AIDA - Abstraction for Advanced In-Database Analytics

Joseph Vinish D'silva
joseph.dsilva@mail.mcgill.ca

Florestan De Moor
florestan.demoor@mail.mcgill.ca

Bettina Kemme
kemme@cs.mcgill.ca

School of Computer Science, McGill University
Montréal, Canada

ABSTRACT

With the tremendous growth in data science and machine learning, it has become increasingly clear that traditional relational database management systems (RDBMS) are lacking appropriate support for the programming paradigms required by such applications, whose developers prefer tools that perform the computation outside the database system. While the database community has attempted to integrate some of these tools in the RDBMS, this has not swayed the trend as existing solutions are often not convenient for the incremental, iterative development approach used in these fields. In this paper, we propose AIDA - an abstraction for advanced in-database analytics. AIDA emulates the syntax and semantics of popular data science packages but transparently executes the required transformations and computations inside the RDBMS. In particular, AIDA works with a regular Python interpreter as a client to connect to the database. Furthermore, it supports the seamless use of both relational and linear algebra operations using a unified abstraction. AIDA relies on the RDBMS engine to efficiently execute relational operations and on an embedded Python interpreter and NumPy to perform linear algebra operations. Data reformatting is done transparently and avoids data copy whenever possible. AIDA does not require changes to statistical packages or the RDBMS facilitating portability.

PVLDB Reference Format:
Joseph Vinish D'silva, Florestan De Moor, Bettina Kemme. AIDA - Abstraction for Advanced In-Database Analytics. *PVLDB*, 11(11): 1400-1413, 2018.
DOI: https://doi.org/10.14778/3236187.3236194

1. INTRODUCTION

The tremendous growth in advanced analytical fields such as data science and machine learning has shaken up the data processing landscape. The most common current approach to develop machine learning and data science applications is to use one of the many statistical languages such as R [21], MATLAB, Octave [13], etc., or packages such as pandas [31], NumPy [55], theano [52], etc., meant to augment a general purpose language like Python with linear algebra support. Should the data to be used reside in an RDBMS, the first step in these programs is to retrieve the data from the RDBMS and store them in user space. From there, all computation is done at the user end. Needless to say, user systems do not possess huge memory capacity nor processing power unlike servers running an RDBMS potentially forcing them to use smaller data sets. Thus, users sometimes resort to big data frameworks such as Spark [58] that load the data into a compute cluster and support distributed computation. But even when enough resources are available, users might choose smaller data sets, in particular during the exploration phase, as transfer costs and latencies to retrieve the data from the database system can be huge. This data sub-setting can be counterproductive, as having a larger data set can reduce algorithm complexity and increase accuracy [12]. Additionally, once the data is taken out of the RDBMS, all further data selection and filtering, which is often crucial in the feature engineering phase of a learning problem, needs to be performed within the statistical package [18]. Therefore, several statistical systems have been enhanced with some relational functionality, such as the DataFrame concepts in pandas [31], Spark [3], and R.

This has the database community pondering about the opportunities and the role that it needs to play in the grand scheme of things [47, 56]. In fact, many approaches have been proposed hoping to encourage data science users to perform their computations in the database. One such approach is to integrate linear algebra operators into a conventional RDBMS by extending the SQL syntax to allow linear algebra operations such as matrix multiplication [60, 27, 2]. However, the syntactical extension to SQL that is required, is quite cumbersome compared to what is provided by statistical packages, and thus, has not yet been well adopted.

Both the mainstream commercial [37, 57] and open source [38, 44, 51] RDBMS have been cautious of making any hasty changes in their internal implementations – tuned for relational workloads – to include native support for linear algebra. Instead, most have been content with just embedding host programming language interpreters in the RDBMS engine and providing paradigms such as user defined functions (UDFs) to interact with them. However, the use of UDFs to perform linear algebraic computation has been aptly noted as not a convenient abstraction for users to work with [44].

In fact, a recent discussion on the topic of the intersection of data management and learning systems [24], points out that even among the most sophisticated implementations

Proceedings of the VLDB Endowment, Vol. 11, No. 11

DOI: https://doi.org/10.14778/3236187.3236194

which try to tightly couple RDBMS and machine learning paradigms, the *human-in-the-loop* is not well factored in. Among the machine learning community, there is already a concern that research papers are focusing solely on accuracy and performance, ignoring the human effort required [12]. Further, [12] also points out that human cycles required is the biggest bottleneck in a machine learning project and while difficult to measure, easiness and efficiency in experimenting with solutions is a critical aspect.

Therefore, unsurprisingly, despite efforts from the database community, studies have consistently shown Python and R being the top favorites among the data science community [40, 10]. In light of these observations, in this paper, we propose an abstraction akin to how data scientists use Python and R statistical packages today, but one that can perform computations inside the database. By providing an intuitive interface that facilitates incremental, iterative development of solutions where the user is agnostic of a database back-end, we argue that we can woo data science users to bring their computation into the database.

We propose AIDA - abstraction for **A**dvanced **i**n-**d**atabase **a**nalytics, a framework that emulates the syntax and semantics of popular Python statistical packages, but transparently shifts the computation to the RDBMS. AIDA can work with both linear algebra and relational operations simultaneously, moving computation between the RDBMS and the embedded statistical system transparently. Most importantly, AIDA maintains and manages intermediate results from the computation steps as elegantly as current procedural language based packages, without the hassles and restrictions that UDFs and custom SQL-extension based solutions place. Further, we demonstrate how AIDA is implemented without making modifications to either the statistical system or the RDBMS engine. We implement AIDA with MonetDB as the RDBMS back-end. We show that AIDA has a significant performance advantage compared to approaches that transfer data out of the RDBMS. It also avoids the hassles of refreshing data sets in the presence of updates.

In short, we believe that AIDA is a step in the right direction towards democratizing advanced in-database analytics. The main contributions of this paper are that we:

- identify the shortcomings of current RDBMS solutions to integrate linear algebra operations (such as UDFs and stored procedures) and propose a new interface for interactive and iterative development of such projects that fit more elegantly with current programming paradigms.
- implement an RMI approach to push computation towards the data, keeping it in the database.
- implement a common abstraction, *TabularData*, to represent data sets on which both relational and linear algebra operations can be executed.
- exploit an embedded Python interpreter to efficiently execute linear algebra operations, and an RDBMS (MonetDB) to execute relational operators.
- transparently provide data transfer between both systems, avoiding data copy whenever possible.
- allow host language objects to reside in the database memory throughout the lifetime of a user session.
- develop a database adapter interface that facilitates the use of different RDBMS implementations with AIDA.
- provide a quantitative comparison of our approach with other in-database and external approaches.

2. BACKGROUND

In this section, we briefly cover the attempts made by statistical packages to offer declarative query support and by RDBMS to facilitate statistical computations. We do not cover specialized DBMS implementations intended for scientific computing such as [5, 50, 28, 7] as our starting point is that data resides in a traditional RDBMS.

2.1 Relational Influence in Statistical Systems

In their most basic form, statistical systems load all data they need into their memory space. Assuming that the data resides in an RDBMS, this results in an initial data transfer from the RDBMS to the client space.

However, incremental data exploration is often the norm in machine learning projects. For instance, determining what attributes will contribute to useful features can be a trial and error approach [12]. Therefore, while statistical systems are primarily optimized for linear algebra, many of them also provide basic relational primitives including selection and join. For example, the DataFrame objects in R [42], pandas [31] package for Python, and Spark support some form of relational join. Although not as optimized as an RDBMS or as expressive as SQL, the fact that linear algebraic operations usually occupy a lion's share of the computation-time makes this an acceptable compromise.

There has been some work done in pushing relational operations into the database, such as in Spark [9]. However, while this is useful at the beginning, when the first data is retrieved from the database, it is less powerful later in the exploration process where users might have already loaded significant amount of data into the statistical framework. In such situation, it might be more beneficial to just load the additional data into the framework and perform, say a join inside the framework rather than performing a join in the database and reloading the entire data result, which can be even more time consuming should the result set be large [45].

As a summary, users typically either fetch more attributes than would be necessary and then do relational computations locally, or they have to perform multiple data transfers. Given that the exploration phase is often long and cumbersome, both approaches do not seem appealing.

2.2 Extending SQL for Linear Algebra

SQL already provides many functionalities required for feature engineering such as joins and aggregations. Thus, exploiting them in the exploration phase is attractive.

