TECHNISCHE UNIVERSITÄT
ILMENAU

RESEARCH PROJECT

SPARKCAD+ : A RICH VISUALIZATION OF SPARK WORKFLOWS AND CLUSTER SIZE RECOMMENDATION

AUTHOR: MUHAMMAD ASLAM ABDUL HUQ

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

SUPERVISOR: M. Sc. Hani Al-Sayeh

Matrikel: 64451

Course: Research in Computer and Systems Engineering (M.Sc.)

CONTENTS

1. ABSTRACT

The efficient management of data in Apache Spark is crucial for optimizing performance and minimizing computational overhead. One key aspect of this management is the caching of intermediate results in memory to avoid recomputing them each time they are needed. However, the limited capacity of memory necessitates careful consideration by developers regarding which datasets to cache and which ones to discard. To make informed decisions and avoid memory overflow situations, developers require knowledge about the size and computational time of each dataset. In this context, a developer decision support tool called SparkCAD+ (SparkCAD plus) proves to be invaluable. It extends the work presented in the paper [1] titled "SparkCAD: Caching Anomalies Detector for Spark Applications" and provides developers with detailed information about dataset size and computational time on the logical plan of Spark application logs. By leveraging SparkCAD+, developers gain insights into the characteristics of their datasets, enabling them to make informed decisions about caching. Knowing the size of a Resilient Distributed Dataset (RDD) and the time required to compute each RDD dataset empowers developers to prioritize caching the most critical and frequently accessed datasets while avoiding unnecessary caching of less significant ones. This information helps prevent memory overflow situations and ensures optimal utilization of available resources. The impact of caching suitable intermediate results cannot be overstated. By avoiding redundant computations, Spark applications can achieve significant performance improvements. Caching allows subsequent operations to directly access the cached data from memory, reducing the overall execution time. With SparkCAD+, developers gain visibility into the time and size aspects of their datasets, facilitating the identification of potential bottlenecks and areas for optimization.

2. INTRODUCTION

Iterative machine learning applications have a huge impact on recent data-intensive applications such as Spark which can help in the case of iterative workloads in improving the performance efficiency. This performance efficiency can be done by caching some particular datasets in the system's memory to avoid recomputing those datasets again and again.

The paper [1] has explained about causes of wrong caching decision which results in the caching anomalies in the application flow and also detailed how to solve those problems. The two types of caching anomalies mentioned in the paper [1] are non-used cached RDD and recomputed RDD. Non-used cached RDD occurs when the application developer caches an RDD that is not reused and that occupies space from already limited memory resources [1]. Recomputed RDD occurs when the application developer does not cache an RDD that is reused multiple times, leading to significant recomputation overhead [1]. These two caching anomalies can be tackled by caching the crucial datasets in the system's memory.

Spark's History Server [4] reads execution logs and provides a web user interface (UI) that application developers can use to get more insights into the execution of their applications [1]. But the directed acyclic graph (DAG) presented by the user interface won't give a full view of RDDs but not the whole overview of the application's data flow. The whole overview here refers to the logical plan of RDDs and transformations across all jobs and stages in a single view [1]. This is one of the reasons the developers say that it is not a completely reliable caching decision support tool when their application becomes more complex. The work and the source code mentioned in the paper [1] will provide a single view of all the stages and transformations between jobs.

But still, there is a drawback with caching the datasets in memory. A user cache the datasets based on the number of times an RDD is recomputed, but they won't know what will be the size and execution time of an RDD. The reason behind the drawback is completely based on the system's memory. If the memory is less and the size of the RDD datasets that are to be cached exceeds then there occurs the out-of-memory error.

To tackle this issue, we present SparkCAD+ (SparkCAD plus), a support tool for developers to efficiently usage of memory and a better caching decision policy. we have proposed ways to calculate the size of an RDD as well as the amount of time it takes for the transformation from one RDD to another. This will greatly help the application developers to decide which RDD to cache depending upon the available memory space in the system.

Firstly, it visualizes the logical plan of the entire application in a single view with various display options [1]. Secondly, it helps application developers to see the size and execution time of an RDD on which job and stage. Thirdly, as an interactive what-if analysis tool, it allows application developers to make new caching decisions and see the impact of their caching decisions without carrying out additional experiments [1]. Fourthly, it provides the developer with a sequence of recommended cache/unpersists commands, which we term Recommended Schedule, to help the developer to know when to cache or unpersist an RDD. In addition, it gives an overview of the memory footprint during the application run [1]. Lastly, It does localization of the information about the datasets when the application is processed for the first time to avoid the processing overhead time for future processes.

3. JUGGLER

In this section, we explore JUGGLER, an autonomous framework that utilizes training-based techniques to optimize the caching of datasets and recommend an optimal cluster configuration. JUGGLER's recommendations to application users are driven by performance prediction and the consideration of performance-cost trade-offs.

JUGGLER employs a dataset combination approach to determine which datasets should be cached. It generates one or more schedules, which are ordered lists specifying the datasets to be cached. These schedules are designed to maximize performance and efficiency based on JUGGLER's analysis.

