DATA+AI
SUMMIT 2022

Aggregate Push Down

Spark Data Source V2 Performance Improvement

Huaxin Gao & DB Tsai
Apple Inc.

ORGANIZED BY databricks

Who are we?

Huaxin Gao

- Apache Spark PMC
- Software Engineer at Apple

DB Tsai

- Apache Spark PMC
- Engineering Manager at Apple

Query of the interest

Queries that contain aggregate functions:

```
SELECT MIN (col1), MAX(col1), COUNT(col1) FROM test
    GROUP BY col2
    WHERE col3 > 100;
```

Without aggregate push down, Spark needs to do a full scan.

Performance improvement for Push Down

J

- No need for a full scan. Only the aggregated results are returned to Spark. Save lots of network I/O and disk I/O.
- Aggregate function could execute faster.
  - SQL based (JDBC):

    data source has index support and better statistics info.
  - File based (Parquet/ORC/Iceberg):

    gets statistics info from footer or manifest files, and uses

    these statistics info to calculate Max/Min/Count.

Data Source API

Data Source API provides Spark the ability to integrate with the external data storage such as Hive, Parquet, ORC, JDBC, Iceberg, etc.

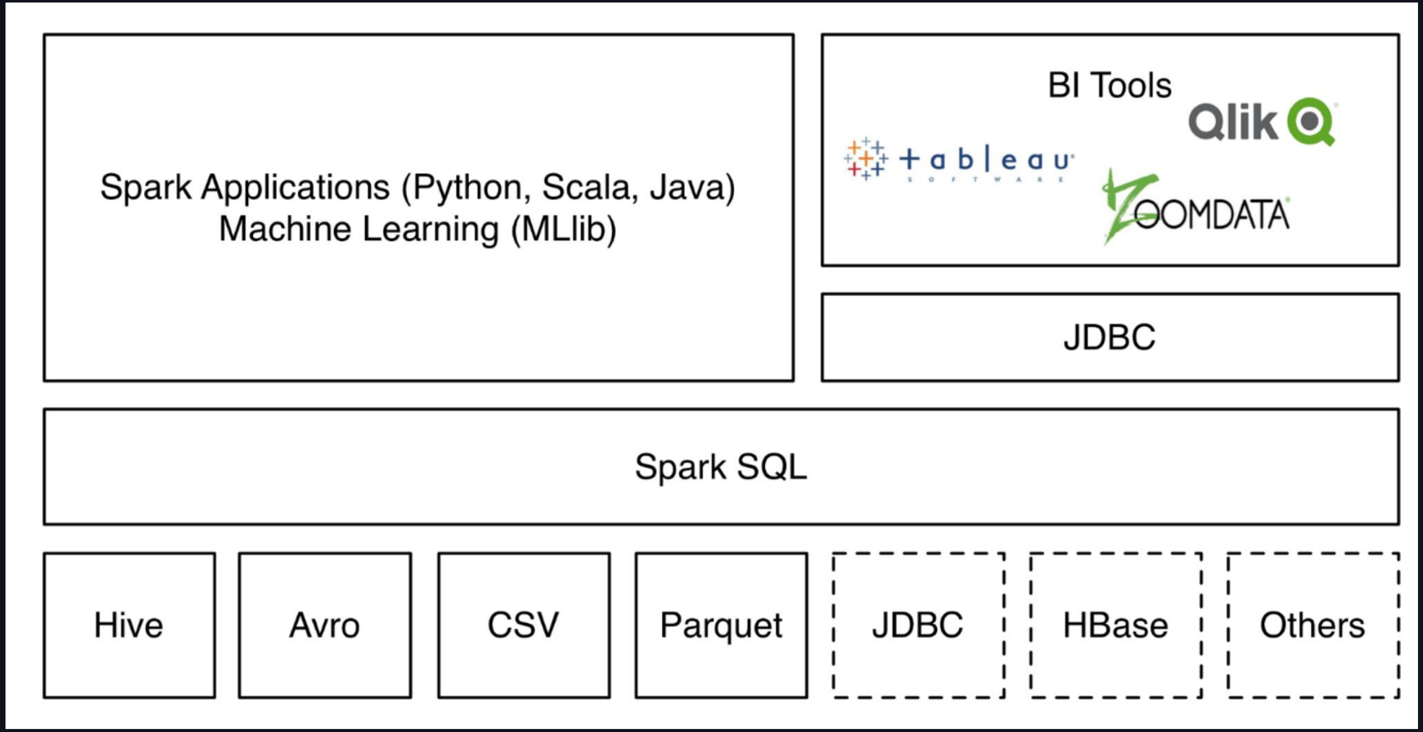


Figure cited from
https://databricks.com/blog/2015/01/09/spark-sql-data-sources-api-unified-data-access-for-the-spark-platform.html

