FRIEDRICH-SCHILLER-
UNIVERSITÄT
JENA

Querying Heterogeneous Data in an In-situ Unified Agile System

Dissertation

zur Erlangung des akademischen Grades
Doktor-Ingenieur (Dr.-Ing.)

vorgelegt dem Rat der Fakultät für Mathematik und Informatik
der Friedrich-Schiller-Universität Jena

von M.Sc. Javad Chamanara
geboren am 23.10.1972 in Eilam

Gutachter

1. **Prof. Dr. Birgitta König-Ries**
   Friedrich-Schiller-Universität Jena, 07743 Jena, Thüringen, Deutschland
2. **Wird vom Fakultätsrat bekannt gegeben Prof. Dr. H. V. Jagadish**
   University of Michigan, 48109-2121 Ann Arbor, Michigan, USA
3. **Wird vom Fakultätsrat bekannt gegeben Prof. Dr. Klaus Meyer-Wegener**
   Friedrich-Alexander-Universität, 91058 Erlangen, Bayern, Deutschland

Tag der öffentlichen Verteidigung: 12. APRIL 2018

Ehrenwörtliche Erklärung

Hiermit erkläre ich,

- dass mir die Promotionsordnung der Fakultät bekannt ist,
- dass ich die Dissertation selbst angefertigt habe, keine Textabschnitte oder Ergebnisse eines Dritten oder eigenen Prüfungsarbeiten ohne Kennzeichnung übernommen und alle von mir benutzten Hilfsmittel, persönliche Mitteilungen und Quellen in meiner Arbeit angegeben habe,
- dass ich die Hilfe eines Promotionsberaters nicht in Anspruch genommen habe und dass Dritte weder unmittelbar noch mittelbar geldwerte Leistungen von mir für Arbeiten erhalten haben, die im Zusammenhang mit dem Inhalt der vorgelegten Dissertation stehen,
- dass ich die Dissertation noch nicht als Prüfungsarbeit für eine staatliche oder andere wissenschaftliche Prüfung eingereicht habe.

Bei der Auswahl und Auswertung des Materials sowie bei der Herstellung des Manuskripts haben mich folgende Personen unterstützt:

- Prof. Dr. Birgitta König-Ries

Ich habe die gleiche, eine in wesentlichen Teilen ähnliche bzw. eine andere Abhandlung bereits bei einer anderen Hochschule als Dissertation eingereicht: Ja / Nein.

Jena, den 12. April 2018

[Javad Chamanara]

To Diana

Deutsche Zusammenfassung

Die Datenheterogenität wächst in allen Aspekten viel rasanter als je zuvor. Daten werden auf verschiedene Art und Weise in vielfältigen Formaten und wechselnden Geschwindigkeiten gespeichert. Softwaresysteme, die diese Daten verarbeiten und verwalten, sind zudem inkompatibel, unvollständig und vielfältig. Datenwissenschaftler müssen oft Daten aus heterogenen Quellen integrieren, um einen Ende-zu-Ende Prozess aufzusetzen und durchführen zu können, der ihnen zu neuen Erkenntnissen verhilft. Zum Beispiel kommt es vor, dass Wissenschaftler Sensordaten aus einer CSV-Datei mit Simulationsergebnissen aus MATLAB-Dateien, Beobachtungsdaten in Excel-Dateien und Referenzdaten aus einer relationalen Datenbank miteinander kombinieren müssen. Diese Daten werden von einer Vielzahl von Werkzeugen und unterschiedlichen Personen für verschiedene Zwecke produziert. In der Regel benötigt der Wissenschaftler nicht alle verfügbaren Informationen aus den Dateien, jedoch ändert sich die erforderliche Datenauswahl über den Forschungszeitraum. Zudem haben Wissenschaftler oft nur eine begrenzte Anzahl an Forschungsfragen und neigen daher dazu, gerade so viele Daten zu integrieren, wie nötig sind um diese Fragen zu beantworten. Ihre Analyse umfasst oft wechselnde Anfragen und volatile Daten in denen sich zum Beispiel die Struktur häufig verändert. Aufgrund dieser Gegebenheiten können Wissenschaftler nicht zu Beginn ihrer Forschungstätigkeit entscheiden, welches Datenschema, welche Werkzeuge und welche Abläufe sie verwenden. Stattdessen würden sie lieber verschiedene Werkzeuge nutzen, um iterativ Daten zu verbinden und zu integrieren. So könnte ein geeignetes Datenschema nur mit den relevanten Teildaten geformt werden. Dieser Prozess generiert eine Vielzahl von ad-hoc ETL-Operationen (Extraction-Transformation-Load), die es erfordern häufig Daten zu integrieren.

Datenintegration stellt eine vereinheitlichte Sicht durch Verknüpfung von Daten aus verschiedenen Quellen dar. Sie beschäftigt sich mit Herausforderungen in Bezug auf die Heterogenität in der Syntax, Struktur und Semantik von Daten. In heutigen multidisziplinären und kollaborativen Forschungsumgebungen werden Daten funktionionsübergreifend produziert und konsumiert. Mit anderen Worten, zahlreiche Forscher verarbeiten Daten in verschiedenen Disziplinen um vielfältige Forschungsumgebungen mit unterschiedlichen Messauflösungen und mehreren Anfrageprozessoren und Analysewerkzeuge zu bedienen. Diese funktionsübergreifenden Datenoperationen sind ein wesentlicher Bestandteil jeder erfolgreichen datenintensiven Forschungsaktivität.

Die zwei klassischen Ansätze in der Datenintegration, die materialisierte und virtuelle Integration, lösen nicht die oben beschriebenen Probleme im Datenmanagement und in der Datenverarbeitung. Beide zielen darauf ab Informationen vollständig zu integrieren. Die Annahme hier ist, dass es sich lohnt, erhebliche Anstrengungen in die Bereitstellung von Langzeit-Informationssystemen zu investieren, die in der Lage sind, eine große Bandbreite an Anfragen zu beantworten. Weitere Faktoren, die die materialisierte Integration erschweren, sind die unbeständige Natur von Forschungsdaten und die oft großen Datenmengen oder die starren Zugangsbestimmungen, die die Datenweitergabe verhindern. Virtuelle Integration ist nicht geeignet, da hier Optimierungsmöglichkeiten für nicht-relationale Datenquellen fehlen.

Die grundlegende Schwierigkeit ist, dass die Daten nicht nur in Syntax, Struktur und Semantik heterogen sind, sondern auch in der Art und Weise wie auf sie zugegriffen wird. Während sich

zum Beispiel bestimmte Daten in funktionsreichen, relationalen Datenbanken befinden, auf die mittels deklarativer Anfragen zugegriffen wird, werden andere durch MapReduce-Programme unter Verwendung prozeduraler Berechnungsmodelle verarbeitet. Des Weiteren werden viele sensorgenerierte Daten in CSV-Dateien verwaltet, ohne dass auf die Verwendung etablierter Formatierungsstandards und grundlegender Datenmanagement-Funktionen zurück gegriffen wird. Sogar verschiedene relationale Datenbanksysteme unterscheiden sich in Syntax, Einhaltung von SQL-Standards und Funktionsumfang. Wir bezeichnen das als *Datenzugriffs-Heterogenität*.

Datenzugriffs-Heterogenität bezieht sich auf Unterschiede im Hinblick auf Berechnungsmodelle (z.B. prozedural, deklarativ), Abfragemöglichkeiten, Syntax und Semantik der durch verschiedene Hersteller oder Systeme bereitgestellten Funktionalität. Weiterhin schließt dies auch die Datentypen und Formate ein, in denen Daten und Abfrageergebnisse zurück geliefert werden. Ein kritischer Aspekt der Datenzugriffs-Heterogenität sind die Unterschiede im Funktionsumfang verschiedener Datenquellen. Während einige Datenquellen, wie zum Beispiel relationale, graphbasierte und Array-Datenbanken, als starke Datenquellen klassifiziert werden, haben schwache Datenquellen, wie zum Beispiel Tabellenkalkulationen kein bewährtes Datenmanagement. Zudem unterstützen nicht alle Datenmanagementsysteme die Funktionen, die von den Anwendern gefordert werden. Daher sollten wir in die Liste der Heterogenität das Interesse von Datenwissenschaftlern an Berechnungen mit Rohdaten aufnehmen. Des Weiteren müssen domänenspezifische Werkzeuge, die Datenwissenschaftler in der Forschung verwenden, ebenso hinzugefügt werden, wie eine Vielzahl an Werkzeugen und Sprachen, die benötigt werden, um Datenanalyse-Aufgaben zu bewältigen.

In dieser Arbeit identifizieren wir den Bedarf an besseren Werkzeugen, um die Gesamtkosten über den vollständigen Datenlebenszyklus von Rohdaten zu Forschungserkenntnissen zu minimieren. Wir argumentieren außerdem, dass diese Werkzeuge demokratisiert werden sollten durch quelloffene, interoperable und leicht bedienbare Systeme. Im Gegensatz zu relationalen Datenbanksystemen, die Funktionen zur Speicherung und Abfrage auf entsprechende Datenmodelle bereitstellen, schlagen wir ein agiles Datenabfragesystem vor. Das ist ein ausdrucksstarkes Datenabruf-System, das nicht an die Datenspeicherung oder darunterliegenden Datenmanagementsysteme mit begrenztem Funktionsumfang gebunden ist. Das Ziel eines solchen Systems sind schnelle Rückmeldungen für Anwender und die Vermeidung der Vervielfältigung von Daten, während gleichzeitig ein vereinheitlichtes Abfrage-und Berechnungsmodell zur Verfügung gestellt wird.

In dieser Arbeit stellen wir `QUIS` (Query In-Situ) vor. `QUIS` ist ein agiles Abfragesystem, ausgestattet mit einer vereinheitlichten Abfragesprache und einer föderierten Ausführungseinheit, die in der Lage ist Abfragen an Ort und Stelle über heterogene Daten auszuführen. Seine Sprache erweitert SQL um Funktionalitäten wie virtuelle Schemas, heterogene Verknüpfungen und polymorphe Präsentationen der Ergebnisse. `QUIS` nutzt das Konzept der Abfrage-Virtualisierung, das auf einem Verbund von Agenten basiert, um eine gegebene Anfrage in einer bestimmten Sprache in ein Berechnungsmodell zu überführen, das sich auf den zugewiesenen Datenquellen ausführen lässt. Während die verteilte Anfrage-Virtualisierung viel größere Flexibilität und Unterstützung von Heterogenität ermöglicht als die zentralisierte Virtualisierung von Daten, hat

sie jedoch den Nachteil, dass einige Teile der Anfrage nicht immer von den zugewiesenen Datenquellen unterstützt werden und dass das Abfragesystem dann als Sicherung agieren muss, um diese Fälle zu komplementieren. QUIS garantiert, dass die Anfragen immer komplett ausgeführt werden. Wenn die Zielquelle die Anforderungen an die Abfrage nicht erfüllt, identifiziert QUIS fehlende Funktionalitäten und ergänzt diese transparent. QUIS bietet Union- und Join-Funktionen über eine unbegrenzte Liste von heterogenen Datenquellen an. Zusätzlich bietet es Lösungen für heterogene Anfrageplanungen und Optimierung an. Zusammengefasst, QUIS zielt darauf ab, die Datenzugriffs-Heterogenität durch Virtualisierung, on-the-fly Transformationen und föderierte Ausführungen abzumildern und stellt dabei folgende Neuerungen bereit:

1. **In-Situ querying:** QUIS transformiert Abfragen in eine Anzahl ausführbarer Jobs, die in der Lage sind auf Rohdaten zuzugreifen und zu verarbeiten ohne sie zunächst in ein Zwischensystem laden oder duplizieren zu müssen;

2. **Agile querying:** QUIS ist ein Abfragesystem und keine relationale Datenbank. Es ermöglicht und fördert häufige ad-hoc Abfragen mit zeitnahen Rückmeldungen;

3. **Heterogeneous data source querying:** QUIS ist in der Lage Abfragen, die mehrere heterogene Datenquellen beinhalten, sowohl darzustellen als auch auszuführen; zusätzlich können Operationen wie Join und Union über mehrere Datenquellen hinweg umgesetzt werden;

4. **Unified execution:** QUIS garantiert Anfragen auszuführen. Es erkennt fehlende Funktionen, die benötigt werden, und ergänzt sie, wenn sie nicht durch die zugewiesenen Datenquellen unterstützt werden;

5. **Late-bound virtual schemas:** QUIS ermöglicht die Deklaration von virtuellen Schemas, die gemeinsam mit der Abfrage gesendet werden können. Diese Schemas haben einen ähnlichen Lebenszyklus wie die Abfragen und brauchen daher nicht vorher definiert oder installiert zu werden; und

6. **Remote execution:** QUIS Abfragen sind in selbstausführende Einheiten kompiliert, die auf entfernte Datenzentren transferiert werden, um direkt auf den Daten ausgeführt werden zu können.

Durch eine Anforderungsanalyse identifizieren wir in dieser Dissertation die Problematik und stellen detailliert da. Weiterhin zeigen wir einen Lösungsansatz auf, um die aufgezeigten Anforderungen zu erfüllen. Die Lösung umfasst eine föderierte Architektur, die aus drei Hauptteilen besteht: Abfragedeklaration, Abfragetransformation und Abfrageausführung. Die Abfragedeklaration stellt sowohl ein System zur Erstellung von Anfragen bereit, als auch für Tokenisierung, Parsen und Validierung. Außerdem konvertiert es die Abfragen in ein internes Abfragemodell, das sich leichter von anderen Komponenten verarbeiten lässt. Abfragetransformation umfasst alle Funktionen, die für die Konstruktion von nativen Berechnungsmodellen benötigt werden und die auf den zugewiesenen Zielquellen ausführbar sind. Abfrageoptimierung wird ebenfalls unterstützt. Die Abfrageausführung zielt darauf ab, aus den transformierten Berechnungsmodellen

ausführbare Einheiten zu generieren und deren Ausführung auf den Zielquellen zu orchestrieren. Weitere Aufgaben dieser Komponente sind das Sammeln, Formatieren und Anzeigen der Abfrageergebnisse. Die Darstellung der Ergebnisse kann dabei auch Visualisierungen und den Austausch von Daten zwischen einzelnen Prozessen umfassen.

Wir stellen eine prototypische Umsetzung zur Verfügung um zu zeigen, dass die vorgeschlagene Lösung praktikabel ist. Obwohl die Implementierung nicht alle Funktionalitäten zu gleichen Teilen abdeckt und vielleicht nicht den optimalen Ansatz darstellt, solche Funktionen zu implementieren, haben wir den Prototyp intensiv evaluiert, um die Effektivität und Effizienz nachzuweisen. Die Dissertation schließt mit einer Diskussion und einem Ausblick auf zukünftige Arbeiten.

Acknowledgments

I would like to express my special appreciation and thanks to my advisor Prof. Dr. Birgitta König-Ries. She believed in my abilities and provided the financial, administrative, and technical infrastructure that were required to accomplish this work. She created an atmosphere that allowed me to frequently obtain her comments and arguments, yet decide independently. I would also like to thank her for her patience and tolerance regarding academic, social, and cultural varieties. I enjoyed it! My thanks also go to Prof. Dr. H. V. Jagadish and Prof. Dr. Klaus Meyer-Wegener, who served as external reviewers for my dissertation. I would like to thank them for the time and effort they devoted to review and evaluate this work.

I appreciate the support I received from Martin Huhmuth and Andreas Ostrowski. They provided me with the software and hardware support I needed for the evaluations I have done in this work. Also, I am thankful for the support I received from Jitendra Gaikwad with interviews and summarization of his work as one of my motivational examples. Additionally, I would like to express my appreciation to Friederike Klan, Sirko Schindler, Felicitas Löffler, and Alseysed Algergawy for their help, advice, and comments on papers, and sections of this work. Vahid Chamanara, my younger brother, helped me with the statistical analysis of the survey results. Thanks Vahid.

I'm also thankful to all the people that contributed to the advancement of this thesis. I would like to particularly thank H. V. Jagadish and Barzan Mozafari from the university of Michigan for giving critical comments and valuable insights. I am grateful for the Mark Schildhauer, Matthew Jones, and Rob DeLine's contributions to the design of the language. They were valuable sources of requirements and features. I am contented and cheerful with the willingness and enthusiasm of those who volunteered as test subjects for the user study I conducted as a part of this work's evaluation.

I'm in dept of gratitude to my parents who, regardless of our geographical distance, continuously supported and encouraged me wherever they could. Thank you! I dedicate this work to my daughter Diana, a source of unending joy and love. Although a kid, she has always been smart, happy, progressing, and sympathetic. She has been wonderfully understanding throughout the process of developing this dissertation. I deeply enjoy the moments we live together, and wish her a bright future.

Abstract

Data heterogeneity, in all aspects, is increasing more rapidly than ever. Data is stored using different forms of representation, with various levels of schema, and at different changing paces. In addition, the software systems used to manage and process such data are incompatible, incomplete, and diverse. Data scientists are frequently required to integrate data obtained from heterogeneous sources in order to conduct an end-to-end process intended to provide insights: For example, a scientist may need to combine sensor data contained in a CSV file with simulation outputs that are stored in the form of a MATLAB mat-file, an Excel file containing field observations, and reference data stored in a relational database. Such data can be produced by different tools and individuals for various purposes. Scientists do not usually require entire sets of available data; however, the portions of data that they require typically change over the course of their research. Also, they often have a narrow set of queries that they want to ask and tend to perform just enough integration according to their research questions only. Their analyses often involve volatile data (i.e., data and/or its structure change frequently) and exploratory querying. These factors prevent the scientists from deciding on the data schema, tool set, and processing pipeline at early stages of their research. Instead, they would use various tools to iteratively merge and integrate data to build an appropriate schema and select the relevant portions of the data. This process creates a loop of ad-hoc ETL operations that requires the scientists to frequently perform data integration.

Data integration provides a unified view of data by combining different data sources. It deals with challenges regarding heterogeneity in the syntax, structure, and semantics of data. In today's multi-disciplinary and collaborative research environments, data is often produced and consumed in a cross-functional manner; in other words, multiple researchers operate on data in different divisions or disciplines in order to satisfy various research requirements, at diverse measurement resolutions, and using different query processors and analysis tools. These cross-functional data operations make data integration a crucial component of any successful data intensive research activity.

The two classical approaches to data integration, i.e., materialized and virtual integration, do not solve the problem encountered in scientific data management and processing. Both aim to provide somewhat complete integration of information. The underlying assumption is that it would be worthwhile to invest significant efforts toward developing a long-term information system capable of providing answers to a wide range of queries. Additional factors that make materialized integration difficult are the volatile nature of research data and the often large volumes of data and rigid access rights that prevent data transfer. Virtual integration is unsuitable due to the typical lack of optimization for non-relational sources.

The fundamental difficulty is that data is heterogeneous not only in syntax, structure, and semantics, but also in the way it is accessed and queried. For example, while certain data may reside in feature-rich RDBMSs accessed by declarative queries, others are processed by MapReduce programs utilizing a procedural computation model. Furthermore, many sensor-generated datasets are maintained in CSV files without well-established formatting standards and lack basic data management features. Even different RDBMS products differ in syntax, conformance to SQL standards, and features supported. We recognize this as *data access heterogeneity*.

Abstract

Data access heterogeneity refers to differences in terms of computational models (e.g., procedural or declarative), querying capabilities, and syntax and semantics of the capabilities provided by different vendors or systems; in addition, it includes data types and the formats in which data and query results are presented. One critical aspect of data access heterogeneity is the heterogeneous capabilities of data sources: While some data sources, e.g., relational, graph, and array databases, are classified as strong data sources, weak data sources, e.g., spreadsheets and files, do not have well-established management systems. Furthermore, not all management systems feature the capabilities requested by users' queries. We should add to these levels of heterogeneity, the interest of data scientists in performing computations over the raw data, the domain-specific tools that data scientists utilize to conduct research, and the various tools and languages required to complete a data analysis task.

In this thesis, we identify a need for superior tools to reduce the total cost of ownership associated with the full data life-cycle, from raw data to insights. We also argue that these tools should be democratized through the development of open-source, interoperable, and easy-to-use systems. In contrast to DBMSs that provide mechanisms for storage and querying respective data models, we propose an agile data query system. An agile query system is an expressive data retrieval facility that is unbound by the mechanics of data storage or the limitations of the capabilities of underlying data management systems. The goal of such a system is to provide rapid feedback and avoid data duplication while simultaneously providing end-users with a unified querying and computation model.

We introduce `QUIS` (QUery In-Situ), an agile query system equipped with a unified query language and a federated execution engine that is capable of running queries on heterogeneous data sources in an in-situ manner. Its language extends standard SQL to provide advanced features such as virtual schemas, heterogeneous joins, and polymorphic result set representation. `QUIS` utilizes the concept of *query virtualization*, which uses a federation of agents to transform a given input query written in its language to a (set of) computation models that are executable on the designated data sources. While federative query virtualization offers much greater flexibility and support for heterogeneity than data virtualization controlled by a central authority, it has the disadvantage that some aspects of a query may not be supported by the designated data sources and that the query engine may then have to act as a backup to complement these cases. `QUIS` ensures that input queries are always fully satisfied. Therefore, if the target data sources do not fulfill all of the query requirements, `QUIS` detects the features that are lacking and complements them in a transparent manner. `QUIS` provides union and join capabilities over an unbound list of heterogeneous data sources; in addition, it offers solutions for heterogeneous query planning and optimization. In brief, `QUIS` is intended to mitigate data access heterogeneity through query virtualization, on-the-fly transformation, and federated execution. It offers the following contributions:

1. **In-Situ querying:** `QUIS` transforms input queries into a set of executable jobs that are able to access and process raw data, without loading or duplicating it to any intermediate system/storage;
2. **Agile querying:** `QUIS` is a query system, not a DBMS. It allows and encourages frequent and ad-hoc querying and provides early feedback;

3. **Heterogeneous data source querying:** QUIS is able to accept and execute queries that involve data retrieval from multiple and heterogeneous data sources; in addition, it can transparently perform composition operations such as join and union;

4. **Unified Execution:** QUIS guarantees the execution of input queries. It detects lacks in terms of the capabilities requested by input queries and complements them if they are not supported by the designated data sources;

5. **Late-bound virtual schemas:** QUIS allows for the declaration of virtual schemas to be submitted alongside queries. These schemas have life cycles that are similar to those of the queries and thus do not need to be predefined or pre-installed on data sources; and

6. **Remote execution:** QUIS queries are compiled to self-contained executable units that can be shipped to remote data centers when it is necessary to directly execute them on data.

Throughout this dissertation, we identify and elaborate on the problem statement by specifying a set of requirements; in addition, we offer a solution intended to satisfy them. This solution proposes a federated architecture that consists of three main components: *query declaration*, *query transformation*, and *query execution*. Query declaration provides a query authoring tool, and tokenization, parsing, and validation services. Additionally, it converts the input queries into a more manageable internal query model that can be used by other components. Query transformation includes all of the activities required for the construction of native computation models that can be run on designated target data sources; it also includes query optimization. Query execution is intended to build executable units from the transformed computation models as well as orchestrating their execution on designated data sources. Collecting, reformatting, and representing query results are also among the responsibilities of this component. The representation of the results may include visualization or inter-process data transmission.

We provide a proof-of-concept implementation to demonstrate the feasibility of the solution. Although the implementation does not address all of the solution features equally and may not represent the optimal approach to implementing such features, we intensively evaluate the suggested implementation in order to prove its effectiveness and efficiency. The dissertation concludes with a discussion and a roadmap for future work.

Contents

List of Figures

List of Tables

Thesis Structure

The dissertation consists of four parts: Part I provides an overview of the general area of heterogeneous data querying and integration, identifies the motives this dissertation, and formulates the hypothesis (Chapter 1). Thereafter, it describes the background of the work (Chapter 2) and provides the problem statement (Chapter 3). Finally, this part defines the specifications and the boundaries of the problem by identifying a set of requirements. These requirements guide the solution proposed in Part II.

Part II proposes and describes the main elements of a solution intended to fulfill the requirements discussed in Chapter 3. It begins by outlining a solution architecture in Chapter 5. The architecture introduces three fundamental components: query declaration, transformation, and execution. Query declaration (Chapter 6) formulates the requirements into a declarative query language that is unified in syntax, semantics, and execution. Query transformation (Chapter 7) specifies and explains the techniques used to convert the queries into appropriate computation models, allowing them to be executed against designated data sources. This chapter also elaborates on the solutions proposed for dealing with queries that access heterogeneous data sources, query rewriting, and data type consolidation. Chapter 8 then explores how the transformed and complemented queries might be executed at the end of the pipeline. Query execution is also responsible for returning the queries' result sets to the client in the format they requested. The extent to which the solution satisfies the requirements, in addition to its limitations and achievements, are summarized in Chapter 9.

Part III is dedicated to the evaluation of the proposed solution. We first present a proof-of-concept implementation in Chapter 10 and utilize it to illustrate the correctness of the hypothesis. To prove that the hypothesis holds, we conduct a set of evaluations and discuss their results in Chapter 11. The evaluations are designed to measure the language's expressiveness, system performance on heterogeneous data, scalability when applied to large data, and usability.

Part IV concludes this dissertation. In Chapter 12, we briefly reiterate our assumptions, the solution we provided, and the results of the evaluation. Thereafter, we bring the dissertation to a close by reviewing its achievements and the extent to which the hypothesis is satisfied. Finally, in Chapter 13, we examine a set of important directives for future work.

Part I.

Problem Definition

This part provides an overview of the general area of heterogeneous data querying and integration, identifies the motives this dissertation, and formulates the hypothesis (Chapter 1). Thereafter, it describes the background of the work (Chapter 2) and provides the problem statement (Chapter 3). Finally, this part defines the specifications and the boundaries of the problem by identifying a set of requirements. These requirements guide the solution proposed in Part II.

1 Introduction

In this chapter, we motivate the work conducted in this thesis by describing the gaps and problems it addresses (Section 1.1). We also demonstrate the work's relevance by identifying real-world usage scenarios (Section 1.2). Based on the challenges identified, we derive and formulate the hypothesis investigated in this dissertation and identify a set of operational objectives, the achievement of which leads to the fulfillment of the hypothesis (Section 1.3).

1.1 Motivation & Overview

Data scientists work in environments that are characterized by multi-faceted heterogeneity, e.g., data and applications. Data can be generated, transferred, stored, and consumed in different ways. Data processing, querying, and visualization tools usually require scientists to reformat and/or reload data according to their particular specifications. Additionally, the systems that are widely used neither support all of the requirements of such scientists nor are compatible with each other. It is also not an easy task to build a workflow that seamlessly integrates multiple systems in order to establish a pipe of data that each system can perform a set of operations on. As Jim Gray mentioned in his last talk [Gra08], the entire discipline of science requires vastly superior tools for the capture, curation, analysis, and visualization of data.

In this dissertation, we elaborate upon the challenges that data scientists are confronted with when performing data-intensive researches. Based on these challenges, we define a problem and specify it in greater detail by identifying a set of requirements. We suggest a solution for such requirements by proposing a unified query language and an execution engine for such a language. Furthermore, we implement a proof of concept to demonstrate that the suggested solution is feasible and practical. Our general intention is to define the specification and the grammar of a science-oriented language that allows scientists to focus on solving their research problems instead of dealing with technical issues related to data management, transformation, and transportation. In the remainder of this section, we elaborate on the terms that are frequently used in this document, define certain aspects of heterogeneity, and explain the challenges.

According to the IFIP[1], data is a representation of facts or ideas in a formalized manner that is capable of being communicated or manipulated by processes. Based on this definition, data

[1] http://www.ifip.org/

has always functioned as the cornerstone of human advancement in all of the three scientific discovery paradigms, namely experimental, theoretical, and computational. Nowadays, with the shift to data-intensive scientific discovery [HTT09b], data is playing a more important role than ever and is an integral part of almost any commercial, industrial, or research institute's value-adding process.

Data is used to identify patterns, anomalies, or outliers that existed in the past or to predict the same for the future. It is used to understand the relationships between complex networks of events, to model or simulate situations in systems, to analyze and reach conclusions regarding behavior, and for countless applications in various disciplines. The field of life sciences, for example, utilizes data for species distribution modeling [MOLMEW17]. In high energy physics, the investigation that proved the existence of the Higgs bosons was only possible as a result of data-intensive research [dBCF$^+$16]. While policies that regulate the use of data exist, the processes of generating and applying data using different techniques, at different times, and for different purposes, are as old as data itself. However, recently, collaborative [TWP16] and reproducible [VBP$^+$13] sciences have gained attention and traction.

Data science is a response to the fast-paced advancements in processing techniques and tools and the analysis, interpretation, and application of data. Data science is the process of systematically extracting insight and understandings from data [Dha13]: It analyzes data in a methodological and reproducible manner in order to extract information or derive conclusions. Data science is characterized by its interdisciplinary nature, as it relies on heterogeneous data, a heavy use of statistical and mathematical methods, modeling, and hypothesis-testing techniques. By combining aspects of statistics, computer science, applied mathematics, and visualization, the data science field offers a powerful approach for making discoveries in various domains, e.g., life sciences [GT12, QEB$^+$09], health care [BWB09], and physics [BHS09].

A process in the data science field is comparable to a data-driven workflow in that each step obtains data from previous steps and/or data sources and performs a set of computations on it. These steps can be executed by machines, humans, or in a machine/human collaboration. The term "computation" refers to a broad set of data operations, including cleansing, filtration, aggregation, decomposition, transformation, processing, storing, transferring, and visualization. Processes may be applied to small amounts of well-formed data or to multiple large datasets with syntactical and structural differences; they may take anything from milliseconds to multiple days to produce a result. Furthermore, the processes involved may be interdisciplinary or inter-departmental, meaning that multiple data scientists may be involved, the applicable policies may differ, and the data-processing tools used may not be compatible. Additionally, such processes may potentially require multiple machines if no single device has the required storage or processing power.

Although the term "data scientist" is commonly used to refer to individuals who deal with the scientific data processing mentioned previously, we use the more general term *data worker* (Definition 1.1) throughout this document to refer to any individual who deals with data, regardless of whether his or her work is considered scientific or not. This definition also includes support and administrative tasks, such as the preparation, transformation, loading, and management

of data. Domain scientists, application users, crowd workers, activists, and citizens are also included [AAA+16].

Definition 1.1: Data Worker *An individual who performs a set of computations on data in order to achieve a result. Such computations can be performed for support, administrative, or operational purposes. This term includes data scientists, data researchers, data analysts, business analysts, statisticians, data miners, predictive modelers, data engineers, computer scientists, and software developers.*

Data workers obtain data from various sources, e.g., sensor logs, simulation outputs, field surveys, and reference data imports. Such data is produced in different ways, by different tools and people, and for various purposes. One of the earliest, and ongoing, activities that data workers engage in is the preparation of data for designed analyses. This requires them to conduct data integration [HRO06]. Although the use of automated methods is preferred in the integration and analysis of data, data workers are required to utilize multiple tools in order to accomplish their tasks. Their tool sets usually consist of a combination of programming languages, database management systems, visualization applications, business intelligence programs, operating systems, and statistical packages [KM14], e.g., R [R C13], Python, SQL, and Excel [Rex13, KM14]. Data integration in environments that feature multi-faceted heterogeneity presents its own challenges. Two of the most important challenges that we discuss in this dissertation are data and system heterogeneities.

Heterogeneity in data is associated with syntax, structure, and semantics [DHI12]. It is because data is produced, transmitted, and stored using a wide variety of means, e.g., sensors, instruments, systems, manual collection, simulations, transformations, communication, and storage devices. In addition, the processes involved are subject to protocols and research constraints and requirements. These heterogeneity dimensions frequently require that data workers perform a series of additional steps that are not part of their analysis work; these steps are generally not considered valuable and are time-consuming, usually challenging, and error-prone. Transforming data to meet the input requirements of a tool or loading data onto a managed database are examples of these kinds of extra loads. Data heterogeneity is not avoidable; hence, the current solution is to mitigate it, frequently through the use of data integration.

Data integration is the science and technology of providing a unified view by combining data from different sources [Len02]. It addresses the three dimensions of heterogeneity in data, namely syntax, structure, and semantics [L+06]. Data integration is crucial in today's multidisciplinary and collaborative research environments, in which data is produced and consumed in a cross-functional manner. The term "cross-functional", in this context, refers to the activities engaged in, e.g., by multiple researchers in different divisions or disciplines in order to satisfy various research requirements, using diverse measurement resolutions and various query processors and analysis tools [HRO06, AAA+16]. Data integration can be applied to tasks of various complexities. For example, it can be applied to trivial task of combining the sensor logs acquired from a field survey with local meteorological records or to more complex pipeline such as the Sloan Digital Sky Survey (SDSS) [STG08].

There are two classical approaches to data integration: *materialized* and *virtual* [DHI12]; both were originally developed with business applications in mind. Materialized data integration is a process designed to extract, transform, and load data from a set of data sources into a single unified database, which can be used to answer queries using the unified data. Virtual integration involves placing a logical access layer on top of a set of data sources in order to hide data heterogeneity from applications and users without loading data in advance [SL90, DHI12]. In virtual integration, instead of data, queries are transformed into and executed on the corresponding data sources. Partial results are integrated by a mediator at the time of query to construct the input query's result set.

Both of these integration approaches assume a degree of schema stability. Materialized integration has a high upfront cost and is not suitable when source data changes frequently, as it relies on the schema for the validity of the Extract, Transform, and Loads (ETLs). It also suffers from a range of side effects, including data duplication and version management, the information and/or accuracy loss caused by transformation, and storage and network requirements both during and after the ETLs. Similarly, virtual integration requires the schema of the member data sources for either global or local view definitions. The processes involved in formulating and applying ETLs in materialized integration or defining and installing views in virtual integration add a set of preparatory steps to the actual scientific work that, among their other side effects, increase the amounts of time required before a data worker can issue the first query.

The traditional integration methods also make it harder to reproduce the results obtained from the original data, as reproduction requires the same integration systems to be set up. This is a challenging requirement to satisfy, because, more often than not, data integration procedures are not captured as part of the data analysis workflow, or it may not even be possible to do so [ABML09]. When it comes to scientific data, however, the data sources are often text files, Excel spreadsheets, or the proprietary formats used by recording instruments. As such, these classes of data integration are often not well-suited for scientific work when one considers the fact that it is not only the scientific data used, but also the query requirements of data workers, that change frequently.

The fundamental difficulty is that data is heterogeneous not only in syntax, structure, and semantics, but also in the ways in which it is accessed and queried. While certain data that reside in feature-rich Relational Database Management Systems (RDBMSs) can be accessed by declarative queries, others are processed by MapReduce programs, utilizing a procedural computation model [DG08]]. Furthermore, many sensor-generated datasets are in Comma Separated Values (CSV) files that lack well-established formatting standards and basic data management features. Finally, the various RDBMS products differ in syntax, conformance to SQL standards, and features supported.

We refer to this heterogeneity dimension as *data access heterogeneity*. Data access heterogeneity includes the varieties of data access methods (e.g., procedural or declarative), the available querying capabilities (e.g., aggregate functions or expressive power), syntax and semantics of the capabilities provided by different standards, vendors, or systems (e.g., Offset/Limit syntax

and the semantic differences between PostgreSQL 9.x and MS SQL Server 2012), and the 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 sets of management systems. Others fall under the so-called *weak data sources* [TRV98], e.g., CSV and spreadsheets, and do not have well-established management systems. In addition, not all management systems support the capabilities requested by users' queries; for example, the MapReduce programming model does not support joins and sorting [F$^+$12, YLB$^+$13].

In order to address this challenge, some researchers have proposed the concept of *data virtualization* [K$^+$15, ABB$^+$12]. Data virtualization abstracts data out of its format and manipulates it regardless of the manner in which it is stored or structured [KAA16]. By providing a framework for posing queries against raw data, such a system can permit the use of data sources with heterogeneous capabilities. However, there are additional important aspects of data access heterogeneity:

- **Heterogeneous joins:** Joins are one of the most expensive operators in relational systems. Likewise, when joining data from various sources, one is constrained by source capabilities in terms of choice of join algorithms. For example, consider an inner join query in which the left side retrieves data from a CSV file and the right side from a relational table: irrespective of input sizes, due to the lack of indexed access capability in the CSV file, an indexed nested loop join must have the CSV file as the outer relation and the relational table as the inner;

- **Heterogeneous Query Planning:** The heterogeneity of source capabilities can also make query planning quite challenging: For example, there may be limited access to metadata such as constraints, data types used, indexes, and statistics. In exploratory or experimental data analyses in which datasets are volatile and queries are ad-hoc, even calculating simple statistics can prove expensive and ineffective;

- **Heterogeneous representation:** Data sources differ in syntax, schema, and type system. Therefore, the potential type conversions required to unify partial result sets may cause loss of information or accuracy. Furthermore, the need to determine an appropriate data type for the reconciled results should be taken into account. Designing systems with interoperability in mind may require considering support for various serialization formats. An agile query system should consider early visual feedback, e.g., drawing a histogram of the values of an attribute; and

- **Multiple versions:** In data-centric research, data is often versioned for various reasons, e.g., data cleansing, exploratory aggregations, hypothesis testing, and ensuring the reproducibility of results [SCMMS12]. The data schema may also differ from one version to another [Rod95, CTMZ08, Zhu03]. The need to support multi-version data sources adds a degree of complexity to data access in heterogeneous environments.

In addition to the data and access heterogeneity dimensions, science projects typically have special requirements in terms of data management and processing practices. Over the previous five years, we have supported a number of large-scale collaborative biodiversity and ecology projects, including the German Center for Integrative Biodiversity Research Halle, Jena, Leipzig-iDiv[2], the Biodiversity Exploratories[3], the Jena Experiment[4], and CRC AquaDiva[5], in data management and integration. In these projects, we worked closely with scientists from the domains of biodiversity and ecology. We also developed data management solutions for these domains in the scope of the BExIS[6] and GFBio[7] initiatives. The following summarizes our findings:

- Data workers are interested in running their computations over raw data [AAB+09] not only to reduce data transformation and duplication efforts but also to retain ownership of the data. Although this is a good motive for adopting a virtual integration system, running and maintaining such a system is not cost- or time-efficient for short-term ad-hoc research activities [H+11];

- General-purpose systems such as RDBMSs, although powerful, cannot replace the domain-specific tools that data workers utilize when conducting research [AAA+16]. One of the main reasons for this is that the majority of the current state-of-the-art data management systems work on data that is loaded into their internal storage or serialized in their pre-defined formats. For instance, RDBMSs store data in terms of tables and rows, the majority of NoSQL systems operate on JSON, and XQuery requires XML. Therefore, researchers often need to export the intermediate results produced in a general-purpose system back to their specific tools (examples of real case scenarios are provided in (Section 1.2). This process causes a series of reverse ETLs; reverse ETLs are usually more difficult to conduct, as the general-purpose systems may provide little/no support when it comes to exporting their managed data to the format/structure required by specific tools;

- Scientific data-intensive research often involves volatile data (i.e., the data itself and/or its structure change frequently) and exploratory queries (i.e., queries are run on small subsets of data). These characteristics limit the efficiency of several RDBMS features, such as indexing, schema design, and tuning. This class of exploratory work shifts the data worker's focus from schema design and database management to agile and interactive query systems;

- According to data volume, storage, and access policies, research centers may prefer to accept certain processes and apply them on data rather than transferring data [Gra08]. This is preferred in cloud-based data centers and among communities that manage copyrighted datasets; and

[2] https://www.idiv.de
[3] http://www.biodiversity-exploratories.de/1/home/
[4] http://www.the-jena-experiment.de/
[5] http://www.aquadiva.uni-jena.de/
[6] http://bexis2.uni-jena.de/
[7] http://www.gfbio.org/

- The number of languages and tools needed to complete a data analysis task is often proportional to the number of data sources. This makes cross-functional data integration tasks cumbersome, time-consuming, and less reproducible. In fact, in many cases, the time spent on the data preparation outweighs that invested in analysis.

Dealing with scientific data has been considered a major barrier to the advancement of science for decades. In 1993, Bouguettaya and King [BK93] noted that, given the growing need for data sharing at the time, a priority was the ability to access and manipulate data independently of the manner in which it is organized and accessed. More than a decade later, Jim Grey suggested that superior tools be developed in order to support the entire research cycle, including the capture, curation, analysis, and visualization of data [HTT09a]. Concurrently, data-querying and processing tools have reached such a degree of inconsistency that, in 2009, a group of database experts collaboratively concluded that radical changes to data-querying systems are needed [AAB+09]. In 2016, the same group stated that coping with increased levels of diversity in data and data management, as well as addressing the end-to-end data-to-knowledge pipeline, are among the top five challenges in the database field [AAA+16]. Even today, data-processing tools fail to adequately cope with requirements and challenges. The lack of complete solutions has led scientists to develop or adopt application-specific solutions [AKD10]; causing software systems to be typically tightly bound to their respective domains and difficult to adapt to changes [AKD10].

The transient nature of scientific and research data, as well as its high volume and short-time usage, make applying a schema and loading it onto conventional systems obsolete. The increasing use of public and private cloud-based data centers and data repositories has reduced the impact of data transfer and duplication issues and provides facilities for high-performance data processing. However, today's cloud data services are considerably more restricted than traditional database systems [AAB+09]. Easy and agile methods that apply computations to data [HTT09b] would be of interest and beneficial for all parties involved.

In the remainder of this chapter, we first introduce two real-world use-cases to demonstrate how deeply heterogeneity influences research and scientific work (Section 1.2). In Section 1.3, we briefly explain the solution we propose to the challenges identified in the introduction. Based on the proposed solution, we establish our hypothesis, define our objectives, and explain the manner in which we test the hypothesis.

1.2 Usage Scenarios

In this section, we present two examples of real-world, multi-source, import-/export-intensive experiments that demonstrate how data and tool heterogeneity result in unnecessary complexity and consume a remarkable portion of researchers' time.

1.2.1. Ecological Niche Modeling Use-Case

Gaikwad et al. conducted a study on the customary medicinal plant species used in Australia in order to predict the ecological niches of the medicinally important species based on bioclimatic

variables [GWR11]. We interviewed the authors and inquired as to how they dealt with their data. The following is their summary of the steps they took from the data retrieval to the result presentation:

"We used MaxEnt [PAS06], the ecological niche-modeling program, for the prediction. To feed the software with appropriate data, we had to obtain data from various sources and transform them accordingly, as follows:

1. ***Species data obtained from CMKb:*** *We queried CMKb [GKV+08] to extract species' names and their respective medicinal uses, downloaded the data as a CSV file, and then imported it into an MS Excel sheet for data cleaning.*
2. ***Distributional data obtained from GBIF[8]:*** *We downloaded the species observation locations as a CSV file and loaded it into a MySQL database table for cleansing. The dataset contained multi-million records; therefore, exceeded MS Excel's capacity. Afterwards, we exported the cleaned relational data back to a CSV file to feed it to DIVA GIS [HGB+12] software. We used DIVA to find and eliminate geographically erroneous location records. We then ported the resulting cleaned distribution data back to MySQL and excluded the species with fewer than 30 locality records. This reduced the number of species to 414. Finally, we transformed the distribution records associated with the 414 species data into a set of CSV files organized in species-specific folders, each for one species.*
3. ***WorldClim data:*** *We downloaded the world climate dataset [H+05]. It is a 2.5 arc-minutes resolution dataset that contains 19 bioclimatic variables in grid (grd) format. We used the R system to choose variables with minimum correlation. We then converted the big grid files into ASCII format using our own program, developed in Delphi.*
4. ***Ecological niche modeling:*** *We generated the ecological niche models [GZ00] using a Python script that ran the MaxEnt software on each of the species folders. Each folder contained the species distribution data, the WorldClim data, and the projected modeling result in a spreadsheet.*
5. ***Species richness Map:*** *We thresholded the generated model results in the R software and overlaid them to derive the species richness map [GTK98]. The map was stored both in ASCII and grid formats.*
6. ***Medicinal Value Map:*** *We calculated the weight for each species based on the number of its unique medicinal uses and added them to the individual species model, followed by overlaying all of the weighted models using R software to generate the medicinal value maps as shown in Figure 5 [GWR11].*

It is worth mentioning that we spent more effort on data preparation, integration, and tool coupling than on the analysis needed to conduct the research."

The scenario identified above occurs frequently in multi-source and collaborative research environments. Those who simultaneously participate in different projects are more likely to be confronted with data and access heterogeneities; hence, a greater number of tools, programming languages, and ad-hoc developments will be required [Rex13].

[8] Global Biodiversity Information Facility `http://www.gbif.org/`

1.2.2. Sloan Digital Sky Survey Use-Case

The Sloan Digital Sky Survey (SDSS) has created a three-dimensional map of the Universe. It contains a large set of multi-color images of the sky and spectra of astronomical objects. To accomplish this goal, SDSS acquires and maintains large multi-dimensional datasets of the sky. It makes this data available through different mechanisms [APA+16]. Raw and processed image data are available through the Science Archive Server (SAS)[9]. The catalogs and derived quantities can be accessed via the Catalog Archive Server (CAS). The server provides interactive[10] and batch[11] querying features as well as synchronous and asynchronous modes.

The SDSS also makes this data available to the public by feeding a virtual observatory system. The virtual observatory integrates both historical and current SDSS data into online repositories for public access. Because of the differences in the formats, resolutions, update rates, and the types of data available through the observatory the SDSS must transform the original data. Transformation is managed by a pipeline of ETL operations that ingests and validates the data and produces the models and records used in the virtual observatory.

SqlLoader is a tool that was developed to implement the SDSS's data-loading pipeline [STG08]. It performs all of the steps required to transform the input raw data into the final designated relational databases that serve the virtual observatory. SqlLoader utilizes a distributed workflow system to orchestrate the required tasks [TSG04]. Tasks are units of processing that are associated with the workflow nodes. The workflow itself is modeled as a directed acyclic graph.

SqlLoader has implemented a number of different tasks. One task ingests the binary data stored in the Flexible Image Transport System (FITS) format that is stored in Linux machines and converts it into CSV files, while another task transfers the CSV files to staging servers running on MS Windows for quality control purposes. There are also tasks to insert the final data into chosen instances of staging and production MS SQL Server databases. Copying staging data to production databases, merging data from multiple databases, checking data integrity, and reindexing are also among the tasks handled by SqlLoader.

Using the workflow management system and parallelizing tasks to operate on chunks of data have remarkably improved the overall performance. However, this pipeline operates in an environment that features large amounts of data and significant tool heterogeneity. It ingests different data formats, e.g., FITS, CSV, Relational data, and ASCII. Also, it deals with various computational environments such as SQL, shell scripts, the Visual Basic programming language, Microsoft Data Transformation Services, and workflow management systems.

1.3 Hypothesis and Objectives

As established in the overview (Section 1.1), the main problem that we decided to address throughout this work is mitigating data access heterogeneity for data workers when dealing with

[9] `data.sdss.org/sas/dr13`
[10] `http://skyserver.sdss.org`
[11] `http://skyserver.sdss.org/casjobs`

data. In other words, we seek to develop a solution intended to transform queries, as opposed to data, in order to address the data access heterogeneity problem. Our goal is to transform a query written in a user language into one or more queries in the designated data-source languages so that, upon execution, the result set is as if it was returned by the original query.

For the specific cases in which a relational database is the data source and an object-oriented application program is the data target, we have extensive previous experience with object-relational mappers such as Hibernate [Red] and LINQ [Mic], in which an object-oriented query definition can be translated to its relational counterpart. Furthermore, federated systems, such as Garlic [HKWY97], as well as polystores, such as [DES+15], have offered solutions intended to overcome such problems. However, the challenge is to allow data workers functioning in heterogeneous environments to process datasets that are stored in different formats and types, at various levels of management.

We propose the concept of *query virtualization*, which uses federated agents to transform submitted queries into native counterparts. These native counterparts are executed in parallel by agents, considering inter-dependencies, and their results are assembled transparently. Federated query virtualization permits much greater flexibility and support for heterogeneity than data virtualization that is controlled by a central authority. This approach is able to deliver a unified query language and complete execution of the submitted queries that are written in the language. The unified nature of the language conceals syntax and semantic incompleteness and the inconsistencies in the target data sources. Unsurprisingly, it comes with the cost that some parts of a query may not be supported by the designated data sources. Hence, we add a special agent to our query engine to act as a backup in order to complement such cases.

Below, we specify the expected outcomes of our work in the form of a hypothesis:

Hypothesis 1 (A Universal Querying System is Feasible and Useful): *Consider a dataset that consists of a set of data containers spread over multiple data sources. In the presence of structural, representational, and accessibility heterogeneity among the data sources; there is a uniform query system that allows a typical data worker to query and manipulate data in-situ. Such a system:*

- *provides a unified data retrieval and manipulation mechanism for heterogeneous data that is independent of data organization (Definition 1.2) and data management;*
- *is expressive enough to support core features of the most frequently used queries; and*
- *reduces the time-to-first-query while maintaining reasonable performance on subsequent queries, is scalable, and is useful in real-world scenarios.*

Definition 1.2: Data Organization *A data organization refers to the unique combination of the key characteristics of a data container. This term includes computation model, serialization format, and presentation model.*

We test the hypothesis by developing a reference implementation and evaluate it accordingly. The implementation serves as a proof of concept that demonstrates the feasibility of the solution. The evaluation proves that the implementation can function in real-world scenarios and meaningfully improves a set of chosen indicators. We call the reference implementation `QUIS` (QUery In-Situ).

`QUIS` is an agile query system with a unified query language and a federated execution paradigm that utilizes late-binding schemas to query heterogeneous data sources in-situ and present polymorphic results. `QUIS` is agile, as it provides feedback rapidly. This agility is achieved by two features: a) `QUIS` reduces the time-to-query to the time required to write a desired query and b) the time required to prepare and load data is eliminated or radically mitigated. Its polymorphic result set representation allows data workers to rapidly obtain visual feedback and refine their queries and/or processing. The system's in-situ feature accesses data in its original format and source and performs composition operations, e.g., join and union, on heterogeneous data. `QUIS`' unified language allows for authoring queries in a data-source-agnostic manner. All statements, clauses, expressions, and functions available at the language level are guaranteed to be executable on any data source supported. The language extends SQL by adding virtual schemas, versioned data querying, heterogeneous joins, and polymorphic result set representation.

`QUIS`'s federated query engine is responsible for dispatching an input query to the available data sources, collecting the partial results returned from members, assembling the final result set by applying composition queries as well as requested representational transformations. The engine detects and complements the features that may be lacking or inconsistent among the member data sources. The late-bound schema feature incorporated into the system allows for the definition of an effective result set schema at the time of query. Utilizing this feature, it is possible to share a schema between different queries, to change the schema of a query without accessing the data source, to provide virtual view of data obtained from various sources, and to decouple queries from the mechanics of concrete data types, schemas, formats, and transformation. The language's integrated result representation makes it possible to transform query results into tables, visualizations, or serialization formats such as XML or JSON.

In summary, QUIS has the following core features:

1. **In-Situ querying:** `QUIS` transforms input queries into a set of executable units, queries, or programs that are written in the computation model of the target data source(s). It also compiles the units, dynamically and on-the-fly, to executable jobs that access and process the raw data according to a query's requirements. This eliminates the need to transform data into a specific format or to add it to a database management system. In-situ querying not only saves the users' time by short-circuiting data transformation and loading, it also eliminates the negative side effects of data duplication. Furthermore, it promotes agile and frequent querying, a feature which is extremely useful in scientific work;

2. **Agile querying:** `QUIS` is a query system, not a database management system. It allows and encourages frequent and ad hoc querying with early feedback. Specific features, such as virtual schema definition and query independency from data organization, ensure a

great degree of data independency as well as portability. Its declarative syntax, as well as its similarity to SQL, reduces the learning curve;

3. **Heterogeneous data source querying:** In addition to single-source querying, QUIS is able to accept and execute compound queries that involve the retrieval of data from multiple heterogeneous data sources. Furthermore, it can transparently perform composition operations such as join and union on the partial results returned by individual data sources in order to assemble the final result set;

4. **Unified execution:** QUIS guarantees the execution of input queries. This means that whatever capabilities are promised by its language would be accepted and executed by the system, regardless of the actual capabilities of the underlying data sources. QUIS' query engine detects the absence of capabilities requested by input queries and complements them if they are not supported by the designated data sources;

5. **Late-bound virtual schemas:** QUIS allows for the declaration of virtual schemas that are submitted alongside queries. These schemas have similar life cycle to those of the queries and thus do not need to be predefined or pre-installed on the data sources; and

6. **Remote execution:** In addition to the unified execution feature, QUIS processes (i.e., sets of queries) can be transformed into self-contained executable units that can be shipped to remote data centers and applied directly to data in order to produce the desired results.

