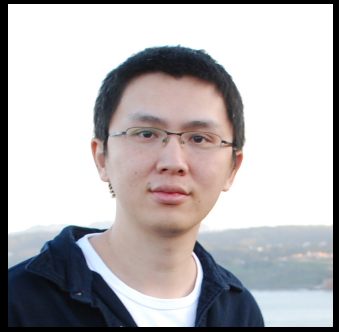
Understanding Query Plans and Spark UIs

Xiao Li @ gatorsmile

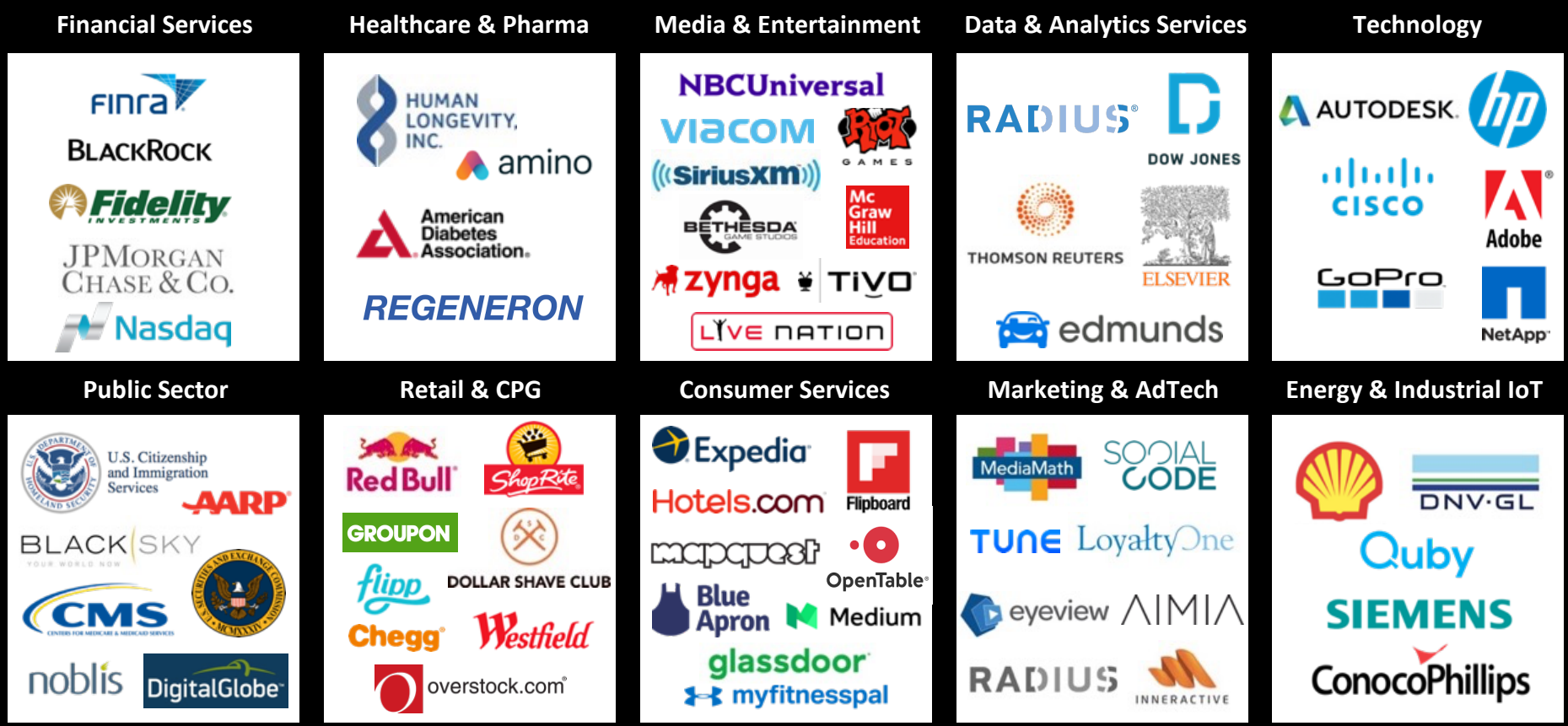
Spark + AI Summit @ SF | April 2019

About Me

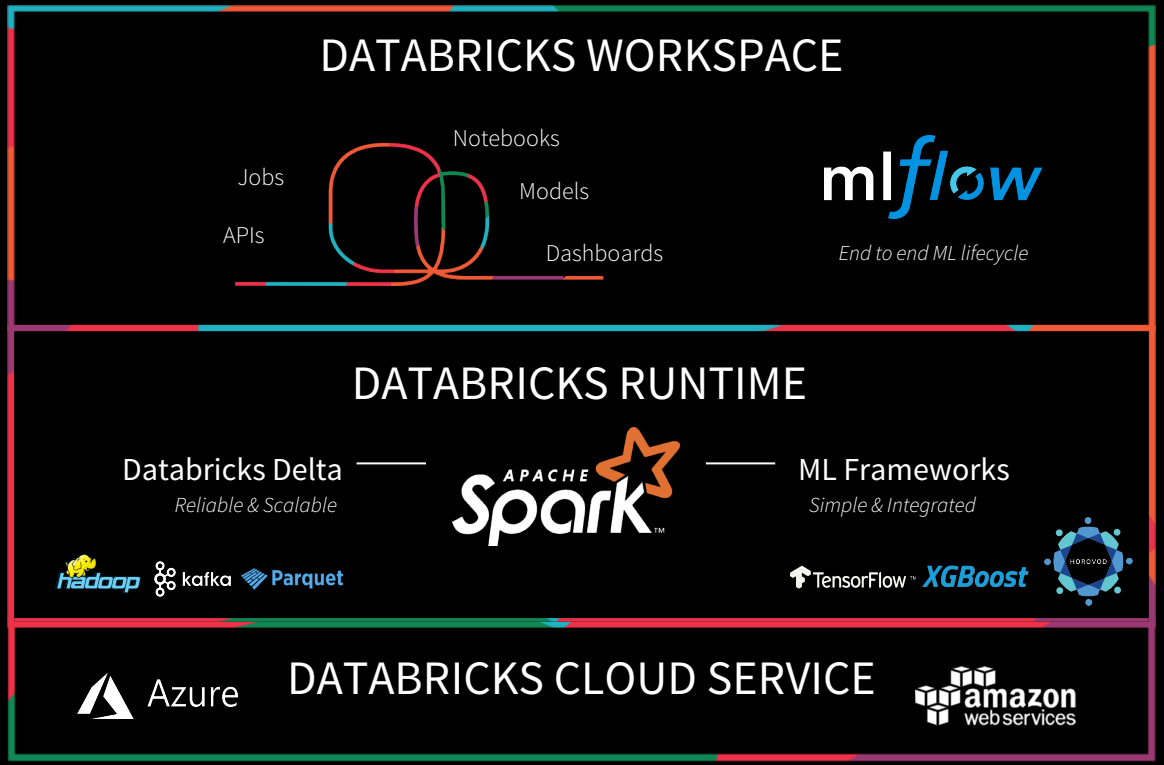


- Engineering Manager at Databricks
- Apache Spark Committer and PMC Member
- Previously, IBM Master Inventor
- Spark, Database Replication, Information Integration
- Ph.D. in University of Florida
- Github: gatorsmile

