CacheBack: Automatic Server-side Caching with PostgreSQL

Jeein Kim*
jeein.kim@student.hpi.de
Hasso Plattner Institute
Potsdam, Germany

Ashish Patel*
ashish.patel@student.hpi.de
Hasso Plattner Institute
Potsdam, Germany

Abstract

Data science is an iterative development paradigm which often involves collaborative efforts and perpetual experimentation with data. The de-facto tool for data exploration and analysis is Jupyter Notebook, which can be used in an isolated local environment, or with collaborative environments such as Google Colab [2] or Amazon Sagemaker [1]. However, the fundamental critique of traditional notebooks is the life cycle of its Kernel, whose interruption ultimately leads to the loss of states of referenced data objects. Hence, a developer willing to continue with the interrupted progress, will have to re-execute all cells sequentially. This introduces friction in the development process in the form of time delays and unnecessary usage of resources. Typically, the data for analysis is stored in relational databases like PostgreSQL or MySQL. Assuming that data is stored on a relational database, we propose a novel solution: CacheBack, that automatically caches materialized variables to the PostgreSQL database, so that the future execution of pipeline can read data directly from the cache, thus bypassing any redundant execution steps of a data science pipeline. CacheBack is designed to abstract out caching and only requires the developer to mark objects that might be used in future runs. We show that selectively caching important variables can significantly reduce overall execution times and resource consumption.

Keywords: Caching, Notebooks, PostgreSQL

1 Introduction

Jupyter notebooks have gained tremendous popularity over the past decades for data science and research projects. The ability to narrate code and create visualizations is one of the key reasons for its increasing popularity [6]. Backed up by an easy-to-use user interface, convenience of documenting code, and visualization support, Jupyter notebooks have completely revolutionized scientific research and statistical analysis.

Nearly all data science workloads are similar in nature from a high-level perspective, i.e. all of these workloads involve general phases like data loading, pre-processing, training and serving, which are always sequential in nature. One important thing to note here is that data science activities often involve continuous experimentation with model parameters and fewer changes are made to the data itself. Hence in the pipeline proposed in the Figure 1, we can state that training and serving are the dynamic phases here, and the data preparation stage is done only once.

We argue that although traditional notebooks are better than regular Python scripts, in the sense that they persist the outputs of each executed cell; however, they still suffer from loss of states when the respective kernel is killed. It is a frustrating problem faced by developers worldwide and the only quick solution is to re-execute all cells from the beginning [11]. This problem becomes increasingly complicated when the analysis relies on vast amounts of data.

Usually, the data used for analysis is ingested from a remote database server like PostgreSQL [7]. As Raasveldt et al. [14] argue, the operations of serializing and deserializing datasets from DBMS to clients are also expensive in terms of time. Consequently, re-executing all cells from the beginning, data needs to be fetched again from the database server, which introduces delay as the data transfer takes place over the network. This problem becomes increasingly challenging as the size of the dataset increases. Thus, this is a common problem faced by data scientists, which has not been solved completely yet.

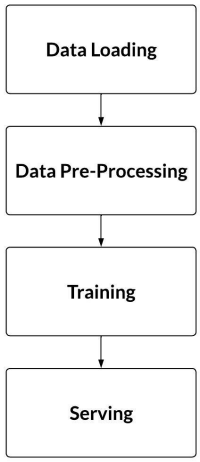


Figure 1. Data Science Workflow

With the aforementioned problems present, it is apparent that the pre-processed data must be cached meticulously.

*Both authors contributed equally to this research.

This approach provides several advantages: 1) Storing data in a persistent storage; 2) Saving time from not running common pre-processing steps; and 3) Users working on an agreed state, which eliminates potential confusion.

The following sections demonstrate our proposed solution CacheBack thoroughly and explicitly mention its implementation details and design assumptions. A comprehensive explanation of the CacheBack architecture is discussed in Section 3 and Section 4 provides evidence on the outcomes of using CacheBack and pragmatically analyzes the applicability of CacheBack.

2 Background

In this section, we aim to illustrate the internals of Jupyter, more specifically we provide information on the life cycle of a Jupyter kernel and state management inside the Jupyter environment.

A Jupyter notebook [16] is a sequence of cells, which may contain code or markdown. Internally these cells contain data such as code or markdown content, execution count, and outputs in the form of JSON.[1] Thus, outputs are saved in text/multimedia format depending on the type of output produced and persisted along with the notebook. This is one of the primary reasons for the popularity of Jupyter notebooks, i.e. outputs from the prior execution can be reviewed without running the respective cells.

The notebook works as a frontend to a kernel that runs the Python processes.[2] This kernel is a separate process responsible for executing the code and capturing the generated outputs. It interacts with the frontend, like Notebooks or Qt console, using JSON messages transferred over ZeroMQ sockets.[3] Each Kernel can be connected to one or more frontends at the same time.

