QUIS: In-Situ Heterogeneous Data Source Querying

Javad Chamanara[†] Birgitta König-Ries[†] H. V. Jagadish[*]

[†]Heinz-Nixdorf-Chair for Distributed Information Systems
Friedrich Schiller University of Jena
Jena, TH, Germany
{first.last}@uni-jena.de

[*]Computer Science and Engineering
University of Michigan
Ann Arbor, MI, USA
{jag}@umich.edu

ABSTRACT

Existing data integration frameworks are poorly suited for the special requirements of scientists. To answer a specific research question, often, excerpts of data from different sources need to be integrated. The relevant parts and the set of underlying sources may differ from query to query. The analyses also oftentimes involve frequently changing data and exploratory querying. Additionally, The data sources not only store data in different formats, but also provide inconsistent data access functionality. The classic Extract-Transform-Load (ETL) approach seems too complex and time-consuming and does not fit well with interest and expertise of the scientists.

With QUIS (QUery In-Situ), we provide a solution for this problem. QUIS is an open source heterogeneous in-situ data querying system. It utilizes a federated query virtualization approach that is built upon plugged-in adapters. QUIS takes a user query and transforms appropriate portions of it into the corresponding computation model on individual data sources and executes it. It complements the segments of the query that the target data sources can not execute. Hence, it guarantees full syntax and semantic support for its language on all data sources. QUIS's in-situ querying facility almost eliminates the time to prepare the data while maintaining a competitive performance and steady scalability.

The present demonstration illustrates interesting features of the system: virtual schemas, heterogeneous joins, and visual query results. We provide a realistic data processing scenario to examine the system's features. Users can interact with QUIS using its desktop workbench, command line interface, or from any R client including RStudio Server.

1. INTRODUCTION

Data heterogeneity is increasing in all aspects, faster than ever [1]. Data is stored in different representations, with various levels of schema, and changes at different paces. Additionally, software systems to manage and process such data are incompatible, incomplete, and diverse.

Proceedings of the VLDB Endowment, Vol. 10, No. 12
Copyright 2017 VLDB Endowment 2150-8097/17/08.

Data scientists often have to integrate data from heterogeneous sources [6] to conduct an end-to-end process to obtain insights. They usually do not need the whole set of available data, but their required portion of data changes over the course of their research. This exploratory nature prevents the scientists from deciding on the data schema, tool set, and pipeline at early stages of their research.

The two classical approaches to data integration, i.e., materialized and virtual integration [5], do not solve scientific data management and processing problems. Both aim to provide somewhat complete integration of information. The underlying assumption is that it is worthwhile to invest significant effort in preparing a long-term information system that is able to answer a wide range of queries. Further specifics that make materialized integration difficult is the volatile nature of the research data and the often large data volume or rigid access rights that prevent transferring the data. Virtual integration is unsuitable because of the typical lack of optimization for non-relational sources.

In contrast, in our scenario, scientists often have a narrow set of queries they want to ask and tend to do just enough integration according to their research questions only.

The fundamental difficulty is that the data is heterogeneous not only in syntax and structure, but also in the way it is accessed and queried. While certain data may be accessed by declarative queries, others are processed by MapReduce programs utilizing a procedural computation model [4]. Furthermore, many sensor-generated datasets are in CSV files that lack basic data management features. We recognize this as *data access heterogeneity.*

Data access heterogeneity covers varieties in computational models (e.g., procedural, declarative), querying capabilities, syntax and semantics of the capabilities provided by different vendors or systems, data types, and presentation formats of the query results.

One critical aspect of data access heterogeneity is the heterogeneous capabilities of data sources. Some data sources, e.g., relational, graph, and arrays, have their own set of management systems. Others fall under the so called *weak data sources* [7], CSV and spreadsheets for example, and do not have a well-established management system. Also, not all management systems support the capabilities requested by users' queries, e.g., the MapReduce programming model does not support joins and sorting [8].

