DATA+AI SUMMIT 2022

Spark SQL Aggregate Improvements at Meta

Shipra Agrawal

Cheng Su

About Us

- Shipra Agrawal
    - Software Engineer at Meta (Data Platform Team)
    - Worked on Spark Core & SQL
- Cheng Su
    - Software Engineer at Anyscale (Ray Data Team)
    - Apache Spark contributor (Spark SQL)
    - Previously worked on Spark, Hive & Hadoop at Meta

Agenda

- Hash aggregate
  - adaptive bypass of partial aggregate
- Object hash aggregate
  - adaptive sort-based fallback based on JVM metrics
- Sort aggregate
  - prefer sort aggregate when data is already sorted
  - code generation
- Data source aggregate
  - aggregate push down for ORC data source
  - efficient statistics collection via file footer

Agenda

- Hash aggregate
  - adaptive bypass of partial aggregate
- Object hash aggregate
  - adaptive sort-based fallback based on JVM metrics
- Sort aggregate
  - prefer sort aggregate when data is already sorted
  - code generation
- Data source aggregate
  - aggregate push down for ORC data source
  - efficient statistics collection via file footer

Hash Aggregation (Partial Aggregation - Mapper Side)

```sql
select avg(score) from student_record group by student;
```

Partition 1

student_record	
student	score
A	70
B	85
B	75
A	90
C	80
A	95
C	70

Grouping Key	Aggregation Buffer	
	Sum	Count
A	70	1
B	85	1
⋮		

Grouping Key	Aggregation Buffer	
	Sum	Count
A	255	3
B	160	2
C	150	2

Hash Aggregation (Partial Aggregation - Mapper Side)

Partition 2

student_record	
student	score
A	70
B	80
C	60
D	90
E	80
A	80
A	75

(1)

Grouping Key	Aggregation Buffer	
	Sum	Count
A	70	1
B	80	1
C	60	1

Spill to external sorter(2)

(3)

Grouping Key	Aggregation Buffer	
	Sum	Count
D	90	1
E	80	1
A	155	2

Merge Sort(4)

Grouping Key	Aggregation Buffer	
	Sum	Count
A	225	3
B	80	1
C	60	1
D	90	1
E	80	1

Shuffle (After Partial Aggregation)

Partition 1

Grouping Key	Aggregation Buffer	
	Sum	Count
A	255	3
B	160	2
C	150	2

Partition 2

Grouping Key	Aggregation Buffer	
	Sum	Count
A	225	3
B	80	1
C	60	1
D	90	1
E	80	1

Shuffle

Partition 3

Grouping Key	Aggregation Buffer	
	Sum	Count
A	255	3
B	160	2
C	150	2
A	225	3
B	80	1
C	60	1
D	90	1
E	80	1

Hash Aggregation (Final Aggregation - Reducer Side)

Partition 3

Grouping Key	Aggregation Buffer	
	Sum	Count
A	255	3
B	160	2
C	150	2
A	225	3
B	80	1
C	60	1
D	90	1
E	80	1

(1)

Grouping Key	Aggregation Buffer	
	Sum	Count
A	480	6
B	240	3
C	210	3

Spill to external sorter(2)

(3)

Grouping Key	Aggregation Buffer	
	Sum	Count
D	90	1
E	80	1

Merge Sort(4)

Grouping Key	Aggregation Buffer	
	Sum	Count
A	480	6
B	240	3
C	210	3
D	90	1
E	80	1

Grouping Key	Avg
A	80
B	80
C	70
D	90
E	80

The Problem

Aggregate reduction ratio = (Input row count - Output row count) / Input row count
2 / 7 = 0.29

Partition 2

student_record	
student	score
A	70
B	80
C	60
D	90
E	80
A	80
A	75

(1)

Grouping Key	Aggregation Buffer	
	Sum	Count
A	70	1
B	80	1
C	60	1

(3)

