Technische Universität Ilmenau

Department of Computer Science and Automation

Research Project Report

Adaptive Buffermanager for Stream Processing Applications

Academic Supervisor:

Prof. Dr.-Ing. habil. Kai-Uwe Sattler

In–House Supervisor: *Timo Räth*

Submitted by:

Hafiz Muhammad Umar Naeem

Program: *RCSE*

Matriculation Number: *63701*

Submission Date: 08.06.2023

Contents

1.0 Introduction

StreamVizzard is an interactive stream processing editor to simplify the pipeline development process. It comes with an assortment of features to support the developer, such as a visual debugger that allows reviewing the pipeline execution.

Data storage has always been an issue when it comes to managing big data. To resolve this issue, StreamVizzard allows developers to monitor and understand the pipeline's execution by traversing its state step-by-step in any direction. The key concept is the dynamic pipeline history, which stores all required information needed to roll back the pipeline state to any point in the past. The user can manually traverse through the data history in any direction, and each step can be undone or redone to transition the overall state of the pipeline. The architecture of the solution can also be adapted for a more distributed execution.

StreamVizzard introduces a buffer manager, which distributes the history data on the main memory and file system. Steps that overflow the available storage will be cycled between the both memory and file system. It implements a queue-based approach to automatically replace the oldest elements of history.

Buffer manager tries to efficiently manage the history of data objects in a pipeline debugger. It manages the memory usage of the data objects in the pipeline, keeps track of their history, and unloads them from memory if the present limit is reached.

1.1 Objectives & Requirements

Our goal is to develop an adaptive buffer manager for StreamVizzard. Based on this, we have established certain key objectives that we need to reach:

1. **The manager should be able to store the data as efficiently as possible**

   While managing the different objects, the manager should be able to store the data with as few function/method calls as possible.

2. **The manager should be able to easily cycle through different storage systems within the hard disk and memory)**

   The manager should have the convenience to move the data to the external storage when the memory limit is reached without difficulty accessing the external storage. For example, using a database, the renewal of the connection may not be required in every database call, and it should be able to follow the data between the different storage easily.

3. **The storage system should be able to store any kind of Python object**

   As there is a broad range of input types, ideally, the external storage should be able to store any Python object.

4. **The application should work efficiently locally**

   Data management approaches should be as similar to closed systems as possible for the safety and stability of the processed data. Using a cloud-based approach to store the data would be unstable since performance would also depend on the internet speed and chosen transfer methods. Additionally, the information would be stored outside of the system, which is a potential security issue.

1.2 Literature Review

This last decade has seen big data grow to become the new holy grail of modern technological developments and solutions. Various sources define big data as consistent and relative to four basic attributes; namely volume (amount of the data acquired), variety (array of sources for acquiring data), velocity (the rate at which data is recorded and examined), and reliability (the accuracy of the acquired data (Wiener, Saunders, & Marabelli, 2020). As such, we can say that a fundamental component of how big data can be used effectively and efficiently is dependent on how quickly and easily we can reach, examine and utilize the necessary information from a big data pool when the need arises. Furthermore, when it comes to big data, security is a vital component, considering how it compromises confidential details such as sensitive sensor data that only certain authorized people are warranted to know. Access and leakage of such data beyond these authorities can have severe consequences for privacy and security (Jagani, Jagani, & Shah, 2021). Consistent of cloud storage systems from external parties, another glaring issue is a severe lack of control and dependence on the external parties as compared to local storage. This can not only limit current management and control but also makes future changes and developments more difficult and applicable as compared to local storage (Secure Storage Services, 2023).

1.3 Research

The main challenge here was figuring out how to store the data efficiently and find a way around the usage of a structured database, which was not working since the data type changes a lot. As such, we checked and went through several database types to find the right one. In this process, it was observed that storing specific Python objects by using a structured or document-based database was not possible since a structured database like SQLite must be structured for multiple reasons, such as storing both NumPy arrays and simple Python lists. This is not achievable as there is no limit to the processed data structures and, thus, a temporary file system was considered. With the correct libraries (i.e., Python Shelves), Python objects can be stored easily.

While Python Shelve was also used to store the objects, it was highly inefficient; there were too many operations, and because of this the database easily became fragmented. To resolve the issue, we tried closing and reopening the database for every fixed number of operations, but that meant that the whole database needed to still be loaded to the memory; a very inefficient use of memory in every way imaginable. This approach also hurts performance with a higher amount of data since loading the database to the memory takes more time. Thus, we concluded that using Shelve was not the right choice for the implementation.