We describe these features in Part I, suggest a detailed solution in Part II, and finally implement and validate our hypothesis in Part III. In essence, our evaluations prove the following:

1. QUIS **reduces the time-to-first-query:** By means of a human study, we demonstrate that QUIS dramatically reduces the initial time-to-first-query. Thus, users are able to start querying data immediately from the beginning (Section 11.2);

2. QUIS**'s query execution time is reasonable:** We demonstrate that the query engine has a reasonable performance and the reduction in time-to-first-query does not come at the cost of a dramatic slowdown of the sub-sequent queries (Section 11.3);

3. QUIS**'s query execution engine is scalable:** We empirically demonstrate that the query engine's performance scales linearly with the size of the queried datasets and outperforms baseline systems (Section 11.4);

4. QUIS **uses effective query optimization:** We study the effect of various optimization techniques in terms of facilitating efficient implementation, showing that our rule-based optimization has a remarkable effect on performance (Section 7.4)[12]; and

5. QUIS **is usable:** Supported by a user study, we demonstrate that QUIS is useful, usable, and satisfactory (Section 11.5). We also compare the language's expressive power with that of related work to show that it is able to express user queries (Section 11.6).

[12] We discuss this subject in Chapter 7 (Query Transformation) in order to situate the evaluation of the optimization rules close to the explanation thereof.

With all of the heterogeneity involved and the inexpensive, rapid, large, and open data accessible to not only data scientists but also to ordinary or occasional researchers, we foresee a wave of small- to medium-sized research teams working autonomously on data and publish their results openly. Thus, there is likely to be a demand for open and free data-querying and processing utilities with low upfront installation costs that are able to deal with a wide spectrum of various data organizations in an agile manner and present the results in an accessible and intuitive manner.

In the remainder of this part we explore the background of this work (Chapter 2) and provide the problem statement (Chapter 3). The problem statement identifies a set of requirements that define the boundaries of the problem. These requirements are considered in developing our suggested solution in Part II. The proposed solution is then implemented and evaluated in Part III in order to demonstrate that the hypothesis holds. A discussion and an overview of the limitations of the solution, as well as possible future work, are presented in Part IV.

2

Background and Related Work

The objective of this thesis is to develop a solution that provides heterogeneous data querying in environments that feature limited and/or inconsistent functionality. In order to address related research areas, we introduce languages, systems, and concepts that overlap with our aim and enumerate both the capabilities that they provide and those that they lack. This assists us to establish a foundation for our work, to identify and scope the areas of interest, to formulate our requirements, and to justify the need for the solution we propose in Part II.

The vast number of database management systems available today is the result of a "no one-size-fits-all" approach [SC05]. Diversity in terms of requirements, disciplines, use-cases, data formats, distribution, scale, and performance has driven multiple development efforts, resulting in an array of Database Management Systems (DBMSs) that are specialized in particular domains.

While major players still prefer to use relational logic, NoSQL systems have been widely adopted by businesses and academia for big data processing in distributed environments. Scientific Database Management Systems (SDBMSs) that rely on multi-dimensional arrays as their primary data model are implemented and adopted in real-case scenarios. In addition, document-based DBMSs are now commonly used in semi- and/or dynamically structured data management use-cases. Heterogeneous database systems have employed the concept of *views* in order to address the problem encountered when attempting to answer queries using sources that have inconsistent query capabilities [Pap16]; Federated Database Management Systems (FDBMSs) and polystores are examples of such systems.

As our work is highly related to data and access heterogeneities, we discuss a wide spectrum of related work: We consider RDBMSs, as the basics of data-querying systems, in Section 2.1, and we discuss the challenges that arise when multiple database systems are involved in data-processing and analysis tasks. Moreover, we explore how federated database systems (Section 2.2) and polystores (Section 2.3) have approached those challenges. We provide background information on NoSQL systems in Section 2.4 and discuss a number of their features. We introduce array-based database systems in Section 2.5 and highlight their importance in numerical/scientific data processing. In addition, we study the emerging approaches and techniques intended to deal with data as is, without or with minimum preparation. We investigate the external file attachments and querying techniques that have been added to conventional RDBMSs

in Section 2.6. Furthermore, we discuss the concepts of adaptive systems, in Section 2.7, and NoDB, in Section 2.8; these are examples of the recent paradigms that propose mechanisms intended to adapt databases to queries or even to create databases upon receiving queries.

2.1 Relational Database Management Systems

A DBMS is a (set of) software used to maintain collections of data. Maintaining data involves a wide range of operations, including transformation, storage, updating, and retrieval. DBMSs rely on data models; a data model is a set of description and constraint constructs that conceal and govern storage details.

A Relational Database Management System (RDBMS) is a DBMS that its data model is *relational*. A *relation* is a set of unordered *n-tuples*, each of which has n uniquely identifiable domains [Cod70]. An instance of a data model that describes a specific dataset is called a *schema*. Schemas provide data independence [Cod70], isolating applications from the ways in which data is structured within an RDBMS, as well as from later changes to those structures. A schema is described in term of *relations* and *constraints* [ABC$^+$76]. Data tuples, or *records*, are formed according to the specifications of the schema's relations, while constraints enforce integrity, uniqueness, and data types.

RDBMSs provide Data Definition Languages (DDLs) for schema manipulation, as well as Data Manipulation Languages (DMLs) for data querying and manipulation. These languages are usually high-level declarative non-procedural formalisms that allow users to formulate expected solutions. Structured Query Language (SQL) was [LCW93], and remains, the most commonly known and used language for accessing and manipulating data in an RDBMS. It is based upon relational algebra and tuple relational calculus. Although SQL has been standardized since 1986, various vendors have implemented the standards differently, thus producing various flavors; some of these flavors even violate the declarative nature of the language by offering procedural constructs. For example, IBM, Oracle, and Microsoft amended SQL PL, PL/SQL, and T-SQL, respectively, to add controls, conditional commands, and procedures to the language.

The primary difficulty with RDBMSs is that they require input data in relational form and provide no or very limited support for other data organizations. This requires RDBMS users and applications to transform data from its original format to its relational equivalent and to transfer and load it into the target database in order to allow it to be queried thereafter. This task has proven to be cost- and time-inefficient, error-prone, and repetitive. The need to design a schema in advance and query only through that schema, as well as enforcing primary and foreign keys, is often an obstacle to agile research environments characterized by dynamic data and query requirements. A large number of DBMSs intended to either reduce users' dependency on schemas or to balance the effort required to create and maintain schemas are merging [JMH16].

2.2 Federated Database Management Systems

A conventional federated system consists of a set of possibly heterogeneous database management systems that are supervised by a mediator [SL90, FGL+98, BS08, DFR15]. It is likely that member databases will have their own query languages [FGL+98], capabilities [DFR15], and optimization preferences [DH02]. A federated database system acts as a virtual database system that accepts queries in the language of the mediator's choice, decomposes them into sub-queries to be executed by the member databases, integrates the sub-queries' results into a final result, and returns the result to the requester. When such a system receives an input query, it generates a federated query execution plan. The generated execution plan determines which member databases should execute each sub-query. The sub-queries are then distributed to the designated members and executed asynchronously. The partial results returned by the members are passed to a chosen member for integration, where operations such as join, union, and aggregation are performed. The final query result is returned to the mediator and passed on to the requester.

The mediator provides a unified query language over the virtual database that is built on top of the member databases of the federation. Utilizing wrappers, input queries written in the mediator's language are transformed into their native counterparts and executed against the designated member databases.

Garlic [HKWY97] is a middleware designed to integrate data from a broad range of data sources (referred to as components). The data sources are anticipated to have different query capabilities. This middleware accesses the data sources via wrappers; the wrappers transform the input queries into their wrapped data sources' languages or programming interfaces. The middleware and the wrappers interact via Garlic's object-oriented data model; hence, the data in the underlying data sources is viewed as objects. The objects are categorized in collections, allowing Garlic to query them. Garlic has a catalog that records the global schema, the associations between data sources and wrappers, and any statistics that may be helpful for querying. Upon receiving an input query, Garlic generates a query plan, optimizes it, and dispatches sub-queries to associated wrappers. It waits for the sub-results and attempts to assemble them by shipping these partial results to capable wrappers.

Garlic utilizes a cost-based optimizer [SL90], which relies heavily on statistics, cost of operations, and the estimated cardinality of the result set. It also has a rule-based optimizer that optimizes queries in a three-step sequence: single-table access; join operations; and projection, selection, and ordering. This ordering prevents the optimizer from applying Select-Project-Join (SPJ) rules, such as push-ahead selection [Cha98], that are utilized to reduce the cardinality of upper operators. The cost of performing operations by wrappers is not known and should be set and tuned manually. The wrappers have the responsibility of estimating the input cardinality, which implies that, in addition to the central catalog, each wrapper should also maintain a local catalog. Methods of estimating costs and collecting statistics on raw data without actually touching and parsing the data have remained open. Garlic assumes a common data model that also expresses the limited capabilities of the remote sources. The mediator rewrites the queries

using the model and the capabilities that the remote sources are identified to be possessing. This introduces the well-known containment [JKV06] problem [TRV98].

DISCO is a heterogeneous distributed database system that mitigates the fragile mediator, source capability, and graceless failure problems [TRV98]. DISCO's data model is an extension to ODMG 2.0 [RC95]. It relies on global uniform view definitions shared between the mediators and data sources for mapping, conflict resolution, and transformation. This approach is usually adopted when the mediator aims to reconcile semantically similar entities into a unified entity.

In such cases, an input query is forked and tailored to a set of local queries, and the results thereof are unioned in order to form the final result set. It is expected that the views will be defined by database administrators (DBAs), who must resolve conflicts among the different models, schemas, and semantics of the data sources in order to construct uniform semantics for the mediator schema and to deploy the schemas to the system before the first query can be issued. DISCO does not support reconciliation functions in its data model; these functions are required to determine how data values from different sources must be combined. DISCO assumes long-term schema stability, implying that, when views and mappings are deployed, they will remain valid for a lengthy period of time, meaning that users can query data and expect that their queries will be executable on the deployed configuration. DISCO performs partial query evaluation only if the normal execution of a query fails due to the unavailability of (some of) data sources. The partial evaluation is based on the availability of nodes in the operator tree; hence, unavailable sub-trees do not return any results. There is no complementing plan in place.

2.3 Polystore Systems

A *polystore* is a mediator system that allows integrated querying over multiple databases. It is much like a federated system, with the exception of *views*. Polystores facilitate querying over multiple data models by allowing users to exploit the features of the native query languages of the target databases. The ideal realization of such a feature results in semantic completeness [DES+15] in that users have access to all of the capabilities of each and every member database. Queries are written in the native languages of the databases that are intended to execute them. A set of compositional queries (or execution directives) are also available to move data from one database to another or to a mediator in order to assemble the final results by performing the requested combine (e.g., join or union) operations.

Franklin et al. [FHM05] discussed the challenges of managing data across loosely connected heterogeneous data collections, classifying them as search and query capabilities, rule enforcement, lineage tracking, and the management of data and metadata evolution. They introduce *dataspaces* as a new abstraction for use in data management. A dataspace models and maintains a catalog of member data sources and their relationships. It then provides search and query functionality over all of the participating data sources, according to the extent to which those sources are integrated; more sophisticated functions are provided on more closely integrated dataspaces. In other words, the operations available to a dataspace are proportional to the level of integration

of the sources in that space. This characteristic gives dataspaces a navigational nature in that a user iteratively queries a chosen dataspace using the associated set of functions and identifies a target group of data items in a more integrated dataspace until he or she reaches the final result.

Dataspaces are intended to accommodate as many data sources as possible. Data sources fall into different classes of expressive power; hence, identifying and maintaining a set of common and useful operation would prove to be a significant challenge. This issue results in users having to constantly verify the correctness, completeness, type matching, and consistency of the semantics of operations.

BigDAWG [DES+15], for example, uses the notion of an information island to refer to a set of databases that can be queried using a single query language. It then provides a cross-island query language that accepts a group of individual island queries as its sub-queries. Island-specific queries are executed on their respective island engines, while the compositions are performed on the islands identified by the *SCOPE* command. An island that performs the compositions requires other sub-queries' results to be transformed into its data model and transferred to its designated engine in order to be able to merge the partial results into an integrated one.

Polystores allow users to use the native languages of the member databases; hence, the final query is a mixture of the components' query languages and the composition elements of the polystores. This hinders readability, maintainability, and reusability. Declarative query languages promise the isolation of syntax from semantics and execution order; however, the semantics of BigDAWG query changes when the composition is switched from one island to another. This requires users to be aware of the scoping and submission order. In addition, the capabilities of the engines chosen to perform the composition queries may vary. Therefore, it is the user's responsibility to ensure that the capabilities required for each and every query are available.

2.4 NoSQLs

According to the CAP theorem [GL02], it is not possible to provide consistency, availability, and partition tolerance simultaneously. This was the main motivation for the development of a new generation of DBMSs, known as NoSQL. NoSQL systems loosen consistency in favor of availability and partition tolerance for distributed, high-throughput, and big data environments. NoSQL, in general, refers to a category of databases that utilize non-relational mechanisms for storing data [Lea10]. In addition, NoSQL databases attempt to simplify design, encourage scaling, reduce schema binding, and provide timely processing of big data.

NoSQL systems utilize various data models to provide the best possible consistency while maintaining high availability and handling network partitioning. Moniruzzaman and Hossain [MH13] classified NoSQL databases into four categories: wide-column stores, document stores, key-value stores, and graph databases. For example, Cassandra stores data column-wise [WT12], MongoDB persists information as documents serialized in JSON [KR13], and Dynamo stores data as key-value pairs [DHJ+07]. Neo4J is a graph database with the ability to assign properties not only to nodes but also to edges [The16].

Column-oriented storage for database tables boosts query performance because it drastically reduces the amount of data that is necessary to load from disk, thus reducing the overall I/O footprint. Distributing columns and rows of data over the various nodes of a cluster improves the performance of the consuming algorithms. This data model is well suited for analytics, data warehousing, and the processing of big data.

Cassandra, for example, is an open-source, column-oriented database that is able to handle large amounts of data across a network of servers [LM10]. It is a highly available system with a tunable consistency model. Unlike a table in a relational database, different rows in the same table are allowed to have different set of columns. Apache HBase is another open-source, column-oriented, distributed NoSQL database [Cat11] that runs on the Apache Hadoop framework. It provides a method of storing large quantities of sparse data using column-based compression and storage.

Column stores provide limited querying functionality. Range queries and operations such as "in" and "and/or" are supported in Cassandra, but the "in" operator can be applied to partition key columns only. Furthermore, support for inequality operators is bound to the ordering preservation of the selected partitioner. Although column stores (specifically Cassandra) offer a SQL-like query language, the provided feature set and execution logic may differ from those of the standard SQL.

A document database is designed to store semi-structured data in the forms of documents. A document is often considered to be a self-sufficient textual representation of an entity, possibly including its satellite entities. Self-sufficiency broadly means that a given document has no pointers to other documents or that such pointers are opaque to the database management system. The schema used by each document can vary. Documents are stored as rows (analogous to relational databases terminology) and are usually serialized as JSON or XML. The majority of document stores are able to index and query documents' contents [MH13].

Document stores offer APIs for querying ranges of values and nested documents. They also accept compositional operations, e.g., "and" and "or", but lack strong support for aggregation. The UnQL project[1] offers a SQL-like syntax for querying JSON, which can be used by a wide spectrum of document stores.

MongoDB, CouchDB, SimpleDB, and Terrastore are among the open-source, high-performance, document-oriented DBMSs [Cat11]. They provide different levels of sharding, replication, document content indexing, and consistency [Ore10]. For example, while MongoDB demonstrates strong consistency at the document level, CouchDB provides scalability by reading, potentially out-of-date, replicas.

Key-value stores are in fact distributed dictionaries [HJ11]. Data is encapsulated in a value and is addressed by a unique key. Values are isolated from and independent of each other; thus, they are completely schema free. The schemas and the possible relationships between values should be handled by the consuming applications. Key-value stores are useful for the rapid recovery

[1] `http://unqlspec.org`

of identifiable data, e.g., user information retrieval on large social networks, web session management, and distributed caching. The APIs of key-value stores largely provide only key-based operations; hence, abstracting them beneath a query language would be unnecessary. The majority of querying features are implemented at the application layer. Dynamo, Voldermort, Riak, Redis, and MemCached are all examples of distributed key/value stores and demonstrate various levels of consistency, persistence, and key distribution [Cat11, HJ11].

NoSQL databases differ in their data models and the query functionalities they offer. They provide various CAP trade-offs, as well as different degrees of schema evolution. Although it is not required, a number of the widely used NoSQL systems provide SQL-like query languages. This, on one hand, allows users to continue to rely on their SQL experience and to maintain their distance from these systems' underlying languages; on the other hand, it provides opportunities for these systems to optimize the input queries. Systems such as Pig [O$^+$08] and Hive [T$^+$09] attempt to draw a declarative querying layer on top of the underlying procedurally programmed MapReduce [DG08] that is used in many big data NoSQL stores. A comprehensive feature comparison of NoSQL systems is conducted in [MH13].

2.5 Scientific Databases

Data is one of the most valuable assets in science, particularly in data-driven science. It is crucial that data can be retrieved and that preliminary processing can be performed in a timely manner [EDJ$^+$03]. Scientific data poses additional considerations for DBMSs: For example, Stratos et al. [IAJA11] explain that the structure of arriving scientific data may change on a daily basis. The attributes of new data may differ from that of the data used previously, and a scientist may need to navigate it differently. In addition, hierarchical data is natural to some domains, such as biology [EDJ$^+$03]. In many disciplines, scientific data can be modeled in multi-dimensional arrays. Libkin et al. [LMW96] highlighted the need for such an array-based scientific query language.

SciDB is a multidimensional array database management system [SBPR11]. In SciDB, an array is defined by its dimensions and attributes; the dimensions can be either unbounded or bounded. Each combination of dimension values defines a cell. Cells can hold scalar, composite, user-defined, or even nested array data values, each of which is called an attribute [Bro10].. In order to access and process the arrays, SciDB uses Array Query Language (AQL) and Array Functional Language (AFL). AQL is the SciDB's SQL-like declarative language, which is used for working with arrays [LMW96, RC13]. Queries written in AQL are compiled into AFL and then passed through the processing pipeline [SBPR11] for execution. AQL includes a counterpart to SQL's DDL, which assists in manipulating the structures of arrays, dimensions, and attributes. Similarly to RDBMSs, SciDB also requires data to be transformed to its array data model and loaded to the system before it can be used. This approach suffers from all of the costs and side effects of ETLs if the original data is not produced as an array from the beginning. Beyond the similarities between AQL and SQL, the former processes queries, e.g., join dimensions, in a

remarkably different way. These differences may cause semantic inconsistencies and lead new adopters to incorrect results/conclusions.

SDS/Q [BWB+14] is an in-situ query processor that operates directly on array-based data such as HDF5 [FHK+11] and netCDF [Uni15]. It eliminates the need to load data onto systems such as SciDB and can be integrated into the larger processing pipelines that are usually required for data analysis. The SDS/Q query execution engine accepts queries in the form of a physical execution plan, in that the leaf operators scan the designated HDF5 datasets or the generated indexes and return relational tuples. SDS/Q has neither notion of data or query virtualization nor heterogeneous querying facilities.

2.6 External Databases

Recently, several open-source and commercial database systems have included the functionality of querying external data. The concept is that a system can read data directly from external files and integrate the read data with other parts of the input queries. It is assumed that data is queried using the same query language used by the system. Nevertheless, current designs do not support any advanced DBMS functionality. In addition they cannot match the performance of a conventional DBMS, as they need to continuously parse the externally read data.

Data Vault [IKM12] utilizes SciQL in order to provide an interaction mechanism between a MonetDB [IGN+12] DBMS and file-based repositories [ZKM13]. It retains data in its original format and provides a transparent access and analysis interface to that data using its query language. Based on the requirements of the incoming query and the metadata of the datasets, Data Vault builds a sequence of operations in order to perform just-in-time data loading. It can load the query results into the hosting DBMS allowing them to be submitted to further traditional queries. Systems such as Pig Latin [O+08], Hive [T+09], and Polybase [D+13] have extended their support to external sources by incorporating data-processing techniques such as MapReduce.

Systems such as Data Vault and the MySQL CSV storage engine [Cor16] have integrated support for querying a pre-defined set of external files into a hosting DBMS. However, a modular design that allows for the registration and integration of new types of data sources is not available. In addition, when supporting arbitrary files or using additional data sources is admitted, the need to manage inconsistencies of the underlying management systems arises.

2.7 Adaptive Query Systems

Data scientists are increasingly interested in running their computations over raw data [AAB+09] not only to reduce the amount of effort invested in data transformation and duplication but also to retain ownership of the data. Such scientists usually tend to store and manage their data in

environments they can control. This represents a good motive for adopting a virtual integration system. However, running and maintaining such a system is not cost- and/or time-efficient when conducting short-term ad-hoc research activities [H$^+$11]. ViDa [K$^+$15] has demonstrated that querying heterogeneous raw data sources is feasible.

ViDa utilizes RAW [K$^+$14] to read data in its raw format. It processes queries using adaptive techniques and just-in-time operators. It generates data access/processing operators on-the-fly and runs them against the data. Statistics are collected during query execution and utilized to generate superior plans for repeated queries. Furthermore, positional maps are generated for text-based data containers such as CSV files. These maps are created and maintained dynamically during query execution to track the positions of data in raw files.

In ViDa, each operator in the query tree reformats the input data according to the requirements of the upper level operator, primarily because data presentation is an integral part of the system. This may result in multiple transformations during the execution of the query plan and consequently lead to an overall query overhead.

In ViDa, the virtual schemas of the result sets are defined in a manner similar to that of SQL projections, which implies that the queries are aware of the structure of the underlying data. The so-called data descriptions also contain connection information. This tightens the queries to the physical/logical structures of the underlying data.

An important characteristic of virtualization is the level of abstraction created to separate interfaces from implementations, e.g., decoupling a query execution engine from the mechanics of parsing, optimization, and execution. ViDa relies on the capabilities of its plug-ins to transform input queries into corresponding query trees. However, the plug-ins and/or data sources may expose inconsistent functionalities or even lack some. For example, ViDa supports aggregate functions at the language level, but not all of the underlying plug-ins/data sources may support the aggregate functions. This issue introduces a degree of uncertainty to query execution. In general scenarios, the non-supported operators are simply omitted from the query plan in the hope of producing a larger (and consistent) result set.

2.8 NoDBs

Organizing data in schemas and databases requires both time and money, particularly in data-intensive systems that feature large amounts of input data. Additionally, the data-processing paradigm is shifting from querying well-structured data (whatever the structure may be) to querying whatever structure is available. This is in the same direction with the Ailamkai's advocate for running queries on raw data and decoupling the query processing from specific data-storage formats [Ail14].

Although the use of traditional database management systems is growing, the growing need for tools that are capable of better handling a variety of emerging data has also been recognized [EDJ$^+$03]. In scenarios in which data arrives in a variety of forms, it is not practical to

decide on an up-front physical design and assume that it will prove optimal for all of the various versions of the data. Enforcing up-front schemas increases time-to-query and requires users to load data to a system with the defined schema. This may lead users to opt for file-based data-processing tools or custom development.

Ioannis Alagiannis et al. [ABB+12] have demonstrated how it is possible to avoid the approach used to load data into traditional databases. They describe a system called NoDB, which provides the features of traditional DBMSs on raw data files. Their research identifies performance bottlenecks that are specific to in-situ processing, namely repeated parsing, tokenization overhead, and expensive data type conversion costs. They proposed solutions intended to overcome these difficulties by, e.g., introducing an adaptive indexing mechanism alongside a flexible caching structure. Their conclusion regarding supporting these types of data-querying approaches is in alignment with the main concept of this thesis, which is the development of a general-purpose query language that runs on specific-purpose query execution systems.

Jaql [BEG+11], as another case, is a declarative scripting language for analyzing large semi-structured datasets in parallel with the use of Hadoop's MapReduce [DG08] framework. Jaql's data model is based on JSON. A Jaql script can start without a schema and evolve over time from a partial to a full-featured schema. Because, like JSON, Jaql uses a self-describing data format, the language is able to obtain metadata about the structure and data types of the underlying data. Jaql is a file-based solution and assumes that files are serialized in JSON format. The use of any specific file format means that scientists must engage in data conversion, particularly if they must load their own data into Jaql or obtain data from it.

In addition to the above-mentioned directives, the scope of this dissertation includes query languages, specifically the declarative languages. We introduce and discuss related query languages in Part II (Approach and Solution). The inclusion of these languages is due to the fact that we also suggest a declarative query language as a component of our solution. Addressing related query languages closer to the discussion and the proposed solution would facilitate the reader's comprehension of this dissertation.

In Chapter 3, we present the problem statement and translate it into a set of requirements. These requirements not only specify the problem but also clarify its boundaries. Thereafter, in Chapter 4, we present a summary, which includes a traceability matrix, to demonstrate how the studied related works satisfy these requirements.

3

Problem Statement

In this chapter, we elaborate on the objectives (see Section 1.3) of this thesis by identifying the functional and cross-cutting requirements of the proposed solution. This process of elaboration establishes the foundation for the specification, proof of concept, and evaluation of the hypothesis. In addition, the identified requirements serve as a basis for establishing the scope of the solution proposed in Part II, as well as in the system design and implementation (Part III). The extent to which these requirements are realized can serve as an indicator of the extent to which the objectives are achieved; traceability matrices in relevant chapters reflect this fact.

Our assumed target working environment involves a team of data workers who query and process heterogeneous data from different sources in order to obtain insights. The team members are assumed to conduct ad-hoc activities to experimentally and/or interactively decide on the queries, processing, and portions of data required for their purposes. They may also, either manually or by utilizing a workflow, use various tools to perform required tasks. In experimental science, data workers may require a small portion of the data that can be well located by means of a first-round exploration. The effort invested in such exploration is usually less than that of involved in reshaping data to a predefined schema and loading the data. This is particularly true when the data volume is much greater than the interested subset and the user has no idea of whether or when the rest of the data will be required. In addition, we assume that the data workers are not necessarily database experts; they usually decide on the importance of data attributes only after they have conducted initial explorations.

Many datasets are represented in file-based two or multi-dimensional arrays, e.g., CSV and NetCDF files. These groups of data are generally processed using specific-purpose tools or are converted to an equivalent relational or array-based dataset and then processed by SQL or a similar query language available in the hosting DBMSs. On the one hand, accessing data in-situ creates a strong dependency on the tools specialized to the file format in question; on the other hand, porting data to a general-purpose DBMS results in additional costs for data ETLs.

There should be a language that allows data workers to specify their data processes. Such processes may include elements for facilitating ETL, querying, analysis, visualization, and transport. The language should be declarative in order to isolate the data workers from the details of implementation. Having a language with appropriately designed elements that satisfies the data-processing requirements of various domains is the basic requirement. Such a language should

additionally allow for backward expressibility; i.e., data workers should be able to express all previous queries using the new language. The language should also be as neutral as possible in terms of the data formats used and functions offered by the target data sources. The syntax of such a language should be natural or close to the workers' daily working experience; in addition, it should be attractive to a diverse range of user groups, including computer programmers. Keeping the syntax close to well-known and frequently used syntaxes will minimize the learning curve and adoption time.

In the following sections, we introduce and specify the functional and non-functional requirements that together satisfy the objectives of our hypothesis. With regard to the requirements, we assume that any given data tuple is completely within a single database on a single machine. Any data replication is assumed to be concealed behind the corresponding system's API or query language. We also assume that the member databases of a federation are autonomous both in the manner in which they execute the queries shipped to them and how they return the results, including the tuple presentation.

3.1 Functional Requirements

Data workers use various tools in different stages of their research. In the majority of cases, data format inconsistency between such tools, lack of features and management functions among tools, and the need to integrate data from multiple data sources lead data workers to load data into feature-rich systems and query it from there. The ability to query bare files, e.g., CSVs, in the same fashion and with the same expressive power as offered by SQL without needing to first load them onto another system would encourage people to better utilize non-managed data and obtain results more rapidly. This would prove crucial during the early stages of research in which queries are exploratory and it is not clear whether data is appropriate for research goals and, if it is, which portions [AAA+16].

Requirement 1 (In-Situ Data Querying)

The system must minimize the need for data loading and duplicating. The language's expressive power must be available for all types of data organizations supported by the system, regardless of the degree of native management that they require. ETL operations should be written as part of the data-querying/analysis processes and executed on target data sources in an as native as possible manner. Alternative plans should also be available should the systems that hold the data are unable to perform the ETLs.

It is not possible to introduce a predefined set of data organizations to a system and to only provide the query language on top of them. Data workers deal with different datasets, which are

stored and formatted differently. In addition, over time, new data organizations are introduced and the existing ones may be upgraded. Therefore, the expressive power of the query language should not only be available for the default data organizations but should also be independent of them. Such independence would allow the system to be extensible in terms of integrating new data organizations.

Requirement 2 (Data Organization/Access Extensibility)

It should be possible to add support for new data organizations to the system at runtime.

Many systems retrieve data as is from sources and apply transformation in memory. For example, Spark SQL ingests external data sources into its *DataFrames* and provides relational operation on top of the data frames [AXL+15], while MapReduce-based systems, such as Hive [T+09] and HBase [Whi12], perform the transformations by creating intermediate files (which are, however, hidden from users). Generating these intermediate files consumes CPU time and disk space: CPU time is wasted because writing to disks, particularly in Hadoop environments, is highly IO bound. Disk space is wasted because the intermediate results of a job are not, by default, available to any other job.

Submitting a query to a federated system should overcome various types of heterogeneities, including structural (in terms of data models), representational (in terms of types and constraints), and accessibility differences (in terms of the expressiveness of query languages and the functionalities exposed through APIs). Retrieving data from such a federated system highlights the need to utilize different data access methods, various query languages, or even writing data retrieval programs. Overcoming the data access heterogeneity problems that exist among data sources is helpful, as it improves vendor independence, skill transfer, and collaboration. It also allows programmers and data workers to write more robust, portable, and reproducible processes; an example thereof would be the ability to write cross-data source joins that retrieve data from data sources with different levels of expressiveness and management.

Requirement 3 (Querying Heterogeneous Data Sources)

The system must be able to conceal structural, representational, and accessibility heterogeneities from end users. The system should also transparently execute compound queries that request data from multiple data sources.

Scientists of need to process and manage their data without being confronted with an excessive number of IT-related complications [AAB+09]. In addition, scientific data management and

processing should be decoupled from vendors, technologies and technical heterogeneity. A unified data access mechanism that allows data workers to author their processes independently of the underlying data sources' expressive powers would provide the required abstraction. Such *unifiedness* would provide a set of capabilities that are available on top of each and every data source. This abstraction would allow users to focus on solving their research problems instead of dealing with the technical issues associated with the management, transformation, and processing of data. Providing such a uniform set of functions at an abstract level would allow users and client applications to be more ignorant of the various functionalities (not) provided by different data sources. The formal specifications associated with data processing (e.g., querying or analysis) should be expressive enough to remain unchanged, independent of the target data organization. Its semantic should also remain independent of the target data organization. Furthermore, the formal specification should facilitate tool integration by allowing the tools to delegate the details of data querying to the language. The abstraction could be defined and set at various levels: syntax, semantics, execution, and presentation.

Requirement 4 (Unified Syntax)

The querying facility must be capable to providing a set of data access methods that are independent of the underlying data organizations (Definition 1.2). The access methods must also be independent of the capabilities of the data sources and/or database management systems that govern the data queried.

It is not enough that a unified syntax is used to formulate queries or analyses; there should also be semantic equivalence. Any language element should convey a clear and constant meaning that is independent of its local meaning in the target data sources.

Requirement 5 (Unified Semantics)

The meanings of query elements and functions should remain the same, regardless of their native meanings at corresponding data sources, in order to provide a means by which semantic independence can be achieved. Feature incompatibility, naming and data type inconsistencies, and/or a lack of features in the designated data sources should not affect the meaning of the constructs of the language.

One of the issues encountered by data workers, especially in collaborative efforts or multi-tool environments, is that there is no guarantee that all of the functions available in one system will also be available in another. Even if such functions exist, they may have inconsistent names,

parameters, and/or data types. To overcome these issues, all of the elements of the proposed language should be equally executable on all of the data sources.

Requirement 6 (Unified Execution)

Given a dataset, the result of executing any query against it must be independent of the data organization of the data source(s) that manage access to the dataset. This independence should include data model, data representation, and data source capability.

When a query is executed against data, its result set should not be bound to the physical schema; instead, the result should be tailored to the needs of the data workers in question or the subsequent processing steps. These needs should be specified and submitted alongside the query, using the same language. Having the result of a query in a different measurement system, a unit of measurement, or a resolution other than that of the original data are few cases of applications of result set schema definition.

Requirement 7 (Unified Result Presentation)

Regardless of the original data organization, the schema of the result set should be determined by user's requirements. The requirements should be captured by the constructs of the query language.

A significant part of scientists' work is dedicated to accessing, visualizing, integrating, and analyzing data that is possibly obtained from a wide range of heterogeneous sources, e.g., observations, sensors, databases, files, and/or previous processes. The data usually needs to go through a series of preparation steps, namely cleansing, data type and/or format conversion, decomposition, and aggregation. In addition to the third-party utilities, data workers tend to develop specific parts of their work by themselves, and they spend a remarkable portion of their time retrofitting data into formats that these tools understand [PJR+11]. For example, the subsequent processing steps in a collaborative workflow may rely on the data created by earlier steps, but, more often than not, this data needs to be reformatted or transformed in some way. These kinds of tasks end in a series of ETL operations, usually using third-party utilities.

Providing a facility that allows for the desired ETLs to be specified in the same language that is used for analysis would not only reduce data workers' burdens but also render the entire process of analysis easier to reproduce. Having ETL specification integrated into the language relieves the users of the need to manually perform data integration tasks and also allows them to avoid

data duplication. Furthermore, it would provide the room required to easily define higher level conceptual entities and write queries against those entities.

It is worth reminding the reader that research is, by its nature, exploratory; hence, data and analyses change over time. Being able to define the same conceptual schemas over varying physical data structures [CMZ08, ABML09] would improve resiliency and hence result in greater data independence. For example, a schema should function equally on both an SQL table and a spreadsheet, provided that the required data items are available in both. In addition, having multiple schemas on the same data would provide an extra degree of flexibility to data workers, allowing them to declare analysis-specific schemas without falling into the ETL pipelines to prepare data for the various analyses. Having such an abstract schema implies the need for a high-level type of system with a similar degree of data independence.

Requirement 8 (Virtual Schema Definition)

It should be possible to define a desired virtual schema and apply it on the actual data without the need to alter the original data. The virtual schema must allow for complex mappings from the data attributes to the virtual attributes; furthermore, it should overcome the heterogeneity in the data types of various data sources. It should also be possible to define and apply multiple virtual schemas to a single dataset. In addition, it should be possible to incorporate a single virtual schema in different queries that potentially access data from various heterogeneous sources. In other words, virtual schemas should not be bound by the data organization but instead only to the data attributes that they require from the datasets.

Virtual schemas are the appropriate mediums for defining data transformations, e.g., when the physical schema of the original data does not satisfy the requirements of a researcher or an algorithm. Merging or splitting columns, computing derived values, aggregating, and temporal resolution alignment are common examples of data transformations that can easily be handled by virtual schemas.

Requirement 9 (Easy Transformation)

It should be possible to easily express data transformations on the data items of the target data sources. These transformations should be expressed in a formal and reproducible manner.

Data conversion, transformation, and aggregation are among the most frequently used operations. Mathematical functions, string manipulation, conversion of units of measurements, merging data items, and format conversion, e.g., date/time, are among the common transformations used by data workers on a daily basis.

Requirement 10 (Built-in Functions)

The system should have a set of built-in functions in order to perform popular operations on data values. Such function may include mathematical, statistical, string manipulation, date/time and unit of measurement conversion.

Providing a collection of these types of operations, although necessary, is not satisfactory. Different scientific domains require different functions; even individuals may require different set of functions for various analyses. Allowing new tuple-based and aggregation functions to be added to the language is a mandatory requirement, as it would allow the system to remain both domain-agnostic and useful.

Requirement 11 (Function/Operation Extensibility)

The system should allow for the development and registration of third party aggregate and non-aggregate functions. Upon registration, they should be available to the queries. Requirement 6 should remain satisfied after the registration of any function extension.

Data independence [Cod70] in an DBMS is a mechanism to isolate data consumers from the details of data storage, organization, and retrieval. An application should not become involved with these issues, as there is no difference in the operations carried out against the data. In our case, the abstraction defined by a virtual schema should apply to all of the constructs of the query language; for example, a filtering predicate should operate on the attributes defined by the query's virtual schema.

Requirement 12 (Data Independency)

Users should remain isolated from changes in data organization and data sources in such a manner that they issue queries against virtual schemas and obtain results represented in virtual schemas.

Result sets should also obey the schemas. The attributes presented in a query result set should have been defined by a virtual schema and not keep track of the original data item(s) that their value was obtained from. However, the result sets are likely to be presented or communicated in different ways: For example, one could request a query result to be presented in JSON in order to feed it into a MongoDB database. The same result set can be serialized in the form of a CSV file to be used by an R script. In addition, scientific work normally includes visualized representations of processed data; hence, every effort to develop a general data-processing tool must also take visualization into account.

Requirement 13 (Polymorphic Resultset Presentation)

Query results should be presented in different ways upon request. Such presentation could take the form of a conventional tabular form for on-screen display, an XML or a JSON for system interoperability, or a human-oriented visualization, such as a chart.

Based on a similar argument to that used for the extensibility of data organization, the result set presentations should also be extensible. It is necessary that the system should have a set of built-in presentation methods; however, it should enable third parties to develop and plug their own presentation methods into the system. Such presentation methods should be accessible through the query language.

Requirement 14 (Resultset Presentation Extensibility)

Beyond the built-in result set presentation methods, the system should allow for adding new presentation methods. The newly added methods should be seamlessly accessible via the queries and produce the intended presentation upon query execution.

The experimental and iterative nature of scientific querying often results in multiple versions of a particular dataset; for example, a data cleansing process generates a new version of a raw dataset. An error correction procedure may compensate for measurement device errors and create another version. These versions may or may not preserve the original format, units of measurement, or attributes. Hence, the language should provide a version-aware querying mechanism that allows data workers to freely choose a schema relevant to the version of interest and start querying it. In addition, Scientific Workflow Management Systems (SWMSs) maintain snapshots of data for provenance reasons [ABML09].

Requirement 15 (Version Aware Querying)

The system should provide a mechanism for querying data from a specific version of a designated dataset. A desired (and relevant) virtual schema may be applied to the queried version.

3.2 Non-functional Requirements

Typically, data is processed in a series of steps, using a range of possibly different tools in a multi-tool or a workflow environment [L+06]. Such tools generally use different data structures, meaning that users frequently need to transform data from one form to another. In these scenarios, the transformations are usually performed by means of the import/export operations provided by the collaborating tools. In many cases, e.g., workflow systems, users are required to write transformation programs or introduce additional steps to the flow solely for the purposes of for data transformation. Making system functionalities and language features available to external tools would improve tool integration and reduce workflow complexities.

Requirement 16 (Tool Integration)

system features should be available as public APIs to be integrated into third party systems such as SWMSs.

A positive effect of fulfilling such a requirement would be that data-processing tools could be seamlessly integrated and rely on the capabilities of this language only. In addition, they could delegate the details of data ETL tasks to it. Furthermore, SWMSs could orchestrate their complex pipelines of data-processing tasks with less effort and fewer internal data transformation steps.

Although system features would be exposed via the language or the APIs, end users are typically more comfortable with a workbench that features an easy-to-use Graphical User Interface (GUI). Such a workbench supports users in organizing their query/analysis scripts, data, and configuration. It also facilitates editing, e.g., by means of syntax highlighting, syntax and semantic error reporting, and presenting query results.

Requirement 17 (IDE-based User Interaction)

The system should be available as a standalone GUI-based desktop system. Users should be able to author, submit, and execute queries and retrieve results from the underlying data sources. The system should be developed with operating system (OS) portability in mind, as individual end-users are assumed to utilize different OSs.

In addition to the above-mentioned requirements, the system should be both easy to use and useful. Our expectation regarding the usability and usefulness of the system are specified in Requirements 18 and 19.

Requirement 18 (Ease of Use)

1. **Ease of Sharing**: The system should make it easy for users to share analytical processes, and it should be possible to execute shared processes in different environments with minimal changes. The potential changes required should be related to information concerning data source connections, credentials, and/or the physical schema of the data in question;
2. **Non-Disclosure of Sensitive Information**: The queries that process data should be kept separate from the credentials required to access the data;
3. **Minimizing Total Cost of Ownership (TCO)**[a]: The TCO of the system should be low in order to render it attractive to data workers, as they are largely researchers who struggle to obtain financial resources;
4. **Minimizing the Learning Curve**: The user-facing features of the system should follow relevant best practices to reduce learning effort and duration; and
5. **Ease of Development**: Improving system functionalities, as well as writing programs using the system, should be straightforward for data workers and programmers [AAB+09].

[a] https://en.wikipedia.org/wiki/Total_cost_of_ownership

Requirement 19 (Usefulness)

1. **Expressiveness**: The query interface should be expressive enough to allow for the authoring and execution of at least the SQL core queries[a].
2. **Performance**: The query execution time of the system must be close to or comparable with that of RDBMSs on the same or similar datasets. In additions, the system's user interface should be responsive; and
3. **Scalability**: The system's performance should remain linear with the volume of data; exponential execution time is not acceptable.

[a] The core queries are defined in Part II.

4

Summary of Part I

Thus far, we have declared our goal as being to mitigate data access heterogeneity through query virtualization, on-the-fly transformation, and federated execution. In order to achieve this goal, we formulated a hypothesis in Section 1.3 concerning the existence, feasibility, and usefulness of a universal query language. By studying existing systems and approaches, as well as roadmaps for the future, we specified and established the boundaries of our hypothesis to the list of requirements as stated in Chapter 3 (Problem Statement). In this summary, we prioritize the requirements with reference to the contributions that they make toward achieving our goal. For this purpose, we first demonstrate how other related works have satisfied these requirements and then present our prioritized list of requirements.

Traceability Matrix 4.1 shows how the systems studied in Chapter 2 (Background and Related Work) fulfill the requirements. This information is important because 1) it indicates the gap between those systems and that which we propose; 2) it provides a basic understanding of which systems have implemented which requirements better, making it possible to learn from them; and 3) it can be used to validate the requirements.

The matrix shows that requirements related to heterogeneity are not widely addressed. For example, the ability to query heterogeneous data sources and support for data organization extensibility are limited to federation-based systems, while there is even less support for in-situ data querying. This is due to the fact that the majority of DBMSs operate on a designated data model and its related calculus. DBMSs usually perform query optimization for the assumed data model and collect statistics and historic data accordingly.

In Part II (Approach and Solution), we focus on the requirements that, on the one hand, make the greatest scientific contributions to this dissertation and, on the other hand, fill the gaps in the current state-of-the-art in the field of database domains. Supporting multiple data organizations, querying data in-situ, providing virtual schemas, guaranteeing unified execution even in the presence of functionality shortage, and polymorphic result set presentation are our top priorities in terms of requirements. While this prioritized list serves as guidance in identifying the solution components, it is not the only source, as we also take into accounts all of the other requirements. However, the architecture of the solution is built around the prioritized requirements.

	RDBMS	FDBMS	Polystore	Array DBMS	NoSQL	NoDB
In-Situ Data Querying	✗	✓	✗	✗	✗	✓
Data Organization/Access Extensibility	✗	✓	✓	✗	✗	✓
Querying Heterogeneous Data Sources	✗	✓	✓	✗	✗	✓
Unified Syntax	✓	✓	✗	✓	✓	✓
Unified Semantics	✓	✓	✗	✓	✓	✓
Unified Execution	✓	✗	✗	✓	✓	✗
Unified Result Presentation	✓	✓	✓	✓	✓	✓
Virtual Schema Definition	✗	✓	✗	✗	✗	✗
Easy Transformation	✓	✓	✓	✓	✓	✓
Built-in Functions	✓	✓	✓	✓	✓	✓
Function/Operation Extensibility	✓	✓	✓	✓	✓	✓
Data Independency	✓	✓	✗	✓	✓	✓
Polymorphic Resultset Presentation	✗	✗	✗	✗	✓	✗
Resultset Presentation Extensibility	✗	✗	✗	✗	✗	✗
Version Aware Querying	✗	✗	✗	✗	✗	✗
Tool Integration	✓	✓	✓	✓	✓	✗
IDE-based User Interaction	✓	✓	✓	✓	✓	✗

Table 4.1.: The related work (see Chapter 2) satisfy the requirements differently. The matrix shows a general overview in that each column is a representative of many systems in its category.

Part II.

Approach and Solution

This part proposes and describes the main elements of a solution intended to fulfill the requirements discussed in Chapter 3. It begins by outlining a solution architecture in Chapter 5. The architecture introduces three fundamental components: query declaration, transformation, and execution. Query declaration (Chapter 6) formulates the requirements into a declarative query language that is unified in syntax, semantics, and execution. Query transformation (Chapter 7) specifies and explains the techniques used to convert the queries into appropriate computation models, allowing them to be executed against designated data sources. This chapter also elaborates on the solutions proposed for dealing with queries that access heterogeneous data sources, query rewriting, and data type consolidation. Chapter 8 then explores how the transformed and complemented queries might be executed at the end of the pipeline. Query execution is also responsible for returning the queries' result sets to the client in the format they requested. The extent to which the solution satisfies the requirements, in addition to its limitations and achievements, are summarized in Chapter 9.

5

Overview of the Solution

In Part I, we described a situation in which an input query was defined to retrieve data from multiple data sources in an in-situ manner. The data sources had different data models, exposed their own computation models, and were not consistent in terms of the functions and operations that they provided; indeed, they did not even ensure that they would execute the entire input query shipped to them.

In this chapter, we illustrate the "big picture" of an ideal solution and then limit it based on our requirements. We design a high-level architecture and introduce the needed components; in addition, we specify the roles and responsibilities of the components, as well as the interactions between them.

Data integration has been extensively studied by scholars attempting to manage data heterogeneity. There are two classical approaches to data integration: *materialized* and *virtual* integration [DHI12], both originally developed with business applications in mind. Materialized data integration is a process designed to extract, transform, and load data from a set of data sources into a single database in which queries are then answered over the unified data. Virtual integration features a logical access layer on top of a set of data sources in order to conceal data heterogeneity from applications and users without loading data in advance [SL90, DHI12]. In virtual integration, queries, as opposed to data, are transformed and executed on the corresponding data sources. Partial results are integrated by a mediator at query time.

Materialized integration has a high upfront cost and is not suitable when data sources change frequently. As such, it is often not well-suited for scientific work. On the other hand, most virtual integration work assumes that the data sources are relational databases, or at least support a relational query interface. The focus of attention is then schema mapping and query transformation. When it comes to scientific data, however, the data sources are often text files, Excel spreadsheets, or the proprietary formats used by recording instruments; in consequence, virtual integration is not possible.

The fundamental difficulty is that data is heterogeneous not only in syntax and structure but also in the manner in which it is accessed and queried. While certain data may reside in feature-rich DBMS and can be accessed using declarative queries, other data is processed by MapReduce programs, utilizing a procedural computation model [DG08]. Furthermore, many

sensor-generated datasets are stored in the form of CSV files that lack well-established formatting standards and basic data management features. Finally, different RDBMS products vary in terms of syntax, conformance to SQL standards, and features supported.

Despite the fact that our problem is different, namely, data access heterogeneity, we seek inspiration from the large body of previous work that has been conducted on data integration. Some researchers have proposed the concept of *data virtualization* [K$^+$15, ABB$^+$12] to address this challenge. By providing a framework that allows queries to be posed against raw data, such a system could permit the use of data sources with heterogeneous capabilities. However, there are additional important aspects of data access heterogeneity that should be addressed, namely heterogeneous joins, heterogeneous query planning, and heterogeneous result representation [AAB$^+$17].

Transforming and importing data into a centralized store does indeed represent one approach to data integration; however, it is usually not the preferred method. In many circumstances, the preferred solution is to transform the query against the target unified database to a set of queries against the component source databases; this is the technique that is primarily used in Federated Database Systems (FDBSs) [SL90].

The primary challenge with FDBSs is that they are usually designed for static integration environments, meaning that the virtual database that is exposed to users is aware of and takes advantage of bindings, mappings, and available transformations. However, our problem statement imposes Requirement 3 (Querying Heterogeneous Data Sources) on any potential solution. This requirement specifies that component databases might have multi-dimensional heterogeneities. It creates a dynamic and loosely integrated environment in which neither data organization (Definition 1.2) nor system capabilities remain static for long periods of time. It is worth emphasizing that we are targeting a research environment that features volatile data and ad hoc queries; hence, a case-based integration would be a better fit. Furthermore, our solution is required to query data in-situ (Requirement 1: In-Situ Data Querying) in order to eliminate the need for data duplication, transformation, and loading. This requirement imposes restrictions on data integration in that not only must the original (and possibly raw) data be used for querying but the original data should not be permanently altered for integration purposes.

To overcome these barriers and effectively address data access heterogeneity, we propose the concept of *query virtualization*, which uses federated agents to transform submitted queries to native counterparts. Federative query virtualization permits much greater flexibility and support for heterogeneity than data virtualization controlled by a central authority. We combine techniques taken from data integration and federated systems to establish the foundation of our query virtualization solution. We use query transformation techniques to transform data organization-independent (Definition 1.2) input queries into a set of native queries to be executed against the designated component data sources of the federation. Compositional queries can then combine the partial results obtained from the components to shape the final result sets.

Transforming an input query to its native counterpart for execution on a component data source requires a system to be aware of the capabilities of that engaged component. This information allows the system to generate a query that is executable by the designated component, although the

component may only partially fulfill the requirements of the input query. The system may consider complementing the residual work that could not be performed by the target data source. In many cases, e.g., in CSV files, there are no or only limited querying capability available. Hence, the system must translate the input query into a set of operations in order to execute the query and obtain its result. This technique requires that the system is able to synthesize and compile code on-the-fly. Overall, the system should be capable of complying to the computation models used by the designated data sources by transforming input queries to either query languages, operational procedures, or API calls that those data sources accept. The system should compose a set of specific programs tailored for execution against the target data organizations, compile them to appropriate executable units, and then run the units on the designated data sources or data. This approach can be referred to as "database on-the-fly", as the data access functionality is dynamically generated based on the data organization and according to the query requirements.

In order to identify the components and their roles and lay a foundation for a solution, we establish a reference architecture. A reference architecture is required in order to determine the architecturally important components of the solution and the manner in which they interact with each other. Each architectural component deals with a subset of the problem and makes it possible to focus on higher level aspects of the system, e.g., the roles assigned to the components and the flow of control and data. It also assists in establishing and satisfying the requirements identified in Chapter 3. Furthermore, we specify a set of design principles and construct the architectural elements on top them. In the remainder of this chapter, we elaborate on these prerequisites in terms of architectural design, study a number of relevant architectures, and finally propose ours.

Any given query undergoes a series of operations in order to yield its result set. Broadly summarized, the query is transformed into an appropriate computational model, which is then executed on a set of designated data sources. The possible partial results obtained from each data source are combined to shape the final result set, which in turn is returned to the requesting agent. This flow indicates the following three major building blocks:

1. *Query Declaration:* Authoring the data requirements in terms of queries and parsing the input queries in order to validate their syntax and semantics are the main duties of query declaration. Additionally, it may build internal query model such as Abstract Syntax Trees (ASTs) and submit these models for execution;

2. *Query Transformation:* This includes all of the activities required for the construction of a set of optimized concrete computation models that are tailored to be run on the designated target data sources; and

3. *Query Execution:* This process is intended to build executable units from the computation models, dispatch the units to the data sources and request for execution, collect and combine the results, and reformat the results according to the query requirements.

