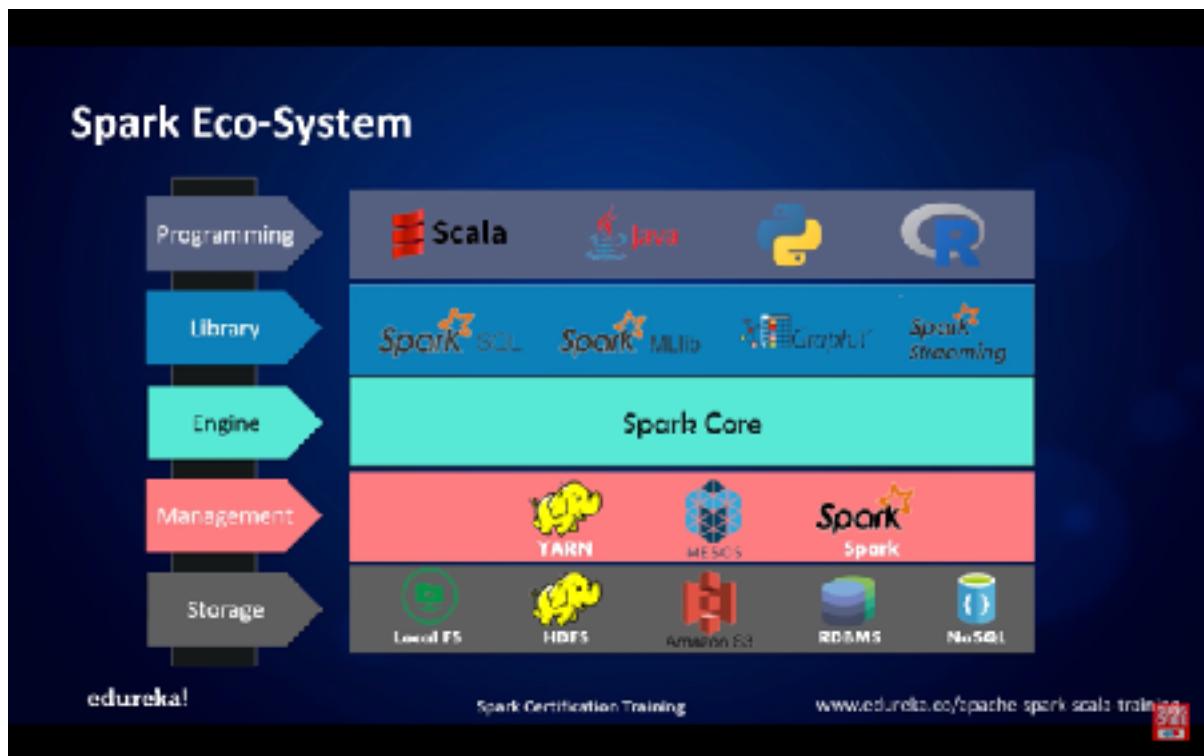
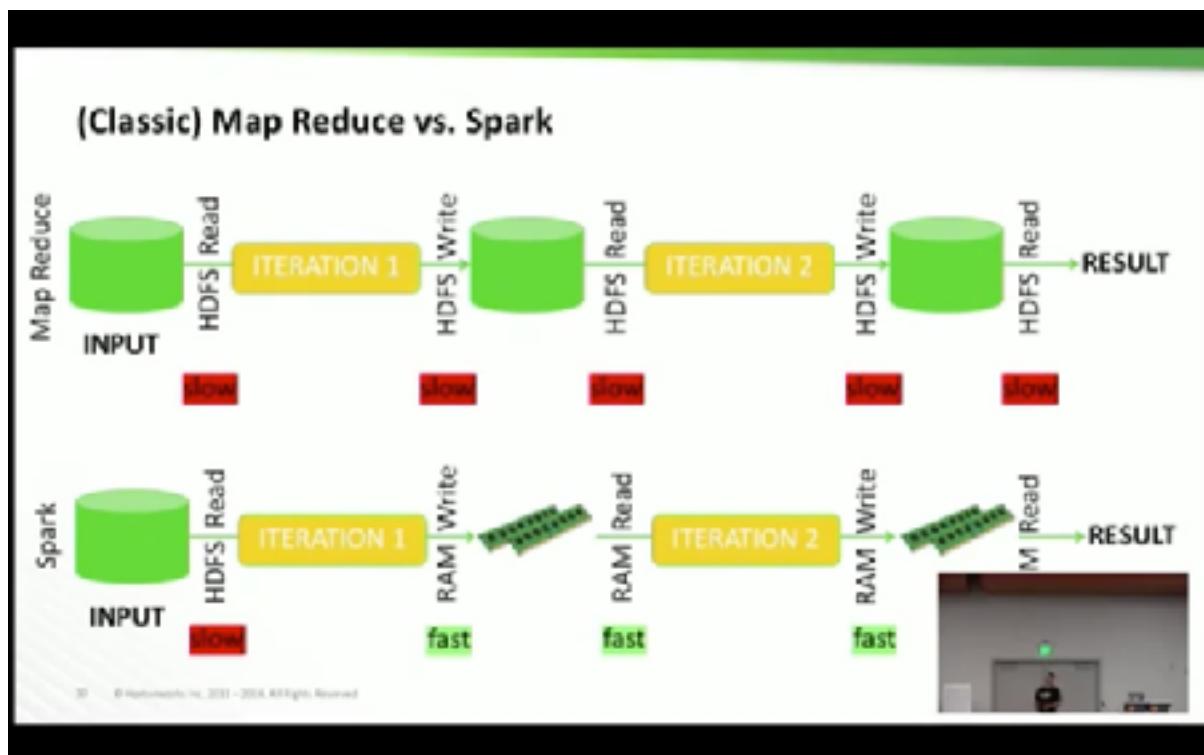


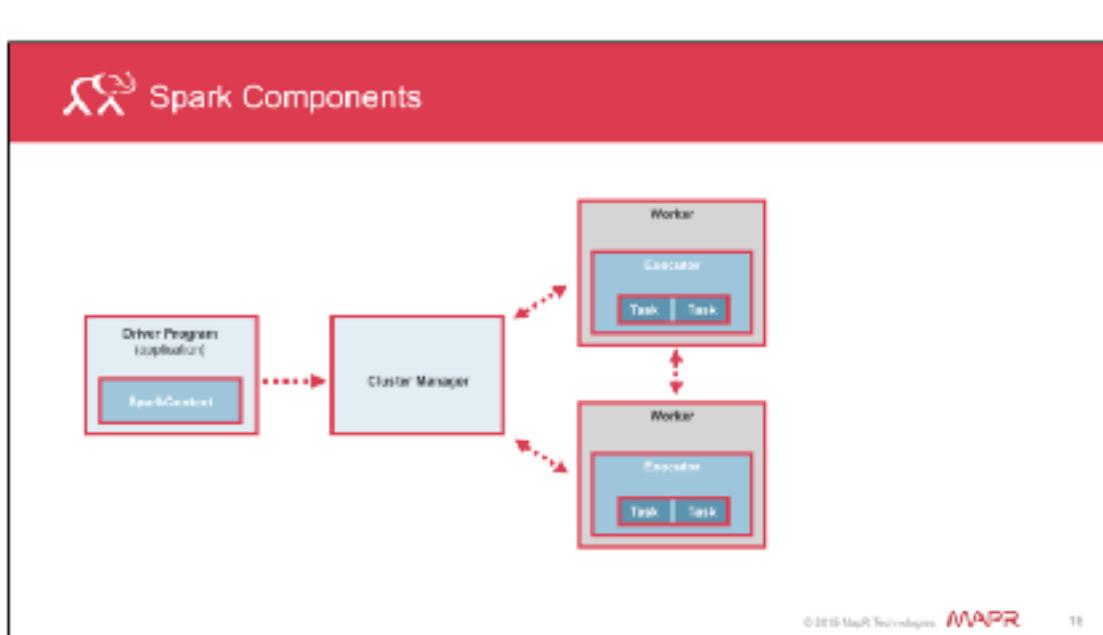
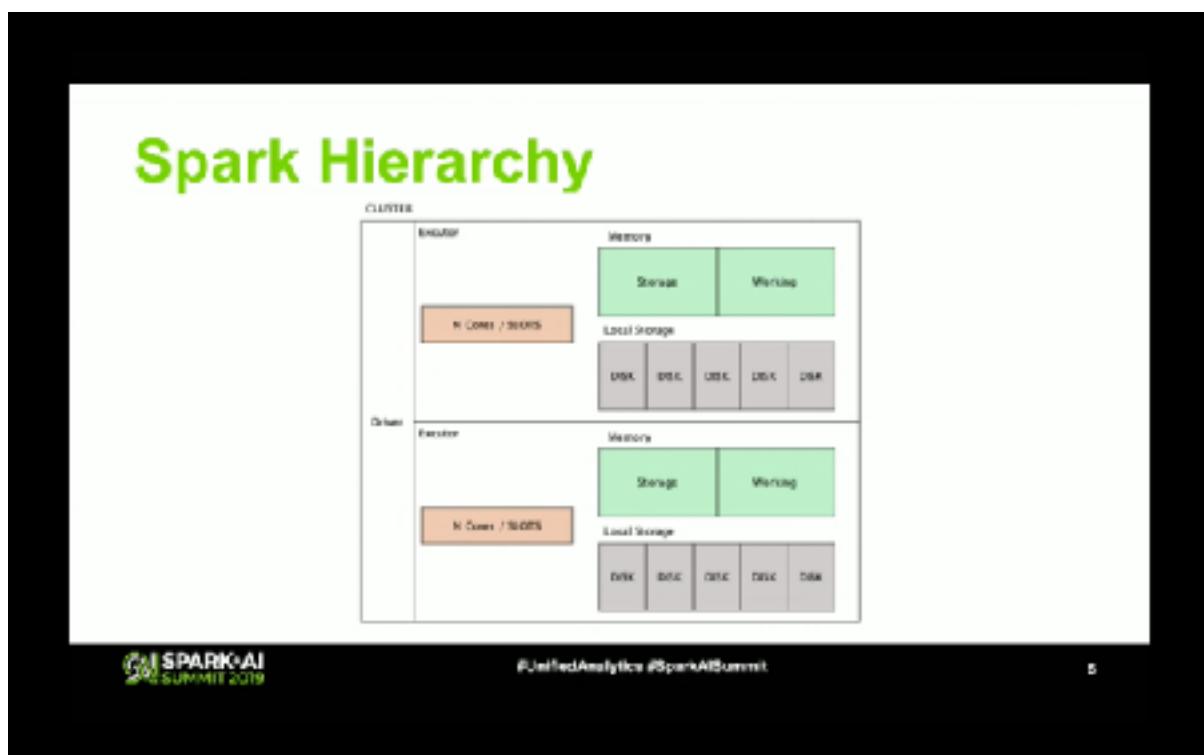
Spark Eco-System



Classic Diff. between MapReduce and Spark(MR vs Spark)



Spark Architecture:



A Spark cluster consists of two processes, a driver program and multiple workers nodes each running an executor process. The driver program runs on the driver machine, the worker programs run on cluster nodes or in local threads.

1. The first thing a program does is to create a `SparkContext` object. This tells Spark how and where to access a cluster
2. `SparkContext` connects to cluster manager. Cluster manager allocates resources across applications
3. Once connected, Spark acquires executors in the worker nodes (an executor is a process that runs computations and stores data for your application)
4. Jar or python files passed to the `SparkContext` are then sent to the executors.
5. `SparkContext` will then send the tasks for the executor to run.
6. The worker nodes can access data storage sources to ingest and output data as needed.

Spark internals:

Reference:

<https://www.youtube.com/watch?v=dmL0N3qfSc8>

Press Esc to exit full screen

Why understand internals?

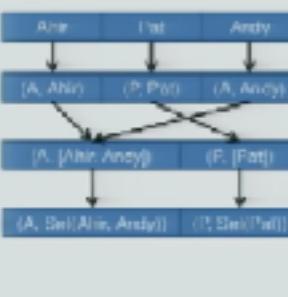
Goal: Find number of distinct names per "first letter"

```
sc.textFile("hdfs://names")  
    .map(name => (name.charAt(0), name))  
    .groupByKey()  
    .mapValues(names => names.toSet.size)  
    .collect()
```

Why understand internals?

Goal: Find number of distinct names per "first letter"

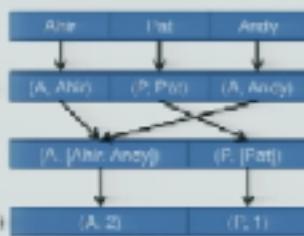
```
sc.textFile("hdfs://names")  
    .map(name => (name.charAt(0), name))  
    .groupByKey()  
    .mapValues(names => names.toSet.size)  
    .collect()
```



Why understand internals?

Goal: Find number of distinct names per "first letter"

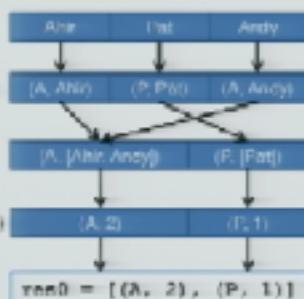
```
sc.textFile("hdfs://names")  
.map(name => (name.charAt(0), name))  
.groupByKey()  
.mapValues(names => names.toSet.size)  
.collect()
```



Why understand internals?

Goal: Find number of distinct names per "first letter"

```
sc.textFile("hdfs://names")  
.map(name => (name.charAt(0), name))  
.groupByKey()  
.mapValues(names => names.toSet.size)  
.collect()
```



Spark Execution Model

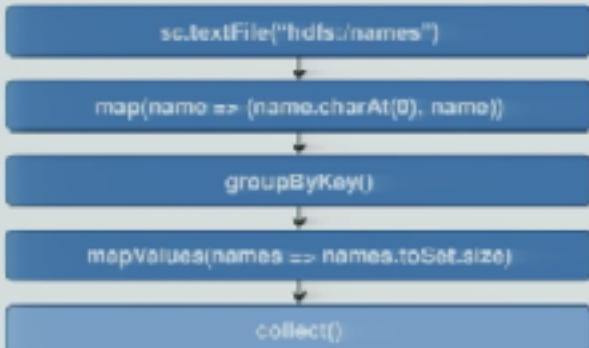
1. Create DAG of RDDs to represent computation
2. Create logical execution plan for DAG
3. Schedule and execute individual tasks



Step 1: Create RDDs

```
sc.textFile("hdfs://names")  
      ↓  
map(name => (name.charAt(0), name))  
      ↓  
groupByKey()  
      ↓  
mapValues(names => names.keySet.size)  
      ↓  
collect()
```

Step 1: Create RDDs



Step 1: Create RDDs



Step 2: Create execution plan

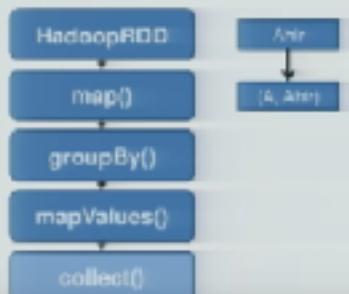
- Pipeline as much as possible
- Split into “**stages**” based on need to reorganize data



A Deeper Understanding of Spark Internals - Aaron Davidson (Databricks)

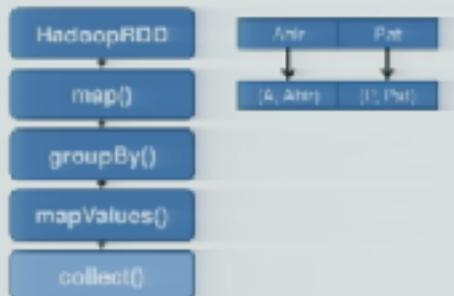
Step 2: Create execution plan

- Pipeline as much as possible
- Split into “**stages**” based on need to reorganize data



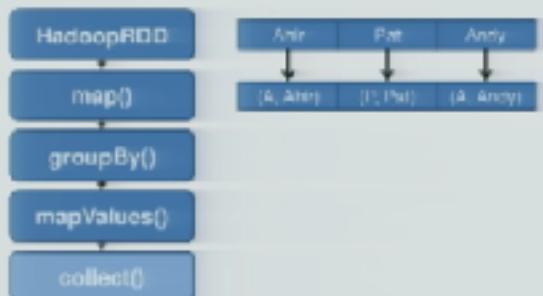
Step 2: Create execution plan

- Pipeline as much as possible
- Split into “**stages**” based on need to reorganize data



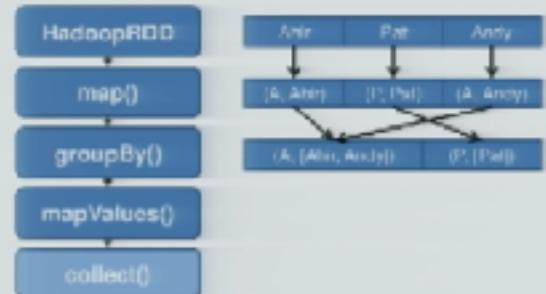
Step 2: Create execution plan

- Pipeline as much as possible
- Split into “**stages**” based on need to reorganize data



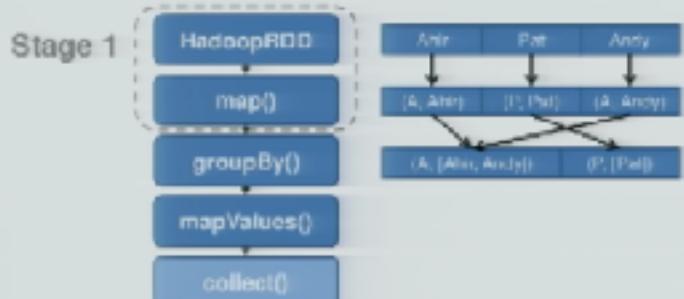
Step 2: Create execution plan

- Pipeline as much as possible
- Split into “**stages**” based on need to reorganize data



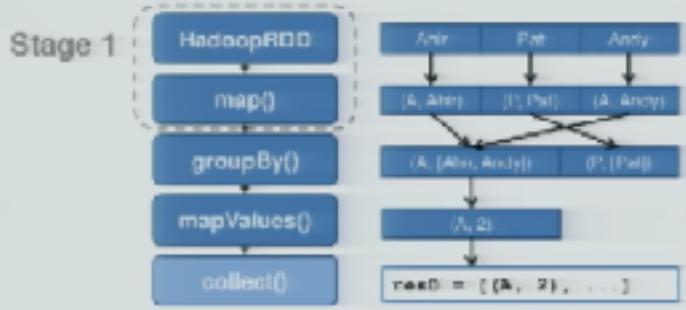
Step 2: Create execution plan

- Pipeline as much as possible
- Split into “**stages**” based on need to reorganize data



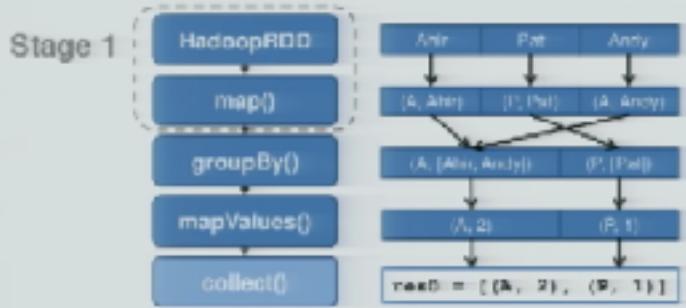
Step 2: Create execution plan

- Pipeline as much as possible
- Split into “**stages**” based on need to reorganize data



Step 2: Create execution plan

- Pipeline as much as possible
- Split into “**stages**” based on need to reorganize data



Step 3: Schedule tasks

- Split each stage into **tasks**

Step 3: Schedule tasks

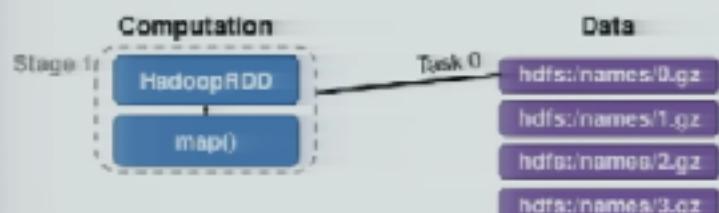
- Split each stage into **tasks**
- A task is data + computation