To make these recommendations, JUGGLER goes through four main stages of processing: Hotspot Detection, Parameter Calibration, Memory Calibration, and Building Execution Time Model [4]. The first three stages involve offline training by

JUGGLER, where it analyses historical data and identifies data hotspots, fine-tunes parameters, and optimizes memory usage [4]. These stages serve as a preparatory phase to ensure accurate and effective caching recommendations.

The following explains those stages in detail,

Hotspot Detection: In this stage, JUGGLER caches the datasets and it will again look in the cached dataset to identify whether any of those datasets has any potential to have an improved performance. The goal of Hotspot Detection will be achieved by JUGGLER by analyzing the application on SparkI to gather the size and timestamp metrics for each dataset.

Parameter Calibration: JUGGLER takes the application parameter from the users and will build models to predict the sizes of those potential datasets [4].

Memory Calibration: JUGGLER works out on calculating the calibration factor in order to make the predictions for the required number of machines to cache the datasets [4].

Building execution time model: JUGGLER constructs an execution time model for predicting the execution time [4].

The Building Execution Time Model stage involves developing a predictive model for estimating the execution time of different dataset caching configurations. This model takes into account various factors such as dataset characteristics, cluster configuration, and caching strategies.

Ultimately, JUGGLER combines these training-based processes to provide intelligent recommendations to end users. These recommendations consider the trade-off between performance and cost, ensuring that caching decisions align with the desired performance goals while optimizing resource utilization [4]. By leveraging offline training and predictive modeling, JUGGLER empowers users to make informed decisions about dataset caching and cluster configuration, enhancing the overall performance and efficiency of their applications.

The goal of SPARKCAD+ has been inspired by JUGGLER to find the size and execution time of each dataset to have appeared on the logical plan of the Spark application. JUGGLER uses the concept called instrumentation which helps in finding the size and timestamp of a particular dataset. This concept of instrumentation from JUGGLER has been inspired in SparkCAD+ in order to collect the size and execution time information about the datasets. The following section will provide a more detailed explanation of the concept of Instrumentation.

4. INSTUMENTATION

In the context of Spark, a task is responsible for processing a dedicated data partition by applying a sequence of narrow transformations. Our primary objective is to obtain accurate and detailed information about the processing time and resulting data partition size for each transformation within all the tasks. However, the default behavior of Spark does not provide this runtime data out of the box. To overcome this limitation, some modifications to the source code of Spark will result in a new version of logs known as Spark Instrumentation (Spark*i*).

Spark*i* introduces an automatic and intelligent mechanism to address the lack of runtime data. It achieves this by injecting a special-purpose transformation called "*mapPartitionsWithIndex*" between each consecutive pair of transformations within a task. This injected transformation serves a dual purpose: profiling timestamps and partition sizes. Moreover, it generates an instrumentation dataset that faithfully replicates the dataset produced by the preceding transformation. Essentially, this means that the data generated by the original transformation is accurately replicated, while additional profiling information is collected during the process.

Within each task, the profiling transformation is responsible for recording the runtime data and storing it in *TaskContext*, a context-specific data structure provided by Spark [4]. Once the task completes its execution, the corresponding low-level runtime data is sent to a centralized profiling database [4]. This database serves as a repository for all the gathered runtime information, which includes timestamps, partition sizes, and any other relevant profiling data [4]. When the entire Spark application comes to an end, the runtime data, which comprises information about the application, individual jobs, stages, and tasks, is consolidated and copied into the profiling database [4]. This comprehensive collection of runtime data allows us to perform detailed analysis and gain insights into the precise processing time and resulting data partition sizes of each transformation within Spark tasks [4].

By leveraging the capabilities of Spark$i$, we are empowered to make informed decisions regarding performance optimization and further analysis of our Spark applications. The availability of this fine-grained runtime data enables us to identify potential bottlenecks, understand the behavior of individual transformations, and fine-tune our data processing pipelines for enhanced efficiency and performance.

Figure 1 depicts the automatic addition of profiling transformations in Spark. Each dataset, represented by "d0" in the figure, is accompanied by an automatically injected profiling transformation during its construction. This profiling transformation generates an instrumentation dataset, denoted as "di0," which is an exact replica of the original dataset. When a transformation, such as a filter operation, is applied to "d0," it produces a new dataset called "d1." Before establishing a parent-child dependency from "d1" to "d0," a crucial check is performed to determine whether an instrumentation dataset is associated with "d0." If an instrumentation dataset exists, a dependency is created from "d1" to the instrumentation dataset, "di0." On the other hand, if no instrumentation dataset is present, a dependency is created directly from "d1" to "d0".