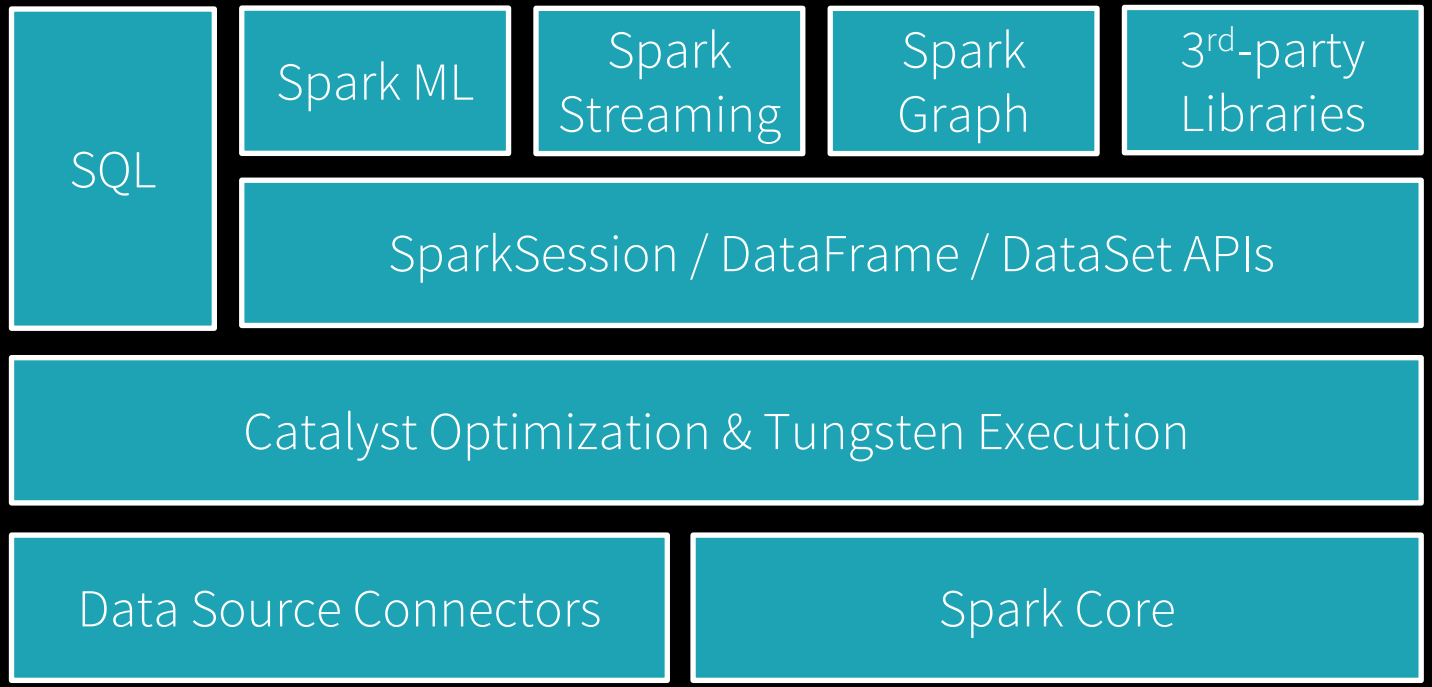
Databricks Customers Across Industries



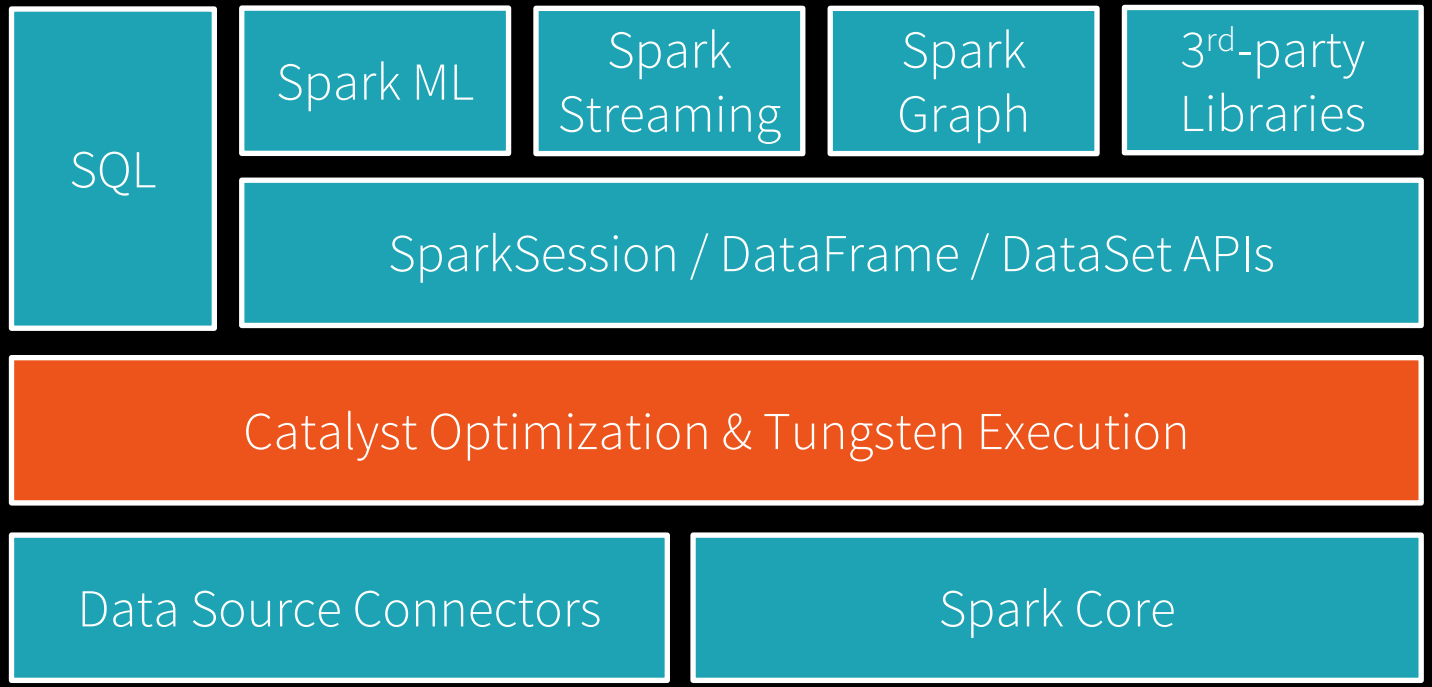
Databricks Unified Analytics Platform



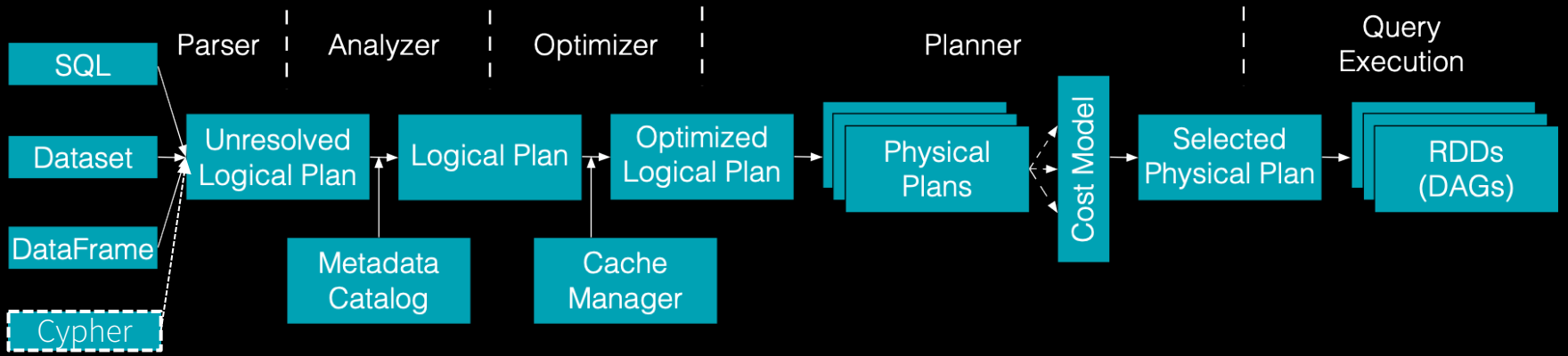
Apache Spark 3.x



Apache Spark 3.x



From declarative queries to RDDs



Maximize Performance

Read Plan.

Interpret Plan.

Tune Plan.

Track Execution.

Read Plans from SQL Tab in either Spark UI or Spark History Server

```
1  spark.sql(
2    """
3      |SELECT
4      |  sum(l_extendedprice * l_discount) AS revenue
5      |FROM
6      |  lineitem
7      |WHERE
8      |  l_shipdate >= DATE '1994-01-01'
9      |  AND l_shipdate < DATE '1994-01-01' + INTERVAL '1' YEAR
10     |  AND l_discount BETWEEN .06 - 0.01 AND .06 + 0.01
11     |  AND l_quantity < 24
12   """.stripMargin).show()
```

▸ (1) Spark Jobs

```
+------------------+
|           revenue|
+------------------+
|6.156313774170997E8|
+------------------+
```

SQL

Completed Queries: 1

Completed Queries (1)

ID	Description	Submitted	Duration	Job IDs
0	show at command-1547544212 +details	2019/03/06 06:42:03	55 s	[0]

Read Plans from SQL Tab in either Spark UI or Spark History Server

```
spark.sql(
  """
    |SELECT
    |  sum(l_extendedprice * l_discount) AS revenue
    |FROM
    |  lineitem
    |WHERE
    |  l_shipdate >= DATE '1994-01-01'
    |  AND l_shipdate < DATE '1994-01-01' + INTERVAL '1' YEAR
    |  AND l_discount BETWEEN .06 - 0.01 AND .06 + 0.01
    |  AND l_quantity < 24
  """.stripMargin).show()
```

▸ (1) Spark Jobs

```
+------------------+
|           revenue|
+------------------+
|6.156313774170997E8|
+------------------+
```

SQL

Completed Queries: 1

Spark 3.0: Show the actual SQL statement? [SPARK-27045]

Completed Queries (1)

ID	Description	Submitted	Duration	Job IDs
0	spark.sql(""" |SELECT | sum(l_exten... +details	2019/04/20 16:35:59	35 s	[0]

