the project.

3. RAW-Proteus integration

In this section, we describe the three main aspects of the RAW-Proteus integration:

- Proteus' catalog is populated with the user's RAW views. This permits a SmartDataLake user to query them. This happens by fetching the related view metadata from Proteus, using RAW's APIs.
- Once RAW views are known to Proteus as its tables and a query is entered, the query is processed by Proteus' optimizer which produces an execution plan. The plan includes calls to the RAW plugin in order to retrieve the content of the user view. However, since RAW is able to execute some fraction of the plan (e.g. some filters and projections coming with the user query could be executed by RAW), the optimizer is also performing some specific transformations to the query plan to push some of its operations to the RAW plugin, which internally turns them into the ad hoc corresponding RAW query, on top of the original simple reference to the user view .
- After the query plan has been optimized and some fraction of it has been offloaded to RAW (at a minimum, the execution of the view alone), Proteus drives the execution of the whole query as both systems execute their share of the plan.

3.1. Exposing RAW views to Proteus as tables

In order to run queries on views created in RAW, Proteus must be able to get the list of views in RAW and its attributes. The RAW list of views can be programmatically obtained using the REST API. At startup Proteus fetches the schema information about the views and materialized views, using the REST API, and adds the information to its catalog to reflect the RAW views. Proteus does not only need to be aware of the views but it also has to get specific metadata about them, in particular their data type (the type of the records each view exposes) which is required information about a data source for its engine to parse and compile a query. Note that at this stage no data present in the RAW views is fetched by Proteus. RAW provided all information as part of the REST API.

3.2. Query transformations for using RAW in Proteus

The query parser parses the SQL query and identifies which relations in the query are RAW views or materialized views. For such relations, RAW is used to access the underlying data. The query planner creates corresponding queries in RQL to issue queries to fetch the data corresponding to these relations.

As an optimization to the above approach, the planner decides to push operators such as selections and projections to RAW. The query planner constructs RQL queries that contain these selections and projections. This allows us to reduce the amount of data that is fetched by RAW. Proteus gets pruned after removing some rows and columns that are not required for the query. The query planner provides the transformed query plan to Proteus after removing the operators that had been pushed to RAW.

3.3. Query execution on Proteus and RAW

RAW provides a REST API for a client to execute a query and retrieve results in a specific format. The communication between RAW and Proteus to retrieve data from a view is implemented using that API. Proteus uses the REST API to issue calls to RAW and fetch results corresponding to the RQL query issued to RAW. Results are retrieved in JSON format from RAW which is then parsed by Proteus and used by Proteus as any other input data source. Proteus is thus agnostic to the data source from which the data is retrieved.

As part of the integration with the storage manager, results of an RQL query will be instead sent to the storage manager and only the storage block identifiers (a very small list of identifiers) are going to be retrieved by Proteus which will then be able to read the actual data using the storage manager.

4. Query Engine User Manual

In this section, we provide installation instructions and demonstrate the usage of the query engine.

4.1. Installation

A RAW installation consists of multiple services which run as dockers. The engine requires a Kubernetes cluster to run services and execute queries, as well as an HDFS installation which it can use for intermediate storage, and which it can also read from if raw datasets are hosted there.

In order to install Proteus, CUDA drivers for Nvidia GPU processing must be installed on the system. Proteus can be installed by downloading the source code and running the make file. The make file downloads the dependencies of Proteus and creates executables for running Proteus.

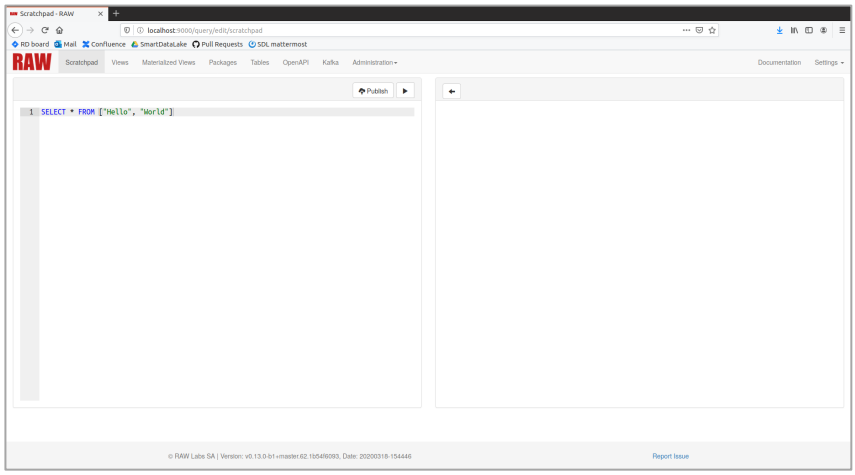
4.2. View definition: RAW language

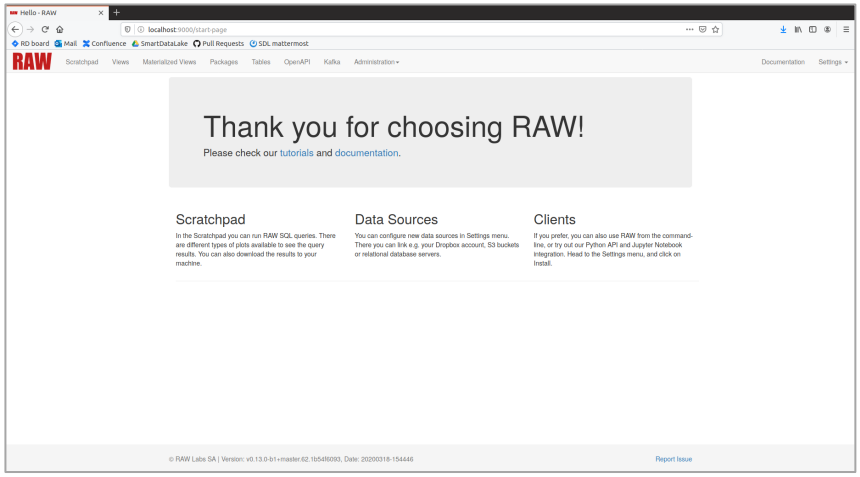
This is the language used when defining a view. The specifications of the language are available online. In this section we show how to use the RAW language to load datasets and encapsulate the processing within a view so that Proteus can then be used to query them.

