Data Transfer Between PostgreSQL and its Embedded Python Environment for In-Database Analytics

Jianhao Cao

School of Computer Science

McGill University, Montreal

August, 2021

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Master of Science

© Jianhao Cao, 2021

Abstract

Data science workflows generally comprise a combination of relational operations and numerical computations. Since traditional Relational Database Management Systems (RDBMSes) lack proper support for linear algebra operations, data scientists have resorted to database-external analytical solutions. While some high-level language (HLL) statistical libraries provide relational operation support, their functionalities are not comparable to the database engine. As many RDBMSes have integrated HLL interpreters into their implementations, it opens the possibility of an in-database analytics solution that can leverage RDBMS's database engine for relational operations and utilize statistical libraries to perform linear algebra computations in the embedded HLL environment. This thesis studies the implications of implementing such a system for PostgreSQL, a row-oriented RDBMS, from the perspective of data movement between the RDBMS and its embedded Python environment. We extend AIDA, a data science framework prototype for Advanced In-Database Analytics, to support in-database analytics in PostgreSQL. The row-based PostgreSQL has different internal data representations from the vector-based data structures that are used in AIDA and Python statistical packages for linear algebra operations. The challenge is that AIDA has to enforce an expensive data-structure transformation when loading SQL result sets into vector-based Python objects or when exposing the Python objects to PostgreSQL to perform SQL queries over them. To alleviate such transformation overhead and facilitate in-database analytics, we look into the optimization probabilities of transforming

PostgreSQL data into Python data representations and vice versa. Instead of running SQL queries over the user defined functions that return Python data, we develop a new optimizer-friendly data exposing mechanism based on PostgreSQL's foreign data wrapper feature and a second approach that loads Python objects into temporary tables for PostgreSQL to read.

Abrégé

Les flux de travail en science des données se composent généralement d'une combinaison d'opérations relationnelles et de calculs numériques. Étant donné que les systèmes de gestion de bases de données relationnelles (SGBDR) traditionnels ne considèrent pas correctement les opérations d'algèbre linéaire, les experts ont recours à des solutions analytiques externes aux bases de données. Bien que certaines bibliothèques statistiques de langage de haut niveau (HLL) soutiennent des opérations relationnelles, leurs fonctionnalités ne sont pas comparables à celles du moteur de base de données. Comme de nombreux SGBDR ont intégré des interpréteurs HLL dans leurs implémentations, il est désormais possible d'avoir une solution analytique dans la base de données qui peut exploiter le moteur de base de données du SGBDR pour les opérations relationnelles. Cette solution peut également utiliser des bibliothèques statistiques pour effectuer des calculs d'algèbre linéaire dans l'environnement HLL embarqué. Cette thèse étudie les implications de la mise en œuvre d'un tel système pour PostgreSQL, un SGBDR orienté lignes, du point de vue du mouvement des données entre le SGBDR et son environnement Python embarqué. Nous étendons AIDA, un prototype de cadre de science des données pour l'analyse avancée dans la base de données, pour soutenir la capacité d'analyse dans la base de données dans PostgreSQL. Le PostgreSQL basé sur des lignes a des représentations de données internes qui sont différentes des structures de données vectorielles utilisées dans les modules statistiques AIDA et Python pour les opérations d'algèbre linéaire. Le défi est qu'AIDA doit imposer

une transformation coûteuse de la structure des données lors du chargement des ensembles de résultats SQL dans des objets Python vectoriels ou lors de l'exposition des objets Python à PostgreSQL pour effectuer des requêtes SQL sur eux. Afin d'alléger cette surcharge de transformation et faciliter l'analyse dans la base de données, nous évaluons les probabilités d'optimisation de la transformation des données PostgreSQL en représentations de données Python et vice-versa. Au lieu d'exécuter des requêtes SQL sur les fonctions définies par l'utilisateur qui renvoient des données Python, nous développons un nouveau mécanisme d'exposition de données, convivial pour les optimiseurs, basé sur la fonctionnalité de wrapper de données étrangères de PostgreSQL. Nous proposons en outre une deuxième approche qui charge des objets Python dans des tables temporaires pour que PostgreSQL puisse ensuite les lire.

Acknowledgements

Foremost, I would like to express my deep gratitude to Prof. Bettina Kemme, a truly caring and thoughtful supervisor, for all of her meticulous guidance and continuous encouragement. Also, I want to thank Dr. Joseph D'Silva. Not only did he inspire me to conduct the research, but he had also been giving me precious and practical feedback that helped me finish the work with his immense knowledge in this research field. I feel fortunate to have been guided by these two supportive mentors throughout my master's studies. This work could not have been done without their kind help and support.

Secondly, I want to thank Sean MacRae for helping me translate the abstract into French. I am also very grateful to all my companions throughout this journey. I especially want to express my appreciation to my fellow members from the Distributed Information System Lab for their valuable thoughts and suggestions. And I would also like to give special thanks to my friends who encouraged me during the rough time.

Lastly, I want to thank my family for their emotional support and unfailing love. My appreciation for them goes beyond words.

Table of Contents

List of Figures

List of Listings

Chapter 1

Introduction

Data analysis plays a pivotal role when it comes to decision-making in many fields. The data-driven analytical requests in such work typically comprise a combination of relational operations and numerical computations. Relational database management systems (RDBMSes) are widely used for data storage and have a vigorous database engine to execute relational queries. However, they lack user-friendly programming paradigms that allow users to perform numerical and linear algebra computations on the data stored inside them. Users have to retrieve the data from the RDBMS and resort to database-external solutions for analytical workflows. Although some high-level language statistical libraries such as pandas [24] and data science frameworks such as Spark [35] also provide relational operation support, their implementations are not comparable to the query optimization capability of RDBMSes. In addition, these database-external analytics solutions have a data transfer overhead as they require loading data from the RDBMS to their workspace or even shipping data across the network if they reside in a remote machine.

The in-database analytics solutions arise from the fact that many RDBMSes start to embed high-level language (HLL) environments into their systems to host in-database programming. These RDBMSes provide an interface to retrieve relational data sets from

the RDBMS to the embedded HLL environment. Such data transfer eliminates the costs of shipping data to an external system. Users can write a user defined function (UDF) with HLL code to manipulate the data and execute the UDF inside the RDBMS. Moreover, it is possible to return an HLL data set in a UDF and then run a SQL query over the UDF to perform relational operations over HLL data. Therefore, with HLL UDFs, users can leverage RDBMS's database engine to execute relational operations and utilize HLL statistical libraries for numerical and linear algebra computations. However, this implementation has one limitation: it does not allow for interactive analytics as users must pre-define operations in a UDF before executing the task.

AIDA (abstraction for Advanced In-Database Analytics) [11] was proposed to address the agility aspect of in-database solutions. It is a Python-based framework that follows a client-server setup. On the client side, AIDA provides a user interface that emulates the syntax and semantics of Python statistical packages and shifts the computation to the server component. The server component of AIDA resides inside the embedded Python environment of an RDBMS to facilitate in-database analytics. It executes linear algebra computations by using the statistical package NumPy and pushes relational operations down to RDBMS's database engine. The current implementation of AIDA in [11] is designed for the column-oriented RDBMS MonetDB. It is convenient to transfer data between MonetDB and AIDA to perform analytical operations because the internal data representation of MonetDB has structural similarities with the vector-based data structures used by the Python statistical package NumPy. However, it remains the fact that many RDBMS implementations that are optimized for relational operations still follow a row-oriented data storage model. Considering the prevalence of row-oriented RDBMSes, it would be beneficial to execute the whole analytics workflow inside a row-oriented RDBMS while leveraging its database engine to process relational operations. In this thesis, we want to analyze the implications of adapting AIDA for a row-oriented RDBMS to support in-database

analytics. Therefore, we develop an implementation for AIDA that can fully interact with PostgreSQL, our row-based RDBMS of choice, to provide in-database analytics support that allows for an interleaved execution of linear algebra computations and relational algebra operations. We use different strategies to facilitate data movement between PostgreSQL and AIDA's workspace; this includes optimizations on loading PostgreSQL data into AIDA's internal representation and on exposing Python data to PostgreSQL to perform relational operations.

To circumvent the problem that SQL queries cannot be directly performed over HLL objects, AIDA leverages UDFs to expose Python data to the underlying RDBMS to perform SQL queries. However, UDFs are perceived as a black-box by the RDBMS and leave room for optimization. The concept of Virtual Table was introduced in [12] and implemented for MonetDB, which again leverages the same data transfer optimization as in the development of AIDA. It allows registering HLL objects as virtual tables inside the database that can later be referenced in a SQL query. An advanced version of AIDA uses virtual tables to facilitate running SQL queries over Python data. Unlike UDFs, virtual tables are optimizer-friendly as they can provide the cardinality of the registered data to the query planner. Since the implementation does not align with PostgreSQL, we delve into how to transfer data from the embedded Python environment to PostgreSQL by following the same optimization logic. In this thesis, we propose an optimizer-friendly data exposing method that exploits PostgreSQL's foreign data wrapper feature to run SQL queries over Python data. This approach has different versions that vary in terms of how the data is transferred to the database engine. We also have an alternative solution that loads Python data into temporary tables for later relational querying. Through various test cases, we evaluate our implementation of AIDA with these two data exposing mechanisms.

The remainder of this thesis is organized as follows:

In Chapter 2, we present the background of this thesis. We first look at query processing in RDBMS and then introduce the row-oriented RDBMS PostgreSQL. Following that, we compare current data analytics solutions and present how the Python data science framework AIDA supports in-database analytics. Lastly, we cover several tools that allow us to transfer Python objects to PostgreSQL.

In Chapter 3, we present how we adjust AIDA's database adapter interface to facilitate in-database analytics inside PostgreSQL's embedded Python environment. To evaluate our AIDA implementation, we experiment with test cases that involve data movement between PostgreSQL and Python space and compare it with other analytical solutions.

In Chapter 4, we demonstrate two optimizer-friendly data exposing mechanisms that can facilitate running SQL queries over Python objects in PostgreSQL. To understand the performance benefits of these optimized approaches, we use the TPC-H Benchmark and an end-to-end data science workflow to evaluate them against the conventional UDF-based approach.

In Chapter 5, we summarize our findings to conclude this thesis and discuss possible directions for future work.

Chapter 2

Background & Related Work

In this chapter, we will have an overview of concepts and previous research related to our work. Section 2.1 introduces relational data querying in RDBMSes and then presents PostgreSQL, a row-oriented RDBMS that we will work on. Section 2.2 describes database-external and in-database analytics solutions and provides an overview of the AIDA framework [11]. Section 2.3 discusses the strategies and optimizations of how to expose HLL data to the RDBMS in order to run SQL queries over them for in-database analytics.

2.1 Relational Database Management Systems

Based on mathematical set theory, Edgar F. Codd [6] proposed the relational model that laid the foundation of RDBMSes. The relational model defines a relation as a set of values with different attributes in the mathematical sense, which provides a logical abstraction for data representation.

A relational database is a collection of one or multiple relations, and the system managing the relational database is referred to as a Relational Database Management System (RDBMS). Regardless of the internal data storage implementation of an RDBMS, a relation is presented as a table comprising rows and columns in the database. Each

row in the table corresponds to a data record, also called a tuple, with column values conforming to the attribute definitions in the relation schema. In order to connect the underlying database storage to the application programming interface, the RDBMS is equipped with a database engine component to process user tasks.

2.1.1 Data Querying of Relational Databases

Given the logical nature of the relational model, Edgar F. Codd defined a relational algebra with high-level operators and a relational calculus, which is essentially an applied predicate calculus [7].

SQL [3] is a declarative query language built upon relational calculus, while its design also leverages the semantics of high-level operators in relational algebra. Being a domain-specific language specialized in managing and interacting with relational databases, it was initially developed for IBM System R [2]. Nowadays, SQL is widely supported by RDBMS implementations because of the versatility and implementation-independent interpretability of its syntax.

Listing 2.1 depicts a SQL query example. This query retrieves the student names and their corresponding faculty names stored in the relational database. The FROM keyword indicates the relations/tables that are required to execute the query. In this example, the records from *students* and *faculties* with the same faculty id values are combined to form a result set in a join operation, following the comparison predicate stated in the WHERE clause. The SELECT function specifies the attributes/columns to return, and thus, it requires performing a projection on *s_name* and *f_name* of the result set.

```
1 SELECT s_name, f_name
2 FROM students, faculties
3 WHERE s_fid = f_fid;
```

Listing 2.1: A SQL query example

As shown in the example above, SQL provides a versatile syntax that enables users to describe a data querying task in terms of high-level logical operations. Because of its declarative and implementation-independent nature, SQL opens the possibility of query optimizations by looking into its semantics. It is plausible to transform a SQL query to another one that is logically equivalent in regard to the output result. Depending on what kinds of transformations are made, there are two different phases of query optimization. Some RDBMSes have a query rewrite component that uses predefined rules to transform an input SQL query into a simplified query expression [16, 28]. The query planner/optimizer, which usually follows the query rewrite component, is responsible for estimating the costs of various query execution paths and applying optimization heuristics to generate an efficient execution plan [4, 18]. Most of the time, the efficiency of query executions depends on the accuracy of cost estimations.

2.1.2 PostgreSQL - A Row-Oriented RDBMS

PostgreSQL[1] is a popular open-source row-oriented RDBMS that has well-developed query optimization functionalities [14]. The relational model, together with the SQL language, decouples data querying from the physical storage and architecture implementation of RDBMSes, giving developers the flexibility to decide how to implement an RDBMS targeting to address specific needs. Early data usage workflows mainly consisted of single-record retrieval and write-intensive requests from the domain of business transaction processing, which developed into the concept of Online Transaction Processing (OLTP). Since the contiguous row-oriented disk storage model allows for efficient query processing and disk I/O operations when the user requests follow an OLTP workflow, contemporary RDBMSes, including PostgreSQL, predominantly adopted a row-oriented architecture that uses a row as the minimal

[1] `https://www.postgresql.org/`

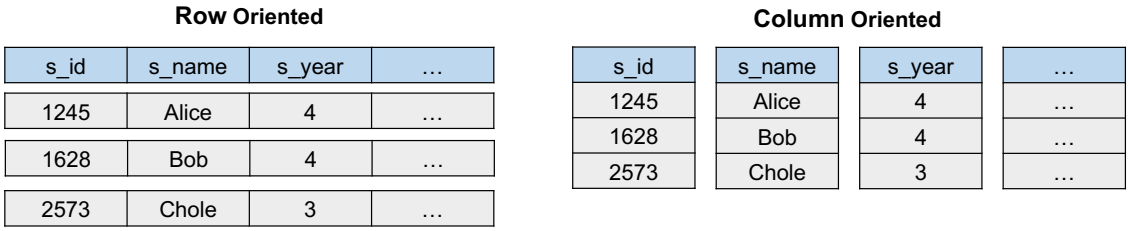

Figure 2.1: Two types of RDBMS data storage

indivisible unit data storage (Figure 2.1) in order to accommodate OLTP-style workflows.

In contrast to OLTP, Online Analytical Processing (OLAP) workflows consist of more complex queries that usually slice the data and work with a subset of columns in the relations. Row-oriented RDBMSes do not perform well on these analytical workloads as they require reading the whole records despite that some columns are unnecessary to the request. C-Store [31] and MonetDB [17] are the attempts from the database community that exploit the column-oriented storage model to implement an RDBMS. The column-oriented implementation, as shown in Figure 2.1, stores the values of each column contiguously in the disk so that the database engine can only load the columns that are relevant to the request.

Although the column-oriented implementation performs relatively well on OLAP-style workflows [15], row-oriented RDBMSes still have certain advantages in OLTP-style queries. Considering the prevalence of row-oriented RDBMSes at the current time, it is of interest to investigate how to apply data analytics facilitation in a row-oriented RDBMS such as PostgreSQL.

PostgreSQL is also known for using a ”process per user” client/server model [10], meaning that each client process corresponds with one server process. When there is a client connection request, the Postgres master process spawns a Postgres backend process to handle requests from that client process. The Postgres master process also allocates a shared buffer pool so that all the backend processes spawned from it can concurrently

access the database data to process client requests. Therefore, this ”process per user” design allows PostgreSQL to handle multiple concurrent client connections.

2.2 Data Analytics

While SQL provides many relational operators and aggregate functions, it is not well suited for analytical workflows because these workflows use many linear algebra operators and require the flow control structures of high-level programming languages that are lacking in SQL. There are recent attempts to extend the syntax of SQL to support linear algebra query processing in RDBMS [9, 36]; however, such newly-developed extensions are not user-friendly when it comes to complex analytical tasks. Conventionally, data scientists rely on specialized data analytics systems to perform relational tasks and numerical computations in a compact workflow.

