AIDA
An Agile Abstraction for Advanced In-database Analytics

Joseph Vinish D'Silva

Doctor of Philosophy

School of Computer Science

McGill University

Montréal, Québec, Canada

July 2020

A thesis submitted to McGill University in partial

fulfillment of the requirements of the degree of

Doctor of Philosophy

Abstract

As a result of accelerated growth in the areas related to data science and machine learning, we are presently witnessing a surge in interest in the development of advanced analytical applications that employ these concepts and techniques. As these projects are "data-hungry", their rapid influx has shaken up the data management landscape. Databases, especially those maintained by Relational Database Management Systems (RDBMSes) are a major source of data for data scientists working on such projects. However, data science workflows require support for computational primitives such as linear algebra, in addition to the data processing support required by traditional RDBMS applications. They are also characterized by long exploratory phases in the project development. As such, data scientists often prefer to use agile development environments that are database-external and therefore, are not necessarily computationally efficient. This thesis makes two important contributions to facilitate the agile development of data science applications and projects that utilizes data stored in RDBMSes, without sacrificing computational efficiency.

Our first contribution is AIDA, a programming abstraction for advanced in-database analytics. AIDA's Python based programming API leverages the familiar agile programming style employed by the popular data science environments that data scientists currently use. Similar to these systems, data scientists can use AIDA to develop data science workflows in an exploratory fashion, following an incremental, iterative, step-by-step approach to determine the solution or to build the predictive model. Data scientists can use AIDA's programming API with a regular Python interpreter or with advanced data science environments such as Jupyter Notebook installed in their workstation. However, unlike these data science environments that extract the data from the database to perform their computations, behind the scenes, AIDA employs a client-server model, transparently pushing the computational logic to the RDBMS, to achieve near-data computation.

AIDA's server component resides inside the embedded Python interpreter of the RDBMS and is responsible for intercepting the transformations sent by the client sessions and executing them over the data sets. In this aspect, AIDA's computational ideology is very similar to current in-database analytics approaches such as User Defined Functions (UDFs) that employ an embedded high-level

language (HLL) interpreter to execute HLL code inside the database. However, unlike UDF-based approaches that lack exploratory programming support, AIDA employs the agile programming paradigm concepts used by popular data science environments.

To facilitate the wide range of data processing capabilities required by data science workflows, AIDA supports the seamless use of both relational operations and numeric computations such as linear algebra, through a unified data set abstraction called TabularData. Internally, AIDA relies on the RDBMS engine to efficiently execute relational operations and on an HLL statistical library, NumPy to perform linear algebra operations. Any data reformatting or handover that is required between the RDBMS and the statistical library is done transparently by AIDA and the actual data copies between these components are avoided whenever possible. AIDA does not require changes to statistical packages or the host RDBMS, facilitating portability. Further, being Python based, AIDA's extensibility features allow data scientists to incorporate the many existing data science libraries into their workflows as custom transformations. AIDA also facilitates complex data visualizations even though it performs data processing at the server. Performance comparisons of AIDA against alternative approaches by implementing end-to-end data science workflows demonstrate that computationally, AIDA is on par with in-database approaches, while providing programming agility that is on par with the popular data science environments.

Our second contribution is a virtual table concept that addresses the performance drawbacks of RDBMS embedded HLL constructs such as UDFs. Virtual tables augment the database UDF concept and allow embedded HLL code, including UDFs, stored procedures, as well as in-database frameworks like AIDA to expose table-like HLL objects to the RDBMS optimizer in order to run SQL queries over them. Virtual tables provide an abstraction that makes HLL objects look like regular database tables to the RDBMS optimizer. As the optimizer is now able to analyze the data characteristics of the HLL objects before producing an execution plan, this facilitates better optimization opportunities for the execution of SQL queries over HLL objects. Virtual tables also minimize the need for performing data copies and conversions between the HLL code and the RDBMS, performing them lazily when and only if required. We perform evaluations over a variety of workloads which demonstrate the performance and programming benefits of virtual tables.

Abrégé

En raison de la croissance accélérée des domaines liés à la science des données et à l'apprentissage automatique, nous assistons actuellement à un regain d'intérêt pour le développement d'applications analytiques avancées utilisant ces concepts et techniques. Ces projets étant "gourmands en données", leur afflux rapide a bouleversé le paysage de la gestion des données. Les bases de données, en particulier celles gérées par les systèmes de gestion de bases de données relationnelles (SGBDR) sont une source majeure de données pour les scientifiques travaillant sur de tels projets. Cependant, les processus liés à la science des données nécessitent le support de primitives de calcul telles que l'algèbre linéaire, en plus du support du traitement des données requis par les applications RDBMS traditionnelles. Ils sont également caractérisé par de longues phases exploratoires au cours du développement du projet. Pour cette raison, les scientifiques des données préfèrent souvent utiliser des environnements de développement agile et externes à la base de données qui, par conséquent, ne sont pas nécessairement efficaces. Cette thèse apporte deux contributions importantes pour faciliter le développement agile d'applications et de projets de science des données qui utilisent des données stockées dans des SGBDR, sans faire de compromis sur l'efficacité des calculs.

Notre première contribution est AIDA, une abstraction de programmation permettant une analyse avancée des données au sein même des base de données. Basée sur Python, l'API d'AIDA utilise le style familier de programmation agile, au coeur des principaux environnements de science des données que les scientifiques utilisent actuellement. Comme dans ces systèmes, les scientifiques des données peuvent utiliser AIDA pour développer des processus de manière exploratoire en suivant une approche incrémentale et itérative pour déterminer la solution ou construire le modèle prédictif. L'API d'AIDA peut être utilisée avec un interpréteur Python ou avec des environnements avancés comme Jupyter Notebook installés sur un poste de travail. Cependant, contrairement à ces environnements qui extraient les données de la base de données pour effectuer leurs calculs en arrière-plan, AIDA utilise un modèle client-serveur, poussant de manière transparente la logique de calcul vers le SGBDR, permettant aux calculs de se faire à proximité des données.

La composante serveur d'AIDA réside à l'intérieur de l'interpréteur Python intégré au SGBDR et est chargé d'intercepter les transformations envoyées par les sessions clientes et de les exécuter sur les ensembles de données. À cet égard, l'idéologie computationnelle d'AIDA est très similaire aux approches actuelles d'analyse in situ des bases de données, comme par exemple les fonctions définies par l'utilisateur (FDU) qui utilisent un interpréteur de langage de haut niveau (LHN) intégré pour exécuter du code LHN à l'intérieur de la base de données. Cependant, contrairement aux approches basées sur FDU qui n'offrent aucun support à la programmation exploratoire, AIDA s'appuie sur les concepts du paradigme agile utilisés par les principaux environnements de science des données.

Pour servir la large gamme de besoin en traitement de donnée requise par les processus en science des données, AIDA permet une utilisation conjointe et sans interuption d'opérations relationnelles et de calculs numériques tels que des calculs d'algèbre linéaire, via une abstraction de donnée unifiée appelée TabularData. AIDA s'appuie sur le moteur SGBDR pour exécuter efficacement les opérations relationnelles et sur une bibliothèque statistique LHN, NumPy pour effectuer des opérations d'algèbre linéaire. Tout reformatage ou transfert de données nécessaire entre le SGBDR et la bibliothèque statistique est effectué de manière transparente par AIDA et la copie des données entre ces composants est évitée dans la mesure du possible. AIDA ne nécessite aucune modification de progiciel statistique ou du SGBDR hôte, ce qui facilite sa portabilité. De plus, étant basées sur Python, les fonctionnalités d'extensibilité d'AIDA permettent aux scientifiques d'incorporer de nombreuses bibliothèques existantes dans leurs processus comme transformations personnalisées. AIDA facilite également les visualisations de données complexes même si un traitement des données est effectué sur le serveur. La comparaison des performances d'AIDA avec des approches alternatives, en implémentant des processus du début à la fin, démontrent que sur le plan computationnel, AIDA a des performances équivalentes aux approches favorisant le traitement des données in-situ, tout en offrant une agilité de programmation comparable aux populaires environnements de science des données.

Notre deuxième contribution est un concept de table virtuelle qui propose une solution à la diminution des performances causée par l'intégration de constructions LHN au SGBDR telles que les FDU. Les tables virtuelles améliorent le concept FDU attaché aux bases de données et permettent au code LHN intégré, aux procédures stockées, ainsi qu'aux structures de base de données in-situ comme AIDA d'exposer des objets LHN de type table à l'optimiseur RDBMS afin qu'il soit possible d'exécuter des requêtes SQL par leur intermédiaire. Les tables virtuelles fournissent une abstraction permettant aux objets LHN d'être interprété comme des tables de base de données régulières par l'optimiseur RDBMS. L'optimiseur étant désormais capable d'analyser

les caractéristiques des données des objets LHN avant de produire un plan d'exécution, ceci crée de meilleures opportunités d'optimisation pour l'exécution de requêtes SQL via des objets LHN. Les tables virtuelles minimisent également le besoin de copier ou convertir les données entre le code LHN et le SGBDR, en les exécutant seulement si nécessaire (évaluation paresseuse). Nous effectuons des évaluations sur une variété de charges de travail qui démontrent les performances et avantages des tables virtuelles.

Acknowledgements

Foremost, I would like to thank my supervisor, Prof. Bettina Kemme, for her support and guidance that always go far and beyond the call of duty. Her ethics for hard work and tenacity to ensure quality in research is truly inspirational and she has contributed immensely as a role model towards grooming me for academic life. I am also grateful to Profs. Muthucumaru Maheswaran and Laurie J. Hendren for their valuable insights and advice as members of my PhD progress committee.

Many of my labmates have contributed to the construction of the system, AIDA, that is proposed in this thesis and I am thankful for the quality work that they have put in to validate and implement some of the ideas and concepts put forth in this thesis. Florestan De Moor, as part of his Master's internship in Winter 2018, worked on integrating the virtual table concept discussed in Chapter 6 to MonetDB RDBMS and did an outstanding job. Ting Gu, during his undergraduate summer internship in 2018, worked on analyzing the performance advantages of various compression libraries in the context of fast data transfers that are covered in Chapter 3. Jianhao Cao, also during his undergraduate summer internship in 2018, was instrumental in building a prototype of AIDA's database adapter for PostgreSQL RDBMS. He is presently pursuing this work as part of his Master's thesis. Finally, Rez GodarzvandChegini, who embarked on a mission to implement distributed machine learning algorithms over a cluster of AIDA servers as part of his Master's thesis, for being the first user of AIDA, helping us to validate its potential beyond a research prototype.

Thank you to Laetitia Fesselier for translating the abstract to French. Also, a special thank you to all of our system support staff, especially Andrew Bogecho and Ron Simpson. Their timely involvement in analyzing and addressing any system issues and software requirements have been extremely helpful throughout our work.

I would also like to express my deep gratitude to my family and friends who have supported and encouraged me when I ventured out on this long academic road. Especially grateful to my parents Rita and Milton, and my brother Royden. Finally, I have been lucky in a loving, caring, and patient wife Elizabeth who has been very supportive in this journey.

For all the wonderful people who have encouraged me in this endeavor, Deo gratias.

Contents

List of Figures

List of Tables

List of Algorithms

Listings

1 Introduction

The tremendous growth in advanced analytical fields such as data science and machine learning has shaken up the data processing landscape. These new applications are characterized by the ubiquitous usage of data processing primitives and paradigms such as *linear algebra*, *iterative computations*, and *incremental model development*, as well as long *exploratory* phases in the project development, that sets them apart from the traditional database applications. As most data science projects want to build some predictive or analytical models, these applications are also "data-hungry", where the availability of larger quantities of data makes the objective easier to accomplish, often with a positive impact on the accuracy of the results. As such, these applications call for systems that are capable of performing a significant amount of data processing, using a variety of computation primitives.

In the case of traditional database applications, the resource intensive tasks associated with data processing are usually performed by the Database Management System (DBMS), whereby the database applications send the data manipulation instructions to the DBMS, which executes them, and sends the results back to the application. This approach is especially popular with Relational Database Management Systems (RDBMSes), which owing to their mathematical set-theory based relational data model, allow database applications to specify data processing tasks in a very concise manner.

This simplicity in expressing data manipulation operations has made RDBMSes the most popular among the various database implementations [sol19]. As such, they are often the custodians for critical data that encompasses the "bread-and-butter" of the organization. However, with the emergence of data science and machine learning systems, they were found lacking in the additional data processing paradigms required by this new breed of applications. This has resulted in the data science community developing their own suite of data processing systems.

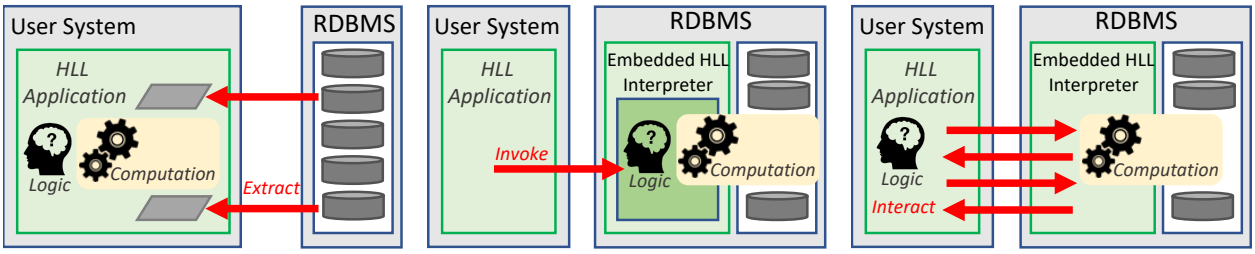
1.1 Data Science Frameworks and Programming Environments

The growth in data science and machine learning fields have been so rapid that the computational and analytical frameworks that are developed to meet the needs of the data science community are predominantly focused on their primary objective, which is to provide sophisticated modeling primitives and efficient support for various numerical computations. As a result, these frameworks were developed to exist in separation from the data management components of the organizations themselves. From the data scientists' perspective, RDBMSes contain a significant amount of vital data, and as demonstrated by various survey results, the majority of the data scientists rely on an RDBMS as their data source [Kag17, KDn14]. However, the current approaches employed by data scientists to work with these data sets generally require the data set to be completely exported from the database, to the statistical system that is used for the data science analysis (see Figure 1.1a).

On the other hand, due to the unconventionally long and complex exploration phases associated with data science projects, the popularity of these data science frameworks and environments are also dictated by how these systems account for the *human-in-the-loop*. That is, they put an emphasis on the productivity of the data scientist, focusing on the *agility* of the programming paradigms offered by the framework. Therefore, the most popular tools with data scientists are those that give them more programming flexibility and freedom during the development phase of the project.

As such, the most common approach to develop machine learning and data science applications is to use one of the many statistical languages and environments such as R [IG96], MATLAB [MMS17], Octave [EBH07], etc., or packages such as pandas [McK11], NumPy [VCV11], TensorFlow [ABC+16], etc., meant to augment a general purpose high-level language (HLL) like Python with linear algebra support. When data scientists have to work with data sets residing in an RDBMS, the first step in these systems is to retrieve the data from the RDBMS and store them in user space. From there, all computation is done at the user end.

Needless to say, user systems do not possess huge memory capacity nor processing power unlike servers running an RDBMS, potentially forcing them to use smaller data sets. Thus, users sometimes resort to big data frameworks such as Spark [ZCF+10] that load the data into a com-



(a) Database-external (b) In-database : UDF (c) In-database : AIDA

Figure 1.1: High-level Concept of Various Data Science Frameworks Approaches

pute cluster and support distributed computation. But even when enough resources are available, users might choose smaller data sets, in particular during the exploration phase, as transfer costs and latencies to retrieve the data from the database system can be huge. This data sub-setting can be counterproductive, as having a larger data set can reduce algorithm complexity and increase accuracy [Dom12]. Additionally, once the data is taken out of the RDBMS, all further data selection and filtering, which is often crucial in the *feature engineering* phase of a learning problem, needs to be performed within the statistical package [GE03]. Therefore, several statistical systems have been enhanced with some relational functionality, such as the DataFrame concepts in pandas [McK11], Spark [AXL+15], and R. As we will discuss later in this thesis, they are not as efficient as RDBMSes in performing these functionalities.

1.2 Emergence of In-database Advanced Analytics

As we can see, the above approaches implemented by the data science community are not only computationally suboptimal but also relegate the role of the database to merely a persistent data storage layer, underutilizing its data processing capabilities. This has the database community pondering about the opportunities and the role that it needs to play in the grand scheme of things [RAB+15, WZC+16]. In fact, many approaches have been proposed hoping to encourage data science users to perform their computations in the database, *near-data*, in an effort to build a *unified system* that can facilitate both conventional data management, as well as support the new capabilities required for data science applications.

One proposed approach is to integrate linear algebra operators into a conventional RDBMS by extending the SQL syntax [ZKM13, LGG+17, ALOR18]. However, the proposed syntactical extensions to SQL are quite cumbersome compared to what is provided by statistical packages, and thus, have not yet been well adopted.

In the past, the database community has successfully built DBMSes that are specialized for scientific computing fields such as satellite imagery, astronomy, genomics, etc., and they do facilitate efficient numeric computations [SBZB13, MS99, Dat16]. However, these implementations either do not support relational operations required by traditional RDBMS applications or are not optimized for them. They have also shown suboptimal performance under more generic application workloads [PPR+09, MDZ+17].

On the other hand, the mainstream RDBMSes have been cautious of making any hasty changes in their internal implementations – tuned for relational workloads – to include native support for linear algebra. Instead, most have been content with just embedding host HLL interpreters that are suitable for data science work in the RDBMS engine and providing paradigms such as User Defined Functions (UDFs) to interact with them [Mil16, WDG+16, ML13, RM16, The17]. UDF based approaches allow users to write small HLL code snippets as a function, that can be then invoked as part of a SQL statement, thus executing them inside the RDBMS, *near-data* (see Figure 1.1b), and thus are considerably faster than performing external computations.

However, HLL code blocks are essentially a black-box to the optimizer, as the later is not equipped to understand HLL constructs. As such, UDF based implementations interfere with the database optimizer's ability to generate an efficient query execution plan, resulting in suboptimal performance. Further, the use of UDFs have shown to be cumbersome, especially during the exploratory/development phase of the project that calls for more *agile* programming paradigms. This is because, UDFs require the entire data processing logic to be predefined as a function before they can be invoked. This is not a convenient mechanism for exploratory workflows, where the complete logic path is not known in advance, as the data scientist is performing a step-by-step analysis of the problem. UDF constructs are also very restrictive when it comes to the ability to perform *incremental* and *iterative* analysis, an important characteristic of the *model development* phase of data science projects. This is because they have little support for maintaining and sharing intermediate HLL objects produced inside UDF execution. They also lack the architectural facilities to provide support for data visualization capabilities, a critical tool for exploration.

1.2.1 Usability vs Performance

While numerous works, such as, Weld [PTS+17], TUPLEWARE [CGD+15], Froid [RPE+17], and HorseIR [CDC+18], have proposed an overhaul of RDBMS architecture to address the computational inefficiency introduced by the integration of embedded HLL based primitives for data science applications, the issue of user-friendly programming paradigms have not been addressed.

In fact, recent discussions on the topic of the intersection of data management and learning systems point out that even among the most sophisticated implementations which try to tightly couple RDBMS and machine learning paradigms, the *human-in-the-loop* is not well factored-in [BKY19, KBY17]. Among the machine learning community, there is already a concern that research papers are focusing solely on accuracy and performance, ignoring the human effort required to build solutions [Dom12]. Further, [Dom12] also points out that human cycles required are the biggest bottleneck in a machine learning project and while difficult to measure, easiness and efficiency in experimenting with solutions is a critical aspect. Lately, various surveys have tried to capture the distribution of the human effort and the nature of the evolution of machine learning project workflows in a more systematic fashion [Mun12, XMSP18]. Of these, [XMSP18] surveyed the machine learning literature to look into the human-in-the-loop aspect of machine learning projects and recommended a list of *desiderata* to analyze the effectiveness of a machine learning system. Their work highlights that performance is only a part of a bigger set of characteristics that these implementations should possess, and that the "human element" is as important a factor for successful data science projects.

Therefore, unsurprisingly, despite efforts from the database community, studies have consistently shown that user-friendly and flexible environments, such as Python, and R are the top favorites among the data science community [Pia17, DCR14, Kag17].

1.3 Contributions

The primary contributions made by this thesis are twofold. In the first contribution, we put forward a new paradigm, AIDA - an agile abstraction for **a**dvanced **i**n-**d**atabase **a**nalytics. AIDA is a framework that provides support for in-database advanced analytics, one that takes into account the productivity of the *human-in-the-loop*, by providing an agile, client-based programming environment that performs server-based computations both seamlessly, and interactively, leveraging the efficient data processing capabilities that are traditionally associated with RDBMS. This principled architecture is shown in Figure 1.1c. In the second contribution, we analyze the drawbacks of contemporary RDBMS embedded HLL constructs such as UDFs, that are employed in integrating HLL computations and data sets into RDBMS query processing, and propose a *virtual table* approach to augment them, that is both computationally efficient, as well as convenient in terms of the programming effort required.

1.3.1 AIDA – An Agile Abstraction for Advanced In-database Analytics

AIDA is our implementation of an in-database data science framework that emulates the syntax and semantics of popular Python based statistical packages for user-friendliness. It provides a client API for data scientists to work with, in an *interactive manner*, but transparently shifts the computation to the RDBMS, performing them *near-data*. AIDA can work with both linear algebra and relational operations simultaneously, by using the query engine of the RDBMS for relational operations and an embedded statistical system for linear algebra, moving computation between these two systems that share the same address space, transparently and efficiently. Thus, it facilitates a *unified* system capable of data management as well as advanced analytics. AIDA maintains and manages intermediate results from the computation steps as elegantly as current HLL based database-external data science systems, without the hassles and restrictions that UDFs and custom SQL-extension based solutions place. AIDA also supports exploratory data *visualization* of data sets stored in the database.

We implement AIDA with MonetDB as the RDBMS back end, but follow a modular approach that facilitates the integration of AIDA into other RDBMSes as well. We show that AIDA has a significant performance advantage compared to approaches that transfer data out of the RDBMS. Using in-database frameworks such as AIDA also avoids the hassles of refreshing data sets in the presence of updates to the source data sets, an issue with the database-external approaches employed by contemporary data science systems.

Our main objectives for AIDA, and how it compares to popular alternatives, are shown in Table 1.1. We believe that AIDA is a step in the right direction towards popularizing advanced in-database analytics with the data science community. In short, we make the following contributions in this aspect.

(i) Identify the shortcomings of current RDBMS solutions such as UDFs to integrate linear algebra operations and propose a new interface for interactive and iterative development that fits more elegantly with current programming paradigms that have been successful with the data science community.

(ii) Implement a Remote Method Invocation (RMI) approach to push computation towards the data, keeping it in the RDBMS.

(iii) Implement a common data abstraction, *TabularData*, to represent data sets on which both relational and linear algebra operations can be executed.

(iv) Exploit an embedded HLL (Python) interpreter to efficiently execute linear algebra operations, and an RDBMS (MonetDB) to execute relational operators.

(v) Transparently provide data handover between the RDBMS and the embedded HLL inter-

preter during data transformations, avoiding data copy whenever possible.

(vi) Provide visualization support for database-resident data.

(vii) Allow host language objects to reside in the database memory throughout the lifetime of a user session to facilitate incremental and iterative development.

(viii) Develop a database adapter interface that facilitates the use of different RDBMS implementations with AIDA.

(ix) Implement an efficient external data transfer approach that is optimized for database-external statistical systems.

(x) Provide quantitative and qualitative comparisons of our approach with other in-database and database-external approaches.

Table 1.1: AIDA Vs. Popular Data Science Tools - Characteristics

System	Languages	Interactive	Incremental	Near-data	Visualization	Unified
AIDA	Python	✓	✓	✓	✓	✓
DB UDF	Python, SQL	✗	✗	✓	✗	✓
pandas	Python	✓	✓	✗	✓	✗
Spark	Scala	✓	✓	✗	✓	✗

1.3.2 Virtual Tables – Optimizer-friendly Integration of HLL Data Sets

As mentioned before, UDFs are a commonly proposed approach to integrate HLL based computations as part of RDBMS query processing, and as such, have found use as underlying primitives in various works associated with data science and machine learning. Our in-database framework, AIDA, also makes use of the UDF constructs internally to expose objects created by the statistical package back to the RDBMS, in order to perform additional relational operations over them using the RDBMS engine. However, database optimizers rely on data characteristics as one of the many parameters to generate an efficient plan. As traditional RDBMS optimizers do not have the capability to analyze the characteristics of HLL objects, the black-box nature often results in suboptimal query execution plans and associated degradation of performance. UDFs are also restricted from a programmability perspective, which makes it difficult to send data back and forth between the HLL interpreter and the RDBMS engine, impeding the development of complex workflows.

To address these drawbacks of conventional UDFs, we propose a virtual table approach to augment the current UDF constructs. Virtual tables expose table-like HLL data sets to the RDBMS optimizer which can now treat them like any other database tables. Virtual tables allow the database

optimizer to understand the characteristics of the HLL objects to generate efficient execution plans when they are used in queries. Embedded HLL constructs such as UDFs, and in-database frameworks such as AIDA can make use of virtual tables to efficiently execute SQL queries over HLL objects. Using virtual tables, HLL objects can be exposed to the RDBMS at any point in the execution of the HLL code. Thus, they are more flexible from a programming perspective, allowing for the development of complex workflows that need to frequently move data between the HLL and the RDBMS. Further, as the data types of the embedded HLL and the underlying RDBMS do not always align, virtual tables can provide automatic data conversions, performing them lazily, and only when the specific column from an HLL object is required as part of some query processing.

In a nutshell, we make the following contributions to improve the performance and usability of the conventional UDF construct.

(i) Propose a virtual table concept that offers an elegant way to expose HLL objects to the RDBMS engine.

(ii) We implement the virtual table concept for the MonetDB columnar-RDBMS that can perform lazy conversion of data sets and provide additional metadata such as heuristics to the optimizer for the generation of an efficient query execution plan.

(iii) Demonstrate the performance benefits of using virtual tables with in-database frameworks such as AIDA, to obtain better query performance over HLL data sets generated inside the framework.

(iv) Demonstrate through some examples as to how leveraging virtual tables reduces the overall programming complexity associated with using conventional UDF based solutions.

(v) Demonstrate through means of a practical example, how the programming flexibility made possible by virtual tables facilitates the exploration and implementation of new, efficient approaches to solving some of the conventional data science problems.

(vi) Provide a detailed evaluation of the performance and usability benefits of virtual tables over contemporary alternative approaches.

1.4 Thesis Outline

The remainder of this thesis is organized as follows.

Chapter 2 provides the background for the thesis. We first provide a brief overview of the *relational* model and RDBMS implementations. We also take a look at the evolution of the various software architectures pertaining to database applications and highlight their limitations in the context of the emergence of data science applications. We will then discuss

the characteristics of contemporary database-external data science systems, the reasons for their popularity, and their architectural drawbacks. Finally, we will cover the efforts made by the database community to facilitate in-database implementation of many functionalities required by data science workflows.

Chapter 3 provides first an overview of AIDA's overall architecture, and its concept of unified data abstraction, TabularData. We will then take a deeper look at the internal architecture of AIDA, and the RMI concepts that make possible the seamless transfer of client side computation requests to the RDBMS for execution.

Chapter 4 introduces the reader to the advanced capabilities of AIDA. We will see how users can implement custom transformations on data sets, encounter AIDA's capability to shift blocks of client side code to the server for execution, and see how AIDA can be used in a distributed environment, including data transfer between multiple AIDA servers. We will also discuss how AIDA is able to integrate some of the sophisticated data visualization tools that are popular with data scientists.

Chapter 5 presents the performance and usability analysis that we performed on AIDA. We will discuss the test setup, data sets, various micro-benchmark evaluations, and a qualitative and quantitative evaluation of AIDA over an end-to-end learning problem. Through our various test cases we will compare and contrast AIDA against some of the contemporary, popular database-external and in-database approaches.

Chapter 6 introduces the concept of virtual tables. We first discuss the performance drawbacks of UDF based approaches and then describe the implementation details of virtual tables. Following this, we will demonstrate through a variety of data sets and test cases as to how virtual tables provide significant performance benefits in query processing and programming flexibility compared to conventional UDF alternatives.

Chapter 7 presents the conclusions of the thesis and outlines some of the future research directions for AIDA.

2

Background & Related Work

In this chapter, we will first take a brief look at the relational model and its characteristics that have made the Relational Database Management System (RDBMS) the most prominent choice for data management. We will then discuss how SQL, the declarative query language for interacting with RDBMS implementations is traditionally used in combination with general-purpose high-level languages (HLLs) to build complex database applications and their prevalent architectural designs. Following that, we will see how the traditional database application architecture is not ideal for data science projects, which has resulted in the evolution of database-external frameworks that are not architecturally optimal. We will also discuss the efforts made by the database community to integrate some of the computational capabilities required by data scientists into database implementations. We will conclude the chapter by comparing the "user-friendliness" of such database-centric programming paradigms to that of the contemporary popular, database-external implementations.

2.1 Relational Database Management Systems

The relational model proposed by Edgar Codd in the 1970s revolutionized the database landscape and paved the way for the evolution of RDBMSes [Cod70]. The relational model is based on mathematical *set theory* and provides a high level of abstraction for data modeling and querying, liberating the database application programmers from having to familiarize themselves with the low-level storage details of the database [Got75]. Database application programmers can use a declarative type of language for querying and data manipulation to express the "what" of the data processing task and the RDBMS implementers will address the "how" of the actual execution process that takes into account a variety of factors to produce an optimal execution plan. Their versatility and ease of use quickly resulted in RDBMSes becoming the *de facto* implementation of choice of organizations for data management.

2.1.1 The Relational Model

An important aspect of the relational model is that the data in the database is stored in the form of multiple *relations* where each relation consists of several *attributes*. A *record* or *tuple* in a relation is basically a set of values associated with each of the attributes of the relation. When it comes to the physical implementation of databases, relations are often referred to as tables and attributes as columns. We will keep this distinction in terminology depending on the context, i.e., whether we are talking about the theoretical aspect or the physical construct. But otherwise, the two terminologies are interchangeable for all practical purposes.

Typically, a relation has a *primary key* $\mathcal{P}$, which is composed of one or more attributes whose values can be used to uniquely identify each record in that relation. Different relations in a database are also generally related to each other, which plays a significant role in analyzing the data stored in an RDBMS, as we will see in Section 2.1.3. A set of attributes $\mathcal{F}$ of a relation $\mathcal{R}$ forms a *foreign key* in $\mathcal{R}$ if the possible values in the attributes $\mathcal{F}$ of the relation $\mathcal{R}$ are a subset of the values of $\mathcal{P}$ where $\mathcal{P}$ is the *primary key* of a relation $\mathcal{S}$ (which could be identical to R). Here, $\mathcal{R}$ is termed the *referencing* relation and $\mathcal{S}$ is called the *referenced* relation.

For example, Figure 2.1 shows the relational data model for the TPC-H benchmark, which is a synthetic benchmark that is commonly used in the performance testing of RDBMS implementations [Tra17]. We will be utilizing this model throughout various examples used in this thesis as well as for some of the performance evaluations.

TPC-H basically defines a concise relational database model that mimics a Business to Customer (B2C) application. Tables `CUSTOMER`, `NATION`, `REGION`, `SUPPLIER`, and `PART` represent their

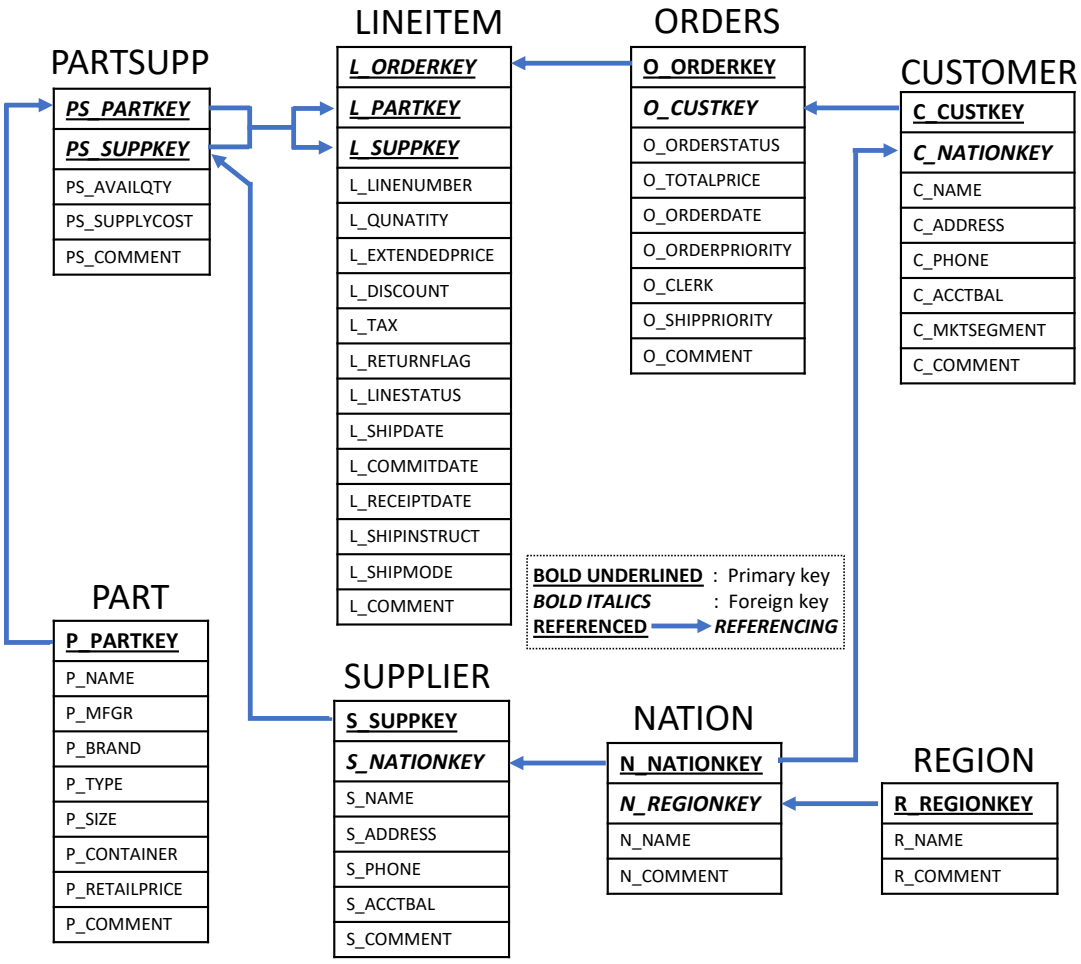


Figure 2.1: Relational model of TPC-H benchmark [Tra17]

namesake real-world entities and are used to capture the corresponding data records. Customers can order parts from various suppliers, and each order is captured in the `ORDERS` table. The individual line-items that constitute an order, i.e., the quantity of each part, the supplier responsible to ship them, etc., are captured in the `LINEITEM` table. Further, `PARTSUPP` table is used to keep track of the suppliers who are distributors for a specific part. In Figure 2.1, the primary key columns of the tables are highlighted with a bold, underlined font, whereas foreign keys are denoted using a bold, italics font. For instance, `C_CUSTKEY` is the primary key of the `CUSTOMER` table and `O_CUSTKEY` is a foreign key of `ORDERS` table that references it. This essentially means that any value of `O_CUSTKEY` in the `ORDERS` table should also be present in the `C_CUSTKEY` of the `CUSTOMER` table.

Figure 2.2 shows an excerpt from an example instance of a TPC-H relational database consisting of the `ORDERS` and `CUSTOMER` tables. However, the reverse is not necessarily true (for example, customer key `712` is not present in `O_CUSTKEY` although it is present in `C_CUSTKEY`). This is a valid

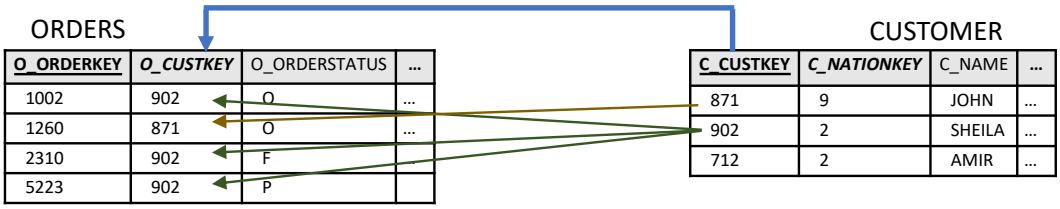


Figure 2.2: An excerpt from an example instance of a TPC-H relational database

scenario, as a customer may not have placed an order yet, whereas every order should have a customer who is responsible for placing that order.

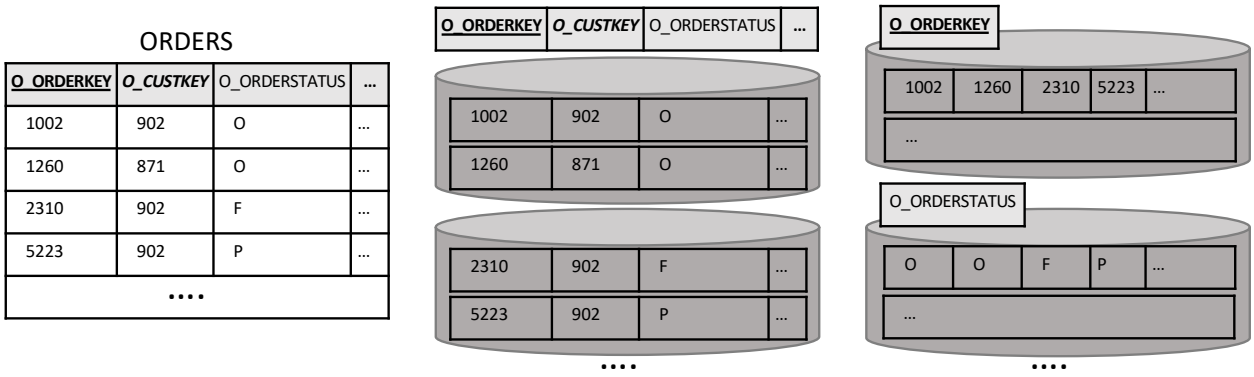
2.1.2 Storage Architectures for RDBMS

The relational model itself is a *logical* model, i.e., it focuses on describing the nature of the data (domains of columns, constraints, etc.) and the relationships between various data items (such as the foreign key concept that we just discussed). This leaves the vendors with a significant amount of flexibility when it comes to the physical implementation details of the RDBMS software. Although there exists a variety of RDBMS implementations, at a high level their physical data storage architectures can be mostly classified as either *row-based* or *column-based*.

Row-based RDBMS

Row-based RDBMS (also known as *row-stores*) implementations store the values of all the columns associated with a record of a table contiguously in the disk block. In row-based storage, accessing a specific record in the table or adding a new record to the table often involves only one disk block of I/O. They are thus popular with database applications that use such *point* queries, formally termed as Online Transaction Processing (OLTP) systems [Cla92]. As most of the initial use cases for database applications were of this nature, many traditional RDBMS implementations follow this storage approach. For example, Figure 2.3a shows an instance of data for the `ORDERS` table and Figure 2.3b shows a row based disk storage of the same. As we can see each row is stored contiguously and contained within a disk block.

However, with the emergence of widespread use of *business analytics* for decision making, the row-based approach turned out to have some performance limitations when answering queries over large data sets. This is because, as all the columns of a row are stored contiguously, and since the basic unit of disk I/O is usually at least as big as a disk block, the database inadvertently ends up reading all of the columns, whereas it may actually require only a smaller subset of columns to answer the query. This is not of much significance if we are only interested in one record.



(a) example orders data (b) row-based storage (c) column-based storage

Figure 2.3: Row-based and column-based disk storage in RDBMS

However, it can be an issue when the query has to process a large number of records. The database now has to perform a considerable amount of I/O, but its "returns" in terms of relevant data from each disk block that it is reading is not optimal. Assuming that the query needs to process the column values for every record in the table, this would mean that the database has to read all the disk blocks associated with that table. For example, if we were interested in finding out the total number of open orders (`ORDERSTATUS = 'O'`), the database will have to read all the disk blocks associated with the `ORDERS` table, even though much of the data in these blocks are irrelevant to the query that is being executed.

***Discussion*:** Although database constructs like *indexes* and *partitions* can reduce the I/O cost in certain scenarios, they do not address all the limitations and have their caveats. For the sake of brevity, we do not delve into those details. These constructs and their characteristics have been well discussed in the database literature [SKS11, MTT12, Ape88, NB99].

Column-based RDBMS

Columnar RDBMS (also known as *column-stores*) implementations have been around for a long time [Raa07, ABH09]. Unlike their row-based counterparts who store each record contiguously, column-based implementations store each column's values contiguously. I.e., all the values for a given column across all the records of the table are stored contiguously in the disk block(s). This is logically very similar to the programming language concept of *array-like* data structures where each column's values would be stored in an "array" of its own. If we look at Figure 2.3c, we can see that all the values corresponding to the `O_ORDERKEY` column are stored on its own disk block(s), as is the case with all the other columns in the `ORDERS` table. Since all the columns of a given record are spread across multiple disk blocks, retrieving a specific record or inserting a new record incurs

far more I/O compared to the row-based approach. Due to this, they are not very efficient for OLTP type of database applications and as a result did not find much favor during the initial decades of RDBMS evolution.

On the other hand, the column-based approach performs very well when a query needs to process only a subset of columns across a large number of records in the table. This is because, as columnar storage results in each column having their own distinct sets of disk blocks, the database does not have to read columns that are irrelevant to the query [HLAM06]. Therefore, only a subset of disk blocks belonging to the table needs to be read. As a result, columnar databases have encountered a surge in interest during recent times due to significant growth in the amount of data being stored combined with the rising popularity of various advanced analytical applications, making these I/O savings very consequential for the performance of database applications that employ such queries.

Going back to our example in Figure 2.3c, we can see how a columnar database can answer the query to find the total number of open orders by only reading the disk blocks associated with the `O_ORDERSTATUS` column, thus incurring a lot less I/O compared to the row-based approach. As we can see, although the high-level relational data model is the same, the underlying implementation details and the associated performance implications are significantly different between these storage architectures.

For an avid reader, [AMH08] and [HLAM06] contain some in-depth analysis between the two storage approaches and their performance characteristics. In our implementation, we use the column-based RDBMS MonetDB as our back end; and we will see that the column-based architecture is particularly convenient if we want to integrate statistical and numerical operations into the RDBMS space.

2.1.3 Querying Relational Databases

An Algebra for the Relational Model

Just as the relational model is based on the mathematical concept of sets, the accompanying *relational algebra* defines the abstract semantic notations and concepts that provide the theoretical foundation for data processing over it [Cod70]. All relational algebra operators take either one or two relations as input and produce exactly one relation as output which will not contain any duplicate records. Below we will discuss some of the fundamental data processing operations that relational algebra defines and their behaviour.

The ***selection*** operator, denoted as $\sigma_c(\mathcal{T})$, is used to select a subset of records from a relation

$\mathcal{T}$. Where c is the predicate check that is to be used to decide whether a record must be included in the operator's output relation. The attributes of the output relation will be the same as that of the input relation. If we consider the example orders data in Figure 2.3, we can identify the records belonging to open orders by applying the selection operator on the ORDERS table, as in $\sigma_{\texttt{O\_ORDERSTATUS}='O'}(\texttt{ORDERS})$.

To perform this operation, a row-based implementation will have to read all the disk blocks of the ORDERS table. While a column-based database can figure out the records which will qualify for the output by merely reading the disk blocks belonging to the O_ORDERSTATUS column, it may still have to read some of the remaining disk blocks of the ORDERS table to fetch the remaining columns of the selected records. In practice, columnar implementations may be able to produce the output with less I/O if only a small fraction of records are selected or if only a smaller subset of columns is required in the output (discussed next). This process of producing the output of queries, known as *materialization*, follows a variety of optimization strategies intended to reduce the overall I/O cost of a database query and is heavily influenced by the underlying storage architecture [AMDM07, AMH08, HLAM06].

The ***projection*** operator, denoted as $\pi_a(\mathcal{T})$, where a is a subset of attributes in the relation $\mathcal{T}$, is used to extract specific attributes of interest from a relation. Therefore, the attributes of the output relation are determined by a. For example, referring to Figure 2.3, if we are only interested in the O_CUSTKEY column of the ORDERS table, this can be performed via a projection, as in $\pi_{\texttt{O\_ORDERKEY}}(\texttt{ORDERS})$. While a row-based implementation will have to scan across all the disk blocks of the table to produce the output of the query, a columnar implementation has to read only the disk blocks used to store the O_ORDERKEY column.

The ***join*** operation, denoted as $\mathcal{T} \bowtie_c \mathcal{S}$, where $\mathcal{T}$ and $\mathcal{S}$ are relations, is used to combine information from associated relations. The predicate condition c determines how each record from either relation are combined. The output relation will contain all the attributes from both the input relations. For example, referring to Figure 2.4, we can associate each order with the corresponding customer information by employing a join operation, as in $\texttt{ORDERS} \bowtie_{\texttt{O\_CUSTKEY=C\_CUSTKEY}} \texttt{CUSTOMER}$. Here the join predicate specifies that a record in the ORDERS table is matched with a record in the CUSTOMER table only if the O_CUSTKEY attribute of the former has the same value as the C_CUSTKEY attribute of the latter. Recall that O_CUSTKEY is a foreign key of the ORDERS table that is referencing the primary key of the CUSTOMER table (C_CUSTKEY). Joins are thus most commonly used in traversing these foreign key references to combine information that are otherwise stored across multiple tables. In this example, the output relation will consist of each order record with all its attributes, as well as all the customer attributes from the corresponding customer record.

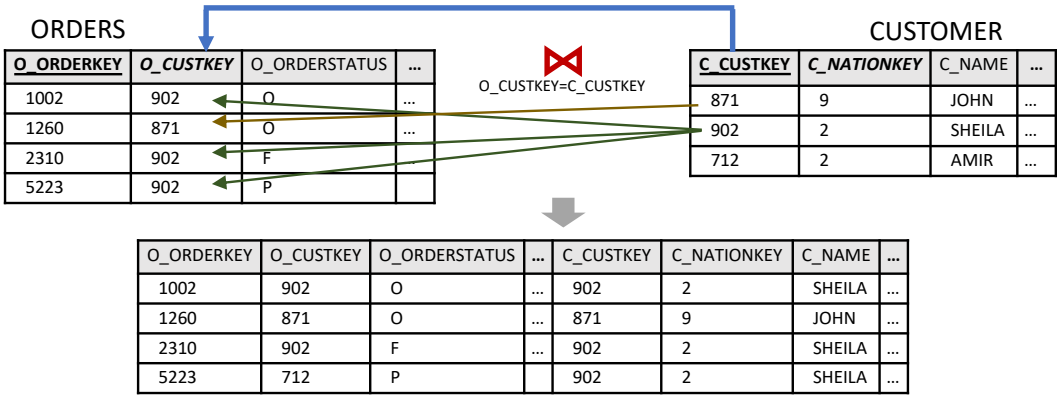


Figure 2.4: A relational *join* example based on TPC-H

Further, since the input and output to relational operators are relations themselves, we can "combine" the operators in interesting ways to perform more sophisticated queries. For example, if we are only interested in the orders and corresponding customer information for open orders, we can accomplish this as a combination of selection and join operations, as in

$\sigma_{\texttt{O\_ORDERSTATUS}='O'}(\texttt{ORDERS}) \bowtie_{\texttt{O\_CUSTKEY=C\_CUSTKEY}} \texttt{CUSTOMER}$

Here, we first perform a selection on `ORDERS` table for open orders and then the ensuing output relation is joined with the `CUSTOMER` table to retrieve their corresponding customer information. Equivalently, one may also perform this as a join followed by the selection with the same end result, as in $\sigma_{\texttt{O\_ORDERSTATUS}='O'}(\texttt{ORDERS} \bowtie_{\texttt{O\_CUSTKEY=C\_CUSTKEY}} \texttt{CUSTOMER})$

As we just saw, most data manipulation operations have a number of equivalent alternatives with potentially different execution costs. However, most modern RDBMS implementations employ a *query rewrite* subsystem to automatically transform a user-written query into an equivalent less resource-intensive form based on its internal architecture [JK84, HSH07]. Therefore, database application programmers are not burdened to deal with such performance logistics [HSH07].

Further relational algebra operators also include the more familiar mathematical set-based concepts such as ***union***, ***intersection***, and ***difference***. As our discussion on relational algebra is meant to be only representative, we do not go into further details. The topic is covered in depth by most senior undergraduate level database textbooks such as [RG03, GMUW13].

From the above discussion, we can see how the relational model along with relational algebra, provides a higher level of abstraction for modeling and processing data in a manner that is agnostic of the physical implementation details of the database. Relational algebra also forms the theoretical foundation for SQL, the *de facto* query language used across RDBMS implementations.

SQL: A Declarative Query Language for RDBMS

Originally proposed by [CB74], SQL is a declarative query language that is primarily built upon relational algebra and is supported across most modern RDBMS implementations. Here, we briefly outline some of the qualitative characteristics of SQL.

Similar to relational algebra, SQL allows database application developers to specify the data processing logic using a high-level syntax that is agnostic to the physical implementation logic. We can write the *selection* example from relational algebra using SQL as follows.

```
SELECT *
FROM ORDERS
WHERE O_ORDERSTATUS = 'O';
```

While the `FROM` keyword in SQL is used to indicate the list of tables involved, the `WHERE` keyword is used to specify the predicate condition, which in our example is used to *select* only the records for open orders. The `SELECT` functions as the *projection* operator in relational algebra. Whereas the default $*$ argument to `SELECT` returns all the columns (as in the above example), we can limit the columns returned in the query by passing it the list of columns that we are actually interested in. For example, we could limit ourselves to the `O_CUSTKEY` column of the `ORDERS` table using the following SQL syntax.

```
SELECT O_CUSTKEY
FROM ORDERS;
```

It must be noted that unlike relational algebra, SQL does not automatically eliminate duplicate records. However, this can be enforced by using the `DISTINCT` keyword in the `SELECT` clause.

Joins in SQL are performed by listing the tables to be joined in the `FROM` clause, and the predicates for the join condition itself are passed to the `WHERE` clause. The SQL snippet below joins the `ORDERS` and `CUSTOMER` tables to produce an output that has each order record listed with the corresponding customer record alongside.

```
SELECT *
FROM ORDERS, CUSTOMER
WHERE O_CUSTKEY = C_CUSTKEY;
```

SQL provides far more features over conventional relational algebra, including the capabilities to perform operations such as aggregations. For example, one could find the total number of orders using the following SQL.

```
SELECT COUNT(*)
FROM ORDERS;
```

As can be seen from the preceding examples, the declarative nature of SQL simplifies the programming effort in terms of data manipulation. Whereas the language provides powerful features, it is agnostic to the physical architecture of the RDBMS implementation itself. This decoupling also has other associated benefits. It gives the RDBMS vendors the liberty to choose the data storage and query execution approaches that are best suited for their overall architecture. Query optimization techniques that are employed in this regard is a well-developed field in database research [JK84, HSH07].

2.2 Traditional Database Application Architectures

Though RDBMSes and SQL provide a powerful paradigm for data management and querying, in the grand scheme of things, they are only one of the multitude of components that together constitute a complex software system. In this section, we will discuss various software architectures with relevance to their interaction with databases.

2.2.1 Combining Query Languages and HLL

As software constructs, the capabilities and functionalities of database systems and their associated query languages are inherently focused on providing support for the storage, retrieval, and manipulation of data. In general, these features alone are not sufficient to build a real-world application.

For example, consider a simple invoicing system. The features that are expected to be provided by such an application may be broken down at a high level into: (i) A Graphical User Interface (GUI) for the clerk to indicate the customer for whom the invoice needs to be generated. (ii) Provision to search through the orders and customer data records to retrieve the relevant orders. (iii) To format the invoicing information into a visually aesthetic layout either within the GUI to present it to the user and/or send it to a printer.

While database systems and query languages are well placed to facilitate feature (ii), they have little support for the remaining features required in the application. These, on the other hand, are the realm of general-purpose programming languages, also termed as high-level languages (HLLs). In fact, database query languages such as SQL, that are broadly categorized under Domain-specific Language (DSL), are designed to be invoked from an HLL based program [LL99a, FG16]. Therefore, practical software applications are built as a combination of both these programming paradigms where the HLL plays the role of the "driver", dictating the overall control flow of the application. Such approaches of combining different programming languages are generally termed as *polyglot programming* [Fje08].

Example 2.1. Listing 2.1 shows a pseudocode program that employs such a programming style to build a bare-bones version of our invoicing system. The program prompts the user (clerk) to enter a customer's phone number which is then used to retrieve the customer's open orders from the database, which are then used to display an invoice.

Here lines 3-4 of the HLL code prompt the user to enter the customer's phone number and reads it to a program variable, `cphone`. In lines 6-11, the program executes a SQL on the database to retrieve the `O_ORDERKEY` and `O_TOTALPRICE` of open orders that are associated with that customer. The set of orders information thus retrieved is stored in the HLL variable `invorders`. The nuances of connecting from an HLL application to a database to execute queries will be discussed in Section 2.2.2. Finally, in lines 13-16, the HLL code iterates over the set of orders retrieved from the database and displays it to the user.

Listing 2.1: Pseudo code for an invoicing system database application

```
1  loop
2  {
3    print("Enter the customer's phone number :");
4    cphone = read();
5
6    invorders = db.execute(
7      SELECT O_ORDERKEY, O_TOTALPRICE
8      FROM ORDERS, CUSTOMER
9      WHERE O_CUSTKEY = C_CUSTKEY
10     AND C_PHONE = '{cphone}' AND O_ORDERSTATUS = 'O'
11   );
12
13   print('Invoice For Open Orders\n\n');
14   print("O_ORDERKEY -- O_TOTALPRICE\n");
15   for order in invorders:
16     print("{order['O_ORDERKEY']} -- {order['O_TOTALPRICE']}\n");
17 }
```

Database Application Classifications

Although database applications vary significantly in their features and use cases, the following two categories are the most prominent among them.