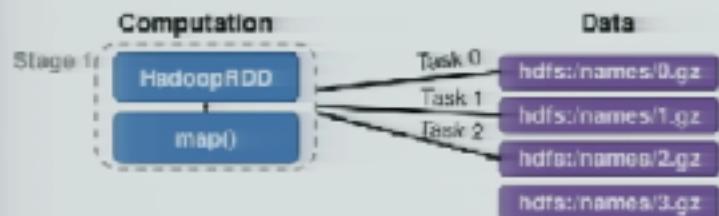
Step 3: Schedule tasks



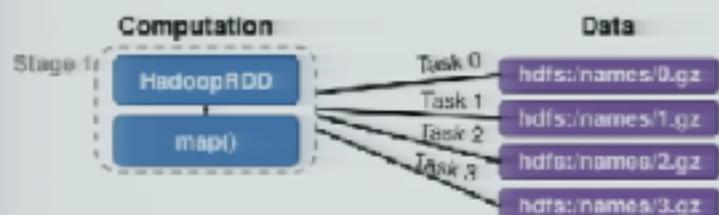
Step 3: Schedule tasks



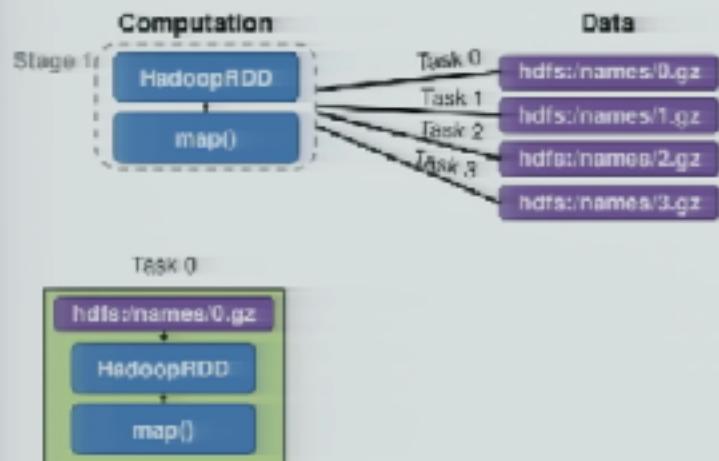
Step 3: Schedule tasks



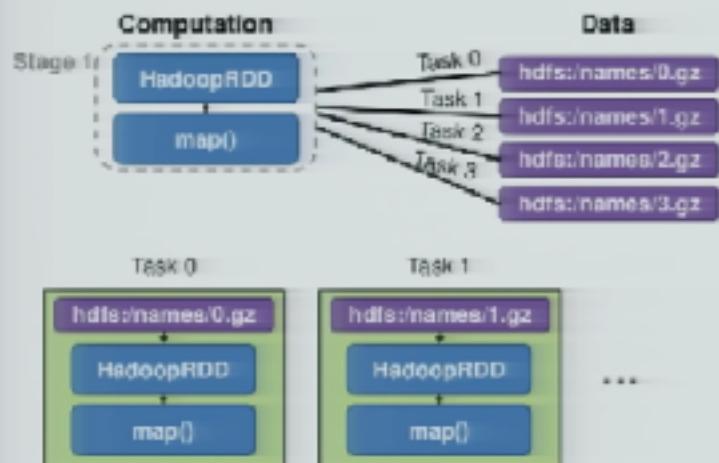
Step 3: Schedule tasks



Step 3: Schedule tasks



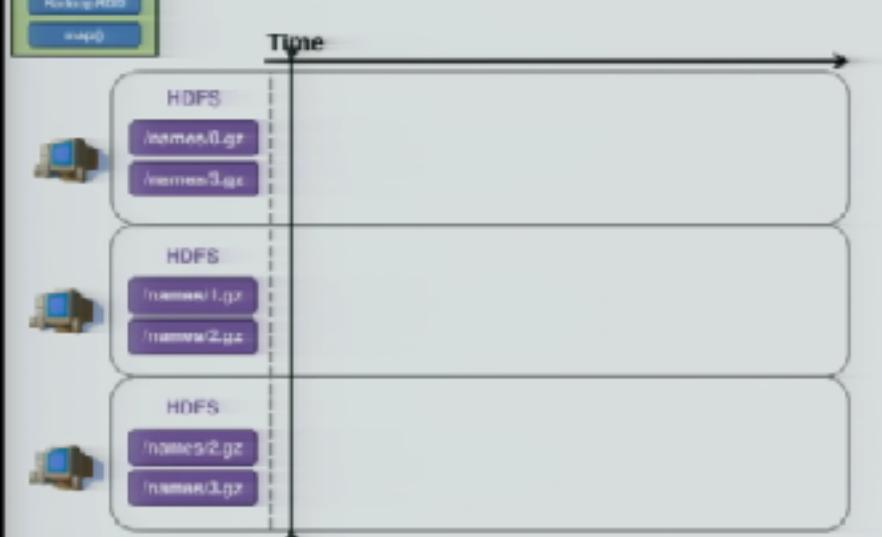
Step 3: Schedule tasks



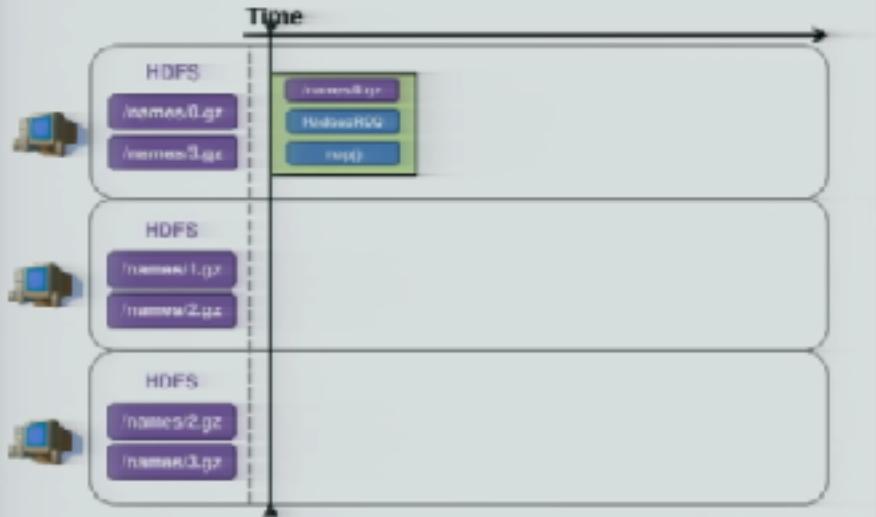
Step 3: Schedule tasks



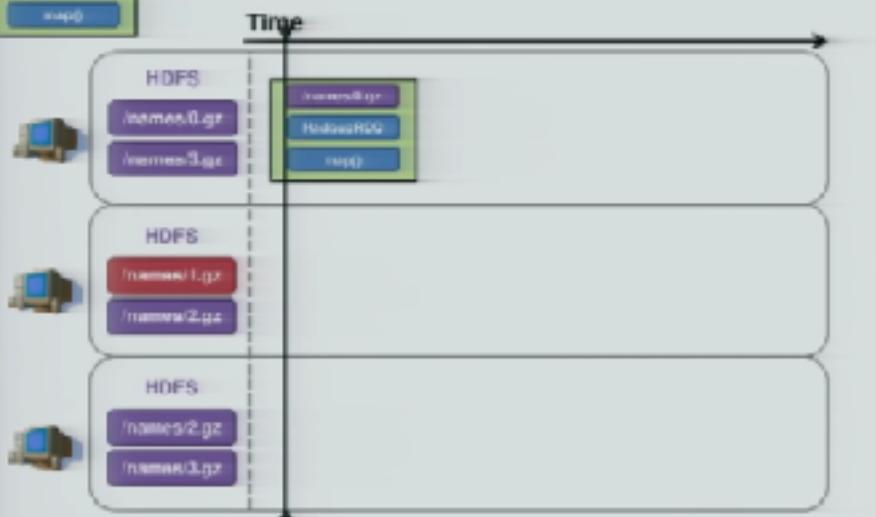
Step 3: Schedule tasks



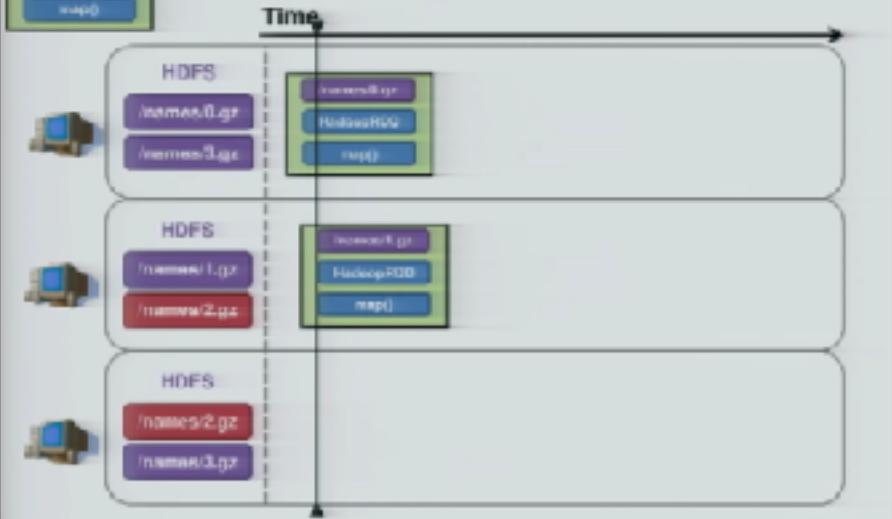
Step 3: Schedule tasks



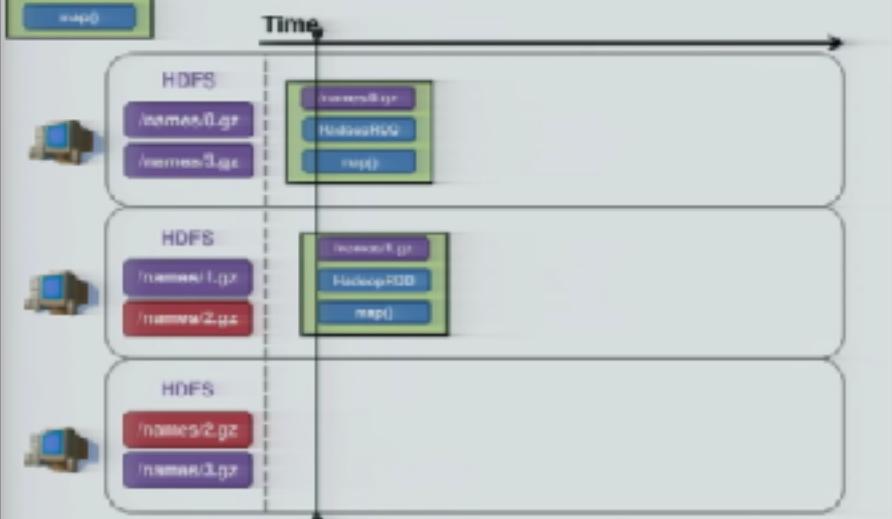
Step 3: Schedule tasks



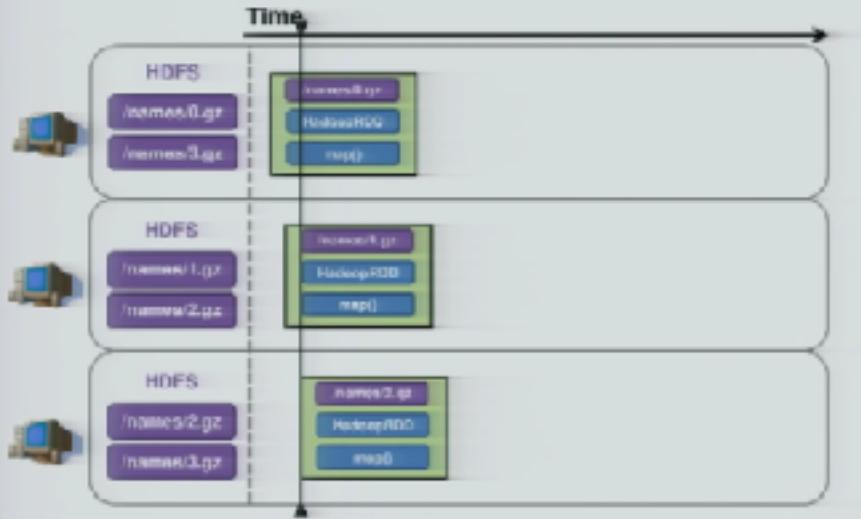
Step 3: Schedule tasks



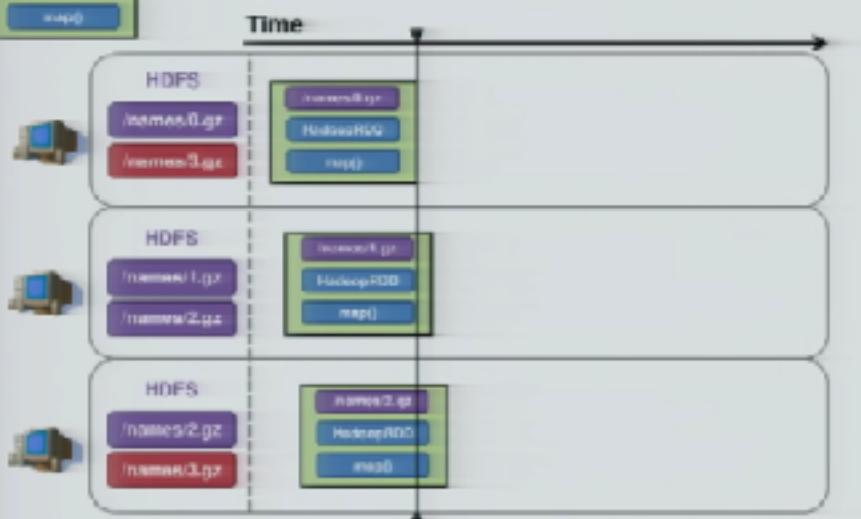
Step 3: Schedule tasks



Step 3: Schedule tasks



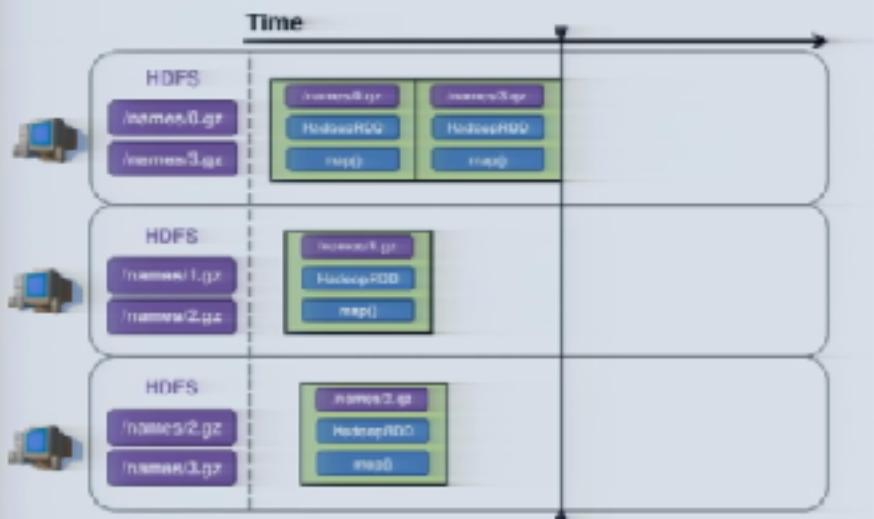
Step 3: Schedule tasks



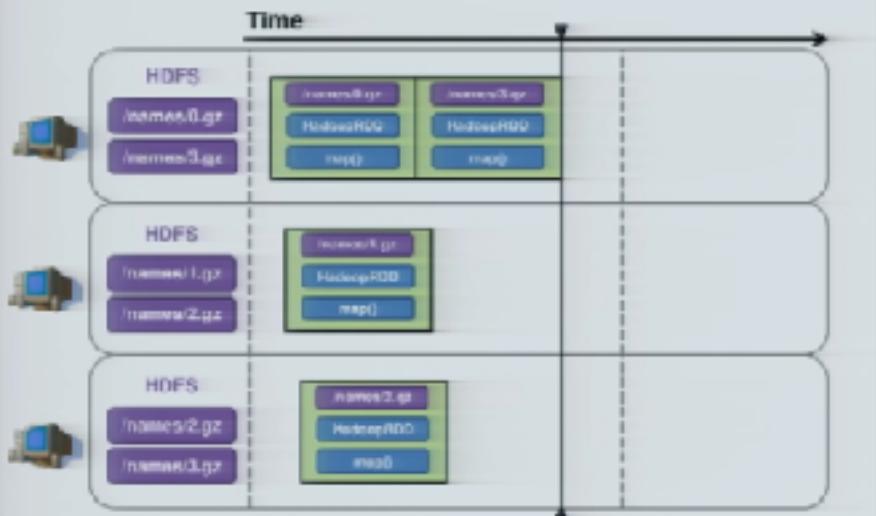
Step 3: Schedule tasks



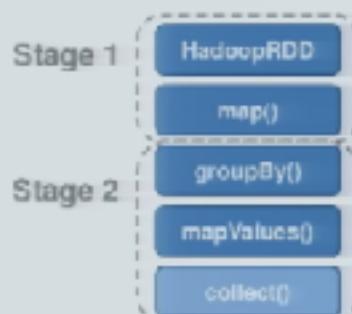
Step 3: Schedule tasks



Step 3: Schedule tasks

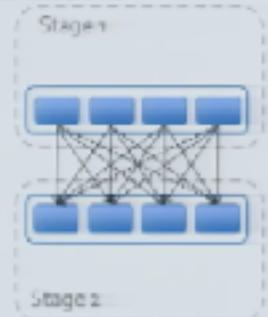


The Shuffle



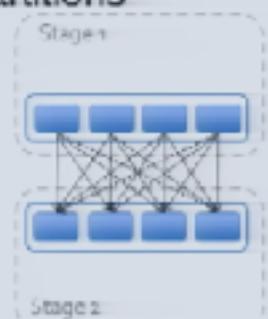
The Shuffle

- Redistributes data among partitions



The Shuffle

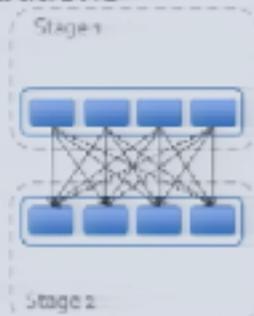
- Redistributes data among partitions
- Hash keys into buckets



The same hashCode contains keys will be moving to same partition/bucket

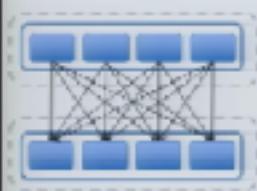
The Shuffle

- Redistributions data among partitions
- Hash keys into buckets
- Optimizations:
 - Avoided when possible, if data is already properly partitioned
 - Partial aggregation reduces data movement



The Shuffle





The Shuffle

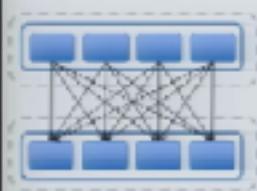
- Pull-based, not push-based



The Shuffle

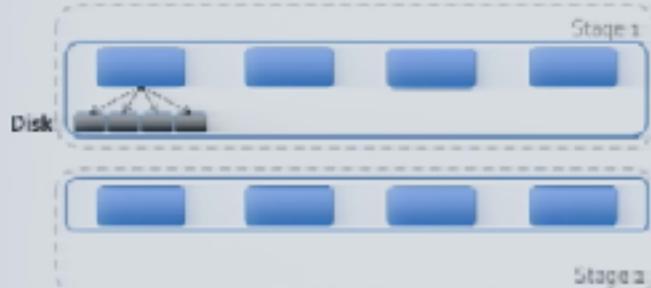
- Pull-based, not push-based
- Write intermediate files to disk





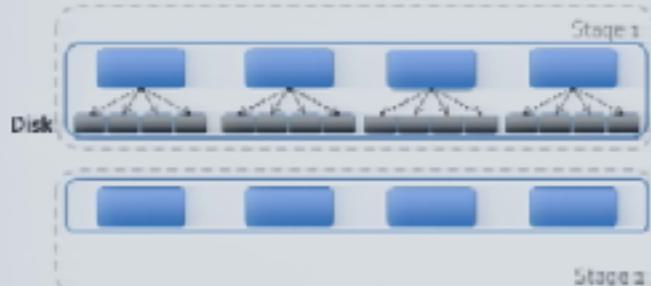
The Shuffle

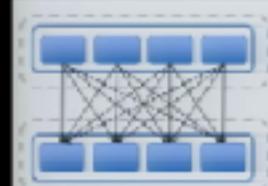
- Pull-based, not push-based
- Write intermediate files to disk



The Shuffle

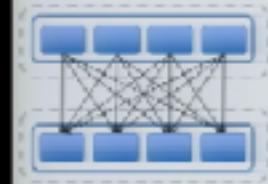
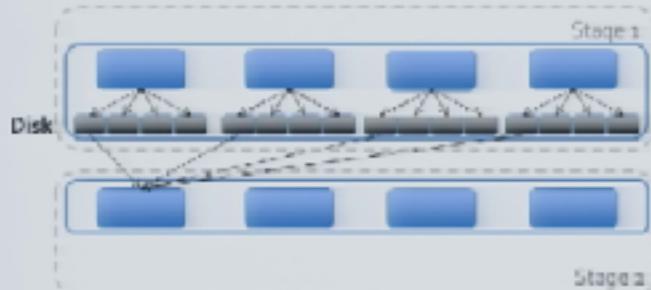
- Pull-based, not push-based
- Write intermediate files to disk





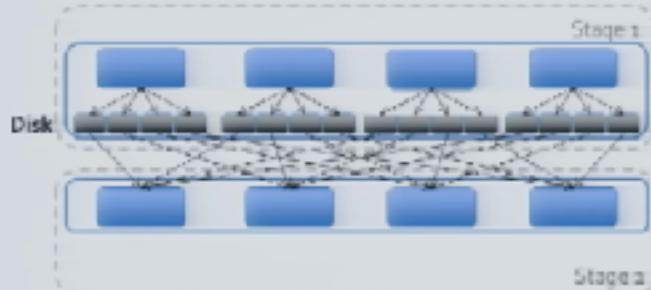
The Shuffle

- Pull-based, not push-based
- Write intermediate files to disk



The Shuffle

- Pull-based, not push-based
- Write intermediate files to disk



Execution of a groupBy()

- Build hash map within each partition

A => [Arsalan, Aaron, Andrew, Andrew, Andy, Ahir, Ali, ...]
E => [Erin, Earl, Ed, ...]

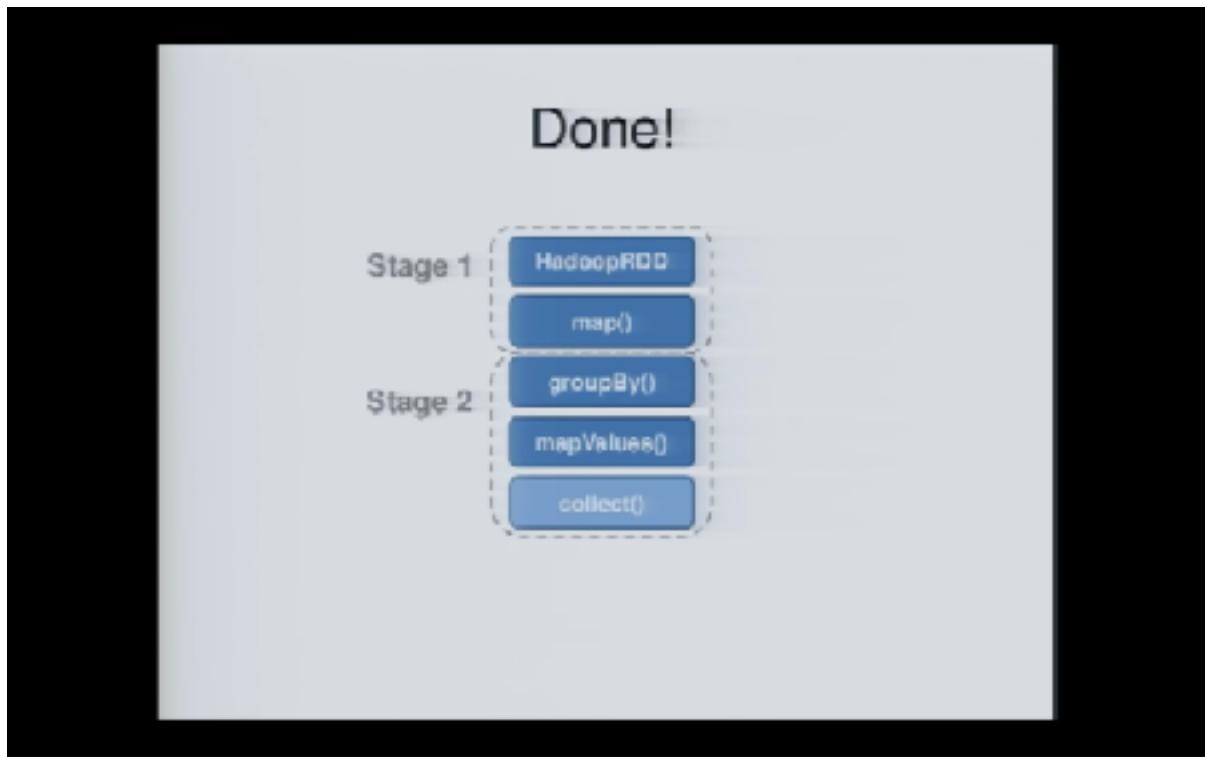
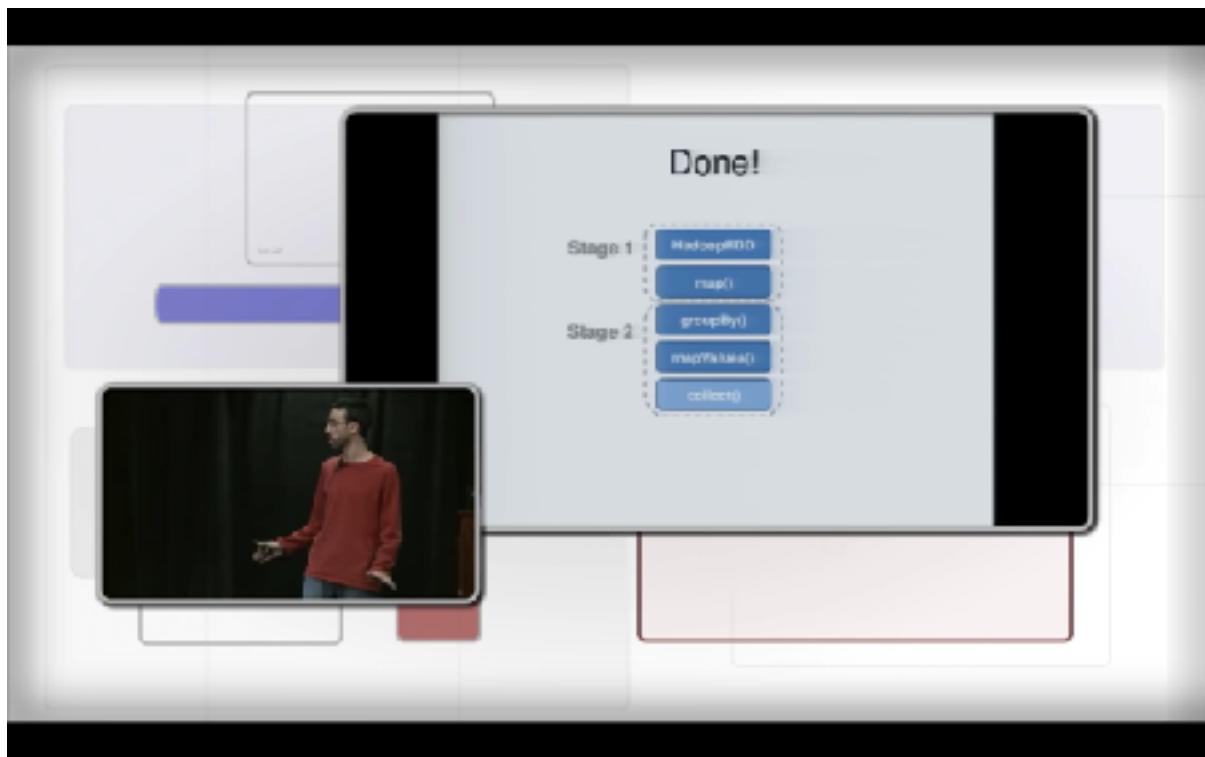
The image shows a video player interface. On the left, there is a small video thumbnail of a man in a red shirt speaking. The main area displays a presentation slide with a black header bar. The slide has a white background and contains the following content:

Execution of a groupBy()

- Build hash map within each partition

A => [Arsalan, Aaron, Andrew, Andrew, Andy, Ahir, Ali, ...]
E => [Erin, Earl, Ed, ...]

- Note: Can spill across keys, but a single key-value pair must fit in memory



What went wrong?

- Too few partitions to get good concurrency
- Large per-key groupBy()
- Shipped all data across the cluster



Common Issue checklist

1. Ensure enough partitions for concurrency
2. Minimize memory consumption (esp. of sorting and large keys in groupBys)
3. Minimize amount of data shuffled
4. Know the standard library



Common issue checklist

1. Ensure enough partitions for concurrency
2. Minimize memory consumption (esp. of sorting and large keys in groupBys)
3. Minimize amount of data shuffled
4. Know the standard library

1 & 2 are about tuning number of partitions!

Importance of Partition Tuning

- Main issue: too few partitions
 - Less concurrency
 - More susceptible to data skew
 - Increased memory pressure for groupBy, reduceByKey, sortByKey, etc.
- Secondary issue: too many partitions
- Need “reasonable number” of partitions
 - Commonly between 100 and 10,000 partitions
 - Lower bound: At least ~2x number of cores in cluster
 - Upper bound: Ensure tasks take at least 100ms

Susceptible : capable / admitting of

Memory Problems

- Symptoms:
 - Inexplicably bad performance
 - Inexplicable executor/machine failures
(can indicate too many shuffle files too)
- Diagnosis:
 - Set spark.executor.extraJavaOptions to include
 - -XX:+PrintGCDetails
 - -XX:+HeapDumpOnOutOfMemoryError
 - Check dmesg for oom-killer logs

A Deeper Understanding of Spark Internals - Aaron Davidson (Databricks)

Memory Problems

- Symptoms:
 - Inexplicably bad performance
 - Inexplicable executor/machine failures
(can indicate too many shuffle files too)
- Diagnosis:
 - Set spark.executor.extraJavaOptions to include
 - -XX:+PrintGCDetails
 - -XX:+HeapDumpOnOutOfMemoryError
 - Check dmesg for oom-killer logs
- Resolution:
 - Increase spark.executor.memory
 - Increase number of partitions
 - Re-evaluate program structure (!)

Inexplicable: unable to be explained

Fixing our mistakes

```
sc.textFile("/tmp/names")
    .repartition(6)
    .distinct()
    .map(name => (name.charAt(0), name))
    .groupByKey()
    .mapValues(names => names.toSet.size)
    .collect()
```

1. Ensure enough partitions for concurrency
2. Minimize memory consumption (esp. of large groupBys and sorting)
3. Minimize data shuffle
4. Know the standard library

Adding repetition & distinct:

Original:

```
scala> sc.textFile("/tmp/names.txt").map(name =>(name.charAt(0),name)).groupByKey().mapValues(names =>
names.toSet.size).collect().foreach(println)
```

Modified:

```
scala> sc.textFile("/tmp/names.txt").repartition(6).distinct().map(name =>(name.charAt(0),name)).groupByKey().mapValues(names => names.toSet.size).collect().foreach(println)
```

Fixing our mistakes

```
sc.textFile("/tmp/names")
  .repartition(6)
  .distinct()
  .map(name => (name.charAt(0), name))
  .groupByKey()
  .mapValues(names => names.size)
  .collect()
```

1. Ensure enough partitions for concurrency
2. Minimize memory consumption (esp. of large groupBys and sorting)
3. Minimize data shuffle
4. Know the standard library

```
scala> sc.textFile("/tmp/names.txt").distinct(numPartitions=6).map(name =>(name.charAt(0),name)).groupByKey().mapValues(names => names.size).collect().foreach(println)
```

Fixing our mistakes

```
sc.textFile("hdfs://names")
  .distinct(numPartitions = 6)
  .map(name => (name.charAt(0), name))
  .groupByKey()
  .mapValues(values => names.size)
  .collect()
```

1. Ensure enough partitions for concurrency
2. Minimize memory consumption (esp. of large groupBys and sorting)
3. Minimize data shuffle
4. Know the standard library

Fixing our mistakes

```
sc.textFile("hdfs://names")
  .distinct(numPartitions = 6)
  .map(name => (name.charAt(0), 1))
  .reduceByKey(_ + _)
  .collect()
```

1. Ensure enough partitions for concurrency
2. Minimize memory consumption (esp. of large groupBys and sorting)
3. Minimize data shuffle
4. Know the standard library

Fixing our mistakes

```
sc.textFile("hdfs://names")
  .distinct(numPartitions = 6)
  .map(name => (name.charAt(0), 1))
  .reduceByKey(_ + _)
  .collect()
```

Original:

```
sc.textfile("hdfs://names")
  .map(name => (name.charAt(0), name))
  .groupByKey()
  .mapValues { names => names.toSet.size }
  .collect()
```

Spark Joins:

<https://www.youtube.com/watch?v=fp53QhSfQcl&t=1146s>

Spark SQL Joins

```
SELECT ...  
FROM TABLE A  
JOIN TABLE B  
ON A.KEY1 = B.KEY2
```

Topics Covered Today

Basic Joins:

- Shuffle Hash Join
 - Troubleshooting
- Broadcast Hash Join
- Cartesian Join

Special Cases:

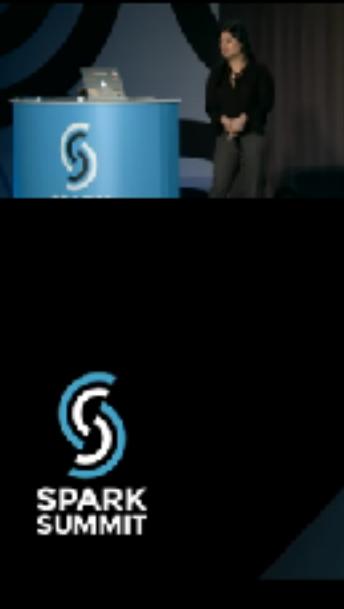
- Theta Join
- One to Many Join

Shuffle Hash Join

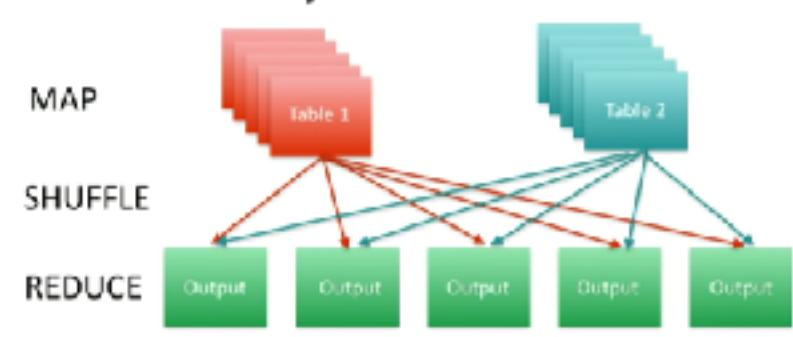
A **Shuffle Hash Join** is the most basic type of join, and goes back to Map Reduce Fundamentals.

- **Map** through two different data frames/tables.
- Use the fields in the join condition as the **output key**.
- **Shuffle** both datasets by the output key.
- In the **reduce** phase, join the two datasets now any rows of both tables with the same keys are on the same machine and are sorted.

[Shuffle Hash \(SH\) Join](#)



Shuffle Hash Join



MAP

SHUFFLE

REDUCE

Output

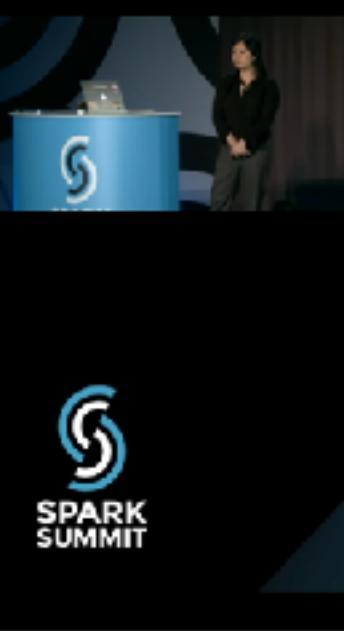
Output

Output

Output

Output

◆ databricks



Shuffle Hash Join Performance

Works best when the DF's:

- Distribute evenly with the key you are joining on.
- Have an adequate number of keys for parallelism.

◆ databricks

Shuffle Hash Join Performance

Works best when the DF's:

- Distribute evenly with the key you are joining on.
- Have an adequate number of keys for parallelism.

```
join_rdd = sqlContext.sql("select *  
    FROM people_in_the_us  
    JOIN states  
    ON people_in_the_us.state = states.name")
```

◆ databricks

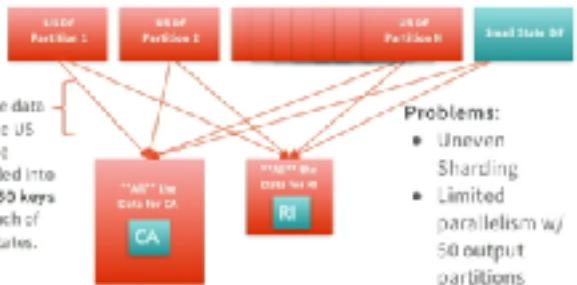
Uneven Sharding & Limited Parallelism,

A larger Spark Cluster will not solve these problems!

◆ databricks



Uneven Sharding & Limited Parallelism,



- Uneven Sharding
- Limited parallelism w/ 50 output partitions

A larger Spark Cluster will not solve these problems!

 databricks



Broadcast Hash (BH) Join



Uneven Sharding & Limited Parallelism,



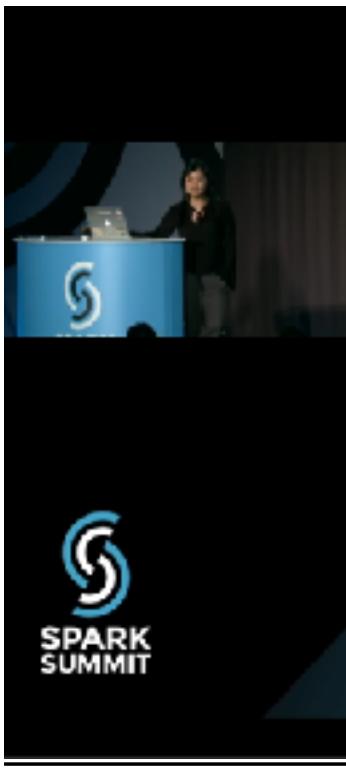
Broadcast Hash Join can address this problem if one DF is small enough to fit in memory.

 databricks



Left Outer Join

All keys will get as output in the final output table

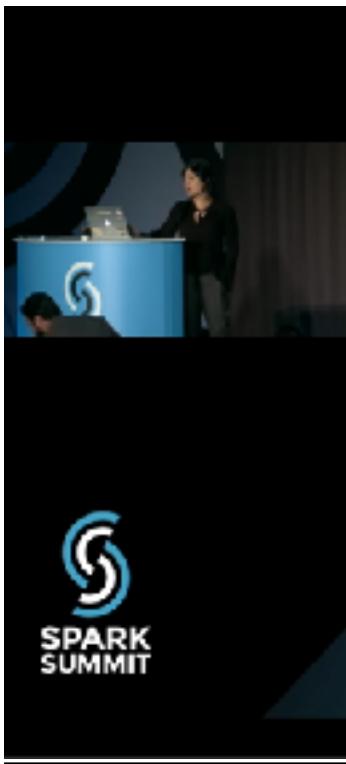


More Performance Considerations

```
join_rdd = sqlContext.sql("select *\n    FROM people_in_california\n    LEFT JOIN all_the_people_in_the_world\n    ON people_in_california.id =\n        all_the_people_in_the_world.id")
```

Final output keys = # of people in CA, so don't need a huge Spark cluster, right?

◆ databricks

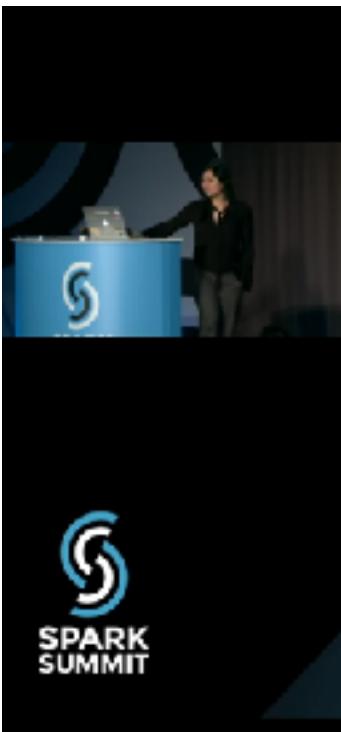


Left Join - Shuffle Step



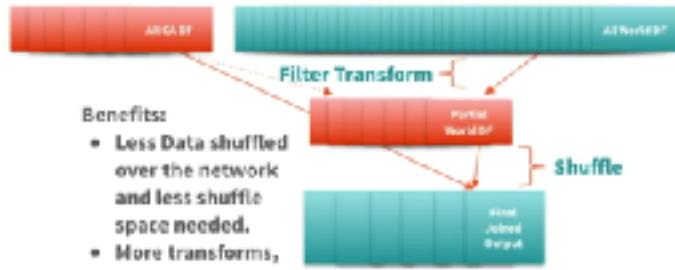
The Size of the Spark Cluster to run this job is limited by the Large table rather than the Medium Sized Table.

◆ databricks



A Better Solution

Filter the World DF for only entries that match the CA ID

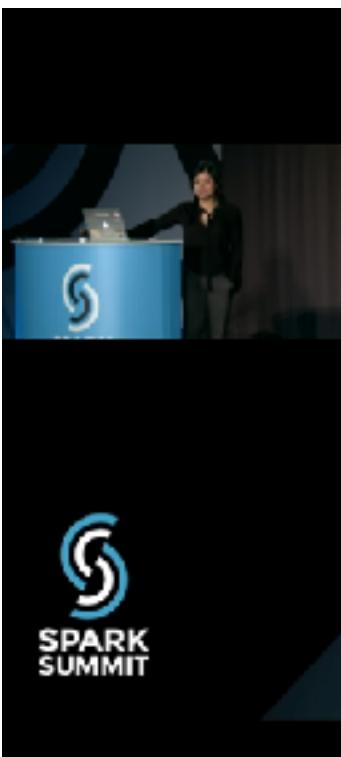


Benefits

- Less Data shuffled over the network and less shuffle space needed.
- More transforms, but still faster.

Final Result

databricks

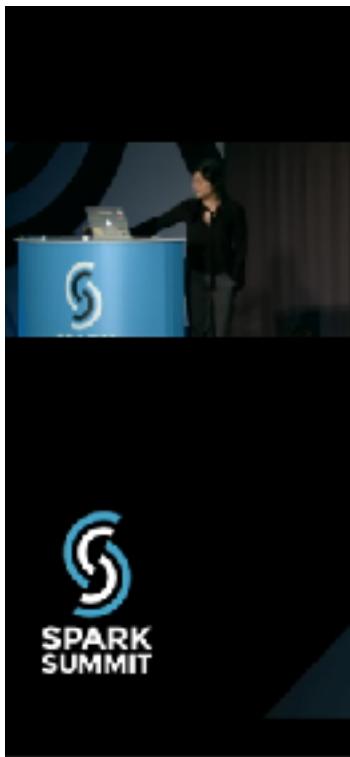


What's the Tipping Point for Huge?

- Can't tell you.

You should understand your data and its unique properties in order to best optimize your Spark Job.

databricks

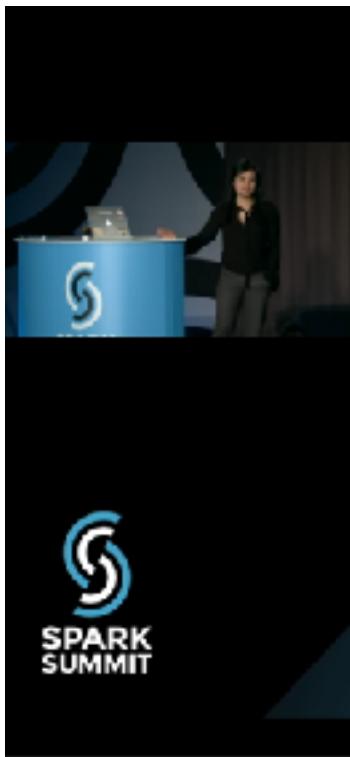


What's the Tipping Point for Huge?