There are a multitude of architectural designs for FDBMSs [HM85, SL90, KK92], heterogeneous data systems [NRSW99, CDA01, KFY+02], query transformation [BDH+95, ALW+06, Ram12], and query shipping [FJK96, Sah02, RBHS04, LP08], each of which are aligned towards

particular aspects of the entire problem that we intend to address. The majority of the existing solutions attempt to solve the problem at hand for a specific domain. By taking advantage of the existing systems, our reference architecture provides elements intended to address the needs of the above-mentioned building blocks and satisfy the requirements described in Chapter 3 (Problem Statement). We construct our solution based on the following three main components:

1. An abstract *query language* that allows for schema and query formulation in a data-organization-agnostic (in terms of source, format, serialization) manner (Chapter 6). Such a unified query language is fundamental to the concept of query virtualization that we are striving towards. Users can write their queries and applications using this query language and rely on the system to perform the necessary transformations required to evaluate the specified queries against heterogeneous data sources;

2. A set of data organization-specific *adapters* that are responsible for transforming and executing queries on their corresponding data sources (Chapter 7). Adapters additionally participate in query rewriting and complementing; and

3. A *query execution engine* that orchestrates adapter selection, query rewriting and dispatching, and result assembly (Chapter 8). Query execution is constructed on top of the above-mentioned two end-points, namely the (abstract) query language and the (concrete) capabilities of the underlying sources accessible via adapters.

Figure 5.1 illustrates our architectural realization of the components identified above. In brief, the architecture consists of three modules: client, agent, and data access, which are described in Chapter 10 (Implementation).

Upon starting up, the runtime system is activated. It is responsible for the configuration, registration, and instantiation of the query execution engine, as well as other components. The system interacts with its clients via a set of APIs that provide the functionality required to exchange queries and results. When a set of queries is submitted, the runtime system selects the active query execution engine, launches it, and then passes the queries to it. A plug-in mechanism allows for swapping the query engine if needed. The query engine accepts the input queries, orchestrates their execution, and returns the results. The APIs provide a mechanism for various types of clients, such as desktop workbenches, third-party tools, and remote services, to interact with the system.

Definition 5.1: DST *A Described Syntax Tree (DST) is an Abstract Syntax Tree (AST) annotated with metadata, e.g., the data types and constraints that are extracted or inferred from the queries or the data. A DST is an intermediate representation of its corresponding query; the nodes of a DST are operators of its associated query. Nodes may be annotated by additional data, such as cost indicators, allowing optimizer, transformer, and executer to benefit from such information.*

During the semantic analysis phase, the parser builds a DST (Definition 5.1) for each query statement and adds them to the ASG (Definition 5.2). The engine optimizes the queries using its rule-based optimizer (see Section 7.4) and then, from the pool of registered adapters, selects

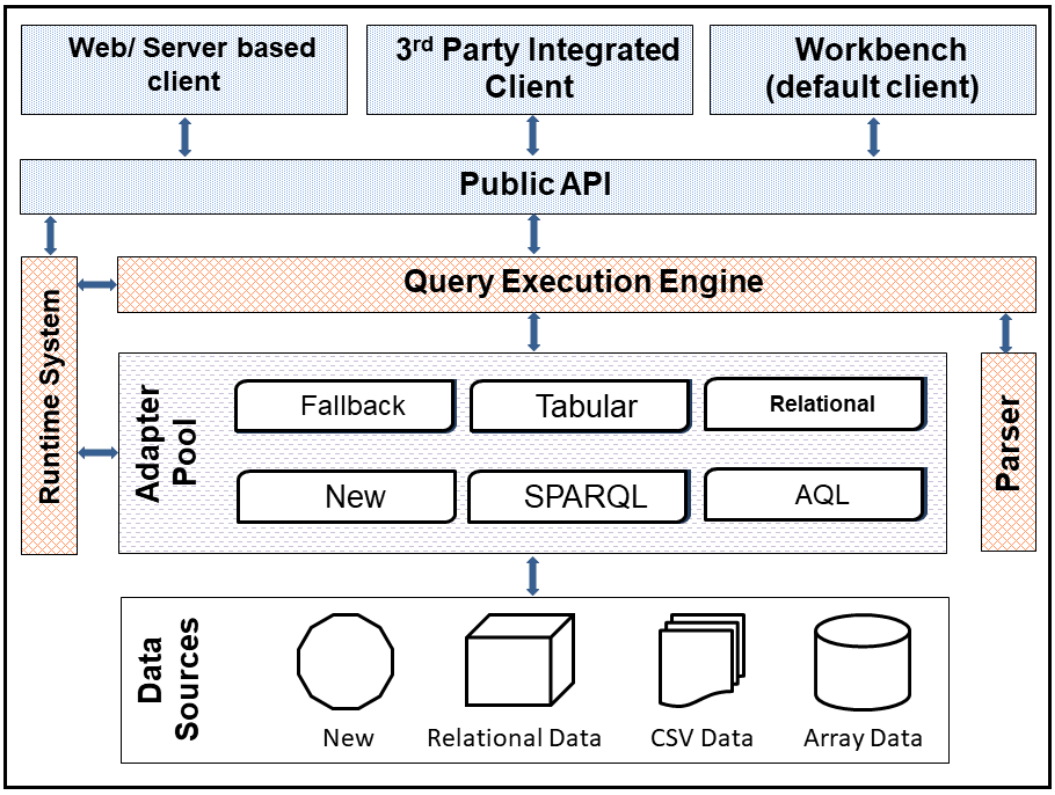

Figure 5.1.: The overall system architecture, showing the interactions between the client, the agent, and the data access modules. The client module consists of the dot-patterned components, while the grid-patterned components are the agent module. All of the other components shape the data access module.

the most suitable adapter for transforming the query into its target computation model. The adapter selection process is based on a negotiation algorithm (see Section 8.1.2) in that the engine compares the requirements of queries with the capabilities of the adapters. When an adapter is chosen, the engine determines which parts of each query are executable on its designated data source.

Adapters accept the DSTs and transform them into their native counterparts to be executed against the corresponding data sources. The output of such transformation depends on the organization of the target data: For example, an RDBMS adapter may transform the input query to a vendor-specific SQL, while a CSV adapter generates a sequence of operations to read, parse, materialize, and filter the records upon execution.

Definition 5.2: ASG *An Annotated Syntax Graph is a directed acyclic graph in which nodes are DSTs and edges indicate inter-statement dependency.*
It is a strongly typed and fully linked representation of the input process. An inter-statement dependency is defined by a data flow, so that if a statement $s1$ *requires data from another statement* $s2$*, then* $s1$ *is dependent upon* $s2$*.*

When all of the adapters return the transformations, the query execution engine packages them as jobs, which are the units of execution. It then constructs an execution pipeline per input query and puts the jobs into the pipeline (see Section 8.1.4). After compiling the jobs, the engine dispatches them to their corresponding adapters for execution. The dispatching algorithm can break the ASG down into a set of disjoint sub-graphs for parallel execution. The adapters execute the jobs on the data sources and return the result sets. The engine then obtains the results and feeds the dependent queries, allowing them to proceed. The final result of each query would be ready at the end of the execution of its pipeline. In addition, the engine assembles the final result sets of heterogeneous queries (see Section 8.1.4.1) to appropriate presentation models [JCE+07].

Those query requirements that were not fulfilled by their target adapters are identified and added to the ASG as complementing DSTs (see Section 7.3). These complementing queries are executed on an adapter specifically designed for this purpose. The engine rewrites the original queries accordingly in order to eliminate the complemented requirements and marks the complementing query as depending upon the rewritten ones. Knowledge of the query capabilities of the data sources or adapters is essential for rewriting queries.

In the following chapters in this part, we describe how queries are declared (Chapter 6), transformed (Chapter 7), and executed (Chapter 8). In each chapter, we explain how the relevant requirements described in Chapter 3 are realized. To do so, we define and classify partial solutions intended to realize the relevant requirements and introduce them as *solution features* (or *features* for short). Every feature is a cross-cut of one or more requirements, meaning that, if the feature is realized, then all of the associated requirements are assumed to be partially fulfilled. If all of the features $(F_1..F_n)$ associated with a requirement R_i are realized, then the requirement is assumed to be fulfilled. We infer the satisfaction of the hypothesis from the overall fulfillment of requirements, and illustrate these dependencies using tractability matrices in Chapter 9.

6

Query Declaration

In Chapter 5 (Overview of the Solution), we proposed a reference architecture that provided the fundamental building blocks for the three major functions of the system, namely query *declaration*, *transformation*, and *execution*. The objective of this chapter is to introduce and define a uniform data access mechanism through the development of a query language. The query language is intended to provide the expressive power required to access various data organization (Definition 1.2). We argue that, by abstracting and isolating the language from implementation and execution semantics, it is possible to develop a single unified declarative language with a sufficient number of constructs to allow it to transform its sentences to the semantically valid computational model of any underlying database. This chapter is dedicated to the design of such a language. We define the components required to declare queries that are able to retrieve, transform, and manipulate data.

Query declaration must satisfy various requirements: Requirement 3 (Querying Heterogeneous Data Sources) states that data of interest is heterogeneous and therefore it should be possible to formulate queries on such data. In addition to the selected data organizations, Requirement 2 (Data Organization/Access Extensibility) requires a mechanism to allow adding support for new data organizations. Requirement 15 (Version Aware Querying) requires that if data sources keep track of various versions of data, the query declaration needs to provide facilities to make designated versions of the data queryable.

Query declaration should provide a syntactically (Requirement 4: Unified Syntax) and semantically (Requirement 5: Unified Semantics) unified means of posing queries on data in such a manner that the two are not affected by the data's organization (Requirement 12: Data Independency). The query language should allow its users to define the desired view of their raw data (Requirement 8: Virtual Schema Definition), meaning that the structure of the query results will be determined by the defined schema and presented in a unified manner (Requirement 7: Unified Result Presentation). It is worth mentioning that such a unified method of presenting results should not be interpreted as being restricted to one way only; users should rather have the option to present results in various ways (Requirement 13: Polymorphic Resultset Presentation), and those ways must be extensible (Requirement 14: Result set Presentation Extensibility).

Users should be able to transform data of interest to virtual schemas (Requirement 9: Easy Transformation), allowing them to query the virtual representation of the data. Built-in functions

(Requirement 10: Built-in Functions) should be integrated into the query language and be easily accessible for transformation and/or aggregation purposes. These functions should be extensible (Requirement 11: Function/Operation Extensibility) to make room for both domain-specific implementations and customization.

We choose to develop a declarative query language as the system's query-authoring mechanism. This language allows clients to formulate their own data retrieval, transformation, and manipulation needs and submit them to the system. Since SQL is widely used for complex data manipulation, at first glance it seems a natural choice for our unified query language. However, standard SQL does not have the extensibility required to support the diversity of data sources and type systems that we anticipate. Furthermore, SQL is a large language with many features that are not of high priority for our use-case. In light of these observations, we have developed our own unified query language (QUery-In-Situ (QUIS)) [CKR12], which can be considered an extension to the SQL core.

`QUIS` expressive power allows its statements to be translated to appropriate counterpart elements in other languages and systems, e.g., SQL, AQL [LMW96], and SPARQL [HS13]. In cases such as CSV, TSV and spreadsheets, where no or limited data source functionalities are available, the language provides enough information to the adapters to enable them to build appropriate native models required to compute a result set.

In the remainder of the chapter, we explain the general programming paradigm in Section 6.1, which is followed by a discussion of our choice of programming in Section 6.2 and the tools selected for writing the language (the meta-language), as well as for lexical and syntactical analysis, in Section 6.3. Thereafter, in Section 6.4, we consider a number of well-known related languages. Finally, we introduce the `QUIS`'s language features, design, and grammar in Section 6.5.

6.1 Programming Paradigm

We begin this section by briefly introducing a number of the key concepts of computer programming languages that are used throughout of this dissertation. We first present a number of definitions and then explain the grammar chosen to formulate the language. Thereafter, we describe the decisions that we made concerning the grammar's meta-languages, their varieties and syntaxes, and the lexer and parser generation tools.

A language is a set of valid sentences; sentences are composed of phrases, which in turn are composed of sub-phrases, which can be broken down into the linguistic building blocks known as words or, more abstractly, as tokens. A token, which is a vocabulary symbol used in a language, can represent a category of symbols such as identifiers and keywords. A programming language is a notation for writing programs, which are the specifications of a computation or an algorithm [Aab04]. More generally, a programming language may describe the computations performed on a particular machine. The machine can be a real one, e.g., an Intel CPU, a soft machine, e.g., a Java virtual machine, or a query execution engine.

Programming languages usually benefit from a level of abstraction when defining and manipulating data structures or controlling the flow of execution. The theory of computation classifies languages by the computations they are capable of expressing. The description of a programming language is usually split into the two components of syntax and semantics. The syntax of a programming language is the set of rules that define the language membership and is concerned with the appearance and structure of programs [Aab04]. The syntax determines whether a stream of letters (a token) or a stream of tokens (a phrase) is a valid member of the corresponding language. Syntax is contrasted with semantics, as the latter is a set of rules and algorithms that determine whether a phrase is meaningful it its context. For example, in the Java language, a local variable should be defined before its first use. Semantic processing generally comes after syntactic processing, but they can be done concurrently or in an interlacing manner if necessary.

The syntax of a language is formally defined by a grammar, which is usually defined using a combination of regular expressions and a variation of Backus–Naur Form (BNF) to inductively specify productions and terminal symbols [FWH08]. Each production has a left-hand side nonterminal symbol and a right-hand side, which consists of terminals and non-terminals. The right-hand side specifies how the members of the production can be constructed; in other words, it determines the valid phrases that a production can produce. The terminals are defined using regular expressions that describe how input characters should be grouped together to form tokens. The set of tokens is considered as the alphabet of the grammar [Aab04].

In order to recognize the membership of an input stream to a grammar, the stream should undergo a chain of steps of lexical, syntactical, and, in some cases, semantic analyses. In a general scenario, the input stream is fed into a lexical analyzer (also known as lexer or tokenizer) in order to create, based on the lexical rules of a grammar, a stream of valid tokens out of it. The token stream is then passed to a syntactic analyzer, a parser, which applies the nonterminal production rules on the tokens and recognizes the phrases and sentences. The output of the parser is represented as a syntax tree, also known as parse tree. A syntax tree is a tree in which input sentences are structured as sub-trees. The name of the sub-tree roots are the corresponding rule names, while the leaves are the tokens presented in the input and detected during the lexical analysis. Semantic analysis, also referred to as context-sensitive analysis, is a process used to analyze the parse tree in order to gather information necessary to validate (and annotate) the parse tree [Aab04]. It usually includes data type, declaration order, and flow of data and/or control checking.

Parsers are usually built to match production rules either top-down or bottom-up. A top-down parser begins with the start symbol of the grammar and uses the productions to generate a string that matches the input token stream. A bottom-up parser, in contrast, attempts to match the input with the right-hand side of a production and, when a match is found, replaces the portion of the matched input with the left-hand side of the production [Aab04].

Both top-down and bottom-up parsers use a variety of approaches and implementations, based on factors such as performance, memory usage, and the size of backtracking and lookahead[1]. A

[1] Lookahead is a mechanism used by parsers to look ahead for (but not consume) a number of tokens in order to decide upon a viable alternative. The number of tokens the parser can look ahead varies from parser to parser and affects its performance, flexibility, and the class of grammars it can parse.

Left to right, Leftmost derivation (LL) parser is able to analyze a subset of context-free languages in a top-down manner. The first L in the name refers to the fact that it parses the input from left to right. The second L implies that the parser performs the left-most derivation of the sentence, hence LL. When parsing a sentence, the parser may need to test different possible alternatives of productions. In order to test these alternatives, the parser looks at, but does not consume, the tokens ahead of its current token. Depending on how many lookahead tokens are required for the parser to recognize the input, it is called LL(k), in which k is the number of tokens that the parser will look ahead when parsing a sentence. If such a parser exists for a certain grammar and it can parse sentences of that grammar without backtracking, it is called an LL(k) grammar [RS70]. If there is no limit on the number of look-ahead tokens, the parser is called LL(*) [PF11]. LL grammars can also be parsed by recursive-descent parsers.

Recursive-descent parsers are a type of top-down parser implementation. They are a collection of recursive procedures in which each procedure represents one of the productions in the grammar. Parsing begins at the root of a parse tree and proceeds towards the leaves. When the parser matches a production, it calls the corresponding procedure to consume the input and call the sub-rules, including the original production itself if necessary. In recursive-descent parsing, the structure of the resulting parser program closely mirrors that of the grammar it recognizes. This parsing technique does not need to backtrack if the grammar is LL(k), for which a positive integer k exists that allows the parser to decide which production to use by examining only the next k tokens of the input [Par13].

Because the syntax of a program in a language usually has a nested or tree-like structure, recursion will be at the core of parsing techniques [FWH08]. One of the most frequently used recursions is left recursion. Left recursion refers to a situation in which a rule invokes itself at the start of an alternative. Arithmetic and logic expressions are examples of left recursion. Modeling these kind of expressions in an LL grammar would require developing a set of sub-rules intended to eliminate recursion and apply precedence. Choosing an LL parser that accepts left-recursion as a first class citizen eases the process of designing arithmetic and logic expressions and makes them more legible and maintainable.

6.2 Choice of Programming

In query languages, the exact procedure of accessing, filtering, materializing, and returning results is not of interest to query clients. These tasks remain the job of query execution engines and their associated optimizers, which may differ from one implementation to another. John W. Lloyd [Llo94] believes that declarative programming has made a significant contribution toward improving programmer productivity. He defines a declarative programming as a method of building the structure and elements of computer programs that expresses the logic of computation without describing its control flow. The benefit of describing a program declaratively is that the programmer describes the desired result (solution) without having to be concerned about the details of how to explain the control flow, how to choose an optimized algorithm or

how to avoid side effects[2] [Han07]. All of these aspects are left up to the language's implementation, making make room for dynamic execution planning, optimization, and parallelism. The greater the extent to which a query language is procedure-ignorant, the easier it is to optimize and use. Database query languages are among the most well-known and successful declarative programing languages.

Declarative programs are made up of expressions, not commands. An expression is any valid sentence in the language that returns a value. The expressions should have referential transparency [Han07], which implies that any expression can be replaced by its return value. This property allows language implementations to consider expression substitution, memoization[3], the elimination of common sub-expression, lazy evaluation, or parallelization.

In addition to referential transparency, a declarative operation should satisfy all of the following conditions [VRH04]:

1. It should be independent, which means it does not depend on any execution state outside of itself. Whenever an operation is called with the same arguments, it returns the same results, independent of any other computation state;

2. It should be stateless (immutable), which implies it has no internal execution state that is remembered between calls; and

3. It should be deterministic, which means that it will always give the same results when given the same arguments.

Roy and Haridi argue that declarative languages should be compositional, in that programs consist of components that can be written individually, tested, and proven correct independently of other components and of their own past histories (previous calls) [VRH04].

6.3 Choice of Meta-language and Tools

Despite the enormous number of languages that have been invented, there are relatively few fundamental language patterns. Token order and token dependency are among the preliminary expectations of any language designer. There are also some common and reusable elements such as identifiers, integers, and strings, that can be easily used in any other language. In general, the patterns can be categorized into the following classes:

Sequence: An ordered list of elements of the language. $x\ y\ z$;

[2] A function or expression is said to have a side effect if it modifies some state or has an observable interaction with calling functions or the outside world. `http://en.wikipedia.org/wiki/Side_effect_(computer_science)`

[3] Memoization is an optimization technique primarily used to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again. `http://en.wikipedia.org/wiki/Memoization`

Choice: Choice of one path among multiple alternatives. $x \mid y \mid z$;

Token dependency: The presence of one token requires another token to be present too. $'(' \, x \, y \, ')'$; and

Nested phrases: self-similar language constructs in which a phrase can have sub- or recursive phrases. $x \mid (y \ (z \mid r))$

These patterns are implemented as well-defined grammar rules in Backus–Naur Form (BNF). BNF formulates rules for specifying alternatives, token references, and rule references. A BNF rule is a nonterminal that is defined as a sequence of alternatives that are separated by the meta-symbol |. Each alternative consists of strings of terminals and nonterminals. The left-hand side, the rule name, is separated from its definition by ::=. Rules are terminated with a ;.

Grammar 6.1 A BNF mini-grammar for a language that accepts simple and block statements in for loops.

⟨*stat-list*⟩ ::= ⟨*statement*⟩ ; ⟨*stat-list*⟩ | ⟨*statement*⟩;

⟨*statement*⟩ ::= ⟨*ident*⟩ = ⟨*expr*⟩
| for ⟨*ident*⟩ = ⟨*expr*⟩ to ⟨*expr*⟩ do ⟨*statement*⟩
| ⟨*stat-list*⟩
| ⟨*empty*⟩
;

Listing 6.1 shows a sample grammar written in BNF: The for, to, and do tokens are terminals, while the $statement$ and $expr$ are nonterminals. The *sequence*, *choice*, *token dependency*, and *nested phrases* patterns are all used in this example.

BNF uses the symbols (⟨, ⟩, |, ::=) for itself, but it does not include quotes around terminal strings. This prevents these characters from being used in languages. Furthermore, it requires a special symbol for an empty string. Options and repetitions cannot be directly expressed in BNF, as they require the use of an intermediate rule or alternative production.

BNF has been extended by the ISO/IEC 14977/ 1996. Extended Backus–Naur Form (EBNF) allows for the grouping of items by wrapping them in a pair of parentheses. Optional and repetitive items can be expressed by enclosing them in [] and {}, respectively. EBNF marks the terminals of the language using quotes, meaning that any character can be defined as a terminal symbol in the language.

There are tools available that are able to generate a lexer from the lexical specification and a parser from the BNF or EBNF representation of a grammar. ANTLR 4 is a Variable length lookahead, Left to right, Leftmost derivation (LL(*)) recursive-descent predictive lexer and parser generator that allows the same syntax to be used to specify both lexical and syntactical rules [PF11, Par13]. In ANTLR, a grammar consists of a set of rules that describe the language

syntax. Productions are called rules. Rules starting with a lowercase latter define the syntactic structure (hence, they are called parser rules), while rules starting with an uppercase letter describe the vocabulary symbols or tokens of the language that constructs the lexical rules.

Like EBNF, the alternatives of a rule in ANTLR are separated by the | operator, and sub-rules can be grouped together by enclosing them in a pair of parentheses. The optionality of an $item$ is indicated by $item^?$, Repetition is expressed by $item^*$ for zero-or-more, and $item^+$ for one-or-more. Operator precedence implicitly follows the order of the alternatives in a rule. For example, if the multiplication alternative is placed before the addition, the parser resolves the precedence ambiguity in favor of multiplication. In ANTLR, operator association[4] is by default from the left to the right, but, when necessary, it can be explicitly declared via $\langle assoc = right \rangle$ or $\langle assoc = left \rangle$ property on the corresponding alternative.

In this dissertation, we use the following methods and materials:

1. An LL(*) grammar. Our proposed language intends to support two dynamic constructs, namely expressions and functions. Expressions can be nested to any level. Functions can have an arbitrary number of arguments, in which each of them can be either an expression or a function. This makes it difficult for the parser to find a distinguishing token by having a specific finite look-ahead k. Therefore, we decided to utilize an LL(*) grammar;

2. A top-down LL(*) parser. The availability of quality tools for lexer and parser generation, as well as support, were among the decision factors;

3. The EBNF meta-language and ANTLR extensions to make it easier for us to formulate closures, repetitions and nesting that we need to author the language grammar.

4. The declarative language paradigm with no procedural feature; in addition, it features referential transparency, independence, and immutability;

5. ANTLR version 4, which supports LL(*), for lexer and parser generation; and

6. Java language version 8 for the development of semantic analysis and all of the other system components.

It is worth mentioning that LL parsers have an intrinsic difficulty with First/First conflicts[5] and left recursion[6]. ANTLR assists with the detection and elimination of direct left recursion only. Indirect left recursion that requires multiple rule traversal and First/First conflict cases are taken care of during the language design process and are solved individually.

[4]Operator associativity: in general, operators of the same priority class are associated from the left to the right, e.g., in $3 * 4/2$, the multiplication is associated before the division. However, in some cases, such as exponentiation, the associativity should go from the right to the left, e.g., 2^{3^4} is calculated as $2^{(3^4)}$ not $(2^3)^4$.

[5]The First/First conflict is a type of conflict in which the FIRST sets of two different grammar rules for the same non-terminal intersect. `https://en.wikipedia.org/wiki/LL_parser`

[6]Left recursion is a special case of recursion in which a string is recognized as part of a language by the fact that it decomposes into a string from that same language (on the left) and a suffix (on the right). `https://en.wikipedia.org/wiki/Left_recursion`

6.4 Related Query Languages

In this section, we briefly introduce the grammars of a set of well-known query languages. We have chosen a diverse set of languages in order to obtain insights into other related query languages and to identify the most useful features. These languages are chosen based on (1) the query operators they support and (2) the data models that they operate on.

6.4.1. SQL

SQL, Structured Query Language, is a relational data management programming language that is primarily based on relational algebra and tuple relational calculus [Lib03]. It is known as a declarative language, but it also includes a number of procedural elements. It was first standardized in 1986 by ANSI, as SQL-86. Different vendors have implemented the standards in various ways, while there are also a number of vendor-specific extensions to the language.

In a relational system, data is represented as a collection of relations, with each relation being depicted as a table. Columns are attributes of the entity modeled by the table, while rows represent individual entities. Certain columns may be designated as the primary key of a table, which uniquely identifies each and every entity. Additionally, tables may benefit from the referential integrity constraints represented by foreign keys.

SQL consists of DDL and DML, which are sub-divided into elements such as clauses, expressions, predicates, queries, and statements [Elm00]. While DDL deals with schema creation and alteration, DML operates on the data for inserting, updating, deleting, and querying [RG00].

An operation that references zero or more tables and returns a table is called a query [ISO08]. In SQL, a query is performed by the declarative SELECT statement [RG00], which by definition has no persistent effect on the database. The SELECT statement specifies the result set, but not how to calculate it. The designated SQL-implementation translates the query into a query plan, based on an optimization algorithm. The optimization may use statistics and heuristics to generate a more efficient plan that involves using indexes, caching, or query rewriting.

The query *statement* depicted in Grammar 6.2 starts with an optional set quantifier that accepts either DISTINCT or ALL options; it determines whether duplicate rows should be eliminated from the result set. The statement then follows with *selectList*, which determines the list of columns to appear in the result set. It forms the query's projection. The projection can be generated in different ways: an asterisk specifying all columns, a qualified asterisk specifying all the columns of a relation, derived columns that are computed during query evaluation, or explicit column names.

The *table expression*, which forms the source of the query, indicates the table reference(s) from which data is to be retrieved. A table reference can be a reference to an ordinary, a derived, or a joined table. A derived table can be a nested query, a view, or the result of a function call. A joined table is the result of a join operation that is performed on a set of participating tables. The

Grammar 6.2 A simplified version of the main components of SQL 2003's query statement [Lef12].

1: $statement$ ::= SELECT $setQuantifier^{?}\ selectList\ tableExpression$
2: $setQuantifier$ ::= DISTINCT | ALL
3: $selectList$ ::= $asterisk$ | ($column$ | $derivedColumn$ | $qualifiedAsterisk)^{+}$
4: $tableExpression$ ::= $from\ where^{?}\ groupBy^{?}\ having^{?}\ window^{?}$
5: $from$ ::= ($table$ | $derivedTable$ | $query$ | $joinedTable)^{+}$
6: $where$ ::= WHERE $searchCondition$
7: $groupBy$ ::= GROUP BY $setQuantifier^{?}\ groupingElement^{+}$
8: $having$ ::= HAVING $searchCondition$

where clause provides a search condition for the query to use to eliminate of all the non-matching records from the result set. The *search condition*, which is not provided in the grammar, is any valid expression that is evaluated to a Boolean. The *group by* clause groups the records that have common values on the provided grouping elements, e.g., columns. The *having* clause applies filtering conditions on the groups, analogous to what the *where* clause does on rows.

At the time of execution, a query optimizer generates a query plan. The clauses of the query statement are then processed in the order defined in the query plan: For example, in the T-SQL used in MS SQL Server 2012, the logical processing order begins with the FROM clause. The WHERE, GROUP BY, HAVING, and SELECT are processed afterwards [Mic12]. At any step, the objects used in the previous steps, i.e., results and names, are visible and available, but not vice versa. PostgreSQL for instance, does not allow projection aliases to be used in the where clause, because the where clause is executed before the projection. As long as the provided effects are identical on the database state, conforming implementations are not required to process the clauses in the same order.

6.4.2. SPARQL

SPARQL [W3C13] is a set of specifications that provides languages and protocols to query and manipulate RDF [CK04] graph content on the Web or in an RDF store. SPARQL contains capabilities for querying required and optional graph patterns, along with their conjunctions and disjunctions [HS13]. It also supports aggregation, sub-queries, negation, and creating values by expressions. Complex queries may include union, optional query parts, and filters. SPARQL supports SELECT queries, which return variable bindings, ASK queries to impose Boolean queries, and CONSTRUCT queries by which new RDF graphs can be constructed from a query result [W3C13]. The results of SPARQL queries can take the form of result sets or RDF graphs [HS13]; however, in order to exchange these results in machine-readable forms, SPARQL supports XML, JSON, CSV, and TSV [W3C13].

The select query in SPARQL comprises a *select*, any number of *datasets*, a *where*, and finally a *solution modifier* clause, as shown in Grammar 6.3. The query is executed on a set of RDF

Grammar 6.3 SPARQL query syntax [HS13].

1: $selectQuery$ ::= $select\ dataset^*\ where\ solutionModifier$
2: $select$ ::= SELECT(DISTINCT | REDUCED)$^?$ (ASTERISK | $selectVariables^+$)
3: $dataset$ ::= FROM NAMED$^?$ $iriRef$
4: $where$ ::= WHERE$^?$ $groupGraphPattern$
5: $solutionModifier$ ::= $group^?\ having^?\ orderBy^?\ limitOffset^?$
6: $selectVariables$ ::= $variable$ | '(' $expression$ AS $variable$ ')'
7: $groupGraphPattern$ ::= '{' $subSelect$ | $triplesBlock^?$ ($graphPatternNotTriples$ '.'$^?$ $triplesBlock^?$)* '}'
8: $group$ ::= GROUP BY ($builtInCall$ | $functionCall$ | '(' $expression$(AS $variable$)$^?$ ')' | $variable$)$^+$
9: $having$ ::= HAVING($expression$ | $builtInCall$ | $functionCall$)$^+$
10: $orderBy$ ::= ORDER BY ((ASC | DESC)$^?$ ($expression$ | $builtInCall$ | $functionCall$ | $variable$))$^+$
11: $limitOffset$ ::= $limit\ offset^?$ | $offset\ limit^?$
12: $limit$ ::= LIMIT **INTEGER**
13: $offset$ ::= OFFSET **INTEGER**

datasets [HS13] and returns a solution set that matches the imposed where clause conditions. An RDF dataset comprises one default non-named graph and zero or more named graphs, so that each named graph is identified by an IRI[7].

The *select* clause, depicted in `line 2`, identifies the variables that are to appear in the query results. Specific variables and their bindings are returned when a list of variable names is given. The list can be provided by an asterisk declaration for all in scope variables, by the explicit names of existing variables, or by introducing new variables into the solution set. The DISTINCT modifier eliminates duplicate solutions from the solution set, and the REDUCED modifier simply permits them to be eliminated if their cardinality is greater than the cardinality of the solution set with no DISTINCT or REDUCED modifier.

A SPARQL query may have zero or more *dataset* clauses, with each adhering to the syntax depicted in `line 3`. The query may specify the dataset to be used for matching by using the *FROM* or the *FROM NAMED* clauses to describe the RDF dataset. The effective dataset resulted from a number of *FROM* or *FROM NAMED* clauses will be either a default graph that consists of a merged RDF of the graphs referred to in the *FROM* clauses or a set of (IRI/graph) pairs, one from each *FROM NAMED* clause.

The *where* clause, as defined in `line 4`, provides a triple/graph pattern to match against the data. It can be composed of conjunctions, filters, operations, and functions. In addition, optional matching is available, which allows non-existing items to be skipped.

The *solution modifier* clause, as shown in `line 5`, is a sequence of optional *grouping*, *having*, *ordering*, and *limit/offset* clauses. If the GROUP BY keyword is used, or if there is implicit grouping due to the use of aggregates in the projection, grouping is performed to divide the solution set into groups of one or more solutions, with the same overall cardinality. The group function can consist of any combination of built-in and user-defined functions, expressions, or variables. The HAVING operator filters the grouped solution sets in the same manner in which the FILTER operates over ordinary ones.

The *order by* clause applies a sequence of order comparators to establish the order and the ordering direction of the solution sequence. The *offset* construct causes the solutions generated to start after the specified number of solutions, and the *limit* clause places an upper bound on the number of solutions returned.

6.4.3. XQuery

In contrast to relational data, hierarchical data is usually nested, without any pre-assumptions concerning depth or breadth. Hierarchical data is usually serialized in XML and JSON. Both allow for defining various elements' schemas, attributes, nesting, and sequences. Schemas are maintained alongside data in order to create standalone self-descriptive documents. Therefore, a search may access the schema as well as the data.

[7] IRIs are a generalization of URIs and are fully compatible with URIs and URLs.

In XML, it is possible to query a document using XQuery [RCDS14], which in turn is constructed upon XPath [BBC+10], for expressing and navigating paths. A search over XML data can reach any level of the document, and the result can contain objects of different types. XQuery is a functional language; therefore, a value can be calculated by passing the result of one expression or function into another expression or function. The language consists of expressions.

XQuery operates on XDM, which is the logical structure of the queried XML document. XDM is generated in a pre-processing step before XQuery is engaged. The pre-processing ends up building an XDM instance, which is assumed to be an unconstrained sequence of items, in which an item is either a node or a (atomic) value. XQuery processing consists of two-phase: a static analysis and a dynamic evaluation. During the static analysis phase, the query is parsed into an operation tree, its static values are evaluated, and its types are determined and assigned. Thereafter, during the dynamic evaluation, the values of the expressions are computed.

Grammar 6.4 A simplified version of XQuery's grammar.

1: $mainModule$ ::= $prolog\ expr$
2: $libraryModule$ ::= $moduleDecl\ prolog$
3: $moduleDecl$ ::= MODULE NAMESPACE $ncName$ '=' $stringLiteral$
...
4: $expr$ ::= $exprSingle^{+}$
5: $exprSingle$ ::= $flworExpr \mid quantifiedExpr \mid typeswitchExpr \mid ifExpr \mid orExpr$
6: $flworExpr$ ::= $(forClause \mid letClause)^{+}$(WHERE $exprSingle$)$^{?}$ $orderByClause^{?}$ RETURN $exprSingle$
7: $forClause$ ::= FOR $forVar^{+}$
8: $forVar$ ::= '$' $qName\ typeDeclaration^{?}$(AT '$' $qName$)$^{?}$ IN $exprSingle$
9: $letClause$ ::= LET $letVar^{+}$
10: $letVar$ ::= '$' $qName\ typeDeclaration^{?}$ ':=' $exprSingle$
11: $orderByClause$ ::= STABLE$^{?}$ ORDER BY $orderSpec^{+}$
12: $orderSpec$ ::= $exprSingle$(ASCENDING | DESCENDING)$^{?}$ (EMPTY(GREATEST | LEAST))$^{?}$ COLLATION $stringLiteral^{?}$
13: $quantifiedExpr$ ::= (SOME | EVERY)$^{?}$ $quantifiedVar^{+}$ SATISFIES $exprSingle$
14: $orExpr$ ::= $arithmeticExpr \mid logicExpr \mid pathExpr$
15: $pathExpr$ ::= '/' $relativePathExpr^{?}$ | '//' $relativePathExpr$ | $relativePathExpr$

As depicted in Grammar 6.4, XQuery is built on two corner stones, namely expressions and paths. Expressions provide the framework necessary for data type declaration and casting, logical and arithmetic formulation, predicate testing, and recursion. Paths, borrowed from XPath, establish mechanisms for forward and reverse traversal on axes, as well as predicate and name

testing on the traversal. XQuery unites these two pillars via FLOWR. FLOWR builds an SQL-like language over expressions and paths that enables XQuery users to access any element, attribute, value, and position in an XML document. Users can also filter these elements and sort the matched result set items. The resulting items can be shaped arbitrarily.

The for clause provides a source for the query. It extracts a sequence of nodes from a bound path and makes it accessible to the following clauses for further operations. The let clause binds a sequence to a variable, without iterating over it. It is beneficial for the following clauses, as they use the bound variable instead of the associated path. The $where$ clause is a conventional selection mechanism that receives a node, tests a Boolean predicate on it, and then either drops or passes the node. The nodes passed through the previous operations may be sorted using the $order\ by$ clause. The $return$ clause is evaluated once per node. It has the ability to apply projections, transformations, and formatting on a node and return the newly built one. The output node does not need to be similar to the input; it even does not need to be formatted in XML.

6.4.4. Cypher

A graph is a finite set of vertices and edges in which each edge either connects two vertices or a vertex to itself. A labeled property graph is a graph in which the vertices and edges contain properties in terms of key-value pairs, vertices can be labeled, and edges are named and directed [RWE15]. A graph database management system is a DBMS that exposes a graph data model and provides operations such as *create*, *read*, *update*, and *delete* against that data [RWE15].

Neo4J is a transactional schema-less property graph DBMS in which graphs record data in nodes [The16]. The nodes have properties and are connected via relationships, which in turn can also have properties. Nodes and edges may have different properties. Nodes and relationships in Neo4J can be grouped by labels in order to restrict queries to a subset of the graphs, as well as to enable model constraints and indexing rules. Indexes are mappings from properties to nodes and relationships, which make it easier to find nodes and relationships according to properties. Querying a graph usually results in a collection of matched sub-graphs. Path expressions are generalized to pattern expressions. Patterns are able to specify the criteria used to match nodes, edges, properties, variable length traversals, predicates, and sub-patterns.

Cypher is a declarative graph query language for Neo4J that is intended to make graph querying simpler. Cypher's pattern-matching allows it to match sub-graphs, extract information, and/or modify data. It can create, update, and remove nodes, relationships, and properties. Cypher is inspired by a number of different approaches and builds upon established practices for expressive querying. While the majority of the keywords, such as WHERE and ORDER BY are inspired by SQL [The13], pattern-matching borrows the expression approaches used by SPARQL[8]. A concise version of the cypher's query syntax is depicted in Grammar 6.5.

[8]SPARQL is also a graph based query language but it operates on a different and more specific data model.

Grammar 6.5 Cypher query grammar version 2.3.7 [The16], based on the Open Cypher EBNF grammar (`http://www.opencypher.org`).

1: $query$ ::= $clause^+$

2: $clause$ ::= $match$ | $unwind$ | $merge$ | $create$ | set | $delete$ | $remove$ | $with$ | $return$

3: $match$ ::= OPTIONAL$^?$ MATCH $pattern$ $where^?$

4: $pattern$ ::= (($variable$ '=')$^?$ $patternElement$)*

5: $where$ ::= WHERE $expression$

6: $with$ ::= WITH DISTINCT$^?$ $returnBody$ $where^?$

7: $return$ ::= RETURN DISTINCT$^?$ $returnBody$

8: $returnBody$ ::= $returnItems$ $order^?$ $skip^?$ $limit^?$

9: $order$ ::= ORDER BY ($expression$ (ASC | DESC)$^?$)$^+$

10: $skip$ ::= SKIP $expression$

11: $limit$ ::= LIMIT $expression$

12: $patternElement$ ::= ($nodePattern$ $patternElementChain^*$) | ('(' $patternElement$ ')')

13: $nodePattern$ ::= '(' $variable^?$ $nodeLabels^?$ $properties^?$ ')'

14: $patternElementChain$ ::= $relationshipPattern$ $nodePattern$

15: $relationshipPattern$ ::= ($leftArrowHead$ $dash$ $relationshipDetail^?$ $dash$ $rightArrowHead$) | ($leftArrowHead$ $dash$ $relationshipDetail^?$ $dash$) | ($dash$ $relationshipDetail^?$ $dash$ $rightArrowHead$) | ($dash$ $relationshipDetail^?$ $dash$)

16: $relationshipDetail$ ::= '[' $variable^?$ $relationshipTypes^?$ $rangeLiteral^?$ $properties^?$ ']'

In Cypher, any query is describing a pattern in a graph. Patterns are expressions that return a collection of paths. So they are able to be evaluated as predicates. Patterns start from implicit or explicit anchor points, which can be nodes or relationships in the graph. Patterns can be sub-graphs traversing over nodes and relationships while conforming to restrictions such as path length and relationship type.

The *match* clause allows specifying a search pattern that Cypher will use to match in the graph. The patterns can be introduced to match all nodes, nodes with a label, nodes having bidirectional or directed relationships, specific relationship types, calling functions that return patterns, and variable length relationship paths.

The *with* clause divides a query into multiple, distinct parts, chaining subsequent query parts and forwards the results from one to the next [RWE13]. For example, in order to obtain a result set of aggregated values that are filter by a predicate, one can write a two-part query piped with a *with* clause: The first part calculates the aggregations, and the second eliminates non-matching aggregate values obtained from the first part. The *with* clause specifies that the first part has to be finished before Cypher can start on the second part.

Beyond from the pattern-matching functionality provided by the *match* clause, it is also possible to filter the result set using the *where* clause. The *where* clause is able to apply Boolean operators and regular expressions on node labels and their properties, as well as on relationship types and their properties. Multiple uses of single-path patterns is also allowed, in combination with other filtering conditions, to eliminate any matched sub-graph from the result set.

Each Cypher query ends in a *return* clause that signals the end of the query and introduces the data that is returned as the query result. The query result is specified in term of *returnItems*, which is a list of expressions, with each evaluated to a solution variable. It is possible to apply *skip*, *limit*, and *order by* clauses within the *return* clause. In read-only queries, Cypher does not actually perform the pattern matching until the result is asked for. As with any declarative language, Cypher can change the order of operators at execution time. Thus, it is possible for query optimizers to decide on the best execution plan in order to reduce the portion of the graph that must be visited to compile the solution.

6.4.5. Array-based Query Languages

MonetDB is an open-source column-store database management system [IGN+12]. It targets analytics over large amounts of data. MonetDB interfaces with its users as a relational DBMS via its support for the SQL:2003 standard. Although it is designed for read-dominant settings such as analytics and scientific workload, it can also be positioned in application domains that feature considerable number of write operations and transactional scenarios. In addition to relational data, MonetDB has built-in support for array data, XML, and RDF. It provides SQL for querying relational data, XQuery for XML, and SPARQL for RDF querying. MonetDB utilizes SciQL to allow users to perform all of the SQL DML operations on array data. MonetDB's querying design paradigm is to keep its interfacing languages as close to SQL as possible. SciQL [KZIN11] satisfies this MonetDB requirement, as its design enhances the SQL:2003

framework to incorporate arrays as first class citizens. The language integrates array-related concepts, such as sets, sequences, and dimensions, into SQL. SciQL also provides a mechanism for accessing cell values in multi-dimensional arrays.

In SciQL, an array is identified by its dimensions [ZKIN11]. These dimensions can be of a fixed size or unbounded. Every combination of index values refers to a cell, which can have a scalar or compound value. Arrays can be used wherever tables are used in SQL, and the SQL iterator concept is adapted to access cells. There is a deep analogy between the relational table and the SciQL array; in fact, SciQL is able to switch between the two and consider them as perspectives on the underlying data.

SciQL's query model is similar to that of SQL [ZKM13][9]: Elements are selected based on predicates, joins, and groupings. The projection clause of a SELECT statement produces an array, which may inherit the dimensions of the query's source. SciQL supports positional access to arrays while preserving dimension orders; thus, range-based access is also possible. This feature allows array slicing based on range patterns. The slicing technique is not only used in queries but also in updating the arrays. Additionally, views are supported, allowing for transposing, shifting, and creating temporary sub-arrays. SciQL grouping has extended SQL, allowing it to accept overlapping groups. This is beneficial when applying aggregate functions on neighborhoods, e.g., in image processing. SQL's windowing feature is supported on arrays and is used to identify groups based on their dimension relationships. In addition to arrays, SciQL has special language constructs used to declare matrices, sparse matrices, and time series.

SciDB is a multi-dimensional array database management system [SBPR11] that utilizes AQL. AQL is an SQL-like declarative language [LMW96, RC13] for working with arrays[9]. AQL is based on nested relational calculus with arrays (NRCA) [LMW96] in that operators, such as join, take one or more arrays as input and return an array as output. Queries written in AQL are compiled into AFL, their functional equivalent, and are then passed through the rest of the processing pipeline, which includes optimization and execution. Beyond the similarity of AQL to SQL, the former processes queries, e.g., joining dimensions, in a remarkably different manner [SBPR11]. AQL includes a counterpart to SQL's DDL that assists in defining and manipulating the structures of arrays, dimensions and attributes.

The studied languages support different feature sets and have various names for similar features. We generalize the most common query features in Table 6.1 and determine how the above-mentioned languages address these features.

All of the languages *query operators*, e.g., selecting and ordering, accept one or more objects from their corresponding data model, operate on them, and return an object. For example, SQL's join operator takes two (or more) tables as input and returns a single table as its output; AQL's join function performs the same operator on arrays and returns a single array. *Functions*, similarly to their mathematical concept, are transformations that accept one (or more) data values, perform the transformation, and return a data value (usually scalar). In contrast, aggregate functions, or *aggregates* for short, accept a collection of values, apply the cumulative operation on

[9] We were unable to locate a formal and citable grammar of either SciDB or AQL.

	SQL	SPARQL	Cypher	SciQL
Access Expression	✓	✓	✓	✓
Source Selection	✓	✓	✓	✓
Projection	✓	✓	✓	✓
Selection (Filtering)	✓	✓	✓	✓
Pattern Matching	✗	✓	✓	✗
Query Partitioning	✗	✗	✓	✗
Data Transformation	✓	✓	✓	✓
Aggregation	✓	✓	✓	✓
Ordering	✓	✓	✓	✗
Grouping	✓	✓	✗	✓
Group Filtering	✓	✓	✓	✓
Set Quantifying	✓	✓	✗	✗
Pagination (Limit/Offset)	✗	✓	✓	✓

Table 6.1.: Query features supported by various query languages.

all of the values of the collection, and finally return a single value. Each of the query languages supports a set of data types that may affect storage, operations, indexing, and conversion.

Access expressions provide a means by which languages can access data attributes. They have different levels of expressiveness. For example, SQL access expressions traverse over schema, table, and Column, while XPath can express variable depth trees, sequences, and cardinality [BBC+10]. Cypher is able to express sub-graphs, node and edge properties, and loops. While pattern matching is supported in SPARQL and Cypher, query partitioning is realized only in Cypher. Standard SQL does not have *Limit/Offset* operators; however, they are integrated into various vendor-specific RDBMSs. Some implementations require the query result to be ordered before the limit/offset is applied. All of these languages, with the exception of Cypher, accept multiple and/or combined sources. Cypher operates on one graph per query and does not accept joins and sub-queries as sources.

6.4.6. Data Model

`QUIS` distinguishes between three data models: input, internal, and output. The input data model is the one that `QUIS` accepts as input and is able to operate on. The studied query languages operate on different data models, e.g., SQL on relational, XQuery on hierarchical data, and Cypher on graph data. In order to harmonize the different terminologies used by the above-mentioned models, we provide the following definitions:

Definition 6.1: Data Source *A data source is a consistent pair of a data organization (Definition 1.2) and a data management system. The data management system organizes and stores the data and provides a level of management over the data.*

A database in a RDBMS or a graph in a graph database are both considered to be conventional examples of a data source. We additionally consider the file system and the supported data files as data sources. For example, an OS-managed directory containing a set of CSV files is considered to be a data source, albeit one with very limited data management facilities. In addition, well-known file types/formats, such as that of MS Excel, are considered to be data sources.

Definition 6.2: Data Container *A data container is a bound set of data in a data source.*

While data sources such as RDBMSs usually manage objects, such as tables and views, other data sources have similar boundaries around sets of data. We generalize these boundaries as data containers. For example, a single CSV file, an RDBMS table, or an MS Excel sheet are considered to be data containers.

Definition 6.3: Data Entity *An individual entity in a given data container that has zero or more attributes and optionally values for those attributes.*

A data entity can be, for instance, a record in a database table, an element in an XML document, a node in a graph, or a line in a CSV file.

Definition 6.4: Data Item *The value of an individual attribute of a data entity.*

A data item can be scalar or compound. In the simplest cases, a data item is a field value of a record in a database table, a single cell in an MS Excel sheet, a value in an array, or a property value on a node of a graph.

While input data can take various forms, we define a query result as a collection of data entities that share a similar schema. This is `QUIS`'s internal representation of data, and it is based on the tuple relational calculus. All of the query operations that access the input data entities have a task that transforms them into equivalent internal representation, allowing the other operators to continue working on the internal representation of data. This is similar to the *scan* operators in RDBMSs that read the disk-stored rows and materialize them as records. This internal representation isolates the higher level query operators, e.g., selecting, inserting, updating, and deleting from the variety of the input data models. In addition, it provides room for adapters tailor entity scanners for the specific data models that they support.

When `QUIS` is requested to communicate query results, it, transforms the internal data model into the requested presentation model based on the query requirements. For example, the presentation model may be a JSON or XML that is used for tool interoperability, a multi-series chart to be used for visual representation, or an R data frame to be used for further statistical processing. This operation is also decoupled from the actual query planning and execution in order to reduce load on the engine and provide a stream-lined data model for superior query optimization. The load is reduced because the representational transformations occurs only once, and on the final query results; hence, there is no need to for instance, transform the tuples and then filter them. This also improves the query performance, as the query operators are optimized for application to the internally represented tuples and do not need to consider different presentation varieties.

6.5 QUIS Language Features

The main focus of the QUIS system is in-situ (Requirement 1: In-Situ Data Querying) querying of different data sources (Requirement 3: Querying Heterogeneous Data Sources), provided that those data sources expose different and inconsistent capabilities. Similarly to SQL, QUIS should also feature DDL and DML, but it is designed to function as an in-situ language with an emphasis on data retrieval. The only area of interest in DDL for the purposes of QUIS is the virtual schema definition (see Feature 3). Therefore, we decided to merge both the DDL and DML into one language. As listed in grammar segment Grammar 6.6 the QUIS's query language consists of two high-level concepts, namely *declarations* and *statements*.

Grammar 6.6 QUIS top level language elements in EBNF[10]

1: $process ::= declaration^* \ statement^+$

In QUIS, the term *process script*, or *process* for short, refers to a sequence of *declarations* and *statements* that are written in a specific order to serve as the data-processing requirements of a designated procedure. While *statements* are the units of execution, *declarations* are non-executable contracts used by statements. Statements may include all kinds of data operations, e.g., querying, manipulating, and deletion, as well as the processing of data by means of applying functions. They consist of *clauses*, *expressions*, and *predicates*. Clauses are used to decompose larger language elements into smaller ones, whereas expressions are objects evaluating to a scalar or a collection of values, and predicates are used in places where a condition is needed. Both expressions and predicates can use functions and aggregates.

6.5.1. Declarations

A declaration can be a definition of the structure of a data object, the information required to connect to a data source, or the constraints that govern the visibility of data containers or the versions of data that are accessible. Declarations are not executable; rather, they act as a contract between other statements or between the language's runtime system and its client. We identify and integrate the following three declaration types into the QUIS language:

6.5.1.1. Connections

In order to obtain data, one must first either connect to a data source or access the data itself. This step may require various types of information: For example, while accessing a local CSV file requires a full file path, accessing an Excel sheet also requires the sheet number/name. Connecting to an RDBMS or a web service not only requires the server URL and port number but

[10]The repetition on *declaration* is for better readability. The complete and accurate grammar is provided in Appendix A.

also credentials. In almost all DBMSs, this concept is not considered to be a part of the query language. The queries of these languages presume that a connection is established and the issued queries are submitted to the system's query execution engine via that connection. We decided to include connection information as part of the language. The main reason behind this choice was that QUIS is a heterogeneous data-querying system in which queries retrieve/manipulate data from various data sources. In these scenarios, QUIS query execution engine should know where to submit individual (sub-)queries. In addition, this feature makes queries more self-descriptive and reproducible. It also makes it possible to run queries on different data sources by simply swapping their connections.

Feature 1 (Connection Information Integrated to Language)

A *connection* models the information required to obtain access to data sources. It may contain connection strings, credentials, network-related information, and data source-specific parameters.

It is worth mentioning that additional or data source-specific configuration parameters may be required to enable the system to obtain data from underlying sources: The values in a CSV files, for example, can be comma- or tab-separated, or a file may have an internal or external header row.

