SQL Performance Improvements At a Glance in Apache Spark 3.0

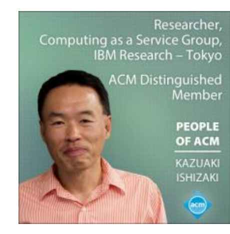
Kazuaki Ishizaki

IBM Research

About Me – Kazuaki Ishizaki

- Researcher at IBM Research – Tokyo
  https://ibm.biz/ishizaki
  - Compiler optimization, language runtime, and parallel processing
- Apache Spark committer from 2018/9 (SQL module)

- Work for IBM Java (Open J9, now) from 1996
  - Technical lead for Just-in-time compiler for PowerPC
- ACM Distinguished Member



- SNS

@kiszk

https://www.slideshare.net/ishizaki/

Spark 3.0

- The long wished-for release…
  - More than 1.5 years passed after Spark 2.4 has been released

Spark 3.0

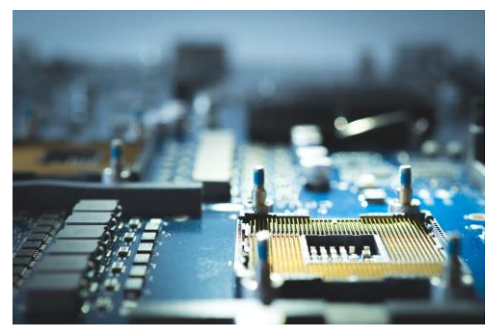
- Four Categories of Major Changes for SQL

Interactions with developers

Dynamic optimizations

Catalyst improvements

Infrastructure updates



When Spark 2.4 was released?

- The long wished-for release…
  - More than 1.5 years passed after Spark 2.4 has been released

Spark 2.4.0 released

November 2, 2018

We are happy to announce the availability of Spark 2.4.0! Visit the release notes to read about the new features, or download the release today.

2018 November

What We Expected Last Year?

- The long wished-for release…
  - More than 1.5 years passed after Spark 2.4 has been released

Spark 2.4.0 released

November 2, 2018

We are happy to announce the availability of Spark 2.4.0! Visit the release notes to read about the new features, or download the release today.

Keynote at Spark+AI Summit 2019

2019 April

Spark 3.0 Looks Real

- The long wished-for release…
  - More than 1.5 years passed after Spark 2.4 has been released

Spark 2.4.0 released

November 2, 2018

We are happy to announce the availability of Spark 2.4.0! Visit the release notes to read about the new features, or download the release today.

Keynote at Spark+AI Summit 2019

2019 November

Preview release of Spark 3.0

November 6, 2019

To enable wide-scale community testing of the upcoming Spark 3.0 release, the Apache Spark community has posted a preview release of Spark 3.0. This preview is **not a stable release in terms of either API or functionality**, but it is meant to give the community early access to try the code that will become Spark 3.0. If you would like to test the release, please download it, and send feedback using either the mailing lists or JIRA.

Spark 3.0 has been released!!

- The long wished-for release…
  - More than 1.5 years passed after Spark 2.4 has been released

3.0.0 has released early June, 2020

Community Worked for Spark 3.0 Release

Version 3.0.0 UNRELEASED

Start date not set Release date not set Release Notes

0 Warnings | 3464 Issues in version | 3463 Issues done | 0 Issues in progress | 1 Issues to do

- 3464 issues (as of June 8th, 2020)
  - New features
  - Improvements
  - Bug fixes

Source https://issues.apache.org/jira/projects/SPARK/versions/12339177

Many Many Changes for 1.5 years

Version 3.0.0 UNRELEASED

Start date not set   Release date not set   Release Notes

0 Warnings	3464 Issues in version	3463 Issues done	0 Issues in progress	1 Issues to do

Hard to understand what's new due to many many changes

Many Many Changes for 1.5 years

Version 3.0.0 UNRELEASED

Start date not set   Release date not set   Release Notes

0 Warnings   3464 Issues in version   3463 Issues done   0 Issues in progress   1 Issues to do

Hard to understand what's new due to many many changes

This session guides you to understand what's new for SQL performance

Seven Major Changes for SQL Performance

1. New EXPLAIN format
2. All type of join hints
3. Adaptive query execution
4. Dynamic partitioning pruning
5. Enhanced nested column pruning & pushdown
6. Improved aggregation code generation
7. New Scala and Java

Seven Major Changes for SQL Performance

1. New EXPLAIN format
2. All type of join hints
3. Adaptive query execution
4. Dynamic partitioning pruning
5. Enhanced nested column pruning & pushdown
6. Improved aggregation code generation
7. New Scala and Java

Interactions with developers (1–2)

Dynamic optimizations (3–4)

Catalyst improvements (5–6)

Infrastructure updates (7)

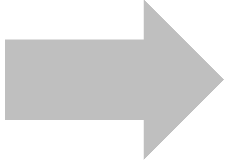
What is Important to Improve Performance?

- Understand how a query is optimized

SPARK-27395

What is Important to Improve Performance?

- Understand how a query is optimized



Easy to Read a Query Plan

SPARK-27395

Read a Query Plan

```
“SELECT key, Max(val) FROM temp WHERE key > 0 GROUP BY key HAVING max(val) > 0”
```

From #24759

SPARK-27395

Not Easy to Read a Query Plan on Spark 2.4

- Not easy to understand how a query is optimized

Output is **too long!!**

```
scala> val query = “SELECT key, Max(val) FROM temp WHERE key > 0 GROUP BY key HAVING max(val) > 0”
scala> sql(“EXPLAIN “ + query).show(false)
```

```
!== Physical Plan ==
*(2) Project [key#2, max(val)#15]
+- *(2) Filter (isnotnull(max(val#3)#18) AND (max(val#3)#18 > 0))
   +- *(2) HashAggregate(keys=[key#2], functions=[max(val#3)], output=[key#2, max(val)#15,
max(val#3)#18])
      +- Exchange hashpartitioning(key#2, 200)
         +- *(1) HashAggregate(keys=[key#2], functions=[partial_max(val#3)], output=[key#2,
max#21])
            +- *(1) Project [key#2, val#3]
               +- *(1) Filter (isnotnull(key#2) AND (key#2 > 0))
                  +- *(1) FileScan parquet default.temp[key#2,val#3] Batched: true,
DataFilters: [isnotnull(key#2), (key#2 > 0)], Format: Parquet, Location:
InMemoryFileIndex[file:/user/hive/warehouse/temp], PartitionFilters: [], PushedFilters:
[IsNotNull(key), GreaterThan(key,0)], ReadSchema: struct<key:int,val:int>
```