We need tools to reduce the total cost of ownership of the full data life-cycle, from raw data to insight. And we need to democratize them; by developing open source, interoperable, and easy to use systems.

We have developed QUIS (QUery In-Situ) to overcome the data access heterogeneity problem. QUIS is an agile query system equipped with a unified query language and a federated execution engine that provides advanced features such as virtual schemas, heterogeneous joins, polymorphic result set presentation, and in-situ data querying. QUIS transforms a given input query written in its language to a (set of) computation models that are executed on the designated data sources. QUIS guarantees that input queries are always fully satisfied. Therefore, if the target data sources do not fulfill all the query requirements, QUIS detects the lacking features and complements them transparently.

Our experiments have proven that i) QUIS dramatically reduces the time-to-first-query, ii) its rule-based query optimizer remarkably boosts performance, and iii) the overall performance scales linearly as the datasets' size grow.

In the rest of this paper, we give an overview of the system's main architectural components in Section 2 and then describe the proposed demonstration in Section 3.

2. SYSTEM ARCHITECTURE

QUIS consists of three main components; a query language (Section 2.1), a set of adapters (Section 2.2), and a query execution engine (Section 2.3). Users write their queries and applications using the query language and submit them to the execution engine. The engine selects the best adapter to *transform* and *execute* each of the queries. It may optimize the queries before shipping them to the adapters (Section 2.4). Also, it may rewrite the input queries to complement any lacking features of the chosen adapters. Query execution is an orchestration between these two endpoints; the (abstract) query language and the (concrete) capabilities of underlying sources accessible via adapters.

2.1 Query Language

Our query language is an extension to the SQL core. Its two main elements are *declarations* and *statements*. Statements are the units of execution. Declarations are non-executable definitions or configuration items used by statements. There are three types of declarations; *connections*, *bindings*, and *perspectives*.

A *connection* encapsulates the information needed to connect to the data source. For example, a server endpoint and credentials are required to connect to an RDBMS. QUIS supports querying specific versions of data, e.g., to support reproducibility [2]. A *binding* is used to associate queries to a specific version of the target data. It also restricts data reachable via the bound connection.

Perspectives allow users to explicitly specify the schema of the query results in term of *attributes*. Attributes are expressed as transformations of the physical data items of the sources. Perspectives differ from RDBMS views as they formulate projection and transformation only, but not selection. Perspectives are not materialized. In contrast, they support inheritance and overriding.

QUIS retrieves and manipulates data via statements, which are based on tuple relational calculus and offer query operators [3]. QUIS supports polymorphic presentation, in that query results can be delivered in many ways, e.g., tabular, visual, or serialized. It also has a virtual type system that manages type matching, inference, and consolidation.

Listing 1: QUIS query to extract, transform, aggregate, and draw a mean daily temperature chart from a CSV dataset.

```
1 CONNECTION cnn1 ADAPTER=CSV SOURCE_URI=
    "data/" PARAMETERS=delimiter:comma,
    fileExtension:csv, firstRowIsHeader:
    true, externalHeader:true
2 BIND b1 CONNECTION=cnn1 SCOPE=fso2014h
    VERSION=Latest
3 SELECT FROM b1.0
    USING avg(temperatureC) AS meanTemp,
    dt.dayOfYear(dateUTC) as dayIndex
    INTO PLOT fsoPlot hAx:dayIndex vAx:
    meanTemp plotType:line hLabel:"Day of
    Year" vLabel:"Mean Temperature (C)"
    plotLabel:"Daily Mean Temperature at
    SFO in 2014"
    ORDER BY dayIndex
```

An example of a QUIS script is presented in Listing 1. It declares a connection to a CSV file (line 1) and binds it to the latest version of the data (line 2). The query (line 3) retrieves data and computes the average of temperature implicitly grouped by the day of year. The result set is ordered and visualized as a line chart as shown in Figure 1.