Business Intelligence (BI) applications play a pivotal role in business analytics, aiding the analysis and decision-making process. Examples of such usage would include identifying products that are underperforming in the market, forecasting future sales, etc. They often facilitate the

business users with a sophisticated, but not analytic *presentation layer* [McL02] that can accept some user inputs, execute any necessary queries on the database, and "report" the results back to the user. For this reason, they are also informally known as *reporting* applications. As software systems that are built for human interaction, they are also characterized by ease of use and intuitive data visualization capabilities.

Extract Transform and Load (ETL) applications, on the other hand, are usually executed as a scheduled/event-driven process, often interacting with other processes and systems that are spread across the organization [ST08]. Their primary objective is to extract data from some source (often a database), transform them according to a set of predefined rules, and load them into a database (which sometimes could be the same as the source). They serve the purpose of "enriching" the information stored in the database and to perform any maintenance required for the upkeep of the stored information. For example, a retail organization may employ an ETL process to extract the daily sales information from the databases that support their point of sale systems (POSs) and update a centralized inventory management database that deals with the *supply chain management* aspect of the organization.

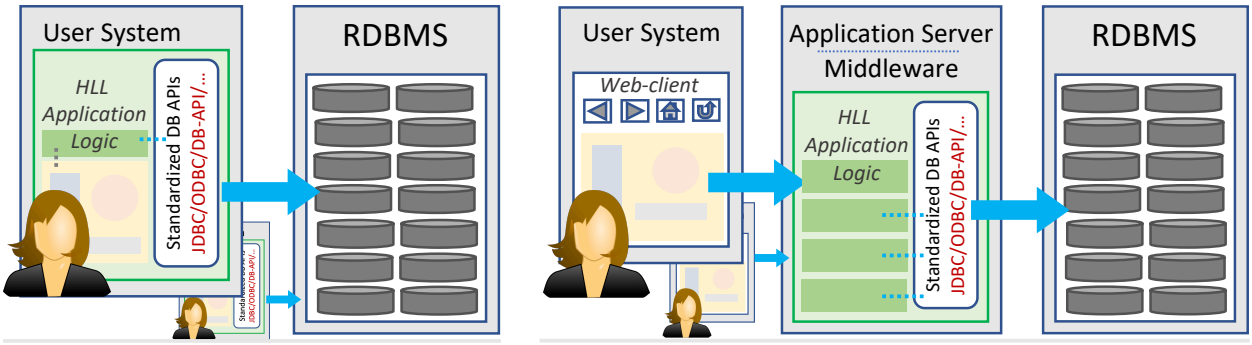
As many of these database applications can be quite complex, various software engineering architectures have evolved to facilitate ease of development and maintainability.

2.2.2 Tiered Architectures

Two-tiered Architectures

A *two-tiered* software system provides a basic "separation of responsibilities" where the database provides the data management services, and the HLL application is responsible for the user-facing functionality (such as presentation services) as well as a significant part of the business logic. The application connects as a client (*tier 1*) to the database server (*tier 2*) to invoke and perform any required data manipulation operations to perform its functionality [SE07]. One way to accomplish this is to have the HLL programs embed the SQL logic inside them [LL99b], such as the pseudocode example that we saw in Listing 2.1. This is done using an HLL and RDBMS vendor-specific syntax often in the form of a Common Language Interface (CLI) library. Initial database applications predominantly followed this simplified architecture. As the use cases and requirements for these database applications were well defined and the number of applications themselves was small in number, this tightly coupled paradigm worked well in the beginning.

However, as database applications became ubiquitous and the number of RDBMS implementations diversified, the lack of standardization in the interaction between client (HLL) applications and database servers became a maintenance issue, as they became a hurdle in porting ap-



(a) Two-tier system

(b) Three-tier system

Figure 2.5: Database application architectures

plications to different platforms or in migrating between various RDBMS implementations. This resulted in the development of Object Database Connectivity (ODBC) and Java Database Connectivity (JDBC) standards that define a set of standardized API calls for HLL applications to interact with a database [Gei95, FEB03]. RDBMS vendors implement drivers that conform to these standards. This made it easier to develop database applications as well as to migrate them between RDBMS implementations as developers could just swap the concerned driver without much impact on the application code itself. Figure 2.5a shows the high-level architecture of such a contemporary two-tiered application.

Although database applications have greatly increased in functionality and complexity, resulting in the emergence of more sophisticated software architectures, many implementations still utilize the two-tiered approach for their simplicity.

Multi-tiered Architectures

With the emergence of the Internet and the increase in user-facing database applications, the two-tiered architecture necessitated the installation of application programs in the computing workstations of several users, posing significant maintenance challenges. This led to the emergence of *three-tiered* or even more complex *N-tiered* architectures [SE07].

In such an approach, the user workstations can be equipped with only a *thin client* to perform the tasks associated with user interaction, reducing the maintenance overhead [Dew98]. Modern browsers, with their support for sophisticated markup and scripting languages such as HTML, JavaScript, etc., have essentially taken up the role of the thin client implementation [DT11, WL02].

Application servers or *middlewares* are employed as intermediate tiers between the end client and the database to manage most of the business logic and in some cases to also provide auxiliary features required for the system architecture [Lin01]. Examples for the latter include support for

transactions that are distributed across several databases [AP03], caching of database query results [DILR00], etc. Middlewares also employ the usage of JDBC and/or ODBC APIs to perform any database interaction. Figure 2.5b shows such an architecture for a three-tiered system.

Most of the contemporary BI tools provide support for a three-tiered architecture. For example, with Microstrategy [Mic12], end users can use a web browser based client to interact with the analytics reports, which themselves are hosted by a centralized *intelligence server* component of Microstrategy. The intelligence server is responsible for interacting with the databases and executing queries to retrieve the data required to construct the reporting dashboards.

2.2.3 Data Flow Optimizations

Many conventional database implementations store data in the order of several Terabytes. Therefore, it is not efficient to transfer such large amounts of data neither to the middleware nor to the end client. As a result, there has been a lot of interest in data workflow optimization techniques [KGS17, LPVM15].

Incidentally, most of the traditional analytics questions can be answered by some form of *aggregate* queries over large data sets, or by *point* queries that are only interested in specific detail records. As such, it makes sense for applications to transform the business logic of the user requests into an equivalent SQL representation to be executed in the database and then only retrieve the relatively small result set [DCSW09]. Commercially termed as *pushdown optimization*, this ensures that the size and number of the data transfers out of the database are kept to a minimum, while also efficiently leveraging the computing power of RDBMS servers.

The success of this approach is affirmed by its widespread use in several commercial BI and ETL implementations [Mic12, SAP17, Inf07, IBM08]. These tools offer easy-to-use front-ends, enhancing developer productivity while still providing fast execution times by leveraging the performance of the RDBMS and associated hardware transparently. For example, Microstrategy [Mic12] provides a sophisticated GUI for developers to build analytical reports which can then be hosted in its centralized server. The server transparently converts the data processing logic of these reports into appropriate SQL statements and executes them in the database. The results returned by the database are then presented to the users, usually in the form of some intuitive visualization (such as tables and charts) that is accessible through a web browser.

As mentioned before, ETL tools are often used to process and move data across different data sources in an organization. These tools need to leverage additional optimizations to reduce processing and data transfer costs. The most commonly employed approach is to perform a *task merge* [KGS17]. In this approach, developers may construct an ETL workflow that consists of a sequence

of smaller transformation tasks starting from the source data set and eventually terminating at a target data set (usually a database table). This allows them to develop complex workflows as a combination of several simpler, smaller steps of data transformation logic, that is more intuitive to develop and maintain. However, instead of executing the logic of each distinct task individually, the ETL tool internally combines their transformation logic into fewer, more complex tasks that produce semantically equivalent results at the target data set [KGS17]. This reduces the overall execution cost compared to executing separate distinct tasks for each transformation step.

One way to accomplish this is to utilize the transformation operations supported by the source and target RDBMS implementations [DCSW09]. This is performed by first analyzing all the transformations required from the source end of the data flow to the target end. The transformations at the source end that are supported by the source RDBMS are grouped to be executed directly in the source database using supported SQL logic. In the next step, a similar analysis is done with the transformations at the target end supported by the target RDBMS. The ETL server itself will execute only those transformations that have no equivalent operation(s) in the corresponding RDBMS. As we can see, this approach advocates transforming as much of the execution logic as possible to be executed within the RDBMS.

2.2.4 Advanced Analytics Applications

An important characteristic of traditional database applications is that their data processing logic is in general expressible as a combination of some form of *relational* operations, making it possible to automatically translate the data-centric parts of the business logic into SQL queries, to be executed in the database. The data flow optimizations that we discussed in the previous section are implemented by capitalizing on this aspect.

However, with the emergence of advanced analytical applications such as data science and machine learning workflows, the situation has changed. While a significant number of data processing steps in these applications are still of relational nature (such as during the *feature engineering* phase), they are also equally inclined to contain a considerable number of numeric computation operations such as *linear algebra* (comprising of computations of matrices and vectors), which is required during *model training*. Linear algebra is also usually the most computationally intensive part of these applications in general. Although traditional RDBMS implementations inherently support and optimize relational operations, the same is not true for operations such as linear algebra. This is compounded by the fact that SQL has very little provisions for supporting operations outside of the relational spectrum.

The immediate solution from the data science community was to build or leverage statistical systems that support and optimize numerical computations. As these systems are external to the database, the intuition was to perform any required relational operations first inside the database, before extracting the data into the statistical system and continue with the numerical computations in the statistical system. This basically creates a two-tiered system. However, such separation of tasks is not as easy in practice. Developers of conventional analytics applications can often leverage the insight captured in the data models to determine the tables and attributes of interest to them to implement the solution to a specific business logic. On the other hand, a data scientist is often "exploring the unknown", trying to find patterns across data sets or construct models to predict certain outcomes. For the most part, it is not possible for them to determine the required attributes right at the beginning of the learning process. This leads to an iterative cycle of exploration and refinement of approaches, an aspect that has been highlighted by the machine learning community as an important characteristic of machine learning projects [Dom12].

This essentially means that the "separation of responsibilities" approach between the RDBMS and the statistical system is not often pragmatic. For a data scientist, a realistic progression of the analysis could involve going back and forth between relational and statistical operations. As an example, if part of the data set is already in the statistical system, which has undergone some processing there and now the data scientist realizes that they need to enrich it with some additional data, i.e., join with other data sets in the database, how do they go about it? The potential options to address such situations are to either (i) equip statistical systems with some traditional database functionality so that additional data sets can be loaded into them as needed without relying on the RDBMS anymore for relational support or (ii) engineer database implementations to incorporate support for numerical computing and such to perform the complete analysis *near-data*.

Naturally, this has led to a significant amount of research and development resulting in the evolution of numerous systems. In the remainder of this chapter, we will discuss some of the common approaches and popular systems in this aspect, but this is not meant to be an exhaustive list. We encourage an avid reader to refer to [BKY19] for a more in-depth study on the intersection of data management and machine learning systems.

2.3 Database-external Advanced Analytics Systems

The most popular contemporary data science tools and frameworks take the data outside of the database into a statistical system for the users to perform their analysis. Based on the complexity and intended scale of these systems some could be set up on a user's computer, while others are devised to be installed on dedicated servers, often leveraging distributed computing capabilities.

2.3.1 End-user Based Systems

End-user based systems are environments that a data scientist can typically set up in their workstation to perform their analysis locally.

The forerunners of this category are dedicated programming languages and environments that have traditionally served the numerical computing needs of the scientific community much before the vigorous growth of data science and machine learning fields. Examples of these include specialized languages for numerical and statistical computing such as FORTRAN [BKK+81], R [IG96], MATLAB [MMS17], GNU Octave [EBH07], and specialized workbenches like WEKA [HDW94b]. Julia [BEKS17] is a more recent addition to this list that attempts to improve the performance of numerical computations by employing modern compiler approaches to the language design.

The increase in popularity of data science and machine learning has also resulted in the development of packages and libraries intended to augment general-purpose programming languages. For instance, MULAN [TSXVV11], MOA [BHKP10], and Java-ML [APS09] are machine learning libraries that have been developed for Java based systems. Perhaps the most iconic example of this aspect is the emergence of Python as the most popular language of choice for data scientists [Pia17, DCR14], augmented by various packages such as NumPy [VCV11], Scikit-learn [PVG+11], or pandas [McK11].

NumPy was originally developed for scientific computing and provides support for multi-dimensional arrays such as matrices, as well as various high-level mathematical functions (e.g. Fourier transform) commonly used in scientific analysis. It provides faster performance than pure Python alternatives as its underlying data structures and functions are implemented in optimized C code that does not incur the overhead of Python's interpreted syntax [BBC+11]. Scikit-learn is a machine learning library that is popular within the data science community and consists of many off-the-shelf learning algorithms that data scientists can use to construct some of the well-known machine learning models. It internally leverages NumPy's capabilities for numeric computing. pandas is a library that provides support for data manipulation in Python. Its implementation rose from the need to provide database-like processing capabilities on data sets inside statistical environments, which as we pointed out in Section 2.2.4, became a necessity due to the exploratory nature of advanced analytical projects. pandas internally uses NumPy data structures.

When using such end-user based environments, data scientists can load data from an RDBMS into these systems using one of the various standardized API drivers that we discussed in Section 2.2.2, similar to how conventional database applications function in a two-tiered architecture. While many of them leverage existing standards like ODBC/JDBC, others have come up with their own variations (e.g. Python DB-API [Kuc98]) which are similar in principle. However, un-

like conventional database applications, where the data transfers are usually small, because often only aggregate information is transferred, data scientists need to work with detail level data that can be quite large and therefore incur noticeable transfer overhead in terms of network delays. Further, as user workstations are not typically equipped with powerful hardware compared to RDBMS servers, such arrangements can force them to work with a subset of data. This data sub-setting can be counterproductive, as having a larger data set can reduce algorithm complexity and increase its accuracy [Dom12]. Sampling is also not ideal for applications dealing with competitive business use cases [HRS+12] and is therefore not favoured by data scientists [CDD+09].

One positive aspect of these systems is that while they may not be always optimized for system performance, they are very user-friendly to work with. Specifically, they provide support for incremental, iterative development that accommodates a trial-and-error exploration path, giving consideration to the productivity of the *human-in-the-loop*. A most notable example in this aspect is the ease with which data science exploration can be performed using Python + pandas in a Jupyter Notebook [KRKP+16] Integrated Development Environment (IDE).

Example 2.2. Figure 2.6 shows such an example exploratory workflow using Jupyter Notebook, where the data scientist is "exploring" a TPC-H benchmark [Tra17] database. In the first *cell* of the Jupyter Notebook, they import the pandas package as well as the Python DB-API driver for MonetDB, pymontedb and establish a connection to the database. In the next cell, they first load the `SUPPLIER` table into a pandas DataFrame object (`supplier`). This is followed by a relational *aggregation* operation to calculate the number of suppliers and their total account balance per country that is then stored in a new DataFrame `si`.

As we will discuss further in Section 2.3.2, for pragmatic reasons, many statistical packages such as pandas are equipped with basic relational support to facilitate such relational operations over data already loaded into them. In the third cell, a *selection* condition is applied to confine the aggregated information to countries with more than 2 million in balance. At this point, the data scientist decides to verify the results and prints a sample of records. If there is a logical error, they can go back and update the code in the appropriate cells and just re-execute them. Further, if there is a syntax or any runtime error in this cell, they will not lose the computations performed in the preceding cells and can merely address the cause of the error and continue from there. Finally, in the fourth cell, they scale the representation of the number of suppliers into hundreds and that of the total balance into millions through a linear algebra division operation using a NumPy array. At this point, the DataFrame stored in `si` has the intended result. Once again, this step is verified for correctness by displaying some sample records from the DataFrame, which as we can see, encapsulates data that is logically "table-like".

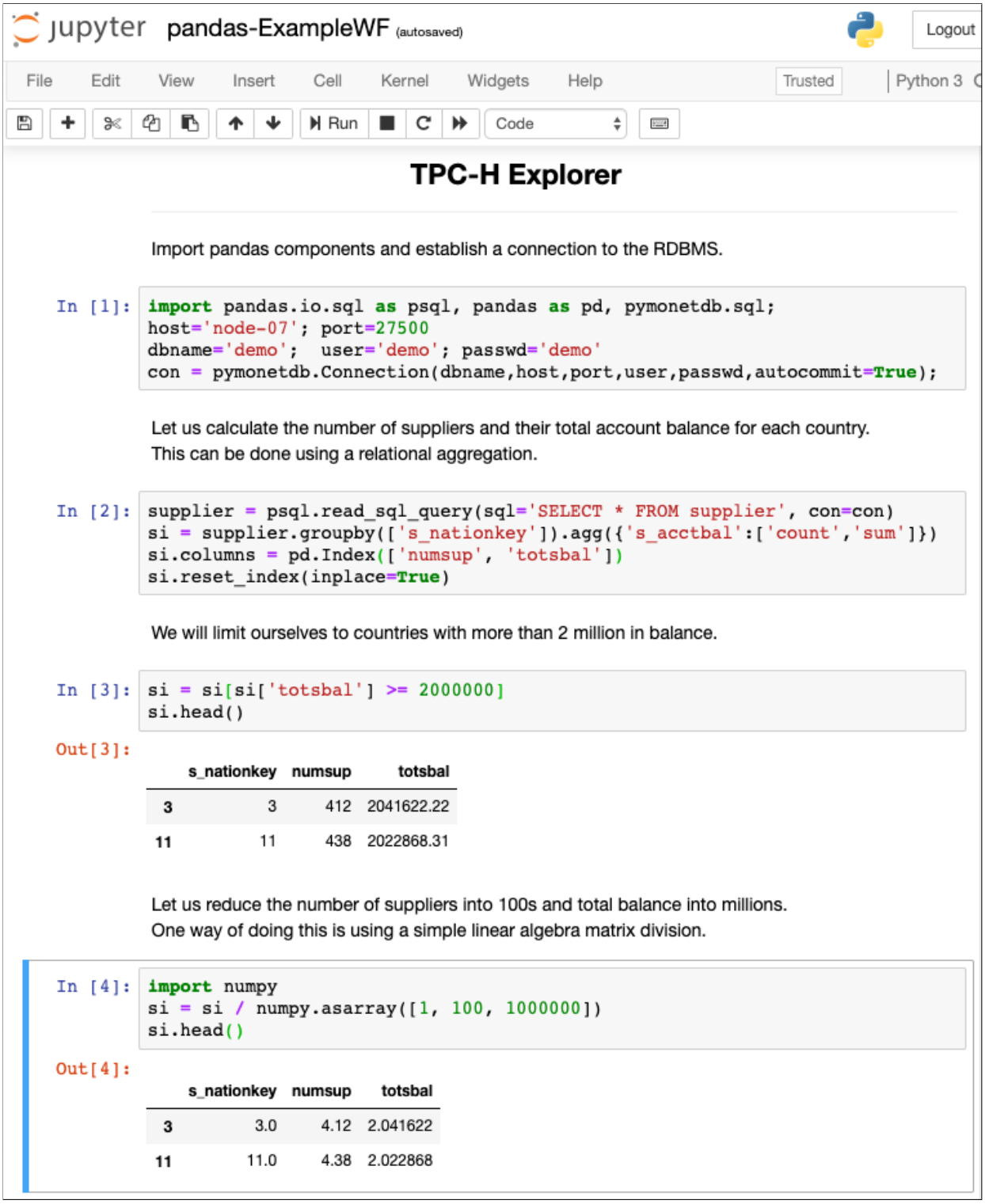


Figure 2.6: Pandas - example workflow in Jupyter Notebook

Looking back at the workflow in the above example as a whole, it is important to note that for other than the first data retrieval step from the database, the database has no more involvement in the workflow, as all the ensuing relational operations on the data are performed by the pandas library at the client-side. Further, as demonstrated by the above example, using programming environments such as this, data scientists can thus perform a statement-at-a-time incremental exploration, verifying intermediate outcomes as needed, before making decisions about what transformations to perform next. Their popularity has been thus ascribed to this *agile development* experience that they provide [SAFFGA18, YLP+17]. Due to their relevance to our research from the usability and functionality perspective, we will be comparing and contrasting our work with some of these popular implementations in our evaluations.

2.3.2 Relational Influence in Statistical Systems

As pointed out in the previous section, in their most basic form, analytical systems, whether they are the end-user based systems that we just discussed, or the *Big data* implementations used for distributed data processing that we will cover in the next section, load all the data they need into their internal data structures and libraries. When the source data resides in an RDBMS, this results in an initial data transfer from the database to the analytical system.

However, as we mentioned in Section 2.2.4, incremental data exploration is often the norm in machine learning projects. Therefore, many analytical systems and Big data frameworks also provide basic relational primitives including *selection* and *join*. This helps users to integrate any additional data to enrich the set of features that they have already constructed. It is also useful if they have to perform any further relational transformations on the data sets that they have already loaded, as we saw in the example pandas workflow in Figure 2.6.

Prominent examples of relational support in analytical systems include the DataFrame objects in R [R C14], pandas [McK11] package for Python, and Spark [ZXW+16]. Extensions to some of these implementations also provide a SQL-like syntax to express relational operations in a declarative way [AXL+15, Lam19]. Although not as optimized as an RDBMS, the fact that linear algebraic operations usually occupy the lion's share of the computation-time makes this an acceptable compromise for the data scientists.

2.3.3 Distributed Implementations

Since the processing power of user workstations are often not enough to tackle large scale analysis, there is the need for dedicated computing infrastructure for such tasks. As a result, the past decade has also seen the rise of several open-source general-purpose cluster computing frame-

works for *Big data* analysis starting with Google's MapReduce approach [DG08] and its open-source equivalent, Hadoop [GMSS15], followed by Spark [ZXW+16]. These frameworks facilitate computations that leverage *data parallelism* while providing *fault tolerance* through programming paradigms that mask the complexity traditionally associated with such distributed processing approaches. For example, Spark provides an abstraction called Resilient Distributed Dataset (RDD) that encapsulates data sets that are distributed over a cluster of machines [ZCF+10]. Operations performed over RDDs are internally parallelized by the framework, which also provides automatic recovery from failures of individual machines involved in the computation.

Since the rapid growth in data science and machine learning followed in on the wake of the emergence and popularisation of these *Big data* frameworks, they became the natural candidates for these implementations. For example, Apache Mahout [ODFA12] is an implementation of distributed machine learning algorithms that originally utilized the MapReduce framework to perform distributed computations, and has since moved on to Spark. SystemML [GKP+11, BDE+16] is similarly a declarative language interface for constructing machine learning algorithms that internally leverages the distributed computational paradigms offered by MapReduce and Spark. Spark MLlib [MBY+16] is a distributed machine learning framework that is built on top of Spark and is currently one of the most popular distributed frameworks used for machine learning [Kag17].

While such dedicated cluster-based Big data frameworks provide more computing powers, users still have to load data sets into them from the respective source database systems. These frameworks also rely on the traditional ODBC/JDBC type standard APIs to interact with the database systems and transfer data. Along with the fact that large data transfers are inherently impacted by network delays, these implementations are also limited by the fact that these traditional database APIs were not designed with large scale data extractions in mind.

2.3.4 Optimizations

As most of the conventional numerical computational systems work on memory-resident data structures, initial attempts to leverage database optimizations into such systems focused on the I/O capabilities offered by RDBMS implementations when working with large data sets. For example, RIOT-DB extends R to leverage the I/O capabilities of an external RDBMS back end by translating expressions in R to operations in SQL by creating view definitions [ZHY09, ZZY10].

In WekaDB [ZMK+06], the authors augment a popular Java-based machine learning workbench, WEKA [HDW94a], with an RDBMS back end. They develop a storage manager that lets WEKA's algorithms run transparently on data stored in an RDBMS by providing an alternate implementation for WEKA's main-memory based data set interface. Copying of records from the

database to WEKA is performed lazily. The database access itself is performed using JDBC. As a result, the performance is several magnitudes slower for data set sizes that can otherwise fit well in WEKA's memory. MATLAB similarly uses a concept called *Tall arrays* to work with data sets that do not fit completely in main memory [HH16]. Among Big data frameworks, there has been some work done in pushing relational operations into the database, such as in Spark [DL17].

Despite such improvements in architecture, any of these *two-system* approaches still require data transfers to the statistical system. While optimizations such as *pushdown* can be useful at the beginning, when the first data is retrieved from the database, they cannot be employed if one of the data sets is already in the framework and the other is still in the database, such as could be the case at a later point in the exploration process. In such a situation, it might be more beneficial to just load the additional data into the framework and perform the relational operation, e.g. the *join*, inside the framework rather than performing a join in the database and reloading the entire data result, which can be even more time consuming should the result set be large [RM17]. As discussed in Section 2.3.2, this has been the driving factor behind the evolution of support for relational operations in analytical systems.

In short, data scientists typically have to either fetch more attributes than would be necessary and then do relational computations locally in the statistical system, or they have to perform multiple data transfers. Given that the exploration phase is often long and cumbersome, both approaches do not seem appealing. Further, as many machine learning processes employ approaches and techniques such as *feature selection* [CS14], *dimensionality reduction* [Sar14], etc., that are intended to reduce or consolidate the number of attributes to be used, it makes sense to be able to perform such data-size-reducing tasks at the source, i.e., inside the database.

2.4 In-database Advanced Analytics

As we saw in Section 2.2.4, many dedicated analytical systems have started duplicating parts of RDBMS functionality, extracting a large amount of data from the database and treating it merely as a storage layer, foregoing many of the data processing optimizations that the RDBMS community has accrued over the decades. The situation is also not ideal in terms of an optimal architecture because of the data transfer overheads. Though many organizations have started adding new dedicated infrastructure to set up external frameworks for advanced analytics, as [YTLL17] points out, having different siloed compute clusters to cater to different frameworks can underutilize the infrastructure. These shortcomings turned out to be a good motivation for the database community, who has been forced to play catch up and has started looking for ways to bring numerical computation capabilities into RDBMS implementations.

2.4.1 Case for A Unified System

A unified system offering both efficient linear algebra and relational algebra support facilitates efficient data exploration. A data scientist having direct access to the source database is better equipped with exploring the various data stored in the system, while at the same time being able to analyze, build and test prototypes quickly via a computational framework embedded into the RDBMS. Such an arrangement can essentially provide the data scientist with an *agile development* environment [LC16]. For example, a data scientist can start model development with a smaller set of features to reduce the computational cost. Additional features can be used if the desired accuracy is not obtained from the original model. Immediate availability of these additional features is made possible by computing near the data and the data scientist does not have to burden themselves with shipping all of the features in advance from the database to an external computational framework or be interrupted waiting for the additional data to be made available.

Further, most RDBMS implementations employ some amount of de-normalization for performance reasons [SS01]. This can result in data sets having many attributes that are derived, and hence correlated. In general, learning algorithms avoid including correlated features in the model. In fact, many machine learning techniques can automatically detect such patterns in the data sets irrespective of whether they are explicitly derived or naturally correlated, without burdening the data scientist [YL03, YL04]. As mentioned in the previous section, steps and techniques such as *feature selection*, *dimensionality reduction*, etc., are used for this purpose and reduce the size of the data to be modeled. Therefore, transferring a lot of data from the database to a computational system in advance without looking into such aspects is not ideal and can result in wastage of both time and resources.

Computing closer to the data also ensures data freshness. The data scientist is not burdened with the task of reloading the data into an external framework in a periodic fashion when data in the database is updated or added. This is an essential aspect also for supporting real-time analytics, which is desirable in many fields [LC16, Hal15]. Finally, having a unified system also reduces the scattering of data, eliminating the need or making it easier to establish data provenance, as this can be a challenge with a heterogeneous infrastructure [PRWZ17].

2.4.2 Specialized DBMS Implementations

Specialized databases such as SciDB [SBZB13] and its predecessors [BDF+98, MS99] evolved from the need to support several niche scientific fields working on data sets such as satellite imagery, astronomy, genomics, etc., which are altogether different in both scale and structure from the conventional domain of relational databases. These implementations use array-based data struc-

tures and query languages such as Array Query Language (AQL) to interact with the database. For numeric computations, SciDB reformats the data and offloads it to ScaLAPACK [CDPW92], an analytical package known for its efficiency with numerical operations. Both SciDB and ScaLAPACK run on the same hardware as separate processes.

As another example of a specialized solution, Machine Learning Database (MLDB) [Dat16] is a database specifically built for machine learning purposes, to address the lack of data management capabilities in the traditional statistical packages and engines. The machine learning functionalities are available via stored procedures and UDF extensions to the standard SQL (We will discuss stored procedures and UDFs in detail in Section 2.4.4). The database itself is schema-free and the data sets are append-only, stored as three-dimensional sparse matrices where time is an additional dimension along with row and column. As such, their support for conventional relational database applications is limited.

Recent studies have shown that compared to such specialized database implementations, contemporary Big data solutions such as Spark perform better on more generic workloads to which many machine learning and data science projects belong [MDZ+17, RMYC13]. Further, RDBMS implementations themselves have demonstrated to be manyfold more efficient compared to the Big data frameworks when it comes to performing traditional relational operations [PPR+09]. In short, relying on the architecture of such specialized scientific databases or Big data frameworks may not be sensible performance-wise for traditional analytics applications that depend on RDBMS implementations currently. Therefore, while some of these implementations may perform well for data science workloads, they will have to be set up as an auxiliary database system that is updated with data from a conventional source RDBMS. In practice, this will make them yet another form of a database-external framework along with its associated drawbacks.

2.4.3 Extending SQL for Linear Algebra

As SQL already provides many functionalities required for feature engineering such as joins and aggregations, exploiting them in the exploration phase is an interesting prospect. Recent research proposes to integrate linear algebra concepts natively into RDBMS by adding data types such as vector and matrix, and extending SQL to work with them [ZKM13, LGG+17, ALOR17, ALOR18]. However, we believe that they might not fare that well in terms of usability. For example, most linear algebra systems use fairly simple notations for matrix multiplication such as:

```
res = A × B
```

In contrast, the SQL equivalents as described in the works mentioned above require several lines of SQL code which might not be as intuitive [BKY19].

Example 2.3. Listing 2.2 is a SQL syntax based on [ALOR18] that performs the multiplication of a matrix by itself. The matrix is stored as a database table that has three columns – columns `i` and `j` indicate the row and column dimensions of each element in the matrix, and `v`, its value.

Listing 2.2: A SQL-based Syntax for Matrix Multiplication [ALOR18]

```
SELECT m1.i, m2.j, SUM(m1.v * m2.v)
FROM matrixTable AS m1, matrixTable AS m2
WHERE m1.j = m2.i
GROUP BY m1.i, m2.j
```

Another inconvenience, which is not obvious at first sight but equally important, is the lack of proper support to store and maintain the results of intermediate computations in above proposals, as users need to explicitly create new tables to store the results of their operations, adding to the lines of code required. This is akin to how in standard SQL one has to create a table explicitly if the result of a query is to be stored for later use (in Listing 2.2, the database returns the results to the client). However, unlike SQL applications, learning systems often go through several thousands of iterations of creating such temporary results, making this impractical. Additionally, RDBMSes generally treat objects as persistent. This is an extra overhead for such temporary objects and burdens the user to perform explicit cleanup[1]. Procedural HLLs such as Python, R, etc., on the other hand, provide much simpler operational syntax and transparently perform object management by automatically removing any objects that are not in use.

As others have pointed out, SQL is awkward to use and lacks many HLL programming features when it comes to non-relational computations such as those that are conventionally supported by HLLs [FG16, BKY19, ZZY10]. The inability of the database to satisfy all of their computational needs even when using such extensions can be a turnoff for many users as they still have to rely on an HLL to perform the rest of their analysis [ZZY10].

2.4.4 HLL UDFs and Stored Procedures

User Defined Functions (UDFs)

UDFs are a well-established mechanism to extend traditional RDBMS functionality [LKD+88]. Figure 2.7 shows a high-level layout of the UDF approach. As we will see in a concrete example

[1]The modern RDBMS concept of *temporary tables* [AK+17, Sch05] only address a small part of the problem.

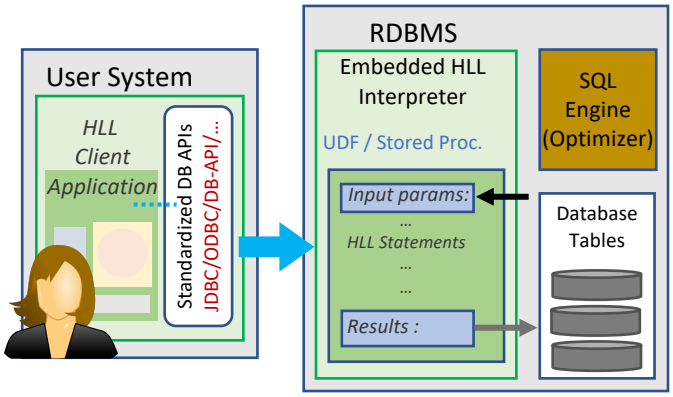


Figure 2.7: Concept of a HLL UDF/Stored procedure

shortly, a UDF is a program snippet written in an HLL, that can be invoked within the RDBMS execution space through a SQL API. Many modern programming language interpreters that are popular with data scientists, such as Python or R, support embedding themselves as part of a larger application [Pyt18, R C99], and RDBMS implementers are increasingly making use of this feature to support UDFs implementations [Mil16, WDG+16, ML13, RM16, The17]. UDFs may or may not take any input parameters, but they are always expected to produce a (possibly empty) output data set. This is akin to how all the relational operators produce an output data set. Therefore, UDFs are easily integrated into SQL statements. When a UDF is invoked as part of a SQL statement, the RDBMS execution engine will leverage the embedded HLL interpreter libraries to execute the UDF source code (see Figure 2.7). Thus, in a nutshell, UDFs provide a computationally efficient mechanism to execute user-defined functionality *near-data* using an HLL. UDFs come in a few variations, often determined by their purpose and usage. A detailed overview is given in [Raa15]. We will limit our discussion in this section to some of the UDF concepts that are relevant to our work.

Given that the UDF concept allows the execution of arbitrary HLL code inside the RDBMS, with the emergence of data science applications, HLL based UDFs have been proposed as a promising mechanism to support in-database analytics including linear algebra operations and machine learning. While implementations such as [LKD+88, LM14, RM16, RHMM18] focus on extending the RDBMS to provide a generic facility for users to write their own UDFs, others such as [OP11], BISMARCK [FKRR12], or MADlib [CDD+09, HRS+12] focus on providing off-the-shelf *learning* algorithms in the form of UDFs for users to directly use in their analysis.

However, so far, they have not found much favor in advanced analytics applications. We believe that this has to do with their usability. As mentioned before, learning is a complex process and can have a considerable exploration phase. This can include a repetitive process of fine-tuning the data

set, training and testing an algorithm, and analyzing the results. The user is actively involved in each step as a decision-maker, often having to enrich/transform the data set to improve the results, or changing the algorithm and/or tweaking its parameters for better results.

UDFs lack the agility for such incremental development for several reasons. First, in UDF implementations, the user has to write the complete workflow of the program before any of it can be executed. This is not feasible for someone who is exploring, as they would generally plan their strategy for the next transformation steps based on their assessment of outcomes of the previous steps. Further, the only way to interact with a UDF is via its input parameters and final output results due to their procedural nature (see Figure 2.7). This prevents the possibility of any intermediate feedback to the user, as is demonstrated by our next example.

Example 2.4. If we are to implement the Jupyter Notebook based Python + pandas workflow in Figure 2.6 as a UDF, it will look like the definition in Listing 2.3, which is based on the MonetDB syntax for Python UDFs [Raa15].

Listing 2.3: UDF example

```
1  CREATE FUNCTION TPCH_Explorer()
2  RETURNS TABLE(s_nationkey INTEGER, numsup FLOAT, totsbal FLOAT)
3  LANGUAGE PYTHON
4  {
5    import pandas as pd
6    #Load supplier table from the database.
7    supplier = pd.DataFrame(_conn.execute('SELECT * FROM supplier;'))
8
9    #Calculate the number of suppliers and their total account balance for each country.
10   si = supplier.groupby(['s_nationkey']).agg({'s_acctbal':['count','sum']})
11   si.columns = pd.Index(['numsup', 'totsbal'])
12   si.reset_index(inplace=True)
13
14   #Confine the aggregation to countries with more than 2 million in balance.
15   si = si[si['totsbal'] >= 2000000]
16
17   #Represent the number of suppliers in 100s and total balance in millions.
18   import numpy
19   si = si / numpy.asarray([1, 100, 1000000])
20
21   return si
22 };
```

Similar to the original workflow in Figure 2.6, the UDF first loads the supplier data from the database into a pandas DataFrame (line 7). It then performs a relational aggregation over

the DataFrame to calculate the country level account balance (lines 10-12) and then restricts the data to those countries with above 2 million in total account balance (line 15). However, unlike the original workflow, where the data scientist could at this juncture verify the results of this intermediate step, the UDF has no provision for this. Finally, once the aggregated supplier information is scaled down numerically (line 19), the results are returned to the database engine.

This result returned by the UDF to the database engine, `si`, is a pandas DataFrame, and thus, has a "table-like" format. Therefore, it can be used in a manner similar to regular database tables in SQL statements. For instance, once it is created in the database, the above UDF can be executed through SQL APIs as follows:

```
SELECT * FROM TPCH_Explorer()
```

UDFs such as the above example, that return a "table-like" data structure to the database engine are commonly referred to as *table UDFs* [RM16]. Table UDFs allow embedded HLL programs to expose non-database *tabular* data sets to the SQL engine of the RDBMS, so that the later can be used to run SQL queries over them [XKG10, HLR02, OHB+11]. As can be seen, syntactically, UDFs can be easily plugged into SQL, requiring only modest language extensions. Although this makes them popular with RDBMS implementers, as we have seen, they are not conducive for exploratory work, and hence have not found favor with data scientists.

Another drawback of UDFs is that all host language objects inside a UDF are discarded after execution. If they are required for further processing (as in another UDF invocation in the same client session), they need to be stored back into the database as part of the UDF execution process. In order to accomplish this, the user needs to explicitly write code to persist the objects to a database table [Raa16], and additional UDFs for later retrieval [RHMM18]. This is an overhead and a significant development effort that is not related to the actual problem that the users are attempting to solve, but attributed to the framework's limitation. UDFs are also often confined in terms of the number and data types of its input and output parameters even though the HLL itself may support such *overloading* concepts. This hinders users from developing generic UDF-based solutions to their learning problems [RM16].

Therefore, even though HLL environments such as Python themselves are very flexible and agile as we discussed in Section 2.3.1, the restrictions imposed by the UDF constructs hamper their agility. Further, as we discussed in Section 2.2.1, and also shown in Figure 2.7, real-world applications anyways require an HLL to be the "driver" managing the control flow of the system. As a result, there is little incentive for the users to take the HLL – SQL/RDBMS – HLL/UDF approach, especially if the UDF is to be written in the same HLL as the parent application and comes

with many hassles that are caused by the SQL/RDBMS component being in the middle. Database-external analytical systems, on the other hand, are architecturally more simplistic, constituting of only an HLL – SQL/RDBMS interaction that does not impede the flexibility provided by HLL constructs. Therefore, while computationally efficient, in their rudimentary form, we believe that UDFs are not the most convenient approach for data scientists to adopt for their exploratory work.

Stored Procedures

Stored procedures were introduced in early RDBMS implementations to make up for the limitations of SQL that lacks many conventional HLL programming constructs such as *loops*, *conditional statements*, etc. They were introduced as an extension to the SQL syntax [FP05, DW02]. This allows programmers to write compact code that contains both regular SQL and some HLL-like control constructs and that is executed by the RDBMS. Similar to UDFs, stored procedures can accept input arguments and return a result. However, in general, most stored procedure implementations allow only a single value to be returned, which is usually a status code or a message. This is because their original intent was to provide language constructs to use SQL also as a "driver" for the control flow of the application, instead of relying on an HLL language for that purpose. A typical SQL invocation of a stored procedure is as follows.

```
CALL dowork_storedproc(...);
```

Therefore, unlike UDFs which can be part of complex SQL statements and can be operated upon like a regular table, stored procedures are executed standalone.

Stored procedure implementations come in two flavours. Originally, RDBMS vendors developed their own language extensions to SQL to support some HLL constructs. However, they offered only limited capabilities compared to general-purpose HLLs such as Python, C++, etc., especially lacking in *object-oriented* concepts and the versatility of built-in libraries. Further, unlike SQL itself, the syntax and semantics for stored procedure extensions vary significantly between the RDBMS implementations, thus providing less incentive for the user community to adopt. As a result, with the advent of embedded HLL interpreters, many modern RDBMS implementations simply use an HLL for supporting stored procedures, as these are already quite versatile and popular with the user community. Therefore, the practical distinctions between many UDF constructs and stored procedures have become blurred, as their internal logic is built upon the same HLL constructs, providing similar capabilities. In fact, in an attempt to further simplify the implementation logistics, many modern RDBMSes such as MonetDB, do not provide an explicit concept of an HLL stored procedure, as for all practical purposes, a table-UDF returning just one column with

a single value can be treated as a functional replacement for the stored procedure concept. The invocation syntax will thus remain the same as that of a table-UDF, as shown below.

```
SELECT * FROM dowork_storedproc(...);
```

As such, in order to simplify the terminologies used in our discussions, we will refer to code segments written in an HLL, producing a result set data structure (such as a table) as its output, possibly intended to be used as part of a complex SQL statement or data processing workflow, as a *UDF*. Code segments that contain a significant amount of workflow logic, and mostly return just a status code, whose purpose is to be invoked in standalone, will be termed as stored procedures. It should be noted that, given their "procedural" nature, stored procedures suffer from the same limitations as UDFs when it comes to support for exploratory use.

So far, our discussion on UDFs and stored procedures were focused around their usability characteristics in the context of exploratory analysis. We will be revisiting them again in Chapter 6 when we take a deeper look into the performance aspect of integrating HLL functionality into database query optimization, which is a quite challenging endeavor.

2.4.5 Optimizations

The concept of mixing HLL and relational programming paradigms has resulted in new optimization challenges and associated research works.

Data Handover

As RDBMS implementations and HLLs evolved independent of each other, a major challenge in this aspect is the efficient integration of these systems that take into account their differences in underlying data structures and primitive data types.

In RICE [GLW+11], authors attempt to integrate the statistical computation framework R with an RDBMS by executing the R process in tandem with a SAP RDBMS engine on the same node. They optimize the data transfer between the two processes by using shared memory. While this reduces any serialization overhead compared to performing a network transfer, as [LM14] points out, it still incurs data copying overhead between the process spaces.

In [LM14], R is integrated with an embedded version of MonteDB [Raa18] to construct a unified system. They accomplish this by making minor modifications to the high-level data structures and memory management routines associated with both the systems to make them compatible with both R and MonetDB in an agnostic manner. They capitalize on the fact that MonetDB [IGN+12]

is a columnar RDBMS [ABH09] and uses C-style array-based data structures for memory buffers which bears similarity to computational structures used by R. As a result, the RDBMS is able to hand over results from database queries to the R environment without making any explicit copies of the original data in the database, a concept they term *zero-copy*. As the implementation is based on embedding the database into an R process, this approach cannot be utilized for standalone RDBMS implementations that are shared across multiple applications and users, which is the norm. More recently, Apache Arrow [Din16] uses a shared memory implementation to facilitate applications that conform to its API to share data between them without making copies.

Other Related Works

A recent work, EmptyHeaded [ALOR17], and it's extension LevelHeaded [ALOR18] attempt to find an acceptable architectural compromise between relational and statistical systems to improve performance. Linear algebra is supported through a SQL-based syntax, similar to prior works [ZKM13, LGG+17] that integrates linear algebra into RDBMS implementations. However, they use a trie based data structure [Bra08] that permits join operations only over key columns, restricting the use of complex analytical SQLs such as those containing subqueries.

As discussed in Section 2.1.3, traditional RDBMS execution engines are designed to optimize relational operations. However, HLLs follow a different programming model and associated optimization techniques compared to SQL. Therefore, this difference in programming constructs poses challenges as to how to optimize such "mixed" program snippets. There have been proposals such as Weld [PTS+17], TUPLEWARE [CGD+15], Froid [RPE+17], Lara [KKS+19], and HorseIR [CDC+18], that develop a common Intermediate Representation (IR) to which all the SQL queries and HLL code are first translated to, thereby facilitating an optimization at this IR level. The research in this area is still in its infancy and requires major RDBMS redesign [Var18]. Besides, as such implementations require the complete workflow to be made available for the optimization algorithms to work, they cannot be employed for exploratory frameworks where the data scientist needs the capability to incrementally execute transformations.

More recently, the iterative nature of machine learning and data science projects have received some attention. HELIX [XMM+18, XML+18] takes this into account to optimize the execution of workflows across multiple iterations. It performs a heuristics-based caching of results to reuse them across different executions of the workflow.

2.5 Background Summary

The evolution of database-external systems has been more or less driven by the immediate needs of the data science community, and as a result, a considerable amount of focus has been given to improve the productivity of the *human-in-the-loop*. However, due to the complex nature of the life cycle of data science projects, the separation of responsibilities between the statistical system and the RDBMS has been blurry at the best. Therefore, contemporary implementations treat the database as a storage layer by just extracting the data out of the database and have nothing else to do with it afterward, often re-implementing some of the RDBMS functionalities themselves. Such approaches have been suboptimal due to their lack of efficient relational optimization engines as well as transfer overheads associated with large data sets.

Efforts by the database community to build RDBMS-centric data science frameworks, on the other hand, have been primarily focused on improving the execution efficiency of bundling relational and numeric computations together in the RDBMS. User-friendliness has not received as much attention compared to contemporary popular database-external frameworks. As a result, the data science community has been reluctant to shift towards using database-centric approaches for their analysis.

3

The Design & Implementation of AIDA

In this chapter, we present AIDA, a Python based in-database framework for data scientists, that facilitates both relational operations and linear algebra through a unified data abstraction. What makes AIDA's approach novel is that AIDA allows data scientists to perform analysis following the statement-at-a-time, incremental, iterative, approach, offering the same degree of usability supported by the popular database-external implementations while being *near-data*, similar to in-database approaches. In order to accomplish this, AIDA provides a client side API that follows the syntax and semantics of popular database-external systems. However, unlike such systems where the data need to be extracted from the database prior to performing any computations on it, AIDA cleverly transfers the computational logic to the server side, near-data, ensuring efficiency similar to contemporary in-database solutions.

In the following, we will first provide a high-level overview of the client-server architecture of AIDA. After that, we will discuss the core data abstraction of AIDA, *TabularData*, that provides a unified abstraction over data sets allowing to perform both relational and linear algebra operations over it. We will then explore the internal data structures used by *TabularData* and discuss the various optimization techniques employed by AIDA. Finally, we will look into the Distributed Object Communication (DOC) framework of AIDA that facilitates seamlessly shifting computation requests from the client side to AIDA's server that resides inside the RDBMS.

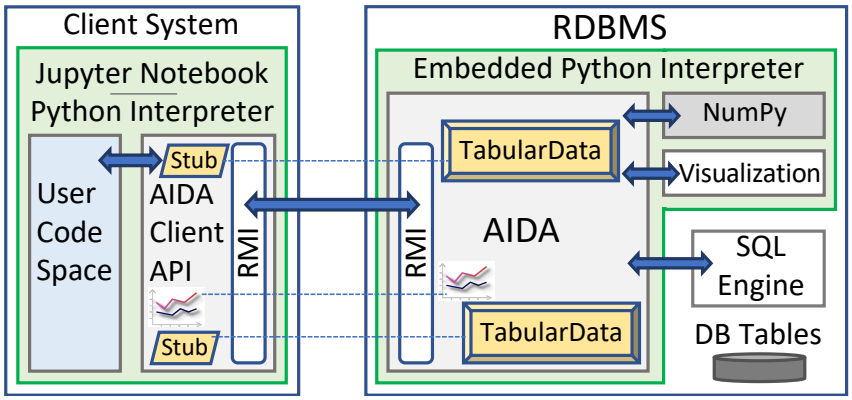


Figure 3.1: AIDA - conceptual layout

3.1 AIDA Overview

3.1.1 Conceptual Architecture

Figure 3.1 depicts a high-level conceptual layout for AIDA with its client-side API library and the server-side component. AIDA's server is embedded in the RDBMS, more precisely, within the Python interpreter embedded in the RDBMS, thus sharing the same address space as the RDBMS. This makes AIDA easily portable since embedded Python interpreters are becoming increasingly common in modern RDBMSes. Embedded interpreters also help us leverage some of the RDBMS optimizations already available in this area, as we will later discuss.

Users can connect to AIDA's server using a regular Python interpreter or more popular data science IDEs like Jupyter Notebook that works with Python, as long as they are equipped with AIDA's client API library. We decided to leverage Python because Python interpreters are ubiquitously available in most data scientists' computing systems as they are often used to work with popular statistical packages such as NumPy and pandas. Using AIDA's client API, data scientists can write Python programs for exploration in an iterative, interactive manner, as they do with these other statistical packages. Figure 3.2 is a screenshot from an interactive workflow written using AIDA's client API in a Jupyter Notebook, and as we can see, bears a lot of resemblance with the similar workflow that we saw in Figure 2.6 which is written using pandas. We will be discussing the syntax and semantics of AIDA's client API in later sections.

Further, as we will see in a use case in the evaluation, using such a generic language such as Python and IDEs like Jupyter Notebook also provides opportunities to the users for integrating other Python based packages into their workflows. Additionally, this minimizes the learning curve for users compared to an approach where we would propose a new Domain-specific Language (DSL), improving the chances of adoption.

Figure 3.2: AIDA - example workflow in Jupyter Notebook

When using AIDA, data scientists use their computers as clients to interact with the data and implement any necessary programming logic, but any data transformations and computations that they initiate are not executed on the client side. Instead, AIDA's client API sends them transparently to the server and receives a *remote reference* which represents the result that is stored in AIDA's server. We implement this interaction between AIDA's client API and server in the form of RMIs, whose details we cover in Section 3.4. RMI is a well-established communication paradigm and known to work in practice. Others have effectively employed similar mechanisms to push computations into a remote database [HLZ+03]. This will also allow us in the future to easily extend AIDA to be part of a fully distributed computing environment where data might be distributed across many RDBMSes.

3.1.2 Data Transformations

[KBY17] lists a seamless integration of relational algebra and linear algebra data transformations as one of the current open research problems. They highlight the need for a holistic framework that supports both the relational operations required for the feature engineering phase and the linear algebra support needed for the learning algorithms themselves. AIDA accomplishes this via a unified abstraction of data called *TabularData*, providing both relational and linear algebra support for data sets.

TabularData: TabularData objects, as shown in Figure 3.1, reside in AIDA's server, and therefore in the RDBMS address space. They remain in memory beyond individual remote method invocations. TabularData objects belong to a class of AIDA objects called Database Memory Resident Objects (DMROs). We will discuss DMROs in detail in Section 3.4.1. TabularData objects can work with both data stored in database tables as well as host language objects such as NumPy arrays. Users perform linear algebra and relational operations on a TabularData object using the client API, regardless of whether the actual data set is stored in the database or in NumPy. Behind the scenes, and as shown in Figure 3.1, AIDA's server utilizes the underlying RDBMS's SQL engine to execute relational operations and relies on NumPy to execute linear algebra. When required, AIDA performs data transformations seamlessly between the two systems (see Figure 3.3) without user involvement. We will discuss the interplay between these two systems in detail over the next sections.

Linear algebra and relational operations: AIDA cashes in on the influence of contemporary popular systems for its client API. For linear algebra, it simply emulates the syntax and semantics of the statistical package it uses: NumPy.

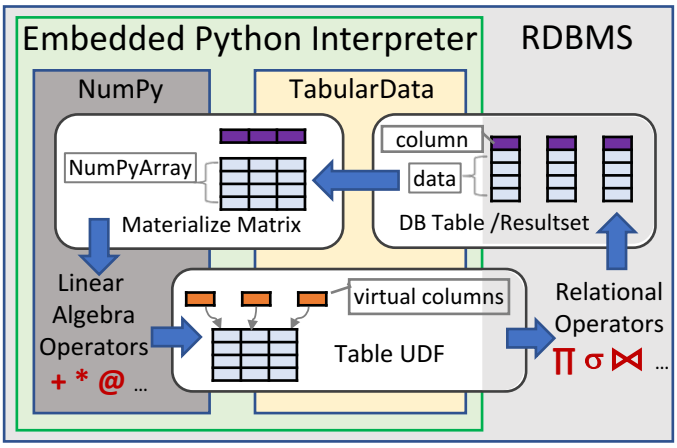


Figure 3.3: TabularData Abstraction

For relational operators, we decided to not use pure SQL. As others have pointed out, such approaches of embedding one language in another (*polyglot programming* style) are awkward to work with [ORS+08, US19], and in our case will make it difficult to provide a seamlessly unified abstraction. Instead, we resort to Object-Relational Mappings (ORMs), which allow an object-oriented view and method-based access to the data in database tables [MAB08]. While not as sophisticated as SQL, ORMs are fairly versatile. ORMs have shown to be very useful for web developers, who are familiar with object-oriented programming but not with SQL. ORMs make it easy to query the database from a procedural language without having to write SQL or work with the nuances of JDBC/ODBC APIs. A recent study indicated that such an approach can be more programmer-friendly compared to the polyglot style due to the large semantic and syntactic distance between SQL and a typical HLL [US19].

Similar object-oriented approaches to exposing relational operations have been also employed by other popular data science tools such as Spark [ZXW+16], pandas [McK11], and R [R C14] for their DataFrame abstraction. By borrowing syntax and semantics from ORMs – we mainly based our system on Django's ORMs module, a popular framework in Python [Dal07] – we believe that data scientists who are familiar with Python and NumPy but not so much with SQL, will be at ease writing database queries with AIDA's client API.

3.1.3 Overview Example

To have a preliminary understanding of how a data scientist could go about analyzing the data stored in the database using AIDA's client API, first, let us take a look at two simple code snippets. A more detailed reference for AIDA's client API is provided in Appendix A.

As AIDA's client API consists of regular Python statements, they can be run in any client-based Python interpreter or even a Jupyter Notebook in an interactive manner. The first code snippet, taken from the example AIDA workflow in Figure 3.2, represents a relational operator as it accesses the `supplier` table of the TPC-H benchmark to calculate the number of suppliers and their total account balance for each country. Here, `supplier` is an object reference to the underlying database table and `agg` is a built-in API provided by AIDA for TabularData objects that represents an aggregation method with appropriate input:

```
si = supplier.agg(('s_nationkey'
                  ,{COUNT('*'): 'numsup'} ,{SUM('s_acctbal'): 'totsbal'})
                  ,('s_nationkey',))
```

The client API of AIDA ships this relational transformation to AIDA's server which returns a remote reference to the client for a TabularData object that represents the result set. The client program stores this reference in the variable `si`. At this point, the data scientist can choose to apply yet another relational transformation or a linear algebra operation over `si`.

