Spark-BDD : A Big Data Debugger

Muhammad Ali Gulzar
UCLA

Tianyi Zhang
UCLA

Seunghyun Yoo
UCLA

ABSTRACT

Apache Spark has become a key platform for Big Data Analytics, yet it lacks complete support for debugging analytics programs. As a result, the development of a new analytical toolkit can be a painstakingly long process. To fill this gap, we developed Spark-BDD (Big Data Debugger), which brings a traditional interactive debugger experience to the Spark platform. Over the course of this project we integrated current version of Spark-BDD with some interactive fault localization and optimization mechanisms. Analytic programmers (e.g., data scientists) can leverage Spark-BDD interactive debugging capabilities to set breakpoints and watchpoints, investigate crashes and failures, identifying straggling data partitions. Along with debugging this tool provides some useful runtime information that can be helpful in improving the performance of Spark in an unpredicted cluster deployment. One of these optimizations is inter-stage repartitioning to avoid data skew during data shuffling where performance improvement is reflected in the Spark job competition time. Most of these debugging features pose minimum overhead and in turn provide valuable debugging with lowest number of iterations.

Keywords

Debugging, Apache Spark, Fault Recovery & Localization, Performance

1. INTRODUCTION

Large scale data processing is the hottest topic these days. As the accessibility of the technology and methods to collect is improved , the amount of data is also increasing exponentially. Processing these massive amount of datasets are both difficult and necessary. As scene from he experience single computer super machines are capable enough to to analyze that scale of data. To overcome that cluster computing frameworks are the right path to perform these kind of analysis. Large companies like Google and Facebook have there own data center which they use to process their data with small companies lease subset of inter connected machines from third party resource provider like amazon. Since the availability of these kind infrastructure is not an issue anymore, the need for software framework that be ran on it is imminent. Apache Spark is one of these large scale data processing framework that can be run cluster computing frameworks with thousands of nodes. Spark is built on MapReduce programming paradigm from LISP but also provides data processing operation like GroupBy and Joins. Several other frameworks before Sparks also encountered the same problem in the same distributed way but most of them were either slow (Hadoop) or complicated (Dryad). Spark cater both of these issued by providing an in-memory model for lighting fast speed and simple Scala, java and python API for easier understanding. The choice of Spark for this project is because of its growing popularity in the community, impressive performance and research support as a lot go research lab are working in or on top of of Spark.

In this project we present a debugging tool called Spark-BDD build on top of Spark to debugging feature that are usually available to user in single process programs. The basic aim was to develop a tool corresponds to the features available to users on conventional debuggers as the users of Spark are mostly ML and Data Science that are are not very familiar with the Spark internals. By hiding the internal details, Spark-BDD introduces a layer of debugging build in the same API and a web based UI to perform debugging both physical and data layer. We support features like breakpoint, watchpoint, stragglers profiling, crash culprit identification and recovery. All of these features supports wide array of use cases and drives a new optimizations and performance improvements from the measurements collected during debugging runtime. Later in the report we also show the overhead of this frameworks and one of the performance improvements based the stragglers profiling information.

2. MOTIVATION

With the abrupt increase in the adaptation of these frameworks, debugging these frameworks are getting challenging too. Current state of these frameworks does not allow any kind of debugging support. The only debugging support that comes build in with analytic frameworks are single process mode where user can run a job in single computer with subset of data. But still this does not add any thing to the fault localization task as it lacks the distributed infrastructure, actual dataset and resource sharing. In case of Spark, the application written by user does not comply with the actual execution because lazy evaluation of RDDs and some optimizations. Running a step by step debugger of Scala and Java like JDB, would not help just like a multithreaded applications. Since everything is distributed, the actual tasks are executing at the works which are hard to track from current logging mechanism. Spark logging can help at the physical layer of the framework where you can have abstract information of the tasks while debugging at data layer is necessary.

Cluster competing frameworks are usually resource hungry

and also consumes a lot time to execute a job. Once the job is submitted the user is kept in the dark until the end of the job or the occurrence of the failure. In case of failure all the previous computations are wasted because of Spark in memory design and the user need to start the job from start by doing the redundant work. This scenario resulted in both resource and time wastage and the later the failure occur more wastage would happen (If not cached). Early detection and identification is crucial for effective debugging and resource saving. All these drawbacks of using analytic frameworks motivate the development of a debugging environment to can enable users detect the logical errors in their implementations and refine the source of the fault on the data plane.

3. APPROACH

Considering the motivations and the current debugging need of Spark, we built a debugging environment integrated within Spark to support most of the features to perform extensive debugging both on physical and data plane. Spark-BDD is build with in the internals/core of Spark to have control over the low level details of then jobs so that even the minute actions could be shown to the user as per need. This debugger also provides an interactive UI designed along the with the current Spark monitoring UI to give low level controls to the user. The idea here is to provide essential details to the user to detect faults and logical errors that can not be detected through the normal logging mechanisms. Interactive user interface provides runtime details to the user to make debugging decisions like step in/out, put monitors, handling crashes, tracing faulty data source and tracking latencies.