2.2 Adapters

An adapter is a data-source-specific plug-in that plays two roles. First, it translates a given input query to a computation model native to the underlying data source(s). Second, it executes the computation model on the designated data source(s) to build a result set. The computation model is either a set of queries in the data sources' languages or a sequence of operations. For example, an RDBMS adapter translates the input query to a vendor-specific SQL, while the CSV adapter generates a set of functions to read, parse, materialize, and filter the records.

QUIS allows adapters with partial language support. But at minimum, operators to retrieve tuples from and insert tuples to the underlying data sources must be provided. Adapters may wrap *dialects* of similar data sources, e.g., PostgreSQL and MySQL.

Currently, CSV, MS Excel, RDBMSs, and in-memory adapters are available. One to operate on big data using Hadoop is under development. Developing a full-fledged adapter takes up to two months for an experienced Java programmer.

2.3 Query Execution Engine

QUIS's query execution engine orchestrates all the activities needed to compute and assemble the result set of a submitted query. Upon submission of a query, the engine parses, validates, builds its syntax tree. It selects an adapter to transform and execute the query, more adapters are chosen if query accesses heterogeneous data sources. The query engine then triggers its optimizer to apply all the relevant optimization rules. Afterwards, the optimized query is shipped to the chosen adapters for transformation. The query engine compiles the transformations on-the-fly to a set of executable jobs, which are then dispatched for execution. Finally, the query's result set is presented to the client in the requested form, e.g., tabular, visual, or serialized.

Although the adapters can expose partial support for the language, the query engine promises full execution. To keep this promise it needs to perform two important tasks; selecting the best available adapter and complementing the features that the selected adapter does not provide.

Assuming R is the set of features required by an input query q and C is the set of features exposed by adapter a, then a supports $Q = \{R \cap C\}$ only. The reminder of the features $P = \{R \setminus C\}$ are handled by a complementing algorithm. The algorithm builds two queries to fulfill q. Query q_a to execute Q on a and query q_f to execute P on a special fallback adapter f. At runtime, the engine executes q_a and passes its result to q_f. The result set of q_f is equivalent to the result of q.

QUIS computes the overall execution cost of all the available adapters and selects the one with minimum cost. The overall cost is the summation of the cost of executing Q on the target adapter and P on the fallback. Each adapter announces the cost of executing each feature it supports. Currently each adapter developer assigns the costs statically.

2.4 Query Optimization

QUIS has an integrated rule-based optimizer that benefits from well-known and custom designed optimization techniques. The well-known techniques are adopted to heterogeneous and in-situ environments. For example, its *selective materialization* rule determines which data items of the target data source are actually used by the query and ensures that only those ones are loaded.

Its *push ahead selection* rule changes the query plan to test the selection's predicate first and materialize the record only if the test passes. The query engine applies this rule even on non-relational data sources.

We have implemented two special optimization rules that although experimental, have shown to be effective. *Running Aggregate Computation* calculates an aggregate function $agg(V)$ on a set of values: $V = \{v_1..v_{n+1}\}$ using the running method $agg_{n+1} = f(agg_n, v_{n+1})$. For example, $average_{n+1} = (sum_n + value_{n+1})/(n+1)$. It only needs to keep a small state object in memory, hence $O(1)$ space.

Weighted Short-Circuit Evaluation assigns a cost factor to functions and operators and builds a weighted evaluation tree for logical expressions. It then evaluates cheaper paths earlier and stops as soon as the expression result is determined. In a predicate $p \Rightarrow (f1 \wedge f2) \ni cost(f1) = 5cost(f2)$, $f2$ is evaluated first. Conventional evaluation would evaluate $f1$ first.

In essence, the system architecture provides a streamlined query virtualization that encourages raw and ad-hoc data processing, reduces the need to utilize multiple languages and tools to deal with data, and promotes query portability.