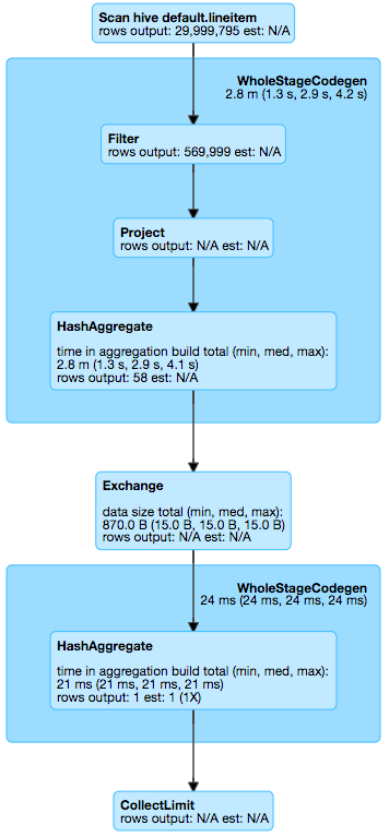
databricks

Page: In Details for SQL Query

Jobs Stages Storage Environment Executors SQL JDBC/ODBC S

Details for Query 0

Submitted Time: 2019/03/06 06:42:03
Duration: 55 s
Succeeded Jobs: 0



Details

```
== Parsed Logical Plan ==
```

Details

```
== Parsed Logical Plan ==
GlobalLimit 21
+- LocalLimit 21
   +- Project [cast(revenue#2 as string) AS revenue#22]
      +- Aggregate [sum((l_extendedprice#8 * l_discount#9)) AS revenue#2]
         +- Filter (((l_shipdate#13 >= cast(8766 as string)) && (l_shipdate#13 < cast(cast(cast(8766 as timestamp) + interval 1 years as date) as string))) && (((l_discount#9 >=
cast(CheckOverflow((promote_precision(cast(0.06 as decimal(3,2))) - promote_precision(cast(0.01 as decimal(3,2)))), DecimalType(3,2)) as double)) && (l_discount#9 <=
cast(CheckOverflow((promote_precision(cast(0.06 as decimal(3,2))) + promote_precision(cast(0.01 as decimal(3,2)))), DecimalType(3,2)) as double))) && (l_quantity#7 < cast(24 as
double))))
            +- SubqueryAlias `default`.`lineitem`
               +- HiveTableRelation `default`.`lineitem`, org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe, [l_orderkey#3, l_partkey#4, l_suppkey#5, l_linenumber#6, l_quantity#7,
l_extendedprice#8, l_discount#9, l_tax#10, l_returnflag#11, l_linestatus#12, l_shipdate#13, l_commitdate#14, l_receiptdate#15, l_shipinstruct#16, l_shipmode#17, l_comment#18]

== Analyzed Logical Plan ==
revenue: string
GlobalLimit 21
+- LocalLimit 21
   +- Project [cast(revenue#2 as string) AS revenue#22]
      +- Aggregate [sum((l_extendedprice#8 * l_discount#9)) AS revenue#2]
         +- Filter (((l_shipdate#13 >= cast(8766 as string)) && (l_shipdate#13 < cast(cast(cast(8766 as timestamp) + interval 1 years as date) as string))) && (((l_discount#9 >=
cast(CheckOverflow((promote_precision(cast(0.06 as decimal(3,2))) - promote_precision(cast(0.01 as decimal(3,2)))), DecimalType(3,2)) as double)) && (l_discount#9 <=
cast(CheckOverflow((promote_precision(cast(0.06 as decimal(3,2))) + promote_precision(cast(0.01 as decimal(3,2)))), DecimalType(3,2)) as double))) && (l_quantity#7 < cast(24 as
double))))
            +- SubqueryAlias `default`.`lineitem`
               +- HiveTableRelation `default`.`lineitem`, org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe, [l_orderkey#3, l_partkey#4, l_suppkey#5, l_linenumber#6, l_quantity#7,
l_extendedprice#8, l_discount#9, l_tax#10, l_returnflag#11, l_linestatus#12, l_shipdate#13, l_commitdate#14, l_receiptdate#15, l_shipinstruct#16, l_shipmode#17, l_comment#18]

== Optimized Logical Plan ==
GlobalLimit 21
+- LocalLimit 21
   +- Aggregate [cast(sum((l_extendedprice#8 * l_discount#9)) as string) AS revenue#22], 21
      +- Project [l_extendedprice#8, l_discount#9]
         +- Filter ((((((isnotnull(l_discount#9) && isnotnull(l_shipdate#13)) && isnotnull(l_quantity#7)) && (l_shipdate#13 >= 1994-01-01)) && (l_shipdate#13 < 1995-01-01)) &&
(l_discount#9 >= 0.05)) && (l_discount#9 <= 0.07)) && (l_quantity#7 < 24.0))
            +- HiveTableRelation `default`.`lineitem`, org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe, [l_orderkey#3, l_partkey#4, l_suppkey#5, l_linenumber#6, l_quantity#7,
l_extendedprice#8, l_discount#9, l_tax#10, l_returnflag#11, l_linestatus#12, l_shipdate#13, l_commitdate#14, l_receiptdate#15, l_shipinstruct#16, l_shipmode#17, l_comment#18]

== Physical Plan ==
CollectLimit 21
+- *(2) HashAggregate(keys=[], functions=[finalmerge_sum(merge sum#25) AS sum((l_extendedprice#8 * l_discount#9))#19], output=[revenue#22])
   +- Exchange SinglePartition
      +- *(1) HashAggregate(keys=[], functions=[partial_sum((l_extendedprice#8 * l_discount#9)) AS sum#25], output=[sum#25])
         +- *(1) Project [l_extendedprice#8, l_discount#9]
            +- *(1) Filter ((((((isnotnull(l_discount#9) && isnotnull(l_shipdate#13)) && isnotnull(l_quantity#7)) && (l_shipdate#13 >= 1994-01-01)) && (l_shipdate#13 < 1995-01-01)) &&
(l_discount#9 >= 0.05)) && (l_discount#9 <= 0.07)) && (l_quantity#7 < 24.0))
               +- Scan hive default.lineitem [l_discount#9, l_extendedprice#8, l_quantity#7, l_shipdate#13], HiveTableRelation `default`.`lineitem`,
org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe, [l_orderkey#3, l_partkey#4, l_suppkey#5, l_linenumber#6, l_quantity#7, l_extendedprice#8, l_discount#9, l_tax#10, l_returnflag#11,
l_linestatus#12, l_shipdate#13, l_commitdate#14, l_receiptdate#15, l_shipinstruct#16, l_shipmode#17, l_comment#18]
```

Parsed Plan

Analyzed Plan

Optimized Plan

Physical Plan

▾Details

```
== Parsed Logical Plan ==
GlobalLimit 21
+- LocalLimit 21
   +- Project [cast(revenue#2 as string) AS revenue#22]
      +- Aggregate [sum((l_extendedprice#8 * l_discount#9)) AS revenue#2]
         +- Filter (((l_shipdate#13 >= cast(8766 as string)) && (l_shipdate#13 < cast(cast(cast(8766 as timestamp) + interval 1 years as date) as string))) && (((l_discount#9 >=
cast(CheckOverflow((promote_precision(cast(0.06 as decimal(3,2))) - promote_precision(cast(0.01 as decimal(3,2)))), DecimalType(3,2)) as double)) && (l_discount#9 <=
cast(CheckOverflow((promote_precision(cast(0.06 as decimal(3,2))) + promote_precision(cast(0.01 as decimal(3,2)))), DecimalType(3,2)) as double))) && (l_quantity#7 < cast(24 as
double))))
            +- SubqueryAlias `default`.`lineitem`
               +- HiveTableRelation `default`.`lineitem`, org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe, [l_orderkey#3, l_partkey#4, l_suppkey#5, l_linenumber#6, l_quantity#7,
l_extendedprice#8, l_discount#9, l_tax#10, l_returnflag#11, l_linestatus#12, l_shipdate#13, l_commitdate#14, l_receiptdate#15, l_shipinstruct#16, l_shipmode#17, l_comment#18]
```

```
== Analyzed Logical Plan ==
revenue: string
GlobalLimit 21
+- LocalLimit 21
   +- Project [cast(revenue#2 as string) AS revenue#22]
      +- Aggregate [sum((l_extendedprice#8 * l_discount#9)) AS revenue#2]
         +- Filter (((l_shipdate#13 >= cast(8766 as string)) && (l_shipdate#13 < cast(cast(cast(8766 as timestamp) + interval 1 years as date) as string))) && (((l_discount#9 >=
cast(CheckOverflow((promote_precision(cast(0.06 as decimal(3,2))) - promote_precision(cast(0.01 as decimal(3,2)))), DecimalType(3,2)) as double)) && (l_discount#9 <=
cast(CheckOverflow((promote_precision(cast(0.06 as decimal(3,2))) + promote_precision(cast(0.01 as decimal(3,2)))), DecimalType(3,2)) as double))) && (l_quantity#7 < cast(24 as
double))))
            +- SubqueryAlias `default`.`lineitem`
               +- HiveTableRelation `default`.`lineitem`, org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe, [l_orderkey#3, l_partkey#4, l_suppkey#5, l_linenumber#6, l_quantity#7,
l_extendedprice#8, l_discount#9, l_tax#10, l_returnflag#11, l_linestatus#12, l_shipdate#13, l_commitdate#14, l_receiptdate#15, l_shipinstruct#16, l_shipmode#17, l_comment#18]
```

```
== Optimized Logical Plan ==
GlobalLimit 21
+- LocalLimit 21
   +- Aggregate [cast(sum((l_extendedprice#8 * l_discount#9)) as string) AS revenue#22], 21
      +- Project [l_extendedprice#8, l_discount#9]
         +- Filter (((((((isnotnull(l_discount#9) && isnotnull(l_shipdate#13)) && isnotnull(l_quantity#7)) && (l_shipdate#13 >= 1994-01-01)) && (l_shipdate#13 < 1995-01-01)) &&
(l_discount#9 >= 0.05)) && (l_discount#9 <= 0.07)) && (l_quantity#7 < 24.0))
            +- HiveTableRelation `default`.`lineitem`, org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe, [l_orderkey#3, l_partkey#4, l_suppkey#5, l_linenumber#6, l_quantity#7,
l_extendedprice#8, l_discount#9, l_tax#10, l_returnflag#11, l_linestatus#12, l_shipdate#13, l_commitdate#14, l_receiptdate#15, l_shipinstruct#16, l_shipmode#17, l_comment#18]
```

```
== Physical Plan ==
CollectLimit 21
+- *(2) HashAggregate(keys=[], functions=[finalmerge_sum(merge sum#25) AS sum((l_extendedprice#8 * l_discount#9))#19], output=[revenue#22])
   +- Exchange SinglePartition
      +- *(1) HashAggregate(keys=[], functions=[partial_sum((l_extendedprice#8 * l_discount#9)) AS sum#25], output=[sum#25])
         +- *(1) Project [l_extendedprice#8, l_discount#9]
            +- *(1) Filter (((((((isnotnull(l_discount#9) && isnotnull(l_shipdate#13)) && isnotnull(l_quantity#7)) && (l_shipdate#13 >= 1994-01-01)) && (l_shipdate#13 < 1995-01-01)) &&
(l_discount#9 >= 0.05)) && (l_discount#9 <= 0.07)) && (l_quantity#7 < 24.0))
               +- Scan hive default.lineitem [l_discount#9, l_extendedprice#8, l_quantity#7, l_shipdate#13], HiveTableRelation `default`.`lineitem`,
org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe, [l_orderkey#3, l_partkey#4, l_suppkey#5, l_linenumber#6, l_quantity#7, l_extendedprice#8, l_discount#9, l_tax#10, l_returnflag#11,
l_linestatus#12, l_shipdate#13, l_commitdate#14, l_receiptdate#15, l_shipinstruct#16, l_shipmode#17, l_comment#18]
```

Understand and Tune Plans

Cmd 1

```
1  %sql
2
3  CREATE TEMPORARY VIEW tempView3 AS SELECT * FROM VALUES
4    ("one", "2.3"), ("two", "0.35")
5    AS t1(col1, col2);
6
7  SELECT * FROM tempView3 WHERE col2 != 0.0;
```

col1	col2
one	2.3
two	0.35

Cmd 1

```
1  %sql
2
3  CREATE TEMPORARY VIEW tempView3 AS SELECT * FROM VALUES
4    ("one", "2.3"), ("two", "0.35")
5    AS t1(col1, col2);
6
7  SELECT * FROM tempView3 WHERE col2 != 0;
```

col1	col2
one	2.3

Different Results!!!

databricks

Read the analyzed plan to check the implicit type casting.