2.2.1 Database-external Analytics Solutions

It is not uncommon to find relational operator implementations in HLL statistical systems that are designed for numerical computing needs. From the perspective of distributed query processing, these systems follow a data shipping strategy [19] that requires loading the data from an RDBMS into the statistical systems where the analytical task is eventually processed. These database-external statistical systems can establish a connection to an RDBMS using standard database APIs to process SQL queries inside the RDBMS and fetch the relational data into their workspace. Once the data is taken out of the RDBMS and loaded into the statistical systems, they can provide a stand-alone analytics solution to perform both relational and statistical computations within HLL flow control structures.

Nowadays, Python is a popular programming language among data scientists as it has a variety of statistical packages and machine learning libraries. NumPy [34] is a

Python package that is commonly used for numerical computing on multi-dimensional arrays. Its internal data structures are implemented in C, and the computing functions are optimized to be faster than purely Python-based computations. Pandas [24] is built on top of NumPy for numerical data analysis. It also implements basic relational operators such as filter, join, and group-by. These operators equip Python with basic RDBMS functionalities so that we can define analytical workflows and perform the analysis locally in the user space. There also exists distributed statistical systems such as Spark [35], which allows distributing data sets across nodes in a cluster for large scale data analysis. Apart from a set of relational functions, it also has a declarative API that supports expressing relation tasks with SQL syntax [1].

Although these database-external analytics solutions provide a user-friendly interface to perform relational operations without relying on the RDBMS, the capability of relational processing in these systems is not comparable to conventional RDBMSes when it comes to query optimizing. Moreover, the data transfer from the RDBMS to the statistical system is usually constrained by the network connection. The communication overheads of shipping raw data sets may hinder the data analysis performance. In order to avoid these shortcomings, there have been attempts to bring data analytics into the RDBMS.

2.2.2 HLL UDFs for In-database Analytics

User defined functions (UDFs) [23] and stored procedures [13] were proposed to extend RDBMS's support for customized functionalities. Like other built-in functions, they are executable inside the RDBMS, thus avoiding data shipping. In principle, a UDF is a function that can be integrated into a SQL statement within the `SELECT`, `FROM`, or `WHERE` clause, while a stored procedure is a stand-alone executable function that can be called through a special interface without using a SQL statement. An example is given in Listing 2.2 in a later section, where we will discuss the usage of UDFs. As any function

can be made a UDF and then invoked by using a dummy SQL `SELECT` statement, we only use the term UDFs to refer to in-database executable functions from here on.

Early analytics-oriented UDF implementations extend RDBMS to support near-data numerical computations inside the RDBMS. However, these UDF implementations are developed in the C language [5,27], making the programming model opaque as it requires knowledge of RDBMS's internal data representations and fails to accommodate complex analytical workflows. However, as many RDBMSes start to integrate high-level language (HLL) interpreters inside their systems, they start to support UDFs written in an HLL. The usage of UDFs in RDBMS's embedded HLL environment as a tool for in-database analytics is explored in [29].

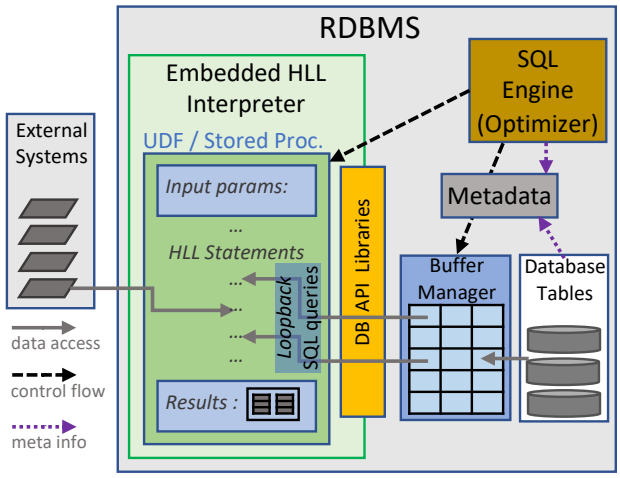

Figure 2.2: An example of HLL UDF [12]

Figure 2.2 from [12] depicts an overview of the UDF construct in RDBMS's embedded HLL environment. The HLL UDF can take input parameters and return a result set. It also has HLL source code in the UDF body for data processing. Unlike early UDF implementations that are defined in C, users can now use an HLL and its statistical packages, such as NumPy and pandas, to define analytical tasks inside the UDFs. Therefore, HLL UDFs equip the RDBMS with the generic facilitation to perform in-database analytics. Ideally, users can define an entire analytical workflow in an HLL UDF and execute it inside the RDBMS.

It is noteworthy that HLL UDFs support loopback queries [29]. These loopback queries allow the users to retrieve relational data sets from the RDBMS at any point inside a UDF. For example, PostgreSQL has a built-in Python module, *plpy*, that is used in the UDFs to push SQL queries to the database engine and bring the HLL result sets to the UDF [32]. These loopback queries are limited to regular SQL statements, meaning that they can only perform relational operations on database objects. Using loopback queries inside a UDF is akin to transferring relational data from the RDBMS to a statistical system so that users can process the data in an HLL. However, loopback queries are executed within the RDBMS; it is more efficient to use them inside a UDF than loading the data into a database-external statistical system.

When the HLL UDF is invoked, the RDBMS will use the embedded HLL interpreter to process the source code in the UDF body. During the execution of the HLL UDF, it is inevitable to transfer data between the RDBMS and its embedded Python environment: loopback queries require loading database data sets into HLL objects, and the HLL return value must be transferred to the RDBMS space so that the database engine can process the result set as part of a SQL query. Due to the fact that the data types may not be compatible across the two environments, any data movement is subject to data copy and conversion, which is a downside of using HLL UDF to process large data sets in regard to the overall execution time. There have been attempts to optimize data handover between the RDBMS and the HLL environment that share similar underlying data structures [21, 29]. They leverage the fact that the RDBMS and the HLL environment access the same memory space to perform zero-copy transfer across the two environments without copying the internal data structures into another representation.

Depending on the purpose and the return value, UDFs can be characterized into the following categories [29]: (1) A scalar UDF operates on individual rows of the data, which is common for numerical computing. (2) An aggregate UDF performs some aggregation function on the data, possibly in groups. (3) A table-UDF returns a table-like data set for

further processing. With these HLL UDFs, users can define analytical logic in an HLL and integrate statistical tasks into SQL queries. The last type of UDFs also allows the users to leverage the RDBMS to perform relational operations on HLL objects. In other words, if the users want to perform relational operations on a data set resulted from HLL computations, they can return the HLL data set in a table-UDF and then integrate the table-UDF into a SQL query that the database engine will execute. We will discuss the usage of table-UDFs in Section 2.3.1.

To summarize, the HLL UDF is a user-friendly interface for in-database analytics. With HLL UDFs, data is allowed to be transferred between the RDBMS and its embedded HLL environment. As a result, users can utilize either HLL statistical functionalities or RDBMS's SQL engine to fulfill analytical needs of different nature. However, it is not an agile programming tool as users are required to pre-define analytical operations inside a UDF that does not allow for interactive analytics.

2.2.3 The AIDA Framework

AIDA (Abstraction for Advanced in-database analytics) [11] is a Python client-server based open-source framework for in-database analytics. In contrast to stand-alone UDFs, AIDA provides a user interface that allows for interactive analytics where clients can submit individual operations to the system in an interactive manner facilitating explorative data analysis. AIDA follows a client-server layout that uses Remote Method Invocations (RMI). Figure 2.3 from [11] depicts the architecture of AIDA with its client-side and server-side components. AIDA's server resides inside the embedded Python environment of the RDBMS and is started by executing a bootstrap stored procedure. AIDA's server component has a connection manager that is responsible for connecting each of AIDA's clients to a database workspace object.

The client component is responsible for shifting user-written tasks from the client side to AIDA's in-database server component, which follows the logic of query

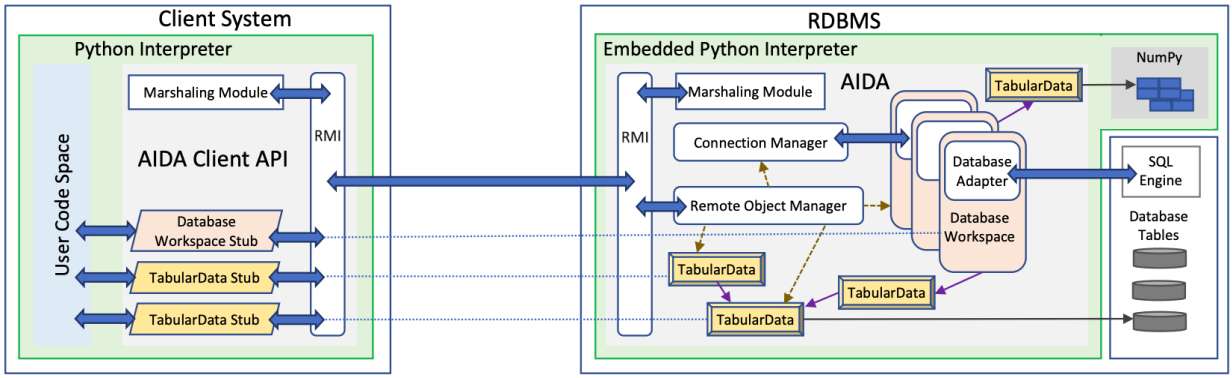

Figure 2.3: High level architecture of AIDA [11]

shipping [19]. Similar to many popular data science frameworks, AIDA's client API is based on Object-Relational Mappings (ORM) [25], which provide object-oriented methods to define operations on the data sets that are stored in the RDBMS. Since AIDA's server resides inside the embedded Python environment of the RDBMS, the database workspace object can use NumPy to perform linear algebra requests and leverage RDBMS internal Python APIs to push down relational requests to the database engine. The database adapter implementation in the database workspace functions as a bridge that interconnects the SQL engine of the underlying RDBMS with the rest of AIDA through RDBMS-specific APIs. This modular design approach makes AIDA portable to another RDBMS by implementing a new database adapter.

The abstraction of data sets in AIDA is referred to as TabularData, and it allows the users to perform both relational operations and linear algebra computations in an interleaved fashion. TabularData objects can encapsulate data sets no matter if they are database tables or Python data represented by NumPy arrays.

TabularData objects reside in AIDA's server component and thus in the embedded Python environment of the RDBMS. A TabularData object has two internal Python representations in the Python environment. In order to accommodate linear algebra operations, both representations are vector-based. The first internal data representation is a dictionary-columnar format that is used to hold data obtained from relational

operations. It is essentially a Python dictionary with column names as keys and the values being the column data in NumPy array format. The second one is a two-dimensional array representation for linear algebra operations such as matrix multiplications. This matrix format is only possible when all the columns in the data set have the same data type.

Applying an operation on a TabularData object will result in a new TabularData object; however, the new TabularData object may not be materialized with an internal data representation. In other words, a TabularData object can have empty internal data representations at a given time. A particular internal representation of the TabularData object will not be materialized until the data representation is requested for the first time, and the materialized data representation can be reused for future operations.

AIDA separates request operations by their nature: relational operations will be pushed down to RDBMS's database engine, and the statistical package NumPy will be in charge of linear algebra computations. AIDA performs relational operations lazily as it combines relational operations and only translates them into a SQL query when it is required to materialize the internal data representation of a TabularData object. With such a lazy approach, AIDA can benefit from the query optimization capability of the RDBMS instead of executing each simple relational operation separately. On the other hand, linear algebra operations require immediate materialization of the result TabularData object as NumPy does not optimize combined linear algebra operations.

In order to facilitate the cohesive execution semantics and both relational operations and linear algebra computations in an interleaved manner, AIDA will transfer the data sets between the embedded Python environment and the underlying RDBMS if they do not exist in the respective execution environment. When AIDA needs to materialize a TabularData object that results from relational operations, the database adapter will push down the translated SQL query to the database engine for execution and then load the result data set from the database to the embedded Python environment. The result

set is used to form the dictionary-columnar format internal representation used by the TabularData object. The data transfer in this direction is managed by RDBMS APIs. In contrast, the data transfer from the Python environment to the RDBMS has several possible implementations. When AIDA needs to perform relational operations on a TabularData object that has been materialized in the Python space, it transfers the TabularData object's internal data representation to the RDBMS. In AIDA's base implementation, the database adapter automatically creates table-UDFs that return the dictionary-columnar data representation for the TabularData objects, which are then fed to the database engine. The data transfer process is one of the impact factors of executing the SQL query that integrates the table-UDF. An advanced implementation of AIDA uses virtual tables [12] to expose the dictionary-columnar data to the RDBMS. We will present these data exposing mechanisms in more detail in Section 2.3.

The first implementation of AIDA [11] is built on top of MonetDB - a column-oriented RDBMS. Given that MonetDB and NumPy share the same underlying C representation of numeric values and vector-based data structures, it is possible to transfer data between MonetDB and its embedded Python environment with a minimum level of data conversions. As the RDBMS and the embedded Python interpreter run in the same process with shared memory, the data transfer requires zero-copy if the data types are compatible across the two environments and no data conversion is needed. The result set returned by a SQL query is handed over to the embedded HLL environment without making copies of the underlying data structures whenever possible. Similarly, when a UDF returns HLL vectors to perform relational operations, MonetDB can access this data without any explicit copy in many cases. That is, the AIDA implementation for MonetDB can leverage such structural similarities to optimize data movement in both directions. Nevertheless, for row-oriented RDBMSes, transferring data between RDBMS and the embedded Python environment will impose inevitable overheads of data conversions. In this thesis, we consider using PostgreSQL

as a row-oriented example to study the implications of such behaviors, and we also aim at optimizing the overall data transfer process.

2.3 Exposing HLL Data to the RDBMS

Executing SQL queries over data sets that are not stored in the RDBMS is cumbersome as it traditionally requires loading the data into a database table; Python DB-API [22] allows the users to prepare SQL INSERT statements with Python data objects and send the SQL statements to the RDBMS. In the scope of our study, since the data sets already reside inside RDBMS's embedded Python environment, we want to explore a more elegant data exposing mechanism to directly perform SQL queries over such data sets.

2.3.1 Table-UDFs

The term table-UDF, as mentioned in Section 2.2.2, is used to describe a UDF that returns an HLL data object which follows a table-like structure with column attributes and records containing data values. Listing 2.2 demonstrates how to define a table-UDF in PostgreSQL's syntax and use it in a SQL query. The body of this table-UDF is a programming snippet written in Python. It uses a pandas function to load student information into PostgreSQL's embedded Python environment from a CSV file, whose filename is given as an input argument of the table-UDF. Once the data set is loaded into the Python environment, further Python operations can be applied in the UDF body. The result data set has to be prepared in a row-based format to be compatible with PostgreSQL's internal data representation before it is returned to PostgreSQL's database engine.

Regarding the usage of a table-UDF, it can appear in a SQL query similar to regular database tables as in the last line of Listing 2.2. However, the data engine actually invokes the HLL UDF that returns a row-based result set instead of scanning through data pages

```
CREATE FUNCTION students (filename TEXT)
RETURNS TABLE (s_id INTEGER, s_name TEXT) AS $$
    # Read the data from a CSV file.
    import pandas as pd
    df = pd.read_csv(filename)

    # Data manipulation ...

    # Prepare the result set in a data structure compatible with the RDBMS.
    result = ...
    return result
$$ LANGUAGE plpython3u;

SELECT s_id FROM students('/data/students.csv') where s_name = 'John Doe';
```

Listing 2.2: An example of a table-UDF and its usage

from the disk. As for the SQL query in Listing 2.2, after the student data set is transferred from the embedded Python environment to the RDBMS space, the SQL engine performs a selection operation on its records to find the student with the name *John Doe*, followed by a projection to find the student id in the row record.

Although table-UDFs provide a programming tool for the RDBMS to execute SQL queries over non-database HLL data objects, it still leaves room for improvement [12]. Below we will discuss some deficiencies of the approach that uses a Table-UDF to perform SQL queries over an HLL data object.

Query execution plans: PostgreSQL has a sophisticated cost-based query planner to optimize execution plans [32]. The query planner uses statistical metadata, such as data set cardinality, to estimate the cost of each individual operation within the SQL query and generate an execution plan that has the smallest overall cost. However, Table-UDFs are perceived as a black-box to the RDBMS SQL engine. Although a table-UDF can predefine how many rows it will return as part of the UDF creation statement in a declarative manner, it usually cannot foresee how many rows the result set will contain, leaving this option irrelevant to UDFs of analytical workflows. In general, a UDF fails to inform the query planner of the cardinality of its result set. Not having such information, the query

planner simply uses a constant value to estimate the execution cost, which can lead to inaccurate cost estimations, and therefore, an execution plan with poor performance.