Grouping Key	Aggregation Buffer	
	Sum	Count
D	90	1
E	80	1
A	155	2

Spill to external sorter(2)

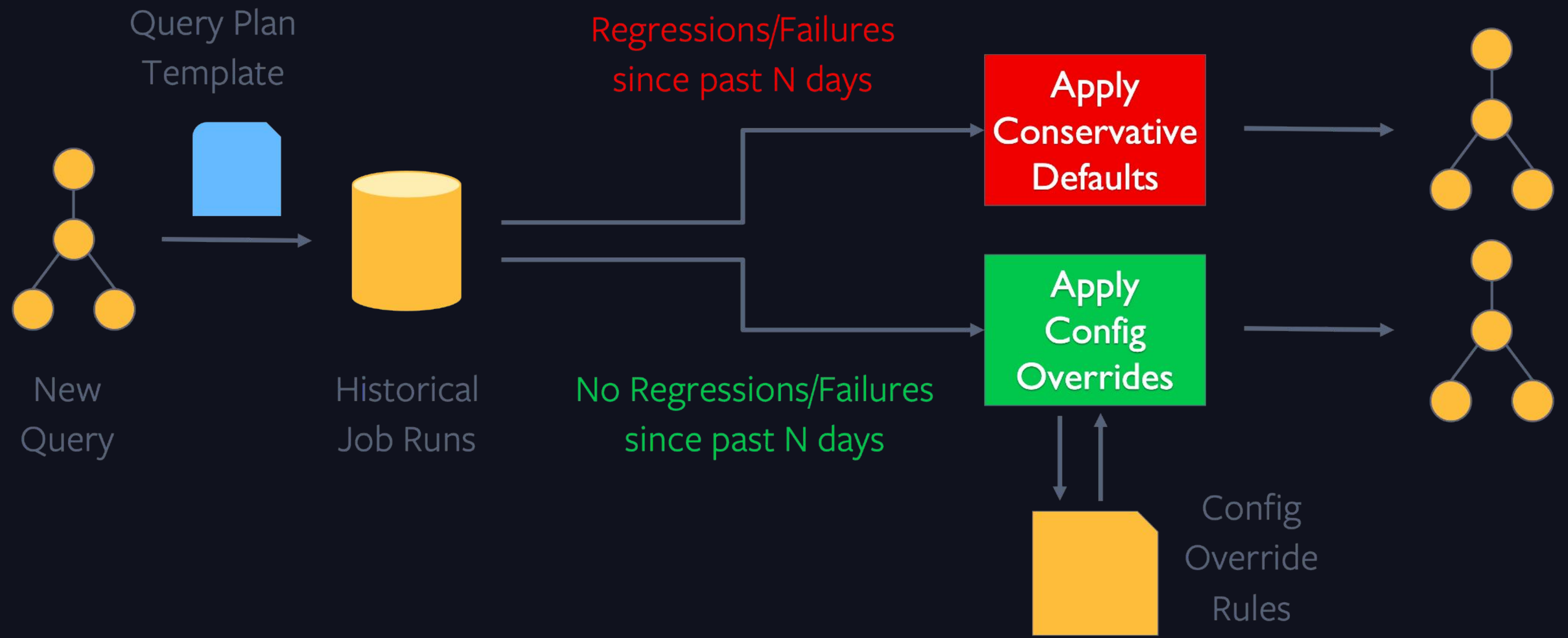
Merge Sort(4)

Grouping Key	Aggregation Buffer	
	Sum	Count
A	225	3
B	80	1
C	60	1
D	90	1
E	80	1

Potential Solutions to Skip Partial Aggregation

- ✗ Decide based on metrics from historical runs.
- ✓ Decide at runtime based on metrics of current job.

History-Based Tuning at Meta



Solution 1: History Based Tuning

- Use ***hash aggregation reduction ratio*** of historical runs.
- Several issues with this approach:
  - Historical statistics might be not available.
  - Using final aggregation ratio may be an overestimate.
  - This has to be done for all tasks in a stage.
  - Input data characteristics across runs, for eg. in case of skew. Historical metrics won't help here.