Tip:

Explicitly cast the types in the queries.

Cmd 3

```
1  %sql
2
3  EXPLAIN EXTENDED SELECT * FROM tempView3 WHERE col2 != 0.0;
```

plan

```
== Parsed Logical Plan ==
'Project [*]
+- 'Filter NOT ('col2 = 0.0)
   +- 'UnresolvedRelation `tempView3`

== Analyzed Logical Plan ==
col1: string, col2: string
Project [col1#435840, col2#435841]
+- Filter NOT (cast(col2#435841 as double) = cast(0.0 as double))
   +- SubqueryAlias `tempview3`
      +- Project [col1#435840, col2#435841]
         +- SubqueryAlias `t1`
            +- LocalRelation [col1#435840, col2#435841]

== Optimized Logical Plan ==
LocalRelation [col1#435840, col2#435841]

== Physical Plan ==
LocalTableScan [col1#435840, col2#435841]
```

Read the analyzed plan to check the implicit type casting.

Tip:

Explicitly cast the types in the queries.

Cmd 3

```
1  %sql
2
3  EXPLAIN EXTENDED SELECT * FROM tempView3 WHERE col2 != 0;
```

plan

```
== Parsed Logical Plan ==
'Project [*]
+- 'Filter NOT ('col2 = 0)
   +- 'UnresolvedRelation `tempView3`

== Analyzed Logical Plan ==
col1: string, col2: string
Project [col1#435840, col2#435841]
+- Filter NOT (cast(col2#435841 as int) = 0)
   +- SubqueryAlias `tempview3`
      +- Project [col1#435840, col2#435841]
         +- SubqueryAlias `t1`
            +- LocalRelation [col1#435840, col2#435841]

== Optimized Logical Plan ==
LocalRelation [col1#435840, col2#435841]

== Physical Plan ==
LocalTableScan [col1#435840, col2#435841]
```

Create Hive Tables

Cmd 2

Syntax to create a Hive Serde table

```
%sql

CREATE TABLE tabDemo1 (col1 int, col2 Timestamp)
PARTITIONED BY (col3 int)
STORED AS ORC;
```

Read Tables

Cmd 4

```
1  %sql
2
3  SET spark.sql.hive.convertMetastoreOrc = false;
4
5  EXPLAIN SELECT * FROM tabDemo1 WHERE col1 > 3;
```

plan

```
== Physical Plan ==
*(1) Filter (isnotnull(col1#454790) && (col1#454790 > 3))
+- Scan hive default.tabdemo1 [col1#454790, col2#454791, col3#454792], HiveTableRelation
`default`.`tabdemo1`, org.apache.hadoop.hive.ql.io.orc.OrcSerde, [col1#454790, col2#454791],
[col3#454792]
```

Hive serde reader

Cmd 3

```
1  %sql
2
3  SET spark.sql.hive.convertMetastoreOrc = true;
4
5  EXPLAIN SELECT * FROM tabDemo1 WHERE col1 > 3;
```

Native reader/writer performs faster than Hive serde reader/writer

plan

```
== Physical Plan ==
*(1) Project [col1#454775, col2#454776, col3#454777]
+- *(1) Filter (isnotnull(col1#454775) && (col1#454775 > 3))
   +- *(1) FileScan orc default.tabdemo1[col1#454775,col2#454776,col3#454777] Batched: true,
DataFilters: [isnotnull(col1#454775), (col1#454775 > 3)], Format: ORC, Location:
CatalogFileIndex[dbfs:/user/hive/warehouse/tabdemo1], PartitionCount: 0, PartitionFilters: [],
PushedFilters: [IsNotNull(col1), GreaterThan(col1,3)], ReadSchema:
struct<col1:int,col2:timestamp>
```

filter pushdown

Create Native Tables

Tip:

Create native data source tables for better performance and stability.

Cmd 2

Syntax to create a Spark native ORC table

```
1  %sql
2
3  CREATE TABLE tabDemo2 (col1 int, col2 Timestamp, col3 int)
4  USING ORC
5  PARTITIONED BY (col3);
6
7  EXPLAIN SELECT * FROM tabDemo2 WHERE col1 > 3;
```

plan

```
== Physical Plan ==
*(1) Project [col1#454829, col2#454830, col3#454831]
+- *(1) Filter (isnotnull(col1#454829) && (col1#454829 > 3))
   +- *(1) FileScan orc default.tabdemo2[col1#454829,col2#454830,col3#454831] Batched: true,
DataFilters: [isnotnull(col1#454829), (col1#454829 > 3)], Format: ORC, Location:
CatalogFileIndex[dbfs:/user/hive/warehouse/tabdemo2], PartitionCount: 0, PartitionFilters: [],
PushedFilters: [IsNotNull(col1), GreaterThan(col1,3)], ReadSchema:
struct<col1:int,col2:timestamp>
```

Push Down + Implicit Type Casting

Tip:

Cast is needed?

Update the constants?

Cmd 3

```
1  %sql
2
3  EXPLAIN SELECT * FROM tabDemo2 WHERE col1 > 3.0;
```

plan

```
== Physical Plan ==
*(1) Project [col1#454829, col2#454830, col3#454831]
+- *(1) Filter (isnotnull(col1#454829) && (cast(col1#454829 as bigint) > 3))
   +- *(1) FileScan orc default.tabdemo2[col1#454829,col2#454830,col3#454831] Batched: true,
DataFilters: [isnotnull(col1#454829), (cast(col1#454829 as bigint) > 3)], Format: ORC, Location:
CatalogFileIndex[dbfs:/user/hive/warehouse/tabdemo2], PartitionCount: 0, PartitionFilters: [],
PushedFilters: [IsNotNull(col1)], ReadSchema: struct<col1:int,col2:timestamp>
```

Not pushed down???

Nested Schema Pruning

Cmd 4

```
1  import org.apache.spark.sql.functions.struct
2
3  spark.range(10)
4    .select(
5      $"id",
6      struct(($"id" + $"id") as "c", ($"id" * $"id") as "d") as "a")
7    .write
8    .format("parquet")
9    .mode("overwrite")
10   .saveAsTable("nestedTab1")
11
12 spark.conf.set("spark.sql.optimizer.nestedSchemaPruning.enabled", false)
13
14 spark.table("nestedTab1").select($"a.c").explain()
```

▸ (1) Spark Jobs

```
== Physical Plan ==
*(1) Project [a#455352.c AS c#455359L]
+- *(1) FileScan parquet default.nestedtab1[a#455352,...batched: true, DataFilters: [], Format: Parquet, Location: InMemoryFileIndex[dbfs:/user/hive/warehouse/nestedtab1], PartitionFilters: [], PushedFilters: [], ReadSchema: struct<a:struct<c:bigint,d:bigint>>
```

#!%?

Not pruned???

Nested Schema Pruning

Cmd 4

```
1  import org.apache.spark.sql.functions.struct
2
3  spark.range(10)
4    .select(
5      $"id",
6      struct(($"id" + $"id") as "c", ($"id" * $"id") as "d") as "a")
7    .write
8    .format("parquet")
9    .mode("overwrite")
10   .saveAsTable("nestedTab1")
11
12 spark.conf.set("spark.sql.optimizer.nestedSchemaPruning.enabled", true)
13
14 spark.table("nestedTab1").select($"a.c").explain()
```

▸ (1) Spark Jobs

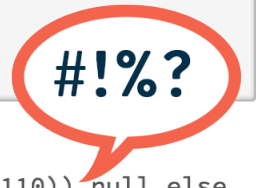
```
== Physical Plan ==
*(1) Project [a#455316.c AS c#455323L]
+- *(1) FileScan parquet default.nestedtab1[a#455316] Batched: true, DataFilters: [], Format: Parquet, L
ocation: InMemoryFileIndex[dbfs:/user/hive/warehouse/nestedtab1], PartitionFilters: [], PushedFilters:
[], ReadSchema: struct<a:struct<c:bigint>>
```

Collapse Projects

Cmd 10

```
1  import org.apache.spark.sql.functions.udf
2
3  val udf1 = udf((value: Int) => if (value > 0) value * value else value)
4
5  spark.table("tabDemo2")
6    .select(udf1($"col1").as("newCol1"))
7    .selectExpr("newCol1 + newCol1", "newCol1")
8    .explain()
```

