Narrow Dependencies

“Transformations consisting of narrow are those for which each input partition will contribute to only one output partition”

wide dependency

“A wide dependency style transformation will have input partitions “contributing to many output partitions. You will often hear this referred to as a shuffle whereby Spark will exchange partitions across the cluster. ”

Transformation and Action:

“Transformations allow us to build up our logical transformation plan. To trigger the computation, we run an action. ”

- Actions to view data in the console
- Actions to collect data to native objects in the respective language
- Actions to write to output data sources

divisBy2.count()

To get the schema information, Spark reads in a little bit of the data and then attempts to parse the types in those rows according to the types available in Spark. You also have the option of strictly specifying a schema when you read in data

```
val flightData2015 = spark
  .read
  .option("inferSchema", "true")
  .option("header", "true")
  .csv("/data/flight-data/csv/2015-summary.csv")

flightData2015.take(3)
... Array([United States,Singapore,1],
[Moldova,United States,1])
```

```
spark.conf.set("spark.sql.shuffle.partitions", "5")
flightData2015.sort("count").take(2)

Array([United States,Romania,15], [United
States,Croatia...

flightData2015
  .groupBy("DEST_COUNTRY_NAME")
  .sum("count")
  .withColumnRenamed("sum(count)",
"destination_total")
  .sort(desc("destination_total"))
  .limit(5)
  .show()

flightData2015.createOrReplaceTempView("flight

val maxSql = spark.sql("""
SELECT DEST_COUNTRY_NAME, sum(count) as
destination_total
FROM flight_data_2015
GROUP BY DEST_COUNTRY_NAME
ORDER BY sum(count) DESC
LIMIT 5
""")
```

DataFrames Versus Datasets:

To say that DataFrames are untyped is aslightly inaccurate; they have types, but Spark maintains them completely and only checks whether those types line up to those specified in the schema at runtime. Datasets, on the other hand, check whether types conform to the specification at compile time. Datasets are only available to Java Virtual Machine (JVM)–based languages (Scala and Java) and we specify types with case classes or Java beans.

To Spark (in Scala), DataFrames are simply Datasets of Type Row. The “Row” type is Spark’s internal representation of its optimized in-memory format for computation. This format makes for highly specialized and efficient computation because rather than using JVM types, which can cause high

garbage-collection and object instantiation costs, Spark can operate on its own internal format without incurring any of those costs."
To Spark (in Python or R), there is no such thing as a Dataset: everything is a DataFrame and therefore we always operate on that optimized format.

A schema is a StructType made up of a number of fields, StructFields, that have a name, type, a Boolean flag which specifies whether that column can contain missing or null values, and, finally, users can optionally specify associated metadata with that column.

Excerpt From: Bill Chambers. "Spark." iBooks.

```
StructType schema =
DataTypes.createStructType(fields);

Dataset<Row> ds = spark.createDataFrame(javaRowRDD,
schema);

List<StructField> fields = new ArrayList<>();

StructField field =
DataTypes.createStructField(metadata.getColumnIndexMa
p().get(Integer.toString(index)).getName(),
DataTypes.StringType, true);
```

<u>Structured API Code Execution</u>:

To execute code, we must write code. This code is then submitted to Spark either through the console or via a submitted job. This code then passes through the Catalyst Optimizer, which decides how the code should be executed and lays out a plan for doing so before, finally, the code is run and the result is returned to the user.

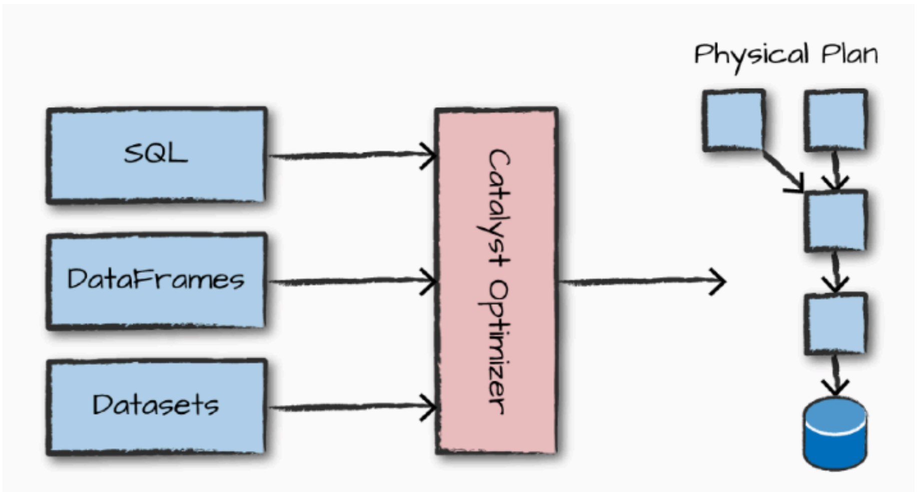
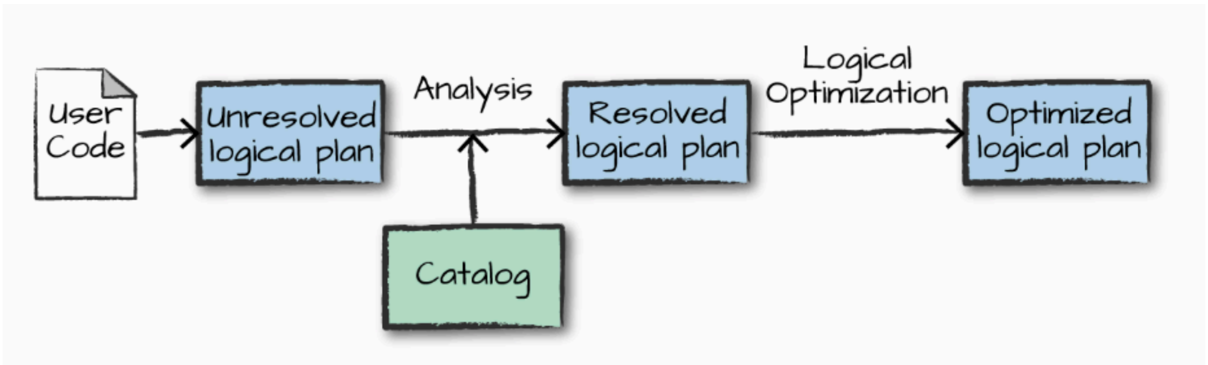