Data Source API

- Provides a pluggable mechanism for accessing structured data from the external storage though Spark SQL
- Efficient data access powered by Spark SQL query optimizer
- Has interfaces to push down operators to data source for optimization
- Filter push down and column pruning can dramatically reduce the amount of data that need to be transferred and processed

Data Source V1 API Limitations

- Dependency on higher level API such as SQLContext and DataFrame
- Lack of support for Columnar Read and Streaming
- Lack of transactional support in write
- Hard to add new operators push down

Data Source V2 API Framework

Introduced in Spark 2.3, Data Source V2 API provides a set of java interfaces. They are located in spark.sql.connector. Some basic APIs are:

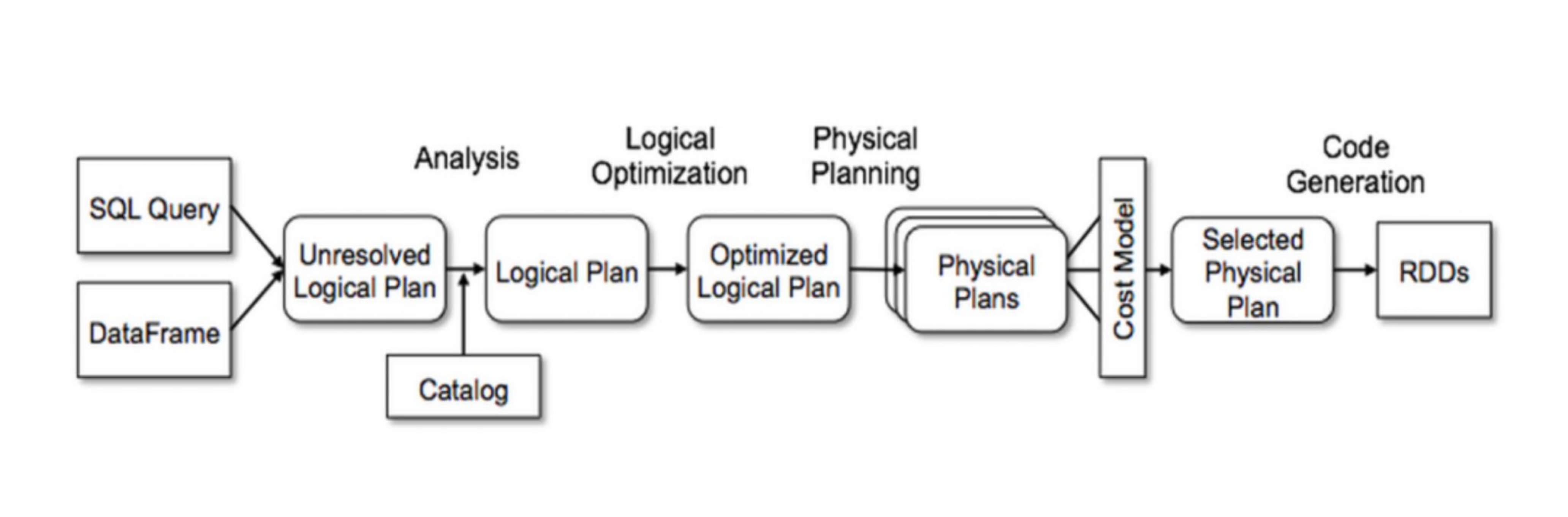
- read
  - Batch
  - Scan
  - ScanBuilder
  - SupportsPushDownFilters
  - SupportsPushDownRequiredColumn
  - SupportsPushDownAggregates
  - SupportsPushDownLimit
  - SupportsPushDownTableSample
- write
- catalog

Data Source V2 API Framework

- Spark driver:
  - determines the schema, and generates the query plans
  - creates and serializes PartitionReaderFactory, and sends to executors.
- Spark executors:
  - create PartitionReader using InputPartition
  - PartitionReader fetches data from external storage
  - process data

Spark Query Plan

- When executing a query, Spark will produce different types of plans.
- Query optimization is done at logical optimization phase. Once the Logical plan has been produced, it will be optimized based on various rules applied on logical operations.