Data conversion and transfer overheads: When data is transferred from the embedded HLL environment to the RDBMS, any HLL data whose types are not compatible with the database data types must be converted to proper database representations. When a table-UDF is invoked from the SQL engine, the whole result set will be transferred to the RDBMS space in an eager manner, meaning that each column with an incompatible data type has to be converted no matter whether the column is actually needed to process the SQL query. As an example, despite that only the column *s_id* is projected in the SQL query in Listing 2.2, the entire data set of student information will be transferred to the RDBMS nevertheless, which causes unnecessary data type conversion overheads during data transfer.

Repeating HLL operations: Although the table-UDF approach is flexible in the way that it allows users to write HLL operations to process the data before handing the result set to the RDBMS, the HLL operations defined in the HLL snippet are indivisible and thus, have to be executed whenever the SQL engine invokes the table-UDF. If a table-UDF is used more than once in a SQL query, the HLL operations defined inside the UDF will be repeatedly and collectively executed for each table-UDF invocation from the RDBMS's SQL engine. As in Listing 2.2, the HLL result set has to be transformed into a data structure that is compatible with the structural representation of PostgreSQL before being sent to the SQL engine. Should the table-UDF be requested several times, even within the same SQL query, the data format transformation of the result set also happens multiple times, wasting the computing resources. Depending on the result set's size, the computation cost of repeated data format transformation may not be negligible.

It is worth mentioning that PostgreSQL has a new feature that allows the users to attach a customized *planner support function* to a target UDF after the release of PostgreSQL 12 [32]. Although this *planner support function* can inform the query planner

about the number of rows that the target UDF will return, the target UDF is still subject to the last two deficiencies mentioned above.

2.3.2 Virtual Tables

The concept of virtual tables [12] provides a new data exposing mechanism to alleviate the drawbacks of table-UDFs. It can facilitate running SQL queries over data objects in RDBMS's embedded HLL environment in an optimizer-friendly fashion. Originally developed in MonetDB, Figure 2.4 from [12] illustrates the architecture of the virtual table implementation with the following implementation designs:

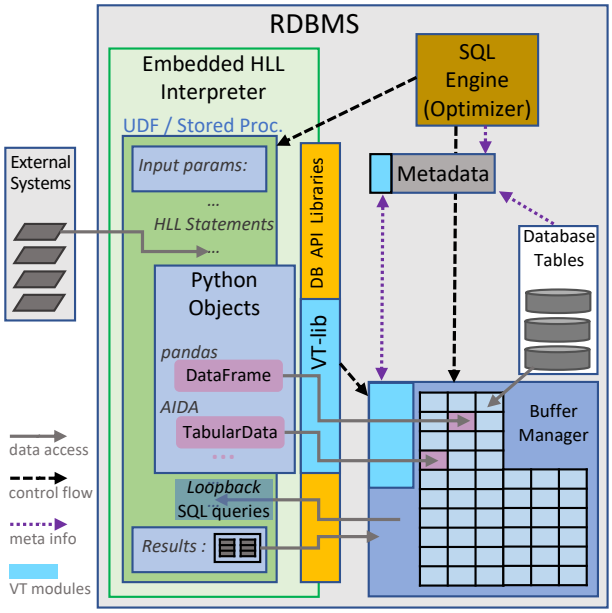

Figure 2.4: Virtual tables implementation in MonetDB [12]

The core of this implementation is the virtual table library which interconnects the RDBMS and its embedded HLL environment. No intrusive changes were made to the RDBMS. The virtual table library provides an HLL API to register HLL data objects that have a table-like data structure in the RDBMS. Despite that the data set is known to the RDBMS, the data set still resides in memory without being materialized down to data storage files.

The library only delivers a virtual view of the registered HLL data set to the RDBMS. The virtual tables can be used in a SQL query in the same way as regular database tables. The underlying data representation is concealed from both the RDBMS SQL engine and the users, where any data type conversion and data transfer will be taken care of by the virtual table library. This enables a lazy conversion strategy that the library only converts and transfers data in the columns when they are explicitly requested by the SQL engine.

In contrast to table-UDFs that follow a black-box setup, the virtual table library can collect metadata information from the registered HLL objects before transferring the data to the RDBMS space. Such information is crucial to the query planner; therefore, virtual tables are optimizer-friendly as they can provide the cardinality of registered data to the query planner in the hope of an optimized execution plan. Virtual tables also provide a more flexible interface to interact with the RDBMS: the virtual table library can register a data set at any point in an analytical workflow, whereas the table-UDF approach can only return a data set at the end of the UDF body, not allowing any further data processing within the UDF. Moreover, once the HLL object is registered as a virtual table, we can then perform SQL queries over the data set without caring how the data is obtained. However, if we use table-UDFs, we have to execute all the source code in the UDF body to expose the return value to the RDBMS, which may lead to unnecessary computing if the table-UDF is invoked more than once, as we described in Section 2.3.1. Therefore, we can use virtual tables within HLL UDFs or in-database analytics systems, such as AIDA, to facilitate data movement between the RDBMS and its embedded HLL environment.

The virtual table implementation for MonetDB in [12] also leverages the fact that the internal representation of MonetDB tables shares similar underlying data structures with many Python objects used in statistical libraries, such as NumPy arrays. MonetDB uses memory-mapped I/O [17]: When MonetDB's buffer manager receives a request to retrieve a column, it can map the data storage file into a memory address as if the column is a memory resident. Since the registered Python data is already an in-memory

resident, MonetDB can directly access the memory address of the Python data to process a SQL query over the virtual table [12].

Although such optimizations for MonetDB do not align with PostgreSQL because of different data representations between the two RDBMSes, a virtual table implementation for PostgreSQL is still possible by following the same logic to generate efficient execution plans even when accessing data inside PostgreSQL's embedded Python environment.

2.3.3 SQL/MED & Foreign Data Wrapper

The foreign data wrapper interface was introduced into the SQL standard as part of the SQL/MED (SQL Management of External Data) extension [26]. This extension defines an interface between the RDBMS and a foreign data wrapper with the objective to provide a standardized access method for external data that is not stored in the local RDBMS. Since we are interested in transferring data from the embedded HLL environment to the RDBMS space, this interface is worth further exploration.

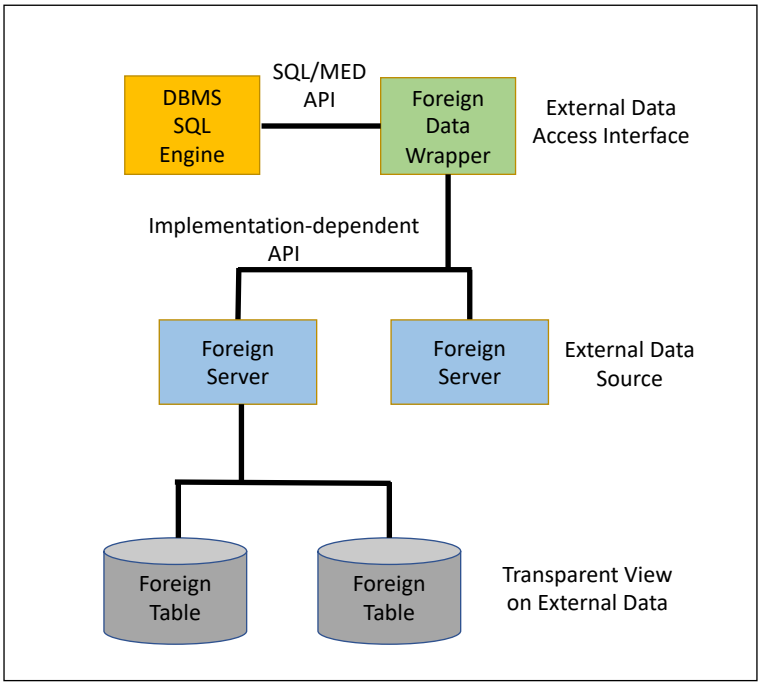

Figure 2.5: SQL/MED components

Figure 2.5 demonstrates the main components of the SQL/MED standard. Since SQL is designed for the relational model, should the external data be expected to fit into the RDBMS seamlessly, it has to be perceived as regular relational tables to the RDBMS. To address this requirement, SQL/MED brings up the notion of foreign tables to represent external data. The foreign tables are designed to provide a transparent view of the external data, which conceals the fact that the data is not stored locally. Therefore, external data is known to the RDBMS as a database table. Since the external data source can contain a collection of different data sets, SQL/MED defines a foreign server concept as the abstraction for the external data source, which allows the RDBMS to access multiple data sets that are maintained in one external data source in the same manner.

Furthermore, if multiple external data sources share a common interface, it is plausible to use a single module within the RDBMS to access the external data managed by such data sources. This module is framed as a foreign data wrapper in SQL/MED. Therefore, the onus of accessing such data sources and regulating their corresponding foreign servers is on the foreign data wrapper.

SQL/MED specifies an interface for the RDBMS and the foreign data wrapper to interact with each other. When the RDBMS's SQL engine and the foreign data wrapper need to work together to process SQL queries over a foreign table, their interaction can be divided into the following two phases [26]:

1. **A query planning phase**: the RDBMS and the foreign data wrapper exchange information to produce an execution plan for the query.

2. **A query execution phase**: the query is executed in accordance with the plan, and the foreign data is transferred to the RDBMS.

PostgreSQL implements a foreign data wrapper interface with routine functions for such interaction [32]. It allows the query planner to inquire about the cardinality

information of external data in order to generate an optimized execution plan and also regulates how the external data is transferred to the SQL engine for query execution.

There have been studies on leveraging PostgreSQL's foreign data wrapper feature to access external data from non-relational databases or the file system [8,30]. Although the foreign data wrapper feature was originally designed to expose external data to PostgreSQL, the same mechanism can be applied to access the data in PostgreSQL's embedded Python environment. In our case, we will explore the possibility of using this feature to provide an optimizer-friendly data exposing mechanism to access the data represented inside AIDA's TabularData objects.

2.3.4 Multicorn

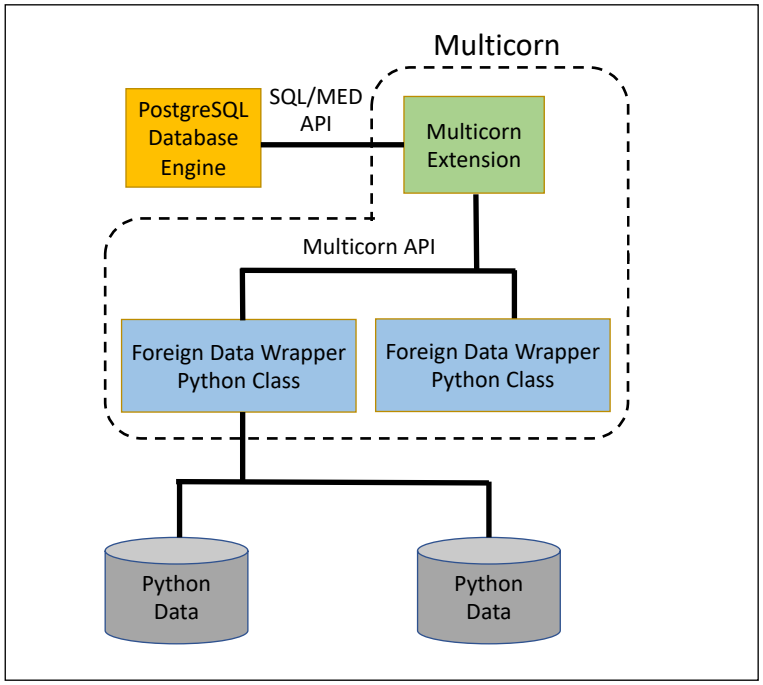

Figure 2.6: The architecture of Multicorn

Multicorn [20] is a third-party foreign data wrapper implementation for PostgreSQL. It has an interface that allows users to develop a foreign data wrapper in Python. Figure 2.6 captures how the SQL/MED components are presented in Multicorn. The Multicorn

extension interacts with the database engine in accordance with PostgreSQL's foreign data wrapper API. It accesses the foreign data through a Python class which manages the actual data retrieval from the external source. Although this Python class is referred to as a "foreign data wrapper" class in Multicorn, it is used to create the foreign server (see Listing 2.3).

There are two steps to set up the connection between PostgreSQL's SQL engine and the foreign data. First, we need to create a foreign server with Multicorn extension as in Listing 2.3. A foreign server needs to be created only once to store it in PostgreSQL's system catalogs. In the `OPTIONS` clause of the `CREATE` statement, it is required to include an option to provide the path to a foreign data wrapper class that we want to use to create the foreign server. Secondly, we need to create a foreign table with that foreign server as in Listing 2.4. At this step, data type information of columns has to be included in the `CREATE` statement, along with other options specific to its foreign server and foreign data wrapper class. The foreign table provides an abstract view of the external data so that it can appear in SQL queries as a regular database table. For every foreign table, there is a running instance of the foreign data wrapper class that is responsible for accessing the foreign data.

```
1 CREATE SERVER server_name FOREIGN DATA WRAPPER multicorn
2 OPTIONS ( wrapper 'path to a Python class' );
```

Listing 2.3: Foreign server creation synopsis

```
1 CREATE FOREIGN TABLE table_name (
2     column_name data_type
3     ... )
4 SERVER server_name
5 OPTIONS ( option_name 'value', ... );
```

Listing 2.4: Foreign table creation synopsis

Multicorn defines a base Python class with signatures of the functions that will be called from the Multicorn extension to access the foreign data. To present the foreign table as a regular database table, Multicorn does not only provides routine functions to process data queries but also supports an API for insert, update, and delete statements. Since our objective is to expose Python data objects to the RDBMS, we only focus on the read aspect of the foreign data wrapper. In order to facilitate query planning, the foreign data wrapper can provide the exact data set's cardinality to the query planner if the corresponding function is implemented. And then, a foreign scan function is invoked for query execution. The foreign data wrapper class has to implement a function that provides the rows of the foreign data in an iterative manner. The Multicorn extension will convert the row-based Python data set to PostgreSQL's internal data representation before sending it to the SQL engine.

Therefore, we can create a specialized Multicorn foreign data wrapper by implementing these methods to regulate how to expose a Python object from the embedded Python environment to PostgreSQL's SQL engine through the Multicorn extension. Similar to using a table-UDF, the foreign data wrapper still requires transferring the data set for every invocation of the foreign table from the SQL engine. However, it is different in the sense that the foreign-data wrapper can provide meta-information to the query planner and allows transferring only a subset of the data depending on the execution request.

Chapter 3

An AIDA Implementation for PostgreSQL

In this chapter, we extend AIDA to be able to interact with the row-based RDBMS PostgreSQL. Some specific characteristics of PostgreSQL make our development a non-trivial process. We present the challenges that we have encountered during the development and our attempts to alleviate their influence on AIDA's usability. Lastly, we evaluate the performance of our AIDA implementation with regard to data movement between PostgreSQL and AIDA's workspace.

3.1 The Database Adapter Interface for PostgreSQL

PostgreSQL has an embedded Python interpreter, in which we can integrate the AIDA server. Programs running inside the embedded Python environment can use the built-in *plpy* module to push down SQL queries to PostgreSQL's database engine and bring the SQL result set to the Python environment. Furthermore, since PostgreSQL supports table-UDFs, it allows exposing AIDA's Tabular Data objects, which are HLL objects, as regular database tables to PostgreSQL's SQL engine for executing relational operations

over them. AIDA's modular design approach allows it to abstract such RDBMS-specific interface by designing a database adapter to interconnect the rest of AIDA with the RDBMS. In order to regulate the data transfer between PostgreSQL and the embedded Python environment, we adjust AIDA's database adapter interface so that it is specific to PostgreSQL.

3.1.1 AIDA Server Management

To start AIDA's server-side component, we need to execute a bootstrap stored procedure that runs AIDA's server component in PostgreSQL's embedded Python environment. The procedure starts up a connection manager for managing AIDA's connection to remote clients and a remote object manager for handling RMI requests from AIDA's client-side. Since PostgreSQL uses a "process per user" client/server model as described in Section 2.1.2, we have to make the stored procedure long-lasting to keep AIDA's server component alive. When we start the stored procedure, a Postgres backend process is spawned by the Postgres master process to manage the connection and execution of that stored procedure. Once the stored procedure finishes, the connection is lost, and thus, the Postgres backend process also vanishes, together with the running instance of AIDA's server.

To resolve this issue, we introduce a "looping mechanism" by having an infinite loop in the bootstrap procedure to prevent the Postgres backend process from terminating and keep the connection manager as well as other Python objects in the embedded Python environment alive indefinitely. Thus, this long-lasting Postgres backend process, which executes the bootstrap procedure, plays the role of AIDA's server process.

However, such an approach means that this process will have to execute all of AIDA's operations from different AIDA clients. In more detail, when a new AIDA client connects to the AIDA server, the connection manager will create a database workspace object for this connection. Since the workspace objects are created inside the looping

bootstrap procedure running inside the PostgreSQL backend process, the SQL engine interface in the Python environment, the *plpy* module, is shared across database workspaces and thus AIDA client connections. This necessitates synchronizing SQL query requests that are generated from different database workspaces. Consequently, we introduce a synchronized queue that is shared by the database workspaces for storing the SQL requests that will be pushed down to PostgreSQL's SQL engine through the *plpy* module.