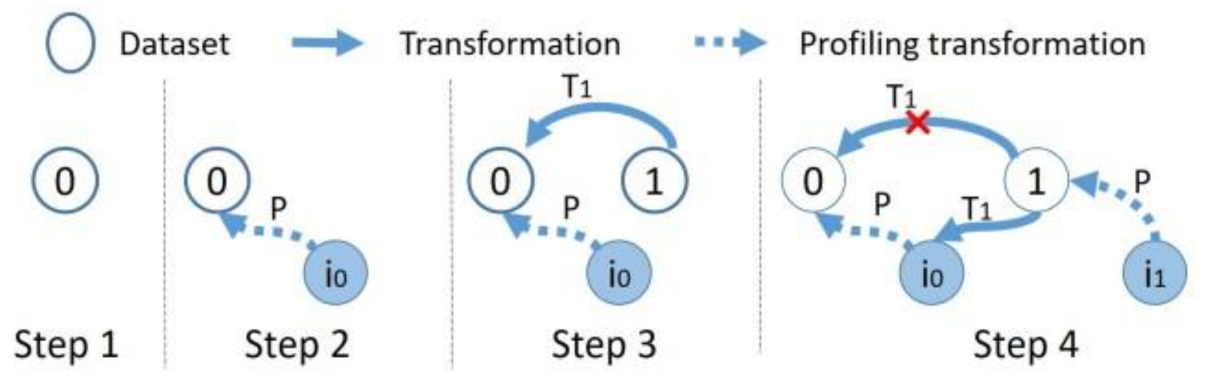

Figure 1: Spark$i$ steps in adding profiling transformations (P). T1 indicates regular transformations [4].

This process continues iteratively until the creation of the final dataset. Figure 2 below provides insight into the internal workings of the tasks within the first stage of the iterative job in the application *Logistic Regression*, starting from the instrumentation datasets. To elaborate further, the addition of profiling transformations in Spark involves seamlessly integrating them into the dataset construction process. Each dataset is accompanied by a corresponding profiling transformation that captures runtime data and generates an instrumentation dataset. This instrumentation dataset serves as a replica of the original dataset, allowing for accurate tracking and analysis of various runtime metrics.

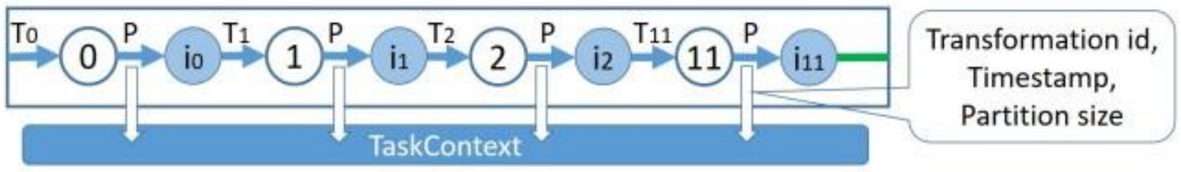

Figure 2: Task internals in the first stage of the iterative job in Logistic Regression proceeding from instrumentations. P indicates profiling transformation.

As transformations are applied to the original datasets, new datasets are derived. However, the creation of dependencies between these datasets depends on the presence of associated instrumentation datasets. If an instrumentation dataset exists, indicating that profiling has been enabled for a particular dataset, a dependency is established from the newly derived dataset to the instrumentation dataset. This enables the profiling data to propagate throughout the transformation chain. On the other hand, if no instrumentation dataset is associated with a dataset, the dependency is established directly from the newly derived dataset to its immediate predecessor. This ensures that dependencies are correctly set up, allowing for efficient and accurate lineage tracking and data flow within the Spark application.

This methodology enables comprehensive runtime profiling, facilitating the analysis of performance characteristics, data lineage, and transformations applied to each dataset within the Spark job. Figure 2 specifically showcases the internal workings of the tasks within the first stage of an iterative job in Logistic Regression. It illustrates how the instrumentation datasets and their associated profiling data are utilized and propagated through the different tasks, providing valuable insights into the execution behavior and performance of the iterative job.

Keep in mind that by integrating Spark with lightweight instrumentations, we may obtain low-level runtime data from application binaries without having the need to get into the source code [4]. With this solution, metrics can be obtained for all datasets, even those that are inaccessible from an application layer [4].

5. SparkCAD+

SparkCAD+ is a valuable decision support tool designed specifically for developers working with Spark applications. It aids in the analysis and optimization of Spark applications by providing a comprehensive visualization of the application's logical plan with the size and execution time of each dataset. This visualization is generated through a three-step process consisting of parsing, analyzing, and visualizing the application.

In the first step, known as "parse," SparkCAD+ extracts and organizes relevant information from the application's log files. This information includes details about the structure of the application, such as jobs, stages, tasks, and RDDs (Resilient Distributed Datasets). By parsing these components, SparkCAD gains a comprehensive understanding of the application's logical plan.

The second step, "analyze," involves processing the cached status of the RDDs. This step will store the information about the caching status of the RDDs and will later help the developers to have a what-if session regarding the cache status of RDDs. With this step, SparkCAD+ leverages its analysis capabilities and provides developers with insights into potential improvements or optimizations.

Finally, in the "visualize" step, SparkCAD presents the analyzed information in a visual format, enabling developers to gain a holistic view of the application's logical plan. This visualization encompasses the entire logical flow of the application, allowing developers to have more information about the RDDs including size and the transformation time visually. By presenting the information in an intuitive and accessible manner, SparkCAD+ empowers developers to make informed decisions and take necessary actions to enhance the performance and efficiency of their Spark applications.