Data Source V2 API optimization strategy

Data Source V2 has optimization rules to push down operators to external storage

New operators push down added in Spark 3.2/3.3:

- Push down aggregates to JDBC (Spark 3.2)
- Push down MIN/MAX/COUNT to Parquet (Spark 3.3)
- Push down MIN/MAX/COUNT to ORC (Spark 3.3)
- Push down Data Source V2 filter (including push down functions in V2 filter) (Spark 3.3)
- Push down Limit (Spark 3.3)
- Push down Table Sample (Spark 3.3)

Data Source V2 API optimization strategy

Data Source V2 has optimization rules to push down operators to external storage.

```
def apply(plan: LogicalPlan): LogicalPlan = {
  val pushdownRules = Seq[LogicalPlan => LogicalPlan] (
   createScanBuilder,
   pushDownSample,
   pushDownFilters,
   pushDownAggregates,
   pushDownLimits,
   pruneColumns)
  pushdownRules.foldLeft(plan) { (newPlan, pushDownRule) =>
   pushDownRule(newPlan)
  }
```

JDBC Aggregate Push Down

```
case class JDBCScanBuilder(
    session: SparkSession,
    schema: StructType,
    jdbcOptions: JDBCOptions)
  extends ScanBuilder
    with SupportsPushDownV2Filters
    with SupportsPushDownRequiredColumns
    with SupportsPushDownAggregates
    with SupportsPushDownLimit
    with SupportsPushDownTableSample
    with SupportsPushDownTopN
    with Logging
```

JDBC Aggregate Push Down

- Data Source V2 has an option pushDownAggregate. The default is false.
- If sets to true, Spark pushes down aggregates (MIN, MAX, COUNT, SUM, AVG) to the JDBC data source.
- GROUP BY can be pushed down as well.
- If there is only one partition, no final aggregate will be needed at Spark.
- If there are more than one partitions, final aggregate will be done at Spark.

JDBC Aggregate Push Down

If there is only one partition, aggregates are completely pushed down to data source. No final aggregation is needed at Spark.

```
val df = sql("SELECT * FROM h2.test.employee")
  .filter("SALARY > 1000.0")
  .groupBy($"DEPT")
  .min("SALARY")
df.explain(true)
```

JDBC Aggregate Push Down

```
== Analyzed Logical Plan ==
Aggregate [DEPT], [DEPT, min(SALARY)]
+- Filter (SALARY > 1000)
      +- RelationV2[DEPT, NAME, SALARY, BONUS, IS_MANAGER] test.employee

== Optimized Logical Plan ==
RelationV2[DEPT, MIN(SALARY)]

== Physical Plan ==
JDBCScan [DEPT,MIN(SALARY)]
  PushedAggregates: [MIN(SALARY)],
  PushedFilters: [SALARY IS NOT NULL, SALARY > 1000.00],
  PushedGroupByExpressions: [DEPT]
```

JDBC Aggregate Push Down

If there are more than one partition, aggregates are partially pushed down to data source. Final aggregation is done at Spark.

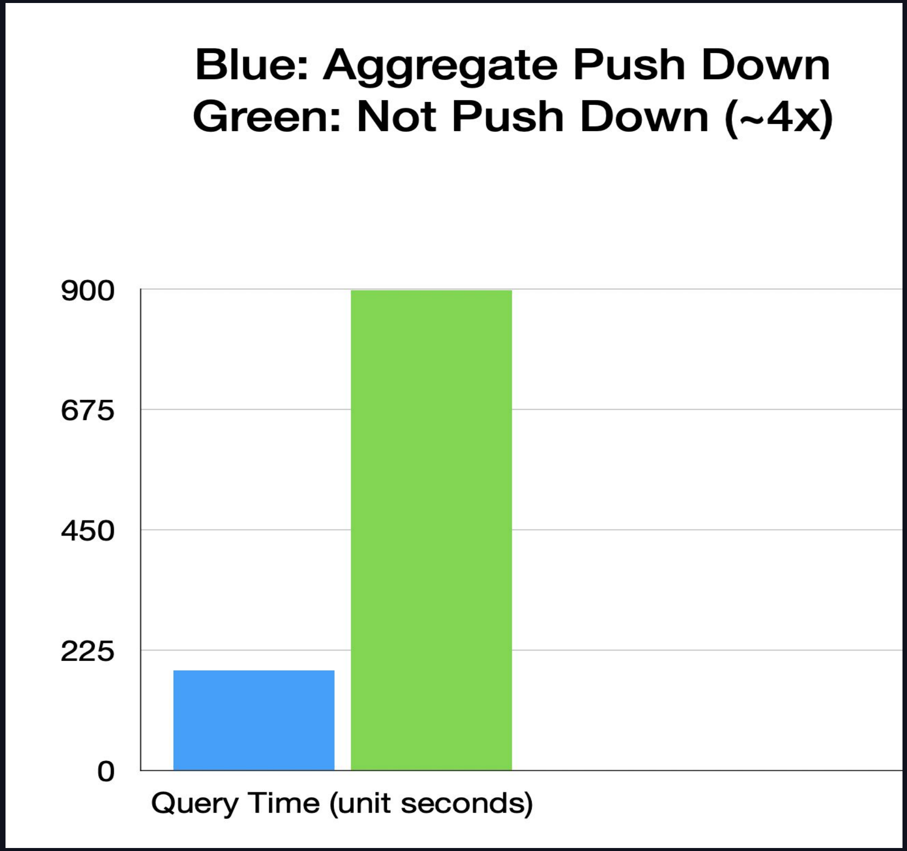
```
val df = spark.read
  .option("partitionColumn", "dept")
  .option("lowerBound", "0")
  .option("upperBound", "2")
  .option("numPartitions", "2")
  .table("h2.test.employee")
  .filter("SALARY > 1000.0")
  .agg(sum($"SALARY").as("sum"))
df.explain(true)
```