The Kernel can be interrupted or shut down manually. In such cases, the kernel is no more active and the respective kernel process is killed from the background. Since the jupyter frontend interacts in an interactive fashion with the kernel, intermediate outputs are saved locally in the form of JSON. Thus, all the outputs generated prior to the interruption of Kernel are still preserved in the `.ipynb` file format [10].

However, the outputs are merely text/multimedia objects and not in-memory objects. Hence, all the cells preceding a given cell, need to be executed again in order to obtain the in-memory state of the object.

In this paper, we demonstrate our approach towards solving this problem and introduce the first prototype of our solution: CacheBack. Overall, we make the following contributions through this paper -

1. We provide a reasonable solution: CacheBack, which utilizes the availability of relational DBMS and uses it as a global cache to maintain and update data science pipelines. We designed CacheBack in a way that all the caching processes are carried out by the DBMS server. This design was adopted since we did not want to transfer pre-processed data from the client to the server again, and thus perform computations closer to the data source. We observed that the major bottleneck in the caching process is writing cached data to DBMS, which relies on Pandas functions that are inherently slow.
2. We prove the potential of using PostgreSQL as an execution environment. Most of the databases today support Python UDF execution, however, the capabilities and performance of the Python interpreter utilized by the database are not well studied. To test the viability of CacheBack, we conducted experiments to test the abilities of PostgreSQL Python interpreter, which we show provides comparable performance as a regular Python kernel.
3. We explain the potential use cases when caching provides better run times and alternatively, the cases when caching pre-processed data degrades overall execution times. The detailed analysis of our findings is explained in the section Evaluation.

3 CacheBack Overview

In this section, we present our novel solution, CacheBack, primarily motivated by the aforementioned problems. CacheBack is a caching mechanism that stores existing variables in a Notebook, either caching them all or manually selecting which ones to cache. Our assumption for when using CacheBack is that the relevant data to retrieve for are located in a remote DBMS. Therefore, CacheBack also utilizes its existing DBMS to "cache back" relevant variables, or pre-processed works. After its initial caching session, a new notebook with the pointers to the cached variables inside DBMS is returned to the users. As a result, the users only need to run the freshly returned notebook, which will retrieve the cached variables automatically. The details of these processes are covered under 3.1 Design and Architecture.

Currently, CacheBack supports caching pandas dataframes [12], with PostgreSQL [7] for the choice of DBMS.

3.1 Design and Architecture

This section describes the design assumptions and architecture of CacheBack. We have approached the design primarily from the user's perspective, i.e. we have ensured minimal user interaction while still maintaining accuracy while caching the precise variables. Section 3.1.1 describes

[1] http://ipython.org/ipython-doc/3/notebook/nbformat.html
[2] https://docs.jupyter.org/en/latest/projects/kernels.html#kernels-langs
[3] https://docs.jupyter.org/en/latest/projects/architecture/content-architecture.html

CacheBack from a user's view point and section 3.1.2 describes the CacheBack architecture in more technical terminologies.

3.1.1 User Perspective: From a user's perspective, they simply need to import our library `cacheback` in their notebooks. This can simply be installed via:

```
pip install cacheback
```

This can be done on any terminal with `pip` installed. Users may select which variables to cache into the DBMS. If they do not specify them, the default behavior is to cache all variables. The following code snippet demonstrates this use case -

```
from cacheback import cache_back
cache_back.add_to_cache(var, 'var_name')
```

Finally, users will use our command line interface (CLI), so they can send a notebook to the DBMS, where CacheBack dependency is pre-installed to support our caching mechanism. Ultimately, after caching is done on the server side, users will receive modified notebooks that contain pointers to the cached variables inside DBMS. The following command illustrates the usage of our CLI -

```
python cli.py --user USER --password PASSWORD \
    --host HOST --port PORT --db DB \
    --notebook NOTEBOOK_TO_CACHE
```

These arguments are needed for a successful connection between the client and the DBMS.

3.1.2 Technical Perspective: Note that when users are using our CLI, the only file that is sent to the server is the notebook itself, not the variables to be cached. This implies that when the server receives a notebook, it will first run the received notebook using the server machine and cache from them. This has a multitude of induced advantages: 1) utilizes the server's powerful resources, and 2) eliminates network overheads.

When a user sends a notebook using the CLI, the server receives the notebook and translates it from `.ipynb` format to `.py` format. This allows the DBMS to utilize `PL/Python`, or Python Procedural Language, which makes PostgreSQL functions and procedures to be written in the Python language. Therefore, having a translated `.py` script into a PostgreSQL function using `PL/Python` allows users to run any code, as long as relevant Python libraries are installed inside the DBMS. With this powerful feature of PostgreSQL, CacheBack will run this translated script inside the DBMS.