Figure 4-1. The Catalyst Optimizer

Phase 1 – (convert it into a logical plan)

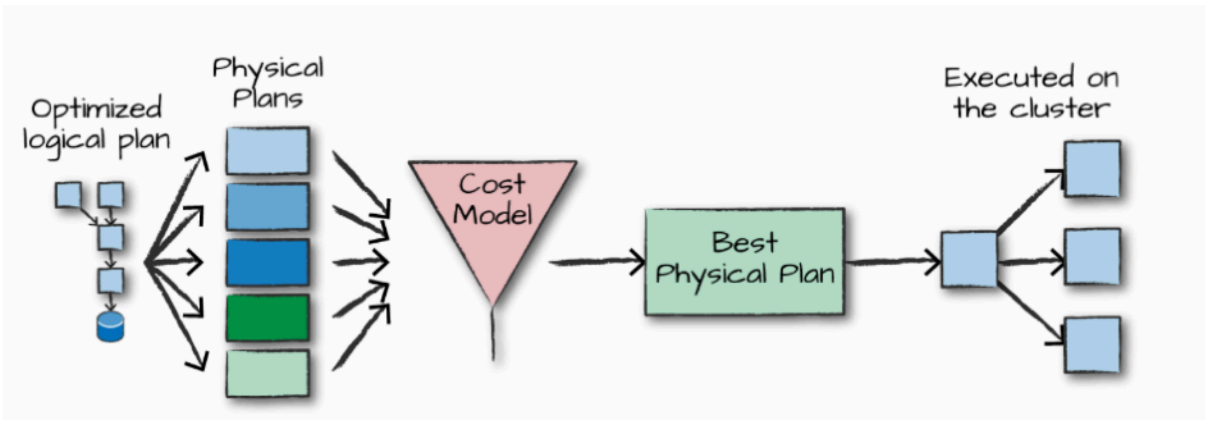
This logical plan only represents a set of **abstract transformations** that do not refer to executors or drivers.

The analyser might reject the unresolved logical plan if the required table or column name does not exist in the catalog. **If the analyzer can resolve it, the result is passed through the Catalyst Optimizer, a collection of rules that attempt to optimize the logical plan by pushing down predicates or selections.** Packages can extend the Catalyst to include their own rules for domain-specific optimizations.



Phase 2 – (Physical Planning)

Physical planning results in a series of RDDs and transformations. This result is why you might have heard Spark referred to as a compiler—it takes queries in DataFrames, Datasets, and SQL and compiles them into RDD transformations for you.



An example of the cost comparison might be choosing how to perform a given join by looking at the physical attributes of a given table (how big the table is or how big its partitions are

Repartition:

Repartition will incur a full shuffle of the data. **This means that you should typically only repartition when the future number of partitions is greater than your current number of partitions.**

df.repartition(5, col("DEST_COUNTRY_NAME"))

Coalesce:

Coalesce, on the other hand, will not incur a full shuffle and will try to combine partitions.

Column:

If you need to refer to a specific DataFrame’s column, you can use the col method on the specific DataFrame.

```
df.col("count")
```

Joins

```
val joinExpression = person.col("graduate_program") ===
graduateProgram.col("id")
var joinType = "inner"

person.join(graduateProgram, joinExpression, joinType).show()
```

Big table–to–big table

When you join a big table to another big table, you end up with a shuffle join.
“node-to-node communication strategy”

“In a shuffle join, every node talks to every other node and they share data according to which node has a certain key or set of keys (on which you are joining). These joins are expensive because the network can become congested with traffic, especially if your data is not partitioned well.”