- Can't tell you.
- There aren't always strict rules for optimizing.
- If you were only considering two small columns from the World RDD in Parquet format, the filtering step may not be worth it.

You should understand your data and its unique properties in order to best optimize your Spark Job.

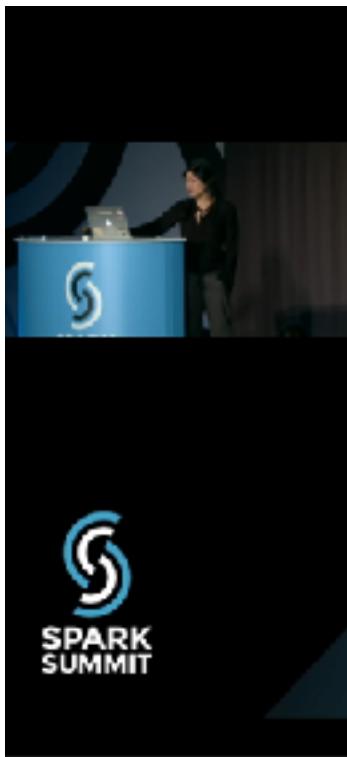
 databricks



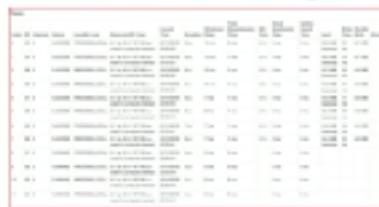
In Practice: Detecting Shuffle Problems

Task ID	Stage ID	Partition ID	Input File	Output File	Progress (%)	Memory Usage (MB)	Shuffle Read (MB/s)	Shuffle Write (MB/s)	Local Read (MB/s)	Local Write (MB/s)	Network In (MB/s)	Network Out (MB/s)
1	1	0	file1	file1	0	100	0	0	0	0	0	0
2	1	1	file1	file1	0	100	0	0	0	0	0	0
3	1	2	file1	file1	0	100	0	0	0	0	0	0
4	1	3	file1	file1	0	100	0	0	0	0	0	0
5	1	4	file1	file1	0	100	0	0	0	0	0	0
6	1	5	file1	file1	0	100	0	0	0	0	0	0
7	1	6	file1	file1	0	100	0	0	0	0	0	0
8	1	7	file1	file1	0	100	0	0	0	0	0	0
9	1	8	file1	file1	0	100	0	0	0	0	0	0
10	1	9	file1	file1	0	100	0	0	0	0	0	0
11	1	10	file1	file1	0	100	0	0	0	0	0	0
12	1	11	file1	file1	0	100	0	0	0	0	0	0
13	1	12	file1	file1	0	100	0	0	0	0	0	0
14	1	13	file1	file1	0	100	0	0	0	0	0	0
15	1	14	file1	file1	0	100	0	0	0	0	0	0
16	1	15	file1	file1	0	100	0	0	0	0	0	0
17	1	16	file1	file1	0	100	0	0	0	0	0	0
18	1	17	file1	file1	0	100	0	0	0	0	0	0
19	1	18	file1	file1	0	100	0	0	0	0	0	0
20	1	19	file1	file1	0	100	0	0	0	0	0	0
21	1	20	file1	file1	0	100	0	0	0	0	0	0
22	1	21	file1	file1	0	100	0	0	0	0	0	0
23	1	22	file1	file1	0	100	0	0	0	0	0	0
24	1	23	file1	file1	0	100	0	0	0	0	0	0
25	1	24	file1	file1	0	100	0	0	0	0	0	0
26	1	25	file1	file1	0	100	0	0	0	0	0	0
27	1	26	file1	file1	0	100	0	0	0	0	0	0
28	1	27	file1	file1	0	100	0	0	0	0	0	0
29	1	28	file1	file1	0	100	0	0	0	0	0	0
30	1	29	file1	file1	0	100	0	0	0	0	0	0
31	1	30	file1	file1	0	100	0	0	0	0	0	0
32	1	31	file1	file1	0	100	0	0	0	0	0	0
33	1	32	file1	file1	0	100	0	0	0	0	0	0
34	1	33	file1	file1	0	100	0	0	0	0	0	0
35	1	34	file1	file1	0	100	0	0	0	0	0	0
36	1	35	file1	file1	0	100	0	0	0	0	0	0
37	1	36	file1	file1	0	100	0	0	0	0	0	0
38	1	37	file1	file1	0	100	0	0	0	0	0	0
39	1	38	file1	file1	0	100	0	0	0	0	0	0
40	1	39	file1	file1	0	100	0	0	0	0	0	0
41	1	40	file1	file1	0	100	0	0	0	0	0	0
42	1	41	file1	file1	0	100	0	0	0	0	0	0
43	1	42	file1	file1	0	100	0	0	0	0	0	0
44	1	43	file1	file1	0	100	0	0	0	0	0	0
45	1	44	file1	file1	0	100	0	0	0	0	0	0
46	1	45	file1	file1	0	100	0	0	0	0	0	0
47	1	46	file1	file1	0	100	0	0	0	0	0	0
48	1	47	file1	file1	0	100	0	0	0	0	0	0
49	1	48	file1	file1	0	100	0	0	0	0	0	0
50	1	49	file1	file1	0	100	0	0	0	0	0	0
51	1	50	file1	file1	0	100	0	0	0	0	0	0
52	1	51	file1	file1	0	100	0	0	0	0	0	0
53	1	52	file1	file1	0	100	0	0	0	0	0	0
54	1	53	file1	file1	0	100	0	0	0	0	0	0
55	1	54	file1	file1	0	100	0	0	0	0	0	0
56	1	55	file1	file1	0	100	0	0	0	0	0	0
57	1	56	file1	file1	0	100	0	0	0	0	0	0
58	1	57	file1	file1	0	100	0	0	0	0	0	0
59	1	58	file1	file1	0	100	0	0	0	0	0	0
60	1	59	file1	file1	0	100	0	0	0	0	0	0
61	1	60	file1	file1	0	100	0	0	0	0	0	0
62	1	61	file1	file1	0	100	0	0	0	0	0	0
63	1	62	file1	file1	0	100	0	0	0	0	0	0
64	1	63	file1	file1	0	100	0	0	0	0	0	0
65	1	64	file1	file1	0	100	0	0	0	0	0	0
66	1	65	file1	file1	0	100	0	0	0	0	0	0
67	1	66	file1	file1	0	100	0	0	0	0	0	0
68	1	67	file1	file1	0	100	0	0	0	0	0	0
69	1	68	file1	file1	0	100	0	0	0	0	0	0
70	1	69	file1	file1	0	100	0	0	0	0	0	0
71	1	70	file1	file1	0	100	0	0	0	0	0	0
72	1	71	file1	file1	0	100	0	0	0	0	0	0
73	1	72	file1	file1	0	100	0	0	0	0	0	0
74	1	73	file1	file1	0	100	0	0	0	0	0	0
75	1	74	file1	file1	0	100	0	0	0	0	0	0
76	1	75	file1	file1	0	100	0	0	0	0	0	0
77	1	76	file1	file1	0	100	0	0	0	0	0	0
78	1	77	file1	file1	0	100	0	0	0	0	0	0
79	1	78	file1	file1	0	100	0	0	0	0	0	0
80	1	79	file1	file1	0	100	0	0	0	0	0	0
81	1	80	file1	file1	0	100	0	0	0	0	0	0
82	1	81	file1	file1	0	100	0	0	0	0	0	0
83	1	82	file1	file1	0	100	0	0	0	0	0	0
84	1	83	file1	file1	0	100	0	0	0	0	0	0
85	1	84	file1	file1	0	100	0	0	0	0	0	0
86	1	85	file1	file1	0	100	0	0	0	0	0	0
87	1	86	file1	file1	0	100	0	0	0	0	0	0
88	1	87	file1	file1	0	100	0	0	0	0	0	0
89	1	88	file1	file1	0	100	0	0	0	0	0	0
90	1	89	file1	file1	0	100	0	0	0	0	0	0
91	1	90	file1	file1	0	100	0	0	0	0	0	0
92	1	91	file1	file1	0	100	0	0	0	0	0	0
93	1	92	file1	file1	0	100	0	0	0	0	0	0
94	1	93	file1	file1	0	100	0	0	0	0	0	0
95	1	94	file1	file1	0	100	0	0	0	0	0	0
96	1	95	file1	file1	0	100	0	0	0	0	0	0
97	1	96	file1	file1	0	100	0	0	0	0	0	0
98	1	97	file1	file1	0	100	0	0	0	0	0	0
99	1	98	file1	file1	0	100	0	0	0	0	0	0
100	1	99	file1	file1	0	100	0	0	0	0	0	0

 databricks



In Practice: Detecting Shuffle Problems

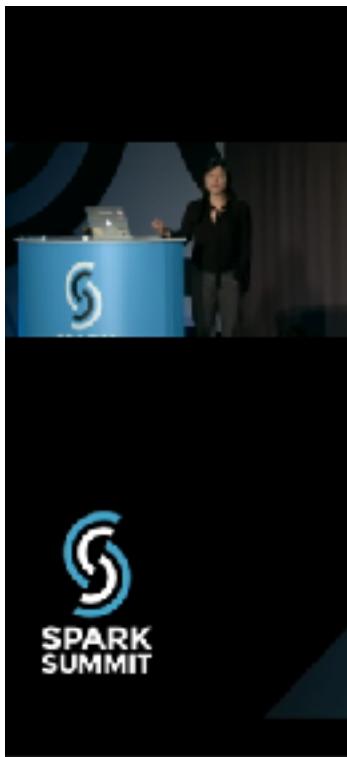


Check the Spark UI
pages for task
level detail about
your Spark job.

Things to Look for:

- Tasks that take much longer to run than others.
- Speculative tasks that are launching.
- Shards that have a lot more input or shuffle output.

databricks



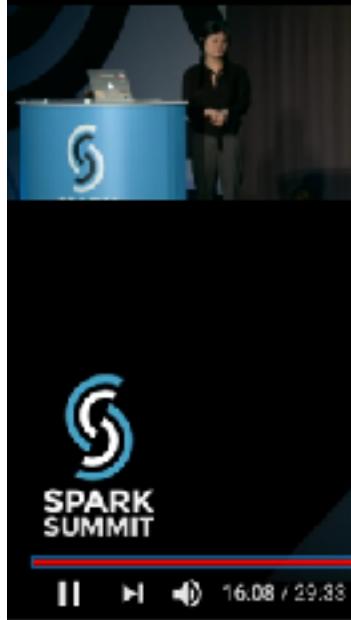
Broadcast Hash Join

Optimization: When one of the DF's is small enough
to fit in memory on a single machine.



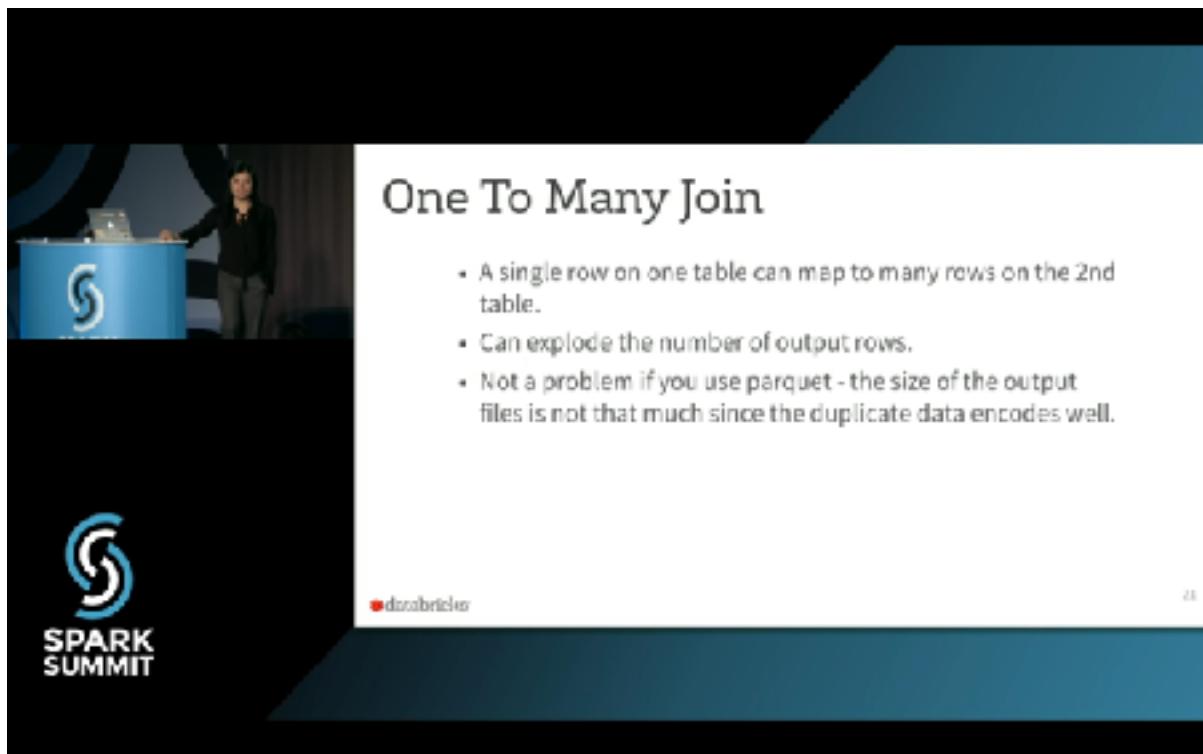
databricks

28

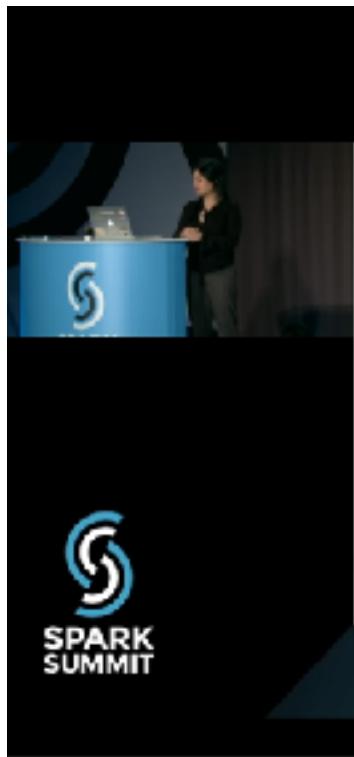


Broadcast Hash Join

- Often optimal over Shuffle Hash Join.
- Use "**explain**" to determine if the Spark SQL catalyst has chosen Broadcast Hash Join.
- Should be automatic for many Spark SQL tables, may need to provide hints for other types.



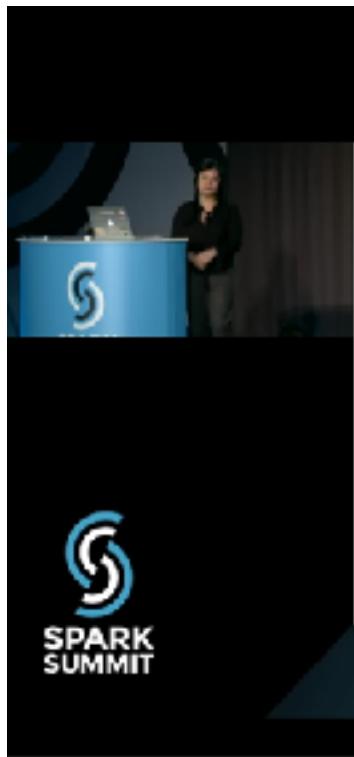
[Theta Join](#)

A photograph of a person speaking at a blue podium. The podium has a white "S" logo on it. To the right of the speaker is a presentation slide titled "Theta Join".

The slide has a dark background with a teal header bar. The title "Theta Join" is in a large, light-colored font. Below the title is a bulleted list:

- Spark SQL consider each keyA against each keyB in the example above and loop to see if the theta condition is met.
- Better Solution - create buckets for keyA and KeyB can be matched on.

A small red "databrickles" watermark is visible in the bottom left corner of the slide. The overall background of the slide is dark with geometric shapes.

A photograph of a person speaking at a blue podium. The podium has a white "S" logo on it. To the right of the speaker is a presentation slide titled "Theta Join".

The slide has a dark background with a teal header bar. The title "Theta Join" is in a large, light-colored font. Below the title is a code snippet and a bulleted list:

```
join_rdd = sqlContext.sql("select *  
    FROM tableA  
    JOIN tableB  
    ON (keyA < keyB + 10)")
```

- Spark SQL consider each keyA against each keyB in the example above and loop to see if the theta condition is met.
- Better Solution - create buckets for keyA and KeyB can be matched on.

A small red "databrickles" watermark is visible in the bottom left corner of the slide. The overall background of the slide is dark with geometric shapes.

Partitions – Right Sizing – Shuffle – Master Equation

- Largest Shuffle Stage
 - Target Size ≤ 200 MB/partition
- Partition Count = Stage Input Data / Target Size
 - Solve for Partition Count

EXAMPLE

Shuffle Stage Input = 210GB
 $x = 210000MB / 200MB = 1050$
spark.conf.set("spark.sql.shuffle.partitions", 1050)
BUT -> If cluster has 2000 cores
spark.conf.set("spark.sql.shuffle.partitions", 2000)

Join Optimization

- SortMergeJoins (Standard)
- Broadcast Joins (Fastest)
- Skew Joins
- Range Joins
- BroadcastedNestedLoop Joins (BNLJ)

Hive Bucketing in Apache Spark - Tejas Patil

The diagram illustrates three join methods for two tables, A and B, which are partitioned into buckets. Table A has 3 blue buckets, and Table B has 4 green buckets.

- Broadcast hash join:**
 - Ship smaller table to all nodes
 - Stream the other table
- Shuffle hash join:**
 - Shuffle both tables
 - Hash smaller one, stream the bigger one
- Sort merge join:**
 - Shuffle both tables
 - Sort both tables, buffer one, stream the bigger one

SPARK SUMMIT

138 / 2516

Hive Bucketing in Apache Spark - Tejas Patil

The diagram shows the Sort Merge Join process. It starts with two tables, A and B, each partitioned into four buckets. The buckets are sorted, and then the sorted data from both tables is shuffled and merged. The final result is shown in four buckets.

Sort Merge Join

Table A

Table B

Shuffle

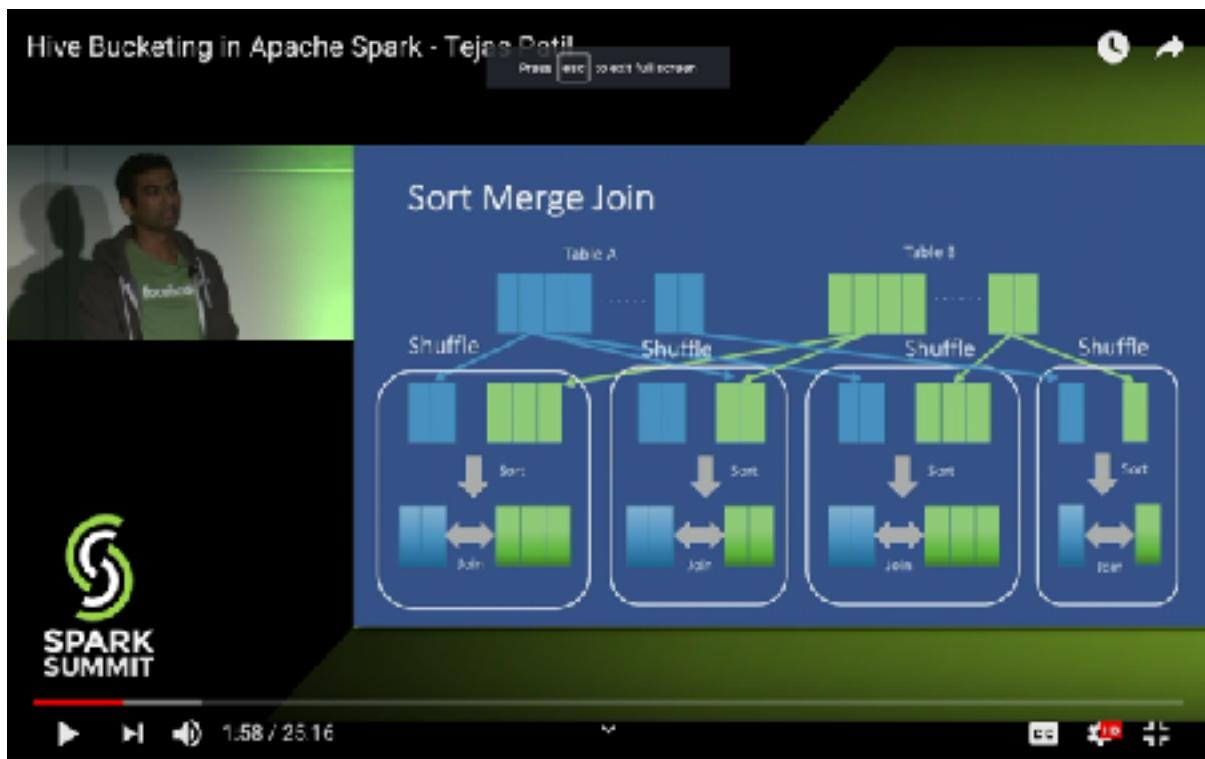
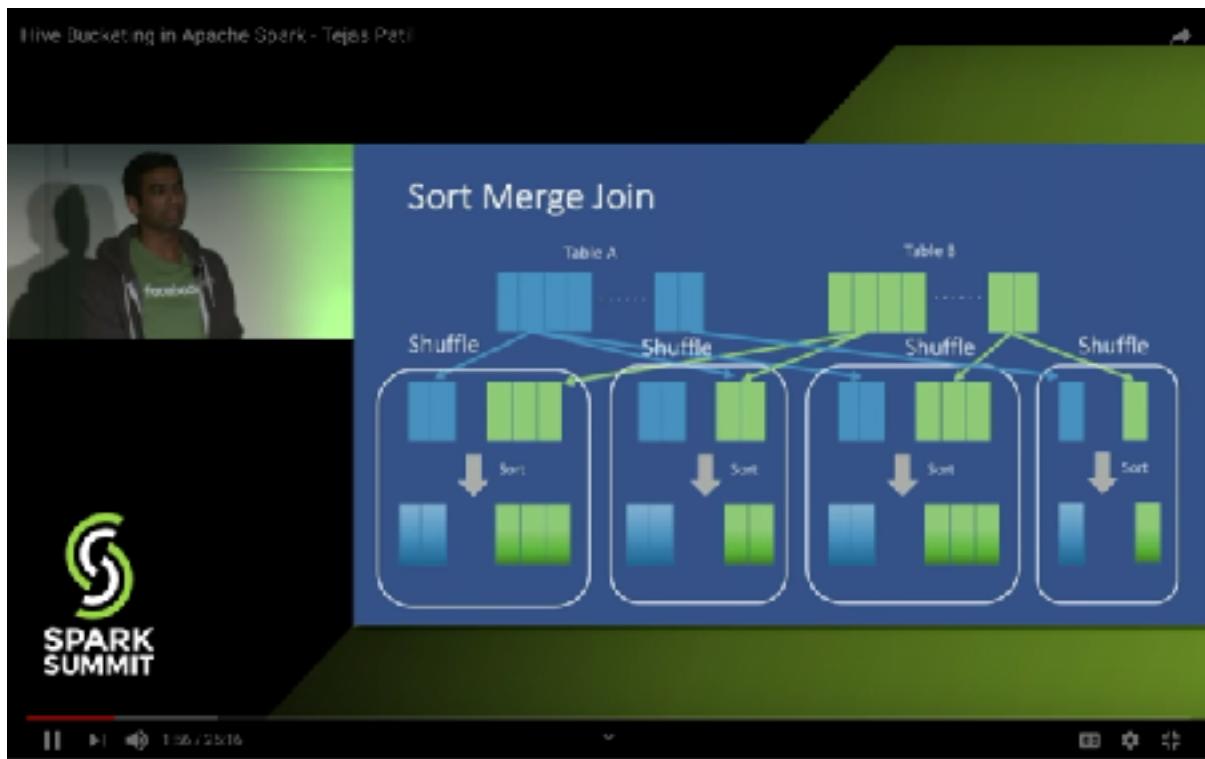
Shuffle

Shuffle

Shuffle

SPARK SUMMIT

138 / 2516



Spark Joins Examples:

```
scala> case class Row(id: Int, value: String)
defined class Row

scala> val r1 = Seq(Row(1, "A1"), Row(2, "A2"), Row(3, "A3"),
Row(4, "A4")).toDS()
r1: org.apache.spark.sql.Dataset[Row] = [id: int, value: string]

scala> val r2 = Seq(Row(3, "A3"), Row(4, "A4"), Row(4, "A4_1"),
Row(5, "A5"), Row(6, "A6")).toDS()
r2: org.apache.spark.sql.Dataset[Row] = [id: int, value: string]
```

Inner join

```
scala> val innerJoin = r1.join(r2, usingColumns = Seq("id"),
joinType="inner").show
+---+---+---+
| id|value|value|
+---+---+---+
|  3|   A3|   A3|
|  4|   A4|   A4|
|  4|   A4| A4_1|
+---+---+---+
innerJoin: Unit = ()
```

Outer join/ Full join / Full outer join: (All are same)

```

scala> val outerJoin = r1.join(r2, usingColumns = Seq("id"),
joinType="outer").show
+---+---+---+
| id|value|value|
+---+---+---+
| 1| A1| null|
| 6| null| A6|
| 3| A3| A3|
| 5| null| A5|
| 4| A4| A4|
| 4| A4| A4_1|
| 2| A2| null|
+---+---+---+
outerJoin: Unit = ()

scala> val fullJoin = r1.join(r2, usingColumns = Seq("id"),
joinType="full").show
+---+---+---+
| id|value|value|
+---+---+---+
| 1| A1| null|
| 6| null| A6|
| 3| A3| A3|
| 5| null| A5|
| 4| A4| A4|
| 4| A4| A4_1|
| 2| A2| null|
+---+---+---+
fullJoin: Unit = ()

scala> val fullOuterJoin = r1.join(r2, usingColumns = Seq("id"),
joinType="full_outer").show
+---+---+---+
| id|value|value|
+---+---+---+
| 1| A1| null|
| 6| null| A6|
| 3| A3| A3|
| 5| null| A5|
| 4| A4| A4|
| 4| A4| A4_1|
| 2| A2| null|
+---+---+---+

```

Left join/ Left Outer join-(both same)

```

scala> val leftJoin = r1.join(r2, usingColumns = Seq("id"),
joinType="left").show
+---+---+---+
| id|value|value|
+---+---+---+
| 1| A1| null|
| 2| A2| null|
| 3| A3| A3|
| 4| A4| A4_1|
| 4| A4| A4|
+---+---+---+
leftJoin: Unit = ()

scala> val leftOuterJoin = r1.join(r2, usingColumns = Seq("id"),
joinType="left_outer").show
+---+---+---+
| id|value|value|
+---+---+---+
| 1| A1| null|
| 2| A2| null|
| 3| A3| A3|
| 4| A4| A4_1|
| 4| A4| A4|
+---+---+---+

```

Left Semi join

```

scala> val left_semiJoin = r1.join(r2, usingColumns = Seq("id"),
joinType="left_semi").show
+---+---+
| id|value|
+---+---+
| 3| A3|
| 4| A4|
+---+---+

```

Left Anti join

```

scala> val left_antiJoin = r1.join(r2, usingColumns = Seq("id"),
joinType="left_anti").show
+---+---+
| id|value|
+---+---+
| 1| A1|
| 2| A2|
+---+---+
left_antiJoin: Unit = ()

```

Right outer join/ right join : both are same

```
scala> val rightJ0in = r1.join(r2, usingColumns = Seq("id"),
joinType="right").show
+---+----+----+
| id|value|value|
+---+----+----+
| 3 | A3 | A3 |
| 4 | A4 | A4 |
| 4 | A4 | A4_1 |
| 5 | null| A5 |
| 6 | null| A6 |
+---+----+----+
rightJ0in: Unit = ()  
  
scala> val right_Outer_J0in = r1.join(r2, usingColumns =
Seq("id"), joinType="right_outer").show
+---+----+----+
| id|value|value|
+---+----+----+
| 3 | A3 | A3 |
| 4 | A4 | A4 |
| 4 | A4 | A4_1 |
| 5 | null| A5 |
| 6 | null| A6 |
+---+----+----+
```

Reference:

<https://stackoverflow.com/questions/45990633/what-are-the-various-join-types-in-spark>

tab1

NUMID
12
14
10
11

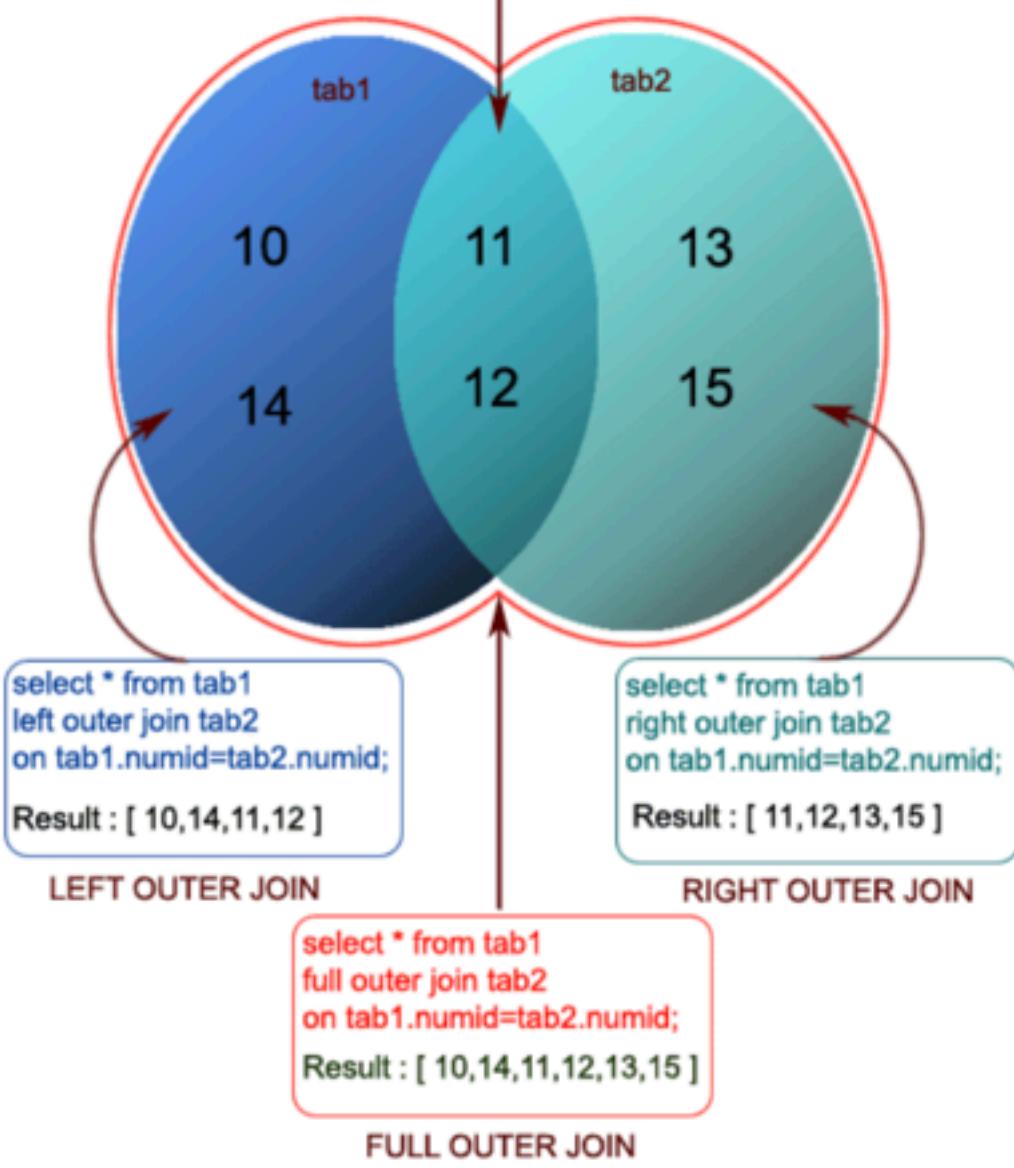
INNER JOIN

```
select * from tab1
inner join tab2
on tab1.numid=tab2.numid;
```

