Haram Kim, Siddhant Navali
Data Engineering Team
Fall 2017 Report

Server Usage Restrictions

We considered several methods to control server resource consumption and to prioritize jobs running on Spark over general use processes. These choices were constrained both by complexity and the server configuration, which relies on team-level accounts as opposed to accounts for individual users. The primary objective of this endeavor was to ensure that a user running a very computationally intensive process on their team account on the master node would not inadvertently interfere with the operation of the Spark cluster.

Option 1 was based on the Auto Nice Daemon, which uses the Unix niceness process flag to change the operating system's CPU time scheduling priorities. The specifics of the operating system's implementation of niceness are opaque, but typically it will allocate CPU time starting with very high niceness processes and leave low niceness processes to hang until higher priority tasks are completed. Some granularity in enforcement was possible, as niceness can range from 19 to -19, but the opacity of the Ubuntu scheduler made it hard to judge how best to specify default niceness values. This solution was attractive because it was easy and intuitive; with the AND configured to start on boot and using a configuration file to automatically reduce the priority of team user account-spawned processes to -19 (or some other similarly low value), we could prevent their processes from interfering with Spark's CPU time. This solution was unattractive because it lacked clearly defined granularity, provided control on only a per-process basis, and did nothing to address problems arising from excessive memory consumption (which could also interfere with the Spark cluster).

Option 2 was the Ulimit system, based on the setting in security.conf. This solution relied upon hard resource caps set for each user in the appropriate configuration file, and enforced by the operating system. It had the advantage of providing control over both CPU priority and memory usage, in addition to restrictions on concurrent logins and other potentially performance-harming behavior. However, it applied its memory limits on a per-process basis. Although in theory this might be sufficient to prevent excessive memory consumption, it would provide no protection if a user attempted to run a large number of intensive programs concurrently.

Option 3 was the Control Groups Rules Engine Daemon, in combination with Unix control groups. Control groups allow a set of member processes to be limited to a shared pool of resources, preventing issues arising from a large number of parallel programs. They also allow control over the two primary areas of concern for this problem: memory and CPU usage. CGroup enforcement is most easily automated using the cgrulesengd daemon, which could assign processes to CGroups based on the creating user and the process name. This allowed spark processes explicitly started on user accounts to be exempted from restrictions, while ensuring that each user account's general use allocation was limited to a specific CPU and memory pool.

The current server setup uses Option 3, with a systemctl service responsible for creating the appropriate CGroups (2 per user; 1 for general use with standard limits, and 1 without limits for exempted processes) from a specification file at boot, and starting the daemon. Changes to general use allocations can be made by changing the limits specified for a user's general use CGroup in the specification file, then restarting the systemctl service. No server restart is required.

Server Usage Monitoring

Remote graphical server usage monitoring is accomplished through Cockpit, an open-source web monitoring application. It is run at boot in webserver form on the master node, which can be connected to by browsers over port 9090. The service allows users to monitor overall CPU, memory, HDD, and network usage, provides access to a local terminal, and allows for the inspection of mounted volumes. The three slave nodes also have Cockpit installed, and are linked to the master node's web interface, allowing the same functionality across the entire cluster. Full cluster monitoring is currently available to the administrator user, however, because Cockpit logs in to the slave nodes using the credentials provided to it when a user logs in over the web, and team accounts currently do not exist on the slave nodes. In the future, a fully deployed version of this system should be able to provide a complete picture of cluster status, and may have its graphs embedded into an external website for convenience.

MySQL on Spark

Spark can connect to MySQL and other SQL databases using Java JDBC connector drivers, which are compatible with Spark's Scala backend. There are four available types of JDBC drivers: JDBC-ODBC, Native-API, Network-Protocol, and Database-Protocol. The driver for MySQL, the MySQL Connector/J, is a type 4, Database-Protocol driver. This type of driver is optimized for connecting to a single type of database, and is typically the fastest. This is because the driver connects directly both the Local Database Management System and the Database server itself. In other drivers, the connection to the Local DBMS is not as simple and involves a number of additional steps including connecting with the Database Vendor's drivers. Type 3 JDBC drivers are helpful when connecting to multiple types of data databases simlultaneously.