The second code snippet converts the number of suppliers to hundreds and the total account balance to millions via a linear algebra division using a NumPy vector[1]:

```
res = si / numpy.asarray([[1, 100,1000000]])
```

The client API ships this linear algebra transformation again to AIDA's server, including the NumPy vector and the reference of `si`, and receives the object reference to the result of this division, which it stores in the local variable `res`.

At the server side, AIDA executes the relational operation (first code snippet) by first generating the SQL query to perform the required aggregation and then using the SQL engine of the RDBMS to execute it. The TabularData object represented by `si` will point to the result set created by the RDBMS (see right top result set format in Figure 3.3). As this object becomes the input of a linear algebra operation (second code snippet), AIDA will transform it into a matrix (see the top left matrix in Figure 3.3), and perform the division operation using NumPy. It then encapsulates the resulting matrix in a TabularData object, returning a remote reference to the client API, which the client stores in the variable `res`.

Should `res` become the input of a further relational operation, AIDA server needs to provide the corresponding data to the SQL engine for execution. AIDA's server achieves this by transparently exposing the data through a *table UDF* to the RDBMS (see the bottom of Figure 3.3). As we discussed in Section 2.4.4, table UDFs are quite handy to expose table-like data sets to the SQL

[1] Although this can be computed using relational/SQL operations as well, we are using a trivial example for the sake of brevity and easiness of demonstrating the concepts.

engine in order to execute queries over them. However, it must be noted that the client is not aware of this UDF-based transfer. That is, the internal UDF-based implementation provided by AIDA's server is transparent to the client.

As AIDA's server transparently handles the complexity of moving data sets back and forth between the RDBMS SQL-engine and the NumPy environment (see Figure 3.3) and hides the internal representation of TabularData objects, it allows client programs to use relational and linear algebra operations in a unified and seamless manner. Programmers are not even aware of where the execution actually takes place.

3.2 The TabularData Abstraction

In this section, we will discuss in detail the TabularData abstraction and the co-existence of different internal representations, how and when exactly linear algebra and relational operations are executed, how AIDA is able to combine several relational operators into a single execution, and how and when it is able to avoid data copying between the execution environments.

All data sets in AIDA are represented as a *TabularData* abstraction, that is conceptually similar to a relational table with columns, each having a column name, and rows. Thus, it is naturally suited to represent database tables or query results. Also, two-dimensional data sets such as matrices that are commonly used for analytics can be conceptually visualized as tables with rows and columns where the column positions can serve as virtual column names.

TabularData objects are *immutable* and applying a linear algebra/relational operation on a TabularData object will create a new TabularData object. This is conceptually similar to Spark's data abstraction of RDDs [ZCD+12]. Immutable TabularData objects also facilitate optimizations such as sharing internal data structures between similar TabularData objects as well as performing lazy transformations. This is because in many cases we can simply keep track of a *lineage* of transformations along with their corresponding source TabularData objects and not actually execute any transformation until its data is required.

A TabularData object can be used to represent a table in the database (that is possibly still on disk), a result set from a query (that may not even be materialized yet), some data in a matrix format, etc. In Section 4.1, we will also see how they can be used to encapsulate user written extensions including data loaded from external systems. A TabularData object maintains information such as the columns in it, data associated with each of the columns and optionally a "matrix" representation of the entire data set. How the data and corresponding meta-information is actually physically stored depends on how the TabularData object has been derived through transforma-

tions. The users themselves are only exposed to the higher-level abstraction of the TabularData object through its APIs and are not involved with the nuances of the data set's internal representation. We will discuss the various options with regard to the TabularData object's internal representations in Section 3.3. Therefore, one can see that TabularData objects function as high-level wrappers to mask the diversity in data representations and transformation functions (relational, linear algebra, etc.), managing them transparently, performing any bridging of the statistical package (NumPy) and the RDBMS engine internally when needed, to provide a unified programming abstraction to the user.

A TabularData object is conceptually similar to a DataFrame instance in pandas and Spark but can be accessed by two execution environments: Python interpreter and SQL engine. Using a new abstraction instead of extending one of these existing DataFrame abstractions, such as pandas, avoids the overhead that these systems have in order to support relational operations on their own. This overhead is mostly attributed to the computation and maintenance of extra metadata and auxiliary data structures such as indices that the pandas DataFrames maintain to facilitate database-like capabilities. As AIDA's server relies completely on the back end RDBMS for performing relational operations, this is not a design concern for AIDA and we prefer to avoid the duplication of such metadata and auxiliary data structures which the RDBMS already maintains. Further, we want to perform some computational optimizations on relational operations in the form of *lazy evaluations* (discussed in Section 3.2.1) and the syntax of contemporary Python based DataFrame APIs such as pandas turned out to be not convenient for this as they are not designed to provide such capabilities.

3.2.1 Relational Operations on TabularData

As mentioned before, to support relational operations, AIDA borrows many API semantics from contemporary ORM systems. AIDA's TabularData abstraction supports SQL/relational operations such as selection, projection, aggregation and join. The full syntax is listed in Appendix A. Here, we provide some examples along which we explain the expressiveness of AIDA's client API and discuss how AIDA exactly performs the transformations.

Example 3.1. The code snippet in Listing 3.1 demonstrates how to use AIDA's client API to perform relational operations on TabularData objects. Here, we connect to a TPC-H database that is running an AIDA server and performs a series of relational transformations on its data sets that returns for each customer their name (`c_name`), account balance (`c_acctbal`), and country (`n_name`).

Listing 3.1: TabularData : Join and Projection

```
1 db = AIDA.connect(host='...', user='...', passwd='...', ...)
2 ct = db.customer
3 nt = db.nation
4 t1 = nt.join(ct,('n_nationkey'),('c_nationkey'))
5 t2 = t1.project(('n_name','c_name','c_acctbal'))
```

The program first establishes a connection to the AIDA server running inside the database (line 1). On success, AIDA's client API receives the reference to the *database workspace* object (`db`) that represents the connection. We will discuss database workspace objects later in detail in Section 3.4.5. Using the reference to the database workspace object, we then obtain the references to TabularData objects that represent the `CUSTOMER` and `NATION` tables in the database, and store these references in the variables `ct` and `nt` respectively. Line 4 *joins* these two TabularData objects, describing the columns on which the join is to be performed. The result is a new TabularData object, whose reference is stored in `t1`. Further, a *projection* operation on `t1` retrieves only the columns of interest, resulting in `t2`.

While the code in Listing 3.1 is using a variable for each intermediate TabularData object, users can use *method chaining* [Fow10] to reduce the number of statements and variables by chaining the calls to operators, as shown below.

```
t = tbl1.<op1>(...).<op2>(...).<op3>(...)
```

Though the original code listing is easier to read and debug, similar to conventional programming language systems, intermediate variables keep intermediate objects alive longer than needed and may hold resources such as memory. However, we will see later that in the case of relational operators, AIDA automatically groups them using a lazy evaluation strategy to allow for better query optimization and to avoid the materialization of intermediate results.

Example 3.2. The source code in Listing 3.2 demonstrates some additional relational operations that are supported by AIDA's client API. It builds upon the previous example to find countries with more than one thousand customers and their total account balances. To compute this, it performs further relational transformations on the TabularData object created in the previous example, referenced by `t2`.

First, we create `t3` by *aggregating* `t2`, grouping over `n_name` and computing the `count`, referencing this column as `numcusts` and the total account balance as `totabal`. Next, we perform the *selection* operation, by applying the condition which limits the records of interest to those with `numcusts` greater than 1000, storing the new TabularData object's reference to

t4. At this point we have the information that we have been trying to compute, so we display it. As a result, AIDA will materialize the data represented by t4.

Listing 3.2: TabularData : Aggregation and Selection

```
t3 = t2.agg(('n_name'
            ,{COUNT('*'): 'numcusts'}
            ,{SUM('c_acctbal'): 'totabal'})
            ,('n_name'))
t4 = t3.filter(Q('numcusts', 1000, CMP.GT))
print(t4.cdata)
```

Data Transfers: In Listing 3.2, the last line displays the columns of the result set referenced by t4. In TabularData, cdata is a special variable. This is the point when AIDA actually transfers data to the client. Often, users are interested only in a small sample of data records or some aggregated information on it, such as in the example we just presented. By not transferring the intermediate results, but only the results the user wants to actually visualize, we significantly reduce the data transfer overhead.

Selection Conditions: Conditions in AIDA's client API are expressed using Q objects, a syntax choice borrowed from Django's API. Q objects take the name of an attribute, a comparison operator, and a constant literal or another attribute with which the first attribute needs to be compared with. Q objects can be combined to denote complex logical conditions with conjunctions and disjunctions, such as those required to express the AND and OR logic in SQL. For example, in the previous source code listing, we could have additionally restricted ourselves to countries with a total balance above zero by including one more selection condition as follows.

```
t4 = t3.filter(Q('numcusts', 1000, CMP.GT)
             & Q('totabal', 0, CMP.GT))
```

Improvements over ORM Approaches: It is important to highlight that most conventional ORM implementations require the user to specify a comprehensive data model that covers the data types and relationships of the tables involved. AIDA does not impose such restrictions and works with the metadata available in the database. Another important distinction is that relational operations on an ORM produce a result set, akin to processing results using JDBC/ODBC APIs. They do not support any further relational transformations on these result sets. AIDA, on the other hand, transforms a TabularData object into another TabularData object, providing endless opportunities

to continue with transformations. Further, compared to contemporary statistical packages such as pandas that provide a similar, sophisticated ORM-style API for relational operations, AIDA stands to benefit in terms of performance optimizations due to its ability to perform lazy evaluations. This is important for building complex code incrementally without completely sacrificing efficiency. Users familiar with big data systems will also notice the similarity of the resulting programming paradigm of AIDA's client API to those such as Pig Latin [ORS+08], Spark [ZCF+10], etc.

3.2.2 Linear Algebra on TabularData

TabularData objects support the standard linear algebra operations that are required for vector/-matrix manipulations. We do so using the overloading mechanism of Python. These overloaded methods then ship the operation using RMI to the corresponding TabularData objects residing on AIDA's server inside the RDBMS. Similar approaches have been used by others [ZZY10]. For those operators that are not natively supported in Python, we follow the NumPy API syntax. This is the case, e.g., with the *transpose* operator for matrices. Therefore, users can apply the same operator on a TabularData object for a given linear algebra operation as they would in NumPy. AIDA's server will then invoke the corresponding operation on its underlying NumPy data structures.

Example 3.3. In Listing 3.3, we continue off from the previous example to compute the average account balance per customer for each country in `t4`, using linear algebra.

Listing 3.3: TabularData : Linear Algebra

```
t5 = t4[['totabal']] / t4[['numcusts']]
```

Further, we can also generate the total account balance across all the countries contained in `t4`, by performing a matrix multiplication operation as shown below[a].

```
t6 = t4[['totabal']] @ t4[['totabal']].T
```

Note that, `@` is the Python operator for matrix multiplication that is overloaded by TabularData and `T` is the name of the method in TabularData used for generating a matrix's transpose, a nomenclature we adopt from NumPy for maintaining familiarity.

[a] It is important to note that `t4[['totabal']]` itself results in an intermediate TabularData object with just one column, viz., `totabal`.

As we saw in the examples in Section 3.1.2, TabularData objects can also work with numeric scalar data types and NumPy array data structures, which are both commonly used by data scientists for statistical computations.

3.3 TabularData Internal Representations

There are two different internal representations for a TabularData object, and it may have zero, one, or both of these representations materialized at any time. One of the internal representations is a matrix format, i.e., a two-dimensional array representation where the entire data set is located in a single contiguous memory space (see Figure 3.3, left-top representation). AIDA's server uses NumPy to perform linear algebra, which requires this format for some operations such as matrix multiplication and transpose.

The second representation is a dictionary-columnar format used to execute relational operators (see Figure 3.3, right-top representation). This is essentially a Python dictionary with column names as keys and the values being the column's data in NumPy array format. While each array is in contiguous space, in contrast to the matrix format, the different columns might (or might not) be spread out in memory. The rationale behind this approach is that AIDA is optimized to work with columnar RDBMS implementations such as MonetDB that offer *zero-copy* data transfer of query results from the database memory space to the embedded language interpreter [LM14]. This means that the RDBMS passes query results to an embedded host language program without having to copy the data into application-specific buffers or perform data conversions. This is made possible by having database storage structures that are fundamentally identical to the host language data structures. Thus, the host language only needs to have the appropriate pointers to the query result to be able to access and read it. In particular, MonetDB creates result sets where each result column is stored in a *C* programming language array. NumPy uses the same format for arrays. MonetDB thus returns a Python dictionary with column names as the keys and references to the underlying data wrapped in NumPy objects. We leverage this optimization as it reduces the CPU/memory overhead. Therefore, the result of a relational query has a dictionary-column format as default.

Although a TabularData object's internal data representation can be in a matrix and/or a dictionary-columnar format, the matrix format is given the preferential treatment by the data transformation operators in AIDA. This is because, as we will discuss later, one can always create a "logical view" where each column of a matrix is considered independently, to build a dictionary-columnar representation from it without making a physical copy of the underlying data structures, whereas the reverse is not possible. However, it is not always sensible to create a matrix data representation by default. This is because, (i) columnar databases such as MonetDB uses the dictionary-columnar format to provide zero-copy data handover, requiring extra effort and cost to build a matrix. (ii) Different columns in a data set typically belong to a variety of data types. Since all elements in a matrix need to be of the same data type, this will require "upgrading" all columns to a data type that can store all data elements, which could be an overkill as many transformations

might not need this. Therefore, as we will see later, matrix representations are produced only in specific scenarios, keeping into account the above concerns.

In short, we can see that the internal data structures used by TabularData objects are based on conventional NumPy/Python structures. Users can access these internal representations if required. For example, in Listing 3.2, we access the dictionary-columnar data representation of `t4` through the special variable `cdata`. Similarly, the matrix representation can be accessed by using the special variable `matrix`. This allows data scientists to use AIDA in conjunction with other Python libraries that work with NumPy data structures. A TabularData object will materialize a particular internal data representation the first time that representation is requested. This happens when the user accesses it through its special variable or from one of AIDA's internal methods. Once materialized, the internal representation will exist for the lifetime of that TabularData object, reused for any further operations as required. AIDA's data structures are essentially memory resident. In the future, we plan to implement a memory manager that can move the internal representations of least used TabularData objects into regular database tables to reduce memory contention. The data in a database table itself is treated by AIDA as though they are materialized in a dictionary-columnar format for all practical purposes. Therefore, in Listing 3.1, the variables `ct` and `nt` refer to TabularData objects that represent `customer` and `nation` tables respectively.

3.3.1 Relational Operations & TabularData Materialization

Relational operations on a TabularData object are always performed lazily, i.e., the resulting TabularData object's internal representation is not immediately materialized. They are materialized only when the user explicitly requests one of the internal data representations through the special variables we just discussed or when a linear algebra operation is invoked on it. This means that by default and as shortly discussed through Algorithm 1, the new TabularData object contains only the reference to its source – which can be a database table, one TabularData object, or two TabularData objects in case of a join – and the information about the relational transformation it needs to perform on the source. This approach is similar to the lineage concept that is followed by Spark [ZCF+10] for their data sets and transforms. The source TabularData object itself may not be materialized yet or can have materialized data in any of the two formats that we discussed.

Algorithm 1 shows the high-level approach used by AIDA to keep track of relational transformations in a lazy fashion. When a relational transform is requested, AIDA first performs any static error checks on the source(s) (line 2). This is very crucial to provide immediate feedback to the user in case of input errors, rather than at some other step further down where the actual execution happens, as that can be confusing. AIDA's server can perform static checks quickly using the

Algorithm 1: Relational Transforms on TabularData

```
1 : On TabularData::relationalTransform(sources, transform):
      [ perform a static error checking                                    ]
2 :   errorObj = transform.check(sources)
3 :   if errorObj then                          [ Abort if any static errors are detected ]
4 :     throw errorObj
5 :   end
      [ create a new TabularData object (lazy materialization)             ]
6 :   return new TabularData(sources, transform)
```

actual database metadata or its own equivalent metadata (such as columns and data types) that it keeps associated with every TabularData object. This helps it to weed out typical programming errors immediately, like incorrect column names, semantically incorrect operations, etc. If the static error checks are passed, AIDA's server will create a new TabularData object that has references to its sources (immediate *ancestors*) as well as the transformation. The materialization itself (i.e., the execution of the transformation) does not happen at this point.

Referring back to our example in Listing 3.2, at line 1, `t3` contains just the references to `t2` and the *aggregation* transform that is described over it. Similarly, in line 5, `t4` contains only the references to `t3` and the *selection* transform.

Algorithm 2 shows the process followed by such TabularData objects to materialize themselves. When such a TabularData object's `materialize` method (line 1) is invoked, it will build the SQL logic required to materialize itself (line 3). To accomplish this, it will first request its immediate source(s) to provide their object's SQL equivalent (line 14). Using the SQL returned by its sources, it will apply its own relational transformation logic over them (line 15). We have implemented this by using the concept of derived tables in SQL [MS02] which allows nesting of SQL logic. The SQL thus generated is then executed using the RDBMS to fetch the data that the TabularData object represents (line 4). Once a TabularData object is materialized, it will discard the references to its source and transformation logic (line 5). As depicted in the `genSQL` method (line 9) in Algorithm 2, we can distinguish three cases for the source.

Source is a Database Table

If the source represents an actual database table, this is as simple as returning a SQL that is essentially a `SELECT` query on the table with all the columns and no filter conditions (line 11).

Algorithm 2: TabularData Materialization (Relational Transforms)

```
1 : On TabularData::materialize():
2 :     if not materialized then
3 :         SQL ← genSQL()                                  [ generate necessary SQL ]
            [ SQL is executed by the RDBMS                                           ]
            [ data generated is in dictionary-columnar form                          ]
4 :         cdata ← DB.execute(SQL)
5 :         delete transform, sources                  [ sources are no longer required ]
6 :         materialized ← True
7 :     end
8 :     return cdata

9 : On TabularData::genSQL():
10 :    if databaseTable then                          [ data is stored in the database ]
11 :        return 'SELECT * FROM <tableName>'
12 :    end
13 :    if not materialized then
            [ get the source SQL representations recursively                         ]
14 :        sourceSQLs ← sources.genSQL()
            [ build this object's SQL transform over its sources                     ]
15 :        return transform.buildSQL(sourceSQLs)
16 :    end
        [ materialized data is in Python, return a SQL over it's UDF                 ]
17 :    return 'SELECT * FROM __tbl_UDF__<XXX>()'
```

Non-materialized Source

On the other hand, if the immediate source TabularData object itself is not materialized, then it would recursively request its own source for the SQL equivalent, apply its transformation on top of it (but will not use it to materialize itself) and return the combined SQL logic (encapsulated by the `if` condition block at line 13). It is easy to envision this approach going down recursively to arbitrary depths to build a complex SQL query.

Example 3.4. Consider the following code snippet that uses AIDA's client API, where we perform a sequence of relational transformations, building one on top of another.

```
supAgg    = supplier.agg(('s_nationkey', {SUM('s_acctbal'):'totbal'})
                         ,('s_nationkey',))
supAggTop = supAgg.filter((Q('totbal', 2000000, CMP.GT),))
```

Here, we first perform an aggregation on the SUPPLIER table in the TPC-H database in order to compute the total account balance across all suppliers at the country level. Next, we restrict ourselves to only those countries with more than 2 million as the total balance by applying the necessary filter.

If we chose to materialize the result `supAggTop` at this point, it will request `supAgg` for its equivalent SQL. `supAggTop` then applies its filter operation over this SQL and ends up generating a SQL that is syntactically similar to the one given in Listing 3.4. This SQL is then executed by the RDBMS SQL engine, materializing itself using the result set in a dictionary-columnar format.

Listing 3.4: Combining Multiple Relational Transformations

```
SELECT s_nationkey, totbal FROM
(
  SELECT s_nationkey, SUM(s_acctbal) AS totbal FROM
    (SELECT s_suppkey, s_nationkey, s_acctbal, ... FROM supplier)t
  GROUP BY s_nationkey
)supAgg
WHERE totbal > 2000000
```

At first glance, the SQL in Listing 3.4 that is automatically generated by our materialization procedure, may look very verbose. An experienced programmer can instead express the same objective in a much succinct manner, such as for example like the SQL given below.

```
SELECT s_nationkey, SUM(s_acctbal) AS totbal
FROM supplier
GROUP BY s_nationkey
HAVING totbal > 2000000
```

However, although these two SQLs look different *prima facie*, their semantic meaning is the same. And as we discussed in Section 2.1.3, modern RDBMS implementations are equipped with a *query rewrite* component that internally translates user queries into an equivalent form that is computationally efficient. Therefore, we do not need to be concerned that our automatically generated verbose queries would lead to performance problems.

Most importantly, lazy evaluation techniques, while allowing users to build complex logic incrementally, ensure that optimization is not sacrificed. By nesting multiple relational transforms together, we can skip the need to materialize the intermediate TabularData objects, such as `supAgg` in the above example.

Materialized Source

Finally, the source might already be materialized in one or both of the internal representations. This data now needs to be provided to the RDBMS SQL engine for it to perform the relational operation. As mentioned in Section 3.1.3, AIDA's server exposes this data by means of a table UDF to the database (see the bottom of Figure 3.3). Table UDFs need to expose the dictionary-columnar representation for the SQL engine to execute queries. This is again because columnar-RDBMS like MonetDB share identical data structures, making data conversions and copies unnecessary. In case the source TabularData object has only a matrix representation so far, it can still provide the dictionary-columnar representation by an abstraction that represents a particular column's data as a column-wise slice of the matrix relevant to that column without making an actual copy of the data. The SQL equivalent of the source is again a simple `SELECT` query on this table UDF (line 17) that represents all the data in the source TabularData object. Such system generated table UDFs are automatically removed when the corresponding TabularData objects are removed from the system.

Example 3.5. Assuming `supAggTop` is now materialized, any future relational transformations over it will be performed over its table-UDF representation, which will have the structure in Listing 3.5.

Here `__tbl_UDF__743` is an AIDA generated table UDF for `supAggTop` that simply returns the internal data representation of `supAggTop` in dictionary-columnar format that becomes now the input to the RDBMS query engine. `743` is a unique id that AIDA's server internally generates for each TabularData object, ensuring that the names of table UDFs thus created by AIDA for the TabularData objects are unique in the database.

Listing 3.5: Internal Table UDF Representing a Materialized TabularData Object

```
CREATE FUNCTION __tbl_UDF__743()
RETURNS TABLE(s_nationkey INTEGER, totbal FLOAT)
LANGUAGE PYTHON
{
  return _getTabularDataObject('743').cdata
}
```

3.3.2 Linear Algebra & TabularData Materialization

Unlike relational operations, linear algebra operations are computed and materialized immediately when the corresponding TabularData object is created. This is primarily because NumPy does not perform any significant optimizations when multiple computational operations are combined.

Scalar Operations

Algorithm 3 outlines the strategy employed by AIDA to perform scalar operations over TabularData objects. Linear algebra transformations involving scalar operations (e.g., dividing all the elements in the data set by 1000) can be performed on either data representation because performing a scalar operation (e.g., division by 1000) on each column's data in dictionary-columnar representation is the same as performing it on the entire matrix. Scalar transformations will first check if the data is available in the matrix form and use it to perform the transformation (line 2). The resulting TabularData object will have its internal data in matrix representation. Matrices are given preference because, as we discussed in Section 3.3.1, a dictionary-columnar representation can be built from it without making another copy, whereas the reverse is not possible.

Alternatively, if an object has only the dictionary-columnar representation, it will apply the transform on each of the columns (line 7) to build the new TabularData object, whose internal representation will also be dictionary-columnar representation.

Additionally, if the source does not yet have an internal representation due to the lazy evaluation of relational operations as discussed in the previous section, it will at this point materialize itself when its `data` is requested. As the data thus returned will be naturally a dictionary-columnar representation (as the relational operations are executed by the SQL engine), the scalar transformation will work with this representation.

In summary, if there is no matrix representation available, we do not build a matrix just because of a scalar operation. This is because, as we mentioned at the beginning of this section, matrix construction takes time and is less versatile when it comes to supporting different data types. Besides, having an additional matrix representation takes up memory.

Vector Operations

Algorithm 4 shows the process employed to execute a linear algebra transformation involving vector operations, such as matrix-matrix multiplication and matrix transpose. To perform such a transformation, the TabularData object will need the matrix format representation of the source(s) involved (line 8). If any of the source(s) currently has no internal data representation due to the lazy

Algorithm 3: Linear Algebra Transforms on TabularData (Scalar)

```
1 : On TabularData::scalarTransform(source, scalar, operation):
        [ source data is available as a matrix                                    ]
2 :     if hasmatrix(source) then
            [ use source matrix for operation.  result is a matrix                ]
3 :         matrix ⟵ numpy.operation(source.matrixdata, scalar)
            [ materialize a new TabularData Object as the result                  ]
            [ with a matrix internal representation                               ]
4 :         return new TabularData(matrix)
5 :     else                                  [ source data is in dictionary-columnar form ]
6 :         cdata ⟵ {}                        [ build a dictionary-columnar representation ]
            [ the source will materialize itself if it has to                     ]
7 :         foreach col in source.data do
                [ perform the operation on each column individually               ]
8 :             cdata[col] ⟵ numpy.operation(source.cdata, scalar)
9 :         end
            [ materialize a new TabularData Object as the result                  ]
            [ with a dictionary-columnar internal representation                  ]
10 :        return new TabularData(cdata)
11 :    end
```

evaluation of relational operators, it will first materialize it, which will have a dictionary-columnar representation (line 3). Alternately, a source may already have a materialized dictionary-columnar representation (such as, for example, if cdata was already called on the object) but not the matrix representation. In either case, it will then proceed to build the matrix representation from the dictionary-columnar representation (line 4). The transformation is then applied to this matrix and the resulting new TabularData object will have a matrix internal representation.

Optimizations

Internally, AIDA employs some clever optimizations to share data between multiple TabularData objects, attributed to their immutable nature. This is possible when a transformation is requesting a subset of columns from a TabularData object's data set. If the TabularData object has a dictionary-columnar representation, then it will generate the new TabularData object in the same format, but with only the requested subset of columns in its dictionary. The data arrays for these columns will be just references to the original data set. For example, remembering that `t4` is materialized when

Algorithm 4: Linear Algebra Transforms on TabularData (Vector)

```
1 : On TabularData::matrixdata:
2 :     if not exists(matrix) then
            [ materialize dictionary-columnar data if required                    ]
3 :         materialize()
4 :         matrix ⟵ toMatrix(cdata)                  [ convert columnar data to matrix ]
5 :     end
6 :     return matrix

7 : On TabularData::vectorTransform(sources, operation):
8 :     matrix ⟵ numpy.operation(sources.matrixdata)
        [ materialize a new TabularData Object as the result                  ]
        [ with a matrix internal representation                               ]
9 :     return new TabularData(matrix)
```

we printed its dictionary-columnar data representation, next, the expression `t4[['totabal']]` results in a temporary tabular data object that has only the `totabal` column of `t4` and hence can be shared with `t4`.

3.3.3 Practicality of Dual Representations

Although the need for dual representation of data sets is primarily attributed to the fact that the RDBMS and statistical packages often have different internal representations, an often overlooked reality is that each of these systems is optimized for their own functional domain. Therefore, any compromise will come at a cost to one or both of the systems. For example, as we saw with the case of NumPy, a statistical package would often need compact two-dimensional data structures such as matrices for computational efficiency that exploits CPU cache, code locality, etc. On the other hand, such a data structure would make data growth and updates (especially the nuances required to deal with variable data types and such) extremely inefficient and impractical for an RDBMS implementation.

The scientific database SciDB, that also implements the dual system strategy of database and statistical package integration, follows the same approach of reformatting the data from the database to conform to the needs of the statistical package that it uses (ScaLAPACK [CDPW92]), when performing linear algebra operations [SBZB13, Aur15]. The authors reason that reformatting costs are negligible enough for larger data sets compared to the amount of time spent in actual

numeric computations for linear algebra and that the years of optimizations put into these statistical packages make it worthwhile to just leverage them over rewriting these operations from scratch into a database. However, we believe that optimizations in integrating these systems, such as the ones presented in this section, and the use of zero-copy whenever possible are of significance in bridging these two worlds.

3.4 Leave Data Behind

A key design philosophy of AIDA is to push computation to the RDBMS, near the data. Data transfer to the client is only needed when explicitly requested. The usual examples of such a scenario would be when the users want to take a look at a sample of records from the detailed data, when aggregated information or final results need to be viewed and analyzed, etc. Thus, the expectation is that this is only a small fraction of the overall data accesses needed for computations.

This execution model necessitates retaining any computational objects in the RDBMS memory, ship the computation primitives to these objects from the client, and transfer data from the server to the client side when required.

The high-level architecture of AIDA that supports this model is depicted in Figure 3.4 and discussed in this section. A *bootstrap stored procedure* loads AIDA's server-side processes when the database starts up.

3.4.1 Database Memory Resident Objects

As discussed in Section 2.4.4, even though modern RDBMS implementations support embedded programming language interpreters to execute UDFs and stored procedures, they only expose limited capabilities. Specifically, host language objects only live within the scope of the UDF/stored procedure source code. In order to work around this problem, we introduce the concept of a Database Memory Resident Object (DMRO). DMROs are maintained by AIDA's server, and therefore stay in the RDBMS memory (the embedded Python interpreter memory to be precise), outside of the scope of any UDF or stored procedure. Users can perform operations on these objects using AIDA's client APIs. The most common DMROs are TabularData objects, the computational component discussed in the previous sections. We will encounter a few other types of DMROs as well as discuss their scope and lifespan in this section.

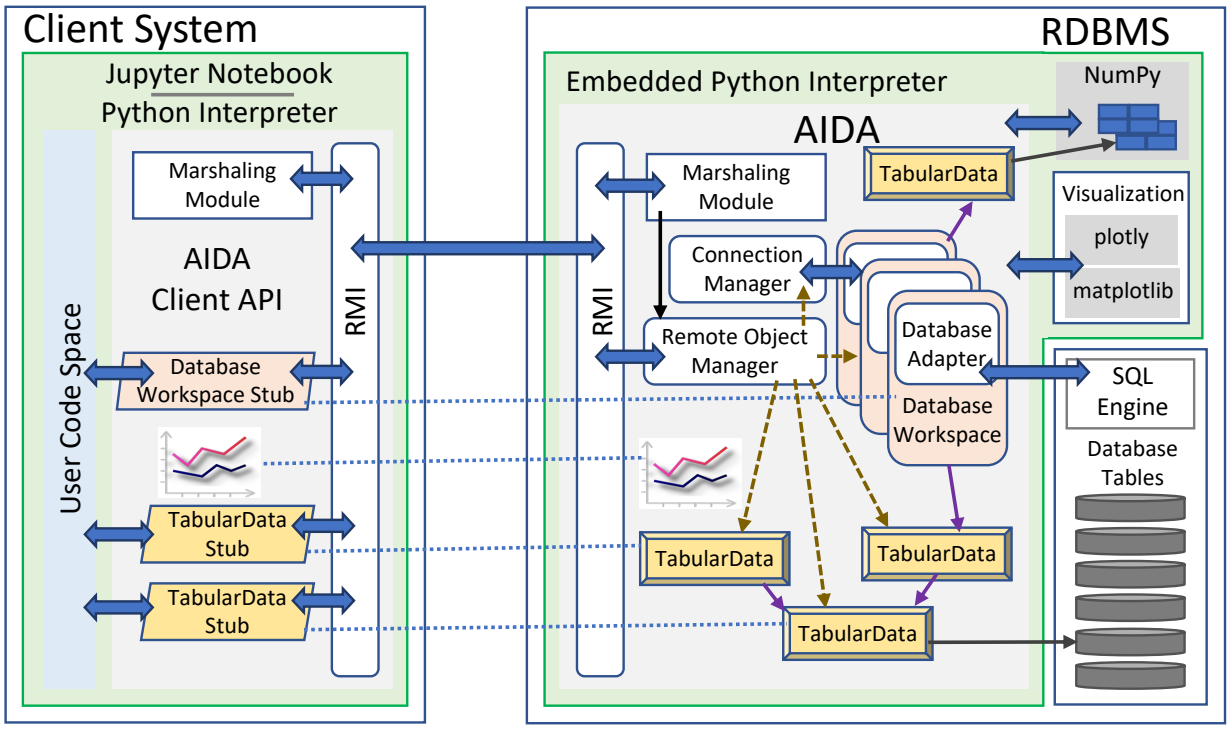


Figure 3.4: Detailed Architecture of AIDA

3.4.2 Distributed Object Communication

The DOC layer allows AIDA's client API to interact with DMROs such as TabularData objects that reside on the server side. AIDA's DOC layer follows the conventional architecture popularized by implementations such as Java RMI [PS98] with minor adaptations to suit AIDA's needs. On the server side, we have an instance of a remote object manager server, which is a DMRO created by the bootstrap stored procedure. The manager acts as a repository to which AIDA's server-side modules can register objects that are to be made available for remote access.

Similar to standard RMI implementations, AIDA's client API uses a *stub* object to interact with these remote objects residing at the server. When a stub object is instantiated at the client side, it connects to the remote object manager and passes on the *id* of the object it is interested in. The manager will create a *skeleton* object for the actual object and send the client the information required to interact with that skeleton object. For the sake of simplicity, we have omitted skeleton objects from Figure 3.4. The references to TabularData objects that we encountered so far in the source code listings (eg. `t1` and `t2` in Listing 3.1) are essentially such stubs. When the client API invokes methods on a stub object, it will *marshal* the input parameters and send them to the corresponding skeleton object on the server side. The skeleton object will unmarshal this

information, execute the intended method on the actual object, and marshal and return the results back to the client side stub.

Where AIDA's RMI implementation differs from the standard approach is in how the transfer of objects between the client and server modules are handled. Specifically, the *marshaling module* at the server-side (see Figure 3.4) has some intelligence built into it regarding the nature of AIDA's DMROs. As such, when it is asked to marshal an object, it first checks if there is a stub data type associated with that object's base type. If it finds a stub data type, it registers this object with the remote object manager and passes on the information regarding the stub data type and the object id provided by the remote object manager to the client. The marshaling module at the client side automatically looks for this information in the data that it unmarshals and uses it to construct the corresponding stub object which is then passed on to the user code. AIDA's marshaling module is built by extending dill [MSS+11], a popular module for marshaling objects in Python.

Often, a method invocation on a stub may contain another object's stub as an argument (for example, in Listing 3.1, `ct` is passed as an argument to the `join` method call on `nt`). Under such circumstances, the marshaling module at the client side will simply pass the stub information (of `ct`) as-is to the server. At the server side, when the arguments are being unmarshalled, the marshaling module will detect the stub information. It will then check with the remote object manager to see if the stub represents an object hosted by the manager. If the remote object manager finds the object in its repository, it will return it to the marshaling module, which then replaces the stub information with the actual object before it is passed on to the skeleton object.

As we will see in Chapter 4, using a tiered software engineering approach to implementing DOC, keeping it separate from AIDA's high-level data abstractions such as TabularData and its transformation modules has its benefits. As these high-level modules are agnostic to the existence of the DOC layer, it makes it easy to shift parts of the client code to the server side in order to implement custom functionality or to improve execution performance by reducing the network communication overhead. Further, this architecture also makes it possible to write applications that span multiple (possibly federated [SL90]) databases in an elegant manner, as we will also see in Chapter 4.

3.4.3 The Database Adapter Interface

Modern RDBMS implementations with embedded HLL interpreters allow UDFs written in a host language to query the database directly without an explicit database connection. However, each vendor implements their own API to facilitate this interaction between an embedded HLL code and the RDBMS engine. As AIDA's server relies on these internal APIs to interact with

the RDBMS it is embedded in, we define a *database adapter interface* to standardize this interaction, and to keep remaining AIDA packages independent of the RDBMS. Therefore, akin to applications that utilize ODBC/JDBC style standardized database connectivity, by implementing a database adapter package conforming to this interface, one can easily port AIDA's server to that specific RDBMS. A database adapter interface implementation must be able to authenticate a user's credentials with the RDBMS, read database metadata, execute a SQL query and return results in dictionary-columnar format, and facilitate the creation of table UDFs.

As part of the current implementation of AIDA, we have developed a database adapter for the MonetDB columnar database. In this implementation, as pointed out in Section 3.3, our adapter exploits the zero-copy optimization provided by MonetDB to retrieve the result sets produced by relational query execution, which are then stored in the dictionary-columnar form in TabularData objects. Many RDBMS, especially row-based RDBMSes, do not offer such optimized data transfers as their physical storage model has no resemblance to the data structures of the embedded language. Therefore, in order to work with such systems, the database adapter interface will have to convert the result set returned by the RDBMS into one of the internal representations of the TabularData object. Even accounting for any such conversion overhead, we believe that such implementations should still benefit from AIDA's approach. This is because even with contemporary client-based statistical systems, they are forced to retrieve row-formatted data from these databases and convert them into the vector formats used by statistical packages. However, unlike these client-based systems, AIDA's implementation should still incur less network transfer overhead as it resides in the database, pushing computations toward data.

3.4.4 The Connection Manager

The connection manager is a DMRO created by AIDA's bootstrap procedure that is responsible for managing client sessions. When a client makes a connection request, the connection manager uses the database adapter to authenticate with the database. On success, it creates a database workspace object (discussed in the next section) and sends its stub information back to the client. Referring back to the first line of code Listing 3.1, `db` is such a stub.

3.4.5 Database Workspace

The database workspace is another kind of DMRO, created for each authenticated client connection in AIDA. Primarily, it provides access to database tables via the TabularData abstraction. For example, `db.nation` provides a reference to a TabularData object that encapsulates the `nation`

table in the database. Such TabularData objects and any new TabularData objects created from them by applying relational or linear algebra operations have a reference to the database workspace object that created the first source TabularData object. A TabularData object utilizes its database workspace (which in turn uses the database adapter) when it needs to execute any relational operations in the RDBMS.

As we will see in Chapter 4, the database workspace also provides a number of sophisticated features such as the ability to load external data, remote execution of client code at the server, access to data visualization functionalities, etc. It also acts as a *context environment* for these components to easily share and access the same objects and variables from across both the client side and server side as we will see in the examples that we will discuss in Chapter 4.

3.4.6 Life Span of a DMRO

As many DMROs can have remote references at the client side, keeping track of their references and ensuring proper garbage collection is a non-trivial process. AIDA's garbage collection infrastructure for DMROs is built on top of Python's garbage collection process.

Since languages such as Python will garbage collect any objects that do not have a valid reference (reachable from the program), we need a mechanism at the server side to ensure that any DMROs that have remote references from a client are not removed by the Python's garbage collector. The challenge here is to make a remote reference from the client "count" towards the active references of the DMRO at the server side. Further, a DMRO that has no direct reference from client stubs, but itself is referenced by another DMRO that has a remote reference from the client, should also be not garbage collected. An example of this scenario is a source TabularData object that is referenced by an un-materialized TabularData object in a relational transform due to lazy evaluation.

Algorithm 5 provides a high-level outline of the garbage collection process employed for DMROs. As discussed in Section 3.4.2, the client-side process interacts with a DMRO by using a stub, which in turn interacts with a skeleton object at the server side that is responsible for performing appropriate operations on the DMRO object. As such, the reference held by the skeleton object on the DMRO is sufficient to prevent the DMRO from being garbage collected. The skeleton objects themselves are kept track of by the remote object manager, preventing them from being garbage collected (line 2). When a skeleton object learns that its corresponding client-side stub is no longer active, it will automatically remove itself from the remote object manager, as well as its own reference to the DMRO. If there are no more active references to the DMRO object

Algorithm 5: Garbage Collection of DMROs

```
1 : On garbageCollectDMRO(dmro):
2 :     if not exists skeletonObjectsFor(dmro) then
3 :         foreach obj in DMROrepository do
4 :             if not exists obj.referencesFor(dmro) then
5 :                 return True
6 :             end
7 :         end
8 :     end
9 :     return False
```

at the server side, the Python garbage collector will now recycle it. On the other hand, a DMRO whose reference is held by another DMRO that has an active remote reference from a client will not be garbage collected, as they are still indirectly reachable from the client space (line 3).

Although this garbage collection process may look straightforward to implement, it is complicated by the fact that the remote object manager itself also has to keep track of any active DMROs. This is necessary because a new client stub requiring access to a DMRO initiates the process by sending the object id it is interested in, to the remote object manager. The remote object manager uses the object id to lookup the corresponding DMRO and instantiates a skeleton object for it to serve the stub. Therefore, the remote object manager needs to maintain a repository of the active DMROs to facilitate such requests. This repository is also important for aiding the reverse mapping of stub information to actual DMRO references, such as when the stubs are sent back to the server by AIDA's client API as arguments to any remote method invocations made on the DMROs (as discussed in Section 3.4.2).

However, maintaining such a repository will mean that the references held by it to the DMROs will prevent them from being automatically garbage collected by Python even if there are no active client references to it. In order to avoid this interference with Python's recycling mechanism, we make use of the *WeakValueDictionary* facility provided by Python. This repository holds only *weak references* [BA19] to the objects in it. A *weak reference* is a programming language feature that allows access to an object similar to a regular programming language reference, but does not "count" towards its garbage collection status. An object that has only weak references associated with it is automatically garbage collected and the weak references themselves are removed from the repository, making them inaccessible from the program.

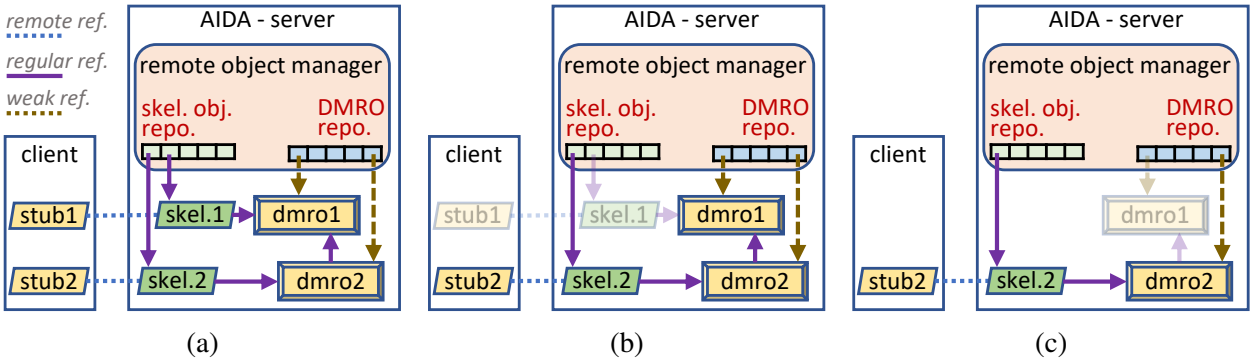


Figure 3.5: Life span of DMRO objects

Example 3.6. Figure 3.5 portrays some of the common scenarios with regards to the lifespan of DMROs based on our discussion so far. In Figure 3.5a, two DMROs, `dmro1` and `dmro2` have remote references from the client stubs through the skeleton objects. As such, they will not be removed by the garbage collection process. Further, `dmro1` is also referenced by `dmro2`. In Figure 3.5b, the client removes `stub1` that is remotely referencing `dmro1` (possibly because `stub1` itself is garbage collected at the client-side). While this results in the corresponding skeleton object (`skel.1`) being removed, `dmro1` itself is not garbage collected as it is still referenced by `dmro2`. In Figure 3.5c, `dmro2` no longer references `dmro1` (as would be the case of a TabularData object that materializes and discards its references to the source TabularData objects). Since, now the only reference to `dmro1` is the weak reference from the remote object manager's DMRO repository, `dmro1` is recycled by the Python garbage collection process, and its weak reference is removed from the DMRO repository.

Remote object manager and connection manager objects are the only DMROs that exist throughout AIDA's uptime and are therefore not garbage collected. As database workspace objects function similar to the concept of *connection* objects in traditional ODBC/JDBC based database applications [FR03], facilitating the client's interaction with the database, they are discarded only when the corresponding client disconnects.

Although TabularData objects themselves are transient due to the fact that they are only memory-resident, users can at any point persist the contents of a TabularData object into a physical database table by using the `_saveTblrData` API method provided by the database workspace object as shown in Listing 3.6.

Listing 3.6: Persisting a TabularData Object into a Database Table

```
db._saveTblrData(t4, 'CustAcctSummary')
```

The persisted TabularData object can be retrieved later by the same or different client connection as shown below.

```
casmry = db.CustAcctSummary
```

3.4.7 Optimizing Client Data Transfers

Although the fundamental philosophy of AIDA is not to take the data outside of the database, this cannot be avoided under all circumstances. For example, the user may want to take a look at some sample records, an intermediate result, the final outcome, etc. Even though these are usually much smaller data transfers, they could still incur an observable network overhead in real-time. Further, as we will see in Section 4.1.2, sometimes certain data sets required for a workflow might reside in another database of the organization and will have to be accessed and transferred locally – a concept that AIDA does support. Therefore, there is an incentive to look into the optimization opportunities for data transfers involving statistical systems.

Contemporary client-based statistical systems have to rely on standard database-client protocols such as ODBC/JDBC for data transfer. These row-based standards were developed in a time when client workstations had extremely small resources, such as main memory. As most of these early database applications were meant to produce "records" in its output (such as a printer), the data transfer protocols were designed and implemented in such a way that the database server held the bulk of the data and the client APIs of these standard protocols fetched few rows at a time, "consumed them", and then requested the sever to send some more rows. However, such approaches are not optimal when it comes to statistical systems that internally use vector-based data structures and often require the entire data set to be acquired to proceed to the next step in processing.

[RM17] has done some in-depth research into the inefficiency of traditional database-client protocols, especially in the context of data science applications. In their work, the authors reviewed the *data serialization* approaches used by various implementations of these protocols to transmit data across the network and their impact on network transfer delays. They concluded that *lightweight data compression* techniques can have a salubrious effect on the network data transfer speeds of such implementations. Their implementation slices the rows from the query result set into multiple chunks, following which each chunk is compressed column-wise (vectorized) prior to network transmission, in order to reduce the size of the data in the network. This is then uncompressed and reconstructed at the client side into a *row-format* to comply with the row-based standards of ODBC/JDBC type protocols. Therefore, although their results are promising, by continuing to rely on the row-based database-client protocols, they incur an unnecessary overhead of

serialization into row-format at the client-side, which has to be reformatted into column (vector) format by the statistical package to build its own computational data structures.

In this aspect, AIDA's implementation is built upon the success of certain proven concepts in columnar-RDBMS implementations, notably on the efficient compressibility of array-like columnar data structures [ABH09, Raa08, AMF06]. As a TabularData object's internal data representations are already in vector format, it is in an optimal form for compression and transmission. Based on the performance analysis of various popular light-weight compression libraries such as LZ4 [Col11], snappy [Goo19], base64 [IS03], and Zlib [DG96] in our network environment, we chose to implement a compressed data transfer channel using LZ4 compression as it gave us an optimal performance across a variety of scenarios. Work done by [RM17] had also concluded that LZ4 is one of the best performing lightweight compression libraries for network data transfer protocols. However, unlike the implementation in [RM17], we do not have the overhead of creating chunks out of result sets at the server side or reformat data into row-based format at the client side. In essence, we compress vectorized data at the server side, which is then reconstructed back directly to vectors at the client side. Therefore, using AIDA's client API, a TabularData object's internal representation can be retrieved much more efficiently than using the traditional database client transfer protocols.

4

An Extensible Framework

Contemporary statistical systems used by data scientists owe a significant amount of their popularity to their extensible nature. This serves various purposes such as implementing custom transformations that are not inherently supported by the framework, often by leveraging external libraries. Many of these systems also facilitate the integration of data sets that are not native to the system into the workflow. AIDA provides capabilities on par with such approaches, allowing users to express them using regular Python code snippets. By means of some practical examples, in this chapter, we will discuss and explore the many extensible features that AIDA supports. We will also see how AIDA can support complex data visualizations even though it performs server side processing.

4.1 Extending the Functionality

4.1.1 Custom Transformations

As we saw in the previous chapter, AIDA's client API supports the fundamental relational operations such as join, selection (filtering), projection, aggregation, etc., and linear algebra operations of addition, subtraction, multiplication, division, exponentiation, etc. However, users might need

to perform operations that do not fit into the realm of relational or linear transformations that are included in the repertoire of the framework. The facility to perform custom transformations implementable by the users is an integral functional requirement expected of any advanced analytical system. As we recapped previously in Section 2.4.4, RDBMSes' attempts to support this via UDFs lack usability. In contrast, AIDA's TabularData objects support this feature through the `_U` operator that is very easy to use.

Custom transformations in AIDA are expressed using regular Python functions and are invoked on the TablularData object. A custom transformation function must accept a TabularData object as its first argument. They can also accept any additional user provided arguments, but must return a data set in one of the formats that TabularData supports internally, which we discussed in Section 3.3. AIDA's client API ships the transformation function transparently to the server side using RMI to be executed at the server. Custom transformations are materialized immediately, and AIDA instantiates a new TabularData object using the data provided by the transformation function. We can perform additional relational or linear algebra operations on this new TabularData object, as is the case with any other TabularData objects.

Example 4.1. In Listing 4.1, we employ a custom transform to create a TabularData object derived from the `CUSTOMER` table of the TPC-H database, that contains only the first name of the customers.

Listing 4.1: Custom transformation example

```
1 def cFname(td):
2   cn = td.cdata['c_name']
3   fn = numpy.asarray([n.split()[0] for n in cn])
4   return {'c_fname':fn};
```

Our transformation function, `cFname`, first accesses the internal data structure of the source TabularData object (`CUSTOMER` table) and retrieves the `C_NAME` column that contains the customer's full name (line 2). Next, it uses Python's string manipulation function to parse each of the names and create an array of strings that contains only the first names of the customer (line 3). The transformation function then returns this newly created "column", giving it a name `c_fname`. Thus, in this example, the return data set of the transformation function is in the dictionary-columnar format.

Now that we have defined the custom transformation function, it can be executed as follows.

```
cfn = customer._U(cFname);
```

At this point, AIDA's client API will ship the transformation function, `cFname`, to the server

using RMI, where AIDA's server component will execute it by passing the `CUSTOMER` table's data into it. The remote reference to the newly created TabularData object is returned to the client API which is then stored in `cfn`.

As can be seen, using AIDA's user defined transform feature is straightforward. Users who are familiar with systems like pandas and Spark will also notice the similarity of constructs with the paradigms employed by those systems for custom transformations. Further, unlike the database UDF construct which imposes a host language - SQL interface that restricts programmability, AIDA exposes a pure Python interface, allowing the users to employ the full programming power of Python. This also allows users to employ other Python-based packages to perform computations inside their user defined transformations.

Example 4.2. In the previous example, if we want to additionally "guess" the gender of the person based on the first name, and include that as a second column in the output, we could do so using a third-party Python library, gender-guesser[a] as shown in Listing 4.2.

Listing 4.2: Custom transformation returning multiple columns

```
1 def cFnameGender(td):
2   import gender_guesser.detector as gender
3   d  = gender.Detector()
4   cn = td.cdata['c_name']
5   fn = numpy.asarray([n.split()[0] for n in cn])
6   g  = numpy.asarray([d.get_gender(n) for n in fn])
7   return {'c_fname':fn, 'c_gender':g};
8
9 cinfo = customer._U(cFnameGender);
```

In this example, we pass the first names of the customers to the gender detection library that returns a value `male`/`female` indicating its best guess (line 6). When we execute this modified version of the custom transformation function, the resulting TabularData object will now contain two columns – the first name (`c_fname`) of the customer as well as their guessed gender (`c_gender`).

[a] https://pypi.org/project/gender-guesser

AIDA's pure Python based implementation of custom transformations also allows users to employ more modular programming approaches and write separate program snippets for different functionality, a key aspect to incrementally building solutions. Since the result of a custom transformation is a TabularData object, which can be transformed further, this allows users to aesthetically "chain" multiple custom transformations.

Example 4.3. For the sake of modularity, we could write a totally separate custom transformation for detecting gender and incrementally build up on our previous transform in Example 4.1, that only extracted the first name of the customers. In the code shown in Listing 4.3, `Gender` accepts two additional arguments `incol` which indicates the name of the TabularData object's column from which it should read its input and `outcol` which indicates the name of the column that it should assign to the output column that it produces.

Listing 4.3: Chaining multiple custom transformations

```
1  def Gender(td, incol, outcol):
2    import gender_guesser.detector as gender
3    d     = gender.Detector()
4    tdata = copy(td.cdata)
5    name  = tdata[incol]
6    g     = numpy.asarray([d.get_gender(n) for n in name])
7    tdata[outcol] = g;
8    return tdata;
9
10 cinfo = customer._U(cFname)
11                 ._U(Gender, 'c_fname', 'c_gender');
```

This transformation essentially produces in its output all the columns that are in the input (via the `copy` function[a] in line 4), and an additional gender column that it computes (line 6). Thus `cinfo` is built by first applying `cFname` on `customer` followed by applying `Gender` on the intermediate TabaularData object that `cFname` returns. Therefore, the TabularData object that is referenced by `cinfo` will have the columns `c_fname` and `c_gender`.

[a] `copy` function only copies the column metadata from the source. Underlying data representations are shared with the source.

As indicated before, we can perform additional transformations over a TabularData object that is produced by a custom transformation. In case we perform a relational transformation over such a TabularData object, the resulting TabularData object itself will not be immediately materialized, and instead will follow the lazy evaluation process that was described in Section 3.3.1. If such a TabularData object (or any another TabularData object that is derived from it through a relational transformation) needs to materialize, AIDA will expose the data sets of the TabularData object produced by the original custom transformation to the RDBMS SQL engine through a table UDF.