Solution 2: Runtime Decision

- Goal is to minimize both false positives and false negatives.
- Partial aggregation is skipped if reduction observed is less than 50% after processing 100,000 rows and it's incurring spill.
- Gives ability to have partial aggregation for some, but not necessarily all tasks in a stage.
  - On average, a stage skipping partial aggregation skipped it for ~ 75% of the tasks.

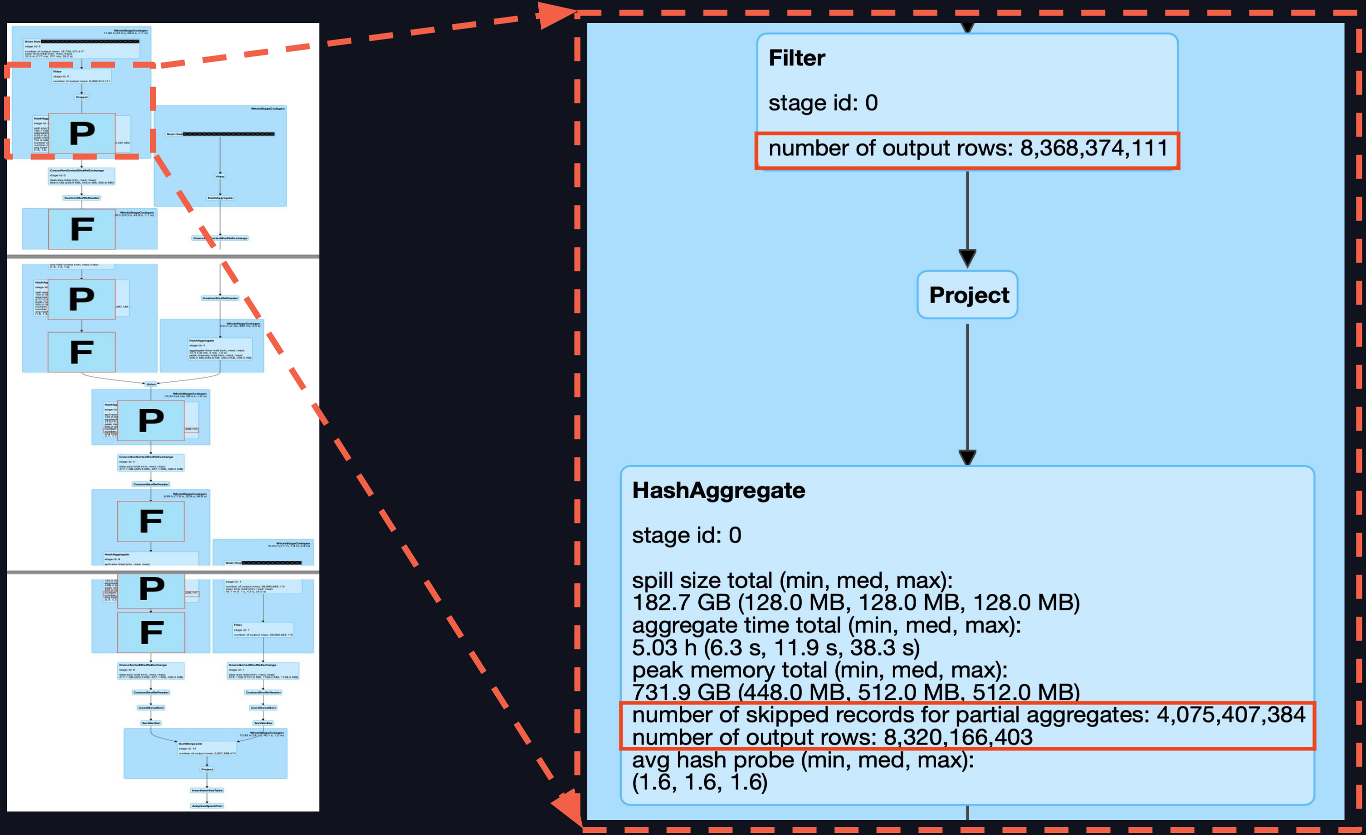
Solution 2: Runtime Decision

- Results:
  - Affected jobs contribute around 35% by CPU, 5% by count.
  - Reduction in Spill: 34%, CPU time: 9%, Reserved memory time: 12%