The ability of the manager to be able to work locally was also one of the basic objectives of this project. Therefore, using cloud-based database providers such as Amazon is not possible, .as it would introduce instability to the system.

Before going into the details of why the temporary file system was chosen, let's summarize the problems faced in the search for the right tool:

- A structured database would not work because the data structure varies a lot. It is not possible to initialize the database to fit any kind of data.
- Using an alternative such as Python Shelve is not feasible because the database gets fragmented pretty easily. Defragmentation is inefficient.
- Using a local database requires the whole database to be loaded on the memory. Which introduces performance problems with a high amount of data.
- Cloud-based database providers are not a valid choice as they introduce system instability.

There are several reasons for a temporary file system to be a better option for solving the problems. Here are some of them briefly explained:

- Storing the objects as individual files lets us store almost any Python object without limitation of the data structure. If the object is pickable (as most objects are), then we can store it.
- There is no indexing and database updates are not made in a single file, the updates are made by overwriting the stored files. So there is no fragmentation.
- The database is not stored on the memory completely, the data is read from the hard disk when it's required.
- Completely local

Since the most important aspect is the storing history of the objects, the object data needs to be stored safely. With security as the top priority, a choice was made in solution design by setting up a temporary file system. With this approach, it has become possible to efficiently store a broad range of Python objects and enable the manager to successfully manage the memory.

2.0 Design and implementation

2.1 Concept

The idea is to improve efficiency by providing greater ease of access and bypassing unnecessary steps and actions. An example is by creating tables that allow users to look up the relevant files more easily and avoid reading all of the temporary folders each time they are moved. Another concept to improve efficiency is by introducing batching. Since the app aims to introduce greater efficiency, batching data to speed up the processing of the data into large chunks is perhaps one of the most viable and reliable support that can be added to the application. There are several advantages of batch files, such as distribution of large files, it also enables parallel processing, downloading of segments etc. however, it additionally requires management and coordination in-order to make sure the integrity as well as proper organization of batch files. After all this The main reason not to select batch file system is the same with databases. We would need to load the subsets of the dataset to memory. Depending on the batch size, this could be reduced. But if we'd need to use smaller batch sizes to not load much more data to memory, why should bother to implement it in the first place? Also "batching" did not seem like a solid performance booster.

The instance of the Manager class manages the history of data objects in a pipeline debugger. It also manages the memory usage of the data objects in the pipeline by keeping track of their history and unloading them from memory if memory limits are reached.

Adjusting the memories and cycling between each other was another difficult task. Calculating the buffer sizes was difficult so it has been thought that custom objects should handle their memory size calculation by themselves. If the object calculates its memory size, then it would be easily used to manage the buffers. There were two storage limits about whom one should be cautious:

- Memory size: When the memory size is exceeded, try to save the objects in the memory to the hard disk. Then if all the objects in the memory are stored, simply deleted the oldest history entry.
- Storage size: When the storage size is exceeded, the manager tries to load the data to the memory and delete the stored file. While this decreases the storage size, it introduces an extra burden on the memory. So, double-check if the memory size is exceeded or not. Then the oldest history entry should be removed again. This approach ensures the limits are not exceeded and are successfully managed by the history manager.

2.2 Implementation

The main data structure is DebugTuple. This class is used to store and manipulate tuples, as well as some extra data associated with them. The most important methods of this class are:

- registerMemoryChangeCallback: This method registers a callback function that is called whenever the memory size of the instance of DebugTuple changes.
- _updateMemorySize: This is the private method that is called whenever the size of the instance of DebugTuple changes. It calculates new memory size using `aside` from the `pympler` module.

The class named HistoryBufferManager provides various methods to manage the history of debugging steps. It is intended to work with a PipelineDebugger instance and it stores all the necessary information about each step executed in the pipeline.

The main purpose of this class is to keep the execution history of the pipeline steps with their input data and the results. It is possible to set memory limits to the history; in StreamVizzard the data was stored on the hard disk if the memory limit is reached. This way, it is possible to store the data and read it again when it was needed. But, there's also a limit defined for the storage size.

If the memory limit is reached, the manager tries to move the data into the hard disk by using the `_storeDT` method.