Call UDF three times!!!




```
== Physical Plan ==
*(1) Project [(if (isnull(col1#110)) null else SpecializedUDF(col1#110) + if (isnull(col1#110)) null else
SpecializedUDF(col1#110)) AS (newCol1 + newCol1)#197, if (isnull(col1#110)) null else SpecializedUDF(col1#
110) AS newCol1#195]
+- *(1) FileScan orc default.tabdemo2[col1#110,col3#112] Batched: true, DataFilters: [], Format: ORC, Loca
tion: CatalogFileIndex[dbfs:/user/hive/warehouse/tabdemo2], PartitionCount: 0, PartitionFilters: [], Pushe
dFilters: [], ReadSchema: struct<col1:int>
import org.apache.spark.sql.functions.udf
udf1: org.apache.spark.sql.expressions.UserDefinedFunction = UserDefinedFunction(<function1>,IntegerType,S
ome(List(IntegerType)))
```

Collapse Projects

Cmd 11

```
1  import org.apache.spark.sql.functions.udf
2
3  val udf1 = udf((value: Int) => if (value > 0) value * value else value).asNondeterministic
4
5  spark.table("tabDemo2")
6    .select(udf1($"col1").as("newCol1"))
7    .selectExpr("newCol1 + newCol1", "newCol1")
8    .explain()
```

```
== Physical Plan ==
*(1) Project [(newCol1#213 + newCol1#213) AS (newCol1 + newCol1)#215, newCol1#213]
+- *(1) Project [if (isnull(col1#110)) null else UDF(col1#110) AS newCol1#213]
   +- *(1) FileScan orc default.tabdemo2[col1#110,col2#111,col3#112] Batched: true, DataFilters: [], Forma
t: ORC, Location: CatalogFileIndex[dbfs:/user/hive/warehouse/tabdemo2], PartitionCount: 0, PartitionFilter
s: [], PushedFilters: [], ReadSchema: struct<col1:int,col2:timestamp>
import org.apache.spark.sql.functions.udf
udf1: org.apache.spark.sql.expressions.UserDefinedFunction = UserDefinedFunction(<function1>,IntegerType,S
ome(List(IntegerType)))
```

Cross-session SQL Cache

- If a query is cached in the one session, the new queries in all the sessions might be impacted.
- Check your query plan!

```
1  val df1 = spark.table("tabDemo2").selectExpr("col1 + col1", "col3")
2  df1.cache()
```

Cmd 11

```
1  spark.table("tabDemo2")
2    .selectExpr("(col1 + col1) as newCol1", "col3")
3    .selectExpr("newCol1 + col3")
4    .explain(true)
```

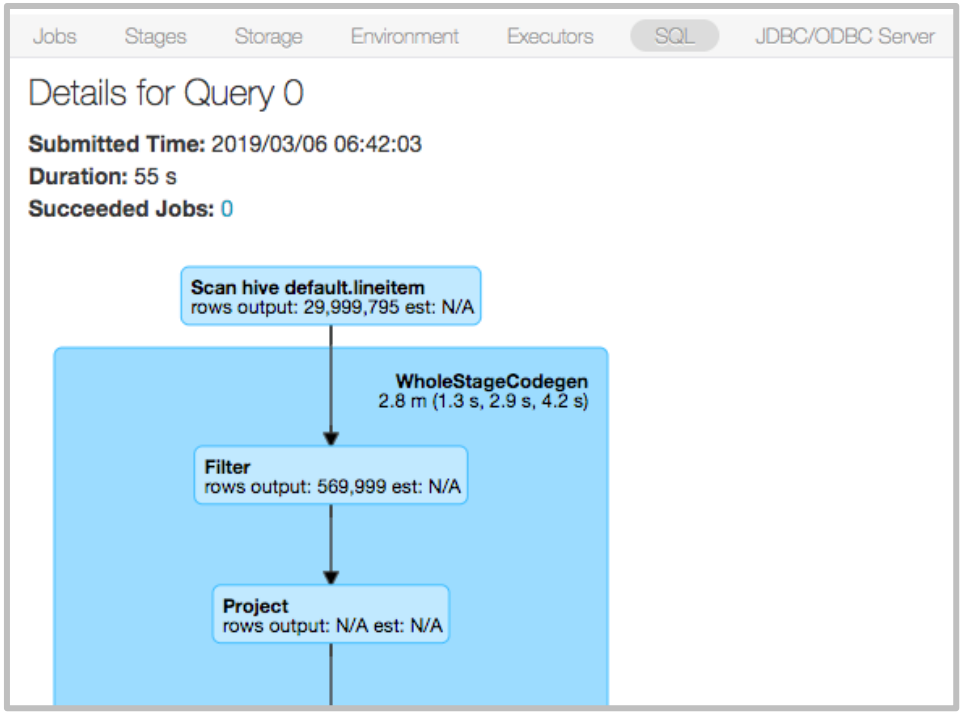
```
== Parsed Logical Plan ==
'Project [('newCol1 + 'col3) AS (newCol1 + col3)#277]
+- Project [(col1#110 + col1#110) AS newCol1#274, col3#112]
   +- SubqueryAlias `default`.`tabdemo2`
      +- Relation[col1#110,col2#111,col3#112] orc

== Analyzed Logical Plan ==
(newCol1 + col3): int
Project [(newCol1#274 + col3#112) AS (newCol1 + col3)#277]
+- Project [(col1#110 + col1#110) AS newCol1#274, col3#112]
   +- SubqueryAlias `default`.`tabdemo2`
      +- Relation[col1#110,col2#111,col3#112] orc

== Optimized Logical Plan ==
Project [(newCol1#274 + col3#112) AS (newCol1 + col3)#277]
+- InMemoryRelation [newCol1#274, col3#112], StorageLevel(disk, memory, deserialized, 1 replicas)
      +- *(1) Project [(col1#110 + col1#110) AS (col1 + col1)#258, col3#112]
         +- *(1) FileScan orc default.tabdemo2[col1#110,col3#112] Batched: true, DataFilters: [], Format:
ORC, Location: CatalogFileIndex[dbfs:/user/hive/warehouse/tabdemo2], PartitionCount: 0, PartitionFilters:
[], PushedFilters: [], ReadSchema: struct<col1:int>
```

Join Hints in Spark 3.0

- BROADCAST
  - Broadcast Hash/Nested-loop Join
- MERGE
  - Shuffle Sort Merge Join
- SHUFFLE_HASH
  - Shuffle Hash Join
- SHUFFLE_REPLICATE_NL
  - Shuffle-and-Replicate Nested Loop Join

Track Execution

From SQL query to Spark Jobs



SQL

Completed Queries: 1

Completed Queries (1)

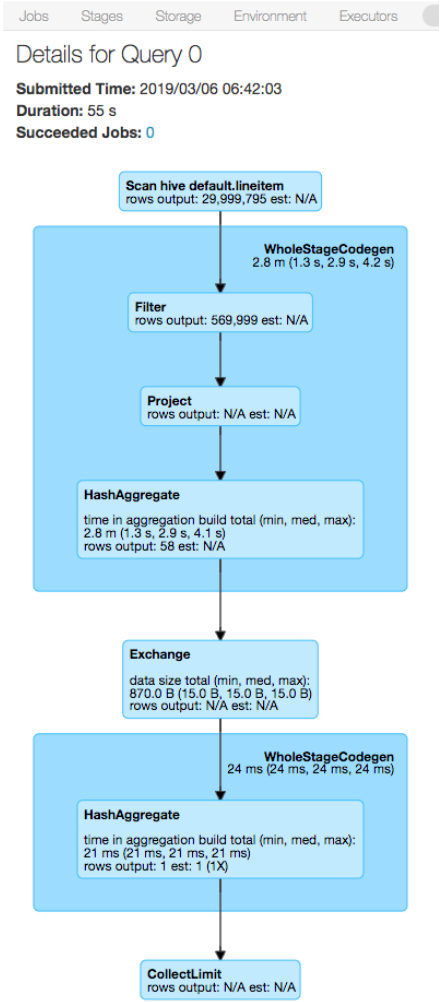
ID	Description	Submitted	Duration	Job IDs
0	show at command-15475442:12 +details	2019/03/06 06:42:03	55 s	[0]

- A SQL query => multiple Spark jobs.

  - For example, broadcast exchange, shuffle exchange, Scalar subquery.
  - External data sources: Delta Lake.
  - New adaptive query execution.