Big table–to–small table

```
SELECT /*+ MAPJOIN(graduateProgram) */ * FROM person JOIN
graduateProgram ON person.graduate_program = graduateProgram.id”
```

Read

```
spark.read.format("csv")
  .option("header", "true")
  .option("mode", "FAILFAST")
  .option("inferSchema", "true")
  .option("path", "path/to/file(s)")
  .schema(someSchema)
  .load()
```

Read modes:

Reading data from an external source naturally entails encountering malformed data, especially when working with only semi-structured data sources. Read modes specify what will happen when Spark does come across malformed records.

Permissive - Sets all fields to null when it encounters a corrupted record and places all corrupted records in a string column called _corrupt_record

dropMalformed -Drops the row that contains malformed records.

failFast -Fails immediately upon encountering malformed records"

<u>Writer</u>:

```
dataframe.write.format("csv")
  .option("mode", "OVERWRITE")
  .option("dateFormat", "yyyy-MM-dd")
  .option("path", "path/to/file(s)")
  .save()
```

Append , overwrite and errorIfExists.

errorIfExists
"Throws an error and fails the write if data or files already exist at the specified location"

<u>countByKey</u>

You can count the number of elements for each key, collecting the results to a local Map.

<u>Broadcast Variables</u>

Broadcast variables are a way you can share an immutable value efficiently around the cluster without encapsulating that variable in a function closure. when you use a variable in a closure, it must be deserialized on the worker nodes many times (one per task). Moreover, if you use the same variable in multiple Spark actions and jobs, it will be re-sent to the workers with every job instead of once.

This is where broadcast variables come in. **Broadcast variables are shared, immutable variables that are cached on every machine in the cluster instead of serialized with every single task.**

```
val supplementalData = Map("Spark" -> 1000,
"Definitive" -> 200,
                           "Big" -> -300, "Simple" ->
100)”

val suppBroadcast =
spark.sparkContext.broadcast(supplementalData)
suppBroadcast.value
```

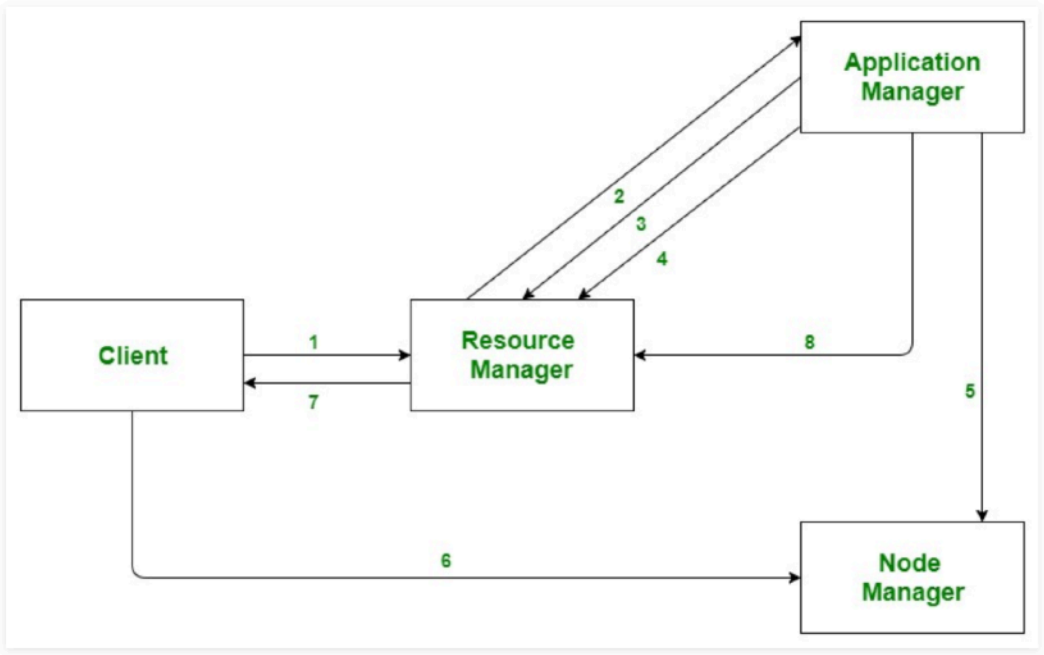
<u>**Accumulators**</u>

Accumulators Spark’s second type of shared variable, are a way of updating a value inside of a variety of transformations and propagating that value to the driver node in an efficient and fault-tolerant way.

Accumulators provide a mutable variable that a Spark cluster can safely update on a per-row basis.

accumulator updates performed inside actions only, Spark guarantees that each task’s update to the accumulator will be applied only once, meaning that restarted tasks will not update the value.