3. DEMONSTRATION

QUIS is an open source software capable of querying local, distributed, and diverse data that are managed by systems with access heterogeneity. It best fits in scenarios such as ad-hoc and agile querying, early insight, data, tool and workflow integration. For the present demonstration we focus on these features: (1) perspectives, (2) heterogeneous joins, and (3) query result visualization. Virtual schemas isolate the processing part from the data formatting, conversion, and type system parts. Heterogeneous joins are simple ways to retrieve data from various sources and combine them transparently without needing the data to be loaded to an intermediate system. Query result visualization provides the user with early insight into data. We have prepared the required tools and data for the audience to participate in the demonstration and afterwards.

3.1 Demonstration Scenario

In order to demonstrate the features of the system, we write the required queries to perform the following task: *Retrieve airport names, location, and average temperature per airport over the full timespan of the weather records. Then visually check if there is a relationship between the airports' elevation and the average temperature. Select the airport with highest average.*

The task requires various QUIS capabilities, e.g., filtering, joining, aggregation, visualization, and in-situ querying.

3.2 Data

We have a heterogeneous dataset that consists of three different data sources. (1) *Meteorological data:* a remote PostgreSQL database table of 5 years of hourly weather records collected from the stations at around 200 different airport meteorological stations. The table contains approximately 10M records. Each record contains temperature ($°C$), humidity (%), wind speed (m/s), timestamp, and station id. (2) *Airport information:* a CSV file containing the list of airports. Each record contains id, code, and name of an airport. (3) *Airport location:* an MS Excel file that its first sheet contains the geographical location of the airports. Records consist of station id, latitude(°), longitude(°), and elevation (m). The exact schemas, connection information, and file locations will be provided at the demo session.

3.3 Tools

We have developed three clients to interact with QUIS[1].

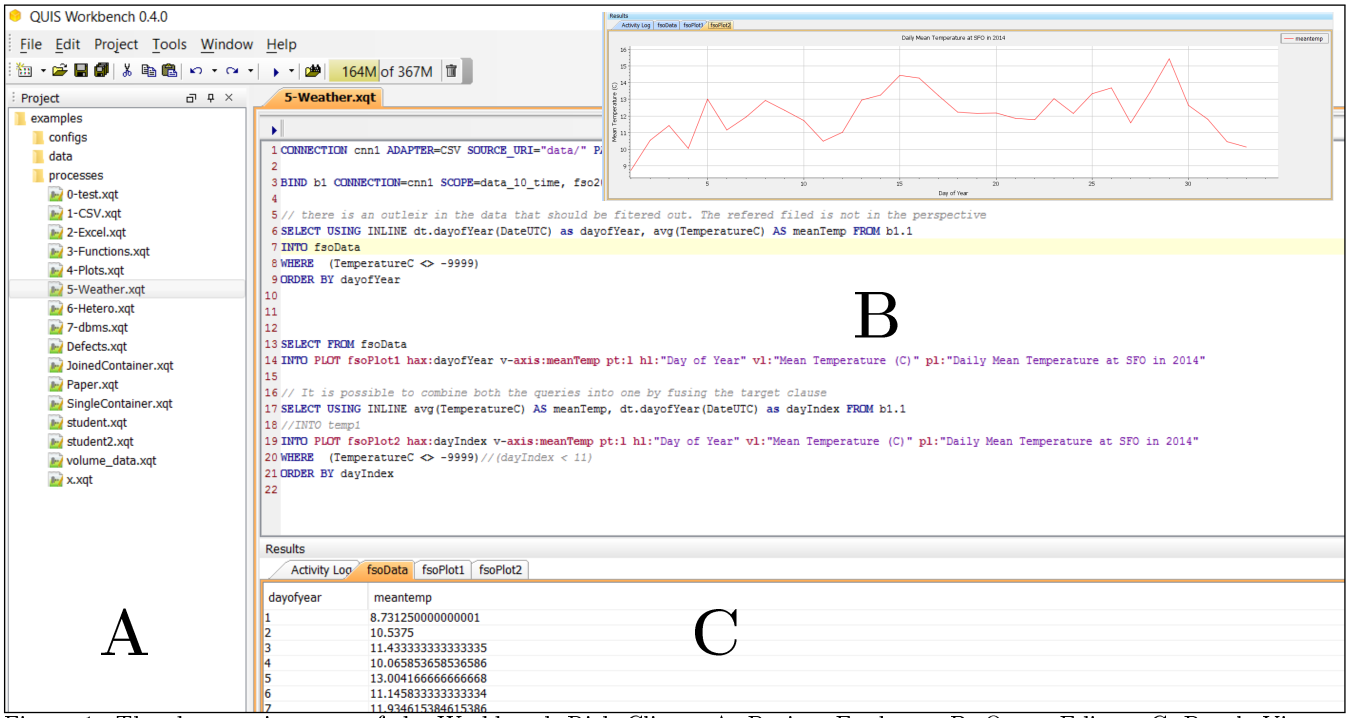
***WRC*:** (Workbench Rich Client) is QUIS's built-in GUI based client. Its main screen is shown in Figure 1.