The main data structures used in this part are `_storedDtQueue` (list), `_memoryDtQueue` (list), and `_historyDtLookup` (dictionary). The used memory and storage size are also updated in the `_storeDT`, `_removeFromStorage`, and `_registerStepDT` methods. The `_removeFromStorage`, method simply loads the data again and deletes the storage file. And `_registerStepDT` adds the given object to the history. Every time an object is added, the `_adjustBuffer` is called. This method checks whether the pre-defined memory limits are reached. If an adjustment is required, then it uses the `_removeOldestStepFromHistory`, `_removeFromStorage`, and `_storeDT` methods to make the adjustments. Every time that an object's memory is updated, `_onDebugTupleMemoryChange` is called. It updates the tracked memory sizes and updates the data stored in the filesystem if the object is stored.

When a DebugTuple is registered to the history, it is added into its uuid to the `_memoryDtQueue`. And when a DebugTuple object is moved to the storage, its uuid is stored in the `_storedDtQueue`. By cycling `UUIDs` between these two queues, we ensure data processing without calling extra file I/O-related methods.

2.2.1 Memory Management: Storing and Loading the Data

At first, it was thought that storing the data in a database would be all right. But as the test cases provided many different data types, it was observed that not using a database technology was more convenient. Therefore, we proceeded with storing the data on a hard disk, as Python Shelve was already been tried and it was pretty inefficient. Because Shelve requires caching the whole data on the memory to make subsequent modifications to the saved data, it was dropped.

The next option was the use of Pickle first, but it's well known for inconsistencies with some Python objects (you can't serialize complex objects such as unique-designed class objects, etc.). As such, the best option left was Dill, since it's able to serialize much more complex objects than the Pickle.

This choice introduced various problems relating to the file I/O; Dill was slower than the Pickle. But the advantages were greater than the disadvantages since Dill was built on the Pickle and just extended the serialization support. The difference in speed was barely even considerable (its test script with cProfile was tried for running).

Therefore, `dill`, with the extended version of `pickle` in Python was used. The whole object to the storage file was dumped, as it contains necessary information too, along with that it was changed to storing only the tuple data and DebugTuple's data.

Here's in figure 2 you can see the comparison of dill and pickle, the scripts ran for different memory limits, then averaged the total execution time in the graph. Then averaged the execution time of each run with different limits. In order to arrive at a robust conclusion.