Recent research proposes to integrate linear algebra concepts natively into RDBMS by adding data types such as vector and matrix and extending SQL to work with them [60, 27]. However, they might not fare that well in terms of usability. For example, most linear algebra systems use fairly simple notations for matrix multiplication such as:

```
res = A × B
```

In contrast, the SQL equivalents as described in [60, 27] require several lines of SQL code which might not be as intuitive. Another inconvenience, which is not obvious at first sight but equally important, is the lack of proper support to store and maintain the results of intermediate computations in above proposals, as users need to explicitly create new tables to store the results of their operations, adding to the lines of code required. This is akin to how in standard SQL one has to create a table explicitly if the result of a query

is to be stored for later use. However, unlike SQL applications, learning systems often go through several thousands of iterations of creating such temporary results, making this impractical. Additionally, RDBMS generally treat objects as persistent. This is an extra overhead for such temporary objects and burdens the user to perform explicit cleanup[1]. Procedural high-level languages (HLL) such as Python and R, on the other hand, provide much simpler operational syntax and transparently perform object management by automatically removing any objects not in use. In short, data science applications require many HLL programming features lacking in SQL and as pointed out in [15], SQL was never intended for such kind of use.

2.3 UDF-ing the Problem

User-defined functions (UDFs) have been proposed as a promising mechanism to support in-database analytics including linear algebra operations [26, 44]. However, so far, they have not found favor in such applications. We believe this has to do with their usability. As mentioned before, learning is a complex process and can have a considerable exploration phase. This can include a repetitive process of fine-tuning the data set, training and testing an algorithm, and analyzing the results. The user is actively involved in each step as a decision maker, often having to enrich/transform the data set to improve the results, or changing the algorithm and/or tweaking its parameters for better results.

UDFs are not cut out for such incremental development for several reasons. The only way to interact with a UDF is via its input parameters and final output results, and all host language objects inside a UDF are discarded after execution. If they are required for further processing, they need to be stored back into the database as part of the UDF. Similar to the SQL extensions described above, the user needs to write code to write the objects to a database table [43], and additional UDFs for later retrieval. This is an overhead and a significant development effort that is not related to the actual problem that the users are attempting to solve, but attributed to the framework's limitation.

UDFs are also often confined in terms of the number and data types of its input and output parameters. This hinders users from developing generic UDF-based solutions to their learning problems [44]. In contrast, high-level languages (HLL) such as Python are much more flexible. Further, as SQL is designed to be embedded into an HLL to build more sophisticated applications [15], there is little incentive for the users to take the HLL – DBMS – UDF approach, especially if the UDF is to be written in the same HLL and comes with many hassles. Therefore, while efficient, in their rudimentary form, we believe that UDFs are not the most convenient approach to adapt.

3. MOTIVATION & APPROACH

AIDA's philosophy is to provide an abstraction that is high in usability in terms of programming paradigms and at the same time takes advantage of the data storage, management and querying capacities of RDBMS.

3.1 Overall Architecture

Figure 1 depicts a high-level conceptual layout for AIDA. AIDA resides in the embedded Python interpreter of the RDBMS, thus sharing the same address space as the RDBMS. This makes AIDA easily portable since embedded Python interpreters are becoming increasingly common in modern RDBMS. Embedded interpreters also help us leverage some of the RDBMS optimizations already available in this area, as we will later discuss. Users connect to AIDA using a regular Python interpreter and AIDA's client API. Below we will take a brief look at some of the key ideas behind AIDA before delving deeper in the later sections.

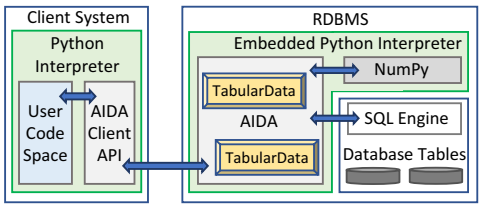

Figure 1: AIDA - conceptual layout

Many contemporary analytical systems provide intuitive interfaces for iterative and interactive programming. AIDA's objective is to be on par with such approaches. For AIDA's client API, we decided to leverage the Python interpreter, as it is ubiquitously available and often used to work with statistical packages such as NumPy and pandas. Using such a generic tool also provides opportunities to the users for integrating other Python-based packages into their programs.

However, data transformations and computations are not executed on the client side. Instead, AIDA's client API sends them transparently to the server and receives a *remote reference* which represents the result that is stored in AIDA. We implement this interaction between client and server in the form of remote method invocations (RMI), whose details we cover in Section 5. RMI is a well-established communication paradigm and known to work in practice. It will also allow us in the future to easily extend AIDA to be part of a fully distributed computing environment where data might be distributed across many RDBMS.

3.2 A Unified Abstraction

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

TabularData. TabularData objects reside in AIDA, and therefore in the RDBMS address space. They remain in memory beyond individual remote method invocations. TabularData objects can work with both data stored in database tables as well as host language objects such as NumPy arrays. Users perform linear algebra and relational operations on a TabularData object using the client API, regardless of whether the actual data set is stored in the database or in NumPy. Behind the scenes, AIDA utilizes the underlying RDBMS's SQL engine to execute relational operations and relies on NumPy to execute linear algebra. When required, AIDA performs data transformations seamlessly between the two systems (see Figure 2) without user involvement, and as we will see later, can often accomplish this without the need to copy the actual data.

[1]While many RDBMS have the concept of a *temporary table* [4, 49], they only address a small part of the problem.

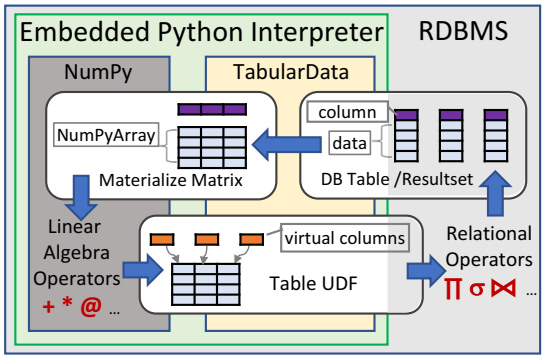

Figure 2: TabularData Abstraction

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

For relational operators, we decided to not use pure SQL as it will make it difficult to provide a seamlessly unified abstraction. Instead, we resort to object-relational mappings (ORMs) [33], which allow an object-oriented view and method-based access to the data in database tables. While not as sophisticated as SQL, ORMs are fairly versatile. ORMs have shown to be very useful for web-developers, who are familiar with object-oriented programming but not with SQL. ORMs make it easy to query the database from a procedural language without having to write SQL or work with the nuances of JDBC/ODBC APIs. By borrowing syntax and semantics from ORM – we mainly based our system on Django's ORM module, a popular framework in Python [6] – we believe that data scientists who are familiar with Python and NumPy but not so much with SQL, will be at ease writing database queries with AIDA.

3.3 Overview Example

Let's have a look at two very simple code snippets that can be run in a client-based Python interpreter using AIDA's client API. The first code snippet represents a relational operator as it accesses the `supplier` table of the TPC-H benchmark [53] to calculate the number of suppliers and their total account balance. `supplier` is an object reference to the underlying database table and `agg` indicates an aggregation method with appropriate input:

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

The client API ships this relational transformation to AIDA which returns a remote reference to the client for a TabularData object that represents the resultset. The client program stores this reference in the variable `si`.

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

```
res = si / numpy.asarray([1000,1000000]);
```

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

At the server side, AIDA executes the relational operation (first code snippet) by first generating the SQL query to perform the required aggregation and then using the SQL engine of the RDBMS to execute it. The TabularData object represented by `si` will point to the result set created by the RDBMS (see right top resultset format in Figure 2). As this object becomes the input of a linear algebra operation (second code snippet), AIDA will transform it to a matrix (see the top left matrix in Figure 2), and perform the division operation using NumPy. It then encapsulates the resulting matrix in a TabularData object, returning a remote reference to the client, which stores it in `res`.

Should `res` become the input of a further relational operation, AIDA needs to provide the corresponding data to the SQL engine for execution. AIDA achieves this by transparently exposing the data through a *table UDF* to the RDBMS (see the bottom of Figure 2). Table UDFs are a standard RDBMS mechanism that allows embedded host language programs to expose non-database data sets to the SQL engine as if they were database tables.