4. RELATED WORK

TODO

5. IMPLEMENTATION - FEATURES

The current state of the debugger performs the features described in the sub sections below. All the features are implemented with in the internal core code of Spark as hooks to provide the most efficient and robust implementation. Along with the debugging backend systems , almost all the features are supported and controlled through both programming API in both Java and Scala and also using the web based UI.

5.1 Breakpoint & Watchpoint

Breakpoint one of the most important and interactive debugging feature to the users. A Spark program written by the user can be paused and resumed when user put a breakpoint between a set instructions. The idea here is to match the debugging experience that the users usually had in conventional debugging environment. Though both UI and programming API user can specify the breakpoint where he/she wants to put a breakpoint. Once placed, a user can run the program and it will start the computing the tasks at different nodes and once that instruction is reached where the breakpoint is placed the debugger will pause the execution right there. It is important to understand the granularity of this feature as the user write a Spark program at the RDD level which is a subtask that runs on each node on the allotted partition and the original input is partitioned into tasks which are assigned to worker nodes. A task can contain subtasks (only one-one dependency RDDs) which are actually the RDDs based instructions written by the user. Whenever there are one to many dependencies between RDD a stage boundary is created and the tasks before had to be completed before it can proceed with the next stage. Spark-BDD provide support to place breakpoints at all levels of physical and programming layer of a Spark program which includes stages, tasks and subtasks.

Pausing the execution itself at a certain point in the program does not provide valuable information about program. We needed some kind of a monitoring tool just like variable watching in normal debugging to watch the data flowing between stages and subtasks. Spark-BDD's Watchpoint catered that need by tapping the intermediate data flows at the breakpoint. User can see the data records monitored at that breakpoint and analyze the intermediate data to detect and eliminate the faults and errors. Considering the scale of these analytical programs the tapped data could in the order of giga bytes and keeping that data in memory is not a viable option as this data need to be showed to the user and since the user only interact with the one node (Driver), sending all this data over the network will cause tremendous overhead.

To avoid this overhead we implemented a filter mechanism for our monitor. Watchpoint filter mechanism is but on top of Scala and Java, so the user can actually write the filters like a normal Spark program. For user to write this filter, he only need to write the implementation of a BDDPredicate class's filter function. The filter function takes in a Key-Value pair and returns a boolean value to indicate whether that record is to be sent to user or not. A user can provide this predicate implementation both in the Spark program or in UI. Flexibility is also provided in the sense that user can dynamically write the filters and submit it into the on going Spark job. The debugger will then dynamically compile it and ship the compiled predicate to the workers. Since watch point is the feature that poses the highest overhead, we adapted a set of optimizations to minimize the network and disk traffic including batch sending, on demand watchpoint and volatile watchpoints. With watchpoint and breakpoint feature at disposal , a user can oversee the intermediate and run some simple and short analysis to detect the outliers , anomalies, faults or any kind of logical errors that can not be detected either through logs or by observing the original Spark program.

5.2 Stragglers

Current version of Spark provides a high level of information on how long did each work take to perform the task. But due to optimization most the transformation are done in a batch and current latency approach does not give you per transformation level latency so that you will exactly which transformation is causing trouble. To localize the abnormal behavior at the most finest level, we implemented straggler detection both at subtask and record level. User can enable the configurations for subtask or record level latency and get the execution/transformation times for each RDD and record respectively. Subtask level latency is important where the counts of records are very huge and most of computations are very basic, this will help in isolating the transformations and workers. Record level latency is crucial where time consuming transformation happen mostly on per

record. This will help us identify the bad or outlying records that are delaying the job. User is usually given the information at runtime that which of the transformations and records are straggling along with the physical layer information like information on workers performance. Worker based stragglers can provide some useful information, which consequently can help to perform runtime scheduling and shuffling optimizations.

5.3 Crash Culprit

Spark doesn't have a mechanism of handling an exception caused by a program at runtime. If a Spark program doesn't handle the exception, the program is going to be terminated because of the unexpected exception. Some of the datasets may contain specific values that are able to raise exceptions. For example, missing values, values not following a predefined type, out of range, and even locale problem can terminate the program. To be specific, the type of value should be strictly integer, but the actual value could be Ń/A; which causes parsing errors. Therefore, a user needs to cleanse the given dataset until there is no further exception. Considering the distribution of values causing the unexpected exceptions is unknown, this procedure involves too much waste of time and resources due to redundant computation.

We added a feature of a crash culprit, which means a dataset that causes a program crash at runtime. The reason we called the dataset as ćulpritís that the cause of an error might be coming from external sources like hardware failure or operating system level failure. We support two way of resolving crash culprits: (1) to skip the dataset, (2) to modify the dataset. Therefore, a user may continue the program execution without making the program terminated by ignoring or fixing the dataset at runtime.

5.4 Crash Recovery

5.5 Re-Partitioning

6. EVALUATION

7. CONCLUSION

8. REFERENCES