- A Spark job => A DAG

  A chain of RDD dependencies organized in a directed acyclic graph (DAG)

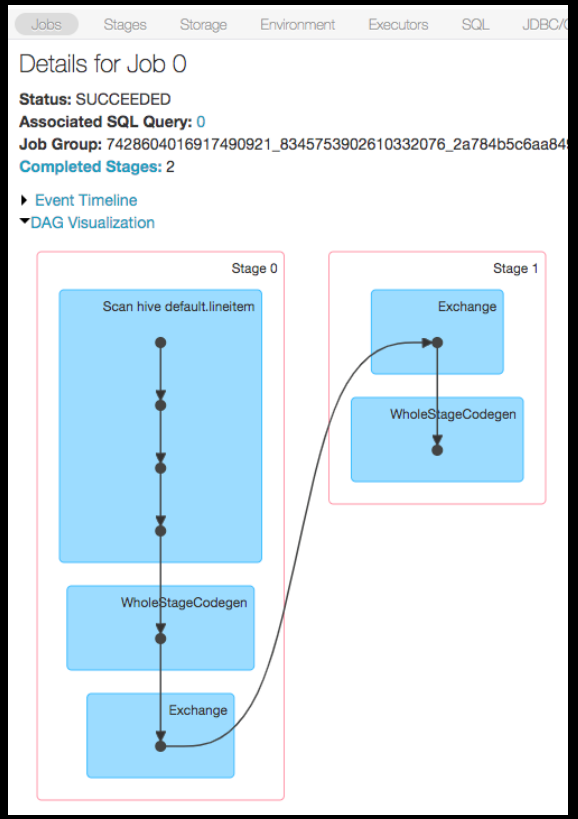


Query Execution

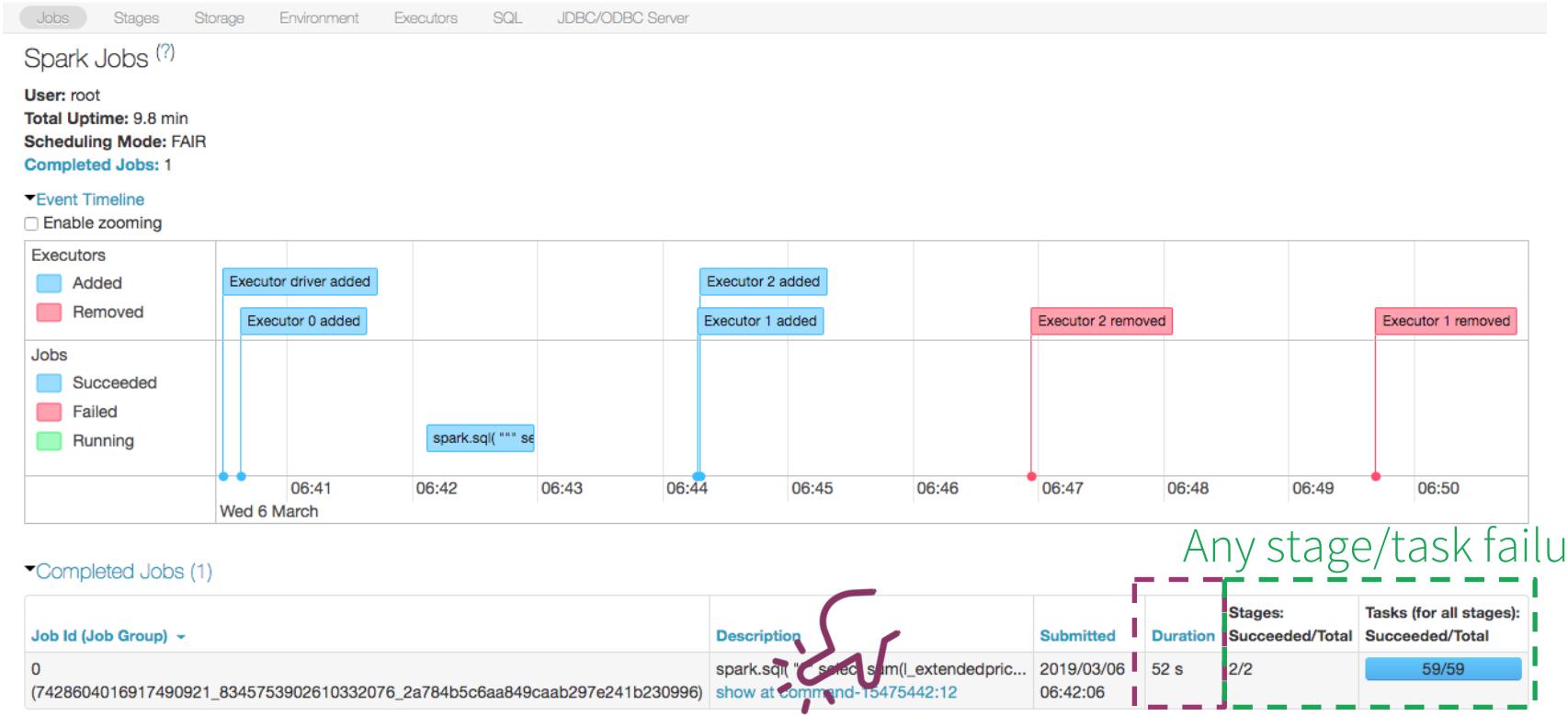
Selected Physical Plan → DAGs

The higher level SQL physical operators.

The low level Spark RDD primitives.



Job Tab in Spark UI



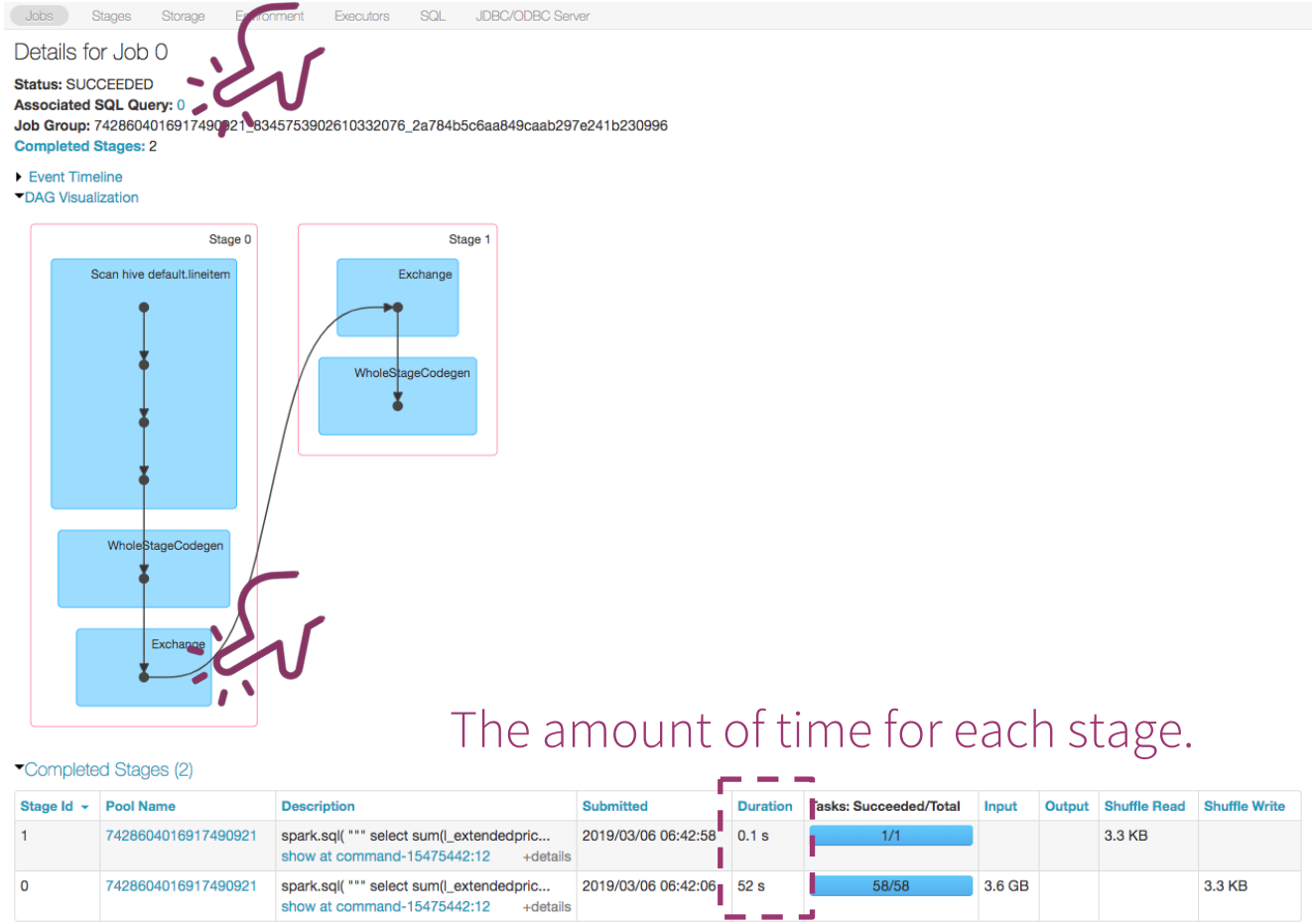
Any stage/task failure?

The amount of time for each job.