From #24759

SPARK-27395

Easy to Read a Query Plan on Spark 3.0

- Show **a query in a terse format** with detail information

```
scala> sql("EXPLAIN FORMATTED " + query).show(false)
```

```
!== Physical Plan ==
Project (8)
+- Filter (7)
   +- HashAggregate (6)
      +- Exchange (5)
         +- HashAggregate (4)
            +- Project (3)
               +- Filter (2)
                  +- Scan parquet default.temp1 (1)

(1) Scan parquet default.temp [codegen id : 1]
Output: [key#2, val#3]

(2) Filter [codegen id : 1]
Input     : [key#2, val#3]
Condition : (isnotnull(key#2) AND (key#2 > 0))

(3) Project [codegen id : 1]
Output    : [key#2, val#3]
Input     : [key#2, val#3]
```

```
(4) HashAggregate [codegen id : 1]
Input: [key#2, val#3]

(5) Exchange
Input: [key#2, max#11]

(6) HashAggregate [codegen id : 2]
Input: [key#2, max#11]

(7) Filter [codegen id : 2]
Input     : [key#2, max(val)#5, max(val#3)#8]
Condition : (isnotnull(max(val#3)#8) AND
(max(val#3)#8 > 0))

(8) Project [codegen id : 2]
Output    : [key#2, max(val)#5]
Input     : [key#2, max(val)#5, max(val#3)#8]
```

SPARK-27395

Seven Major Changes for SQL Performance

1. New EXPLAIN format
2. All type of join hints
3. Adaptive query execution
4. Dynamic partitioning pruning
5. Enhanced nested column pruning & pushdown
6. Improved aggregation code generation
7. New Scala and Java

Only One Join Type Can be Used on Spark 2.4

Join type	2.4
Broadcast	BROADCAST
Sort Merge	X
Shuffle Hash	X
Cartesian	X

SPARK-27225

All of Join Type Can be Used for a Hint

Join type	2.4	3.0
Broadcast	BROADCAST	BROADCAST
Sort Merge	X	SHUFFLE_MERGE
Shuffle Hash	X	SHUFFLE_HASH
Cartesian	X	SHUFFLE_REPLICATE_NL

Examples

```
SELECT /*+ SHUFFLE_HASH(a, b) */ * FROM a, b
       WHERE a.a1 = b.b1
```

```
val shuffleHashJoin = aDF.hint(“shuffle_hash”)
                         .join(bDF, aDF(“a1”) === bDF(“b1”))
```


SPARK-27225

Seven Major Changes for SQL Performance

1. New EXPLAIN format
2. All type of join hints
3. Adaptive query execution
4. Dynamic partitioning pruning
5. Enhanced nested column pruning & pushdown
6. Improved aggregation code generation
7. New Scala and Java

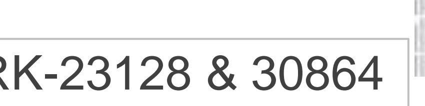
Automatically Tune Parameters for Join and Reduce

- Three parameters by using runtime statistics information (e.g. data size)
  1. Set the number of reducers to avoid wasting memory and I/O resource
  2. Select better join strategy to improve performance
  3. Optimize skewed join to avoid imbalance workload

Without manual tuning properties run-by-run

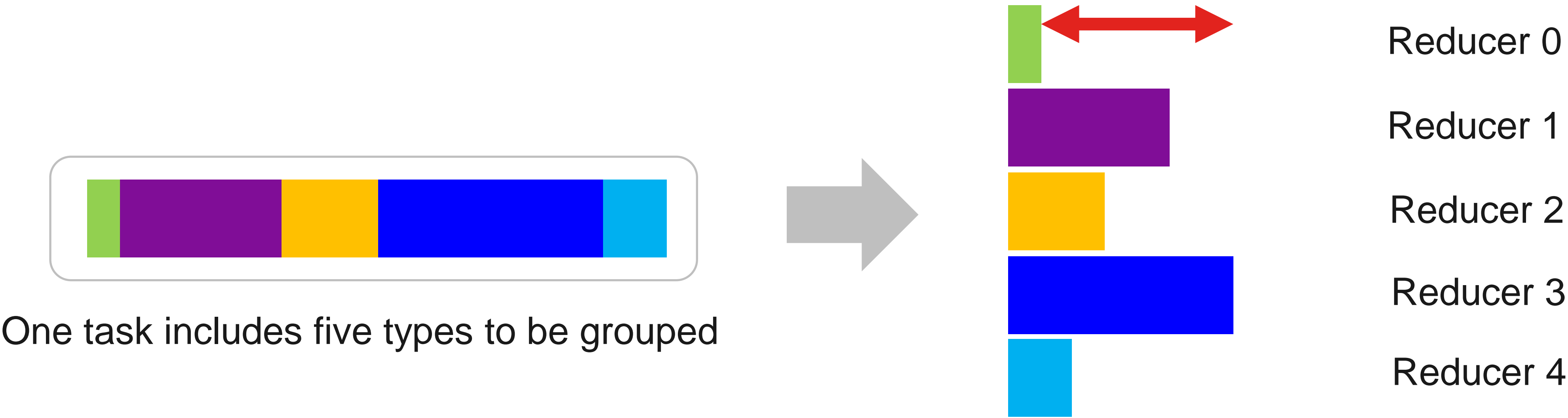
Yield 8x performance improvement of Q77 in TPC-DS

Source: Adaptive Query Execution: Speeding Up Spark SQL at Runtime



Used Preset Number of Reduces on Spark 2.4

- The number of reducers is set based on the property `spark.sql.shuffle.partitions (default: 200)`



Five reducers for five partitions

SPARK-23128 & 30864

Tune the Number of Reducers on Spark 3.0

- Select the number of reducers to meet the given target partition size at each reducer