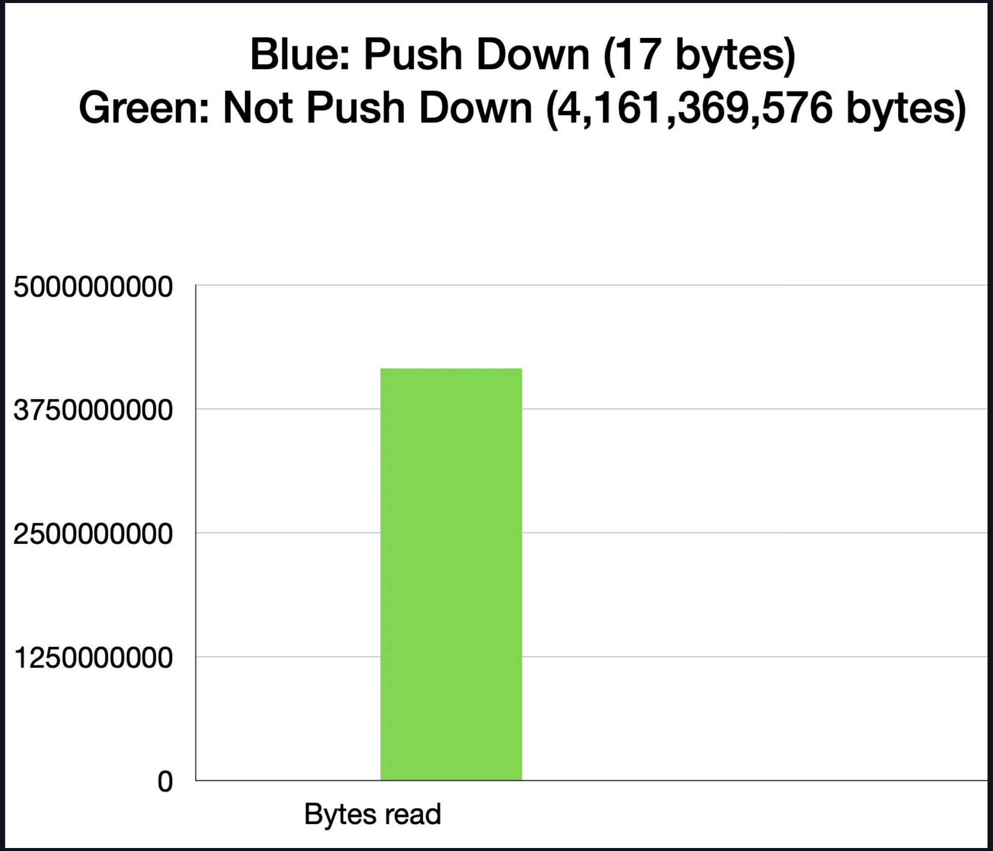
JDBC Aggregate Push Down

```
== Analyzed Logical Plan ==
Aggregate [sum]
+- Filter (SALARY > 1000.0 )
     +- RelationV2[DEPT, NAME SALARY, BONUS, IS_MANAGER]

== Optimized Logical Plan ==
Aggregate [SUM(SALARY)]
+- RelationV2[SUM(SALARY)]

== Physical Plan ==
+- HashAggregate
      +- JDBCScan [SUM(SALARY)]
      PushedAggregates: [SUM(SALARY)],
      PushedFilters: [SALARY IS NOT NULL, SALARY > 1000.00]
```

JDBC Aggregate Push Down

Benchmark results using TPCH Query 6





Parquet Aggregate Push Down

Parquet Layout

- Multiple Row groups
- Each Row group splits into column chunks
- Metadata for row group and column chunks is stored in the footer.