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

In the following, we will discuss in Section 4 in detail the `TabularData` abstraction and the co-existence of different internal representations, how and when exactly linear algebra and relational operations are executed, how AIDA is able to combine several relational operators into a single execution, and how and when it is able to avoid data copying between the execution environments. In Section 5, we then present in more detail the overall architecture and its implementation.

4. A TABULAR DATA ABSTRACTION

All data sets in AIDA are represented as a *TabularData* abstraction, that is conceptually similar to a relational table with columns, each having a column name, and rows. Thus, it is naturally suited to represent database tables or query results. Also, two-dimensional data sets such as matrices that are commonly used for analytics can be conceptually visualized as tables with rows and columns where the column positions can serve as virtual column names. A TabularData object is conceptually similar to a DataFrame instance in Pandas and Spark but can be accessed by two execution environments: Python interpreter and SQL engine. Using a new abstraction instead of extending the existing DataFrame (such as pandas) implementations also avoids the metadata overhead that these systems have in order to support relational operations on their own. TabularData objects are *immutable* and applying a linear algebra/relational operation on a TabularData object will create a new TabularData object. We will discuss the internal implementation details of `TabularData` objects in Section 4.3.

4.1 Relational Operations on TabularData

As mentioned before, to support relational operations, AIDA borrows many API semantics from contemporary ORM systems. AIDA's TabularData abstraction supports SQL/relational operations such as selection, projection, aggregation and join. Listing 1 shows a code snippet on a TPC-H database that returns for each customer their name (`c_name`), account balance (`c_acctbal`), and country (`n_name`).

The program first establishes a connection to the database (line 1), then creates references to TabularData objects that represent the `customer` and `nation` tables in the database, and stores these references in the variables `ct` and `nt` respectively. Line 4 *joins* these two TabularData objects, describing the columns on which the join is to be performed. The result is a new TabularData object, whose reference is stored in `t1`. Further, a *projection* operation on `t1` retrieves only the columns of interest, resulting in `t2`.

Listing 1: TabularData : Join and Projection

```
1 db = aida.connect(user='tpch', pass='rA#2.p')
2 ct = db.customer
3 nt = db.nation
4 t1 = nt.join(ct,('n_nationkey'),('c_nationkey'))
5 t2 = t1.project(('n_name','c_name','c_acctbal'))
```

While the code in listing 1 is using a variable for each intermediate TabularData object, users can use *method chaining* [16] to reduce the number of statements and variables by chaining the calls to operators, as shown below.

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

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

As another example, the source code listing 2 finds countries with more than one thousand customers and their total account balance by first aggregating and then filtering.

Listing 2: TabularData : Aggregation and Selection

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

Conditions in AIDA are expressed using Q objects, similar to Django's API. Q objects take the name of an attribute, a comparison operator, and a constant literal or another attribute with which the first attribute needs to be compared with. Q objects can be combined to denote complex logical conditions with conjunctions and disjunctions, such as those required to express the `AND` and `OR` logic in SQL.

The last line displays the columns of the result set referenced by `t4`. `cdata` is a special variable that refers to the dictionary-columnar representation of the object (more on data representations in Section 4.3). This is the point when AIDA actually transfers data to the client. Often, users are interested only in a small sample of data records or some aggregated information on it, such as in the example we just presented. This significantly reduces the size of actual data that needs to be transferred to the client.

Discussion: It is perhaps important to mention that most ORM implementations require the user to specify a comprehensive data model that covers the data types and relationships of the tables involved. AIDA does not impose such restrictions and works with the metadata available in the database. Another important distinction is that relational operations on an ORM produce a resultset, akin to processing results using JDBC/ODBC APIs. They do not support any further relational transformations on these resultsets. AIDA, on the other hand, transforms a TabularData object into another TabularData object, providing endless opportunities to continue with transformations. This is important for building complex code incrementally.

4.2 Linear Algebra on TabularData

TabularData objects support the standard linear algebra operations that are required for vector/matrix manipulations. We do so using the overloading mechanism of Python. These overloaded methods then ship the operation using RMI to the corresponding TabularData objects. For those operators that are not natively supported in Python, we follow the NumPy API syntax. This is the case, e.g., with the *transpose* operator for matrices. Similar approaches have been used by others [61]. Therefore, users can apply the same operator on a TabularData object for a given linear algebra operation as they would in NumPy. AIDA will then invoke the corresponding operation on its underlying NumPy data structures. For example, in the code listing below, where continuing off from the previous section, we compute the average account balance per customer for each country in `t4`, using linear algebra.

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

Further, we can also generate the total account balance across all the countries contained in `t4`, by performing a matrix multiplication operation as shown below [2].

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

Where `@` is the Python operator for matrix multiplication that is overloaded by TabularData and `T` is the name of the method in TabularData used for generating a matrix's transpose, a nomenclature we adopt from NumPy for maintaining familiarity. As we saw in the examples in Section 3.2, TabularData objects can also work with numeric scalar data types and NumPy array data structures also commonly used by data scientists for statistical computations.

4.3 One Object, Two Representations

There are two different internal representations for a TabularData object, and it may have zero, one, or both of these representations materialized at any time. One of the internal representations is a matrix format, i.e., a two-dimensional array representation where the entire data set is located in a single contiguous memory space (see Figure 2, left-top representation). AIDA uses NumPy to perform linear algebra, which requires this format for some operations such as matrix multiplication and transpose.

The second representation is a dictionary-columnar format used to execute relational operators (see Figure 2, left-top representation). This is essentially a Python dictionary with column names as keys and the values being the column's data in NumPy array format. While each array is in contiguous space, in contrast to the matrix format, the different columns might be spread out in memory. The rationale behind this approach is that AIDA is optimized to work with columnar RDBMS implementations such as MonetDB that offer *zero-copy* data transfer of query results from

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

the database memory space to the embedded language interpreter [25]. This means that the RDBMS passes query results to an embedded host language program without having to copy the data into application-specific buffers or perform data conversions[3]. This is made possible by having database storage structures that are fundamentally identical to the host language data structures. Thus, the host language only needs to have the appropriate pointers to the query result to be able to access and read it. In particular, MonetDB creates resultsets where each result column is stored in a *C* programming language array. NumPy uses the same format for arrays. MonetDB thus returns a Python dictionary with column names as the keys and references to the underlying data wrapped in NumPy objects. We leverage this optimization as it reduces the CPU/memory overhead. Therefore, the result of relational operators has a dictionary-column format as default.

Thus, the internal data structures used by TabularData are conventional NumPy/Python structures. Users can access these internal representations if required. For example, in listing 2, we access the dictionary-columnar data representation of `t4` through the special variable `cdata`. Similarly, the matrix representation can be accessed by using the special variable `matrix`. This allows users to use AIDA in conjunction with other Python libraries that work with NumPy data structures. A TabularData object will materialize a particular internal data representation the first time that representation is requested. That happens when the user accesses it through its special variable or from one of AIDA's internal methods. Once materialized, the internal representation will exist for the lifetime of the object, reused for any further operations as required. AIDA's data structures are essentially memory resident. In the future, we plan to implement a memory manager that can move least used TabularData objects into regular database tables to reduce memory contention. The data in a database table itself is treated by AIDA as though they are materialized in a dictionary-columnar format for all practical purposes. Therefore, in listing 1, the variables `ct` and `nt` refer to TabularData objects that represent `customer` and `nation` tables respectively.

4.3.1 Relational Operations & Data Representations

Relational operations on a TabularData object are always performed lazily, i.e., the resulting TabularData object's internal representation is not immediately materialized. They are materialized only when the user explicitly requests one of the internal data representations through the special variables we just discussed or when a linear algebra operation is invoked on it. This means that by default, the new TabularData object contains only the reference to its source – which can be a database table, one TabularData object or two TabularData objects in case of a join – and the information about the relational transformation it needs to perform on the source. This approach is very similar to the lineage concept that is followed by Spark [58] for their data sets and transforms. The source TabularData object itself may not be materialized yet or can have materialized data in any of the two formats that we discussed.

[3]Apache ecosystem has recently introduced the Apache Arrow [11] platform which can facilitate data movement between applications by using shared memory data structures.