The following figures 3 and 4 below will explain the main difference between the old version “SparkCAD” and the new version “SparkCAD+”. In figure 3, which shows the overview of SparkCAD, the flow starts from collecting execution log files from Spark applications and then the tool SparkCAD will draw the DAG graph with the details collected and processed from the logs. The important thing to note here is that these logs won’t contain information about the size and transformation time of each RDDs.

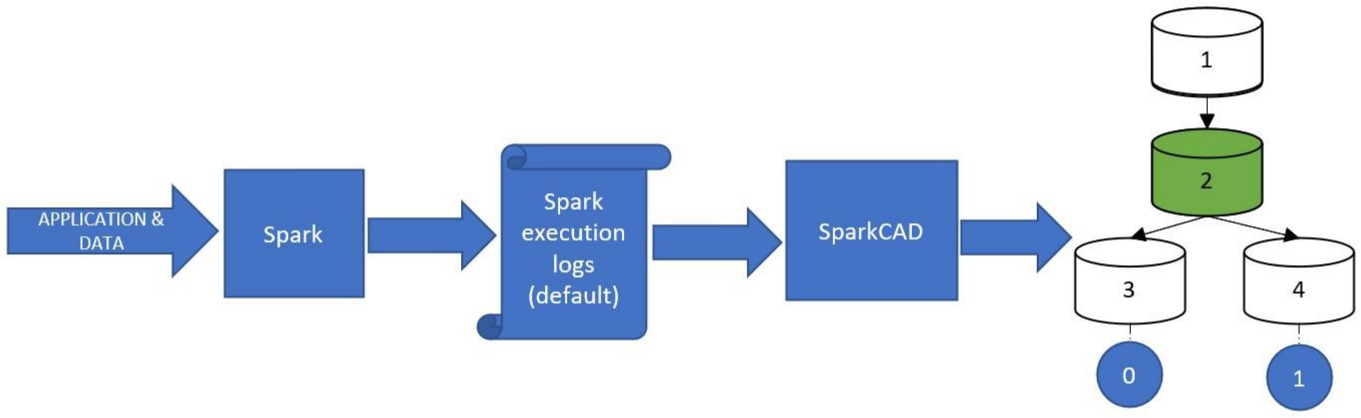

Figure 3: Overview of SparkCAD

Below Figure 4 provides an overview diagram of how SparkCAD+ transforms the raw application data into a meaningful visualization that aids in showing the size and transformation time of the datasets. Here, in this diagram, the flow starts from SparkI (Instrumented) logs instead of Spark logs. As mentioned in section 4 about instrumentation, one can generate SparkI logs by manipulating the source code of Spark. SparkI logs contain information about the size and transformation time of each RDD dataset which helps the tool SparkCAD+ in drawing a DAG graph with our desired output.

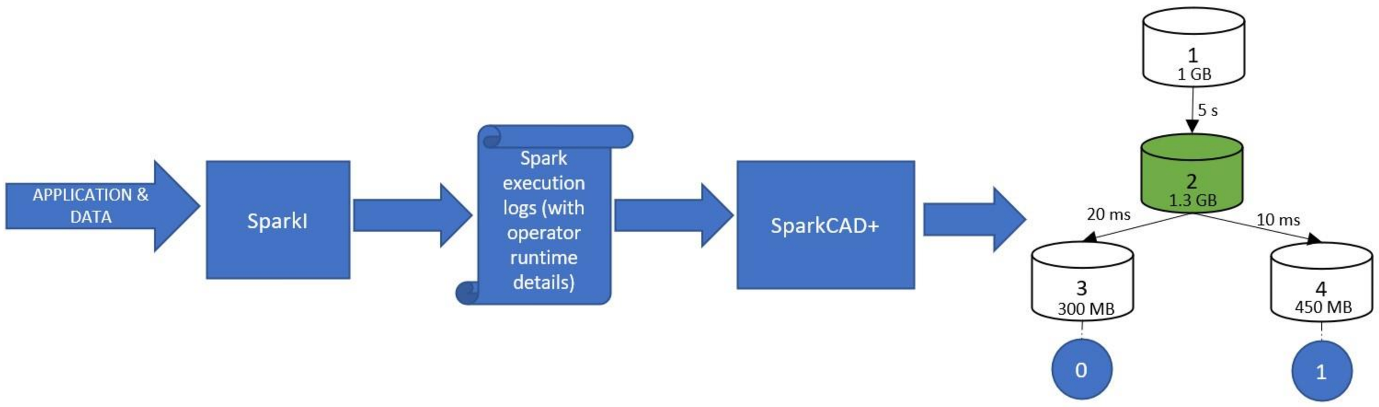

Figure 4: Overview of SparkCAD

The following implementation section will explain the functionalities parse, analyze and visualize in detail.