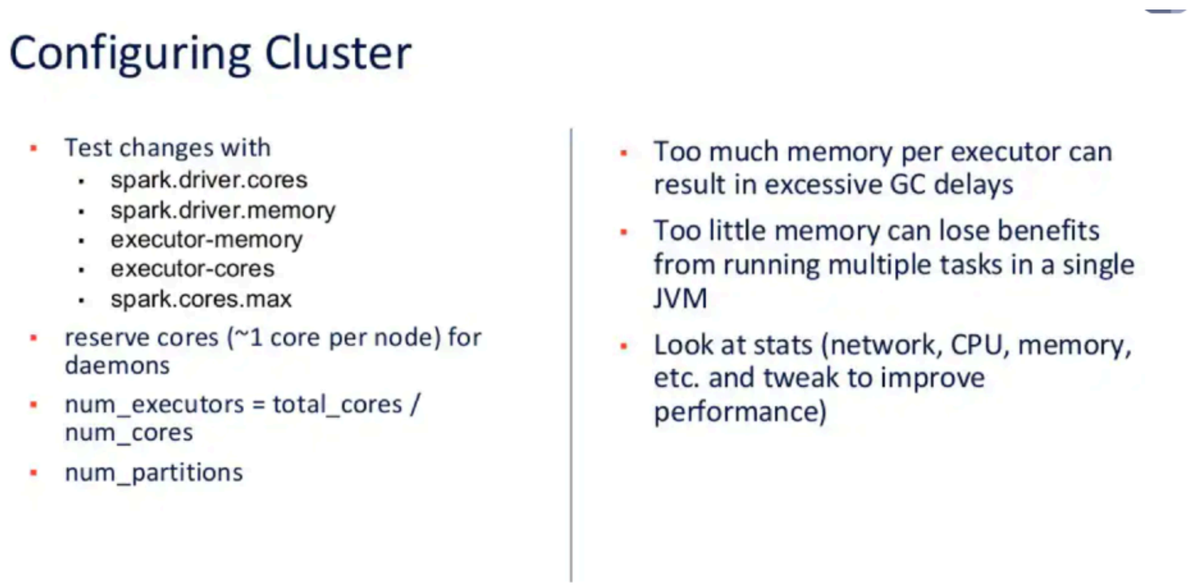
```
val accUnnamed = new LongAccumulator
val acc = spark.sparkContext.register(accUnnamed)
```



1. Client submits an application
2. The Resource Manager allocates a container to start the Application Manager
3. The Application Manager registers itself with the Resource Manager
4. The Application Manager negotiates containers from the Resource Manager
5. The Application Manager notifies the Node Manager to launch containers
6. Application code is executed in the container
7. Client contacts Resource Manager/Application Manager to monitor application’s status
8. Once the processing is complete, the Application Manager un-registers with the Resource Manager

Optimization

1) Cluster

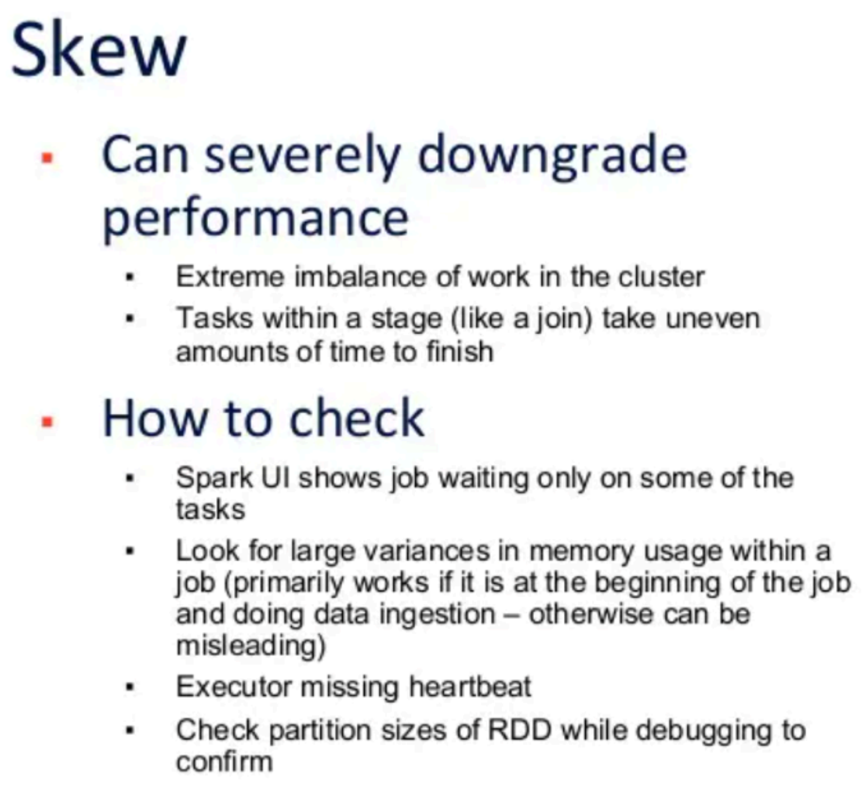


Configuring Cluster

- Test changes with
  - spark.driver.cores
  - spark.driver.memory
  - executor-memory
  - executor-cores
  - spark.cores.max
- reserve cores (~1 core per node) for daemons
- num_executors = total_cores / num_cores
- num_partitions