Defining connections independently with their own parameters' specifications not only makes it possible to have a uniform interface for data access at the language level but also provides a base for future extensions. The connection information for new data sources can be easily accommodated as a set of parameter/value pairs and recognized by the system and/or the responsible adapters.

6.5.1.2. Bindings

When a connection to a data source is established, queries may request different versions of data (Requirement 15: Version Aware Querying). The queries may require data from previous versions, join different version of a dataset, or reproduce a snapshot of a dataset as it was used in a research publication [RAvUP16]. QUIS has a versioning scheme at the language level that permits ordinal- (by version number), temporal- (by version date), and label- (by version name) based version identification. Users can specify the version of data to be queried: For example, tools such as DataHub [B$^+$14, Ope15], and BExIS [bex] support internal versioning and provide access to each version of a specific dataset, as well as information about the available versions. QUIS queries allow users to request specific versions of the datasets stored in such systems. The actual versioning implementation may differ from system to system: For example, file-based

data sources may employ complete or differential file duplication, while relational databases may utilize techniques such as version per table, version per view, or version per query.

`QUIS` supports all of the variants of the versioning schemes mentioned above at the language level. Bindings additionally set a visibility scope in order to limit queries to accessing only the declared set of data containers. During query transformation, the chosen adapters translate the version identifiers to filename patterns so that, during the execution phase, the nominated files can be accessed. It is possible to share one connection between multiple bindings and to apply different versioning schemes or version identifiers. Different versions of a single dataset can be accessed via different schemas, as long as the attributes referred to in the schema are present in the data.

Feature 2 (Version Aware Data Querying)

A *binding* establishes a relationship between a connection and a specific version of the bound data. Bindings set a visibility scope that restricts access to the data containers of the target data source.

A *binding* is modeled as a triple $<$ `Connection`, `Versioning Scheme`, `Version Selector` $>$ that establishes a linkage between a connection and a specific version of data that is accessible via that connection. This link isolates the statements from the connections and allows for transparent changing of connections and/or data sources. Thus, it would be easy to redirect a query statement to another version or even to another data source by simply altering its binding information.

6.5.1.3. Perspectives

The schema of data should be known in advance in order to allow query operators to effectively operate on that data. The schema may be obtained from a catalog, e.g., as in RDBMSs, from the data container itself, e.g., in XML, JSON, Avro [SSR+14], or from an external source, e.g., MySQL's external files [Cor16]. These schemas formulate the actual data as it exists in the underlying containers. In many cases, using the original data schema is not satisfactory: For instance, it may not satisfy the requirements of a specific analysis task. Furthermore, computing derived values or reshaping the data, e.g., aggregating, combining, or splitting columns, are common practices that imply the use of a schema other than the original.

As described in Requirement 8 (Virtual Schema Definition) scientific data analysis requires more than physical schemas alone, as i) changing the schema causes data transformation and duplication, ii) the individuals who work with data have different requirements, iii) schemas can

change/evolve over time [Rod95], and iv) various analysis tasks expect data in a format upon which they can operate.

QUIS utilizes *perspectives* to allow users to formally specify the schema of the query results. A perspective consists of the *attributes* that are the individual dimensions of the desired query results. Each attribute defines how it is constructed from the underlying physical data fields; they also determine the reverse mapping for data persistence scenarios. In addition, attributes capture data types and constraints either implicitly or explicitly (see Feature 4 for details). Data types belong to the set of virtual data types defined by the language and are independent of the actual data sources' type systems. The values of attributes can be restricted or formatted by applying constraints on them: For example, a date/time attribute can have short, long, or standard representations. All of these variations should be recognized and ingested. Attributes can additionally be semantically annotated with, e.g., a unit of measurement for automatic transformation.

Perspectives play an important role in query virtualization, as they are defined independently from the subsequent queries and isolate them from the mechanics of data formatting, transformation, and type conversion. Perspectives differ from RDBMS views, as they formulate only projection and transformation, but not selection. Furthermore, perspectives are not materialized. However, they do support single-parent inheritance and overriding. Attribute overriding is done via the use of a base name in a derived perspective; hence, the deepest attribute overrides any attribute defined in the ancestor perspectives. Perspectives are sharable among queries, processes, and people as well.

Feature 3 (Virtual Schema Definition)

A *perspective* is the schema of a query's result set from a user's point of view. It consists of *attributes*, which are individual dimensions of data.

Perspectives build the end-users' view of the data, so they can be defined as the schema of the result sets. Using perspectives, users are able to do the following:

1. Define a schema for their data container of interest, even when the data container has an intrinsic physical or logical schema. Schema definition takes place at the same time as query authoring and undergoes the same life cycle, without affecting the original data or its schema;

2. Define multiple schemas to similar data. This is useful in the following situations:

   a) *Presentation variety:* Different query statements may be required to access the same data but return the results differently.

   b) *Versioning variety:* Queries access different versions of data, which may have different schemas.

c) *Process variety:* Different data analysis tasks require data to be formatted according to their requirements.

3. Define each and every attribute of a schema and allow attributes to be mapped to the actual data items of the designated data container using logical and arithmetic expressions and non-/aggregate functions;

4. Determine the data types of and enforce constraints applicable to attributes using an abstract type system that is independent from, but inter-convertible to, the underlying supported data sources; and

5. Inherit from previously defined or well-known perspectives, as well as overriding inherited schema attributes.

The attributes of perspectives can be used for any kind of data transformations, e.g., changing the units of measurement and the precision of data, correcting the measurement of sensor errors, replacing missing values, or applying domain-specific transformations.

Adding the three declaration types, i.e., connection, binding, and perspective to Algorithm 6.6 results in Algorithm 6.7. The declaration clause allows for the definition of an arbitrary number of *perspectives*, *connections*, and *bindings*. The grammar also defines the order in which these elements can appear in a `QUIS` process.

Grammar 6.7 `QUIS` declarations' grammar. For the complete grammar, see Appendix A.

1: $process \ ::= \ declaration^{*} \ statement^{+}$
2: $declaration \ ::= \ perspective^{*} \ connection^{*} \ binding^{*}$
3: $perspective \ ::= \ identifier \ (\text{EXTENDS } \textbf{ID})^{?} \ attribute^{+}$
4: $attribute \ ::= \ samrtId \ (\text{MAPTO} = expression)^{?} (\text{REVERSEMAP} = expression)^{?}$
5: $connection \ ::= \ identifier \ adapter \ dataSource (\text{PARAMETERS} = parameter^{+})^{?}$
6: $binding \ ::= \ identifier \ \text{CONNECTION} = \textbf{ID} (\text{SCOPE} = bindingScope^{+})^{?}$
$(\text{VERSION} = versionSelector)^{?}$
7: $smartId \ ::= \ \textbf{ID} \ (\!(: dataType)(:: semanticKey)^{?})^{?}$

As shown in Grammar 6.7, each perspective has a name that is generated by the $identifier$ rule, can be extended to another perspective, and contains at least one attribute. The name of each attribute, its optional data type, semantic annotation, and constraints are governed by the $smartId$ rule. Optional forward and reverse mappings specify how an attribute should be materialized from/to actual data fields. Missing data types, as well as attribute mappings, are addressed during the query transformation and execution phases (see Section 7.2.4).

Each connection also has a name and a set of directives that assist the query engine in selecting an appropriate adapter. The $dataSource$ is the construct used to describe the information required to access the data source. Additional data source-specific information is encapsulated in name/value parameters.

Listing 6.1 displays an exemplary usage of a QUIS script that utilizes the above-mentioned language elements. The process begins with a perspective definition (line 1) that has two attributes mapped to physical fields. Mappings are formulated using transformation expressions. The connection information required to access the data source is defined in $dbCnn$ (line 7). In this example, the data source is a local relational database named $soilDB$. The binding $b1$ utilizes the $dbCnn$ connection to establish a visibility scope on the $campaigns$ and $measurements$ containers (line 8). It additionally implies that the latest version of the in-scope data should be used.

Listing 6.1 A sample QUIS process script that defines and uses declarations.

```
1 PERSPECTIVE soil
2 {
3   ATTRIBUTE Temp_Fahrenheit MapTo=1.8*Temp_Celsius+32,
4   // SN: amount of Nitrogen per a volume unit of soil
5   ATTRIBUTE SN_mg MapTo = SN_g * 1000,
6 }
7 CONNECTION dbCnn ADAPTER = DBMS SOURCE_URI = "server:
    localhost, db=soilDB, user:u1, Password:pass1"
8 BIND b1 CONNECTION =dbCnn SCOPE=campaigns, measurements
    VERSION = Latest
9 SELECT PERSPECTIVE soil FROM b1.1 INTO resultSet
```

The query simply retrieves data from the $b1.0$ data container that resolves to the $campaigns$ table in the $soilDB$ database. It then transforms the physical result by applying the $soil$ perspective's attribute mappings and returns the result in $resultSet$. The details of the querying features of QUIS are provided in Section 6.5.2.

Changing the data source is as easy as introducing new connections and/or bindings and referencing them in the queries. Listing 6.2 replaces the DBMS connection with a connection to an Excel spreadsheet, without the need to change the query. After this change, $b1.0$ and $b1.1$ point to the $sheet1$ and $sheet2$ sheets in the $soilData1.xls$ Excel file. A similar change is applicable to the bound perspective.

Listing 6.2 Replacement and redirecting of queries to different data sources.

```
1 CONNECTION excelCnn ADAPTER=SP SOURCE_URI= "d:\data\
    soilData1.xls"
2 BIND b1 CONNECTION = excelCnn SCOPE = sheet1, sheet2 VERSION
     = Latest
3 SELECT PERSPECTIVE soil FROM b1.1 INTO resultSet
```

6.5.1.4. Virtual Type System

The variety of type systems used by different data sources affects both query transformation and execution. In order to conceal this variety and its potential inconsistencies from the user, we have

incorporated a virtual type system into the query language. Such a virtual type system provides enough metadata to allow transforming data items to and from underlying type systems. In addition, it facilities inferring types when the underlying system does not expose such metadata e.g., in a CSV file with a header line that contains only column names. The type system provides an appropriate mechanism for defining perspectives' attributes in an abstract form by isolating them from the implementation and data conversion details. The choice of the programming language to be used to implement the execution engine may also affect the type system and data conversion process.

Feature 4 (Virtual Type System)

The virtual type system provides an appropriate mechanism for defining perspectives' attributes in an abstract form by isolating them from the details of implementation and data conversion.

`QUIS`'s virtual type system supports data types such as integer and floating point numbers, sate, Boolean, and string. These data types are converted or cast to their native counterparts during query transformation. When the data type of an attribute is not explicitly declared, `QUIS`'s type inference utility chooses one based on heuristics. It utilizes various techniques, e.g., the return type of functions, data type of parameters, and operators applied to the data, to make a workable guess. If necessary, it touches the data source/container to ask the fields' data types and infer the attributes' types based on those types.

6.5.1.5. Path Expression

A unified path expression is a sequential representation of a pattern, comprised of data entities, relationships, attributes, expressions, and iterations, that specifies matching criteria to address a sub-model of a data model. According to the requirement that `QUIS` must be able to operate on various data models (see Section 6.4.6), we need to support a uniform path expression that enables users to express data item accesses in a manner independent of the target data organization. For example, if the underlying data model is relational, the path will usually take the form of $schema.table.field$. This pattern can be generalized to other tabular data models, e.g., CSV. However, array databases add the dimension concept to the path expressions. Hierarchical data such as XML and JSON require variable depth paths, axes, and sequences. Additionally, graphs may need to match sub-graphs, loop over paths, and access node and edge properties.

Feature 5 (Uniform Access to Data Items)

The unified access pattern provides a sequential representation for expressing paths and patterns to access the data items of the supported data models.

Grammar 6.8 `QUIS`'s Path Expression Grammar.

1: $pathExpression \ ::= \ (path \ attribute^{?}) \mid (path^{?} \ attribute)$
$path \ ::= \ (path \ relation \ path) \mid (path \ relation) \mid (relation \ path)$
2: $\mid (\text{`('}(label \ \text{`:'})^{?} \ cardinality^{?} \ path \ \text{`)'})$
$\mid (step) \mid (relation)$
3: $step \ ::= \ (unnamedEntity \mid (namedEntity \ sequenceSelector^{?}))\ predicate^{*}$
4: $attribute \ ::= \ \text{`@'}(namedAttribute \mid \text{`*'} \mid predicate)$
$relation \ ::= \ forward\_rel \mid backward\_rel \mid non\_directional\_rel$
5: $\mid bi\_directional\_rel$
6: $forward\_rel \ ::= \ \text{`->'} \mid (\text{`-'} \ label \ \text{`:->'}) \mid (\text{`-'}(label \ \text{`:'})^{?} \ taggedScope \ \text{`->'})$
7: $backward\_rel \ ::= \ \text{`<-'}(label \ \text{`:-'} \mid (label \ \text{`:'})^{?} \ taggedScope \ \text{`-'})^{?}$
8: $non\_directional\_rel \ ::= \ \text{`-'}(\text{`-'} \mid label \ \text{`:-'} \mid (label \ \text{`:'})^{?} \ taggedScope \ \text{`-'})$
9: $bi\_directional\_rel \ ::= \ \text{`<-'}(\text{`>'} \mid label \ \text{`:->'} \mid (label \ \text{`:'})^{?} \ taggedScope \ \text{`->'})$
10: $taggedScope \ ::= \ (tag \ (\text{`|'} \ tag)^{*})^{?} \ relationScope$
$relationScope \ ::= \ sequenceSelector \ predicate \mid cardinalitySelector \ predicate$
11: $\mid sequenceSelector \mid cardinalitySelector \mid predicate$
12: $sequenceSelector \ ::= \ \text{`('}\ \textbf{NUMBER}\ \text{`)'}$
13: $predicate \ ::= \ \text{`['}\ expression\ \text{`]'}$
$cardinalitySelector \ ::= \ \text{`\{'}(($\textbf{NUMBER}$^{?}$ \text{`..'}\ \textbf{NUMBER}$^{?}$)$
14: $\mid \textbf{NUMBER} \mid \text{`*'} \mid \text{`+'} \mid \text{`?'})\ \text{`\}'}$

`QUIS` is equipped with a canonic path express at the language level. This facility provides enough expressive power to define tabular, hierarchical, and graph-based paths. It can be used for both matching and querying. For instance, one can use this facility to match a sub-graph in a graph database or to retrieve the value of an attribute.

Access to data items of interest in a chosen data container is expressed by the $pathExpression$ rule, as defined in Grammar 6.8. A $pathExpression$ can be a $path$ that is optionally followed by an $attribute$ or an $attribute$ alone. In its simplistic form, a path is chain of $steps$ linked together using $relations$. However, it can start or end with a $relation$, or it can be a $step$ or a $relation$ only. $\mathrm{a} \rightarrow \mathrm{b}$ represents step a, which has a directed relationship to step b. Step a can be a table, an element, or a node in a relational database, an XML document, or a graph database. The relation may also be interpreted differently by different parsers, query engines, or adapters.

At each step, it is possible to refer to an entity type, a specific entity, or a set of entities that satisfy a set of predicates. Predicates can check for the existence of attributes as well as values. They benefit from the full power of expressions (see Grammar 6.10). `QUIS` supports four types of relations: forward, backward, bi-, and non-directional. The forward type can be used to traverse traditional tabular data; it also represents children and descendant axes in XPath, as

well as directed forward relations in graph query languages such as Cypher. The other relation types serve similar purposes.

Furthermore, relations in QUIS accept cardinality, sequencing, and selection constraints. The cardinality constraint sets a minimum and/or maximum length of a path, with support for closures. This is useful in expressing paths of variable length. The sequencing constraint causes a path to point to a specific element by its position and the predicate allows for testing the properties of visiting relationships. As an example, in a property graph database of friends, assume relationships means "friendship" and each relationship has a "start_date". Using the above constructs, QUIS is able to easily express "networks of at most five friends who have been in relationships for more than three years". A verbose table that demonstrates the expressive power of QUIS's path expression is presented in Appendix B. It illustrates QUIS's coverage of XPath and Cypher. Expressing path expressions for tabular data is trivial: In its most verbose form, it takes the form of $\text{server} \rightarrow \text{database} \rightarrow \text{table} \rightarrow \text{field}$. Using the proposed path expression grammar, it is not only easy to express such tabular paths but also to formulate constraints and joins. For example, $\text{students}[@\text{id} == 21]@\text{userName}$ matches the username for a user with $id = 21$, while $\text{students}[@\text{id} == 21]\text{-}[@\text{id} == @\text{studentId}] \rightarrow \text{courses}@\text{name}$ expresses a join on $students$ and $courses$ tables on $students.id$ and $courses.studentId$ fields. The path also applies a predicate constraint on the students table to match $id == 21$ only. Finally, it accesses the $name$ field of the courses.

6.5.2. Data Retrieval (Querying)

QUIS is designed to focus on data retrieval. Hence, it should have a query language expressive enough to address the most useful features of the related languages, as described in Section 6.4. QUIS introduces the statement as a language element that may have a persistent effect on data or that may control an execution procedure. A *query* is a sub-class of statement for data retrieval that does have any persistent (or side) effect on the data.

Feature 6 (Heterogeneous Data Source Querying)

In conjunction with declarations, QUIS queries are able to retrieve, transform, join, and present the data obtained from various and heterogeneous data sources.

Based on the query features summarized in Table 6.1, we design our query to realize the features expressed in Grammar 6.9. As the Grammar illustrates, a statement (`line 2`) can either retrieve, insert, update, or delete data. QUIS grammar for data querying is defined by the $selectStatement$ rule (`line 3`).

The minimum query is constructed by a **`SELECT`** keyword, followed by the source-selection clause. Such a query retrieves data from the specified source without performing any operation

Grammar 6.9 `QUIS` query grammar.

1: $process$ ::= $declaration\ statement^+$

2: $statement$ ::= $selectStatement$ | $insertStatement$ | $updateStatement$ | $deleteStatement$

3: $selectStatement$ ::= SELECT $setQualifierClause^?\ projectionClause^?\ sourceSelectionClause\ filterClause^?\ orderClause^?\ limitClause^?\ groupClause^?\ targetSelectionClause^?$

4: $projectionClause$ ::= USING PERSPECTIVE $identifier$ | USING INLINE $inlineAttribute^+$

5: $inlineAttribute$ ::= $expression$(AS $identifier$)$^?$

6: $sourceSelectionClause$ ::= FROM $containerRef$

7: $containerRef$ ::= $combinedContainer$ | $singleContainer$ | $variable$ | $staticData$

8: $combinedContainer$ ::= $joinedContainer$ | $unionedContainer$

9: $unionedContainer$ ::= $containerRef$ UNION $containerRef$

10: $joinedContainer$ ::= $containerRef\ joinDescription\ containerRef$ ON $joinKeys$

11: $joinDescription$ ::= INNER JOIN | OUTER JOIN | LEFT OUTER JOIN | RIGHT OUTER JOIN

12: $joinKeys$ ::= $identifier\ joinOperator\ identifier$

13: $joinOperator$ ::= EQ | NOTEQ | GT | GTEQ | LT | LTEQ

14: $filterClause$ ::= WHERE LPAR $expression$ RPAR

15: $orderClause$ ::= ORDER BY $sortSpecification^+$

16: $sortSpecification$ ::= $identifier\ sortOrder^?\ nullOrder^?$

17: $sortOrder$ ::= ASC | DESC

18: $nullOrder$ ::= NULL FIRST | NULL LAST

19: $limitClause$ ::= LIMIT(SKIP = **UNIT**)$^?$(TAKE = **UNIT**)$^?$

20: $groupClause$ ::= GROUP BY $identifier^+$(HAVING LPAR $expression$ RPAR)$^?$

21: $targetSelectionClause$ ::= INTO($plot$ | $variable$ | $singleContainer$)

22: $plot$ ::= PLOT$^?$ $identifier$ HAXIS:$^?$ $identifier$ VAXIS:$^?$ $identifier^+$ PLOTTYPE:$^?$($plotTypes$ | **STRING**) HLABEL:$^?$ **STRING** VLABEL:$^?$ **STRING** PLOTLABEL:$^?$ **STRING**

23: $plotTypes$ ::= LINE | BAR | SCATTER | PIE | GEO

on it; the other features are optional. Beyond the minimum query, there is a set of operators that take over various functionalities. We describe the most important characteristics of these operators bellow:

6.5.2.1. Source Selection

QUIS queries obtain data from *data containers* (line 6). Data containers are registered and managed by adapters and provide access to persisted data in a format known to the respective adapters. In addition to persistent data containers, QUIS incorporates a special kind of data container called a *variable*. A variable is an immutable in-memory container that can be loaded statically or by query results. It can be used as a normal data container by subsequent queries; thus, it serves to provide a query chaining/partitioning mechanism. In addition, variables can play the role of sub-queries and reduce the amount of data actually retrieved when shared among queries.

Feature 7 (Query Chaining)

Query chaining is a feature incorporated in QUIS that makes it possible to pass the result set of a previously executed query to one or more other queries.

A QUIS query is able to access and combine (i.e., join or union) multiple data containers through declared bindings. Data containers can belong to single/multiple bindings/connections; in addition, they can be homo/heterogeneous. The combination of variables and other data containers, in any permutation, is also supported. More specifically, data can be obtained from a) a single container determined by an associated binding, b) a variable that holds the result set of a previously executed statement, c) a static data defined inline within the query, and d) a combined set of two containers of any combination, including other combined containers (line 7). In the case of join, the join type, join keys, and the join operator between any keys are also determined (line 10). The logic of union and join operations is that of relational tuple calculus. However, in contrast to SQL, which matches the unioned column names to their order, we match by their case-insensitive names.

6.5.2.2. Projection

The main and recommended vehicle for declaring a projection is to define and apply an explicit perspective, as specified by the $projectionClause$ in line 3. The projection clause itself (line 4) provides two options for declaring a perspective for a query: 1) an explicit declaration by referencing an already defined perspective and 2) defining the perspective inline with the

query. All inheritance and overriding rules apply, and an effective schema is associated with the query. This shapes the schema of the query's result set. The attributes of the effective perspective are visible to the subsequent query features according to the query execution plan. If no explicit perspective is declared, the query execution engine will infer and assign one by analyzing the query's container(s) or the actual data (see Section 7.2.3 (Schema Discovery)).

6.5.2.3. Selection

In `QUIS`, the selection clause (`line 14`) is designed to remove non-matching objects from the result set. It is based on predicates that evaluate to true or false. These predicates can be constructed from logical, arithmetic, or combined expressions. The use of non-aggregate function calls is also allowed. In addition to basic mathematical and logical operations, precedence, associativity, nesting, and call to functions are supported. All of the attributes of the ambient effective perspective are available to the selection clause. Furthermore, the selection's predicate has access to the physical data items of the queried data containers. This cherry-picking technique assists in filtering data objects by the properties of the actual data that are not part of the query's perspective.

6.5.2.4. Ordering

The $orderClause$ (`line 15`) sorts the query's result sets. The clause begins with **`ORDER BY`** followed by at least one sort specification, $sortSpecification$. Each sorting specification consists of a sort key that refers to an attribute in the query's schema, a sorting direction, and the order of NULL values. When more than one sorting specifications are present, the result set is first sorted by the left-most one, then, for data objects that are equal on that key, the next specification is applied. The NULL values can be chosen to appear at the top or bottom.

6.5.2.5. Pagination

Pagination is a technique used to truncate the result set and return only one slice of it. It is included in the grammar as $limitClause$ (`line 19`), but it can accomplish offsetting as well. The clause makes it possible to optionally *skip* over a non-negative number of data objects s and then optionally take a non-negative number of data objects t. Omitting the **`SKIP`** means taking a maximum of t data objects from the beginning and omitting the **`TAKE`** means skipping the first s data objects and returning the remaining. If there is only $k \ni k < t$ data objects available, the **`TAKE`** clause returns those available k objects.

6.5.2.6. Grouping

Similarly to the related languages, the grouping clause (`line 20`) in `QUIS`'s grammar groups the result set based on the provided attributes. Only the attributes of the effective perspective are accessible to the grouping. However, grouping creates another perspective for the result set, which is applied during query execution. Grouping is usually declared explicitly, but it is possible for the effective perspective of the query to contain aggregates. The presence of the aggregates leads the query engine to rewrite the query to an equivalent query with a grouping clause based on the non-aggregate attributes of the perspective. Groups can be further filtered by introducing a **`HAVING`** predicate. This predicate is as expressive as the selection's predicate, but it only has access to the attributes of the generated grouping perspective.

6.5.2.7. Target Selection

The target selection clause $targetSelectionClause$ (`line 21`) specifies how query results should be delivered to the requester. The requester can be either an end-user or a system. A tool such as R system [R C13] may require the result in the form of a data frame[11], while an SWMS may require the result set to be delivered in JSON format. A human user would normally prefer a tabular presentation or a visualized form, such as a chart.

The target clause can route the result set of the query to different destinations. One common option is presenting the result set in a tabular form according to its bound perspective. It is also possible to persist the result set using a specified serialization format. The serialization format is chosen from the list of formats supported by the registered adapters; JSON and XML are supported by default. Persisting the result sets allows queries to read from one or more data containers and write to another; this way, `QUIS` can act as a bridge for data transformation and system integration. Registering new and/or richer adapters provides a greater number of supported serialization formats to the target selection clause.

Feature 8 (Polymorphic Query Result Presentation (PQRP))

Query results can be presented to clients in many ways upon request.

In addition to the above-mentioned targets, `QUIS` presents result sets in visual forms: For example, a user can request that a line chart be drawn from a query result set. This feature provides users with agile feedback and allows them to rephrase their queries in order to achieve the optimal result set.

[11] `http://www.r-tutor.com/r-introduction/data-frame`

Feature 9 (Visual Resultset Presentation)

In QUIS, it is possible to present a query result set in visual form.

As expressed in (line 22), plots can draw single or multi-series line, bar, scatter, pie, and geographical bubble charts. The types, features, and appearance of the charts are controlled by the parameters declared in respective queries. QUIS's default chart realization is performed by its fallback adapter. Therefore, not only are the other adapters not required to realize this feature but it is also guaranteed that the feature will always be available. The PQRP feature relies on the overall system's plug-in architecture; thus, it is a straightforward procedure to replace existing visualizations or add new ones.

6.5.2.8. Data Processing

QUIS handles data transformation and processing by applying expressions to data objects. It relies on its functions and aggregates, as well as arithmetic and logical operators. While functions operate on single data values, aggregates are functions that obtain a collection of data values and return single values. In addition, they are used in conjunction with the grouping operator. Functions have no side effects[12] and are referentially transparent[13]. QUIS includes a set of built-in function packages that can operate on strings, numbers, and date and time, as well as aggregate functions, such as average, sum, and count. More sophisticated aggregate functions are grouped into the statistics package. Furthermore, QUIS also supports user-defined functions. User-defined functions can be developed as *function packages* and can be imported into the system by utilizing its plug-in architecture. It is also possible to override the built-in functions. The transformations can be declared in perspectives, inline with queries' projection phrases, or in the expressions used in the selection and having clauses.

Feature 10 (Data Processing)

QUIS provides an extensible mechanism for data processing with a comprehensive built-in set of frequently used functions and aggregates.

The grammar of the *expression* is listed in Grammar Algorithm 6.10. Beyond the basic arithmetic and logical operations, such as addition, multiplication, and comparison, its design allows for both simple and nested function calls, negation, and user-defined functions.

[12] http://en.wikipedia.org/wiki/Side_effect_(computer_science)
[13] http://en.wikipedia.org/wiki/Referential_transparency_(computer_science)

Grammar 6.10 `QUIS`'s Expression Grammar.

$expression$ ::= NEGATE $expression$ | $expression$ (MULT | DIV | MOD) $expression$
| $expression$ (PLUS | MINUS) $expression$
| $expression$ (AAND | AOR) $expression$
| $expression$ (EQ | NOTEQ | GT | GTEQ | LT | LTEQ | LIKE) $expression$
| $expression$ IS NOT$^{?}$(NULL | NUMBER | DATE | ALPHA | EMPTY)
1: | NOT $expression$
| $expression$ (AND | OR) $expression$
| $function$
| LPAR $expression$ RPAR
| $value$
| $identifier$
2: $function$ ::= $(identifier\ .)^{?}\ identifier$ LPAR $argument^{*}$ RPAR
3: $argument$ ::= $expression$

The recursive nature of an expression is also reflected in its grammar. In the majority of the alternatives the expression is formulated in terms of an operation on one or two other expressions. For example, a *function* can accept any number of *arguments*, which are expressions. This enables functions to accept scalar values, arithmetic or logical expressions, and other function calls as input. The rules are matched from upper alternatives downwards and from the left to the right in each alternative. Therefore, negation is designed to have higher precedence in comparison to multiplication, which is in turn evaluated before division.

Combining all of the requirements, we have designed a grammar that offers a set of uniform and expressive constructs for providing an abstraction layer over the actual data sources' capabilities and the underlying data processing techniques.

7

Query Transformation

Requirement 4 (Unified Syntax) implies that the system must offer its end users a unified query language. In Chapter 6 (Query Declaration), we described the features, design, and syntax of such a language. Requirement 5 (Unified Semantics) implies that all of the elements of input queries must convey a unique meaning, even if the underlying data sources realize them differently. We satisfy this requirement through the use of *query transformation*, which is a set of writing and rewriting techniques that generates from a given input query appropriate computation models tailored to run on designated data sources.

Query transformation should consider the syntactical and semantic differences between various data sources. In join and union queries, the sides should be transformed according to their corresponding data sources. Additionally, as Requirement 1 (In-Situ Data Querying) demands that data be queried in-situ, any query transformation effort should consider working on the original data. Hence, the system must generate the target queries in such a manner that they access the data without duplicating or loading it onto any intermediate medium. Transient transformations intended to comply with input query or representation requirements, as well as internal data integration must be performed in a manner that is transparent to users and client systems.

Although we address Requirement 6 (Unified Execution) in Chapter 8 (Query Execution), it has an implication for query transformation that we consider here. The implication is that the result set of an input query should be independent of both the actual data organization as well as the functions available in the queried data sources.

As explained in Chapter 5 (Overview of the Solution), each input query is first transformed into its internal representative DST. The DSTs are then passed on to chosen adapters to be transformed into their target counterpart computation models. In this chapter, we first elaborate on how queries are represented internally in Section 7.1. Thereafter, we explain the transformation techniques used to build appropriate computational models for the input queries in Section 7.2. These techniques transform a given input query into either a set of queries in the target data sources' languages or, when there is no query language available, into a set of operations on the target data. Joins can combine these techniques and transform the input queries into a mixed computation model. Section 7.3 is dedicated to techniques that detect and complement inconsistencies between the input queries and the capabilities of the designated data sources. The proposed algorithms demonstrate that having a data source that satisfies a minimum set of required capabilities would be sufficient to ensure unified semantics and execution of input queries.

In Section 7.4 we introduce and evaluate a set of optimization rules to demonstrate how, and to what extent, the performance of input queries can be improved.

7.1 Query Plan Representation

Input queries undergo multiple processing phases when they are prepared to be transformed into their target counterparts. At the heart of the preparation process, each submitted query is validated by a parser and converted into a DST. Having prepared and registered the DSTs in the process model, the query engine determines which adapters are responsible for transforming each DST and assigns them to the corresponding query nodes in the process model. The details of the adapter selection mechanism is described in Section 8.1.2; Figure 7.1 depicts the assignment.

There are four queries, $q1 - q4$, which use three adapters, a_1, a_2, and a_m. While $q1$ and $q2$ are transformed by a_1 and a_2, respectively, a_m is assigned to both the q_3 and q_4 queries. The figure also shows a data dependency between queries: q_3 depends upon q_2 and q_4 upon q_1. This dependency indicates that the dependent queries q_3 and q_4 (partially) obtain their data from the q_2 and q_1 queries.

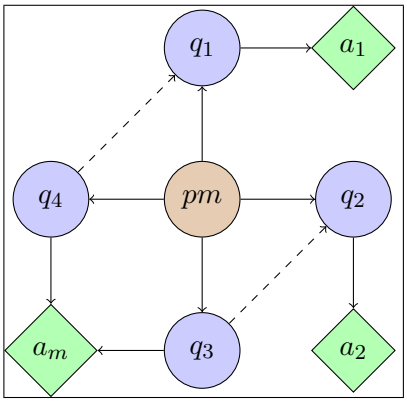

Figure 7.1.: A sample instance diagram of an ASG with adapters assigned to queries. $q_1 - q_4$ are the queries. a_1 and a_2 are the adapters responsible for transforming q_1 and q_2, respectively. a_m is the adapter responsible for q_3 and q_4. The q_4 and q_3 queries are data dependent upon q_1 and q_2, respectively.

Assume we have a simple query, as in Listing 7.1 that retrieves from a database a limited number of student records that match a set of criteria and then sorts them.

Listing 7.1 A QUIS query that retrieves a maximum of 10 student records whose last names contain 'Ch' from a database and orders them by the last name.

```
1  PERSPECTIVE student{
2    ATTRIBUTE FirstName:String  MapTo: student_Name,
3    ATTRIBUTE LastName:String   MapTo: student_lastName
4    ATTRIBUTE Gender:Integer    MapTo: student_gender
5  }
6
7  CONNECTION ds ADAPTER=DBMS SOURCE_URI='' PARAMETERS=server:
       localhost, database:edu, user:user1, password:pass1
8  BIND occ CONNECTION=ds SCOPE=Students VERSION=Latest
9
10 SELECT
11 USING PERSPECTIVE student
12 FROM occ.0
13 INTO result
14 WHERE (str.indexof('Ch', LastName) >= 0)
15 ORDER BY LastName DESC
16 LIMIT TAKE 10
```

Figure 7.2 represents the process model of the query depicted in Listing 7.1. In this figure, the $q1$ node represents the query itself. The children are the top-level query features such as the selection, projection, and ordering clauses. The query node is linked to the process model pm. This linkage is used for building dependency on/from other queries. The nodes representing query features are described at various transformation phases, mostly during the syntax and semantic analyses. The type of information gathered varies from feature to feature; for example, the pagination feature (LIMIT clause) requires the number of records to skip over and/or to take, while the ordering clause needs a list of $\langle sort\ key, sort\ direction, null\ ordering \rangle$ tuples.

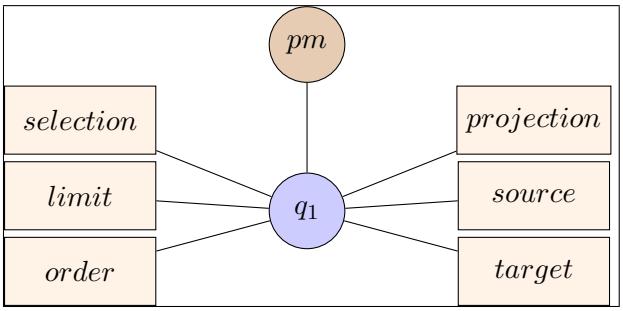

Figure 7.2.: A query $q1$ is represented as a node in the ASG graph, rooted at pm. The query features are modeled as nodes associated to the query node $q1$.

During the semantic analysis, queries are examined for their data sources. Each query retrieves data from one or more data containers and directs its result into a target, which can also be a

data container. The queries use bindings to access data containers; hence, the semantic analyzer checks the existence and the correctness of the bindings and their associated connections. If valid, the bindings are linked to the query nodes in the corresponding DSTs. A pair of queries that are chained to share a binding is depicted in Listing 7.2. Figure 7.3 illustrates two queries, q_1 and q_2, that use the bindings b_1 and b_2 as their data sources; the binding b_2 is shared between the two. Bindings provide access to multiple data containers; therefore, q_2 is able to retrieve data from one container and store it in another.

Listing 7.2 Two chained QUIS queries that share a binding to retrieve and store data.

```
1  PERSPECTIVE student{
2    ATTRIBUTE FirstName:String  MapTo: student_Name,
3    ATTRIBUTE LastName:String   MapTo: student_lastName
4    ATTRIBUTE Gender:Integer    MapTo: student_gender
5  }
6
7  CONNECTION ds ADAPTER=DBMS SOURCE_URI='' PARAMETERS=server:
       localhost, database:edu, user:user1, password:pass1
8  BIND b1 CONNECTION=ds SCOPE=Students VERSION=Latest
9
10 CONNECTION csvCnn ADAPTER=CSV SOURCE_URI='/home/data'
       PARAMETERS=delimiter:tab, fileExtension:csv,
       firstRowIsHeader:true
11 BIND b2 CONNECTION=csvCnn SCOPE=students, searchResult
       VERSION=Latest
12
13 SELECT
14 USING PERSPECTIVE student
15 FROM b1.0
16 INTO b2.0
17 LIMIT SKIP 100 TAKE 1000
18
19 SELECT
20 USING PERSPECTIVE student
21 FROM b2.0
22 INTO b2.1
23 WHERE (str.indexof('Ch', Last) >= 0)
```

7.2 Query Transformation Techniques

The role of query transformation is to build an optimized executable version of a given input query. In RDBMSs, query transformation is mainly employed by an optimizer to rewrite queries

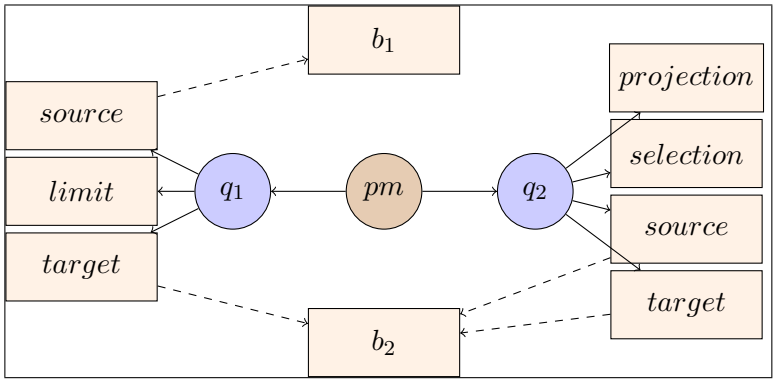

Figure 7.3.: The ASG of the two queries that share a binding (as in Listing 7.2). q_1 retrieves data from b_1 and writes into b_2; q_2 reads the data written by q_1, filters the data, and inserts the result into $b2$. However, q_2 operates on different containers for reading and writing.

in order to reduce their execution time, the number of records touched, or both [Cha98]. In FDBMSs, there is one phase of transformation that must be performed before optimization can be even considered. This first phase of query transformation translates the input query to an equivalent set of queries written in the language of the component databases. In `QUIS`, the heterogeneity of the component data sources is broader than is usually than the case in FDBMSs. `QUIS`'s data sources can vary from bare CSV files to feature-rich RDBMSs or graph databases that are equipped with their own query languages.

The broad variety and openness of data sources leads to a multitude of challenges within the scope of transformation. The first challenge is that the target data sources are likely to use different query languages, which makes query transformation more complex. Transforming joins, nested queries, and nested joins introduce even greater complexity. The second challenge is that not all data sources have query languages; For example, CSV files have no access utility, Excel files are equipped with APIs, and online data sources are accessible through Web Services (WSs). This class of weak data sources [TRV98] requires the input query to be transformed into a set of operations written in a programming language that calls either their APIs or accesses the data directly. The third challenge is that different data sources demonstrate different levels of functionality, meaning that a data source may not be able to satisfy a subset of features requested by a query, while another data source may demonstrate lack of support for another subset.

In the following sections, we elaborate upon these challenges and explain our proposed solutions. We begin by transforming `QUIS` queries into other query languages in Section 7.2.1. We explore our approach to the transformation for weak data sources by suggesting appropriate computation models in Section 7.2.2. In Section 7.2.3 we explain the our approach to schemas and schema discovery. Thereafter, in Section 7.2.4 we describe how data types are handled during query transformation. Our solution for overcoming functional heterogeneities in the capabilities of data sources is described in Section 7.3. Finally, in Section 7.4 we introduce our optimization techniques and discuss their effectiveness.

7.2.1. Query to Query Transformation

Assume $F(L)$ yields the feature set (or operators) of language L and $F(q_l)$ is the set of features of language l required by query q. We define $F(L_s) = \{f_1, f_2, ..., f_n\}$ as the features provided by QUIS's language. Presuming q_s is a query in L_s that requires $F(q_s) = \{f_a, f_b, ..., f_k\}$ features, our goal is to transform query q_s to its equivalent in language L_d. L_d provides $F(L_d) = \{d_1, d_2, ..., d_m\}$ features through a selected adapter $a \in \{a_1, a_2, ..., a_p\}$. We want to have the following:

$$\forall\, q_s \in L_s, \exists\, q_d \in L_d \ni q_d = t_a(q_s) \tag{7.1}$$

Here, t is the transformation function implemented by adapter a. Because each adapter receives only that sub-query (DST) which it can transform (see Section 7.3), t is guaranteed to be able to generate a transformation of q_s; hence, a function t exists. The transformation function does indeed transform two elements of the source query, namely the features and the order. The declarative nature of the language relaxes the transformation techniques to preserve the result set equivalence only. This means that the order of execution remains flexible and can be decided upon at execution phase. According to Equation (7.1), adapter a is guaranteed to have individual transformations for the features requested by q_s; therefore, for each feature f_i in query q_s there is a feature transformer t_{a_i} in adapter a that, when applied, generates the features equivalent $q_{d_{f_i}}$.

$$\forall f_i \in q_s, \exists\, t_{a_i} \in a \ni q_{d_{f_i}} = t_{a_i}(f_i) \tag{7.2}$$

Function t_{a_i} utilizes the available features of the target language $F(L_d) = \{d_1, d_2, ..., d_m\}$ to conduct the transformation. The overall transformation t would be a substitution of the partial transformations:

$$q_d = t_a(q) = substitute(f_i, t_{a_i}(f_i) \mid \forall f_i \in F(q_s)) \tag{7.3}$$

The substitution function does not need to preserve the order of the features of the input query. The feature transformers t_{a_i} are realized by utilizing templates and data transformation functions. The templates are used to convert the syntax of the input feature(s) to the appropriate syntax of the target language. Data transformation functions are used for data type consolidation, expression evaluation, and query projection building. For example, for the selection predicate $math.abs(credit) > 2400\ AND\ math.min(score, udf.score(balance)) > 4$, the selection feature transformer will not only substitute the mathematical functions used but may also rewrite the expression's evaluation tree, e.g., in order to simplify or reorder its evaluation.

The query transformer has access to the source query's DST node, which contains detailed information concerning the query and each of its features. For example, the transformer can trace a container identifier to its name in the related binding and then to its connection. In this manner, it can translate the container's name to, e.g., a relational table name. Or, as another example, the transformer tracks an attribute att_1 referred to by the query's predicate to its definition in

perspective p_1 and obtains its mapping expression exp_1. Attribute att_1 is then substituted with exp_1, possibly after the expression itself has been transformed. The aggregate and non-aggregate functions used in such expressions are also transformed into their native counterparts.

The final transformed query q_d is embedded in a job that acts as an execution context for the query. The job consists of the operations required to establish a connection to the designated data source, to execute the transformed query, to format the result set, and to handle errors. This job is then submitted, as a unit of execution, to the Query Execution Engine (QEE) for further processing, compilation, and execution.

7.2.2. Query to Operation Transformation

Weak data sources are those that generally lack data management and/or access capabilities. Usually, they do no feature a declarative language that can be used to query their managed data. They also suffer from a lack of feature-rich or standardized APIs. In cases such as CSV and JSON, the data is provided as is, and it is the responsibility of users or tools to access and query it. The remote data behind WSs can also fall into this category.

Assume $O(ds) = \{o_1, o_2, ..., o_n\}$ yields the set of operations available in a non-declarative data source ds. Each operation o_i is an atomic unit of work to be performed on the data; it can be a row scan, a value parse, or a string manipulation operation. The ds data source is said to be weak regarding query q_s, if $\exists\, q_s \in L_s, \ni t_a(q_s) = \emptyset$. Additionally, $O(ds) = \emptyset$ is considerable. We want to transform query q_s to its equivalent sequence of operations in O_{ds} through adapter a. We declare the following:

$$\forall q_s \in L_s, \exists\, p \ni p = t_a(q_s) \tag{7.4}$$

Here, t is the transformation function implemented by adapter a. It builds a procedure p that is result-equivalent to query q_s, so that $p = (o_1, o_2, .., o_k \mid o_i \in O(ds))$ is an ordered sequence of operations in the ds data source. Transformer t consists of individual transformations for the features requested by the query. Therefore:

$$\forall f_i \in q_s, \exists\, t_{a_i} \in a \ni p_{f_i} = t_{a_i}(f_i) \tag{7.5}$$

In which t_{a_i} is a function that transforms the input feature f_i to a procedure p_{f_i} using the operations available in O_{ds}. The overall transformation t would be a second-level sequence of the procedures generated for each feature:

$$p = t_a(q) = sequence(t_{a_i}(f_i) \mid \forall f_i \in F(q_s)) \tag{7.6}$$

$$p = t_a(q) = sequence(p_{f_i}) \tag{7.7}$$

Once again, the sequencing function $sequence$ does not need to preserve the order of features in the input query; rather it decides on the order of the procedures based on the overall optimization rules and the adapter's preferences.

7.2.3. Schema Discovery

Each query must have a perspective that shapes its result set. Perspectives can be declared in various ways. We use schema discovery to identify, consolidate, and assign a perspective to a query. QUIS's schema-discovery process checks the following sources in order to identify a perspective:

1. Explicit perspectives: Each query can declare its perspective. Such an explicit perspective should have been defined in advance. The benefit of explicit perspective declaration is its potential reusability among other queries;

2. Inline perspectives: If a perspective is not designed or intended for reuse, it can be declared alongside the containing query as part of the query's projection clause. Inline attributes cannot declare explicit data types; the data types are inferred from the expressions that the inline attributes are built upon. However, an inline perspective attribute is able to reuse an attribute from an already existing explicit perspective;

3. Implicit perspectives: When a query operates on the result set of another query without altering its schema, it can implicitly inherit its perspective. This scenario is mostly designed for query chaining that applies a series of different selections, paging, and/or sorting; and

4. Inferred perspectives: If a query does not declare or inherit a perspective, the schema of its underlying data container will be assumed. This scenario is used in agile queries that do not require ETL operations, as well as for schema extraction when the actual data schema is unknown to the user.

When the perspective(s) of a query are identified, we apply any of the following cases to derive an effective perspective for the query. Consolidation occurs in the following cases:

1. Perspective inheritance: In QUIS, perspectives can inherit from each other using the single-parent inheritance paradigm. Therefore, if a perspective $child$ has extended another perspective $parent$, we build a union of their attributes;

2. Attribute overriding: If a child perspective declares an attribute with a name that is present in one of its descendants, we override the attribute in favor of the child. Overriding, maintains the forward and reverse mappings as well as the data type of the child and removes the inherited counterparts; and

3. Perspective merging: In case of compositional queries, e.g., join and union, we merge the attributes of the sides into a new perspective. Name conflicts will be solved by prefixing the attributes' names.

The consolidated perspective is assigned to the query as its effective (runtime) schema. The result set of the query is structured with reference to the effective schema. The schema-discovery process begins with the parsing, but it can be deferred to the transformation phase if it needs to be performed by accessing the data source's actual schema.

7.2.4. Transforming Data Types

At the end of the semantic analysis, each query is bound to a perspective, either explicit or implicit. As shown in Grammar 6.7 each attribute has a virtual data type that can be declared explicitly or inferred from the query's underlying data source. `QUIS` supports *Boolean*, *byte*, *integer*, *long*, *real*, *date*, and *string* data types. When data types are not explicitly declared, `QUIS` attempts to infer them from their context: It first looks up the schema of underlying data; thereafter, it attempts to extract the type from the expressions that the attribute is used in. `QUIS` also examines the aggregate and non-aggregate function parameters and return types, as well as the arithmetic and logic operators that are applied to the attributes. In any case, the attributes' data types must be known during the transformation phase. This is because the transformers need to build casting and/or converting transformations from virtual data types to concrete counterparts, in addition to reformatting query results in order to comply with the original query's data types.

Each adapter is required to provide a two-way mapping table that translates each of the virtual data types to its corresponding concrete type. If an adapter supports more than one dialect, it should provide a table for each of the dialects. The tables additionally indicate whether a conversion method or a casting should be applied. This adapter- (and dialect-) specific table is used by the data type transformer sub-routine to apply the appropriate templates and/or functions onto the referenced data items of the target query as well as its projection operator.

7.3 Query Complementing

One of our fundamental goals was to address the problem of data access heterogeneity. Data sources vary in their capabilities, which contribute to data access heterogeneity. The capabilities of interest may not be equally available on all data sources; for example, selection is not available via spreadsheet system APIs, sorting is not an out-of-the-box feature of MapReduce systems, and the domain-specific aggregates used in scientific research may not be of interest for general purpose tools and hence may not be available. Our core solution for dealing with heterogeneity was federation. In addition, because of our focus on in-situ querying, we shifted from database-centric solutions to data-querying systems. In order to address functional heterogeneity and maintain our language's unifiedness, we incorporate *query complementing*.

Query complementing is a technique used to transparently detect and compensate for the features that a chosen adapter may lack when executing a given query. It consists of two phases, namely capability negotiation and query rewriting. Capability negotiation enumerates all of the capabilities announced by the available adapters and matches them against the requirements of

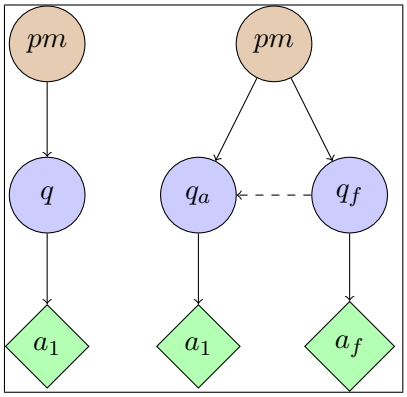

Figure 7.4.: The query q is not fully satisfied by the capabilities of its associated adapter, a_1. Hence, a complementing query is built. The right-hand side figure depicts the ASG after complementing is performed, in that q is broken into q_a and q_f to be executed on the original adapter a_1 and the fallback adapter a_f, respectively. The complementing query q_f depends upon the rewritten version of the original query, q_a.

the input query in order to nominate an adapter that provides the best possible coverage (see Section 8.1.2). The query rewriter breaks the input query down into a partial query that can be run on the selected adapter plus into another complementing query comprised of the rest of the input query that can be run on a special adapter, namely the fallback adapter. The query rewriter additionally binds these two together so that their sequential execution produces the expected result set of the original input query.

Figure 7.4 depicts a query q that has been chosen (by the adapter selection algorithm) to be executed on adapter a_1. Assume that q requires features $R = \{f_1, f_2, f_3, f_4\}$ but a_1 only exposes capabilities $C = \{f_1, f_4\}$; therefore, the query must be complemented. This results the fallback adapter being engaged to accept $R \setminus C = \{f_2, f_3\}$ features and to leave only $R \cap C = \{f_1, f_4\}$ to a_1. The right-hand side of the figure shows that q is broken down into q_a and q_f in order to be executed on the original adapter a_1 and the fallback adapter a_f, respectively. The directed-arrow from q_f towards q_a indicates that the complementing query q_f depends upon q_a.

Feature 11 (Query Complementing)

If there are mismatches between the query requirements and the capabilities of the winning adapter, the query transformer rewrites the original query to whatever the chosen adapter is able to perform and introduces a complementing query to be executed internally.

The query complementing technique is shown in Algorithm 7.1. The chosen adapter $adapter$ accepts $R \cap C$ query features only; hence, the algorithm builds a partial DST dst_a for the

features that it supports (`line 8`). The algorithm additionally builds a new DST dst_f on the fallback adapter $fallback$ to fulfill the missing $R \setminus C$ query features (`line 9`). The $build$ function accepts a set of query features and builds a DST out of it. The dst_f DST is declared to be dependent upon dst_a. This dependency directs the execution engine to first run dst_a and to pass its result to dst_f. The result set of the original input query is then obtained by executing the dst_f DST.

Algorithm 7.1 Query Complementing

Input: An input sub-/query in the form of a DST node and an adapter $adapter$.