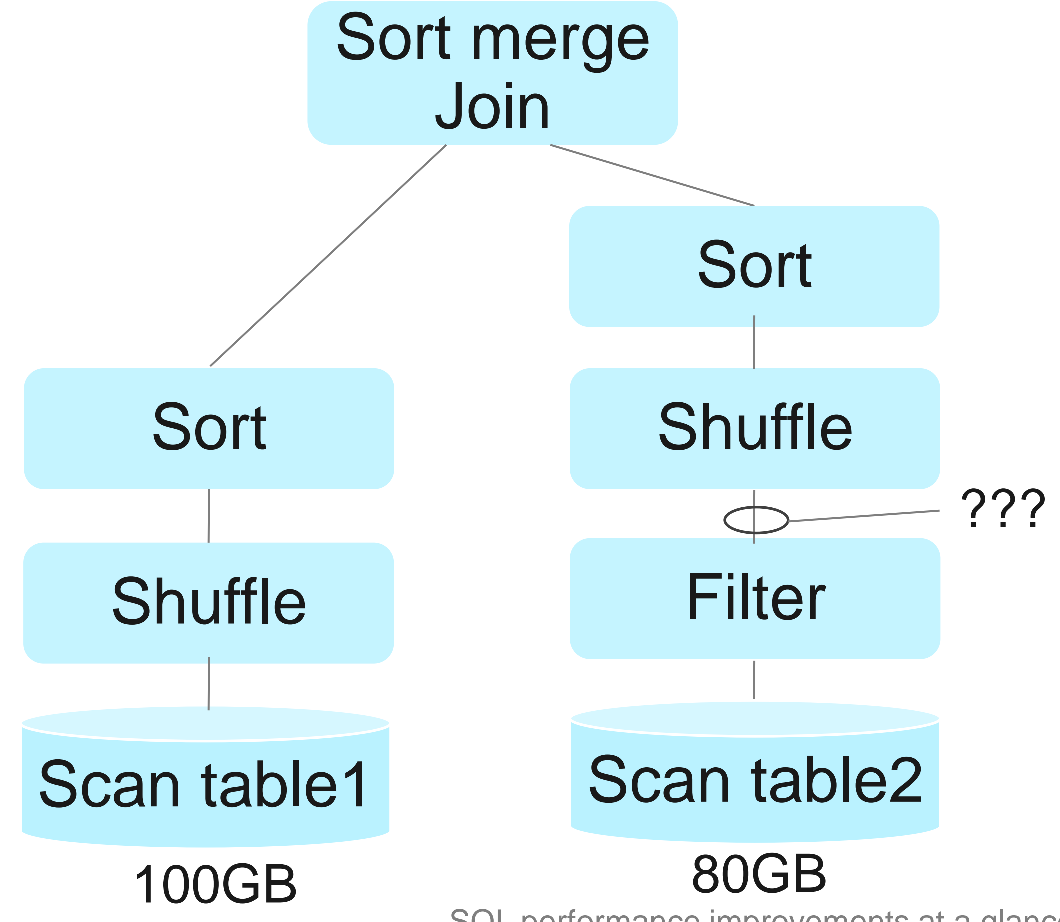
Three reducers for five partitions

```
spark.sql.adaptive.enabled -> true (false in Spark 3.0)
spark.sql.adaptive.coalescePartitions.enabled -> true (false in Spark 3.0)
```

SPARK-23128 & 30864

Statically Selected Join Strategy on Spark 2.4

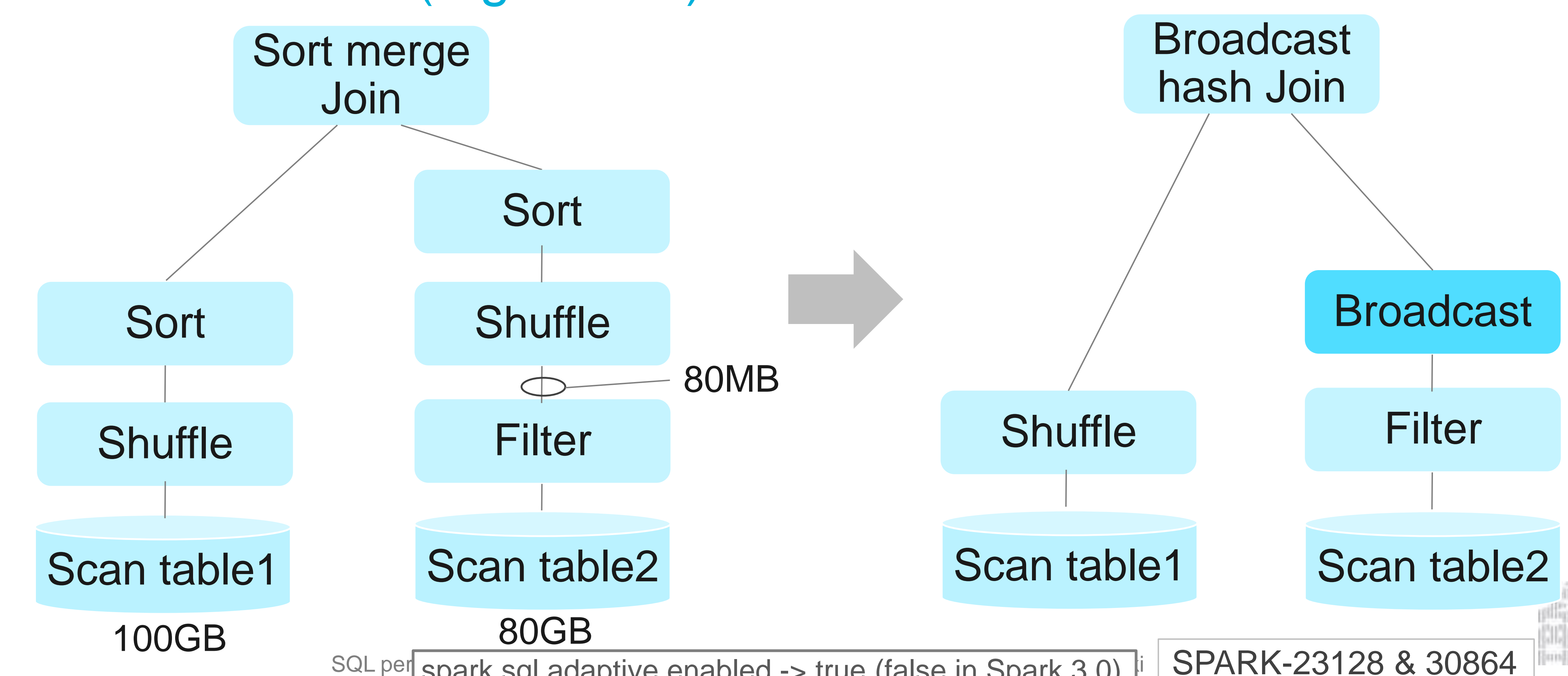
- Spark 2.4 decided sort merge join strategy using statically available information (e.g. 100GB and 80GB)



