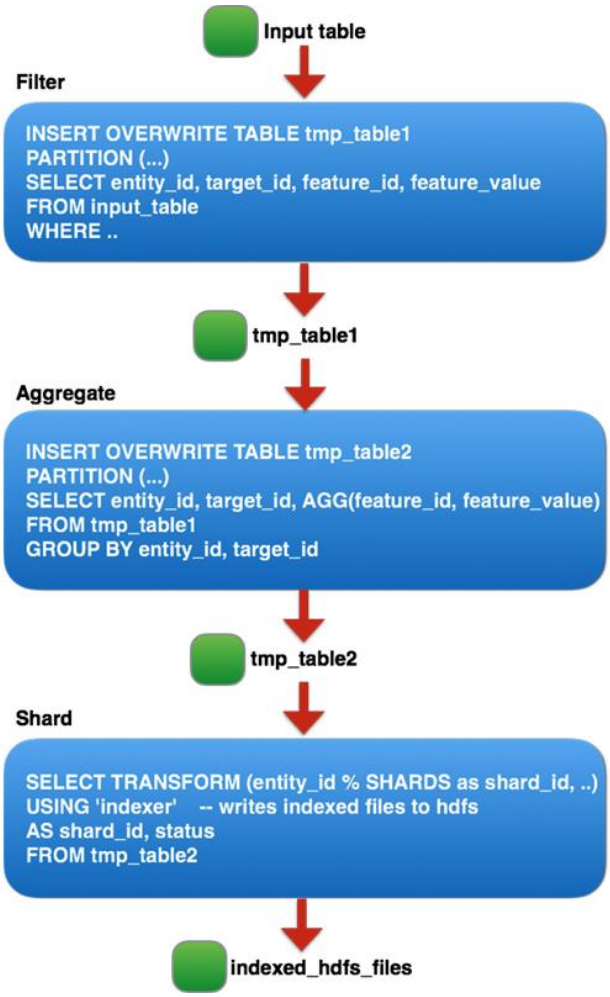
Client: Capital One

Use case: Feature preparation for entity ranking using Bigdata stack

Real-time entity ranking is used in a variety of ways at Capital One. For online serving platforms raw feature values are generated offline with Hive and data loaded into its real-time affinity query system. The existing Hive-based infrastructure built years ago was computationally resource intensive and challenging to maintain because the pipeline was sharded into hundreds of smaller Hive jobs. In order to enable fresher feature data and improve manageability, Wipro took one of the existing pipelines and tried to migrate it to Spark.

Previous Hive implementation

The Hive-based pipeline was composed of three logical stages where each stage corresponded to hundreds of smaller Hive jobs sharded by entity_id, since running large Hive jobs for each stage was less reliable and limited by the maximum number of tasks per job.



The three logical steps can be summarized as follows:

1. Filter out non-production features and noise.
2. Aggregate on each (entity_id, target_id) pair.
3. Shard the table into N number of shards and pipe each shard through a custom binary to generate a custom index file for online querying.

It was also challenging to manage, because the pipeline contained hundreds of sharded jobs that made monitoring difficult. There was no easy way to gauge the overall progress of the pipeline or calculate an ETA. When considering the limitations of the existing Hive pipeline, we decided to attempt to build a faster and more manageable pipeline with Spark.

Reliability fixes

Dealing with frequent node reboots

In order to reliably execute long-running jobs, we want the system to be fault-tolerant and recover from failures (mainly due to machine reboots that can occur due to normal maintenance or software errors). Although Spark is designed to tolerate machine reboots, we found various bugs/issues that needed to be addressed before it was robust enough to handle common failures.

- Make PipedRDD robust to fetch failure (SPARK-13793): The previous implementation of PipedRDD was not robust enough to fetch failures that occur due to node reboots, and the job would fail whenever there was a fetch failure. We made change in the PipedRDD to handle fetch failure gracefully so the job can recover from these types of fetch failure.
- Configurable max number of fetch failures (SPARK-13369): With long-running jobs such as this one, probability of fetch failure due to a machine reboot increases significantly. The maximum allowed fetch failures per stage was hard-coded in Spark, and, as a result, the job used to fail when the max number was reached. We made a change to make it configurable and increased it from four to 20 for this use case, which made the job more robust against fetch failure.
- Less disruptive cluster restart: Long-running jobs should be able to survive a cluster restart so we don't waste all the processing completed so far. Spark's restartable shuffle service feature lets us preserve the shuffle files after node restart. On top of that, we implemented a feature in Spark driver to be able to pause scheduling of tasks so the jobs don't fail due to excessive task failure due to cluster restart.