When such a TabularData object needs to materialize its internal data representation, it will first request its immediate source to provide its object's SQL equivalent. Using this SQL, it will apply its own relational transformation logic over this source SQL. This is accomplished by using the concept of derived tables in SQL [34] which allows nesting of SQL logic, an approach also used in other implementations such as [29]. Once a TabularData object is materialized, it will discard the references to its source and transformation logic. We can distinguish three cases for the source.

Database table. First, if the source represents an actual database table, this is as simple as returning a `SELECT` query on the table with all the columns and no filter conditions.

Non-materialized source. Second, if the immediate source TabularData object itself is not materialized, then it would request its own source for the SQL equivalent, apply its transformation on top of it (but will not use it to materialize itself) and return the combined SQL logic. It is easy to envision this approach going down to arbitrary depths to build a complex SQL query, as shown in the example below. Here, first, we create `tx` by performing an aggregation on a table in the database, `dtbl`. Next, we filter `tx` by applying a filter to generate `ty`.

```
tx = dtbl.agg(('c1', {SUM('c2'):'c2total'})
              ,('c1',))
ty = tx.filter((Q('c2total', 100, CMP.GT),))
```

If we chose to materialize `ty` at this point, it will request `tx` for its equivalent SQL. `tx` then applies its filter operation over this SQL and ends up generating the SQL given below, that is then executed by the RDBMS SQL engine, materializing itself using the resultset in dictionary-columnar format.

```
select c1, c2total from
 (select c1, sum(c2) as c2total from
  (select c1, c2, c3 from dtbl)t group by c1)tx
where c2total > 100
```

Lazy evaluation techniques, while allowing users to build complex logic incrementally, ensures that optimization is not sacrificed. By nesting multiple relational transforms together, it can skip the need to materialize the intermediate TabularData objects, such as `tx` in the above example.

Materialized source. As a third case, the source might already be materialized in one or both of the internal representations. This data now needs to be provided to the RDBMS SQL engine for it to perform the relational operation. As mentioned in Section 3.3, AIDA exposes this data by means of a table UDF to the database (see the bottom of Figure 2). Table UDFs need to expose the dictionary-columnar representation for the SQL engine to execute queries. This is again because columnar-RDBMS like MonetDB share identical data structures, making data conversions and copies unnecessary. In case the source TabularData object has only a matrix representation so far, it can still provide the dictionary-columnar representation by an abstraction that represents a particular column's data as a column-wise slice of the matrix relevant to that column without making an actual copy of the data. The SQL equivalent of the source is again a simple `SELECT` query on this table UDF that will return all the data in the source TabularData object.

4.3.2 Linear Algebra & Data Representations

Unlike relational operations, linear algebra operations are computed and materialized immediately. This is primarily

because NumPy do not perform any significant optimizations when multiple computational operations are combined.

Scalar Operations. Linear algebra transformations involving scalar operations (e.g., dividing all elements in the data set by 1000) can be performed on either data representation because performing a scalar operation (e.g., division by 1000) on each column's data in dictionary-columnar representation is the same as performing it on the entire matrix. Therefore, such transformations will first check if the data is available in the matrix form and use it to perform the transformation. The resulting TabularData object will have its internal data in matrix representation. Matrices are given preference because, as we saw in Section 4.3.1, a dictionary-columnar representation can be built from it without making another copy, whereas the reverse is not possible.

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

Finally, if there is not yet an internal representation due to lazy-evaluation of relational operations as discussed in the previous section, it will at this point materialize it. As this will be naturally a dictionary-columnar representation as the relational operations are executed by the SQL engine, it will work with this representation.

In summary, if there is no matrix representation available, we do not build a matrix just because of a scalar operation. First, matrix construction takes time and is less versatile when it comes to supporting different data types for different columns as we need to upgrade all columns in the data set to a data type that can store all data elements. This can be an overkill as many operations do not need a uniform data type across the entire data set. Besides, having an additional matrix representation takes up memory.

Vector Operations. For linear algebra transformations involving vector operations, such as matrix-matrix multiplication and matrix transpose, the TabularData object will need the matrix format representation. If it currently has no internal data representation due to the lazy evaluation of relational operators, it will first materialize it, which will have dictionary-columnar representation. It then builds the matrix representation from it. The transformation is then applied to this matrix and the resulting new TabularData object will have a matrix internal representation.

Optimizations: AIDA employs some clever optimizations to share data between multiple TabularData objects, attributed to their immutable nature. This is possible when a transformation is requesting a subset of columns from a TabularData object's data set. If the TabularData set has a dictionary-columnar representation, then it will generate the new TabularData object in the same format, but with only the requested subset of columns in its dictionary. The data arrays for these columns will be just references to the original data set. For example, remembering that `t4` is materialized when we printed its dictionary-columnar data representation, next, the expression `t4[['totabal']]` results in a temporary tabular data object that has only the `totabal` column of `t4` and hence can be shared with `t4`.

4.3.3 Practicality of Dual Representations

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

4.4 User Extensions

Custom Transformations. AIDA provides a rich client API capable of performing relational operations such as join, selection (filtering), projection, aggregation, etc., as well as prominent linear algebra operations like addition, subtraction, multiplication, division, exponentiation with both scalar and vector operands, to name a few. However, users often need to perform custom transformations not provided by the framework. As we recapped previously, RDBMS' attempts to support this via UDFs lack usability. AIDA's TabularData objects support this feature through the `_U` operator that is very easy to use. In the example below, we use this approach to create a TabularData object from `t2` which contains only the first name of the customers.

```
def cTrans(td):
  cn = td.cdata['c_name']
  fn = np.asarray([n.split()[0] for n in cn])
  return {'c_name':fn};

t7 = t2._U(cTrans);
```

Custom transformations are expressed using regular Python functions. They accept a TabularData object and any additional user provided arguments, and must return a data set in one of the formats that TabularData supports internally. Custom transformations are materialized immediately. In our example, the custom transformation `cTrans` returns a dictionary. It reads the customer names from the source TabularData object and uses Python's string manipulation function to parse them and create an array of only first names. The dictionary that it returns is then used to construct the new TabularData object, whose reference is stored in `t7`. AIDA's client API ships the transformation logic (`cTrans` function in our example) transparently to the server side using RMI to be executed at the server. We can perform additional relational or linear algebra operations on `t7` as is the case with any other TabularData objects.

Remote Execution. Many advanced analytics applications such as learning algorithms often need to do repeated iterations of certain computations to fine tune the algorithm parameters, often to the extent of several thousand repetitions. Using AIDA, this can easily translate to a large amount of RMI messages and may pose a non-negligible overhead for a large number of iterations. Under such circumstances, users can make use of the remote execution operator (`_X`) functionality to shift the iteration logic to the server side and bypass the RMI overhead. Unlike other operators in AIDA, which are part of the TabularData abstraction, the remote

execution operator allows code execution that is not tightly coupled to a particular TabularData object. Instead, AIDA attaches it to the database workspace object, which is associated with a user session. The details of the workspace object are discussed in Section 5. The example below computes the n^{th} power of each element in `t6`, where $n > 0$.

```
def exp(td, n):
  r = td
  for i in range(0, n):
    r = r * td
  return r

res = db._X(exp, t6, 10)
print(res.cdata)
```

The user function, `exp` takes as arguments a TabularData object `td` and n, the power for exponentiation, performs the computation and returns the result. The remote execution operator `_X` executes the function on the server side, passing it the required arguments, and thus bypasses the RMI overhead for each iteration. The resulting TabularData object's stub is then returned to the client who displays it in this example. The remote execution operator is a suitable alternative to the stored procedure concept of RDBMS, but fits neatly into the host language, without the hassles and restrictions of UDFs and stored procedures.

Custom transformations and the remote execution operator are mechanisms to ship entire functions to the server side to be executed in server space. Thus, they are well suited for the development of learning algorithms, as they allow data scientists to integrate their favorite packages such as Scikit-learn [39] into the AIDA computation framework.

5. LEAVE DATA BEHIND

A key design philosophy of AIDA is to push computation to the RDBMS, near the data. Data transfer to the client is only needed when explicitly requested, i.e., when aggregated information or final results need to be viewed and analyzed; thus, the expectation is that this is only a small fraction of the overall data accesses needed for computations.

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

The high-level architecture of AIDA that supports this model is depicted in Figure 3 and discussed in this section. A *bootstrap stored procedure* loads AIDA's server-side processes when the database starts up.