Example 4.4. Here, we perform a relational aggregation operation over the TabularData object referenced by `cinfo` from Example 4.3, to compute the number of customers in each gender category.

```
cgendrdist = cinfo.agg(('gender', {COUNT('*'):numcusts}), ('gender',))
```

Unlike `cinfo`, which is materialized immediately, `cgendrdist` is not immediately materialized, as it is derived using a relational transformation. In the event that `cgendrdist` or any another TabularData object that is derived from `cinfo` using a relational transformation need to materialize, the Python data structures of `cinfo` will be exposed to the RDBMS engine using a table UDF and the transformation logic will be executed as a SQL statement over it.

4.1.2 Loading External Data

Many real-world data analysis workflows involve integrating data sets from a variety of sources. For example, it is common for data scientists across different groups in an organization to share data sets of interest between themselves (often exported from traditional analytics tools) in the form of spreadsheets [BLSG17, Par19]. Other, more formal sources of external data sets include the multitude of data processing systems that are spread throughout the organization.

As such, facilitating access to external data sets, and to be able to integrate them seamlessly into an analytical workflow is an important functionality expected of modern data science frameworks. AIDA provides this capability through *external data transforms*. Using this feature, users can load data that are external to the database (and therefore, external to AIDA), such as spreadsheets stored in their own client systems, data files stored on server filesystems, or even provided by an external data management system into AIDA.

In order to accomplish this, users write Python functions to perform the data load operation. These functions may accept any user provided arguments, but must return a data set that conforms to TabularData's internal data representation format. These transforms are then executed using the `_L` load operator. AIDA's client API ships the transform function transparently to the server, to be executed there. As such, the data movement happens directly between the system hosting the source data set and AIDA's server. Unlike the custom transforms that we discussed in Section 4.1.1, that were applied over a TabularData object, the load operator itself is executed by the database workspace object. Similar to the TabularData objects created by custom transforms, these TabularData objects are also materialized immediately and we can perform additional transformations such as relational operations or linear algebra over the resulting TabularData object.

Example 4.5. Consider a scenario in which we are given a list of customer keys in a comma-separated values (CSV) file that we have stored inside our client workstation, which we want to use to retrieve relevant customer information from the database for further analysis. Listing 4.4 shows how we can do this in AIDA using the external data load transform.

Listing 4.4: Loading external data into TabularData objects

```
1 import pandas as pd
2 ckeysdf = pd.read_csv("custkeys.csv")
3
4 def loadPandasDataFrame(df):
5     return df
6
7 custkeys = db._L(loadPandasDataFrame, ckeysdf)
```

First, we read the contents of the CSV file into a pandas DataFrame (`ckeysdf`) at the client-side, as shown in line 2. Next, we define a transformation function `loadPandasDataFrame` that accepts a pandas DataFrame as an argument and returns it (line 4). We can now invoke the `_L` operator to execute this transformation function at AIDA's server, also passing the pandas DataFrame `ckeysdf` that has the customer keys from the CSV file as its argument. AIDA's client API will ship both the transformation function and the pandas DataFrame to the server side where it will be executed. The reference to the resulting TabularData object will be returned to the client side, where it will be stored in `custkeys`.

Now that we have the customer keys of interest in a TabularData object, we can use it along with regular relational transformations to retrieve relevant customer records from the database, by querying the `CUSTOMER` table as follows.

```
custinfo = customer.filter(Q('c_custkey', custkeys, CMP.IN))
```

Since this is a relational transformation, `custinfo` is not materialized immediately. If AIDA has to materialize `custinfo`, it will expose the data set contained in `custkeys` through a table UDF to the RDBMS SQL engine.

As the internal data format of the pandas DataFrame and the TabularData object to be generated, `custkeys`, is identical, namely an array, the transformation function in this example, `loadPandasDataFrame`, is quite simple and does not perform any processing of its own. In practice, users can write complex processing logic inside them. Appendix A contains a more complex example in this aspect.

AIDA & Distributed Databases

A special use case of the `_L` operator is that it can be used to perform data analysis that spans multiple database implementations that host their own AIDA servers. In such scenarios, we can connect first to a first AIDA server from our client, perform appropriate transformations, and then pass the TabularData objects generated there to a second AIDA server. We can then continue the analysis at the second AIDA server, by using these TabularData objects in conjunction with the data sets managed by the second server. As AIDA's client API only maintains the stubs for TabularData objects, in essence, it passes this stub information to the second AIDA server. When the marshaling module of the second AIDA server encounters these stubs, it will detect that they are not for TabularData objects that it maintains. Therefore, it will instantiate stub objects of its own that point to the TabularData objects maintained by the first AIDA server. That is, the second AIDA server acts as a client to the first AIDA server.

In Section 3.3 we discussed how AIDA allows clients to explicitly fetch data to the client-side using special variables such as `cdata`. We can exploit this aspect to write an external data load function and then execute it at the second AIDA server. This will essentially copy data from the TabularData object residing in the first AIDA server to the second AIDA server and use it to construct TabularData objects that are local to the second AIDA server. In principle, this approach is similar to the *task merge* concept that we discussed in the context of ETL applications in Section 2.2.3.

Example 4.6. Consider a scenario where we have the `nation` table hosted in the database server `dbnode-1` whereas the `customer` table is hosted in `dbnode-2`. If we have to join these data sets to enrich the customer records with the associated information about their country of residence, we can do that as shown in Listing 4.5.

Listing 4.5: Writing Multi-database Applications Using AIDA

```
db1 = AIDA.connect(host='dbnode-1', ...)
db2 = AIDA.connect(host='dbnode-2', ...)

nt = db1.nation; ct = db2.customer

def loadExtTabularData(t):
    return t.cdata

nt = db2._L(loadExtTabularData, nt)
cn = nt.join(ct,('n_nationkey'),('c_nationkey'))
```

The client API first establishes connections to both the AIDA servers and obtains refer-

ences to the corresponding database workspace objects (lines 1-2). It then obtains the references (stubs) to the `nation` and `customer` tables from the respective servers (line 4), which are stored in `nt` and `ct`, respectively (also see Figure 4.1a). In line 6, we define a function `loadExtTabularData` that simply returns the internal columnar data representation of the TabularData object that is passed to it as argument. Next, we invoke the load operator on the database workspace object associated with the server `dbnode-2`, `db2`, to execute this function and pass the reference to the `nation` object of `dbnode-1`, `nt` as its argument. At this point, the client API will ship both the `loadExtTabularData` function and the stub information contained in `nt` to the AIDA server in `dbnode-2`.

The marshaling module associated with the AIDA server of `dbnode-2` will detect that the stub information from `nt` does not belong to an object that is maintained by its own remote object manager. Therefore, it will construct a new stub object in `dbnode-2` which points to the `nation` TabularData object of `dbnode-1` (see Figure 4.1b). When the database workspace of `dbnode-2` executes the `loadExtTabularData` function, it will result in the internal representation of the `nation` TabularData object being accessed and shipped to `dbnode-2`. This is in dictionary-columnar representation (i.e., a Python object) and is wrapped in a new TabularData object that is now hosted in `dbnode-2`. The reference to this TabularData object is then returned to our client, which stores it in the variable `nt` in line 9 (see Figure 4.1c). This also results in the `nt` TabularData object in `dbnode-1` being garbage collected as there are no more active references to it, as per the discussion we had in Section 3.4.6. In the next step, we perform a join between `nt` and `ct` in our client, storing the resulting TabularData object's reference in `cn`. The actual execution of the join itself will happen (during lazing evaluation) in `dbnode-2` as both the objects involved in the join are now hosted by `dbnode-2`.

In short, we can see that using standalone AIDA systems, one could easily build workflows that need to access data across multiple databases. Although the "driver" code managing the high-level logic flow is at a client system, the actual data movement itself happens directly between the involved database systems. While this overall approach may not look as transparent as systems that automatically build and optimize workflows across multiple data sources, this is still an auxiliary benefit of AIDA's sophisticated Distributed Object Communication (DOC) architecture that we discussed in Section 3.4, and is very handy when it comes to performing exploratory work. Further, complex *distribution-aware* abstractions can be potentially built as a smart layer on top of a cluster of standalone AIDA systems by leveraging the base functionality that is already provided by AIDA. In fact, [God19] has successfully utilized AIDA to implement a suite of federated learning algorithms over distributed relational database systems.

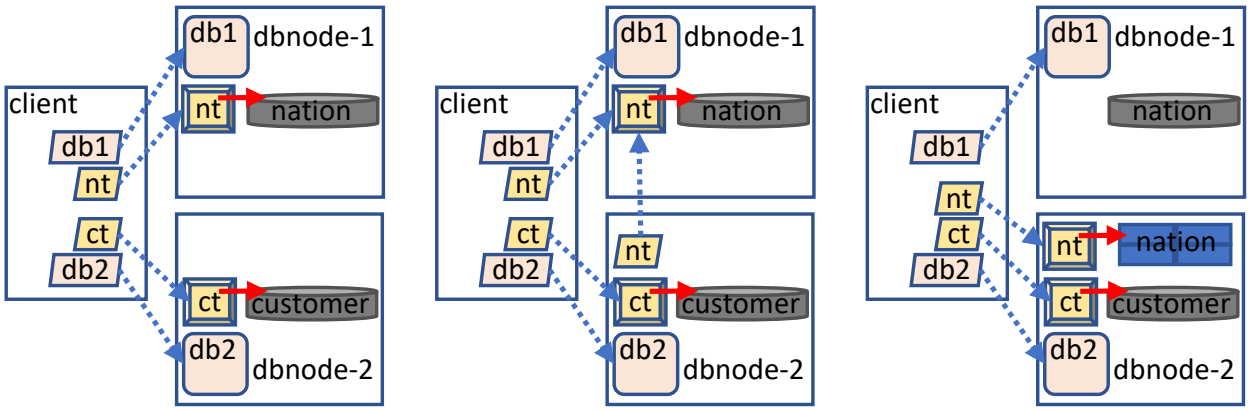


Figure 4.1: Using AIDA to work with distributed data

4.1.3 Remote Execution Operator

While the custom transformations and the external data transformations that we discussed in the previous sections are transparently shipped to the server side for "remote execution", they are a restricted API that serve a specialized purpose, i.e., transforming one TabularData object to another, and loading external data sets, respectively. In contrast, the remote execution operator serves a different purpose. Many advanced analytics applications such as learning algorithms often need to do repeated iterations of certain computations to fine tune the algorithm parameters, usually to the extent of several thousand repetitions. Using AIDA, this can easily translate to a large amount of RMI messages and may pose a non-negligible overhead for a large number of iterations. Under such circumstances, users can make use of the remote execution operator (`_X`) functionality to shift the iteration logic to the server side and bypass the RMI overhead.

The remote execution operator is attached to the database workspace object. All functions for remote executions must take a database workspace object as its first argument, followed by any optional arguments that the function itself requires. Such functions might or might not return a value (possibly a complex object) at the end of its execution. The remote execution operator executes the function on the server side, passing it the required arguments, and thus bypasses the RMI overhead for each iteration performed inside the function itself. In case the function returns a value, it is returned to the client side as the return value of the remote execution operator, `_X` itself.

As the user function in remote execution has access to the database workspace object, it can use the database workspace object as a *context environment* to store arbitrary variables, including those referencing other TabularData objects. These variables are also accessible at the client side through the database workspace object. The important advantage of this is that it provides the facility for

data scientists to incrementally improve upon a work, as we will see in an example shortly. Such abilities to "pickup and continue" are vital considerations for the human-in-the-loop to improve the productivity of data scientists and are found lacking in the database UDF based approaches. The remote execution operator is a suitable alternative to the stored procedure concept of RDBMS that we discussed in Section 2.4.4, but fits neatly into the host language, without the hassles and restrictions of UDFs and stored procedures.

Example 4.7. Consider a TabularData object `ti` with all numeric columns. Listing 4.6 shows how we can make use of the remote execution operator to perform the iterations required to compute the n^{th} power of each element in `ti`, where $n > 0$.

Listing 4.6: Iteration via remote execution

```
1 def exp(db, td, n):
2   for i in range(0, n):
3     db.res = db.res * td
4
5 db.res = ti
6 db._X(exp, ti, 9)     //res is now power to 10
7 print(db.res.cdata)
```

Here, the user function, `exp` takes as arguments a TabularData object `td`, and n, the power for exponentiation. It performs the required computation by performing multiplications iteratively (lines 2-3). At the client side, we first set the default value of the result for $n = 1$ (line 5). We then invoke the remote execution operator to find the 10^{th} power of `ti` (line 6). The client API will ship the exponentiation function and the arguments to the server, where it gets executed. The results are then displayed at the client side (line 7).

In the above code listing, we are using the database workspace object to store a variable (`res`) that is used both by the client to set up the initial value (line 5), as well as by the exponent function to store its intermediate and final results (line 3). Thus, as can be seen, database workspace objects provide a convenient mechanism to share some form of "state" between the source code artifacts that execute on the client-side and the server-side. In the case of our example, it provides a "context" for the exponentiation function to continue from where it left off at the end of its last invocation, if there is such a need.

For instance, if we are not "satisfied" by the number of iterations in the previous step, we can simply continue from there to perform some more iterations as shown in the following code snippet, where we continue from where we left off previously.

```
db._X(exp, t6, 5)     //res is now power to 15
print(db.res.cdata)
```

4.2 Data Visualizations

Data scientists routinely rely on visualizations to analyze and explore the characteristics of the data sets that they are working with [XMSP18]. Visualization plays an important role in incorporating human perception into the exploratory process [Kei02, FL03]. Database UDF based approaches are not capable of providing such functionality. Further, systems that focus on optimizing the overall workflow execution ignore or do not have the architectural paradigms to integrate an exploratory visualization component into the process flow [PTS+17, RPE+17, CDC+18].

Specialized data visualization tools are primarily designed to extract the data from the source for visualization purposes and therefore suffer from the same drawbacks as statistical systems that have to extract the data from the database to perform any analysis. [KPHH12] proposes that visual analysis tools should be built over existing infrastructures for data processing to limit data migration. To this extent, they suggest building systems that use an HLL based approach for analysis and visualizations to ensure interoperability.

Owing to the fact that AIDA is implemented in Python, we have leveraged proven visualization packages from the data science community and integrated them into AIDA's architecture. Presently AIDA provides two options for users to visualize data. They both work in tandem with the Jupyter Notebook Python environment.

4.2.1 Static Data Visualization

Visualization of large data sets can be tricky, especially when the data resides at a different place (RDBMS) from the location where the visual element (i.e., the image), is displayed (such as a Jupyter Notebook of a client computer). Our approach creates the visual element at the server side, writing it into a memory buffer. The buffer is then serialized and sent to the client side and displayed. This approach eliminates the need to transfer the actual data that is the input for visualization, to the client side. The only network overhead is incurred from the image transfer. In practice, the image size is much smaller than the actual base data, and in most cases, is too small in the context of current network speeds to be consequential for real-time analysis.

For static data visualization, data scientists can write a Python function that uses matplotib [Hun07], a plotting library that is very popular in the data science community, to construct their charts. Such visualization functions must accept a database workspace object as the first argument, followed by any additional arguments required by the function. These additional arguments are usually used to pass the references of the TabularData objects that contain the data that needs to be visualized. AIDA requires the visualization functions to return a matplotlib *figure* object.

This visualization function can be then executed using the `_Plot` operator associated with the database workspace. Similar to the custom transformation and remote execution operators that we discussed earlier, AIDA's client API will ship the logic from client space to the server side transparently, where it is executed, and the result (the figure generated by the visualization function) is sent back to the client API. While the visualization function will have to access the internal representation of necessary TabularData objects to produce the chart, since the code itself is executed in the server, there is no data transfer involved to the client side. Figures thus generated can be displayed by graphics-capable IDEs such as Jupyter Notebook. They can also be exported by users to many of the popular image formats such as PNG and JPEG, and can be used in any regular document processing applications.

Example 4.8. Let us consider a scenario where a data scientist is exploring the open orders in the TPC-H data. They can identify the orders that are still open by employing the relational transformations that we discussed in Section 3.2.1. In the following code snippet, they apply a filter on the `ORDERS` table to select only those order records that are open (i.e., has an order status of `O`).

```
openOrds = db.orders.filter(Q('o_orderstatus', C('O')))
```

At this point, the data scientist might be interested in understanding the distribution of the number of open orders with respect to their total prices. This is conventionally visualized in the form of a histogram. They can accomplish this by writing a visualization function using matplotlib, encoding the logic to generate the desired *figure* object as shown in Listing 4.7.

Listing 4.7: Static data visualization in AIDA

```
1 def openOrdHist(db, oord):
2     oorddata  = oord.cdata
3     fig       = plt.figure()
4     ax        = fig.add_subplot(111)
5     ax.hist(oorddata['o_totalprice'], bins=100, color='c')
6     ax.set(title='TPC-H Open Orders Total Price Distribution')
7     plt.xlabel('Total price'); plt.ylabel('Number of orders')
8     return fig
```

The above visualization function accepts the TabularData object that represents the open orders as its second argument. In order to visualize this data, it first accesses the TabularData object's internal dictionary-columnar data representation (line 2). Next, it instantiates a matplotlib figure object (line 3). In line 5, the visualization function uses the total price attribute of the orders to plot the histogram using the matplotlib API. It then sets the axis labels and the title

of the chart (lines 6-7). The matplotlib figure thus created is then returned by the visualization function. The visualization function is then executed using the `_Plot` operator as shown below.

```
img=db._Plot(openOrdHist, openOrds);
show(img);
```

AIDA's client API will ship both the visualization function `openOrdHist`, and the reference to the TabularData object for open orders, `openOrds`, to the server, where it is executed. The resulting matplotlib figure object is returned to the client, which is stored in `img`. This is then displayed using AIDA's client side API function, `show`, that displays it within the Jupyter Notebook environment (see screenshot in Figure 4.2).

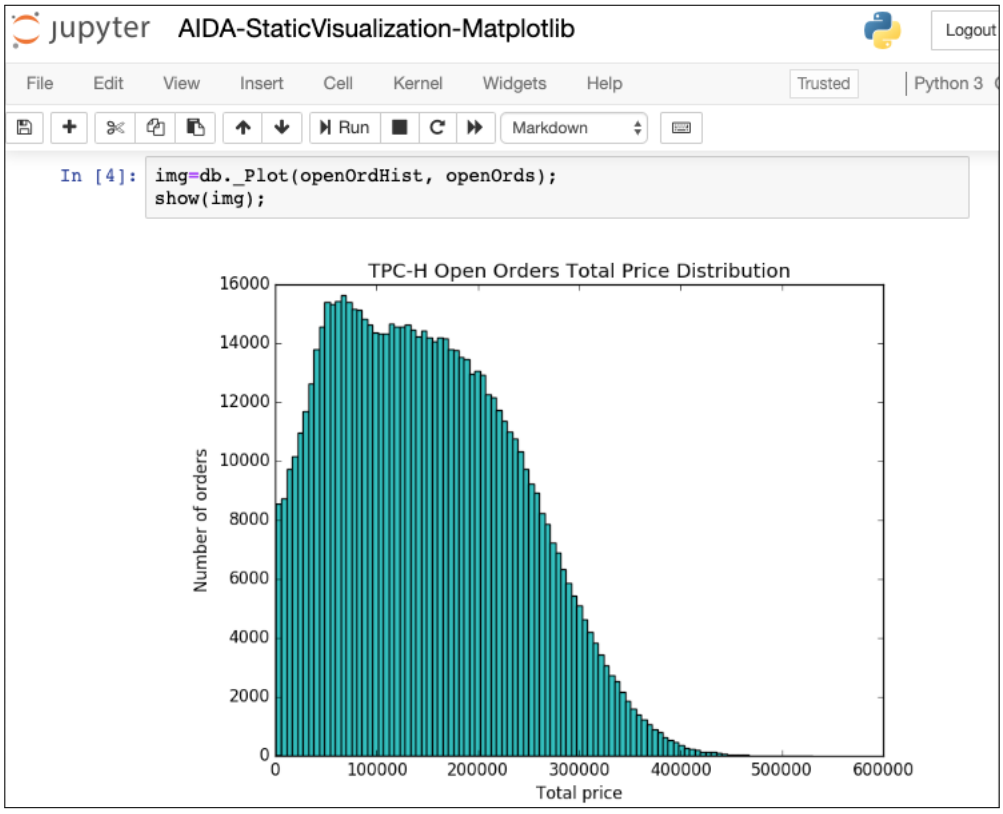


Figure 4.2: AIDA static data visualization in Jupyter Notebook

4.2.2 Interactive Visualization

While static visualizations are very useful, many advanced users wish to utilize interactive data exploration capabilities with visualizations. Although this is the domain of specialized visualization software systems such as Tableau [Lot19], QlikView [Sin16], etc., they are not very flexible for analysis work outside of the visualization paradigms. Studies have shown that the lack of flexibility in performing data manipulation operations in such specialized visualization tools results in users approaching these softwares as more of an explanatory (reporting) component rather than an exploratory tool [KPHH12]. This is akin to how RDBMS implementations have struggled to provide support for programming paradigms outside of relational operations in a user-friendly manner. Based on a user study, [KPHH12] postulates that using the same tool for visualization and analysis will allow users to iterate between these tasks with ease.

AIDA provides programming support for user-interactive visual elements by employing plotly dash[1], a popular Python library for interactive graphing. Advanced users can leverage plotly's complex API to build interactive charts. The system generates a JavaScript based visual interface that can be made accessible through a web browser-like tool. Browser-based approaches to interactive visualizations have been implemented by others previously to facilitate progressive analytics [FP16]. As most data scientists who use Python for their analysis use Jupyter Notebook, which is a browser-based IDE, we implemented support in AIDA's client API to display these visual interfaces inside the user's notebook.

Plotly functions for use in AIDA are required to accept a database workspace object as the first argument and a plotly *dash* object as the second argument, followed by any arguments that the function needs. These functions are expected to return a plotly *layout* object that describes the properties of the charts. The dash object can be used by a plotly function to install any *callback* functions to update the charts and visual components based on user interaction. As is the case with static data visualization functions, plotly functions can be passed references to TabularData objects representing data sets of interest as arguments.

Users can execute plotly functions using the `_Page` operator associated with the database workspace. AIDA's client API transports the logic to the server side, along with any additional arguments, where it gets executed. The resulting visual interface is then displayed at the client side, inside Jupyter Notebook. Further, similar to how the remote execution operator can utilize the database workspace object as a *context environment* to store states (see Section 4.1.3), plotly functions can do the same, allowing the callback functions installed by them to execute transforms over TabularData objects based on user input and store the results in the corresponding database

[1] https://dash.plot.ly/

workspace object. A working example of using plotly function with AIDA is available in Appendix A.

Additionally, the JavaScript based visual interfaces produced by plotly functions are accessible in a web browser through its own URL, outside of the Python workflow used to construct them. As such, data scientists can use this facility to develop a workflow to analyze some new concept and only publish the end-user facing visualizations or interactive widgets to their user community, that they can access through a web browser, without having to share the complex programming nuances associated with typical data science workflows. Such approaches can help streamline the *explore - prototype - user validate* process flow of data science projects.

However, given that contemporary interactive visualization libraries such as plotly embed various aspects of data into their display widgets (which is then constructed at the client side), this is, in essence, a situation where the data does leave the server in some form. Therefore, they are not ideal for visualizations that directly embed a large amount of detail data, such as histograms and scatter diagrams. This is, however, not a pressing issue as such diagrams can as well be visualized using the static visualization techniques that we discussed previously. Interactive visualizations can be on the other hand used to portray high-level summaries of such detail data. With AIDA, such summaries can be easily computed at the server side efficiently instead of shipping all the detail data to the client side, which is the case when using any specialized data visualization tools. In fact, the work done by [KPHH12] concluded that in order to achieve reasonable interactive speed at the client side, visual analytics tools should leverage server-side pre-processing and aggregation techniques.

While AIDA's support for data visualization is not intended to compete with the capabilities of specialized data visualization softwares such as Tableau and QlikView, the programmability and the flexibility of the overall framework is intended to be appealing to the "hacker" type data scientists who are proficient programmers and accustomed to writing complex workflows [KPHH12]. Further, we believe that the emergence of easily extendable unified systems like AIDA will provide incentives for the development of interactive visualization libraries that take into account data transfer efficiency. Such libraries could perform data transfer lazily on a need basis, leveraging the data manipulation capabilities of in-database frameworks like AIDA.

4.3 The Usability Aspect of Extensions

As we saw from the examples in the previous sections, custom transformations and the remote execution operator are mechanisms to ship entire functions to the server side to be executed in

server space. They also facilitate the integration of existing packages and libraries with ease.

Although the extensions in AIDA have a procedural format similar to database UDF based approaches, unlike the latter, they are not impeded by the polyglot programming style where any object transfer between a database and an HLL must be performed over a custom database internal API. For example in the case of database UDFs, a Python object cannot be passed freely from one UDF to another even within a single SQL statement, requiring users to program around such implementation drawbacks. As AIDA is written in pure Python, this is not an issue, and as we saw in the examples, the objects generated inside extensions executed on the server side are easily accessible at the client side and vice versa. In this aspect, AIDA's support for extensions draw parallels to those provided by contemporary statistical systems like pandas, Spark, etc., with the difference that AIDA manages to keep the data and computation at the server-side.

5

Experimental Analysis and Comparisons

In this chapter, we will present and discuss the performance and usability analysis that we performed on AIDA. At a high level, our testing objectives are:

(i) To understand the cost benefits of keeping data at the server compared to transferring it to the client and processing it there.

(ii) To measure the cost benefits of using a data transfer mechanism that is optimized for statistical data structures that AIDA provides.

(iii) To compare AIDA's performance for linear algebra operations with executions on the client side using a standard statistical package.

(iv) To observe how statistical packages fare when it comes to executing relational operations compared to AIDA, which pushes them into the optimized RDBMS.

(v) To measure AIDA's framework overhead compared to database UDF-based solutions.

(vi) And finally, to compare the performance of AIDA against these systems for an end-to-end learning problem.

We evaluate the following systems. **AIDA** is our base AIDA implementation. We also analyze variations of AIDA (e.g., with and without using remote execution functionality). **DB-UDF** is a UDF-based solution that, just as AIDA, performs execution within the RDBMS. Furthermore, we

implemented three solutions that transfer data out of the RDBMS for execution at the client using the following frameworks: (i) **NumPy**, (ii) **pandas** and (iii) **Spark**.

Database UDFs are meant to be representative of the most popular genre of database-centric works that focuses on computational efficiency, which we covered in Section 2.4.4. Of the database-external approaches employed by data scientists that we discussed in Section 2.3, NumPy and pandas are the most ubiquitous end-user based systems. Further, even though there exists a variety of other Pythonic implementations for statistical and data science libraries, NumPy and pandas have similar use cases and underlying structures as our own implementation (they leverage NumPy array-based statistical computations). As one of the main focuses of our work is on transparently shifting the computational aspect of statistical systems to the server side, this also helps us in doing a fair comparison of the performance implications of using an RMI based approach in our implementation without any variables introduced by the difference in underlying statistical computational libraries. The nuances of exploring the concept of AIDA in the context of other computational libraries is discussed in the future work. In addition to the above end-user based systems that we are comparing with AIDA, by including a single node setup of Spark, we would like to validate through our performance analysis that such specialized implementations for cluster computing cannot measure up to the alternatives in resource-restricted environments like an end-user workstation.

5.1 General Test Setup

Hardware: We run the client (including the database-external systems) and the RDBMS on identical nodes (Intel® Xeon® CPU E3-1220 v5 @ 3.00GHz and 32 GB DDR4 RDIMM main memory, running on Ubuntu 16.04). The nodes are connected via a Dell™ PowerConnect™ 2848 switch that is part of a Gigabit private Local Area Network (LAN). In practical implementations, the RDBMS would be likely on a high-end server with much larger resources compared to a client's computer, and the interconnect would be less powerful. However, keeping the two systems similar ensures fairness in the performance metrics that we are comparing.

Software: For software, we use MonetDB v11.29.3, pymonetdb 1.1.0, Python 3.5.2, NumPy 1.13.3, pandas 0.21.0, and Spark 2.3.0 using MonetDB JDBC driver 2.27. Unless explicitly specified, default settings are used for all software. In all our experiments we only start measuring once a warm-up phase has completed and average over several executions. Where relevant, we also discuss the *coefficients of variation* (c_v) of the observed data points to understand their dis-

persion. The coefficient of variation is a commonly used statistical measurement that allows one to compare across data distributions without being concerned about the relative difference in their magnitudes [Abd10].

Data sets: We choose a diverse number of data sets for our testing, each intended to focus on the specific aspect of the characteristic of the systems that are being tested. As such, we have synthetic data sets (explained in the relevant test cases) used for various micro-benchmarks, and we also use the TPC-H Benchmark, to leverage the variety of data types and distributions that it provides in order to test the behaviour of compression-based client data transfers. Finally, to compare the systems' characteristics when performing end-to-end real-world analysis, we make use of a data set that is based on the city of Montréal's public bicycle sharing system, Bixi.

5.2 Loading Data to Computational Objects

As one of the arguments for database-centric systems such as AIDA is that database-external systems incur a significant amount of data transfer overhead, we start out our test cases by studying this aspect in depth. In this test case, we try to understand the cost benefits of not having to transfer data for computation from the database into the client space. We experiment using synthetic data sets ranging from 1 to 1 million rows stored in database tables. Each table has 100 columns consisting of randomly generated floating point numbers. We chose numeric data because one of the objectives of this test case is also to build two-dimensional computational structures (matrices) in the systems being compared, and numeric data is a natural fit for such use cases.

For NumPy and pandas, we use pymonetdb, MonetDB's Python DB-API [Kuc98] implementation to fetch data from the database and build a *NumPy array* resp. a *pandas DataFrame*. Python DB-API is a row-based data transfer standard, in line with the ODBC/JDBC implementations that we discussed in Section 2.2.2. As the performance of retrieving data is dependent on the connection buffer size in pymonetdb, we test with the default buffer size of 100 but also an optimized buffer size of 1 million, reflecting the size of our largest data set. The latter setting is recorded as `NumPyOpt` and `pandasOpt` in the performance figures. For Spark, we load the data using the JDBC connection to build a Spark *matrix*. The default fetch size of JDBC is set to 1 million.

For DB-UDF, we use a Python UDF to load data from the database into a *NumPy matrix*. For AIDA, we build a *TabularData object*. AIDA, by default, materializes the result of a relational data load in dictionary-columnar representation. We also measure the cost for *additionally* building the matrix representation of the data, indicated as `AIDA-Matrix` in the chart. Recall that pandas,

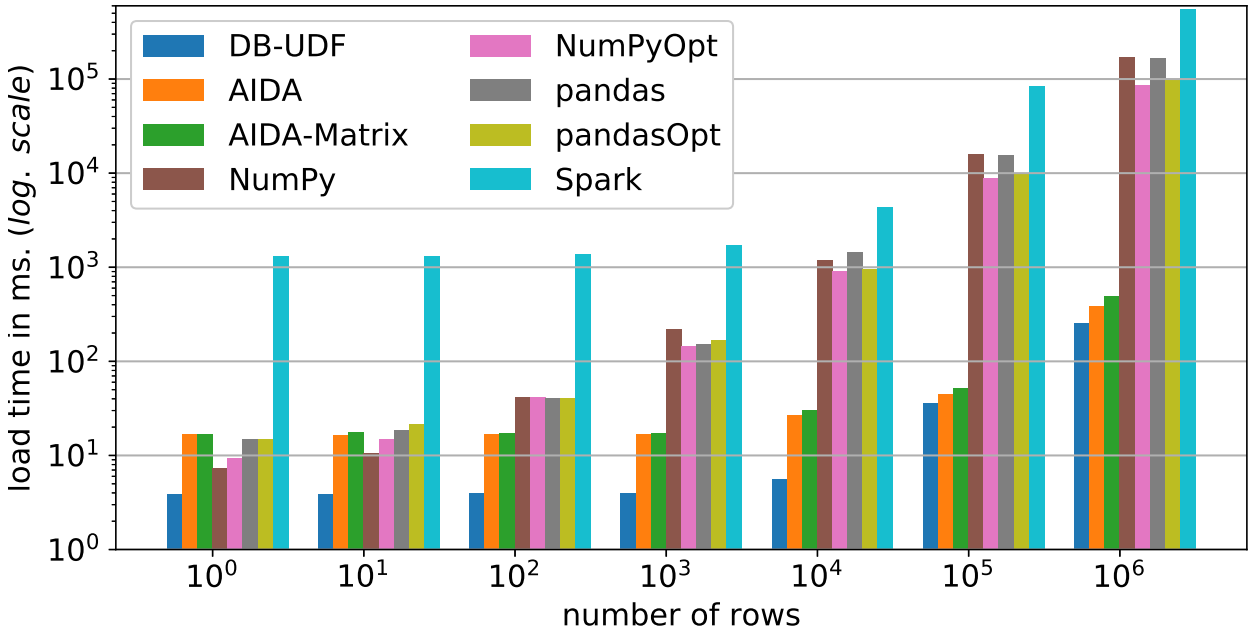


Figure 5.1: Time to load data.

NumPy, and Spark load the data into the client space. In contrast, both AIDA and DB-UDF keep their computational data structures on the server (RDBMS) side.

Figure 5.1 shows the loading time on a logarithmic scale. As a first observation, our Spark implementation performs worse than any other solution by orders of magnitude, including the other client-based solutions. We would like to note that Spark is optimized for large-scale distributed batch computations; that is, it has quite different target applications and cannot deliver good performance in non-clustered environments like an end-user workstation.

Clearly, for small data sets less than 100 rows, AIDA has no real advantage over the other client-based solutions. In fact, for a 1-row table, AIDA needs twice as long as NumPy, and 13% longer than pandas. However, this is still in the range of a few milliseconds, and thus, not noticeable for an interactive user. But as the data size increases to 100 rows, the trend changes with AIDA now having an upper hand over client based approaches. At 100 rows, AIDA takes only 40% of the time compared to loading with NumPy and pandas. This trend continues as data set size increases. At 1 million rows, AIDA is roughly 440 times faster than the default connection for NumPy and pandas, and 240 times faster if we use optimized connections for NumPy and pandas. Even if we force AIDA to build matrices after loading data, it is still able to load data about 188 times faster than the optimized connections with NumPy and pandas. In absolute numbers, while NumPy and pandas took about a minute and a half to load 1 million rows with an optimized connection, AIDA manages to load data and also build an additional matrix representation of its data under half a

second.

DB-UDF, also being server-based and having no framework overhead, is the fastest across all table sizes. For small tables, it is around 4 times faster than AIDA, and with larger sizes, it takes around 65% of AIDA's time. Thus, for a simple data load, AIDA has no benefit over the alternative server-based solution of using UDFs. On the other hand, while there is no observable benefits when working with small data sets, being server-resident, when using large data sets, AIDA is able to load data hundreds of times faster than even the hand optimized client-based systems, as it does not involve any data transfer, making a noticeable impact to a real-time user.

While the response times of the near-data approaches – DB-UDF and AIDA, demonstrate very low c_v throughout (around 10%), the client-based systems start out with a c_v close to 50% for small number of rows before they eventually fall under 10% at 1 million records. As client-based systems have to move the data that they need across the network, their response times are more influenced by any variations in the network's behaviour. Given the transfer times are small for test cases with smaller numbers of rows, they are impacted with transient network activity only intermittently, resulting in large c_v values. On the other hand, with larger numbers of rows and the resulting longer data transfer sessions, almost every data transfer session is impacted by transient network activity, resulting in smaller c_v values.

5.3 Micro-benchmarks for Numeric and Relational Operations

In this section, we analyze the most fundamental and computationally-intensive building blocks of data science workflows, viz., the operators to perform linear algebra and relational transforms. At a high level, we would like to see what is the overhead for AIDA due to RMI when it pushes computations to the server side and how statistical systems fare in their relational capabilities compared to AIDA that uses a well-optimized database back end for the same.

5.3.1 Linear Algebra Operations

To measure the overhead of the AIDA framework on linear algebra operations, we multiply a matrix with a vector. This is a familiar computational step in many machine learning algorithms such as *linear regression*, especially during the *model training* phase of the analysis. This involves transforming a training data set consisting of m attributes and n samples (i.e., a $m \times n$ matrix) by multiplying it with a vector of m *parameters* to compute the *dependent variable* vector, which in turn will have n elements.

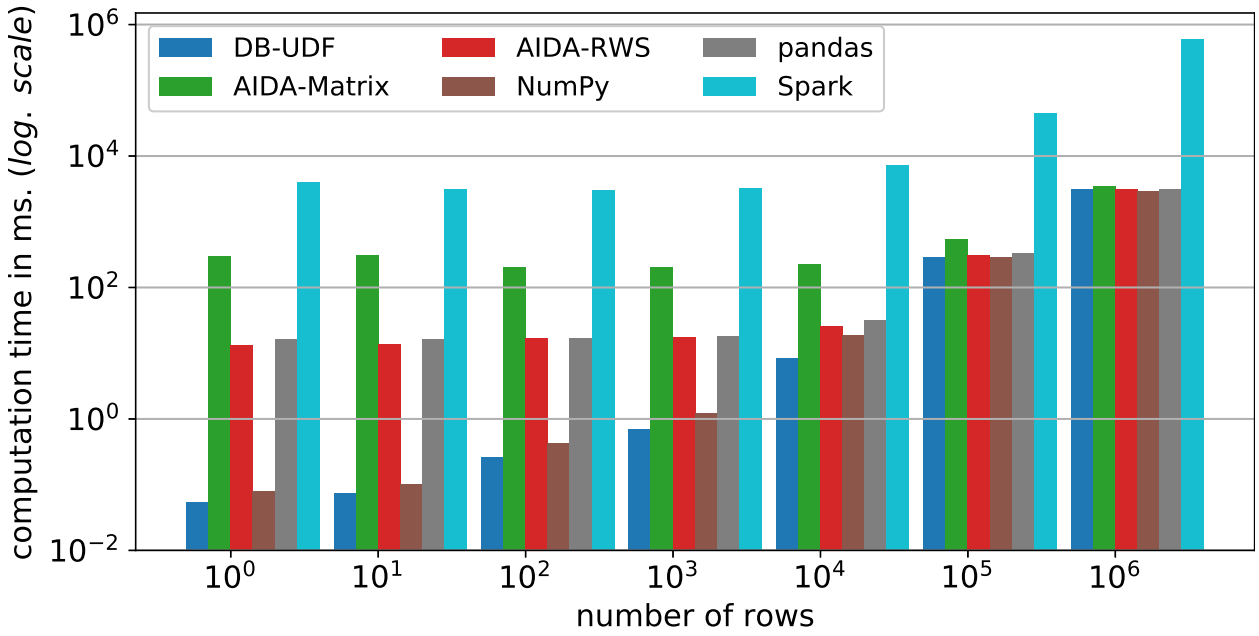


Figure 5.2: Matrix × Vector perf.

For our test case, the primary matrix is built again from the same table as in the first test case discussed in Section 5.2, made of numeric data with 100 columns and up to 1 million rows. The vector is represented as a matrix with 100 columns stored as one row. Therefore, the vector is transposed before performing the multiplication. The operation can be visualized in code as shown below.

```
res = M1 @ M2.T
```

Here `M1` can be considered analogous to a training data set consisting of 100 attributes, and `M2` the parameters vector, and `res` the *dependent variable* vector predicted using the said parameters.

The test measures the cost of performing a total of 100 matrix multiplications after the initial objects are loaded. Such iterative scenarios are very common in learning algorithms as parameters are continuously adjusted to bring the error rate down. In order to observe the overhead due to RMI calls, we also implement this iteration using the *remote execution operator* functionality of AIDA's database workspace where the iteration logic is executed within the server (recorded as `AIDA-RWS`). That is, AIDA-RWS has only one RMI call compared to 100 RMIs with AIDA. Figure 5.2 shows the execution time in a logarithmic form. The results shown here do not include connection or load times between client and server but only the join execution times. In pandas and NumPy, we do not show results for the optimized (`Opt`) version. Since we do not consider the time for the data transfer, the optimized versions have the same performances as the non-optimized ones.

Again, Spark performs significantly worse. The difference in programming paradigms might have an impact. Having little or no framework overhead, NumPy, and DB-UDF (which is using NumPy) are the fastest, showing similar performance for all data sizes. AIDA-RWS and pandas also have similar performance (with AIDA-RWS being up to 20% better than pandas) but they are worse than NumPy or DB-UDF. Both have the overhead of meta-information for their TabularObject resp. DataFrame. At a data size of 1 row, this has a huge impact, and they are around 250 times slower than DB-UDF or NumPy. But as this is still in the range of a few milliseconds, it will not have a significant impact on an interactive user. As the number of rows in the matrix increases, the cost is shifting to the computation itself making metadata and framework overhead a smaller fraction, barely noticeable once we reach 100K rows and execution times in the hundreds of milliseconds. AIDA performs worse than AIDA-RWS and pandas due to the RMI overhead (100 calls vs. 1 call), as each RMI call adds around 2-3 ms. Again, with an increasing number of rows, this fixed overhead has less and less impact as computation cost becomes the predominant factor. At 1 million rows AIDA performs only a bit more than 10% worse than the other approaches.

While we can see that AIDA does incur a framework overhead compared to other "Pythonic", client-based or database UDF-based solutions, such overheads shrink to a smaller fraction of the overall computation cost with increase in data set sizes. Further, AIDA's remote execution operator demonstrates itself to be an important concept that data scientists can leverage to effectively reduce the framework overhead of AIDA that stems from its RMI-based software architecture.

Further, if we look at the overall dispersion of the measured data points, we notice that for test cases with small number of rows, the c_v values are large across the systems, reaching up to 130%. As the actual computation time associated with these data points are very small, any transient activity in the nodes adds significant variance to these measurements. However, this does not influence the relative performance rankings of these systems and after 100 thousand rows, the measurements stabilize at c_v values that are below 2%.

5.3.2 Relational Joins On Matrices

In this section, we compare the join implementations in client-based pandas and Spark with server-based AIDA and DB-UDF, that can leverage MonetDB's optimized join implementation. As NumPy is a package that is meant for numeric computing and does not provide any convenient APIs for relational operations, we do not include it in this test case. Instead, pandas is the most popular alternative used by data scientists to perform relational operations over Python data sets. As was in the previous test case, the results shown here do not include connection or load times between client and server but only the join execution times. Therefore, we have not included the

optimized versions for NumPy and pandas in this test case either.

We use two data sets with 11 integer columns and one million rows each. In the test, we gradually add more columns to the join condition. One of the columns is unique and identical in both data sets, functioning as the key. Given two rows with the same keys in the two data sets, the remaining columns may differ between the two data sets with a small probability. Using integer columns make it easy to generate data using probability functions. Further, while different systems have different data type storage structures, fundamental data types such as integers seldom differ when it comes to the underlying implementation, unlike more complex types such as strings. This helps us focus better on the join cost itself, with minimal interference from the differences in underlying storage paradigms. With this data, the key column join between the two data sets produces all the million rows in the output, while adding more columns into the join condition reduces the result's cardinality. The output of the join contains all columns. As join conditions are a major influential factor in RDBMS implementations when selecting optimal execution approaches, we surmise that by keeping the data set constant and varying the number of columns involved in the join, we should be able to observe and compare the performance behaviours of these systems in that aspect.

For AIDA, we test two scenarios that it might have to face. In the first case, the data is in RDBMS tables and AIDA, therefore, executes a SQL internally in the RDBMS to produce the result which is then stored in a TabularData object. In the second case, the input data is materialized in TabularData objects in the dictionary-columnar format. AIDA exposes them via table UDFs to the RDBMS and has it execute a SQL performing a join on the table UDFs. This approach is denoted as `AIDA-TableUDF`. DB-UDF executes an SQL join over the tables inside the database and loads the contents into a NumPy array. Spark and pandas similarly perform joins over the data sets that are already in their DataFrame objects.

Figure 5.3 shows (non-logarithmic) response times (left axis) and cardinality of the result (right axis). As the data set for this test case is quite large and the response times are above 100ms, any impact due to transient system activity in the nodes is small, keeping the c_v values below 10%. From the figure we can see that the two AIDA implementations and DB-UDF perform much better than pandas and Spark as they can leverage the underlying RDBMS' optimizations. Response times for AIDA and DB-UDF are always less than 150 ms showing that MonetDB is well equipped to handle complex joins and/or large result sets.

For the key-join, pandas takes 97% more time compared to AIDA working with database tables and 87% more time compared to when AIDA needs to resort to table UDFs. With both AIDA scenarios, the cost goes down slightly as more non-key columns are added into the join condition

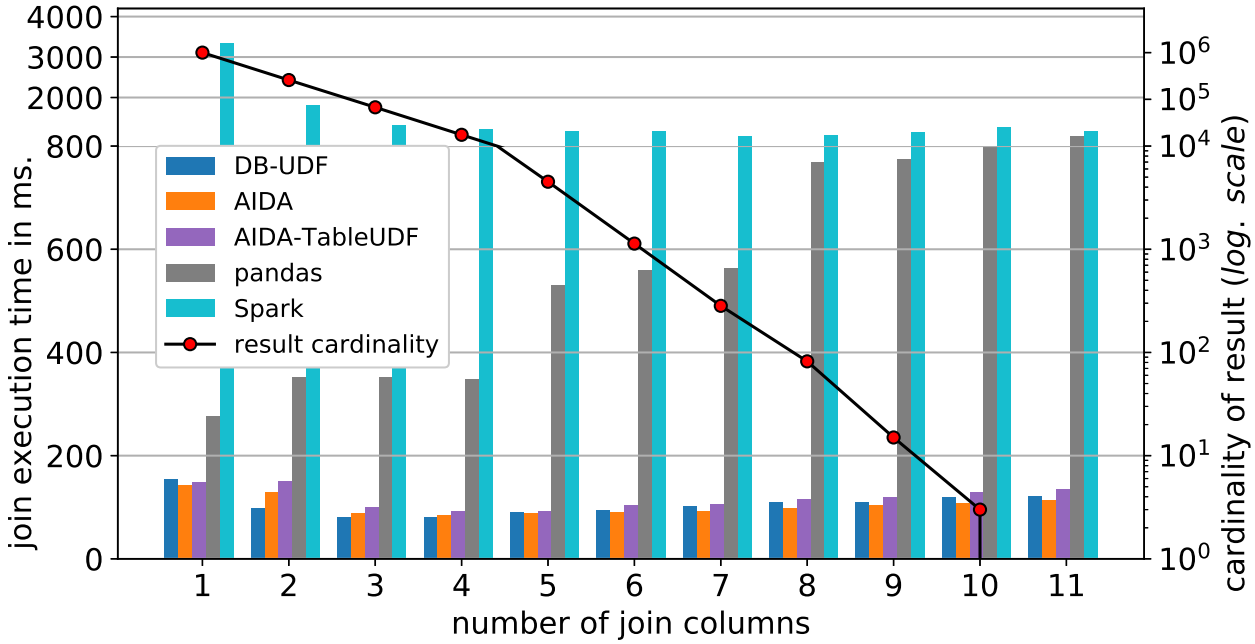


Figure 5.3: Joining two data sets.

before it eventually climbs back up. Initially, a significant amount of AIDA's cost comes from materializing the results as the number of join columns is less (therefore higher cardinality of result), whereas, as more columns are included in the join, that cost starts to reduce, and at some point join computation becomes the dominant cost.

Finally, with all the 11 columns involved in the join, when no row makes it to the final result, pandas takes about 9 times the amount of time that AIDA takes when the latter's data is in a database table, and 8 times compared to AIDA using a table UDF on NumPy arrays. At the bigger picture, we also observe that AIDA's join cost remains reasonably stable over the entire test spectrum with a standard deviation of around 20, whereas for pandas it is about 240. As RDBMS implementations such as MonetDB are equipped with a gamut of join algorithm optimizations, exploiting them gives AIDA a leading edge over pandas.

In short, both client-based solutions are significantly less efficient. pandas takes nearly twice the time for the 1-column join, and around 8 times longer with more than 7 join columns compared to the server-side solutions. Spark is even worse, in particular with few join columns. It looks like it struggles with large results sets more than with complex joins as its performance improves with smaller result sets. Therefore, we can see that, while these client-based solutions may provide support for key relational operations, they do not have the sophisticated optimization capabilities of a traditional RDBMS engine to develop cost-effective execution plans.

5.4 Data Transfer Throughput Tests

It is clear from our evaluation in Section 5.2, that AIDA has a distinct advantage over any client-based implementations when it comes to creating the computational objects from the data stored in a database. This is because, aside from the fact that AIDA is creating the computational objects *near-data*, as we mentioned in Section 2.3.1, data science tools are still held back by the conventional standardized data transfer mechanisms (to which DB-API also belongs) that are inefficient for statistical packages. Even though AIDA's design philosophy is to keep the data at the server, we also provide a faster mechanism for data transfer, when there is a need for one, as discussed in Section 3.4.7.

To understand the performance advantages of AIDA's data transfer implementation, we consider the scenario in which a data scientist wants to construct a pandas DataFrame object at their client side Python interpreter using the data stored in a database table. They can do this by passing the appropriate SQL query to a pandas API (`psql`) along with the database connection (Python DB-API/`pymonetdb`), as shown in the example in Listing 5.1. This is the approach that is commonly used to load database-resident data in Python-based data science projects.

Listing 5.1: Creating a pandas DataFrame using SQL & DB-API.

```
df = psql.read_sql_query('SELECT * FROM lineitem', con)
```

Using AIDA, users can similarly build a client-side pandas DataFrame as shown in Listing 5.2, by accessing the `cdata` internal data representation of the corresponding TabularData object. This will result in AIDA transporting the relevant data from the database side to the client, which is then used to build the DataFrame object.

Listing 5.2: Creating a pandas DataFrame using AIDA.

```
df = pandas.DataFrame(db.lineitem.cdata)
```

For our test, we compare pymonetdb **DB-API**, **AIDA**'s default implementation that has compression enabled and an implementation with compression turned off, **AIDA(noComp)**. Similar to the test case in Section 5.2 to load computational objects, we also test pymonetdb by manually setting a larger buffer size of 1 million, **DB-API(Opt)**. Further, as network dynamics are an important factor when analyzing data transfer throughput, we test three different scenarios for the client: (i) one in which both our client and database/AIDA server nodes are connected to the common **Switch** (Gigabit), (ii) another where the client is a system connected through a **LAN** (we used a desktop that is connected over a 100Mbps LAN), (iii) and finally, from a laptop that is accessing the data across the **Internet** over a Virtual Private Network (VPN).

The response times that we measure also include the query execution times at the database. This cost is the same for all approaches, as all require the database to execute the select query, and therefore incur identical overhead. Further, since this is a simple select, the time spent by the database in "query processing" is minuscule in comparison to the typical network overheads that we are out to measure. Additionally, since we perform warm-ups before the actual measurements, there is no overhead in the database in terms of I/O, for the retrieval of the table's data from the disk, as the data set will already be in memory. Therefore, the lion's share of time is spent in the serialization of the data at the database side, transmitting it across the network, followed by de-serialization into the format required by the client system.

5.4.1 Transfer of a Large Table

Figure 5.4 shows the performance throughput for creating a pandas DataFrame at the client side for the three network types, containing data from the `LINEITEM` table of the TPC-H Benchmark for a database of Scale Factor (SF) 1[1]. We chose this table, as it represents a fairly large sized data set consisting of a variety of data types and distributions. Though we did analyze a few other TPC-H tables, the overall pattern is very similar. Therefore, we do not include them for the sake of brevity. Looking at the performance metrics for the Switch (Figure 5.4a), we can see that AIDA can perform data transfer 17.5 times faster with compression on, and is also 24 times faster than DB-API. Even when we optimize the DB-API transfer buffers, AIDA is still 18.5 times faster than DB-API(Opt). On the other hand, DB-API(Opt) has almost the same transfer speed as AIDA(noComp). In the absence of compression, both have around the same amount of bytes to transfer, resulting in a similar speed.

For a LAN network (Figure 5.4b), some interesting patterns emerge. We see that AIDA is 6 times faster with compression on, and is 17.5 times faster than DB-API and 12.5 faster than DB-API(Opt). Although AIDA is still much faster, as it compresses the data, it does not push as many bytes into the network compared to others, which is essential to obtain an overall high throughput in high latency environments. This is attributed to the transient, *slow-start* phase of the underlying Transmission Control Protocol (TCP) [FAG+11, CP02], whose effect on observed throughput diminishes with larger data transfers. We also observe that even without compression, AIDA is 2 times faster than DB-API(Opt) over the LAN. This is in contrast to the case with the Switch, where they were on similar terms. Once again, this boils down to the fundamental design principles of conventional data transfer protocols like DB-API where the data is transferred as several batches of records (often involving application-level *handshake* to retrieve each new

[1] Considered equivalent to a 1GB database.

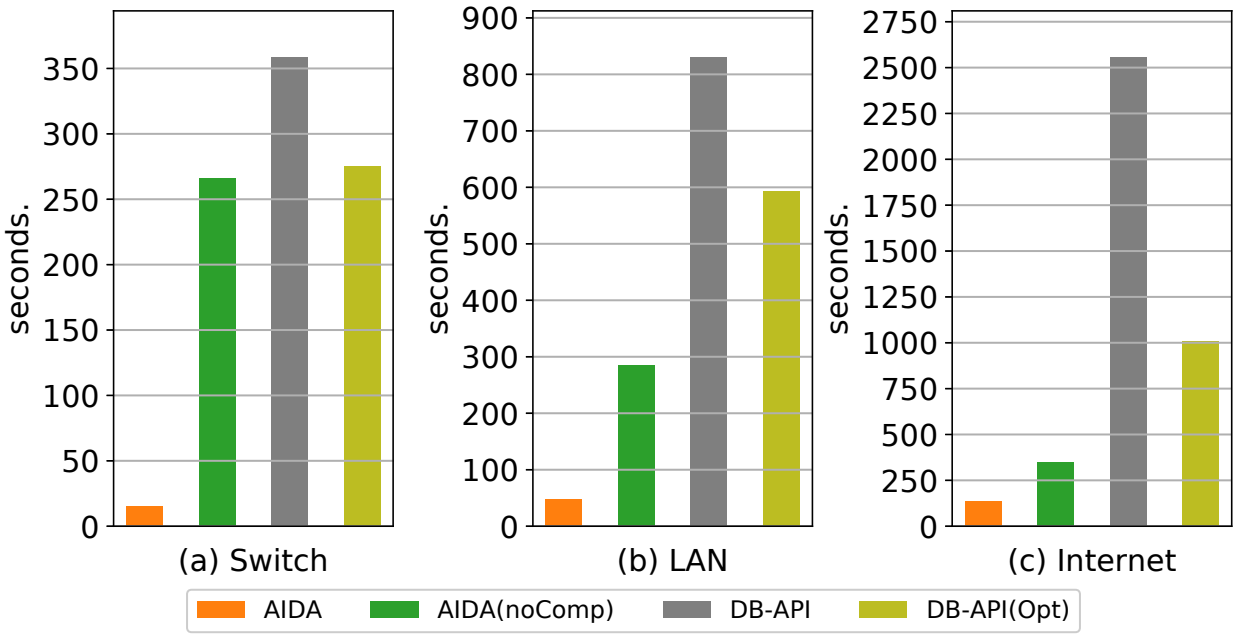


Figure 5.4: Transfer time for `lineitem` table, TPC-H scale factor 1.

batch). While such "control messages" can be passed quickly in a low latency environment, as the network latency increases, such as in a LAN, the time spent waiting for these messages to be exchanged starts increasing, resulting in an overall increase to data transmission times. As AIDA has no such handshake implementation at the application layer, it pushes the data continuously to the network, allowing the network layer to be utilized to the maximum.