The following code snippet is how the translated Python code is prepared for the execution as `PL/Python` function:

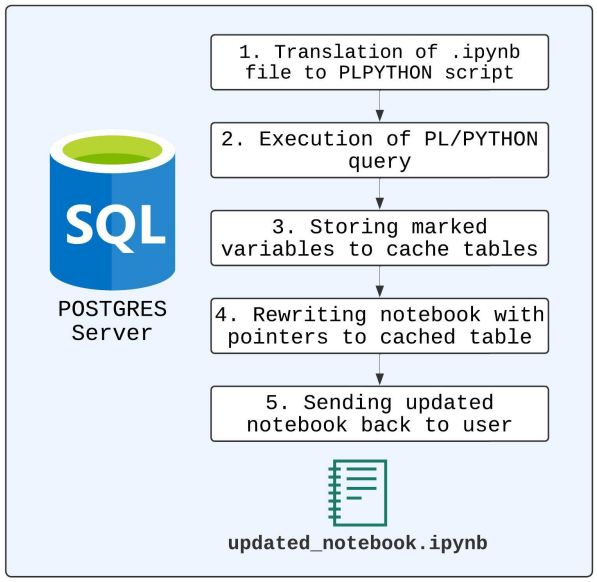


Figure 2. Caching workflow on PostgreSQL server.

```
CREATE OR REPLACE FUNCTION {notebook_name}()
RETURNS TEXT
AS $$
    [translated .py script here]
$$
LANGUAGE plpython3u;
```

Note that the language of `PL/Python` function is `plpython3u`. The u here stands for "untrusted". This means that the system does not restrict its users, in terms of what they can do.[4] A future work of CacheBack will include such mechanism that prevents users from potentially damaging the DBMS server.

As previously mentioned, since the server only takes a Notebook file as input, the server needs to run the translated script. When CacheBack finds the caching code in the middle, i.e., `add_to_cache()`, CacheBack will keep the list of the variables to cache for after the execution of the original script. If no such code exists in a file, CacheBack caches all Pandas dataframes by default. Right before the execution happens inside `PL/Python`, CacheBack will add header and footer codes that will correctly call the necessary CacheBack library functions for proper caching.

Pandas dataframes have a method named `to_sql()`.[5] We utilize this method for caching the variables into the DBMS. This method requires a connection string for DBMS connection. Using the arguments that was provided when executing

[4] https://www.postgresql.org/docs/current/plpython.html
[5] https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.to_sql.html

the CLI, the connection string is made. Again, these are automatically added in the header and the footer codes.

When multiple notebooks try to take advantage of CacheBack, sometimes there might be the same variable names amongst different notebooks. Currently, our design relies on the variable name as the table name on the DBMS. In order to solve this conflict, we are not only adding the variable name but also appending the hash of the notebook name, using `MD5` algorithm. In this way, as long as there are no two notebooks with the exact same name, no tables will be overwritten unintended.

After the caching is done inside the DBMS, CacheBack will return the notebook to its users with the updated pointers to the cached tables. This means that if the original notebook had code snippets that ask for caching, then these codes are automatically commented out (since caching them twice does not make sense), and the code that calls the cached tables is automatically written.

Again, when the caching mechanism is being done, the network overhead will only be caused by these two files: the original notebook and the returned notebook.

The following code snippet is an example of a re-written notebook:

```
# raw_data = load_data()
# cache_back.add_to_cache(raw_data, 'raw_data')

cursor.execute(
    'SELECT * FROM raw_data_d41d8')
raw_data =
    pd.DataFrame(cursor.fetchall(),
    columns = \
    [desc[0] for desc in cursor.description])
```

The original code lines where it normally executes and caches are automatically commented out. Instead, two additional lines are added to retrieve the cached tables from the DBMS. With the pre-defined cursor, which is automatically created by our CacheBack, it will issue an SQL query and get results back to the cursor. Note that the name of the table is the variable name plus the hash of the notebook name. Finally, CacheBack will create a Pandas dataframe from the cursor, including the column names of the table. This way, users do not need to run the original code, but simply retrieve the cached tables from the DBMS.

4 Evaluation

Our primary testing objectives are (i) to establish the feasibility of using postgreSQL as an execution environment; (ii) to understand the relationship between overall execution runtimes and data set size; (iii) to study the overhead of caching on initial run and justify how one expensive initial run provides extremely better runtimes in subsequent runs; (iv) to evaluate the end-to-end runtime performance of data science pipelines using CacheBack and native Python execution.

Against our **CacheBack**, its comparison group will be simply non-CacheBack runs; that is, executing a notebook as-is. We call this **non-CacheBack**. Our primary interest in CacheBack is achieving a faster runtime. Therefore, we primarily focus on measuring overall runtime and also runtime for individual steps, which will be explained in detail below. We were also interested in measuring the overhead of caching, such as how much storage the cached tables take up inside the DBMS. Therefore, we also measure the sizes of these tables for comparison: 1) tables that are used/needed for running the notebook; 2) tables that were produced from caching, which are simply the variables marked for caching. We discuss the interpretations of the results with respect to different scenarios.