SPARK-23128 & 30864

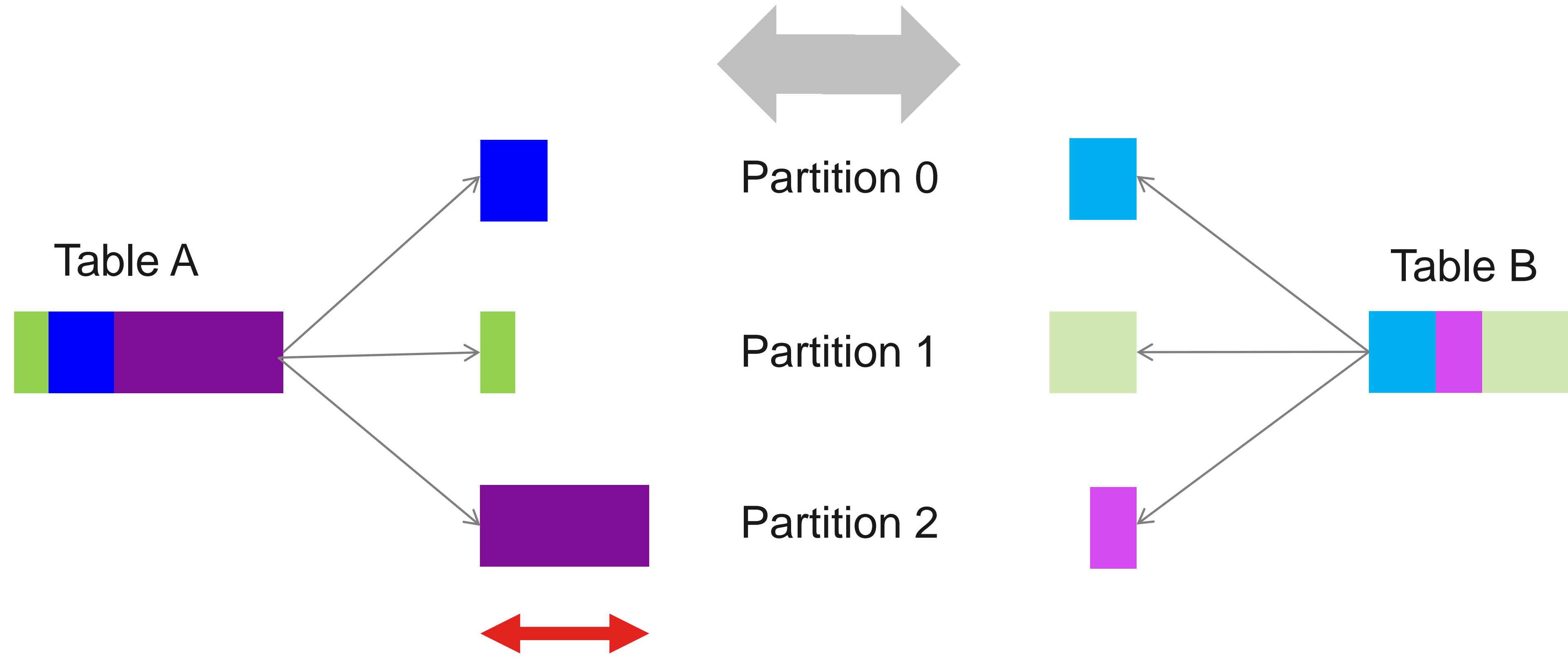
Dynamically Change Join Strategy on Spark 3.0

- Spark 3.0 dynamically select broadcast hash join strategy using runtime information (e.g. 80MB)



Skewed Join is Slow on Spark 2.4

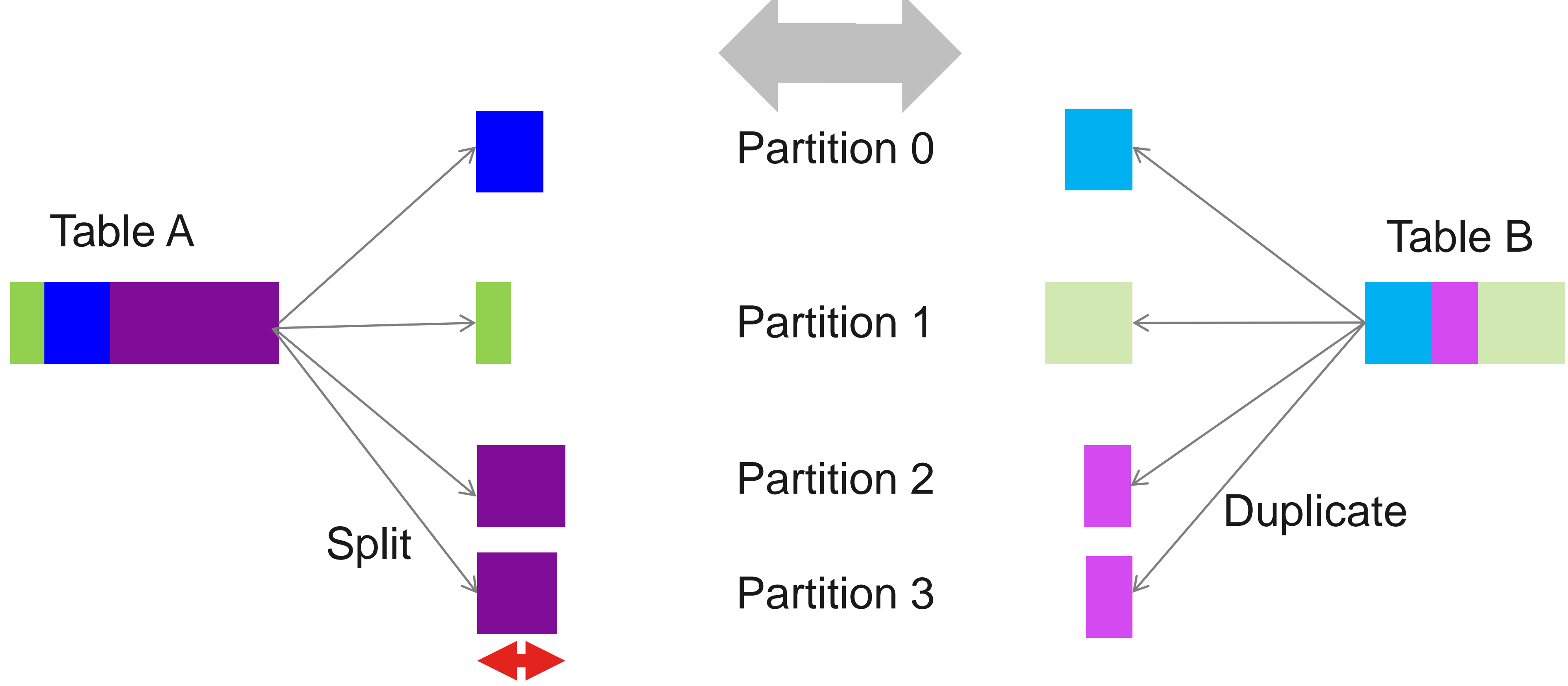
- The join time is dominated by processing the largest partition



SPARK-23128 & 30864

Skewed Join is Faster on Spark 3.0

- The large partition is split into multiple partitions



spark.sql.adaptive.enabled -> true (false in Spark 3.0)
spark.sql.adaptive.skewJoin.enabled-> true (false in Spark 3.0)

SPARK-23128 & 30864

Seven Major Changes for SQL Performance

1. New EXPLAIN format
2. All type of join hints
3. Adaptive query execution
4. Dynamic partitioning pruning
5. Enhanced nested column pruning & pushdown
6. Improved aggregation code generation
7. New Scala and Java

Dynamic Partitioning Pruning

- Avoid to read unnecessary partitions in a join operation
  - By using results of filter operations in another table
- Dynamic filter can avoid to read unnecessary partition