4.2.1. RAW web frontend

The web frontend is a web interface where one can run queries and edit configuration. It permits a user to execute an important fraction of the workflow in a graphical environment. Thus, it is the preferred way to get started.

Open a browser and connect to the frontend. At the first use of the frontend, you will see a welcome screen. After the first use, the welcome screen is skipped and the scratchpad is displayed. If the welcome screen appears, select the Scratchpad. It is now possible to execute a query to test that the system is working.





4.2.2. Executing a RAW expression

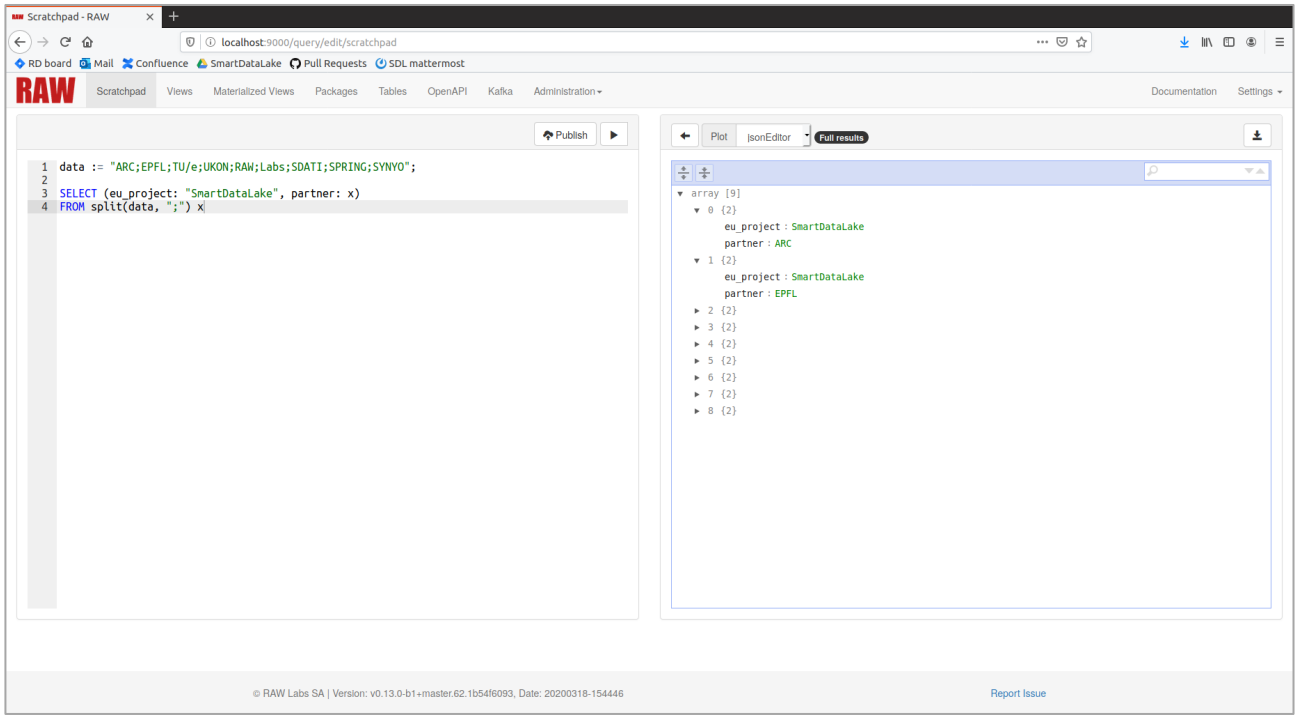
A query usually runs over a dataset stored externally; however, as an exercise, it is also possible to start with a tiny *inlined* set of items as a test. Copy and paste the code below to the scratchpad editor.

```
data := "ARC;EPFL;TU/e;UKON;RAW;Labs;SDATI;SPRING;SYNYO";

SELECT (eu_project: "SmartDataLake", partner: x)
FROM split(data, ";") x
```

This expression splits the embedded string into a collection of substrings which are then wrapped as a record before being returned.

Click the 'play' icon at the top to trigger the execution. The results appear on the right side.



Several viewers are available after a query has returned. By default, the frontend selects a safe JSON editor which is the most flexible viewer since it supports whichever type the results have. Several other options are available when the data shape allows it.

4.2.3. Reading a dataset from an authenticated storage

In the above query, the dataset was inlined in the query. Let's now read one from storage.