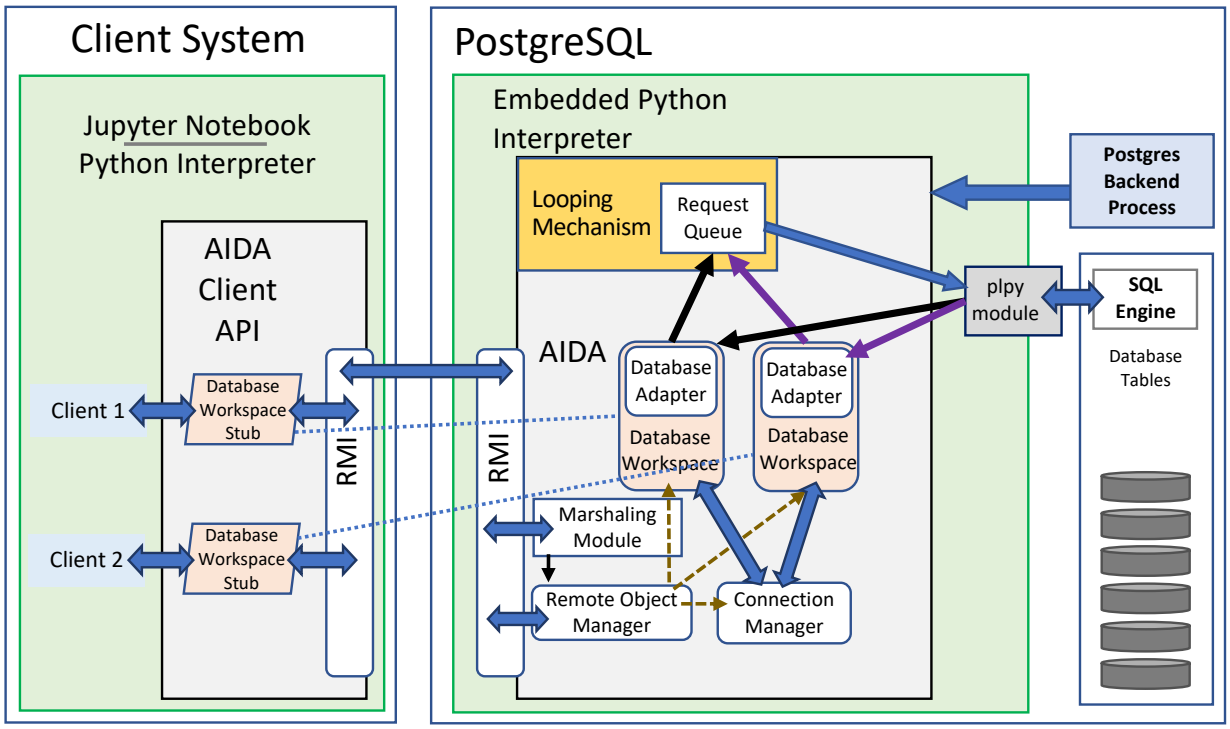

Figure 3.1: Architecture of AIDA adjusted to PostgreSQL

Figure 3.1 depicts how the looping mechanism and the synchronized queue collaborate in the adjusted AIDA framework. When AIDA's client-side requires to materialize a TabularData object from a sequence of relational operations, the database adapter in the corresponding database workspace adds the translated SQL request to the shared request queue and waits until the query result is added to a result queue that belongs to this database workspace. Each iteration of the infinite loop in the bootstrap procedure loads a request from the queue and gives the corresponding database adapter

exclusive access to the *plpy* module for executing the SQL query. Subsequently, the result set returned by the SQL engine will be processed for formatting and then put into the result queue of the database workspace so that the original request caller can retrieve the result set from the queue. A limitation of this approach is that the SQL query requests from two different AIDA clients cannot be executed in parallel as their requests will be synchronized.

3.1.2 SQL Result Set Data Structure Conversion

When AIDA needs to execute relational operations to materialize a TabularData object, it is the database adapter's job to submit the translated SQL query to PostgreSQL and prepare the query result in a column-based format. As described in Section 2.2.3, the TabularData object in AIDA has two internal representations: (i) the first representation is a dictionary-columnar format, which is essentially a Python dictionary of column names paired with the column data in NumPy arrays, (ii) the second representation is built on top of a matrix of contiguous columns, where all the columns have the same data type.

The original implementation of AIDA for MonetDB [11] took advantage of the zero-copy optimization that was possible because of the similarity of their data structures. Thus, formatting MonetDB's result set into the TabularData object's internal representation was nearly instantaneous in most cases without copy or transformation with the exception of string values. However, the row-oriented PostgreSQL has distinctly different internal storage data structures from the vector-based data structures used in AIDA and the statistical packages on which AIDA relies. The *plpy* module returns the result set in a row-by-row format. Therefore, the database wrapper has to perform a data structure transformation to transform this row-by-row result set into the TabularData object's dictionary-columnar representation. Nevertheless, such row-to-column transformation is expensive to perform if the result set is relatively large.

A naive conversion method performs the transformation in Python space. To alleviate the transformation cost, we implemented two advanced methods to perform the transformation on top of the internal data structures. All three methods are depicted in Figure 3.2.

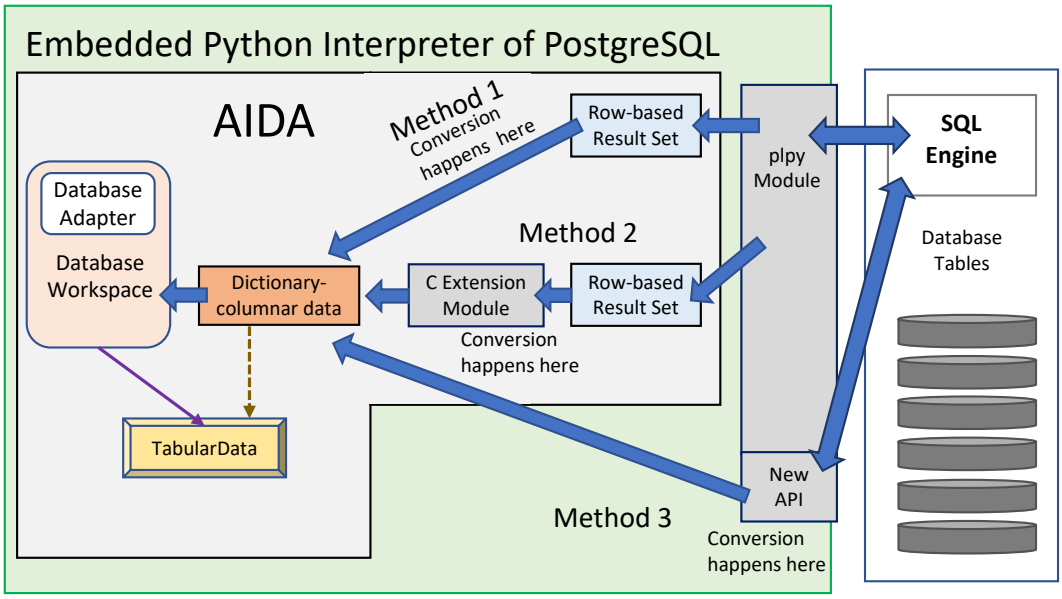

Figure 3.2: High Level Execution Flow of Each Conversion Method

1. **The naive conversion method**: When the *plpy* module returns a row-based result set to the embedded Python environment in which AIDA's server resides, we first create empty NumPy arrays for each of the columns in the result set and then iterate over each row to fill these NumPy arrays. At the end, the appropriate annotations are created to build a full-fledged TabularData object in the dictionary-columnar representation. These conversion steps are written in Python and are executed by the embedded Python interpreter of PostgreSQL.

2. **Convert with an extension module written in C**: Instead of performing the conversion in the Python environment, we use Python/C API[1] to loop over the

[1] `https://docs.python.org/3/c-api/`

rows and perform the transformation on top of the internal data structures as it is faster to do so in C. That is, once we receive the Python row-based result set via the *plpy* module, we pass it to the extension module to transform the result set into the dictionary-columnar format.

3. **Perform the transformation within the *plpy* module:**. In this version, we add a new method to PostgreSQL's *plpy* module, which performs the data structure conversion before returning the result set to the Python environment. Instead of returning a Python row-based, this method directly creates a dictionary-columnar result set from the output produced by PostgreSQL's SQL engine and wraps the internal C representation of the result set as a Python object. When AIDA's database adapter needs to execute an SQL query, it invokes this method rather than the original execution method. Therefore, there is no further operation required after the object is received in the embedded Python environment. It is notable that this last option modifies the *plpy* module provided by PostgreSQL. Strictly speaking, it does not consider PostgreSQL as a black-box anymore, but PostgreSQL itself was extended to support AIDA.

3.1.3 Relational Operations on TabularData Objects

As we have described in Section 2.2.3, TabularData objects that are created from a relational operation are not immediately materialized. Instead, a lazy execution strategy takes place to avoid the materialization of intermediate results. Thus, a TabularData object is only materialized when the client either explicitly requests its data or because it becomes an input for a linear algebra operation. If it depends on other non-materialized objects, AIDA will translate the combination of relational operations into a single SQL query and use PostgreSQL's database engine to execute the query.

If a materialized TabularData object then becomes an input of subsequent relational operations, the object eventually needs to be handed over to PostgreSQL to run the SQL query over it. The default data exposing mechanism in the original implementation of AIDA [11] is to use a table-UDF, as described in Section 2.2.3, to transfer the TabularData object's dictionary-columnar representation to PostgreSQL's SQL engine.

In principle, this is also how the original AIDA implementation for MonetDB [11] exposes TabularData objects to MonetDB. However, it performs a much simpler task to transfer the data from AIDA to the embedded Python environment as the dictionary-columnar format is exactly the format needed by MonetDB.

Again, since PostgreSQL is a row-orient RDBMS, the data exposed to the SQL engine is expected to be in a row-based format. Thus, this introduces yet another data structure conversion, but in the opposite direction, namely from the dictionary-columnar structure to a row-based format. Within the table-UDF, it essentially does the opposite of the previously mentioned row-to-column data structure conversion; that is, it transfers the dictionary-columnar data to a row-based format that resembles the result set returned by PostgreSQL's *plpy* module.

In contrast, as a stand-alone data analysis solution, pandas has a set of relational operators for its own DataFrame objects that are conceptually similar to AIDA's TabularData objects. If relational operations are performed on data that is already retrieved from the RDBMS and maintained in the DataFrame objects, then pandas uses its own implementation, which works on the column-based DataFrame objects and does not require any conversion. Hence, the data structure conversion from the columnar structure to the row-based format in order to use a row-oriented relational database engine is an extra overhead that is not encountered by other systems besides our implementation of AIDA for PostgreSQL.

In our AIDA implementation for PostgreSQL, we design a workaround to avoid such column-to-row data structure conversion by also supporting a row-based representation

for the TabularData objects in AIDA. In particular, for a materialized TabularData object that is the result of relational operations, while we get the Python row-based result set from PostgreSQL and convert it into the dictionary-columnar format, we can also cache the row-based result set as an extra representation. Therefore, when this TabularData object needs to be exposed to PostgreSQL for running SQL queries over, we can directly return the cached row-based result set instead of performing the expensive column-to-row conversion. However, this workaround is only possible with the first two result set preparation methods that we have described in Section 3.1.3 and is incompatible with the extended *plpy* module approach as we do not receive a row-based result set with this implementation. Also, it is noteworthy that such optimization has a space versus efficiency trade-off as it needs to maintain both the dictionary-columnar and row-based representations to avoid the column-to-row conversion in the hope of a performance improvement. Furthermore, it is only possible for a subset of the TabularData objects; a row-based result set is not available for TabularData objects that are derived from linear algebra operations. Finally, while this avoids the column-to-row conversion, PostgreSQL still uses a row-by-row iteration model to retrieve the rows, which includes copying each row into the database space; this overhead cannot be avoided with the current PostgreSQL interface.

3.2 Evaluation

3.2.1 Test Setup

With our AIDA implementation for PostgreSQL, we have to transform the row-based SQL result set to a columnar format when retrieving data from PostgreSQL. In the other direction, we have to convert AIDA's TabularData objects from a dictionary-columnar format to a row-based structure before feeding them to PostgreSQL to perform relational operations.

The question is whether this data structure conversion overhead is more expensive than bringing the data to the client space and executing analytical operations as done with external data analytics solutions. In order to evaluate the performance of our AIDA implementation for PostgreSQL, we use the numerical synthetic data sets from [11] to test our system. The details of the data sets will be described in the corresponding experiments.

To measure the overall overhead of our AIDA implementation for PostgreSQL, we evaluate the performance of AIDA in terms of response time against a purely UDF-based approach for data analytics as described in Section 2.2.2. We also run the experiments with NumPy and pandas as client-based applications that transfer the data to the client side and process the data in user space with the respective package.

We run the experiments on two nodes with identical specifications of Intel® Xeon® CPU E3-1220 v5 @ 3.00GHz and 32 GB DDR4 RDIMM. The two nodes are connected via a Dell™ PowerConnect™ 2848 switch in a Gigabit private LAN. One node runs PostgreSQL with AIDA embedded or not, depending on the system tested. The other runs the AIDA client or the external data analytics systems, NumPy and pandas, with which we compare.

Unless explicitly specified, we use PostgreSQL 12.3, Python 3.5.2, NumPy 1.12.1, pandas 0.23.3, and psycopg2 2.7.5. with default settings. In all the experiments, we start to take time measurement after a warm-up phase.

3.2.2 Making SQL Result Set Computational

We use this test case to study the cost of transforming the row-based SQL result sets into a columnar format that is compatible with linear algebra operations. We run the experiments with database tables that each has 100 columns of randomly generated floating-point numbers. The number of rows in the data tables varies from 1 to 1 million. We took these numerical synthetic data sets from [11].

For AIDA, we materialize TabularData objects to load the database tables. The row-based SQL result set is converted to the dictionary-columnar representation during the materialization. We also examine how the conversion methods that we described in Section 3.1.2 can influence the performance of the materialization process. We denote them as follows:

- **AIDA**: We convert the row-based result set that we receive from the *plpy* module to a columnar format in Python space. That is the naive conversion.
- **AIDA-CM**: We use the extension module to perform the conversion with the underlying data structure of the row-based result set in C.
- **AIDA-MAPI**: We modify the *plpy* module and use the customized function that directly returns a dictionary-columnar representation of the SQL result to the embedded Python environment.

We measure the loading time at AIDA's client side that initiates the load request. Although the data itself still resides in the server end, the cost of RMI communication is included in the loading time. However, we expect this to be small because no actual data is transferred to the client side. Apart from the AIDA variants mentioned above, we also run the experiments with the following test candidates:

- **DB-UDF**: Instead of using the AIDA framework, we use a Python UDF to load the data and convert the result set into the dictionary-columnar format as a TabularData object. The Python UDF is executed on the server side. Thus, there is no client-server communication cost and no AIDA runtime overhead.
- **NumPy** and **pandas** client-based applications: We use psycopg2, a Python DB-API implementation for PostgreSQL, to retrieve the SQL result set from PostgreSQL to user space and perform the data structure transformation on the client side. The NumPy implementation creates a dictionary-columnar representation that is

similar to the internal representation of TabularData objects, while pandas creates its DataFrame object that contains labeled axes for rows and columns. As the performance of retrieving data from PostgreSQL depends on the cursor buffer size of psycopg2, we test with the default buffer size of 100 and also an optimized buffer size of 1 million, to accommodate the largest database table in this test case. This optimized setting is recorded as **NumPyOpt**. The loading time of these two applications is measured at the client side, which includes the communication and data shipping cost.

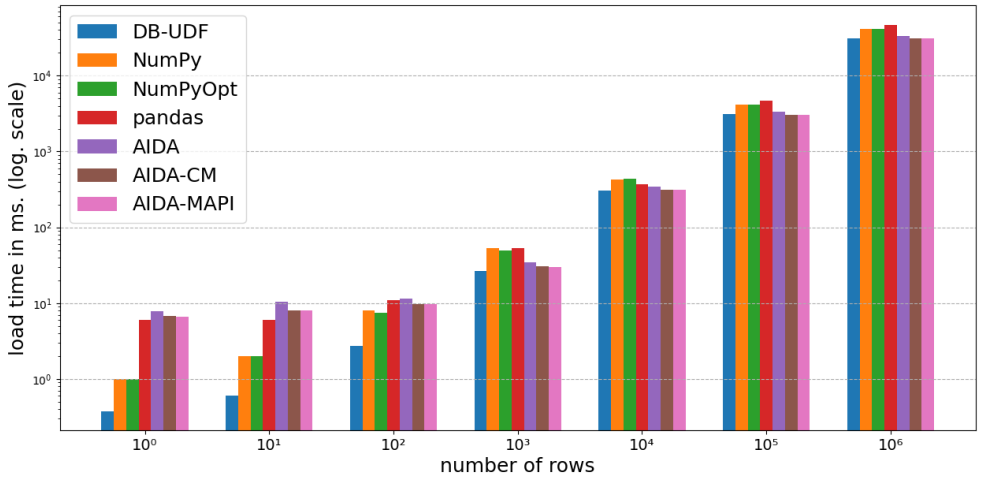

Figure 3.3: Time to load data (The server and client communicate across a switch)

Figure 3.3 shows loading time on a logarithmic scale. For small data sets less than 100 rows, the AIDA variants take about 20 times longer than **DB-UDF**, approximately five times more than **NumPy**, and 20% slower than **pandas**. Such worse performance can be attributed to AIDA's framework overhead of handling RMI communication and managing the creation and meta-data of TabularData objects at runtime.

