SQL queries over Python objects using an embedded column-based relational database

Student: Weiming Guo Supervisors:Prof. Bettina Kemme, Dr. Joseph D'silva
McGill University

Abstract

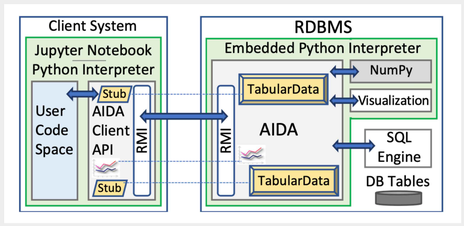
Modern data science workflows require support for linear algebra, in addition to relational algebra provided by traditional database engines. AIDA (An Agile Abstraction for Advanced In-database Analytics) is an in-database data science framework that allows data scientists to perform linear algebra operations using NumPy. The fundamental differences in the data storage and processing needs of NumPy and traditional databases mean that they do not seamlessly integrate well with each other, requiring data format conversions between the systems.
This summer, my research focused on exploring how to avoid the cost of such data conversions, thus improving the efficiency of data processing.

Introduction

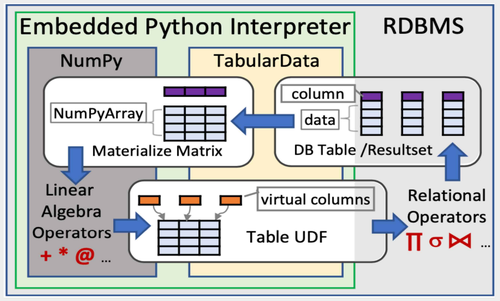
Existing research has shown that using columnar (each column is stored as a separate array of its own) databases reduces the need for data conversions between NumPy data and RDBMS as the internal data structures are very similar to that of the statistical systems.
In this summer, I spent most time on exploring how these benefits offered by column-based databases could be leveraged for AIDA's implementations on row-based databases such as PostgreSQL where all the data for a record is stored continuously as a separate array and is therefore different from the storage format employed by the statistical systems.

My first approach was to embed a columnar database to AIDA framework. The high-level idea was to use this embedded database for queries on data that has been materialized in the statistical system, as this would reduce the need for data conversion costs associated with passing it to the host database. However, the embedded columnar database did not integrate well with PostgreSQL when I initialized it inside AIDA framework. As such, I switched to a second approach, which was to create two AIDA servers for both MonetDB and PostgreSQL and use MonetDB AIDA for queries over materialized data. In the next few sections, I will explain the second approach in detail and show you the performance of the second approach when processing large data sets.

Background



The above picture depicts a high-level conceptual layout of the client-server architecture of AIDA. "AIDA's server is embedded within the Python interpreter embedded in the RDBMS, thus sharing the same address space as the RDBMS. Users can connect to AIDA's server using a regular Python interpreter or more popular data science IDEs like Jupyter Notebook that works with Python, as long as they are equipped with AIDA's client API library. " [1]When using AIDA, data scientists implement programming logic on the client side, but any data transformations and computations are executed on the server side. More precisely, AIDA's client API sends them transparently to the server and receives a remote reference which represents the result that is stored in AIDA's server. This client-server interaction is implemented by using Remote Method Invocation (RMI), a well-established communication mechanism, and known to work in practice. This will also allow us to easily extend AIDA to be part of a fully distributed computing environment where data might be distributed across many RDBMSes.



As discussed before, modern data science applications require a holistic framework that supports both relational and linear algebra operations. AIDA accomplishes this through a unified abstraction of data called TabularData.TabularData object can process data stored in the database and the data stored in host language objects such as NumPy. Internally, AIDA uses RDBMS's SQL engine to execute relational operations and utilizes NumPy to execute linear algebra operations. When needed, AIDA allows data transformations to be performed transparently and seamlessly between the two systems.

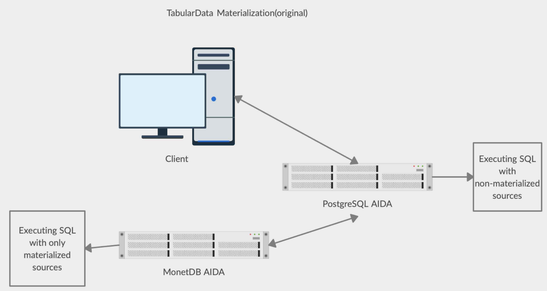
There are two internal representations for TabularData. One of the representations is a matrix format (a two-dimensional array representation where the entire data set is located in single contiguous memory space) used to execute linear algebra operations and the other is a dictionary-columnar format (a python dictionary with column names as keys and the values being the column's data in NumPy array format) used to execute relational operators. Users can access the dictionary internal representations by using the special variable cdata. Similarly, the matrix representation can be accessed by using the special variable matrix.