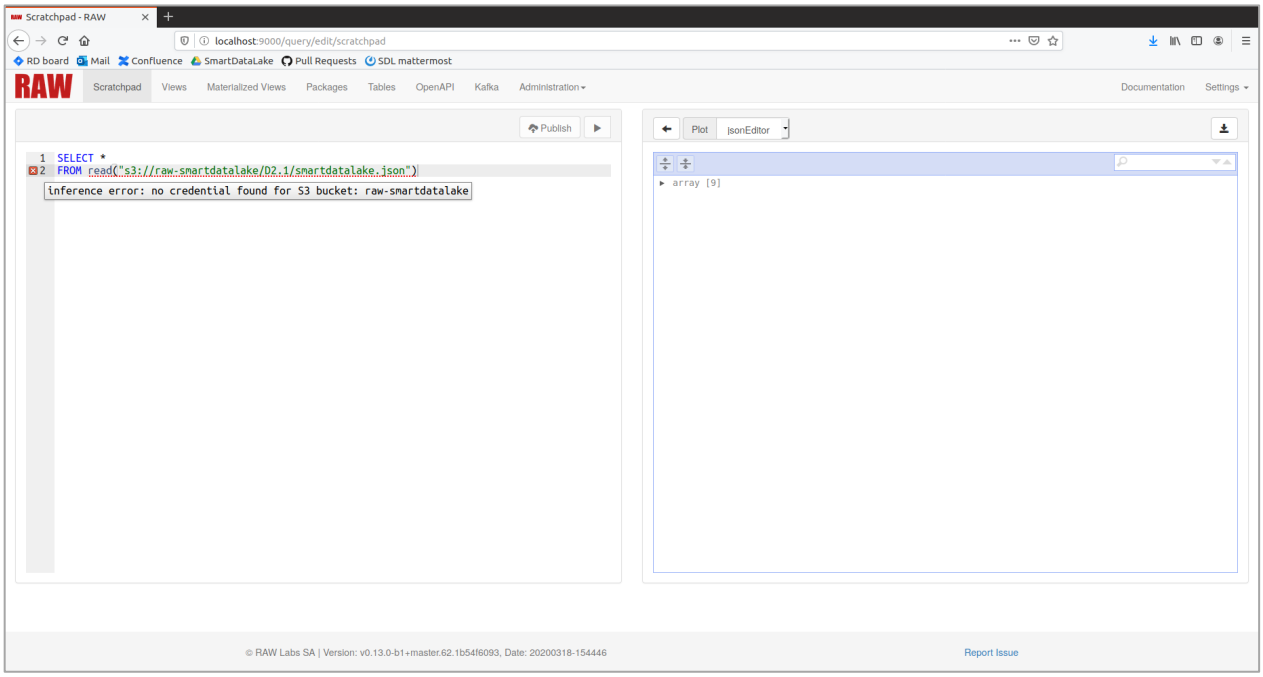
A JSON file describing the SmartDataLake team members is available online as a file in a public server. Here is the content of the file.

```
[{"partner": "ARC",
  "people": ["Person A. One",
             "Person A. Two",
             "Person A. Three",
             "Person A. Four"]},
 {"partner": "EPFL",
  "people": ["Person B. One",
             "Person B. Two",
             "Person B. Three",
             "Person B. Four",
             "Person B. Five"]},
 {"partner": "TU/e",
  "people": ["Person C. One",
             "Person C. Two",
             "Person C. Three",
             "Person C. Four",
             "Person C. Five"]},
 {"partner": "UKON", "people": ["Person D. One"]},
 {"partner": "RAW Labs", "people": ["Person E. One"]},
 {"partner": "SDATI", "people": ["Person F. One"]},
 {"partner": "SPRING", "people": ["Person G. One", "Person G. Two"]},
 {"partner": "SYNYO", "people": ["Person H. One"]}]
```

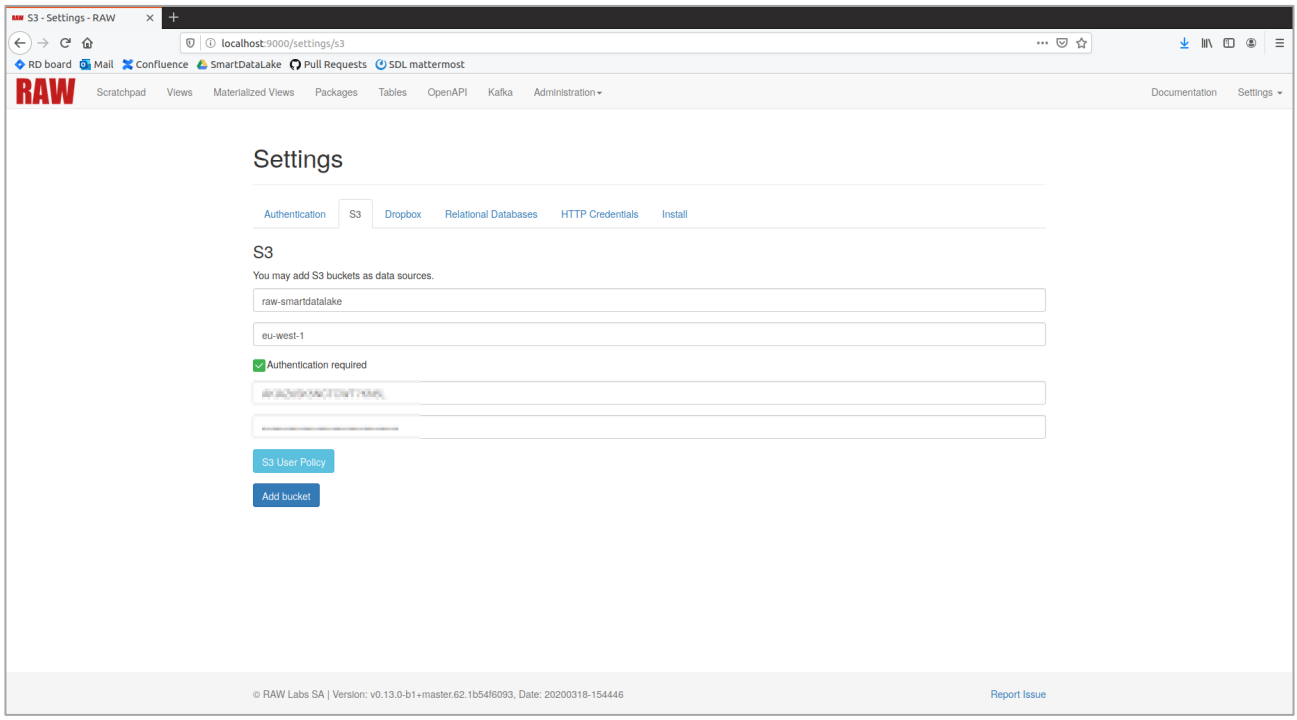
Enter the following query in the scratchpad to read it.