Result : [11,12]

tab2

NUMID
13
15
11
12



List of spark supported join between two dataframes

1. inner : 'inner'
2. cross: 'cross'
3. outer: 'outer'
4. full: 'full'
5. full outer: 'fullouter'
6. left : 'left'
7. left outer : 'leftouter'
8. right : 'right'
9. right outer : 'rightouter'
10. left semi: 'leftsemi'
11. left anti: 'leftanti'

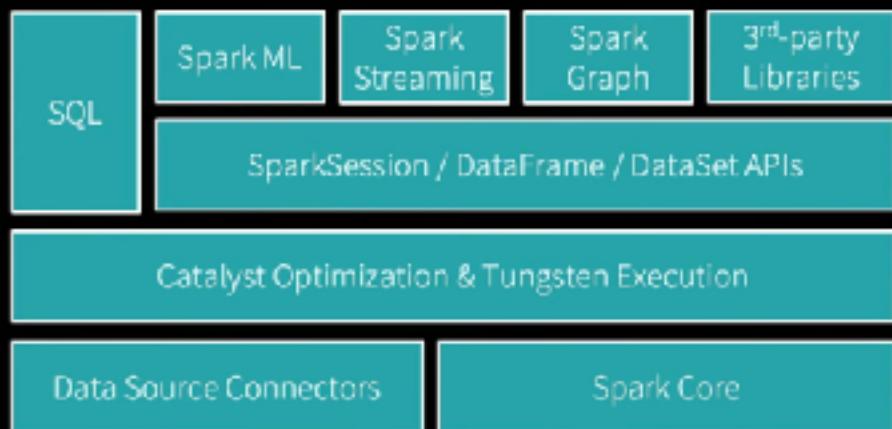
Omit Expensive Ops

- Repartition
 - Use Coalesce or Shuffle Partition Count
- Count – Do you really need it?
- DistinctCount
 - use approxCountDistinct()
- If distincts are required, put them in the right place
 - Use dropDuplicates
 - dropDuplicates BEFORE the join
 - dropDuplicates BEFORE the groupBy

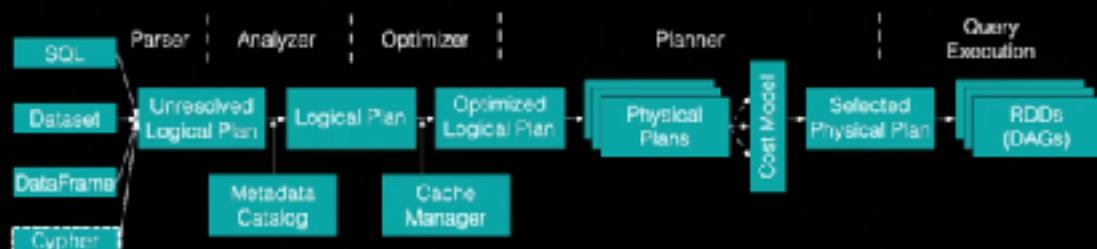
Summary

- Utilize Lazy Loading (Data Skipping)
- Maximize Your Hardware
- Right Size Spark Partitions
- Balance
- Optimized Joins
- Minimize Data Movement
- Minimize Repetition
- Only Use Vectorized UDFs

Apache Spark 3.x



From declarative queries to RDDs



• databricks

7

Join Hints in Spark 3.0

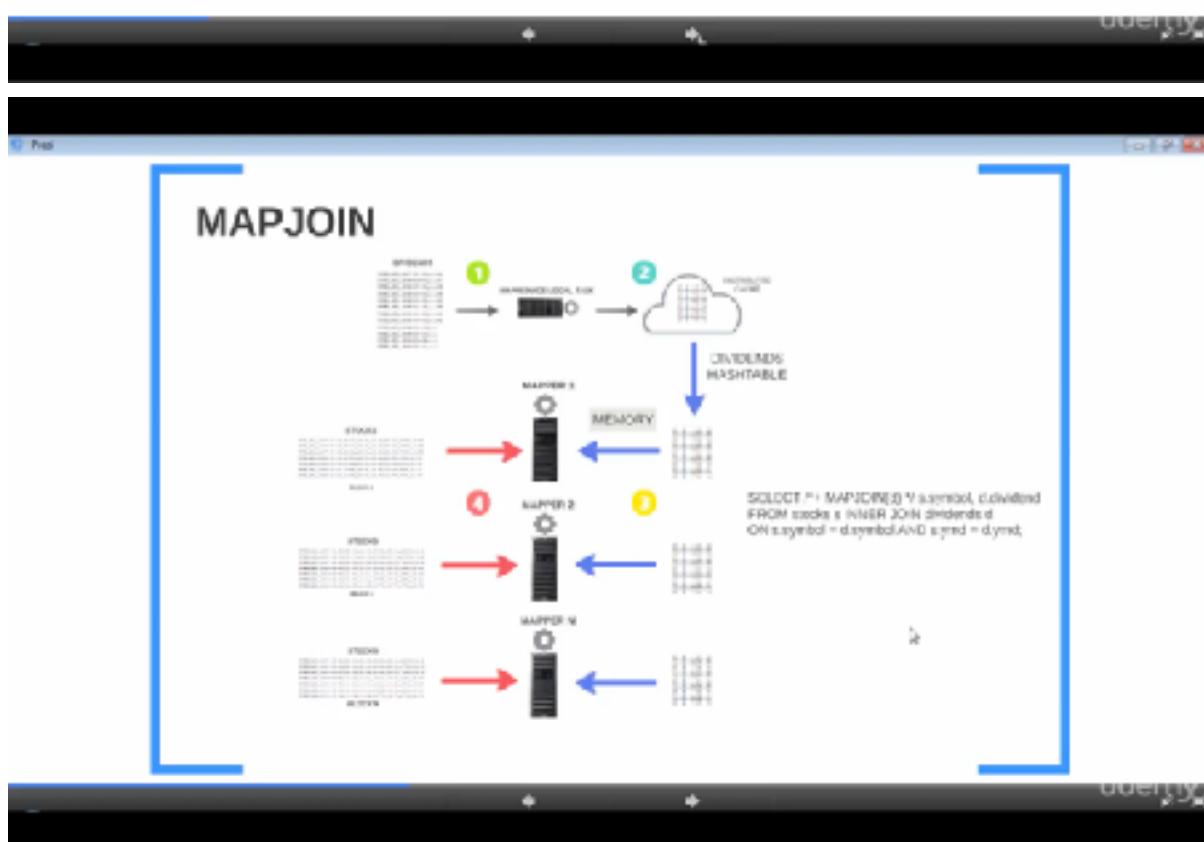
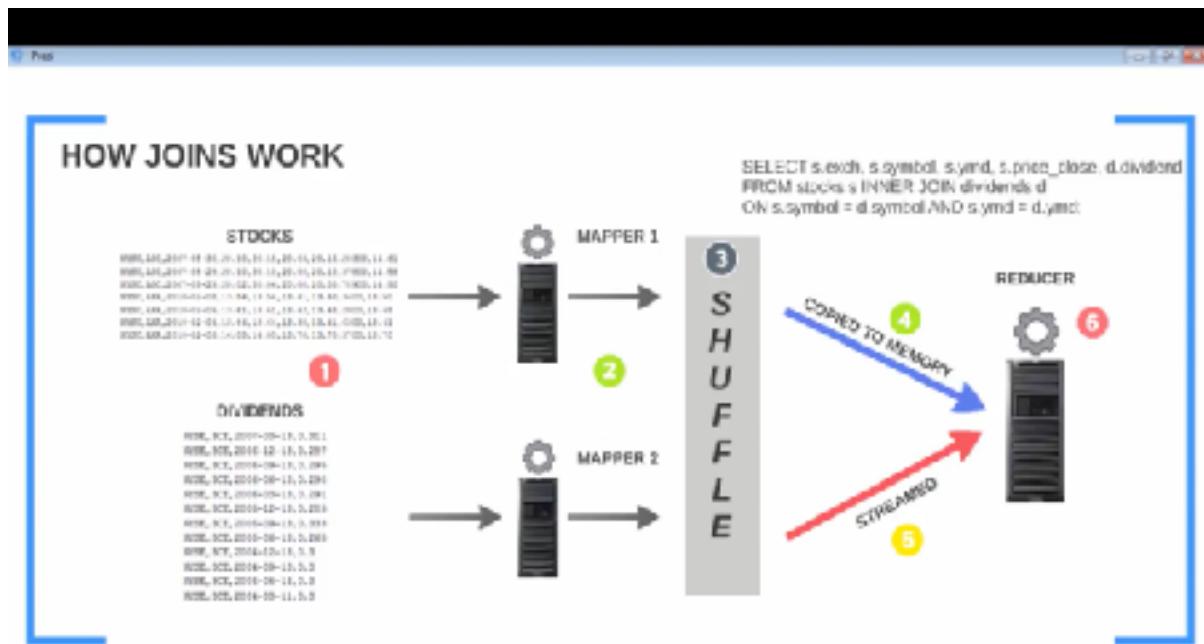
- **BROADCAST**
 - Broadcast Hash/Nested-loop Join
- **MERGE**
 - Shuffle Sort Merge Join
- **SHUFFLE_HASH**
 - Shuffle Hash Join
- **SHUFFLE_REPLICATE_NL**
 - Shuffle-and-Replicate Nested Loop Join

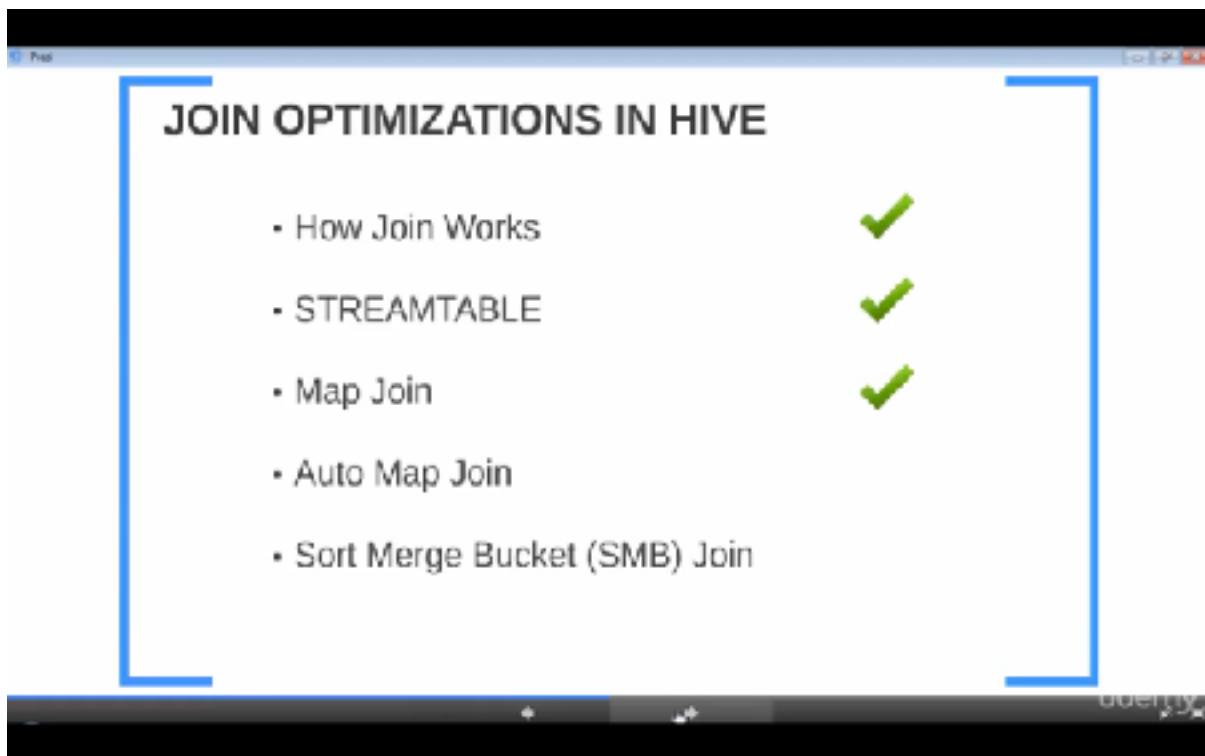
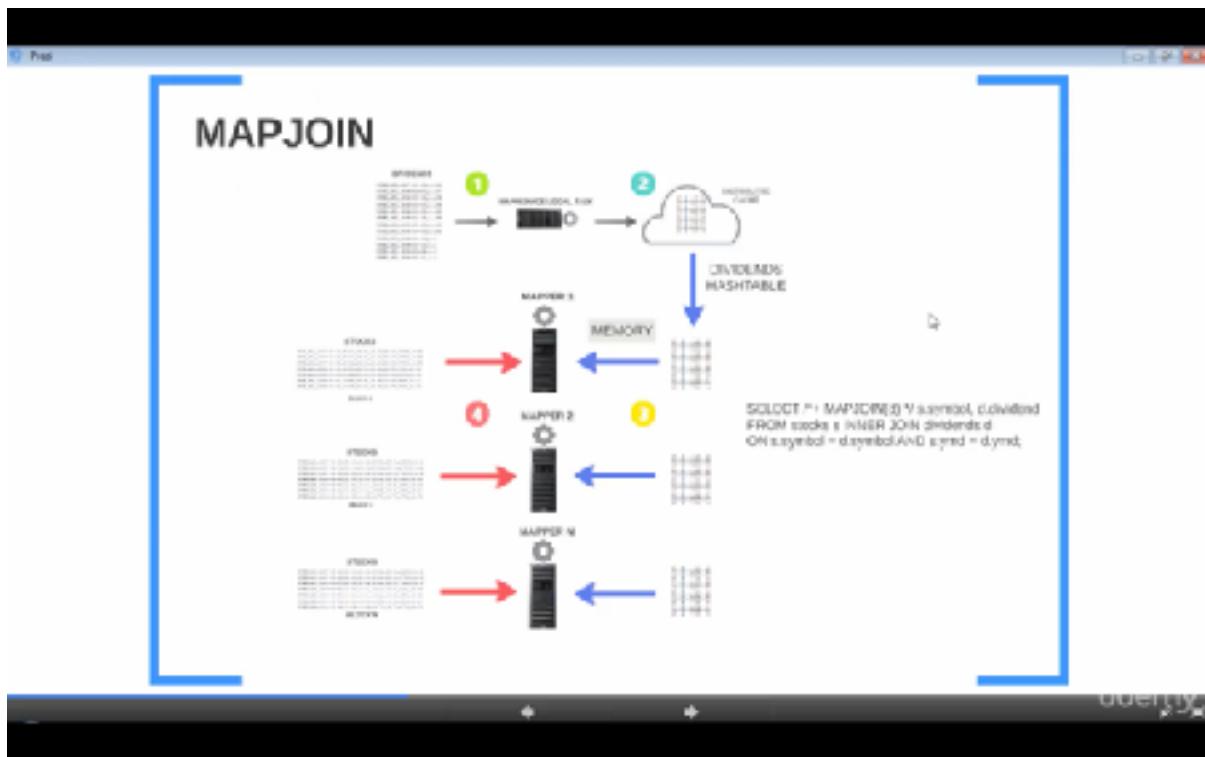
• databricks

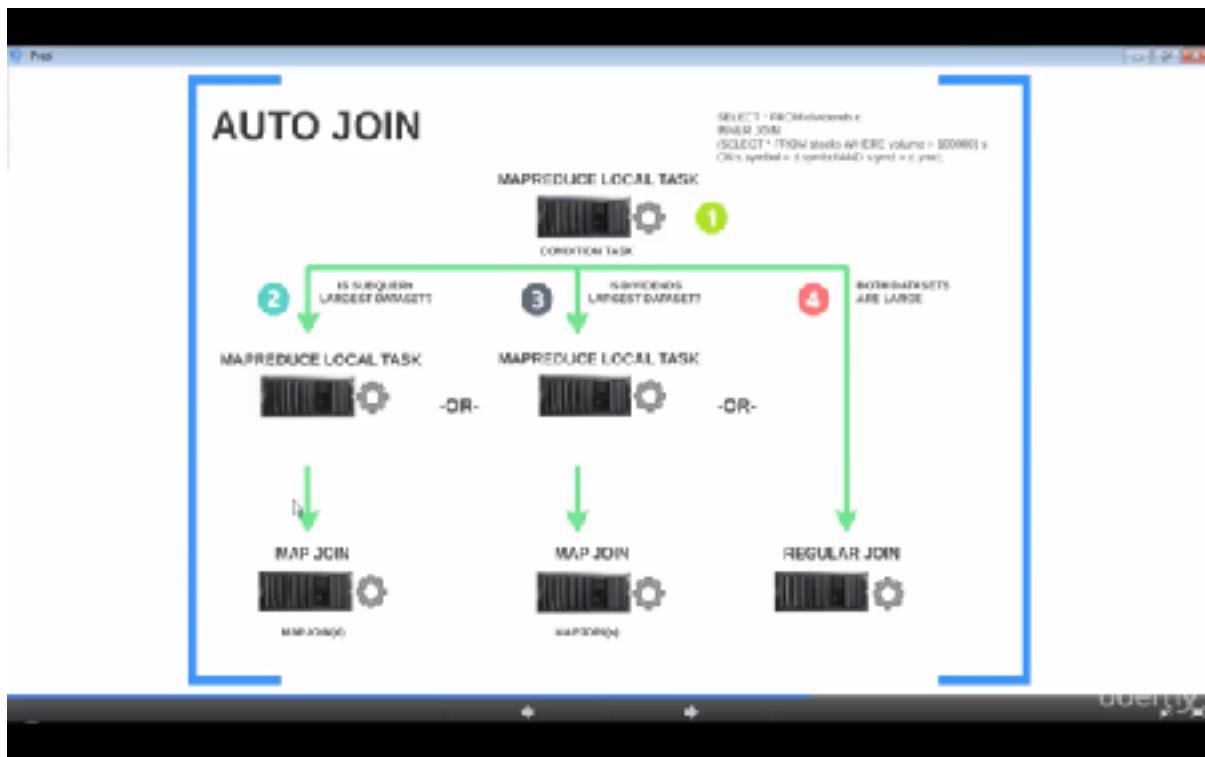
29

<https://www.youtube.com/watch?v=HJvuU0CQS44>

Hive Joins



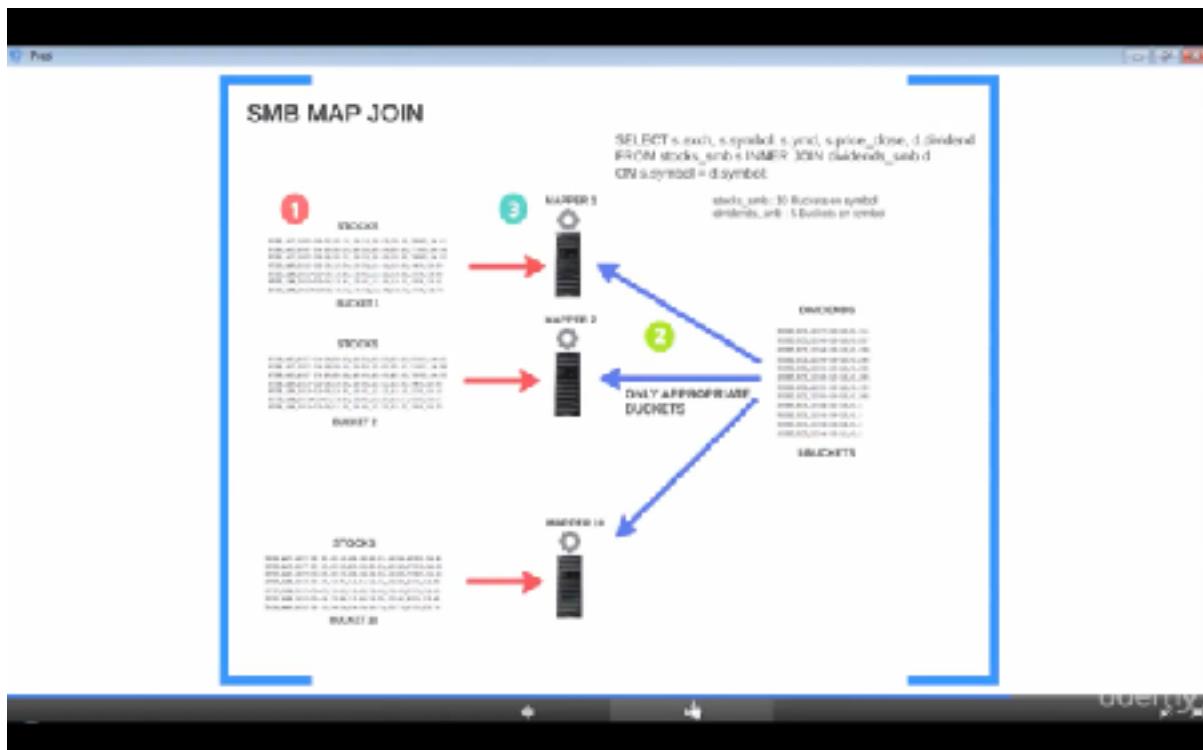




```

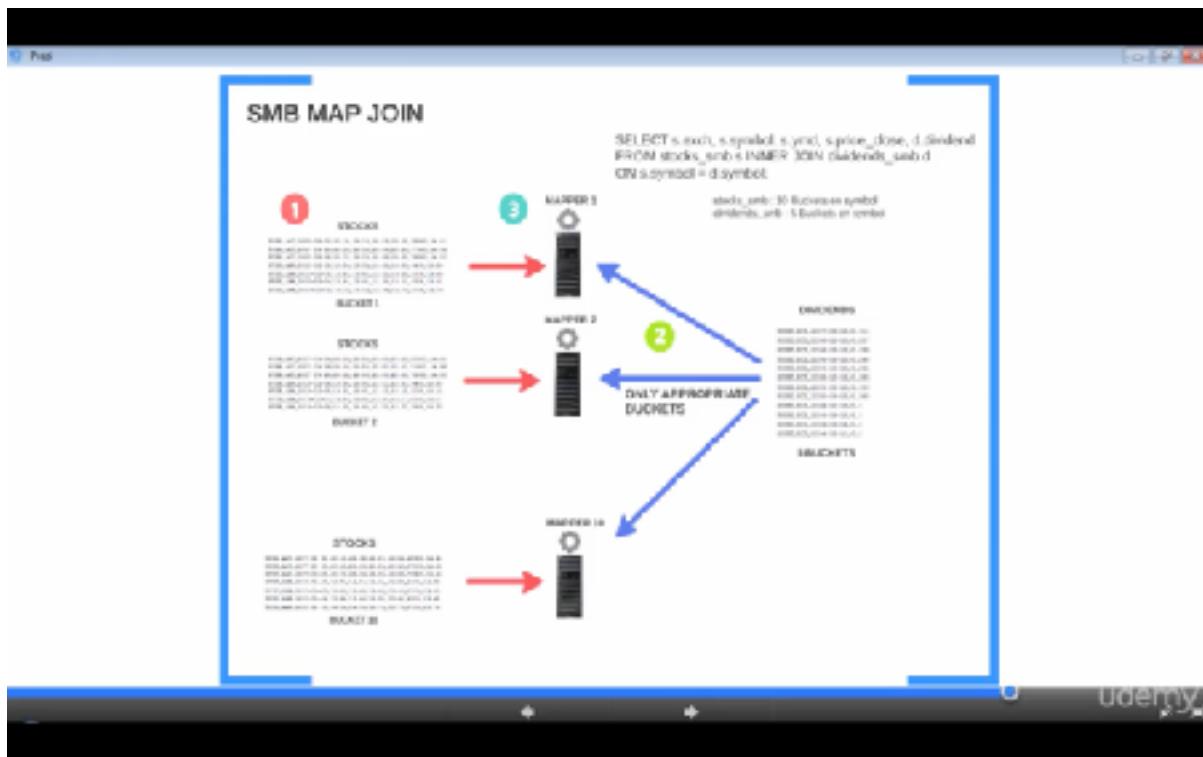
C:\Hadoop\Project\CreateJoin\working\bin\sql\join\optimizer\join.sql -P http://127.0.0.1:8080
File Edit Search View Browser Language Settings Menu Run Plugins Nodes ?
File -> new->sql->join->join
13 ON a.symbol = b.symbol AND a.year = b.year
14
15 *** STREAM A TABLE TO REDUCE 414
16
17 bview SELECT /*+ STREAMTABLE(a) */ a.year, a.symbol, a.vol, a.volume_gb, a.dividend
18 FROM stock a INNER JOIN dividends d
19 ON a.symbol = d.symbol AND a.year = d.year
20
21
22 *** ENABLE AUTO COPILOT JOIN 414
23
24 bview SET bview.auto.convert.join = true
25
26 *** MATCH REE 414
27
28 --Records from Dividends will be loaded in memory and joins will be performed entirely on the side
29 bview SELECT /*+ MATCHREED(414) */ a.year, a.symbol, a.vol, a.volume_gb, a.dividend
30 FROM stock a INNER JOIN dividends d
31 ON a.symbol = d.symbol AND a.year = d.year
32
33 bview SELECT a.year, a.symbol, a.vol, a.volume_gb, b.dividend
34 FROM stock a INNER JOIN dividends b
35 ON a.symbol = b.symbol AND a.year = b.year
36
37
38 *** JOIN WITH DISTINCT DERIVED AT THE TIME 414
39
40 bview SELECT /*+ FROM OLTPBROWSE C */
41 bview JOIN (SELECT * FROM stocks WHERE volume > 1000000
42 ON symbol = symbol AND year = year)
43
44 --Temporarily set max file size for "small" files
45 bview limit=available-1000000000
46
47
48 --ENABLE RECORDING
49 bview SET bview.recordsize = 1000

```

```
infomsg: 10000ms
File Edit Tools Help
Current
HSW Cluster 192.168.1.100 [http://192.168.1.100]
[Start] [Stop] [Reboot] [Power Off] [Save] [Cancel]
Time taken: 23.423 seconds. Processed: 5641 records
BROWSE CREATE TABLE IF NOT EXISTS stocks_mab_4
> stock_symbol,
> year_symbol,
> month_symbol,
> day_symbol,
> close_high_float,
> close_low_float,
> close_close_float,
> volume_int,
> price_avg_close_float
> (BROWSED BY (symbol)) SORTED BY (symbol) INTO 10 RECORDS
> ROW PEGGED SELECTED FIELDS TERMINATED BY ","
on
Time taken: 0.085 seconds
BROWSE CREATE TABLE IF NOT EXISTS clw_bonds_mab_1
> stock_symbol,
> symbol_symbol,
> end_symbol,
> dividend_symbol
>
> (BROWSED BY (symbol)) SORTED BY (symbol) INTO 5 RECORDS
> ROW PEGGED SELECTED FIELDS TERMINATED BY ","
off
Time taken: 0.041 seconds
BROWSE SET max_connections=1024
BROWSE
```

```
infomsg4: infomsg4
File Edit Tools Help
Current
HDFS Cluster HDFS [infomsg4 word]
+ put word1000
> prunes open FLOAT
> prunesign FLOAT
> pruneslow FLOAT
> pruneslowest FLOAT
> reduce INT
> prunesign_CROSS FLOAT
> CLUSTERED BY (word1000) SORTED BY (word1000) INTO 16 BLOCKS
> ROW FORMATS DELIMITED BY FIELD TERMINATED BY ","
or
Time taken: 0.261 seconds
hadoop> CREATE TABLE IF NOT EXISTS wordIndex AS
> SELECT * FROM word1000
> WHERE word1000
> AND word1000
> REVERSE FLOAT
> ;
> CLUSTERED BY (word1000) SORTED BY (word1000) INTO 3 BLOCKS
> ROW FORMATS DELIMITED BY FIELD TERMINATED BY ","
or
Time taken: 0.041 seconds
hadoop> HDFS Admin metastore metadata = true;
hadoop> metastore initdb command failed: metastore_m
> SELECT * FROM metastore;
local metastore_psnow = 1
localhost:21000:21000:1
SUMMARY OF DESIGN: word described as composite type.
In order to change the average load for a column, the system
get time from metastore system per row per column
In order to limit the maximum number of reducers
get time from metastore maxReducers
In order to set a minimum number of reducers
get minReducers
Starting Job = job_1411560200072_0010, Tracking URL = http://node10001:54310/jobs/job_1411560200072_0010
All Cleaned : /user/hadoop/wordIndex/part-r0001 job_1411560200072_0010
Mapreduce framework for map1: number of mappers: 12 number of reducers: 12
2015-09-25 19:58:41,611 Stack-1 map=0%, reduce=0%
[100%]
```



Spark Best Performances:

Spark Best Practices Checklist

	Data Serialization	<ul style="list-style-type: none">✓ Use Kryo Serializer with SparkConf, which is faster and compact.✓ Tune KryoSerializer buffer to hold large objects
	Garbage Collection	<ul style="list-style-type: none">✓ Clean up cached/persisted collections when they are no longer needed.✓ Tuned concurrent abortable preclean time from 10sec to 30sec to push out stop the world GC
	Memory Management	<ul style="list-style-type: none">✓ Avoided using executors with too much memory
	Parallelism	<ul style="list-style-type: none">✓ Optimize number of cores & partitions*
	Action-Transformation	<ul style="list-style-type: none">✓ Minimize shuffles on join(s) by broadcasting the smaller collection✓ Optimize wider transformations as much as possible*
	Caching & Persisting	<ul style="list-style-type: none">✓ Used MEMORY_AND_DISK storage level for caching large✓ Repartition data before persisting to HDFS for better performance in downstream jobs

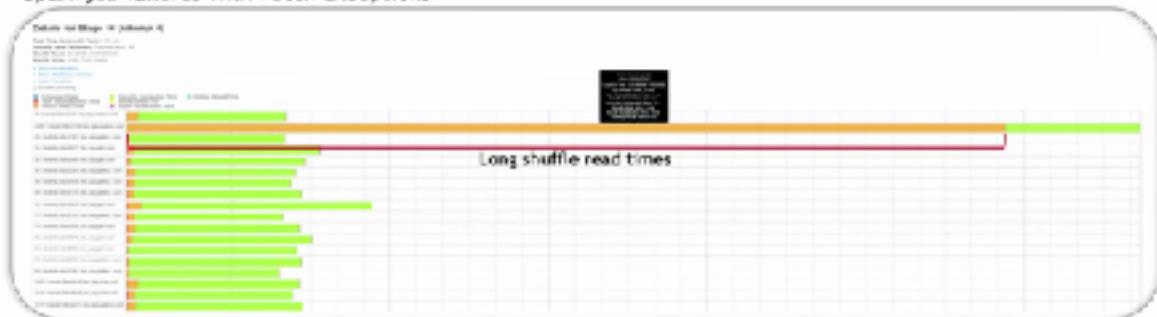
*Specific examples later

© 2014 PayPal Inc. Confidential and proprietary.

19

Learnings

Spark job failures with Fetch Exceptions



Observations

- Executor spends long time on shuffle reads. Then times out, terminates and results in job failure
- Resource constraints on executor nodes causing delay in executor node

Resolution

To address memory constraints, tuned

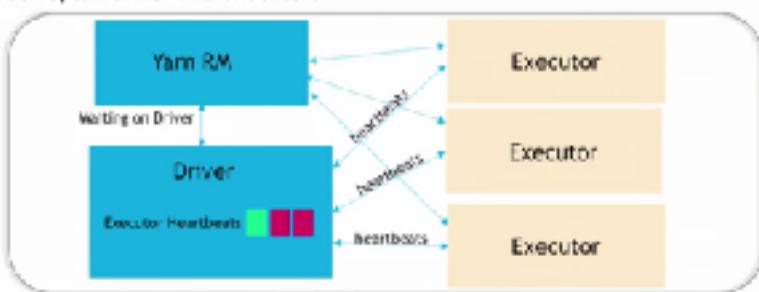
1. config from 200 executor * 4 cores to 400 executor * 2 cores
2. executor memory allocation (reduced)

© 2014 PayPal Inc. Confidential and proprietary.

16

Learnings

Tuning between Spark driver and executors



Observations

- Spark Driver was left with too many heartbeat requests to process even after the job was complete
- Yarn kills the Spark job after waiting on the Driver to complete processing the Heartbeats

Resolution

- The setting "spark.executor.heartbeatInterval" was set too low. Increasing it to 50s fixed the issue
- Allocate more memory to Driver to handle overheads other than typical Driver processes



Learnings

Optimize joins for efficient use of cluster resources (Memory, CPU etc.,)



Learnings

Optimize joins for efficient use of cluster resources (Memory, CPU etc...)