Database Memory Resident Objects. As discussed in section 2.3, even though modern RDBMS implementations support embedded programming language interpreters, they only expose limited capabilities. Specifically, host language objects only live within the scope of the UDF source code. In order to work around this problem, we introduce the concept of a database memory resident object or DMRO. A DMRO stays in the RDBMS memory (the embedded Python interpreter memory to be precise), outside of the scope of a UDF or a stored procedure. Users can perform operations on these objects using AIDA's client APIs. The most common DMROs are TabularData objects, the computational component discussed in the previous section. We will encounter a few others shortly.

Distributed Object Communication. The distributed object communication layer (DOC) allows AIDA's client API to invoke computational primitives on the TabularData objects residing in the RDBMS memory via RMI. AIDA's DOC layer follows the conventional architecture popularized by implementations such as Java RMI [41] with minor adaptations to suit AIDA's needs. On the server side, we have an instance of a remote object manager server, which is a DMRO created by the bootstrap stored procedure. The manager acts as a repository to which AIDA's server-side modules can register objects that are to be made available for remote access. The client invokes methods on a *stub* object residing on the client side, which *marshals* the input parameters and sends them to a *skeleton* object on the server side. The skeleton object will unmarshal this information, execute the intended method on the actual object, and marshal and return the results back to the client side stub. For the sake of simplicity, we have omitted skeleton objects from Figure 3. We extend dill [30], a popular module for marshaling objects in Python, with some customization for AIDA. In particular, if the result of an RMI is of type TabularData (such as when a transformation on a TabularData returns the resulting TabularData object), we do not send the actual data but the stub information for the object.

The Database Adapter Interface. Modern RDBMS implementations with embedded interpreters allow UDFs written in a host-language to query the database directly without an explicit database connection. However, each vendor implements their own API. As AIDA relies on these internal APIs to interact with the RDBMS it is embedded in, we define a *database adapter interface* to standardize interaction, and to keep remaining AIDA packages independent of the RDBMS. Therefore, akin to JDBC/ODBC applications, by implementing a database adapter package conforming to this interface, one can easily port AIDA to that specific RDBMS. A database adapter interface implementation must be able to authenticate, read database metadata, execute a SQL query and return results in dictionary-columnar format, and facilitate the creation of table UDFs.

As discussed in Section 4.3, we exploit the zero-copy optimization of MonetDB to return resultsets stored in TabularData objects. Many RDBMS, especially row-based RDBMSes, do not offer such optimized data transfers as their physical storage model has no resemblance to the data structures of the embedded language. Therefore, in order to work with such systems, the database adapter interface will have to convert the resultset returned by the RDBMS into one of the internal representations of the TabularData object.

The Connection Manager. The connection manager is a DMRO created by AIDA's bootstrap procedure that is responsible for managing client sessions. When a client makes a connection request, the connection manager uses the database adapter to authenticate with the database. On success, it creates a database workspace object (discussed in next section) and sends its stub back to the client. Referring back to the first line of code listing 1, `db` is such a stub.

Database Workspace. This is another kind of DMRO, created for each authenticated client connection in AIDA. Primarily, it provides access to database tables via the TabularData abstraction. For example, `db.nation` provides a reference to a TabularData object that encapsulates the `nation` table in the database. Such TabularData objects and any

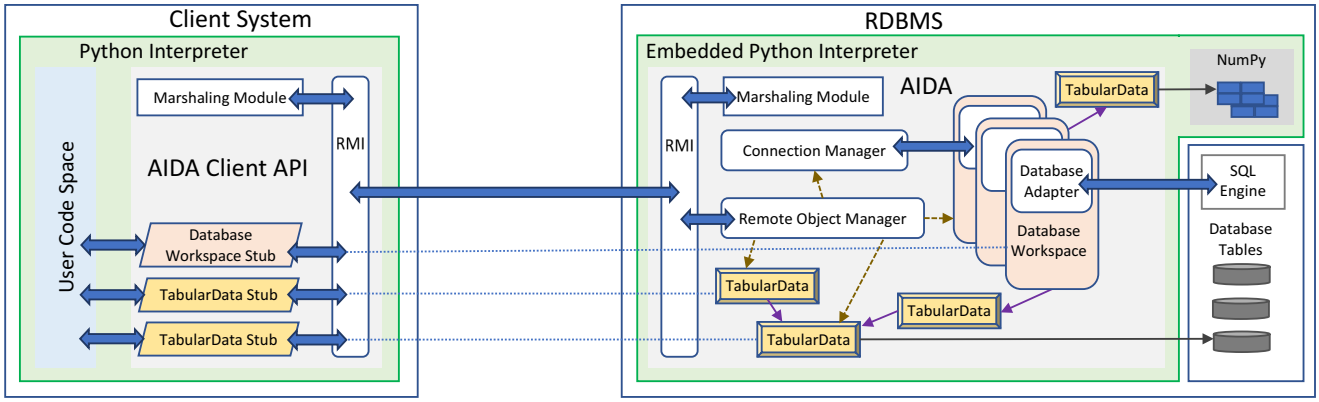

Figure 3: High Level Architecture of AIDA

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

Life Span of a DMRO. Remote object manager and connection manager objects are the only DMROs that exist throughout AIDA's uptime. Database workspace objects are discarded when the corresponding client disconnects. TabularData objects get discarded if (i) no client stub references it, (ii) no database workspace has a reference to it (e.g., set via a remote execution function), (iii) and it is not a source for any TabularData instance that has not been materialized yet, which in itself is not selected to be discarded.

6. EVALUATION

Our testing objectives are: (i) to understand the cost benefits of keeping data at the server compared to transferring it to the client and processing it there; (ii) to compare AIDA's performance for linear algebra operations with executions on the client side using a standard statistical package; (iii) to observe how statistical packages fare when it comes to executing relational operations compared to AIDA, which pushes them into the optimized RDBMS; (iv) to measure AIDA's framework overhead compared to a rudimentary database UDF-based solution; (v) and finally, to compare the performance of AIDA against these systems for an end-to-end learning problem.

We evaluate the following systems. **AIDA** is our base AIDA implementation. We also analyze variations of AIDA (e.g., with and without using remote execution functionality). **DB-UDF** is a UDF-based solution that, just as AIDA, performs execution within the RDBMS. Furthermore, we implemented three solutions that transfer data out of the RDBMS for execution at the client using the following frameworks: (i) **NumPy**, (ii) **pandas** and (iii) **Spark**.

6.1 Test Setup

We run the client and the RDBMS on identical nodes (Intel® Xeon® CPU E3-1220 v5 @ 3.00GHz and 16 GB DDR4 RDIMM main memory, running on Ubuntu 16.04.). The nodes are connected via a Dell™ PowerConnect™ 2848 switch that is part of a Gigabit private LAN. In practical implementations, the RDBMS would be likely on a high-end server with much larger resources compared to a client's computer, and the interconnect would be less powerful. However, keeping the two systems similar ensures fairness in the performance metrics that we are comparing.

For software, we use MonetDB v11.29.3, pymonetdb 1.1.0, Python 3.5.2, NumPy 1.13.3, pandas 0.21.0, and Spark 2.3.0 using MonetDB JDBC driver 2.27. Unless explicitly specified, default settings are used for all software. In all our experiments we only start measuring once a warm-up phase has completed and average over several executions.

6.2 Loading Data to Computational Objects

In this test case, we try to understand the cost benefits of not having to transfer data for computation from the database into the client space. We experiment with tables ranging from 1 to 1 million rows. There are 100 columns consisting of randomly generated floating point numbers.

For NumPy and pandas, we use pymonetdb, MonetDB's Python DB-API [23] implementation to fetch data from the database and build a NumPy array resp. a pandas DataFrame. As the performance of retrieving data is dependent on the connection buffer size in pymonetdb, we test with the default buffer size of 100 but also an optimized buffer size of 1 million, reflecting the size of our largest data set. The latter setting is recorded as `NumPyOpt` and `pandasOpt` in the performance figures. For Spark, we load the data using the JDBC connection to build a Spark matrix. The default fetch size of JDBC is set to 1 million. For DB-UDF, we use a Python UDF to load data from the database into a NumPy matrix. For AIDA, we build a TabularData object. AIDA by default materializes the result of a relational data load in dictionary-columnar representation. We also measure the cost for *additionally* building the matrix representation of the data, indicated as `AIDA-Matrix` in the chart.