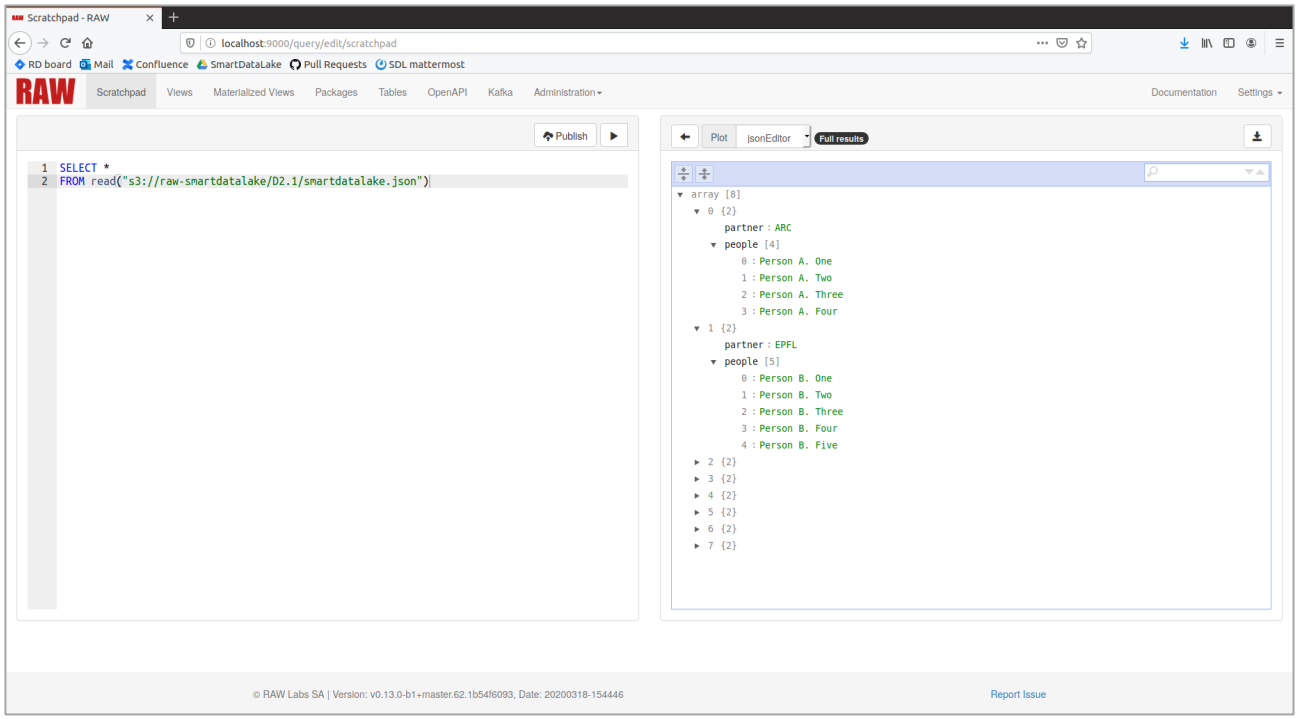
```
SELECT *
FROM read("s3://raw-smartdatalake/D2.1/smartdatalake.json")
```

The read call is underlined in red: the expression cannot be compiled or executed. If hovering the mouse over the red cross an error message is displayed: the URL points to a storage protected by credentials and we haven't entered credentials yet. The "play" button is grayed-out making it impossible to click on it.



Click on the "Settings" tab, then "S3" and enter the missing credentials. Go back to the scratchpad and execute the query again. Since the user is now authenticated to use the S3 storage, this time it works and results are displayed on the right.





4.2.4. Automatic inference

We use a generic call (read) to load the dataset. That call does not pass any information about the dataset format or type, while the expression still types and runs successfully. Since the parsing parameters are not specified, RAW implicitly prefetches a sample of the file and infer its type and parsing details. This can be extremely convenient when the dataset has a complex type.

While inference is quite sophisticated, its logic can always be confused by a complex file. When this is a problem it is always possible to specify parameters to the reader manually. The expression below is equivalent but does not rely on automatic inference: the json parser is instead generated from the type specification passed as a parameter.