6. IMPLEMENTATION

6.1 PARSE

The log files generated by manipulating the source code of Spark contain a sequential collection of runtime events, stored in JSON format. SparkCAD+ leverages these log files as a valuable source of information. It specifically focuses on the log files that contain details about the datasets and their instrumentation. SparkCAD+ will use the log files with information about the datasets and their instrumentation. The utilization of log files allows SparkCAD+ to access a wealth of data related to the dataset's behavior, transformations applied, and caching status. This information is vital for performing comprehensive analysis and making informed decisions regarding caching strategies, memory allocation, and overall optimization of Spark applications. SparkCAD+ will take the following relevant events from the log files.

- SparkListenerApplicationStart – where the name of the application will be extracted
- SparkListenerJobStart – provides details about each job within the application. It includes information such as the job's name, ID, and the RDDs associated with it. Additionally, it provides insights into the parent RDDs, allowing for a better

understanding of the data lineage. The event also indicates the cache status of the RDDs, providing visibility into which RDDs are cached for efficient access.

- SparkListenerStageSubmitted – where the information about the executed stages will be extracted. Executed stages means the stages that are not actually skipped.
- SparkListenerStageCompleted – where the information about the name of the stage, its ID, information about the RDDs contained in it, and the parent details about the RDDs will be extracted.
- SparkListenerTaskEnd – where the information about the task ID, the stage ID of the task, its launch time, and finish time will be extracted. In addition to these, SparkListenerTaskEnd will also contain information about the SparkI (Instrumentation Spark) operator details which contains the information about the instrumented RDD's ID, its timestamp, and the size of the partition.

prepare_from_task_end_events_for_timestamp – This method will use the SparkListenerTaskEnd event to analyze and calculate the time taken by each RDD for their transformation. The calculated time will then be stored in a Python dictionary called FactHub.operator_timestamp, where the keys are RDD ids and values are their respective transformation times. There are three steps to prepare the FactHub.operator_timestamp dictionary. Firstly, to find the time of the first RDD of the task. This can be achieved by taking the difference between the task's launch time and the timestamp of the first operator. Secondly, to find the time of the last RDD, this can be achieved by taking the difference between the timestamp of the last RDD and the task's finish time. Thirdly, to find the time of the rest of the RDDs other than root RDD and the last RDD of a task. This can be achieved by taking the difference between an RDD's timestamp and its parent RDD's timestamp. A particular operator can appear in multiple tasks, so if the same operator ID from a different task is encountered then the dictionary value will be updated by adding the existing time for the particular RDD id.

prepare_from_job_start_events – This method will use the SparkListenerJobStart event from the application logs and will collect information about job IDs and Stage information.

prepare_leaf_from_task_end_events – This method will use the SparkListenerTaskEnd event to analyze and calculate the size of the leaf RDDs. Here the leaf RDD means the last RDD in each job

prepare_root_from_stage_completed_events – This method will use the SparkListenerStageCompleted event to analyze and calculate the size of the root RDDs.

prepare_from_task_end_events – This method will use the SparkListenerTaskEnd event to calculate the partition size from the SparkIOperatorDetails. The partition size is the size of each instrumentation dataset which represents its parent's size. These sizes will be stored in a dictionary called stage_operator_partition and will be contained in the data structure FactHub.

prepare_from_stage_submitted_events – This method will use the SparkListenerStageSubmitted event to collect the stage IDs information of the Jobs and will store them in a set named submitted_stages and will be contained in the data structure called FactHub.

All the extracted information from the above events will be stored in a data structure called FactHub for the purpose of referencing and using that in the later part of the process to achieve the desired results. The section FactHub has been explained in detail in the later section of this report.

6.2 ANALYZE

Based on the parent-child dependencies of the RDDs, SparkCAD+ generates the set of transformations between RDDs [1]. If the parent and the child RDDs are at the same stage, it considers that dependency as a narrow transformation; if not then it considers that as a wide transformation [1]. SparkCAD+ looks and traverses every stage that is submitted and it starts from the last RDD and traverses backward in a recursive manner towards its root RDDs to determine the number of usages for each RDD [1]. All the information collected in the Analyze function part will be stored as dictionaries, lists, and arrays and kept in the data structure called AnalysisHub. The data in the AnalysisHub data structure is not fixed and will be a modifiable one. For better understanding, once the program reads an application log and fills up the AnalysisHub data structure and when the user runs the program for the same application, the program will flush the AnalysisHub data structure, which means it will make the data in it as empty and fills up the data from the application log again even though if it previously processed the same application log file. This is quite opposite to the working concept of FactHub. The idea behind keeping the AnalysisHub as a modifiable one is to enable the users to have a what-if interaction session with the source code.

Let us consider below Figure 4 and Figure 5, where these two pictures depict the results of what-if interaction with the user. In figure 4, RDD 1 has a computation time of 3 s and it is being cached so that the child RDDs of RDD1 have their computation time without being added to their parent RDD. If the user gives the command to change the caching plan by not caching the RDD1 then the computation time of the child RDDs will have an impact which has been shown in Figure 5. When the user doesn't cache the RDD1 then the child RDDs of RDD1 which includes RDD2, RDD3, RDD4, and RDD5 will add the computation time with their parent RDD's computation time which in this case is RDD1.