We observe a similar behaviour for testing across the Internet (Figure 5.4c), where AIDA is 2.5 times faster with compression, and is 17 times faster than DB-API and 7.5 times faster than DB-API(Opt). As discussed in Section 3.4.7, although some interesting research has been performed in optimizing the data transfer throughput of conventional database protocols in the context of data science applications [RM17], they are still not optimal for statistical systems due to their reliance on conventional row-based data transfer protocols standards. They have also not made it to any mainstream RDBMS releases yet.

We also observe that the LAN and Switch environments are quite stable in terms of network response times, as observed by coefficients of variation that are under 4% for all of the systems involved. On the other hand, for the Internet test case, while DB-API does not show any significant changes to c_v , for AIDA we observe a c_v of up to 38% when using compression. This is because, as AIDA's data transfer sessions are generally shorter, in some instances it is able to perform them quickly without any intermittent network congestion, while at other times it would be impacted due

to the congestion. This results in more variance in AIDA's observed data transfer times. Compared to AIDA, DB-API has relatively longer data transfer sessions, making them consistently vulnerable to intermittent network congestion, though that shows up (perceived) as more stable transfer times in the form of smaller coefficients of variation values.

5.4.2 Transfer Costs and Data Characteristics

Compression performance metrics are quite complex and can also depend on the data type and distribution of the data values. Therefore, in this section, we look at the transfer time for individual columns with different data types and distributions. We use a representative set of columns across a variety of data types from the `LINEITEM` table (listed in Table 5.1) and measure the performance of transferring them individually.

Table 5.1: Columns and data types used in data transfer testing.

Column	comment	returnflag	orderkey	shipdate	discount
DB Type	vchar(44)	char(1)	int	date	dec(15,2)

The resulting query that is executed in the database is of the form.

```
SELECT <col> FROM lineitem
```

The response time includes the query execution times, as was in the preceding test scenario when we queried the entire table. Once again, the associated overhead from query execution itself is identical for all the systems involved and is not significant.

Figure 5.5 shows the performance of these systems for the switch. We can see that for `VHAR(44)` column, AIDA is 10 times faster with compression enabled, is 13 times faster than DB-API, and 5.8 times faster than DB-API(Opt). Interestingly, AIDA without compression is slightly slower than DB-API(Opt). This is because as AIDA's internal data structures are Python based, data types such as `VCHAR` are stored using Python string objects, which are quite complex, and by default, serializing them also includes a lot of the object metadata as part of each string. As standard data transfer protocols such as DB-API are more purpose-driven, they just transmit the underlying string data (such as a character array), without the overheads imposed by object-oriented languages such as Python. However, as AIDA uses compression, libraries such as LZ4, which performs *dictionary encoding*, can automatically and efficiently remove the heavy footprint of object metadata as these are repetitive in nature.

When we look at the columns `CHAR(1)`, and `INT`, both of which have relatively small data values, we find something interesting. Although AIDA performs the data transfers under a second

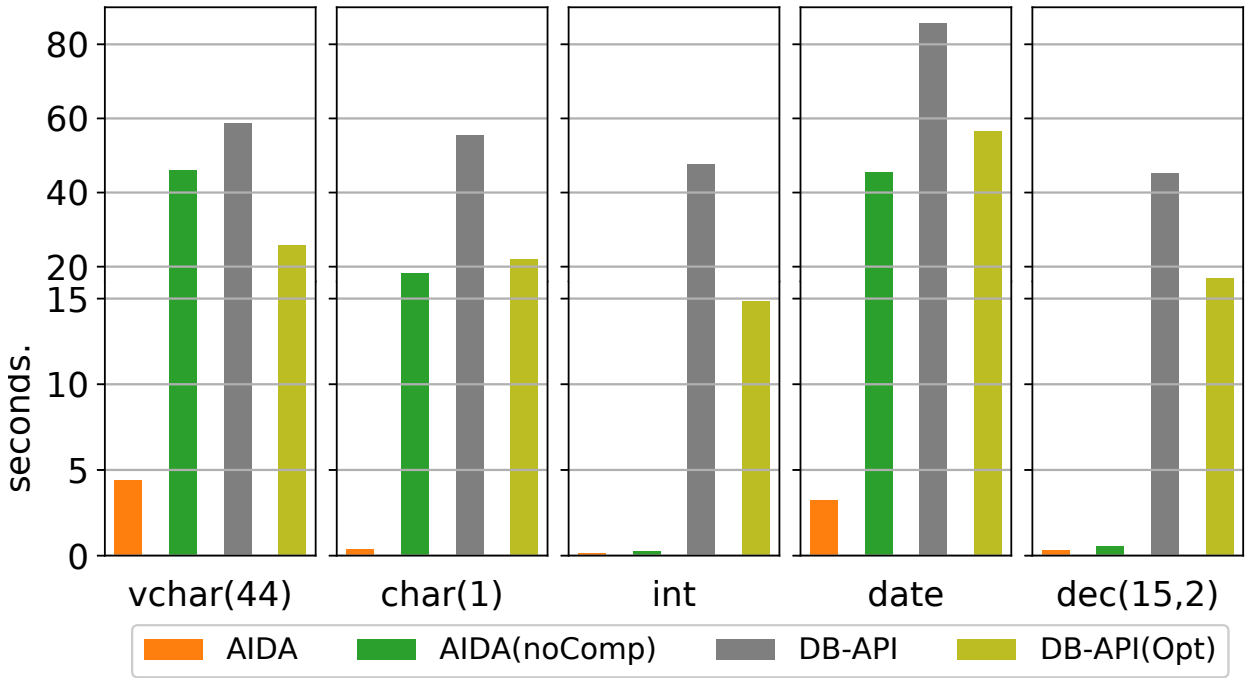


Figure 5.5: Transfer time for different columns across a Switch.

for both; without compression, it is efficient only for `INT` whereas, for `CHAR(1)`, it takes 45 times more without using compression. This is because, as in the case of `VARCHAR(44)` column that we just discussed, as AIDA's internal data structures are in Python, `CHAR(1)` is processed as Python string object, which has object metadata overhead. However, since it has only 3 distinct values (being the `RETURNFLAG` column), this makes it highly compressible, thus reducing the actual data transfer overhead when using compression.

Basic data types such as `INT` are transferred very efficiently by AIDA, even without compression. This is because there is no object overhead for such data types in Python, and as such, even when uncompressed, they have comparable sizes in the network with database protocol implementations. DB-API implementations on the other hand, because of their row-based approach of data transfer (even though there is only one column), still transmit data in chunks and rebuild them at the client side into a vector format. Further, the general trends for `DATE` column, which is a complex object with very few distinct values follows the behavioural characteristic of `CHAR(1)` column. Similarly, `DEC(15,2)`, being a numeric column, also follows the observations for `INT` column.

The performance observations for LAN (Figure 5.6) follow the same relative trend as the ones for Switch. The only exception is that for `VCHAR(44)` column, AIDA is faster than DB-API(Opt) even without compression. This is because, being higher latency environments than Switch, the approach of transferring data as multiple chunks becomes less efficient in LAN and Internet, as

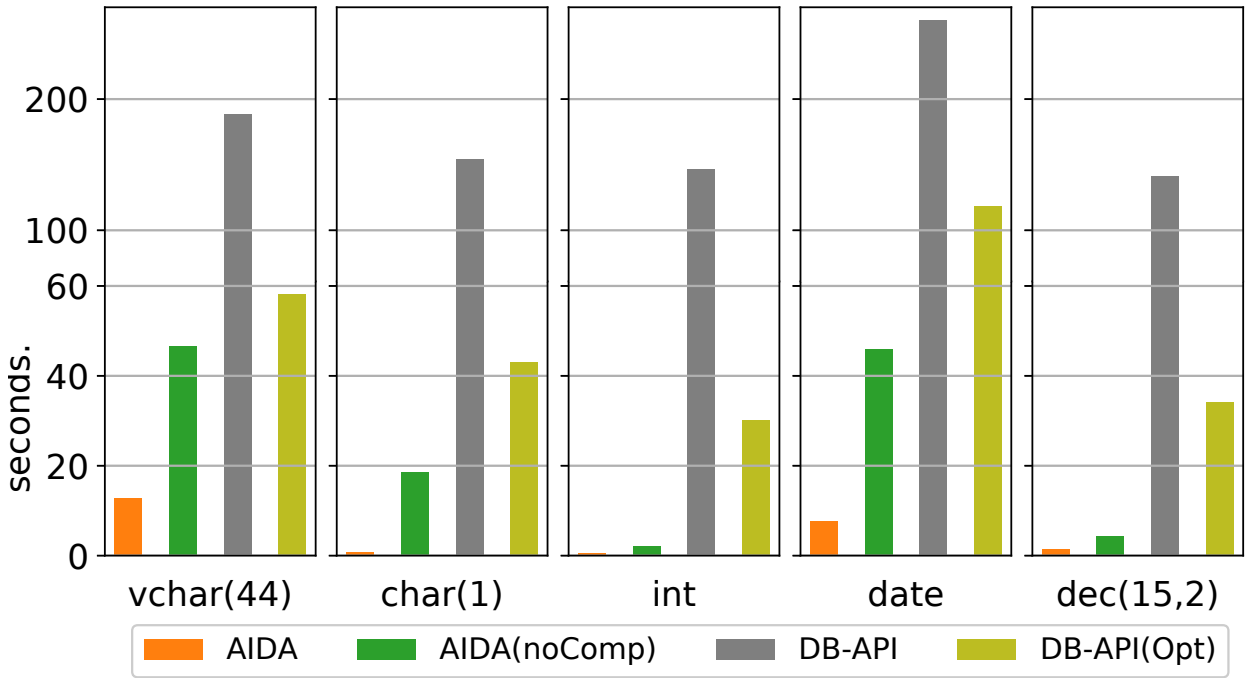


Figure 5.6: Transfer time for different columns across a LAN.

the "handshake" control messages now encounter more turnaround time, resulting in suboptimal utilization of the network. As the relative performance of data transfers across the Internet are very similar to that of the LAN test case, we have not included it for the sake of brevity.

In summary, we can see that the most common approach of extracting data from the database using conventional approaches such as DB-API fares poorly for data science applications. Even when some optimizations are applied to it by manually tuning the data transmission buffers, AIDA is still faster than such conventional data transfer approaches as it maintains data in a vector format that is suitable for statistical client systems. Also, by utilizing compression techniques internally, AIDA is able to further optimize the overall transfer time by reducing the network footprint of the data that is transmitted. We hope that the performance benefits demonstrated by AIDA opens the case for new, dedicated, vector-based data transfer protocols for statistical systems.

5.5 End-to-End Learning Problem – Linear Regression

Finally, to understand the performance and usability of AIDA on a real learning problem, we have developed solutions for a more complex test case. The data set is based on the city of Mon-

tréal's public bicycle sharing system, Bixi[2]. The learning objective is to predict the duration of a trip, given the distance between the start and endpoints of the trip. This lends itself well to the application of a linear regression algorithm. We do not use any built-in libraries such as Scikit-learn [PVG+11] or Spark MLlib [MBY+16] to avoid any bias from the implementation differences of these libraries. Instead, the linear regression algorithm is written using relational and linear algebra operations on the computational structures (DataFrames for Spark and pandas, matrices for DB-UDF, TabularData for AIDA), as would be the case when a user develops their own algorithm from scratch.

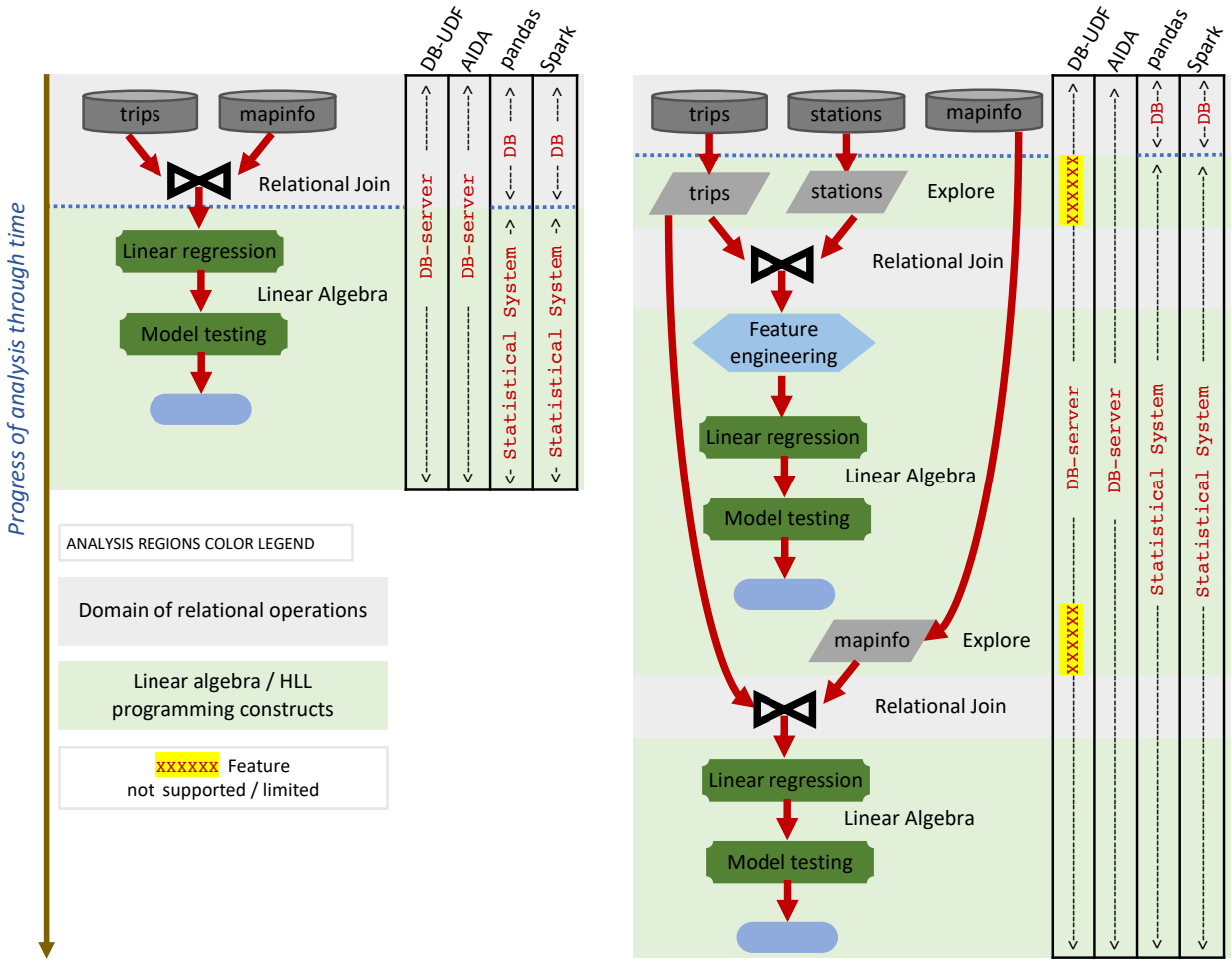
Workflows We have built two workflows depicting how a data scientist could explore the problem. The implementations of these workflows for each of the systems (DB-UDF, AIDA, pandas, Spark) are available on github[3]. For AIDA and pandas they are depicted in Jupyter Notebooks. For Spark and DB-UDF, we formatted the source code of their implementations using github's markdown file format which allows for reading the source code along with the output results. For the sake of brevity and completeness, we have included a high-level outline of the workflows in Figure 5.7 and an indication of the progression of analysis by the data scientist through time. In the workflow diagram, relational operations are depicted in a grey background, whereas statistical and other HLL usage is shown in a green background. Further, when a particular approach cannot support a specific functionality, it is marked out as `XXX` in a yellow background.

A first short workflow shown in Figure 5.7a does not perform any data exploration but assumes the data scientist knows exactly the data set and features required and the best model for training. An important aspect is that the client-based solutions (pandas, Spark) can retrieve from the database at the beginning of the workflow exactly the data that is needed using a join between the trip and map data which contains road distances between Global Positioning System (GPS) coordinates. As can be seen in Figure 5.7a, there is a clear "separation of responsibilities" where all the relational operations are performed first in the database. This offers a best-case scenario for these systems. Any further operations in the workflow are primarily linear algebra, and this can be efficiently performed by the client-side statistical framework.

The second example workflow shown in Figure 5.7b depicts a more complex path where a data scientist explores the data sets. When using client-based statistical systems, this usually involves first exporting the data set out of the database for exploration. As a result, any further relational operations on the data sets (such as joins) have to be performed inside the statistical system. They then build and analyze different models before choosing a promising approach. The data scientist

[2]https://www.kaggle.com/aubertsigouin/biximtl

[3]See https://github.com/joedsilva/vldb2018



(a) Short workflow

(b) Long workflow

Figure 5.7: Bixi workflows.

first decides to build a distance feature from the GPS coordinates available in the bicycle stations data using a geodesics-based formula [Vin75] as part of *feature engineering* and uses it to build a model for the trip data. They then explore the idea of using a map data set that has actual road distances to enrich the trip data. For pandas and Spark, this involves joining the map data in the database to the trip data which is already at the client side. For that, the additional, smaller (map) data set is retrieved from the database and the join performed inside the statistical framework. With DB-UDF and AIDA, everything is executed at the server side (RDBMS). The data scientist then compares the error rate of both approaches and decides to settle for the later. It must be highlighted that with DB-UDF, there is very little exploratory functionality available to the data scientists compared to the other approaches. As can be seen in Figure 5.7b, because we go back and forth multiple times between relational and linear algebra/HLL operations while performing

this analysis, there is no point in the analysis timeline, where we can effectively separate out the tasks that can be optimally executed by the RDBMS vs that belonging to statistical systems/HLL. As such, for pragmatic reasons, we are forced to leverage the suboptimal relational functionality provided by these statistical systems.

Additionally, we look at two performance metrics. First, we want to understand how easy they are to implement in the different systems. Second, we analyze the execution times.

Complexity Usability is very subjective, and it is difficult to measure the relative complexity of different programming paradigms in quantitative terms. As such, we employ two different metrics to assess this aspect. (i) First, as a simple approximation, we measure source code complexity as the amount of source code for each implementation, using non-white space/no-comments character count as the metric. We do not use the typical lines of code (LOC) metric as it may not be able to accurately portray the nature of different programming languages/paradigms[4]. To be fair, we chose similar variable and function names for all systems. With the exception of DB-UDF, all workflows are written using a single programming language (Python or Scala). In contrast, the DB-UDF implementations contain a mix of Python UDFs and fairly complex SQL statements. This metric is depicted in the left blue bar in Figure 5.8. (ii) As an additional metric, for pure-Python implementations (AIDA and pandas), we use the Python tokenize module[5] which is a lexical scanner for Python source code, to count the number of non-comment tokens in those workflows. As DB-UDF workflows also contain non-Python code (SQL) and Spark workflows are written in Scala, we cannot use this metric on them. The token counts thus reported by the tokenize module are shown in the right red bar in Figure 5.8. As the optimized versions of the pandas workflows have almost identical code complexity as the normal ones, we only show code complexity for the normal, unoptimized workflows for pandas.

For the short workflow, source code complexity as captured by the character count is similar for nearly all systems. This is because exploration and the linear regression algorithm itself are straightforward. Spark's use of Scala and a slightly more "wordy" API result in a relatively larger source code complexity. Looking at the token analysis, we can see that both AIDA and pandas have comparable numbers.

The long workflow has around 2x the source code complexity than the short workflow for all systems except for DB-UDF where it is 3.5x that of the short workflow if we look at the character count metric. Most of this can be attributed to the fact that UDFs cannot interact directly with each other and that the scope of the host language objects built inside the UDF cannot outlive the UDF's

[4]In particular, a SQL statement can be written in one line or spread over several lines.
[5]https://docs.python.org/3/library/tokenize.html

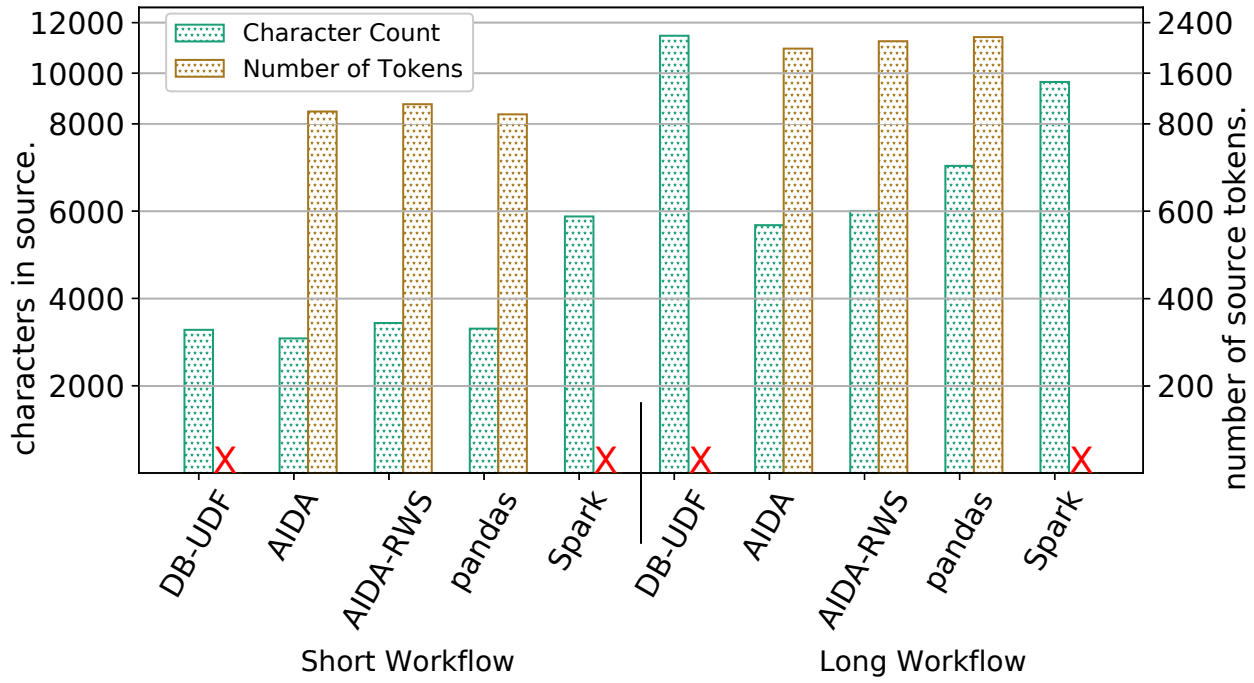


Figure 5.8: Source code analysis.

execution itself. Due to this, UDFs have reduced the reusability of HLL components created inside them, requiring the users to often duplicate some aspects of the source code across multiple UDFs. Recall also that, unlike interactive systems like AIDA, there is no mechanism in a UDF to keep reporting the changes in error rate as the training progresses. Further, although our workflows do not deliberately demonstrate this aspect, errors in an interactive system only require the user to address and resubmit the erroneous statement (This is especially convenient with a Jupyter Notebook like environment). Errors in UDFs, on the other hand, are hard to debug [HRK17], and an error in the last statement in a UDF can waste the computation performed in it up to that step and would require the user to re-write the UDF, increasing the user effort and programming complexity. Additionally, even for the long, exploratory type of workflows, AIDA is able to maintain the number of language tokens involved on par with popular interactive implementations like pandas.

Performance Figure 5.9 shows the computation time for each system not including any think or development time (non-logarithmic in seconds). As before, DB-UDF always performs best. Note that DB-UDF does not compute and report intermediate error rates during training, and thus, performs slightly less computation. AIDA-RWS has roughly 1.5x the execution time of DB-UDF, and AIDA has 3-4x the execution time, the bulk due to model training. Reducing RMI overhead

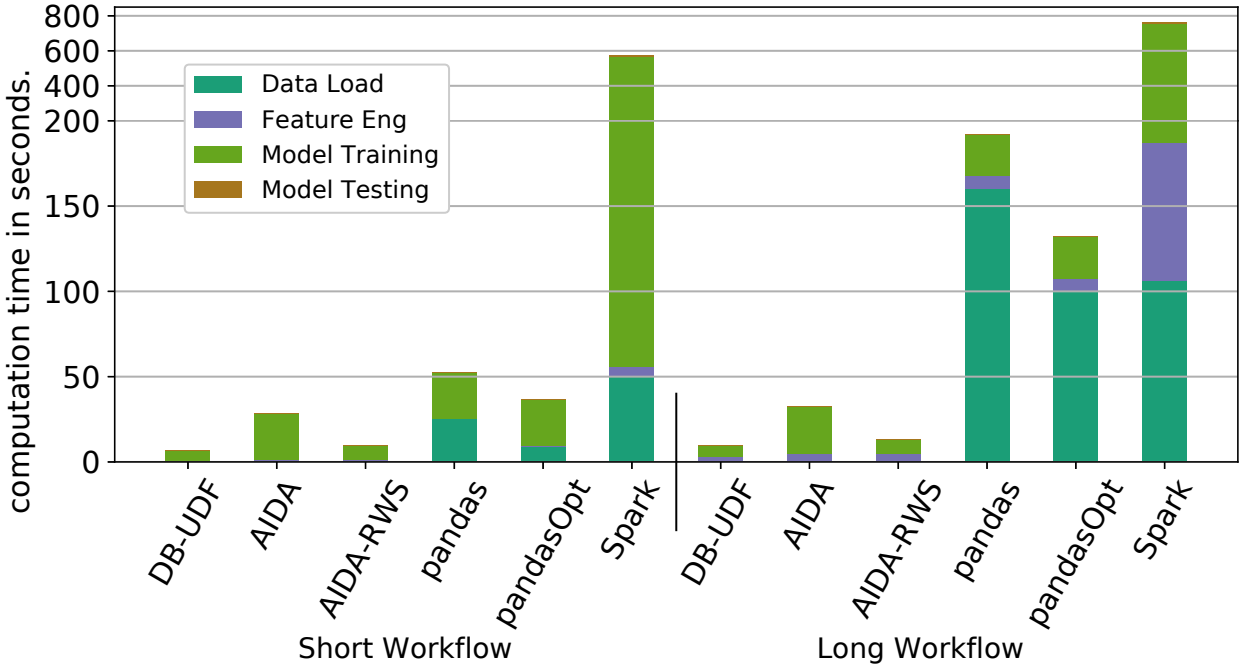


Figure 5.9: Linear regression on Bixi data set.

by pushing iterations to the server is benefiting AIDA-RWS. For both DB-UDF and AIDA, there is no large performance difference between short and long workflows because the main extra part for the long workflow is feature engineering which takes little execution time at the server side.

AIDA and AIDA-RWS always perform better than pandas and even the optimized pandas, and this by a very large margin for the long workflow. This is mainly due to the data load for pandas that AIDA avoids but also due to the feature engineering as pandas relational operators are not as efficient as MonetDB's. Spark is again the slowest by far. Although a *pushdown* of joins into the database has been proposed for Spark, they are still not available in the current release. As such, Spark has to perform its own relational joins, which, as we saw in Section 5.3.2 have a significant cost.

In short, we can see that while database UDF based approaches are computationally efficient, they are not very flexible to work with for exploratory analysis. On the other hand, while client-based systems are very user-friendly, they are interruptive with data transfer delays and are not on par with RDBMS optimizers when it comes to relational capabilities. AIDA, on the other hand, is providing a good alternative by being close to database UDFs in computational efficiency, while being on par with contemporary client-based systems in usability.

5.6 Evaluation Summary

AIDA, by imitating the agile programming style of interactive statistical frameworks, maintains their degree of usability. Data scientists, who have experience with pandas and NumPy should be able to learn AIDA fairly quickly. At the same time, AIDA can provide significant performance benefits compared to client-side packages as it performs computations *near data* and translates data in a way that is optimized for the vector-based data structures used by statistical libraries.

Compared to the alternative, database-centric, UDF-based solution, AIDA is generally slower; this holds mainly for smaller data sets where AIDA's framework and meta-data overhead has more impact. But in such cases, overall execution times are very small and the differences are not likely to be noticeable for the user. In contrast, we believe that UDF-based solutions are considerably more difficult to implement for data scientists, as the required SQL statements and the development steps can be complex, as shown in our example workflows. Furthermore, exploratory steps are badly supported in such implementations, hindering the usability aspect of such solutions.

6

Optimizing Queries over HLL Objects with Virtual Tables

Even though UDFs and stored procedures are not ideal for exploratory work due to their lack of agility, being computationally efficient compared to the alternative of extracting data into database-external systems makes them interesting for non-exploratory works. Further, our in-database framework AIDA that addresses the usability aspect of UDFs and stored procedures, internally makes use of UDFs to expose HLL data to the SQL execution engine.

As AIDA allows users to switch back and forth between linear algebra and relational operations, it internally has to pass data between HLL modules and the RDBMS. While optimized data transfers like *zero-copy* make passing data from the RDBMS to HLL easy and efficient, AIDA has to rely on building table UDFs to pass data in the other direction. However, contemporary implementations of UDF constructs follow generally a black-box setup, providing very little details about the data to be passed to the RDBMS optimizer, restricting its ability to produce an optimal query execution plan.

To address this, in this chapter, we propose and implement the concept of *virtual tables* that can be used to expose data set objects maintained by the embedded HLL interpreter to the query

engine for executing relational operations. The virtual table concept can be used by AIDA, as well as standalone UDFs, stored procedures, and other in-database analytics frameworks.

6.1 Introduction

Since UDFs facilitate efficient *near-data* computation, they are a popular choice with RDBMS implementers to provide some HLL based library support for algorithms and primitives required for in-database execution of data science and machine learning applications [KBY17, KLG+19, BS17, BS16, HRS+12]. For instance, [BS17] provides a predefined UDF that can be used by SQL queries and workflows to analyze the *semantic similarity* between the items purchased by different customers. Additionally, as we have demonstrated with our implementation of AIDA, it is possible to build in-database frameworks with exploratory support that address the usability aspect of UDFs by internally and automatically managing their creation and maintenance on an as-needed basis.

Table UDFs, such as the ones that AIDA internally uses to expose HLL data sets generated by its statistical library to the RDBMS in order to execute SQL queries over them, are very handy to build complex workflows.

Example 6.1. Listing 6.1 shows a simple Python table UDF that reads a CSV file containing user reviews for businesses (consisting of attributes `uid`, `bid`, and `review`) and exposes it to the RDBMS SQL engine.

Listing 6.1: A table UDF example.

```
CREATE FUNCTION ubreviews(filename STRING)
RETURNS TABLE (uid INTEGER, bid INTEGER, review STRING)
LANGUAGE PYTHON
{
  import pandas as pd
  df = pd.read_csv(filename, header=0)
  return df
};
```

As table UDFs produce a table-like structure in their output (in this example it is a pandas DataFrame), they can be easily integrated into SQL statements. For instance, in the SQL below, we are performing an aggregation to compute the number of reviews received by each business.

```
SELECT bid, COUNT(*)
FROM ubreviews('/data/reviews.csv')
GROUP BY bid;
```

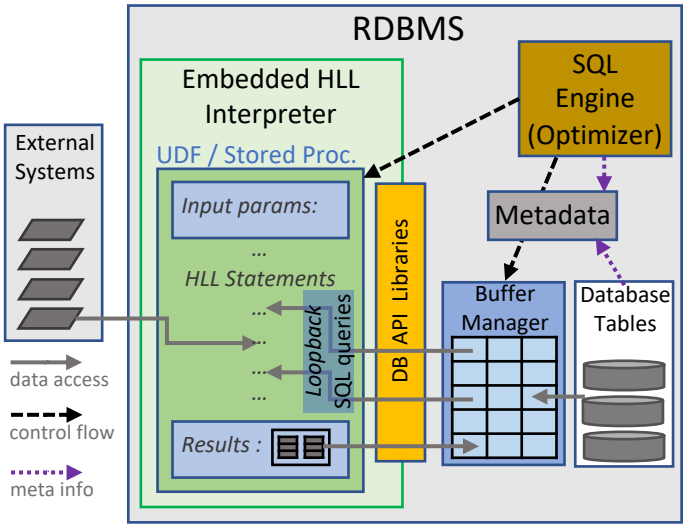


Figure 6.1: An example UDF concept.

However, the rise of such UDF-based approaches has also raised some new performance questions to be addressed. As was pointed out in Section 2.1.3, one of the keystones behind the success of RDBMS implementations is an optimizer that is capable of generating and analyzing the costs of different possible execution plans for a user request to choose an optimal approach for query execution. The introduction of UDFs into the system has disrupted this plan optimization process.

Figure 6.1 depicts the high-level concept of an HLL UDF/stored procedure and associated components, which we will explore in detail as we progress. While the code within the UDF can be optimized by the corresponding embedded HLL interpreter, the UDF itself is a black-box to the RDBMS optimizer, having very little insight into the behavioural characteristics of the UDF [RM16]. However, modern RDBMS optimizers rely on some kind of heuristics about the data (see Figure 6.1, metadata) to estimate the costs associated with execution plans [MCS88, JK84]. For example, most optimizers consider the cardinality of the tables involved in the SQL query to generate an optimal plan. For complex SQL queries, e.g., those involving several joins, such meta-information can be vital for the optimizer. The cardinality of the tables can be used by the optimizer to choose a sequence of processing steps that reduces the amount of data processed early in the query execution in order to minimize the overall execution time of the query and any associated resource requirements.

Non-availability of such meta-information can be an issue when using *table UDFs*. When a table UDFs is used in a SQL query, the optimizer has no insight into the characteristics of the data generated by the table UDF, and thus, might generate a suboptimal plan resulting in poor performance. We will be revisiting performance aspect of table UDFs in detail in Section 6.2.

Another issue is programmability. UDFs and stored procedures can usually execute SQL queries internally (see Figure 6.1, *loopback* queries), at any point in their execution to fetch additional data sets. However, data transfer in the other direction (i.e., back to the RDBMS) is more cumbersome. If an HLL computation produces data that should become the input of a SQL query, the only way so far to do so is to write a table UDF where that data is the return value of the UDF. Data transfer to a SQL query in the middle of an HLL computation is not possible (see *data access* paths in Figure 6.1). As we will see later in our examples, many complex workflows can benefit from programming flexibility if the HLL code is allowed to pass data sets back-and-forth with the RDBMS as required. We will also see that in the absence of this capability, users are often forced to program workarounds, causing the business logic to be broken across multiple source code artifacts, an inconvenience for source code maintenance, and often not very efficient.

Finally, as discussed in Section 2.4.5, UDF implementations such as [RM16, ML13] use a concept called *zero-copy* to optimize data transfer from the RDBMS to the HLL code, allowing the HLL in certain conditions to directly access the result of SQL queries in the RDBMS buffer without having to make a copy into HLL space. This is also beneficial for an in-database framework, such as AIDA that resides in the embedded HLL interpreter of the RDBMS, for it to access data from database tables as well as the result sets from SQL query execution, with minimal overhead. However, as we will see in Section 6.2, when the data returned by the table UDFs become the input of a SQL query, data conversions are so far handled in a much less elegant way.

In light of these drawbacks of contemporary UDF implementations, we propose a seamless and elegant approach to *register* table-like HLL objects (such as AIDA's TabularData object or a pandas DataFrame) to the RDBMS in the form of *virtual tables* to facilitate running SQL queries involving them. Therefore, the virtual table concept can be used by UDFs and stored procedures as well as RDBMS-embedded statistical frameworks such as AIDA. A virtual table exposes any necessary metadata, such as cardinality heuristics, to the optimizer, enabling the optimizer to generate better execution plans. It also allows UDFs and stored procedures to produce input for SQL queries during their runtime and not only as a return value.

Virtual tables also facilitate zero-copy data access from the embedded HLL code to the RDBMS whenever possible. When data conversions are required between the HLL and database data types, virtual tables perform this lazily, whenever the column is actually accessed by a query. This is more efficient than table UDF implementations, where an HLL attribute is converted to database format and materialized at the end of the UDF execution, even if it is not used further. Such capabilities can be crucial for the performance of workflows (such as the exploratory workflows of data scientists using AIDA) that move data back-and-forth between the HLL code and the RDBMS.

Furthermore, given that with this approach a data set can be easily accessed by both the embedded HLL interpreter as well as the RDBMS query engine, the user has the flexibility to choose the system most suitable for a specific computation. For instance, certain transformations associated with data science applications (such as *label encoding*) can be performed by using an HLL library but also through SQL queries. Depending on the scenario, one implementation might be faster than the other. The virtual table concept allows the user to choose the option that works best.

As our work on AIDA is already implemented and tested on MonetDB, and the fact that MonetDB already has support for zero-copy optimizations from RDBMS to embedded HLL code, we extend MonetDB for our implementation.

In summary, in this chapter, we will:

(i) Highlight some performance impacts and programming restrictions of current UDF-based approaches in integrating HLL code as part of RDBMS query processing.

(ii) Propose a virtual table based solution to use HLL data sets in SQL queries.

(iii) Develop a virtual table implementation for the MonetDB columnar-RDBMS that can perform lazy conversion of data sets and provide additional metadata such as heuristics to the optimizer for plan generation.

(iv) Extend AIDA's MonetDB database adapter to leverage virtual tables.

(v) Demonstrate the benefits of retrofitting conventional stored procedures and UDFs to use virtual tables to reduce programming complexity.

(vi) Demonstrate with an example, how virtual tables facilitate an efficient SQL implementation of data science functionality (*label encoding*), something that is conventionally supported by HLL statistical libraries.

(vii) Provide a detailed evaluation of the benefits of using virtual tables and comparisons with alternative approaches.

6.2 Database UDF Internals

As we had briefly introduced in Section 2.4.4, UDFs provide an elegant mechanism to integrate some HLL processing capabilities into an RDBMS with minimal extensions to the SQL syntax, allowing them to be easily integrated into typical SQL-based data processing workflows. This has made them quite popular with RDBMS implementers for extending traditional RDBMS functionality with minimal engineering work. Below we will continue to discuss some additional UDF concepts that are relevant to our work with virtual tables.

6.2.1 Integrating a Host Language to an RDBMS

The high-level details of providing HLL based UDF support in an RDBMS implementation can be broken down as follows:

Language Interpreter: Originally, host language based UDFs were written in compiled languages such as PASCAL, C, etc. [LKD+88], which required the RDBMS to compile the user code and load them as a library into the RDBMS address space. However, as this posed a considerable risk for crashing an RDBMS due to bugs in UDFs [Raa15, GMSvE98], the initial reception for the UDF ideology was not quite warm. Attempts to have a dedicated, standalone process, in conjunction to the RDBMS server to handle UDF processing to address the security issue was found to be inefficient due to the data communication overhead between the two separate processes [GMSvE98].

More recently, support for interpreted languages popular with data scientists such as Python, R, etc., coupled with a widespread application demand for such functionality is changing this trend [Mil16, WDG+16, ML13, RM16, The17]. These interpreted languages alleviate the safety concern associated with crashing the RDBMS server due to a faulty UDF. They also provide language interpreters as libraries than can be embedded inside a host application [Pyt18, R C99]. The RDBMS execution engine can thus invoke the HLL interpreter libraries to execute the UDF source code. This approach results in a shared address space between the RDBMS and the HLL interpreter. As we will discuss next, this provides some interesting optimization opportunities in passing data sets between the two systems, a technique that will also play a crucial role in our virtual table implementation.

Data Exchange: In general, the data types supported by the RDBMS and the host language need not be identical. Some of these could be fundamental differences such as the number of bits used to support a basic data type (e.g. *integer*). On the other extreme, there may not be a straight-forward corresponding mapping for a particular data type, such as the *interval* data types in SQL, requiring it to be represented with more generic data types such as *strings*. Therefore, any interaction between the two systems should also take into account the need for data conversions.

It is important to highlight that these data conversion needs are not unique to UDF implementations. As discussed in Section 2.2, applications written in HLLs are a common mode of accessing and interacting with data stored in an RDBMS. Such interactions, therefore, often necessitate similar data conversions. Most of these data conversions are defined as part of the ODBC, JDBC, etc., standards and are implemented in the corresponding driver libraries used by these applications to

interact with the database. In these conventional database application architectures, the application layer is either at the source or target end of the database processing steps, depending on whether they are sending data to the RDBMS or retrieving data (result set) after a query execution, respectively. As such, the conversion techniques and conversion costs are in general independent of the query execution plan of the RDBMS.

UDFs, on the other hand, can place HLL code right in between a set of RDBMS execution steps associated with an SQL query. Therefore, any data conversion that happen during the execution of a UDF will add up to the overall execution time of the SQL query that is using that UDF. These data conversions between the embedded HLL constructs such as UDFs and the RDBMS are usually handled by a database API layer (see Figure 6.1) that interfaces between the embedded HLL interpreter and the database memory buffers.

As discussed in Section 2.4.5, MonetDB applies a concept called *zero-copy* in which references to query result sets are handed over to embedded HLL modules, without making a copy or performing any conversion if the data types being transferred are compatible [LM14]. The same holds when the result of a table-UDF is used as input for an SQL query; the reference is handed over to the query engine, avoiding copy and conversion when possible. This is possible because many basic data types of the database are compatible with the HLL data types. However, whenever conversions are necessary because of incompatibility, they are done in an *eager* fashion, immediately upon the execution of a table-UDF. This can incur some unnecessary cost if some of these attributes are not used further in the processing of the SQL query in which the UDF is used. Further, as the HLL data objects inside the UDF do not exist beyond the execution of the particular SQL statement using it, optimizations like zero-copy cannot be employed across multiple SQL statements in a session, even though they are part of the same data processing workflow.

Additionally, as frameworks like AIDA often internally generate complex SQLs that consist of several table UDFs, scenarios like *self-join* where a data set is joined with itself [ME92], or SQL statements where a table UDF is referenced more than once, are a common occurrence. Although the data set produced by a given table UDF in AIDA is static, this is not a piece of information that the optimizer is privy to, given the UDF architecture. Hence, under such circumstances, the database might perform multiple data set conversions on the data set returned by the table UDF instead of leveraging a single instance.

Integration to SQL: Although newer versions of the SQL standard have caught up by providing SQL language extension guidelines on the integration of "external functions" such as HLL based UDFs and stored procedures [Eis96], as many of the contemporary implementations pre-date the

standard, minor variations exist between RDBMS vendors with respect to their approaches. However, most of these variations are pertaining to the Data Definition Language (DDL) aspect of SQL, as in the syntactic nuances involved in creating UDFs in the database. Given the limited scope of possibilities in which a UDF can interact in a SQL query statement, the Data Manipulation Language (DML) aspect of using UDFs in SQL queries have barely any variations between RDBMS implementations, most of them being merely cosmetic.

Example 6.1 already introduced the definition of a simple table UDF and its usage in a SQL query. We will be encountering several more instances of the definition and usage of table UDFs in the remainder of this chapter.

6.2.2 Table-UDFs

As we have seen in Section 2.4.4, a *table UDF* is a UDF that returns a table-like host language data structure. I.e., this data structure has columns and possibly multiple records. As columnar databases like MonetDB store and process their data sets using array-like structures to represent a column's data, host language data structures that have a dictionary-like interface where the column names are keys and their data is represented in some array/list format, make ideal candidates to be used as return types of table UDFs. Concrete examples of such data structures include the *dictionary-columnar* internal representation of *TabularData* objects in AIDA that we discussed in Section 3.3, as well as pandas' *DataFrame* objects.

Example 6.2. Listing 6.2 shows a table UDF, `ureviews`, which is a slightly augmented version of the table UDF in Example 6.1.

Listing 6.2: A table UDF to process user reviews

```
1  CREATE FUNCTION ureviews(filename STRING)
2  RETURNS TABLE (uid INTEGER, bid INTEGER, review STRING, lang STRING)
3  LANGUAGE PYTHON
4  {
5    from langdetect import detect
6    import pandas as pd
7    df = pd.read_csv(filename, header=0)
8    df['lang'] = df['review'].apply(detect)
9    return df
10 };
```

Similar to the original example, this table UDF reads a CSV file that contains user reviews for businesses. The CSV filename is passed to the UDF as an argument and it uses the pandas library to read the data from the CSV file (line 7). However, this modified version of table UDF then

proceeds to "enrich" the review data that it has read from the CSV file. It uses the review text to determine the language in which the review is written. For this purpose, it uses a Python-based language detection library, `langdetect`, to detect the language of the review text (line 8), storing it as a new attribute of the pandas DataFrame. The UDF then returns this pandas DataFrame, which now contains the original data from the CSV file, as well as the generated attribute, language (`lang`) of the review text, to the database.

The following SQL shows a usage of this table UDF.

Listing 6.3: Invoking a table UDF using SQL

```
SELECT * FROM ureviews('/data/reviews.csv')
WHERE lang = 'en';
```

The output produced is shown below.

```
| uid    | bid   | review                          | lang |
+========+======+=================================+======+
| 893412 | 8712  | This is such a great place!     | en   |
| 612351 | 1562  | The service is fast.            | en   |
```

Notice how a table UDF can be used in a manner similar to a regular database table with the exception of integrating a standard programming language function call. The column names of the data generated by the table UDF can be referred to in the SQL similar to regular table column names (in the above example a `WHERE` clause is applied to select reviews written in English). Referring back to Figure 6.1, the steps for this SQL execution consist of first executing the UDF using the embedded HLL interpreter, then converting the result data set of the UDF into the database data format and storing it in the database memory buffers, and finally performing the actual SQL query on this data set, that is, the conventional relational *selection* operation. As demonstrated by the above example, a table UDF can be used in complex SQLs akin to regular database tables. They can even be joined with other table UDFs or regular database tables. However, unlike regular database tables, the optimizer may not be able to generate an optimal query execution plan for such SQL statements, as it has no prior information on the characteristics of the data set produced by the UDF during the plan generation.

6.2.3 Executing SQL from an HLL - Loopback Queries

UDFs/stored procedures can typically execute SQL statements on the database through a database API (see Figure 6.1). Termed *loopback queries* [Raa15], this allows for the development of com-

plex HLL code that can read multiple data sets from the database.

Example 6.3. Python UDFs in MonetDB can execute SQL queries from within as shown below.

```
res = _conn.execute('SELECT * FROM users;')
```

Here `res` is a Python object returned by the database API after executing the SQL query passed to it as the argument.

As previously mentioned, MonetDB has optimized the movement of such data into the HLL using the zero-copy concept. However, the scope of loopback queries is limited to regular SQL statements, being able to run queries only over regular database objects such as tables, views, etc. For example, if we go back to the code in Listing 6.2, one cannot run a loopback SQL query over the `df` object in the Python UDF, even though it has a table-like data structure. The only way to execute a SQL over it is to return it to the RDBMS engine at the end of the UDF execution so that we can write a regular SQL statement over the UDF itself, as demonstrated in the original example using Listing 6.2. Of course, this will result in terminating the HLL code flow of the UDF at the `return` statement, not allowing the programmer to include any additional HLL processing logic in the UDF. As we will see in the next section, this limits the programmability of such solutions, resulting in unnecessarily convoluted code fragments.

6.2.4 Complex UDF-based Workflows

Implementing complex workflows with UDFs/stored procedures still faces several programming challenges.

Example 6.4. Let us say if we want to do some data cleaning and discard reviews by user ids that are not present in our database `users` table. We could do that as follows.

```
SELECT * FROM ureviews('/data/reviews.csv')
WHERE uid IN (SELECT uid FROM users);
```

The RDBMS first executes the `ureviews` UDF, followed by the *selection* operation which uses the subquery to retrieve valid user ids from `users` table to discard any irrelevant user review records returned by the UDF. But this means that we will perform language detection on many review texts though they are potentially discarded later, as they do not belong to a known user. Ideally, we would like to discard the reviews from unrecognized user ids before performing language detection, as the later is an expensive operation. However, this requires to perform the *selection* SQL on `df`, which is a pandas (Python) object. In order to do so with current UDF frameworks, one needs two

UDFs and a SQL with additional processing logic to glue them together as demonstrated by our next example.

Example 6.5. The first table UDF, `loadrvs`, loads the data from the CSV file into a pandas DataFrame, which is then passed on to the RDBMS. The RDBMS then filters this data set to select only the reviews by valid user ids, passing them on as the input to the next table UDF, `detectlang`, that performs the language detection and adds the additional language column.

Listing 6.4: Processing only valid user reviews

```
CREATE FUNCTION loadrvs(filename STRING)
RETURNS TABLE ...
{
  ...
  df = pd.read_csv(filename, header=0)
  return df
};

CREATE FUNCTION detectlang(..., review STRING)
RETURNS TABLE ...
{
  from langdetect import detect
  return {..., 'lang':[detect(r) for r in review]}
};

SELECT * FROM detectlang
  (SELECT * FROM loadrvs('/data/reviews.csv')
   WHERE uid IN (SELECT uid FROM users));
```

As we can see from the above example, this results in code fragmentation, and now requires three source code artifacts, which is quite cumbersome.

6.2.5 UDFs and RDBMS Optimizer Query Execution Plans

Cost models of query optimizers take various statistical characteristics of the input data into account to find the best execution plan for a given query, such as the cardinality of the data sets [MCS88, JK84]. This is, for instance, the main characteristic taken into account by MonetDB. High-end commercial RDBMSes, on the other hand, can often leverage additional characteristics, such as histograms of distributions of individual columns, outliers(skew), etc., [GMP02, Ioa96, Lyn88]. Such statistical metadata is especially important for the optimizer if the query is complex and there are many possible execution plans to consider.

Use of UDFs, especially table UDFs, disrupts such cost-based optimizations. Taking Listing 6.4 as an example, the optimizer has no insight into the cardinality or other characteristics of the data set returned by the table UDF `loadrvs` which can result in a suboptimal execution plan being chosen. The more complex an RDBMS optimizer's cost profile model is, the bigger the difference could be between the chosen and optimal execution plans.

To understand this issue better, let us consider a simple scenario where there is an index on the `uid` column of the `users` table. Indexes are very useful if we are only interested in a small subset of records in the table [SKS11]. This is because, indexes provide an "alternate path" to find the records of interest in a table, without having to search through the entire table in the disk, as that could be very expensive for large tables. Since our objective is to ascertain that a user who has written a review also exists in our `users` table, this can be accomplished through a SQL *join* between the `uid` in the reviews data set with the `uid` of the `users` table.

Internally, the RDBMS optimizer has a variety of options to execute the join operation. As there is an index on the `uid` column of the `users` table, an *index-join* is a possible alternative to the conventional *block-nested loop join* [ME92, Zho18]. However, deciding between these two alternatives requires the optimizer to know the cardinality of the reviews data set, among other things. This is because it has been shown that an index-join is cheaper than a block-nested loop join only if the number of lookups is small. Thus, if we have only a few reviews and thus few users to find, an index-join is the best solution. On the other hand, if there are many reviews, then probably the block-nested loop join will be the better option.

Therefore, we can see how such data heuristics are crucial for the optimizer to generate an efficient execution plan, and how the black-box nature of the UDFs interferes with it. As a workaround for this issue, one can conceive of breaking down a SQL statement involving table UDFs into multiple statements. We could execute a table UDF, dump its data into a *temporary table*, and use this temporary table in place of the table UDF in the original SQL query. As mentioned before, since table UDFs produce table-like data sets in its output, swapping them in SQL statements with a real database table can be easily accomplished in code.

Example 6.6. In Listing 6.5, we employ this modified approach of using temporary tables.

Compared to the original approach in Example 6.5, where we did all the data processing in a single SQL statement, in this example, we are doing the data processing in two steps. The first SQL statement involves executing the `loadrvs` table UDF and storing its output as `newureviews` temporary table. The second SQL statement is tasked with selecting the relevant user reviews from `newureviews` and executing the language detection UDF, `detectlang`, over them.

Listing 6.5: Processing only valid user reviews using temporary tables.

```
CREATE TEMPORARY TABLE newureviews
AS SELECT * FROM loadrvrs('/data/reviews.csv');

SELECT * FROM detectlang
  (SELECT * FROM newureviews
   WHERE uid IN (SELECT uid FROM users));
```

Although with such an approach the database optimizer can now derive the data characteristics from the temporary table to generate an optimal plan, the benefits of zero-copy optimization are lost. Within a SQL statement, zero-copy optimization will allow a UDF to handover data sets to the next processing step directly when the HLL data types match with that of the database. However, once we break this into multiple SQL statements, even if the output of the UDF is materialized into a database table, zero-copy cannot be used anymore. This is because the HLL objects created by the UDF do not exist anymore after the execution of the SQL statement that invoked them. Keeping this data around for the next SQL statements as a database table requires a copy to be made into the RDBMS buffers at the end of the execution of the first SQL statement.

Yet another alternative to the above approach is to iterate over the HLL object from a UDF or stored procedure and insert them record by record using loopback queries into a database table, on which the SQL query can then be executed. While this allows us to write the complete data processing logic in a single UDF/stored procedure, such iterative approaches are not efficient. This is because they are not capable of moving data in bulk, nor do they support optimizations such as zero-copy which functions only when the entire underlying data structure is accessible by the other system.

Example 6.7. Listing 6.6 shows this approach being applied to our original problem of selecting valid user reviews and applying language detection over them.