In *WRC* queries are organized in files, and files in projects. Projects are loaded to and managed in the *project explorer* **(pane A)**. The *query editor* **(pane B)** is a multi-tab area that allows multiple query authoring and parallel execution. Each query editor has its own *result viewer* **(pane C)**, where each query result is shown individually. An execution summary is also presented. Query results can be visualized as single or multi-series bar, line, scatter, or pie charts. An exemplary line chart that illustrates the output of Listing 1 is shown as an overlay on top-right corner of Figure 1.

***CLI*:** (Command Line Interface) is a terminal based client for QUIS that accepts queries in files and persists the results in files as well. It can be integrated into the host operating systems' shell scripting as well as other tools and workflow management systems. It is possible to trigger *CLI* with a customized JVM configuration, e.g., to allocate heap memory in advance for operating on large datasets.

***RQUIS*:** is an R package that offers the full functionality of QUIS to R users. Users can write queries inline with R or in separated *.xqt* files. The query results and their schema can be obtained as R data frames by calling $quis.getVariable()$ and $quis.getVariableSchema()$, respectively. Figure 2 shows an R script that utilizes RQUIS and *ggplot2* packages to retrieve and visualize data, respectively.

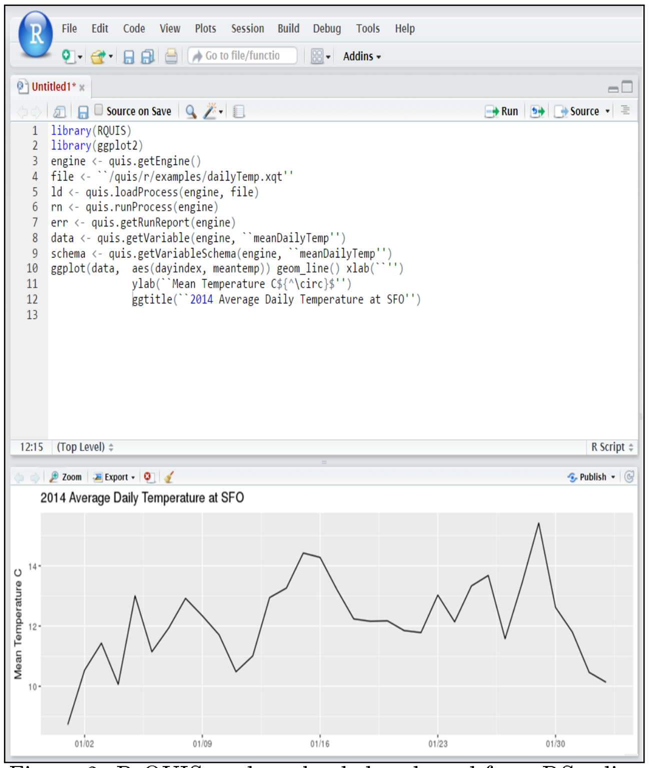
[1]`http://fusion.cs.uni-jena.de/projects/quis/`



```
CONNECTION cnn1 ADAPTER=CSV SOURCE_URI="data/" PA

BIND b1 CONNECTION=cnn1 SCOPE=data_10_time, fso2

// there is an outleir in the data that should be fitered out. The refered filed is not in the perspective
SELECT USING INLINE dt.dayofYear(DateUTC) as dayofYear, avg(TemperatureC) AS meanTemp FROM b1.1
INTO fsoData
WHERE  (TemperatureC <> -9999)
ORDER BY dayofYear



SELECT FROM fsoData
INTO PLOT fsoPlot1 hax:dayofYear v-axis:meanTemp pt:l hl:"Day of Year" vl:"Mean Temperature (C)" pl:"Daily Mean Temperature at SFO in 2014"

// It is possible to combine both the queries into one by fusing the target clause
SELECT USING INLINE avg(TemperatureC) AS meanTemp, dt.dayofYear(DateUTC) as dayIndex FROM b1.1
//INTO temp1
INTO PLOT fsoPlot2 hax:dayIndex v-axis:meanTemp pt:l hl:"Day of Year" vl:"Mean Temperature (C)" pl:"Daily Mean Temperature at SFO in 2014"
WHERE  (TemperatureC <> -9999)//(dayIndex < 11)
ORDER BY dayIndex