Output: Two complementing queries dst_a and dst_f that together satisfy the requirements of dst.

```
1:  function COMPLEMENT(dst, adapter)
2:      R ← enumerateRequirements(dst)
3:      C ← enumerateCapabilities(adapter)
4:      if (R \ C = ∅) then
5:          return dst
6:      end if
7:      fallback ← fallbackAdapter()
8:      dst_a ← build(R ∩ C)
9:      dst_f ← build(R \ C)
10:     processModel.add(dst_a)
11:     processModel.addDependency(dst_f, dst_a)
12:     return dst_f
13: end function
```

It is worth mentioning that, during later steps of the execution flow, the $adapter$ and $fallback$ adapters transform their built queries dst_a and dst_f into their relevant computation models. For example, the chosen adapter may transform dst_a into a relational query, while the fallback adapter generates an imperative function to satisfy the operators delegated to it. The rewritten and newly generated queries are added to the process model alongside the dependency.

One of the scenarios in which query complementing is very helpful is in queries that perform join operations on heterogeneous data sources. It is a common case that none of the data sources of the sides of the join operation ingest and operate on the data obtained from the other side. In addition, due to the diversity in terms of the capabilities of data sources, we may often be limited in the choice of join algorithms available to us. Our solution is to use the query complementing technique to rewrite such heterogeneous joins and develop a basic join algorithm that can perform adequately even with data sources that have limited capabilities and then to potentially add more efficient join methods where data sources can support them.

`QUIS` compares both sides of joins to determine whether they are accessing heterogeneous data sources. If this proves to be the case, it breaks the original query down into a left-deep join tree in which the leaf nodes are the data access nodes and all of the upper level nodes are the

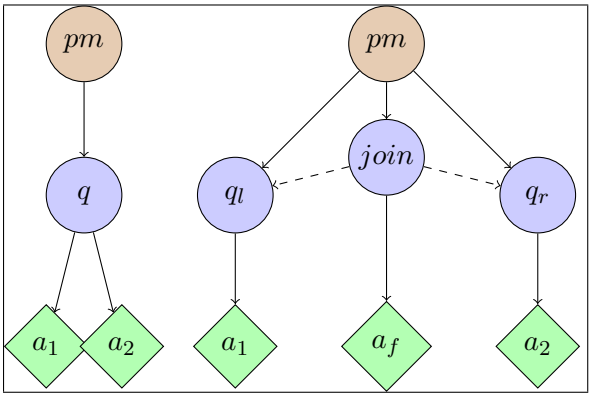

Figure 7.5.: The query q is a join of two heterogeneous data sources; each should be transformed and executed by its associated adapter, a_1 and a_2, respectively. The query is rewritten as two standalone side queries, q_l and q_r plus a composition query $join$ that performs the join on the results of the two side queries.

joins. It then transforms the leaf nodes into standalone queries tailored to retrieve data from their corresponding data sources. The engine also creates a compositional query for each join node in the tree, meaning that the composition query accepts the result sets of the two sides, performs the requested join operation, and bubbles the result up the tree.

Figure 7.5 shows a query q that performs a join operation on two heterogeneous data sources. The adapter selection algorithm has assigned two adapters a_1 and a_2 to it, meaning that each data source will be accessed via one of the adapters. Assume that none of the assigned adapters have capabilities required for the requested join operation: The QEE activates the complementing algorithm to i) rewrite the original query into two sub-queries that each access one of the data sources via the assigned adapters and ii) to factor out the join operation and substitute it with a complementing query to be run on the fallback adapter. The resulting process model shows that the join's left-hand side q_l will be executed on a_1, the join's right-hand side q_r will be executed on a_2, and the $join$ query utilizes a_f for execution. The directed arrows from $join$ towards q_l and q_r indicate that the complementing query $join$ depends upon q_l and q_r and therefore waits for their result sets to be ready before performing its job.

The standalone side queries undergo the same query-complementing phase and may be further break down accordingly. Figure 7.6 shows a query q that is similar to that featured in Figure 7.5. However, in addition to the join operation, q uses a sorting operator on both of the data sources, which is a feature that none of the assigned adapters have matching capabilities for. The QEE rewrites the query as described in the example provided in Figure 7.5. As the preparing algorithm recursively attempts to detect and complement the lacking features, it recognizes the lack of sorting operation on both sides. It therefore initiates a second round of query complementing, this time once for q_l and once for q_r. In plain text, q_{l_c} and q_{r_c} sort the result sets of q_l and q_r, respectively. They feed the sorted partial results into the $join$, which in turn performs the requested join operation and yields the final result set. The resulting process model shows the

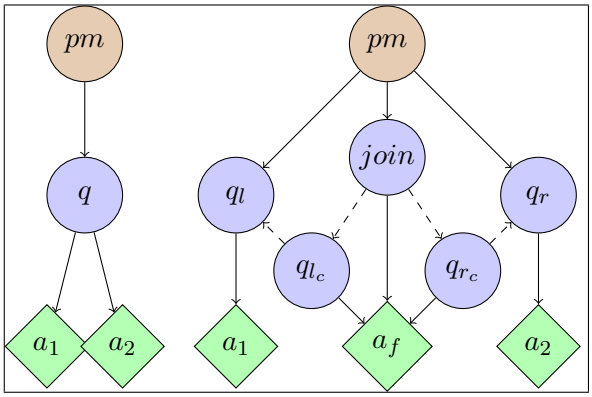

Figure 7.6.: The query q is a join of two heterogeneous data sources; each should be transformed and executed by its associated adapter, a_1 and a_2, respectively. The query is rewritten as two standalone side queries q_l and q_r, plus a joining query, $join$. However, the nominated adapters do not satisfy the capabilities required by the side queries; hence, two complementing queries q_{l_c} and q_{r_c} are constructed for the sides. The right-hand side figure depicts the ASG after the joining and complementing have been performed.

join's left-hand side q_{l_c}, q_{r_c}, and $join$ are assigned to the fallback adapter a_f for execution. The dependencies also control the execution and result set flow.

Using this technique, QUIS is able to run join queries on any combination of RDBMS, Excel sheets, CSV files, and previously fetched result sets. Homogeneous joins are easier to manage, as both their sides use the same data source or adapter. The query engine does not split the sides; instead, the entire join clause is shipped to the designated adapter for transformation.

7.4 Query Optimization

In the preceding sections, utilizing transformation techniques, we demonstrated how we can effectively manage data access heterogeneity through query virtualization. The result is a system in which users are able to include data from remote data sources in complex queries with very little work. However, such a system will not be used if it demonstrates poor performance. In this section, we describe how QUIS satisfies this requirement. The systemic challenge is query optimization over heterogeneous data sources that often demonstrate unpredictable performance and possess unknown statistics [DES+15].

QUIS is intended to support exploratory research in environments that feature volatile data, in which very limited auxiliary data is available to the engine; therefore, query optimization must often be performed in a zero-knowledge environment. This absence of cost and size information limits us to rule-based optimizations; we use the federated structure of the system to address this dilemma. Query planning can be performed at three levels, namely the query engine, the adapter,

and the data source. While the query engine uses rule-based optimizations, the adapters and data sources may perform cost-based re-optimizations to the extent that they have access to auxiliary data for their sub-queries. For example, an RDBMS data source obviously performs its own (traditional) optimization before executing a query shipped to it [Cha98]. More interestingly, adapters can optimize queries by utilizing the cost factors measured at the time transformation, e.g., file size, record size, or historical and/or statistical data.

7.4.1. Optimization Rules

Assuming that a query q operates on a data container D so that $|D|$ is the cardinality of D, A_D is the set of attributes of D, $A_{qD} = \{a|a \in q(D)\}$ is the set of attributes of D requested by q, and $X(A_{qD}) = \{a|a \in q_x(D)\}$ is the set of attributes of D requested by operator X of query q. X is either one of the projection(π), selection(σ), join($\bowtie$), grouping(Ω), or ordering(O) operators. We define $f_{qD} = |A_{qD}| \,/\, |A_D|$ as the fraction of the attributes of D retrieved by q and selectivity ratio s_{qD} to be the selectivity% normalized to $0 \leq s_{qD} \leq 1$.

7.4.1.1. Selective Materialization

If the projection clause of a query statement requests a subset of the attributes of a dataset, the query plan loads only the union of the attributes declared by the projection, selection, join, ordering, and grouping clauses. Assuming $A_{qD} \subset A_D$, the query plan materializes A_{qD} attributes only.

This improves the query performance by a factor proportional to $(1 - f_{qD}) * |D|$. It also reduces the memory footprint by the same factor, as it avoids materializing unnecessary attributes. In column-based data stores, this rule prevents the query executer from touching non-requested columns. These savings have a greater impact while processing file-based data containers, where obtaining a tuple includes expensive operations such as file reading, string parsing, tokenization, type conversion, and object materialization.

7.4.1.2. Lazy Materialization

Lazy materialization is a technique that is applicable to queries that have a WHERE clause. This rule causes the optimizer to derive the attributes referred to by the WHERE clause's predicate and rewrite the query in such a fashion that, at the time of execution, only those attributes are materialized. In other words, assuming $A_{eff} = \sigma(A_{qD})$, the query plan tests the query's predicate on A_{eff} only. It materializes the rest of the attributes, $\{A_{qD} \setminus A_{eff}\}$, only if the test passed; otherwise, the $|\{A_{qD} \setminus A_{eff}\}|$ attributes of the examined record are left untouched. Other rules, such as selective materialization (Section 7.4.1.1), can then be applied. The total performance gain would be proportional to $(1 - (|A_{eff}|/|A_{qD}|)) * (1 - s_{qD}) * |D|$.

7.4.1.3. Push-Ahead Selection

In an inner join, if a selection is present and its predicate refers to the attributes of the schema of the join's outer side only, it is possible to evaluate the predicate using the loaded outer records in order to generate a more efficient query plan [AK98]. Therefore, in $R \bowtie S$, iff $\sigma(A_{q(R \bowtie S)}) \subseteq A_R$ holds, then instead of $\sigma(R \bowtie S)$, $\sigma(R) \bowtie S$ is computed [AK98]. This avoids the need to access the records of S for those records of R that the predicate evaluates to false. The total reduction in the query execution cost would be: $((1-f_{qR})*(1-s_{qR})*|R|)+((1-s_{qR})*|R|*|S|)$. The first part of the formula is similar to performance gain obtained by the application of the lazy materialization rule (Section 7.4.1.2). The second part of the formula is obtained by skipping the inner relation for the outer records with the failed predicate. As selection, if present, must be performed in any case, pushing it ahead of the join operation does not increase the query cost; it only changes the order of execution.

7.4.1.4. Eager Join Key Materialization

If a query contains a join $R \bowtie S$, it is possible to evaluate the join condition first. This requires materializing $A_j = \bowtie (A_{q(R \bowtie S)})$ attributes only, which is $JoinKey(R) \cup JoinKey(S)$, excluding those attributes already materialized for the WHERE clause by other rules. If the join condition fails, the right-hand side record can be safely ignored, and (depending on the join type) no resulting record needs to be materialized. The total performance gain of this rule is $(1-f)*(1-s_{q(R \bowtie S)})*|R \bowtie S|$, in which $f = |A_j|/(|A_{qR}| + |A_{qS}|)$. Instead of union, $+$ is used to indicate that R and S have no join key in common. If present, the push-ahead selection rule (Section 7.4.1.3) may apply before this rule in order to reduce the cardinality of the sides.

7.4.1.5. Running Aggregate Computation

Aggregate functions use the running mechanism $agg_{n+1} = f(agg_n, value_{n+1})$ to calculate values. Instead of requiring and waiting for all of the inputs to arrive, they simply compute the next running values on the data items as they arrive and maintain an internal state of the computation performed. The state is then updated upon the arrival of new input items. At any given time, the state object will hold the correct value. For example, the running version of the average function can be formulated as $avg_{n+1} = (avg_n*n+value_{n+1})/(n+1)$. Implementation of the state object can be done using either (avg_n and n) or (sum_n and n). This technique relieves the aggregate functions of the need to keep track of all input records and reduces the memory footprint dramatically, especially for large datasets and multi-aggregate queries.

7.4.1.6. Join Ordering

Joins are associative and commutative [Cha98], which gives us the liberty to reorder them. A proper join order can dramatically affect the overall performance of a query. In non-DBMS

data sources, specifically in file-based data sources, there is no easy way to calculate query cost, as it heavily relies on the estimation of the number of tuples passing through the query operations. A raw estimation of the number of tuples can be obtained by dividing the file size by the tuple size. A realistic estimation of the tuple size can be obtained from the tuple schema. Thus, the file size divided by the schema size yields an estimation of the number of tuples, which can be used to decide whether to retain or swap the join order. In heterogeneous file-based joins, it is better to have the larger data container in the inner side in order to reduce the number of `file open` and/or `file seek` operations. If a swap is required but is not performed, the cost would be $max(n, m) * (cost(min(file\ open, file\ seek)))$; otherwise it reduces to $min(n, m) * (cost(min(file\ open, file\ seek)))$. Thus, the total gain is proportional to $max(n, m) - min(n, m)$, hence to $abs(n - m)$.

7.4.1.7. Weighted Short-Circuit Evaluation

The conventional minimal expression evaluation method evaluates functions and/or operators from left to right, considering precedence, and returns the evaluation result as early as it is determined. By assigning a cost factor to functions and operators and building a weighted evaluation tree, it is possible to calculate the cost of each evaluation path and evaluate the cheaper paths earlier. This improves the overall performance of expression evaluation by determining the expression's value through evaluating the cheapest paths. For example, in expression $(f1 \wedge f2) \ni cost(f1) > cost(f2)$, the conventional evaluation would call the $f1$ first and, if necessary, $f2$ thereafter. Building the weighted evaluation tree allows the $f2$ to be called first in order to reduce the overall expression evaluation cost. The weight of each function/operation is proportional to its required CPU time and is preserved as part of its metadata; adapters can override the weights if needed. `QUIS` uses an experimental weighting scheme to measure the effectiveness of this rule.

7.4.1.8. Right Outer Join Equivalence

Right outer joins are transformed to their left outer join counterparts [RG00] by swapping the left and right data containers. For example, `(A` **`RIGHT OUTER JOIN`** `B)` can be transformed to its equivalent `(B` **`LEFT OUTER JOIN`** `A)`.

This is not a standalone optimization, but it makes it possible to apply other rules. For example, when swapped, the optimizer checks the application of the push-ahead selection rule (Section 7.4.1.3), eager join key materialization (Section 7.4.1.4), and join ordering (Section 7.4.1.6) in order to further optimize the query.

7.4.1.9. Result set Bubbling

RDBMSs execute optimized query plan that are represented as an operation tree. The execution begins with the leaves, which access data (records and/or indexes) and pass the resulting relations

to their corresponding upper operation nodes in the tree. As operations accept zero or more relations and return a single relation, a set of temporary relations are built during the query execution process in order to retain the intermediate results. Query optimizers attempt to reduce this cost by maximizing the utilization of the pipelined evaluation [RG00].

QUIS also, constructs a pipeline representation of the query tree. However, it uses the result set schema to build and compile tailored concrete operators on-the-fly in order to minimize the need for inter-operator data transformation and/or temporary memory allocation. It also builds a tuple structure based on the effective perspective of the input query. These dynamically built operators are executed at runtime against the data source(s) and build and materialize matching tuples. Therefore, all of the operators are able to operate on a single result set that bubbles through the operation tree. The result set is eventually populated. This mitigates the need to maintain multiple intermediate relations and perform merging at upper level nodes. The rule does not apply in some cases, e.g., when both ordering and limiting clauses are present.

7.4.2. Optimization Effectiveness

In order to gauge the effectiveness of our optimization rules, we conducted a set of comparative experiments, switching on and off the rules and measuring the elapsed times (for more details, see Chapter 11). The queries were executed on the FNO and SMV datasets (see Section 11.1.1). Table 7.1 summarizes the results.

Rule	Average	Maximum
Selective Materialization	47%	70%
Lazy Materialization	14%	29%
Push Ahead Selection	26%	51%
Eager Join Key Materialization	22%	43%
Weighted Short-Circuit Evaluation	18%	18%

Table 7.1.: Effectiveness of the optimization rules. The values indicate the performance gained by enabling the rules.

The selective materialization rule improved the query time by an average of 47% at different projection ratios and up to a maximum of 70% at a projection ratio of 5%. The lazy materialization rule resulted in a 14% performance gain on average when the selection predicate accessed 2.5% of the dataset's attributes. Its greatest improvement was 29%, which was obtained at 0% selectivity. Enabling the push-ahead selection rule on the SMV dataset with a projection ratio of 20% resulted in an average performance gain of 26%. The greatest impact, of approximately 51%, was obtained at a selectivity of 0%. The impact of the eager join key materialization rule was 22% on average and 43% at maximum. Its overall behavior was similar to that of the push-ahead selection rule, as join keys performed similarly to pushed ahead attributes. Applying the weighted short-circuit evaluation rule on a query with a predicate $(f1(x) \wedge f2(x)) \ni w(f1) = 5w(f2)$ boosted performance by approximately 18% at all selectivity levels.

8 Query Execution

Query execution, although focused on the execution of input queries, manages all aspects of queries' life-cycles. The life-cycle begins by parsing the input queries, continues with building the DSTs, selecting adapters, complementing the queries if needed, and finishes by executing the queries and returning the result sets to the client in an appropriate format. These tasks are managed and orchestrated by a Query Execution Engine (QEE).

In brief, the QEE first compiles a given input query to a DST (Section 8.1.1) and then selects the appropriate adapter(s) responsible for the transformation and execution of the DST (Section 8.1.2). Query complementing (see Section 7.3) is also performed in this step. When the DST is transformed, its transformed computation model(s) is compiled to the executed jobs (Section 8.1.3). The executed jobs are submitted to the adapters for execution, and their results are formatted according to the input query requirements (Section 8.1.4).

Although the QEE realizes the workflow, not all of the steps involved are necessarily parts of the QEE. For example, adapters, as an important part of `QUIS` in that they provide the majority of the transformation and execution services, are not part of the engine. However, they are integrated, and interact, with the engine. We explain the specification of adapters in Section 8.2. Furthermore, complementing and compositional queries are executed on the fallback adapter.

8.1 The Query Execution Engine

As the depiction of our system architecture in Figure 5.1 shows, the QEE is activated by the runtime system upon the submission of one or more queries by a client through the APIs. The QEE's workflow is defined in Algorithm 8.1: It consists of four classes of tasks, namely preparation, transformation, compilation, and execution. Among these classes, we discussed DST construction and transformation in Chapter 7. We detail the other classes in this section, after describing the execution flow algorithm.

Lines `3` to `6` of Figure 5.1 list the preparation tasks: The input process, which consists of all of the submitted queries, is parsed and converted into a set of *Described Syntax Trees (DSTs)* (Definition 5.1). These DSTs are added as the nodes of an ASG *Annotated Syntax Graph (ASG)* (Definition 5.2). The ASG acts as the intermediary contract between the language, the QEE, and

Algorithm 8.1 Query Execution Flow

Input: A set of input queries $queries$.
Output: The result sets of $queries$, maximum one result per $query$.

```
1:  processModel ← newASG()
2:  function EXECUTE(queries)
3:      for query ∈ queries do
4:          dst ← parse(q)
5:          prepare(dst)
6:      end for
7:      for dst ∈ processModel do
8:          optimize(dst)
9:          dst.transformation ← transform(dst, dst.adapter)
10:     end for
11:     compile(processModel)
12:     assignGeneration(processModel)
13:     for dst ∈ processModel.generations.ordered(ASC) do
14:         parallel dst.result ← dst.job.execute()
15:     end for
16:     for dst ∈ processModel do
17:         if isRoot(dst) then
18:             return present(dst.result)
19:         end if
20:     end for
21: end function
```

the adapters. The parser detects data flow between queries and creates inter-query dependency links among the DSTs in the ASG. This dependency information is used during transformation and execution. The preparation routine (see Section 8.1.1) selects a (set of) adapter(s) that best fit the query requirements (see Section 8.1.2), decomposes the DSTs of each of the queries, and determines whether complementing is required (see Section 7.3).

In the transformation block (`lines` $7-10$), the QEE calls on the optimizer to apply the relevant optimization rules to each of the DSTs (see Section 7.4). It then ships the optimized queries to the designated adapters and requests them to transform the queries into their corresponding native computation models (see Chapter 7 (Query Transformation)).

The end of the transformation signals the QEE to begin the compilation process (`lines` $11-12$). Compilation is the process of wrapping the transformed queries into a set of execution units, i.e., jobs, and actually compiling them on-the-fly into machine-executable code. The compiled jobs are assigned a *generation* index proportional to their dependency depth in the ASG. Further details concerning these two steps are provided in Section 8.1.3.

The execution block (`lines` $13-20$) performs the final two steps: executing the jobs and presenting the result sets to the requesting clients in the specified format (see Section 8.1.4). As shown in line `13`, the QEE selects all of the jobs associated with a generation index at each iteration, starting from generation zero and moving upwards. It then calls for the execution of all of the selected jobs in parallel. The result set of each query comes back in the form of `QUIS`'s data model (see Section 6.4.6), which is converted to the presentation format requested by the input query. Converting the result sets into presentation formats is the task of the $present()$ function line `18`. It can convert the result set to a table, a chart, or an XML or JSON (see Section 8.1.4.1).

8.1.1. DST Preparation

While we outlined the parsing and preparation activities in Chapter 5 (Overview of the Solution), we describe them in greater detail in this section. Query preparation begins with parsing. Parsing follows the three stages of lexical, syntactical, and semantical analyses. Lexical and syntactical analyses are traditional parsing processes that, in our solution, result in a process model that is formulated as an ASG. The semantic analysis revisits the ASG to discover the declared schemas (see Section 7.2.3), resolve data types (see Section 7.2.4), and validate bindings against their corresponding connections. These activities enhance the DSTs' metadata, which is used in the following stages.

When the semantic analysis has been performed and the DSTs are adequately annotated, the query preparation, which is listed in Algorithm 8.2, begins. This algorithm has two roles: it selects appropriate adapters to transform and execute each of the DSTs and complements query requirements that were not fulfilled by the chosen adapters.

The main block of the preparation algorithm (`lines` $7-12$) selects an adapter that best serves the requirements of the given input query dst (see Algorithm 8.3). However, it is not guaranteed

Algorithm 8.2 Query Preparation

Input: The dst of an input query $query$.
Output: An adapter to serve the query as well as a complementing query in required.

```
1:  function PREPARE(dst)
2:      if isComposite(dst) then
3:          l ← prepare(dst.left)
4:          r ← prepare(dst.right)
5:          dst ← compose(l, r, dst)
6:      else
7:          dst.adapter ← selectAdapter(dst)
8:          R ← enumerateRequirements(dst)
9:          C ← enumerateCapabilities(dst.adapter)
10:         if (R \ C ≠ ∅) then
11:             dst ← complement(dst, adapter)
12:         end if
13:     end if
14:     processModel.add(dst)
15: end function
```

that the selected adapter will fulfill all of the requirements of the input query dst. Therefore, the algorithm enumerates both the adapter's capabilities and the query requirements in terms of query features (see Section 6.5) and computes their differences. If the difference is not an empty set, the algorithm complements the query by assigning the difference (the features not supported by the adapter) to the fallback adapter (see Section 7.3).

If the input query is a composite query that consists of joins or unions, the algorithm recursively decomposes it into its components, prepares the components individually, and recomposes the prepared versions (`lines 2 − 5`). The recursive nature of the algorithm allows it to traverse multi-side joins, select the appropriate adapters for each side, and complement the sides if the assigned adapters have shortcomings in terms of capabilities. The *compose* function, which fuses the component queries after preparation, may use the complementing technique to perform the composition operation (join/union) if none of the component adapters are capable of handling it. The final prepared DSTs are registered in the process model (ASG) for subsequent operations.

8.1.2. Adapter Selection

Adapters differ in the capabilities that they provide as well as their execution costs. Assume F is the set of all supported query features, a given query q with requirements $R_q \in F = \{r_1, r_2, ..., r_n\}$, and a set of available adapters $A = \{a_1, a_2, ..., a_k\}$. Each adapter a_i has the capability set $C_i \in F = \{c_{i1}, c_{i2}, ..., c_{im}\}$. The execution of each capability c_{ij} on adapter a_i costs w_{ij}. Our goal is to select the adapter with the minimum cost $\min_{a \in A} cost(q, a)$ in that

$cost(q, a)$ is the total cost of executing query q on adapter a plus the cost of executing the lacking features, those features that adapter a can not execute, on the fallback adapter, if necessary. Setting higher costs on the fallback adapter encourages more feature-rich and faster data source-specific adapters. The fallback adapter reposts unbound (very high) costs on concrete data access operators in order to avoid replacing the actual adapters. We have designed an adapter selection algorithm that follows the above-mentioned specification; it is illustrated in Algorithm 8.3.

Algorithm 8.3 Adapter Selection

Input: The DST (Described Syntax Tree) of a query.

Output: Adapter that best satisfies the requirements of the DST. If DST accesses more than one data source, the algorithm may select more than one adapters.

```
1:  function SELECTADAPTERS(dst)
2:      fallback ← fallbackAdapter()
3:      costs[] ← ∞
4:      R ← enumerateRequirements(dst)
5:      for dataSource ∈ dst.Datasources do
6:          for adapter ∈ catalog do
7:              C ← enumerateCapabilities(adapter)
8:              D ← R \ C
9:              S ← R ∩ C
10:             ct ← cost(S, adapter) + cost(D, fallback)
11:             if ct < costs[dataSource] then
12:                 costs[dataSource] ← ct
13:                 selectedAdapters[dataSource] ← adapter
14:             end if
15:         end for
16:     end for
17:     return selectedAdapters
18: end function
```

Upon receiving the DST node of a (sub-)query, the algorithm enumerates its requirements (`line 4`). It then enumerates the capabilities of each registered adapter (`line 7`) and calculates the total execution cost (`line 10`) of running the query on each adapter. The algorithm selects the adapter with the lowest overall execution cost. The execution cost is computed by adding the cost of running $R \cap C$ query features on the chosen adapter to the cost of the remaining features, $R \setminus C$, on the fallback adapter. The cost function $cost(C, a_i) = \sum_{j=1}^{|C|} w_{ij} \ |c_i \in C \wedge C \in F$ sums up the costs of all c_i features of the C as reported by adapter a_i.

The required information for the $enumerateRequirements$ function is obtained from the query's DST node. The $enumerateCapabilities$ function accesses the capabilities that are exposed by the adapter; this is a service that all of the adapters must provide (see Section 8.2). The granularity of the capabilities is flexible and can be adjusted by the query engine. In our proof of

concept implementation, we set them at the feature level, as listed in Section 6.5 (QUIS Language Features). It is possible that the algorithm may return more than one adapter if the query contains heterogeneous data sources.

Feature 12 (Capability Negotiation)

The system is be able to detect a target data source's partial inability to run a given query and select the adapter that can execute it with the lowest overall cost.

8.1.3. Query Compilation

The QEE run jobs as units of execution, in which each job represents a transformed version of a DST. Jobs must be created before execution; furthermore, they must be executable on their designated data source and native to the computation model. We compile queries to create jobs. Query compilation, as shown in Algorithm 8.4, is straight-forward: It first wraps the transformations of all the queries into a set of job sources (`lines 2 – 5`) and then compiles all the job sources to executable jobs, on-the-fly and all at once (`line 6`). Each job obtains its executable code from the executable units generated by the compiler (`line 8`).

The ASG contains inter-query dependencies that are either imposed by the chained queries or the results of query complementing and/or composition. The $assignGeneration$ function traverses

Algorithm 8.4 Query Compilation

Input: The process model with all its DST nodes.
Output: The compiled and executable jobs for each query in the process model.

```
1:  function COMPILE(processModel)
2:      for dst ∈ processModel do
3:          dst.job.source ← applyTemplate(dst.transformation)
4:          compilationUnit.add(dst.job.source)
5:      end for
6:      executableUnits ← compiler.compile(compilationUnit)
7:      for dst ∈ processModel do
8:          dst.job.executable ← executableUnits[dst.job.source]
9:          dst.job.generation ← assignGeneration(dst)
10:     end for
11: end function
```

the ASG to detect the dependencies of the current dst. When found, it assigns a *generation* index that is equal to the dst's longest dependency depth. The generation index of the DSTs with no

dependencies is set to zero. For composite queries with multiple data sources, the generation index is set to the maximum of the generation indexes of the component queries.

Using this technique, the first generation consists of those (partial-)queries that have no dependencies (except for their own data sources). The next generations can consist of queries that have JOINs, UNIONs, or complementing queries. The number of generations depends upon various factors, e.g., depth of JOINs and the adapters' abilities to address the input queries. The latter affects query complementing.

Feature 13 (Dynamic Query Compilation)

The system wraps the transformed queries into executable units and compiles them on-the-fly to create independent and self-sufficient jobs..

8.1.4. Job Execution

The overall flow of executing compiled jobs is illustrated in Algorithm 8.1, lines $13-15$. As the first step, the QEE categorizes the jobs for parallel execution. To do so, it sorts all of the DSTs in ascending order based on their generation index and selects them iteratively. During each iteration, the QEE selects a subset of DSTs that have the same generation index. The generation assignment (which is accomplished during query compilation) ensures that any given job is executed only when all of its containers are ready. The QEE then concurrently ships the selected jobs to their corresponding adapters for parallel execution. Serial execution is also possible if insufficient computational resources are available. In such a case, the jobs are put into a single queue and are shipped sequentially to the corresponding adapters.

Upon receiving the jobs, the designated adapter executes them against the data source. Each job carries its raw transformation, as well as the DST and the relevant parts of the input query. This provides the adapter and/or the underlying data source with the information required for potential further optimization. Complementing queries are shipped to the fallback adapter using a similar job shipping procedure; thus, the executor does not need to distinguish between the fallback adapter and the others. Complementing and composition queries are always dependent upon other queries; hence, they are scheduled for the second or later generations. This ensures that their containers are ready by the time of execution. By executing jobs in generations, the QEE eventually builds the query results, which are later presented to the query clients.

8.1.4.1. Query Result Presentation

The result set of each query is returned in the form of `QUIS`'s data model (see Section 6.4.6). In summary, the data model is a bag of potentially duplicate tuples. Each tuple consists of a list

of values. The values obtain their definitions from the attributes of the effective perspective that is associated with the query's projection. These result sets should be presented to the clients who requested them; however, clients may request alternative representations. QUIS provides a mechanism that allows the query authors to declare how each query result should be presented. We call this mechanism *Polymorphic Query Result Presentation (PQRP)*.

PQRP is responsible for determining the declared presentation method from the query, applying it to the result set in order to build the requested presentation, and delivering it to the client. The supported presentation methods are included in the target selection clause of QUIS's grammar (see Section 6.5.2.7 (Target Selection)). We explain them in greater detail bellow:

1. Tabular query result presentation: The query result is arranged in a bag of rows in such a fashion that each row represents a single data entity. For each row, the output displays a set of columns, with each representing one of the attributes of the query's effective perspective. This method of presentation is similar to the result rendering in traditional IDEs of RDBMSs, e.g., those of MS SQL Server, Oracle, PostgreSQL, and MySQL;

2. Serialized query result presentation: In many cases, data workers need to transform and then transfer data to other tools for further processing and/or analysis. In these situations, the target of the query can be set to one of the supported serialization formats. By default, QUIS supports XML and JSON serialization; however, adapters could optionally register new methods. It is also possible to exploit this feature to transfer data between various data sources. For example, it is possible to query data from a spreadsheet and directly submit it to a table in an RDBMS;

3. Visual query result presentation: Query results can be visualized in order to provide superior insight into the data. QUIS allows for different types of visualizations to be declared as the targets of queries. In these cases, the result sets are directed to the chosen visualization instead of being presented or persisted. For example, it is possible to draw a line chart that depicts a country's population in different years by introducing the *year* and *population* attributes of the query's result set to the charting clause. Bar, pie, and scatter charts are also supported; and

4. Client query result presentation: If a presentation is requested by a third-party client system that interacts with the system's APIs, e.g., the R-QUIS package for R, the required presentation methods are not handled by the language; they are instead delegated to the client, allowing to perform its own transformations (see Figure 5.1 in Chapter 5 (Overview of the Solution)). In such a case, the QEE assigns the result set to the variable nominated by the **INTO** clause of the input query in order to make it accessible to the client.

We have designed the QEE to perform PQRP after the result sets are materialized. This design decision 1) reduces the load on the target adapters and/or their underlying data sources, 2) centralizes the presentation concept, logic, and implementation in one location, 3) isolates the query operators from the presentation requirements, and 4) ensures that the PQRP feature is always available. In our design, the fallback adapter is equipped with a special component that is intended to handle PQRP.

8.2 Adapter Specification

Input queries operate on various data sources and demand different capabilities. Capabilities are available through adapters; each adapter supports query execution on one or more data sources. Similar data sources are categorized as dialects. Therefore, adapters may support multiple dialects.

Adapters play two important roles: They are responsible for query transformation (see Chapter 7) and query execution (addressing in the current chapter). In order to realize these two roles, adapters must have the following elements:

1. In order to satisfy the input query requirements, adapters should provide transformations capabilities for each query feature to ensure that a given input query can be transformed to its equivalent in the target computation model. However, the query complementing technique allows the adapters to provide less than full capabilities. At minimum, each adapter must expose capabilities for reading and writing records from and to its underlying source; the rest is managed by the complementing algorithm and the fallback adapter;
2. The adapter selection algorithm (Algorithm 8.3) enumerates the capabilities of adapters to identify the best match for the query requirements. Hence, the adapters are required to expose the capabilities that they support alongside their execution costs. In its simplest case, this takes the form of a list of supported capabilities, each of which has an associated execution cost index. This information is also used during query complementing (see Section 7.3); and
3. Input queries utilize virtual data types (see Section 6.5.1.4), which may differ from the actual adapter's data types. For this reason, the query transformation process includes a step that performs type resolution (see Section 7.2.4). Therefore, each adapter is required to provide forward/reverse data-type mapping information in order for the type resolution to operate.

All of the functionalities of the selected adapters are used in the execution pipeline that is orchestrated by the QEE. The complementing process and existence of the fallback adapter is transparent to all of the adapters. In addition, the existence of each adapter is transparent to the other. The cost of development of adapters highly depends upon the complexity of the underlying data and the management level available to handle such data. Furthermore, the cost of executing similar capabilities differs from adapter to adapter due to various factors, e.g., access methods, type conversion, data entity parsing, tokenization, and the level of optimization available to a data source.

The query engine is equipped with a built-in adapter called *fallback* that is capable of performing all of the query operators. By design, the fallback adapter is prevented from performing actual data access. The reason for this is that the record read/write operations are data organization-dependent; hence, each adapter must provide specific implementation for its own supported data sources. The fallback adapter applies its query operators on the read and loaded records obtained from the target adapters. It is by design an in-memory adapter, but implementations can store the in-transit records in persistent media if necessary, e.g., for big data processing.

9

Summary of Part II

In this part we detailed our solution with regard to its unified query language. The solution has been established on top of a unified query language that was transformed to the native computation models of target data sources with the help of a set of adapters. The generated computation models were compiled into executable units of work and then executed under the supervision of the execution engine. All of the background activities, such as query validation, runtime schema declaration and discovery, data type consolidation, capability negotiation, and query complementing, are performed in a transparent manner. The solution also provides a built-in set of aggregation and non-aggregate functions with support for user-defined functions. The solution suggests visualized query results and plugs them into the language design as first-class citizens. Overall, the proposed solution provides a late schema-binding and agile querying environment characterized by consistent capability exposure for different data organizations.

However, the proposed solution has own limitations: First and foremost, it is not a full-fledged application. It is instead a specification for a data-organization-ignorant, no DBMS, agile, unified, and federated querying system. To demonstrate that such a solution is feasible, we provide in Part III a reference implementation as a proof of concept.

Second, we designed the perspective concept to allow for two-way attribute mappings. While the forward mappings are used for data retrieval, the reverse ones are meant for data persistence. However, we did not detail the data manipulation aspects of the language. The rationale for not doing so was a) to keep the scope of this work under control and b) to stay focused on the concept of an agile querying system. The latter was a result of the motivation of this work, namely to propose a solution for volatile data and ad-hoc querying in research environments that feature high levels of data and tool heterogeneity.

Another limitation is that, while the requirements ask for a declarative language, our design is not completely satisfactory in terms of those requirements. As explained in Section 7.4 (Query Optimization), our aggregate computation technique utilizes a running method that implicitly maintains a state object. Although this state object does not cause any side-effects, it should be carefully implemented to avoid any possible rule violation.

9.1 Realization of the Requirements

In this section, we explore the extent to which the solution satisfies the requirements identified previously. Table 9.1 illustrates the traceability between the features and the requirements; for each feature, F_i, its corresponding row, displays two pieces of information: a) whether F_i should contribute to the realization of any of the requirements R_j, and b), if so, whether the feature does so. These pieces of information are indicated by plain and circled checkmarks, respectively.

Table 9.2 summarizes the overall extent to which the requirements were satisfied. Although neither the requirements nor the features are of equal weight and complexity, a simple average of the requirements' satisfaction rates indicates a satisfaction level of approximately 92%. This means that the features adequately satisfy the requirements and thus the scope of the problem. With the exception of Requirement 15 (Version Aware Querying)[1], all of the requirements experienced a fulfillment level of greater than 80%. This fact indicates that the solution has satisfactorily addressed the problem statement's scope.

Implementation-specific requirements such as Requirement 16 (Tool Integration) and Requirement 17 (IDE-based User Interaction) are realized in Chapter 10 (Implementation). Usability-related requirements, e.g., Requirement 18 (Ease of Use) and Requirement 19 (Usefulness) are discussed in Chapter 11 (System Evaluation).

[1] We intentionally reduced the priority of this requirement, due to time limitations. However, a partial implementation is provided.

	R1: In-Situ Data Querying	R2: Data Organization/Access Extensibility	R3: Querying Heterogeneous Data Sources	R4: Unified Syntax	R5: Unified Semantics	R6: Unified Execution	R7: Unified Result Presentation	R8: Virtual Schema Definition	R9: Easy Data Transformation	R10: Built-in Functions	R11: Function/Operation Extensibility	R12: Data Independency	R13: Polymorphic Resultset Presentation	R14: Resultset Presentation Extensibility	R15: Version Aware Querying
F1: Connection Information Integrated to Language	⊘	⊘	⊘	⊘								⊘			⊘
F2: Version Aware Data Querying	✓	⊘	✓	⊘	⊘	⊘						⊘			⊘
F3: Virtual Schema Definition	⊘	⊘	⊘	⊘	⊘	⊘	⊘	⊘	⊘	⊘	⊘	⊘	⊘	⊘	
F4: Virtual Type System		⊘	⊘	⊘	⊘	⊘	⊘	⊘	⊘	⊘	⊘	⊘			
F5: Uniform Access to Data Items	⊘	✓	✓	⊘	⊘	✓	✓		⊘	⊘	⊘	✓			
F6: Heterogeneous Data Source Querying	⊘		⊘	⊘	⊘	⊘	⊘	⊘				⊘			✓
F7: Query Chaining	⊘	⊘		⊘	⊘	⊘	⊘	⊘		⊘	⊘		⊘		
F8: Polymorphic Resultset Presentation				⊘	⊘	⊘	⊘		⊘				⊘	⊘	
F9: Visual Resultset Presentation				⊘	⊘	⊘	⊘		⊘				⊘	⊘	
F10: Data Processing	⊘	⊘	⊘	⊘	⊘	⊘			⊘	⊘	⊘	⊘			
F11: Query Complementing		⊘	⊘	⊘	⊘	⊘						⊘			
F12: Capability Negotiation		⊘	⊘			⊘						⊘			
F13: Dynamic Query Compilation	⊘	⊘	⊘			⊘		⊘	⊘	⊘	⊘	⊘			✓

Table 9.1.: Traceability matrix demonstrating the extent to which the requirements are fulfilled by the solution's features. A checked cell indicates that the feature (corresponding row) is relevant to the requirement (in the corresponding column) and must satisfy it. A circle-check-marked cell indicates that the feature is relevant and contributes to the fulfillment of the requirement. A blank cell shows that the feature is not related to the corresponding requirement.

	R1: In-Situ Data Querying	R2: Data Organization/Access Extensibility	R3: Querying Heterogeneous Data Sources	R4: Unified Syntax	R5: Unified Semantics	R6: Unified Execution	R7: Unified Result Presentation	R8: Virtual Schema Definition	R9: Easy Data Transformation	R10: Built-in Functions	R11: Function/Operation Extensibility	R12: Data Independency	R13: Polymorphic Resultset Presentation	R14: Resultset Presentation Extensibility	R15: Version Aware Querying
Expected Features	8	10	10	11	10	12	7	5	7	6	6	10	4	3	4
Contributed Features	7	9	8	11	10	11	6	5	7	6	6	9	4	3	2
Overall Satisfaction	88	90	80	100	100	92	86	100	100	100	100	90	100	100	50

Table 9.2.: Requirement satisfaction matrix that expresses to what extent each feature fulfills its related requirements. The *expected* row indicates the number of features that have to contribute to each of the requirements. The *contributed* row is the number of features that contribute to each requirement. And the *overall satisfaction* row is the requirements' satisfaction rate computed as percentage of contributed features versus expected ones.

Part III.

Proof of Concept

In Chapter 1 (Introduction) we suggested a hypothesis, then in Chapter 3 (Problem Statement) scoped its boundaries. Also, we proposed a solution for the stated problem in Part II (Approach and Solution). We dedicate this part to the evaluation of the proposed solution. We first present a proof-of-concept implementation in Chapter 10 and utilize it to illustrate the correctness of the hypothesis. To prove that the hypothesis holds, we conduct a set of evaluations and discuss their results in Chapter 11. The evaluations are designed to measure the language's expressiveness, system performance on heterogeneous data, scalability when applied to large data, and usability.

10 Implementation

In this chapter, we explore the implementation of QUIS's components. QUIS was developed as a proof of concept and a test bed for evaluating the hypothesis [CKRJ17]. The overall architecture of the system was presented in Chapter 5 (Overview of the Solution). Based on the architectural overview depicted in Figure 5.1, we explain the implementation techniques used in QUIS's components. Based on the tasks they are designed to perform, we divide the architectural components into three modules.

The agent module, as described in Section 10.1, receives input queries from the API. It is responsible for parsing, validating, model construction, adapter selection, execution coordination, and result set assembly. The data access module, which we elaborated on in Section 10.2, has two fundamental functions: query transformation and query execution (see Chapters 7 and 8 for definition and features). Both of these functions are handled by adapters. Therefore, this module, in addition to the main functions, provides a plug-in mechanism for managing adapter registration, discovery, and engagement. The fallback adapter is also within the data access module. In Section 10.3, we explain how the client module makes the query language accessible to other tools and systems. This module accepts user queries, passes them to the agent module, and presents users with the query results, as well as diagnostic and instrumentation information. It fulfills its requirements by exposing a set of well-defined APIs that any client can use to submit queries and receive result sets. The clients can interact with end-users or act as brokers that communicate queries and data between systems. In addition, in Section 10.4, we describe the implementation techniques used for tuple materialization, data type consolidation, perspective construction, aggregate computation, and adapter and user-defined function registration.

10.1 Agent Module

The agent module is comprised of the runtime system, a collection of Query Execution Engines (QEEs), and the language parser. These components receive an execution request from a client, compile the solution to the request, and deliver the results. A client request is a set of queries that are submitted alongside related declarations.

Upon starting the agent, the runtime system is activated. Having received a client request, the API requests that the runtime system assign a QEE to the request. This dynamic QEE assignment makes it possible for each client request to run in isolation on its own QEE. In addition,

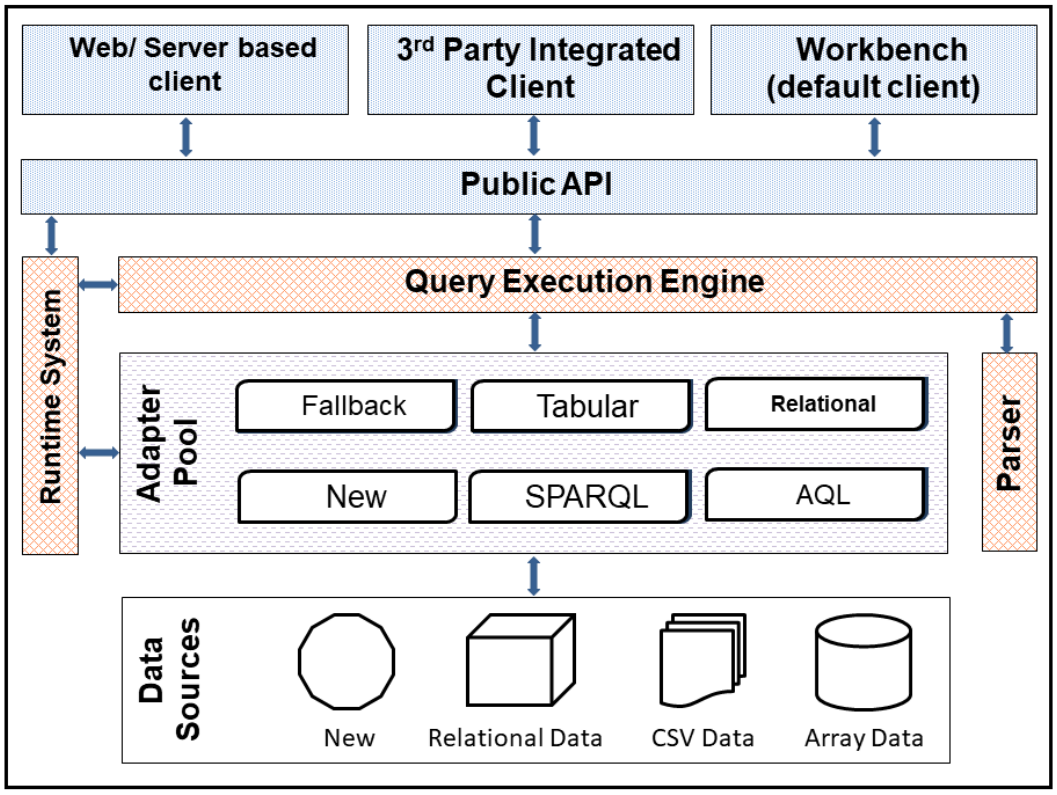

Figure 10.1.: The overall system architecture, showing the interactions between the client, the agent, and the data access modules. The client module consists of the dot-patterned components; the grid-patterned components are the agent module. All of the other components shape the data access module.

requests may be run on different cores, CPUs, or machines to allow for concurrent request processing. QUIS was developed with a built-in QEE, but third parties can develop and register more advanced ones (see Section 10.4.3). QUIS's plug-in mechanism allows the default QEE to be replaced with a user-provided one.

When a QEE is selected, the runtime system activates the engine and forwards the client request to it. The QEE parses, validates, and builds an ASG for the input queries. It then selects the adapters that will be used to transform and execute each and every query. In the following sections, we explain the elements and the features of the agent module.

10.1.1. Parsing

Language recognition in QUIS follows the general parsing workflow: It starts with lexical analysis, which is used to tokenize the input; continues with syntactical parsing , which recognizes the input token stream and builds parse trees; and ends with semantic analysis in order to perform type checking, language binding, and dependency detection.

We use ANTLR 4 [Par13] as the language recognizer. It generates both the lexical and syntactical analyzers, as well as the extension points that are used to plug a semantic analyzer into the flow. ANTLR 4's parser generator embeds two types of event subscription hooks (namely listener and visitor) into the generated parser. A hook is a proxy method that, when triggered, can call a designated function to accomplish the actual task. These hooks can be easily used to integrate the parser into a larger ambient application.

ANTLR generates listener hooks for entry to and exit from each rule, as well as rule alternates. QUIS, as the ambient application, subscribes to the designated entry/exit hooks to perform its logic. These hooked functions are then automatically activated during parsing. We have hooks for input validation, DST and ASG construction, declaration checking, query-chain checking, perspective building, and completing of missing elements.

In order to create the final ASG and to perform cross-statement validations, we needed to traverse the ASG multiple times. For this purpose, we utilized ANTLR's visitation hooks. We developed additional visitor methods for traversing the parse tree and triggering the visitation points. Our semantic analysis relies heavily on this selective visitation to build DSTs, infer effective perspectives, and determine inter-query dependencies.

10.1.2. Dynamic Compilation

As mentioned in Section 8.1.3 (Query Compilation), QUIS compiles queries into executable jobs. This is done for two reasons: First, some data sources utilize imperative computational models; for example, CSV files are processed by running operations on them, MapReduce requires jobs in terms of Java classes and libraries, and web services require their web APIs to

be called via HTTP requests. Additionally, complementing and compositional queries are executed by the fallback adapter, which also uses an imperative computation model. Second, due to significant differences in the syntaxes and connection methods of declarative data sources, we decided to wrap any transformed query into an executable package in order to conceal potential complexities from both the default and other QEEs. This technique provides an additional side benefit: The jobs are standalone and fully self-contained packages that can be shipped to remote systems for execution, archived for later use, or preserved with research results for reproduction purposes. The jobs' source codes are generated in Java.

The dynamic code compilation (also referred to as just-in-time code compilation) component consists of two services: code generation and code compilation. The code generation service, in association with designated adapters, generates partial Java source codes for each query feature. It also generates wrapper classes for non-imperative transformations, e.g., perspectives. The service has a set of predefined templates for each query feature, which adapters can (and are encouraged to) override. We use the Rythm Templating Engine[1] as our templating engine, due to its simplicity and performance. The templates are populated and instantiated according to the query features' requirements, as modeled in their respective DST nodes. The instantiated templates are then assembled to generate a set of valid Java classes that are equivalent to the incoming DST. We generate two classes for each query, with an entity class that represents the query's effective perspective and a driver class that performs the requested query operations on the data source and then populates and return a bag of entity instances. In the case of grouping, we may generate an additional intermediate entity class. However, if a perspective is reused in multiple queries, we also reuse its corresponding entity class. This service assists in creating the best matching and ensuring the most concrete data access and processing operations possible. It not only reduces the need to have a pre-built database to query data but also improves querying performance.

When code generation has been performed for all of the DSTs, we compile the jobs' source codes using Java's JDK compiler utility. We place all of the source codes into a single compilation unit and compile them in one pass on-the-fly. Having a single compilation unit reduces the compilation time and avoids errors caused by dependencies between the generated classes (which arise from data dependencies between the queries).

It is possible to cache the generated source and/or the compiled jobs and attach them to the queries for later use. Currently, we cache the jobs for a single live session; hence, re-running the queries will bypass the code generation and compilation phases. The cashing techniques can be extended to persist the compiled jobs alongside with the process script file for long-term archival, backward compatibility, remote execution, and reproducibility.

[1] `http://rythmengine.org/`

10.2 Data Access Module

The main role of the data access module is to provide the base classes and interfaces that adapters must implement in order to be able to interact with the QEE. The following are the functions each adapter must implement:

1. **Metadata:** An *expose* function reports the module's metadata. It features a tree data structure that indicates which features of the query language are supported and displays their estimated costs of execution. The tree also contains information concerning the supported aggregate and non-aggregate functions. The actual schema of a queried data container is also retrieved and reported;

2. **I/O:** This function provides the actual implementation of the tuple scan and insert operators. The scan operator reads one record from a designated data source, tokenizes it into values, and converts the values into appropriate data types according to the query requirements. The output is a list of objects, with each representing a value of a field of the data source record. The insert operator performs the reverse operation: It serializes the input list of objects to the underlying data source in its native format;

3. **Querying:** For each exposed query feature, the adapter implements its corresponding implementation in term of a parametric template. The template is later instantiated during the transformation process. However, the adapters can provide alternative implementations;

4. **Transformation:** A *transform* function receives a DST, applies its templates to each query feature that exists in that DST, and instantiates assembles the templates. The assembly process yields a set of Java source classes. The entity class forms tuples (see Section 10.4.1), while the driver class executes the query features; and

5. **Execution:** An *execute* function receives the compiled driver class, the job, triggers its entry point, and collects the results returned. It sends the result set to the QEE to be used in subsequent queries or presented to the requesting client. The execution follows the pipeline of operations assembled by the transformer. The flow begins with scanning and materializing the tuples, applying perspective's attributes' mappings, and then calling query operators in the order determined by the optimizer.

Adapters that do not implement the full language's query features are called *weak adapters*. Weak adapters need to be supported by the query complementing technique (see Section 7.3) to be able to fulfill all of the input query requirements. In addition, they usually demonstrate slower performance and potentially retrieve more data. We utilize a fallback adapter to take over the features that a weak adapter does not support. Such a fallback adapter must be complete in order to allow the engine to support all of the query features and must be sound in order to return correct results when complementing for the features that a weak adapter lacks. We have implemented such a fallback adapter and plugged it in to the query complementing; however, it could be replaced by third-party fallback adapters. The fallback adapter does not have I/O operators but implements all of other above-mentioned required functionality. It uses a templating

technique that tailors the complemented query features to operate on memory-loaded datasets. Its *expose* function intentionally reports a high execution cost, resulting in the adapter selection algorithm (see Section 8.1.2) preferring to choose from among the registered adapters.