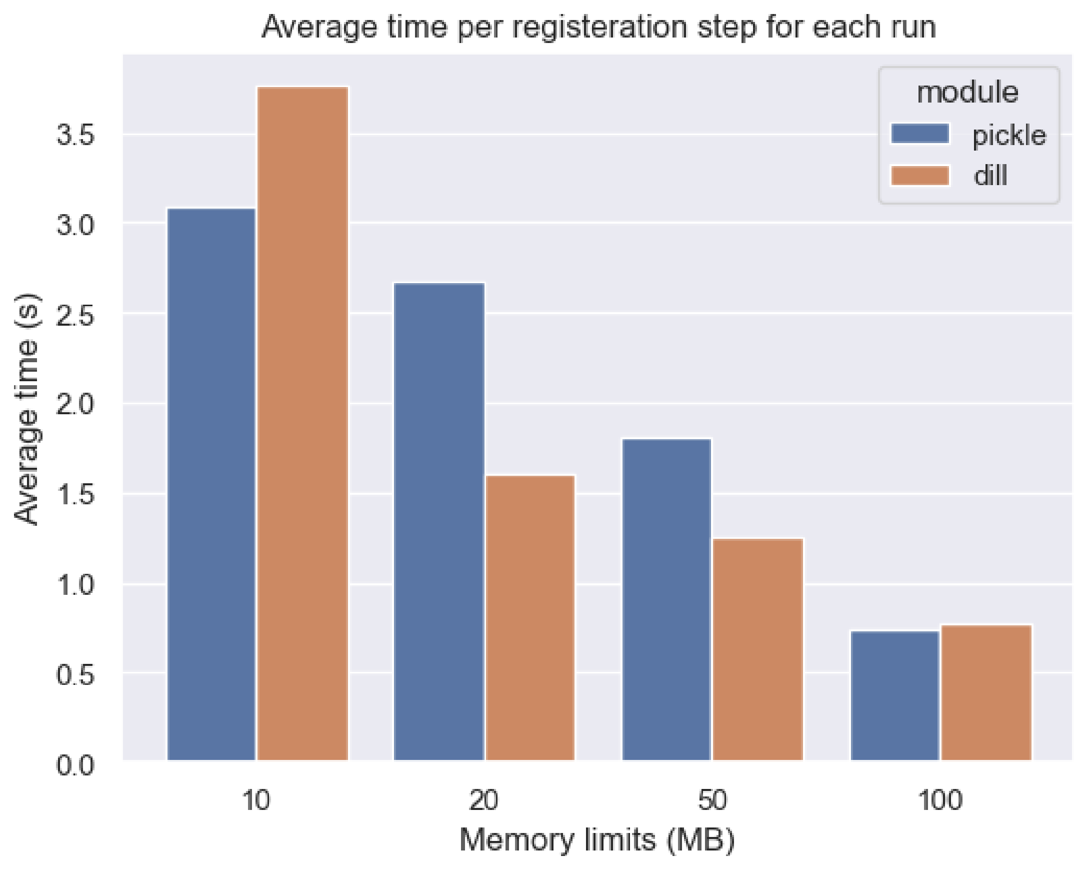
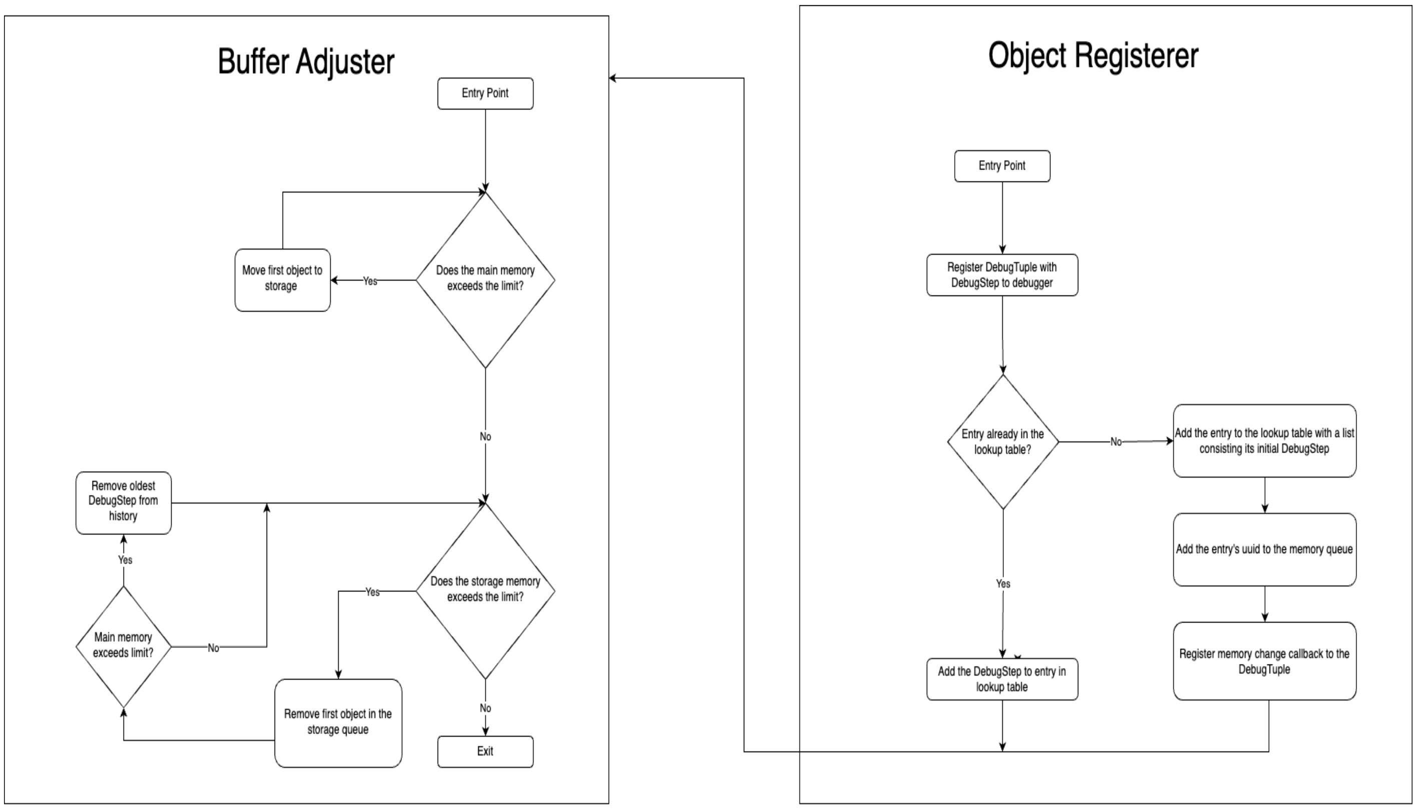


Figure 2: Dill vs Pickle

Hence file storage was implemented by using a temporary directory, so the data is deleted when the program exits. There's also the `_removeFromStorage` method which removes the DebugTuple object's saved data from the disk.