- Too much memory per executor can result in excessive GC delays
- Too little memory can lose benefits from running multiple tasks in a single JVM
- Look at stats (network, CPU, memory, etc. and tweak to improve performance)

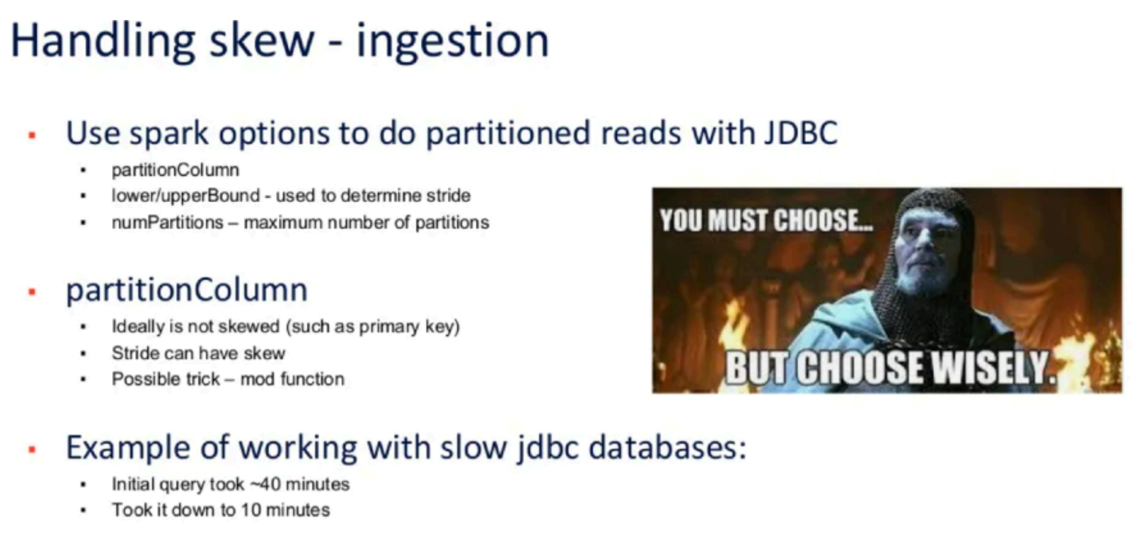
2) Skew



Skew

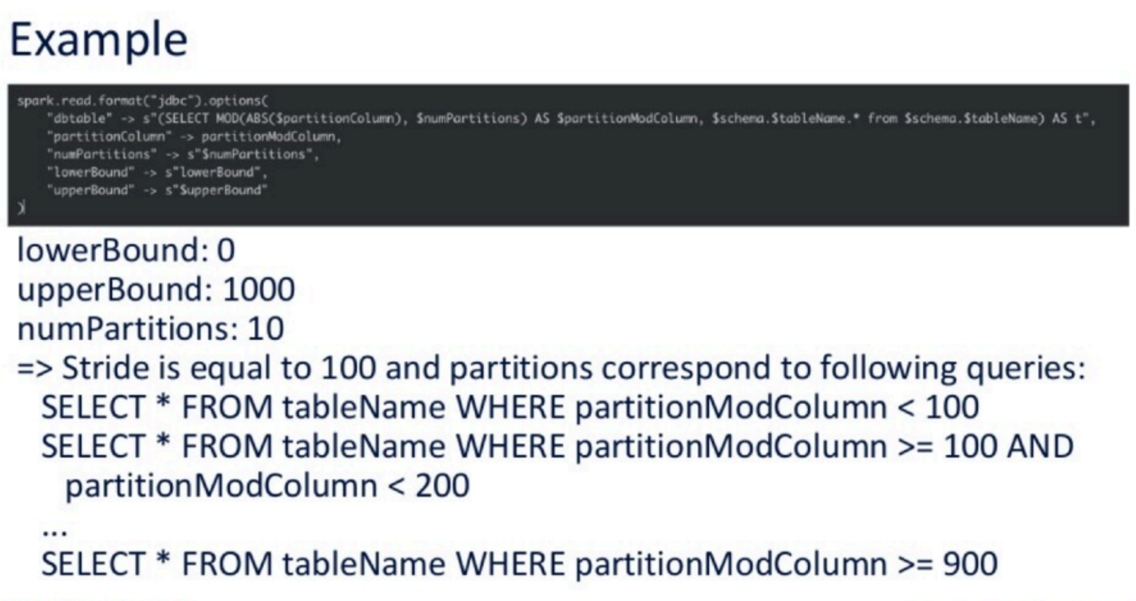
- Can severely downgrade performance
  - Extreme imbalance of work in the cluster
  - Tasks within a stage (like a join) take uneven amounts of time to finish
- How to check
  - Spark UI shows job waiting only on some of the tasks
  - Look for large variances in memory usage within a job (primarily works if it is at the beginning of the job and doing data ingestion – otherwise can be misleading)
  - Executor missing heartbeat
  - Check partition sizes of RDD while debugging to confirm