For larger tables with more than 100 rows, AIDA starts to show the benefits of keeping data inside the RDBMS. Since **DB-UDF** avoids any client-server communication and does not have framework overhead, it has relatively better performance than others.

When we compare AIDA variants with **DB-UDF**, the framework overhead becomes less noticeable as the size of the database table increases. For the small table with 100 rows, **DB-UDF** is 4 times faster than AIDA solutions, but with the larger table of 1 million rows, the two optimized AIDA variants are actually slightly faster than **DB-UDF**. Both **AIDA-CM** and **AIDA-MAPI** are faster than **AIDA** by around 10% because of their optimization of the data structure conversion process. **AIDA-MAPI** is slightly faster than **AIDA-CM**, making it the most efficient AIDA implementation for PostgreSQL. However, the difference between these two variants is not to a noticeable degree.

Pandas has extra computation cost when it constructs a pandas DataFrame, which contains labeled axes for both rows and columns. For tables with more than 10 rows, it becomes slower than AIDA solutions. **AIDA-MAPI** takes 10% less time than pandas at 100 rows and 35% less time when the table size reaches 1 million rows. For large tables, the performance of **NumPy** lies in between pandas and the AIDA solutions; this is mainly because of data shipping cost.

Considering **NumPy** and **pandas** take data out of PostgreSQL and transfer it to the client side, their performance is significantly constrained by the client-server network condition and the hardware on the client side. On the other hand, AIDA performs the data structure conversion in PostgreSQL's embedded Python environment, and thus it is not likely to be affected by the network condition while leveraging the high-end server computation power. To emulate a more realistic client-server connection, we re-run the experiments with the client on a remote machine that communicates with the server across the Internet. The remote machine has the following hardware: Intel® Core™ CPU i5-5257U @ 2.70GHz and 8GB DDR3 RAM. Since we have witnessed that **AIDA-MAPI** is the most efficient conversion method, we will run the rest of the evaluations with this conversion method unless explicitly specified.

Figure 3.4 shows the loading time of the cases that the client communicates with the server across the Internet on a logarithmic scale. While **AIDA-MAPI** is not being

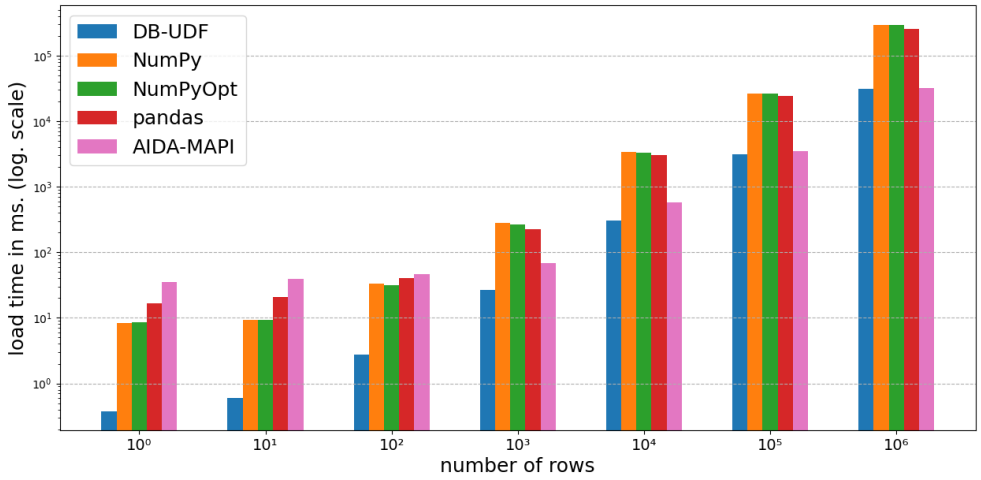

Figure 3.4: Time to load data (The server and client communicate across the Internet)

affected too much by the network condition (as only one RMI request is sent across the Internet), **NumPy** and **pandas** applications need to transfer the data from the server and process it in the user space. We see a huge increase in their loading time. When the table has 1000 rows, **AIDA-MAPI** outperforms **NumPy** and **pandas**, where **AIDA-MAPI** takes only around 30% of the time needed by **NumPy** and **pandas** to load the table into a computational object. This trend continues as the data set size increases. At 1 million rows, **AIDA-MAPI** is 9 times faster than **NumPy** or **pandas**. The performance of **DB-UDF** remains unchanged as it does not have client-server communication.

3.2.3 Relational Joins

In this test case, we want to analyze how AIDA can leverage PostgreSQL's SQL engine to perform a join operation in comparison with the join functionality implemented in pandas. We experiment with two data sets taken from [11]; each has 11 integer columns and 1 million rows. One of the columns, the primary key, is unique and identical in both data sets, while the rest of the fields may share the same value between the two data sets or not.

We have two baseline test cases: (1) We use the merge function in **pandas** to join the data sets represented by pandas DataFrame objects. We do not consider the time to load the data set to DataFrame objects as we have seen the overhead in the previous section. That is, all data is already on the client side. Note that the client we use is as powerful as the machine that hosts PostgreSQL and AIDA. (2) **DB-UDF** assumes the data sets are stored in relational database tables within PostgreSQL. Therefore, it executes an SQL query joining the tables inside the UDF body and then loads the result set into a dictionary-columnar format. That is, data transformation is performed only once.

Regarding AIDA, the join operation will be translated into a SQL query on these two data sets. In terms of where the data sets reside, we consider the following three scenarios: (1) The data sets are stored as database tables in PostgreSQL. Therefore, AIDA simply pushes down a SQL query to the SQL engine. We refer to this case as **AIDA**. (2) The data sets to be joined reside in materialized TabularData objects that resulted from linear algebra operations. In order to perform the join, AIDA needs to expose these two data sets to PostgreSQL. As described in Section 3.1.3, we use table-UDFs to perform the data structure conversion and then return the Python object to PostgreSQL. We refer to this case as **AIDA-UDF**. (3) The data sets reside in materialized TabularData objects that resulted from relational operations; thus, they can have previously cached row-based result sets. We directly expose the cached row-based result set to PostgreSQL for the join. We refer to this case as **AIDA-cache**. To summarize, all AIDA variants produce a TabularData object with the dictionary-columnar representation. Therefore, **DB-UDF** and all AIDA variants have the overhead of transforming the row-based result set to a columnar format. **AIDA-UDF** has the additional overhead to transform columnar TabularData objects to rows and transfer the rows to PostgreSQL, while **AIDA-cache** only has the additional overhead to transfer the previously cached rows of AIDA's TabularData objects.

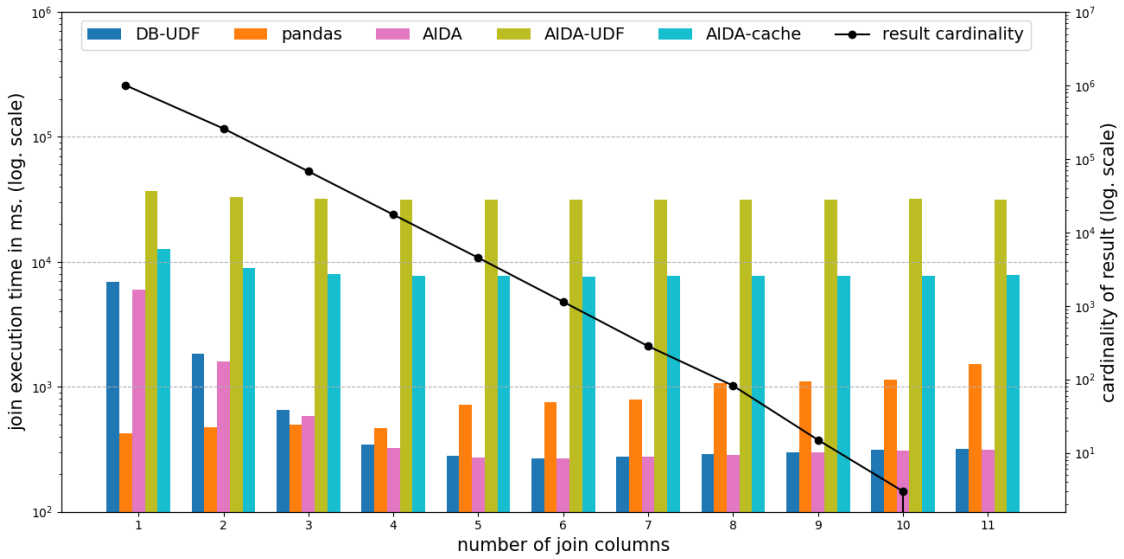

Figure 3.5: Joining two data sets

Figure 3.5 shows the join execution time in milliseconds (left axis) and the cardinality of the result set (right axis); both on a logarithmic scale. As the number of join columns increases, the join query becomes more complex and yields less result records. The result set has 1 million rows when we join only on the primary key and zero rows when we perform the join on all the columns.

Interestingly, with one to three join columns, **pandas** has the best performance. Such behavior can be attributed to the fact that pandas performs little optimization for the join operation. While the basic join plan is sufficient for simple joins, it fails to accommodate complex joins, even though **pandas** has no data conversion overhead as the data sets are already loaded into pandas DataFrame objects.

AIDA outperforms **pandas** when we join on more than 3 columns. The execution time of **AIDA** decreases from 1 to 5 join columns and increases only slightly after. The reason is that with few join columns the SQL result set is large and the row-to-column data structure conversion for the result set is expensive. The conversion cost decreases as the result set becomes smaller. Note that AIDA assumes the data sets reside in PostgreSQL

tables; thus, it can take advantage of PostgreSQL's sophisticated query planner for the complex joins. The only data structure conversion needed in this case is for the SQL result set.

DB-UDF has similar performance to **AIDA**, also executing the join within the RDBMS on database tables. While it does not have a framework overhead, it spends slightly more time than **AIDA** to finish the entire execution because **AIDA** benefits from the optimized conversion of the join result set into a dictionary-columnar structure.

For **AIDA-UDF** and **AIDA-cache**, we have to expose the TabularData objects to the database engine, which incurs extra overhead because MonetDB-style zero-copy data transfer is not possible. This cost is the dominant factor by far. Regarding **AIDA-UDF**, we barely see any difference in the execution time with different join columns because the actual join execution time and the final conversion of the SQL result set into a TabularData object play a much smaller role. For the key-join, **AIDA-UDF** takes 3.5 times longer than **AIDA**, and for an 11-column join, **AIDA-UDF** takes nearly 100 times longer, the only reason being the costly data transfer to the SQL engine.

With **AIDA-cache**, we do not need to perform a column-to-row conversion to expose the data sets; Therefore, we can see that **AIDA-cache** spends about 70% less time than **AIDA-UDF** in general. But we also notice that **AIDA-cache** still takes twice as much time as **AIDA** for a single-column join and 25 times longer for an 11-column join because of the significant overhead caused by sending data via table-UDFs to the SQL engine, even though the cached data is in a row-based format and compatible with PostgreSQL. This shows that the row/column data structure conversion is only one part of the problem. The other is that the interface used to feed data to PostgreSQL not only still copies the data but also does so on a record-by-record basis, making the data transfer very inefficient.

With these results, we can conclude that as long as the data resides in PostgreSQL, it is beneficial for AIDA to use the RDBMS's sophisticated join implementation for large data sets. However, it is not favourable to use a row-based RDBMS for relational operations if

the data to be manipulated does not reside in the database or cannot be exposed to it via a special mechanism such as zero-copy transfer. Even a simple join implementation that works directly on the data, such as implemented in pandas, will be more efficient as data conversion becomes the dominant factor.

Chapter 4

Exposing Python Data to PostgreSQL

In this chapter, we have a much closer look at the data transfer from the embedded Python environment to PostgreSQL and explore possibilities of optimizing SQL queries over Python data. Following the same optimization logic of virtual tables in [12], we want to implement the concept of virtual tables for PostgreSQL. First, we develop a foreign table-based approach that leverages PostgreSQL's foreign data wrapper feature to provide an optimizer-friendly data exposing mechanism with lazy strategies on data transfer. Moreover, we consider using temporary tables to present another approach to transfer the Python object to PostgreSQL. In order to understand the cost benefits of these designs, we evaluate these two methods against the conventional table-UDF approach that we used in our AIDA implementation described in the last chapter. We use the TPC-H Benchmark and an end-to-end data science workflow for our evaluation.

4.1 Virtual Table Designs for PostgreSQL

As pointed out in Section 2.3.2, the virtual table construct implemented in MonetDB has been proved to be an efficient data exposing mechanism when compared to table-UDFs in terms of running SQL queries over table-like data objects maintained in the embedded Python environment of an RDBMS.

To preserve the versatility of virtual tables, we adapt the same library design as in the MonetDB implementation: the functionality of virtual tables is implemented in a virtual table library that exposes a Python API to the client. This API can be used inside UDFs, stored procedures, or data science frameworks such as AIDA. The virtual table library allows for data registration and query execution, and it also separates the underlying data transfer implementation from the user interface. Users only need to change a flag attribute to alter the underlying data exposing mechanism; therefore, different implementations can be easily exchanged and tested.

Listing 4.1 shows a basic usage example of the virtual table library for PostgreSQL. The syntax is similar to the one used by the virtual table library implemented for MonetDB [12]. The virtual table manager object enables the registration of Python objects as virtual tables in the database system (`regTable`) and the execution of SQL queries on virtual tables (`executeQry`). It also enables that the result set returned is a Python object.

```
CREATE FUNCTION minor_students (filename TEXT)
RETURNS TABLE (sid INTEGER, sname TEXT) AS $$
    from vtlib import VTManager
    import pandas as pd
    vtm = VTManager()
    df = pd.read_csv(filename)
    ...
    # register the Python object as a virtual table
    vtm.regTable(df, 'students')

    # execute a SQL query over the registered data
    result = vtm.executeQry('SELECT sid, sname FROM students WHERE age < 18;')
    result = map(nameToUpper, result)
    return result
$$ LANGUAGE plpython3u;
```

Listing 4.1: An example of virtual table library usage

In contrast to MonetDB, PostgreSQL is a row-oriented system and requires further accommodation for data transfer. The architecture and design differences between these two systems make it impossible to directly migrate or have the same virtual table implementation for PostgreSQL as what was developed for MonetDB.

When implementing virtual tables for PostgreSQL, we face the challenge that a data-structure conversion is necessary if the registered object is stored in a column-based format, as this is the case for TabularData objects in AIDA or DataFrame objects in pandas, because PostgreSQL processes SQL query in a row-based manner. Thus, our design choices must take the impact of data structure conversion into consideration.

The internal implementations of these methods for PostgreSQL use two quite different mechanisms. The first one uses the foreign wrapper feature of PostgreSQL, which allows PostgreSQL to access external data via foreign tables. In the context of this foreign data wrapper approach, we have implemented several variations. Alternatively, as a more versatile approach, we also devise a temporary table approach that does not need to introduce extra PostgreSQL extensions but directly loads the Python data into temporary tables for short-term access. In the remainder of this section, we describe both approaches in detail.

4.1.1 The Foreign Table Approach

Figure 4.1 depicts the high-level architecture of the approach using foreign tables. The virtual table library provides a Python API to register a table-like Python object as virtual table as shown in Listing 4.1, concealing the details of how to do that from the user interface. At the time of registration, the virtual table library caches the table-like Python object with its registration name as a Python key-value pair. For each column of the data set, the library internally maps its Python data type to a database data type and uses this meta-information to create a foreign table in PostgreSQL's system catalog so that the virtual table can be used in SQL queries as if it were a regular database table.