Example

P Partial Aggregation

F Final Aggregation



Future Work

- Handle skew by evaluating reduction ratio for each grouping key.
- Add improvement to object hash aggregation.
- Contribute back to Apache Spark.

Agenda

- Hash aggregate
  - adaptive bypass of partial aggregate
- Object hash aggregate
  - adaptive sort-based fallback based on JVM metrics
- Sort aggregate
  - prefer sort aggregate when data is already sorted
  - code generation
- Data source aggregate
  - aggregate push down for ORC data source
  - efficient statistics collection via file footer

Object Hash Aggregate

- Used in aggregate functions like *collect_list*, *percentile* etc. where each aggregation buffer can have a different size.
- Supports arbitrary-sized JVM objects as aggregation states.
- Differences from Hash aggregate:
  - Uses safe rows as aggregation buffers in an ObjectAggregationMap.
  - Spills the map after it reaches a certain entry count. (set to a very small value).
  - Sorts all the remaining input rows, while hash aggregation does this for a reduced number of rows.
- Observation: JVM heap memory underutilized at only around 20%.
- Problem: premature spilling and extra processing cost for the remaining rows.

Solution: Track Heap Memory Usage

- Solution: use JVM heap memory usage along with map entry count to decide when to spill.
- Ensure both performance and reliability.
- Configs for memory usage threshold and row count interval.
  - By fixing memory usage threshold at 70% and row count interval at 100, we limit OOMs to 5-6 jobs.
- Limitation: some JVM OOMs inevitable in cases of skew.

Improvements

- Almost always deferred spill. Spilled bytes reduced by >10%.
- Prevented spilling entirely for almost half of all Spark tasks.
- On-heap memory utilization improved from 20% to 80%. Reserved memory time reduced by >30%.
- Reduced pressure on off-heap memory reduced pre-existing off-heap OOMs.

Future Work

- Change to ‘push notification’ model for detecting memory usage threshold crossing.
- Explore replicating hash aggregate fallback mechanism to reduce number of rows being sorted.
- Contribute back to Apache Spark.

Agenda

- Hash aggregate
  - adaptive bypass of partial aggregate
- Object hash aggregate
  - adaptive sort-based fallback based on JVM metrics
- Sort aggregate
  - prefer sort aggregate when data is already sorted
  - code generation
- Data source aggregate
  - aggregate push down for ORC data source
  - efficient statistics collection via file footer

Sort Aggregate

- Local sort is needed on aggregate keys before sort aggregate.
- Process sorted data and aggregate rows with same keys.
- Differences from hash aggregate:
  - No need to maintain hash table, and so no memory spill or fallback.
  - Optimizer prefers to use hash aggregate over sort aggregate
  - No implementation for code generation

Prefer Sort Aggregate if Data Is Sorted

- Add physical plan rule (ReplaceHashWithSortAgg) to check if child of aggregate is sorted on aggregate keys. If yes, then use sort aggregate, instead of hash and object hash aggregate.
- Improve performance of aggregate when data is already sorted on keys.
  - Eliminate the cost of constructing and looking up hash table.
- The feature is merged in Spark 3.3.
- Enable this feature by setting configuration spark.sql.execution.replaceHashWithSortAgg=true.