Database:



Handling skew - ingestion

- Use spark options to do partitioned reads with JDBC
  - partitionColumn
  - lower/upperBound - used to determine stride
  - numPartitions – maximum number of partitions
- partitionColumn
  - Ideally is not skewed (such as primary key)
  - Stride can have skew
  - Possible trick – mod function
- Example of working with slow jdbc databases:
  - Initial query took ~40 minutes
  - Took it down to 10 minutes

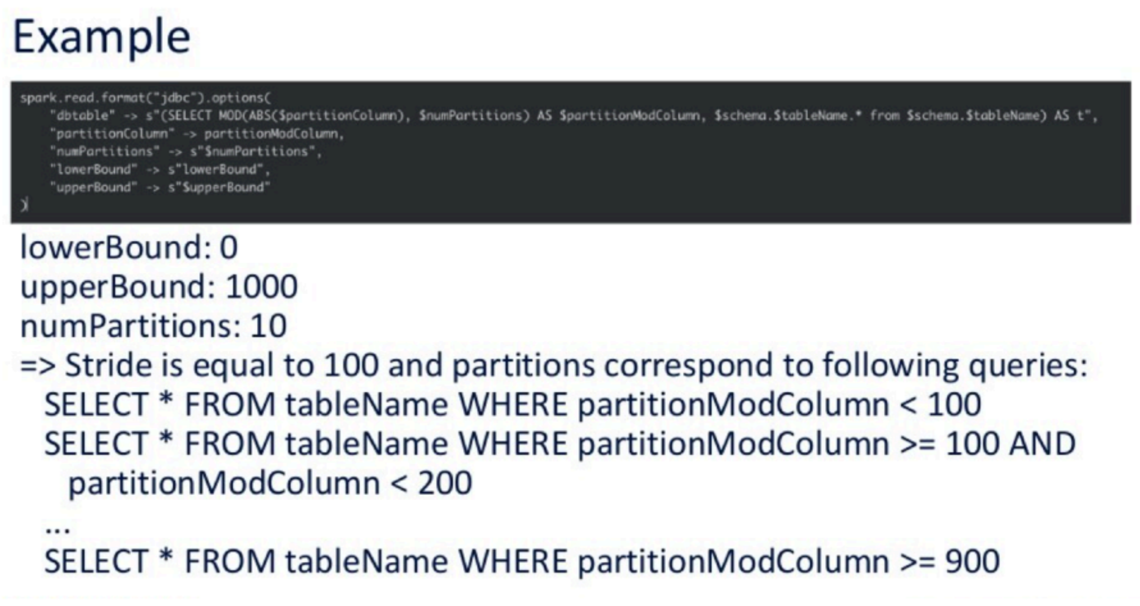


Example

```
spark.read.format("jdbc").options(
    "dbtable" -> s"(SELECT MOD(ABS($partitionColumn), $numPartitions) AS $partitionModColumn, $schema.$tableName.* from $schema.$tableName) AS t",
    "partitionColumn" -> partitionModColumn,
    "numPartitions" -> s"$numPartitions",
    "lowerBound" -> s"lowerBound",
    "upperBound" -> s"$upperBound"
)
```

lowerBound: 0
upperBound: 1000
numPartitions: 10
=> Stride is equal to 100 and partitions correspond to following queries:
  SELECT * FROM tableName WHERE partitionModColumn < 100
  SELECT * FROM tableName WHERE partitionModColumn >= 100 AND
    partitionModColumn < 200
  ...
  SELECT * FROM tableName WHERE partitionModColumn >= 900