The computation time of each RDD depends on the parent RDD's computation time. There are two scenarios to be considered while calculating the computation time of an RDD. One is when the parent RDD of a particular RDD is cached and when the Parent RDD of a particular RDD is not cached. Let us consider Figure 4 below, which explains the first scenario when a parent RDD is cached. The formula to calculate the computation time in this scenario is as follows,

Computation time of current RDD = Transformation time of current RDD

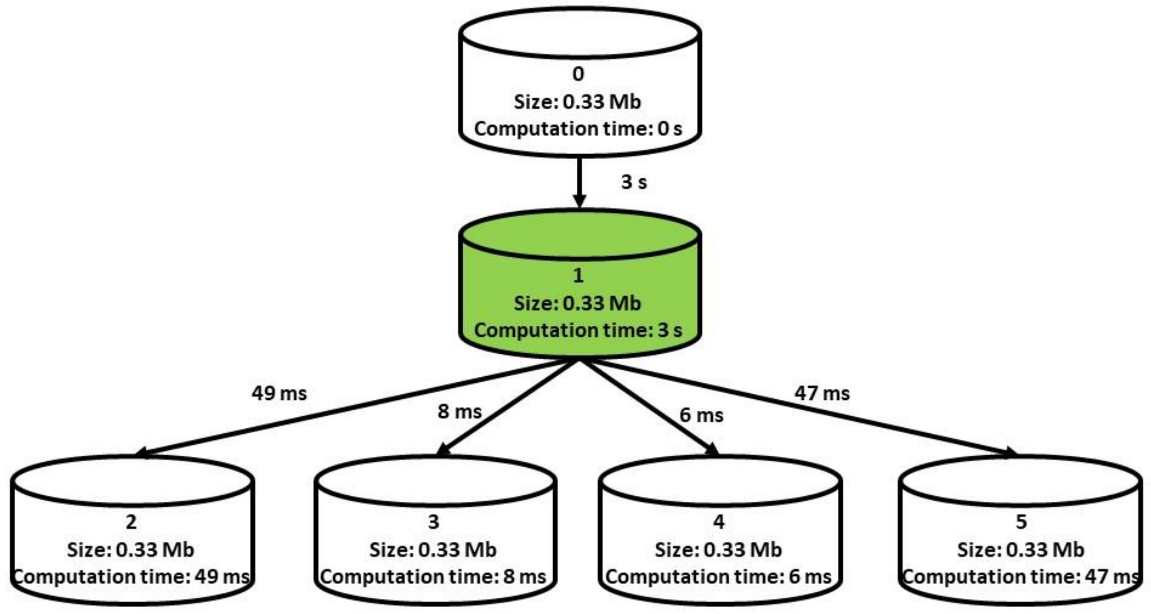

Figure 4. Computation time unaffected with parent being cached

Figure 5 below, explains the second scenario when a parent RDD is not cached. The formula to calculate the computation time in this scenario is as follows,

Computation time of current RDD = Computation time of Parent RDD + Transformation time of current RDD