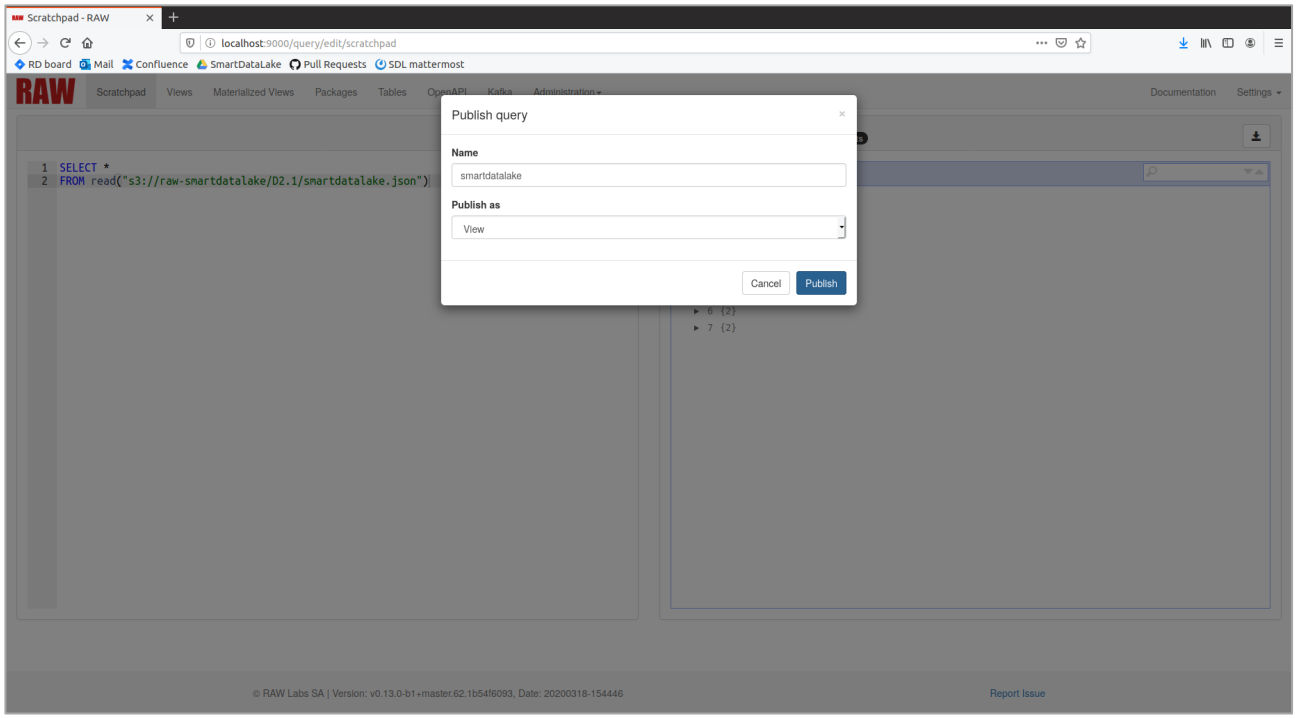
```
SELECT *
FROM read_json[collection(record(partner: string, people: collection(string)))]
    ("s3://raw-smartdatalake/D2.1/smartdatalake.json")
```

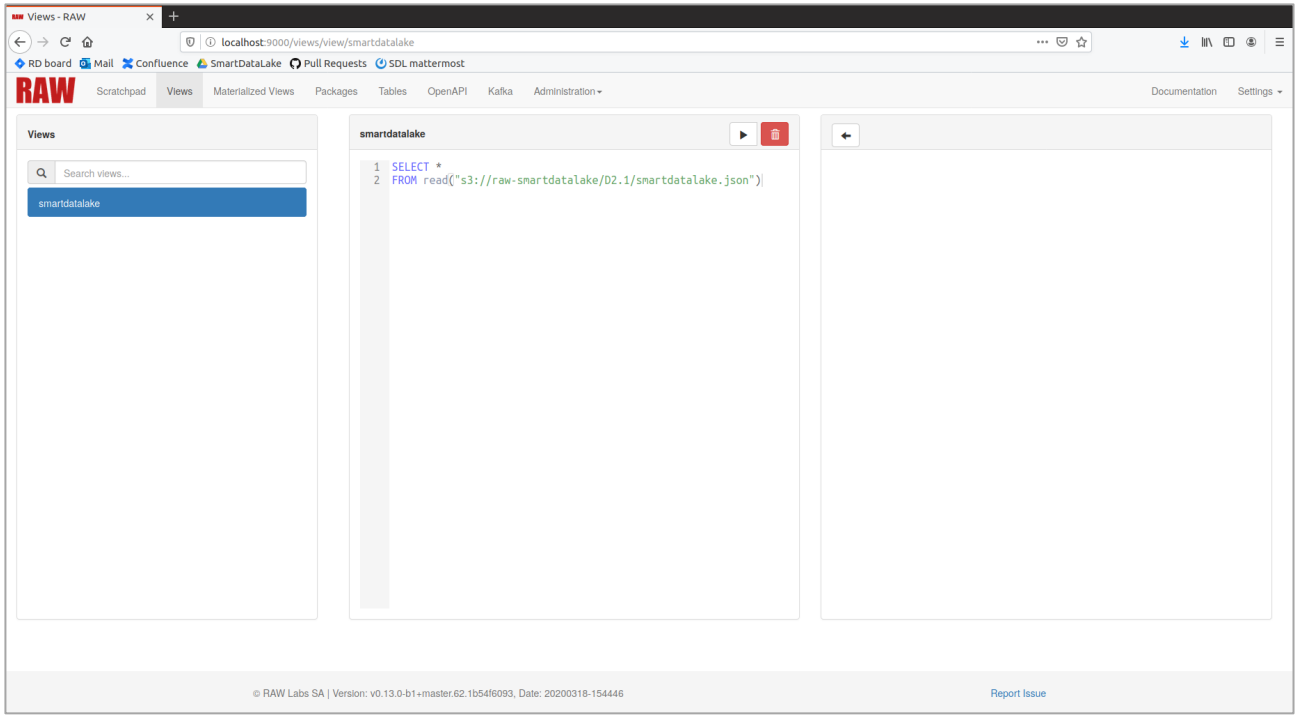
4.2.5. Defining a view

All these expressions we are entering in the frontend can only be evaluated by RAW itself. In SmartDataLake, Proteus can query the datasets using RAW. This is possible once a RAW expression has been published as a view or a materialized view.