In addition to its main role, the data access module manages adapters' life cycles. For example, it provides facilities for registering adapters, adding new function packages to existing adapters, and updating the system catalog. The adapter selection algorithm relies on the system catalog and the metadata provided through each module's *expose* function.

Each adapter is compiled and bundled in a single Java jar package. The registration procedure simply copies the package to a designated directory and updates the catalog. Each adapter can be accompanied with a set of satellite function packages. Function packages override/customize the transformation and/or execution of the aggregate and non-aggregate functions provided by the language. Overrides can be associated with a specific dialect or can be applied to all of the supported dialects of the designated adapter.

Queries usually retrieve data from persistent containers and deliver it to variable containers. Variable containers are used for result set presentation, query chaining, or consumption by clients. However, queries may retrieve data from and write data to persistent containers simultaneously. For example, a query could retrieve data from an RDBMS table, apply filtering and projection on it, and write the result set to a CSV file. In this and in similar cases where the input and output containers of a query are both persistent, we bypass the memory by short-circuiting the input to the output container. Furthermore, the input and output containers may be bound to data sources that are managed by different adapters; for this purpose, we utilize Java 8's steaming feature. We create a reading stream on the input container and plug it into the output. Meanwhile, we perform all of the necessary operations, e.g., projection or selection, on the tuples while they are in transit. The writing operator may apply another set of tuple formation according to the query's perspective.

This technique not only eliminates the need to load the input tuples into memory but also increases the degree of parallelism in multi-core environments. This technique particularly demonstrates its value in low-memory systems, big data querying, or workflow management systems. It is, however, not applicable in the presence of sorting or aggregation. We detect these cases and transparently fallback to the normal execution path.

10.3 Client Module

In order to access `QUIS`'s functionality from any client, we have developed an API that exposes a set of service points. These service points can be consumed by a client to interact with `QUIS`. In a normal scenario, a client submits a set of queries for execution, obtains the results, and presents the result sets. A client can interact with end users or other applications. It can be a desktop workbench such as QUIS-Workbench; an external tool, such as R-QUIS, that enables R users to interact with `QUIS`; or a server application that opens `QUIS`'s APIs to remote users via, e.g., a web interface.

We have developed three open-source clients[2]: A GUI-based workbench (QUIS-WRC), a command-line interface, (QUIS-CLI) and an R package (R-QUIS). All of these clients use the provided API to interact with the QEE. In the following section, we explain the technical details of the client module's components.

10.3.1. Application Programming Interface (API)

The API is the access point to the system functions. It is a set of Java classes and interfaces that provides three type of functions: information, execution, and presentation.

1. Information functions report on the system version, executed queries diagnostics and logs, and performance metrics such as execution time and result set size;

2. Execution functions accept queries in various formats, e.g., files, strings, and command-line arguments and execute them; and

3. Presentation functions are designed to transform query results into formats that are consumable/requested by clients. For example, R-QUIS requires query results in R's data frame format.

Our default QEE is bound to one active client request at a time and keeps track of its state. We do not share agents between various clients' requests. This allows us to support isolated agent instances for each request in the client's process space. Using this isolation level, clients are able to submit multiple requests in parallel and interact with their users utilizing, e.g., MDI[3] or TDI[4] UIs.

10.3.2. QUIS-Workbench

We have two default workbenches[5], QUIS-WRC and QUIS-CLI. Both of these workbenches utilize the APIs in a similar manner. However, they differ in the way they interact with users, as well as in degree of parallel execution.

QUIS-WRC is a rich-client workbench that provides a multi-tab graphical environment. Users can load or type in queries in different tabs and request their execution. In QUIS-WRC, queries are organized in files and files in projects; however, the workbench only contains one active project at a time. Projects are loaded to and managed in the *project explorer*, as shown in Figure 10.2 (pane **A**). The *query editor* (pane **B**) is a multi-tab area that allows multiple query editors to be opened simultaneously and runs them in parallel. Each editor tab provides code-editing features such as syntax highlighting, undo, redo, and copy and paste, among others.

[2]Source: https://github.com/javadch
[3]http://en.wikipedia.org/wiki/Multiple_document_interface
[4]http://en.wikipedia.org/wiki/Tabbed_document_interface
[5]http://fusion.cs.uni-jena.de/javad/quis/

Figure 10.2.: The three main panes of the rich-client workbench: A: Project explorer. B: Query editor. C: Results viewer. The overlay chart depicts the visual representation of a query result. However, it also appears on its own tab in the results viewer pane.

Furthermore, each query editor has its own *result viewer* (pane **C**), in which each query result is shown individually in a tab. A summary of execution results is also presented. Query results can be visualized in the form of single or multi-series bar, line, scatter, or pie charts. An exemplary line chart that illustrates mean daily temperature over a time interval is shown as an overlay in Figure 10.2.

QUIS-CLI is a terminal-based client for QUIS that accepts queries in files and also persists the results in files. It can be integrated into the host operating systems' shell scripting, as well as other tools and workflow systems. It is possible to trigger QUIS-CLI with a customized JVM configuration, e.g., to allocate heap memory in advance for operating on large datasets. For example, `java -Xmx128g -jar workbench.cli.jar demo.xqt` directs the JVM to allocate as much as 128GB of memory for QUIS-CLI to run the `demo.xqt` query file. When set, QUIS-CLI can report execution information to the terminal and/or write it to log files.

10.3.3. R-QUIS Package

R-QUIS is an R package that offers the full functionality of QUIS to R users[6]. Users can write queries inline with R scripts or in separate *.xqt* files. The package exposes a set of functions that allow R users to formulate their data access requirements in QUIS's query language and

[6] http://fusion.cs.uni-jena.de/javad/quis/latest/rquis/

obtain the result sets as R data frames. Additional functions for obtaining result set schemas and accessing the results of previously executed queries are also available. Listing 10.1 showcases the package's functionality. The actual queries are not shown, in the interests of brevity.

Listing 10.1 Computing daily average temperature using the RQUIS package from winthin R.

```
1 library(RQUIS)
2 library(ggplot2)
3 engine <- quis.getEngine()
4 file <- ``/quis/r/examples/dailyTemp.xqt''
5 quis.loadProcess(engine, file)
6 quis.runProcess(engine)
7 data <- quis.getVariable(engine, ``meanDailyTemp'')
8 schema <- quis.getVariableSchema(engine, ``meanDailyTemp'')
9 ggplot(data, aes(dayindex, meantemp)) geom_line() xlab(``'')
      ylab(``Mean Temperature C°'') ggtitle(``2014 Average
     Daily Temperature at SFO'')
```

The script queries a CSV dataset that contains hourly meteorological data and computes a daily mean. The R script first obtains an API handle called engine (`line 3`). It then loads the query file into the engine (`line 5`) and runs it. The *getVariable*(`engine`, `variable`) function populates the data frame, `data`, that is used by the plot function `ggplot` to draw the requested chart, as shown in Figure 10.3.

Each variable points to a result set that is associated with a perspective. R users can obtain the perspective of any result set by calling the *getVariableSchema*(`engine`, `variable`) function, which returns a data frame that presents the basic properties of the perspective's attributes, e.g., `name`, `datatype`, and `size`.

R-QUIS mitigates the need to use different packages for querying heterogeneous data and provides users with a unified data querying syntax. More importantly, it does not require the data to be loaded before processing. Instead, it performs all of the query operators, such as filtering, grouping, and limiting, in-situ and only loads the result set. This makes R-QUIS up to 30% faster than R while also maintaining a lower memory footprint. It is also possible to join data between various data sources.

In addition, we have built a Docker image[7] with all of the software and settings required for easy deployment. When instantiated on a Docker machine, it runs a web-based RStudio server that has R-QUIS loaded and operational. The image can be pulled by issuing a `docker pull javadch/rquis` command.

[7] `https://hub.docker.com/r/javadch/rquis/`

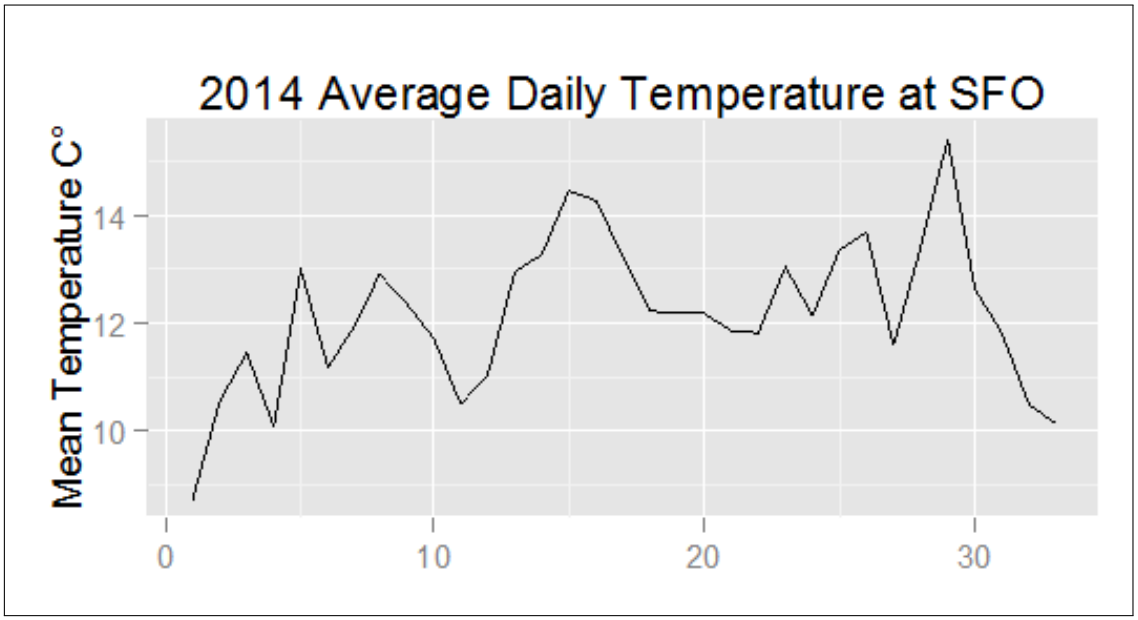

Figure 10.3.: The average daily temperature at SFO airport in 2014.

10.4 Special Techniques

In this section, we explain the techniques that we have used to implement interesting features, e.g., tuple materialization, data type consolidation, perspective construction, aggregate computation, and adapter and user-defined function registration. These features are not explained as elements of the three main components; however, they illustrate how in-situ querying and on-the-fly job compilation affect the core elements of a heterogeneous database or query system.

10.4.1. Tuple Materialization

Listing 10.2 illustrates a query that uses a perspective `resultSchema` for its projection clause. The perspective consists of a set of attributes each of which is mapped to the relevant fields of the underlying container via mapping expressions. The actual data source behind the container is determined by the `ds` connection (`line 5`) and the `occ` binding (`line 6`). The query's virtual schema is known during the transformation phase; therefore the projection attributes and the mapping expressions are accessible to the transformation functions. In order to apply declared mappings, we extract the schema of the underlying container, the physical schema. Using both the virtual and physical schemas we generate a Java entity class that implements the attributes of the virtual schema as transformation functions of the fields of the physical schema. The entity class has a set of `populate` methods that accept an input record of the physical schema's format. These methods populate the entity's attributes by applying the virtual schema's respective

transformations on the incoming records. Thereafter, they release the physical record to preserve memory. This technique is referred to as tuple materialization.

Materialization performance is influenced by sequential or direct access to the fields and the need for data type conversion. A data source may only provide sequential access to the fields; for instance, a line in a CSV file represents a single record, meaning that its fields cannot be accessed randomly. In these cases, we materialize the attributes in such an order that minimizes the number of passes over the input record. Sequential access may impose prerequisites, such as parsing and tokenization. Currently, we tokenize the entire line, even when not all of the fields are required by the query. However, there exist approaches that can detect the shortest continuous sub-string that contains all the required fields. In addition, maintaining position indexes/maps can improve overall performance, but adds additional complexity when building the and updating the index.

Listing 10.2 A sample QUIS query for studying tuple materialization. The query obtains 100 rows from the data source. It uses the projection defined by the `resultSchema` perspective.

```
1  PERSPECTIVE resultSchema{
2    ATTRIBUTE Name:String          MapTo = scientificname,
3    ATTRIBUTE observedIn:Integer   MapTo = str.toInteger(year)
       ,
4  }
5  CONNECTION dss ADAPTER=CSV SOURCE_URI='data/' PARAMETERS=
       fileExtension:csv, firstRowIsHeader:true
6  BIND occ CONNECTION=dss SCOPE=Occurrences VERSION=Latest
7
8  SELECT
9  USING PERSPECTIVE resultSchema
10 FROM occ.0 INTO dataPage
11 WHERE(observedIn > 2005)
12 LIMIT TAKE 100
```

In QUIS, tuples are materialized in multiple steps, according to the generated query plan. For example, during the first step, only the attributes needed to evaluate the selection's predicate are materialized. If a join operation is present, then the attributes used as join key are populated. Finally, if selection and/or join operations pass, the attributes of the projection clause are materialized. Therefore, if the input record is rejected at any step, the system saves the computation and space that would be required to perform the redundant next steps. At each step, the system avoids materializing the already populated attributes. This multi-staged entity population reduces the overall query response time in proportion to cardinality (see Section 7.4 (Query Optimization)).

If the type system of the input record does not match the virtual schema's type system or when the input record does not have any associated type information, a type conversion must be performed. These actions are provisioned during the transformation and are performed during the

execution phases. Each adapter has enough information about its underlying data sources to decide on the appropriate conversion. Listing 10.3 shows the Java entity generated according to the perspective defined in Listing 10.2. It owns two public fields, `name` and `observedin`, which represent the attributes `Name` and `ObservedIn`, respectively.

The tuple is materialized in two phases: The first phase is handled in the class' constructor, which receives a tokenized input record and uses it to populate `observedin`. As shown in `line 9`, a token representing the `year` field is obtained, converted to an integer, and assigned to the `observedin` attribute. The row is also kept for potential further population phases. The second population phase is performed in the `populate()` method. However, it is the driver class (see Listing 10.4) that determines whether this phase should be executed. Its decision is based on the validity of previous population phases and satisfaction of the selection's predicate.

Listing 10.3 The dynamically generated entity representing the perspective defined in Listing 10.2.

```
1  public class Stmt_1_Entity {
2      public String name;
3      public int observedin;
4      public boolean isValid = true;
5      private String[] row;
6
7      public Stmt_1_Entity (String[] row){
8          try {
9              observedin = (int)((xqt.adapters.builtin.functions.String.toInteger(String.valueOf(
                   row[1]))));
10         } catch (Exception ex) {
11             isValid = false;
12         }
13         if(isValid){
14             this.row = row;
15         }
16     }
17
18     public Stmt_1_Entity populate(){
19         try {
20             name = (String.valueOf(row[0]));
21         } catch (Exception ex) {
22             LoggerHelper.logDebug("Object Stmt_1_Entity failed to
                   populate. Reason:" + ex.getMessage());
23             isValid = false;
24         }
25         row = null;
26         return this;
27     }
28 }
```

Listing 10.4 is the Java equivalent of the generated query plan. It reads the input data, in this case a CSV file, record by record, bypassing the header and commented lines. Starting with the tokenization process, each record is passed through a pipeline of operations.

Listing 10.4 The dynamically generated query plan for the SELECT statement, as declared in Listing 10.2.

```
1  public class Stmt_1_Reader implements DataReader<Stmt_1_Entity, Object, Object> {
2      BufferedReader reader;
3      BufferedWriter writer;
4      String columnDelimiter = ",";
5      String quoteMarker = "\"";
6      String typeDelimiter = ":";
7      String unitDelimiter = "::";
8      String commentIndicator = "#";
9      String missingValue = "NA";
10     String source = "";
11     String target = "";
12     boolean bypassFirstRow = false;
13     boolean trimTokens = true;
14     LineParser lineParser = new DefaultLineParser();
15
16     public List<Stmt_1_Entity> read(List<Object> source1, List<Object> source2) throws
           FileNotFoundException, IOException {
17         lineParser.setQuoteMarker(quoteMarker);
18         lineParser.setDilimiter(columnDelimiter);
19         lineParser.setTrimTokens(trimTokens);
20         reader = new BufferedReader(new FileReader(new File(source)));
21         if(this.bypassFirstRow){
22             reader.readLine();
23         }
24         List<Stmt_1_Entity> result =
25         reader.lines()
26             .filter(p -> !p.trim().startsWith(commentIndicator))
27             .map(p -> lineParser.split(p))
28             .map(p -> new Stmt_1_Entity(p))
29             .filter(p -> (p.isValid == true) && (((p.observedin) > (2005))))
30             .limit(100)
31             .map(p -> p.populate())
32             .filter(p -> p.isValid)
33             .collect(Collectors.toList());
34         return result;
35     }
36 }
```

The pipeline represents the generated and optimized query plan. The driver calls for the first population phase in `line 28` and then checks the entity's validity status, as well as the query's where clause, in `line 29`. Because the query declares a `LIMIT` clause, the optimizer defers the

second population phase until the limit is applied (`line 30`). This dramatically reduces the time required to populate the entities that do not appear in the result set. In other words, it minimally populates the first 100 entities that satisfy the predicate and then fully populates only those.

Because each entity class is generated according to its particular concrete schema, we do not need to utilize general purpose materialization or scanning operators. Instead, we dynamically build the exact operators required to retrieve the records of interest and produce the necessary entities. As mentioned previously, we even insert applicable optimization rules into the materialization logic. Therefore, both classes are highly optimized for the purpose.

We have additionally fine-tuned the Java classes to eliminate unnecessary function calls. For instance, we generate the entity attributes as Java public fields and omit the getter and setter access methods around the fields. This saves a large number of function calls and also reduces heap usage. In a query that requests 10 attributes on data container with a million records, this technique saves at least 10 million `set` and `get` operations per query execution. This technique is similar to Java's getter/setter in-lining. We implemented it internally, as neither the Java compilers nor the JVMs guarantee that it will be performed in a deterministic manner.

10.4.2. Aggregate Computation

Conventional RDBMSs emit the aggregate calculation for each group of records only when the execution engine processes the last record in the table [SC05]. Streaming applications continue indefinitely, and there is no notion of "end of table"; hence, they cannot use this technique. In streaming systems, aggregates are computed over a window of stream or accumulated as stream continues. As presented in Section 7.4 (Query Optimization), we use a combination of both techniques to calculate the aggregates.

We execute each query as a series of operations that are chained together in a pipeline. The pipeline builds a stream that has as its input the records of the queried data source and as its output the solution to the query. We build this stream using Java 8's streaming features and apply the query operators using the Java's lambdas. An exemplary pipeline is provided in Listing 10.4.

We transform each aggregate function to a Java class with a state object and a lambda function that updates the state. The state object holds all of the information required to yield, the most recent value of the aggregate upon request. The update logic is encapsulated in the `move` function. We maintain one aggregate object per group of records. The groups are built upon the unique combination of the grouping keys declared by the input query. Each time the aggregate is called, its `move` method receives the new value from the stream and updates the state. At the end of the stream, the aggregate value is ready to be consumed.

This technique allows the QEE to reduce memory usage to an amount that is proportional to the size of the state object and to ensure that it remains independent from the size of the queried data. Furthermore, the aggregates are non-blocking and do not need the stream to be finished to yield their results; the results are available after each and every update.

10.4.3. Plug-ins

A plug-in is a component with a specified set of related functionalities that is injected into a host application at runtime. It usually implements an interface. If more than one implementation for a specific interface is available, the injection mechanism can select one based on parameters, configuration files, or other metadata. The implementation component can be a single class or a complete JAR package. The plug-in management has a simple design that follows a basic Inversion of Control (IoC) design pattern. `QUIS` utilizes plug-ins in the following scenarios:

1. **Query Execution Engines:** It is possible to replace the default QEE or have multiple query engines run side by side for various workloads. The current implementation supports configuration-based manual engine selection. The selected query engine is injected into the runtime system at startup and responds to input queries according to its isolation level, as described in Section 10.1;

2. **Adapters:** Each adapter is a realized as a set of Java classes bundled in a JAR. The entry point of the adapter implements the `DataAdapter` interface, which requires the adapter to implement the methods detailed in Section 10.2. When the QEE selects an adapter to run a query, it asks the plug-in manager to load the adapter. The plug-in manager queries the adapter catalog file to locate the requested adapter's JAR file and load it, if found. It then uses Java reflection to find the entry point and instantiates the plug-in using IoC techniques. Finally, it returns the instantiated object to the engine; and

3. **Function Packages:** All of the aggregate and non-aggregate functions defined by the language have their own default implementations. Adapter developers, however, can customize the default implementations according to the capabilities of their underlying data sources. Multi-dialect adapters may provide different implementations of each of their supported dialects. Custom implementations are bundled into a JAR file and shipped alongside the adapter package. The bundles are registered within the adapters. The query transformation process utilizes the adapter-/dialect-specific implementation of the functions whenever necessary.

In addition to the above-mentioned implementation notes, we also assumed the following constraints: 1) The built-in functions skip NULL input values; 2) the input data submitted to the statements and functions, as well as the result sets are immutable; 3) logical expressions use effective Boolean value (early-out) evaluation pattern; 4) multiple aggregate calls per attribute in a single expression are valid and possible–however, aggregate functions can not be nested in each other or in other functions; and 5) aggregate functions accept only one parameter, which is either a reference to an attribute of the incoming data record or a non-aggregate function call.

Expressions containing aggregate functions may have non-aggregate sub-expressions; however, the sub-expressions can not use the attributes of the incoming records. For example, $a : count(x) + math.sqrt(100)$ is a valid expression while $b : count(x) + math.log(y)$ is not. The reason for this is that the aggregation part needs the entire input stream, which makes it

impossible to pass the y to the $math.log()$ function. In these scenarios, the non-aggregate part is applied to the result of the aggregation at the end of computation.

If the effective perspective of a statement contains aggregate and non-aggregate attributes, the non-aggregate ones build an implicit GROUP BY clause. The GROUP BY will include all of the attributes that have not participated in the aggregate functions. It is worth noting that WHERE clauses are executed before GROUP BY; hence, they do not have access to the aggregated attributes.

11 System Evaluation

In this chapter, we evaluate the overall performance of the system from a variety of different perspectives. We first explain, in Section 11.1, the evaluation methodology, including the objectives, the experiments designed and conducted, the testing environment, and the data. Thereafter, we elaborate on the individual evaluations.

In Section 11.2, we measure the time-on-task of a group of test subjects performing an assigned task on `QUIS` and a tool set of their choice. We observe the time-to-first-query (see Definition 11.1). Thereafter, we demonstrate that the improvement in time-to-first-query does not come at the cost of reduced performance of query execution. We explain `QUIS`'s performance on heterogeneous data in Section 11.3 and evaluate its performance at scale in Section 11.4.

In order to evaluate the system from the users' perspective, we report on the results of a user study that we conducted in Section 11.5. The study required the test subjects to perform a specific task on `QUIS` and a baseline system and then provide their rankings of both systems using a questionnaire. The study measures different aspects of system usability indicators in a realistic daily work scenario. We define and observe six indicators: time-on-task, execution time, code complexity, ease of use, usefulness, and satisfaction. While the first three are accurately measurable, the later three take the form of user opinions, which we analyze in order to derive insights from. Because this was a detailed evaluation that generated a large set of supporting documents, the most important materials are provided in Appendix C. Finally, in Section 11.6, we compare the features of our language to that of SQL and the other languages studied (see Section 6.4) in order to illustrate the language's expressiveness.

Definition 11.1: Time-to-first-query *is the total up-front preparation time required before the first query can be issued. It includes cleansing, transforming, and loading data to the designated query systems.*

11.1 Evaluation Methodology

We evaluate different aspects of the system. For each, we design and conduct one or more experiments. Each experiment may have its own method, input data, or tool set. However, in this section, we introduce the commonalities.

11.1.1. Evaluation Data

Species Medicinal Value (SMV) Dataset: The SMV dataset is the equivalent of the Ecological Niche Modeling use-case (Section 1.2.1 (Ecological Niche Modeling Use-Case)). It is stored in a number of different sources. This dataset contains all of the data attributes and records concerning species, medicinal uses, and species location observation data. The SMV's dataset schema consists of a species table with a one-to-many relationship to a locations table and a many-to-many relationship to a symptoms table via a medicinalUses table. Each species can be observed at many locations. Symptoms can be treated by many species, and each species can treat multiple symptoms. The medicinal uses refer to the symptoms that the species can treat.

Fungi Occurrences (FNO) Dataset: Fungi Occurrences is a dataset of observations of fungi occurrences recorded between 1600 and 2015. It consists of 42 data attributes (including scientific name, taxonomic classification, observation time, location, and metadata concerning the recording time, individuals involved, and rights). The dataset is a composite of many other datasets created by north European and American institutions merged and maintained by the GBIF [GBI15a]. We chose two versions of the dataset (i) FNO-SML, which consists of 10M records in a 5GB CSV file [GBI15b] and (ii) FNO-BIG, which consists of 651 million records in a single 280GB CSV file [GBI16].

11.1.2. Tools

Table 11.1 shows a summary of the tools we used in the different experiments. We chose an RDBMS and the R[1] system as our comparison baselines in order to demonstrate that QUIS performance is comparable to well-known systems. RDBMSs are used in almost every domain and have state-of-the-art performance, while R is widely used in scientific data analysis activities.

Tool	Version	Purpose of Use
PostgreSQL	9.4.4	To execute SQL queries
RStudio Desktop	0.99.484	To interact with R via GUI
R [R C13]	3.2.2	To execute R scripts
DBI [CEN+14]	0.3.1	To interact with RDBMSs from R
RPostgreSQL [CEN+12]	0.4	DBI implementation for PostgreSQL
QUIS-Workbench [Cha15]	0.5.0	To execute QUIS queries in via GUI
R-QUIS [CYV]	0.5.0	To execute QUIS queries from R

Table 11.1.: The tools used in QUIS evaluation scenarios.

[1] http://www.r-project.org/

11.1.3. Test Machines

We used the following hardware and software to run the experiments:
SMALL-MACHINE: Intel i7-2620M/ 2.70GHz/ 2 cores/ 64 bits, Storage: 250GB SSD Seq. R/W rate: 550/520 MB/s, Memory: 16GB, Java: 1.8, JVM Xmx: 8GB.
BIG-MACHINE: 16 cores/ 64 bits, Storage: 1TB NAS, Memory: 128GB, Java: 1.8, JVM Xmx: 128GB.

We performed the experiments on both machines. In order to isolate the results from the out-of-control factors such as network latency, bandwidth, and stability, we mainly reported the results obtained using the SMALL-MACHINE configuration. However, the results obtained from BIG-MACHINE confirm the system behavior demonstrated on SMALL-MACHINE. In all of the experiments that involve time measurements, we repeat each query five times, with an intervening buffer flush, and report the average.

11.2 Measuring Time-to-first-query

The amount of incoming data is massive, and loading and preparing it for the first use may take a remarkable length of time, while the validity or relevance of the data is unknown. In such an uncertain situation, a user may prefer to determine whether the currently available data is of any interest, and it should be necessary to determine this in a manner that is fairly rapid when compared to the usual prepare, load, and tune procedure.

To better understand the costs associated with data preparation, we worked with a group of subjects with various skills, i.e., data scientists from the BExIS project, the authors of the species niche modeling use-case, and master students enrolled in a research data management course. The subjects were from different backgrounds, including bio-informatics, geo-informatics, computer science, biology, and ecology. Each subject was provided with sufficient training to be able to perform similar tasks comfortably before being assigned the task. We categorized the subjects into three groups: (SG1), (SG2), and (SG3). We asked the subjects, independently of the other groups, to create and load a full relational dataset from the SMV dataset (Section 11.1.1). They were free to use the tool sets of their choice. We executed a set of test queries against the resulting databases to determine the correctness of the setup databases.

For each group, we separately report the time required to complete each of the three preparatory tasks: (T1) extracting, cleaning, and transforming the source data, (T2) creating the target database schema, and (T3) importing the source data (from all of the data sources) into the target database (the time required to prepare the subjects and the computing environment is not included in the measurements reported below.). Task T1 was performed by the subjects with the assistance of tools, T2 was almost entirely completed by the subjects, and T3 was performed by an automated script specified by the subjects.

QUIS, in comparison, requires none of these steps. However, it does require the establishment of connections to these data sources and the establishment of corresponding bindings. We recorded the time taken by each of our subjects to complete the three tasks as well as the QUIS ones and report their averages per group in Table 11.2.

Task	SG1	SG2	SG3	Average
T1	45	56	37	46
T2	10	14	12	12
T3	1	1	1	1
Total ETL	56	71	50	59
Connection	1.5	1.5	1	1.5
Binding	1	1	1	1
Total QUIS	2.5	2.5	2	2.5

Table 11.2.: Time-to-first-query observation result (minutes). SG is the subject group and T is the task. Values indicate the average time spent by a subject group on a task. Each group had two members. The measurements are rounded up to half a minute.

The results clearly indicate that the time-to-first query poses a serious burden on scientists. SMV is a relatively small dataset (it has less than one million rows) with only 3 data sources. Even in this scenario, QUIS reduced the time-to-first-query by a factor of 30; with larger, more heterogeneous data sets, we expect QUIS to excel by an even greater margin. However, it is worth mentioning that the cleaning task that the subjects completed during the data preparation phase assisted them later while querying and consuming data. QUIS, on the other hand, while it allows for fast startup, requires data workers to perform data cleaning during later steps, i.e., concurrently with the process of querying data. This is a part of QUIS's design paradigm as an agile querying system. The rationale behind this is the exploratory nature of the task at hand and the dynamic structure of the data. The assumption is that the data worker will search for the appropriate data in a large and changing dataset.

11.3 Performance on Heterogeneous Data

In Section 11.2, we observed that the in situ nature of QUIS's design provides great savings in terms of time-to-first-query. Next, we show that the reduction in time-to-first-query does not come at the cost of an unreasonable increase in query execution time (see Definition 11.2). The crucial issue that we need to study is the performance penalty paid for query execution by QUIS, given that there is no time spent on data preparation. Thus, the main goals of our performance experiments are as follows: (i) to determine whether QUIS has a reasonable execution time and (ii) to observe how it scales when querying large datasets (Section 11.4).

Definition 11.2: Query execution time *Refers to the user-observed (wall-clock) elapsed time from when the user submits a query to the moment that the client application presents the last record of the query result to the user.*

The goal of this experiment is to study the system's performance when querying heterogeneous datasets. We measure the cumulative time required to run a sequence of queries on QUIS and compare it with that of a set of chosen competitors. We designed experiment Exp. 1 to run the ecological niche modeling use-case (Section 1.2.1) on the SMV dataset. It performs the following steps:

1. For each species that has been observed in more than 30 locations and has at least one medicinal use, select the species' name and the number of symptoms it treats; and

2. Select S% of the resulting records, where S is the selectivity parameter. Use the longitude field to enforce the desired selectivity.

The experiment consisted of a set of scenarios, Exp. 1.0–Exp. 1.3. We designed each scenario to execute the task on a different organization of the dataset. The data organizations are shown in Table 11.3. The scenarios allowed us to study the impact of both individual data sources and heterogeneity on QUIS's performance. We use the heterogeneity index (HI) to indicate the diversity of the data sources involved in a query. More specifically, HI is the number of distinct data sources involved in a given query.

Scenario	HI	Data Sources
Baseline	1	DBMS
Exp. 1.0	1	CSV
Exp. 1.1	2	CSV, DBMS
Exp. 1.2	2	CSV, Excel
Exp. 1.3	3	CSV, Excel, DBMS

Table 11.3.: Data source settings for the performance on heterogeneous data experiment.

In addition, w designed a baseline scenario for comparison purpose. To run the baseline scenario, we transformed and imported the experiment's dataset into an appropriate relational model in the chosen RDBMS (see Section 11.1.2). We also created primary and foreign keys, as well as indexes. The relational schema used in this baseline scenario is shown in Figure 11.1.

We executed the baseline query with 0%, 10%, 30%, 50%, 70%, 90%, and 100% selectivity, fetched the resulting records, and measured the execution time. The zero selectivity level was used to capture the constant overhead of the query, e.g., for warm up, index scans, logging, and security checks; the 100% selectivity level was used to exclude the performance gain achieved through indexing. We wrote the QUIS equivalent of the baseline query, ran it on the scenarios, and measured the query execution time. Figure 11.2 shows the query execution times of the baseline and all of the scenarios in Exp. 1.

The Baseline chart depicts the linear response time of the baseline for different selectivities. Exp. 1.0's scenario shows a higher initial query time but outperforms the baseline on selectivities above 20%. Exp. 1.2 starts at a higher offset, having a similar trendline as that of Exp. 1.0, and passes the baseline at selectivities of 55% and higher. The main factor that causes a longer

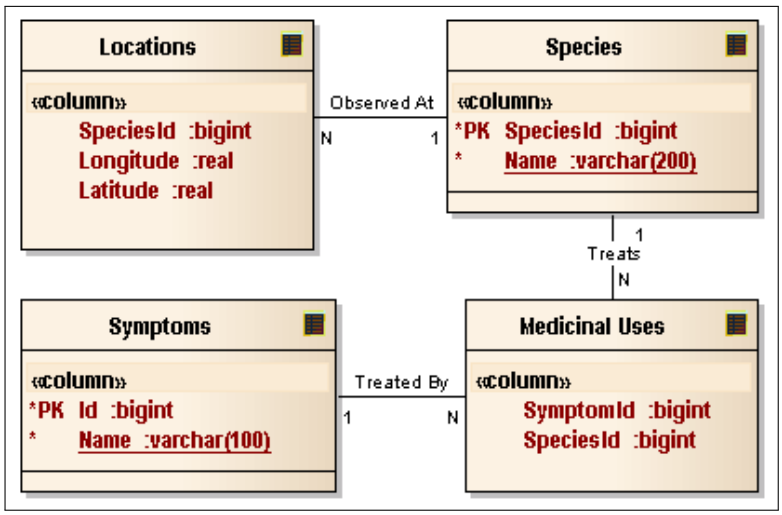

Figure 11.1.: The relational model of the SMV dataset prepared for the baseline. Each species can be observed at many locations. Symptoms can be treated by many species and each species can treat multiple symptoms. The medicinal uses are the symptoms the each species can treat.

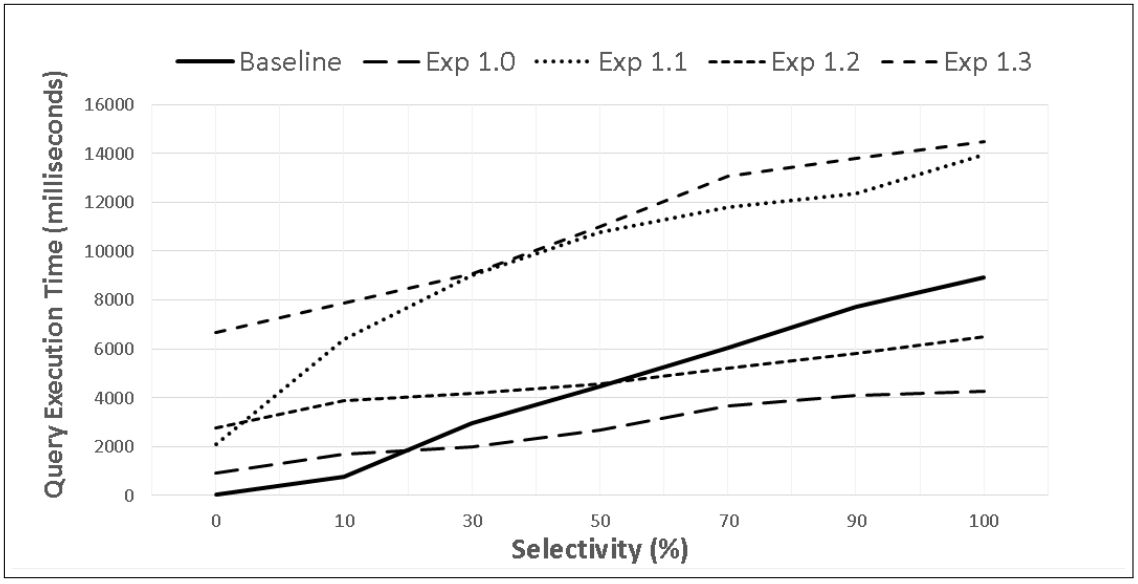

Figure 11.2.: QUIS performance evaluation on heterogeneous data. Here, Exp. 1.2 shows QUIS's performance on the original data, whereas Exp. 1.0, Exp. 1.1, and Exp. 1.3 denote QUIS query execution times for different data source combinations. The baseline is the performance on PostgreSQL.

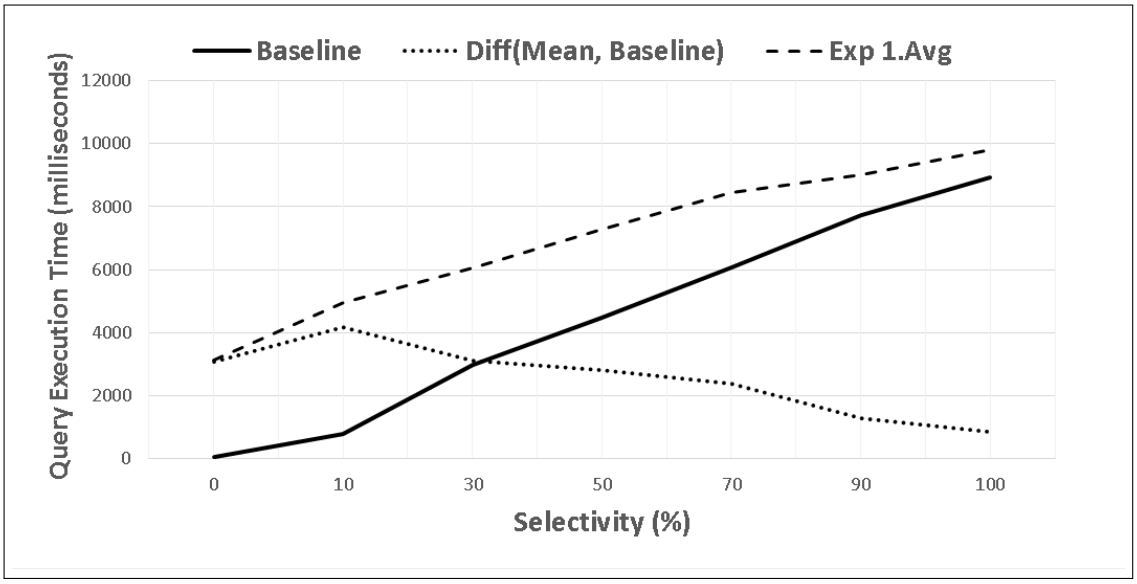

Figure 11.3.: QUIS's average performance versus the baseline. Here, Exp. 1. Avg is the average QUIS performance for all scenarios. The baseline is the PostgreSQL performance, and Diff(Mean, Baseline) is the difference between the two.

initial query time for lower selectivities is the fact that QUIS does not use indexing; hence, it needs to touch a subset of the columns for all records to be able to evaluate the WHERE clause predicates (and join keys if they are present in the query).

The QUIS scenarios Exp. 1. 1 and Exp. 1. 3, which involved access to the relational database, showed a steady performance, with a similar trend to that of the baseline. However, they were slower than the baseline by an average of 5.75 seconds. This is the penalty that QUIS pays for supporting join operations on heterogeneous data sources.

The Exp. 1. Avg chart in Figure 11.3 shows QUIS's average performance for all of the Exp. 1's scenarios. It is an indicator of QUIS's performance when applied to heterogeneous data. Although the chart scales linearly alongside the baseline, it keeps a distance from the baseline. However, as shown by the Diff(Mean, Baseline) chart, it decreases for larger selectivities.

As expected, QUIS had a higher query surcharge, i.e., the query time at selectivity 0% divided by the query time at selectivity 100%. This is because the QUIS engine creates a dynamic operator tree per query statement and compiles it on the fly. This process has a constant cost that depends upon the number and complexity of statements, schema, and capabilities exposed by the corresponding adapters, but is independent of the data.

For selectivities greater than 10%, the best, the average, and the worst query processing times for QUIS were respectively 10%, 47%, and 105% slower than their baseline counterparts. We omitted selectivities below 10% in order to isolate the impact of the constant time needed by QUIS dynamic code compilation. QUIS's best overall run was two times faster than the baseline,

while its worst run was seven times slower. During the repetition of the measurements, we found that baseline times had negligible deviation. In comparison, QUIS showed an average of 23% deviation between its fastest and slowest runs per scenario.

We additionally compared QUIS's performance with a non-relational query system, ViDa [K+15]. ViDa uses RAW [K+14] for querying raw data. We reproduced the data and the queries of the RAW experiment[2] and ran them on similar hardware, namely BIG-MACHINE. In addition, we used two storage configurations to observe the network effect.

On average, running the reproduced query on QUIS for all selectivity levels took 151 seconds on a local SSD and 226 seconds on a networked RAID system. RAW reported a 170-second response time. The RAW experiment used memory-mapped files to improve performance by allocating more memory. Disabling this feature caused the query to be approximately three times slower. QUIS uses a bound memory mapping that sets an upper limit of 2GB to the mapped portion of the designated file and uses buffering techniques to process the mapped data.

11.4 Scalability Evaluation

In addition to evaluating QUIS's ability to query heterogeneous data, we studied its scalability under datasets with a larger numbers of records and attributes. Here, we compared QUIS with R and DBI (see Section 11.1.2). The SMV dataset, though heterogeneous, was not large enough for this purpose; thus, we ran our scalability experiment on FNO, a larger dataset from the same environmental science use-case. While certain scientific datasets are massive in size (e.g., in astrophysics), other scientific datasets are often reported to be relatively small [Rex13]. This is consistent with the datasets that we have observed in our collaborations with Biodiversity Exploratories [Bio]. Thus, we ran this experiment on the FNO-SML dataset. However, we proved the system behavior on the larger variant, FNO-BIG.

We use the dataset to compute the temporal distribution of the frequency of occurrences of globally observed fungi on a yearly basis. This is a basic practice in environmental science that illustrates the fungi's histogram. We designed and ran experiment Exp. 2 to perform the above-mentioned task. Preparing the data required for such a task required counting the number of occurrences per fungus per year and ordering the results for each fungus by year.

We divided the experiment into three scenarios: Exp. 2. Baseline, Exp. 2. R, and Exp. 2. QUIS. The Exp. 2. QUIS scenario was conducted in two settings, one in which the workbench was used and one in which the R-QUIS package from inside R was used. While the Exp. 2. Baseline used a relational counterpart of the dataset, we kept the FNO dataset in its original form for the Exp. 2. QUIS and Exp. 2. R scenarios.

[2] We asked the authors of ViDa to access their system/data to conduct a comparison to QUIS. Unfortunately neither the system nor the data could be shared at the time of our request, but they advised us that running the RAW queries as explained in [K+14] on a regenerated version of their data would represent a fair comparison.

For the baseline, we used PostgreSQL's batch insert utility to load the dataset into a table. In addition, we added indexes on all of the attributes used in filtering, ordering, and group by clauses, namely `year`, `scientificName`, and `decimalLatitude`. We used the `year` attribute as our selectivity parameter. Since the `year` attribute was originally of the string type, we added an integer equivalent of the year (named `intYear`) to the table (and also built an index on it) to improve efficiency. Loading the data and indexing it took 763 and 305 seconds, respectively. Unfortunately, the standard R functions used to load a CSV file took too long to respond, so we omitted them from the final results.

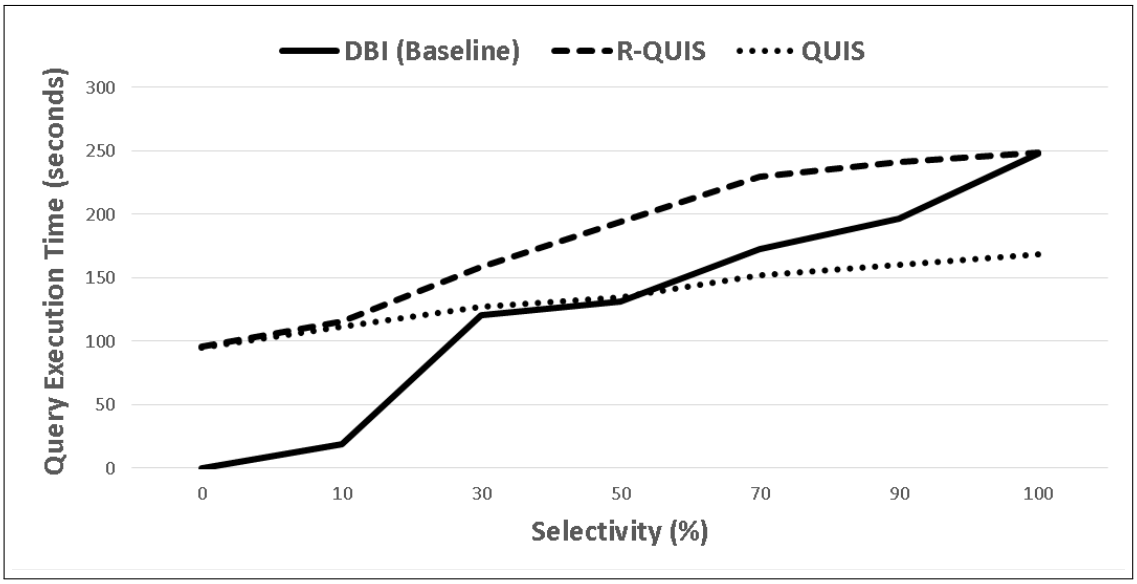

Figure 11.4.: `QUIS`'s performance results when applied to large data. Here, `DBI` is the baseline, `QUIS` is the `QUIS`'s query execution times run from its default workbench, and `R-QUIS` is the R package that utilizes `QUIS` to access data.

The results are depicted in Figure 11.4. The figure shows that the performance of `QUIS`'s query execution engine remains fairly constant over all the selectivity levels, both when called from its own workbench (the chart labeled `QUIS`) or from within R (the chart labeled `R-QUIS`). However, R-QUIS pays an extra cost to transform the query results into R data frames and it is thus slower. At 30% selectivity, all of the scenarios approach the neighborhood of 100 seconds and diverge thereafter. Our inspection revealed that this is the point at which the PostgreSQL query planner switches from an index scan to a table scan. This is also the point at which QUIS starts to outperform the `DBI`.

11.5 User Study

In the previous sections of the evaluation chapter, we showed that our `QUIS` proof of concept implementation is functional, demonstrates adequate performance, and is scalable. In this section,

through the analysis of the results of a user study, we show that QUIS is usable and useful in real-world application scenarios. To do so, we conducted a user study, in which a group of subjects were assigned a task (see Appendix C.2 (Task Specification)) to perform on both the baseline system and on QUIS. The task required working with heterogeneous data, joining, aggregation, and transformation (see Appendix C.3 (Task Data)). Our goal was to prove that QUIS's performance on a defined set of indicators is meaningfully superior to that of the baseline. To clarify the scope of the study and provide a basis for statistical analysis, we defined six indicators and set the data-obtaining methods for all of them. The indicators are time-on-task, machine time, code complexity, ease of use, usefulness, and satisfaction. The subjects were randomly chosen to begin with either the baseline or QUIS. The definitions of the indicators and the procedures are described in detail in Appendix C.1 (User Study Methods).

For each indicator I_i, we designed a null hypothesis $H0_i : \bar{x}_{b_i} \neq \bar{x}_{c_i} \rightarrow \mu_{b_i} = \mu_{c_i}$, in which b, and c are the baseline and the introduced change, respectively. The hypothesis declares that having different sample means does not imply a meaningful difference in the population mean. In other words, each $H0_i$ claims that the change introduced does not meaningfully improve I_i. Our desired result would be that the study rejects these hypotheses.

We designed a questionnaire intended to collect the subjects' responses to a set of questions designed to gauge the six indicators (see Appendix C.4 (Questionnaire)). We called for volunteer participation using different channels, e.g., R user groups, project members, colleagues, friends, and the faculty's students. In total, we received 32 task results, which included the scripts and the answer sheets. Our subjects were highly diverse in many dimensions, e.g., nationality, field of study, age, and gender. Among the participants, 62% were male and 38% were female, ranging between 24 and 38 years of age. More than 55% were German. The rest were from various countries including China, Iran, Russia, South Korea, and Ukraine. The study was conducted in a prepared laboratory. However, eight subjects managed to finish the assignment remotely. About 88% were master's students, while the other 22% were either PhD students or PhD holders.

Upon completion of the survey, we collected the answer sheets and prepared the raw data according to the requirements of the corresponding indicators. The resulting tables for both scenarios are presented in Appendix C.5 (Raw Data). We chose to use the t-test method, as we wanted to decide on the population mean based on the sample mean. Based on the fact that our subjects were identical in both evaluation scenarios, we had to use the paired samples t-test method. This method requires the data to expose a normal distribution. As our sample size $n = 32 < 50$, we used the Shapiro-Wilk [SW65] techniques to test the normality. The test results, as shown in Table 11.4, indicate that all of the significances (the $\texttt{sig}_{\texttt{sw}}$ column), are greater than 0.05 thus pass the test.

Table 11.5 shows the analysis of the null-hypotheses tests in that the t-values and their significances are computed for $Q - R$. The analysis clearly shows that QUIS has driven a meaningful change to the baseline on all of the indicators, with the exception of ease of use (EU). In the following paragraphs, we briefly explain the survey results.

Indicator	sc	$\bar{x}$	$\tilde{x}$	σ	v	sig_{sw}
Time-on-task (TT)	R	49.563	50.000	7.075	50.060	0.055
	Q	37.125	40.000	7.183	51.597	0.060
Machine Time (MT)	R	15.406	15.500	1.932	3.733	0.062
	Q	12.750	12.500	3.750	14.065	0.069
Code Complexity (CC)	R	3.935	3.935	0.427	0.183	0.072
	Q	3.318	3.386	0.342	0.117	0.067
Ease of Use (EU)	R	-0.464	-0.500	0.913	0.833	0.341
	Q	-0.223	-0.214	0.944	0.891	0.514
Usefulness (UF)	R	-0.224	-0.083	1.202	1.446	0.135
	Q	0.411	0.417	1.127	1.271	0.055
Satisfaction (SF)	R	0.568	0.667	0.849	0.720	0.132
	Q	0.151	0.500	1.013	1.026	0.067

Table 11.4.: Descriptive statistics of the survey results. sc: evaluation scenario, $\bar{x}$: mean, $\tilde{x}$: median, σ: standard deviation, v: variance, and sig_{sw}: the Shapiro-Wilk normality test's significance. The sig_{sw} value is above 0.05 for all the indicators, which means they pass the normality test required by the paired samples t-test.

Hypothesis	t-value	t-sig	H0 result	$\bar{x}_Q$-$\bar{x}_R$	boost%
$H0_{TT}$	-6.934	0.000	Rejected	-12.438	25
$H0_{MT}$	-3.790	0.001	Rejected	-2.656	17
$H0_{CC}$	-7.186	0.000	Rejected	-0.617	16
$H0_{EU}$	-2.022	0.052	Holds	0.241	51
$H0_{UF}$	2.165	0.038	Rejected	0.635	283
$H0_{SF}$	-2.247	0.032	Rejected	-0.417	-73

Table 11.5.: User study hypothesis test results for $t(31)$. t-sig: P(T<=t) t-significance (two-tailed). The t-values and their significances are computed for $Q - R$.

Time-on-Task (TT): The t-test rejects $H0_{TT}$ with a t-sig of zero, as shown in Table 11.5. This conveys the message that the time-on-task indicator has certainly been improved. The majority of the subjects used the total planned TT allocated quota of 45 minutes to accomplish the baseline scenario. They could exceed the quota by an additional five minutes. However, on average they spent 25% less time when using QUIS, which decreased the baseline mean of 49.6 minutes by more than 12 minutes. The subjects' time-on-task chart in Figure 11.5 clearly depicts the difference. In addition, the histogram in Figure 11.6 shows that not only does QUIS have more values around its mean but that it also shifted the mean towards the left.