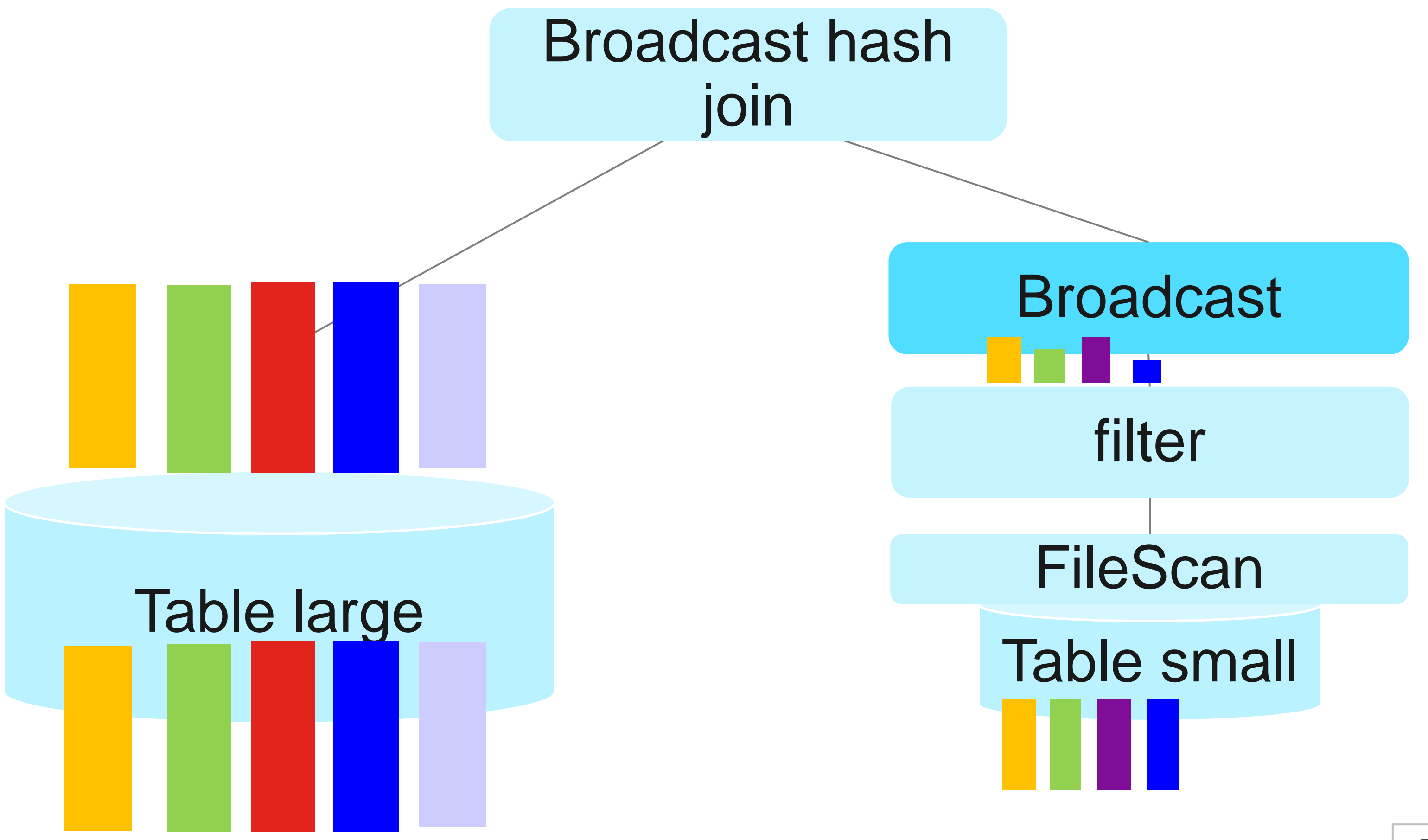
Yield 85x performance improvement of Q98 in TPC-DS 10TB

Source: Dynamic Partition Pruning in Apache Spark

SPARK-11150

Naïve Broadcast Hash Join on Spark 2.4

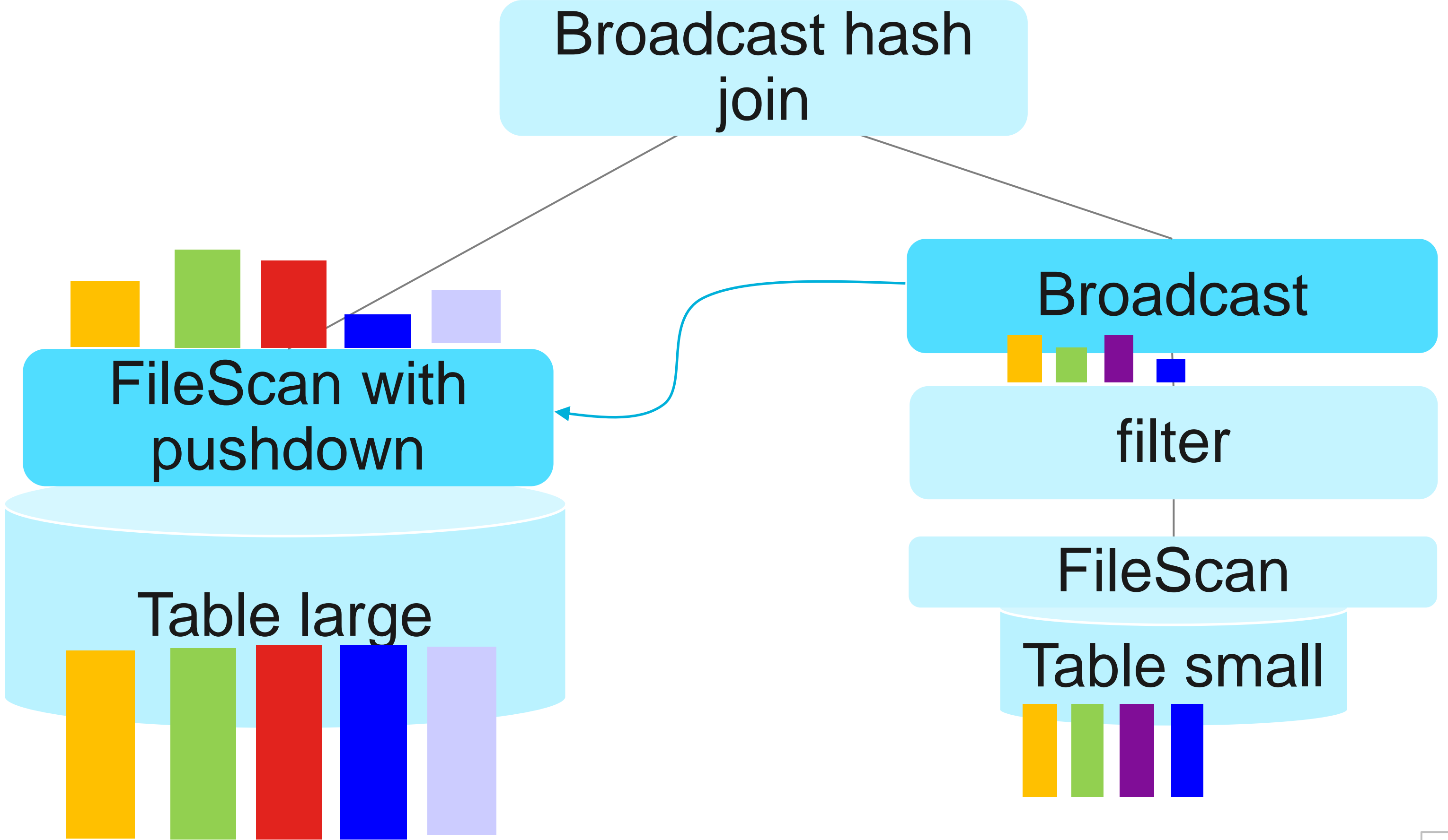
- All of the data in Large table is read