We also examined the caching that occurs when Spark is used with MySQL. At the top level, when a query is run, Spark automatically caches the result of the query so that if it is run again before any data has been altered, the results can be quickly pulled directly from memory. The MySQL server also does this type of query caching by default, although this behavior can be disabled. This type of caching is useful for cases in which large numbers of similar SELECT queries are expected, with few modifications to the database. In other cases (such as those where the database is changed frequently), the cache becomes stale (that is, containing data that has changed on disk) more quickly and may actually lead to a performance decrease [1]. Query caching is also not supported for partitioned tables, which can be used with Spark to greatly speed up querying of attached MySQL databases.

In addition to the potential problems of query caching, it is clear that the presence of a query cache in both Spark and MySQL is redundant, as they will both attempt to store the most recent non-stale queries in memory. If the Spark session and MySQL server are running on the same machine, this will waste memory that might be allocated to other tasks. In such a use case, query caching should be disabled on either Spark or the MySQL server. This issue is further complicated by the fact that the InnoDB database engine also performs caching through its buffer pool. This system differs from Spark and MySQL query caching because it does not keep recent query results in memory. Instead the buffer pool attempts to keep both table indices and data in memory to avoid disk operations where possible, or to make lookups more rapid when not [2]. The buffer pool works well with MySQL query caching and can improve performance greatly when memory is available. However, for the use case mentioned previously, a large buffer pool may not necessarily be the most efficient use of available memory, especially if Spark itself is instructed to cache required tables in memory.

Spark Profiling

We explored the possibility of profiling the performance of Spark applications visually using JVM profiling tools, such as VisualVM. This type of profiling can complement monitoring provided by services like Cockpit because they can report the memory actually used by the application, as opposed to just the total heap allocated to the JVM (which is never fully used, as the JVM expands the heap whenever it approaches full occupation). We were successful in profiling running Scala-Spark applications in local tests (please see Figure 1), and observed a significant disparity between heap size and used memory, which suggested a potential space for improvement. However, we discovered that it was not feasible to profile larger applications on the servers (although it was possible to attach profilers to a remote JVM) because each node in the Spark cluster had 5 separate JVM instances for each of the running nodes. Although VisualVM and other profilers theoretically allowed multiple concurrent connections, this would require 20 for the current server configuration, all of which would have to be monitored and integrated. The recommended profiling tool for Spark distributed processes, YourKit, is unfortunately closed-source and commercial, so unfortunately continued pursuit of this possibility is presently prohibitive.

Catalyst Optimizer

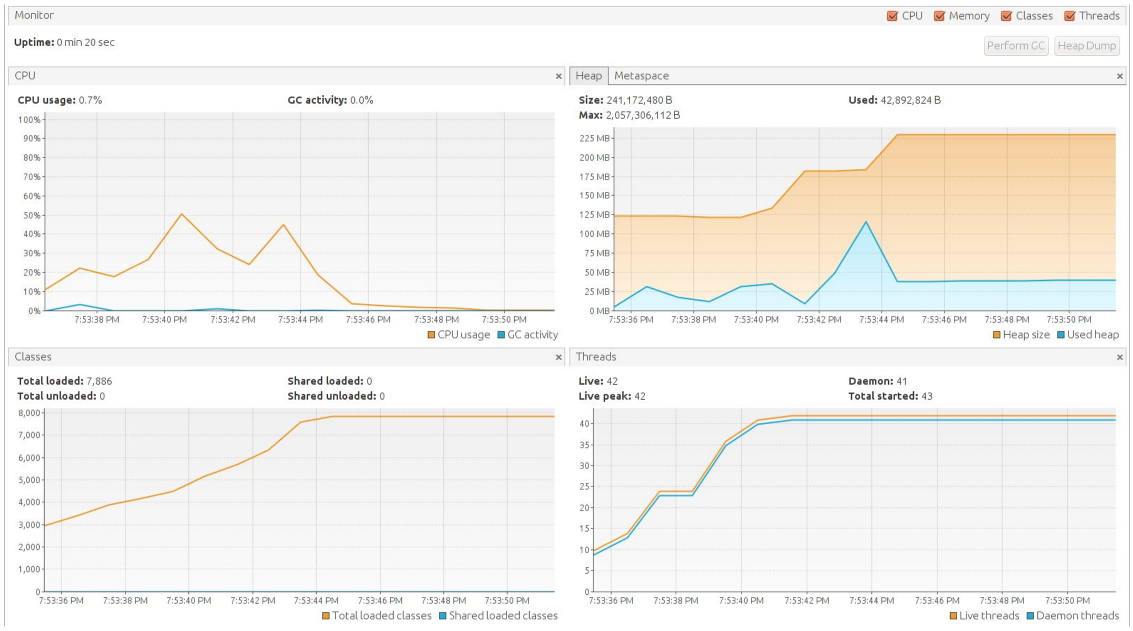
When looking into the low-level specifics behind the SQL in Spark, we found out about the Catalyst Optimizer. This at the center of how Spark speeds up SQL processes.
The entire process can be broken down into 4 steps (as can be seen in Figure 2):