databricks

Job Tab

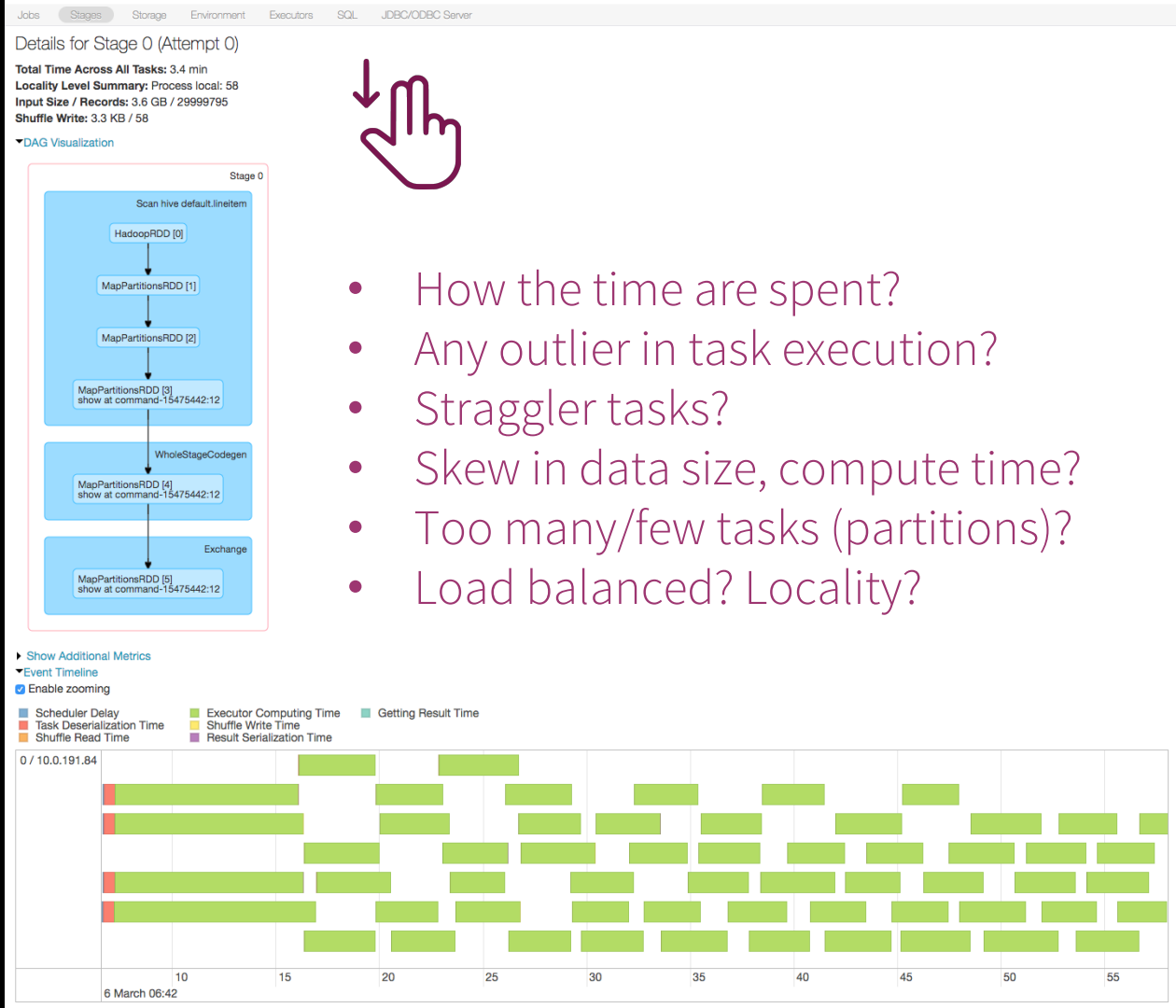
- Jobs
- Stages
- Tasks



databricks

Stages Tab

Tasks specific info



- How the time are spent?
- Any outlier in task execution?
- Straggler tasks?
- Skew in data size, compute time?
- Too many/few tasks (partitions)?
- Load balanced? Locality?

databricks

Summary Metrics for 58 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	1 s	3 s	3 s	3 s	10 s
Scheduler Delay	5 ms	8 ms	11 ms	17 ms	84 ms
Task Deserialization Time	1 ms	2 ms	3 ms	10 ms	0.5 s
GC Time	0 ms	10 ms	13 ms	18 ms	0.3 s
Result Serialization Time	0 ms	0 ms	0 ms	0 ms	2 ms
Getting Result Time	0 ms	0 ms	0 ms	0 ms	0 ms
Peak Execution Memory	0.0 B	0.0 B	0.0 B	0.0 B	0.0 B
Input Size / Records	36.2 MB / 294076	64.0 MB / 519617	64.0 MB / 519676	64.0 MB / 523649	64.0 MB / 528652
Shuffle Write Size / Records		B / 1	59.0 B / 1	59.0 B / 1	59.0 B / 1

Which executor's log we should read?

▾Aggregated Metrics by Executo

Executor ID ▲	Address	ailed Tasks	Killed Tasks	Succeeded Tasks	Input Size / Records	Shuffle Write Size / Records	Blacklisted
0 stdout stderr	10.0.191.84:3358	)	0	58	3.6 GB / 29999795	3.3 KB / 58	false

Balanced?

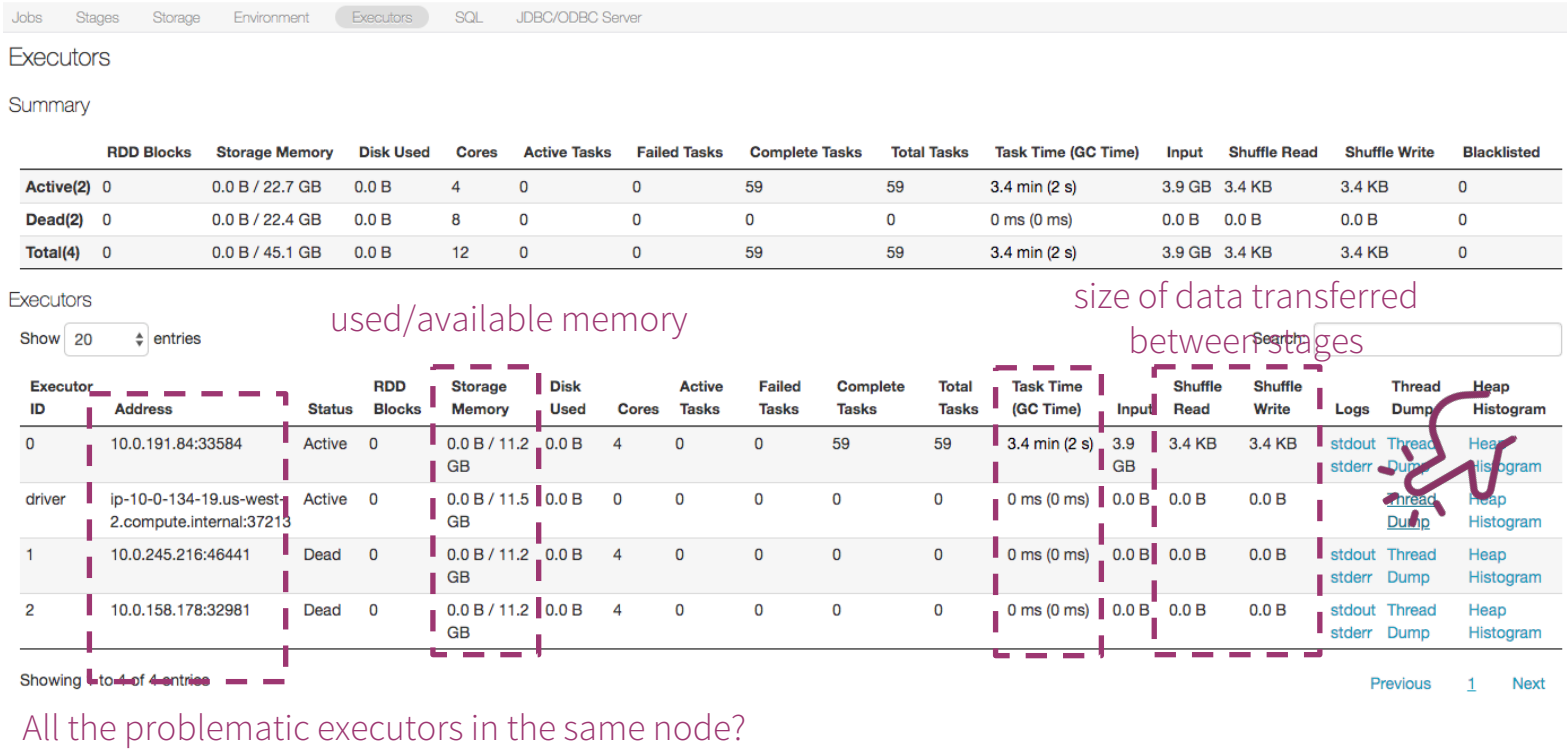
Skew?

▾Tasks (58)

Killed?

Index	ID	Attempt	Status	Locality Level	Executor ID	Host	Launch Tim	Duration	Scheduler Delay	Task Deserialization Time	GC Time	Result Serialization Time	Getting Result Time	Peak Execution Memory	Input Size / Records	Write Time	Shuffle Write Size / Records	Errors
0	0	0	SUCCESS	PROCESS_LOCAL	0	10.0.191.84 stdout stderr	2019/03/06 06:42:06	10 s	83 ms	0.5 s	0.3 s	0 ms	0 ms	0.0 B	64.0 MB / 528652		59.0 B / 1	
2	2	0	SUCCESS	PROCESS_LOCAL	0	10.0.191.84 stdout stderr	2019/03/06 06:42:06	9 s	71 ms	0.5 s	0.2 s	1 ms	0 ms	0.0 B	64.0 MB / 523796		59.0 B / 1	
3	3	0	SUCCESS	PROCESS_LOCAL	0	10.0.191.84 stdout stderr	2019/03/06 06:42:06	9 s	73 ms	0.5 s	0.2 s	0 ms	0 ms	0.0 B	64.0 MB / 523738	10 ms	59.0 B / 1	
1	1	0	SUCCESS	PROCESS_LOCAL	0	10.0.191.84 stdout stderr	2019/03/06 06:42:06	9 s	84 ms	0.5 s	0.2 s	2 ms	0 ms	0.0 B	64.0 MB / 527402	8 ms	59.0 B / 1	
12	12	0	SUCCESS	PROCESS_LOCAL	0	10.0.191.84 stdout stderr	2019/03/06 06:42:22	4 s	17 ms	18 ms	18 ms	0 ms	0 ms	0.0 B	64.0 MB / 523739		59.0 B / 1	

Executors Tab



Summary

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Blacklisted
Active(2)	0	0.0 B / 22.7 GB	0.0 B	4	0	0	59	59	3.4 min (2 s)	3.9 GB	3.4 KB	3.4 KB	0
Dead(2)	0	0.0 B / 22.4 GB	0.0 B	8	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	0
Total(4)	0	0.0 B / 45.1 GB	0.0 B	12	0	0	59	59	3.4 min (2 s)	3.9 GB	3.4 KB	3.4 KB	0

Executors

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Logs	Thread Dump	Heap Histogram
0	10.0.191.84:33584	Active	0	0.0 B / 11.2 GB	0.0 B	4	0	0	59	59	3.4 min (2 s)	3.9 GB	3.4 KB	3.4 KB	stdout stderr	Thread Dump	Heap Histogram
driver	ip-10-0-134-19.us-west-2.compute.internal:37213	Active	0	0.0 B / 11.5 GB	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B		Thread Dump	Heap Histogram
1	10.0.245.216:46441	Dead	0	0.0 B / 11.2 GB	0.0 B	4	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump	Heap Histogram
2	10.0.158.178:32981	Dead	0	0.0 B / 11.2 GB	0.0 B	4	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump	Heap Histogram

Spark UI

Hostname: ec2-54-185-31-39.us-west-2.compute.amazonaws.com Spark Version: 2.4.0

Jobs | Stages | Storage | Environment | Executors | SQL | JDBC/ODBC Server

Thread dump for executor driver

Updated at 2019/04/21 01:57:38

Expand All

Search: hiveclient

- Interacting with Hive metastore?
- Slow query planning?
- Slow file listing?