SPARK-11150

Prune Data with Dynamic Filter on Spark 3.0

- Large table can reduce the amount of data to be read using pushdown with dynamic filter



SPARK-11150

Example of Dynamic Partitioning Pruning

```
scala> spark.range(7777).selectExpr("id", "id AS key").write.partitionBy("key").saveAsTable("tableLarge")
scala> spark.range(77).selectExpr("id", "id AS key").write.partitionBy("key").saveAsTable("tableSmall")
scala> val query = "SELECT * FROM tableLarge JOIN tableSmall ON tableLarge.key = tableSmall.key AND tableSmall.id < 3"
scala> sql("EXPLAIN FORMATTED " + query).show(false)
```

```
|== Physical Plan ==
* BroadcastHashJoin Inner BuildRight (8)
:- * ColumnarToRow (2)
:  +- Scan parquet default.tablelarge (1)
+- BroadcastExchange (7)
   +- * Project (6)
      +- * Filter (5)
         +- * ColumnarToRow (4)
            +- Scan parquet default.tablesmall (3)

(1) Scan parquet default.tablelarge
Output [2]: [id#19L, key#20L]
Batched: true
Location: InMemoryFileIndex [file:/home/ishizaki/Spark/300RC1/spark-3.0.0-bin-hadoop2.7/spark-
warehouse/tablelarge/key=0, ... 7776 entries]
PartitionFilters: [isnotnull(key#20L), dynamicpruningexpression(key#20L IN dynamicpruning#56)]
ReadSchema: struct<id:bigint>
```

Source: Quick Overview of Upcoming Spark 3.0

SPARK-11150

Seven Major Changes for SQL Performance

1. New EXPLAIN format
2. All type of join hints
3. Adaptive query execution
4. Dynamic partitioning pruning
5. Enhanced nested column pruning & pushdown
6. Improved aggregation code generation
7. New Scala and Java

Catalyst improvements (items 5–6)

Nested Column Pruning on Spark 2.4

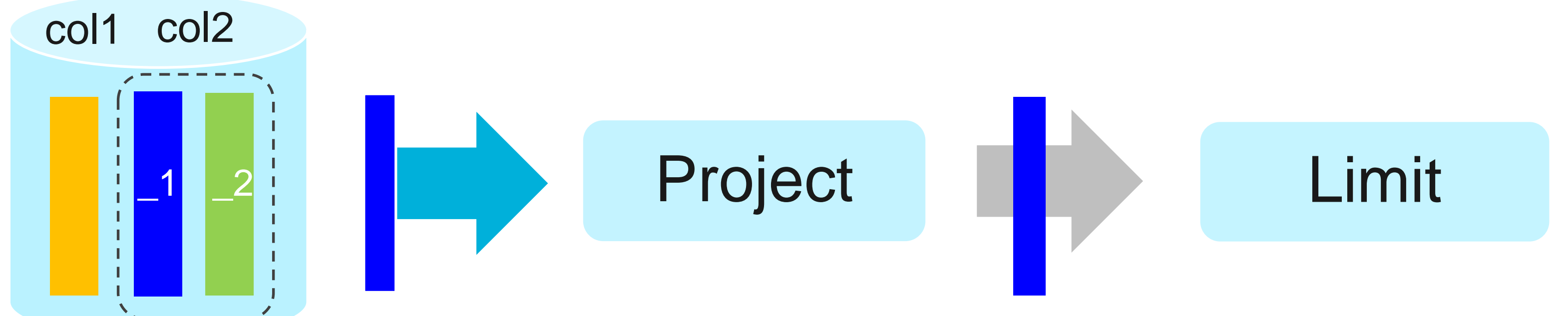
- Column pruning that read only necessary column for Parquet
  - Can be applied to limited operations (e.g. LIMIT)

Source: #23964

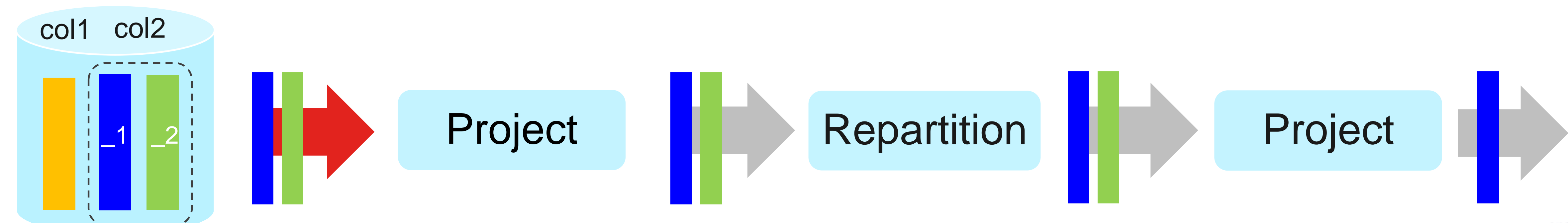
SPARK-25603 & 25556

Limited Nested Column Pruning on Spark 2.4

- Column pruning that read only necessary column for Parquet
  - Can be applied to limited operations (e.g. LIMIT)



  - Cannot be applied other operations (e.g. REPARTITION)