4.1 Experiment Setup

All experiment was conducted under the Hasso Plattner Institute's Scientific Computing Operations and Research Environment Lab. We have setup the client and the DBMS under the same machine using docker containers. This means there is no additional network overhead. The evaluations were carried out on a machine with AMD Ryzen Threadripper PRO 3995WX Prozessor (64 Kerne, 128 Threads, 32 MB Cache, up to 4.20 GHz), 64 GB DDR4 memory, 2 of NVIDIA RTX A5000 24 GB with NVIDIA NVLink Bridge, and 1 TB M.2 SSD.

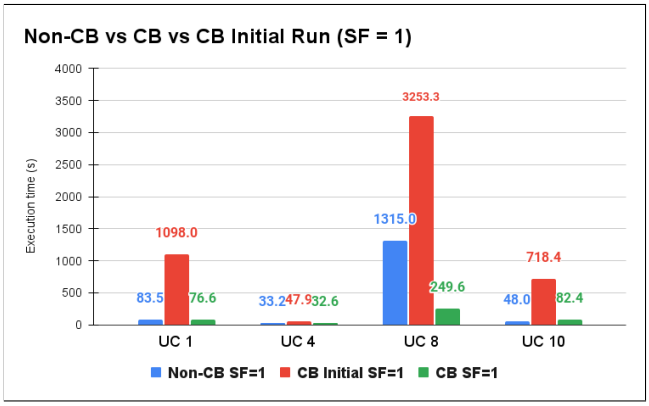
The choice of DBMS is PostgreSQL, and we use Jupyter Notebook for the notebook environment. In addition, we utilise Docker container [9], so the transition of machines does not affect our development and testing. We have a Docker container with Ubuntu 22.04.1 LTS, and inside this container, PostgreSQL (15.1) Docker container is located. The PostgreSQL is equipped with an embedded Python interpreter, which our `PL/Python` uses. Its version is 3.9.2. Some other common Python libraries include Pandas 1.5.2, Numpy 1.23.5, psycopg2-binary 2.9.5, and SQLAlchemy 1.4.45.

4.2 Description of Use Cases

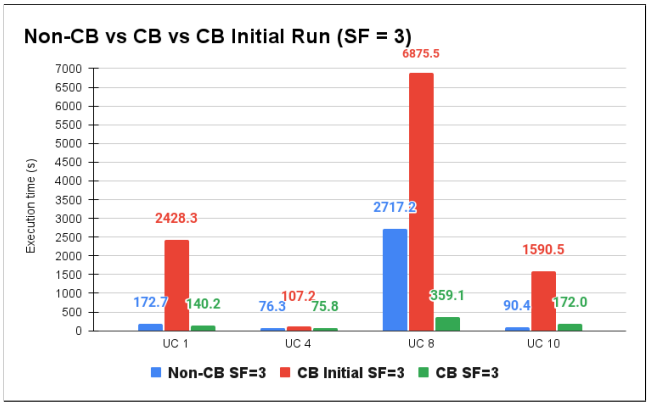
To create different but consistent scenarios, we take advantage of `TPCx-AI`, an end-to-end AI benchmark standard developed by the Transaction Processing Performance Council (TPC). [3] The `TPCx-AI` benchmark comprises 10 use cases that can be used with different scale factors. For simplicity, we have only excluded use cases that rely on images or audio. Therefore, out of the 10 workloads we have picked 4 relevant use cases which deal with textual data. For the evaluations we have used scale factors 1 and 3, which correspond to 1GB and 3GB of the data. The specifications of each use case are explained in detail on the official `TPCx-AI` documentation.[6] The use cases used for evaluations are described below -

[6] https://www.tpc.org/tpc_documents_current_versions/pdf/tpcx-ai_v1.0.2.pdf

1. **Use Case 1 (Customer Segmentation):** This use case is designed to segregate customers based on their spending behavior. It uses K-means clustering algorithm and uses the Customer, Order, Lineitem, and Order_returns tables from the database.
2. **Use Case 4 (Spam Detection):** This use case performs Spam detection on the reviews data. It trains a Naïve Bayes model to generate accurate predictions. This use case loads data from database eliminates duplicates, fits the model, and computes predictions. It uses the Reviews table.
3. **Use Case 8 (Classification of Taxi Trips):** Use Case 8 is an emulation for the classification of shopping trips, where it has an underlying assumption that different trip types have different characteristics. This use case contains some complex pre-processing steps, which include data cleaning, feature extraction (binarising categorical values), and item aggregation.
4. **Use Case 10 (Fraud Detection):** This use case emulates Fraud detection and aims to train a Logistic regression model to classify a transaction as fraudulent. It uses the Financial_Account and Financial_Transaction tables. It performs data loading and joins those tables, cleans data, performs some calculations to derive relevant data, trains a model, and computes predictions.