Thread ID	Thread Name	Thread State	Thread Locks
1331	WRAPPER-ReplId-10b19-7ae6f-00815-e	RUNNABLE	Lock(java.util.concurrent.locks.ReentrantLock$NonfairSync@111358199}), Monitor(org.apache.spark.sql.hive.client.HiveClientImpl@608273722}), Monitor(sun.security.ssl.AppInputStream@229024694}), Monitor(org.apache.spark.sql.execution.command.DataWritingCommandExec@1613603167}), Monitor(line407480b3f69249989060248fc235f7ee88.$eval$@1213128957}), Monitor(java.lang.Object@1798611689})

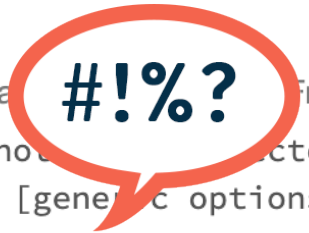
```
java.net.SocketInputStream.socketRead0(Native Method)
java.net.SocketInputStream.socketRead(SocketInputStream.java:116)
java.net.SocketInputStream.read(SocketInputStream.java:171)
java.net.SocketInputStream.read(SocketInputStream.java:141)
```

Insert Partitioned Hive Table

```
1  spark.sql(
2    s"""
3      |CREATE TABLE hiveTab1 (key INT, value STRING, p1 STRING)
4      | USING hive OPTIONS (fileformat 'parquet')
5      |  PARTITIONED BY (p1)
6    """.stripMargin)
7
8  spark.time {
9    spark.sql(
10       s"""
11         |INSERT INTO TABLE hiveTab1
12         |  SELECT id AS key, id AS value, CAST((id % 50) AS STRING) p1
13         |  FROM RANGE(0, 5000)
14        """.stripMargin)
15 }
```

OR "STORED AS PARQUET"

5000 partitions took almost 8 minutes!!!



▸ (1) Spark Jobs

▸ res30: org.apa… …Frame

```
-chgrp: '' does no… …cted pattern for group
Usage: hadoop fs [gene… …c options] -chgrp [-R] GROUP PATH...
Time taken: 464115 ms
```

```
sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.j
java.lang.reflect.Method.invoke(Method.java:498)
org.apache.spark.sql.hive.client.Shim_v0_12.loadDynamicPartitions(HiveShim.sca
org.apache.spark.sql.hive.client.HiveClientImpl$$anonfun$loadDynamicPartitions
org.apache.spark.sql.hive.client.HiveClientImpl$$anonfun$loadDynamicPartitions
org.apache.spark.sql.hive.client.HiveClientImpl$$anonfun$loadDynamicPartitions
```

Insert Partitioned Native Table

```
1  spark.sql(
2    s"""
3      |CREATE TABLE nativeTab1 (key INT, value STRING, p1 STRING)
4      | USING parquet
5      |  PARTITIONED BY (p1)
6    """.stripMargin)
7
8  spark.time {
9    spark.sql(
10       s"""
11         |INSERT INTO TABLE nativeTab1
12         |  SELECT id AS key, id AS value, CAST((id % 50) AS STRING) p1
13         |  FROM RANGE(0, 5000)
14        """.stripMargin)
15 }
```

Reduced from almost 8 minutes to less than 1 minute !!!

▸ (1) Spark Jobs

▸ res32: org.apache.spark.sql.DataFrame

Time taken: 48730 ms

Insert Partitioned Delta Table

```
1  spark.sql(
2    s"""
3      |CREATE TABLE deltaTab1 (key INT, value STRING, p1 STRING)
4      | USING delta
5      |  PARTITIONED BY (p1)
6    """.stripMargin)
7
8  spark.time {
9    spark.sql(
10       s"""
11         |INSERT INTO TABLE deltaTab1
12         |  SELECT id AS key, id AS value, CAST((id % 50) AS STRING) p1
13         |  FROM RANGE(0, 5000)
14        """.stripMargin)
15 }
```

Reduced from almost 8 minutes to 27 seconds!!!

▸ (4) Spark Jobs

▸ res17: org.apache.spark.sql.DataFrame

```
Time taken: 27735 ms
```

Typical Spark Performance Issues

The table has thousands of partitions
- Hive metastore overhead

This table can have 100s of thousands to millions of files
- File system overhead - listing takes forever!

New data is not immediately visible
- Need to invoke a command “Refresh Table” with the SQL engine they were using

The above issues can add 10s of minutes to the response time!

Delta Lake + Spark

Scalable metadata handling @ Delta Lake

Store metadata in transaction log file instead of metastore

The table has thousands of partitions

- Zero Hive Metastore overhead

The table can have 100s of thousands to millions of files

- No file listing

New data is not immediately visible

- Delta table state is computed on read

How do I use Delta?

format(“parquet”) -> format(“delta”)

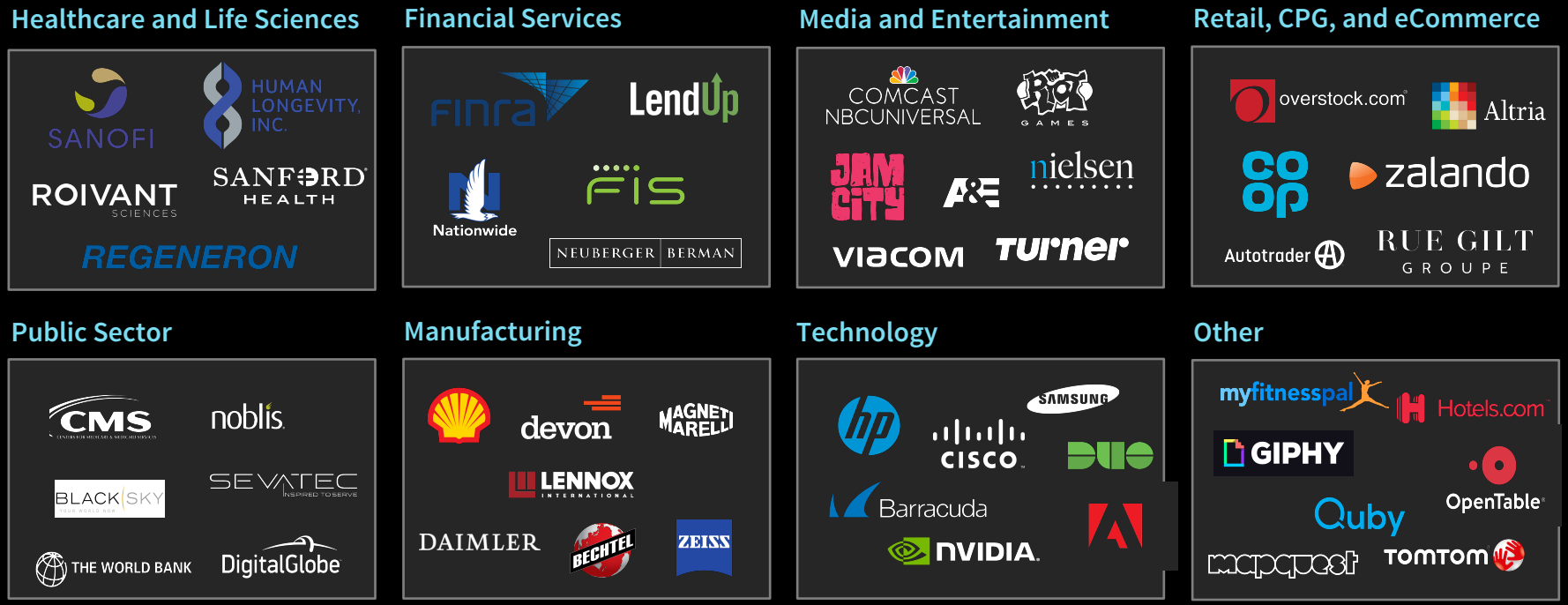
Delta Lake + Spark

- Full ACID transactions
- Schema management
- Data versioning and time travel
- Unified batch/streaming support
- Scalable metadata handling
- Record update and deletion
- Data expectation

Delta Lake: https://delta.io/
For details, refer to the blog
https://tinyurl.com/yxhbe2lg

Delta Usage Statistics

More than 1 exabyte processed (10^{18} bytes) monthly



databricks

Additional Resources

- Apache Spark document: https://spark.apache.org/docs/latest/sql-programming-guide.html
- Blog: https://databricks.com/blog/category/engineering/spark
- Previous summit: https://databricks.com/sparkaisummit/north-america/sessions
- Delta Lake document: https://docs.delta.io
- Databricks document: https://docs.databricks.com/
- Books: https://www.amazon.com/s?k=apache+spark
- Databricks academy: https://academy.databricks.com
- Databricks ebooks: https://databricks.com/resources/type/ebooks

Thank you

Xiao Li

(lixiao@databricks.com)