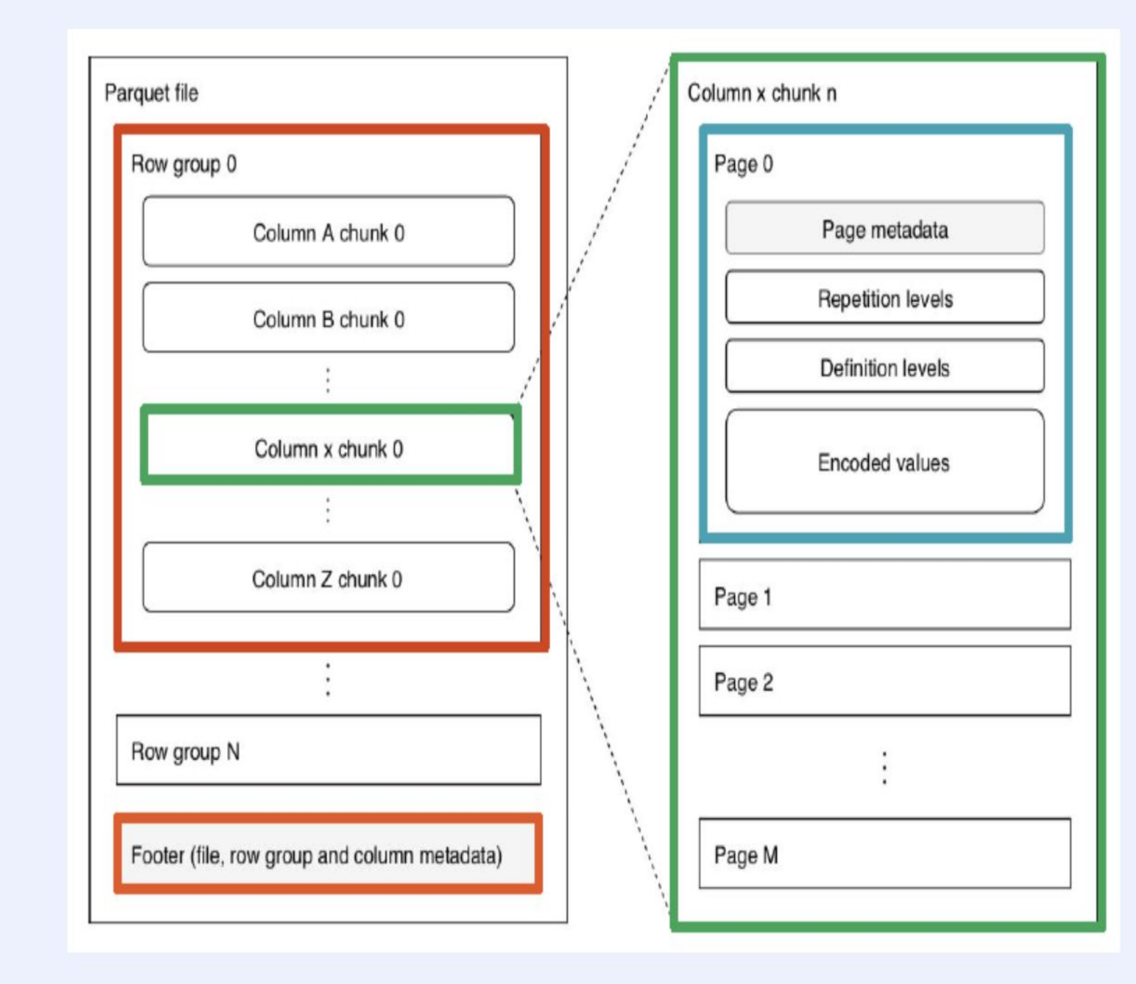