The flow chart below describes the process; once the data has entered the program and gone through the debugger, the debug step entry is added to the lookup table and the memory queue heading towards the buffer adjuster. It then checks whether the main memory has exceeded the limit, and if so it moves the very first object to storage. It then checks the storage memory and proceeds to exit unless the memory is exceeded. If so, It moves the very first object in the storage queue. It checks the main memory and removes the oldest DebugStep from the memory to make room if it exceeds the limit. Whether it does or doesn't exceed the memory, it will eventually loop to check if the storage memory exceeds the limit until the answer is 'no 'and proceed to exit.



Buffer Adjuster
Entry Point
Does the main memory exceeds the limit?
Yes
Move first object to storage
No
Remove oldest DebugStep from history
Yes
Main memory exceeds limit?
No
Does the storage memory exceeds the limit?
Yes
Remove first object in the storage queue
No
Exit
Object Registerer
Entry Point
Register DebugTuple with DebugStep to debugger
Entry already in the lookup table?
No
Add the entry to the lookup table with a list consisting its initial DebugStep
Add the entry's uuid to the memory queue
Register memory change callback to the DebugTuple
Yes
Add the DebugStep to entry in lookup table

Figure 1: Flowchart of Buffer Manager

2.2.2 Memory Management: _adjustBuffer Method

If the memory limits are reached, the oldest DebugTuple objects in memory are moved to the storage, and if the storage limit is reached, the oldest objects are removed from the storage. A bottleneck was expected if both limits are reached, that's when one removes the oldest steps from the execution history. It can be seen from Figure 1 that this ensures that there's valid cycling between different storages.

At the start, everyone was checking the files by simply finding all files in the temporary directory, it was later on changed to using a queue called `_storedDtQueue`. It is defined at the object initialization and the files that contain the DebugTuple data are stored in it. Therefore, the first element will be the oldest DebugTuple object. Thus, one can simply load the oldest object to the memory again. If loading the data to the memory exceeds the memory limit, he removes the oldest step from the history. This ensures limits are not exceeded.

In summary, the class manages memory usage by moving old objects to storage when the memory limit is reached. If the storage memory limit is also reached, the oldest objects are removed from storage.

3.0 Discussion & Conclusion

3.1 Motivation

The major challenge in this project was to make it possible to store the data most efficiently and conveniently. Several systems have been tried to achieve the purpose. In the process of finding a suitable system for data storage, a lot of learning has happened automatically. The things which were initially only learned theoretically, got a first-hand experience on them in this whole process. My motivation and spirits were kept high to excel in the project. A famous saying is “You are your own biggest critique”. This was kept in mind so that efficiency could not be compromised. All the issues are approached to increase learning opportunities.

3.2 Future Recommendations

While this project can help curb and manage a lot of issues regarding quick retrieval, security, and accessibility of relevant data using on-site methods, there can still be issues if the data exceeds the maximum capacity that the local storage can hold. In the case of the expected increase in future data influx and retention needs, expanding on-site storage consistently might be necessary. For management needs, it’s also possible to introduce batching to stored data for convenience in accessing and calling relevant data when required.

3.3 Conclusion

Store data and objects are one of the main achievements for any developer. Everyone faces lots of challenges while designing and searching for a tool that is more accessible for them. Our data was getting stored in the main library which has very limited space. So, the aim was to get a system or database on which we can store the data for the time being the respective program is in process and once the program is completed, it will delete. Hence file system was chosen for this. As it was able to manage everything in the best way.

4.0 Reference

1. Räth, T., & Sattler, K. U. (2022, June). StreamVizzard: an interactive and explorative stream processing editor. In *Proceedings of the 16th ACM International Conference on Distributed and Event-Based Systems* (pp. 186-189).
2. Jagani, N., Jagani, P., Shah, S., & Somaiya, K. J. (2021). Big Data in Cloud Computing: A Literature Review. *International Journal of Engineering Applied Sciences and Technology*, *5*.
3. Secure Storage Services. (2023). *Pros and Cons of Cloud Storage*. Retrieved from Secure Storage Services: https://www.securestorageservices.co.uk/article/11/pros-and-cons-of-cloud-storage
4. Wiener, M., Saunders, C., & Marabelli, M. (2020). Big-data business models: A critical literature review and multiperspective research framework. *Journal of Information Technology*, *35*(1), 66-91