Figure 4 shows loading time in logarithmic scale. As a first observation, our Spark implementation performs worse than any other solution by orders of magnitude, including the other client-based solutions. We would like to note that Spark is optimized for large-scale distributed batch computations; that is, it has quite different target applications. Also, MonetDBs pymonedb driver used for NumPy and pandas, is highly optimized, providing data-transfer rates several times faster than many other database implementations [45], likely including the JDBC driver used for Spark.

Clearly, for small data sets less than 100 rows, AIDA has no real advantage over the other client-based solutions. In fact, for a 1-row table, AIDA needs twice as long as NumPy, and 13% longer than pandas. However, this is still in the

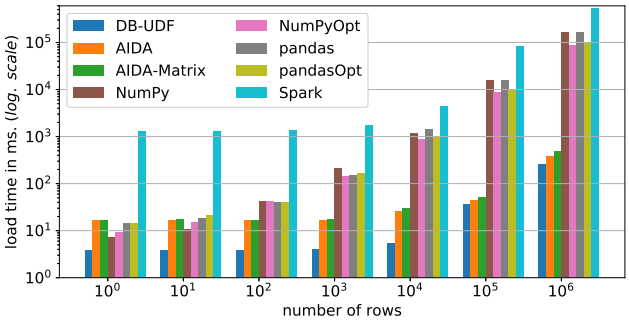

Figure 4: Time to load data.

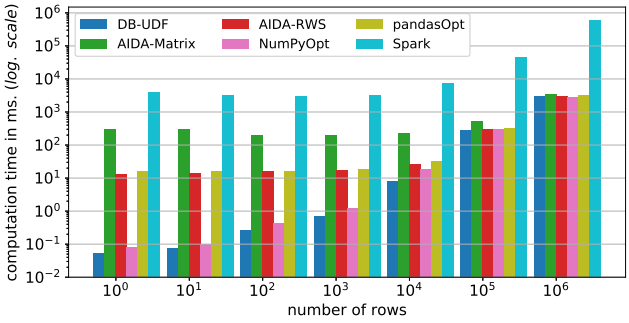

Figure 5: Matrix × Vector perf.

range of a few milliseconds, and thus, not noticeable for an interactive user. But as the data size increases to 100 rows, the trend changes with AIDA now having an upper hand over client based approaches. At 100 rows, AIDA takes only 40% of the time compared to loading with NumPy and pandas. This trend continues as data set size increases. At 1 million rows, AIDA is roughly 440 times faster than the default connection for NumPy and pandas, and 240 times faster if we use optimized connection for NumPy and pandas. Even if we force AIDA to build matrices after loading data, it is still able to load data about 188 times faster than the optimized connections with NumPy and pandas. In absolute numbers, while NumPy and pandas took about a minute and a half to load 1 million rows with an optimized connection, AIDA manages to load data and also build an additional matrix representation of its data under half a second. This is impressive, given that MonetDB has a highly optimized data transfer [45].

DB-UDF, being server-based and having no framework overhead, is the fastest across all table sizes. For small tables, it is around 4x faster than AIDA, and with larger sizes, it takes around 65% of AIDA's time. Thus, for a simple data load, AIDA has no benefit over a UDF-based solution.

6.3 Linear Algebra Operations

To measure the overhead of the AIDA framework on linear algebra operations, we multiply a matrix with a vector. The primary matrix is built again from the same table as in the previous test case, with 100 columns and up to 1 million rows. The vector is represented as a matrix with 100 columns stored as one row. Therefore, the vector is transposed before performing the multiplication. The operation can be visualized in code as shown below.

```
res = m1 @ m2.T
```

The test measures the cost of performing a total of 100 matrix multiplications after the initial objects are loaded. Iterative scenarios are very common in learning algorithms as parameters are continuously adjusted to bring the error rate down. In order to observe the overhead due to RMI calls, we also implement this iteration using the remote execution functionality of AIDA's database workspace where the iteration logic is executed within the server (recorded as `AIDA-RWS`). That is, AIDA-RWS has only one RMI call compared to 100 RMIs with AIDA.

Figure 5 shows the execution time in logarithmic form. Again, Spark performs significantly worse. The difference in programming paradigms might have an impact.

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

6.4 Relational Joins On Matrices

In this section we compare the join implementations in client-based pandas and Spark with server-based AIDA and DB-UDF, that can leverage MonetDB's optimized join implementation. The results shown do not show connection or load times between client and server. We use two data sets with 11 integer columns and one million rows each. In the test, we gradually add more columns to the join condition. One of the columns is unique and identical in both data sets, functioning as the key. A given key's remaining columns may differ between the two data sets with a small probability. With this, the key column join between the two data sets produces all the million rows in the output, while adding more columns into the join condition reduces the result's cardinality. The output contains all columns.

For AIDA, we test two scenarios that it might have to face. In the first case, the data is in RDBMS tables and AIDA, therefore, executes a SQL internally in the RDBMS to produce the result. In the second case, the data is materialized in a TabularData object in dictionary-columnar format. AIDA exposes this via table UDFs to the RDBMS and has it execute a SQL performing a join on the table UDFs. This approach is denoted as `AIDA-TableUDF`. DB-UDF executes an SQL join over the tables inside the database and loads the contents into a NumPy array.

Figure 6 shows (non-logarithmic) response times (left axis) and cardinality of the result (right axis). The two AIDA implementations and DB-UDF perform much better than pan-

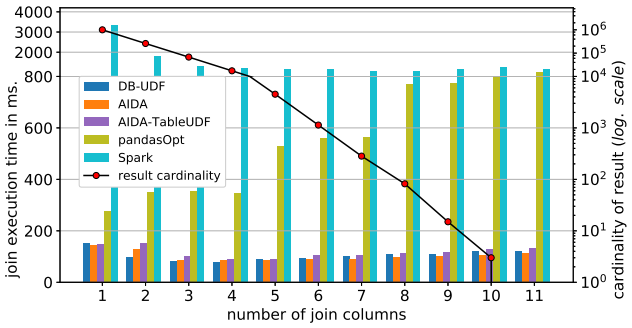

Figure 6: Joining two data sets.

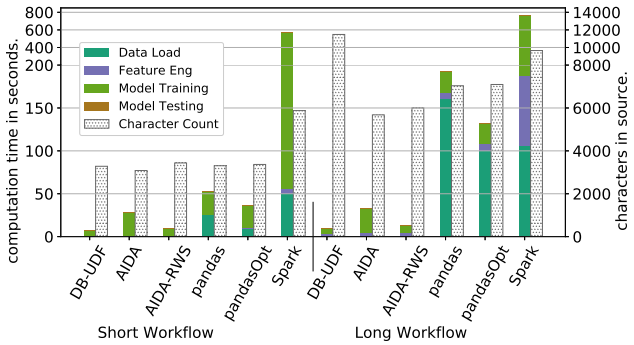

Figure 7: Linear regression on Bixi data set.

das and Spark as they can leverage the underlying RDBMS' optimizations. Response times for AIDA and DB-UDF are always less than 150 ms showing that MonetDB is well equipped to handle complex joins and/or large result sets. Execution times first decrease with increasing number of join columns as there are less and less records in the result set to be materialized but then increase again as the join computation time becomes larger with more join columns.

Both client-based solutions are significantly less efficient. Pandas takes nearly twice the time for the 1-column join, and around 8 times longer with more than 7 join columns compared to the server-side solutions. Spark is even worse, in particular with few join columns. It looks like it struggles with large results sets more than with complex joins as its performance improves with smaller result sets.

6.5 Performing Linear Regression

Finally, to understand the performance and usability of AIDA on a real learning problem, we have developed solutions for a more complex test case. The data set is based on the city of Montréal's public bicycle sharing system, Bixi[4]. The learning objective is to predict the duration of a trip given the distance between start and end points of the trip. This lends itself well to the application of a linear regression algorithm. We do not use any built-in libraries such as Scikit-learn or Spark MLlib [35] to avoid any bias from the implementation differences of these libraries. Instead, the linear regression algorithm is written using relational and linear algebra operations on the computational structures (DataFrames for Spark and pandas, matrices for DB-UDF, TabularData for AIDA), as would be the case when a user develops their own algorithm from scratch.