Click the Publish icon at the top of the editor. Pick the "view" option and enter “smartdatalake” as a name.

You are automatically taken to the view tab where the new view is now showing in read-only.





4.2.6. Reading a dataset from a public storage system

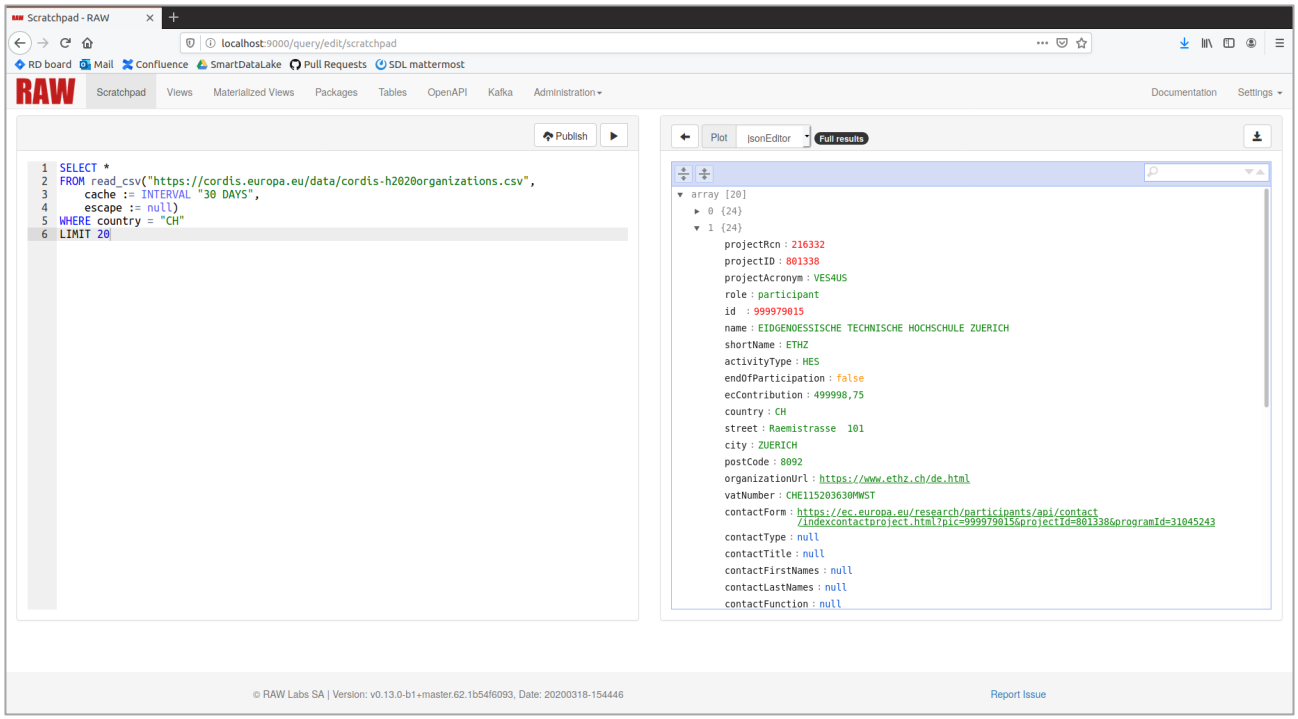
If a dataset is made available publicly, reading works the same way except that no credentials are required. Here is how to read the first couple of entries of one of the EU cordis datasets. This is a dataset containing the names of European project partners. More specifically, it is a CSV file whose format and parsing details can be *mostly* inferred automatically by RAW. However, not everything is figured out: one has to specify that there is no escape character, otherwise some lines fail to parse when evaluating the view.

The dataset is large, and is located in remote storage. To avoid downloading and loading the dataset in each execution, we specify a cache duration, which lets RAW the possibility to cache and reuse intermediate results for a whole week.

The query expression includes a filter on the country and it limits the number of results to 20. This is fine when investigating the content of a dataset however it should not remain when the view is defined.

```
SELECT *
FROM read_csv("https://cordis.europa.eu/data/cordis-h2020organizations.csv",
    cache := INTERVAL "30 DAYS",
    escape := null)
WHERE country = "CH"
LIMIT 20
```

The expression triggers the file download, type inference and parsing. Results are eventually displayed.