Observation

With the default shuffle partitions of 200, the Join Stage was running with too many tasks causing performance overhead

Resolution

Reduce the `spark.sql.shuffle.partitions` settings to a lower threshold

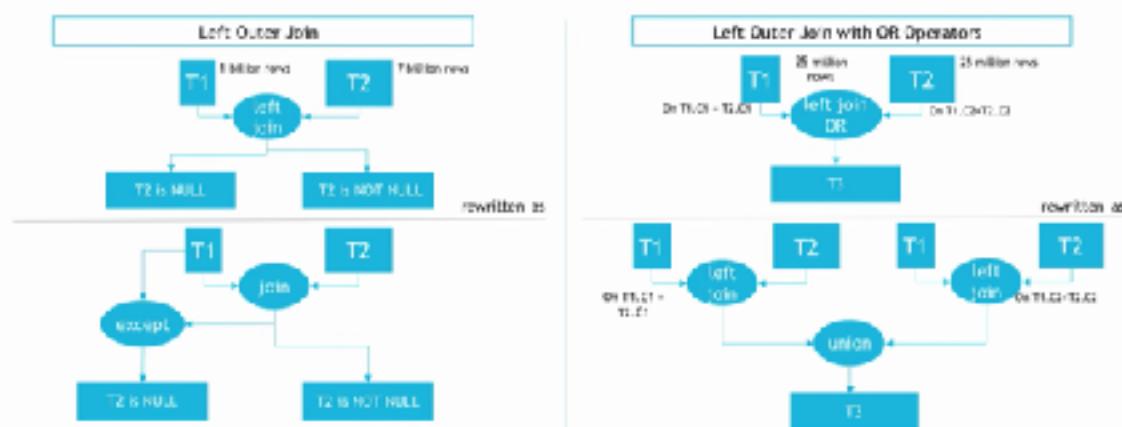


© 2019 PayPal Inc. Confidential and Proprietary

19

Learnings

Optimize wide transformations



© 2019 PayPal Inc. Confidential and Proprietary

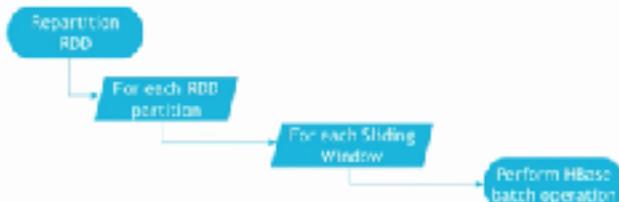
20

Learnings

Optimize throughput for HBase Spark Connection

Observations

- Batch puts and gets slow due to HBase overloaded connections
- Since our HBase row was wide, HBase operations for partitions containing larger groups were slow

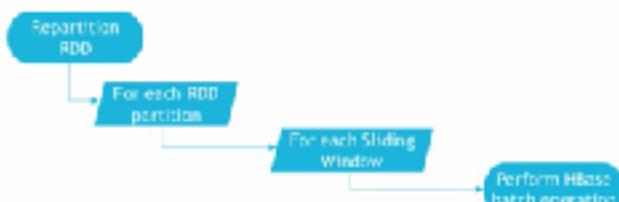


Learnings

Optimize throughput for HBase Spark Connection

Observations

- Batch puts and gets slow due to HBase overloaded connections
- Since our HBase row was wide, HBase operations for partitions containing larger groups were slow



Example

```
val rdd = sc.parallelize(1 to 1000).mapPartitions(p =>
  p.sliding(2000, 2000).foreach{ window =>
    val connection = new HBaseConnection("localhost")
    connection.setOperationTimeout(1000)
    connection.createTable("test", schema)
    connection.close()
  }
)
```



PayPal merchant ecosystem using Apache Spark, Hive, Druid, and HBase

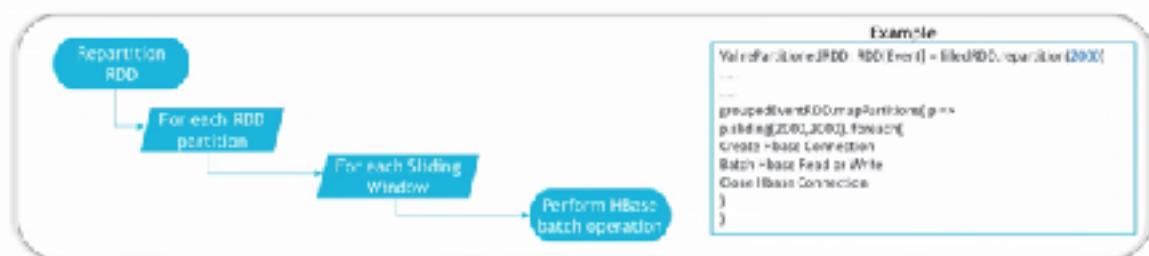


Learnings

Optimize throughput for HBase Spark Connection

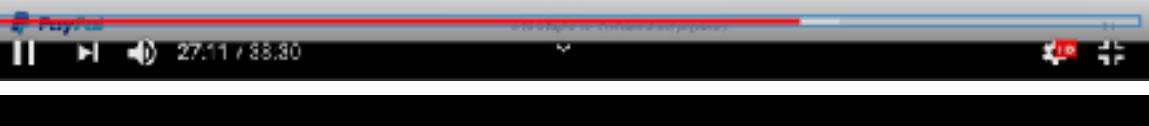
Observations

- Batch puts and gets slow due to HBase overloaded connections
- Since our HBase row was wide, HBase operations for partitions containing larger groups were slow



Resolution

- Implemented sliding window for HBase Operations to reduce HBase connection overload



Learnings

Optimize throughput for HBase Spark Connection

Observations

- Batch puts and gets slow due to HBase overloaded connections
- Since our HBase row was wide, HBase operations for partitions containing larger groups were slow



Resolution

- Implemented sliding window for HBase Operations to reduce HBase connection overload



Behavioral Driven Development

- While Unit tests are more about the implementation, BDD emphasizes more on the behavior of the code.
- Writing "Specifications" in pseudo-English.
- Enables testing at external touch-points of your application.

Feature : Identify the activity related to an event

Scenario: Should perform an Bddicision events and join to activity table and identify the activity name

```
Given I have a set of events
[book_id(String) | page_id(String) | last_activity(String)]
[HTTP://TUTORIALHOME.COM/ | login_provider (review_next_page)
[BIRTHDAY | PROFILE_CREATION | PROFILE_CREATION]

And I have a Activity table
[base_activity_id(String) | article_id(String) | activity_name(String)]
[review_next_page | 19911028080808 | Reviewing Next Page]
[profile_creation | 1234567890123456 | Provide Credentials |]

When I implement the cutt ActivityJoin

Then the final event is
[book_id(String) | article_id(String) | activity_name(String)]
[base_activity_id(String) | activity_id(String) | activity_name(String)]
[HTTP://TUTORIALHOME.COM/ | 19911028080808 | Reviewing Next Page]
[BIRTHDAY | 23232323232323 | Provide Credentials |]
```



© 2013 PayPal Inc. Confidential and proprietary.

13

Catalyst Optimizer

At the core of [Spark SQL](#) is the Catalyst optimizer, which leverages advanced programming language features (e.g. [Scala's pattern matching and quasi quotes](#)) in a novel way to build an extensible query optimizer.

Catalyst is based on functional programming constructs in Scala and designed with these key two purposes:

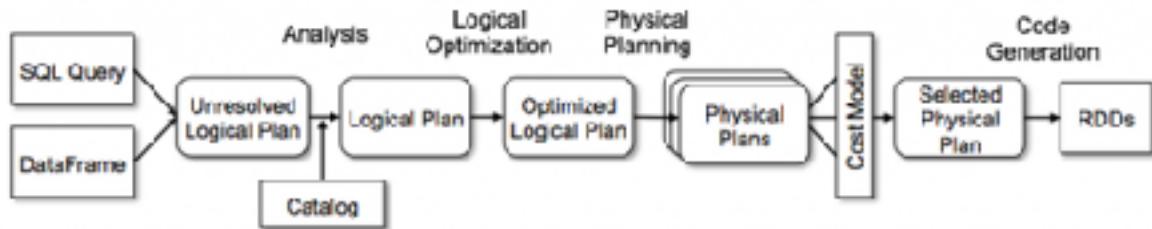
1. Easily add new optimization techniques and features to Spark SQL

- Catalyst contains a general library for representing trees and applying rules to manipulate them.
- On top of this framework, it has libraries specific to relational query processing (e.g., **expressions**, **logical query plans**), and several **sets of rules** that handle different phases of query execution:
 - Analysis,
 - Logical optimization,
 - Physical planning,
 - Code generation to compile parts of queries to Java bytecode

2. Enable external developers to extend the optimizer

(e.g. adding data source specific rules, support for new data types, etc.)

- a. it uses another Scala feature, **quasiquotes**,
- b. that makes it easy to generate code at runtime from composable expressions.
- c. Catalyst also offers several public extension points, including external data sources and user-defined types.
- d. As well, Catalyst supports both rule-based and cost-based optimization



References:

<https://databricks.com/glossary/catalyst-optimizer>

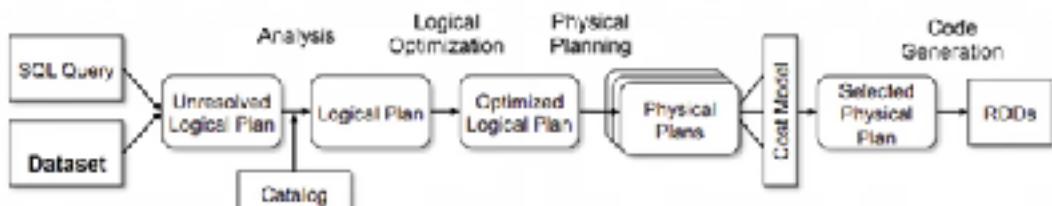
<https://databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html>

Example for Catalyst Optimizer:

Spark SQL Architecture

Terminology:

- > Logical and Physical plans are trees representing query evaluation
- > Logical plan is higher-level and algebraic
- > Physical plan is lower-level and operational



Spark SQL Catalyst Code Optimization using Function Outlining with Madhusudana

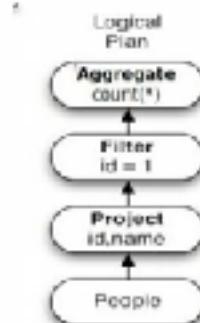
Catalyst Optimizer Phases

- **Analysis:**
 - > resolve unresolved attribute references or relations
- **Logical Optimization:**
 - > standard sql query optimizations like constant folding, predicate pushdown, project pruning are applied.
- **Physical Planning:**
 - > one or more physical plans are formed from the optimized logical plan, using physical operator matching the Spark execution engine
- **Code Generation:**
 - > fuses multiple physical operators together into a single optimized function
 - > to avoid large amount of branches and virtual function calls, generates Java code to run on each machine
 - > generated code is compiled using Janino compiler

Logical Plan Optimization Example

- An example query

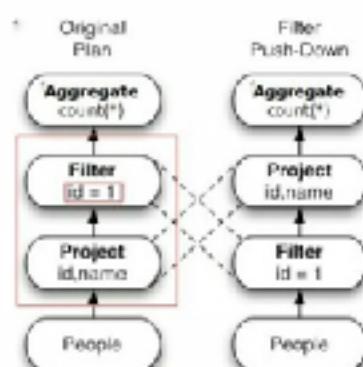
```
SELECT count(*)  
FROM (  
    SELECT id, name  
    FROM People ) p  
WHERE p.id = 1
```



Logical Plan Optimization Example (Contd..)

- Optimization Rules example

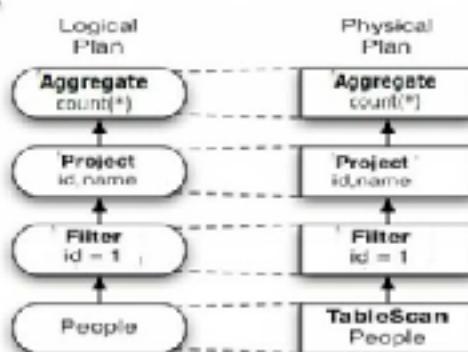
- Find Filters on top of projections.
- Check that the filter can be evaluated without the result of the project.
- If so, switch the operators.



Physical Planning

- Native query planning

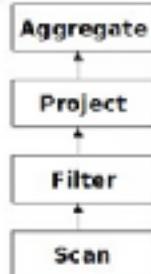
```
SELECT count(*)  
FROM (  
    SELECT id, name  
    FROM People ) p  
WHERE p.id = 1
```



Catalyst Code Generation

- Without Code Generation, each of the operator in the tree has to be interpreted for each row of data by walking down the tree
 - large amounts of branches and virtual function calls
 - unnecessary memory allocations/de-allocations from creating new rows
- With Code Generation, this can be rewritten as one iterator which takes in a row
 - creates one row object per output row, and doesn't use any extraneous method calls
 - allows the compiler/processor optimizations to kick in

```
int count = 0;  
for (Row r : rows) {  
    if (<filter>) {  
        count++;  
    }  
}  
return new Row(count);
```

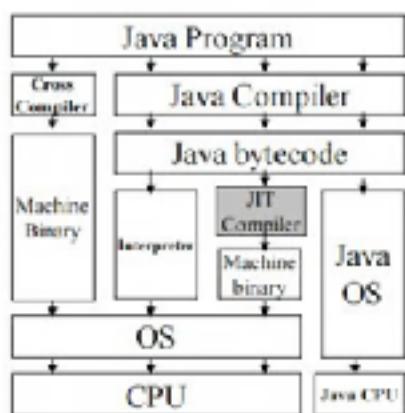


Traditional Java Compilation and Execution

- A Java Compiler compiles high level Java source code to Java bytecode readable by Java Virtual Machine (JVM)
- JVM interprets bytecode to machine instructions at runtime
- Advantages
 - platform independence (JVM present on most machines)
- Drawbacks
 - needs memory
 - not as fast as running pre-compiled machine instructions



Java JIT Compiler Overview:



- A just-in-time (JIT) compiler is a compiler that compiles code into machine instructions during program execution, rather than ahead of time.
- Combines speed of compiled code w/ flexibility of interpretation
- JVM interprets the application code initially
- Collects execution statistics for each method
- Invokes JIT compiler for identified hot methods
- JIT code generally offers far better performance than interpreted code



Another Example for above [Catalyst: The Apache Spark Optimizer](#)



Why structure APIs?

```
Dataframe  
data.groupBy("dept").avg("age")  
  
SQL  
select dept, avg(age) from data group by 1  
  
RDD  
data.map { case (dept, age) => dept -> (age, 1) }  
.reduceByKey { case ((a1, c1), (a2, c2)) => (a1 + a2, c1 + c2)}  
.map { case (dept, (age, c)) => dept -> age / c }
```

● datadriven 5

The slide shows a speaker at a podium on the left and a presentation slide on the right. The slide title is 'Why structure APIs?'. It displays three code snippets: a Dataframe API call, an SQL query, and an RDD-based implementation. The RDD code uses map, reduceByKey, and map operations to achieve the same result as the Dataframe API. The SPARK SUMMIT logo is visible in the bottom left corner of the slide.



How to take advantage of optimization opportunities?

● datadriven 5

The slide shows a speaker at a podium on the left and a presentation slide on the right. The slide title is 'How to take advantage of optimization opportunities?'. The SPARK SUMMIT logo is visible in the bottom left corner of the slide.



Get an optimizer that automatically finds out
the most efficient plan to execute data
operations specified in the user's program



databricks



Catalyst: Apache Spark's Optimizer



databricks

A video frame showing a presentation slide. The slide has a dark background with a green diagonal bar. In the top left corner is a small video window of a speaker. The main title 'How Catalyst Works: An Overview' is centered in white text. Below the title is a diagram showing three stacked components: 'SQL AST', 'DataFrame', and 'Dataset (Java/Scala)'. To the left of the diagram is the Spark Summit logo. At the bottom left is the text '● dambrides' and at the bottom right is a small number '11'.

How Catalyst Works: An Overview

The diagram illustrates the Catalyst architecture. It shows three input boxes on the left: 'SQL AST', 'DataFrame', and 'Dataset (Java/Scala)'. Arrows point from each of these boxes to a central blue box labeled 'Query Plan'. Above the 'Query Plan' box is the word 'Catalyst' in large letters.

A video frame showing a continuation of the presentation slide. The layout is identical to the first slide, featuring a dark background with a green diagonal bar, a small video window of the speaker in the top left, and the main title 'How Catalyst Works: An Overview' centered above the diagram. The diagram itself is also identical, showing the flow from SQL AST, DataFrame, and Dataset to a 'Query Plan' through Catalyst.

A video frame showing a presentation slide. The slide has a dark background with a green diagonal bar on the right. In the top left corner, there is a small video window showing a man speaking. The main title 'How Catalyst Works: An Overview' is at the top. Below it is a diagram illustrating the Catalyst process. The input sources are SQL AST, DataFrame, and Dataset (Java/Scala), which point to a 'Query Plan' box. From the 'Query Plan' box, two arrows emerge: one labeled 'Transformations' pointing to an 'Optimized Query Plan' box, and another labeled 'Code Generation' pointing to an 'RDDs' box.

How Catalyst Works: An Overview

SQL AST
DataFrame
Dataset (Java/Scala)

Catalyst

Transformations

Optimized Query Plan

Code Generation

RDDs

SPARK SUMMIT

A video frame showing the continuation of the presentation slide. The layout is identical to the first slide, featuring the same title, diagram, and footer elements. The small video window in the top left corner shows the speaker continuing his presentation.

How Catalyst Works: An Overview

SQL AST
DataFrame
Dataset (Java/Scala)

Catalyst

Transformations

Optimized Query Plan

Code Generation

RDDs

SPARK SUMMIT

How Catalyst Works: An Overview

The diagram illustrates the Catalyst architecture. It shows various input sources (SQL AST, DataFrame, Dataset (Java/Scala)) on the left, which feed into a central 'Catalyst' component. The 'Catalyst' component is enclosed in a dashed blue box and contains three main stages: 'Query Plan', 'Optimized Query Plan', and 'RDDs'. The flow starts with 'Transformations' leading from the 'Query Plan' to the 'Optimized Query Plan', which then leads to 'RDDs'. Below the 'Catalyst' box, a double-headed arrow labeled 'Abstractions of users' programs (Trees)' connects the 'Query Plan' and 'Optimized Query Plan' stages.

Trees: Abstractions of Users' Programs

```
SELECT sum(v)
FROM (
  SELECT
    t1.id,
    1 + 2 + t1.value AS v
  FROM t1 JOIN t2
  WHERE
    t1.id = t2.id AND
    t2.id > 50000) tmp
```



Trees: Abstractions of Users' Programs

Expression

```
SELECT sum(v)
FROM (
  SELECT
    t1.id,
    1 + 2 + t1.value AS v
  FROM t1 JOIN t2
  WHERE
    t1.id = t2.id AND
    t2.id > 50000) tmp
```

• dambrides 14

 SPARK SUMMIT



Trees: Abstractions of Users' Programs

Expression

```
SELECT sum(v)
FROM (
  SELECT
    t1.id,
    1 + 2 + t1.value AS v
  FROM t1 JOIN t2
  WHERE
    t1.id = t2.id AND
    t2.id > 50000) tmp
```

- An expression represents a new value, computed based on input values
 - e.g. $1 + 2 + t1.value$

• dambrides 14

 SPARK SUMMIT



Trees: Abstractions of Users' Programs

Expression

```
SELECT sum(v)
FROM (
  SELECT
    t1.id,
    1 + 2 + t1.value AS v
  FROM t1 JOIN t2
  WHERE
    t1.id = t2.id AND
    t2.id > 50000) tmp
```

- An expression represents a new value, computed based on input values
 - e.g. $1 + 2 + t1.value$
- Attribute: A column of a dataset (e.g. $t1.id$) or a column generated by a specific data operation (e.g. v)

● dambrides 14

SPARK SUMMIT



Trees: Abstractions of Users' Programs

Query Plan

```
SELECT sum(v)
FROM (
  SELECT
    t1.id,
    1 + 2 + t1.value AS v
  FROM t1 JOIN t2
  WHERE
    t1.id = t2.id AND
    t2.id > 50000) tmp
```

● dambrides 15

SPARK SUMMIT

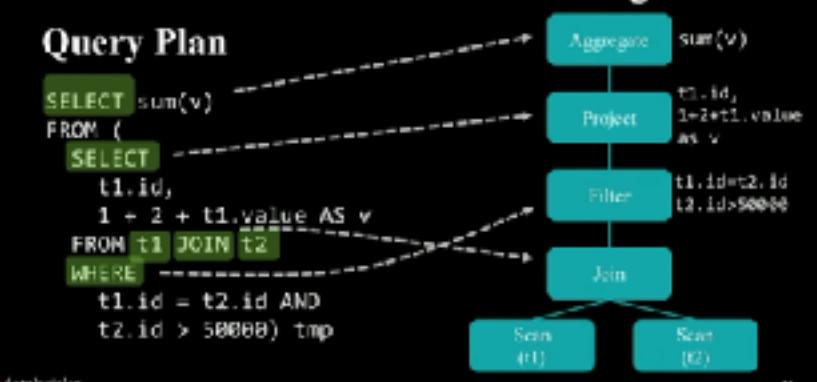


Trees: Abstractions of Users' Programs

Query Plan

```
SELECT sum(v)
FROM (
  SELECT t1.id,
         1 + 2 * t1.value AS v
  FROM t1 JOIN t2
  WHERE t1.id = t2.id AND t2.id > 50000) tmp
```

● dambrides



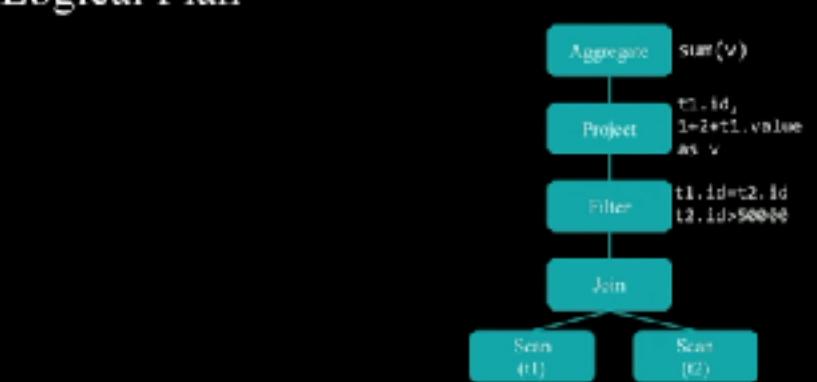
```
graph TD
    QP[Query Plan] --> AP[Aggregate sum(v)]
    AP --> P[Project t1.id, 1+2*t1.value AS v]
    P --> F[Filter t1.id=t2.id, t2.id>50000]
    F --> J[Join]
    J --> S1[Scan(t1)]
    J --> S2[Scan(t2)]
```

16

SPARK SUMMIT



Logical Plan



```
graph TD
    AP[Aggregate sum(v)] --> P[Project t1.id, 1+2*t1.value AS v]
    P --> F[Filter t1.id=t2.id, t2.id>50000]
    F --> J[Join]
    J --> S1[Scan(t1)]
    J --> S2[Scan(t2)]
```

● dambrides

16

SPARK SUMMIT

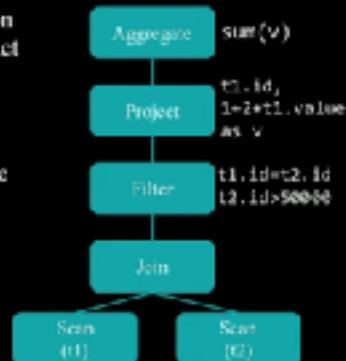


SPARK
SUMMIT

Logical Plan

- A Logical Plan describes computation on datasets without defining how to conduct the computation
- **output:** a list of attributes generated by this Logical Plan, e.g. [$t2.id$, v]
- **constraints:** a set of invariants about the rows generated by this plan, e.g. $t2.id \geq 50000$
- **statistics:** size of the plan in rows/bytes. Per column stats (min/max/ndv/nulls).

• dambrides



16

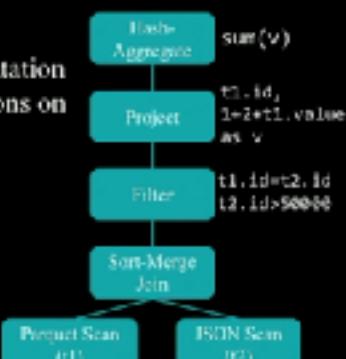


SPARK
SUMMIT

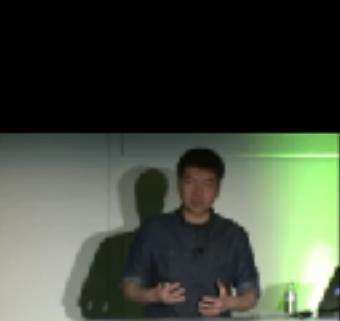
Physical Plan

- A Physical Plan describes computation on datasets with specific definitions on how to conduct the computation
- A Physical Plan is executable

• dambrides



17



How Catalyst Works: An Overview

SQL AST
DataFrame
Dataset (Java/Scala)

Code Generation

Transformations

Query Plan

Optimized Query Plan

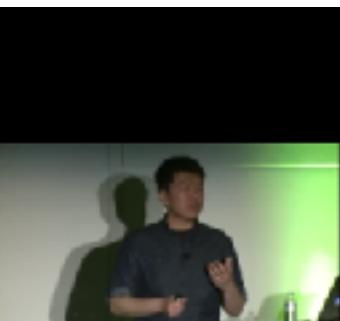
RDDs

Abstractions of users' programs (Trees)

SPARK SUMMIT

● dambrides 18

This slide provides an overview of the Catalyst optimizer. It shows the input sources (SQL AST, DataFrame, Dataset) which feed into the Catalyst system. Catalyst consists of three main components: Transformations, Query Plan, and Optimized Query Plan. The Optimized Query Plan leads to RDDs via Code Generation. The system also maintains Abstractions of users' programs (Trees). The Catalyst logo features a stylized green 'C' with a spark icon.

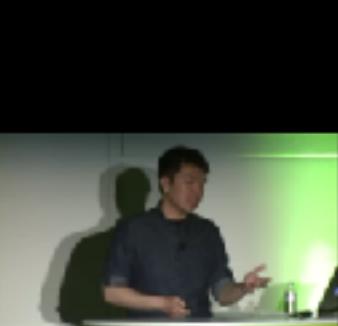


Transformations

SPARK SUMMIT

● dambrides 19

This slide focuses on the 'Transformations' component of the Catalyst optimizer. It features a video thumbnail of a speaker and the Spark Summit logo.



Transformations

- Transformations without changing the tree type (Transform and Rule Executor)
 - Expression \Rightarrow Expression
 - Logical Plan \Rightarrow Logical Plan
 - Physical Plan \Rightarrow Physical Plan
- Transforming a tree to another kind of tree
 - Logical Plan \Rightarrow Physical Plan



SPARK
SUMMIT

● dambrides

29



Transform

- A function associated with every tree used to implement a single rule

$1 + 2 + \text{t1}.value$



● dambrides

29

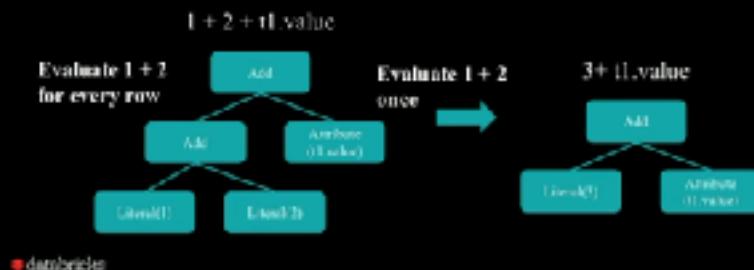


SPARK
SUMMIT



Transform

- A function associated with every tree used to implement a single rule



29



Transform

- A transformation is defined as a Partial Function
- Partial Function: A function that is defined for a subset of its possible arguments

```
val expression: Expression = ...
expression.transform {
    case Add(Literal(x, IntegerType), Literal(y, IntegerType)) =>
        Literal(x + y)
}
```

Case statement determines if the partial function is defined for a given input

● dambrides

30





Combining Multiple Rules

```
graph TD; Project[Project: t1.id, 3*t1.value as v] --> Filter[Filter: t1.id=t2.id, t2.id>50000]; Filter --> Join[Join]; Join --> Scan1[Scan (t1)]; Join --> Scan2[Scan (t2)];
```

● dambrides 22

The slide shows a speaker at a Spark Summit presentation. The title is "Combining Multiple Rules". A query plan diagram is displayed, starting with a "Project" node followed by a "Filter" node, which then leads to a "Join" node. The "Join" node has two children, both labeled "Scan". The "Project" node has the predicate $t1.id, 3*t1.value \text{ as } v$. The "Filter" node has the predicates $t1.id=t2.id$ and $t2.id>50000$. The "Join" node has no explicit predicates listed.



Combining Multiple Rules

Predicate Pushdown

```
graph LR; Project1[Project: t1.id, 3*t1.value as v] --> Filter1[Filter: t1.id=t2.id, t2.id>50000]; Filter1 --> Join1[Join]; Join1 --> Scan11[Scan (t1)]; Join1 --> Scan12[Scan (t2)]; Project2[Project: t1.id, 3*t1.value as v] --> Filter2[Filter: t1.id=t2.id, t2.id>50000]; Filter2 --> Join2[Join]; Join2 --> Scan21[Scan (t1)]; Join2 --> Scan22[Scan (t2)];
```

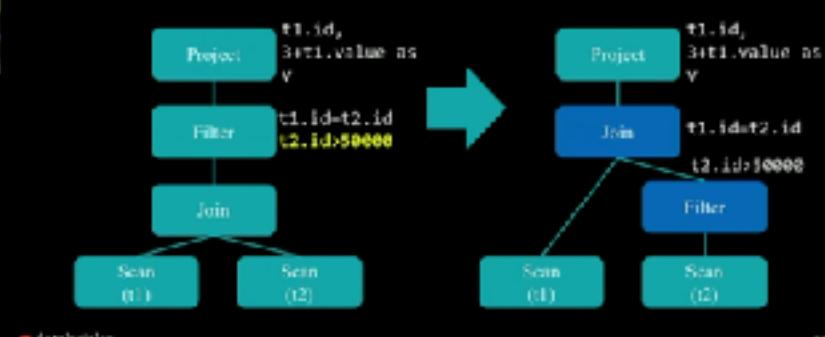
● dambrides 22

This slide continues the "Combining Multiple Rules" topic. It illustrates the process of "Predicate Pushdown". On the left, the original query plan is shown: a "Project" node followed by a "Filter" node, which then leads to a "Join" node. The "Join" node has two children, both labeled "Scan". The "Project" node has the predicate $t1.id, 3*t1.value \text{ as } v$. The "Filter" node has the predicates $t1.id=t2.id$ and $t2.id>50000$. The "Join" node has no explicit predicates listed. An arrow points from this plan to a modified version on the right. In the modified version, the predicates from the "Filter" node have been moved down to the "Join" node, resulting in two separate "Project" nodes. Each "Project" node is followed by its own "Filter" node, which then leads to a "Join" node. The first "Join" node has one child "Scan" (t1) and the second "Join" node has one child "Scan" (t2). Both "Project" nodes have the predicate $t1.id, 3*t1.value \text{ as } v$. Both "Filter" nodes have the predicates $t1.id=t2.id$ and $t2.id>50000$.