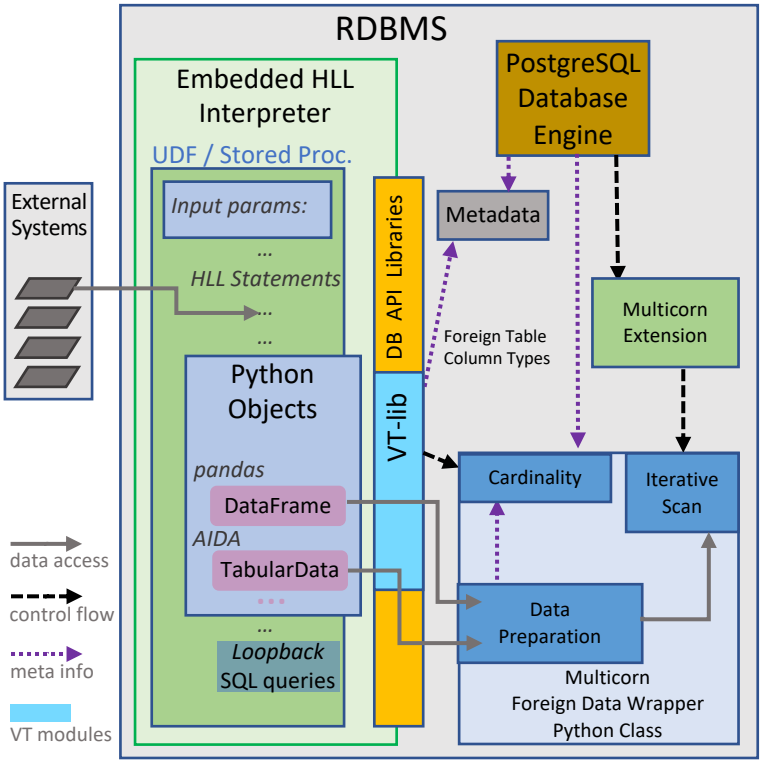

Figure 4.1: The foreign table approach

Using the foreign table approach as a virtual table implementation requires us to integrate the Multicorn extension and implement a Multicorn foreign data wrapper class. As described in Section 2.3.4, the API functions to be implemented in the Multicorn foreign data wrapper class facilitate both the query planning phase and query execution phase. We can expose the meta-information to the query planner and only transfer a subset of the registered data during the query execution. Therefore, the onus of regulating how to access the registered data lies on a concrete Multicorn foreign data wrapper class.

Multicorn Foreign Data Wrapper Class Initialization and Data Preparation

Although PostgreSQL obtains an abstract view of the Python data after the registration, the Multicorn foreign data wrapper class is only initialized when the foreign table is requested for the first time in a SQL query. In the constructor of the Multicorn foreign

data wrapper class, we find the cardinality of the registered data set. That is, the meta-information will be collected only once at the initialization time of the Multicorn foreign data wrapper class. The same applies to other data preparation operations that we only want to execute once for a foreign data set.

Since column-based Python data objects cannot be directly transferred to PostgreSQL's database engine, we need to convert the data set into a row-based format before offering the tuples to the Multicorn extension. We have analyzed several options. In the first implementation, we perform all the conversion in one shot at the initialization time of a Multicorn foreign data wrapper class and provide the full tuples to the SQL engine every time when the foreign table receives a scan request. In an alternative approach, we provide the SQL engine at query execution time with reformatted tuples that only contain the attributes needed for the query. We outline the approaches in detail further below.

Query Execution

As mentioned in Section 2.3.3, the execution of SQL queries over a foreign table can be divided into two phases: the query planning phase and the query execution phase.

At the query planning phase, the foreign data wrapper can use the column data type information to estimate the expected width of a tuple. In our specific Multicorn foreign data wrapper class, we have to implement the corresponding functions that provide such information to PostgreSQL's query optimizer. In particular, if the mapped PostgreSQL data type of a column has a fixed size, we directly add the size to the expected tuple width. Otherwise, we use a hard-coded constant value to estimate the width of a variable-length column and apply this estimated size to the expected tuple width. For each query planning inquiry from PostgreSQL's planner, we return the number of rows in the data set as well as the estimated tuple width of the columns that are needed to process the query for the query planner to estimate any plan cost.

PostgreSQL offers an iterative foreign scan interface to Multicorn. What we have to implement for a Multicorn foreign data wrapper class is a function that fetches one row at a time from the prepared row-based Python data set and then transfers the row to the Multicorn extension. Python data objects in that row are translated to C strings at the Multicorn extension before being sent to PostgreSQL SQL engine as pass-by-reference PostgreSQL *Datums*. However, such iteration and data type conversion are executed for every foreign scan of the foreign table. We want to explore the possibility of reducing the data transfer, and thus, the data type conversion overhead.

To analyze how the two aspects above will influence the performance of foreign tables, we have three different implementation variants, each using a different strategy for data format conversion and data transfer in the Multicorn foreign data wrapper class.

1. **The Trivial Implementation:** This implementation performs the column-to-row data format conversion on the entire Python data in one shot eagerly when the Multicorn foreign data wrapper class is initialized. The row-based data set is still a Python object, and this row-based Python data set will be used for subsequent query execution of the foreign table. The iterative foreign scan function sends one row of the Python data at a time to the Multicorn extension for data type conversion. The Multicorn extension converts the Python data to *Datum*, transforming the row into a virtual tuple table slot with pass-by-reference *Datum* attributes before sending it to PostgreSQL's database engine. This tuple table slot contains data for all the columns, no matter if some columns are required in the SQL query or not.

2. **The Need-Based Transfer Implementation**: This implementation also performs the column-to-row data format conversion at the Python level in one shot during the initialization of the Multicorn Foreign data wrapper class. However, when a particular SQL query invokes the foreign scan function, the foreign data wrapper class prepares each row with only the columns that are needed in the query as

projections are pushed down to the foreign data wrapper class. Therefore, only the data in the requested columns are sent to the Multicorn extension for data type conversion. The Multicorn extension then inserts nulls at the column positions that are not needed in the SQL query and provides the SQL engine with a tuple table slot that only has the requested data. Thus, less data is transferred at execution time.

3. **The Lazy Conversion Implementation:** This implementation performs a lazy data structure conversion for columns on an ad-hoc basis. It does not transform the columnar Python data set to a row-based structure at the initialization time. Instead, it performs the column-to-row conversion for the requested columns over time as queries are submitted for the foreign table. A first query on the foreign table only takes the columns that are requested by the query and transforms them into row format. Further queries then might add data from other columns to the cached rows whenever they are the first query that requests that column. Like the need-based data transfer implementation, this implementation only sends the requested columns in the cached rows to the Multicorn extension during the iterative foreign scan. The positions of other non-inquired columns in the virtual tuple table slots will be inserted with nulls before the Multicorn extension sends them to PostgreSQL's database engine.

Our foreign table approach performs the column-to-row data structure conversion for each column only once. It also provides cardinality information to PostgreSQL's query optimizer in the hope of a better execution plan and attempts to reduce the transfer load by only providing requested column data. Although it has such advantages over the table-UDF approach, data type conversion and data transfer from the embedded Python environment to PostgreSQL are still performed for each foreign scan request. Similar to the table-UDF approach described in the previous chapter, the foreign table approach also transfers the data on a per-record basis when it exposes a Python object to PostgreSQL.

4.1.2 The Temporary Table Approach

As pointed out in [12], loading the table-UDF's result set into a temporary table can circumvent the drawbacks of repeatedly using the table-UDF to a certain degree. The data transfer between the embedded HLL environment and the RDBMS only happens once, and the cardinality information will be known to the RDBMS afterward as the data set is stored in a temporary table. Therefore, the SQL queries over the temporary table can be optimized by the query planner. In this section, we use temporary tables to present a second data exposing mechanism to facilitate running SQL queries over Python objects.

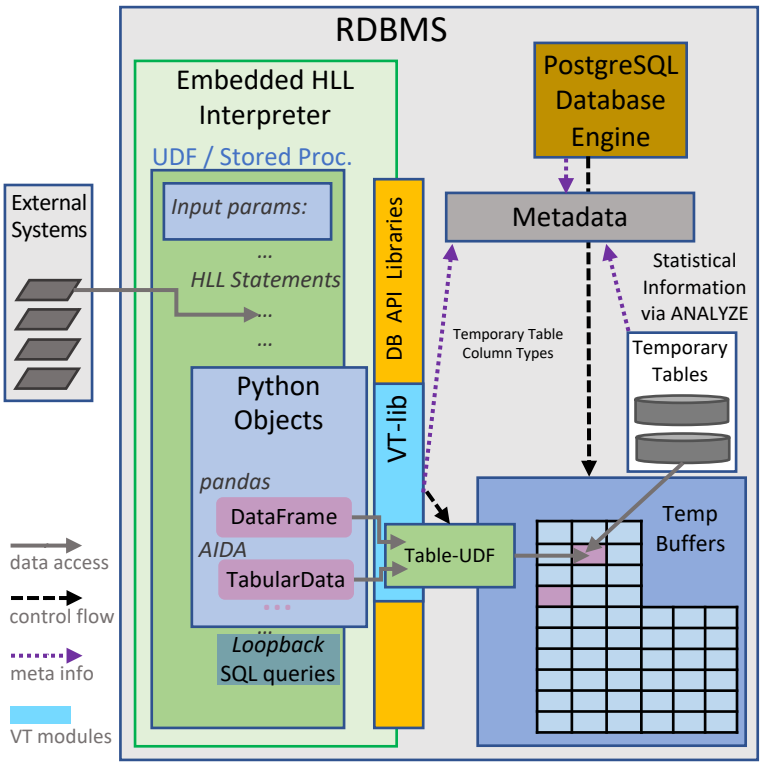

Figure 4.2: The temporary table approach

Figure 4.2 illustrates the high-level architecture of the temporary table approach. It is lightweight to use a temporary table for short-term access to registered data as a temporary table only exists for the database session in which it is created. Temporary tables have dedicated temporary buffers maintained by individual PostgreSQL backend

processes; they may have a higher cache hit rate in the temporary memory to access the data without being affected by operations on regular database tables that use shared buffers. Therefore, it is plausible to load the registered data from the embedded Python environment to a temporary table and then use it to query the data. This avoids the repeated data transfer overhead that occurs to both table-UDF and foreign table approaches when we run multiple SQL queries over the same registered Python object. Note that these temporary tables require extra space. Should memory space be too small, this might lead to swapping.

```
CREATE FUNCTION studentsIntermUDF()
RETURNS TABLE (sid INTEGER, sname TEXT) AS $$
    # data preparation
    result = ...
    return result
$$ LANGUAGE plpython3u;

CREATE TEMP TABLE students AS SELECT * FROM studentsIntermUDF();
SELECT sid FROM students WHERE sname = 'John Doe';
```

Listing 4.2: Temporary Table Approach

Listing 4.2 shows the high level flow of the temporary table-based approach. The virtual table library defines a table-UDF to return the Python object to the PostgreSQL database engine. It then uses the `CREATE TEMP TABLE AS` command to create a temporary table whose input is the result set of the Table-UDF. The temporary table's columns will have the same names and data types as in the output result of the SELECT command on the Table-UDF. When PostgreSQL executes the table-UDF, it transfers the Python data returned from the Table-UDF directly into the temporary table. Thus, data transfer only happens once when the temporary table is created. However, as we have discussed before, it leads to a data copy on a per-record basis. The temporary table can then be used in SQL queries, just as any other regular database table. For any SQL

queries over the temporary table, the database engine will directly access the tuples in the temporary buffers; this reduces the execution cost compared to our previous solutions, which transfer data from the embedded Python environment to PostgreSQL for every query execution.

Although the data that resides in a temporary table is known to PostgreSQL, it provides very little meta-information about itself to the query planner. PostgreSQL has a feature called *autovacuum*, whose purpose is to automate the execution of `VACUUM` and `ANALYZE` commands [32]. The *autovacuum* daemon maintains and updates the statistical information of database tables. It is automatically executed after a large number of inserted, updated, or deleted tuples. First, `VACUUM` collects the storage space occupied by the table tuples that are either deleted or obsoleted by an update. Then, `ANALYZE` collects statistical information from the cleaned tuples of the table. The collected statistical information of that table is stored in PostgreSQL's system catalog and can be used by the query planner to generate an optimal execution plan. Because the *autovacuum* daemon cannot access the temporary tables [32], PostgreSQL's query planner by default estimates the characteristics of a temporary table by counting how many buffer pages a temporary table occupies and by using the column data type information to estimate the cardinality of the temporary table. However, the estimation often diverges from the actual value, especially for temporary tables with variable-length columns. Ideally, the query planner needs more accurate and extensive statistical information about a temporary table to generate the optimal execution plan.

To collect the statistical information of a temporary table and its columns, we can explicitly run an `ANALYZE` command on that temporary table. Once the statistics about the temporary table is added into PostgreSQL's system catalog, the query planner can use the meta-information to generate an efficient execution plan for SQL queries over the temporary table. Depending on the size of the temporary table, collecting statistical metadata may be expensive. We are interested in evaluating the trade-off between using

a simpler execution plan to access the temporary table data vs. spending extra time to collect temporary table statistics in the hope of executing the SQL query with an optimized plan.

4.2 Evaluation

4.2.1 Test Setup

Our implementations provide alternative approaches to expose Python data to PostgreSQL, and we would like to evaluate the usability of these implementations and compare their performance with the conventional table-UDF approach over practical test cases. AIDA, as a data science framework, supports relational operations by running SQL queries over Python data objects. By default, it uses Table-UDFs as the default data exposing mechanism. For the purpose of the evaluation, we replace the table-UDF generation component with the virtual table library and evaluate our implementations with test cases of business-oriented queries in the TPC-H Benchmark and an end-to-end data science workflow that requires frequent data movement between PostgreSQL and the embedded Python environment. We measure the performance in terms of the response time of operation requests with each data exposing approach.

We use different data exposing mechanisms in AIDA's server component to run the experiments and set up the client-server connection on the same nodes as mentioned in Section 3.2.1: The hardware specifications are Intel® Xeon® CPU E3-1220 v5 @ 3.00GHz and 32 GB DDR4 RDIMM. The node that runs PostgreSQL with AIDA's server component and the other node that runs the AIDA client are connected via a Dell™ PowerConnect™ 2848 switch in a Gigabit private LAN. For software, we use PostgreSQL 12.3, Python 3.5.2, NumPy 1.12.1, pandas 0.23.3, psycopg2 2.7.5, and Multicorn 1.4.0. Unless explicitly specified, we use default settings for all software. In all the experiments, we only start to take time measurement after a warm-up phase.

4.2.2 TPC-H Queries

To evaluate the usability of our data exposing implementations for PostgreSQL and compare them with table-UDFs, we first run experiments with the TPC-H Benchmark [33]. The TPC-H Benchmark captures a Business-to-Consumer system consisting of tables such as customer, orders, line-item, etc. The database can be created with different scale factors (SFs), where SF 1 leads to a database of roughly 1 GB. The TPC-H queries are modelled after real-world commercial questions, which are complex and made up of a variety of relational operations. Notably, the richness of selectivity conditions in these queries will help us better understand how the data characteristics will influence PostgreSQL's query planner. With regard to the scale factor of the TPC-H Benchmark, we choose to use SF 1 in our experiments as 1 GB of data is already large enough to analyze how the data transfer affects the performance of virtual tables. We use AIDA's ORM-style API to run Python workflows that are translated from TPC-H SQL queries 1-20 and 22. Query 21 is left out because it cannot be translated by using AIDA's syntax. We set up AIDA's server component with the following data exposing mechanisms:

- **db-tbl**: The data set is in regular database tables. No data conversion takes place when running queries over those tables, and PostgreSQL has full access to the characteristics of these tables. This is our base case, which has the lowest overhead by construction.
- **tbl-udf**: The data set is stored in Python objects, and AIDA uses table-UDFs to expose the data to PostgreSQL.
- **ft**: The data set is stored in Python objects, and AIDA uses the trivial implementation of foreign tables to expose the data to PostgreSQL. The foreign data wrapper eagerly transforms the columnar object to a row-based format at the initialization time and transfers all the data to PostgreSQL for each query request.

- **ft-needed-cols**: The data set is stored in Python objects, and AIDA uses the needed-based implementation of foreign tables to expose this data to PostgreSQL. The foreign data wrapper class still eagerly transforms the columnar object at the initialization time but only transfers the requested columns to PostgreSQL at the runtime of a SQL query.

- **ft-lazy**: The data set is stored in Python objects, and AIDA uses the lazy conversion implementation of foreign tables to expose this data to PostgreSQL. The foreign data wrapper class lazily performs the column-to-row data transformation for the columns when they are requested for the first time and caches them in a row-based structure for later use. This implementation also only transfers the requested columns for a query request.

- **tp**: The data set is stored in Python objects, and each Python object is loaded into a temporary table by using a table-UDF. PostgreSQL will then use the temporary table to access the data.

- **tp-analyzed**: After the data is loaded into the temporary table, we run an `ANALYZE` command on the temporary table to collect statistical information from the temporary table.