1. Resolve references in Logical Plan
2. Optimizing Logical Plan
3. Creating a Physical Plan
4. Generating the code

The first step involves taking in an abstract syntax tree that is generated by an SQL parser. From there, all the tables and column names are checked for and resolved from the SQLContexts that were created in the Spark process. From there optimizations are made using the rules available. These rules change the shape of the of the original logical plan by merging parts of the query together to minimize the number of computations, and by reordering the query if possible. The second part is where much of the performance increases occur, and we can start creating our own specific rules to help further speed up processes. The rules created utilize Scala's pattern matching featuresThe third step involves creating several physical plans and running them through a cost model to determine which one will have the highest performance. From there, a Scala's quasiquotes can be used to generate code and run the process.
The general idea of the Catalyst Optimizer is that Abstract Syntax Trees are created and the manipulated using rules so that the SQL query (or operation on a DataFrame) can do as little computation as possible.

Figures

Figure 1: VisualVM Profiling of Sample Scala-Spark Application


Monitor

Uptime: 0 min 20 sec

CPU

CPU usage: 0.7%
GC activity: 0.0%

Heap | Metaspace

Size: 241,172,480 B
Max: 2,057,306,112 B
Used: 42,892,824 B

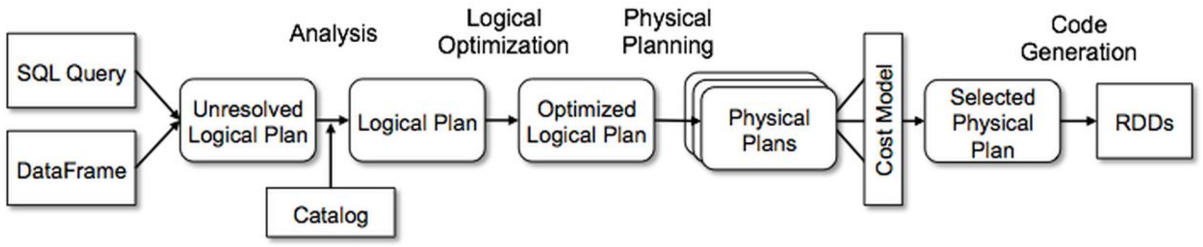
Classes

Total loaded: 7,886
Total unloaded: 0
Shared loaded: 0
Shared unloaded: 0

Threads

Live: 42
Live peak: 42
Daemon: 41
Total started: 43

Figure 2: The Catalyst Optimizer in a Flow Chart



References

1. MySQL Reference Manual, section 8.10.3 The MySQL Query Cache. https://dev.mysql.com/doc/refman/5.7/en/query-cache.html
2. MySQL Reference Manual, section 14.9.2.1 The InnoDB Buffer Pool. https://dev.mysql.com/doc/refman/5.5/en/innodb-buffer-pool.html
3. DataBricks, Paper on Catalyst Optimizer http://people.csail.mit.edu/matei/papers/2015/sigmod_spark_sql.pdf