Other reliability fixes

- Unresponsive driver: Spark driver was stuck due to O(N^2) operations while adding tasks, resulting in the job being stuck and killed eventually. We fixed the issue by removing the unnecessary O(N^2) operations.
- Excessive driver speculation: We discovered that the Spark driver was spending a lot of time in speculation when managing a large number of tasks. In the short term, we disabled speculation for this job. We are currently working on a change in the Spark driver to reduce speculation time in the long term.
- TimSort issue due to integer overflow for large buffer: We found that Spark's unsafe memory operation had a bug that leads to memory corruption in TimSort. Thanks to

Databricks folks for fixing this issue, which enabled us to operate on large in-memory buffer.

- Tune the shuffle service to handle large number of connections: During the shuffle phase, we saw many executors timing out while trying to connect to the shuffle service. Increasing the number of Netty server threads (spark.shuffle.io.serverThreads) and backlog (spark.shuffle.io.backLog) resolved the issue.

Performance improvements

After implementing the reliability improvements above, we were able to reliably run the Spark job. At this point, we shifted our efforts on performance-related projects to get the most out of Spark. We used Spark's metrics and several profilers to find some of the performance bottlenecks.

Tools we used to find performance bottleneck

- Spark UI Metrics: Spark UI provides great insight into where time is being spent in a particular phase. Each task's execution time is split into sub-phases that make it easier to find the bottleneck in the job.
- Spark Linux Perf/Flame Graph support: Although the two tools above are very handy, they do not provide an aggregated view of CPU profiling for the job running across hundreds of machines at the same time. On a per-job basis, we added support for enabling Perf profiling (via libperfagent for Java symbols) and can customize the duration/frequency of sampling. The profiling samples are aggregated and displayed as a Flame Graph across the executors using our internal metrics collection framework.

Performance optimizations

- Fix memory leak in the sorter (30 percent speed-up): We found an issue when tasks were releasing all memory pages but the pointer array was not being released. As a result, large chunks of memory were unused and caused frequent spilling and executor OOMs. Our change now releases memory properly and enabled large sorts to run efficiently. We noticed a 30 percent CPU improvement after this change.
- Snappy optimization (10 percent speed-up): A JNI method — (Snappy.ArrayCopy) — was being called for each row being read/written. We raised this issue, and the Snappy behavior was changed to use the non-JNI based System.ArrayCopy instead. This change alone provided around 10 percent CPU improvement.
- Reduce shuffle write latency (up to 50 percent speed-up): On the map side, when writing shuffle data to disk, the map task was opening and closing the same file for each partition. We made a fix to avoid unnecessary open/close and observed a CPU improvement of up to 50 percent for jobs writing a very high number of shuffle partitions.
- Configurable buffer size for PipedRDD (10 percent speed-up): While using a PipedRDD, we found out that the default buffer size for transferring the data from the sorter to the piped process was too small and our job was spending more than 10 percent of time in copying the data. We made the buffer size configurable to avoid this bottleneck.

After all these reliability and performance improvements, we are pleased to report that we built and deployed a faster and more manageable pipeline for one of our entity ranking systems, and we provided the ability for other similar jobs to run in Spark.

Spark pipeline vs. Hive pipeline performance comparison

We used the following performance metrics to compare the Spark pipeline against the Hive pipeline. Please note that these numbers aren't a direct comparison of Spark to Hive at the query or job level, but rather a comparison of building an optimized pipeline with a flexible compute engine (e.g., Spark) instead of a compute engine that operates only at the query/job level (e.g., Hive).

Conclusion and future work

Wipro uses performant and scalable analytics to assist in product development. Apache Spark offers the unique ability to unify various analytics use cases into a single API and efficient compute engine. We challenged Spark to replace a pipeline that decomposed to hundreds of Hive jobs into a single Spark job. Through a series of performance and reliability improvements, we were able to scale Spark to handle one of our entity ranking data processing use cases in production. In this particular use case, we showed that Spark could reliably shuffle and sort 90 TB+ intermediate data and run 250,000 tasks in a single job. The Spark-based pipeline produced significant performance improvements (4.5-6x CPU, 3-4x resource reservation, and ~5x latency) compared with the old Hive-based pipeline, and it has been running in production for several months.