When sweet and cute isn't enough anymore: Solving scalability issues in Python Pandas with Grizzly

Stefan Hagedorn
TU Ilmenau, Germany
stefan.hagedorn@tu-ilmenau.de

1. MOTIVATION

The giant panda bear is very popular, not only because of the (seemingly) cute and friendly face and behavior. However, due to their poor diet they are slow and clumsy. In this sense, Python Pandas is very similar to the bears: The framework has a nice and user-friendly appearance, but, under the hood, requires a lot of resources (memory, CPU time) even to process data sets of moderate size. Relational DBMS are highly optimized for storing and querying large amounts of data, but complex analysis tasks are often difficult or even impossible to express in SQL. Thus, easy-to-learn scripting languages such as Python or R became very popular and the de-facto standard for data science tasks. The Pandas *DataFrame*s re-implement operations known from SQL, such as projection, selection, join, grouping etc. Therefore, a sequence of Pandas operations could also be expressed as SQL and executed in a DBMS or SparkSQL.

2. THE GRIZZLY FRAMEWORK

Grizzly is a transpiler from a Pandas-like Python API to SQL used to push Python operations into the DBMS where data is stored in order to benefit from its optimized data processing capabilities. The lightweight architecture does not require changes to be made to the DBMS and can therefore be used with any SQL engine.

Grizzly supports projection, selection, join, aggregation with and without grouping and experiments have shown that execution time as well as memory consumption on the user's PC were reduced to only a fraction compared to Pandas.

While the prototype is fully functional[1], there are a few challenges and opportunities being addressed in our ongoing work:

1. Dealing with External Data: Although we argue that for most cases data is already stored in an RDBMS, there are use cases where data is present in (local) files only. For such cases, already existing techniques known from various projects (PostgresRaw, Foreign Data Wrappers in PostgreSQL, or External Tables in Actian Vector, etc.) can be utilized. For this scenario, we further envision a cost model to decide when to import the external data as regular tables into the DBMS.

2. User Defined Functions: While most standard operations to analyze a data set's characteristics are already present in the DBMSs, users often need to implement use case specific algorithms to process the data. In order to execute such UDFs within the DBMS, their Python code needs to be transferred to the DBMS. This can be achieved several ways: (1) If the DBMS supports Python as a language for stored procedures/functions, the source code of the UDF can be fetched via reflection tools and be sent to and be installed at the DBMS, or (2) the Python code is transpiled into a language the DBMS supports, e.g., PL/pgSQL for PostgreSQL.

3. Recycling Intermediate Results: When many researchers in a team work with the same data sets, many operations on the input data are performed again and again. To avoid the costly repeated execution of the same operations, a caching strategy can be applied to identify frequently used sub-expressions and materialize their results for later re-use. In [1] we presented a cost model to materialize and reuse such intermediate results and showed that this approach can have a enormous benefit on query execution time. Recently, [2] also showed that materializing the result of common sub-expressions of SQL queries reduces the *machine hours* in data centers up to 40%.

4. Self-tuning: The framework can further be extended by approaches from self-tuning and even self-driving databases to automatically derive an optimal partitioning scheme or create indexes on-the-fly.

5. Multiple Target Platforms: Currently, we generate SQL queries only. However, since the Pandas API is widespread and well-known, we believe that it could be used in various cases. Thus, we are working on decoupling the SQL generation from the core and allow to add custom code generation plugins. This way, and with the respective language extensions, Grizzly could be used to create queries for various platforms, including stream processing.

[1] https://github.com/dbis-ilm/grizzly

10th Annual Conference on Innovative Data Systems Research (CIDR '20) January 12-15, 2020, Amsterdam, Netherlands.

3. REFERENCES

[1] S. Hagedorn and K. Sattler. Cost-based sharing and recycling of (intermediate) results in dataflow programs. In *ADBIS*, pages 185–199. Springer, 2018.

[2] A. Jindal, K. Karanasos, S. Rao, and H. Patel. Selecting subexpressions to materialize at datacenter scale. *Proc. VLDB Endow.*, 11(7):800–812, 2018.