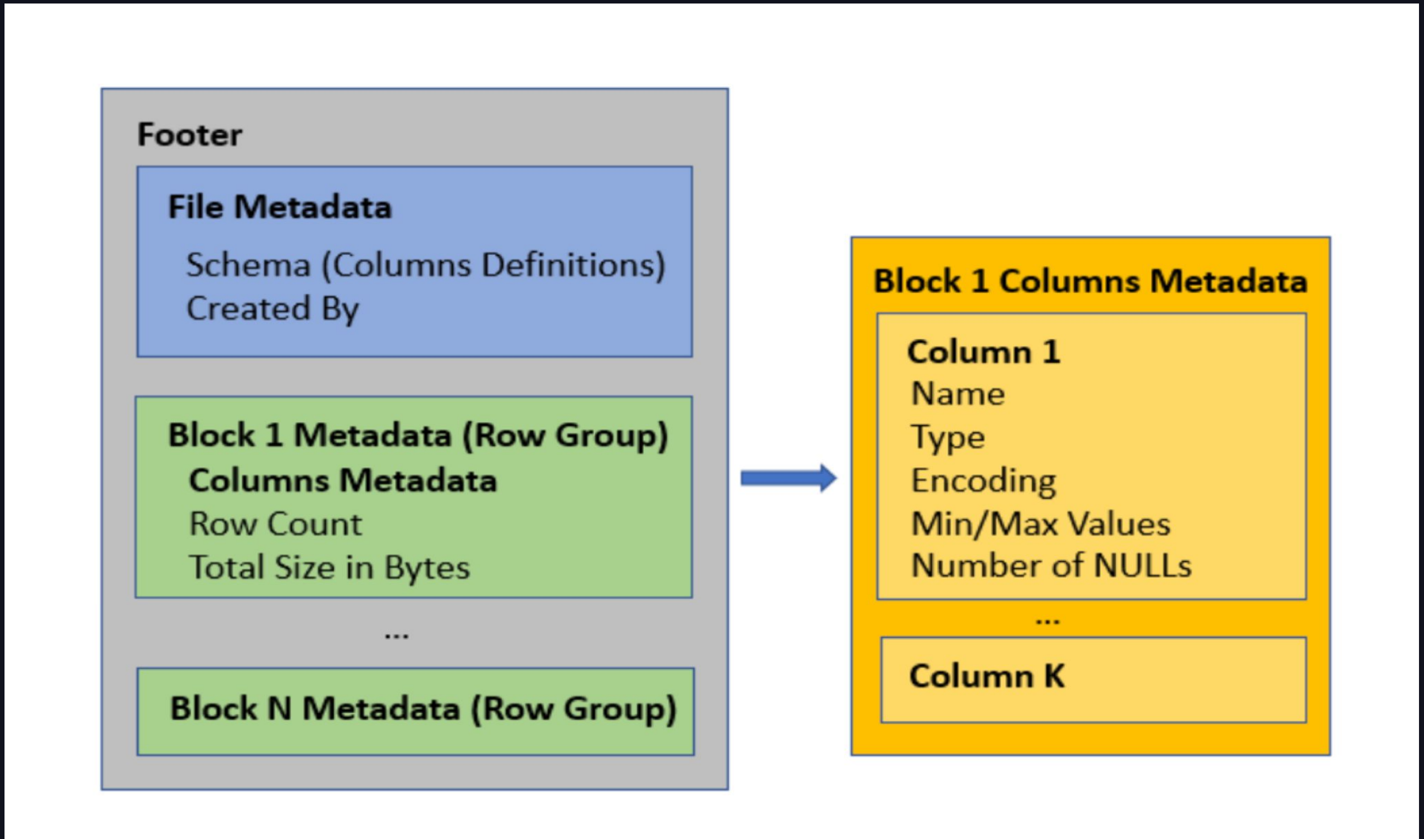