(a) Scale Factor 1



(b) Scale Factor 3

Figure 3. Overall End-to-End runtimes of both Non-CacheBack, CacheBack's initial run and CacheBack final pipeline, for each use cases, with Scale Factors = 1 and 3 respectively

4.3 End-to-End Runtime Analysis

When measuring the runtimes of both CacheBack and non-CacheBack, we not only measure the overall runtime, but also the overall runtime divided into four categories: load, pre-process, train, and serve. Each use case has different functions inside of these categories, but the common part is that, out of these four categories, the first two: load and pre-process, are intended to be optimized by CacheBack. Although the loading and pre-processing phases are separated, the loading phase also contains some pre-processing operations. Therefore, a performance boost is expected in these two categories.

Each use case is measured for ten times, and we average the result. Note that CacheBack requires an initial run to cache the variables marked for caching. This means that during the initial run, the whole script will be run, plus the caching part comes in. Therefore, it is expected to take longer than simply running the script for the first time, but we hope that all the subsequent runs after caching show performance improvements. This means that for non-CacheBack, simply the script is run ten times, and for CacheBack, the initial caching run is executed once, and the subsequent runs are executed nine times. We compare the non-CacheBack, CacheBack initial, and CacheBack subsequent runtimes in Figure 5.

4.4 Initial Execution with CacheBack

The deployment of the notebook to the postgreSQL server is made possible by the CLI as described in the section 3.1. After the notebook has been sent to the server, the initial execution of the pipeline begins on the PostgreSQL `PL/Python` interpreter. This initial run has additional overheads, hence showing a larger overall execution time. The overhead of setting up Python instance on the server has some minor performance penalties. Figure 3 illustrates the overall runtime comparison between native execution of pipeline with initial notebook execution on PostgreSQL server.

A detailed evaluation of the performance differences between regular Python environments and `PL/Python` is covered in Section 4.5. Additionally, the overhead of caching variables contributes a vast majority of the initial execution time. This is because writing cached variables to DBMS relies on `pandas.to_sql()` function, which is an extremely

costly I/O operation. At the same time, deciding the appropriate variables to cache is an essential step that requires a developer's sophisticated understanding of the pipeline.

4.5 Subsequent Executions

After variables have been cached, the subsequent runs fetch data directly from the cache, and hence the advantages of caching become apparent. Figure 4 and Figure 5 elucidate the breakdown of runtime in several stages of the pipeline execution for use cases 8 and 10 respectively. These stages are data loading, pre-processing, training, serving, and total execution times. The relevant stages are the loading and pre-processing steps and also the overall execution time. For scale factor 3, use case 8 observed a huge reduction in execution times from 45 min to under 6 minutes, thus proving that CacheBack accelerates runtime for subsequent runs drastically.

Use cases 1 and 4 are not covered in evaluations since they do not involve complex pre-processing steps. Moreover, use cases 1 and 4 depicted the same performance benefit yet the performance gain is not significant enough to be discussed separately.

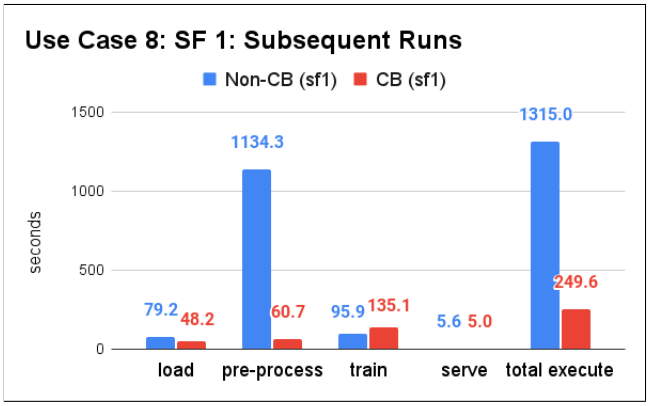
We noticed that the trend is consistent across all the use cases except for use case 10, where CacheBack performed worse. A detailed analysis on why caching is not a good strategy for specific workloads is given in Section 4.6.

4.6 Caching Overhead

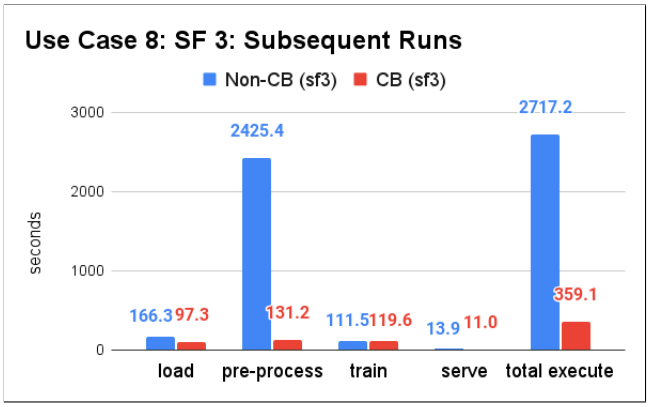
The final step in the CacheBack workflow is writing the marked variables to cache; i.e., to a unique table inside the PostgreSQL database. Automating this process is one of the key contributions of this project. However, it is important to analyze the overhead of caching itself and critically evaluate whether caching large data objects is a reasonable action. In this section, we highlight two main use cases - 8 and 10, which demonstrate the dichotomy of caching.