Workflows. We have built two workflows depicting how a data scientist could explore the problem. The implementations of these workflows for each of systems (DB-UDF, AIDA, pandas, Spark) are available on github[5]. For AIDA and pandas they are depicted in Jupyter notebooks, a widely used tool that supports interactive development and source code exchange. For Spark and DB-UDF, we formatted the source code of their implementations using github's markdown file format which allows for reading the source code along with the output results.

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

The second example workflow depicts a more complex path where a data scientist explores the data sets and builds and analyzes different models before choosing a promising approach. The data scientist first decides to build a distance feature from the GPS coordinates available in the trip data using a geodesics-based formula [54] and uses it to build a model. He/she then explores the idea of using a map data set that has road distances to enrich the trip data. For Pandas and Spark this involves joining the map data in the database to the trip data which is already at the client side. For that, the additional, smaller (map) data set is retrieved from the database and the join performed inside the statistical framework. The data scientist then compares the error rate of both approaches, and decides to settle for the later.

For both workflows we analyze execution times, and we also want to understand how easy they are to implement for the different systems. Figure 7 shows for each system with the left bar computation time not including any think or development time (non-logarithmic in seconds) and with the right bar our attempt to measure source code complexity.

Complexity. Usability is very subjective, and it is difficult to measure the complexity of different programming paradigms. As a simple approximation, we measure source code complexity as the amount of source code for each implementation, using non-white space/no-comments character count as the metric. We do not use the typical lines of code (LOC) metric as it may not be able to accurately portray the nature of different programming languages/paradigms[6]. To be fair, we chose similar variable and function names for all systems. With the exception of DB-UDF, all workflows are written using a single programming language (Python or Scala). In contrast, the DB-UDF implementations contain a mix of Python UDFs and fairly complex SQL statements.

For the short workflow, source code complexity is similar for nearly all systems. This is because exploration and the linear regression algorithm itself are straightforward. Spark's use of Scala and a slightly more "wordy" API results in a relatively larger source code complexity.

The long workflow has around 2x the source code complexity than the short workflow for all systems except for

[4]https://www.kaggle.com/aubertsigouin/biximtl
[5]See https://github.com/joedsilva/vldb2018

[6]In particular, a SQL statement can be written in one line or spread over several lines.

DB-UDF where it is 3.5x that of the short workflow. Most of this can be attributed to the fact that UDFs cannot interact directly with each other and that the scope of the host language objects built inside the UDF cannot outlive the UDF's execution itself. Due to this, UDFs have reduced reusability of components, requiring the users to often duplicate some aspects of the source code across multiple UDFs. Also, unlike interactive systems like AIDA, there is no mechanism in a UDF to keep reporting the changes in error rate as the training progresses. Further, although our workflows do not deliberately demonstrate this aspect, errors in an interactive system only require the user to address and resubmit the erroneous statement. Errors in UDFs, on the other hand, are hard to debug [19], and an error in the last statement in a UDF can waste the computation performed in it up to that step and would require the user to re-write the UDF, increasing the user effort and programming complexity.

Performance. As before, DB-UDF always performs best. Note that DB-UDF does not compute and report intermediate error rates during training, and thus performs slightly less computation. AIDA-RWS has roughly 1.5 the execution time of DB-UDF, and AIDA has 3-4x the execution time, the bulk due to model training. Reducing RMI overhead by pushing iterations to the server is benefiting AIDA-RWS. For both DB-UDF and AIDA, there is no large performance difference between short and long workflows because the main extra part for the long workflow is feature engineering which takes little execution time at the server side.

AIDA and AIDA-RWS always perform better than pandas and even the optimized pandas, and this by a very large margin for the long workflow. This is mainly due to the data load for pandas that AIDA avoids but also due to the feature engineering as pandas relational operators are not as efficient as MonetDB's. Spark is again the slowest by far. Although a pushdown of joins into the database has been proposed for Spark, they are still not available in the current release. Therefore, Spark has to perform its own relational joins, which as we saw in Section 6.4 has a significant cost.

6.6 Summary

AIDA, by imitating the programming style of interactive statistical frameworks, maintains their degree of usability. Data scientists, who have experience with pandas and NumPy should be able to learn AIDA fairly quickly. At the same time, AIDA can provide significant performance benefits compared to client-side packages.

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

7. RELATED WORK

Specialized databases such as SciDB [50] and its predecessors [5, 28] use array-based data structures and are aimed specifically at scientific computing, providing their own query languages. Recent studies have shown that contemporary Big data solutions such as Spark perform better on more generic scientific workloads to which many machine learning and data science projects belong [32, 46].

Recent works, EmptyHeaded [1], and it's extension LevelHeaded [2] attempts to find an acceptable architectural compromise between relational and statistical systems. Linear algebra is supported through a SQL-based syntax, similar to prior works [60, 27] that integrated linear algebra into RDBMS implementations. Joins are however possible only over key columns, restricting the use of various analytical SQL expressions such as subqueries.

Implementations such as BISMARCK [14] improvise on the conventional UDF approach by standardizing and implementing some of the specific learning algorithms as UDFs, sparing the users from the hassles, but restricting them to the algorithms exposed by the system. [7] is a schema-free append-only database for machine learning. Although it provides a SQL interface for querying the data, any learning algorithms must be written as stored procedures.

In [17], the authors use a shared memory approach to exchange data between an R process and SAP database. While this reduces any serialization overhead for network transfer, as [25] points out, still incurs data copying overhead. More recently, Apache Arrow [11] uses shared memory to facilitate applications that conform to its API to share data without copying them. AIDA, being embedded into the RDBMS, can leverage zero-copy optimizations between the two systems without a shared memory data structure.

Pushdown optimization was originally popularized by the Business Intelligence (BI) and Extract, Transform and Load (ETL) community [8, 36, 48, 22, 20] and has lately made inroads into statistical frameworks. RIOT-DB [59, 61] extends R to leverage the I/O capabilities of an external RDBMS back-end by translating expressions in R to operations in SQL by creating view definitions. Push down join optimizations has been recently attempted in Spark [3, 9], though it is still not available in the release. However, such two-system approaches cannot perform pushdown joins if one of the data sets is already in the framework and the other is still in the database. As AIDA stores all its data sets in the RDBMS memory, it can perform pushdown joins even in such cases using table UDFs.

8. CONCLUSION & FUTURE WORK

In this paper, we proposed AIDA, a unified abstraction that supports both relational and linear algebra operations seamlessly using the familiar syntax and semantics of popular Python ORM and linear algebra implementations. Execution is moved into the database system reducing data transfer costs and facilitating the use of the SQL engine whenever possible. AIDA allows users to write custom code, but without the hassles of UDF implementations. We believe AIDA is a step in the right direction to facilitate user-friendly computational interfaces to interact with an RDBMS.

Currently, AIDA works completely based on in-memory data, similar to pandas and NumPy. We are studying a least-recently-used (LRU) style memory manager module that can off-load TabularData objects that are not being actively used into database tables to reduce memory usage. We are also working on a distributed version of AIDA that could be used for very large data explorations. We also plan to develop extensions to use AIDA with existing learning and data visualization libraries.

9. REFERENCES

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

[2] C. R. Aberger, A. Lamb, K. Olukotun, and C. Ré. LevelHeaded: A Unified Engine for Business Intelligence and Linear Algebra Querying. In *ICDE*. IEEE, 2018.

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

[4] L. Ashdown, T. Kyte, et al. *Oracle Database Concepts, 12c Release 2 (12.2)*. Oracle, 2017.

[5] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann. The Multidimensional Database System RasDaMan. In *SIGMOD*, pages 575–577. ACM, 1998.

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

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

[8] U. Dayal, M. Castellanos, A. Simitsis, and K. Wilkinson. Data Integration Flows for Business Intelligence. In *EDBT*, pages 1–11. ACM, 2009.

[9] Delaney, Ioana and Li, Jia. Extending Apache Spark SQL Data Source APIs with Join Push Down, 2017. [https://databricks.com/session/extending-apache-spark-sql-data-source-apis-with-join-push-down, accessed 20-April-2018].

[10] N. Diakopoulos, S. Cass, and J. Romero. Data-Driven Rankings: the Design and Development of the IEEE Top Programming Languages News App. In *Symposium on Computation+ Journalism*, 2014.

[11] T. W. Dinsmore. *In-Memory Analytics*, pages 97–116. Apress, Berkeley, CA, 2016.