Machine Time (MT): The t-test rejects $H0_{MT}$ with a t-sig of 0.001. This means that QUIS executes queries more rapidly than the baseline. Table 11.5 shows an average of a 17% performance boost on the machine time. However, it is worth mentioning that QUIS, as depicted in Figure 11.8, has a near uniform distribution of frequencies over a wider span. This fact

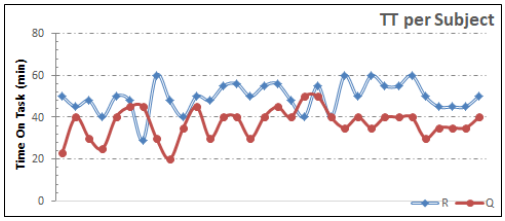

Figure 11.5.: comparison chart of the time-on-task indicator per subject.

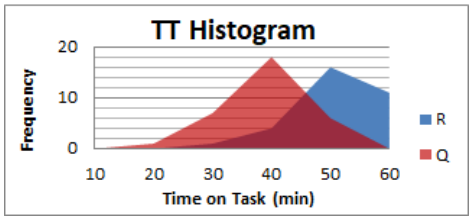

Figure 11.6.: Histogram of the time-on-task indicator on the baseline and QUIS.

is also reflected by the fluctuating line chart in Figure 11.7 and the larger standard deviation shown in Table 11.4. As we ran the subjects' tasks on a reference machine, this behavior can be attributed to our implementation techniques, especially on-the-fly compilation, schema and object caching, and also Java's garbage collection. This indicates that QUIS's query execution engine and implementation of adapters are not sufficiently mature to remain within a well-bound response time. The baseline, however, demonstrates superior stability here.

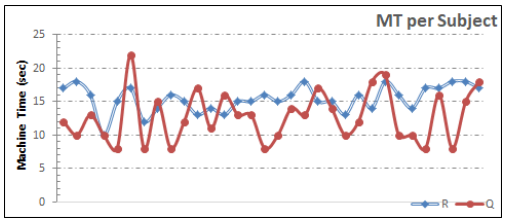

Figure 11.7.: Comparison chart of the machine time indicator per subject.

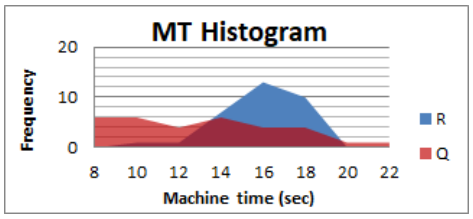

Figure 11.8.: Histogram of the machine time indicator on the baseline and QUIS.

Code Complexity (CC): The t-test rejects HO_{CC} with a t-sig of zero. The code complexity in the baseline is not only on average higher than that of QUIS but also fluctuates more (see Figure 11.9). This is because the subjects were required to use multiple packages to perform joins and aggregates, and access the different data sources; in addition, they had to use at least two languages, R and SQL, in order to complete the task. The histogram chart depicted in Figure 11.10 confirms this fluctuation. While the majority of the occurrences were between 3.0 and 4.0 for QUIS, the baseline's code complexity indexes were distributed over the $3.0 - 4.6$ range, with a lower frequency per point. Overall, QUIS reduced code complexity by 16%.

Ease of Use (EU): The t-test does not reject HO_{EU}, and it holds with a marginal t-sig value of 0.052. The observed difference between the sample means (0.241) is not convincing enough to claim a difference in terms of population means. Although both the systems show a very close mean per subject, as shown in Figure 11.11, QUIS has a more uniform histogram, which indicates the subjects' uncertainty about its usefulness (see Figure 11.12). The baseline has

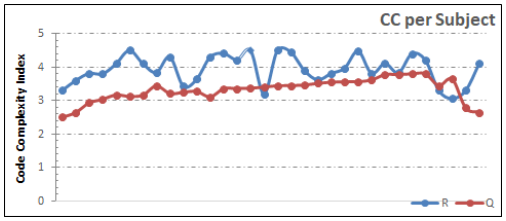

Figure 11.9.: Comparison chart of the code complexity indicator per subject.

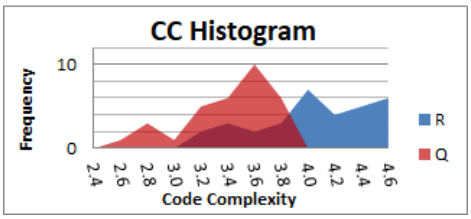

Figure 11.10.: Histogram of the code complexity indicator on the baseline and QUIS.

a slightly more normal histogram. Interestingly, QUIS's mean is slightly greater than of the baseline. This indicates that an effort to improve the EU would likely result in it passing the test.

In general, tests that measure human behavior have lower confidence. We can refer to many factors that may have contributed to this result; For example, embedding QUIS's SQL-like syntax in R negatively affects ease of use. Although we conducted the study using a native R package, R-QUIS, and attempted to minimize the mixture of languages, it is likely the major factor that weakened the results. Additionally, providing richer documentation during the study and from inside the IDEs could improve ease of use. Furthermore, more accurate and to-the-point error messages, as well as code completion features, are needed.

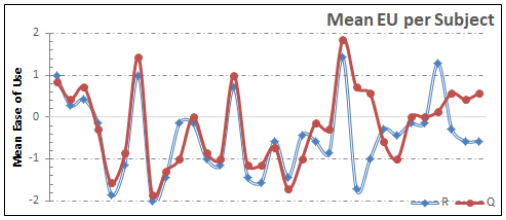

Figure 11.11.: Comparison chart of the ease of use indicator per subject.

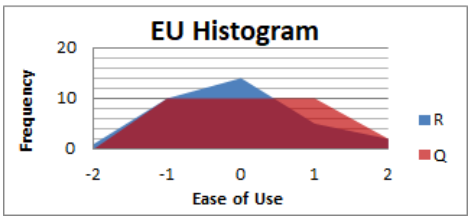

Figure 11.12.: Histogram of the ease of use indicator on the baseline and QUIS.

Usefulness (UF): The t-test rejects $\mathrm{HO}_{\mathrm{UF}}$ with a t-sig of 0.038. This is an excellent indicator that the subjects felt that QUIS was useful. Indeed, QUIS dramatically improved this indicator. Not only are its mean and median greater that of the baseline but it also obtained a positive mean (0.411), while the baseline suffered from a negative value of -0.244 for usefulness.

Despite its proven improvement, QUIS's per-subject mean responses, shown in Figure 11.13, reflect the fluctuations in the subjects' opinions. This pattern is also reflected in the histogram shown in Figure 11.14. The histogram shows two condensed areas around 0 and 2 that although both positive show a degree of fragmentation among the subjects. One possible reason for this dual peak histogram can be driven from the assumption that those who voted near zero were mostly the subjects who were satisfied with R and who used conventional single data source

analyses. These categories of data workers frequently prepare their data in advance, possibly using different systems, and perform their analyses on integrated data sets using tools such as R. The other category of the subjects saw QUIS as being more useful due to its ability to obtain and query data from multiple data sources simultaneously and on-the-fly. This class of data workers usually obtains raw data and tend to do Extract, Transform, Load, and Analyze (ETL-A) operations in an exploratory manner. QUIS represents a better fit for these kinds of usage scenarios.

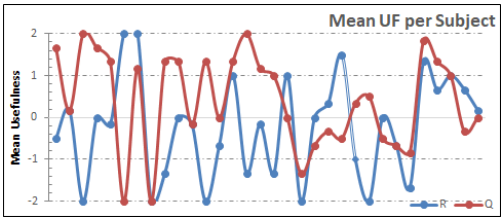

Figure 11.13.: Comparison chart of the usefulness indicator per subject.

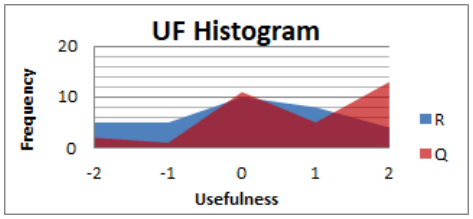

Figure 11.14.: Histogram of the usefulness indicator on the baseline and QUIS.

Satisfaction (SF): Despite the rejection of the HO_{SF} null hypothesis by the t-test with a t-sig of 0.032, QUIS's mean difference to the baseline is -0.417. This means that QUIS negatively affected the satisfaction indicator relative to the baseline. Therefore, we confirm that QUIS failed on the satisfaction indicator. Figure 11.15 conveys the overall fluctuating pattern of the subjects' responses. However, it illustrates that QUIS should have a lower mean. Table 11.4 shows that QUIS and the baseline have a close median but a remarkable mean difference. This mean difference is most likely rooted in the long tail of the $\{-2..0\}$ responses, as depicted in the satisfaction histogram of Figure 11.16. Despite this rejection, QUIS has obtained a satisfaction mean of 0.151, which we consider a step forward for an infant system implemented as a proof of concept.

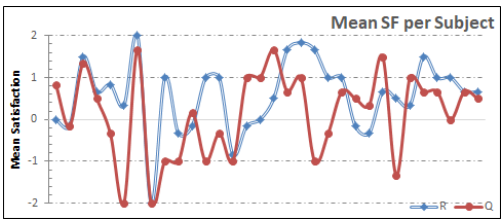

Figure 11.15.: Comparison chart of the satisfaction indicator per subject.

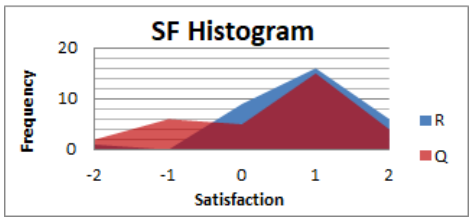

Figure 11.16.: Histogram of the satisfaction indicator on the baseline and QUIS.

Overall, we found out that, while the measured indicators (TT, MT, and CC) illustrated a clear boost, the observed ones (EU, UF, and SF) communicated mixed results. For example, UF shows a dramatic increase in QUIS's usefulness. However, the EU and SF convey the message that

the subjects did not think that QUIS was easier to use than the baseline and that they were not satisfied with it. Considering that they had only a brief introduction to QUIS and that the tool itself was a proof of concept with technical, UI, and documentation issues, we consider the survey findings to be in support of our solution.

11.6 Language Expressiveness

In Table 11.6, we compare the features of the QUIS's language with their counterparts in the languages studied in Section 6.4. The languages previously discussed provides a solid foundation for the comparison and indicate the expressiveness of QUIS's language. The feature comparison is performed at the language level. Therefore, the level of conformance may differ from implementation to implementation.

It is worth mentioning that some of QUIS's language constructs, such as *perspectives* and *bindings*, are QUIS-specific concepts that have no counterparts in other languages. These features, while useful, are not part of the comparison.

The *projection* feature supports explicit, implicit, and inline perspectives. Implicit projections can be inferred form either previous queries or the underlying data sources. One of the main contributions of this feature is the ability to extract and construct a complete projection from external sources. Although SQL and SciQL are marked as supporting implicit projection construction, they only accomplish this using their own data sets, namely table and arrays.

The *selection* feature has access to perspective attributes as well as physical fields of the underlying data containers. This allows QUIS to filter queried records based on variables that are not intended to appear in result sets. At the same time, QUIS fully supports projection aliases in the selection predicates, a feature that some RDBMSs, e.g., PostgreSQL, do not support.

QUIS allows a data worker to declare and use heterogeneous data sources, a feature no other system can match. In addition, it supports query chaining, meaning that the result of a query can be fed into one or more following queries. On top of its data source selection features, QUIS supports all types of joins over any combination of heterogeneous data. QUIS not only retrieves data from heterogeneous sources but is also able to persist query results into different data sources and, on request, in various formats. It can also visualize the results. Furthermore, it integrates this feature into the language, and thus makes it always available to the end-user. It is also noteworthy that SPARQL and Cypher, although they operate on their own respective data models, return query results in a tabular form. They may construct compound cell values for the matched elements with multiple attributes.

Last, but not least, QUIS is equipped with a path expression language that enables it to uniformly express tabular, hierarchical, and graph paths and patterns. The paths can traverse in any valid direction, have cardinality, and include conditions and sequences, touch elements, attributes, and relations, as relation properties. It is also possible to express sub-trees, sub-graphs, and cycles.

	QUIS	SQL	SPARQL	Cypher	SciQL
PROJECTION					
Implicit from the data	✓	✓	✓	✗	✓
Implicit from other queries	✓	✓	✓	✗	✓
Explicit projection	✓	✓	✓	✓	✓
Use of expressions	✓	✓	✓	✓	✓
Use of aggregates	✓	✓	✓	❋	✓
Use of multi-aggregates	✓	✓	✓	❋	✓
SELECTION					
Use of expressions	✓	✓	✓	✓	✓
Access to the projection's aliases	✓	❋	✗	✗	✗
DATA SOURCE					
Heterogeneous data querying	✓	✗	✗	✗	✗
Chained query results	✓	✗	✗	✗	✗
Sub-queries	✗	✓	✓	✓	✓
Homogeneous joins	✓	✓	✓	❋	✓
Heterogeneous joins	✓	✗	✗	✗	✗
QUERY TARGET					
Tabular result set	✓	✓	✓	✓	✓
Hierarchical result set	✗	✗	✗	✗	✗
Graph result set	✗	✗	✗	✗	✗
Homogeneous persistence	✓	✓	✓	✓	✓
Heterogeneous persistence	✓	✗	✗	✗	✗
Visualized result set	✓	✗	✗	✗	✗
RESULT SET PAGING					
Skip on objects	✓	❋	✓	✓	✓
Take objects	✓	❋	✓	✓	✓
ORDERING					
NULL ordering	✓	✓	✗	✗	✓

Table 11.6.: Comparison of QUIS's features with those of the related work. Only the important features that are supported in different ways are shown. ❋ indicates partial support or implementation variety.

Part IV.

Conclusion and Future Work

This part concludes this dissertation. In Chapter 12, we briefly reiterate our assumptions, the solution we provided, and the results of the evaluation. Thereafter, we bring the dissertation to a close by reviewing its achievements and the extent to which the hypothesis is satisfied. Finally, in Chapter 13, we examine a set of important directives for future work.

12

Summary and Conclusions

We explained, and with the aid of a deep literature review, demonstrated that scientific data is stored using different representations, with various levels of schema, and changes at different rates. In addition, we illustrated that the software systems used to manage and process such data are incompatible, incomplete, and diverse. We also established that data workers often have to integrate data from heterogeneous sources in order to conduct the end-to-end processes that are required to obtain meaningful results. Their usual patterns of dealing with data differ from those of business applications. Scientists usually do not need an entire set of available data; rather, their portions of data that they require change over the course of their research. This exploratory nature prevents scientists from deciding on data schemas, tool sets, and pipelines during early stages of their research. They usually need to perform data integration not only to prepare data for planned analyses but also to enable various tools to function together in pipelines or workflows. However, as discussed, the two classical approaches to data integration, i.e., materialized and virtual integration, do not solve these scientific data management and processing problems.

We identified the concept of *data access heterogeneity* as an important root-cause of the data integration problem. This term refers to diversity in terms of the computational models (e.g., procedural and declarative), querying capabilities, syntaxes, and semantics of the capabilities provided by different vendors or systems, data types, and presentation formats of query results. Therefore, assuming data workers (Definition 1.1) to be the principal stakeholders, in Chapter 3 (Problem Statement) we formulated and established boundaries for the problem. Supported by numerous studies, we argued that this is a multi-dimensional problem that is rooted in heterogeneity in data organizations (Definition 1.2), as well as in data management systems. Furthermore, we observed that these heterogeneities are still in the diverging phase and will continue for at least the following decade. Based on this reality, we offered a solution that embraces data and system heterogeneities. Our suggested solution draws an abstraction layer on top of a chosen heterogeneous environment in order to provide an integrated and unified data access mechanism. The term *"unified"* guarantees that syntax, semantics, and execution of input queries remains identical throughout the heterogeneous environment. To address various requirements, we divided the solution into three components: declaration, transformation, and execution. We designed a declarative language intended to provide an expressive interface for data workers to define and declare their data retrieval, processing, and persistence requirements independently of underlying data organization, data sources, and the various capabilities of such sources. In order to guarantee the execution of queries written in our suggested unified language, we also

specified a unified execution mechanism. This execution mechanism promises to fully execute all of the language's constructs, regardless of the capabilities of the underlying concrete data management systems. Between these two components, we introduced a query transformation component, the main role of which is to transform user queries written in our unified language into a set of appropriate computational models that the execution component can seamlessly execute. These components are orchestrated by a query execution engine. They jointly provide important features, such as in-situ querying, heterogeneous joins, query complementing, and polymorphic result presentation.

We implemented `QUIS` to prove the feasibility of the suggested solution. `QUIS` is an agile query system that is equipped with a unified query language and a federated execution engine that is able to run queries on heterogeneous data source in an in-situ manner. Its language extends standard SQL to provide advanced features, such as virtual schemas, heterogeneous joins, and polymorphic result set presentation. `QUIS` provides union and join capabilities over an unbound list of heterogeneous data sources. In addition, it offers solutions for heterogeneous query planning and optimization.

This proof-of-concept implementation greatly satisfied this thesis' hypothesis that a universal querying system would prove both feasible and useful. The hypothesis identified the following three objectives:

- To provide a unified data retrieval and manipulation mechanism on heterogeneous data that is independent of data organization and data management;
- The provided mechanism should be expressive enough to support the core features of the most frequently used queries; and
- The mechanism should reduce the time-to-first-query while maintaining reasonable performance for subsequent queries, be scalable, and be useful in real-world scenarios.

Unified data retrieval was achieved, as discussed in Section 6.5 (QUIS Language Features) and summarized in Chapter 9 (Summary of Part II). We conducted a series of experiments and surveys intended to demonstrate that the performance (Section 11.2: Measuring Time-to-first-query) and (Section 11.3: Performance on Heterogeneous Data), scalability (Section 11.4: Scalability Evaluation), and usefulness (Section 11.5: User Study) requirements were satisfied. The expressiveness of the suggested language was examined in Section 11.6 (Language Expressiveness).

In the course of this thesis, we realized the features required by the solution, satisfying the requirements and thus the objectives. Furthermore, we introduced a handful of novel algorithms and techniques that not only fulfill our requirements and ensure unified query execution but can also be generalized and used in other systems. For example, our query complementing algorithm detects capability mismatches between an input query and a designated data source. The difference is then complemented by an automatic and transparent mechanism, allowing the system to fully execute any input query. It also enables the adapters to provide various levels of

support for the query language and to evolve over time. An interesting capability of our solution is its dynamic query compilation, packaging, and execution. While the transformation component generates a concrete counterpart of each and every input query, the execution component groups them together based on their data dependencies and compiles them on-the-fly into standalone jobs. These jobs are shipped to the execution engine and can be easily triggered to launch and run. These techniques eventually led to the concept of *OFF-DBMS* (on-the-fly federated DBMS). Meanwhile, these techniques produced two other remarkable benefits: First, jobs can be shipped to remote data. This yields an unmatched performance gain for big data, data repositories, data centers, and wherever access to data is restricted. Second, the jobs can be maintained alongside research findings for reproducibility. Considering the engine's small footprint, its open-source and free licensing scheme, and its self-contained nature, it would be easy to maintain the jobs (and the engine) as part of the publication of research.

Among our optimization rules, the weighted short-circuit evaluation (see Section 7.4.1.7) takes into account the cost of executing each node in a query's predicate evaluation tree and executes the cheapest path. This is a remarkable performance improvement that can be generalized and utilized by other DBMSs. Additionally, under some circumstances, it could also be used in the compilers of imperative languages.

One of the heterogeneity dimensions of various systems is the manner in which each system expresses access to data. Our universal path expression grammar (see Section 6.5.1.5) relieves its users from the need to be bound to data persistence and serialization variety. They are able to freely express access to tabular, hierarchical, and graph data using a single unified syntax and leave the complexity of understanding and mapping them to the actual data to `QUIS`'s transformation component.

`QUIS`'s perspectives and the polymorphic result presentation technique decouple data workers' processes from the mechanics of data organization. A perspective creates a late-bound virtual schema that is appropriate for the logical flow of the process and conceals the actual underlying data source, data types, formats, and units of measurement. At the other end, the result set is also presented in the structure and form requested by the data worker. The data structure complies with the bound effective perspective; moreover, it can be presented in either tabular or visual forms. Furthermore, the data can be serialized to formats that are consumable by external tools.

13
Future Work

The goal of this thesis was to demonstrate the feasibility of a unified execution system in order to illustrate the benefits and the potential of a heterogeneous agile querying system that operates on a mixed federation of data sources with multi-dimensional heterogeneity. Achieving this goal required us to consider a large number of different research issues and involved the application and adoption of a wide range of techniques. We decided to focus on the core elements of our suggested solution and provided a proof of concept with enough evidence to indicate its feasibility and usability. This assisted us to remain within the defined scope of the developed hypothesis. We expect that the contributions of this thesis can be extended and improved in several directions. In the rest of this chapter, we briefly identify possible future directions.

Cost-based query optimization: In Section 7.4, we introduced our rule-based optimizations and justified our approach to building a zero-knowledge query optimizer. However, cost-based query optimization has it merits [Cha98]. The challenge with cost-based optimization is applying it in heterogeneous data environments, in which the structure of data is not known to the optimizer in advance [ACPS96]. In `QUIS`, cost-based query optimization can be done at the language and/or adapter level.

Adapters have knowledge of their own managed data organizations and are the best places in which to perform data-source-specific optimizations. Based on each designated data organization, adapters may use different techniques to compute query costs. The cost functions could require a query engine to generate and maintain metadata and statistics. For example, improving access to a CSV file may rely on maintaining individual or clustered mapping to the field positions [ABB+12].

Moreover, other types of cost functions that better describe a specific data organization or estimate cost using indirect indicators are emerging. Adaptive partitioning [OKA+17] observes data access patterns and dynamically decides to partition the data, and indexes the partitions iteratively. In contrast to conventional optimization techniques that rely on cardinality estimation based on pre-computation, progressive optimization suggests gradual and concurrent (to query execution) cost estimation and adjustment [EKMR06]. However, the frequency and the cost of such re-optimizations remain high. Indirect cost estimation offers a cheaper re-optimization. For example, it is possible to use performance metrics to build a model for rapid estimation during the query execution of in-memory databases [ZPF16]. Adapters, as well as QEEs can incorporate these kinds of cost functions.

Visualization recommendation: Visualization is an important component of the presentation and communication of scientific research [War12]. Selecting an appropriate form of visualization is as important as the information that will be communicated via the chosen visualization. The search space required to find and select an appropriate visualization grows with the amount of data and the increasing variety of visualization techniques [CEH+09]. Therefore, offering assistance in choosing proper visualization would represent a value-added service for scientific data exploration tools. Recommendation systems have been integrated in many domains for some time now [KKR10, KR14]. Data science could reap real benefits from such recommendation systems. Providing visualizations of data in a (semi-)automatic fashion based on the characteristics of the data and queries, as well as historical information and user feedback, could meaningfully improve the usability and satisfaction of any data analysis system [KOKR15].

Vartak et al. [VHS+17] classified the factors that a system that provides visualization recommendations should possess: Such a system should consider data characteristics, the intended task or insight, semantics and domain knowledge, visual ease of understanding, and user preferences and competencies. Many of these factors can be, to a great extent, extracted from data and queries. The implicit and/or explicit user responses to previous visualizations can be accumulated and used to offer superior recommendations. Furthermore, semantics and domain knowledge can be acquired by the integration of general purpose and domain-specific ontologies [GSGC08]. However, visualization recommendation is not an easy task, particularly in the presence of user-centric factors such as usability, scalability, quality, and causality [Che05].

Deep complementing: A number of possible improvements and extensions to our proposed approach refer to the query complementing concept introduced in Section 7.3. As explained previously, although it can theoretically traverse the input query to its stem level, the query complementing process currently operates only at the level of the input query feature. In other words, the transformation component negotiates the features with the adapters and determines whether they support the entire feature as a unit. Although this mechanism is good enough for most scenarios, a dynamic and fine-grained query complementing algorithm capable of transforming sub-features or even elements such as functions and operators would enhance the overall performance of a system on weak data sources as well as on weak adapters. One such improvement could be to decompose the predicates to their conjunctives and evaluate which segments could be pushed to the data source.

Schema mapping: We perform a linear expression-based schema mapping between the attributes of perspectives and their counterparts in underlying data sources. Although our path expression is able to express hierarchical and graph-shaped paths and patterns, the schema-mapping techniques utilize it only partially. This represents an extension point to the language that could enable it to express hierarchical and networked data. Incorporating this extension would allow data scientists to easily target XML, JSON, and social data. It would also open the door for life science disciplines to express molecules, proteins, and chemical formulas. Furthermore, effective schema-mapping techniques, either in the form of complete [MMP+11,

BKCS10, GJSD09] or partial mappings [KOKS15], could be incorporated into QUIS's perspective declaration to assist data scientists to (semi-)automatically describe their data.

Query shipping and remote execution: We were able to transform input queries and compile them on-the-fly to build a set of natively and independently executable jobs. The jobs are serialized to IO, storage, or networks. We did this primarily for performance reasons, but these techniques could be used in many other scenarios. For example, a job can be persisted with the data or the results as a matter of proof or for purposes of reproducibility [VBP+13].

The availability of clouds and data centers empowers Jim Gray's recommendation [SB09] that processes be sent to data. This, in turn, implies that all process aspects should be transferred to the data, including authorization and configuration information. One useful application of QUIS's jobs is that they can be shipped to data centers for remote execution. As the jobs are standalone, self-contained, stateless execution units that run on JVM, dispatching them to data centers should be trivial. However, jobs may be designed to work in a flow. Therefore, inter-job communication and the orchestration of execution of jobs in a distributed environment could represent an interesting topic of research in data processing and in distributed software development. Security concerns should be added to the complexity of such an ecosystem.

Real-world application: It would be desirable to deploy our approach in a real-world system to not only evaluate it but also to obtain real-world user scenarios and expand upon and/or improve the language and the related components. As we demonstrated in Chapter 11 (System Evaluation), QUIS is fast, scalable, and competitive. Therefore, plugging it into one or more widely used system would not only facilitate the above-mentioned goals but also reduce users' burdens. Embedding QUIS in an RDBMS, e.g., PostgreSQL, or a data processing framework, e.g., Apache Flink [CKE+15] to empower them to access heterogeneous data would be an appropriate test for validating QUIS and determining whether it could generate additional benefits.

References

[AAA+16] Daniel Abadi, Rakesh Agrawal, Anastasia Ailamaki, Magdalena Balazinska, Philip A. Bernstein, Michael J. Carey, Surajit Chaudhuri, Jeffrey Dean, AnHai Doan, Michael J. Franklin, Johannes Gehrke, Laura M. Haas, Alon Y. Halevy, Joseph M. Hellerstein, Yannis E. Ioannidis, H. V. Jagadish, Donald Kossmann, Samuel Madden, Sharad Mehrotra, Tova Milo, Jeffrey F. Naughton, Raghu Ramakrishnan, Volker Markl, Christopher Olston, Beng Chin Ooi, Christopher Ré, Dan Suciu, Michael Stonebraker, Todd Walter, and Jennifer Widom. The Beckman Report on Database Research. *Communications of the ACM*, 59(2):92–99, February 2016. url = http://doi.acm.org/10.1145/2845915.

[Aab04] Anthony A Aaby. *Introduction to programming languages*. Walla Walla College, draft version 0.9 edition, July 2004.

[AAB+09] Rakesh Agrawal, Anastasia Ailamaki, Philip A. Bernstein, Eric A. Brewer, Michael J. Carey, Surajit Chaudhuri, Anhai Doan, Daniela Florescu, Michael J. Franklin, Hector Garcia-Molina, Johannes Gehrke, Le Gruenwald, Laura M. Haas, Alon Y. Halevy, Joseph M. Hellerstein, Yannis E. Ioannidis, Hank F. Korth, Donald Kossmann, Samuel Madden, Roger Magoulas, Beng Chin Ooi, Tim O'Reilly, Raghu Ramakrishnan, Sunita Sarawagi, Michael Stonebraker, Alexander S. Szalay, and Gerhard Weikum. The Claremont Report on Database Research. *Commun. ACM*, 52(6):56–65, 2009.

[AAB+17] Serge Abiteboul, Marcelo Arenas, Pablo Barceló, Meghyn Bienvenu, Diego Calvanese, Claire David, Richard Hull, Eyke Hüllermeier, Benny Kimelfeld, Leonid Libkin, et al. Research Directions for Principles of Data Management (Dagstuhl Perspectives Workshop 16151). *arXiv preprint arXiv:1701.09007*, 2017.

[ABB+12] Ioannis Alagiannis, Renata Borovica, Miguel Branco, Stratos Idreos, and Anastasia Ailamaki. NoDB in action: adaptive query processing on raw data. *Proceedings of the VLDB Endowment*, 5(12):1942–1945, 2012.

[ABC+76] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim N Gray, Patricia P. Griffiths, W Frank King, Raymond A. Lorie, Paul R. McJones, James W. Mehl, et al. System R: relational approach to database management. *ACM Transactions on Database Systems (TODS)*, 1(2):97–137, 1976.

[ABML09] Manish Kumar Anand, Shawn Bowers, Timothy Mcphillips, and Bertram Ludäscher. Exploring scientific workflow provenance using hybrid queries over nested data and lineage graphs. In *International Conference on Scientific and Statistical Database Management*, pages 237–254. Springer, 2009.

[ACPS96] S. Adali, K. S. Candan, Y. Papakonstantinou, and V. S. Subrahmanian. Query Caching and Optimization in Distributed Mediator Systems. *SIGMOD Rec.*, 25(2):137–146, June 1996.

[Ail14] Anastasia Ailamaki. Running with Scissors: Fast Queries on Just-In-Time Databases. 30th IEEE International Conference on Data Enginreeing, April 2014.

[AK98] Jose Luis Ambite and Craig A Knoblock. Flexible and Scalable Query Planning in Distributed and Heterogeneous Environments. In *AIPS*, pages 3–10, 1998.

[AKD10] Anastasia Ailamaki, Verena Kantere, and Debabrata Dash. Managing Scientific Data. *Communications of the ACM*, 53(6):68–78, 2010.

[ALW+06] Rafi Ahmed, Allison Lee, Andrew Witkowski, Dinesh Das, Hong Su, Mohamed Zait, and Thierry Cruanes. Cost-based Query Transformation in Oracle. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 1026–1036. VLDB Endowment, 2006.

[APA+16] Franco D Albareti, Carlos Allende Prieto, Andres Almeida, Friedrich Anders, Scott Anderson, Brett H Andrews, Alfonso Aragon-Salamanca, Maria Argudo-Fernandez, Eric Armengaud, Eric Aubourg, et al. The Thirteenth Data Release of the Sloan Digital Sky Survey: First Spectroscopic Data from the SDSS-IV Survey Mapping Nearby Galaxies at Apache Point Observatory. *arXiv preprint arXiv:1608.02013*, 2016.

[AXL+15] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark SQL: Relational data processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394. ACM, 2015.

[B+14] Anant Bhardwaj et al. DataHub: Collaborative Data Science & Dataset Version Management at Scale. *arXiv*, 2014.

[BBC+10] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, and Jérôme Siméon. XML Path Language (XPath) 2.0 (Second Edition). W3C Recommendation, World Wide Web Consortium, December 2010.

[BDH+95] Peter Buneman, Susan B Davidson, Kyle Hart, Chris Overton, and Limsoon Wong. A Data Transformation System for Biological Data Sources. In *21st Conference on Very Large Data Bases*, pages 158–169, Zuerich, Switzerland, 1995.

[BEG+11] Kevin Beyer, Vuk Ercegovac, Rainer Gemulla, Andrey Balmin, Mohammed Eltabakh, Carl-Christian Kanne, Fatma Özcan, and Eugene Shekita. Jaql: A Scripting Language for Large Scale Semistructured Data Analysis. In *Proceedings of the 37th International Conference on VLDB*, volume 4 of *Proceedings of the VLDB Endowment*, pages 1272–1283, Seattle, USA, 2011. VLDB Endowment.

[bex] BExIS++, Biodiversity Exploratory Information System. http://fusion.cs.uni-jena.de/bexis. Accessed: 2015-10-11.

[BHS09] Gordon Bell, Tony Hey, and Alex Szalay. Beyond the Data Deluge. *Science*, 323(5919):1297–1298, 2009.

[Bio] Biodiversity Exploratories. Exploratories for Large-Scale and Long-Term Functional Biodiversity Research. http://www.biodiversity-exploratories.de. German Research Foundation (DFG) Priority Programm No. 1374.

[BK93] Athman Bouguettaya and Roger King. Large Multidatabases: Issues and Directions. In *Proceedings of the IFIP WG 2.6 Database Semantics Conference on Interoperable Database Systems (DS-5)*, pages 55–68, Amsterdam, The Netherlands, The Netherlands, 1993. North-Holland Publishing Co.

[BKCS10] Shawn Bowers, Jay Kudo, Huiping Cao, and Mark P Schildhauer. ObsDB: A System for Uniformly Storing and Querying Heterogeneous Observational Data. In *IEEE 6th International Conference on e-Science*, pages 261–268. IEEE, 2010.

[Bro10] Paul G. Brown. Overview of sciDB: Large Scale Array Storage, Processing and Analysis. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 963–968, New York, NY, USA, 2010. ACM.

[BS08] Stefan Berger and Michael Schrefl. From Federated Databases to a Federated Data Warehouse System. In *Proceedings of the 41st Annual ICSS*, pages 394–394, 2008.

[BWB09] Iain E Buchan, John M Winn, and Christopher M Bishop. A Unified Modeling Approach to Data-Intensive Healthcare, 2009.

[BWB+14] Spyros Blanas, Kesheng Wu, Surendra Byna, Bin Dong, and Arie Shoshani. Parallel Data Analysis Directly on Scientific File Formats. In *ACM SIGMOD ICMD*, pages 385–396, 2014.

[Cat11] Rick Cattell. Scalable SQL and NoSQL Data Stores. *ACM Sigmod Record*, 39(4):12–27, 2011.

[CDA01] Silvana Castano and Valeria De Antonellis. Global Viewing of Heterogeneous Data Sources. *IEEE Transactions on Knowledge and Data Engineering*, 13(2):277–297, 2001.

[CEH+09] Min Chen, David Ebert, Hans Hagen, Robert S Laramee, Robert Van Liere, Kwan-Liu Ma, William Ribarsky, Gerik Scheuermann, and Deborah Silver. Data, Information, and Knowledge in Visualization. *IEEE Computer Graphics and Applications*, 29(1), 2009.

[CEN+12] Joe Conway, Dirk Eddelbuettel, Tomoaki Nishiyama, Sameer (during 2008) Kumar Prayaga, and Neil Tiffin. R Interface to the PostgreSQL Database System. http://cran.r-project.org/web/packages/RPostgreSQL/index.html, January 2012. 0.4 edition.

[CEN+14] Joe Conway, Dirk Eddelbuettel, Tomoaki Nishiyama, Sameer Kumar Prayaga, and Neil Tiffin. DBI: R Database Interface, 2014. 0.3.1 edition.

[Cha98] Surajit Chaudhuri. An Overview of Query Optimization in Relational Systems. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '98, pages 34–43, New York, NY, USA, 1998. ACM.

[Cha15] Javad Chamanara. QUIS Workbench, the Default Workbench for QUIS. https://github.com/javadch/SciQuest/tree/0.3.0, 2015. Accessed: 2015-11-10.

[Che05] Chaomei Chen. Top 10 Unsolved Information Visualization Problems. *IEEE computer graphics and applications*, 25(4):12–16, 2005.

[CK04] Jeremy Carroll and Graham Klyne. Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C recommendation, W3C, February 2004. http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/.

[CKE+15] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink: Stream and Batch Processing in a Single Engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[CKR12] Javad Chamanara and Birgitta König-Ries. SciQL: a Query Language for Unified Scientific Data Processing and Management. In *PIKM*, pages 17–24. ACM, 2012.

[CKRJ17] Javad Chamanara, Birgitta König-Ries, and H. V. Jagadish. QUIS: In-situ Heterogeneous Data Source Querying. *Proc. VLDB Endow.*, 10(12):1877–1880, August 2017.

[CMZ08] Carlo A. Curino, Hyun J. Moon, and Carlo Zaniolo. Graceful Database Schema Evolution: the PRISM Workbench, 2008.

[Cod70] Edgar F Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, 1970.

[Cor16] Oracle Corporation. The MySQL's CSV Storage Engine. https://dev.mysql.com/doc/refman/5.7/en, 2016. Accessed: 2016-04-13.

[CTMZ08] Carlo A. Curino, Letizia Tanca, Hyun J. Moon, and Carlo Zaniolo. Schema Evolution in Wikipedia: Toward a Web Information System Benchmark. In *ICEIS*, 2008.

[CYV] Javad Chamanara, Clayton Yochum, and Ellis Valentiner. R-QUIS: The R Package for QUIS. https://github.com/javadch/RQt/releases/tag/0.1.0. Accessed: 2015-11-11.

[D+13] David J. DeWitt et al. Split Query Processing in Polybase. In *ICMD*, pages 1255–1266, 2013.

[dBCF+16] Jorge de Blas, Marco Ciuchini, Enrico Franco, Diptimoy Ghosh, Satoshi Mishima, Maurizio Pierini, Laura Reina, and Luca Silvestrini. Global Bayesian Analysis of the Higgs-Boson Couplings. *Nuclear and Particle Physics Proceedings*, 273:834–840, 2016.

[DES+15] Jennie Duggan, Aaron J Elmore, Michael Stonebraker, Magda Balazinska, Bill Howe, Jeremy Kepner, Sam Madden, David Maier, Tim Mattson, and Stan Zdonik. The BigDAWG Polystore System. *ACM Sigmod Record*, 44(2):11–16, 2015.

[DFR15] Akon Dey, Alan Fekete, and Uwe Röhm. Scalable Distributed Transactions Across Heterogeneous Stores. In *ICDE*, pages 125–136, 2015.

[DG08] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *ACM*, 51:107–113, 2008.

[DH02] Amol Deshpande and Joseph M Hellerstein. Decoupled Query Optimization for Federated Database Systems. In *Proceedings of 18th International Conference on Data Engineering*, pages 716–727. IEEE, 2002.

[Dha13] Vasant Dhar. Data Science and Prediction. *Commun. ACM*, 56(12):64–73, December 2013.

[DHI12] AnHai Doan, Alon Halevy, and Zachary Ives. *Principles of Data Integration*. Elsevier, 2012.

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007.

[EDJ+03] Barbara Eckman, Kerry Deutsch, Marta Janer, Zoé Lacroix, and Louiqa Raschid. A Query Language to Support Scientific Discovery. In *Proceedings of the Bioinformatics Conference*, pages 388–390. IEEE Computer Society, 2003.

[EKMR06] Stephan Ewen, Holger Kache, Volker Markl, and Vijayshankar Raman. *Progressive Query Optimization for Federated Queries*, pages 847–864. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[Elm00] Ramez Elmasri. *Fundamentals of Database Systems*. Adison-Wesley, 2000.

[F+12] Pedro Ferrera et al. Tuple MapReduce: Beyond Classic MapReduce. In *ICDM*, pages 260–269, 2012.

[FGL+98] Peter Fankhauser, Georges Gardarin, Moricio Lopez, J Munoz, and Anthony Tomasic. Experiences in Federated Databases: From IRO-DB to MIRO-Web. In *VLDB*, pages 655–658, 1998.

[FHK+11] Mike Folk, Gerd Heber, Quincey Koziol, Elena Pourmal, and Dana Robinson. An Overview of the HDF5 Technology Suite and Its Applications. In *EDBT/ICDT*, pages 36–47, 2011.

[FHM05] Michael Franklin, Alon Halevy, and David Maier. From Databases to Dataspaces: a New Abstraction for Information Management. *ACM SIGMOD Record*, 34(4):27–33, 2005.

[FJK96] Michael J. Franklin, Björn Thór Jónsson, and Donald Kossmann. Performance Tradeoffs for Client-server Query Processing. In *Proceedings of the International Conference on Management of Data*, SIGMOD '96, pages 149–160, New York, NY, USA, 1996. ACM.

[FWH08] Daniel P Friedman, Mitchell Wand, and Christopher Thomas Haynes. *Essentials of Programming Languages*. MIT press, third edition, 2008.

[GBI15a] GBIF. Global Biodiversity Information Facility (GBIF). http://www.gbif.org/, 2015. Accessed: 2015-10-22.

[GBI15b] GBIF. Global Fungi Occurrences Dataset, September 2015. DOI: 10.15468/dl.3le67x.

[GBI16] GBIF. Global Fungi Occurrences Dataset, March 2016. DOI: 10.15468/dl.4uc5ad.

[GJSD09] Jürgen Göres, Thomas Jörg, Boris Stumm, and Stefan Dessloch. GEM: A Generic Visualization and Editing Facility for Heterogeneous Metadata. *CSRD*, 24(3):119–135, 2009.

[GKV+08] Jitendra Gaikwad, Varun Khanna, Subramanyam Vemulpad, Joanne Jamie, Jim Kohen, and Shoba Ranganathan. CMKb: a Web-Based Prototype for Integrating Australian Aboriginal Customary Medicinal Plant Knowledge. *BMC bioinformatics*, 9(Suppl 12):S25, 2008.

[GL02] Seth Gilbert and Nancy Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *SIGACT News*, 33(2):51–59, June 2002.

[Gra08] Jim Gray. Technical Perspective: The Polaris Tableau System. *Communications of the ACM*, 51(11):74–74, 2008.

[GSGC08] Owen Gilson, Nuno Silva, Phil W Grant, and Min Chen. From Web Data to Visualization via Ontology Mapping. In *Computer Graphics Forum*, volume 27, pages 959–966. Wiley Online Library, 2008.

[GT12] Casey S Greene and Olga G Troyanskaya. Data-Driven View of Disease Biology. *PLoS Computational Biology*, 8(12):e1002816, 2012.

[GTK98] Antoine Guisan, Jean-Paul Theurillat, and Felix Kienast. Predicting the Potential Distribution of Plant Species in an Alpine Environment. *Journal of Vegetation Science*, 9(1):65–74, 1998.

[GWR11] Jitendra Gaikwad, Peter D Wilson, and Shoba Ranganathan. Ecological Niche Modeling of Customary Medicinal Plant Species used by Australian Aborigines to identify Species-rich and Culturally Valuable Areas for Conservation. *Ecological Modelling*, 222(18):3437–3443, 2011.

[GZ00] Antoine Guisan and Niklaus E Zimmermann. Predictive Habitat Distribution Models in Ecology. *Ecological Modelling*, 135(2):147–186, 2000.

[H+05] Robert J Hijmans et al. Very High Resolution Interpolated Climate Surfaces for Global Land Areas. *International Journal of Climatology*, 25(15):1965–1978, 2005.

[H+11] Bill Howe et al. Database-as-a-Service for Long-Tail Science. In *Scientific and Statistical Database Management*, SSDBM, pages 480–489, 2011.

[Han07] Michael Hanus. Multi-paradigm Declarative Languages. In *ICLP*, volume 4670 of *Lecture Notes in Computer Science*, pages 45–75. Springer, 2007.

[HGB+12] RJ Hijmans, L Guarino, C Bussink, P Mathur, M Cruz, I Barrentes, and E Rojas. DIVA-GIS 5.0. A Geographic Information System for the Analysis of Species Distribution Data. *Versão*, 7:476–486, 2012.

[HJ11] Robin Hecht and Stefan Jablonski. NoSQL Evaluation: A Use-case Oriented Survey. In *Cloud and Service Computing (CSC), 2011 International Conference on*, pages 336–341. IEEE, 2011.

[HKWY97] Laura Haas, Donald Kossmann, Edward Wimmers, and Jun Yang. Optimizing Queries Across Diverse Data Sources. In *23rd VLDB International Conference*, pages 276–285, 1997.

[HM85] Dennis Heimbigner and Dennis McLeod. A Federated Architecture for Information Management. *ACM Transactions on Information Systems (TOIS)*, 3(3):253–278, 1985.

[HRO06] Alon Halevy, Anand Rajaraman, and Joann Ordille. Data Integration: the Teenage Years. In *In Proceedings of the 32nd international conference on very large data bases*, pages 9–16. VLDB Endowment, 2006.

[HS13] Steven Harris and Andy Seaborne. SPARQL 1.1 Query Language. Technical report, W3C, March 2013. http://www.w3.org/TR/2013/REC-sparql11-query-20130321/.

[HTT09a] A.J.G. Hey, S. Tansley, and K.M. Tolle. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research Redmond, WA, 2009.

[HTT09b] Tony Hey, Stewart Tansley, and Kristin M Tolle. Jim Gray on eScience: a Transformed Scientific Method, 2009.

[IAJA11] Stratos Idreos, Ioannis Alagiannis, Ryan Johnson, and Anastasia Ailamaki. Here are my Data Files. Here are my Queries. Where are my Results? In *CIDR*, pages 57–68, 2011.

[IGN+12] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, Sjoerd Mullender, Martin Kersten, et al. MonetDB: Two Decades of Research in Column-Oriented Database Architectures. *A Quarterly Bulletin of the IEEE Computer Society Technical Committee on Database Engineering*, 35(1):40–45, 2012.

[IKM12] Milena Ivanova, Martin Kersten, and Stefan Manegold. Data Vaults: a Symbiosis between Database Technology and Scientific File Repositories. In *SSDM*, pages 485–494, 2012.

[ISO08] ISO. Information Technology — Database Languages — SQL — Part 1: Framework (SQL/Framework). International Organization for Standardization ISO/IEC 9075-1:2008, July 2008. concepts: p:13, definition of query.

[JCE+07] HV Jagadish, Adriane Chapman, Aaron Elkiss, Magesh Jayapandian, Yunyao Li, Arnab Nandi, and Cong Yu. Making Database Systems Usable. In *Proceedings of the International Conference on Management of data*, pages 13–24. ACM, 2007.

[JKV06] T. S. Jayram, Phokion G. Kolaitis, and Erik Vee. The Containment Problem for Real Conjunctive Queries with Inequalities. In *Proceedings of the Twenty-fifth SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '06, pages 80–89, New York, NY, USA, 2006. ACM.

[JMH16] Shrainik Jain, Dominik Moritz, and Bill Howe. High Variety Cloud Databases. In *32nd International Conference on Data Engineering Workshops (ICDEW)*, pages 12–19. IEEE, 2016.

[K+14] Manos Karpathiotakis et al. Adaptive Query Processing on RAW Data. In *40th VLDB*, 2014.

[K+15] Manos Karpathiotakis et al. Just-In-Time Data Virtualization: Lightweight Data Management with ViDa. In *7th CIDR*, 2015.

[KAA16] Manos Karpathiotakis, Ioannis Alagiannis, and Anastasia Ailamaki. Fast Queries over Heterogeneous Data through Engine Customization. *Proceedings of the VLDB Endowment*, 9(12):972–983, 2016.

[KFY+02] Boris Katz, Sue Felshin, Deniz Yuret, Ali Ibrahim, Jimmy Lin, Gregory Marton, Alton Jerome McFarland, and Baris Temelkuran. Omnibase: Uniform Access to Heterogeneous Data for Question Answering. In *Natural Language Processing and Information Systems*, pages 230–234. Springer, 2002.

[KK92] Magdi N Kamel and Nabil N Kamel. Federated Database Management System: Requirements, Issues and Solutions. *Computer Communications*, 15(4):270–278, 1992.

[KKR10] Friederike Klan and Birgitta König-Ries. Enabling Trust-Aware Semantic Web Service Selection a Flexible and Personalized Approach. In *Proceedings of the 12th International Conference on Information Integration and Web-based Applications and Services*, iiWAS '10, pages 83–90, New York, NY, USA, 2010. ACM.

[KM14] John King and Roger Magoulas. Data Science Salary Survey: Tools, Trends, What Pays (and What Doesn't) for Data Professionals. Sebastopol: O'Reilly, 11 2014.

[KOKR15] Pawandeep Kaur, Michael Owonibi, and Birgitta König-Ries. Towards Visualization Recommendation-A Semi-Automated Domain-Specific Learning Approach. In *GvD*, pages 30–35, 2015.

[KOKS15] Verena Kantere, George Orfanoudakis, Anastasios Kementsietsidis, and Timos Sellis. Query Relaxation Across Heterogeneous Data Sources. In *24th ACM*, CIKM, pages 473–482, 2015.

[KR13] Karamjit Kaur and Rinkle Rani. Modeling and Querying Data in NoSQL Databases. In *International Conference on Big Data*, pages 1–7. IEEE, 2013.

[KR14] Friederike Klan and Birgitta König Ries. Recommending Judgment Targets for Rating Provision. In *International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT)*, volume 2, pages 327–334. IEEE/WIC/ACM, 2014.

[KZIN11] M. Kersten, Y. Zhang, M. Ivanova, and N. Nes. SciQL, a Query Language for Science Applications. In *Proceedings of the EDBT/ICDT Workshop on Array Databases*, AD '11, pages 1–12, New York, NY, USA, 2011. ACM.

[L+06] Bertram Ludäscher et al. Managing Scientific Data: From Data Integration to Scientific Workflows. *Geological Society of America Special Papers*, 397:109–129, 2006.

[LCW93] Hongjun Lu, Hock Chuan Chan, and Kwok Kee Wei. A Survey on Usage of SQL. *SIGMOD Rec.*, 22(4):60–65, December 1993.

[Lea10] Neal Leavitt. Will NoSQL Databases Live up to Their Promise? *Computer*, 43(2), 2010.

[Lef12] Jonathan Leffler. BNF Grammar for ISO/IEC 9075-2:2003 - Database Language SQL (SQL-2003) SQL/Foundation. http://savage.net.au/SQL/sql-2003-2.bnf.html, June 2012.

[Len02] Maurizio Lenzerini. Data Integration: A Theoretical Perspective. In *21st SIGMOD symp. on Principles of database systems*, pages 233–246, 2002.

[Lib03] Leonid Libkin. Expressive Power of SQL. *Theoretical Computer Science*, 296(3):379–404, 2003.

[Llo94] John W Lloyd. Practical Advantages of Declarative Programming. In *Joint Conference on Declarative Programming, GULP-PRODE*, volume 94, page 94, 1994.

[LM10] Avinash Lakshman and Prashant Malik. Cassandra: a Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[LMW96] Leonid Libkin, Rona Machlin, and Limsoon Wong. A Query Language for Multidimensional Arrays: Design, Implementation, and Optimization Techniques. In *Proceedings of the International Conference on Management of Data*, volume 25, 2 of *ACM SIGMOD Record*, pages 228–239, New York, NY, USA, June 4–6 1996. ACM Press.

[LP08] Hua-Ming Liao and Guo-Shun Pei. Cache-Based Aggregate Query Shipping: An Efficient Scheme of Distributed OLAP Query Processing. *Journal of Computer Science and Technology*, 23(6):905–915, 2008.

[MH13] ABM Moniruzzaman and Syed Akhter Hossain. NoSQL Database: New Era of Databases for Big Data Analytics-Classification, Characteristics and Comparison. *arXiv preprint arXiv:1307.0191*, 2013.

[Mic] Microsoft Corporation. LINQ: Language-Integrated Query. http://msdn.microsoft.com/en-us/library/bb397926.aspx. Accessed: 2015-9-14.

[Mic12] Microsoft. MS SQL Server 2012 SELECT (Transact SQL). http://msdn.microsoft.com/en-us/library/ms189499.aspx, 2012. Visited Nov. 2013.

[MMP+11] Bruno Marnette, Giansalvatore Mecca, Paolo Papotti, Salvatore Raunich, Donatello Santoro, et al. ++Spicy: an Open-Source Tool for Second-Generation Schema Mapping and Data Exchange. *Clio*, 19:21, 2011.

[MOLMEW17] Alejandra Morán-Ordóñez, José J Lahoz-Monfort, Jane Elith, and Brendan A Wintle. Evaluating 318 Continental-Scale Species Distribution Models Over a 60-year Prediction Horizon: What Factors Influence the Reliability of Predictions? *Global Ecology and Biogeography*, 26(3):371–384, 2017.

[NRSW99] LHRMB Niswonger, M Tork Roth, PM Schwarz, and EL Wimmers. Transforming Heterogeneous Data with Database Middleware: Beyond Integration. *Data Engineering*, 31, 1999.

[O+08] Christopher Olston et al. Pig Latin: A Not-so-Foreign Language for Data Processing. In *ICMD*, pages 1099–1110, 2008.