Use case 8 proves that caching is extremely beneficial when the pre-processing steps are computationally intensive and consume more time as data size grows. This use case is dominated by pre-processing steps which involve cleaning data, feature extraction, aggregations and a number of transformations to obtain columns relevant to training. CacheBack was used to cache the pre-processed data, thus pre-processed clean data could be directly read from the cache. As shown in Figure 4 subsequent runs show drastic improvements in execution times, since the new pipeline completely eliminated the data loading and pre-processing steps.

By contrast, Use Case 10 illustrates a typical scenario when caching performs worse. This use case had simple pre-processing steps, which involves joining one big table with about ~2000x smaller table. As an example, for scale factor 1 the tables to be joined were 908 MB and 400 KB in size respectively. Joining these tables in memory is faster and



(a) Scale Factor 1



(b) Scale Factor 3

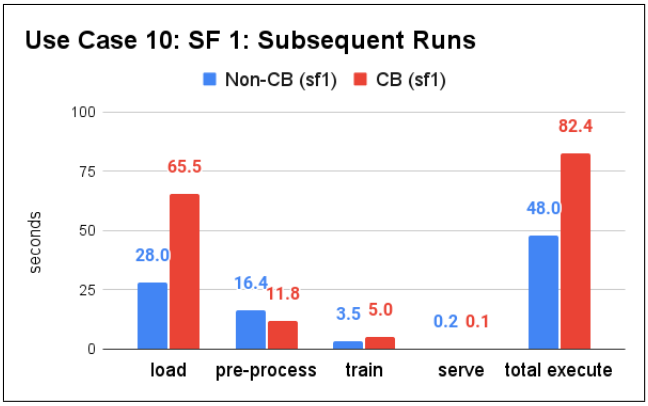
Figure 4. Use Case 8: Subsequent Runtimes of Both Non-CacheBack and CacheBack, divided into different categories, with Scale Factors = 1 and 3, respectively

does not involve computationally intensive operations. However, when the resultant DataFrame of this join is cached, the output table on the database was 1257 MB.

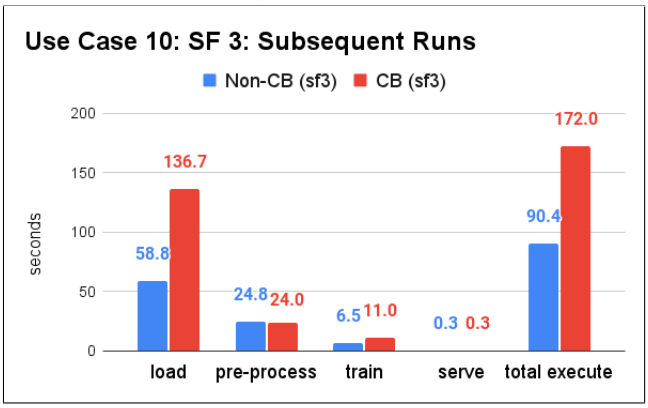
Figure 5 shows that reading cached data from the database adds notable delay to the execution time of the subsequent runs. It is worth noticing that the tradeoff between reading pre-processed data directly from the cache and carrying out pre-processing steps on-site, depends largely upon the nature of the workload. Hence, we conclude that in such scenarios when the pre-processing steps are simple and produce a sparse or wide DataFrame; caching is not a wise option as caching these variables would induce a significant overhead on subsequent runs.

4.7 PostgreSQL as an Execution environment

We also evaluated the overall run time performance of these workloads in two different environments. Firstly, we recorded the execution time of the use cases using a native Python interpreter. Then we executed the same scripts on PostgreSQL using the `PL/Python` extensions. We recorded the overall



(a) Scale Factor 1



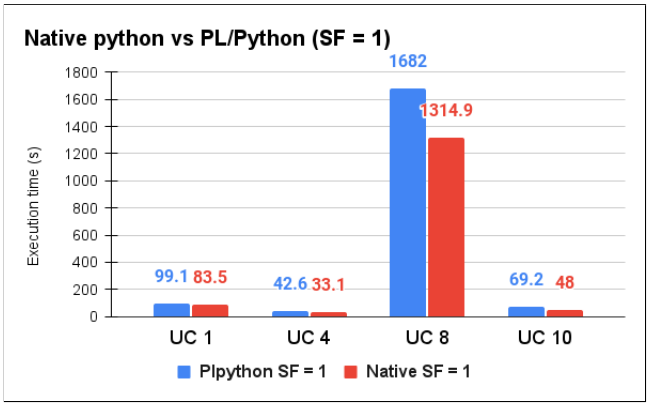
(b) Scale Factor 3

Figure 5. Use Case 10: Subsequent Runtimes of Both Non-CacheBack and CacheBack, divided into different categories, with Scale Factors = 1 and 3, respectively