```

dayofyear	meantemp
1	8.731250000000001
2	10.5375
3	11.433333333333335
4	10.065853658536586
5	13.004166666666668
6	11.145833333333334
7	11.934615384615386

Figure 1: The three main panes of the Workbench Rich Client. A: Project Explorer. B: Query Editor. C: Result Viewer. The overlay chart depicts the visual representation of a query result.



```
library(RQUIS)
library(ggplot2)
engine <- quis.getEngine()
file <- ``/quis/r/examples/dailyTemp.xqt''
ld <- quis.loadProcess(engine, file)
rn <- quis.runProcess(engine)
err <- quis.getRunReport(engine)
data <- quis.getVariable(engine, ``meanDailyTemp'')
schema <- quis.getVariableSchema(engine, ``meanDailyTemp'')
ggplot(data,  aes(dayindex, meantemp)) geom_line() xlab(``'')
              ylab(``Mean Temperature C${^\circ}$'')
              ggtitle(``2014 Average Daily Temperature at SFO'')

```

Figure 2: R-QUIS package loaded and used from RStudio.

It has been executed on an RStudio server on a docker container on the cloud.

In summary, RQUIS provides unified data querying syntax for R users, eliminates the need to load the data, and performs all the query operators directly on the raw data.

Acknowledgments

This work was supported by DFG (German Research Foundation) in the BExIS++ project.

4. REFERENCES

[1] D. Abadi et al. The beckman report on database research. *Commun. ACM*, 2016.

[2] C. L. Borgman. Research data: Who will share what, with whom, when, and why? *China-North America Library Conference*, 2010.

[3] E. F. Codd. A relational model of data for large shared data banks. *Comm. of the ACM*, 13(6):377–387, 1970.

[4] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *ACM*, 2008.

[5] A. Doan, A. Halevy, and Z. Ives. *Principles of data integration*. Elsevier, 2012.

[6] B. Ludäscher et al. Managing scientific data: From data integration to scientific workflows. *Geological Society of America Special Papers*, 2006.

[7] A. Tomasic, L. Raschid, and P. Valduriez. Scaling access to heterogeneous data sources with DISCO. *IEEE Trans. on Knowledge and Data Engg.*, 1998.

[8] J. Yin, Y. Liao, M. Baldi, L. Gao, and A. Nucci. Efficient Analytics on Ordered Datasets Using MapReduce. 2013.