Combining Multiple Rules

Predicate Pushdown



Project
Filter
Join
Scan (t1)
Scan (t2)

t1.id = t2.id
t2.id > 50000

Join
Scan (t1)
Scan (t2)

t1.id = t2.id
t2.id > 50000

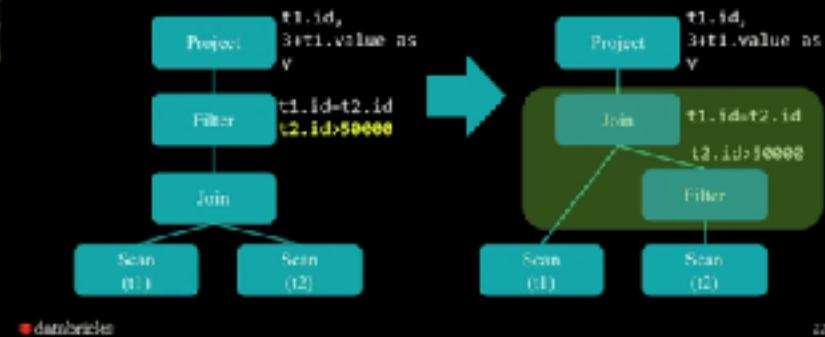
Project
Filter

● dambrides 22



Combining Multiple Rules

Predicate Pushdown



Project
Filter
Join
Scan (t1)
Scan (t2)

t1.id = t2.id
t2.id > 50000

Join
Scan (t1)
Scan (t2)

t1.id = t2.id
t2.id > 50000

Project
Filter

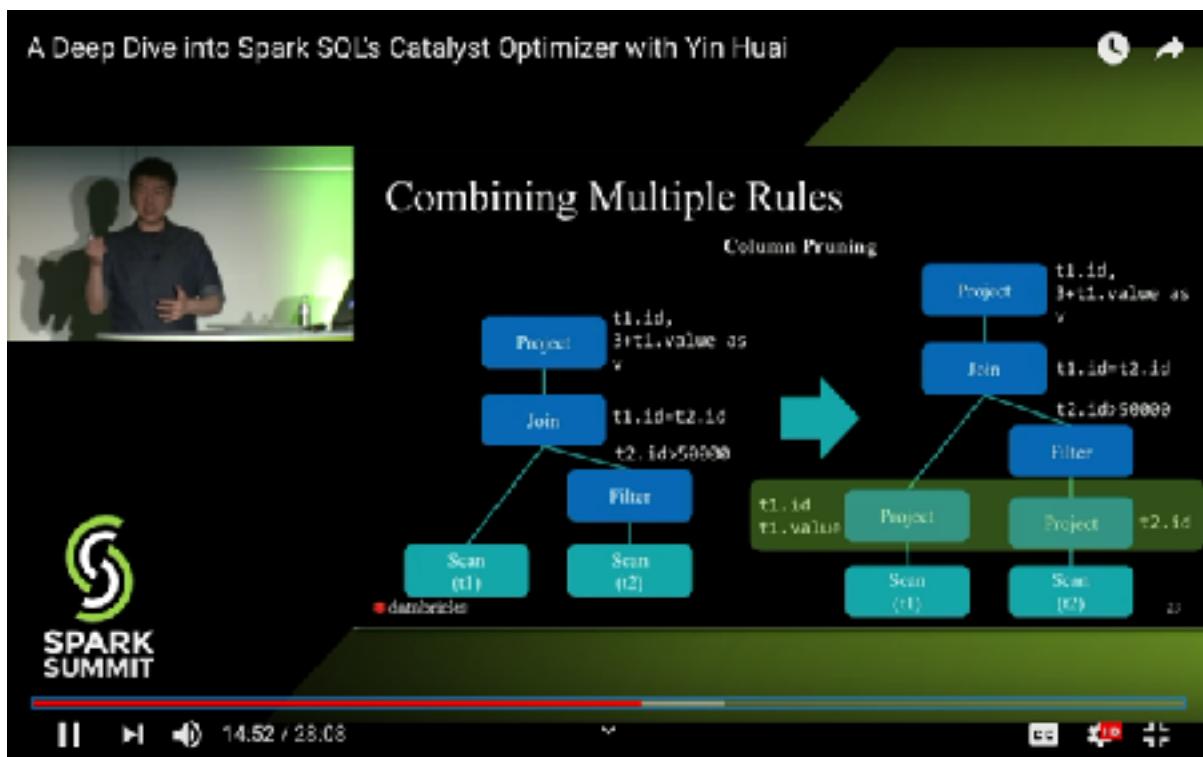
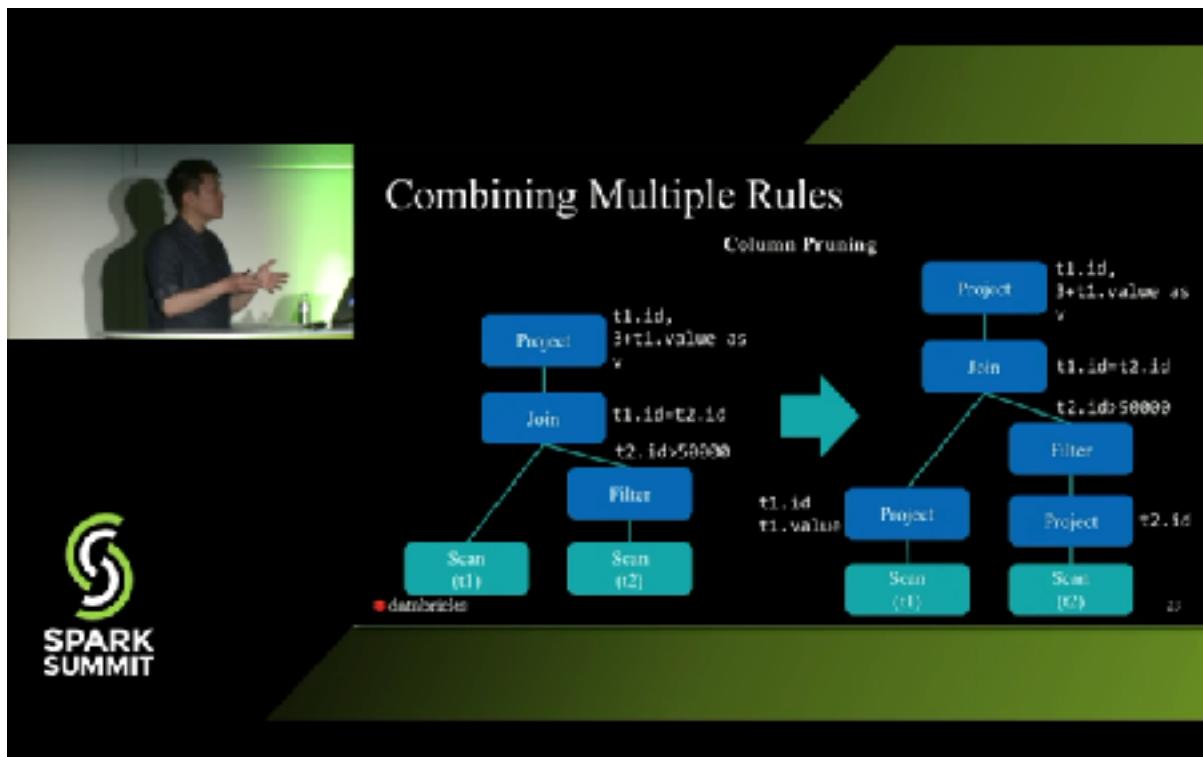
Join

Scan (t1)
Scan (t2)

t1.id = t2.id
t2.id > 50000

Project
Filter

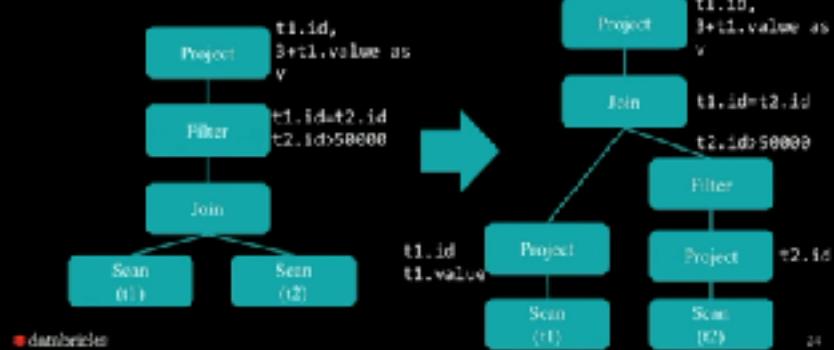
● dambrides 22





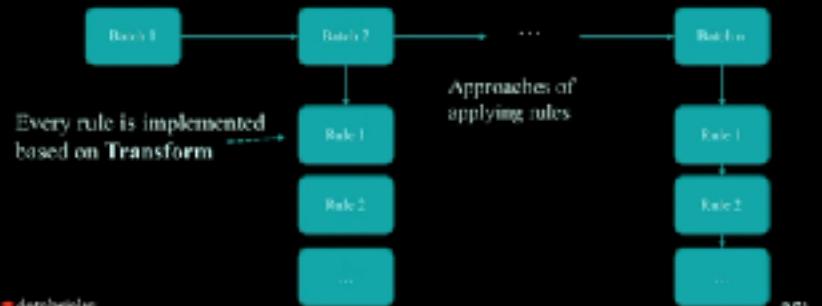
Combining Multiple Rules

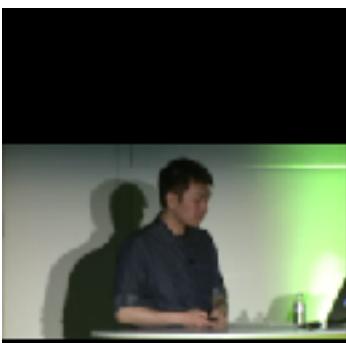
Before transformations



Combining Multiple Rules: Rule Executor

A Rule Executor transforms a Tree to another same type Tree by applying many rules defined in batches





Combining Multiple Rules: Rule Executor

A Rule Executor transforms a Tree to another same type Tree by applying many rules defined in batches.

Every rule is implemented based on Transform

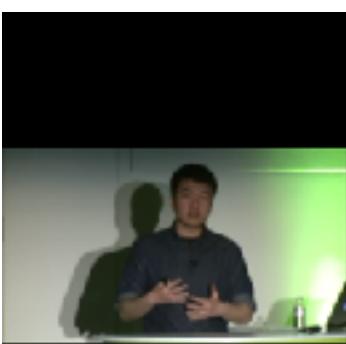
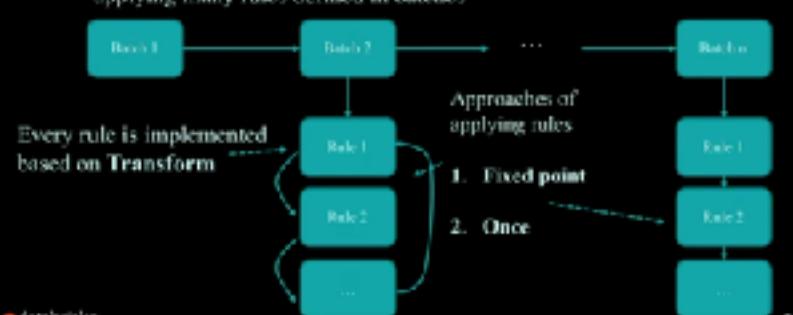
Approaches of applying rules

1. Fixed point
2. Once

Rule 1 → Rule 2 → ... → Rule n

Rule 1
Rule 2
...
Rule n

● dambrides 25



Transformations

- Transformations without changing the tree type (Transform and Rule Executor)
 - Expression \Rightarrow Expression
 - Logical Plan \Rightarrow Logical Plan
 - Physical Plan \Rightarrow Physical Plan
- Transforming a tree to another kind of tree
 - Logical Plan \Rightarrow Physical Plan

● dambrides 26





From Logical Plan to Physical Plan

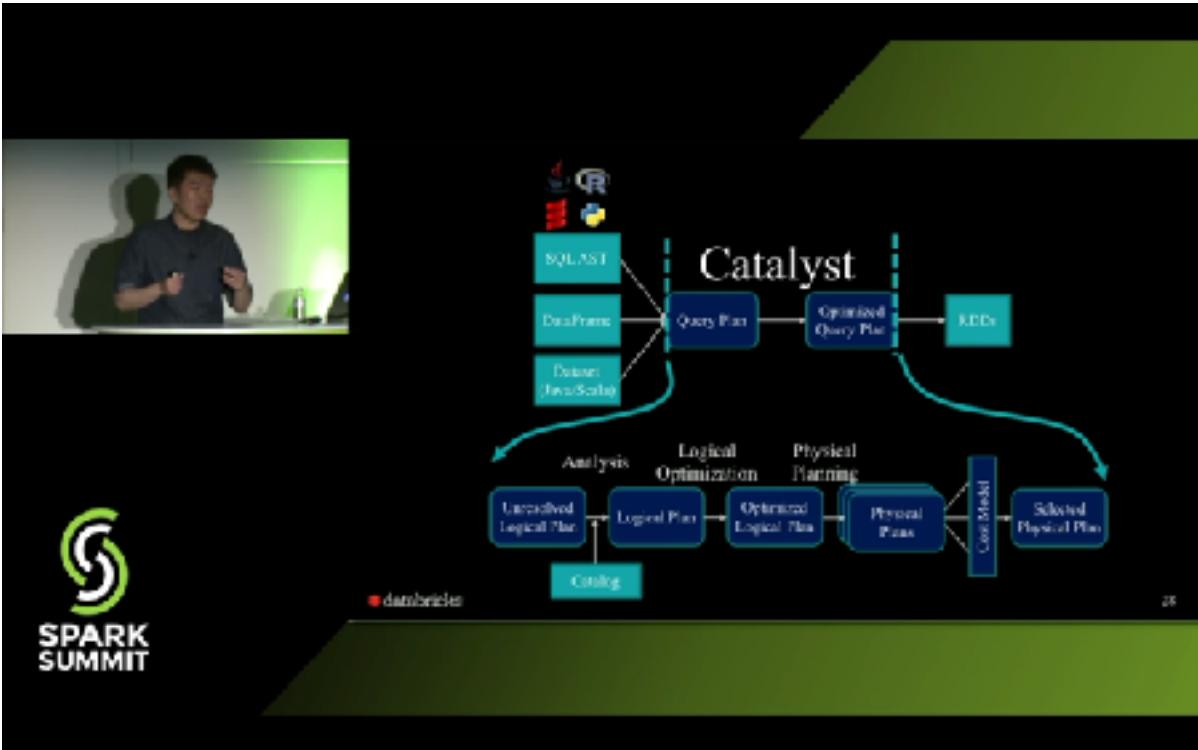
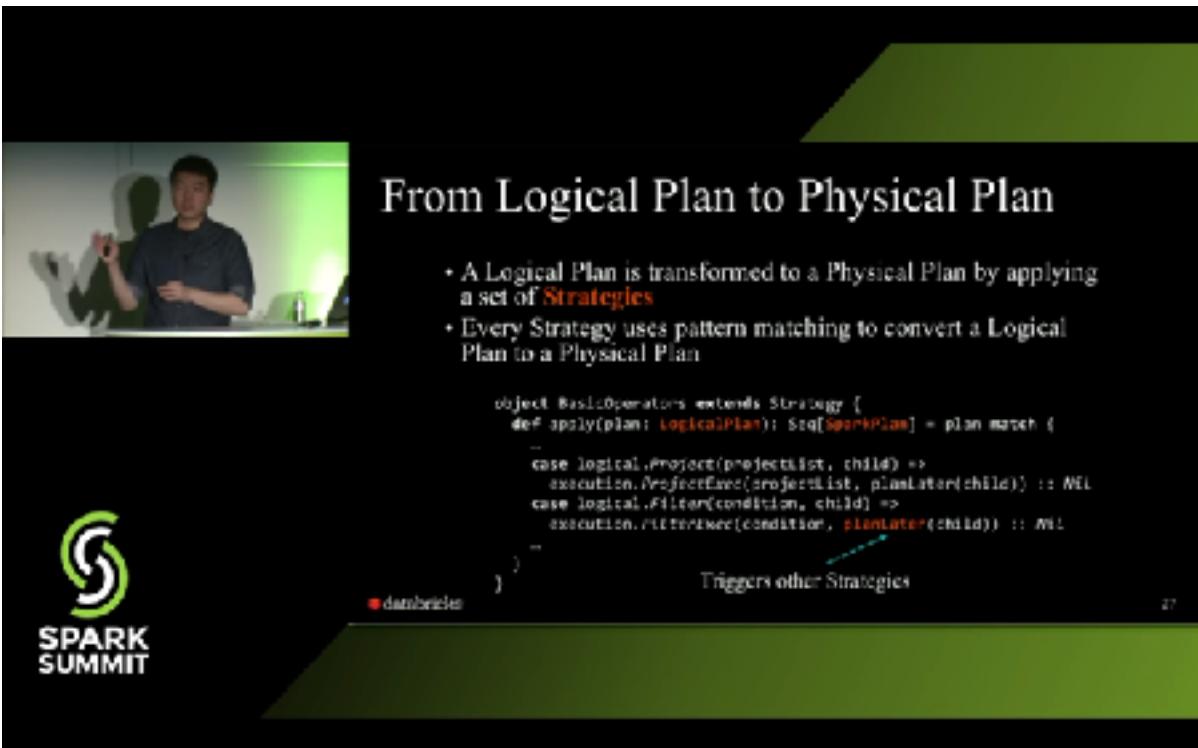


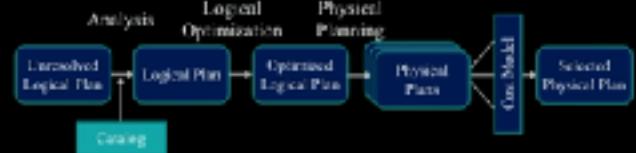
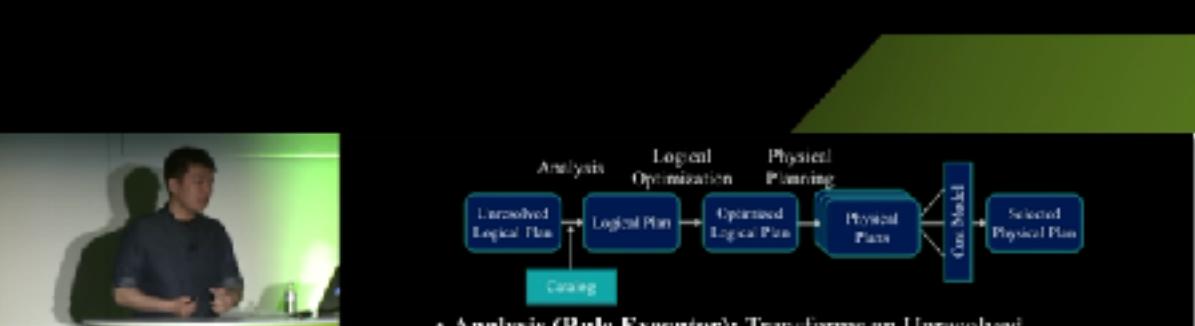
From Logical Plan to Physical Plan

- A Logical Plan is transformed to a Physical Plan by applying a set of **Strategies**
 - Every Strategy uses pattern matching to convert a Logical Plan to a Physical Plan

```
object BasicOperators extends Strategy {
  def apply(plan: LogicalPlan): Seq[SparkPlan] = plan match {
    case logical.Project(projectList, child) =>
      execution.ProjectExec(projectList, planLister(child)) :: NIL
    case logical.Filter(condition, child) =>
      execution.FilterExec(condition, planLister(child)) :: NIL
```





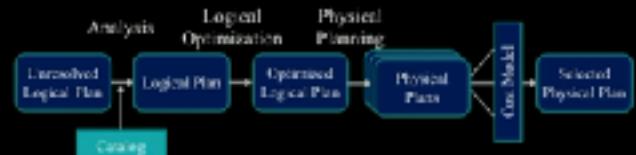


- **Analysis (Rule Executor)**: Transforms an Unresolved Logical Plan to a Resolved Logical Plan



● dambrides

29



- **Analysis (Rule Executor)**: Transforms an Unresolved Logical Plan to a Resolved Logical Plan
 - Unresolved => Resolved: Use Catalog to find where datasets and columns are coming from and types of columns
- **Logical Optimization (Rule Executor)**: Transforms a Resolved Logical Plan to an Optimized Logical Plan



● dambrides

29

The slide illustrates the Spark query optimizer's workflow:

```
graph LR; A[Unresolved Logical Plan] --> B[Logical Plan]; B --> C[Optimized Logical Plan]; C --> D[Physical Plan]; D --> E[Selected Physical Plan]; C <--> F[Catalog];
```

Key components and their descriptions:

- Analysis (Rule Executor):** Transforms an Unresolved Logical Plan to a Resolved Logical Plan
 - Unresolved => Resolved: Use Catalog to find where datasets and columns are coming from and types of columns
- Logical Optimization (Rule Executor):** Transforms a Resolved Logical Plan to an Optimized Logical Plan
- Physical Planning (Strategies + Rule Executor):**
 - Phase 1: Transforms an Optimized Logical Plan to a Physical Plan
 - Phase 2: Rule executor is used to adjust the physical plan to make it ready for execution
- Catalog:** Used during Analysis and Physical Planning.

SPARK SUMMIT

dambricelis 29

Put what we have learned in action

SPARK SUMMIT

dambricelis 29



Roll your own Planner Rule

```
import org.apache.spark.sql.functions._  
  
// tableA is a dataset of integers in the range of [0, 19999999]  
val tableA = spark.range(20000000).as('a)  
// tableB is a dataset of integers in the range of [0, 999999]  
val tableB = spark.range(10000000).as('b)  
// result shows the number of records after joining tableA and tableB  
val result = tableA  
    .join(tableB, $"a.id" === $"b.id")  
    .groupBy()  
    .count()  
result.show()
```

● databricks 12

SPARK SUMMIT



Roll your own Planner Rule

```
import org.apache.spark.sql.functions._  
  
// tableA is a dataset of integers in the range of [0, 19999999]  
val tableA = spark.range(20000000).as('a)  
// tableB is a dataset of integers in the range of [0, 999999]  
val tableB = spark.range(10000000).as('b)  
// result shows the number of records after joining tableA and tableB  
val result = tableA  
    .join(tableB, $"a.id" === $"b.id")  
    .groupBy()  
    .count()  
result.show()
```

This takes 4-8s on Databricks Community edition

● databricks 12

SPARK SUMMIT



Roll your own Planner Rule

```
result.explain()  
= Physical Plan =  
+HashAggregate(keys=(), functions=[count(1)])  
+- Exchange SinglePartition  
  +- +HashAggregate(keys=(), functions=[partial_count(1)])  
    +- +Project  
      +- +SortMergeJoin [id#642L], [id#646L], Inner  
        :- +Sort [id#642L ASC NULLS FIRST], false, 0  
        :  +- Exchange hashpartitioning(id#642L, 200)  
        :  +- +Range (0, 20000000, step=1, splits=8)  
        +- +Sort [id#646L ASC NULLS FIRST], false, 0  
          +- Exchange hashpartitioning(id#646L, 200)  
            +- +Range (0, 10000000, step=1, splits=8)
```

● dambrides 33



Roll your own Planner Rule

```
result.explain()  
= Physical Plan =  
+HashAggregate(keys=(), functions=[count(1)])  
+- Exchange SinglePartition  
  +- +HashAggregate(keys=(), functions=[partial_count(1)])  
    +- +Project  
      +- +SortMergeJoin [id#642L], [id#646L], Inner  
        :- +Sort [id#642L ASC NULLS FIRST], false, 0  
        :  +- Exchange hashpartitioning(id#642L, 200)  
        :  +- +Range (0, 20000000, step=1, splits=8)  
        +- +Sort [id#646L ASC NULLS FIRST], false, 0  
          +- Exchange hashpartitioning(id#646L, 200)  
            +- +Range (0, 10000000, step=1, splits=8)
```

● dambrides 33



Roll your own Planner Rule

Exploit the structure of the problem

We are joining two intervals; the result will be the intersection of these intervals



A
B
A ∩ B

● dambrides

24

SPARK SUMMIT



Roll your own Planner Rule

```
// Import internal APIs of Catalyst
import org.apache.spark.sql.Strategy
import org.apache.spark.sql.catalyst.expressions.{Alias, Equalto}
import org.apache.spark.sql.catalyst.plans.logical.{LogicalPlan, Join, Range}
import org.apache.spark.sql.catalyst.plans.Inner
import org.apache.spark.sql.execution.ProjectExec, RangeExec, SparkPlan

case object IntervalJoin extends Strategy with Serializable {
  def apply(plan: LogicalPlan): Seq[SparkPlan] = plan match {
    case Join(
      Range(start1, end1, 1, part1, Seq(e1)), // matches tableA
      Range(start2, end2, 1, part2, Seq(e2)), // matches tableB
      Inner, Some(Equalto(e1, e2))) // matches the Join
      if ((e1.semanticEquals(e2)) || (e1.semanticEquals(e2))) && (e2.semanticEquals(e1))) =>
      // see next page for rule body
    case _ => Nil
  }
}
```

● dambrides

25

SPARK SUMMIT



Roll your own Planner Rule

```
// matches cases like:  
// tableA: start1-----end1  
// tableB: ...-----end2  
if ((end1 >= start2) && (end2 <= end1)) {  
    // start of the intersection  
    val start = math.max(start1, start2)  
    // end of the intersection  
    val end = math.min(end1, end2)  
    val part = math.max(part1.getUrbits(200), part2.getUrbits(200))  
    // Create a new Range to represent the intersection  
    val result = RangeExec[Range](start, end, 1, Some(part), ci :: Nil)  
    val twoColumns = ProjectExec(  
        Alias(e1, e1.name)(exprId = e1.exprId :: Nil,  
        result)  
        twoColumns :: Nil  
    ) else {  
        Nil  
    }  
}
```

● dambrides 26



Roll your own Planner Rule

Hook it up with Spark

```
spark.experimental.extraStrategies = IntervalJoin :: Nil
```

Use it

```
result.count()
```

This now takes ~0.5s to complete

● dambrides 27



Roll your own Planner Rule

```
result.explain()  
  
-- Physical Plan --  
*HashAggregate(keys=[], functions=[count(1)])  
-- Exchange SinglePartition  
+- *HashAggregate(keys=[], functions=[partial_count(1)])  
  +- *Project  
    +- *Project [id#642L AS id#642L]  
      +- *Range (0, 10000000, step=1, splits=8)
```

SPARK SUMMIT

● dambrides 38



Roll your own Planner Rule

```
result.explain()  
  
-- Physical Plan --  
*HashAggregate(keys=[], functions=[count(1)])  
-- Exchange SinglePartition  
+- *HashAggregate(keys=[], functions=[partial_count(1)])  
  +- *Project  
    +- *Project [id#642L AS id#642L]  
      +- *Range (0, 10000000, step=1, splits=8)
```

SPARK SUMMIT

● dambrides 38

Tungsten :

improving the efficiency of *memory and CPU* for [Spark applications](#),
[TMC- (Tungsten – Memory -CPU)

TMC- M2-BP-CA-CG

M2 = Memory Management

BP- Binary Processing

CA- Cache Aware

CG- Code Generation

]

Project Tungsten: Bringing Apache Spark efficiency Closer to Bare Metal

Project Tungsten will be the largest change to Spark's execution engine since the project's inception. It focuses on substantially improving the efficiency of *memory and CPU* for [Spark applications](#), to push performance closer to the limits of modern hardware.

This effort includes three initiatives:

1. Memory Management and Binary Processing:

Leveraging application semantics to manage memory explicitly and eliminate the overhead of JVM object model and garbage collection

2. Cache-aware computation:

Algorithms and data structures to exploit/utilize memory hierarchy

3. Code generation:

Using code generation to exploit/utilize modern compilers and CPUs

Project tungsten uses **Janino compiler** to reduce code generate time.

Reference

<https://www.youtube.com/watch?v=5ajs8EIPWGI>

Goals of Project Tungsten

Substantially improve the memory and CPU efficiency of Spark applications.

Push performance closer to the limits of modern hardware.

In this talk

- Motivation: why we're focusing on compute instead of IO
- How Tungsten optimizes memory + CPU
- Case study: aggregation
- Case study: record sorting
- Performance results
- Roadmap + next steps

Many big data workloads are now compute bound

NSDI'15:

Making Sense of Performance in Data Analytics Frameworks

Kay Damerow¹, Ryan Rast^{1,2}, Sylvia Ratnasamy¹, Scott Shenker¹, Byung-Gon Chun¹
¹UC Berkeley, ²ICSI, ³VMware, ⁴Southern National University



5

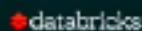
Many big data workloads are now compute bound

NSDI'15:

Making Sense of Performance in Data Analytics Frameworks

Kay Damerow¹, Ryan Rast^{1,2}, Sylvia Ratnasamy¹, Scott Shenker¹, Byung-Gon Chun¹
¹UC Berkeley, ²ICSI, ³VMware, ⁴Southern National University

- “Network optimizations can only reduce job completion time by a median of at most 2%.”
- “Optimizing or eliminating disk accesses can only reduce job completion time by a median of at most 19%.”
- We’ve observed similar characteristics in many Databricks Cloud customer workloads.



5

Why is CPU the new bottleneck?

- Hardware has improved:
 - Increasingly large aggregate I/O bandwidth, such as 10GbE links in networks
 - High bandwidth SSD's or striped HDD arrays for storage
- Spark's I/O has been optimized:
 - many workloads now avoid significant disk I/O by pruning input data that is not needed in a given job
 - new shuffle and network layer implementations

• databricks

intel experience what's inside

Spark summit 2015
DATA SCIENCE AND ENGINEERING AT SCALE
SAN FRANCISCO - JUNE 15 - 17, 2015

Why is CPU the new bottleneck?

- Hardware has improved:
 - Increasingly large aggregate I/O bandwidth, such as 10GbE links in networks
 - High bandwidth SSD's or striped HDD arrays for storage
- Spark's I/O has been optimized:
 - many workloads now avoid significant disk I/O by pruning input data that is not needed in a given job
 - new shuffle and network layer implementations
- Data formats have improved:
 - Parquet, binary data formats

• databricks

intel experience what's inside

Spark summit 2015
DATA SCIENCE AND ENGINEERING AT SCALE
SAN FRANCISCO - JUNE 15 - 17, 2015

Why is CPU the new bottleneck?

- Hardware has improved:
 - increasingly large aggregate IO bandwidth, such as 10GbE links in networks
 - High bandwidth SSDs or striped HDD arrays for storage
- Spark's IO has been optimized:
 - many workloads now avoid a significant disk IO by pruning input data that is not needed in a given job
 - new shuffle and network layer implementations
- Data formats have improved:
 - Parquet, binary data formats
- Serialization and hashing are CPU-bound bottlenecks

• datastax.com



Spark
summit2015

DATA SCIENCE AND ENGINEERING AT SCALE

SAN FRANCISCO - JUNE 15 - 17, 2015

jmmr20



How Tungsten improves CPU & memory efficiency

- **Memory Management and Binary Processing:** leverage application semantics to manage memory explicitly and eliminate the overhead of JVM object model and garbage collection
- **Cache-aware computation:** algorithms and data structures to exploit memory hierarchy
- **Code generation:** exploit modern compilers and CPUs; allow efficient operation directly on binary data

• datastax.com



Spark
summit2015

DATA SCIENCE AND ENGINEERING AT SCALE

SAN FRANCISCO - JUNE 15 - 17, 2015

summit20



The screenshot shows a video player interface. On the left, a presentation slide has the following text:

Generality has a cost, so we
should use semantics and
schema to exploit specificity
instead

On the right, a video frame shows a man in a green t-shirt speaking on stage. The background of the video frame features the "Spark summit 2015" logo and the Golden Gate Bridge at sunset. The video player interface includes a progress bar at 526/2529.

The screenshot shows a video player interface. On the left, a presentation slide has the following text:

The overheads of Java objects
“abcd”

On the right, a video frame shows a man in a green t-shirt speaking on stage. The background of the video frame features the "Spark summit 2015" logo and the Golden Gate Bridge at sunset. The video player interface includes a progress bar at 526/2529.