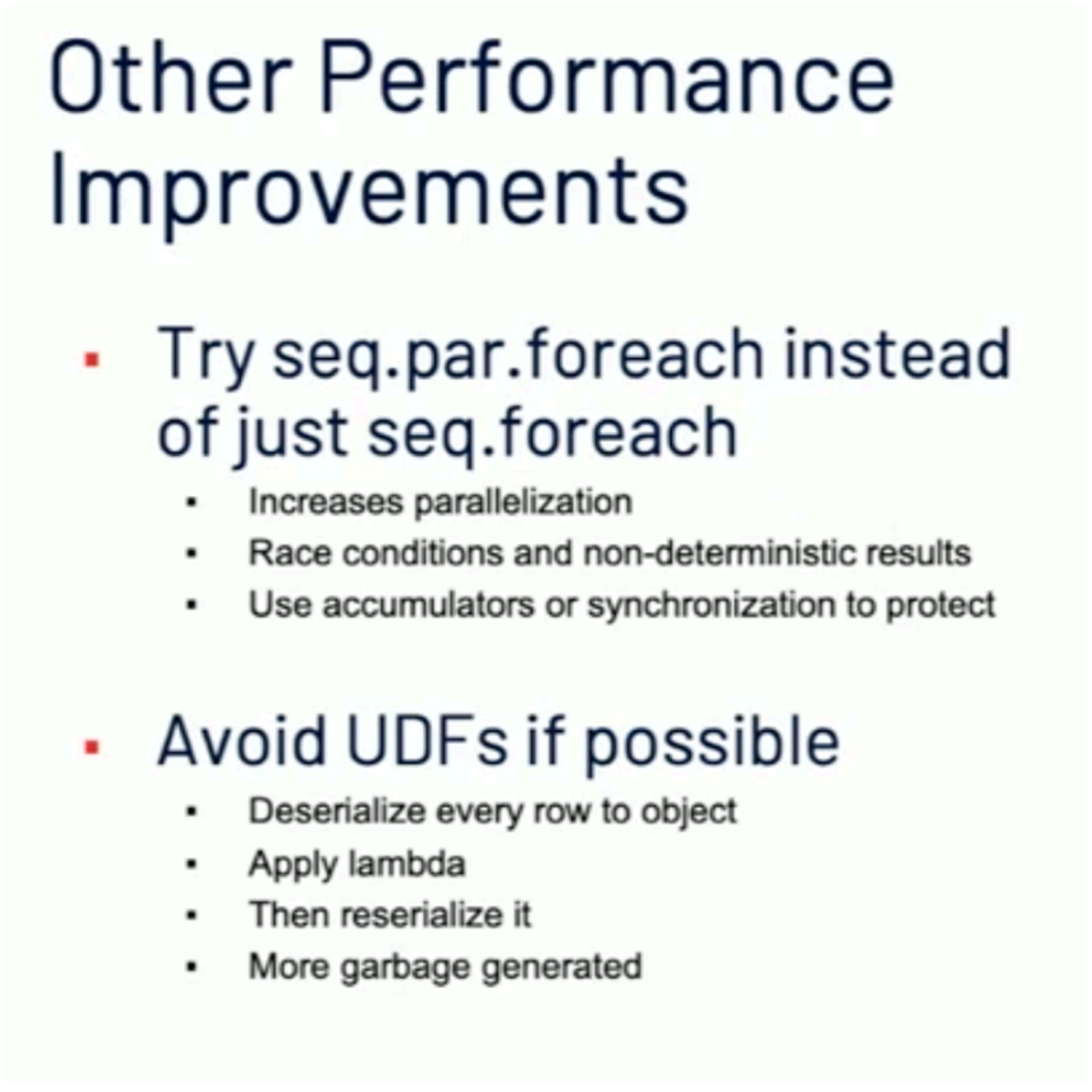
Cache/Persist



Example

```
spark.read.format("jdbc").options(
    "dbtable" -> s"(SELECT MOD(ABS($partitionColumn), $numPartitions) AS $partitionModColumn, $schema.$tableName.* from $schema.$tableName) AS t",
    "partitionColumn" -> partitionModColumn,
    "numPartitions" -> s"$numPartitions",
    "lowerBound" -> s"lowerBound",
    "upperBound" -> s"$upperBound"
)
```

lowerBound: 0
upperBound: 1000
numPartitions: 10
=> Stride is equal to 100 and partitions correspond to following queries:
  SELECT * FROM tableName WHERE partitionModColumn < 100
  SELECT * FROM tableName WHERE partitionModColumn >= 100 AND
    partitionModColumn < 200
  ...
  SELECT * FROM tableName WHERE partitionModColumn >= 900

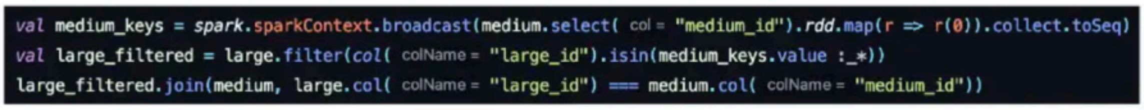


Other Performance Improvements

- Try seq.par.foreach instead of just seq.foreach
  - Increases parallelization
  - Race conditions and non-deterministic results
  - Use accumulators or synchronization to protect
- Avoid UDFs if possible
  - Deserialize every row to object
  - Apply lambda
  - Then reserialize it
  - More garbage generated

Join :

Filter Trick

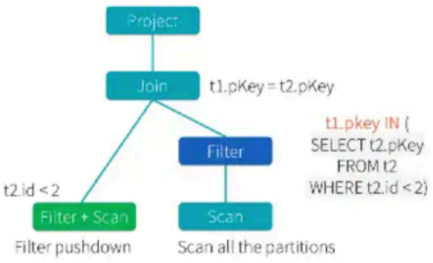


```
val medium_keys = spark.sparkContext.broadcast(medium.select( col = "medium_id").rdd.map(r => r(0)).collect.toSeq)
val large_filtered = large.filter(col( colName = "large_id").isin(medium_keys.value :_*))
large_filtered.join(medium, large.col( colName = "large_id") === medium.col( colName = "medium_id"))
```