Here, once we load the pandas DataFrame from the CSV file, we iterate over each row in it and insert them into the temporary table, `newureviews`[a] (line 8). Now that the reviews are loaded into a database table, we can execute a loopback query to select only the valid reviews (line 12), and apply the language detection on just those selected reviews (line 15). This allows the optimizer to make an efficient query plan for the selection of valid user reviews. This is because `newureviews` is a database table, and for SQL queries executed through loopback API, the execution plan for that SQL query is generated only when the loopback API is invoked and not at the beginning of the UDF's execution. However, such an iterative approach of moving

data into the database now introduces a new performance overhead, as it cannot perform bulk data transfers or leverage optimizations such as zero-copy.

Listing 6.6: Processing only valid user reviews using a single UDF

```
CREATE FUNCTION ureviewslang(filename STRING)
RETURNS ...
{
  from langdetect import detect
  import pandas as pd

  df = pd.read_csv(filename, header=0)
  for index, row in df.iterrows():
    _conn.execute("INSERT INTO newureviews
                   VALUES({uid}, {bid}, '{review}');".format(**row))

  df = pd.DataFrame(_conn.execute(
      "SELECT * FROM newureviews
       WHERE uid IN (SELECT uid FROM users);"))
  df['lang'] = df['review'].apply(detect)

  ...
};
```

[a]For simplicity, we have not shown the table creation steps here.

In the performance evaluations that we will discuss in Section 6.4.3, we will be comparing the cost overhead of these alternative approaches that we discussed above.

6.3 The Virtual Table Approach

When developing our solution aiming in addressing the drawbacks of UDFs, we had the following design objectives.

(i) Minimizing the amount of change that needs to be made to the RDBMS engine is a key aspect when adapting legacy RDBMSes to novel ideas. Therefore, our approach is to present the host language data as a table-like "view" to the RDBMS components and take care of any data conversion necessary. Thus, the RDBMS does not need to be concerned with the data representation format. We refer to this as a *virtual table*.

(ii) No new syntax will be added to the SQL language to make use of virtual tables. Virtual tables will be accessible in SQL statements in a manner similar to regular database tables.

(iii) Users should be able to use an HLL API (Python-based) to register HLL language objects that have a table-like data structure. This can be performed at any point in the HLL code, unlike UDF implementations where the bulk data transfers could be only performed at the end of its execution, in the form of its result.

(iv) HLL objects already have a lot of metadata associated with them (e.g., data types, the size of the data set, etc.). Therefore, we will leverage this information and expose it to the optimizer, when possible, in order to facilitate an efficient execution plan.

(v) The virtual table implementation needs to facilitate lazy data conversion. That is, if a column requires data conversion because of incompatible data types, this conversion takes place only if an SQL query actually accesses the column.

(vi) The data set associated with a virtual table is cached for the scope of the entire UDF/stored procedure execution (or a client session, if the user chooses this option).

(vii) The purpose of the virtual table implementation is not to replace the UDF/stored procedure concept as a whole, but to address their specific drawbacks. Therefore, where applicable, they need to co-exist beneficially. I.e., a UDF or a stored procedure should be able to make use of a virtual table to write easier and/or efficient code.

Figure 6.2 shows a high-level architecture of the virtual table concept. The database components that need to be extended to facilitate virtual tables are highlighted in blue. At the heart of this implementation is the virtual table library (*vt-lib*), that is loaded when the database starts up. vt-lib consists of two components, (i) a Python API that users and frameworks can invoke from their stored procedures and UDFs, (ii) and an RDBMS module that facilitates the other RDBMS components to interact with virtual table objects.

Although our implementation is built by extending MonetDB, we believe a similar architecture can be employed for other RDBMS implementations. In particular, for any column-based RDBMS that provides some support for zero-copy, we expect an easy integration that provides high efficiency in workflow executions.

6.3.1 Virtual tables API

In Listing 6.6 we have seen a table UDF that is able to "send" data to the RDBMS without compromising the optimizer's access to data heuristics, and maintains the code in a single artifact. But it is inefficient due to the iterative nature of data transfer to the RDBMS which also compromises the zero-copy optimization. We will see how the virtual table based approach addresses these specific drawbacks, without sacrificing any of the original benefits. Stored procedures or UDFs that wish to utilize the facility of the virtual tables can make use of the Python API of the library.

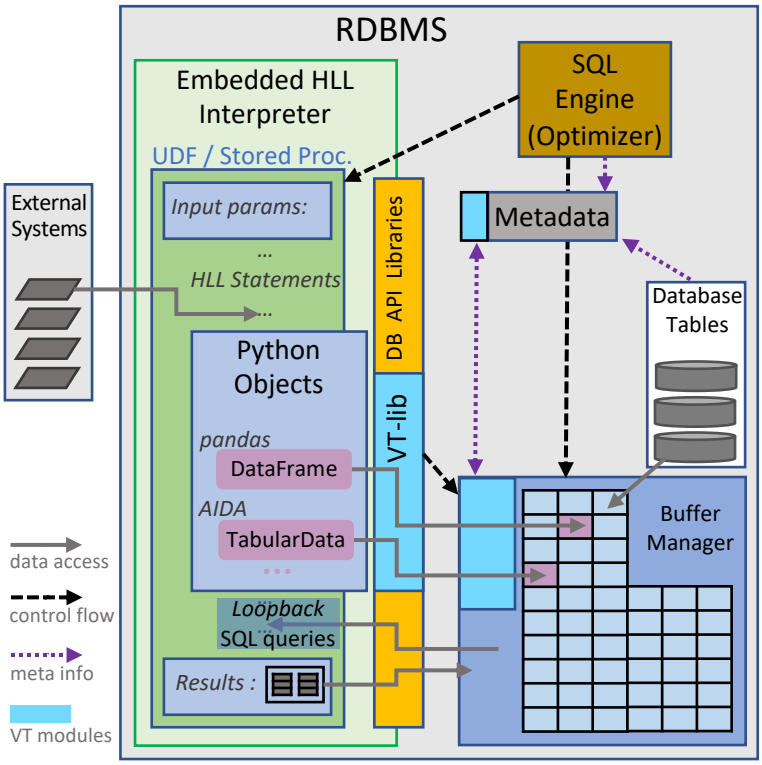


Figure 6.2: Virtual tables implementation.

The virtual table API allows users to register Python objects that provide some concept of a "key" (for column names) and "values" (an array or list type structure that represents the values for a given column). Some examples would be the DataFrame object of the pandas library, or the *dictionary-columnar* internal data representation of the TabularData objects in AIDA.

Example 6.8. Listing 6.7 shows a modified version of the user review processing table UDF from Listing 6.6, which makes use of vt-lib.

Here, the UDF first creates a virtual table manager, which is responsible for keeping track of the Python objects that a UDF is using as virtual tables (line 6). Once the UDF has loaded the user reviews from the CSV file to a pandas DataFrame, it can now register the pandas DataFrame as a virtual table (line 9), giving it a name, `rdata`, which can be referred to in SQL statements. This replaces the iterative statement that is in Listing 6.6 which is inefficient. Virtual-tables enable bulk data transfer back to RDBMS from HLL code and as we will see in the next section, facilitate zero-copy optimizations in doing so. Using this virtual table, `rdata`, now we can execute an SQL query to filter out the invalid user ids (line 11) and run the language detection library only on the filtered data. As virtual tables appear like regular database tables

to the optimizer, with access to its data heuristics, this will also enable the optimizer to produce efficient query execution plans.

Listing 6.7: VT-lib usage example

```
CREATE FUNCTION ureviewslang(filename STRING)
RETURNS ...
{
  ...
  from vtlib import VTManager
  vtm = VTManager()

  df  = pd.read_csv(filename, header=0)
  vtm.regTable(df, 'rdata')

  df  = pd.DataFrame(vtm.executeQry(
        'SELECT uid, bid, review FROM rdata
         WHERE uid IN (SELECT uid FROM users);'))
  df['lang'] = df['review'].apply(detect)

  ...
}
```

Programmability: As we can see, virtual tables make it easier to write data processing workflows as there is no more the need to write multiple table UDFs and SQL statements to glue all the bits together (such as the one that we saw in Listing 6.4). Instead, users can easily expose a Python object to the database via a call to the Python API of the vt-lib. Therefore, while we needed three different source code artifacts in Listing 6.4, only one is needed with the virtual table approach in Listing 6.7.

Scope: By default, when a UDF/stored procedure's execution is completed, the virtual table manager will automatically remove any virtual tables that were registered. However, the user can pass optional arguments to the API to indicate that the virtual table is maintained beyond the execution scope of the UDF or stored procedure that created it. Under such circumstances, the onus of when to remove a virtual table is on the user and must be done explicitly. As virtual tables are basically a table-like "view" for the RDBMS into a host language object, they are practically memory resident and do not survive any restarts by default. Users can pass additional options to the API to indicate that a particular virtual table must be persisted, in which case the data is copied into database buffers and the virtual table becomes a regular table. This transformation into a regular table is useful if the users want to store some of the transformed data sets of their workflow for future use.

6.3.2 Interfacing RDBMS with Virtual tables

When the HLL code registers a Python object, the vt-lib module updates the metadata maintained by the RDBMS to indicate that there is a new table. In Listing 6.7, for instance, Python object `df` will be registered as table `rdata`. For this purpose, we have augmented the RDBMS metadata fields to capture some additional information. One of these is an indicator to denote whether this metadata entry actually refers to a Python object or a regular database table. Heuristics contained in the Python object, in our case cardinality, are also recorded in the system metadata. The vt-lib module also updates the column metadata with the attribute information associated with the Python object, indicating the names and data types of the columns contained in it. Once a virtual table is registered, if required, the user can collect additional heuristics explicitly by using the database specific commands (e.g. `ANALYZE` in MonetDB) similar to regular database tables. Of course, running `ANALYZE` and alike can be a costly operation itself, and the overhead might only be worth if a virtual table is used in many queries. When a query that uses this virtual table (e.g. line 11 in Listing 6.7) is submitted to the RDBMS, the latter now has access to this metadata and heuristics to produce an optimal execution plan. This works because, as mentioned before, the plans of such a loopback query are generated at the time the query is called and not at the beginning of the enclosing UDF/stored procedure's execution.

Sharing Data: How the actual data itself is represented is based on the data type of the column. As [Raa15] points out, for many basic data types such as integers, MonetDB uses a *C-style* array to store data. This is fundamentally identical to the underlying storage structure used by Python objects such as those in NumPy, pandas, AIDA, etc., for such data types. In fact, [Raa15] leverages this to their advantage while reading data from the database into a UDF, performing the zero-copy optimization that we discussed before. Therefore, we rely on the same principles and instead of making a copy of this data, it is retained as-is. Although not essential, this is simplified by the fact that MonetDB uses *memory-mapped I/O*[1], a concept that allows programs to access data from files as though they were memory resident. Essentially, when the database buffer manager (see Figure 6.2) sees a request for a particular column, it maps that column's file into a memory location. The rest of the RDBMS modules basically work with the column's data as though it is a memory resident array. In the case of virtual tables, as the buffer manager sees that the data is already in the memory (as HLL objects are memory resident), it will just pass that location to the RDBMS for further processing.

[1] https://en.wikipedia.org/wiki/Memory-mapped_I/O

Lazy Conversions: However, there are other data types that require conversion into corresponding database types. For example, *string* in MonetDB is stored differently compared to Python. This requires conversion from the Python data type to the corresponding RDBMS data type. Vt-lib delays this conversion process until the column's data is really needed for the first time. If a column is never used in any further SQL statement, the overhead of conversion never occurs. This is particularly useful for exploratory frameworks like AIDA that do not know in advance which columns from a data set will be requested in a SQL statement in the future. Therefore, the virtual table concept allows such frameworks to expose the entire data set to the RDBMS without worrying that it might have columns that are never used afterward. These columns will not pose a conversion overhead with virtual tables. In contrast, in the case of table UDFs, MonetDB immediately converts all the columns of the return value that need conversion. While lazy conversion could in principle also be possible, it might be challenging to determine the columns that are actually used when the execution logic is complex.

Lazy conversion is implemented in vt-lib as follows. When a column that needs conversion is registered, vt-lib attaches a "conversion" function to the column's metadata. vt-lib has a repository of conversion functions to convert from most of the commonly used Python basic data types into a corresponding database type. When the column is actually used in a query and the buffer manager looks up the column metadata, it will notice the conversion function and activate it. The conversion function produces a memory-resident data structure compatible with the corresponding database data type. This process happens only once, namely when the column is first accessed. The buffer manager will use this converted data structure for any future reference to the column.

Handling modifications: Lazy conversions can pose issues when data is modifiable. Once vt-lib performs the conversion of a column, there are essentially two copies of data for this column. One maintained by vt-lib in the database format, and another in the original HLL object, if that is still used in the HLL code. This is not an issue in the case of AIDA. Its TabularData objects are immutable, as transformations always create new TabularData objects. Thus the two versions of the column (one part of the HLL object, the other in database format), will always be identical.

Problems can arise with libraries such as pandas that allow *in-place* modification of data sets. If a user makes a change to a column in the HLL object that was already converted, that change will not reflect on the copy maintained by vt-lib. To prevent such accidents, vt-lib allows users to request for strict enforcement of data consistency. With this option, any data set that is registered as a virtual table is marked as *read-only*. This will ensure that any attempts to modify the data set in the HLL code will result in an error notification. Others have applied similar techniques [Raa15],

and users can always create a modified copy of the data set to work with that does not change the original data set.

A more elegant approach is possible if libraries that allow in-place modifications provide an *event-listener* type interface to alert data modifications. The vt-lib module can be easily extended to create such event-listeners upon the registration of an HLL object as a virtual table, and automatically discard the converted column data if the original host language data for the column has been changed, restarting the lazy-conversion cycle.

Data Characteristics for the Optimizer: In order to provide the RDBMS optimizer with information on the data set's characteristics, the virtual table implementation provides a *profile-function* that the optimizer can invoke for this purpose. This profile-function is capable of accessing the HLL object's metadata and passing this information back to the optimizer. As many HLL objects already maintain metadata such as the size of the data set, the data types, etc., this information can often be obtained with negligible cost.

Virtual tables and AIDA: We implemented an alternative version of AIDA's database adapter for MonetDB, to take advantage of virtual tables. In this version, the database adapter, instead of creating table UDFs for any materialized TabularData objects over which it needs to execute a SQL query, simply registers the TabularData object's internal dictionary-columnar data representation as a virtual table. In fact, the alternative adapter is much simpler in its structure, as virtual tables are much less clumsy than the code in the original adapter that automatically generated the table UDFs. As the changes to the framework are modular and confined to the database adapter implementation, this does not impact any of the remaining AIDA modules, most importantly, the client APIs of AIDA. We will see later that by leveraging virtual tables instead of conventional table UDFs, AIDA is able to provide improved performance for many of the relational operations occurring in data science workflows.

Virtual tables and other HLLs: While our implementation is in Python, in principle, the concept can be easily extended to any other HLL embedded in the RDBMS. However, zero-copy optimizations may be limited to HLLs with data structures that are similar to the underlying RDBMS, such as the case with the R language and MonetDB [LM14].

Virtual tables and Row-stores: Even though column-store architectures have become commonplace for large scale analytics, there is still a significant number of row-store based implementations in existence. The primary concern in applying the virtual table concept to row-stores is that

row-stores have record formats that are often complex and do not align with the vector-computing data models of statistical packages. Thus, there is less opportunity for zero-copy optimization. Nevertheless, a virtual table implementation can still expose structure and heuristics to the optimizer for efficient plan generation. Furthermore, lazy conversions can still be possible if the database can *pushdown* projections into the virtual table layer.

6.3.3 Opportunities for New Solutions

As discussed in Section 2.3.2, the evolution and popularity of data science libraries and frameworks external to RDBMS have led to the duplication of several functionalities that have traditionally been the stronghold of RDBMS. UDF based implementations and frameworks like AIDA address the issue to some extent by pushing back the responsibility of executing conventional relational operations into the RDBMS [RM16].

Still, data science computations apply an increasingly larger set of standardized data transformations for data preparation and data analysis that seem to be more naturally expressed through HLL concepts. Therefore, statistical packages offer these transformations conveniently through libraries, such as scikit-learn [PVG+11]. An example transformation is *label encoding* which is commonly used to replace a set of string values with a numeric encoding that leverages a dictionary. Interestingly, some of these transformations, including label encoding, can also be executed by the RDBMS, although the corresponding SQL statement might be quite verbose and somewhat complex. Nevertheless, it might be very attractive to go via the SQL path if the RDBMS can perform such transformations faster than the HLL interpreter. This is where the concept of virtual tables can play an important role. By using virtual tables, HLL data sets become easily accessible to the RDBMS engine with very low transfer overhead and great potential for optimization. Thus, it facilitates implementations of data transformations that are executed by the SQL engine instead of the HLL interpreter.

To facilitate execution through SQL equivalents, an HLL wrapper library can abstract away the tediousness involved in generating a complex SQL, by automating several aspects of it, as has been demonstrated [FEB18] previously. Stored procedures, UDFs or frameworks such as AIDA can then use these libraries in combination with the virtual table concept to expose data sets to the RDBMS, and execute automatically generated SQLs to produce the required transformations without increasing programming complexity. We will cover such an example in our evaluations.

6.3.4 Virtual Tables vs. Table UDFs – Summary of Comparison

To summarize, we can see that the virtual table concept has several advantages over an approach that is purely based on conventional table UDFs.

(i) *Easier to program:* Unlike table UDFs that can return data sets to the database only at the end of its execution, virtual tables can be used at any point in an HLL code to expose HLL data sets to the RDBMS. This allows users to write compact code when they need to use both HLL and SQL functionality in tandem.

(ii) *More flexibility in where to execute tasks:* As we discussed in the previous section, virtual tables enable the data sets to be easily passed back-and-forth between the HLL and RDBMS systems, opening up new programming possibilities. In particular, depending on the task and data at hand, certain computations can either be done by the HLL interpreter or by the RDBMS query engine.

(iii) *Optimization Opportunity:* Compared to the black-box approach of table UDFs, virtual tables expose the data characteristics to the optimizer, offering more potential to come up with efficient plans for complex queries.

(iv) *Smart Data Conversion:* As virtual tables follow a lazy conversion approach, attributes are converted only when they are first accessed, reducing the cost of queries when a SQL statement does not access all the attributes from the virtual table.

(v) *Use cases - Virtual tables vs. UDFs:* As we mentioned in the objectives, virtual tables are not a comprehensive replacement for UDFs. Instead, it is what we believe a more elegant programming model for use within UDFs, stored procedures or server-based frameworks like AIDA, to provide a better interface with the database system. Virtual tables do not perform any computation of their own, but a given virtual table is associated with a particular host language object at a given point and allows for a seamless and more efficient integration with the RDBMS. A virtual table offers an effective way of exposing the actual data to the RDBMS for an optimal query processing strategy, providing performance and programming advantages over a plain table UDF based approach. That is, virtual tables replace table UDFs when they are awkward or inconvenient to use or when they hinder performance.

6.4 Evaluation

6.4.1 Test Setup

Our test node's hardware configuration is Intel® Xeon® CPU E3-1220 v5 @ 3.00GHz and 32 GB DDR4 RDIMM main memory, running on Ubuntu 16.04. For software, we use MonetDB v11.29.3, pymonetdb 1.1.0, Python 3.5.2, NumPy 1.13.3 and pandas 0.21.0. Unless explicitly specified, default settings are used for all software. In all our experiments we only start measuring once a warm-up phase has been completed for the database to eliminate any I/O overheads.

As our objective is to measure a variety of use cases that include performance, programmability, etc., we have leveraged a variety of data sets, including some that have been used in other works, for the purpose of evaluating our implementation. Thus, our test data sets consist of TPC-H [Tra17], that we already discussed in previous chapters, North Carolina voters data set used in [RHMM18] for conventional UDF based data science solutions, and the Bixi bicycle trip data set that we originally used in Section 5.5 to demonstrate the capabilities of AIDA on a real-world end-to-end data science analysis. Additionally, for some test cases, we also use some synthetic data that we will describe in the corresponding test section.

In our experiments, we did not run any `ANALYZE` function on virtual tables to collect advanced heuristics, but only provide the cardinality information contained in the host language object to the RDBMS. This is because performing `ANALYZE` itself is a costly process, and is not always useful. Therefore, the criteria for choosing the tables and columns that should be *analyzed* by the RDBMS are often determined by the database administrators by studying the database workloads [MCS88]. While in some cases it could be beneficial, the transient and dynamic nature of virtual tables makes them probably less favourable candidates to do this. Further, not incorporating an explicit `ANALYZE` function to collect additional, resource intensive heuristics will put virtual tables on a fair comparison platform with table UDFs, as the latter does not have any such facility.

6.4.2 A Study Using TPC-H Queries

To understand the performance implications of using virtual tables instead of table UDFs, we first decided to test using the TPC-H Benchmark. As TPC-H queries are fairly complex, this will help us better analyze the influence of data characteristics that the optimizer uses to generate the execution plan. For this purpose, we translate the SQL queries in TPC-H using AIDA's API. This is convenient for testing as AIDA supports the execution of relational operations on both database tables and host language objects. The database adapter of AIDA can automatically build table-

UDFs or virtual tables on such objects in order to execute SQL queries over such objects. Our test suite consists of all the TPC-H queries, except queries 20 and 21 which we could not translate using the current API of AIDA. We also test the queries across various scale factors (SF) of TPC-H, viz., SF 1, 2, 3 and 4.

We setup AIDA to test the following scenarios:

(i) Data set is in regular database tables (`db-tbl`). This is our base case, expected to give the best performance as there are no data conversions involved and the database optimizer has access to the data heuristics of the table to generate efficient execution plans.

(ii) Data set is stored in python objects and AIDA uses table UDFs to expose the data to the RDBMS (`tbl-udf`). Most of these queries will incur data conversion costs as well as end up with suboptimal execution plans.

(iii) Data set is stored in python objects and AIDA uses virtual tables to expose this data to the RDBMS, but we force them to materialize eagerly, similar to table UDFs (`v-tbl`). While having the data heuristics exposed to the optimizer should provide these queries with efficient execution plans, they still incur overhead for any data types that require conversions.

(iv) Data set is stored in python objects and AIDA uses virtual tables to expose this data to the RDBMS and conversion is performed lazily `v-tbl(lazy)`. This is the ideal scenario for executing queries over HLL objects. The data heuristics exposed to the optimizer will be beneficial for query plan generation. Furthermore, while virtual tables will still need to perform some data conversions, such conversions will be restricted to only those attributes that are being actually used in the SQL query and will be performed when they are first used.

Each query is executed twice by the client, and we record the execution cost for each instance separately with suffixes `-1` and `-2` respectively. We expect the second execution to be faster for the virtual table implementation as it performs any data conversions only once for a given attribute. The execution times do not include the time for setting up data sets or any data transfer to the client. We would also like to note that the coefficients of variation for both of the test cases are under 10%, showing low dispersion of the observed data points. Thus, we do not discuss them further.

All Data Sets as Python Objects

In a first test case, for all scenarios except `db-tbl`, we first load all the TPC-H data sets into Python objects. That is, for `tbl-udf` and the two `v-tbl` versions, all the data for the queries reside in HLL objects. In AIDA, this is implemented as a TabularData object, which, as discussed in Section 3.3,

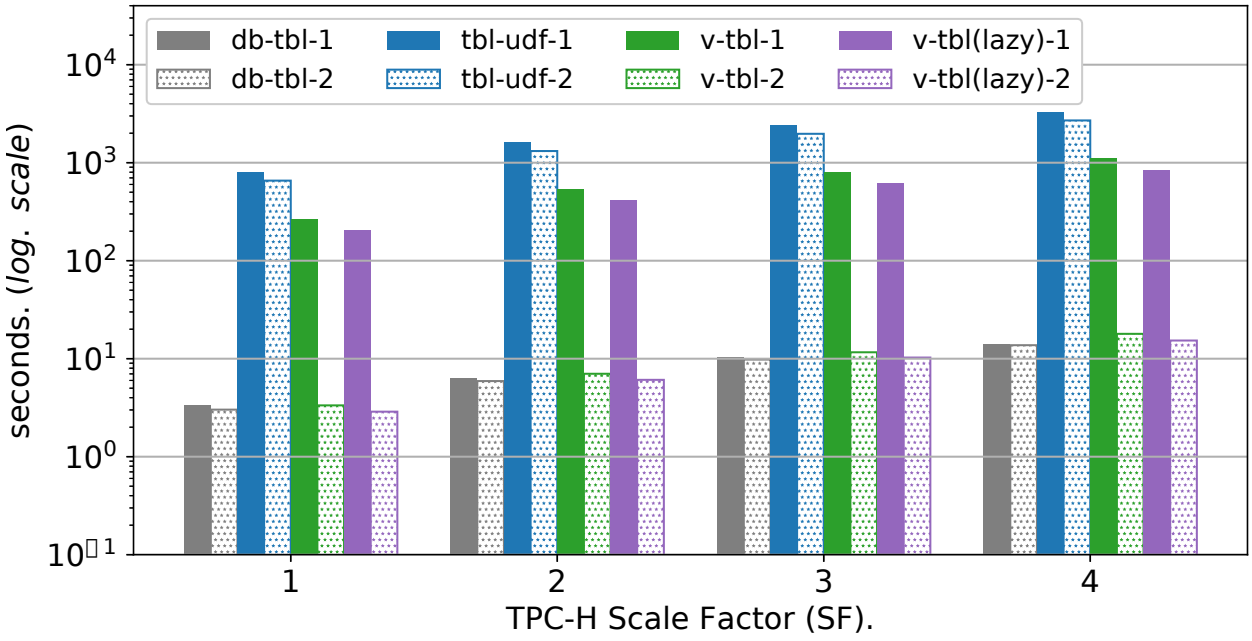


Figure 6.3: TPC-H Queries (all data sets as Python objects).

internally contains a dictionary-columnar structure where a key is a column name and the value corresponds to an array that has the data for that column.

Figure 6.3 shows the sum of all execution time in seconds for all the queries for each of the different scale factors. Individual queries show a similar trend. Unsurprisingly, queries are efficient when run directly on database tables (`db-tbl`). The second runs (`db-tbl-2`) are slightly more efficient compared to the first runs (`db-tbl-1`), predominantly due to minor changes like instruction cache warm-up, but not significantly.

Table UDFs, on the other hand, are the most expensive, costing around 240 times more than running queries against database tables. On the second run, they are still around 210 times more expensive. This is also exacerbated by the fact that even for the second execution, the database performs the data conversions again on the output of the table UDFs, although it is the same data.

When we analyze the performance of the virtual tables with eager conversion, we can see that during the first run, although it costs roughly 80 times compared to database tables, this is still about 3 times faster than table UDFs. Looking at the absolute numbers for SF 4, queries using table UDFs took 3318 seconds compared to 1114 seconds for virtual tables. Given that both table UDFs and virtual tables are doing conversions eagerly on all the output attributes, the difference in their execution cost can be attributed to the difference in execution plans. With table UDFs, the optimizer has no data characteristics to work with and ends up with much worse execution plans than with virtual tables. Looking at the second run of the queries, we can see

that as virtual tables have already performed the conversions during the first run, their costs drop significantly, taking only 18 seconds against 2700 seconds of table UDFs, 150 times faster.

Further, enabling lazy conversion on virtual tables reduces the cost of the first run to 836 seconds, which is about 4 times faster than table UDFs. This also implies a savings of 278 seconds (the difference with eager conversion in virtual tables) from not converting attributes that are not used in the query.

Only One Data Set as a Python Object

To analyze the performance impact for less complex scenarios, we re-run the previous test setup with only one of the data sets as a Python Object while the others are all standard database tables[2]. To also account for the impact of the size of the data sets on performance, we select the data set for either `nation` (a very small table) or `lineitem` (a very large table) to be represented as a Python object. We confine ourselves to TPC-H queries 5, 7, 8, 9, and 10, which have these two tables in common.

First, we re-run the test case by keeping only `nation`, which has 25 records, as a Python object. Figure 6.4 shows the execution costs in milliseconds of individual queries for SF 4. Other scale factors show similar trends. Other than query 7, there is barely any noticeable difference in performances between the various test candidates. For query 7, the cost of execution using table UDF is 2.3 times (627 ms) compared to all others (under 270 ms). Although `nation` is a small data set, incurring very little conversion cost, it seems that the lack of access to data characteristics can still cause the optimizer to generate bad execution plans.

The results for keeping the `lineitem` data set, which has SF $\times$ 6 million records as a Python Object, are shown in Figure 6.5. In general, we can see that queries are faster against the database tables, followed by the virtual table implementations, and table UDF being the worst performer. Their overall trend is very similar to what we previously observed. Unsurprisingly, with larger data sets, there is more conversion cost involved. Additionally, for table UDFs, the impact on the cost of query execution due to the optimizer selecting an inefficient plan is higher for all the queries. Virtual tables with eager conversion at 133 milliseconds are 4.8 times faster than table UDFs that took 648 milliseconds, indicating that the choice of a sub-optimal plan with table UDFs leads to a penalty of over 500 milliseconds.

Taking this one step further, with lazy evaluation, it takes only 19 milliseconds for virtual tables (35 times faster), indicating that it saved 114 milliseconds by avoiding unnecessary data conversions. We can also observe that for queries 5, 8 and 9, the cost of virtual table's lazy evaluation

[2]Again, for `db-tbl` all data is in database tables.

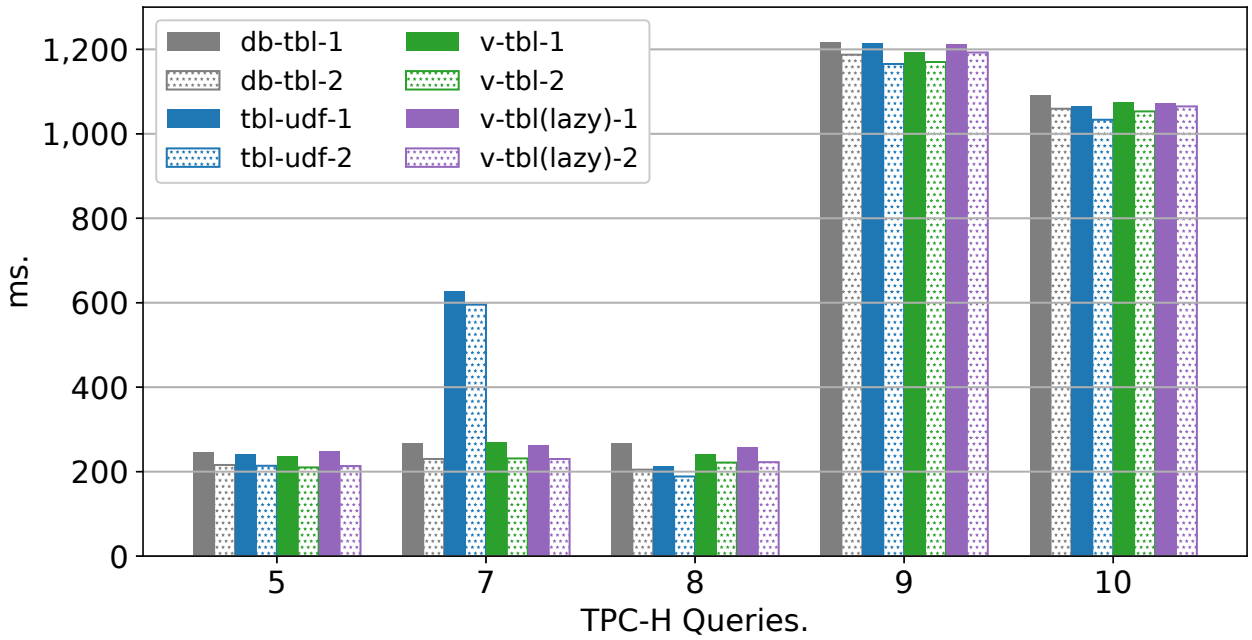


Figure 6.4: TPC-H, SF4 (nation as a Python object).

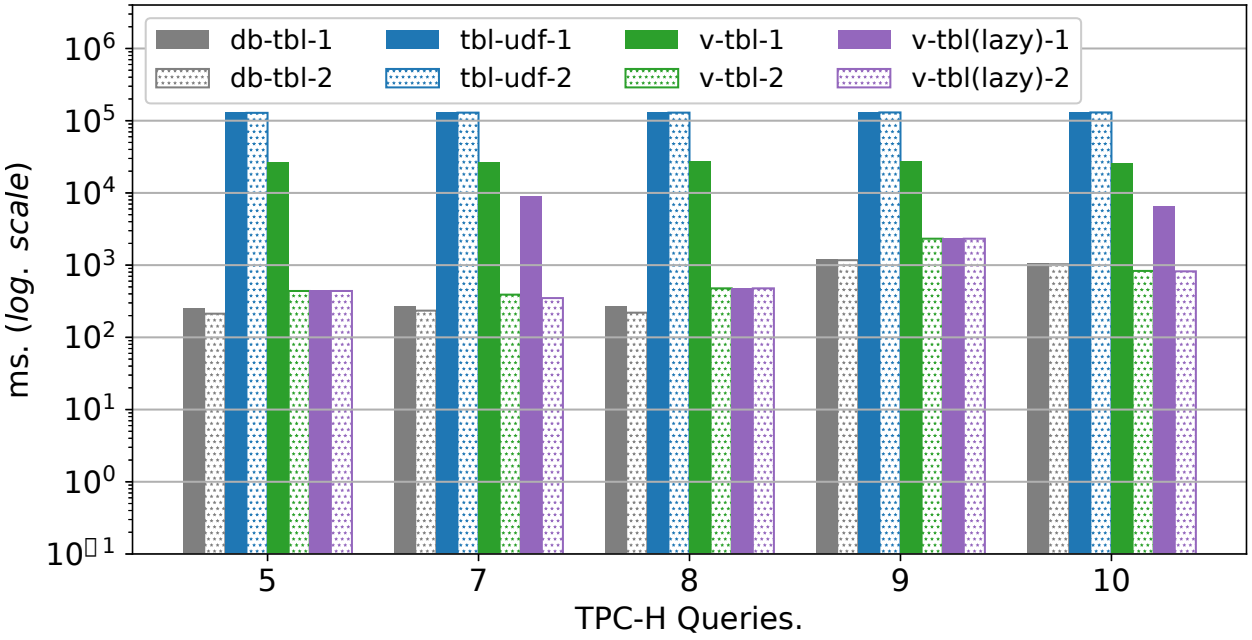


Figure 6.5: TPC-H, SF4 (lineitem as a Python object).

implementation is already on par with that of querying database tables directly. This is because all the `lineitem` columns referenced in these queries have data types that are compatible between the HLL and the database, eliminating any need for conversions. Further, since the rest of the columns are not converted in the lazy evaluation implementation, overall there is no conversion cost for running these queries, highlighting the importance of this approach.

To summarize, we can see that for large data sets table UDFs are severely impacted due to data conversions and sub-optimal execution plans. Even though conversion costs are low for smaller data sets, there are still cases where a poor execution plan increases the cost of query execution severalfold. In comparison, virtual tables, with its lazy evaluation and optimizer-friendly interface can facilitate much faster query execution on HLL objects.

6.4.3 Virtual table Creation

As we saw in the previous test case, table UDFs incur a lot of cost due to bad execution plans. As discussed in Section 6.2.5, a workaround is to create a temporary table in the database with the relevant data and then use this temporary table in the SQL instead of the table UDF. Full access to the temporary database table allows the optimizer to produce an efficient execution plan.

One way to accomplish this is to first execute the table UDF and then materialize its output into a temporary table, such as the example we discussed in Listing 6.5. This represents a bulk transfer of the data to the RDBMS. An alternative approach that appears to be more elegant to code as it can be implemented within a single stored procedure is to traverse the HLL object and insert its data into a temporary table in the database using the loopback query API, record by record, such as the example in Listing 6.6. We implement both the approaches and compare their costs against the cost of registering a virtual table. For a fair comparison with UDFs, we disable the lazy evaluation for virtual tables so that they are also eagerly materialized.

First, we analyze the performance when the data types require conversion. For this, we use seven synthetic data sets ranging from 1 to 1 million records, each with a hundred attributes that are of 10 character length. Strings are data types that require conversions to be performed between MonetDB and the embedded HLL interpreter (Python). We then execute scripts to:

(i) Materialize a temporary database table using the output of a table UDF: `tmp-tbl-udf`.

(ii) From a stored procedure, iterate over the HLL object and insert it into a temporary table, record by record using the loopback query API: `sp-loopback`.

(iii) From a stored procedure, register an HLL object as a virtual table that is eagerly materialized: `sp-v-tbl`.

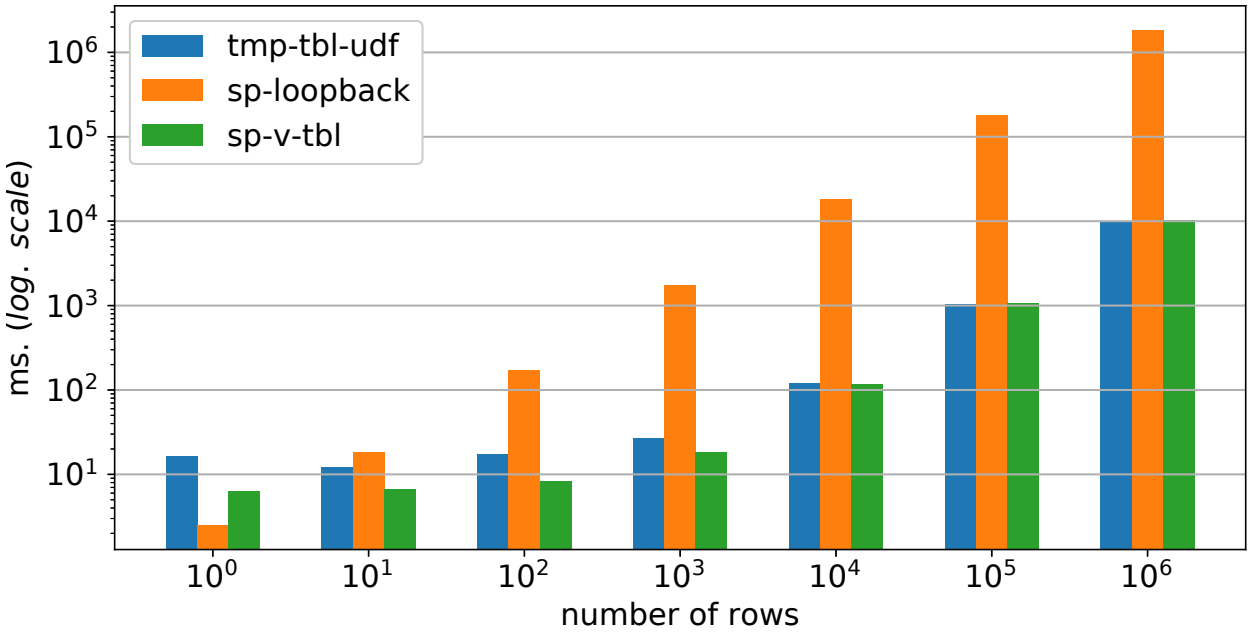


Figure 6.6: Temporary table creation, character columns.

Looking at the performance results in Figure 6.6, where we have the cost of constructing a temporary table structure in milliseconds along the y-axis in logarithmic scale and the number of records in the data set along the x-axis, we can clearly see that, from the cost perspective, virtual tables have no advantage over using a conventional table UDF to create temporary tables when data conversion is involved, and that their performance is on par. On the other hand, the naive approach of inserting data into a temporary table using loopback queries has a huge performance penalty unless we are populating just a single record. But then, all the systems cost just a few milliseconds for this to be of any consequence. At 10 records, the cost of using loopback queries is 12 milliseconds, compared to 6.5 milliseconds for the virtual table. The disparity increases with the number of records, with loopback queries taking 30 minutes to load 1 million records compared to 10 seconds for the virtual table, giving the latter a 180 times faster response.

Next, we change all data types to float so that virtual tables do not need to perform any data conversion. The performance for this test is shown in Figure 6.7. The creation and registration of the virtual tables take 6.5 milliseconds independently of the number of records showing the power of virtual tables when no conversion is required. Inserting records one-by-one has similar costs as with strings (see Figure 6.6) as not the conversion but the individual insert operations are the main cost contributing factor. As table UDFs need to copy data into RDBMS buffers, its cost increases in proportion to the number of records, reaching 590 milliseconds for 1 million records.

Therefore, we can see that while table UDFs are able to match the “data transfer” cost asso-

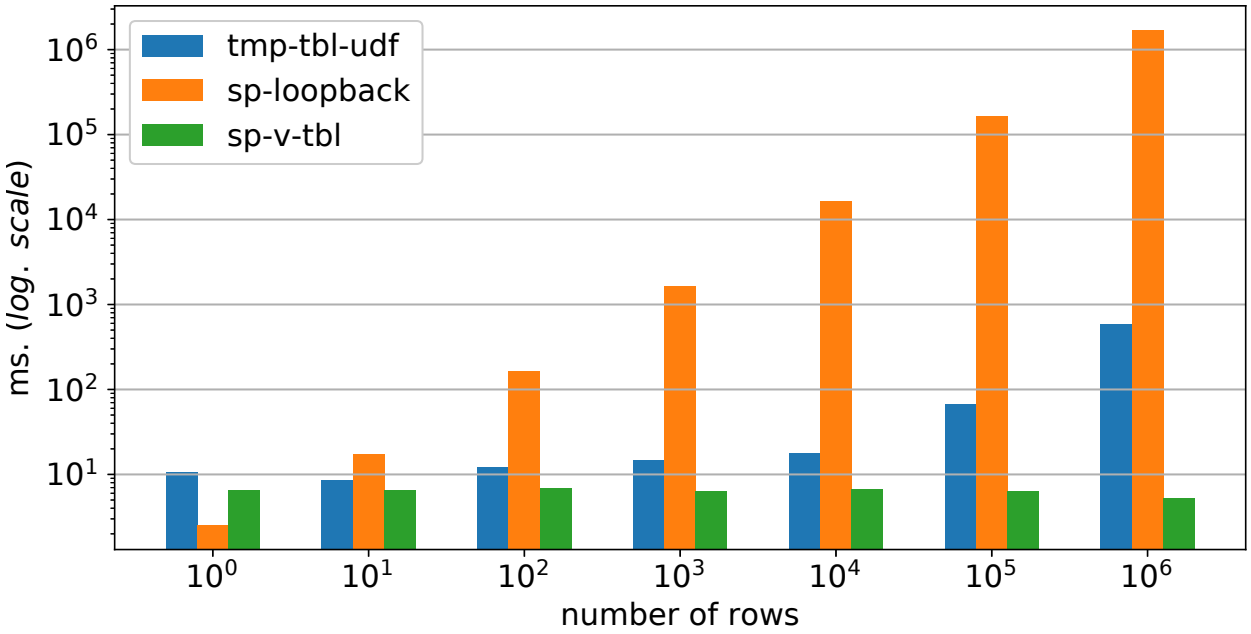


Figure 6.7: Temporary table creation, float columns.

ciated with virtual tables when data conversions are involved, in the absence of such conversions, the zero-copy approach of virtual tables is significantly advantageous. Moreover, the iterative approach of building temporary database tables from within an HLL code, while resulting in a compact code, is prohibitively expensive for any reasonable sizes of data sets and not practical.

While the coefficients of variation for the observed data points start off high (around 100%) for smaller number of rows, they taper off after 1000 rows to below 10%. The only exception to this is the virtual tables for floating point data which has a c_v of up to 66% even with a larger number of rows. Considering the execution costs, we can see that for smaller number of records it is often possible to finish the task without being impacted by any transient system activity, while other times it might be influenced by them, resulting in higher c_v values. As the number of rows (and therefore, the cost) involved increases, all measurements are evenly impacted by transient system activity resulting in consistent numbers. As virtual tables have a constant, low execution cost for floating point data that is independent of the number of rows, their c_v remains high. These variations however, do not influence the relative performance conclusions drawn in the test cases.

6.4.4 Data Science Workflows Using AIDA

To understand the performance implications of virtual tables on the end-to-end exploratory data science workflow that we tested in Section 5.5 using the Bixi bicycle trip data set, we execute

it with the modified version of MonetDB's database adapter that uses virtual tables instead of table UDFs.

In general, an AIDA workflow's computational cost is spread across the execution of linear algebra, HLL code, relational operations, etc. The relational operations themselves can be classified as SQL queries that contain only regular database tables and queries that also contain HLL objects. Of the various cost components, we are specifically interested only in the cost of these SQL queries containing HLL objects, as these are the only queries that will be impacted by using virtual tables vs table UDFs.

Using table UDFs, the cost for these queries comes to a total of 3475 milliseconds, whereas using virtual tables, the cost comes down to 2747 milliseconds, a 21% reduction. When we analyze the actual SQL queries involved in this, we notice that there are only three which have a join, with a maximum of three tables being joined. Therefore these queries are not as complex as the TPC-H queries where we saw a large magnitude of difference in the performance between table UDFs and virtual tables. Further, we do not observe any significant savings from lazy-conversion. This is because most of the attributes in the HLL objects do become part of some SQL queries in the lifetime of the workflow, resulting in eventually all of them undergoing data type conversion.

6.4.5 Label Encoding

Virtual tables are an elegant way to expose table-like Python objects to the RDBMS in order to perform database operations on it. This opens up many programming possibilities that were hitherto difficult to implement. In this section, we explore a possible SQL implementation of *label-encoding*, a transformation process common in data science and machine learning approaches to replace data attributes that are string literals with artificially generated numeric values. Such data transformations are required in many cases when the model to be trained can work only with numeric data. We implement a Python library for labeling transformation with an API modeled after the scikit-learn [PVG+11] Python library, which is the most popular Python package for labeling.

Our library accepts either a Python object such as the dictionary-columnar representation of a TabularData object, a pandas DataFrame or a database table. The library can return the transformation as a Python object or store it in the database itself as a table. Thus, this approach provides the capability for the user to store encodings and the transformations for a later use as a database table, something which is not trivial to accomplish with scikit-learn.

The internal implementation consists of creating and executing a set of SQL statements that are based on the input data schema. Should the input data be a database table, the SQL queries are

executed directly on the tables. Python input objects are exposed as virtual tables to the RDBMS for executing the SQL queries.

To test the performance of this approach, we used the North Carolina voter classification data set from [RHMM18] and tested the library for varying sets of records in the data set considering the following scenarios.

(i) For a base comparison, we perform the label encoding using scikit-learn (`sklearn`) on data that is stored in a Python object, and the result returned is also a Python object.

(ii) Data is stored in a database table and is to be encoded and returned to the calling stored procedure as a Python object (`dbtbl2py`).

(iii) Data is stored as a Python object and is to be encoded and returned to the calling stored procedure as a Python object (`py2py`).

Looking at the performance comparisons in Figure 6.8, we can see that our SQL-based implementation outperforms scikit-learn across all the data set sizes. The SQL-based implementation is fastest when it works directly on a database table. This is because when it needs to work on a Python object, it needs to first expose it to the RDBMS as a virtual table, and as character columns undergo conversion, there is an extra cost. Their performance trends are similar across all the data sizes. If we take a close look at costs for 4 million records, we can see that scikit-learn took 28 seconds, compared to the SQL implementation which took 1.55 seconds when working on a database table (18 times faster) and 3.47 seconds when working on a Python object (8 times faster).

That is, our database implementation of label encoding is an order of magnitude faster than the scikit implementation, showing the promise of even pushing advanced data science transformations, that seem to be more naturally implemented in HLL, into the database. The flexibility of doing so is provided through the virtual table concept.

6.4.6 Programmability: Virtual tables in UDFs

Finally, one of the benefits of using virtual tables is that it makes it possible to write compact code. Unlike the conventional approach which requires multiple table UDFs along with an additional driver script or a stored procedure, a single stored procedure can be employed to perform the entire task without loss of performance. Although the loopback query API can be used to transfer data into the RDBMS to write the logic as a single stored procedure, as we demonstrated earlier, the cost for this is extremely high.

To analyze this facet of virtual tables, we translated the voter classification workflow from [RHMM18] to make use of virtual tables. At a high-level, this workflow analyzes the voter in-

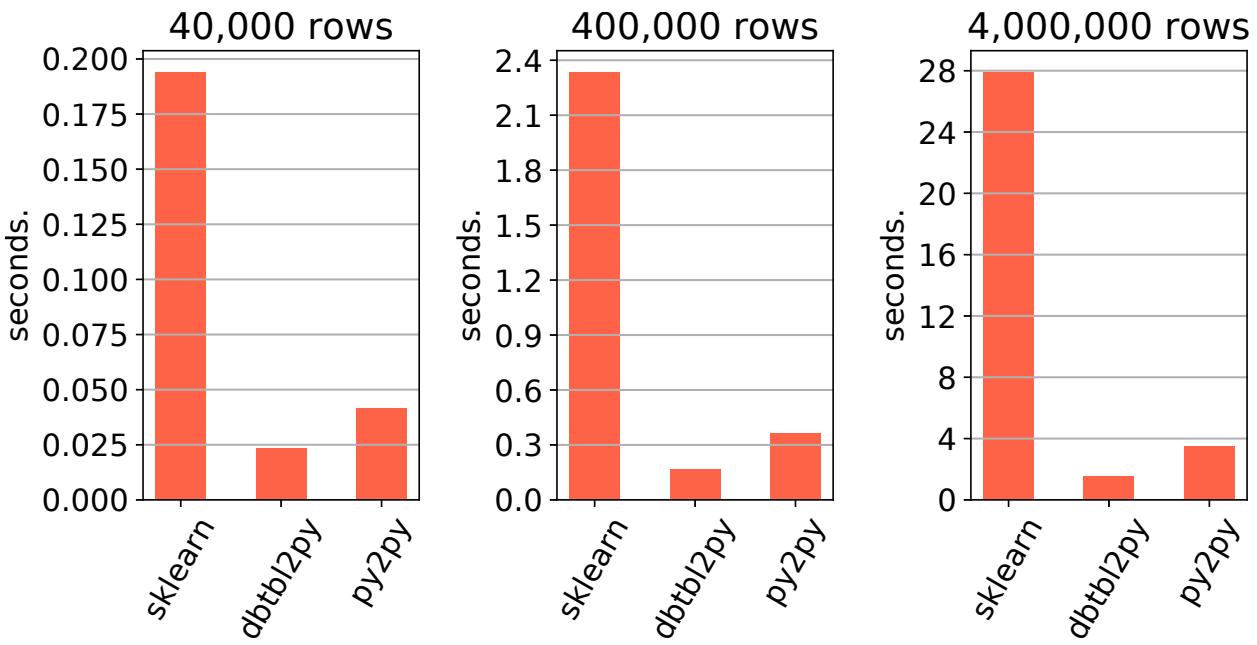


Figure 6.8: Label encoding via virtual tables.

formation and their localities to determine which party they would vote for. Along with relational processing, the workflow also uses scikit-learn to solve the classification (party voted for) problem.

The first box in Figure 6.9 shows the number of source code artifacts required to write the workflow. The table UDF based solution requires 4 table UDFs and a driver script with SQL statements to glue their logic together. In contrast, all other approaches can be written using a single stored procedure. The further boxes in the figure show the performance for the various number of rows. We can see that the performance of table UDFs and virtual tables are on par, whereas stored procedures using loopback queries, `sp-loopback`, are about 100 times costlier. As this workflow is already hand-optimized to output only the required attributes from the table UDFs and most attributes are strings, virtual tables have no advantage in terms of performance. Additionally, we also put to test the label encoding library that we developed and whose test results were covered in section 6.4.5. This library is used to replace the label encoding process performed in scikit-learn, in the original solution. We can see that this approach, `sp-v-tbl+enc` is roughly 3 times faster compared to the table UDF based approach and is 300 times better than the `sp-loopback` based approach.

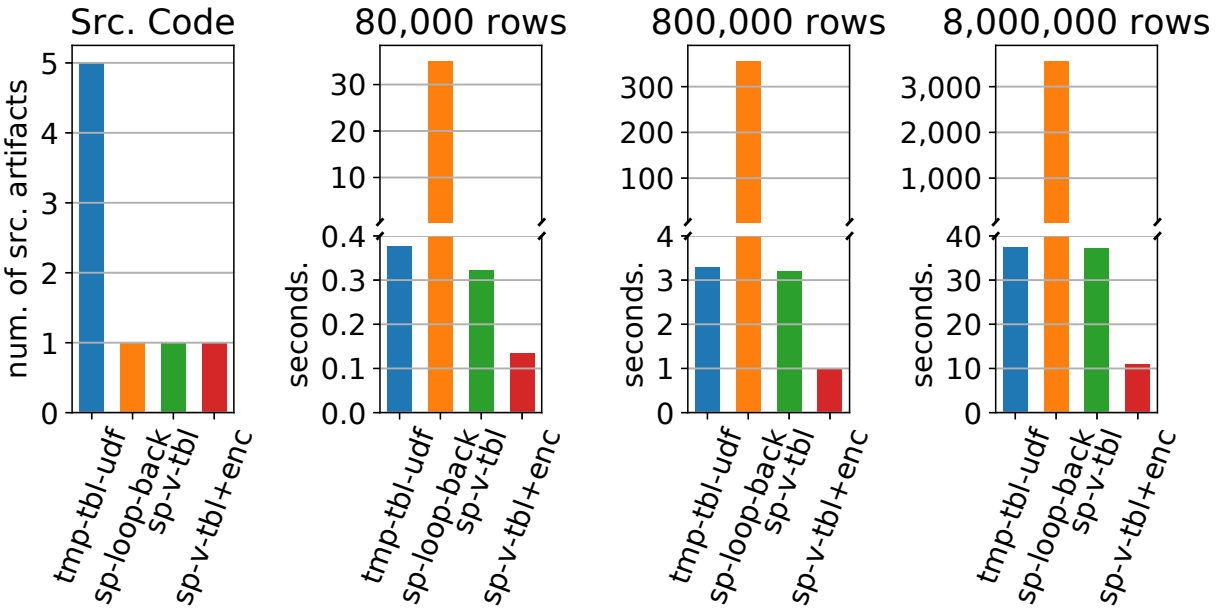


Figure 6.9: Voter classification UDF workflows.

