ABSTRACT

Spark Framework and its successors have recently demonstrated its position as a new fault-tolerant eco-system for large-scale batch data analysis. They have shown to be highly scalable, support coarse-grained fault tolerance and provide easy to learn Application Programming Interface (API). Most applications are built to execute different jobs on these frameworks where they often share similar work (for instance, several jobs may use the same input data and/or produce the same output data which is used by next job). Hence, we can spot many opportunities to optimise the execution plan performances for majority of batch jobs. In this report, we plan to explore possible caching techniques in the literature for multi-job optimisation specifically for spark framework. We plan to go further and propose a simple yet efficient caching techniques and policies. Our contribution in this project would be surveying the current and recent literature and proposals of caching and optimisation algorithms that given an input batch of jobs, produces an optimal plan while identifying caching opportunities. Our other contribution is proposing a straightforward caching algorithm that would improve batch job's performance. If possible, we will report our experimental results on Spark deployment to demonstrate that our technique would improve the average completion time of Spark jobs.

1. INTRODUCTION

Motivated by lack abstractions for leveraging distributed memory and means of storing intermediate results which would benefit majority of Batch processing applications. It was found that many of these applications like logistic regression and interactive data mining involve the reuse of similar intermediate results multiple times and performing almost same operations on them. Hence, they proposed storing these as in-memory JAVA objects to provide a resilient distributed dataset for such queries along with lineage feature for fault recovery. To this end, Apache Spark [2] was proposed as a general purpose distributed data processing framework that provides fault tolerance through the concept of Resilient Distributed Dataset (RDD).

ACM ISBN 123-4567-24-567/08/06.
DOI: 10.475/123_4

An RDD is an immutable representation of a dataset that is either reliable by nature, i.e. stored in reliable external storage, or could be computed from the reliable datasets. Instead of storing the actual data, an RDD stores only its lineage information, i.e. the source of data and all transformations specified to compute the data. If any transformation fails under fault, Spark can recover by recomputing the dataset using its reliable ancestors. They also provided an API programming interface through Scala Language to ease the implementation of various programming models on top of Spark. The results are quite staggering and show impressive improvements over Hadoop framework. To summarise we were able to identify the following strength and weaknesses of Spark Framework.

1.1 Spark Strengths

We could identify the following strengths from Spark Framework which motivated our choice of seeking further improvement of its completion times.

1. It complements and address a missing feature of the modern large dataset mining applications which is storing intermediate results.
2. It provides an easy programming interface for faster adoption by many application developers and data analytics.
3. It allows for efficient while less storage-wise costly alternative for fault recovery through leveraging lineage of job stages.
4. RDDs are read-only meaning they can be written out in the background without any program pauses or read-write locking mechanisms. This is quite a promising feature for making any caching algorithm tractable.
5. Spark was 20X and 40X faster than Hadoop for iterative and a real-world data analytics as well as it can scan 1 TB dataset with 5âĂŞ7s latency.
6. The framework has been evaluated in research (controlled environment) as well as in a real application deployment by Conviva Inc and Mobile Millennium Project.

2. CACHING IN SPARK

A typical Spark program will construct the actions in an incremental way. The user will usually construct the first RDD using an external data source, usually from file systems or databases. A new RDD is constructed, either explicitly or implicitly, for each transformation applied to the existing RDDs. The Spark program will usually end with one or more actions that either prints out the results

or stores the transformed data to external storage. All the RDDs will form a Directed Acyclic Graph (DAG) and the output vertices given by these actions will invoke the actual computation for all their RDD dependencies.

However, the current implementation of Spark evaluates each action as a single Spark job. For Spark programs that contain multiple jobs, we can reduce the computation and storage overhead if the dependency graphs for different jobs share a common substructure. Intuitively, if an RDD is a dependency for multiple Spark jobs, it will be beneficial to avoid re-computation by caching the computed result after its first evaluation. Conversely, if an RDD is a dependency for only a single Spark job, its results will not be reused and its cache could be eliminated after computation to save the storage space.

The following code snippet shows a possible caching optimisation opportunity in the execution of this job:

Listing 1: Possible Caching Example

```
val lines = spark.textFile("hdfs
    ://...")
val num_error_php = errors.filter(_.
    contains("php")).count()
val num_error_mysql = errors.filter(_.
    contains("mysql")).count()
val errors = lines.filter(_.startsWith
    ("ERROR"))
errors.cache() ← this is where the caching kicks in

Cashed version of errors dataset can be used in two
    different jobs
val num_error_php = errors.filter(_.
    contains("php")).count()
val num_error_mysql = errors.filter(_.
    contains("mysql")).count()
```

Note that the last two variables specify two independent Spark jobs. The line of code with an arrow here is important because without it, the first two lines of code will evaluate each time for the counting task. However, after the multi-job optimisation, the caching statement should be injected to the user program without manually specifying it as shown in Figure 1.

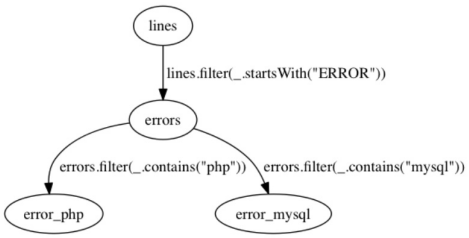

Figure 1: Injection of caching mechanism into the jobs execution path of spark

2.1 Work Plan

The key observation from the aforementioned example is that each Spark RDD is represented as a immutable variable in the Scala program, either explicitly declared using val keyword or implicitly declared in the return value of chaining method calls. The dependency graph of all RDDs is isomorphic to the data flow graph of RDD variables, which is available at compile-time when constructing the abstract syntax tree for that program. Hence, a straightforward approach would be to analyze the out degrees of the data flow graph or RDD variables after pruning the non-action paths then there is a chance for the caching policy to be applied at this point. More complex cases involving control flow analysis, e.g. with if conditions and while/for loops, will also be investigated in this project.

3. RELATED WORK

A number of recent works addressed this shortcoming of Spark Framework to improve the execution time and reduce resource usage to be able to pack more jobs within the cluster. [1] proposed two new techniques for multi-job optimization targeting the MapReduce framework. A grouping technique to merge multiple jobs into a single job via sharing both the scan operation of input file and the communication of the common map output. And a materialization technique that share input/ouput through partial materialization of the map output of some jobs. They achieve this by constructing an optimal execution plan by partitioning jobs into groups and assign necessary processing to each group.

4. REFERENCES

[1] G. Wang and C.-Y. Chan. Multi-query optimization in MapReduce framework. *Proceedings of the VLDB Endowment*, 7(3):145–156, Nov. 2013.

[2] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.