The overheads of Java objects

“abcd”

- Native: 4 bytes with UTF-8 encoding
- Java: 48 bytes

```
java.lang.String object internals:
OFFSET  SIZE   TYPE DESCRIPTION          VALUE
    0      4   (object header)
    4      4   (object header)
    8      4   (object header)
   12     4   char[]  String.value        "a"
   16      4   int   String.hash         8
   20      4   int   String.hash32       8
Instance size: 24 bytes (reported by Instrumentation API)
```



Garbage collection challenges



- Many big data workloads create objects in ways that are unfriendly to regular Java GC.
- Guest blog on GC tuning: tinyurl.com/db-gc-tuning



sun.misc.Unsafe

- JVM internal API for directly manipulating memory without safety checks (hence "unsafe")
- We use this API to build data structures in both on- and off-heap memory

```
graph LR; A[Data structures with annotations] --> B[Flat data structures]; B --> C[Complex examples]
```

• datacollector



DATA SCIENCE AND ENGINEERING AT SCALE
SAN FRANCISCO - JUNE 15 - 17, 2015



Java object-based row representation

3 fields of type (int, string, string)
with value (123, "data", "bricks")

```
graph LR; GM[GenericMutableRow] --> A[Array]; A --> B[BigInteger(123)]; A --> C[String("data")]; A --> D[String("bricks")]
```

3+ objects; high space overhead; expensive hashCode()

• datacollector



DATA SCIENCE AND ENGINEERING AT SCALE
SAN FRANCISCO - JUNE 15 - 17, 2015



Tungsten's UnsafeRow format

null bit set (1 byte / field)	values (#bytes / field)	variableLength
-------------------------------	-------------------------	----------------

- Bit set for tracking null values.
- Every column appears in the fixed-length values region:

• datacollector 82

intel experience what's inside

Spark summit 2015
DATA SCIENCE AND ENGINEERING AT SCALE
SAN FRANCISCO - JUNE 15 - 17, 2015

immit 201

A man in a green shirt is speaking on stage at a conference.

Tungsten's UnsafeRow format

null bit set (1 byte / field)	values (#bytes / field)	variableLength
-------------------------------	-------------------------	----------------

Other variable length data ↑

- Bit set for tracking null values.
- Every column appears in the fixed-length values region:
 - Non-null values are integers
 - For variable-length values (strings), we store a relative offset into the variable-length data section

• datacollector 81

intel experience what's inside

Spark summit 2015
DATA SCIENCE AND ENGINEERING AT SCALE
SAN FRANCISCO - JUNE 15 - 17, 2015

summ

A man in a green shirt is speaking on stage at a conference.

Tungsten's UnsafeRow format

The diagram illustrates the memory layout of an UnsafeRow. It shows a header with three fields: `null bit mask (1 byte/field)`, `values (8 bytes/field)`, and `variableLength`. Below the header, a note states: "That is, we length data". The `variableLength` field is highlighted with a red arrow pointing to the text "Other is variable data".

- Bit set for tracking null values.
- Every column appears in the fixed-length values region:
 - Integers are int32
 - Any variable-length values (String, List, Map) are stored relative offset in the variable length data section
- Rows are always 8-byte word aligned (size is multiple of 8 bytes)
- Equality comparison and hashing can be performed on raw bytes without requiring additional interpretation

• databricks

Deep Dive Into Project Tungsten Bringing Spark Closer to Bare Metal! - Josh Rosen (Databricks)

The diagram shows an example of an UnsafeRow with the tuple `(123, "data", "bricks")`. It highlights the `null tracking bitmap`, `object ref lengths`, and `field lengths`. A callout box points to the `variableLengthData` section. The `variableLengthData` is annotated with arrows pointing to the string values "data" and "bricks".

• databricks

Spark summit 2015
DATA SCIENCE AND ENGINEERING AT SCALE
SAN FRANCISCO - JUNE 15 - 17, 2015

it 2015

intel experience what's inside

The image shows a presentation slide titled "How we encode memory addresses" on the left, a speaker in a green shirt on the right, and a background featuring the Golden Gate Bridge and an Intel advertisement.

How we encode memory addresses

- Off heap: addresses are raw in-memory pointers.
- On heap: addresses are base object + offset pairs.
- We use our own "pogo table" abstraction to enable more compact encoding of on-heap addresses:

page | offset in page

0
1
-
$n-1$

dataobject PageTable

Spark summit 2015
DATA SCIENCE AND ENGINEERING AT SCALE
SAN FRANCISCO - JUNE 15 - 17, 2015

summit 2015

intel experience what's inside

The image shows a presentation slide titled "How we encode memory addresses" on the left, a speaker in a green shirt on the right, and a background featuring the Golden Gate Bridge and an Intel advertisement.

How we encode memory addresses

- Off heap: addresses are raw in-memory pointers.
- On heap: addresses are base object + offset pairs.
- We use our own "pogo table" abstraction to enable more compact encoding of on-heap addresses:

page | offset in page

0
1
-
$n-1$

dataobject PageTable

Data page (base object)

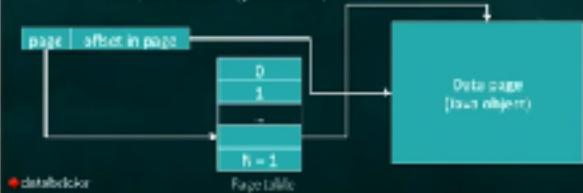
Spark summit 2015
DATA SCIENCE AND ENGINEERING AT SCALE
SAN FRANCISCO - JUNE 15 - 17, 2015

summit 2015

intel experience what's inside

How we encode memory addresses

- Off heap: addresses are raw in-memory pointers.
- On heap: addresses are base object + offset pairs.
- We use our own “page table” abstraction to enable more compact encoding of on-heap addresses:



Spark
summit2015
DATA SCIENCE AND ENGINEERING AT SCALE
SAN FRANCISCO - JUNE 15 - 17, 2015



java.util.HashMap

- Huge object overheads
- Poor memory locality
- Size estimation is hard

• databricks

• intel

Spark
summit2015
DATA SCIENCE AND ENGINEERING AT SCALE
SAN FRANCISCO - JUNE 15 - 17, 2015



Tungsten's BytesToBytesMap

Memory page

key	value	key	value
key	value	key	value
key	value	key	value

array

• data[addr]

- Low space overheads
- Good in-memory locality, especially for scans

intel experience what's inside

Spark summit 2015
DATA SCIENCE AND ENGINEERING AT SCALE
SAN FRANCISCO - JUNE 15 - 17, 2015

Code generation

- Generic evaluation of expression logic is very expensive on the JVM
 - Virtual function calls
 - Branches based on express or type
 - Object creation due to primitive boxing
 - Memory consumption by boxed primitive objects
- Generating custom bytecode can eliminate these overheads

Including SELECT & FILTER overhead

generated bytecode	Java
Debug	~10%
Minimizer	~1%

intel experience what's inside

Spark summit 2015
DATA SCIENCE AND ENGINEERING AT SCALE
SAN FRANCISCO - JUNE 15 - 17, 2015

Code generation

- Generic evaluation of expression logic is very expensive on the JVM
 - Virtual function calls
 - Branches based on expression type
 - Object creation due to primitive boxing
 - Memory consumption by boxed primitive objects
- Generating custom bytecode can eliminate these overheads

Evaluating "SELECT a + a - a"
(query time in seconds)

Implementation	Time (seconds)
Interpreted Project	36.66
Code gen	9.36
Handwritten	9.33

Deep Dive Into Project Tungsten: Bringing Spark Closer to Bare Metal | Josh Rosen (Databricks) 18

Code generation

- Project Tungsten uses the Janino compiler to reduce code generation time.
- Spark 1.5 will greatly expand the number of expressions that support code generation:
 - [SPARK-8159](#)

● databricks 19

Example: aggregation optimizations in DataFrames and Spark SQL

```
df.groupBy("department").agg(max("age"), sum("expense"))
```



SAN FRANCISCO - JUNE 15 - 17, 2015

Example: aggregation optimizations in DataFrames and Spark SQL



SAN FRANCISCO - JUNE 15 - 17, 2015

intel experience what's inside



SAN FRANCISCO - JUNE 15 - 17, 2015



Optimized record sorting in Spark SQL + DataFrames ([SPARK-7082](#))

- AlphaSort-style prefix sort:
 - Store prefixes of sort keys inside the sort pointer array
 - During sort, compare prefixes to short-circuit and avoid full record comparisons
- Use this to build external sort-merge join to support joins larger than memory

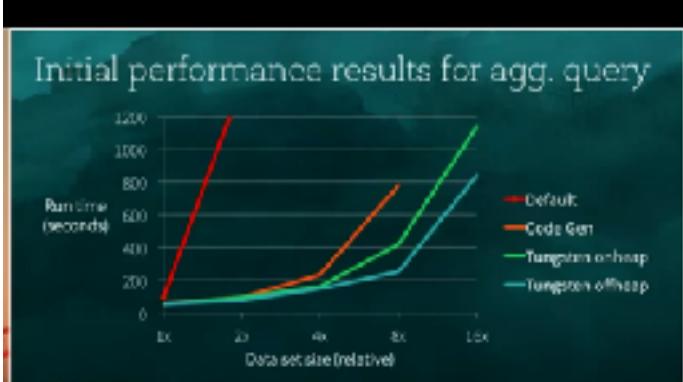
Native layout 

Code-friendly layout 

• [datacollector](#)



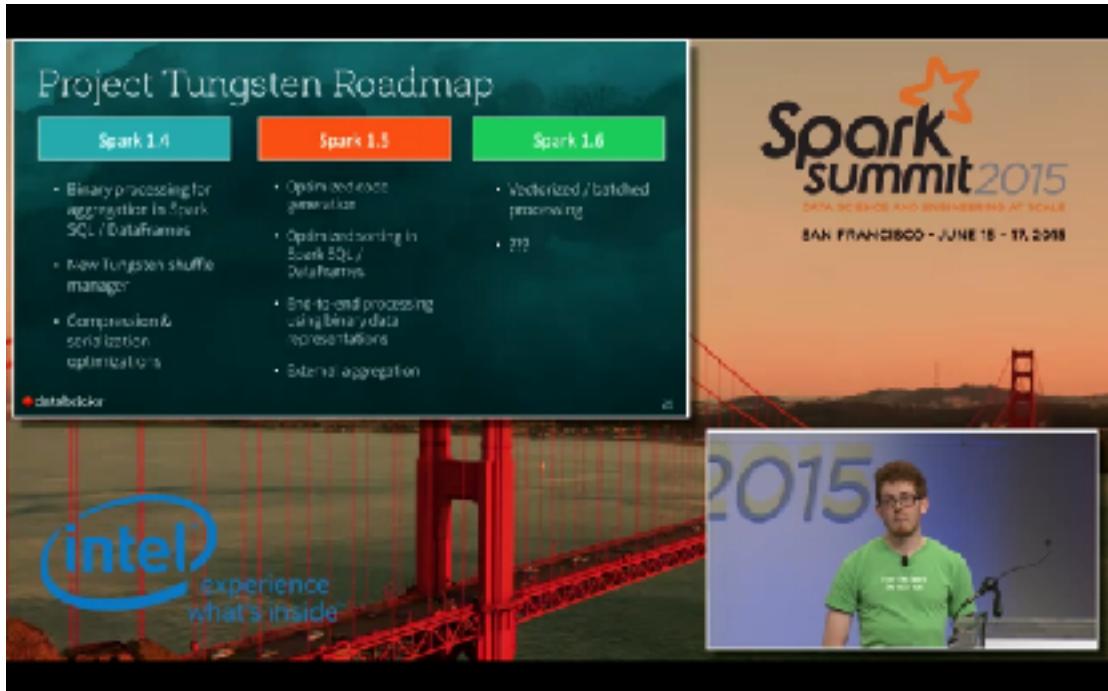
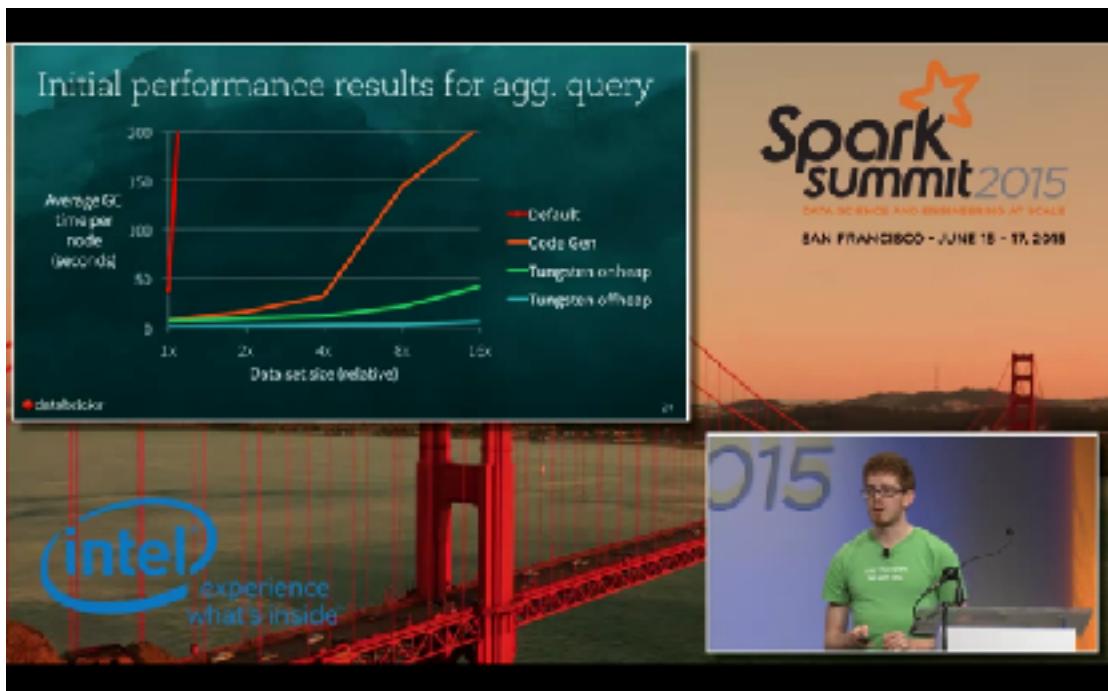
Initial performance results for agg. query



Data set size (relative)	Default (s)	Code Gen (s)	Tungsten on-heap (s)	Tungsten off-heap (s)
1x	~100	~100	~100	~100
2x	~400	~200	~150	~100
4x	~1000	~400	~250	~150
8x	~2000	~800	~400	~250
16x	~4000	~1600	~800	~400

• [datacollector](#)





Which Spark jobs can benefit from Tungsten?

- **DataFrames**
 - Java
 - Scala
 - Python
 - R
- **Spark SQL queries**
- **Some Spark RDD API programs**, via general serialization + compression optimizations

```
logs.join(  
  users,  
  logs.userId == users.userId,  
  "left_outer") \\  
.groupByKey("userId").agg({"#": "count"})
```

• databricks

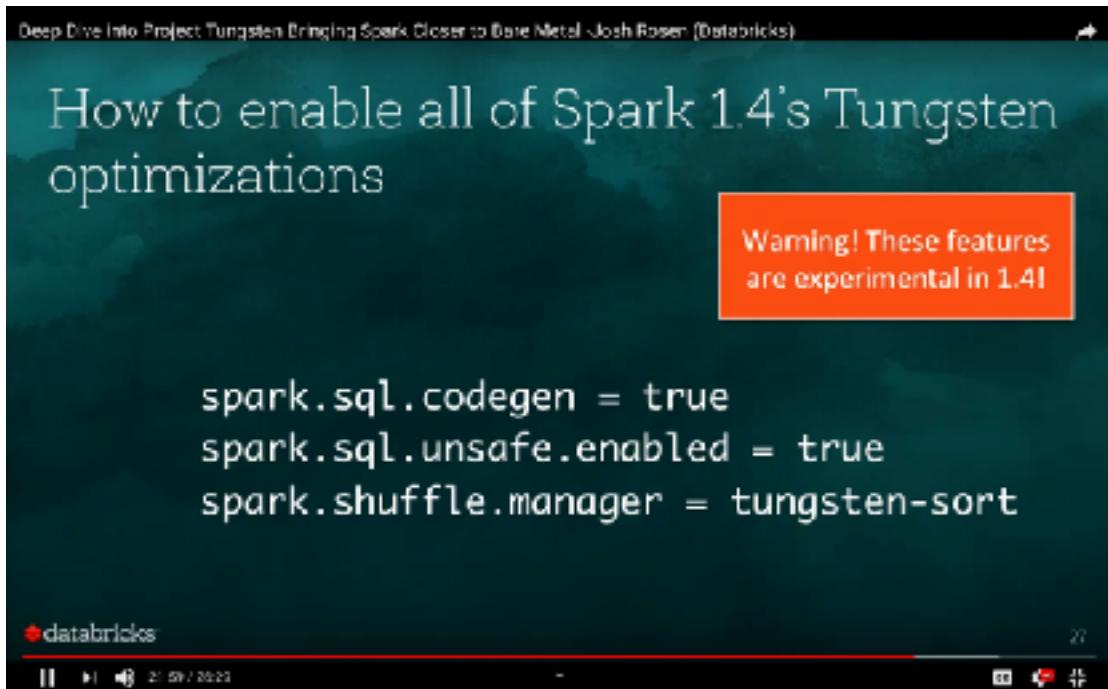
Spark summit 2015
DATA SCIENCE AND ENGINEERING AT SCALE
SAN FRANCISCO - JUNE 15 - 17, 2015

Which Spark jobs can benefit from Tungsten?

- **DataFrames**
 - Java
 - Scala
 - Python
 - R
- **Spark SQL queries**
- **Some Spark RDD API programs**, via general serialization + compression optimizations

```
logs.join(  
  users,  
  logs.userId == users.userId,  
  "left_outer") \\  
.groupByKey("userId").agg({"#": "count"})
```

• databricks



<https://community.hortonworks.com/articles/72502/what-is-tungsten-for-apache-spark.html>

References:

<https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html>

SparkContext(sc)

It resides in Driver program to communicate between the Spark Driver and cluster to get connect with ClusterManager(Mesos, Yarn)

When spark-shell launch, SparkContext is created by default as "SC"

```
$spark-shell --master yarn
```

```
Spark context Web UI available at http://play2:4040  
Spark context available as 'sc' (master = yarn, app id = application_1
```

```
version 2.4.0
```

SC:

- The driver program use the SparkContext to connect and communicate with the cluster and it helps in executing and coordinating the Spark job with the resource managers like YARN or Mesos.
- Using SparkContext you can actually get access to other contexts like SQLContext and HiveContext.

- Using SparkContext we can set configuration parameters to the Spark job.
- If you are in spark-shell, a SparkContext is already available for you and is assigned to the variable sc.
- If you don't have a SparkContext already, you can create one by first creating a SparkConf first.

```
//set up the spark configuration
val sparkConf = new SparkConf().setAppName("hirw").setMaster("yarn")
//get SparkContext using the SparkConf
val sc = new SparkContext(sparkConf)
```

SQL Context (sqlContext)

SQLContext is your gateway to Spark-SQL. Here is how you create a SQL-Context using the SparkContext.

```
// sc is an existing SparkContext.
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
```

Hive Context(hiveContext)

- HiveContext is your gateway to Hive.
- HiveContext has all the functionalities of a SQLContext.
- In fact, if you look at the API documentation you can see that HiveContext extends SQLContext, meaning, it has support the functionalities that SQLContext support plus more (Hive specific functionalities)

```
public class HiveContext extends SQLContext implements Logging

//Way to create HiveContext
// sc is an existing SparkContext
val sqlContext = new org.apache.spark.sql.hive.HiveContext(sc)
```

Spark Session(spark)

SparkSession was introduced in Spark 2.0 to make it easy for the developers so we don't have worry about different contexts and to streamline the access to different contexts. By having access to SparkSession, we automatically have access to the SparkContext

Or

A SparkSession can be created using a builder pattern. The builder will automatically reuse an existing SparkContext if one exists; and create a SparkContext if it does not exist. Configuration options set in the builder are automatically propagated over to Spark and Hadoop during I/O.

```
// A SparkSession can be created using a builder pattern
import org.apache.spark.sql.SparkSession
val sparkSession = SparkSession.builder
```

```
.master("local")
.appName("my-spark-app")
.config("spark.some.config.option", "config-value")
.getOrCreate()
```

In Glance:

[Explanation from spark source code under branch-2.1](#)

SparkContext (SC)

- Main entry point for Spark functionality.
- A SparkContext represents the connection to a Spark cluster, and can be used to create RDDs, accumulators and broadcast variables on that cluster.
- Only one SparkContext may be active per JVM.
- You must stop() the active SparkContext before creating a new one.
- This limitation may eventually be removed; see SPARK-2243 for more details.

JavaSparkContext:

- A Java-friendly version of [[org.apache.spark.SparkContext]] that returns [[org.apache.spark.api.java.JavaRDD]]s and works with Java collections instead of Scala ones.
- Only one SparkContext may be active per JVM.
- You must stop() the active SparkContext before creating a new one.
- This limitation may eventually be removed; see SPARK-2243 for more details.

SQLContext (sqlContext)

- The entry point for working with structured data (rows and columns) in Spark 1.x.
- As of Spark 2.0, this is replaced by [[SparkSession]].
- However, we are keeping the class here for backward compatibility.

SparkSession (spark)

The entry point to programming Spark with the **Dataset** and **DataFrame** API.

NOTE:

This way we can create SparkSession for Sql operation on Dataframe

```
val sparkSession=SparkSession.builder().getOrCreate();
```

Second way is to create SparkSession for Sql operation on Dataframe as well as Hive Operation.

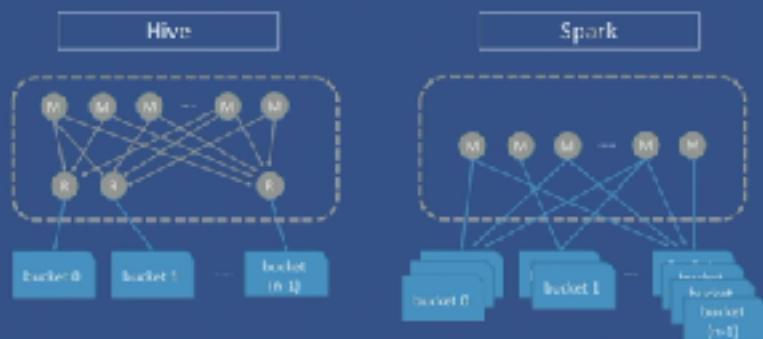
```
val sparkSession=SparkSession.builder().enableHiveSupport().getOrCreate()
```

Bucketing Hive vs Spark

Bucketing =
pre-(shuffle + sort) inputs
on join keys



Bucketing semantics of Spark vs Hive





SPARK SUMMIT

Bucketing semantics of Spark vs Hive

	Hive	Spark
Model	Optimizes reads, which are costly	Writes are cheaper; reads are costlier
Unit of bucketing	A single file represents a single bucket	A collection of files together comprise a single bucket
Sort ordering	Each bucket is sorted globally. This makes it easy to optimize queries reading this data	Each file is individually sorted but the "bucket" as a whole is not globally sorted
Hashing Function	Hive's built-in hash	Murmur3 hash

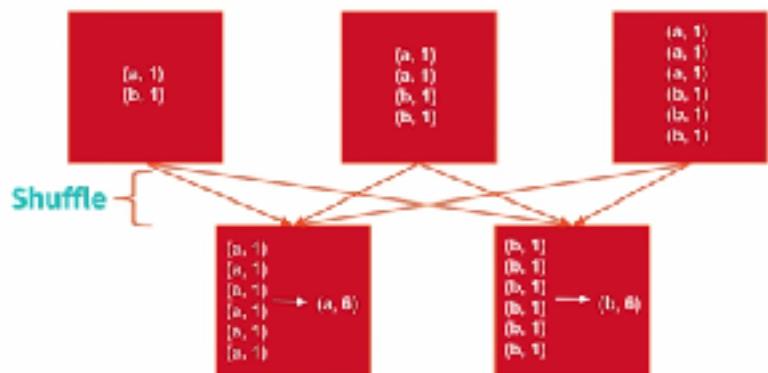
[groupByKey\(\) vs ReduceByKey\(\)](#)

Strata+ Hadoop WORLD



STRATACONF.COM
FEB 17-20, 2015
SAN JOSE, CA

GroupByKey: Shuffle Step



With GroupByKey, all the data is wastefully sent over the network and collected on the reduce workers.

databricks

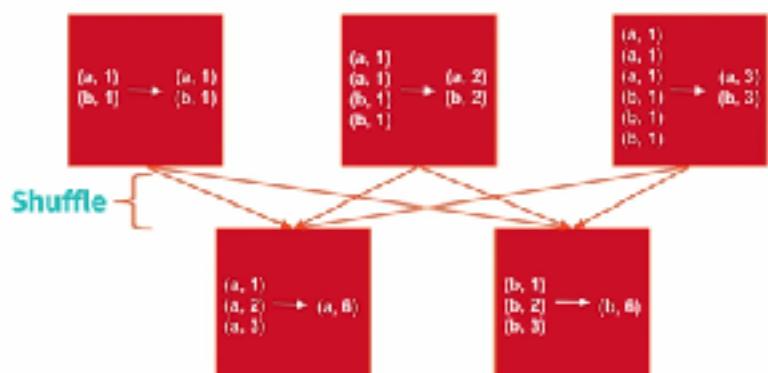
Everyday I'm Shuffling - Tips for Writing Better Apache Spark Programs

Strata+ Hadoop WORLD



STRATACONF.COM
FEB 17-20, 2015
SAN JOSE, CA

ReduceByKey: Shuffle Step



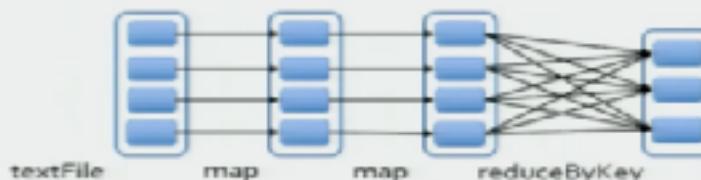
With ReduceByKey, data is combined so each partition outputs at most one value for each key to send over the network.

databricks

Example

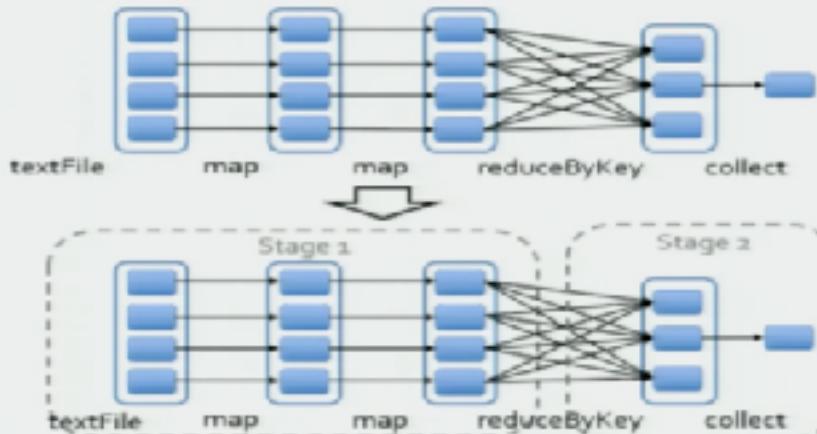
```
sc.textFile("/some-hdfs-data")
    .map(line => line.split("\t"))
    .map(parts =>
        (parts[0], int(parts[1])))
    .reduceByKey(_ + _, 3)
```

RDD[String]
RDD[List[String]]
RDD[(String, Int)]
RDD[(String, Int)]



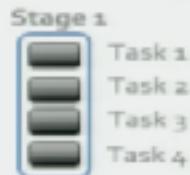
databricks

Execution Graph



databricks

Stage execution



Create a task for each partition
in the new RDD

Serialize task

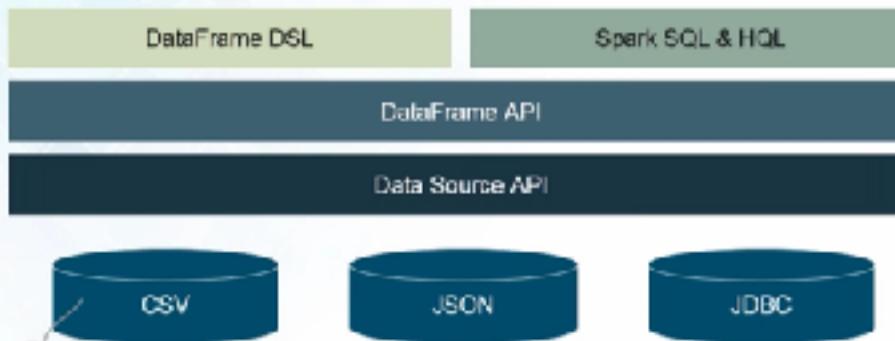
Schedule and ship task to slaves

 databricks

Datasets & DataFrames

Spark SQL Architecture

edureka!





Some code to read Wikipedia

```
val rdd = sc.textFile("http://www.wikipediacounts.gz")
val parsedRDD = rdd.flatMap { line => line.split("\t\t\t\t") match {
    case Array(project, page, numRequests, ...) => Some((project, page, numRequests))
    case _ => None
  }
}

// Filter only English pages ; count pages and requests to it.
parsedRDD.filter { case (project, page, numRequests) => project == "en" }.
  map { case (_, page, numRequests) => (page, numRequests) }.
  reduceByKey(_ + _).
  take(100).foreach { case (page, requests) => println(s"page: $page requests: $requests") }
```

 **SPARK SUMMIT**
powered by 





Structured APIs In Spark

	SQL	DataFrames	Datasets
Syntax Errors	Runtime	Compile Time	Compile Time
Analysis Errors	Runtime	Runtime	Compile Time

Analysis errors are reported before a distributed job starts.

 **SPARK SUMMIT**
powered by 



DataFrame API code.

```
// convert RDD to DF with column names  
val df = parsedRDD.toDF("project", "page", "numRequests")  
//filter, groupBy, sum, and then agg()  
df.filter($"project" === "en").  
  groupBy($"page").  
  agg(sum($"numRequests").as("count")).  
  limit(100).  
  show(100)
```

project	page	numRequests
en	23	45
en	24	200

databricks



SPARK
SUMMIT

powered by databricks

Why structure APIs?

DataFrame

```
data.groupBy("dept").avg("age")
```

SQL

```
select dept, avg(age) from data group by 1
```

RDD

```
data.map { case (dept, age) => dept -> (age, 1) }  
.reduceByKey { case ((a1, c1), (a2, c2)) => (a1 + a2, c1 + c2)}  
.map { case (dept, (age, c)) => dept -> age / c }
```

databricks



SPARK
SUMMIT

powered by databricks

Using Catalyst in Spark SQL

The diagram illustrates the Catalyst optimization pipeline. It starts with SQL AST, DataFrames, and Datasets as inputs. These feed into the Unresolves Logical Plan stage. This is followed by the Logical Plan stage, which is then optimized into an Optimized Logical Plan. This plan is then converted into a Physical Plan. The Physical Plan interacts with a Catalog and a Code Model to produce a Selected Physical Plan, which finally results in RDDs.

Analysis: analyzing a logical plan to resolve references

Logical Optimization: logical plan optimization

Physical Planning: Physical planning

Code Generation: Compile parts of the query to Java bytecode

SPARK SUMMIT
organized by databricks

DataFrame Optimization

The diagram shows the optimization of a DataFrame query. The query is defined by the code:

```
users.join(events, users("id") === events("uid")).  
filter(events("date") > "2015-01-01")
```

The optimization process is shown in three stages:

- Logical Plan:** A join node with two children: eventsFile and usersTable. A filter node is above the join node.
- Physical Plan:** A join node with two children: scanEvents and filter. The filter node has a child: scanUsers.
- Physical Plan with Predicate Pushdown and Column Pruning:** A join node with two children: optimizedScanEvents and optimizedScanUsers.