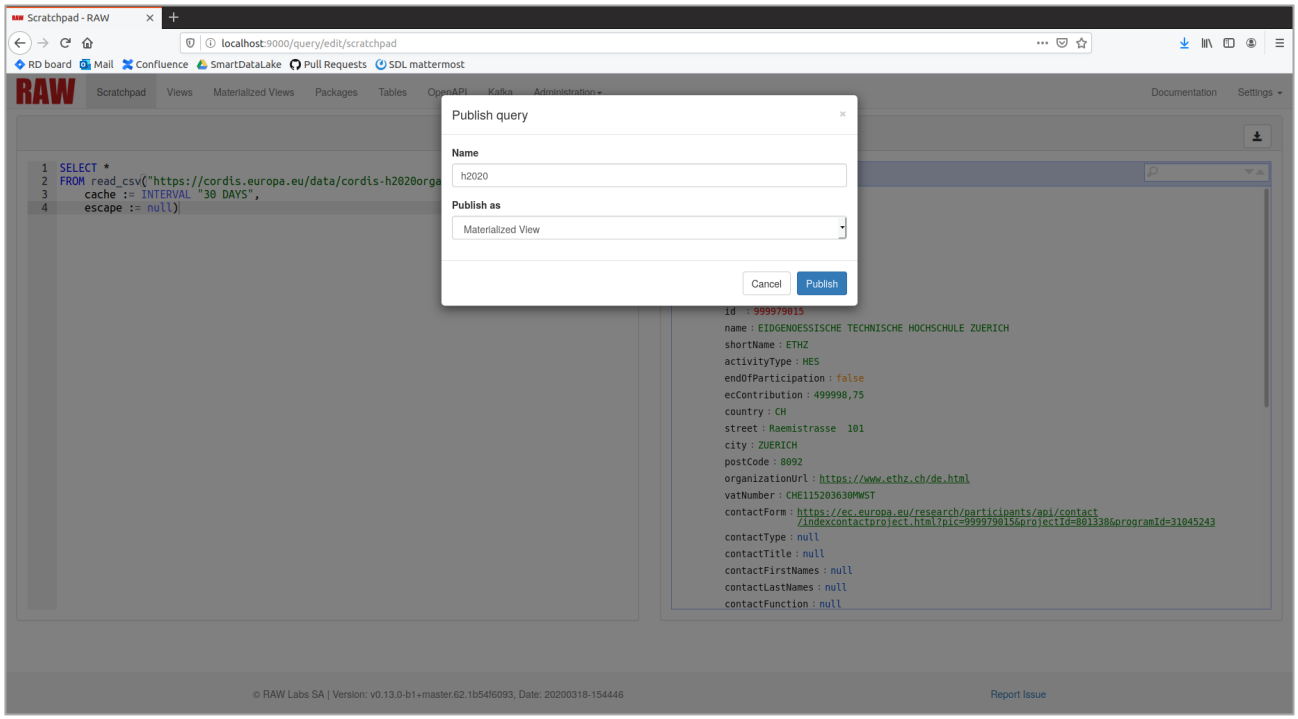
The fact that we specified a cache duration does not imply that the dataset was cached, since this is only a hint for RAW. The best approach is to define a *materialized view*.

4.2.7. Defining a materialized view

The above dataset is large and remote and we would like to use it often. While RAW does caching automatically, it is also possible to specifically cache the output of a view so that the cache is reused as much as possible when the view is used.

Replace the expression by the one below so that it evaluates to the full file content (we remove the WHERE and the LIMIT clauses), and click the Publish icon at the top of the editor. This time pick the "materialized view" option and enter "h2020" as a name.

```
SELECT *
FROM read_csv("https://cordis.europa.eu/data/cordis-h2020organizations.csv",
    cache := INTERVAL "30 DAYS",
    escape := null)
```



You are automatically taken to the materialized view tab where the new view is now showing. Materialized views are like views, they are visible to Proteus by their name.

Now you can go and query these two views (smartdatalake and h2020) from Proteus.

4.2.8. RAW command-line tool

Everything one can do with the frontend is also available using a command-line tool called rawcli which is installed by SmartDataLake. The only required step is to have it configured to use the SmartDataLake server.

Configuration

In the frontend select Settings and then "Authentication". Download the interactive clients configuration. It is a file called "raw.ini". Save it in your home folder.

From a shell run rawcli. Without parameters it shows the available list of commands.

Running the command-line tool

The command-line tool is practical to run prepared queries in a shell script or to try something quickly. It can also be used to publish a query as a materialized view, list the existing views, delete them, etc. but also add credentials like we did using the frontend.

Here is how to run a query.

```
rawcli query 'SELECT * FROM smartdatalake'
```

The tool prints the results of the query in text on the screen.

```
$ rawcli query 'SELECT * FROM read("s3://raw-
smartdatalake/D2.1/smartdatalake.json")'
partner    | people
-----------+-------------------------------------------------------------------
-----------------------
"ARC"      | ["Person A. One", "Person A. Two", "Person A. Three", "Person A.
Four"]
"EPFL"     | ["Person B. One", "Person B. Two", "Person B. Three", "Person B.
Four", "Person B. Five"]
"TU/e"     | ["Person C. One", "Person C. Two", "Person C. Three", "Person C.
Four", "Person C. Five"]
"UKON"     | ["Person D. One"]
"RAW Labs" | ["Person E. One"]
"SDATI"    | ["Person F. One"]
"SPRING"   | ["Person G. One", "Person G. Two"]
"SYNYO"    | ["Person H. One"]
```

If a query is too long to be inlined as an argument, it can also be read from a file, or from standard input.

```
rawcli query -i myquery.rql
cat myquery.rql | rawcli query
```

4.3. Running SQL queries on Proteus

Proteus can run SQL queries defined over local data as well as over views or materialized views defined in RAW. The interface to connect to Proteus is either a standard database connector like JDBC or the command line.