An individual test unit consists of one of the TPC-H queries and one of the implementation options described above. Within each test unit, the query is executed twice. We measure the execution time for each run separately and show the results with suffixes of -1 and -2, respectively. The virtual tables requested in the query are created when the query is executed for the first time. Thus, the first two foreign table variants perform the column-to-row transformation on the entire data set in the first run of the query. In contrast, the third foreign table variant performs the transformation lazily for the columns that are requested in the query during the first foreign table scan. Similar to foreign tables, the temporary table approach only loads the data into a temporary table

at its creation time. Therefore, we expect the second run of each query to take less time for our virtual table implementations. Note that the execution time does not include any data transfer from the server to the client side; the query execution solely occurs on the server side. Since we experiment with AIDA, we include the time spent on loading the SQL result set into a TabularData object. This overhead occurs for all test candidates.

All Data Sets as Python Objects

In our first test case, we load all the TPC-H data sets into AIDA's TabularData objects in the dictionary-columnar representation. All test candidates except **db-tbl**, where the data is stored in regular database tables, have the Python objects at the start-point.

Figures 4.3 - 4.5 depict each query's execution time with different data exposing mechanisms in seconds (y-axis in a logarithmic scale). Although most queries follow the same trend, there are three queries that deviate from others (i.e. queries 2, 19, and 22). We will discuss these three queries case by case later.

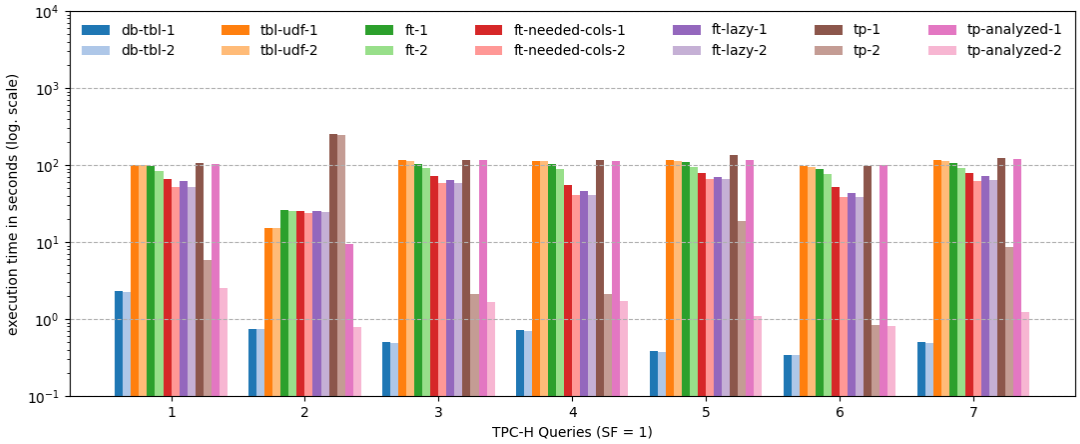

Figure 4.3: TPC-H queries: 1-7
(all data sets as Python objects).

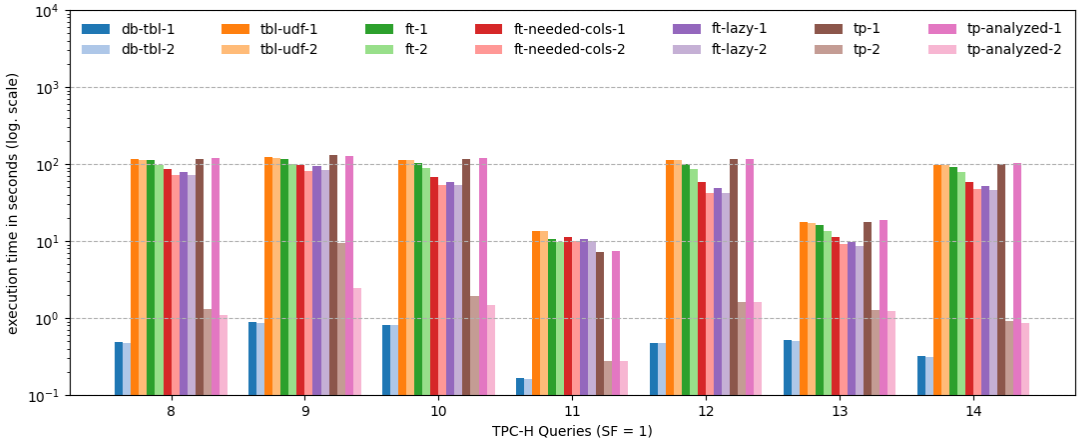

Figure 4.4: TPC-H queries: 8-14
(all data sets as Python objects).

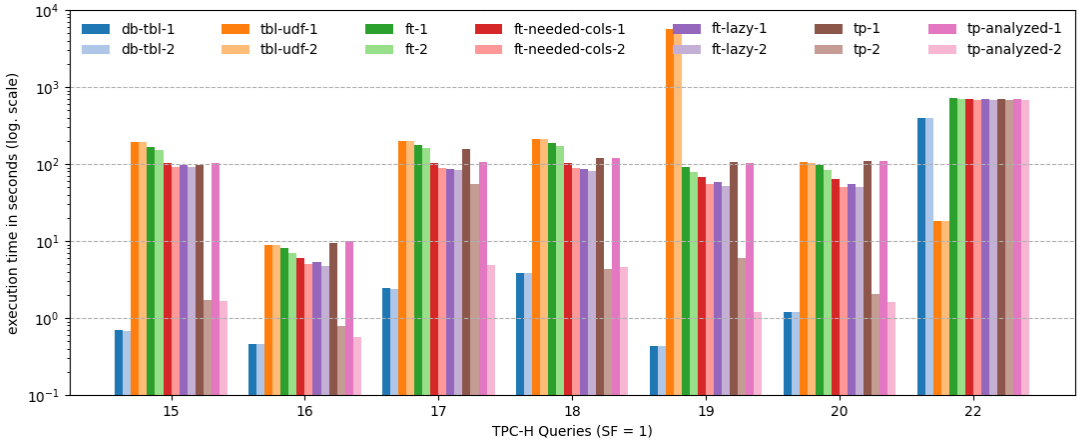

Figure 4.5: TPC-H queries: 15-20, and 22
(all data sets as Python objects).

Overall Trend: For the present, we confine ourselves to the 18 queries that follow the general trend. In the following, when we discuss the execution time, we calculate them as the sum of the execution times of all 18 queries.

We have the best performance when running the queries directly on database tables (**db-tbl**), with a total execution time of 17 seconds. This comes as no surprise because no data conversion is needed and the query planner has full access to precise

meta-information. In contrast, Table-UDFs, bearing the deficiencies mentioned in Section 2.3.1, are relatively expensive to use. The first runs of table-UDFs (**tb-udf-1**) take about 1960 seconds in total, which is 115 times more than running the queries with regular database tables on average. The execution time difference between the first and second runs of Table-UDFs is not significantly noticeable because PostgreSQL needs to re-perform all the expensive operations, such as data structure transformation and data type conversion for both executions of the query.

Regarding our implementations of virtual tables, we start by looking at the trivial foreign table implementation (**ft-1** and **ft-2**). The first executions take approximately 105 times more than the regular database tables, which is also about 90% of the execution time of table-UDFs. We determined that running the same query with table-UDFs and foreign tables can result in different execution plans. The reason is that foreign table-based implementations inform PostgreSQL's query planner of the cardinality information about the data sets, whereas table-UDFs cannot foresee how many rows they will return. The query planner uses a constant value of 1,000 rows to estimate the output of table-UDFs. Since table-UDFs and the naive foreign table implementation perform similar conversions in the first execution, the performance improvement from using foreign tables can be attributed to the fact that the query planner can benefit from the cardinality information provided by the foreign tables and generate efficient plans. Since the column-to-row data transformation of the Python data is only performed once when the foreign table first appears in a query call (**ft-1**), the subsequent executions (**ft-2**) do not need to repeat the transformation. Thus, we can see there is a 12% decrease in execution time when we compare the second executions to the first runs when using the naive foreign table implementation.

Our two other foreign table variants aim at further reducing the data transfer overhead. These two variants will have the same execution plan as the naive foreign table approach because they provide the same cardinality information to the query

planner. Therefore, they optimize the data transfer process in seek of performance improvement. Although the first optimized variant (**ft-needed-cols**) still performs the column-to-row data structure transformation in one shot, it only transfers the tuples with the columns that are requested in a query during the foreign table scan. Not transferring the columns that are not needed in the queries reduces the execution time of first runs (**ft-needed-cols-1**) to 1170 seconds, which is roughly 35% faster than naive foreign tables and 40% faster than Table-UDFs. We can see the same improvement in the second runs as well. The last implementation variant (**ft-lazy**) performs both the column-to-row data structure transformation and the data exposing process lazily. There is a further 10% improvement when we compare **ft-needed-cols-1** and **ft-lazy-1**, as we avoid adding unnecessary columns into the row-based data structure cached by the foreign data wrapper class. These two variants have roughly the same execution times for the second runs as they share the same data transfer optimization strategy.

For the temporary tables, we set the size of temporary buffer space to 128 MB, which is the same as the default size of the shared memory buffers in PostgreSQL; we do so to reduce the influence of disk I/O requests when we compare using regular database tables and temporary tables. The two temporary table-based approaches take a very long time for the first runs. In contrast to the other approaches, where we directly run the SQL queries over either table-UDFs or foreign tables, the first query execution of the temporary table-based approach consists of two separated parts: first load Python data sets into temporary tables via Table-UDFs, and then run SQL queries over these temporary tables. In the second run, the execution time is purely the time spent on running the SQL query over the data stored in these temporary tables. The difference between the first and the second run reveals the time spent on loading Python data into temporary tables via Table-UDFs. In the first runs of the vanilla temporary table approach (**tp-1**), loading the data into temporary tables costs around 1,670 seconds in total, taking up 93% of the execution time. Nevertheless, when we compare the first runs of the temporary table approach (at

1785 seconds in total) and the table-UDF approach (at 1960 seconds in total), we observe a 9% drop in the overall execution time, given that the temporary table approach performs the data transfer exactly once even if the data set could be requested multiple times in some queries.

Remember that the temporary table does not store the data size information by default. PostgreSQL's query planner estimates the temporary table's row number based on how many pages it occupies. However, this estimation is not always accurate and will not always lead to an efficient execution plan. Therefore, we have a second solution that explicitly runs an `ANALYZE` command to collect the statistical information about a temporary table and its columns. The time spent on running `ANALYZE` commands on the temporary tables is about 25 seconds, which is merely 1.5% of the total execution time of **tp-analyzed-1**. In addition, with the collected statistical information, running the SQL queries over the temporary tables alone is 70% faster than the vanilla temporary table approach and more than 30 times faster than the foreign table approaches in the second runs (**tp-analyzed-2**). Therefore, this approach is favourable when we want to execute SQL queries over the exposed data multiple times as we only transfer the data once instead of repeating data transfer for each request from the SQL engine.

Although temporary tables have dedicated memory buffers, the performance of the second runs on the temporary tables is not always close to running the query with regular database tables. The scans of temporary tables are currently not performed in parallel in PostgreSQL, whereas the pages of a regular database table can be divided among parallel worker processes for scanning. Thus, the second execution of using analyzed temporary tables (**tp-analyzed-2**) is still roughly 50% slower than directly running the queries on regular database tables. Nevertheless, as data transfer is avoided at the second run, this is more optimal than the other data exposing mechanisms.

Outlier Queries: There are three queries that deviate from the general trend (i.e. queries 2, 19 and 22). We will have a detailed look at them in this subsection.

Running query 2 with table-UDFs is faster than most of our optimization methods except **tp-analyzed**. The reason is that PostgreSQL's inaccurate selectivity estimates coincidentally yield an optimal execution plan for table-UDFs. The TPC-H query 2 re-written in AIDA syntax consists of multiple joins. PostgreSQL's query planner cannot predict the exact number of rows that will be produced by a join or a selection unless it has the relevant statistical information about the attributes in the condition. Although foreign tables provide the cardinality information for the data set, none of these data exposing mechanisms provides statistical information about columns in the data sets. Therefore, the query planner might yield inaccurate selectivity estimates and choose a sub-optimal plan. In fact, when we run query 2 with foreign tables or temporary tables that are not analyzed, PostgreSQL ends up performing nested loop joins on the intermediate result sets that the query planner estimates to be small but are actually large. In contrast, tables-UDF is perceived as a black-box that constantly returns 1,000 rows to the query planner regardless of how many rows the data set actually contains. With this cardinality information, the query execution plan of query 2 happens to be a chain of hashed joins between table-UDFs and intermediate join results, which leads to better performance. As the analyzed temporary table approach provides statistical information about the columns to PostgreSQL, it benefits from an efficient execution plan. Even bearing the data transfer time in the first run, it is still faster than the other data exposing mechanisms for query 2.

When looking at query 19, we find it takes surprisingly long if we run the query with table-UDFs. This query involves joining two large data sets. When we run the query with table-UDFs, the query planner perceives that the two function scans will only produce 1,000 rows each and decides to use a nested loop to join these two data sets. However, the two data sets actually have 800,000 rows and 6,001,215 rows, respectively. Performing a nested loop join on these two data sets is a very expensive operation and leads to an inefficient execution plan. All of our virtual table implementations avoid the

expensive nested loop join because they expose the correct cardinality information to the query planner.

Lastly, it is unexpected to see that running query 22 with regular database tables is slower than using table-UDFs. After digging into the execution plans, we find that the problem is related to how PostgreSQL handles the NOT IN operator. In query 22, there is an operation that selects the rows with an attribute that is not in another data set. The data set in the NOT IN operator has 1,500,000 rows; thus, it is large. To perform the NOT IN operation, PostgreSQL is inclined to build a hash table for the data inside the NOT IN operator and traverses the outer row set to verify if the attribute is in the hash table. However, if the query planner finds that the size of the hash table exceeds the *work_mem* size in PostgreSQL's configuration file, it will use a naive NOT IN validation in the execution plan instead. Since regular database tables, foreign tables, and temporary tables provide the cardinality information to PostgreSQL's query planner, they are all affected by this decision rule and run the query with the expensive naive NOT IN validation. Table-UDFs, on the other hand, only provide a constant value of 1,000 rows to the query planner despite that this number diverges from the real size of the data sets. Therefore, running this query on Table-UDFs will follow an execution plan that uses a hash table to perform the NOT IN operation. And it turns out that this is more efficient than the naive solution despite the lack of memory space for the hash table.

One Data Set as a Python Object

In order to understand the performance impact for scenarios where there are fewer Python data objects involved in a SQL query, we run the experiments where only one data set is loaded as a Python object. The other data sets are still stored as regular database tables in PostgreSQL. Considering the impact of the data size on data transfer, we load a small data set, `nation`, and a large data set, `lineitem`, as a Python object,

respectively. In this test case, we re-run TPC-H queries 5, 7, 8, 9, and 10, which have these two data sets in common, and analyze their execution costs.

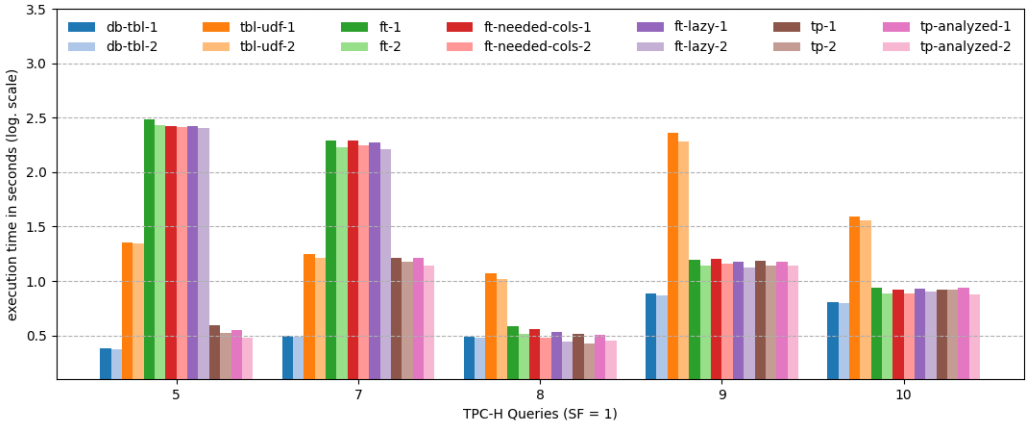

Figure 4.6: TPC-H queries 5, 7, 8, 9, and 10
(`Nation` as a Python object).