Code Generation for Sort Aggregate

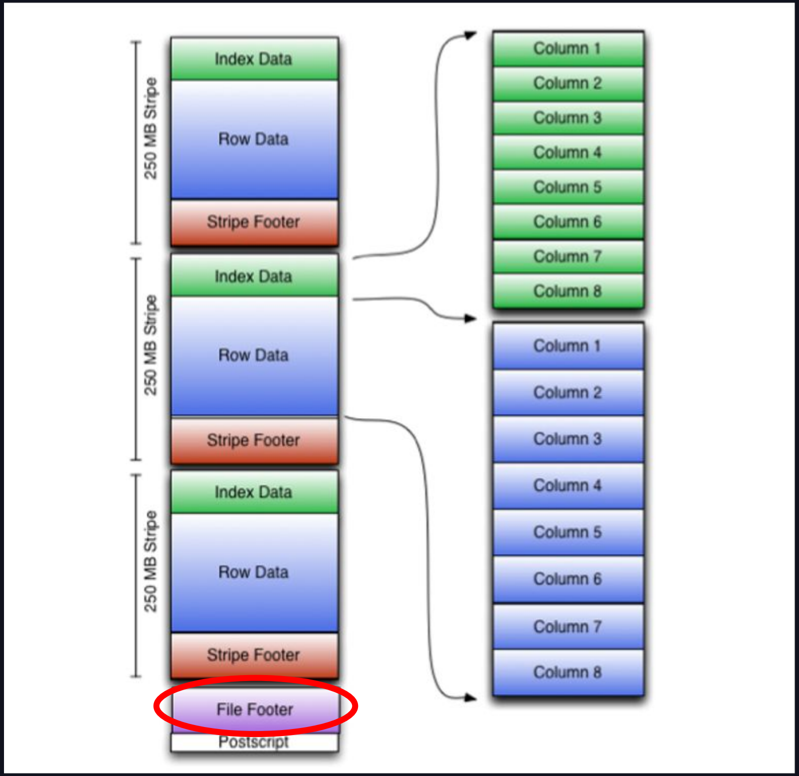
- Spark has whole stage code generation for many operators (filter, project, hash aggregate, etc), but not for sort aggregate.
- Add code generation for sort aggregate to improve performance of job.
- Code is merged in Spark 3.3 to support sort aggregate without keys.
- Future release will support sort aggregate with keys.
- The feature is enabled by default.

Agenda

- Hash aggregate
  - adaptive bypass of partial aggregate
- Object hash aggregate
  - adaptive sort-based fallback based on JVM metrics
- Sort aggregate
  - prefer sort aggregate when data is already sorted
  - code generation
- Data source aggregate
  - aggregate push down for ORC data source
  - efficient statistics collection via file footer

Background: Apache ORC

- Columns are stored separately.
- Rows are divided into multiple groups.
- File footer stores columns statistics
  - Rows count
  - Non-null values count
  - Min, max value



Aggregate Push Down for ORC Data Source

- Use file footer column statistics to short-cut aggregate processing.
- Example query: SELECT MIN(id) FROM users
    - Get min statistics for column “id” in each file footer.
    - Aggregate min statistics together.
    - No need to process actual rows in files.
- The feature is merged in Spark 3.3. Only work for Data source v2.
- Enable this feature by setting configuration spark.sql.orc.aggregatePushdown=true.

Efficient Statistics Collection via File Footer

- Partition/table statistics = Collection(files statistics for the partition/table)
- Example of partition/table statistics:
    - Rows count
    - Total files size
    - Min, max values of each column
- Accurate up-to-date partition/table statistics is useful for query optimizer to generate better query plan.
- Traditional statistics collection is a separate job to reprocess ALL rows from each file. Inefficient and hard to manage.

Efficient Statistics Collection via File Footer

- Our solution: statistics collection by only opening files footer (that's enough for ORC and Parquet!).
- Eliminate cost of reprocess actual rows in each file.
- Enforce statistics collection automatically right after inserting to table. Make sure statistics of partition/table is always accurate and up-to-date.

Thank You