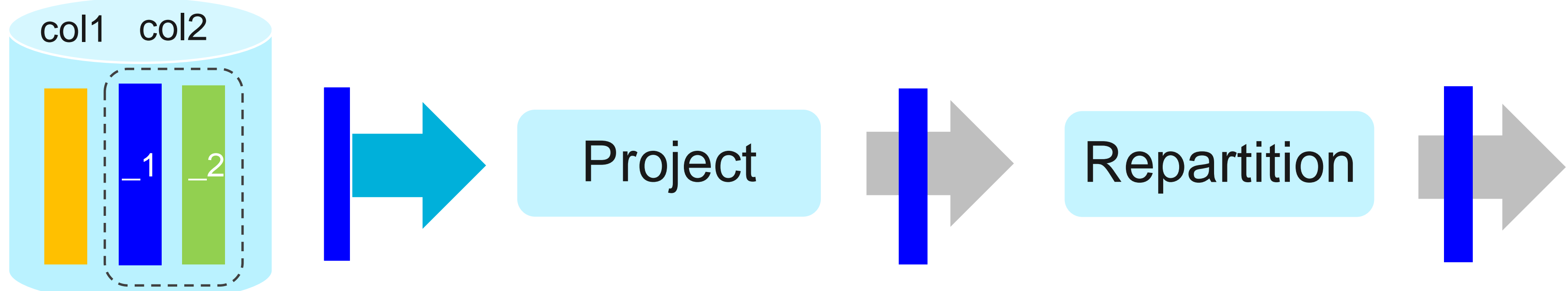


Source: #23964

SPARK-25603 & 25556

Generalize Nested Column Pruning on Spark 3.0

- Nested column pruning can be applied to all operators
  - e.g. LIMITS, REPARTITION, …



Source: #23964

SPARK-25603 & 25556

Example of Nested Column Pruning

- Parquet only reads col2._1, as shown in **ReadSchema**

LIMIT

```
scala> spark.range(1000).map(x => (x, (x, s"$x" * 10))).toDF("col1", "col2").write.parquet("/tmp/p")
scala> spark.read.parquet("/tmp/p").createOrReplaceTempView("temp")
scala> sql("SELECT col2._1 FROM (SELECT col2 FROM tp LIMIT 1000000)").explain
```

```
== Physical Plan ==
CollectLimit 1000000
+- *(1) Project [col2#22._1 AS _1#28L]
   +- *(1) FileScan parquet [col2#22] ..., ReadSchema: struct<col2:struct<_1:bigint>>
```

Repartition

```
scala> sql("SELECT col2._1 FROM (SELECT /*+ REPARTITION(1) */ col2 FROM temp)").explain
```

```
== Physical Plan ==
Exchange RoundRobinPartitioning(1)
+- *(1) Project [col2#5._1 AS _1#11L]
   +- *(1) FileScan parquet [col2#5] ..., PushedFilters: [], ReadSchema: struct<col2:struct<_1:bigint>>
```

Source: #23964

SPARK-25603 & 25556

No Nested Column Pushdown on Spark 2.4

- Parquet cannot apply predication pushdown

```
scala> spark.range(1000).map(x => (x, (x, s"$x" * 10))).toDF("col1", "col2").write.parquet("/tmp/p")
scala> spark.read.parquet("/tmp/p").filter("col2._1 = 100").explain
```

Spark 2.4

```
== Physical Plan ==
*(1) Project [col1#12L, col2#13]
+- *(1) Filter (isnotnull(col2#13) && (col2#13._1 = 100))
   +- *(1) FileScan parquet [col1#12L,col2#13] ..., PushedFilters: [IsNotNull(nested)], ...
```

Source: #28319

SPARK-25603 & 25556

Nested Column Pushdown on Spark 3.0

- Parquet can apply pushdown filter and can read **part of columns**

```
scala> spark.range(1000).map(x => (x, (x, s"$x" * 10))).toDF("col1", "col2").write.parquet("/tmp/p")
scala> spark.read.parquet("/tmp/p").filter("col2._1 = 100").explain
```

Spark 3.0

```
== Physical Plan ==
*(1) Project [col1#0L, col2#1]
+- *(1) Filter (isnotnull(col2#1) AND (col2#1._1 = 100))
   +- FileScan parquet [col1#0L,col2#1] ..., DataFilters: [isnotnull(col2#1), (col2#1.x = 100)],
..., PushedFilters: [IsNotNull(col2), EqualTo(col2._1,100)], ...
```

Source: #28319

SPARK-25603 & 25556

Seven Major Changes for SQL Performance

1. New EXPLAIN format
2. All type of join hints
3. Adaptive query execution
4. Dynamic partitioning pruning
5. Enhanced nested column pruning & pushdown
6. Improved aggregation code generation
7. New Scala and Java