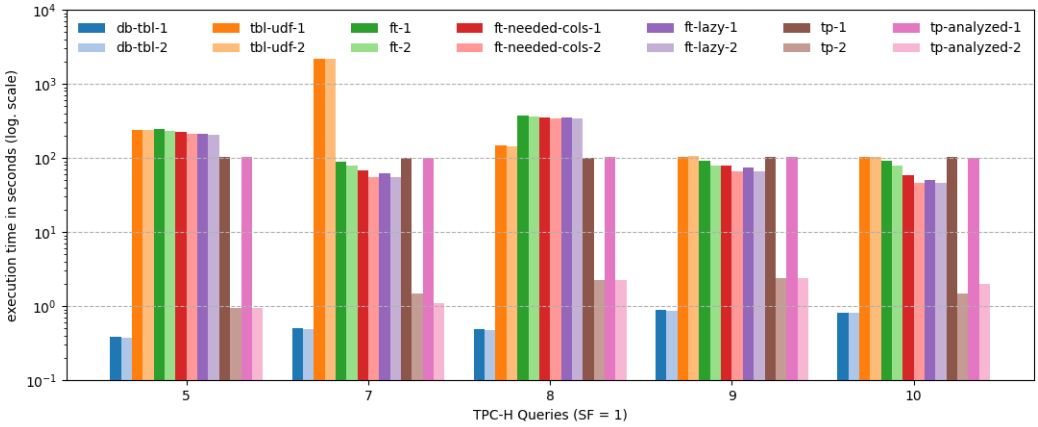

Figure 4.7: TPC-H queries 5, 7, 8, 9, and 10
(`Lineitems` a Python object).

To start with `nation`, we load it as a Python object but keep all other tables within the database and re-run the 5 queries mentioned above. Figure 4.6 shows the execution times of each query in seconds. The `nation` data set only has 25 rows and 4 columns, and thus, it introduces very little data conversion and transfer overheads. We can see that

there is no significant difference between the execution costs of the first and second runs of any particular test candidate. Although our virtual table implementations outperform table-UDFs in queries 8, 9 and, 10, there exist scenarios like queries 5 and 7, where using a foreign table is slower than a table-UDF. PostgreSQL's query planner assumes every table-UDF would produce exact 1,000 rows. Given that this estimate is relatively close to the actual row number in the scale of RDBMS, it is not clear which approach will lead to an optimal execution plan.

In contrast to `nation`, `lineitem` is a large data set consisting of 6,000,125 rows. The execution cost of running the queries over the loaded Python object is shown in Figure 4.7, with the y-axis on a logarithmic scale. We notice that the overall trend that we have seen in the previous test case can also be observed in this experiment. Using the foreign table-based approach can still lead to a sub-optimal execution because of PostgreSQL's query planner's inaccurate selectivity estimates of intermediate operations. This holds true for query 8. But summing up the execution times of all the queries, the trivial foreign table approach is up to 70% faster than table-UDFs. As expected, a large data set introduces more data conversion and transfer overheads. We see a 14% drop in total execution time by only transferring the columns that are requested in SQL queries in **ft-reduced-cols** and **ft-lazy** compared to the trivial foreign table approach (**ft**) in the second runs. Also, the first runs of **ft-lazy** are 5% faster than **ft-reduced-cols** as it performs the column-to-row data structure conversion lazily. For the temporary table-based implementations, there is no significant difference in the query costs no matter whether we analyze the temporary tables or not. Loading the data into temporary tables takes up around 98% of the execution costs in the first runs. If we only look at the time spent on running the queries over temporary tables in the second runs, it is over 300 times faster than table-UDFs and relatively close to using regular database tables.

To summarize, cardinality information is crucial to the query planner when it comes to execution plan decision-making. This is not only limited to the Python objects that we

want to expose to PostgreSQL but also applies to the selectivity estimates of result sets produced by intermediate operations. The query planner tends to yield an efficient execution plan if we provide cardinality information to it. Compared to table-UDFs, our foreign table-based implementation can optimize SQL queries over Python objects because it can provide accurate cardinality information and applies an ad hoc data transfer strategy. However, a significant amount of time is still spent on transferring the data from the embedded Python interpreter to the SQL engine for each query request. On the other hand, the temporary table-based implementation eliminates the overhead of recurrent data transfer in subsequent query invocations. Furthermore, it can collect additional statistical information to facilitate generating execution plans for complex SQL queries.

Nevertheless, as observed several times in this thesis, the iterative data transfer approach in which tuples have to be copied and fed to PostgreSQL, no matter which interface is used (table-UDF, foreign table, or temporary table), leads to the dominant performance impact.

4.2.3 Data Science Workflows

AIDA supports complex analysis such as end-to-end data science workflows, which usually perform linear algebra and relational operations collaboratively. In such workflows, AIDA needs to translate the relational operations into SQL queries and move data back and forth between the embedded Python environment and PostgreSQL for in-database analytics. In this test case, we experiment with the short BIXI workflow from [11], which uses bicycle trip data[1] to predict the duration of a trip given the trip distance. In the short BIXI workflow, the data scientist is assumed to have domain knowledge on what data sets and features are required to train the machine learning model. The data sets used in this workflow are stored as distinct tables in the database.

[1] `https://www.kaggle.com/aubertsigouin/biximtl`

In order to obtain the prediction variables and target feature, the data scientist uses both relational and linear algebra operations to build the training and test sets in AIDA. These two data sets are then used to train and evaluate a linear regression model. To analyze how AIDA performs on such a data science workflow with different data exposing mechanisms, we only measure the execution time of the SQL queries that run on Python objects.

Because data science workflows are usually written with an interleaved combination of linear algebra and relational operations, it is unlikely to find many complex relational operations in such workflows. In fact, the SQL queries that are translated from the operations in the BIXI workload are relatively simple compared to TPC-H queries; they merely have a maximum of three data sets being joined. When looking at the entire workflow, there is a total of 19 Python data sets that are being exposed to PostgreSQL to run SQL queries over. Most of them are numerical result sets that are accessed by PostgreSQL's SQL engine only once, with the exception of two data sets that appear in SQL queries twice and three times, respectively. The first one is a data set with 4,880 rows of 1 column, and the second has 2,256,283 rows and 4 columns.

Running all SQL queries found in the BIXI Workload using Table-UDFs takes 87.638 seconds. When we use the trivial foreign table approach, the cost comes down to 83.578 seconds. The other two foreign table-based virtual table implementations both see a 5% decrease compared to the trivial foreign table approach due to the same data transfer optimization strategy. As for the temporary table-based virtual table implementations, the measured time consists of two parts: loading Python data into PostgreSQL's temporary buffers via Table-UDFs and running the translated queries on the temporary tables. Using the temporary table approach yields a total execution time of 62.757 seconds, or 63.636 seconds if we run the queries with analyzed temporary tables and include the time spent on collecting statistical information. This is quite remarkable as only 2 queries benefited from the objects already being stored in temporary tables.

With these results, we can conclude that these translated queries are simple enough that the query planner cannot benefit from additional meta-data other than the cardinality information. The real execution burden is on the data transfer process. When we use a table-UDF or a foreign table to access a Python object, the data must be transferred to the SQL engine for every query request. Although the foreign table variants have an ad hoc data transfer strategy, the data type conversion that occurs in each data transfer process is still remarkably expensive. To circumvent the recurrent data transfer, we can use the temporary table-based implantation where the data is only transferred to PostgreSQL once at the creation time of the temporary table. With this approach, we can avert the repetition of data transfer and boost the query performance, especially if the SQL engine needs to access an enormous data set more than once. We see that even if this does not happen frequently, it can benefit overall execution.

Chapter 5

Conclusions & Future Work

In this chapter, we will summarize the achievements of this thesis, present our findings, and lastly, discuss possible directions for future work.

5.1 Conclusions

In this thesis, we extended the Python-based AIDA framework for the row-based RDBMS PostgreSQL to support interactive in-database analytics. Because PostgreSQL's internal data representation is not aligned with the statistical package used in AIDA, we cannot benefit from the data transfer optimizations that were possible with the original implementation of AIDA for MonetDB [11]. While AIDA can still switch between linear and relational algebra operations, it is very expensive to convert the row-based PostgreSQL data to the columnar Python data representation suitable for linear algebra operations and vice versa. To address this problem, we designed a database adapter interface to facilitate transferring data from PostgreSQL to AIDA's workspace for executing analytical tasks. We have devised optimized data conversion methods that directly work on the underlying structure of the transferred data in C. Through experiments, we have proven that such data transfer inside AIDA is more efficient than

loading PostgreSQL data into a database-external environment to form a computational object for data analytics.

In the other direction, we find that there are certain limitations to transferring Python data to PostgreSQL by means of table-UDFs. Although AIDA leverages PostgreSQL's database engine to execute relational operations, the costly performance of the SQL queries that run over table-UDFs makes the use of PostgreSQL's database engine less attractive. To optimize running SQL queries over Python data, we developed two optimizer-friendly data-exposing mechanisms for implementing the virtual table concept proposed in [12]: the first one is based on PostgreSQL's foreign data wrapper feature, and the second one uses temporary tables to hold the data. They provide an alternative approach to expose table-like Python data to PostgreSQL and facilitate running SQL queries over them. Although these two designs also transfer data on an iterative record-by-record basis, both of them can provide data characteristics to PostgreSQL's query planner in the hope of an optimized execution plan. The foreign table-based implementations transfer the data in each column on an ad hoc basis to reduce the data transferring overheads in each foreign table scan. Although there are still scenarios that the query planner will produce a sub-optimal execution, using foreign tables to expose Python data to PostgreSQL tends to have better performance compared to the conventional Table-UDF approach. As another option, the temporary table-based approach only transfers the Python data to PostgreSQL once and stores the transferred data in temporary buffers for later query requests. If a Python object is to be accessed in multiple SQL queries, it is favourable to use this method to avoid repeated data transfer overheads.

5.2 Future Work

While bringing data into another environment to perform relevant operations, we want to avoid iteratively copying the data on a record-by-record basis. One promising optimization would be to modify the memory management component of the database system to perform a bulk transfer of the data between PostgreSQL and AIDA's workspace. However, how to work around the challenge of row/column data structure conversion has to be further explored. Another optimization possibility is to divide the overall data transfer process and execute sub-tasks in parallel. Such a parallel copy mechanism can help to boost the overall performance.

Looking at another direction that targets the optimization of in-database analytics in general, it seems promising if we can reduce the amount of data transferred across the two environments to perform interleaved relational and numerical computations. As in the case of the optimized foreign data wrapper, we have seen the performance improvement of only transferring the columns of the Python data that are needed in a SQL query. Therefore, we can possibly apply a similar ad hoc data transfer strategy to transfer only a subset of the rows by verifying query conditions inside the foreign data wrapper. This approach reduces data transfer overheads by only sending the qualified Python rows to PostgreSQL's database engine. However, it would be necessary to analyze the exact cost benefits of pushing down query conditions to the Python data set versus leveraging PostgreSQL's database engine to validate each individual row from the transferred data.

At the most extreme, we can seek efficient Python relational operator implementations for AIDA's columnar data structure so that we do not need to expose Python data to PostgreSQL at all and perform relational operations within Python. It is also worth trying to only transfer Python data for complex queries, where PostgreSQL's SQL engine significantly outperforms the implementations that exist within Python, making the data transfer worthwhile. In addition, we can track data lineage in AIDA to

avoid unnecessary data transfer. For any operation, if it requires a data set that can be reassembled from other data sets in the working environment, we can try to form such a new data set without loading the data from another environment.

Bibliography

[1] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark SQL: Relational data processing in Spark. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1383–1394, 2015.

[2] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, et al. System R: Relational approach to database management. *ACM Transactions on Database Systems (TODS)*, 1(2):97–137, 1976.

[3] D. D. Chamberlin and R. F. Boyce. Sequel: A structured english query language. In *Proceedings of the ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control*, pages 249–264, 1974.

[4] S. Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 34–43, 1998.

[5] Q. Chen, M. Hsu, and R. Liu. Extend UDF technology for integrated analytics. In T. B. Pedersen, M. K. Mohania, and A. M. Tjoa, editors, *Data Warehousing and Knowledge Discovery*, pages 256–270, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

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

[7] E. F. Codd. *Relational completeness of data base sublanguages*. IBM Corporation, 1972.

[8] M. Deng. Using foreign data wrapper in PostgreSQL to expose point clouds on file system. Master's thesis, Delft University of Technology, Delft, Netherlands, 2020.

[9] O. Dolmatova, N. Augsten, and M. H. Böhlen. A relational matrix algebra and its implementation in a column store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 2573–2587, 2020.

[10] J. D. Drake and J. C. Worsley. *Practical PostgreSQL*. O'Reilly Media, Inc., 2002.

[11] J. V. D'silva, F. De Moor, and B. Kemme. AIDA: Abstraction for advanced in-database analytics. *Proceedings of the VLDB Endowment*, 11(11):1400–1413, 2018.

[12] J. V. D'silva, F. De Moor, and B. Kemme. Keep your host language object and also query it: A case for SQL query support in RDBMS for host language objects. In *Proceedings of the 31st International Conference on Scientific and Statistical Database Management*, pages 133–144, 2019.

[13] A. Eisenberg. New standard for stored procedures in SQL. *ACM SIGMOD Record*, 25(4):81–88, 1996.

[14] Z. Fong. The design and implementation of the POSTGRES query optimizer. *MS Report, University of California, Berkeley, CA*, 1986.

[15] S. Harizopoulos, V. Liang, D. J. Abadi, and S. Madden. Performance tradeoffs in read-optimized databases. In *Proceedings of the International Conference on Very Large Data Bases*, pages 487–498, 2006.

[16] J. M. Hellerstein, M. Stonebraker, and J. Hamilton. *Architecture of a database system*. Now Publishers Inc, Hanover, MA, USA, 2007.

[17] S. Idreos, F. Groffen, N. Nes, S. Manegold, S. Mullender, and M. Kersten. MonetDB: Two decades of research in column-oriented database. *IEEE Data Engineering Bulletin*, 2012.

[18] M. Jarke and J. Koch. Query optimization in database systems. *ACM Computing surveys (CSUR)*, 16(2):111–152, 1984.

[19] D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys (CSUR)*, 32(4):422–469, 2000.

[20] Kozea. Multicorn, 2014. https://multicorn.readthedocs.io/, accessed December 11, 2020.

[21] J. Lajus and H. Mühleisen. Efficient data management and statistics with zero-copy integration. In *Proceedings of the 26th International Conference on Scientific and Statistical Database Management*, pages 1–10, 2014.

[22] M.-A. Lemburg. Python Database API Specification v2.0, 2001.

[23] V. Linnemann, K. Küspert, P. Dadam, P. Pistor, R. Erbe, A. Kemper, N. Südkamp, G. Walch, and M. Wallrath. Design and implementation of an extensible database management system supporting user defined data types and functions. In *Proceedings of the International Conference on Very Large Data Bases*, pages 294–305, 1988.

[24] W. McKinney et al. pandas: a foundational Python library for data analysis and statistics. *Python for High Performance and Scientific Computing*, 14(9):1–9, 2011.

[25] S. Melnik, A. Adya, and P. A. Bernstein. Compiling mappings to bridge applications and databases. *ACM Transactions on Database Systems (TODS)*, 33(4):1–50, 2008.

[26] J. Melton, J.-E. Michels, V. Josifovski, K. Kulkarni, P. Schwarz, and K. Zeidenstein. SQL and management of external data. *ACM SIGMOD Record*, 30(1):70–77, 2001.

[27] C. Ordonez and J. García-García. Vector and matrix operations programmed with UDFs in a relational DBMS. In *Proceedings of the ACM International Conference on Information and Knowledge Management*, pages 503–512, 2006.

[28] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/rule based query rewrite optimization in Starburst. *ACM Sigmod Record*, 21(2):39–48, 1992.

[29] M. Raasveldt. Vectorized UDFs in Column-Stores. Master's thesis, Utrecht University, Utrecht, Netherlands, 2015.

[30] J. Roijackers. Bridging SQL and noSQL. Master's thesis, Eindhoven University of Technology, Eindhoven, Netherlands, 2012.

[31] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: A column-oriented DBMS. In *Proceedings of the International Conference on Very Large Data Bases*, pages 553–564, 2005.

[32] The PostgreSQL Global Development Group. Postgresql 12.3 documentation, 2020.

[33] Transaction Processing Performance Council. TPC Benchmark H, 2017.

[34] S. Van Der Walt, S. C. Colbert, and G. Varoquaux. The NumPy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011.

[35] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica, et al. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[36] Y. Zhang, M. Kersten, M. Ivanova, and N. Nes. SciQL: bridging the gap between science and relational DBMS. In *Proceedings of the 15th Symposium on International Database Engineering & Applications*, pages 124–133, 2011.

Acronyms

AIDA Abstraction for Advanced In-Database Analytics.

API Application Programming Interface.

CSV comma-separated values.

HLL high-level language.

LAN Local Area Network.

OLAP Online Analytical Processing.

OLTP Online Transaction Processing.

ORM Object-Relational Mappings.

RDBMS Relational Database Management System.

RMI Remote Method Invocation.

SF Scale Factor.

SQL Structured Query Language.

SQL/MED SQL Management of External Data.

UDF User Defined Function.