Figure cited from https://saince.io/2020/08/13/data-lakes-apache-parquet/

Parquet Aggregate Push Down

Parquet Layout

Footer contains metadata info: schema, row groups and column statistics.



Footer

File Metadata

Schema (Columns Definitions)
Created By

Block 1 Metadata (Row Group)
Columns Metadata
Row Count
Total Size in Bytes

...

Block N Metadata (Row Group)

Block 1 Columns Metadata

Column 1
Name
Type
Encoding
Min/Max Values
Number of NULLs

...

Column K

Figure cited from http://cloudsqale.com/2021/01/15/parquet-1-x-file-format-footer-content/

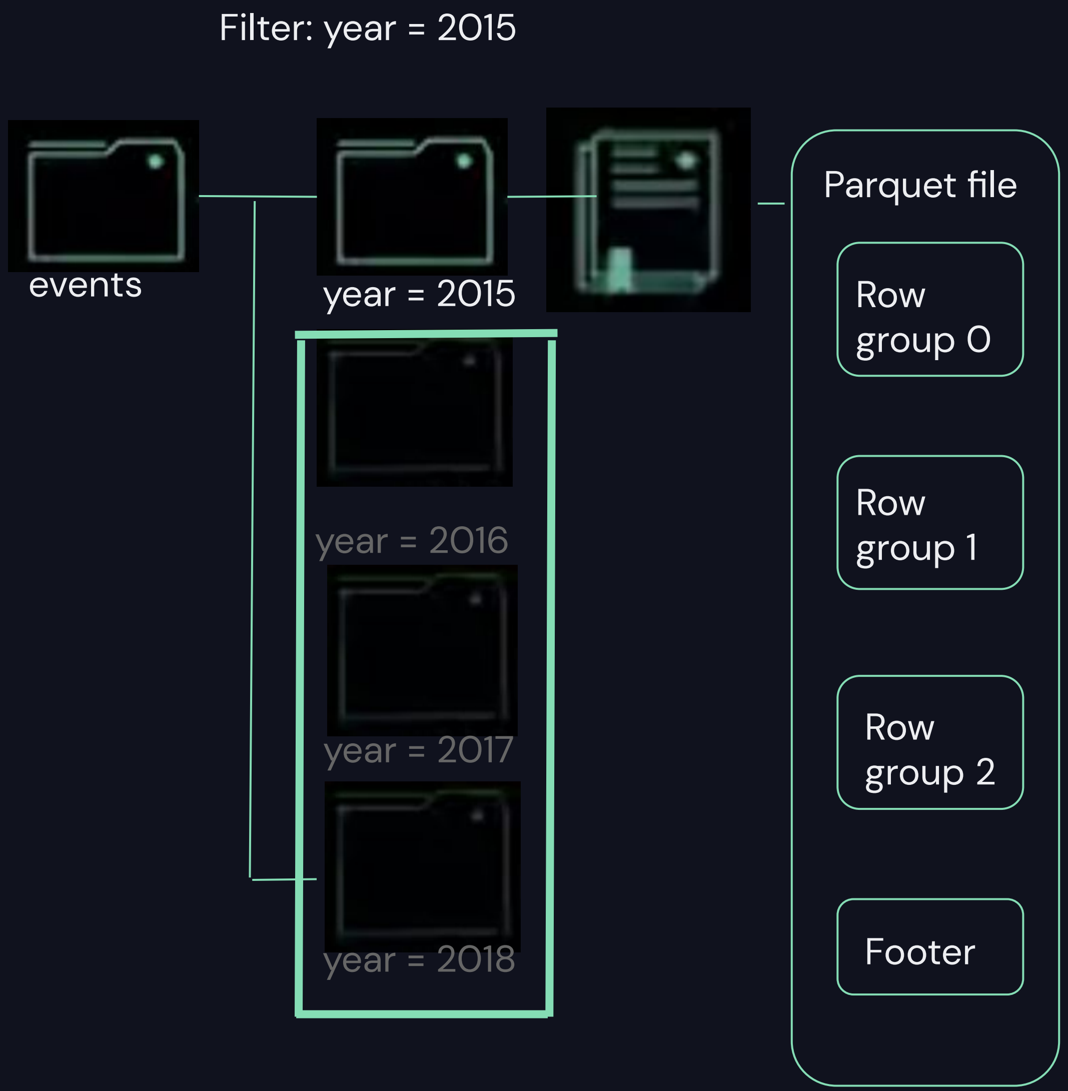
Parquet Partial Aggregate Push Down

- Property `spark.sql.parquet.aggregatePushdown`, defaults to false.
- Push down MIN/MAX/COUNT to Parquet
- Use the statistics information in Parquet Row group metadata and calculate the MIN/MAX/COUNT
- Have final aggregate at Spark

Parquet Partial Aggregate Push Down

- Aggregate with filter can be pushed down only if the filter is on partition columns.

- Aggregate with GROUP BY can be pushed down only if GROUP BY is on partition columns.



Parquet Aggregate Push Down

```
val conf = new SparkConf()
           .set(SQLConf.USE_V1_SOURCE_LIST, "")
           .set(SQLConf.PARQUET_AGGREGATE_PUSHDOWN_ENABLED.key, "true")

val agg = sql("SELECT max(id), min(id), count(id) FROM tmp WHERE p = 0")
agg.explain(true)
```

Parquet Aggregate Push Down

Before Aggregate push down

```
== Physical Plan ==
+- HashAggregate
  +- BatchScan[id]
     ReadSchema: struct<id:bigint>
```

After Aggregate push down

```
== Physical Plan ==
+- HashAggregate
     +- BatchScan[max(id), min(id), count(id)]
        ReadSchema: struct<max(id):bigint,min(id):bigint,count(id):bigint>  partitionFilter p = 0
        PushedAggregation: [MAX(id), MIN(id), COUNT(id)]
```