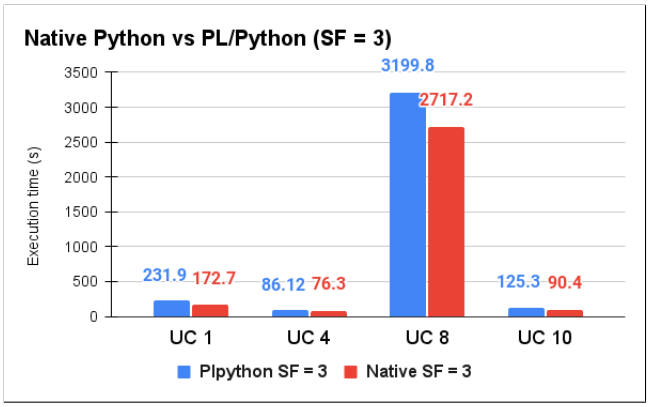
runtimes for each mentioned use cases and compared them against native performance.

The Python interpreter hosted with PostgreSQL can support any operation that can be executed as a Python function. This means that the PostgreSQL Python interpreter does not experience any limitations on libraries used, computations performed or resources consumed. This is an important observation as this would allow us to run compute-intensive scripts on the DBMS server. Moreover, we can observe in Figure 6, the Python interpreter on the PostgreSQL server shows comparable performance as compared to native Python performance with some minor performance degradation. The performance drop is experienced due to the overhead of setting up the Python interpreter and allocating necessary resources by the DBMS server. However, overall runtimes are similar and the observations are consistent.

By analyzing the results of these experiments, we conclude that PostgreSQL can be used as an execution environment and can be used to utilize the high-end server hardware for executing complex workloads, with minimal performance degradation. Hence, the proposition of moving computations close to data source has enormous potential and feasibility for large-scale applications.



(a) Scale Factor 1



(b) Scale Factor 3

Figure 6. Overall End-to-End runtime for native Python interpreter and PostgreSQL PL/Python interpreter, for each use case, with Scale Factors = 1 and 3 respectively

5 Scope of Improvement and Future work

We presented the initial prototype of CacheBack in this paper. Although CacheBack serves as an easy-to-adopt caching abstraction for developers, there are a number of limitations and pitfalls of the current design. We identify the following major areas of improvement in the future -

1. **Automated Caching:** As we observed in evaluations, developers must be aware of the variables they are caching and think about the consequences of caching them as well. Thus, the selection of variables to be cached is a manual process and is subject to human error. One major improvement to this project would be the ability to decide or suggest frequently used dataframes and adding them to cache. This would involve cost estimation and heuristics to determine the preferreed variables to cache with an appropriate level of accuracy.

2. **Dependency Versioning:** Dependency management has been handled to some extent in the initial prototype by using AST and tracking imports. However, versioning is yet another challenge that would pose problems towards executing independent pipelines. One pipeline may require completely different version of a specific library and might need to install the right version on PostgreSQL interpreter. Approaches such as Containerized UDFs [15] and Virtual environments [8] can be integrated for addressing this problem.

6 Related Work

Improving the drawbacks of Jupyter notebooks has been a popular research area. The problem of losing states is an existing problem that has not been solved completely yet. Most of the relevant work in this area has been done for creating reproducible notebooks [13] since reproducibility implies the ability to recover states without losing any information [17]. The root problem concluded by the former research area is the lack of state recovery in notebooks, which leads to the re-execution of all cells in the right sequence. We selected some of the existing solutions for solving this problem -

1. **SerDe Libraries :** Serialization/Deserialization libraries such as Dill [5] and Pickle [4] are commonly used for restoring states. These libraries serialize all the in-memory objects to a byte stream, which can later be deserialized to obtain the original object. This solution, however, adds the overhead of creating the serialized objects and writing them to disk, thus increasing storage consumption. We observed that the size of the exported file increases rapidly when the notebook deals with DataFrames and similar data objects. Hence, this approach is not applicable when dealing with gigabytes of data ingested from a database.
2. **Cloud Notebook Providers:** Augmenting the capabilities of Jupyter notebooks, commercial collaborative notebook environments such as Google Colab [2] are becoming ubiquitous. These products provide cloud integration to a kernel along with a web-browser based interactive frontend. Fundamentally Google Colab is built on top of Jupyter Notebooks, hence the problem of saving states still exists. However, Google Colab provides a tangible solution to save progress. Any Colab notebook can be authorized with a valid google account and exported to Google Drive. This allows users to work on a common working directory and have direct access to files. To preserve states, one can simply use serialization libraries such as Pickle or Dill, and dump the progress which can be loaded again in the subsequent runs, thereby reducing the overhead of running all the cells again.
3. **iPython-Kerner-Changes:** Wannipurage et al. have approached this problem by considering the local Jupyter kernel into account [18]. This approach aims to export all dependencies to a session file which can later be used to restore the absolute state of notebooks. However, we argue that this approach works best for smaller codebases which are relatively simpler as compared to the modern real-world data science workloads.