6.4.7 Evaluation Summary

We performed an extensive evaluation of our implementation across various data sets and use cases. Our virtual table implementation is able to speed up complex queries such as TPC-H, by as much as 4 times. These savings are attributed to both better plans generated by the optimizer using the data characteristics provided by the virtual table, as well as the cost savings from not having to convert unused attributes because of lazy evaluation. When integrated into AIDA, we observed a 21% cost reduction for operations involving virtual tables compared to using table UDFs for our exploratory data science workflow. As virtual tables make the movement of large data sets to RDBMS easy, we implemented a SQL version of the label encoding process used in data science and found it to perform 8 times faster than scikit-learn, a popular Python library that is usually used for this purpose. Further, as a bonus feature, complex hand-optimized table UDF based workflows can be written in a compact manner using a single UDF/stored procedure and virtual tables, reducing the number of source code artifacts.

6.5 Related Work

While the IR based approaches that we discussed in Section 2.4.5, such as Weld [PTS+17], TUPLEWARE [CGD+15], Froid [RPE+17], and HorseIR [CDC+18], tries to address the issue of UDFs by providing a unified execution platform for both database and HLL functionality, they require a major overhaul of RDBMS internals and is therefore not easily adaptable for traditional RDBMS implementations [Var18]. Besides, as such approaches require the entire workflow to be defined to generate an optimized IR for execution, they cannot be easily employed for exploratory frameworks like AIDA where the complete program path is not known in advance. As such, we believe that our proposed approach of virtual tables is an attractive alternative to such a major re-design and at the same time offers many of the advantages of a tight integration that these IR based systems promise.

Frameworks such as LINQ that integrate query capabilities into programming languages are client-centric, i.e., data often gets moved to the client side from the database [MEW08]. Virtual tables, on the other hand, provide improvements to database-centric programming frameworks such as AIDA, as well as conventional UDFs and stored procedures that retain the data in the database, without sacrificing programmability.

Apache Arrow uses a shared memory concept to facilitate different applications that conform to its API to access the shared data [Din16]. This, however, limits any operations on such data sets to what is supported by the module, besides the fact that applications and libraries will have to be re-written to interface with it. The virtual table concept is more flexible as it only deals with smart data interfacing between the systems and does not impose any restrictions on the computational possibilities on the data.

PostgreSQL has a *foreign data adapter* concept which allows users to expose external data sets (such as a CSV file or a table from another database) to the RDBMS as *foreign tables* [OH17]. Unlike table UDFs, foreign-tables expose a more versatile API to the user. However, the implementation requires the user to write numerous data processing APIs if they desire to optimize the query, including support for cost estimation and plan generation. The implementation copies over data from the external system to the database space, necessitating a way to reduce the data movement. Such implementations are cumbersome to develop and maintain, as any future optimizations in the RDBMS could likely require changes to be made to the APIs. Since we are concerned with the processing of data sets already in the RDBMS address space, but stored in an HLL object, our virtual table approach is able to efficiently piggy-back the RDBMS query processing system, which treats these data sets in a way similar to real database tables, reducing maintenance effort significantly.

Finally, those familiar with Spark SQL [AXL+15] might notice a syntactic similarity with how virtual tables APIs are used. However, behind the scenes, they serve two fundamentally different purposes. Spark SQL functions as *syntactic sugar* for users to express relational operations on Spark data sets instead of using the conventional Spark API. I.e., they both work on the same underlying data structures. The ideology behind virtual tables, on the other hand, is to bridge data sets from one system (embedded HLL) to another (RDBMS). Further, systems like Spark work external to the database, whereas the embedded HLL and the RDBMS that are linked by the virtual table concept reside within the same runtime environment and memory space.

7

Final Conclusions & Future Work

In this chapter, we will briefly recap the reasons that motivated the research presented in this thesis, present our conclusions, and lay out the road map for future work.

7.1 Final Conclusions

In this thesis, we reviewed the evolution of data science frameworks. We saw how the lack of support in conventional RDBMS implementations for the data processing primitives and paradigms required by data science projects has led to the data science community developing database-external frameworks that are architecturally suboptimal. We also discussed how RDBMS vendors prioritize functional support for traditional database applications, and as such, are hesitant to make any hasty changes involving major redesign of their internal architecture to efficiently incorporate the new computational primitives required by data scientists.

We then discussed the most common, "minimal engineering" approach employed by contemporary RDBMS vendors to incorporate some support for the computational primitives required by data science applications. We described how this approach, implemented by embedding an HLL interpreter in the RDBMS, can be used to extend RDBMS functionality. We discussed that this

approach, mainly supported through UDFs is not a convenient paradigm for the exploratory work involved in data science projects. Further, we also highlighted the disruptive nature of UDF constructs when it comes to query optimizations, owing to their back-box nature, affecting the query performance adversely.

As such, we make two important contributions in this thesis to address both (i) the lack of user-friendliness of the programming paradigms provided for in-database data science analysis, (ii) and the performance implications introduced by incorporating HLL constructs such as UDFs into database query processing.

Our first contribution, AIDA, provides a unified data abstraction that supports both relational and linear algebra operations using the familiar syntax and semantics of popular end-user based Python ORM and linear algebra implementations. AIDA's programming API is client-based, allowing data scientists to use their familiar programming environments such as Jupyter Notebook to perform their analysis. However, AIDA's client API transparently moves the execution into the database system, eliminating the data transfer costs associated with using an external data science system. AIDA's server component resides in the embedded HLL interpreter of the RDBMS, facilitating *near-data* execution of computations. This also allows AIDA to leverage the underlying SQL engine of the RDBMS for executing relational operations on data sets. AIDA internally leverages an existing HLL statistical library, NumPy to perform numeric computations. Data "handover" between the RDBMS and the statistical library is done automatically by AIDA, without any user involvement. Further, it can often accomplish this efficiently, without having to perform any data copying or reformatting between the RDBMS and the statistical library.

AIDA allows users to write custom code to be executed inside the framework, but without the programming limitations of contemporary HLL database constructs such as UDFs. Also, unlike such database constructs, AIDA provides data visualization capabilities, a critical tool for exploratory analysis. Finally, AIDA also defines an in-database data science framework architecture that allows other RDBMS implementations to port AIDA by minimally implementing AIDA's database adapter interface.

Our performance evaluations show that AIDA has a significant performance advantage compared to approaches that transfer data out of the RDBMS. We are also able to demonstrate through qualitative analysis that AIDA provides better programmability to the user compared to contemporary UDF based approaches, without incurring any significant framework overhead in performance. Further, the qualitative analysis shows that AIDA's programmability is on par with the popular database-external systems presently popular with the data science community. Therefore, AIDA addresses the usability aspect of in-database data science approaches while maintaining the

performance aspect of *near-data* solutions. We believe that AIDA is a step in the right direction to facilitate user-friendly computational interfaces for data scientists to interact with an RDBMS.

Our second contribution, virtual tables, addresses the performance aspect of incorporating embedded HLL constructs such as UDFs, in database query processing. The virtual table concept augments the current UDF constructs and can be used by any embedded HLL program to expose HLL table-like data sets to the RDBMS for relational operations. Virtual tables facilitate the database optimizer to have insight into the characteristics of the HLL objects, allowing the generation of efficient execution plans. They also provide a more efficient data handover between the HLL and the RDBMS, delaying data conversions of any associated HLL attributes till the moment it is required by the RDBMS, thus saving overhead from unnecessary data conversions. The programming flexibility and performance improvement introduced by virtual tables allow users to develop more compact code, as well as explore new and efficient solutions to some conventional data science problems. Our evaluation of the virtual table implementation over a variety of data sets and use cases demonstrate that they have a significant performance advantage over the alternative of using conventional UDF constructs, while also providing better programmability.

7.2 Future Work

7.2.1 Re-visiting Workload Management in RDBMS

Modern RDBMS implementations already have to handle a variety of traditional database applications, each with their own query characteristics and organizational priorities. As such, the ability to manage a variety of query workloads and to be able to provide differential treatment to their resource requirements are described as very desirable traits for high-end RDBMS implementations [ZMPC18]. This workload management landscape will be significantly impacted when in-database advanced analytics is widely adopted.

Traditional database application analytics queries are predominantly I/O bound, as a significant amount of query processing is spent on accessing data sets from the disk. In comparison, their CPU usage, while not insignificant, is generally less prominent, and is spent on computations performed in memory (such as joins, aggregations, etc.). While a conventional data science workflow will exhibit similar characteristics during its initial, *feature engineering* phase, where the transformations are predominantly of relational nature, once it starts performing *model training*, it can become extremely CPU bound. Thus a given data science task usually starts out being I/O bound and eventually switches over to being heavily CPU bound. At this stage, it might compete

and interfere with other workloads requiring CPU. Hence, to ensure a reasonable balance in an RDBMS implementation that supports data science applications along with traditional database workloads, the database workload manager may have to prioritize the CPU for traditional database workloads.

This also poses a novel challenge of defining, analyzing and comparing the behavioural characteristics of such workload management approaches. While conventional RDBMS implementations have relied on the benchmarks developed by the Transaction Processing Performance Council (TPC) for performance analysis, and NoSQL[1] systems similarly rely on the Yahoo! Cloud Serving Benchmark (YCSB) [CST+10], they do not incorporate the workload characteristics of CPU intensive machine learning tasks. On the other hand, though there have been some independent initiatives for benchmarks targeting machine learning primitives [WMC+13, RDVP16], they do not address the much wider spectrum of traditional database workloads. As such, there is a need for more detailed research in this field, and possibly the development of mixed workload benchmarks that address a much broader suite of database applications.

Even though this is still an open problem, organizations can still leverage some of the approaches that they have traditionally employed to protect their mission-critical database workloads from runaway exploratory queries. Many organizations maintain a passive replica of their primary production database to support high-availability in the event of a failure of the primary database. As these replicas are usually idle and do not part-take in active production tasks, organizations allow exploratory users to utilize them as *sandbox* environments to run custom analysis queries. This allows for efficient utilization of existing hardware infrastructure, while also protecting mission-critical production tasks. Therefore, such infrastructure could be utilized for the kind of data science work that AIDA is intended for, without having grave concerns about its impact on other, mission-critical database workloads.

7.2.2 Memory Management for TabularData Objects

Presently, any materialized TabularData objects in AIDA maintain their internal data representation in the form of Python data structures (specifically NumPy arrays). As these data structures are memory-resident, this limits the number of materialized TabularData objects that can be maintained by AIDA at any given instance. Since AIDA already facilitates explicitly persisting TabularData objects as regular database tables, which themselves are easily read into TabularData objects, one way to tackle this issue is to have a *memory manager* component in AIDA that can temporar-

[1] In the modern context, the term NoSQL is used to refer to non-relational database implementations that were developed to address the needs of Big data applications [HHLD11].

ily and transparently persist TabularData objects to relieve the pressure on memory usage, loading them back when required.

One approach would be to employ a least recently used (LRU) algorithm to decide which TabularData objects must be persisted. A threshold setting could be added to AIDA that dictates the total amount of memory that materialized TabularData objects can occupy. The memory manager can actively monitor the memory usage by materialized TabularData objects, persisting those ones selected by the algorithm. Such a persisted TabularData object can be treated similar to a TabularData object that is pointing a regular database table, loading the data back when requested by an operation that requires it to be materialized. The only difference that they have with TabularData objects that points to regular database tables is that, such temporarily persisted database tables must be automatically removed when their corresponding TabularData objects are removed.

While in principle this approach should be sufficient for many scenarios, we would also want to take into account the fact that some TabularData objects may have both the dictionary-columnar and matrix data formats materialized. In such cases, it might make sense for the memory manager to just discard the dictionary-columnar data representation of such TabularData objects, as the dictionary-columnar data representation can always be provided as a "logical view" of the matrix format.

Further, we will also have to consider the special scenarios where the underlying data structures of TabularData's internal data representation are shared between multiple TabularData objects. In such cases persisting just one of the TabularData objects will not release any memory, as the underlying data structure is still being actively referenced by another TabularData object. Therefore, a sophisticated algorithm will have to take into account this aspect as well in deciding which TabularData objects must be persisted. As such, we will have to move all of the relevant TabularData objects into a "persisted status" together, tied to a single underlying database table.

7.2.3 GPU Support in AIDA

Although AIDA is primarily designed for the interactive exploration phase of data analysis, it contains mechanisms such as the *remote execution* operator to ship the execution of entire functions to the database server. Thus, computationally intensive code, such as the iterations to be performed during the *model training* of certain learning algorithms, can be fully executed at the server, without any continuous interaction with the client. Interestingly, several of these computationally intensive tasks pertaining to machine learning often perform far better on GPUs compared to CPUs. However, traditional RDBMS implementations are optimized for CPUs [BHS+14]. In general, the performance advantage of using GPUs for a particular operation must be weighed in against

the overhead associated with data transfer to GPU RAM, which is presently bottlenecked by the PCIe Bus bandwidth that dictates the data transfer speeds between main memory and GPU RAM [BHS+14].

Nevertheless, we would like to explore extending AIDA to provide a new operator akin to the *remote execution* operator, that can automatically ship code and associated data sets to GPUs to perform such computationally intensive tasks associated with machine learning primitives. Also, such support for GPUs can ameliorate the impact of data science workloads on traditional database application queries by moving the compute-intensive aspect of data science workflows into GPUs. The decision of whether a function should be executed within the CPU or the GPU can be left to the user or done by the system. It will have to consider the trade-off between an increase in execution speed with the potential impact of data copying into GPU memory.

7.2.4 Extending AIDA to Support Trained Models as First-class Citizens

Although AIDA's extensions such as the *remote execution* operator can be used with existing Python libraries such as Scikit-learn to build various models used in machine learning, AIDA presently has no explicit mechanism to persist them to be used at a later point, or to be used by sessions other than the one that constructed it.

We believe that an explicit framework feature to persist trained models and to access them at a later point is an important and useful addition to AIDA, as often the uses of a trained model are intended to outlive the scope of the workflow that built it. Although works such as ModelDB [VSL+16] is capable of capturing and storing the transformations used to construct a data science workflow and the associated models built by it, they exist external to the RDBMS that is the source of the data. On the other hand, rudimentary approaches followed by UDF based implementations such as [RHMM18] result in the UDFs having to load the model from the database every time it needs to be used, incurring an unnecessary overhead.

AIDA can be extended to treat these trained models as first-class citizens. The concept of a DMRO can be extended to include such trained models, treating them akin to other DMROs in AIDA, that exist independent of any user sessions. This will allow data scientists to "publish" a trained model, that is then accessible across other AIDA user sessions. Further, rudimentary persistence techniques such as those employed in [RHMM18] can be internally used by AIDA to store a new model into the database and to retrieve it later. However, unlike the limitations of the UDF based approach described in [RHMM18], as AIDA keeps the model available in the database memory in the form of a DMRO, they need to be loaded from the database only once and will

be immediately accessible for any further requests without any additional overhead, facilitating real-time *scoring*[2] of new data.

7.2.5 A Statistical Library Adapter for AIDA

Our current implementation of AIDA uses NumPy as its statistical library. This design decision was in part due to the fact that we can leverage optimizations such as *zero-copy* of MonetDB that works with NumPy libraries. As NumPy data structures also form the internal data representation of AIDA's TabularData objects, currently AIDA's data transformation modules are tightly integrated with NumPy. Therefore, unlike how database adapters can be swapped out to use AIDA with another RDBMS implementation, the same is not feasible when it comes to statistical libraries in AIDA's current architecture. Therefore, we will have to spawn a new architectural component in AIDA, a *statistical library adapter*, akin to the database adapter, to make the rest of the components of AIDA agnostic to the statistical library that is being used.

Further, there have emerged a variety of other Pythonic implementations for statistical and data science libraries, such as TensorFlow [ABC+16], PyTorch [Ket17], and dask [Roc15], most of them with their own underlying data structures and API syntax that are not compatible with NumPy. While both TensorFlow and PyTorch have support for GPUs based computations, they have different programming models. TensorFlow requires *data-flow graphs* [Wol12] to be built in order to execute any transformations[3]. As AIDA's programming API follows the more user-friendly, imperative programming approach[4] used by popular Python libraries such as NumPy and pandas, the statistical library adapter might have to internally build miniature data-flow graphs to execute individual statistical transformations in order to maintain AIDA's current programming APIs. We will have to analyze how significant the performance overhead of this approach would be. On the other hand, PyTorch follows an imperative programming model similar to that of AIDA. Although this results in slightly less performance compared to approaches such as TensorFlow, they are much more user-friendly to work with [PGM+19]. Therefore, implementing a statistical library adapter for PyTorch should be a lot more easier, with possibly much smaller performance overhead. While dask also employs a data-flow graph based programming model, their programming APIs bear a lot more syntactic similarity compared to NumPy array APIs, making them also potentially an easier candidate to port.

[2]Scoring is a machine learning term used to denote the application of a model to new data, to perform predictions over them.

[3]Data-flow graphs in a nut-shell are a sequence of transformations (nodes) connected in the form of a directed-graph that allow data sets to "flow" through them, applying transformations in doing so.

[4]This is basically the statement-at-a-time approach that was discussed in Section 2.3.1

Finally, AIDA's current architecture puts the onus of formatting the relational data queried from the RDBMS into the statistical library format (NumPy) on the database adapter implementations. In a platform that supports several RDBMS implementations and several statistical libraries, this is not a modular approach that can be maintained easily. Therefore, the data format conversions might have to be moved to a new architectural component in AIDA, that is specifically tasked with it, such that the database adapters and statistical library adapters can function agnostic to the data formatting needs of their counterparts.

7.2.6 AIDA For Row-based RDBMS

As discussed in Section 3.4.3, the database adapter separates the database architecture dependent implementation of AIDA from the rest of its components. It is also responsible for building data representations that are compatible with the statistical package that AIDA uses. While MonetDB's zero-copy optimization facilitates some advantages in this aspect, implementing a database adapter efficiently in the context of a row-based RDBMS architecture [AMH08], such as PostgreSQL, can pose some performance challenges.

Our initial attempt to build a functional prototype of a database adapter for PostgreSQL has shown promise. However, since this prototype is a pure Python implementation, the overhead of converting the row-based result set returned by PostgreSQL into the vector format required by the statistical library that AIDA uses, inside the database adapter is not negligible. This is because interpreted languages such as Python are inefficient for computationally intensive tasks. As MonetDB produces data sets that are compatible with AIDA's statistical library, NumPy, this is not an issue with MonetDB's database adapter. Conventionally, such performance concerns are addressed by building libraries using computationally efficient languages such as C, which can be then invoked through Python wrappers. In fact, many statistical packages such as NumPy rely on this approach to provide performance. Our current idea is to employ a similar approach to implement a data format converter as an efficient C library, which can transform the row-based data sets produced by PostgreSQL into a vector format. However, as data conversions are a costly affair, we believe there is scope for implementing this in a "lazy fashion", similar to the approach we used for data conversion in virtual tables. Such an approach would defer the data transformations until the particular column(s) are requested in a transformation from the statistical package.

7.2.7 Virtual Tables and Row-based Systems

We would like to explore the possibility of implementing our virtual table concept for a row-based RDBMS, such as PostgreSQL. As discussed in Section 6.5, PostgreSQL has a *foreign data wrapper* concept that allows the RDBMS to execute SQL queries over non-database objects [OH17]. We believe that extending this functionality of PostgreSQL would allow us to develop a less intrusive implementation for virtual tables for PostgreSQL. Further, unlike columnar-RDBMS like MonetDB that expect each column's data to be provided as an independent vector, row-based systems like PostgreSQL require the data to be provided in a record format, thus requiring conversions. It needs to be analyzed as to how this can be done efficiently, given that each query may request a different set of columns from a data set. In this aspect, we will have to analyze the cost benefits of performing all the conversion in one-shot and letting the database doing the required column projection versus having the foreign data adapter module produce different record formats depending on the columns required by the query. Concepts like caching previously constructed results will have to be investigated to optimize this kind of approach.

7.2.8 Virtual Tables - Further Optimizations

A fundamental characteristic of virtual tables is that they perform any necessary data conversions on HLL data sets to the database format, for the RDBMS to execute queries over them. While the *lazy conversion* approach provides some optimization by delaying the data conversions of HLL attributes till (and only if) they are actually used, this can be improved upon. There are various scenarios where the database does not have to know the actual values of some of these HLL columns. For example, consider a simple query that performs *selection* based on a certain set of columns on an HLL data set. From the database's perspective, it needs to check the values of those columns involved in the *selection* predicate (i.e., the `WHERE` clause), and as such, would need to convert these columns into the database format for it to perform the predicate evaluation. The rest of the columns can be returned as-is if the result set is being sent back to the embedded HLL code. However, such implementations will require more intrusive changes to the database query execution engine than we have presently performed for our current implementation of virtual tables.

Contributions & List of Publications

Published

[DDMK18] AIDA-Abstraction for Advanced In-database Analytics.
Joseph Vinish D'Silva, Florestan De Moor, and Bettina Kemme.
PVLDB, 2018.

In this paper, we investigate the database-external development environments that are popular with the data scientists and the lack of usability in the alternative database-centric approaches. We propose a novel approach, AIDA, that provides a programming API that is as programmer-friendly as the popular data science systems. While AIDA's Python API allows data scientists to use familiar development environments like Jupyter Notebook that is installed in their local workstations, instead of pulling the data into the client workstations, it follows a client-server model that seamlessly pushes the computations to AIDA's server that resides inside the embedded Python interpreter of the Relational Database Management System (RDBMS). AIDA's programming API is very versatile, supporting both relational operations and linear algebra. Internally, AIDA utilizes the host RDBMS to execute relational operations and a statistical library (NumPy) to perform linear algebra, seamlessly performing any data handover between these systems. We performed extensive performance and usability comparisons with other approaches. This work was done by me under the guidance of my advisor. Florestan performed the test case executions involving Spark. The contributions made by this paper are covered in Chapters 3, 4 & 5.

[DDMK19b] Making an RDBMS Data Scientist Friendly: Advanced In-database Interactive Analytics with Visualization Support.
Joseph Vinish D'Silva, Florestan De Moor, and Bettina Kemme.
PVLDB (Demo), 2019.

In this demonstration, we augment the system proposed in [DDMK18] to integrate the capability to perform both static and interactive data visualizations at the client workstation, even

though the data resides in the server. Advanced data scientists can use matplotlib or plotly Python packages to visualize their data sets. By means of a real-life data science workflow, we demonstrate how data scientists can use this visualization capability to monitor the progression of error rate in real-time as a model is being trained at the server side. This work was done by me under the guidance of my advisor. Florestan performed the test case executions involving Spark. The contributions made by this paper are covered in Chapter 4.

[DDMK19a] Keep Your Host Language Object and Also Query It: A Case for SQL Query Support in RDBMS for Host Language Objects.
Joseph Vinish D'Silva, Florestan De Moor, and Bettina Kemme.
Int. Conference on Scientific and Statistical Database Management, 2019.

In this paper, we propose a virtual table concept that facilitates running SQL queries over high-level language (HLL) objects from embedded HLL code. The virtual table proposal addresses the limitations of user defined functions (UDFs) that are traditionally used for this purpose. The black-box nature of the UDFs results in the RDBMS optimizer generating inefficient execution plans. On the other hand, the optimizer can analyze the data characteristics of virtual tables similar to how it does the same for regular database tables, facilitating efficient plan generation. The idea and design of the virtual table concept was done by myself under the guidance of my advisor. Florestan performed the changes to MonetDB RDBMS modules to integrate the virtual table libraries into it under my guidance. The contributions made by this paper are covered in Chapter 6.

Bibliography

[ABC+16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *Symposium on Operating Systems Design and Implementation*, pages 265–283. USENIX Association, 2016.

[Abd10] Hervé Abdi. Coefficient of Variation. *Encyclopedia of Research Design*, 1:169–171, 2010.

[ABH09] Daniel J. Abadi, Peter A. Boncz, and Stavros Harizopoulos. Column-oriented Database Systems. *PVLDB*, 2(2):1664–1665, 2009.

[AK+17] Lance Ashdown, Tom Kyte, et al. *Oracle Database Concepts, 12c Release 2 (12.2)*. Oracle, 2017.

[ALOR17] Christopher R Aberger, Andrew Lamb, Kunle Olukotun, and Christopher Ré. Mind the Gap: Bridging Multi-domain Query Workloads with EmptyHeaded. *PVLDB*, 10(12):1849–1852, 2017.

[ALOR18] Christopher R Aberger, Andrew Lamb, Kunle Olukotun, and Christopher Ré. LevelHeaded: A Unified Engine for Business Intelligence and Linear Algebra Querying. In *ICDE*. IEEE, 2018.

[AMDM07] Daniel J Abadi, Daniel S Myers, David J DeWitt, and Samuel R Madden. Materialization Strategies in a Column-Oriented DBMS. In *ICDE*, pages 466–475. IEEE, 2007.

[AMF06] Daniel Abadi, Samuel Madden, and Miguel Ferreira. Integrating Compression and Execution in Column-oriented Database Systems. In *SIGMOD*, pages 671–682. ACM, 2006.

[AMH08] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. Column-stores vs. Row-stores: How Different Are They Really? In *SIGMOD*, pages 967–980. ACM, 2008.

[AP03] Mauricio Arregoces and Maurizio Portolani. *Data Center Fundamentals*, chapter "Application Architectures Overview. Cisco Press, 2003.

[Ape88] Peter M. G. Apers. Data Allocation in Distributed Database Systems. *ACM Transactions on Database Systems*, 13(3):263–304, 1988.

[APS09] Thomas Abeel, Yves Van de Peer, and Yvan Saeys. Java-ML: A Machine Learning Library. *Journal of Machine Learning Research*, 10(Apr):931–934, 2009.

[Aur15] Bouteiller Aurélien. *Fault-Tolerance Techniques for High-Performance Computing*, chapter Fault-Tolerant MPI. Computer Communications and Networks. Springer International Publishing, 2015.

[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 *SIGMOD*, pages 1383–1394. ACM, 2015.

[BA19] J. Burton Browning and Marty Alchin. *Pro Python 3: Features and Tools for Professional Development*, chapter Object Management, pages 269–303. Apress, 2019.

[BBC+11] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. Cython: The Best of Both Worlds. *Computing in Science Engineering*, 13(2):31–39, 2011.

[BDE+16] Matthias Boehm, Michael W. Dusenberry, Deron Eriksson, Alexandre V. Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick R. Reiss, Prithviraj Sen, Arvind C. Surve, and Shirish Tatikonda. SystemML: Declarative Machine Learning on Spark. *PVLDB*, 9(13):1425–1436, 2016.

[BDF+98] Peter Baumann, Andreas Dehmel, Paula Furtado, Roland Ritsch, and Norbert Widmann. The Multidimensional Database System RasDaMan. In *SIGMOD*, pages 575–577. ACM, 1998.

[BEKS17] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A Fresh Approach to Numerical Computing. *SIAM review*, 59(1):65–98, 2017.

[BHKP10] Albert Bifet, Geoff Holmes, Richard Kirkby, and Bernhard Pfahringer. MOA: Massive Online Analysis. *Journal of Machine Learning Research*, 11:1601–1604, 2010.

[BHS+14] Sebastian Breß, Max Heimel, Norbert Siegmund, Ladjel Bellatreche, and Gunter Saake. GPU-Accelerated Database Systems: Survey and Open Challenges. *Transactions on Large-Scale Data - and Knowledge-Centered Systems*, pages 1–35, 2014.

[BKK+81] Gerd Bohlender, Edgar Kaucher, Rudi Klatte, U Kulisch, Willard L Miranker, Ch Ullrich, and J Wolff v Gudenberg. FORTRAN for Contemporary Numerical Computation. *Computing*, 26(4):277–314, 1981.

[BKY19] Matthias Boehm, Arun Kumar, and Jun Yang. Data Management in Machine Learning Systems. *Synthesis Lectures on Data Management*, 11(1):1–173, 2019.

[BLSG17] David Birch, David Lyford-Smith, and Yike Guo. The Future of Spreadsheets in the Big Data Era. In *European Spreadsheet Risks Interest Group Conference*. EuSpRIG, 2017.

[Bra08] Peter Brass. *Advanced Data Structures*, volume 1. Cambridge University Press, 2008.

[BS16] Rajesh Bordawekar and Oded Shmueli. Enabling Cognitive Intelligence Queries in Relational Databases using Low-dimensional Word Embeddings. *arXiv preprint arXiv:1603.07185*, 2016.

[BS17] Rajesh Bordawekar and Oded Shmueli. Using Word Embedding to Enable Semantic Queries in Relational Databases. In *Workshop on Data Management for End-to-End Machine Learning*, page 5. ACM, 2017.

[CB74] Donald D Chamberlin and Raymond F Boyce. SEQUEL: A Structured English Query Language. In *ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control*, pages 249–264. ACM, 1974.

[CDC+18] Hanfeng Chen, Joseph Vinish D'Silva, Hongji Chen, Bettina Kemme, and Laurie Hendren. HorseIR: Bringing Array Programming Languages Together with Database Query Processing. In *SIGPLAN International Symposium on Dynamic Languages*, pages 37–49. ACM, 2018.

[CDD+09] Jeffrey Cohen, Brian Dolan, Mark Dunlap, Joseph M. Hellerstein, and Caleb Welton. MAD Skills: New Analysis Practices for Big Data. *PVLDB*, 2(2):1481–1492, 2009.

[CDPW92] Jaeyoung Choi, Jack J Dongarra, Roldan Pozo, and David W Walker. ScaLAPACK: A Scalable Linear Algebra Library for Distributed Memory Concurrent Computers. In *Symposium on the Frontiers of Massively Parallel Computation*, pages 120–127. IEEE, 1992.

[CGD+15] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Carsten Binnig, Ugur Cetintemel, and Stan Zdonik. An Architecture for Compiling UDF-centric Workflows. *PVLDB*, 8(12):1466–1477, August 2015.

[Cla92] Bill G. Claybrook. *OLTP: Online Transaction Processing Systems*. Wiley, 1992.

[Cod70] Edgar F Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, 1970.

[Col11] Yann Collet. RealTime Data Compression: Lz4 explained, 2011. [http://fastcompression.blogspot.com/2011/05/lz4-explained.html, accessed 18-October-2019].

[CP02] Rajiv Chakravorty and Ian Pratt. WWW Performance over GPRS. In *International Workshop on Mobile and Wireless Communications Network*, pages 527–531. IEEE, 2002.

[CS14] Girish Chandrashekar and Ferat Sahin. A survey on feature selection methods. *Computers & Electrical Engineering*, 40(1):16 – 28, 2014.

[CST+10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *ACM Symposium on Cloud Computing*, pages 143–154. ACM, 2010.

[Dal07] Liza Daly. *Next-Generation Web Frameworks in Python*. O'Reilly Short Cut. O'Reilly Media, 2007.

[Dat16] Datacratic. The Machine Learning Database. White paper, Datacratic, 2016.

[DCR14] Nicholas Diakopoulos, Stephen Cass, and Joshua Romero. Data-Driven Rankings: the Design and Development of the IEEE Top Programming Languages News App. In *Symposium on Computation+ Journalism*, 2014.

[DCSW09] Umeshwar Dayal, Malu Castellanos, Alkis Simitsis, and Kevin Wilkinson. Data Integration Flows for Business Intelligence. In *EDBT*, pages 1–11. ACM, 2009.

[DDMK18] Joseph Vinish D'Silva, Florestan De Moor, and Bettina Kemme. AIDA-Abstraction for Advanced In-Database Analytics. *PVLDB*, 11(11):1400–1413, 2018.

[DDMK19a] Joseph Vinish D'Silva, Florestan De Moor, and Bettina Kemme. Keep Your Host Language Object and Also Query It: A Case for SQL Query Support in RDBMS for Host Language Objects. In *International Conference on Scientific and Statistical Database Management*, pages 133–144. ACM, 2019.

[DDMK19b] Joseph Vinish D'Silva, Florestan De Moor, and Bettina Kemme. Making an RDBMS Data Scientist Friendly: Advanced In-database Interactive Analytics with Visualization Support. *PVLDB*, 12(12):1930–1933, 2019.

[Dew98] Dawna Travis Dewire. *Thin Clients*. Web enterprise computing. McGraw-Hill, 1998.

[DF13] Michael M. David and Lee Fesperman. *Advanced Standard SQL Dynamic Structured Data Modeling and Hierarchical Processing*, chapter Standard SQL Join Types and Their Operation. Artech House, 2013.

[DG96] Peter Deutsch and Jean-Loup Gailly. ZLIB Compressed Data Format Specification version 3.3. Technical report, RFC Editor, 1996.

[DG08] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[DILR00] Louis Degenaro, Arun Iyengar, Ilya Lipkind, and Isabelle Rouvellou. A Middleware System Which Intelligently Caches Query Results. In *IFIP/ACM International Conference on Distributed Systems Platforms*, pages 24–44. Springer-Verlag, 2000.

[Din16] Thomas W. Dinsmore. *In-Memory Analytics*, pages 97–116. Apress, Berkeley, CA, 2016.

[DL17] Ioana Delaney and Jia Li. Extending Apache Spark SQL Data Source APIs with Join Push Down, 2017. [https://databricks.com/session/extending-apache-spark-sql-data-source-apis-with-join-push-down, accessed 20-April-2018].

[Dom12] Pedro Domingos. A Few Useful Things to Know About Machine Learning. *Communications of the ACM*, 55(10):78–87, October 2012.

[DT11] John David N. Dionisio and Ray Toal. *Programming with JavaScript: Algorithms and Applications for Desktop and Mobile Browsers*, chapter Distributed Computing. G - Reference, Information and Interdisciplinary Subjects Series. Jones & Bartlett Learning, 2011.

[DW02] Joshua D. Drake and John C. Worsley. *Practical PostgreSQL*. O'Reilly Media, 2002.

[EBH07] John Wesley Eaton, David Bateman, and Søren Hauberg. *GNU Octave*. Free Software Foundation, 2007.

[Eis96] Andrew Eisenberg. New Standard for Stored Procedures in SQL. *SIGMOD Record*, 25(4):81–88, 1996.

[FAG+11] Markus Fiedler, Patrik Arlos, Timothy A. Gonsalves, Anuraag Bhardwaj, and Hans Nottehed. Time is Perception is Money — Web Response Times in Mobile Networks with Application to Quality of Experience. In *International Conference on Performance Evaluation of Computer and Communication Systems: Milestones and Future Challenges*, pages 179–190. Springer-Verlag, 2011.

[FEB03] Maydene Fisher, Jon Ellis, and Jonathan Bruce. *JDBC API Tutorial and Reference*. Java series. Addison-Wesley, 2003.

[FEB18] Edouard Fouché, Alexander Eckert, and Klemens Böhm. In-Database Analytics with ibmdbpy. In *EDBT*. ACM, 2018.

[FG16] Elvis C. Foster and Shripad Godbole. *Database Systems: A Pragmatic Approach*, chapter Limitations of SQL. Apress, 2016.

[Fje08] Hans-Christian Fjeldberg. Polyglot Programming. Master's thesis, Norwegian University of Science and Technology, Norway, 2008.

[FKRR12] Xixuan Feng, Arun Kumar, Benjamin Recht, and Christopher Ré. Towards a Unified Architecture for in-RDBMS Analytics. In *SIGMOD*, pages 325–336. ACM, 2012.

[FL03] Maria Cristina Ferreira de Oliveira and Haim Levkowitz. From Visual Data Exploration to Visual Data Mining: A Survey. *IEEE Transactions on Visualization and Computer Graphics*, 9(3):378–394, 2003.

[Fow10] Martin Fowler. *Domain-Specific Languages*. Pearson Education, 2010.

[FP05] Steven Feuerstein and Bill Pribyl. *Oracle PL/SQL Programming*. O'Reilly Media, 2005.

[FP16] Jean-Daniel Fekete and Romain Primet. Progressive Analytics: A Computation Paradigm for Exploratory Data Analysis. *CoRR*, abs/1607.05162, 2016.

[FR03] Martin Fowler and David Rice. *Patterns of Enterprise Application Architecture*, chapter Mapping to Relational Databases. A Martin Fowler signature book. Addison-Wesley, 2003.

[GE03] Isabelle Guyon and André Elisseeff. An Introduction to Variable and Feature Selection. *Journal of Machine Learning Research*, 3(Mar):1157–1182, 2003.

[Gei95] Kyle Geiger. *Inside ODBC*. Microsoft Press, 1995.

[GKP+11] Amol Ghoting, Rajasekar Krishnamurthy, Edwin Pednault, Berthold Reinwald, Vikas Sindhwani, Shirish Tatikonda, Yuanyuan Tian, and Shivakumar Vaithyanathan. SystemML: Declarative Machine Learning on MapReduce. In *ICDE*, pages 231–242. IEEE, 2011.

[GLW+11] Philipp Große, Wolfgang Lehner, Thomas Weichert, Franz Färber, and Wen-Syan Li. Bridging Two Worlds with RICE Integrating R into the SAP In-Memory Computing Engine. *PVLDB*, 4(12):1307–1317, 2011.

[GMP02] Phillip B. Gibbons, Yossi Matias, and Viswanath Poosala. Fast Incremental Maintenance of Approximate Histograms. *Transactions on Database Systems*, 27(3):261–298, 2002.

[GMSS15] Mark Grover, Ted Malaska, Jonathan Seidman, and Gwen Shapira. *Hadoop Application Architectures: Designing Real-World Big Data Applications*. O'Reilly Media, 2015.

[GMSvE98] Michael Godfrey, Tobias Mayr, Praveen Seshadri, and Thorsten von Eicken. Secure and Portable Database Extensibility. In *SIGMOD*, pages 390–401. ACM, 1998.

[GMUW13] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*, chapter Algebraic and Logical Query Languages. Pearson, 2013.

[God19] AhmadReza GodarzvandChegini. Federated Learning Algorithms on top of Distributed Relational Database Systems. Master's thesis, McGill University, Montréal, Quebec, Canada, 2019.

[Goo19] Google. snappy | A fast compressor/decompressor, 2019. [http://google.github.io/snappy, accessed 18-October-2019].

[Got75] Leo R. Gotlieb. Computing Joins of Relations. In *SIGMOD*, pages 55–63. ACM, 1975.

[Hal15] Fern Halper. Next-Generation Analytics and Platforms for Business Success. *TDWI Best Practices Report: First Quarter*, 2015.

[HDW94a] Geoffrey Holmes, Andrew Donkin, and Ian H Witten. Weka: A Machine Learning Workbench. In *Australian and New Zealand Conference on Intelligent Information Systems*, pages 357–361. IEEE, 1994.

[HDW94b] Geoffrey Holmes, Andrew Donkin, and Ian H. Witten. WEKA: A Machine Learning Workbench. In *Australian New Zealand Intelligent Information Systems*, pages 357–361. IEEE, 1994.

[HH16] Desmond J. Higham and Nicholas J. Higham. *MATLAB Guide*, chapter Large Data Sets. Society for Industrial and Applied Mathematics, 2016.

[HHLD11] Jing Han, E Haihong, Guan Le, and Jian Du. Survey on NoSQL Database. In *International Conference on Pervasive Computing and Applications*, pages 363–366. IEEE, 2011.

[HLAM06] Stavros Harizopoulos, Velen Liang, Daniel J. Abadi, and Samuel Madden. Performance Tradeoffs in Read-optimized Databases. In *PVLDB*, pages 487–498. VLDB Endowment, 2006.

[HLR02] Laura M Haas, E. T. Lin, and Mary Roth. Data integration through database federation. *IBM Systems Journal*, 41(4):578–596, 2002.

[HLZ+03] Kurt Hornik, Friedrich Leisch, Achim Zeileis, et al. Statistical Computing and Databases: Distributed Computing Near the Data. In *Distributed Statistical Computing*, page 2, 2003.

[HRK17] Pedro Holanda, Mark Raasveldt, and Martin Kersten. Don't Hold My UDFs Hostage-Exporting UDFs For Debugging Purposes. In *International Conference on Simpósio Brasileiro de Banco de Dados (SSBD)*, 2017.

[HRS+12] Joseph M. Hellerstein, Christoper Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, and Arun Kumar. The MADlib Analytics Library: Or MAD Skills, the SQL. *PVLDB*, 5(12):1700–1711, 2012.

[HSH07] Joseph M. Hellerstein, Michael Stonebraker, and James Hamilton. Architecture of a Database System. *Foundations and Trends® in Databases*, 1(2):141–259, February 2007.

[Hun07] John D Hunter. Matplotlib: A 2D Graphics Environment. *Computing in Science & Engineering*, 9(3):90, 2007.

[IBM08] IBM. IBM InfoSphere DataStage Balanced Optimization. White paper, IBM, June 2008.

[IG96] Ross Ihaka and Robert Gentleman. R: A Language for Data Analysis and Graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, 1996.

[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. *IEEE Data Engineering Bulletin*, 35(1):40–45, 2012.

[Inf07] Informatica Corporation. How to Achieve Flexible, Cost-effective Scalability and Performance through Pushdown Processing. White paper, Informatica Corporation, November 2007.

[Ioa96] Yannis E. Ioannidis. Query Optimization. *ACM Computing Surveys*, 28(1):121–123, 1996.

[IS03] Martin Isenburg and Jack Snoeyink. Binary Compression Rates for ASCII Formats. In *International Conference on 3D Web Technology*, pages 173–ff. ACM, 2003.

[JK84] Matthias Jarke and Jurgen Koch. Query Optimization in Database Systems. *ACM Computing Surveys*, 16(2):111–152, June 1984.

[JWHT17] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning: with Applications in R*, chapter Linear Regression. Springer Texts in Statistics. Springer New York, 2017.

[Kag17] Kaggle. The State of ML and Data Science 2017 | Kaggle, 2017. [https://www.kaggle.com/surveys/2017, accessed 5-December-2019].

[KBY17] Arun Kumar, Matthias Boehm, and Jun Yang. Data Management in Machine Learning: Challenges, Techniques, and Systems. In *SIGMOD*, pages 1717–1722. ACM, 2017.

[KDn14] KDnuggets. What data types/sources you analyzed in the past 12 months?, 2014. [https://www.kdnuggets.com/polls/2014/data-types-sources-analyzed.html, accessed 5-December-2019].

[Kei02] Daniel A. Keim. Information Visualization and Visual Data Mining. *IEEE Transactions on Visualization and Computer Graphics*, 8(1):1–8, 2002.

[Kel18] Benjamin Walter Keller. *Mastering Matplotlib 2.x: Effective Data Visualization techniques with Python*. Packt Publishing, 2018.

[Ket17] Ketkar, Nikhil. *Deep Learning with Python: A Hands-on Introduction*, chapter Introduction to PyTorch, pages 195–208. Apress, 2017.

[KGS17] Georgia Kougka, Anastasios Gounaris, and Alkis Simitsis. The Many Faces of Data-centric Workflow Optimization: A Survey. *CoRR*, abs/1701.07723, 2017.

[KKS+19] Andreas Kunft, Asterios Katsifodimos, Sebastian Schelter, Sebastian Breß, Tilmann Rabl, and Volker Markl. An Intermediate Representation for Optimizing Machine Learning Pipelines. *PVLDB*, 12(11):1553–1567, 2019.

[KLG+19] Torsten Kilias, Alexander Löser, Felix Gers, Ying Zhang, Richard Koopmanschap, and Martin Kersten. IDEL: In-Database Neural Entity Linking. In *International Conference on Big Data and Smart Computing*, pages 1–8. IEEE, 2019.

[KPHH12] Sean Kandel, Andreas Paepcke, Joseph M. Hellerstein, and Jeffrey Heer. Enterprise Data Analysis and Visualization: An Interview Study. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2917–2926, 2012.

[KRKP+16] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, et al. Jupyter Notebooks—a publishing format for reproducible computational workflows. *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, pages 87–90, 2016.

[Kuc98] Andrew M Kuchling. The Python DB-API. *Linux Journal*, 1998(49es):8, 1998.

[Lam19] Greg Lamp. pandasql, 2019. [https://pypi.org/project/pandasql, accessed 24-December-2018].

[LC16] Deanne Larson and Victor Chang. A review and future direction of agile, business intelligence, analytics and data science. *International Journal of Information Management*, 36(5):700 – 710, 2016.

[LGG+17] Shangyu Luo, Zekai J Gao, Michael Gubanov, Luis L Perez, and Christopher Jermaine. Scalable Linear Algebra on a Relational Database System. In *ICDE*, pages 523–534. IEEE, 2017.

[Lim08] Rudy Limeback. *Simply SQL*, chapter The WHERE clause. SitePoint, 2008.

[Lin01] Lisa E. Lindgren. *Application Servers for E-Business*. Auerbach Publications, 2001.

[LKD+88] Volker Linnemann, Klaus Küspert, Peter Dadam, Peter Pistor, R Erbe, Alfons Kemper, Norbert Südkamp, Georg Walch, and Mechtild Wallrath. Design and Implementation of an Extensible Database Management System Supporting User Defined Data Types and Functions. In *VLDB*, pages 294–305, 1988.

[LL99a] Alexis Leon and Mathews Leon. *SQL: A Complete Reference*, chapter Programming with SQL. TaTa McGraw-Hill, 1999.

[LL99b] Alexis Leon and Mathews Leon. *SQL: A Complete Reference*, chapter Embedded SQL. TaTa McGraw-Hill, 1999.

[LM14] Jonathan Lajus and Hannes Mühleisen. Efficient Data Management and Statistics with Zero-Copy Integration. In *International Conference on Scientific and Statistical Database Management*, pages 12:1–12:10. ACM, 2014.

[Lot19] A. Loth. *Visual Analytics with Tableau*. Wiley, 2019.

[LPVM15] Ji Liu, Esther Pacitti, Patrick Valduriez, and Marta Mattoso. A Survey of Data-Intensive Scientific Workflow Management. *Journal of Grid Computing*, 13(4):457–493, 2015.

[Lyn88] Clifford A. Lynch. Selectivity Estimation and Query Optimization in Large Databases with Highly Skewed Distribution of Column Values. In *VLDB*, pages 240–251. Morgan Kaufmann Publishers Inc., 1988.

[MAB08] Sergey Melnik, Atul Adya, and Philip A Bernstein. Compiling Mappings to Bridge Applications and Databases. *Transactions on Database Systems*, 33(4):22, 2008.

[MBY+16] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. MLlib: Machine Learning in Apache Spark. *The Journal of Machine Learning Research*, 17(1):1235–1241, 2016.

[McK11] Wes McKinney. pandas: a Foundational Python Library for Data Analysis and Statistics. *Python for High Performance and Scientific Computing*, pages 1–9, 2011.

[McL02] Brett McLaughlin. *Building Java Enterprise Applications: Architecture*, chapter Blueprints. Building Java Enterprise Applications. O'Reilly, 2002.

[MCS88] Michael V. Mannino, Paicheng Chu, and Thomas Sager. Statistical Profile Estimation in Database Systems. *ACM Computing Surveys*, 20(3):191–221, September 1988.

[MDZ+17] Parmita Mehta, Sven Dorkenwald, Dongfang Zhao, Tomer Kaftan, Alvin Cheung, Magdalena Balazinska, Ariel Rokem, Andrew Connolly, Jacob Vanderplas, and Yusra AlSayyad. Comparative Evaluation of Big-Data Systems on Scientific Image Analytics Workloads. *PVLDB*, 10(11):1226–1237, 2017.

[ME92] Priti Mishra and Margaret H. Eich. Join Processing in Relational Databases. *ACM Computing Surveys*, 24(1):63–113, 1992.

[MEW08] Fabrice Marguerie, Steve Eichert, and Jim Wooley. *LINQ in Action*. Manning, 2008.

[Mic12] MicroStrategy. Architecture for Enterprise Business Intelligence. White paper, MicroStrategy, Incorporated, 2012.

[Mil16] Tim Miller. Using R and Python in the Teradata Database. White paper, Teradata, 2016.

[Mit18] Ryan Mitchell. *Web Scraping with Python: Collecting More Data from the Modern Web*. O'Reilly Media, 2018.

[ML13] Hannes Mühleisen and Thomas Lumley. Best of Both Worlds: Relational Databases and Statistics. In *International Conference on Scientific and Statistical Database Management*, pages 32:1–32:4. ACM, 2013.

[MMS17] Wendy L Martinez, Angel R Martinez, and Jeffrey Solka. *Exploratory Data Analysis with MATLAB*. Chapman and Hall/CRC, 2017.

[MS99] Arunprasad P Marathe and Kenneth Salem. Query Processing Techniques for Arrays. In *SIGMOD*, pages 323–334. ACM, 1999.

[MS02] Jim Melton and Alan R. Simon. *SQL:1999: Understanding Relational Language Components*. Morgan Kaufmann Series in Data. Morgan Kaufmann, 2002.

[MSS+11] Michael M. McKerns, Leif Strand, Tim Sullivan, Alta Fang, and Michael A.G. Aivazis. Building a Framework for Predictive Science. In *Python in Science Conference*, pages 67 – 78, 2011.

[MTT12] Yannis Manolopoulos, Yannis Theodoridis, and Vassilis J. Tsotras. *Advanced Database Indexing*. Advances in Database Systems. Springer, 2012.

[Mun12] M Arthur Munson. A Study on the Importance of and Time Spent on Different Modeling Steps. *SIGKDD Explorations*, 13(2):65–71, 2012.

[Nai14] Vineeth G. Nair. *Getting Started with Beautiful Soup*. Packt Publishing, 2014.

[NB99] Amin Y. Noaman and Ken Barker. A Horizontal Fragmentation Algorithm for the Fact Relation in a Distributed Data Warehouse. In *International Conference on Information and Knowledge Management*, pages 154–161. ACM, 1999.

[ODFA12] Sean Owen, Ted Dunning, Ellen Friedman, and Robin Anil. *Mahout in Action*. In Action Series. Manning Publications, 2012.

[OH17] Regina O Obe and Leo S Hsu. *Postgresql: Up and Running: A Practical Guide to the Advanced Open Source Database*. O'Reilly Media, 2017.

[OHB+11] Fatma Özcan, David Hoa, Kevin S. Beyer, Andrey Balmin, Chuan Jie Liu, and Yu Li. Emerging Trends in the Enterprise Data Analytics: Connecting Hadoop and DB2 Warehouse. In *SIGMOD*, pages 1161–1164. ACM, 2011.

[OP11] Carlos Ordonez and Sasi K Pitchaimalai. One-pass Data Mining Algorithms in a DBMS with UDFs. In *SIGMOD*, pages 1217–1220. ACM, 2011.

[ORS+08] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: A Not-so-foreign Language for Data Processing. In *SIGMOD*, pages 1099–1110. ACM, 2008.

[Par19] Aditya Parameswaran. Enabling Data Science for the Majority. *PVLDB*, 12(12):2309–2322, 2019.

[PG17] Josh Patterson and Adam Gibson. *Deep Learning: A Practitioner's Approach*. O'Reilly Media, 2017.

[PGM+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Conference on Neural Information Processing Systems*, pages 8024–8035, 2019.

[Pia17] Gregory Piatetsky. New Leader, Trends, and Surprises in Analytics, Data Science, Machine Learning Software Poll, 2017. [https://www.kdnuggets.com/2017/05/poll-analytics-data-science-machine-learning-software-leaders.html, accessed 07-December-2017].

[PPR+09] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J Abadi, David J DeWitt, Samuel Madden, and Michael Stonebraker. A Comparison of Approaches to Large-scale Data Analysis. In *SIGMOD*, pages 165–178. ACM, 2009.

[PRWZ17] Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. Data Management Challenges in Production Machine Learning. In *SIGMOD*, pages 1723–1726. ACM, 2017.

[PS98] František Plášil and Michael Stal. An Architectural View of Distributed Objects and Components in CORBA, Java RMI and COM/DCOM. *Software-Concepts & Tools*, 19(1):14–28, 1998.

[PTS+17] Shoumik Palkar, James J Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, Matei Zaharia, and Stanford InfoLab. Weld: A Common Runtime for High Performance Data Analytics. In *CIDR*, 2017.

[PVG+11] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12(Oct):2825–2830, 2011.

[Pyt18] Python Software Foundation. Embedding Python in Another Application, 2018. [https://docs.python.org/3/extending/embedding.html, accessed 19-December-2018].

[R C99] R Core Team. Writing R Extensions. *R Foundation for Statistical Computing*, 1999.

[R C14] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2014.

[Raa07] David Raab. How to Judge a Columnar Database. *DM Review*, 17(12):33, 2007.

[Raa08] David M Raab. How to Judge a Columnar Database, Revisited. *Information Management*, 18(10):15, 2008.

[Raa15] Mark Raasveldt. Vectorized UDFs in Column-Stores. Master's thesis, Utrecht University, The Netherlands, 2015.

[Raa16] Mark Raasveldt. Voter Classification Using MonetDB/Python, 2016. [https://www.monetdb.org/blog/voter-classification-using-monetdbpython, accessed 07-December-2017].

[Raa18] Mark Raasveldt. MonetDBLite: An Embedded Analytical Database. In *SIGMOD*, pages 1837–1838. ACM, 2018.

[RAB+15] Christopher Ré, Divy Agrawal, Magdalena Balazinska, Michael Cafarella, Michael Jordan, Tim Kraska, and Raghu Ramakrishnan. Machine Learning and Databases: The Sound of Things to Come or a Cacophony of Hype? In *SIGMOD*, pages 283–284. ACM, 2015.

[RDVP16] Petar Ristoski, Gerben Klaas Dirk De Vries, and Heiko Paulheim. A Collection of Benchmark Datasets for Systematic Evaluations of Machine Learning on the Semantic Web. In *International Semantic Web Conference*, pages 186–194. Springer, 2016.

[RG03] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*, chapter Relational Algebra and Calculus. Irwin Computer Science. McGraw-Hill, 2003.

[RHMM18] Mark Raasveldt, Pedro Holanda, Hannes Mühleisen, and Stefan Manegold. Deep Integration of Machine Learning Into Column Stores. In *EDBT*, pages 473–476. OpenProceedings.org, 2018.

[RM16] Mark Raasveldt and Hannes Mühleisen. Vectorized UDFs in Column-Stores. In *International Conference on Scientific and Statistical Database Management*, pages 16:1–16:12. ACM, 2016.

[RM17] Mark Raasveldt and Hannes Mühleisen. Don't Hold My Data Hostage–A Case For Client Protocol Redesign. *PVLDB*, 10(10):1022–1033, 2017.

[RMYC13] Lavanya Ramakrishnan, Pradeep K Mantha, Yushu Yao, and Richard S Canon. Evaluation of NoSQL and Array Databases for Scientific Applications. In *DataCloud Workshop*, 2013.

[Roc15] Matthew Rocklin. Dask: Parallel Computation with Blocked algorithms and Task Scheduling. In *Python in Science Conference*, pages 126–132, 2015.

