SPIP: Incremental Stats Collection

Author: Rakesh Raushan(raksonrakesh@gmail.com)

Q1. What are you trying to do? Articulate your objectives using absolutely no jargon.

The aim of this SPIP is to collect the table, partition and column statistics incrementally as its name suggests. This would allow users to take full advantage of Spark's Cost Based Optimizer(CBO) as we would be keeping our statistics updated at all the time without full table scan. This can be especially beneficial for frequently updated tables. We would update our statistics automatically with each DML query over tables.

Q2. What problem is this proposal NOT designed to solve?

Q3. How is it done today, and what are the limits of current practice?

Currently, after every run of a DML query, statistics are invalidated and thus CBO becomes non-functional. Statistics can be updated again using following two approaches:

a.) ANALYZE TABLE COMPUTE STATISTICS can be used to update the statistics. This is an expensive query as it will do a full scan over the table. Thus running it after every DML statement is not feasible.

b.) Set `spark.sql.statistics.size.autoUpdate.enabled` to true which would update table and partition statistics. But this itself can be costly in some scenarios.

Q4. What is new in your approach and why do you think it will be successful?

We would use the internal APIs introduced by SPARK-21669. *WriteTaskStatsTracker* would be extended to track task level statistics and *WriteJobStatsTracker* would be extended to aggregate the collected task level statistics on the driver. After aggregation, statistics would be persisted in the metastore. These trackers would be added to all data changing commands.

Statistics are collected on 3 levels,

a.) Table level (numRows and size)

b.) Partition level (numRows and size)

c.) Column level (distinctCount, min, max, nullCount, avgLen, maxLen, distinctCountForIntervals)

Collecting stats on table and partition levels would be relatively straightforward using trackers. Each WriteTaskStatsTracker would provide numRows and size per task which would later be aggregated.

In a similar way we can also track and update min, max, nullCount, avgLen and maxLen fields for column statistics. For distinct count we would use the newly introduced Datasketches HllSketch. We would not update histogram incrementally for now.

Why would it be successful?

Majority of the incremental collection would have very little impact over query performance and whatever performance impact collecting distinctCount introduces would be outweighed by the benefits it can provide to end users. Also it is quite common in DBMS to keep the stats updated at all the time. There are quite a few spark JIRAs enquiring about the same in spark as well.

Q5. Who cares? If you are successful, what difference will it make?

This would benefit all spark SQL users. It would allow CBO to be functional at all the time for all the tables as we are keeping stats updated.

Q6. What are the risks?

- Automatic collection of few column level statistics(distinctCount) might degrade performance of some DML queries.
- This proposal will guarantee freshness of stats only if the tables are updated using spark queries. Updating external table locations manually can lead to inconsistent stats.

Q7. How long will it take?

As per my estimates, this would take 3-4 months.

Q8. What are the mid-term and final exams to check for success?

Mid-term: In the first stage, complete implementation for table and partition stats.

Final: In the final phase, implement column level statistics.

Appendix B. Optional Design Sketch: How are the goals going to be accomplished? Give sufficient technical detail to allow a contributor to judge whether it's likely to be feasible. Note that this is not a full design document.

```
case class IncrementalWriteTaskStats(numRows: BigInt, size: BigInt) extends
WriteTaskStats

class IncrementalWriteTaskStatsTracker extends WriteTaskStatsTracker {
 override def newPartition(partitionValues: InternalRow): Unit = {
   // A new partition is being written, increment the partition count
 }
 override def newFile(filePath: String): Unit = {
   // A new file is being written increment the file count
 }
 override def newRow(filePath: String, row: InternalRow): Unit = {
   // A new row is getting added, increment row count, also use the row information
   // to update min, max, nullCount, avgLen, maxLen, etc
 }
 override def getFinalStats(taskCommitTime: Long): WriteTaskStats = {
   // get the size information for all the written files
   IncrementalWriteTaskStats(0, 0)
 }
 override def closeFile(filePath: String): Unit = {
   // Process the information that current file is being closed
 }
}

class IncrementalWriteJobStatsTracker() extends WriteJobStatsTracker {
 override def newTaskInstance(): WriteTaskStatsTracker =
   new IncrementalWriteTaskStatsTracker
 override def processStats(stats: Seq[WriteTaskStats], jobCommitTime: Long): Unit = {
   // Aggregate all the writeTaskStats and update table level, partition level
   // and column level stats to the metastore
 }
}
```

We can define tracker classes as shown above.
`IncrementalWriteJobStatsTracker` would be added to all the data changing commands i.e. `InsertIntoHadoopFsRelationCommand`, `InsertIntoHiveTable`, `AlterTableDropPartitionCommand`, etc.