CacheBack approaches this problem with stronger, yet more rational assumptions. Data is often loaded from a relational database. We have considered PostgreSQL [7] for this project since it is a widely used database, and is adopted as the commercial database in industries. This assumption allows us to utilize DBMS as a global cache and delegate the caching processes to the DBMS machine. Unlike SerDe libraries, users don't have to manage the dumped session data or import it again explicitly. The cached data resides on the DBMS server and users retrieve an updated pipeline both in `.ipynb` and `.py` formats. CacheBack requires minimal user efforts for caching necessary data to the DBMS—all they need to do is add variables to the cache and deploy the notebook using the provided CLI.

7 Conclusion

In this paper, we presented CacheBack, a solution for caching necessary variables to reduce redundant computations over subsequent executions. This approach takes the states of the intended variables produced at the end of a pipeline and caches them to eliminate re-execution of all cells during successive runs. CacheBack is designed to work in setups where data already resides in a relational database. This way CacheBack treats DBMS as a global cache and leverages the server for pipeline execution and caching operations.

The problem of state recovery is a well-studied research area, however, existing systems do not consider a relational database as a caching solution. We argue that our approach has enormous potential and viability in the real world, where relational databases are ubiquitous in industries. We, thus, prove CacheBack's feasibility as a fair solution by showing the capabilities of PostgreSQL as a competitive execution environment and promising evaluation results on standard data science workloads.

References

[1] Amazon. 2023. *Amazon SageMaker.* https://aws.amazon.com/sagemaker/
[2] Google Colaboratory. 2022. *Google Colaboratory.* https://research.google.com/colaboratory/
[3] Transaction Processing Performance Council. 2021. *TPCx-AI.* https://www.tpc.org/tpcx-ai/default5.asp
[4] Python Software Foundation. 2023. *Pickle: Python object serialization Library.* https://docs.python.org/3/library/pickle.html
[5] The Uncertainty Quantification Foundation. 2023. *Dill: Python Object Serialization Library.* https://dill.readthedocs.io/en/latest/
[6] Brian E. Granger and Fernando Pérez. 2021. Jupyter: Thinking and Storytelling With Code and Data. *Computing in Science & Engineering*

23, 2 (2021), 7–14. https://doi.org/10.1109/MCSE.2021.3059263

[7] The PostgreSQL Global Development Group. 2023. *PostgreSQL: The World's Most Advanced Open Source Relational Database.* https://www.postgresql.org/

[8] William Eugene Hart. 2010. Installing python software packages : the good, the bad and the ugly. (2010). https://www.osti.gov/biblio/1030380

[9] Docker Inc. 2023. *Docker: Accelerated, Containerized Application Development.* https://www.docker.com/

[10] IPython interactive computing. 2020. *IPython Kernel.* https://ipython.org/

[11] Jeremiah W. Johnson. 2020. Benefits and Pitfalls of Jupyter Notebooks in the Classroom. In *Proceedings of the 21st Annual Conference on Information Technology Education* (Virtual Event, USA) *(SIGITE '20)*. Association for Computing Machinery, New York, NY, USA, 32–37. https://doi.org/10.1145/3368308.3415397

[12] The pandas development team. 2023. *pandas - Python Data Analysis Library.* https://pandas.pydata.org/

[13] João Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. 2019. A Large-Scale Study About Quality and Reproducibility of Jupyter Notebooks. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 507–517. https://doi.org/10.1109/MSR.2019.00077

[14] Mark Raasveldt and Hannes Mühleisen. 2017. Don't Hold My Data Hostage: A Case for Client Protocol Redesign. *Proc. VLDB Endow.* 10, 10 (jun 2017), 1022–1033. https://doi.org/10.14778/3115404.3115408

[15] Karla Saur, Tara Mirmira, Konstantinos Karanasos, and Jesús Camacho-Rodríguez. 2022. Containerized Execution of UDFs: An Experimental Evaluation. *Proc. VLDB Endow.* 15, 11 (sep 2022), 3158–3171. https://doi.org/10.14778/3551793.3551860

[16] Jupyter Team. 2015. *Jupyter notebook documentation.* https://docs.jupyter.org/en/latest/

[17] Jiawei Wang, Tzu-yang Kuo, Li Li, and Andreas Zeller. 2020. Restoring Reproducibility of Jupyter Notebooks. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 288–289. https://doi.org/10.1145/3377812.3390803

[18] Dimuthu Wannipurage, Suresh Marru, and Marlon Pierce. 2022. A Framework to Capture and Reproduce the Absolute State of Jupyter Notebooks. In *Practice and Experience in Advanced Research Computing* (Boston, MA, USA) *(PEARC '22)*. Association for Computing Machinery, New York, NY, USA, Article 6, 8 pages. https://doi.org/10.1145/3491418.3530296