[RPE+17] Karthik Ramachandra, Kwanghyun Park, K. Venkatesh Emani, Alan Halverson, César Galindo-Legaria, and Conor Cunningham. Froid: Optimization of Imperative Programs in a Relational Database. *PVLDB*, 11(4):432–444, 2017.

[SAFFGA18] Andres Suárez, Miguel A. Alvarez-Feijoo, Raquel Fernández González, and Elena Arce. Teaching optimization of manufacturing problems via code components of a Jupyter Notebook. *Computer Applications in Engineering Education*, 26(5):1102–1110, 2018.

[SAP17] SAP. *SAP BusinessObjects Web Intelligence User's Guide*. SAP SE, 2017.

[Sar14] Alireza Sarveniazi. An Actual Survey of Dimensionality Reduction. *American Journal of Computational Mathematics*, 4(02):55, 2014.

[SBZB13] Michael Stonebraker, Paul Brown, Donghui Zhang, and Jacek Becla. SciDB: A Database Management System for Applications with Complex Analytics. *Journal of Computing in Science & Engineering*, 15(3):54–62, 2013.

[Sch05] Robert D. Schneider. *MySQL Database Design and Tuning*. Pearson Education, 2005.

[SE07] Sai Sumathi and Sankaralingam Esakkirajan. *Fundamentals of Relational Database Management Systems*, chapter Overview of Database Management System, pages 1–30. Studies in Computational Intelligence. Springer Berlin Heidelberg, 2007.

[Sin16] Chandraish Sinha. *QlikView Essentials*. Packt Publishing, 2016.

[SKS11] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts*, chapter Indexing and Hashing. McGraw-Hill, 2011.

[SL90] Amit P. Sheth and James A. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys*, 22(3):183–236, 1990.

[sol19] solid IT. DB-Engines Ranking per database model category, 2019. [https://db-engines.com/en/ranking_categories, accessed 5-December-2019].

[SS01] G Lawrence Sanders and Seungkyoon Shin. Denormalization Effects on Performance of RDBMS. In *Hawaii International Conference on System Sciences*. IEEE, 2001.

[ST08] Alkis Simitsis and Dimitri Theodoratos. *Encyclopedia of Data Warehousing and Mining*, chapter Data Warehouse Back-End Tools. Idea Group Reference, 2008.

[The17] The PostgreSQL Global Development Group. Procedural Languages. In *PostgreSQL 10.0 Documentation*, 2017.

[Tra17] Transaction Processing Performance Council. TPC Benchmark H, 2017.

[TSXVV11] Grigorios Tsoumakas, Eleftherios Spyromitros-Xioufis, Jozef Vilcek, and Ioannis Vlahavas. MULAN: A Java Library for Multi-Label Learning. *Journal of Machine Learning Research*, 12(Jul):2411–2414, 2011.

[US19] Phillip Merlin Uesbeck and Andreas Stefik. A Randomized Controlled Trial on the Impact of Polyglot Programming in a Database Context. In *Workshop on Evaluation and Usability of Programming Languages and Tools*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019.

[Var18] Mihai Varga. Just-in-time compilation in MonetDB with Weld. Master's thesis, Centrum Wiskunde & Informatica, The Netherlands, 2018.

[VCV11] Stéfan Van der Walt, S Chris Colbert, and Gael Varoquaux. The NumPy Array: A Structure for Efficient Numerical Computation. *Journal of Computing in Science & Engineering*, 13(2):22–30, 2011.

[Vin75] Thaddeus Vincenty. Direct and Inverse Solutions of Geodesics on the Ellipsoid with Application of Nested Equations. *Survey Review*, 23(176):88–93, 1975.

[VSL+16] Manasi Vartak, Harihar Subramanyam, Wei-En Lee, Srinidhi Viswanathan, Saadiyah Husnoo, Samuel Madden, and Matei Zaharia. ModelDB: A System for Machine Learning Model Management. In *Workshop on Human-In-the-Loop Data Analytics*, pages 14:1–14:3. ACM, 2016.

[WDG+16] Buck Woody, Danielle Dea, Debraj GuhaThakurta, Gagan Bansal, Matt Conners, and Tok Wee-Hyong. *Data Science with Microsoft SQL Server 2016*. Microsoft Press, 2016.

[WL02] Hugh E. Williams and David Lane. *Web Database Applications with PHP, and MySQL*, chapter Database Applications and the Web. O'Reilly, 2002.

[WMC+13] David R. White, James McDermott, Mauro Castelli, Luca Manzoni, Brian W. Goldman, Gabriel Kronberger, Wojciech Jaśkowski, Una-May O'Reilly, and Sean Luke. Better GP Benchmarks: Community Survey Results and Proposals. *Genetic Programming and Evolvable Machines*, 14(1):3–29, 2013.

[Wol12] Marilyn Wolf. *Computers as Components: Principles of Embedded Computing System Design*, chapter Program Design and Analysis. Engineering Professional Collection. Morgan Kaufmann, 2012.

[WZC+16] Wei Wang, Meihui Zhang, Gang Chen, HV Jagadish, Beng Chin Ooi, and Kian-Lee Tan. Database Meets Deep Learning: Challenges and Opportunities. *SIGMOD Record*, 45(2):17–22, 2016.

[XKG10] Yu Xu, Pekka Kostamaa, and Like Gao. Integrating Hadoop and Parallel DBMs. In *SIGMOD*, pages 969–974. ACM, 2010.

[XML+18] Doris Xin, Litian Ma, Jialin Liu, Stephen Macke, Shuchen Song, and Aditya Parameswaran. HELIX: Accelerating Human-in-the-loop Machine Learning. *PVLDB*, 11(12):1958–1961, 2018.

[XMM+18] Doris Xin, Stephen Macke, Litian Ma, Jialin Liu, Shuchen Song, and Aditya Parameswaran. HELIX: Holistic Optimization for Accelerating Iterative Machine Learning. *PVLDB*, 12(4):446–460, 2018.

[XMSP18] Doris Xin, Litian Ma, Shuchen Song, and Aditya Parameswaran. How Developers Iterate on Machine Learning Workflows. In *IDEA Workshop at KDD*, 2018.

[YCY18] Aldrin Yim, Claire Chung, and Allen Yu. *Matplotlib for Python Developers: Effective techniques for data visualization with Python, 2nd Edition*. Packt Publishing, 2018.

[YL03] Lei Yu and Huan Liu. Feature Selection for High-Dimensional Data: A Fast Correlation-Based Filter Solution. In *International Conference on Machine Learning*, pages 856–863, 2003.

[YL04] Lei Yu and Huan Liu. Efficient Feature Selection via Analysis of Relevance And redundancy. *Journal of Machine Learning Research*, 5(Oct):1205–1224, 2004.

[YLP+17] Dandong Yin, Yan Liu, Anand Padmanabhan, Jeff Terstriep, Johnathan Rush, and Shaowen Wang. A CyberGIS-Jupyter Framework for Geospatial Analytics at Scale. In *Practice and Experience in Advanced Research Computing*, pages 18:1–18:8. ACM, 2017.

[YTLL17] Jane Yu, Kathy Tzeng, and Janis Landry-Lane. A Reference Architecture for High Performance Analytics in Healthcare and Life Science. Technical white paper, IBM, March 2017.

[ZCD+12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Symposium on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[ZCF+10] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing With Working Sets. *HotCloud*, 10(10-10):95, 2010.

[Zho18] Jingren Zhou. *Encyclopedia of Database Systems*, chapter Index Join, pages 1845–1845. Springer New York, 2018.

[ZHY09] Yi Zhang, Herodotos Herodotou, and Jun Yang. RIOT: I/O-Efficient Numerical Computing without SQL. In *CIDR*, 2009.

[ZKM13] Ying Zhang, Martin Kersten, and Stefan Manegold. SciQL: Array Data Processing Inside an RDBMS. In *SIGMOD*, pages 1049–1052. ACM, 2013.

[ZMK+06] Beibei Zou, Xuesong Ma, Bettina Kemme, Glen Newton, and Doina Precup. Data Mining Using Relational Database Management Systems. In *Advances in Knowledge Discovery and Data Mining*, pages 657–667, 2006.

[ZMPC18] Mingyi Zhang, Patrick Martin, Wendy Powley, and Jianjun Chen. Workload Management in Database Management Systems: A Taxonomy. *Transactions on Knowledge and Data Engineering*, 30(7):1386–1402, 2018.

[ZXW+16] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J

Franklin, et al. Apache Spark: A Unified Engine for Big Data Processing. *Communications of the ACM*, 59(11):56–65, 2016.

[ZZY10] Yi Zhang, Weiping Zhang, and Jun Yang. I/O-Efficient Statistical Computing with RIOT. In *ICDE*, pages 1157–1160. IEEE, 2010.

Acronyms

AQL Array Query Language

B2C Business to Customer

BI Business Intelligence

CLI Common Language Interface

CSV comma-separated values

DBMS Database Management System

DDL Data Definition Language

DML Data Manipulation Language

DMRO Database Memory Resident Object

DOC Distributed Object Communication

DSL Domain-specific Language

ETL Extract Transform and Load

GPS Global Positioning System

GUI Graphical User Interface

HLL high-level language

IDE Integrated Development Environment

IR Intermediate Representation

JDBC Java Database Connectivity

LAN Local Area Network

LOC lines of code

LRU least recently used

MLDB Machine Learning Database

ODBC Object Database Connectivity

OLTP Online Transaction Processing

ORM Object-Relational Mapping

POS point of sale system

TCP Transmission Control Protocol

TPC Transaction Processing Performance Council

RDBMS Relational Database Management System

RDD Resilient Distributed Dataset

RMI Remote Method Invocation

SF Scale Factor

UDF User Defined Function

VPN Virtual Private Network

YCSB Yahoo! Cloud Serving Benchmark

A

AIDA Programming API Reference

The purpose of this appendix is to cover the client-side programming API of AIDA in detail for those who are interested in exploring the practical aspects of the system. We assume a rudimentary knowledge of Python and familiarity of data processing concepts associated with data science workflows such as relational operations and linear algebra. Although some topics related to the programming API were covered in the previous Chapters, for the sake of continuity and completeness, we will reiterate some of their characteristics, while providing fresh examples. Further, as AIDA provides data visualization support by leveraging existing systems, users who want to make use of data visualization capabilities in AIDA are expected to be familiar with these systems. For convenience, we have included references to some sources with detailed coverage on the APIs of these systems in the relevant sections.

A.1 Relational Operations

AIDA's TabularData objects provide support for a variety of relational operations through an ORM-style API. As pointed out in Section 3.1, many of its syntactic conventions are derived from Django's ORM model [Dal07]. In this section, we will further explore the relational capabilities

Table A.1: Basic binary comparison operators for Q objects

Operator	**Description** and **Example**

EQ	Checks if the two operands have the same value (equal).
	`customer.filter(Q('c_mktsegment', C('BUILDING'), CMP.EQ))`
	This is the default comparison and hence could be omitted and written as.
	`customer.filter(Q('c_mktsegment', C('BUILDING')))`

NE	Checks if the two operands have different values (not equal). `customer.filter(Q('c_mktsegment', C('BUILDING'), CMP.NE))`
GT	Checks if the first operand is greater than the second one. `lineitem.filter(Q('l_shipdate', DATE('1994-01-01'), CMP.GT))`
GTE	Checks if the first operand is greater or equal to the second one. `lineitem.filter(Q('l_shipdate', DATE('1994-01-01'), CMP.GTE))`
LT	These are the less-than operator counterparts to the greater-than operators.
LTE	

supported by TabularData objects and their syntactic nuances with additional examples. TabularData objects provide support for the relational operations *selection*, *projection*, *aggregation*, and *join* as discussed below.

A.1.1 Selection

TabularData objects support the relational *selection* operation (equivalent of the SQL statement's `WHERE` clause) through the `filter` API call.

```
tabularDataObject.filter(cond1, cond2, cond3, ...)
```

Where `cond1`, etc., are predicate conditions represented using `Q` objects which are then combined through a logical `AND` operation. In its most fundamental form, a `Q` object takes two columns and a *comparison operator* as its argument.

```
Q(col1, col2, oper)
```

The columns used in the conditional expressions of `Q` objects can be a column of the source TabularData object or a constant literal. If a constant literal is to be used in the conditional expression, then it must be wrapped using a `C` object for string and numeric literals and using a `DATE` object for date literals. The *comparison operators* are accessed through the `CMP` object. The basic *binary comparison operators* are listed in Table A.1.

The logical AND (&) and OR (|) operators allow complex conditions to be built by combining multiple Q objects to arbitrary depths, by constructing complex Q objects.

```
Q(col1, col2, oper1) & Q(col3, col4, oper2)
Q(col1, col2, oper1) | Q(col3, col4, oper2)
```

Example A.1. In the following example based on TPC-H, we select only those records from the LINEITEM table such that either their shipment date or commit date is on or after 1994−01−01 and the specified shipment mode is by AIR.

```
lineitem.filter((Q('l_shipdate', DATE('1994-01-01'), CMP.GTE)    |
                 Q('l_commitdate', DATE('1994-01-01'), CMP.GTE)
                ) & Q('l_shipmode', C('AIR')))
```

Q objects also provides complex *list comparison operators* that allow a column's value to be compared simultaneously with multiple elements in a list.

```
Q(col, list, oper)
```

The list itself could be a regular Python list/tuple object or a TabularData object with a single source column. These operators are shown in Table A.2 along with example usage.

Q objects also supports *unary operators* (see Table A.3). The NOT operator is used to negate a conditional expression represented by a (possibly complex) Q object. NL and NNL are used to check if a certain column is NULL or NOT NULL, respectively, offering the same functionality as their familiar SQL counterparts.

```
Q(col, oper)
```

Finally, Q objects also support the string pattern comparisons that are traditionally provided in SQL through its LIKE operators [Lim08].

```
Q(col, pattern, oper)
```

Here pattern is a string literal representing a pattern that conforms to the SQL standard. Being a string literal, it must be encapsulated using a C object. The API usage is shown in Table A.4.

Table A.2: List comparison operators for Q objects

Operator	**Description** and **Example**
IN	Used to check if a column's value is in the list. `customer.filter(Q('c_mktsegment', ('BUILDING','FURNITURE'), CMP.IN))`
NOTIN	Negation of the IN operator.

GTALL	Check if the column's value is greater-than all of the elements in the list.
	`lineitem.filter(Q('l_shipdate'`
	`,(DATE('1994-01-01'),DATE('1994-07-01'))`
	`,CMP.GTALL))`

GTANY	Check if the column's value is greater-than at the least one of the elements in the list.
	`lineitem.filter(Q('l_shipdate'`
	`,(DATE('1994-01-01'),DATE('1994-07-01'))`
	`,CMP.GTANY))`

GTEALL	These are the greater-than or equal to counterparts to the two greater-than operators described above.
GTEANY	
LTANY	These are the less-than counterparts to the greater-than operators described above.
LTALL	
LTEANY	
LTEALL	

Table A.3: Unary operators for Q objects

Operator	**Description** and **Example**
NOT	Can be used to produce the negation of a (possibly complex) `Q` object. `customer.filter(Q(Q('c_mktsegment', C('BUILDING')), CMP.NOT))`
NL	Used to check if the source column is `NULL`. `customer.filter(Q('c_mktsegment', CMP.NL))`
NNL	Used to ensure that the source column is `NOT NULL`. `customer.filter(Q('c_mktsegment', CMP.NNL))`

Table A.4: List comparison operators for Q objects

Operator	**Description** and **Example**
LIKE	Verifies if the column's value matches the specified string pattern. `customer.filter(Q('c_mktsegment', C('BUILD%'), CMP.LIKE))`
NOTLIKE	Ensures that the column's value does not match the specified string pattern. `customer.filter(Q('c_mktsegment', C('BUILD%'), CMP.NOTLIKE))`

A.1.2 Projection

The relational *projection* operation (equivalent of the SQL `SELECT` clause) is supported by TabularData objects through the `project` API call. This API expects a list of source columns as its argument.

```
tabularDataObject.project((col1, col2, col3, ...))
```

Additionally, the projection transformation allows building mathematical expressions over the selected source columns as well as renaming source columns. This is accomplished by passing a key-value pair as an argument for each such column/expression, where the key is the expression and value is the column name to be given to the output of the said expression. Expressions can refer to the values of source columns by wrapping the corresponding column name using the `F` objects. The mathematical operations of addition, subtraction, multiplication, and division can be applied on `F` objects that represents numeric columns. These operations will result in complex `F` objects being created, which themselves could be used in similar mathematical expressions, allowing for arbitrary complexities. Additionally, the `EXTRACT` function can be used to derive the YEAR/MONTH/DAY component of a date column.

Example A.2. In the following excerpt, we are projecting the `L_SUPPKEY`, and `L_ORDERKEY` columns of the `LINEITEM` table. However, we rename the `L_ORDERKEY` column to `L_ORDERNUM`. Further, we are also computing two new columns. The first one is `VOLUME`, that represents the gross discounted revenue of the item, computed by using the values from its `L_EXTENDEDPRICE` and `L_DISCOUNT` columns through the `F` objects. The expression is given the column name `VOLUME` by pairing the expression and the intended column name as a key-value pair. The second derived column is the shipping year of the item, `L_YEAR`, which we use the `EXTRACT` function to compute from the `L_SHIPDATE` column.

```
lineitem.project(('l_suppkey'
                  ,{'l_orderkey':'l_ordernum'}
                  ,{F('l_extendedprice')*(1-F('l_discount')):'volume'}
                  ,{EXTRACT('l_shipdate',EXTRACT.OP.YEAR):'l_year'}
                 ))
```

Projection also facilitates the standard `SUBSTRING` operation over character columns.

```
SUBSTRING(col, start, length)
```

Where `col` is the character column for which the `SUBSTRING` operation is to be performed, `start` is the starting position and *length* is an optional argument to indicate the length of the strength to be included.

Example A.3. In the following excerpt, we project the C_CUSTKEY and C_ACCTBAL columns from the CUSTOMER table and additionally use the SUBSTRING operation on the C_PHONE column to derive the country code (first two digits) of the telephone number associated with each customer, naming this derived column CNTRYCODE.

```
customer.project(('c_custkey', 'c_acctbal'
                 ,{SUBSTRING('c_phone', 1, 2):'cntrycode'}
                 ))
```

Finally, in projection, one can also write complex conditional expressions that utilizes the CASE statement to generate a column's output. Its functionality is similar to the SQL construct with the same name and other standard programming language constructs like the switch, that have a similar purpose. The CASE expression is passed a list of "pairs" where the first item in the pair is a Q object that represents a predicate and the second item is the value to be used if the predicate check is passed. If a constant literal is to be used as the value, it can be specified as-is. On the other hand, if a column's value needs to be used as the value, then it must be wrapped using an F object. The CASE expression can also be provided with a default value to be assigned in a scenario where none of the condition checks succeeds. In the absence of a default value, a NULL value is assigned. The Q objects are constructed following the same syntactic and semantic notations that we discussed in Section A.1.1. The output of the CASE statement is wrapped as a key-value pair, where the key is the CASE expression and the value is the name of the output column produced by the CASE statement.

```
{CASE( ((Q1, val1), (Q2, val2), ...), default): 'outputcolname'}
```

Example A.4. In the following excerpt we use the CASE expression to generate an indicator column, HO, for each record in the ORDERS table to tag high priority orders.

```
orders.project(('o_orderkey', 'o_orderdate', 'o_totalprice'
 ,{CASE(((Q('o_orderpriority',('1-URGENT','2-HIGH'),CMP.IN),1) ,), C(0)):'ho'}
))
```

The HO column is set to 1 if the O_ORDERPRIORITY column's value is either 1-URGENT or 2-HIGH, otherwise it is assigned the default value of 0.

A.1.3 Aggregation

Aggregation operations (equivalent of the SQL `GROUP BY` syntax) is performed by using the `agg` API call. The `agg` API takes two lists as its arguments. The first list contains the columns that are to be included in the output of the aggregation and the second list contains the columns that are to be used for grouping/categorizing. If the aggregation is to be performed over the entire data set without any grouping/categorizing, then the second list can be omitted.

```
tabularDataObject.agg((col1, col2, ...), (colg1, colg2, ...))
```

Here `col1`, etc., can be the regular columns of the source TabularData object or an aggregation over a source column. `colg1`, etc., that is used for grouping, must be the names of the source columns. Aggregations must be specified as key-value pairs, where the key is an aggregation function and the value is the column name to be assigned to the ouput of the aggregation function. The aggregation functions accept the name of a source column as its input. The `agg` API presently supports `MAX`, `MIN`, `SUM`, `AVG`, and `COUNT` aggregation functions. A special case for `COUNT` aggregation function is that, one can pass an asterisk, `'*'` as its argument to compute the number of records in a particular group.

Example A.5. In the following excerpt, we aggregate the `LINEITEM` table based on the return flag and the line status associated with each item. The grouping is performed over the `L_RETURNFLAG` and `L_LINESTATUS` columns, which are also included in the output.

```
lineitem.agg(('l_returnflag', 'l_linestatus'
              ,{SUM('l_quantity'): 'sum_qty'}
              ,{COUNT('*'): 'order_count'})
             ,('l_returnflag', 'l_linestatus'))
```

The `SUM` aggregation function is used to compute the total quantity of items (naming it `SUM_QTY`) and the `COUNT` aggregation function is used to count the number of records in each grouping (named as `ORDER_COUNT`).

Example A.6. In the following excerpt, we are computing the total price across all the items in the `LINEITEM` table by using the `SUM` aggregation function to add up its `L_EXTENDEDPRICE` column. Since the aggregation is performed over the entire data set without any grouping, we have omitted the list of grouping columns to the `agg` API.

```
lineitem.agg(({SUM('l_extendedprice'):'tot_price'},))
```

The COUNT aggregation function also accepts an optional argument to indicate if it should count the distinct values ignoring duplicates. By default, the distinct operation is not applied.

```
tabularDataObject.agg(({COUNT(col1, distinct=False):'opcol1name'}, ))
```

Example A.7. In the following excerpt, for each market segment, we count the number of countries to which the customers belong.

```
customer.agg(('c_mktsegment', {COUNT('n_nationkey', distinct=True): 'n_count'})
            ,('c_mktsegment',))
```

In order to compute this, we first group the customer records over the C_MKTSEGMENT column of the CUSTOMER table and then use the COUNT aggregation function to count only the distinct values associated with the N_NATIONKEY column, giving the output column the name N_COUNT.

Data Set Summary Through Describe

As a convenient way to take a high-level look at the distribution of values in the various attributes in a data set, TabularData objects support the describe API call that performs automatic aggregations and provides a statistical summary of each attribute in the data set.

```
tabularDataObject.describe()
```

Example A.8. In the following excerpt, we use the describe API to take a high-level look at the distribution of values in the CUSTOMER table.

```
customer.describe()
```

The output produced at the client side has the following metrics in its summary (some columns of the source data set are truncated for brevity).

```
            c_custkey    c_mktsegment       c_acctbal
count    [  150000.00]   [  150000.00]   [  150000.00]
unique   [  150000.00]   [       5.00]   [  140187.00]
nulls    [       0.00]   [       0.00]   [       0.00]
max      [  150000.00]   [  MACHINERY]   [    9999.99]
min      [       1.00]   [ AUTOMOBILE]   [    -999.99]
avg      [   75000.50]   [            ]   [    4495.51]
median   [   75000.00]   [            ]   [    4477.17]
25%      [   37501.00]   [            ]   [    1757.63]
50%      [   75000.00]   [            ]   [    4477.17]
75%      [  112500.00]   [            ]   [    7246.31]
stddev   [   43301.27]   [            ]   [    3174.31]
```

As can be seen, non-numeric data types, such as character columns like C_MKTSEGMENT, do not contribute to some of the metrics such as avg, etc., which are relevant only for numeric data.

A.1.4 Join

TabularData objects provide support for the relational *join* operation through the `join` API. The `join` API call takes as its arguments the other TabularData object to be joined with, the type of the join, the columns to be used in the join, and the list of columns to be included from either of the tables in the resulting TabularData object.

```
tabularDataObject1.join(tabularDataObject2, joinCols1, joinCols2
                        ,tbl1Cols, tbl2Cols, joinType)
```

Where `joinCols1` and `joinCols2` are the list of columns to be used from the respective TabularData objects to build the join condition. `tbl1Cols` is the list of columns from the first TabularData object that should be included in the output, whereas `tbl2Cols` is the list of columns from the second TabularData object. The list of columns can also be replaced by two special literals, `NONE` or `ALL`, to indicate if none of the (or all of the, resp.) columns from the corresponding TabularData object must be included in the output of the join. The `joinType` can be either of `INNER`, `OUTER`, `LEFT`, `RIGHT`, or `CROSS` join types, with the same semantic meaning as the SQL counter parts. The default join type is `INNER` if the join type argument is omitted. [DF13] provides an in-depth coverage on the various types of joins that are available in SQL with detailed examples.

Example A.9. In the excerpt below, we perform an `INNER` join between the `CUSTOMER` table and the `ORDERS` table over their customer key columns. The columns of the resulting TabularData object includes only the customer name from the `CUSTOMER` table, but has all the columns from the `ORDERS` table. As this is an inner join, we have omitted the join type.

```
customer.join(orders, ('c_custkey',),('o_custkey',) ,('c_name',), COL.ALL)
```

Below, in a slightly modified version of the above code, we perform a left outer join between the `CUSTOMER` and `ORDERS` tables. As it is a left outer join, all the customer records will make it to the output, irrespective of whether they have any matching orders or not.

```
customer.join(orders, ('c_custkey',),('o_custkey',) ,COL.ALL, COL.ALL, JOIN.LEFT)
```

A.1.5 Duplicates Elimination and Ordering of Data Sets

In addition to the standard relational operators, TabularData objects also provides some additional functionality to match some of the convenient features that are provided by SQL implementations. The first of this is the `distinct` API call which creates a new TabularData object in which all the rows are unique and is useful for eliminating duplicate rows from the source TabularData object.

```
tabularDataObject.distinct()
```

Finally, one can also perform ordering of data sets, by making use of the `order` API call. The `order` API accepts a list of source columns as its argument.

```
tabularDataObject.order(orderlist)
```

The default ordering is ascending. However, the ordering imposed on a column can be changed explicitly to descending by suffixing the column names with a `'#desc'` tag.

```
tabularDataObject.order(('col1', 'col2#desc', ...))
```

Example A.10. In the following source code listing, we first project the `L_ORDERKEY`, `L_SHIPDATE`, and `L_SHIPMODE` columns from the `LINEITEM` table, and then select only the distinct rows, ordering them first by `L_ORDERKEY` ascending, and then by `L_SHIPDATE` descending.

```
lineitem.project(('l_orderkey','l_shipdate', 'l_shipmode'))
        .distinct()
        .order(('l_orderkey','l_shipdate#desc'));
```

The final, resulting TabularData object will therefore have three columns – `L_ORDERKEY`, `L_SHIPDATE`, and `L_SHIPMODE`. Further, when that TabularData object is materialized, its internal data representation will be ordered as described above.

A.2 Data Slicing and Stacking

As TabularData objects represent two-dimensional data sets, they support positional access to individual data elements and transformations that are conventionally expected out of a *matrix* data structure. Therefore, we can *slice* specific rows and columns out of a TabularData object to create a new TabularData object with a subset of columns and rows from the source TabularData object (such as when one would need to separate out the test and training data sets while constructing a typical machine learning model). On the other hand, we can also *stack* multiple TabularData objects either horizontally or vertically, to add more columns or rows, respectively.

The slicing syntax followed by TabularData objects is based on those of NumPy arrays and has the following general syntax.

```
tabularDataObject[rowslice, colslice]
```

Here `rowslice` decides which rows will make it to the result and `colslice` indicates the columns that must be included in the resulting TabularData object.

The slices have the following format.

```
start:end:skip
```

Where `start` is the starting index for the operator (e.g., row 5), and `end` is the ending index for the operator (e.g., row 15). `skip` is used to denote the number of positions to skip between every selected index (e.g., skip 2 rows). The starting index is inclusive, whereas the ending index is excluded. While the columns to be sliced can be indicated using their numeric positions or column names, rows of interest must be always expressed through numeric positions. Table A.5 shows some examples of how to perform slicing on TabularData objects.

Further, we can find the dimensions (number of rows and columns) of a TabularData object by accessing its `shape` property.

```
tabularDataObject.shape
```

The shape property provides a Python tuple of the form:

```
(numRows, numColumns)
```

TabularData objects also allow themselves to be stacked horizontally (increasing the number of columns) or vertically (increasing the number of rows). In order for two TabularData objects to be stacked horizontally, they should have the same number of rows. Similarly, to stack two TabularData objects vertically, they should have the same number of columns and must be of compatible data types. The horizontal stacking and vertical stacking are supported through the `hstack` and `vstack`, respectively API calls of the TabularData object that accepts a list of TabularData objects as its argument.

```
tabularDataObject.hstack((tabularDataObject1, tabularDataObject2, ...), colprfxes)
tabularDataObject.vstack((tabularDataObject1, tabularDataObject2, ...))
```

`hstack` optionally takes a `colprfxes` argument, which is a list of string literals. The prefixes are used by the transformation to rename a source's TabularData object's column name by prefixing it with the corresponding string literal. This approach can be used to address issues where a given column name appears in more than one TabularData object that is being stacked.

Table A.5: Slicing TabularData objects

Syntax	Description
`customer[0]`	Creates a new TabularData object with only the first row in the in the source TabularData object. All the source columns are included in the new TabularData object.
`customer[5:]`	Creates a new TabularData object that contains every row, starting from the sixth row in the source TabularData object. All the source columns are included in the new TabularData object.
`customer[5:9]`	Creates a new TabularData object that contains rows six through nine in the source TabularData object. All the source columns are included in the new TabularData object.
`customer[::2]`	Creates a new TabularData object that contains every second row, starting from the first in the source TabularData object. All the source columns are included in the new TabularData object.
`customer[5::2]`	Creates a new TabularData object that contains every second row, starting from row six in the source TabularData object. All the source columns are included in the new TabularData object.
`customer[:,0]`	Creates a new TabularData object with only the first column from the source TabularData object. All the rows from the source is included in the new TabularData object.
`customer[:, ['n_nationkey']]`	Creates a new TabularData object with only the `N_NATIONKEY` column from the source TabularData object. All the rows from the source is included in the new TabularData object.
`customer[:, 1:5]`	Creates a new TabularData object with only the second through fifth columns from the source TabularData object. All the rows from the source is included in the new TabularData object.
`customer[:, 1:5:2]`	Creates a new TabularData object that contains every second column, starting with the second column until the fifth, from the source TabularData object. All the rows from the source is included in the new TabularData object.
`customer[0:10, 0:3]`	Creates a new TabularData object that contains the first three columns of the first ten records in the source TabularData object.

Example A.11. In the source code listing below, we first create two TabularData objects, `c1` and `c2` by using the slicing operations on the `CUSTOMER` table. The TabularData object `c1` contains only the columns `C_CUSTKEY` and `C_NATIONKEY`, whereas `c2` contains only the `C_ACCTBAL` column. Next, we create a new TabularData object, `c3`, using `hstack` on `c1` and `c2`. We also pass the column prefixes `'c1_'` and `'c2_'` as arguments to `hstack`.

```
c1 = customer[:, ['c_custkey', 'c_nationkey']]
c2 = customer[:, ['c_acctbal']]
c3 = c1.hstack((c2,), ('c1', 'c2'))
```

The resulting TabularData object referenced by `t3` will have three columns from the corresponding source TabularData objects, but will be renamed to `C1C_CUSTKEY`, `C1C_NATIONKEY`, and `C2C_ACCTBAL` respectively.

A.3 Numerical Computations and Linear Algebra

All the columns in a TabularData object must be of a numeric type in order to perform any numerical computations over them. TabularData objects support the basic arithmetic operations – *addition*, *subtraction*, *multiplication*, *division*. In general they have the following form.

```
operand1 oper operand2
```

As these are binary operators, one of the operands can be a scalar instead of being a TabularData object. If one of the operands is a scalar, then the numeric operation is performed between each element in the TabularData object and the scalar operand. On the other hand, if both the operands are TabularData objects then they must have the same dimensions (number of rows and columns) and the arithmetic operation is applied between their corresponding elements in the same position (row and column). These operators are shown in Table A.6.

TabularData objects also support the matrix transformations of *matrix multiplication* and *matrix transpose*. For matrix multiplication, both the operands must be TabularData objects. Further, similar to standard matrix multiplication, the number of columns in the first TabularData object must be the same as the number of rows in the second TabularData object. The resulting TabularData object will have the same number of rows as the first TabularData object and the same number of columns as the second TabularData object. When a matrix-transpose is performed on a TabularData object, a new TabularData object is created where the rows of the source TabularData object becomes its columns and the source rows becomes its columns.

Table A.6: Arithmetic operators for TabularData objects

Operator	**Description** and **Example**

`+`	Numeric addition.
	`tabularDataObject + operand`
	`operand + tabularDataObject`

`-`	Numeric subtraction.
	`tabularDataObject - operand`
	`operand - tabularDataObject`

`*`	Numeric multiplication.
	`tabularDataObject * operand`
	`operand * tabularDataObject`

`/`	Numeric division.
	`tabularDataObject / operand`
	`operand / tabularDataObject`

Table A.7: Matrix for TabularData objects

Operator	**Description** and **Example**
`@`	Perfroms a matrix multiplication. `tabularDataObject1 @ tabularDataObject2` If `tabularDataObject1` has m rows and n columns and `tabularDataObject2` has n rows and p columns, the resulting TabularData object will have m rows and p columns.
`T`	Matrix transpose. `tabularDataObject.T` If `tabularDataObject` has m rows and n columns then the the resulting TabularData object will have n rows and m columns.

A.4 APIs for Custom Extensions

A.4.1 Custom Transformations

A custom transform, as was discussed in Section 4.1.1, allows data scientists to perform transformations on TabularData objects that are not possible with the built-in APIs. It also allows them to integrate third party packages into AIDA's framework. As we had discussed the API and usage of custom transformations in detail in Section 4.1.1, we will only discuss it briefly in this section for the sake of completeness.

A custom transformation function is basically a Python function that accepts a TabularData object as its first argument. It may accept any additional arguments other than this, including other TabularData objects. The custom transformation function can access the internal data representation of the TabularData object passed as its argument and do any data processing over it, employing other Python packages if required.

```
def ctransformFunc(tabularDataObject, arg1, arg2, ...):
  # Import any Python packages required.
  import ...
  # Access the internal data representation of tabularDataObject.
  ... = tabularDataObject.cdata
  # Do any necessary processing to produce a new data set.
  ...
  # Return an internal data representation for the new TabularData object.
  return ...
```

The transformation function itself is executed using the `_U` operator that is associated with the TabularData object. The transformation function and necessary arguments are automatically shipped to the server by AIDA's client API and is executed there. AIDA's server uses the internal data representation provided by the transformation function to instantiate a new TabularData object, returning its reference to the client API.

```
tabularDataObject._U(ctransformFunc, arg1, arg2, ...)
```

Several detailed examples on custom transformations are available in Section 4.1.1.

A.4.2 Loading External Data

As we saw in Section 4.1.2, external data sets can be integrated into AIDA's workflow by writing Python functions that read these data sets into one of the internal data formats of TabularData. They

can also leverage other Python packages that are useful to execute any data processing or computational tasks that they need to perform. These functions can accept any arbitrary arguments, and as such they are syntactically similar to the functions that are used with custom transformations. Their general logical structure is as shown below.

```
def dataLoadFunc(arg1, arg2, ...):
  # Import any Python packages required.
  import ...
  # Load the external data set into Python space.
  ...
  # Do any necessary processing to convert it into TabularData's format.
  ...
  # Return an internal data representation for the new TabularData object.
  return ...
```

They are executed using the `_L` operator associated with the database workspace object.

```
db._L(dataLoadFunc, arg1, arg2, ...)
```

As the load function (and any arguments passed to it) is shipped to and executed at the server side, there is no overhead associated with the external data being transferred through the client (unless the external data set resides in the client). The data set returned by the load function is used to instantiate a new TabularData object, whose reference is then returned to the client API.

Example A.12. In the following source code listing, we use Wikipedia to extract information about the world population at country level by scraping the relevant Wikipedia web page [Mit18].

```
def CountryPopulation():
  import wikipedia, bs4
  page = wikipedia.page('countries population').html() # Lookup the relevant web page.
  soup = bs4.BeautifulSoup(page,'html5lib')
  country=[]; pop=[];
  for items in soup.find('table', class_='wikitable') \ # Find the HTML table.
                   .find_all('tr')[1::1]:
    data = items.find_all(['th','td']) # Parse the HTML table for data items.
    country.append(data[1].a.text)
    pop.append(int(data[2].contents[0].replace(',','')))
  return {'country':country, 'population':pop} # Return dictionary-columnar data.

cntryInfo = db._L(CountryPopulation)
```

We make use of existing specialized Python packages BeautifulSoup [Nai14], and wikipedia, to do the weightlifting such as parsing the web page for the relevant data. We write a load

function that basically "wraps" the functionality provided by these packages and produces a data structure that conforms to TabularData's internal representation.

After we execute the load function, `cntryInfo` references a TabularData object that contains two columns; `COUNTRY`, which is used to store the names of the countries, and `POPULATION` which is used to store their populations.

A.4.3 The Remote Execution Operator API

As discussed in Section 4.1.3, the *remote execution operator* is a very useful mechanism to execute arbitrary Python logic at the server side. However, their main objective is to shift iterative computations over TabularData objects to the server side in order to reduce the RMI overhead. Further, unlike the custom transformations and data load functions, they are not required to produce any data set at the end of their execution and therefore, the execution of remote execution operator does not necessarily result in a new TabularData object (unless the user-written function internally creates new TabularData objects as part of its execution logic).

Users can write their processing logic as a regular Python function. The function must accept a database workspace object as its first argument, but can have any other additional arguments. These arguments can be regular Python data structures, TabularData objects, or even other Python functions. Although these functions are not required to return anything at the end of their execution, if required, they may do so, and the return values are automatically sent back to the client at the end of their execution. These return values can be regular Python objects or even TabularData objects. If a TabularData object is returned, then only the remote reference is sent to the client, which will then construct a stub to interact with the server-resident TabularData object, as is the case with any other built-in TabularData transformations in AIDA.

```
def remoteFunc(db, arg1, arg2, ...):
  # Import any Python packages required.
  import ...
  # Execute the processing logic.
  ...
  # Optionally return something (will be sent back to the client).
  return ...
```

A function meant for remote execution can be invoked through the `_X` operator associated with the database workspace object. The function is then shipped along with any necessary arguments and executed at the server side. If the function has a return value, it will be sent back to the client. Otherwise, a default return value of `None` is assigned.

```
# res will be None, if remoteFunc does not return any value.
res = db._X(remoteFunc, arg1, arg2, ...)
```

Example A.13. In this example, we will see how we can make use of the remote execution operator to build a *linear regression* model by performing *gradient descent* algorithm iteratively. For those not familiar with the approach, [JWHT17] and [PG17] provides a detailed discussion on the fundamentals with some examples. First, we define two Python functions, `gradDesc` that implements the standard gradient descent algorithm, and `squaredErr` that computes the mean squared error. We store these two functions in the database workspace.

```
def gradDesc(actual, predicted, trainDataSet, numSamples):
    return (predicted-actual).T @ trainDataSet / numSamples

def squaredErr(actual, predicted, numSamples):
    return ((predicted-actual)**2).sum()/(2*numSamples)

db.squaredErr = squaredErr; db.gradDesc = gradDesc;
```

Next, we define a Python function, `trainModel`, to perform the model training for linear regression. The function accepts the following parameters – a training data set, the actual values of the dependent variable for the training data set samples, an initial set of *model parameters*, all of which are TabularData objects. It also takes as arguments the number of samples in the training data set, the number of training iterations to perform, and the *learning rate* hyperparameter, `alpha`.

```
def trainModel(db, trainDataSet, trainDataSetActual, params
               ,numSamples, numiters, alpha):
    for i in range(0, numiters):
        pred = trainDataSet @ params.T
        params = params - alpha*db.gradDesc(trainDataSetActual, pred
                                            ,trainDataSet, numSamples)
    return (db.squaredErr(trainDataSetActual, pred, numSamples), params)
```

The training function executes the gradient descent function (accessed through the database workspace) for the specified number of iterations, updating the model parameters with each iteration. At the end of the specified number of iterations, it returns a tuple containing both the squared error and the current set of model parameters. Since the function, and there by all the iterations, are executed at the server space without the client involvement, there is no RMI involved, thus eliminating its overhead.

The training function is executed by passing the necessary arguments to it, using the `_X`

operator. In the following excerpt, we execute the training function to perform 100 iterations.

```
(err, params) = db._X(trainModel, trainDataSet, trainDataSetActual
                      ,initialParams, numSamples, 100, 0.1)
```

At the end of its execution, we can analyze the squared error value returned by the training function. If we are not satisfied with the training, we can continue executing the training function, this time starting from the previously returned set of model parameters.

```
(err, params) = db._X(trainModel, trainDataSet, trainDataSetActual
                      ,params, numSamples, 100, 0.1)
```

A.5 Data Visualization

A.5.1 Using matplotlib for Data Visualization

As discussed in Section 4.2, data scientists can use matplotlib to perform static data visualization. As matplotlib is a very complex and sophisticated package, our discussion here is only meant to demonstrate how to execute matplotlib code using AIDA's API. For those not familiar with matplotlib, there are several sources such as a [YCY18], and [Kel18] that provides an in-depth coverage on the topic.

As data is resident at the server, AIDA creates the image object at the server and then ships it to the client side to visualize it. To accomplish this, the data scientist have to write the matplotlib code as a Python function. The function should accept the database workspace object as the first argument, in addition to any other arguments that it needs. It must return a matplotlib figure object, that contains the plotting logic incorporated.

```
def matPlot(db, arg1, arg2, ...):
    # Access and process any data sets as required.
    # Create a matplotlib figure object.
    fig = plt.figure()
    # Include the plotting logic into the figure object.
    ...
    # Return the figure object.
    return fig
```

The plotting function can be then executed using the `_Plot` API associated with the database workspace object. Similar to the user extensions that we covered in the previous section, the

plotting function and the arguments are shipped to and executed at the server space. The `_Plot` operator returns an image to the client side that represents the visualization logic incorporated by the figure object returned by the plotting function.

```
img = db._X(matPlot, arg1, arg2, ...)
```

The image can be displayed/saved into a file at the client side. With Jupyter Notbook environments, AIDA's client API provides a wrapper (`show`) to display it using Jupyter Notebooks visualization APIs.

```
show(img)
```

Example A.14. In this example that is based on the TPC-H `PART` table, we develop a function to generate a matplotlib bar chart that shows the minimum retail price of a particular type of part, offered by various manufacturers. Our plotting function accepts the part type as an argument, allowing us to use this function to plot charts for any part type. Within the function, we first access the `PART` table through the database workspace object and perform relational operations using TabularData object's relational API. This creates a TabularData object that captures the minimum retail price offered by each manufacturer for the particular part. We then use the dictionary-columnar internal data representation of this TabularData object and use it construct a bar chart figure in matplotlib, returning the figure at the end of the function execution.

```
def partsPriceBar(db, partType):
    from aida.aida import Q,C,MIN; import numpy as np;
    # Find minimum retail prices.
    pinfo = db.part.filter(Q('p_type', C(partType)),Q('p_size', C(1)))\
                   .agg(('p_mfgr',{MIN('p_retailprice'):'min_rprice'}),\
                        ('p_mfgr',))
    pdata  = pinfo.cdata # Access the dictonary-columnar internal data structure
    # Plot the data into a matplotlib figure object.
    fig    = plt.figure()
    ax     = fig.add_subplot(111)
    ydata  = pdata['min_rprice']; xdata = pdata['p_mfgr'];
    xticks = np.arange(0, len(ydata))+0.10
    ax.bar(xticks, ydata, color='b')
    ax.set(title='Min. Retail Pricing for ' + partType)
    ax.set_xticklabels(xdata); plt.xticks(rotation=15)
    plt.ylabel('price ($)')
    return fig; # Return the figure object.
```

We can now plot this chart by executing it using the `_Plot` operator, passing the part type we are interested in as the argument.

```
img=db._Plot(partsPriceBar, 'LARGE BRUSHED BRASS')
show(img)
```

Figure A.1 shows the chart produced by the function, as displayed in a Jupyter Notebook.

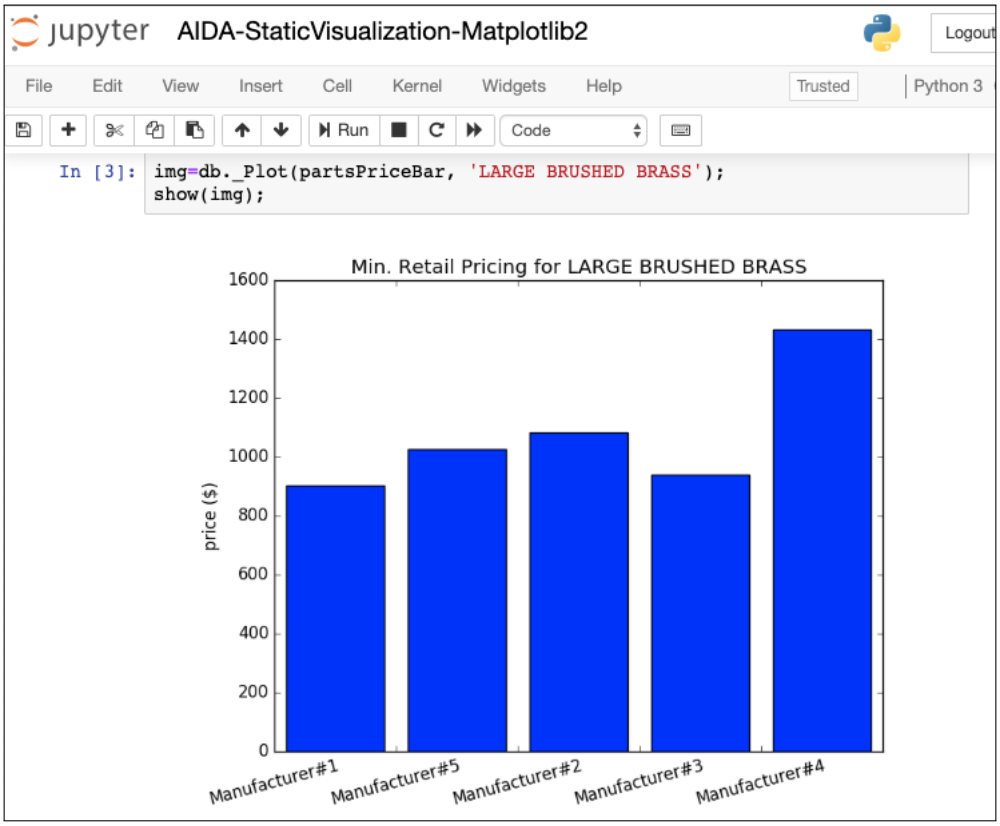


Figure A.1: AIDA bar chart using matplotlib.

A.5.2 Using plotly for Interactive Data Visualization

plotly is an advanced data visualization environment that allows data scientists who are expert programmers to build sophisticated data science workflows. The details of writing visualization applications using plotly can be found in its online documentation[1]. Data scientists can utilize plotly to create interactive visual interfaces in AIDA by constructing the interface inside a Python function. AIDA expects the Python function to accept the database workspace object as the first argument and a plotly app object as the second argument followed by any additional arguments

[1] https://dash.plot.ly

that the function requires. The plotting function is expected to return a layout object that contains all the visualization elements including any charts.

```
def plotlyPlot(db, app, arg1, arg2, ...):
    # Access and process any data sets as required.
    # Create a plotly visual interface layout object.
    layout = ...
    # Include the visual elements into the layout object.
    ...
    # Return the layout object.
    return layout
```

The plotting function is executed using the `_Page` operator of the database workspace object.

```
page = db._Page(plotlyPlot, arg1, arg2, ...)
```

As the operator returns a URL of the web page that hosts the visualization, it can be accessed using a regular web browser, or in the Jupyter Notebook as follows.

```
show(page)
```

Example A.15. In this example, we re-implement the matplotib based plotting function from Example A.14 using plotly. Here, we create an interface that allows us to choose the part type that we are interested in through a drop-down menu, and the chart will automatically fetch the relevant data from the database and update itself.

```
1  def partsPriceExplorer(db, app):
2    from aida.aida import MIN, Q, C
3    import dash_core_components as dcc, dash_html_components as html
4    from dash.dependencies import Input, Output
5
6    #Find all available part types to create a drop down.
7    pTypes = db.part.project(('p_type',)).distinct().cdata['p_type']
8    layout = html.Div([
9       html.Div(id=db.genDivId('pInfo')) # The price bar graph.
10     ,dcc.Dropdown(id=db.genDivId('pTypes') # Part types dropdown.
11                  ,options=[{'label':p, 'value':p} for p in pTypes]
12                  ,value=pTypes[0])])
13
14   # Callback event to update the bar graph.
15   @app.callback(Output(db.getDivId('pInfo'), 'children')
16               ,[Input(db.getDivId('pTypes'), 'value')])
17   def updateGraph(selPType):
18     # Fetch the data corresponding to the selected part type.
19     db.pinfo = db.part.filter(Q('p_type', C(selPType)),Q('p_size', 1))\
20                       .agg(('p_mfgr', {MIN('p_retailprice'):'min_rprice'})
21                           ,('p_mfgr',))
22     pdata = db.pinfo.cdata
23     # Update the bar graph with a new plot figure.
24     fig = {'data' : [{'x':pdata['p_mfgr'],'y':pdata['min_rprice'],'type':'bar'}]
25           ,'layout' : {'title':'Min. Retail Pricing for ' + selPType} })
26     return dcc.Graph(figure=fig)
27
28   return layout
```

The interface is then activated using the `_Page` operator.

```
pltp = db._Page(partsPriceExplorer)
show(pltp)
```

Figure A.2 shows the visual interface created by this plotting function as displayed in a Jupyter Notebook. Inside the plotting function, we create a drop-down menu and populate with all the available part types, by retrieving them from the PART table (line 7). When a part type is selected using the drop-down, the callback function gets triggered (line 17). The callback

function then retrieves the minimum retail price of the part that is currently selected, and updates the bar chart It also saves the TabularData object corresponding to this data set in the database workspace as `pinfo` (line 19). This makes the data set accessible to any other transformation function or the client API for any further analysis.

```
print(db.pinfo.cdata);
```

A.5 Data Visualization

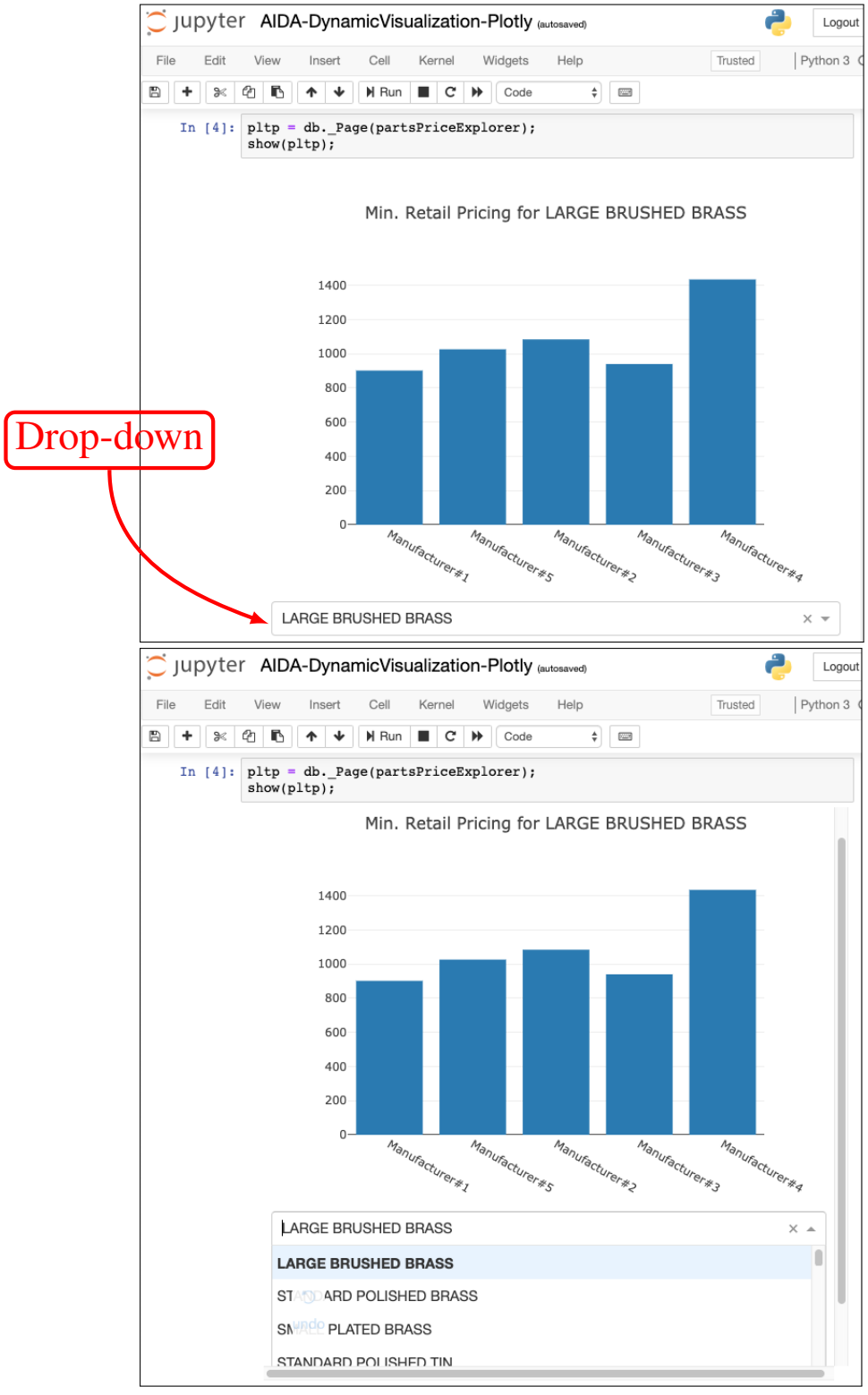


Figure A.2: AIDA interactive data visualization using plotly in Jupyter Notebook.