SPARK SUMMIT
organized by databricks

Dataset API in Spark 2.x

Type-safe: operate on domain objects with compiled lambda functions

```
val df = spark.read.json("people.json")
// Convert data to domain objects.
case class Person(name: String, age: Int)
val ds: Dataset[Person] = df.as[Person]
```

```
val filterDS = ds.filter(p => p.age > 3)
```

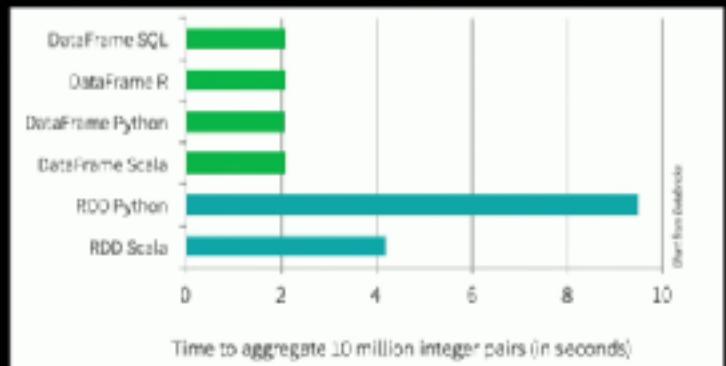


SPARK
SUMMIT

organized by Databricks

databricks

DataFrames are Faster than RDDs



SPARK
SUMMIT

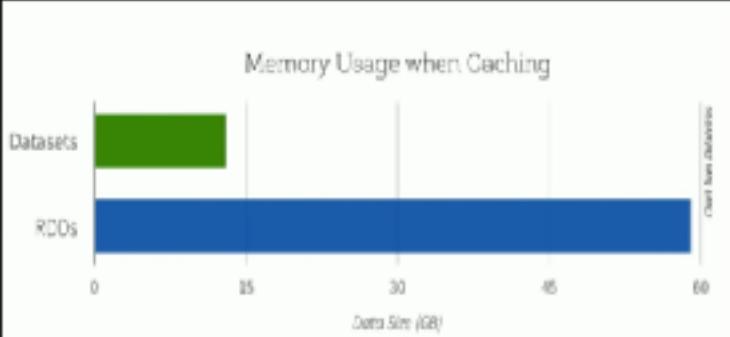
organized by Databricks

databricks



Datasets < Memory RDDs

Memory Usage when Caching



Storage Type	Data Size (GB)
Datasets	~14
RDDs	~58

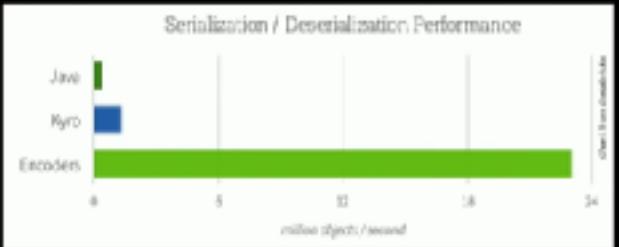
Source: [databricks](#)

SPARK SUMMIT
organized by 



Datasets Faster...

Serialization / Deserialization Performance



Format	million objects / second
Java	~1.5
Pyro	~2.5
Encoders	~28

Source: [databricks](#)

SPARK SUMMIT
organized by 

A Tale of Three Apache Spark APIs: RDDs, DataFrames, and Datasets -...

Watch later Share

DataFrames & Datasets

Why

- High-level APIs and DSL
- Strong Type-safety
- Ease-of-use & Readability
- What-to-do

When

- Structured Data schema
- Code optimization & performance
- Space efficiency with Tungsten

SPARK SUMMIT
organized by databricks

27.59 / 31.18

YouTube

Putting all Together: Conclusion

Datasets

RDDs

- Functional Programming
- Type-safe

Dataframes

- Functional
- Catalyst query optimization
- Tungsten direct/packed RAM
- All code generated
- Sorting/partitioning without repartitioning

Spark

SPARK SUMMIT
organized by databricks

Adaptive Execution Engine/shuffling

Catalyst Optimizer:

Catalyst is an excellent optimizer in SparkSQL, provides open interface for **rule-based optimization in planning stage.** However, the static (rule-based) optimization will not consider any data distribution at runtime.

Adaptive Execution Engine:

A technology called Adaptive Execution has been introduced since Spark 2.0 and aims to cover this part, but still pending in early stage. We enhanced the existing Adaptive Execution feature, **and focus on the execution plan adjustment at runtime** according to different staged intermediate outputs, like

- set partition numbers for joins and aggregations,
- avoid unnecessary data shuffling and disk IO,
- handle data skew cases, and
- even optimize the join order like CBO etc..

In the benchmark comparison experiments, this feature save huge manual efforts in tuning the parameters like

- the shuffled partition number, which is error-prone and misleading.

Challenges with Catalyst optimizer:

- Shuffle Partition
- Number Best Execution Plan
- Data Skew

RunTime Decisions:

- Auto Setting the Number of Reducers
- Execution Plan Optimization at Runtime
- Handling Skewed Join / HandlingSkewedShuffleJoinInputData
- DeterminingtheJoinStrategy

For more info check design doc at:

<https://issues.apache.org/jira/browse/SPARK-9850>

<https://software.intel.com/en-us/articles/spark-sql-adaptive-execution-at-100-tb>

An Adaptive Execution Engine For Apache Spark SQL - Carson Wang

Watch later Share

intel

AN ADAPTIVE EXECUTION ENGINE FOR APACHE SPARK SQL

Carson Wang (carson.wang@intel.com)
Yucai Yu (yucai.yu@intel.com)
Hao Cheng (hao.cheng@intel.com)

SPARK SUMMIT

hosted by databricks

II 0:09 / 29:32 YouTube

An Adaptive Execution Engine For Apache Spark SQL - Carson Wang

Watch later Share

Agenda

- Challenges In Spark SQL* High Performance
- Adaptive Execution Background
- Adaptive Execution Architecture
- Benchmark Result

*Other names and brands may be claimed as the property of others.

SPARK SUMMIT

hosted by databricks

II 0:18 / 29:32 YouTube

An Adaptive Execution Engine For Apache Spark SQL - Carson Wang

Watch later Share



Challenges in Tuning Shuffle Partition Number

- Partition Num $P = \text{spark.sql.shuffle.partition}$ (200 by default)
- Total Core Num $C = \text{Executor Num} * \text{Executor Core Num}$
- Each Reduce Stage runs the tasks in (P / C) rounds

* Other names and brands may be claimed as the property of others.

SPARK SUMMIT

sponsored by databricks

II 0:42 / 29:32 YouTube



Shuffle Partition Challenge 1

- Partition Num Too Small : Spill, COM
- Partition Num Too Large : Scheduling overhead, More IO requests, Too many small output files
- Tuning method: Increase partition number starting from $C, 2C, \dots$ until performance begin to drop



An Adaptive Execution Engine For Apache Spark SQL - Carson Wang

Watch later Share

Shuffle Partition Challenge 2

- The same Shuffle Partition number doesn't fit for all Stages
- Shuffle data size usually decreases during the execution of the SQL query

Question: Can we set the shuffle partition number for each stage automatically?

SPARK SUMMIT

powered by databricks

2:56 / 29:32

YouTube

Spark SQL* Execution Plan

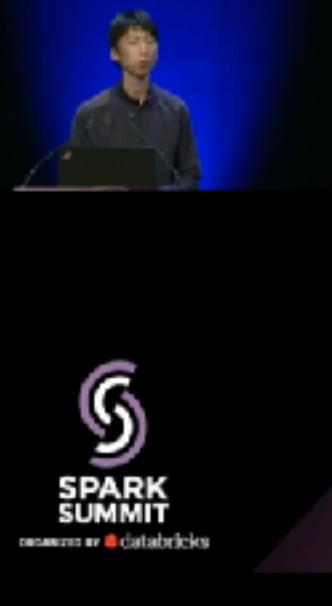
```
graph LR; SQLAST[SQL AST] --> UnresolvedLogicalPlan[Unresolved Logical Plan]; Dataframe[Dataframe] --> UnresolvedLogicalPlan; Datasets[Datasets] --> UnresolvedLogicalPlan; UnresolvedLogicalPlan --> LogicalPlan[Logical Plan]; LogicalPlan --> OptimizedLogicalPlan[Optimized Logical Plan]; OptimizedLogicalPlan --> PhysicalPlan[Physical Plan]; PhysicalPlan --> SelectedPhysicalPlans[Selected Physical Plans]; SelectedPhysicalPlans --> RDD[RDD];
```

The execution plan is fixed after planning phase.

*Other names and brands may be claimed as the property of others.
Image from <https://databricks.com/blog/2015/07/01/spark-and-gremlin-from-alpha-to-gamma.html>

SPARK SUMMIT

powered by databricks

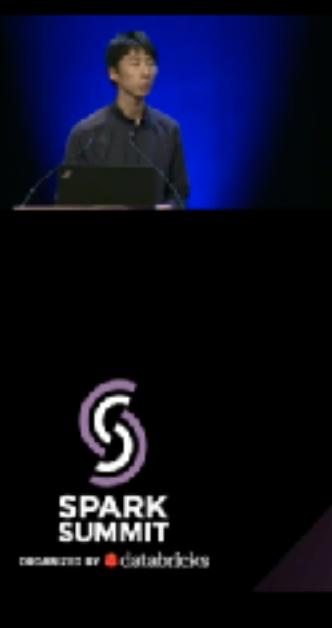


Spark SQL* Join Selection

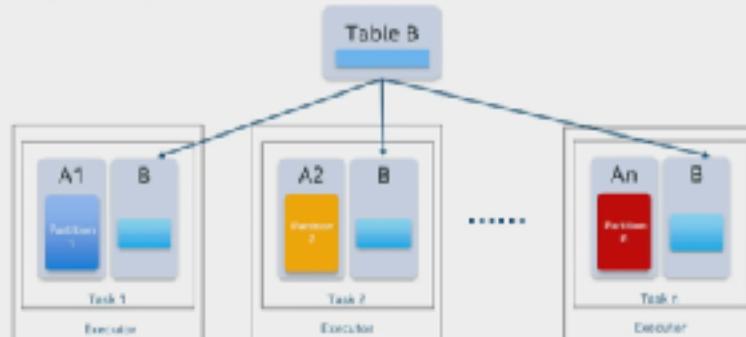
```
SELECT xxx  
FROM A  
JOIN B  
ON A.Key1 = B.Key2
```

*Other names and brands may be claimed as the property of others.

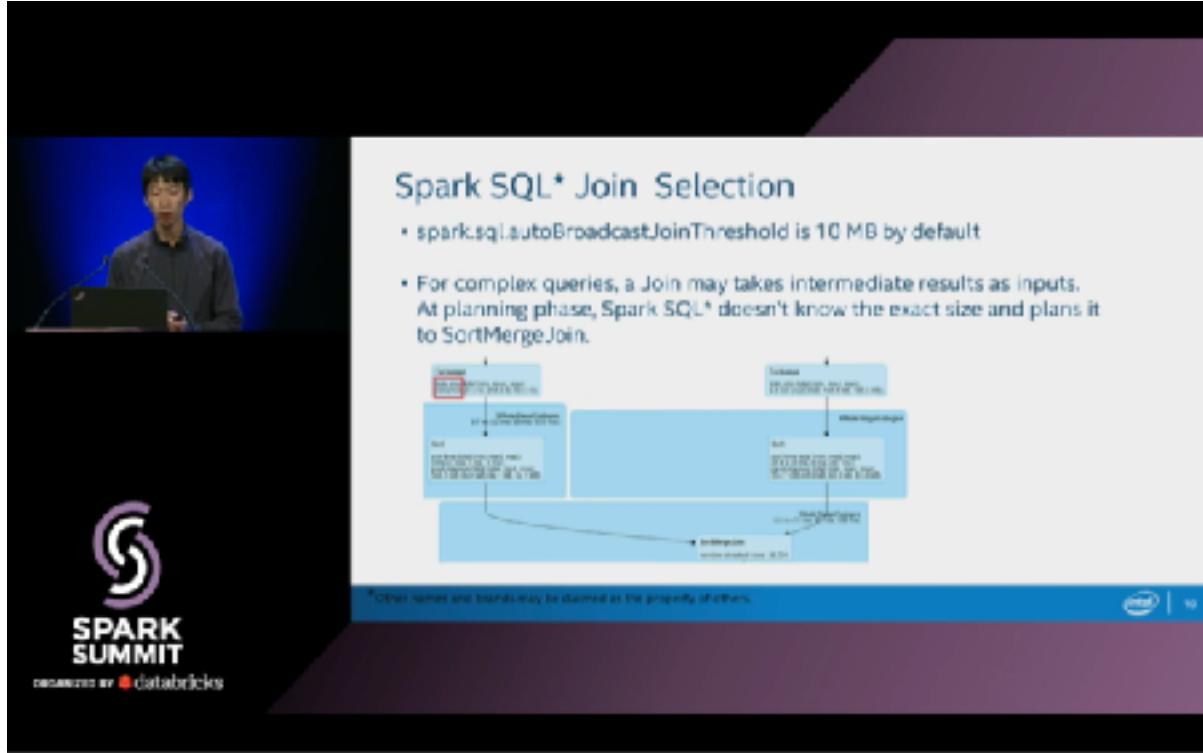
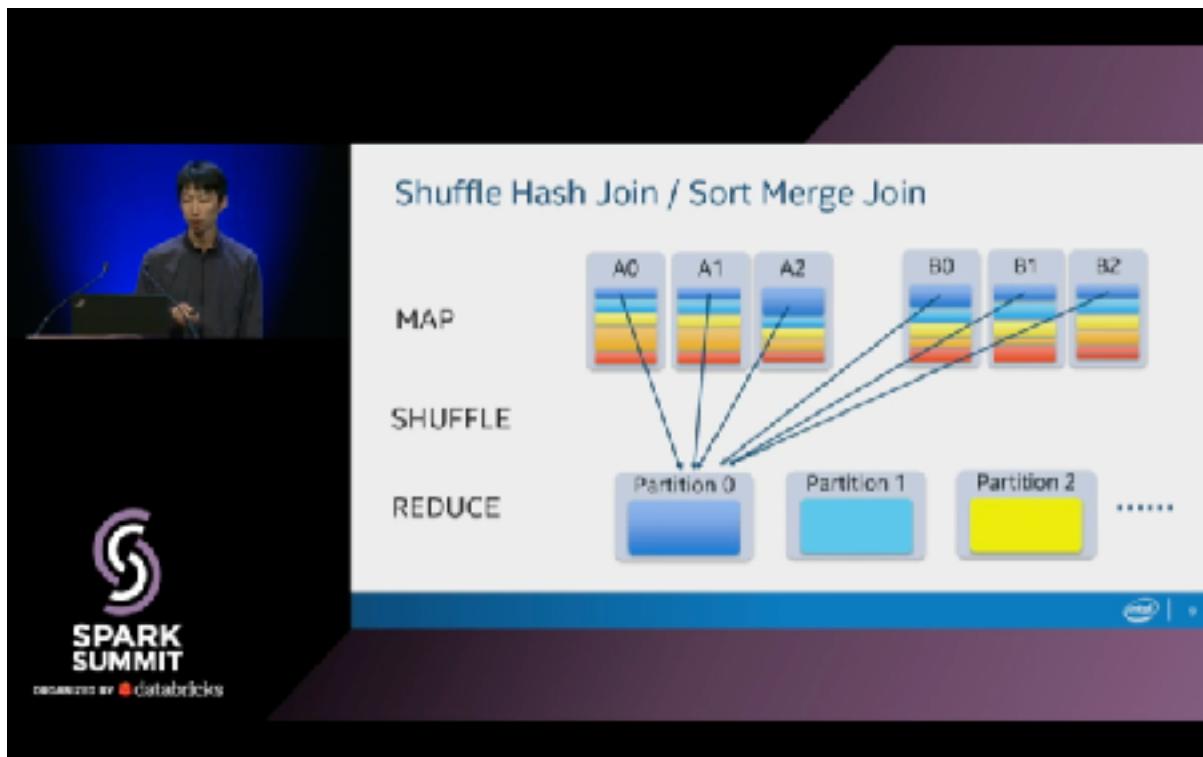
SPARK SUMMIT
powered by databricks



Broadcast Hash Join



SPARK SUMMIT
powered by databricks





SPARK SUMMIT
powered by  databricks

Spark SQL* Join Selection

- `spark.sql.autoBroadcastJoinThreshold` is 10 MB by default.
- For complex queries, a Join may takes intermediate results as inputs. At planning phase, Spark SQL* doesn't know the exact size and plans it to SortMergeJoin.



Question: Can we optimize the execution plan at runtime based on the runtime statistics?

Other names and brands may be trademarks of their respective owners.

© Intel | 10



SPARK SUMMIT
powered by  databricks

Data Skew in Join

- Data in some partitions are extremely larger than other partitions.
- Data skew is a common source of slowness for Shuffle Joins.

Statistic	Min	Q1 (Median)	Median	Q3 (95th Percentile)	Max
Execution Time	0.000	0.00	0.00	0.00	0.000
Input Inspection Time	0.000	0.00	0.00	0.00	0.00
old Time	0.000	0.00	0.00	0.00	0.00
Row Annotations Time	0.000	0.00	0.00	0.00	0.00
Sampling Row Time	0.000	0.00	0.00	0.00	0.00
TFB ExecutionTime	0.000	0.00	0.00	0.00	0.00
Shuffle Data Transfer Time	0.000	0.00	0.00	0.00	0.00
Shuffle Read Time Percentile	0.000	0.000	0.000	0.000	0.000
Shuffle Write Time Percentile	0.000	0.000	0.000	0.000	0.000
Shuffle Write Rows Percentile	0.000	0.000	0.000	0.000	0.000
Shuffle Write Time	0.000	0.00	0.00	0.00	0.000
Shuffle Write Bytes	0.000	0.00	0.00	0.00	0.000

© Intel | 11



Ways to Handle Skewed Join nowadays

- Increase shuffle partition number
- Increase BroadcastJoin threshold to change Shuffle Join to Broadcast Join
- Add prefix to the skewed keys
-



next | u



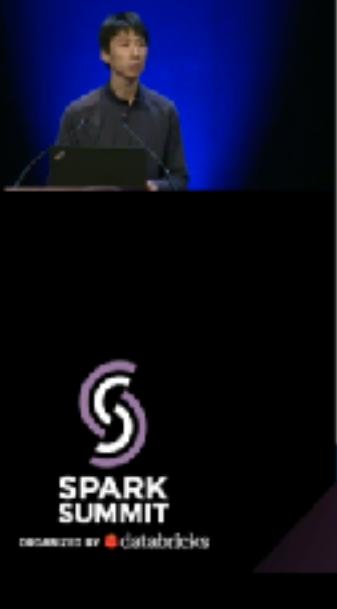
Ways to Handle Skewed Join nowadays

- Increase shuffle partition number
- Increase BroadcastJoin threshold to change Shuffle Join to Broadcast Join
- Add prefix to the skewed keys
-

Question 3: These involve many manual efforts and are limited. Can we handle skewed join at runtime automatically?



next | u



Adaptive Execution Background

- SPARK-9850: Adaptive execution in Spark*
- SPARK-9851: Support submitting map stages individually in DAGScheduler
- SPARK-9858: Introduce an ExchangeCoordinator to estimate the number of post-shuffle partitions.

*Other names and brands may be claimed as the property of others.

 | 11

SPARK SUMMIT
powered by 



A New Adaptive Execution Engine in Spark SQL*

*Other names and brands may be claimed as the property of others.

 | 11

SPARK SUMMIT
powered by 

Adaptive Execution Architecture

Execution Plan → Spark SQL → DAG of RDDs → Execute the Stage

Overhead Exchange → Adaptive Execution → Execute the Stage

Spark SQL
Adaptive Execution

DAG of RDDs

Execute the Stage

Overhead Exchange

Overhead Exchange → Execute the Stage

Intel | 18

Auto Setting the Number of Reducers

- 5 initial reducer partitions with size [70 MB, 30 MB, 20 MB, 10 MB, 50 MB]
- Set target size per reducer = 64 MB. At runtime, we use 3 actual reducers.
- Also support setting target row count per reducer.

Map Task 1

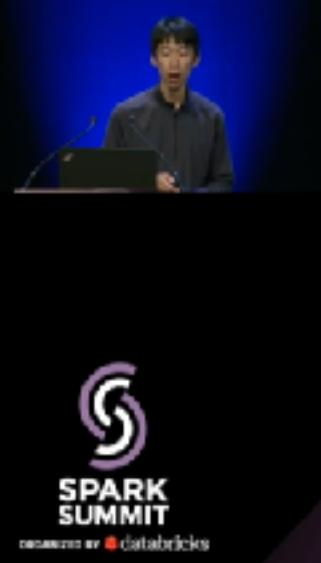
Map Task 2

Reduce Task 1

Reduce Task 2

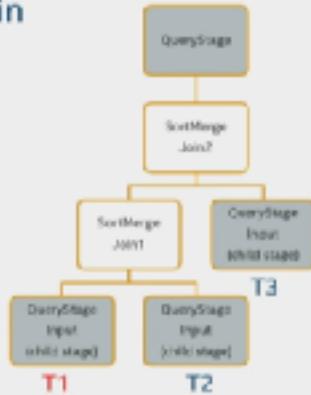
Reduce Task 3

Intel | 18



Shuffle Join => Broadcast Join Example 1

- T1 < broadcast threshold
- T2 and T3 > broadcast threshold
- In this case, both Join1 and Join2 are not changed to broadcast join

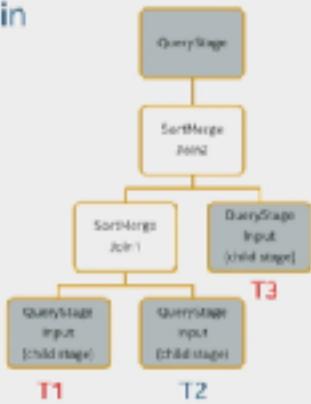


10 | 11



Shuffle Join => Broadcast Join Example 2

- T1 and T3 < broadcast threshold
- T2 > broadcast threshold
- In this case, both Join1 and Join2 are changed to broadcast join



10 | 11

Remote Shuffle Read => Local Shuffle Read

The diagram illustrates the transformation of a remote shuffle read into a local shuffle read. It shows two stages of data processing:

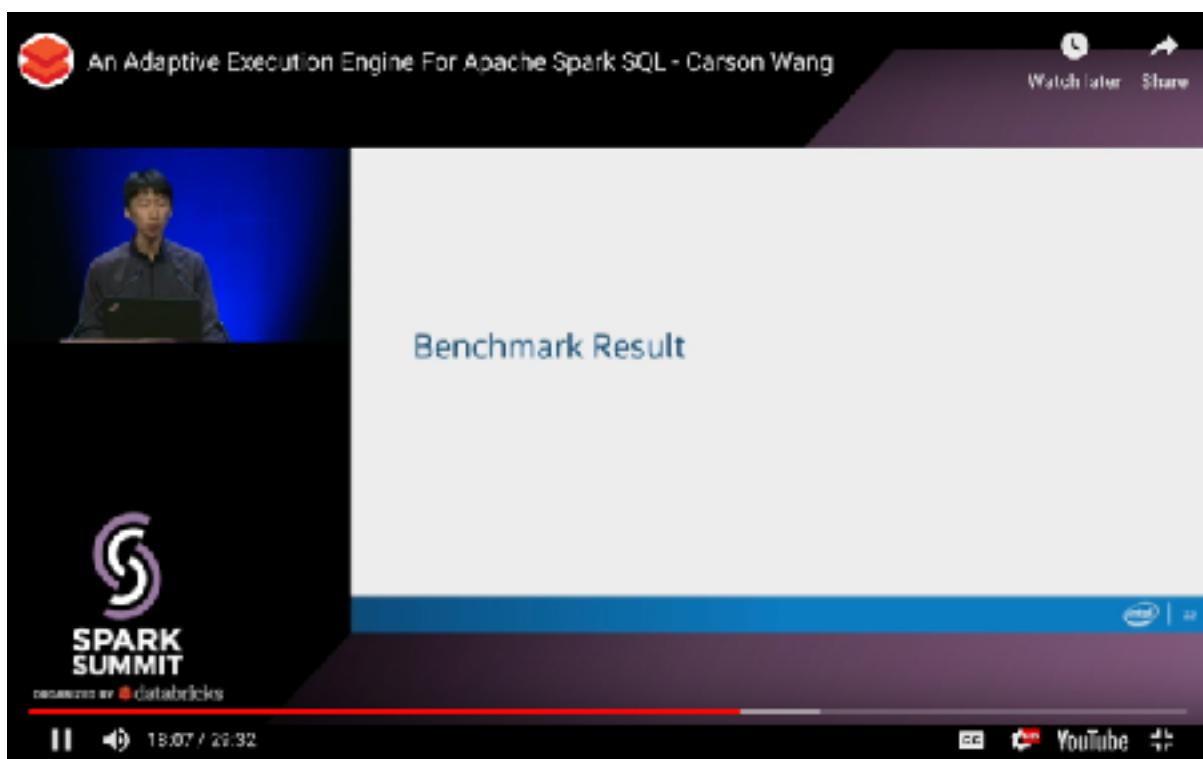
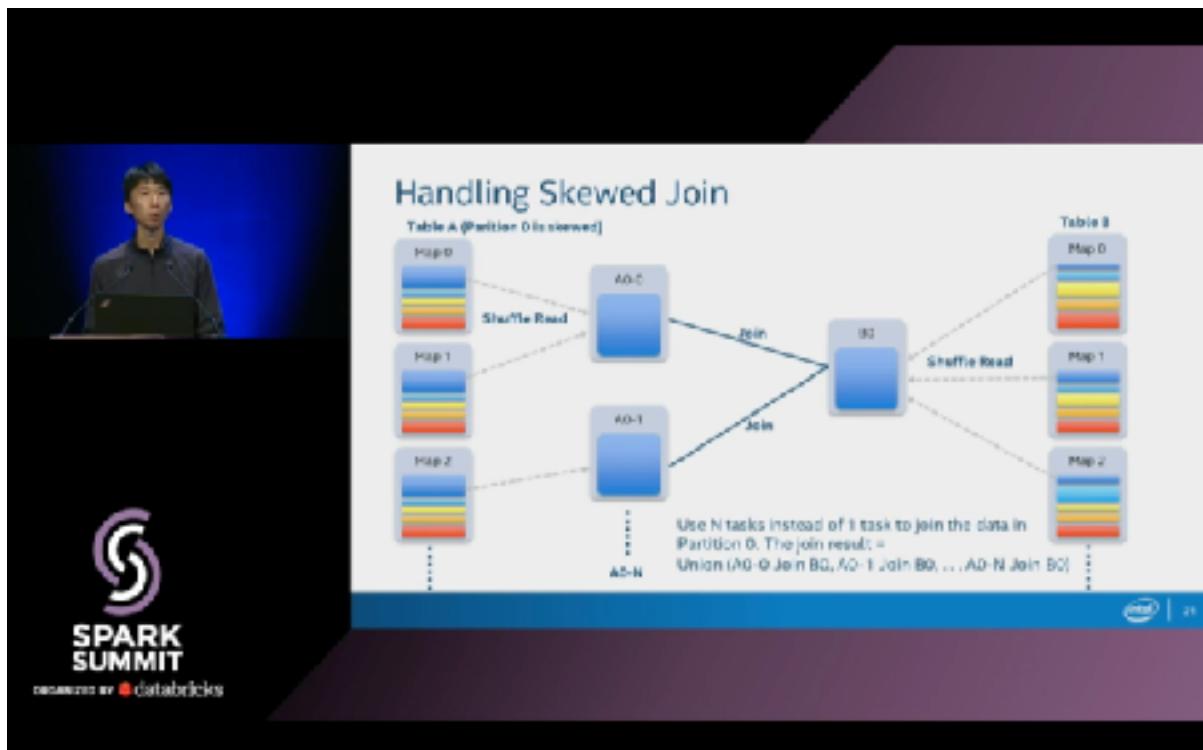
- Remote Shuffle Read:** This stage involves two nodes, Node 1 and Node 2. Node 1 has map output partitions A0 and B0. Node 2 has reduce tasks Task 1, Task 2, and Task 3. An arrow labeled "Shuttle Join" points from Node 1 to Node 2, indicating the movement of data from one node to another.
- Local Shuffle Read:** This stage shows the data after the shuttle join. Node 1 now has map output partitions A1 and B1. Node 2 has two reduce tasks: Task 1 (containing A0 and B0) and Task 2 (containing A1). An arrow labeled "Broadcast Join" points from Node 1 to Node 2, indicating that the joined data is broadcasted to Node 2 for local processing.

SPARK SUMMIT
sponsored by databricks

Skewed Partition Detection at Runtime

- After executing child stages, we calculate the data size and row count of each partition from MapStatus.
- A partition is skewed if its data size or row count is N times larger than the median, and also larger than a pre-defined threshold.

SPARK SUMMIT
sponsored by databricks





Cluster Setup

Hardware		Specs
Slave	Processor	Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.20GHz (8 cores)
	Memory	256 GB
	Disk	7x 400 GB SSD
	Network	10 Gigabit Ethernet
Master	CPU	Intel(R) Xeon(R) CPU E5-2696 v4 @ 2.20GHz (16 cores)
	Memory	256 GB
	Disk	7x 400 GB SSD
	Network	10 Gigabit Ethernet
Software		
OS	CentOS® Linux release 6.9	
Kernel	2.6.32-52.2.2.1.el6.x86_64	
Spark*	Spark™ master (2.0) / Spark™ master (2.0) with adaptive execution patch	
Hadoop™/HDFS™	hadoop-2.7.3	
JDK	1.8.0_40 (Oracle® Corporation)	

*Other names and brands may be claimed as the property of others.
For more complete information about performance and benchmarks results, visit www.intel.com/benchmarks.

 | 23



TPC-DS* 100TB Benchmark

Spark SQL v.s. Adaptive Execution

Query ID	Spark SQL Duration [s]	Adaptive Execution Duration [s]
q0	8.22	1.09
q81	1.09	1.00
q20	1.00	1.00
q51	1.79	1.00
q61	1.60	1.00
q60	1.00	1.00
q70	1.00	1.00
q57	1.00	1.00
q82	1.00	1.00
q96	1.00	1.00
q31	1.00	1.00
q19	1.00	1.00
q41	1.00	1.00
q24	1.00	1.00
q91	1.00	1.00

*Other names and brands may be claimed as the property of others.
For more complete information about performance and benchmarks results, visit www.intel.com/benchmarks.

 | 24



Auto Setting the Shuffle Partition Number

- Less scheduler overhead and task startup time.
- Less disk IO requests.
- Less data are written to disk because more data are aggregated.

Partition Number: 10976 (q30)

Stage	Task ID	Start Time	End Time	Duration	Shuffle Read	Shuffle Write	Disk I/O	Memory
Map	10976_0	2017-09-17 11:36:01	2017-09-17 11:36:01	0:00:00	0:00:00	0:00:00	0:00:00	6.1GB
Map	10976_1	2017-09-17 11:36:01	2017-09-17 11:36:01	0:00:00	0:00:00	0:00:00	0:00:00	6.1GB

Partition Number changed to 1084 and 1029 at runtime. (q30)

Stage	Task ID	Start Time	End Time	Duration	Shuffle Read	Shuffle Write	Disk I/O	Memory
Map	1084_0	2017-09-17 11:36:01	2017-09-17 11:36:01	0:00:00	0:00:00	0:00:00	0:00:00	6.1GB
Map	1084_1	2017-09-17 11:36:01	2017-09-17 11:36:01	0:00:00	0:00:00	0:00:00	0:00:00	6.1GB
Map	1029_0	2017-09-17 11:36:01	2017-09-17 11:36:01	0:00:00	0:00:00	0:00:00	0:00:00	11.7 GB
Map	1029_1	2017-09-17 11:36:01	2017-09-17 11:36:01	0:00:00	0:00:00	0:00:00	0:00:00	11.7 GB

*For more complete information about performance and benchmark results, visit www.intel.com/benchmarks


SPARK SUMMIT
powered by 



SortMergeJoin -> BroadcastJoin at Runtime

- Eliminate the data skew and straggler in SortMergeJoin
- Remote shuffle read -> local shuffle read.
- Random IO read -> Sequence IO read

SortMergeJoin (q8):

Stage	Task ID	Start Time	End Time	Duration	Shuffle Read	Shuffle Write	Disk I/O	Memory
Map	q8_0	2017-09-17 11:36:01	2017-09-17 11:36:01	0:00:00	0:00:00	0:00:00	0:00:00	6.1GB
Map	q8_1	2017-09-17 11:36:01	2017-09-17 11:36:01	0:00:00	0:00:00	0:00:00	0:00:00	6.1GB
Map	q8_2	2017-09-17 11:36:01	2017-09-17 11:36:01	0:00:00	0:00:00	0:00:00	0:00:00	6.1GB

BroadcastJoin (q8 Adaptive Execution):

Stage	Task ID	Start Time	End Time	Duration	Shuffle Read	Shuffle Write	Disk I/O	Memory
Map	q8_0	2017-09-17 11:36:01	2017-09-17 11:36:01	0:00:00	0:00:00	0:00:00	0:00:00	6.1GB
Map	q8_1	2017-09-17 11:36:01	2017-09-17 11:36:01	0:00:00	0:00:00	0:00:00	0:00:00	6.1GB
Map	q8_2	2017-09-17 11:36:01	2017-09-17 11:36:01	0:00:00	0:00:00	0:00:00	0:00:00	11.7 GB

*For more complete information about performance and benchmark results, visit www.intel.com/benchmarks


SPARK SUMMIT
powered by 

References:

<https://databricks.com/session/an-adaptive-execution-engine-for-apache-spark-sql>

<https://issues.apache.org/jira/browse/SPARK-9850>