Complex Aggregation is Slow on Spark 2.4

- A complex query is not compiled to native code

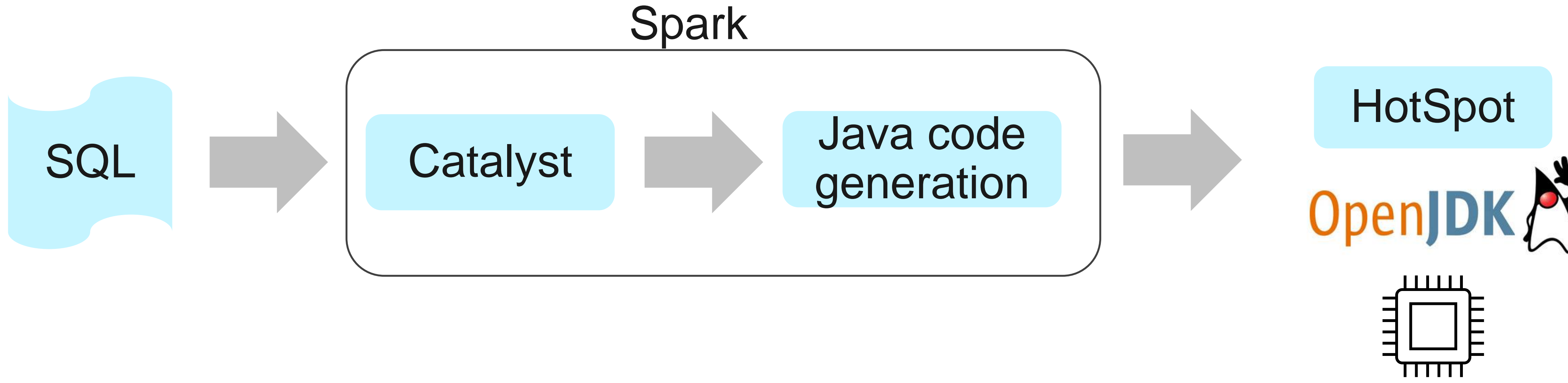
Not good performance of Q66 in TPC-DS

Source: #20695

SPARK-21870

How SQL is Translated to native code

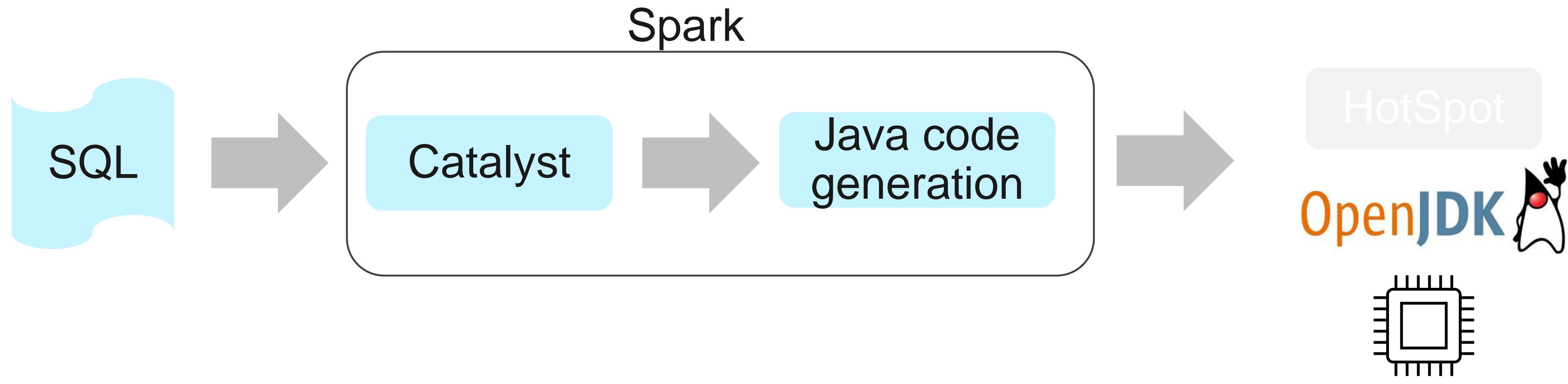
- In Spark, Catalyst translates a given query to Java code
- HotSpot compiler in OpenJDK translates Java code into native code



SPARK-21870

How SQL is Translated to native code

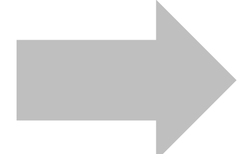
- In Spark, Catalyst translates a given query to Java code
- HotSpot compiler in OpenJDK **gives up** generating native code for more than **8000 Java bytecode instruction per method**



SPARK-21870

Making Aggregation Java Code Small

- In Spark, Catalyst translates a given query to Java code
- HotSpot compiler in OpenJDK **gives up** generating native code for more than **8000 Java bytecode instruction per method**



Catalyst splits a large Java method into small ones to allow HotSpot to generate native code

SPARK-21870

Example of Small Aggregation Code

- Average function (100 rows) for 50 columns

```
scala> val numCols = 50
scala> val colExprs = (0 until numCols).map { i => s"id AS col$i" }
scala> spark.range(100).selectExpr(colExprs: _*).createOrReplaceTempView("temp”)
scala> val aggExprs = (0 until numCols).map { I => s”AVG(col$i)" }
scala> val query = s"SELECT ${aggExprs.mkString(", ")} FROM temp“
scala> import org.apache.spark.sql.execution.debug._
scala> sql(query).debugCodegen()
```

```
Found 2 WholeStageCodegen subtrees.
== Subtree 1 / 2 (maxMethodCodeSize:3679; maxConstantPoolSize:1107(1.69% used); numInnerClasses:0) ==
...
== Subtree 2 / 2 (maxMethodCodeSize:5581; maxConstantPoolSize:882(1.35% used); numInnerClasses:0) ==
```

Source: PR #20965

SPARK-21870

Example of Small Aggregation Code

- Average function (100 rows) for 50 columns

```
scala> val numCols = 50
scala> val colExprs = (0 until numCols).map { i => s"id AS col$i" }
scala> spark.range(100).selectExpr(colExprs: _*).createOrReplaceTempView("temp”)
scala> val aggExprs = (0 until numCols).map { I => s”AVG(col$i)" }
scala> val query = s"SELECT ${aggExprs.mkString(", ")} FROM temp“
scala> import org.apache.spark.sql.execution.debug._
scala> sql(query).debugCodegen()
```

```
Found 2 WholeStageCodegen subtrees.
== Subtree 1 / 2 (maxMethodCodeSize:3679; maxConstantPoolSize:1107(1.69% used); numInnerClasses:0) ==
...
== Subtree 2 / 2 (maxMethodCodeSize:5581; maxConstantPoolSize:882(1.35% used); numInnerClasses:0) ==
...
```

Disable this feature

```
scala> sql("SET spark.sql.codegen.aggregate.splitAggregateFunc.enabled=false")
scala> sql(query).debugCodegen()
```

```
Found 2 WholeStageCodegen subtrees.
== Subtree 1 / 2 (maxMethodCodeSize:8917; maxConstantPoolSize:957(1.46% used); numInnerClasses:0) ==
...
== Subtree 2 / 2 (maxMethodCodeSize:9862; maxConstantPoolSize:728(1.11% used); numInnerClasses:0) ==
...
```

Source: PR #20965

SPARK-21870

Seven Major Changes for SQL Performance

1. New EXPLAIN format
2. All type of join hints
3. Adaptive query execution
4. Dynamic partitioning pruning
5. Enhanced nested column pruning & pushdown
6. Improved aggregation code generation
7. New Scala and Java

Support New Versions of Languages

- Java 11 (the latest Long-Term-Support of OpenJDK from 2018 to 2026)
  - Further optimizations in HotSpot compiler
  - Improved G1GC (for large heap)
  - Experimental new ZGC (low latency)
- Scala 2.12 (released on 2016 Nov.)
  - Newly designed for leveraging Java 8 new features

NOTE: Other class libraries are also updated

SPARK-24417 & 25956

Takeaway

- Spark 3.0 improves SQL application performance
  1. New EXPLAIN format
  2. All type of join hints
  3. Adaptive query execution
  4. Dynamic partitioning pruning
  5. Enhanced nested column pruning & pushdown
  6. Improved aggregation code generation
  7. New Scala and Java

Please visit https://www.slideshare.net/ishizaki/ tomorrow if you want to see this slide again

Resources

- Introducing Apache Spark 3.0: Now available in Databricks Runtime 7.0
  - https://databricks.com/jp/blog/2020/06/18/introducing-apache-spark-3-0-now-available-in-databricks-runtime-7-0.html
- Now on Databricks: A Technical Preview of Databricks Runtime 7 Including a Preview of Apache Spark 3.0
  - https://databricks.com/blog/2020/05/13/now-on-databricks-a-technical-preview-of-databricks-runtime-7-including-a-preview-of-apache-spark-3-0.html
- Quick Overview of Upcoming Spark 3.0 (in Japanese)
  - https://www.slideshare.net/maropu0804/quick-overview-of-upcoming-spark-30

Resources…

- Madhukar’s Blog
  - https://blog.madhukaraphatak.com/
- Adaptive Query Execution: Speeding Up Spark SQL at Runtime
  - https://databricks.com/blog/2020/05/29/adaptive-query-execution-speeding-up-spark-sql-at-runtime.html
- Dynamic Partition Pruning in Apache Spark
  - https://databricks.com/session_eu19/dynamic-partition-pruning-in-apache-spark