[OKA+17] Matthaios Olma, Manos Karpathiotakis, Ioannis Alagiannis, Manos Athanassoulis, and Anastasia Ailamaki. Slalom: Coasting Through Raw Data via Adaptive Partitioning and Indexing. *Proceedings of the VLDB Endowment*, 10(10):1106–1117, 2017.

[Ope15] Open Knowledge Foundation. DataHub. http://datahub.io, 10 2015. Accessed: 2015-10-13.

[Ore10] Kai Orend. Analysis and Classification of NoSQL Databases and Evaluation of their Ability to Replace an Object-Relational Persistence Layer. page 100, 2010.

[Pap16] Yannis Papakonstantinou. Polystore Query Rewriting: The Challenges of Variety. In *EDBT/ICDT Workshops*, 2016.

[Par13] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2013.

[PAS06] Steven J Phillips, Robert P Anderson, and Robert E Schapire. Maximum Entropy Modeling of Species Geographic Distributions. *Ecological Modelling*, 190(3):231–259, 2006.

[PF11] Terence Parr and Kathleen Fisher. LL (*): the Foundation of the ANTLR Parser Generator. In *ACM SIGPLAN Notices*, volume 46, pages 425–436, New York, NY, USA, June 2011. ACM.

[PJR+11] Prakash Prabhu, Thomas B. Jablin, Arun Raman, Yun Zhang, Jialu Huang, Hanjun Kim, Nick P. Johnson, Feng Liu, Soumyadeep Ghosh, Stephen Beard, Taewook Oh, Matthew Zoufaly, David Walker, and David I. August. A Survey of the Practice of Computational Science. In ACM, editor, *SC '11 State of the Practice Reports*, pages 19:1–19:12, pub-ACM:adr, 2011. ACM Press.

[QEB+09] Xiaohong Qiu, Jaliya Ekanayake, Scott Beason, Thilina Gunarathne, Geoffrey Fox, Roger Barga, and Dennis Gannon. Cloud Technologies for Bioinformatics Applications. In *Proceedings of the 2Nd Workshop on Many-Task Computing on Grids and Supercomputers*, MTAGS '09, pages 6:1–6:10, New York, NY, USA, 2009. ACM.

[R C13] R Core Team. R: A Language and Environment for Statistical Computing. http://www.R-project.org/, 2013. Accessed: 2015-9-25.

[Ram12] Prakash Ramanan. Rewriting XPath Queries Using Materialized XPath Views. *Computer and System Sciences*, 78(4):1006–1025, July 2012.

[RAvUP16] Andreas Rauber, Ari Asmi, Dieter van Uytvanck, and Stefan Pröll. Identification of Reproducible Subsets for Data Citation, Sharing and Re-Use. *Bulletin of IEEE Technical Committee on Digital Libraries, Special Issue on Data Citation*, 2016.

[RBHS04] Christopher Re, Jim Brinkley, Kevin Hinshaw, and Dan Suciu. Distributed XQuery. In *Workshop on Information Integration on the Web*, pages 116–121, 2004.

[RC95] Louiqa Raschid and Ya-Hui Chang. Interoperable Query Processing from Object to Relational Schemas Based on a Parameterized Canonical Representation. *International Journal of Cooperative Information Systems*, 4(01):81–120, 1995.

[RC13] Florin Rusu and Yu Cheng. A Survey on Array Storage, Query Languages, and Systems. *arXiv preprint arXiv:1302.0103*, 2013.

[RCDS14] Jonathan Robie, Don Chamberlin, Michael Dyck, and John Snelson. XQuery 3.0: An XML Query Language. *Recommendation, W3C*, 2014.

[Red] Redhat. Hibernate ORM. http://hibernate.org/. Accessed: 2015-11-01.

[Rex13] Karl Rexer. 2013 Data Miner Survey. www.RexerAnalytics.com, 2013.

[RG00] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw Hill, 2nd edition, 2000.

[Rod95] John F Roddick. A Survey of Schema Versioning Issues for Database Systems. *Information and Software Technology*, 37(7):383–393, 1995.

[RS70] Daniel J Rosenkrantz and Richard Edwin Stearns. Properties of Deterministic Top-Down Grammars. *Information and Control*, 17(3):226–256, 1970.

[RWE13] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph Databases*. O'Reilly Media, 2013.

[RWE15] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph Databases: New Opportunities for Connected Data*. O'Reilly Media, Inc., 2015.

[Sah02] Arnaud Sahuguet. ubQL: A Distributed Query Language to Program Distributed Query Systems, January 01 2002.

[SB09] A.S. Szalay and J.A. Blakeley. Gray's Laws: Database-Centric Computing in Science. *The fourth paradigm: data-intensive scientific discovery*, pages 5–11, 2009.

[SBPR11] Michael Stonebraker, Paul Brown, Alex Poliakov, and Suchi Raman. The Architecture of SciDB. In *23rd International Conference on SSDBM*, pages 1–16, 2011.

[SC05] Michael Stonebraker and Ugur Cetintemel. " one size fits all": an idea whose time has come and gone. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 2–11. IEEE, 2005.

[SCMMS12] Adam Seering, Philippe Cudre-Mauroux, Samuel Madden, and Michael Stonebraker. Efficient Versioning for Scientific Array Databases. In *ICDE*, pages 1013–1024, 2012.

[SL90] Amit P. Sheth and James A. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM*, 22(3):183–236, 1990.

[SSR+14] Ken Smith, Len Seligman, Arnon Rosenthal, Chris Kurcz, Mary Greer, Catherine Macheret, Michael Sexton, and Adric Eckstein. Big Metadata: The Need for Principled Metadata Management in Big Data Ecosystems. In *Proceedings of Workshop on Data analytics in the Cloud*, pages 1–4. ACM, 2014.

[STG08] Alex Szalay, Ani R. Thakar, and Jim Gray. The sqlLoader Data-Loading Pipeline. *Computing in Science & Engineering*, 10(1):38–48, 2008.

[SW65] Samuel Sanford Shapiro and Martin B Wilk. An Analysis of Variance Test for Normality. *Biometrika*, 52(3/4):591–611, 1965.

[T+09] Ashish Thusoo et al. Hive: a Warehousing Solution Over a Map-Reduce Framework. *VLDB*, 2(2):1626–1629, 2009.

[The13] The Neo4J Team. *The Neo4J Manual*. Neo Technology, v1.9.5 edition, September 2013.

[The16] The Neo4J Team. *The Neo4J Manual*. Neo Technology, v2.3.7 edition, August 2016.

[TRV98] Anthony Tomasic, Louiqa Raschid, and Patrick Valduriez. Scaling Access to Heterogeneous Data Sources with DISCO. *IEEE Transactions on Knowledge and Data Engineering*, 10(5):808–823, 1998.

[TSG04] A Thakar, Alexander S Szalay, and Jim Gray. From FITS to SQL-Loading and Publishing the SDSS Data. In *Astronomical Data Analysis Software and Systems (ADASS) XIII*, volume 314, page 38, 2004.

[TWP16] Masashi Tanaka, Guan Wang, and Yannis P Pitsiladis. Advancing Sports and Exercise Genomics: Moving from Hypothesis-Driven Single Study Approaches to Large Multi-Omics Collaborative Science. *Physiological genomics*, pages physiolgenomics–00009, 2016.

[Uni15] Unidata. Network Common Data Form (NetCDF). http://www.unidata.ucar.edu/software/netcdf/, 3 2015. Version 4.5.5, Accessed: 2015-10-12.

[VBP+13] Nicole A Vasilevsky, Matthew H Brush, Holly Paddock, Laura Ponting, Shreejoy J Tripathy, Gregory M LaRocca, and Melissa A Haendel. On the Reproducibility of Science: Unique Identification of Research Resources in the Biomedical Literature. *PeerJ*, 1:e148, 2013.

[VHS+17] Manasi Vartak, Silu Huang, Tarique Siddiqui, Samuel Madden, and Aditya Parameswaran. Towards Visualization Recommendation Systems. *ACM SIGMOD Record*, 45(4):34–39, 2017.

[VRH04] Peter Van-Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT press, 2004.

[W3C13] W3C SPARQL Working Group. SPARQL 1.1 Overview. W3C, 2013.

[War12] Colin Ware. *Information Visualization: Perception for Design*. Elsevier, 2012.

[Whi12] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2012.

[WT12] Guoxi Wang and Jianfeng Tang. The NoSQL Principles and Basic Application of Cassandra Model. In *International Conference on Computer Science and Service System (CSSS)*, pages 1332–1335. IEEE, 2012.

[YLB+13] Jiangtao Yin, Yong Liao, Mario Baldi, Lixin Gao, and Antonio Nucci. Efficient Analytics on Ordered Datasets Using MapReduce. In *Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC, pages 125–126, 2013.

[Zhu03] Ningning Zhu. Data Versioning Systems. Technical report, Computer Science Dept. State University of New York at Stony Brook, April 11 2003.

[ZKIN11] Ying Zhang, Martin Kersten, Milena Ivanova, and Niels Nes. SciQL: Bridging the Gap Between Science and Relational DBMS. In *Proceedings of the 15th Symposium on International Database Engineering and Applications*, IDEAS '11, pages 124–133, New York, NY, USA, 2011. ACM.

[ZKM13] Ying Zhang, M. L. Kersten, and S. Manegold. SciQL: Array Data Processing Inside an RDBMS. In *ICMD*, pages 1049–1052, 2013.

[ZPF16] Steffen Zeuch, Holger Pirk, and Johann-Christoph Freytag. Non-Invasive Progressive Optimization for In-Memory Databases. *Proceedings of the VLDB Endowment*, 9(14):1659–1670, 2016.

Part V.

Appendix

A

QUIS Grammar

Grammar A.1 QUIS Grammar - (Statements)

1: $process$ ::= $declaration\ statement^{+}$

2: $statement$ ::= $selectStatement$ | $insertStatement$ | $updateStatement$ | $deleteStatement$

3: $selectStatement$ ::= SELECT $setQualifierClause^{?}\ projectionClause^{?}\ sourceSelectionClause\ filterClause^{?}\ orderClause^{?}\ limitClause^{?}\ groupClause^{?}\ targetSelectionClause^{?}$

4: $projectionClause$ ::= USING PERSPECTIVE $identifier$ | USING INLINE $inlineAttribute^{+}$

5: $inlineAttribute$ ::= $expression$(AS $identifier$)$^{?}$

6: $sourceSelectionClause$ ::= FROM $containerRef$

7: $containerRef$ ::= $combinedContainer$ | $singleContainer$ | $variable$ | $staticData$

8: $combinedContainer$ ::= $joinedContainer$ | $unionedContainer$

9: $unionedContainer$ ::= $containerRef$ UNION $containerRef$

10: $joinedContainer$ ::= $containerRef\ joinDescription\ containerRef$ ON $joinKeys$

11: $joinDescription$ ::= INNER JOIN | OUTER JOIN | LEFT OUTER JOIN | RIGHT OUTER JOIN

12: $joinKeys$ ::= $identifier\ joinOperator\ identifier$

13: $joinOperator$ ::= EQ | NOTEQ | GT | GTEQ | LT | LTEQ

14: $filterClause$ ::= WHERE LPAR $expression$ RPAR

15: $orderClause$ ::= ORDER BY $sortSpecification^{+}$

16: $sortSpecification$ ::= $identifier\ sortOrder^{?}\ nullOrder^{?}$

17: $sortOrder$ ::= ASC | DESC

18: $nullOrder$ ::= NULL FIRST | NULL LAST

19: $limitClause$ ::= LIMIT(SKIP = **UNIT**)$^{?}$(TAKE = **UNIT**)$^{?}$

20: $groupClause$ ::= GROUP BY $identifier^{+}$(HAVING LPAR $expression$ RPAR)$^{?}$

21: $targetSelectionClause$ ::= INTO($plot$ | $variable$ | $singleContainer$)

22: $plot$ ::= PLOT$^{?}$ $identifier$ HAXIS:$^{?}$ $identifier$ VAXIS:$^{?}$ $identifier^{+}$ PLOTTYPE:$^{?}$($plotTypes$ | **STRING**) HLABEL:$^{?}$ **STRING** VLABEL:$^{?}$ **STRING** PLOTLABEL:$^{?}$ **STRING**

23: $plotTypes$ ::= LINE | BAR | SCATTER | PIE | GEO

Grammar A.1 QUIS Grammar - (Declarations and Expressions)

1: $declaration$ ::= $perspective^{*}$ $connection^{*}$ $binding^{*}$

2: $perspective$ ::= $identifier$ (EXTENDS **ID**)$^{?}$ $attribute^{+}$

3: $attribute$ ::= $samrtId$ (MAPTO = $expression$)$^{?}$(REVERSEMAP = $expression$)$^{?}$

4: $connection$ ::= $identifier$ $adapter$ $dataSource$(PARAMETERS = $parameter^{+}$)$^{?}$

5: $binding$ ::= $identifier$ CONNECTION = **ID**(SCOPE = $bindingScope^{+}$)$^{?}$
(VERSION = $versionSelector$)$^{?}$

6: $smartId$ ::= **ID** ($\!($: $dataType$)(:: $semanticKey$)$^{?}$)$^{?}$

$expression$ ::= NEGATE $expression$ | $expression$ (MULT | DIV | MOD) $expression$
| $expression$ (PLUS | MINUS) $expression$
| $expression$ (AAND | AOR) $expression$
| $expression$ (EQ | NOTEQ | GT | GTEQ | LT | LTEQ | LIKE) $expression$
| $expression$ IS NOT$^{?}$(NULL | NUMBER | DATE | ALPHA | EMPTY)

7: | NOT $expression$
| $expression$ (AND | OR) $expression$
| $function$
| LPAR $expression$ RPAR
| $value$
| $identifier$

8: $function$ ::= ($identifier$.)$^{?}$ $identifier$ LPAR $argument^{*}$ RPAR

9: $argument$ ::= $expression$

Grammar A.1 `QUIS` Grammar - (Path Expressions)

1: $pathExpression \ ::= \ (path\ attribute^{?}) \mid (path^{?}\ attribute)$

$path \ ::= \ (path\ relation\ path) \mid (path\ relation) \mid (relation\ path)$

2: $\quad \mid (\text{`('}(label\ \text{`:'})^{?}\ cardinality^{?}\ path\ \text{`)'})$

$\quad \mid (step) \mid (relation)$

3: $step \ ::= \ (unnamedEntity \mid (namedEntity\ sequenceSelector^{?}))\ predicate^{*}$

4: $attribute \ ::= \ \text{`@'}(namedAttribute \mid \text{`*'} \mid predicate)$

$relation \ ::= \ forward\_rel \mid backward\_rel \mid non\_directional\_rel$

5: $\quad \mid bi\_directional\_rel$

6: $forward\_rel \ ::= \ \text{`->'} \mid (\text{`-'}\ label\ \text{`:->'}) \mid (\text{`-'}(label\ \text{`:'})^{?}\ taggedScope\ \text{`->'})$

7: $backward\_rel \ ::= \ \text{`<-'}(label\ \text{`:-'} \mid (label\ \text{`:'})^{?}\ taggedScope\ \text{`-'})^{?}$

8: $non\_directional\_rel \ ::= \ \text{`-'}(\text{`-'} \mid label\ \text{`:-'} \mid (label\ \text{`:'})^{?}\ taggedScope\ \text{`-'})$

9: $bi\_directional\_rel \ ::= \ \text{`<-'}(\text{`>'} \mid label\ \text{`:->'} \mid (label\ \text{`:'})^{?}\ taggedScope\ \text{`->'})$

10: $taggedScope \ ::= \ (tag\ (\text{`|'}\ tag)^{*})^{?}\ relationScope$

$relationScope \ ::= \ sequenceSelector\ predicate \mid cardinalitySelector\ predicate$

11: $\quad \mid sequenceSelector \mid cardinalitySelector \mid predicate$

12: $sequenceSelector \ ::= \ \text{`('}\ \textbf{NUMBER}\ \text{`)'}$

13: $predicate \ ::= \ \text{`['}\ expression\ \text{`]'}$

$cardinalitySelector \ ::= \ \text{`\{'}(\textbf{NUMBER}^{?}\ \text{`..'}\ \textbf{NUMBER}^{?})$

14: $\quad \mid \textbf{NUMBER} \mid \text{`*'} \mid \text{`+'} \mid \text{`?'})\ \text{`\}'}$

B

Expressiveness of QUIS's Path Expression

XPath	QUIS	**Description**
/	/	root
.	.	current
/@name	/@name	Selects the name attribute of the current node
@name	@name	Selects "name" attribute
@*	@*	Matches any attribute
/a/b/c/@*	/–a–b–c@*	selects all the attributes for all c elements in the path: root/a/b/c.
/a/b/c/@name	/–a–b–c@name	selects the $name$ attributes for all c elements in the path: root/a/b/c
/a//b	/–a-{*}-b	Selects all b elements having any distance to an a, starting from the root
a//b	a-{*}-b	Selects all b elements having any distance to an a. a can be anywhere. Also can be written as a-{..}-b
//a	a	Selects all a elements no matter where
x:a	(x:a)	Namespace or label for a. QUIS supports labels on relations, too, e.g., a-(x:)-b
*[@name]	a[@name]	All elements with "name" attribute
a/*	a<-CHILD-x	All the x elements that are direct children of a
a[@name="Tom"] /[0..3] //b[@livesAbroad]	a(@name="Tom")-CHILD{..3}->x-{*}->b(@livesAbroad)	Decedents of the Tom's maximum three levels ancestors that live abroad
store/-book[1]/@title	store–book(1)@title	The title of the first book in the bookstore
store/book[1]/title	store–book(1)–title	The title of the first book in the bookstore. Title is an element. Sequence applies before relation
store/+title	store-{+}-title	Relation of one or more level deep. Title is an element.
store/[0..1]title	store-{?}-title	Relation of maximum depth of one
store/-books[title="t1"]	store–books(title="t1")	Title is an element which its text is evaluated by the predicate
store/-books[@title="t1"]	store–books(@title="t1")	Title is an attribute which its value is evaluated by the predicate
author[@first-name][3]	author[@first-name][3]	Applying multiple predicates on an element

Table B.1.: QUIS path expression coverage for XPath

Cypher	QUIS	**Description**
(a)->(b)	a->b	A directed relation of length 1 from node a to node b
(a)->()->(b)	a-{2}->b	A relation between a and b with one node in between
a->b->()	a->b->	The path ends in a relation
(a)–>(b)–>(c)->(a)	a->b->c->a	Cycles
(a)–>(b)<–(c)	a->b<-c	A sub-tree rooted in b
(a)-[:RELTYPE]->(b)	a-RELTYPE->b	Direct relation from a to b of type $RELTYPE$
(a)-[r:TYPE1,TYPE2]->(b)	a-r:TYPE1|TYPE2->b	Direct relation from a to b of type $TYPE1$ or $TYPE2$ so that the matched relations are named r
(a)-[*2]-(a)-[*3..5]->(b)->(b)	a-{..2}-a-{3..5}->b->b	A loop of maximum length 2 over a followed by a chain of 3-5 nodes to b and then followed by a direct loop over b
(a:User)–>(b)	User:a->b	Setting a qualifier for the an element
(a:Person name:“Alice”)	Person:a(@name = “Alice”)	Elements of type $Person$ that their name attribute has the value of “Alice”
(a:Person name:“Alice”)-[:ACTEDIN]-(m:Movie)	Person:a (@name =“Alice”)-ACTEDIN-> Movie:m	All the movies Alice acted in
(a:Person name:“Alice”)-[:ACTEDIN{year>2015}]-(m:Movie)	Person:a (@name =“Alice”)-ACTEDIN(@year>2015)-> Movie:m	All the movies Alice acted in after year 2015
Not available	a->(r:{2..5}b->c)->d	Sub pattern repetition and identification. After visiting an a, the sub pattern named r will be matched 2 to 5 times, then a tailing d is expected.
()-[{disabled:TRUE}]->()	-(disabled)->	All the forward relations that have an attribute named “disabled”
(h1:Hydrogen)-[BIND]->(o:Oxygen)-[BIND]->(h2:Hydrogen)	Hyrogen:h1-BIND-Oxygen:o-BIND-Hyrogen:h2	H2O formula

Table B.2.: QUIS path expression coverage for Cypher

C

Evaluation Materials for the User Study

C.1 User Study Methods

The goal of the user study is to measure a set of indicators on a baseline system as well as on QUIS and to test whether QUIS has made a meaningful improvement to any of those indicators. Based on its popularity among data scientists, we chose the R system as the baseline, meaning that it is the representative of the current situation. We then introduce QUIS as an alternative to the baseline in order to be able to measure the effectiveness of the introduced change. To determine whether the observed changes are meaningful, we design and perform a *paired-samples t-test*. In short, a t-test is an inferential statistics that checks if two means are reliably different from each other and if any differences can be generalized from the samples to the population. A paired t-test is used to compare two population means in which there are two samples; the observations made in one sample can be paired with observations from the other.

The chosen indicators are as following:

1. Time-on-task (TT): The total human time spent on the assigned task, from the beginning to the presentation of the results. This is mainly the coding time. It is measured in minutes by the subject, using a wall clock;

2. Machine time (MT): The execution time observed by end-user to perform the task and present the result by running the subject's submitted code on a reference machine. It is measured in seconds;

3. Code complexity (CC): The average number of tokens used per line of code, no matter which and how many languages are used;

4. Ease of use (EU): Also known as usability, it is the degree to which software can be used by specific consumers. This value is obtained using the survey results and statistical techniques. More specifically, it is the mean value of the answers of each subject to questions 1-7 in the usability section of the survey questionnaire (see Appendix C.4);

5. Usefulness (UF): Indicates to what extent and how well the system addresses the users' use-cases. For each subject, it is the mean value of the answers to questions 8-13 in the usability section of the survey questionnaire (see Appendix C.4); and

6. Satisfaction (SF): This is an indicator that shows to what extent a user enjoys using the system. For each subject, it is the mean value of the answers to questions 14-19 in the usability questions section of the survey's questionnaire (see Appendix C.4).

For each indicator I_i, we design a null hypothesis $H0_i : \bar{x}_{b_i} \neq \bar{x}_{c_i} \rightarrow \mu_{b_i} = \mu_{c_i}$, in which b, and c are the baseline and the introduced change, respectively. The hypothesis declares that having different sample means does not lead to a meaningful difference in the population mean. In other words, each $H0_i$ claims that the change introduced does not meaningfully improve I_i. Our desired outcome would be that the study rejects these hypotheses.

In order to perform the data analysis task (see Appendix C.2), we chose a group of volunteer subjects from the iDiv[1] and BExIS[2] projects, as well as students from the FUSION[3] chair. We chose subjects based on their willingness to participate, provided that they are generally familiar with data processing tools as well as R and SQL. The subjects chosen vary in multiple dimensions, including culture, nationality, level of education, field of study, and gender. Each subject performed the given task in two runs: one on the baseline and another on QUIS. Therefore, the samples are paired. In order to eliminate the bias effect of learning from the first run, we randomly divided the subjects into two groups, G_q and G_b. While the members of G_q began with the QUIS run, the G_b members were asked to start with the baseline run. In order to maintain pair independency, we ensured that the subjects a) were not aware of the test beforehand and b) did not exchange relevant information during the test runs. In both runs, we observed/measured the indicators and collected answers to the questionnaire. We use the outcome of the two runs to prove that the change is meaningful and can be generalized.

C.2 Task Specification of the User Study

[1] https://www.idiv.de/en.html
[2] http://bexis2.uni-jena.de/
[3] http://fusion.cs.uni-jena.de

Dear survey participant,

Subject ID:

Thank you for taking part in this survey. Please read this document carefully and follow the instructions provided.

You are given a task to be performed in two scenarios. During (and following the completion of) the evaluation, you will be asked to answer some questions. Please perform the task, record the requested information, and return the results. Use the subject identifier (Subject ID) assigned to you on all of the material (see Section 4) that you return.

Please record current time in (hh:mm) format:

1. Task Specification

You are given a dataset (see Section 7) that contains meteorological data, the airports that the meteorological stations were located at, and the geographical locations of the airports. You are required to perform the following task on the dataset and record the requested measures:

Compute the max, min, and average temperature per airport over the full time span of the records. For each station join the computed variables with the respective airport's name and location. Return the result set as a CSV as explained in Section 5. Be aware that some records may have missing values.

The following is an example of the expected result:

```
stationid,code,name,latitude,longitude,elevation,max_temperature,min_temperature,avg_temperature
1,DSM,DES MOINES,41.53395,-93.65311,294.0,4,37.8,-26.7,10.756333926981313
2,DBQ,DUBUQUE,42.39835,-90.70914,329.0,5,33.3,-34.4,8.671347375750598
3,IOW,IOWA CITY,41.63939,-91.54454,204.0,7,36.7,-31.1,10.12492741971506
```

2. Task Runs

You are required to run the task in two configurations. Both of the runs should be performed in the R system, but you will be required to utilize a different set of R packages from each run. The runs are named "Baseline" and "QUIS".

ATTENTION: If your Subject ID starts with the letter "Q", perform the QUIS run first; otherwise perform the Baseline first.

1. Baseline run: Develop an R solution that generates the requested output using the provided dataset. You are free to use any R standard command, as well as packages found on CRAN. Do not use the RQUIS package for this run.
2. QUIS run: Develop an R solution that generates the requested output using the given dataset. All the data access aspects of the task must utilize the RQUIS package. Additionally, you are allowed to utilize other R packages.

For each run, there is a moderator that can help you in overcoming technical issues or explaining the task, if required. However, they do not guide you towards any solution.

A short QUIS tutorial[1] and a tool setup document are available online[2].

3. Time Allocation

1. Developing a solution for each run should take no longer than 45 minutes
2. Before starting each run, record its starting time in (hh:mm) format in the following table. Record the moment you started reading the task specification.
3. After finishing each run, record its finish time in (hh:mm) format in the following table. Use the timestamp recorded in your latest log files or tool output.
4. You will need to copy this information to the survey form later.
5. Record the start and finish times, regardless of whether the task was executed successfully or not. If you made multiple attempts, record the overall end-to-end duration.

[1] http://fusion.cs.uni-jena.de/javad/quis/latest/docs/QUIS_Tutorial.pptx
[2] http://fusion.cs.uni-jena.de/javad/quis/latest/docs/ToolsSetup.pdf

	Start Time	Finish Time
Baseline Run		
QUIS Run		

4. Task Results

When you are done, you will be given a survey form. The survey form (the questionnaire) can be downloaded from http://fusion.cs.uni-jena.de/javad/eval/. It is provided in both MS Word and PDF formats.

Please first transfer the task runs' start/finish times to the survey form and then answer the questions carefully. Return the following items to the moderators or zip and send them to this email address:
javad.chamanara@uni-jena.de

1. This task sheet
2. The completed survey form
3. For the baseline run:
    a. The script file. Please name the file as <Subject ID>.r.R
    b. The result set file. Please name the file as <Subject ID>.r.csv
4. For the QUIS run:
    a. The script file. Please name the file as <Subject ID>.quis.R
    b. The result set file. Please name the file as <Subject ID>.quis.csv

5. Measurement of the Success

The task consists of many steps; five among them are of particular interest. The following table explains these steps and their contributions to this final success. Using this table, you can calculate your success as a percentage between 0 and 100 and enter it to the table. Later, you will be asked to enter these success rates on the survey form.

Step	Percentage	Baseline	QUIS
Obtaining weather data and calculating the aggregates requested	20%		
Obtaining airport information	15%		
Obtaining airports' locations	15%		
Putting all of the data together and creating the result set in R	30%		
Writing the result set to a CSV file	20%		

6. Result Format

The result sets of the both runs must be stored in individual CSV files. The header record of the CSV file is as follows:

Column Name	Data Type	Description
stationID	Integer	The identifier of the station
code	String	The 3 letter international identifier of the airport
name	String	The name of the airport
latitude	Double	The airport's latitude
longitude	Double	The airport's longitude
Elevation	Double	The airport's elevation
max_temperature	Double	Maximum temperature at the station with stationID
min_temperature	Double	Minimum temperature at the station with stationID
avg_temperature	Double	Average temperature at the station with stationID

- All the column names are case-insensitive.
- Data types are guidelines only. You can use other data types if needed.

7. Task Data

The dataset that you are going to use is a collection of three related data sources that contain information about environmental data measured by a range of stations spread over a set of airports. As you can see, each data source is stored and accessed differently.

c. **Weather Records:** 5 years of 1-hour resolution weather records collected from the meteorological station at approximately 200 different airports. Each record contains temperature (°C), humidity (%), wind speed (m/s), timestamp (date/time), and the station identifier. All data items are measured in SI. This data is stored in a remote PostgreSQL database table named "*airportsweatherdata*". The table schema is as follows:

Table 1: All the column names are in lower case

Column Name	Data Type	Description
stationed	Integer	The identifier of the station
date	character varying	The timestamp of the record
temperature	Double	
dew	Double	
humidity	Double	
windspeed	Double	

In order to connect to the database, use the following information:

i. **Host:** bx2train.inf-bb.uni-jena.de
ii. **Port:** 5432
iii. **User:** postgres
iv. Password: 1
v. Database: quisEval
vi. Table: *airportsweatherdata*

d. **Airport Information:** A list of airports containing ids, codes, and names. The airport information is stored in a comma separated CSV file named *airports.csv*. The first line of the file is the column names. http://fusion.cs.uni-jena.de/javad/eval/data/airports.csv

e. **Airport Location:** An MS Excel file that contains the geographical locations of the airports. It contains one record per airport, so that each record consists of the station id, latitude (°), longitude (°), and elevation (m). The file name is *airportLocations.xlsx* and the location data is stored in a sheet named "locations". It is the first sheet of the MS Excel workbook. http://fusion.cs.uni-jena.de/javad/eval/data/airportLocations.xlsx

8. Auxiliary Resources

There are a number of R and QUIS examples available at the following URLs:

- R scripts to obtain data from a PostgreSQL: http://fusion.cs.uni-jena.de/javad/eval/PostgreSQLSample.R
- QUIS examples to be used within the workbench: http://fusion.cs.uni-jena.de/javad/quis/latest/examples.zip

9. QUIS Querying Hints

Please keep in mind that QUIS and its R package, RQUIS, are prototypes implemented as a proof of concept of a work that is currently under development. As such, they may not support all expected scenarios, fail unexpectedly, or present technical error messages. The following is a list of hints that may help you in successfully accomplishing the task:

- MS Excel returns integer data as float when queried via its APIs. The "stationed" in the Excel file needs to be declared as "Real" so that QUIS is able to use it in join operations. You can do this by either of the following ways:
  - Declare an inline schema inside your QUIS queries and set the data type of "stationed" as Real.
  - Enhance the Excel sheet headers with required data type. QUIS can parse <columnName>:<datatype> pattern (e.g., stationed:Real) in headers. If you do so for one column, do so for all.
  - Define an external header file and declare the column names and their data types there. Use a comma separated list of <columnName>:<datatype>. The header file name is <ExcalFileName>.<ExcelFileExtension>.<SheetName>.hdr
- There are some reserved words (e.g., class, long, integer, and char) that you cannot use to name the columns. If used, QUIS will attempt to add a "qs_" prefix to them and run the query. Usually, no problems results, but, if they do, consider changing the column names.
- When joining data, duplicate columns names should be prefixed with an "R_". If you join the result of a join to another data that contains same names, it is possible to have more than one identical column name that starts with "R_".

C.3 User Study's Task Data

The working dataset is composed of the following data containers:

1. **Weather records:** 5 years of 1-minute resolution weather records collected from the meteorological stations at 100 of the country's airports. Each record contains temperature data concerning (°C), humidity (%), wind speed (m/s) and direction (degree), timestamp (date/time), and the station identifier, which is the three-letter airport code. This data is stored in a remote PostgreSQL database named `AirportsWeatherData`. All data items are measured in SI;

2. **Airport Information:** A complete list of all of the airports' names published by IATA[4]. It provides the code and name of each airport, in addition to the cities and the countries in which they are located. The airport information is stored in a comma-separated CSV file, in which the first line contains the column names; and

3. **Airport location:** An MS Excel file that contains the geographical locations of the airports. It contains one record per airport, with each record consisting of the airport code, longitude (°), latitude (°), and elevation (m).

C.4 User Study's Questionnaire

[4] `http://www.iata.org`

PPOST-STUDY USABILITY SURVEY

Subject ID:

1. Task Execution

Over the course of this survey, you have performed a data analysis task using two different tool families. Please provide information on your work below. If you are not sure or do not want to answer, please mark the "N/A" option.

Task Completion	R		QUIS	
		N/A		N/A
1. In which order did you run the task? (enter 1 for the tool you used first, and 2 for the other tool)				
2. How many languages/tools/packages did you use to accomplish the task? (count of all the tools, languages, and packages used in the task implementation)				
3. Task Start Time (Copy the task start time from the task sheet. use the hh:mm format)				
4. Task Finish Time (Copy the task finishing time from the task sheet. use the hh:mm format)				
5. How long did you work on the task? (task finish time – task start time in minutes)				
6. How long did the execution of the task take on your machine? (seconds)				
7. To what extent were you able to complete the task? (a percentage between 0 and 100, use the task specification Section 6 as guideline)				
8. How often did you use external help to accomplish the task? (0 for no use, 5 for using external help for each and every line of the code)				

2. Expertise Questions

	Your Answer	NA
1. Which operating system do you regularly use? (e.g., Linux, Windows, Mac. Choose one)		
2. Which operating system did do you run the task on? (e.g., Linux, Windows, Mac. Choose one)		
3. How many languages/tools do you usually use to accomplish similar tasks? (1, 2-4, 5-8, more)		
4. How many programming languages do you know? (Only those languages that you actually use)		
5. How proficient are you in SQL or other query languages (like SPARQL)? (never used before, beginner, intermediate, expert)		
6. How proficient are you in R? (never used before, beginner, intermediate, expert)		
7. What programming language did you typically use to analyze data in the past year?		
8. What size datasets did you typically analyze in the past year? (number of records: 1-10K, 10-100K, 100K-10M, 10M-100M, more)		
9. How often in the past year have you participated in data analysis tasks? (weekly, monthly, a few times, once)		

3. Demography Questions

Age:	Current education Level (If you are a student): (Bachelor, Diploma, Master, Ph.D., Post Doc.)	Highest level of education attained: (Bachelor, Diploma, Master, Ph.D., Post Doc.)
Gender:	Field of Study:	Current job title (If you are working):
Country of Origin:	Fluency in English (Intermediate, Advanced, Native)	Does your job involve data analysis? (Everyday, Few hours a week, Few hours a month, Few hours a year, No use)

Subject ID:

4. Usability Questions

Please rate the following statements in regard to your personal experience with the task. Circle the item (-2, -1, 0, 1, 2) that best fits your judgment of the statement's quality.
Possible ratings range from "strongly disagree" (SD) over neutral to "strongly agree" (SA). Please keep in mind that the distances between any two consecutive items are considered equal.
If you do not want to rate a particular statement, please mark the "N/A" option on its relevant row.

Survey Item	R						QUIS					
	SD				SA	N/A	SD				SA	N/A
Ease of Use												
1. I could easily use the system.	-2	-1	0	1	2		-2	-1	0	1	2	
2. I could easily integrate data from heterogeneous sources.	-2	-1	0	1	2		-2	-1	0	1	2	
3. I could easily aggregate data.	-2	-1	0	1	2		-2	-1	0	1	2	
4. I could easily load and work with big datasets.	-2	-1	0	1	2		-2	-1	0	1	2	
5. It required the fewest steps possible to accomplish what I wanted to do with the system.	-2	-1	0	1	2		-2	-1	0	1	2	
6. I think both occasional and regular users would like the system.	-2	-1	0	1	2		-2	-1	0	1	2	
7. It was easy to write SQL-like queries from inside the R system.	-2	-1	0	1	2		-2	-1	0	1	2	
Usefulness												
8. Using the system increases my productivity on the job. (It helped me do things more quickly)	-2	-1	0	1	2		-2	-1	0	1	2	
9. Using the system enhances my effectiveness on the job. (Using the system helped me to do things in the right way)	-2	-1	0	1	2		-2	-1	0	1	2	
10. The system makes the things I wanted to accomplish, easier to get done.	-2	-1	0	1	2		-2	-1	0	1	2	
11. The system covers all my data access requirements.	-2	-1	0	1	2		-2	-1	0	1	2	
12. The system covers all the aggregate and non-aggregate functions I needed.	-2	-1	0	1	2		-2	-1	0	1	2	
13. The system is useful to me.	-2	-1	0	1	2		-2	-1	0	1	2	
Satisfaction												
14. The system is fast and responsive on different sizes of data.	-2	-1	0	1	2		-2	-1	0	1	2	
15. The system is fast and responsive on different combinations of data sources.	-2	-1	0	1	2		-2	-1	0	1	2	
16. Comparing to my usual work, the system reduces the number of languages/tools I need to accomplish my tasks.	-2	-1	0	1	2		-2	-1	0	1	2	
17. I would use the system in other analysis tasks as well.	-2	-1	0	1	2		-2	-1	0	1	2	
18. I would recommend the system to a friend.	-2	-1	0	1	2		-2	-1	0	1	2	
19. I am satisfied with the system.	-2	-1	0	1	2		-2	-1	0	1	2	

C.5 User Study's Raw Data

Subject	Usability Questions																		Measurements		
	Ease of Use							Usefulness						Satisfaction							
	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q1	Q2	Q3	Q4	Q5	Q6	Q1	Q2	Q3	Q4	Q5	Q6	TT	MT	CC
S1	1	1	1	2	1	1	0	-1	-2	-1	1	1	-1	1	1	-1	-1	0	0	50	17	3.3
S2	1	1	1	-1	0	0	0	0	0	1	0	0	0	-1	0	-1	1	0	0	45	18	3.6
S3	0	-1	1	1	1	0	1	-2	-2	-2	-2	-2	-2	1	1	1	2	2	2	48	16	3.81
S4	0	0	-1	0	-1	2	-1	2	0	0	0	-1	-1	1	1	-2	2	1	1	40	10	3.8
S5	-2	-2	-2	-1	-2	-2	-2	-1	-1	0	0	0	1	1	1	0	1	1	1	50	15	4.1
S6	1	1	-2	-2	-2	-2	-2	2	2	2	2	2	2	-1	-1	1	1	1	1	48	17	4.5
S7	0	-2	2	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	29	12	4.1
S8	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	60	14	3.84
S9	-2	-2	-2	1	-2	-1	-2	-2	-2	-1	-1	-1	-1	1	1	1	1	1	1	48	16	4.31
S10	1	0	1	-1	-1	-1	0	0	0	0	0	0	0	-1	-1	-1	-1	1	1	40	15	3.43
S11	1	1	1	-1	-2	-1	0	0	0	-1	-1	1	0	-2	0	0	0	0	1	50	13	3.64
S12	0	-1	-1	-1	0	-2	-2	-2	-2	-2	-2	-2	-2	1	1	1	1	1	1	48	14	4.3
S13	-1	-2	-1	0	0	-2	-2	-1	-1	0	0	-1	-1	1	1	1	1	1	1	55	13	4.41
S14	2	2	1	-2	1	1	0	1	1	1	1	1	1	-2	-2	-1	0	0	0	56	15	4.22
S15	-2	-2	-1	-1	-1	-1	-2	-2	-2	-1	-1	-1	-1	-1	-1	-1	0	1	1	50	15	4.5
S16	-2	-2	-1	-2	-2	-1	-1	0	0	-2	-1	1	1	-2	0	0	0	1	1	55	16	3.2
S17	0	1	-1	-1	-1	-1	-1	-2	-2	-1	-1	-1	-1	-1	-1	0	1	2	2	56	15	4.5
S18	-2	-2	-2	-1	-1	-1	-1	0	0	1	1	2	2	2	2	1	1	2	2	48	16	4.45
S19	0	0	-1	-1	-1	0	0	-2	-2	-2	-2	-2	-2	1	2	2	2	2	2	40	18	3.9
S20	-1	0	-1	0	-1	-1	0	-1	-1	0	0	1	1	2	1	1	2	2	2	55	15	3.62
S21	1	1	-1	-2	-2	-2	-1	0	0	0	0	1	1	1	1	1	1	1	1	40	15	3.81
S22	1	2	2	1	1	2	1	1	1	1	2	2	2	1	2	1	0	1	1	60	13	3.97
S23	-1	-1	-2	-2	-2	-2	-2	-1	-1	-1	0	-1	-2	-1	-1	0	0	1	0	50	16	4.48
S24	-1	-1	-1	-1	-1	-1	-1	-2	-2	-2	-2	-2	-2	0	0	-1	-1	0	0	60	14	3.81
S25	0	0	-1	-1	0	0	0	-1	-1	1	1	0	0	1	1	0	1	1	0	55	18	4.11
S26	-1	-1	0	0	0	0	-1	-1	-1	0	0	-1	-1	-1	0	1	1	1	1	55	16	3.83
S27	0	0	-1	-1	0	0	1	-2	-2	-2	-1	-1	-2	0	0	0	1	1	0	60	14	4.4
S28	-1	0	0	0	-1	0	1	2	2	1	1	1	1	2	2	1	1	2	1	50	17	4.21
S29	2	2	1	1	1	1	1	1	1	0	0	1	1	1	1	1	1	1	1	45	17	3.3
S30	-1	-1	0	0	1	1	-2	1	1	1	1	1	1	1	1	1	1	1	1	45	18	3.06
S31	-2	-1	1	1	-1	-1	-1	0	1	0	1	1	1	1	1	0	0	1	1	45	18	3.3
S32	0	0	0	0	-1	-1	-2	1	0	-1	-1	1	1	1	0	0	1	1	1	50	17	4.1

Table C.1.: Survey raw data for the baseline system

Subject	Usability Questions																		Measurements		
	Ease of Use							Usefulness						Satisfaction							
	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q1	Q2	Q3	Q4	Q5	Q6	Q1	Q2	Q3	Q4	Q5	Q6	TT	MT	CC
S1	1	1	1	0	1	0	2	1	1	2	2	2	2	0	1	1	1	1	1	23	12	2.50
S2	0	1	1	2	0	-1	0	0	0	1	0	0	0	0	0	-1	0	0	0	40	10	2.64
S3	1	1	1	0	0	1	1	2	2	2	2	2	2	1	2	1	1	1	2	30	13	2.93
S4	-1	1	-1	-1	1	-1	0	1	1	2	2	2	2	1	-1	0	1	1	1	25	10	3.04
S5	-2	-2	-2	-2	-2	-2	1	1	1	1	1	2	2	-2	-2	1	1	0	0	40	8	3.17
S6	-2	-2	-2	0	1	1	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	45	22	3.12
S7	1	2	1	2	0	2	2	1	1	1	1	1	2	2	2	2	2	1	1	45	8	3.16
S8	-2	-2	-2	-2	-2	-2	-1	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	30	15	3.44
S9	-2	-2	-1	-1	-1	-1	-1	1	1	1	1	2	2	-1	-1	-1	-1	-1	-1	20	8	3.21
S10	-1	-1	-1	-1	-1	-1	-1	1	1	1	1	2	2	-1	-1	-1	-1	-1	-1	35	12	3.25
S11	0	-1	-1	0	2	-1	1	1	1	0	-1	-1	-1	-1	0	1	-1	1	1	45	17	3.27
S12	0	-1	-1	-1	-1	-1	-1	1	1	1	1	2	2	-1	-1	-1	-1	-1	-1	30	11	3.11
S13	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	2	2	-2	-1	-1	0	1	1	40	16	3.36
S14	1	1	1	1	1	1	1	0	0	2	2	2	2	-1	-1	-1	-1	-1	-1	40	13	3.36
S15	0	-1	-1	-1	-1	-2	-2	2	2	2	2	2	2	2	2	0	0	1	1	30	13	3.37
S16	-2	-1	-1	-1	-1	-1	-1	1	1	2	1	1	1	1	1	1	1	1	1	40	8	3.40
S17	-1	-1	-1	-1	0	0	-1	1	1	1	1	1	1	2	1	1	2	2	2	45	10	3.45
S18	-2	-2	-1	-2	-2	-1	-2	0	0	0	0	0	0	2	1	1	0	0	0	40	14	3.45
S19	-1	-1	-1	-1	-1	-1	-1	-2	-2	-2	-2	0	0	1	2	1	0	1	1	50	13	3.46
S20	0	0	-1	0	0	0	0	-2	-2	0	0	0	0	-1	-1	0	0	-2	-2	50	17	3.52
S21	0	0	-1	0	-1	0	0	-1	-1	0	0	0	0	0	0	-1	-1	0	0	40	14	3.55
S22	1	2	2	2	2	2	2	-2	-2	1	0	0	0	1	1	0	1	1	0	35	10	3.56
S23	1	1	0	1	0	1	1	0	0	2	0	0	0	-1	0	1	1	1	1	40	12	3.57
S24	1	1	0	0	0	1	1	0	0	0	1	1	1	0	0	0	1	1	0	35	18	3.62
S25	0	0	0	1	-1	-2	-2	0	0	0	-1	-1	-1	2	2	1	1	2	1	40	19	3.78
S26	-1	-1	-1	-1	-1	-1	-1	-2	-2	-2	0	1	1	-2	-2	-1	-1	-1	-1	40	10	3.78
S27	0	0	0	0	0	0	0	-2	-2	-2	-1	1	1	1	1	1	1	1	1	40	10	3.80
S28	0	0	0	0	0	0	0	2	2	2	1	2	2	1	1	0	0	1	1	30	8	3.80
S29	1	0	0	0	0	0	0	2	2	1	1	1	1	1	0	0	1	1	1	35	16	3.45
S30	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	35	8	3.64
S31	0	1	0	1	0	1	0	-1	-1	0	0	0	0	0	0	1	1	1	1	35	15	2.78
S32	1	1	0	0	0	1	1	0	0	0	0	0	0	1	1	0	0	1	0	40	18	2.65

Table C.2.: Survey raw data for the QUIS system

C.6 Descriptive Statistics of the User Study Results

Descriptive statistics of the survey data												
Subjects	R						Q					
	TT	MT	CC	EU	UF	SF	TT	MT	CC	EU	UF	SF
s1	50.00	17.00	3.30	1.00	-0.50	0.00	23.00	12.00	2.50	0.86	1.67	0.83
s2	45.00	18.00	3.60	0.29	0.17	-0.17	40.00	10.00	2.64	0.43	0.17	-0.17
s3	48.00	16.00	3.81	0.43	-2.00	1.50	30.00	13.00	2.93	0.71	2.00	1.33
s4	40.00	10.00	3.80	-0.14	0.00	0.67	25.00	10.00	3.04	-0.29	1.67	0.50
s5	50.00	15.00	4.10	-1.86	-0.17	0.83	40.00	8.00	3.17	-1.57	1.33	-0.33
s6	48.00	17.00	4.50	-1.14	2.00	0.33	45.00	22.00	3.12	-0.86	-2.00	-2.00
s7	29.00	12.00	4.10	1.00	2.00	2.00	45.00	8.00	3.16	1.43	1.17	1.67
s8	60.00	14.00	3.84	-2.00	-2.00	-2.00	30.00	15.00	3.44	-1.86	-2.00	-2.00
s9	48.00	16.00	4.31	-1.43	-1.33	1.00	20.00	8.00	3.21	-1.29	1.33	-1.00
s10	40.00	15.00	3.43	-0.14	0.00	-0.33	35.00	12.00	3.25	-1.00	1.33	-1.00
s11	50.00	13.00	3.64	-0.14	-0.17	-0.17	45.00	17.00	3.27	0.00	-0.17	0.17
s12	48.00	14.00	4.30	-1.00	-2.00	1.00	30.00	11.00	3.11	-0.86	1.33	-1.00
s13	55.00	13.00	4.41	-1.14	-0.67	1.00	40.00	16.00	3.36	-1.00	0.00	-0.33
s14	56.00	15.00	4.22	0.71	1.00	-0.83	40.00	13.00	3.36	1.00	1.33	-1.00
s15	50.00	15.00	4.50	-1.43	-1.33	-0.17	30.00	13.00	3.37	-1.14	2.00	1.00
s16	55.00	16.00	3.20	-1.57	-0.17	0.00	40.00	8.00	3.40	-1.14	1.17	1.00
s17	56.00	15.00	4.50	-0.57	-1.33	0.50	45.00	10.00	3.45	-0.71	1.00	1.67
s18	48.00	16.00	4.45	-1.43	1.00	1.67	40.00	14.00	3.45	-1.71	0.00	0.67
s19	40.00	18.00	3.90	-0.43	-2.00	1.83	50.00	13.00	3.46	-1.00	-1.33	1.00
s20	55.00	15.00	3.62	-0.57	0.00	1.67	50.00	17.00	3.52	-0.14	-0.67	-1.00
s21	40.00	15.00	3.81	-0.86	0.33	1.00	40.00	14.00	3.55	-0.29	-0.33	-0.33
s22	60.00	13.00	3.97	1.43	1.50	1.00	35.00	10.00	3.56	1.86	-0.50	0.67
s23	50.00	16.00	4.48	-1.71	-1.00	-0.17	40.00	12.00	3.57	0.71	0.33	0.50
s24	60.00	14.00	3.81	-1.00	-2.00	-0.33	35.00	18.00	3.62	0.57	0.50	0.33
s25	55.00	18.00	4.11	-0.29	0.00	0.67	40.00	19.00	3.78	-0.57	-0.50	1.50
s26	55.00	16.00	3.83	-0.43	-0.67	0.50	40.00	10.00	3.78	-1.00	-0.67	-1.33
s27	60.00	14.00	4.40	-0.14	-1.67	0.33	40.00	10.00	3.80	0.00	-0.83	1.00
s28	50.00	17.00	4.21	-0.14	1.33	1.50	30.00	8.00	3.80	0.00	1.83	0.67
s29	45.00	17.00	3.30	1.29	0.67	1.00	35.00	16.00	3.45	0.14	1.33	0.67
s30	45.00	18.00	3.06	-0.29	1.00	1.00	35.00	8.00	3.64	0.57	1.00	0.00
s31	45.00	18.00	3.30	-0.57	0.67	0.67	35.00	15.00	2.78	0.43	-0.33	0.67
s32	50.00	17.00	4.10	-0.57	0.17	0.67	40.00	18.00	2.65	0.57	0.00	0.50
Mean	49.563	15.406	3.935	-0.464	-0.224	0.568	37.125	12.750	3.318	-0.223	0.411	0.151
Median	50.000	15.500	3.935	-0.500	-0.083	0.667	40.000	12.500	3.386	-0.214	0.417	0.500
SD	7.075	1.932	0.427	0.913	1.202	0.849	7.183	3.750	0.342	0.944	1.127	1.013
Var	50.060	3.733	0.183	0.833	1.446	0.720	51.597	14.065	0.117	0.891	1.271	1.026

Table C.3.: Descriptive statistics of the survey data

C.7 Analytic Statistics of the User Study Results

Table C.4 summarizes the analytics of the user study. It includes the survey result for each indicator as well as their normality test, significance, null hypothesis test result, and the difference of the means.

I	sc	$\bar{x}$	$\tilde{x}$	σ	v	sig_{sw}	t-Stat	P	H0	$\bar{x}_Q$-$\bar{x}_R$
TT	R	49.563	50.000	7.075	50.060	0.055	-6.934	0.000	Rejected	-12.44
	Q	37.125	40.000	7.183	51.597	0.060				
MT	R	15.406	15.500	1.932	3.733	0.062	-3.790	0.001	Rejected	-2.66
	Q	12.750	12.500	3.750	14.065	0.069				
CC	R	3.935	3.935	0.427	0.183	0.072	-7.186	0.000	Rejected	-0.62
	Q	3.318	3.386	0.342	0.117	0.067				
EU	R	-0.464	-0.500	0.913	0.833	0.341	2.022	0.052	Holds	0.24
	Q	-0.223	-0.214	0.944	0.891	0.514				
UF	R	-0.224	-0.083	1.202	1.446	0.135	2.165	0.038	Rejected	0.64
	Q	0.411	0.417	1.127	1.271	0.055				
SF	R	0.568	0.667	0.849	0.720	0.132	-2.247	0.032	Rejected	-0.42
	Q	0.151	0.500	1.013	1.026	0.067				

Table C.4.: Descriptive statistics of the survey results. I: indicator, sc: evaluation scenario, $\bar{x}$: mean, $\tilde{x}$: median, σ: standard deviation, v: variance, and sig_{sw}: Shapiro-Wilk normality test's significance, t-Stat: t-statistic, P: P(T<=t) t-significance (two-tailed), H0: Null hypothesis result, $\bar{x}_Q$-$\bar{x}_R$: Mean difference between QUIS and the baseline.