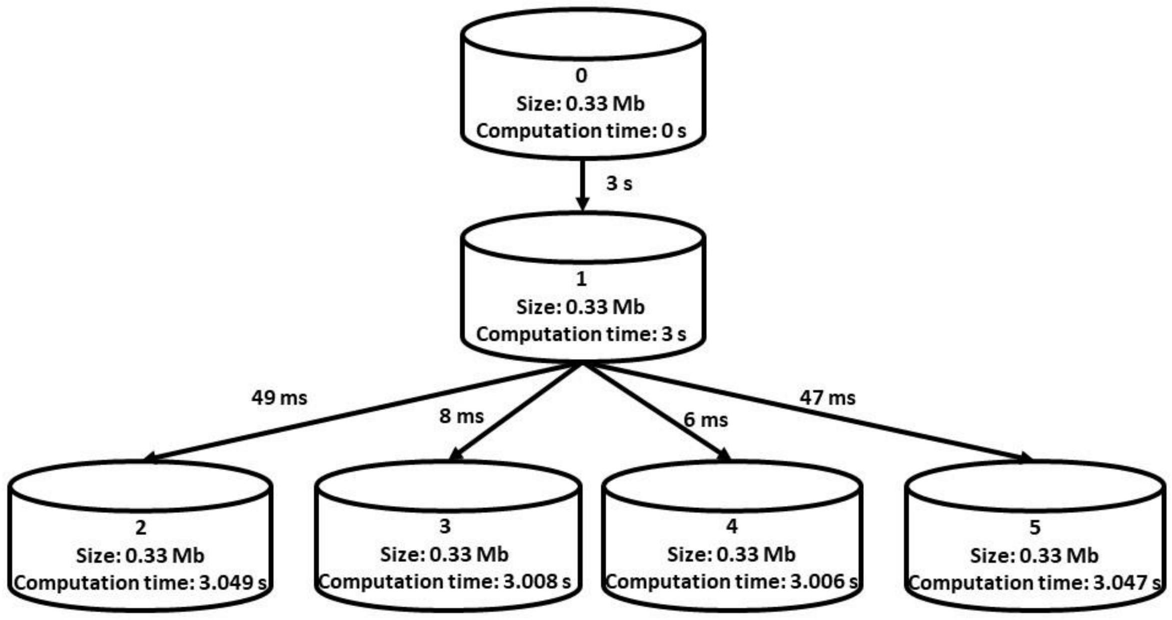

Figure 5. Computation time being affected with parent not cached

6.3 VISUALIZE

The logical view of an application from SparkCAD+ will use Graphviz for representation in the form of a Directed Acyclic Graph (DAG) of nodes and edges, where the nodes represent RDDs and the edges are the transformations that take place between them [1]. SparkCAD+ offers various configurations for the DAG graph to the users with which they can manipulate them to see the cache status of an RDD about whether it is cached or not, see whether an RDD’s transformation is narrow or wide, also about the caching anomalies occurrences, their computation time, size, name, their transformation time, etc. Although SparkCAD+ is used for Spark applications, it is important to note that the idea behind it can be applied to any other dataflow processing system (such as Flink and Storm) by updating the parse step (5.1) [1].

7. FACTHUB

In SparkCAD+, we introduced a powerful data structure called FactHub, which serves as a centralized repository to store comprehensive information about the application's jobs, stages, tasks, RDDs, and their relevant details. To populate the FactHub, we employed the parse method, which meticulously extracted and organized the required information from the application's log files. This process involved parsing each event and storing the relevant data in Python dictionaries, lists and sets within FactHub.

However, during the development of SparkCAD+, we identified a significant drawback in the original SparkCAD approach. When a user loaded a log file, the parse method had to traverse the entire file, processing and stripping events, which resulted in substantial waiting time, especially for complex applications. For instance, loading a sizable log file of 50 MB could take approximately 20 minutes. Furthermore, if the user wanted to make changes to the caching status of an RDD or visualize the DAG graph, they had to endure an additional 20 minutes for the parse method to reprocess and load the relevant information into the FactHub. As log file sizes increased, the processing time required by the parse method grew exponentially, further exacerbating the issue.

To overcome this drawback and improve the user experience, we implemented a caching mechanism in SparkCAD+. After the initial processing by the parse function, the FactHub data structure is serialized and cached in the local folder of the system. This caching step greatly accelerates subsequent operations on the application's data. When a user wishes to modify the caching status of an RDD or perform other actions that rely on the FactHub, only the relevant data is updated, while the previously cached FactHub is utilized for efficient retrieval and visualization tasks such as drawing the DAG graph. As a result, users experience significantly reduced waiting times and can interact with the application's data more promptly.

To enable caching functionality, we leveraged the capabilities of the pickle module in Python. This module provides binary protocols for serializing and de-serializing Python objects. In SparkCAD+, when a user loads a log file, the load_facthub_data function first checks if a. pickle extension already exists in the dedicated FactHubs folder within the current directory. If the .pickle file does not exist, the load_facthub_data function triggers the parse function to extract and organize the relevant information, subsequently populating the FactHub data structure. The FactHub is then serialized into a byte stream and stored as a .pickle file with a name corresponding to the application log file.

On subsequent calls to load_facthub_data, if the. pickle file for the log file already exists in the FactHubs folder, the function utilizes the unpickle operation to efficiently deserialize the stored byte stream back into the FactHub data structure. This way, the previously processed and cached FactHub is quickly retrieved and loaded, eliminating the need for repetitive parsing and significantly reducing the waiting time for users.

The introduction of the caching mechanism in SparkCAD+ has revolutionized the user experience by dramatically reducing the time required to perform various operations on the application's data. By taking advantage of the cached FactHub, users can save considerable time that would have otherwise been spent on repetitive parsing and data processing. SparkCAD+ effectively enhances productivity, enabling users to work more efficiently and effectively with their Spark applications.

8. DEMONSTRATION

Using Jupyter Notebook [6], a user can interactively run SparkCAD+ with the commands as demonstrated below:

Step 1: load_file('*filename*'): Will load the file from the "*logs*" folder in the current directory. The function passes the filename and checks if the same filename exists in the FactHubs folder with the "*.pickle*" extension. If exists, then the function will deserialize the "*.pickle*" file from a byte stream to an object hierarchy, storing them in the FactHub class.