4.3.1. Connecting to Proteus Query Engine

Proteus query engine can be started by running the command

```
$ make run-server
```

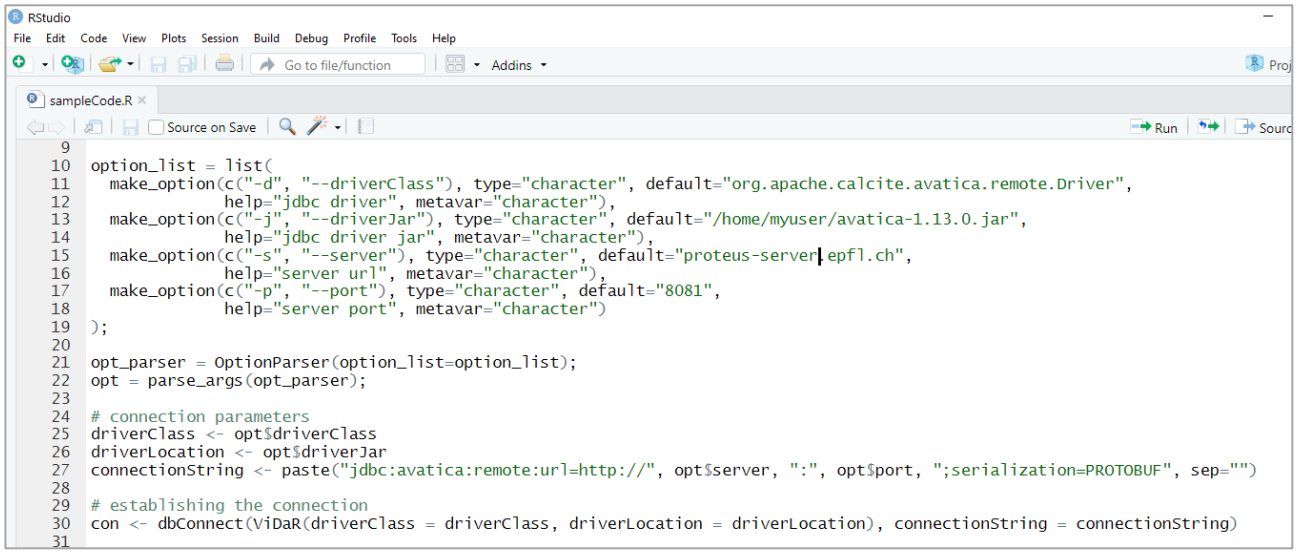
from the project directory.

To connect to Proteus can run queries from the command line uses can run

```
$ make run-client
```

This opens a command line SQL terminal.

To connect to Proteus in a program a standard database connecter can be used. The process of connecting to the database and running queries is similar to a standard database like MySQL. A sample connection code written is R is shown in the figure below.



4.3.2. Listing tables and attributes

The metadata about all tables present in Proteus are present in the "metadata.TABLES" and "metadata.COLUMNS" relations. The TABLES relation contains the name of the tables along with the schema name and the type of table. The COLUMNS relation contains the column name, the table information to which the column belongs and the data type of the column. The views and materialized views defined in RAW are also available in Proteus along with their attributes and attribute types.

4.3.3. Running queries in Proteus

Query Example 1

Let us consider a query on the RAW materialized view "smartdatalake" defined in the previous section.

```
SELECT s.partner, Count(p)
FROM   smartdatalake s, Unnest(people) p
GROUP  BY partner;
```

In this query, we want to find the number of people from each institute that were present in the meeting. Since the people attribute is a list, we need to unnest the *people* attribute to get access to each record within *people.* Each value in the people table is now available as a row in the table. The GROUP BY groups the table by partners and the count gives the number of people per partner that were present.

Query Example 2

Let us now consider a query that does a join between the "smartdatalake" materialized view and the “h2020” view. Note that these datasets are present at different sources. In a conventional database system, we would only be able to query data that is available locally.

```
SELECT h.country, Count(*)
FROM   smartdatalake s, h2020 h
WHERE  s.partner = h.shortname
GROUP  BY h.country
```

This query does an equijoin between the "smartdatalake" and "h2020" data based on matching the names of the institutes. The GROUP BY groups the tuples based on countries and the query finds out how many partners are there from each country in this project.

Query Example 3

In this example, we show another join query between the "smartdatalake" and “h2020” relations. In this query we want to find number of people per country in "smartdatalake".

```
SELECT h.country, Count(p)
FROM   smartdatalake s, Unnest(people) p, h2020 h
WHERE  s.partner = h.shortname
GROUP  BY h.country
```

Similar to example 2, this query also does an equijoin between the two datasets and does a group by on country. However, since we need to find the number of people which is an attribute of the list type, we first unnest the list and then do the count on the unnested list.

5. Conclusion

In data virtualization layer of SmartDataLake, we have built a query engine that can run queries across heterogeneous data formats and heterogeneous data sources over heterogeneous hardware. The users do not need to be aware of the heterogeneities when running the queries unlike traditional data warehouse systems where users need to perform expensive ETL before they can run any query. We have integrated the RAW and Proteus systems to provide these functionalities. In future, we are working on elastic scaling out of the query engine and on a storage manager to automatically manage storage tiering.

References

[1] M. Karpathiotakis, I. Alagiannis, A. Ailamaki: Fast Queries Over Heterogeneous Data Through Engine Customization. PVLDB 9(12): 972-983 (2016)

[2] P. Chrysogelos, M. Karpathiotakis, R. Appuswamy, Anastasia Ailamaki: HetExchange: Encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines. PVLDB 12(5): 544-556 (2019)

[3] L. Fegaras and D. Maier. Optimizing object queries using an effective calculus.ACM Trans. Database Syst., 25(4):457–516, 2000.

[4] The LLVM Compiler Infrastructure. https://llvm.org/