AIDA performs relational operations on TabularData lazily, ie. the TabularData object's internal representation is only materialized when the user explicitly requests it through the special variables mentioned in the previous section. "This means that, before materialized, the new TabularData 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." [2] If a TabularData object is requested to be materialized, AIDA would recursively request its sources and build SQL logic on top of it. The resulting combined SQL logic would then be executed in the host RDBMS. Once materialized, the internal representation will exist for the lifetime of that TabularData object, reused for any further operations as required.

While this method is optimized to work with column-based RDBMS implementations such as MonetDB that requires no data transfer costs associated with passing materialized TabularData to it, it's not optimal for row-based databases such as PostgreSQL. As a result of the fundamental difference in data storage format between row-based RDBMS and statistical system, materialized TabularData that is in a dictionary-columnar format needs to be transformed before being executed to accommodate row-based RDBMS, thus resulting in the overhead of data format conversion.

AIDA allows data scientists to perform data analysis across multiple database implementations that host their own AIDA servers by using _L operator. In such scenarios, we can connect first to a first AIDA server from our client, perform appropriate transformations, and then pass the TabularData objects generated there to a second AIDA server. We can then continue the analysis on this TabularData object at the second AIDA server. In the next section, I will discuss how this functionality can be applied to the approach of reducing the overhead of data format conversion.

Method



The above picture depicts the high-level architecture of the approach of leveraging the benefits offered by columnar RDBMS MonetDB for AIDA's implementations on row-based RDBMS PostgreSQL. When performing data analysis, the user will connect to PostgreSQL AIDA from client, and internally, the PostgreSQL AIDA's adapter will open a connection to MonetDB AIDA during initialization.

When a user requests the materialization of a TabularData object from client, PostgreSQL AIDA will first build up SQL for it and then recursively check if the TabularData object has only NumPy sources, ie if its sources are materialized in the statistical system NumPy. If the TabularData object has only NumPy sources, AIDA would send all materialized sources to MonetDB AIDA server by using _L operator as discussed before and generate necessary table UDFs there. Then the SQL would be executed in MonetDB that offers zero-copy data transfer of NumPy sources from statistical system to database memory space, thus reducing the overhead of data format conversions.

Results

	test 1	test 2	test 3	test 4	test 5	AVG
case1 (None NumPy Data)	0.02471	0.02127	0.02514	0.02597	0.02355	0.024128
case2 (One NumPy Data)	0.07509	0.07695	0.07556	0.07039	0.06877	0.073352
case3 (All NumPy Data)	0.30762	1.39039	0.31849	0.28243	0.30166	0.520118

	test 1	test 2	test 3	test 4	test 5	AVG
case1 (None NumPy Data)	0.02495	0.02127	0.02456	0.02462	0.02496	0.024072
case2 (One NumPy Data)	0.07977	0.07343	0.07583	0.08481	0.07793	0.078354
case3 (All NumPy Data)	0.09154	0.09154	0.08312	0.08604	0.07978	0.086404

I used three cases to test the modified version of TabularData Materialization. In test case 1, the TabularData object that needs to materialize has no NumPy sources. In test case 2, the TabularData object has one NumPy source. In test case 3, the TabularData object has only NumPy sources. MonetDB AIDA is only required in case3 when the TabularData object has only materialized sources.
To compare the efficiency of materialization with and without MonetDB AIDA, I imported a powerful python package time to measure the execution time of each case.
From the pictures above, we can see that it's slightly better to perform case2 with AIDAM. However, it takes a longer time to perform case3 with AIDAM, which is not very efficient as we expected before.

Future Work

Ideally, sending NumPy data to MonetDB AIDA would reduce the cost of data conversions between statistical systems and the RDBMS query engine since they both store data in the columnar storage format. However, the test results show that this method has no advantage over the previous one in terms of efficiency. In the future, we would try to find out the part that contributes to the extra time and improve this method.

References

[1] De Moor F. Kemme B. D'silva, J. V., editor. AIDA - An Agile Abstraction for Advanced In-database Analytics. Proceedings of the VLDB Endowment, 2018.