Step 2: config.read('config.ini'): The config.read() method from the "*configparser*" module in Python is used to read configuration files in the INI file format. It allows users to parse and retrieve values from these files in a structured manner. Once the user has called config.read() with the correct file path, the ConfigParser object will parse the INI file, allowing users to access its sections and options using various methods provided by the configparser module. Note that the configuration file should have the "*.ini*" extension and follow the INI file format, which consists of sections enclosed in square brackets and key-value pairs within each section. For example, the user can enable or disable an option with the following commands,

- config['Drawing']['show_rdd_size'] = 'true'
- config['Drawing']['show_rdd_name'] = 'true'
- config['Drawing']['show_rdd_computation_time'] = 'true'

Like the above configuration commands with *true or false*, the user can able to enable or disable an option to see the results in the DAG graph. Note that, in the program, some of the configuration by default carries the value as *false.*

Step 3: cache(RDD id): This command will pass the RDD id given by the user and stores them in the set "*cached_rdds_set*" under the class AnalysisHub and subsequently remove the RDD id from the set "*non_cached_rdds_set*". The cached RDD will be displayed in the DAG graph in green colour and the computation times of the child RDDs to the cached RDD will get affected as explained in section 6.2.

Step 4: don’t_cache(RDD Id): cache(RDD id): This command will pass the RDD id given by the user and stores them in the set “*non_cached_rdds_set*” under the class AnalysisHub and subsequently remove the RDD id from the set “*cached_rdds_set*”. The non-cached RDD will be displayed in the DAG graph without any colour and the computation times of the child RDDs to the cached RDD will get affected as explained in section 6.2.

Step 5: Draw_DAG(): Calling this function without any parameter as a final command will call the functions responsible for analyze and visualize, which in turn will generate the DAG graph file.

9. FUTURE WORK

SparkCAD+ offers a comprehensive solution for processing the application logs and providing insights into the size and computation time of each RDD within the logical view of the application. With this information, users can make informed decisions about which RDDs to cache in the system's memory and which ones to exclude from caching.

However, there may be cases where the size of a particular RDD exceeds the available memory capacity, posing challenges in achieving an optimized caching plan. In such situations, users can consider caching RDDs in the available disk space. By leveraging disk caching, datasets can be stored both in memory and on disk, allowing for efficient utilization of available resources.

To effectively implement disk caching, users need to be aware of various memory and disk parameters. This includes considering the parameters such as read and write capacities of the memory and disk, size and computation time of RDDs, to know which particular RDDs to be cached and cluster configuration. By incorporating this information as input, users can devise a caching plan that balances the usage of memory and disk resources.

The decision-making process involves assessing the trade-offs between memory and disk caching, taking into account factors such as the criticality of the datasets, the performance impact of reading from disk, and the limitations of available resources. By considering these aspects, users can determine which datasets are more suitable for in-memory caching and which ones can be effectively cached on disk.

This approach paves the way for future work in optimizing RDD caching strategies. It enables users to make informed decisions about the allocation of datasets to memory and disk, taking into account the specific characteristics of the application and the available resources. By leveraging the information provided by SparkCAD+ and considering memory and disk parameters, users can design a caching plan that maximizes performance and resource utilization for their Spark applications.

10. CONCLUSION

SparkCAD+ enhances the caching decision-making process by leveraging the information extracted from application logs. It processes the logs and organizes the data into two distinct data structures: FactHub and AnalysisHub. FactHub contains immutable data that cannot be modified once processed and stored. This data serves as a reliable source of information and forms the foundation for subsequent analysis and visualization. On the other hand, AnalysisHub stores data that can be modified, allowing users to engage in interactive sessions and explore different scenarios for caching purposes. This flexibility enables users to make informed decisions by conducting "what-if" analyses and evaluating the impact of different caching strategies. By utilizing FactHub and AnalysisHub, SparkCAD+ provides valuable insights into the size and computation time of each RDD dataset within the logical view of the application. These insights are then visualized through DAG (Directed Acyclic Graph) graphs using tools like Graphviz. The DAG graphs visualize the RDD relationships and enable developers to understand the flow of data and computations in their Spark applications.

Overall, SparkCAD+ empowers Spark application developers by furnishing them with detailed information about RDDs. This information allows developers to make more informed caching decisions, optimize resource utilization, reduce computational overhead, and enhance overall application performance. By leveraging the data structures and visualization capabilities of SparkCAD+, developers can gain a deeper understanding of their applications and improve their caching decision plans.

11. REFERENCES

[1] Al-Sayeh, Hani, Muhammad Attahir Jibril, Muhammad Waleed Bin Saeed, and Kai-Uwe Sattler. "SparkCAD: caching anomalies detector for spark applications." *Proceedings of the VLDB Endowment* 15, no. 12 (2022): 3694-3697.

[2] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. In 2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10).

[3] Yinghao Yu, Wei Wang, Jun Zhang, and Khaled Ben Letaief. 2017. LRC: Dependency-aware cache management for data analytics clusters. In IEEE INFOCOM 2017-IEEE Conference on Computer Communications. IEEE, 1–9.

[4] Al-Sayeh, H., Memishi, B., Jibril, M. A., Paradies, M., & Sattler, K. U. (2022, June). Juggler: autonomous cost optimization and performance prediction of big data applications. In ACM SIGMOD

[5] [n.d.]. Python documentation. https://docs.python.org/3/library/pickle.html. Accessed: 2023-03-12

[6] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B Hamrick, Jason Grout, Sylvain Corlay, et al. 2016. Jupyter Notebooks-a publishing format for reproducible computational workflows. Vol. 2016.