Parquet Aggregate Push Down

Benchmark results

- Use Spark SqlBasedBenchmark
- Create a table with a single column, insert 100M random number,

  so it has 100M rows
- Test Integer, Long and Double

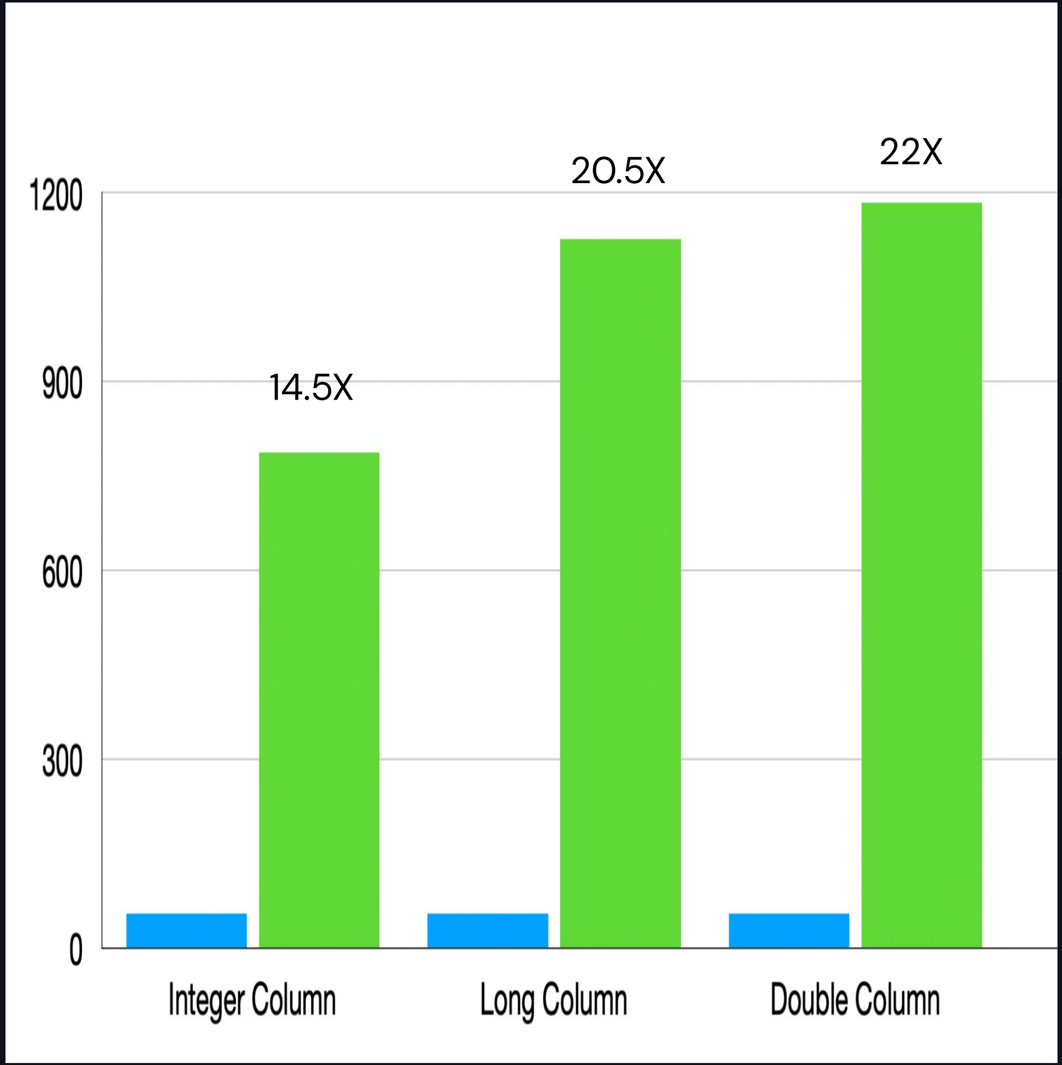
SELECT MIN(col), MAX(col), COUNT(col)

FROM tmp;

Integer Col 54ms vs. 787ms

Long Col 55ms vs. 1126ms

Double Col 54ms vs. 1185ms



Iceberg Aggregate Push Down

Take advantage of the statistics information in Iceberg manifest files and push down MIN/MAX/COUNT to Iceberg.

DATA+AI
SUMMIT 2022

Thank you

ORGANIZED BY databricks

Huaxin Gao & DB Tsai