[12] P. Domingos. A Few Useful Things to Know About Machine Learning. *Communications of the ACM*, 55(10):78–87, Oct. 2012.

[13] J. W. Eaton, D. Bateman, and S. Hauberg. *GNU Octave*. Free Software Foundation, 2007.

[14] X. Feng, A. Kumar, B. Recht, and C. Ré. Towards a Unified Architecture for in-RDBMS Analytics. In *SIGMOD*, pages 325–336. ACM, 2012.

[15] E. C. Foster and S. Godbole. *Database Systems: A Pragmatic Approach*. Apress, 2016.

[16] M. Fowler. *Domain-Specific Languages*. Pearson Education, 2010.

[17] P. Große, W. Lehner, T. Weichert, F. Färber, and W.-S. Li. Bridging Two Worlds with RICE Integrating R into the SAP In-Memory Computing Engine. *PVLDB*, 4(12):1307–1317, 2011.

[18] I. Guyon and A. Elisseeff. An Introduction to Variable and Feature Selection. *Journal of Machine Learning Research*, 3(Mar):1157–1182, 2003.

[19] P. Holanda, M. Raasveldt, and M. Kersten. Don't Hold My UDFs Hostage-Exporting UDFs For Debugging Purposes. In *International Conference on Simpósio Brasileiro de Banco de Dados (SSBD)*, 2017.

[20] IBM. IBM InfoSphere DataStage Balanced Optimization. White paper, June 2008.

[21] R. Ihaka and R. Gentleman. R: A Language for Data Analysis and Graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, 1996.

[22] Informatica Corporation. How to Achieve Flexible, Cost-effective Scalability and Performance through Pushdown Processing. White paper, November 2007.

[23] A. M. Kuchling. The Python DB-API. *Linux Journal*, 1998(49es):8, 1998.

[24] A. Kumar, M. Boehm, and J. Yang. Data Management in Machine Learning: Challenges, Techniques, and Systems. In *SIGMOD*, pages 1717–1722. ACM, 2017.

[25] J. Lajus and H. Mühleisen. Efficient Data Management and Statistics with Zero-Copy Integration. In *International Conference on Scientific and Statistical Database Management*, pages 12:1–12:10. ACM, 2014.

[26] 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 *VLDB*, pages 294–305, 1988.

[27] S. Luo, Z. J. Gao, M. Gubanov, L. L. Perez, and C. Jermaine. Scalable Linear Algebra on a Relational Database System. In *ICDE*, pages 523–534. IEEE, 2017.

[28] A. P. Marathe and K. Salem. Query Processing Techniques for Arrays. In *SIGMOD*, number 2, pages 323–334. ACM, 1999.

[29] D. Marten and A. Heuer. A Framework for Self-Managing Database Support and Parallel Computing for Assistive Systems. In *International Conference on PErvasive Technologies Related to Assistive Environments*, page 25. ACM, 2015.

[30] M. M. McKerns, L. Strand, T. Sullivan, A. Fang, and M. A. Aivazis. Building a Framework for Predictive Science. *CoRR*, abs/1202.1056, Feb. 2012.

[31] W. McKinney. pandas: a Foundational Python Library for Data Analysis and Statistics. *Python for High Performance and Scientific Computing*, pages 1–9, 2011.

[32] P. Mehta, S. Dorkenwald, D. Zhao, T. Kaftan, A. Cheung, M. Balazinska, A. Rokem, A. Connolly, J. Vanderplas, and Y. AlSayyad. Comparative Evaluation of Big-Data Systems on Scientific Image Analytics Workloads. *PVLDB*, 10(11):1226–1237, 2017.

[33] S. Melnik, A. Adya, and P. A. Bernstein. Compiling Mappings to Bridge Applications and Databases. *Transactions on Database Systems*, 33(4):22, 2008.

[34] J. Melton and A. R. Simon. *SQL:1999: Understanding Relational Language Components*. Morgan Kaufmann Series in Data. Morgan Kaufmann, 2002.

[35] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, et al. MLlib: Machine Learning in Apache Spark. *The Journal of Machine Learning Research*, 17(1):1235–1241, 2016.

[36] MicroStrategy. Architecture for Enterprise Business Intelligence. White paper, MicroStrategy, Incorporated, 2012.

[37] T. Miller. Using R and Python in the Teradata Database. White paper, Teradata, 2016.

[38] H. Mühleisen and T. Lumley. Best of Both Worlds: Relational Databases and Statistics. In *International Conference on Scientific and Statistical Database Management*, pages 32:1–32:4. ACM, 2013.

[39] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12(Oct):2825–2830, 2011.

[40] G. Piatetsky. New Leader, Trends, and Surprises in Analytics, Data Science, Machine Learning Software Poll, 2017. [https://www.kdnuggets.com/2017/05/poll-analytics-data-science-machine-learning-software-leaders.html, accessed 07-December-2017].

[41] F. Plášil and M. Stal. An Architectural View of Distributed Objects and Components in CORBA, Java RMI and COM/DCOM. *Software-Concepts & Tools*, 19(1):14–28, 1998.

[42] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2014.

[43] M. Raasveldt. Voter Classification Using MonetDB/Python, 2016. [https://www.monetdb.org/blog/voter-classification-using-monetdbpython, accessed 07-December-2017].

[44] M. Raasveldt and H. Mühleisen. Vectorized UDFs in Column-Stores. In *International Conference on Scientific and Statistical Database Management*, pages 16:1–16:12. ACM, 2016.

[45] M. Raasveldt and H. Mühleisen. Don't Hold My Data Hostage–A Case For Client Protocol Redesign. *PVLDB*, 10(10):1022–1033, 2017.

[46] L. Ramakrishnan, P. K. Mantha, Y. Yao, and R. S. Canon. Evaluation of NoSQL and Array Databases for Scientific Applications. In *DataCloud Workshop*, 2013.

[47] C. Ré, D. Agrawal, M. Balazinska, M. Cafarella, M. Jordan, T. Kraska, and R. Ramakrishnan. Machine Learning and Databases: The Sound of Things to Come or a Cacophony of Hype? In *SIGMOD*, pages 283–284. ACM, 2015.

[48] SAP. *SAP BusinessObjects Web Intelligence User's Guide*. SAP SE, 2017.

[49] R. D. Schneider. *MySQL Database Design and Tuning*. Pearson Education, 2005.

[50] M. Stonebraker, P. Brown, D. Zhang, and J. Becla. SciDB: A Database Management System for Applications with Complex Analytics. *Journal of Computing in Science & Engineering*, 15(3):54–62, 2013.

[51] The PostgreSQL Global Development Group. Procedural Languages. In *PostgreSQL 10.0 Documentation*, 2017.

[52] Theano Development Team. Theano: A Python Framework for Fast Computation of Mathematical Expressions. *arXiv e-prints*, abs/1605.02688, May 2016.

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

[54] T. Vincenty. Direct and Inverse Solutions of Geodesics on the Ellipsoid with Application of Nested Equations. *Survey Review*, 23(176):88–93, 1975.

[55] S. v. d. Walt, S. C. Colbert, and G. Varoquaux. The NumPy Array: A Structure for Efficient Numerical Computation. *Journal of Computing in Science & Engineering*, 13(2):22–30, 2011.

[56] W. Wang, M. Zhang, G. Chen, H. Jagadish, B. C. Ooi, and K.-L. Tan. Database Meets Deep Learning: Challenges and Opportunities. *SIGMOD Record*, 45(2):17–22, 2016.

[57] B. Woody, D. Dea, D. GuhaThakurta, G. Bansal, M. Conners, and T. Wee-Hyong. *Data Science with Microsoft SQL Server 2016*. Microsoft Press, 2016.

[58] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing With Working Sets. *HotCloud*, 10(10-10):95, 2010.

[59] Y. Zhang, H. Herodotou, and J. Yang. RIOT: I/O-Efficient Numerical Computing without SQL. In *CIDR*, 2009.

[60] Y. Zhang, M. Kersten, and S. Manegold. SciQL: Array Data Processing Inside an RDBMS. In *SIGMOD*, pages 1049–1052. ACM, 2013.

[61] Y. Zhang, W. Zhang, and J. Yang. I/O-Efficient Statistical Computing with RIOT. In *ICDE*, pages 1157–1160. IEEE, 2010.