Included in Spark 3.0

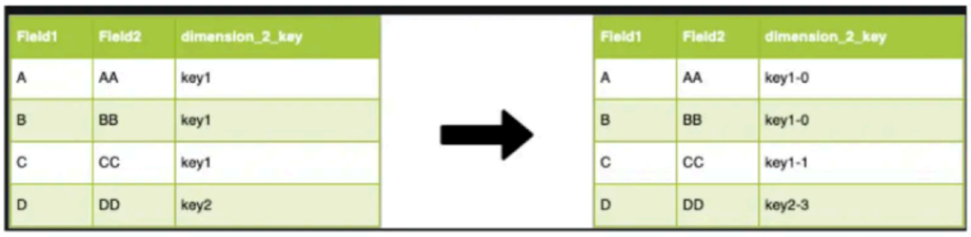
Dynamic Partition Pruning

```
SELECT t1.id, t2.pKey
FROM   t1
  JOIN t2
    ON t1.pKey = t2.pKey
    AND t2.id < 2
```



Salting :

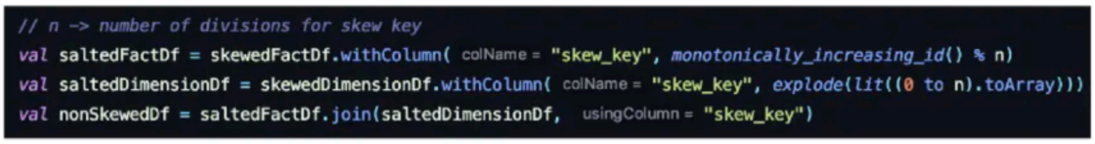
Salting – Reduce Skew



Field1	Field2	dimension_2_key
A	AA	key1
B	BB	key1
C	CC	key1
D	DD	key2

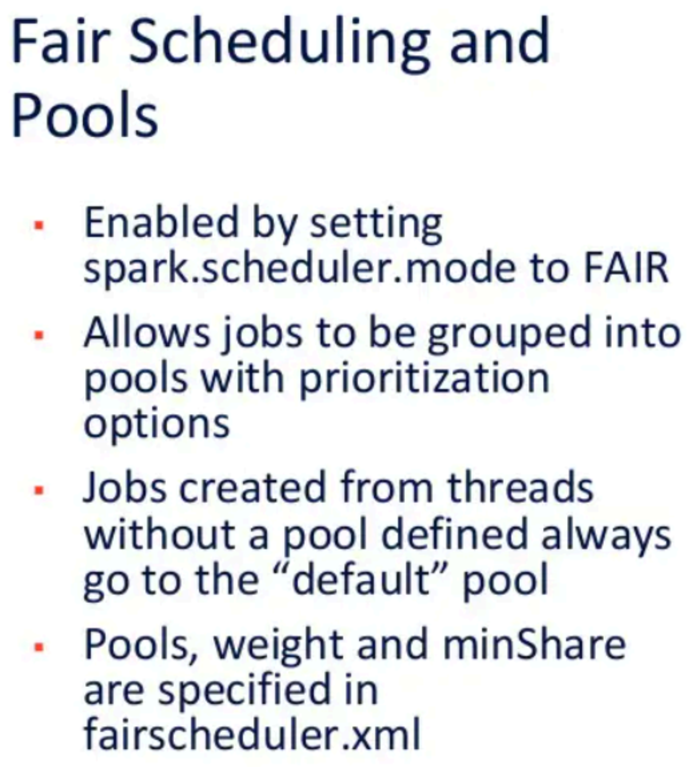
→

Field1	Field2	dimension_2_key
A	AA	key1-0
B	BB	key1-0
C	CC	key1-1
D	DD	key2-3



```
// n -> number of divisions for skew key
val saltedFactDf = skewedFactDf.withColumn( colName = "skew_key", monotonically_increasing_id() % n)
val saltedDimensionDf = skewedDimensionDf.withColumn( colName = "skew_key", explode(lit((0 to n).toArray)))
val nonSkewedDf = saltedFactDf.join(saltedDimensionDf,  usingColumn = "skew_key")
```

Fair Scheduling



Fair Scheduling and Pools

- Enabled by setting spark.scheduler.mode to FAIR
- Allows jobs to be grouped into pools with prioritization options
- Jobs created from threads without a pool defined always go to the "default" pool
- Pools, weight and minShare are specified in fairscheduler.xml

Structured Streaming

Structured Streaming is a scalable and fault-tolerant stream processing engine built on the Spark SQL engine. You can express your streaming computation the same way you would express a batch computation on static data.

Structured Streaming provides fast, scalable, fault-tolerant, end-to-end exactly-once stream processing without the user having to reason about streaming.

Delivering end-to-end exactly-once semantics was one of key goals behind the design of Structured Streaming.

```
Every streaming source is assumed to have offsets
(similar to Kafka offsets, or Kinesis sequence
numbers) to track the read position in the stream.
The engine uses checkpointing and write ahead logs to
record the offset range of the data being processed
in each trigger.
```

The streaming sinks are designed to be idempotent for handling reprocessing. Together, using replayable sources and idempotent sinks, Structured Streaming can ensure **end-to-end exactly-once semantics** under any failure.

Streaming DataFrames can be created through the `DataStreamReader` interface

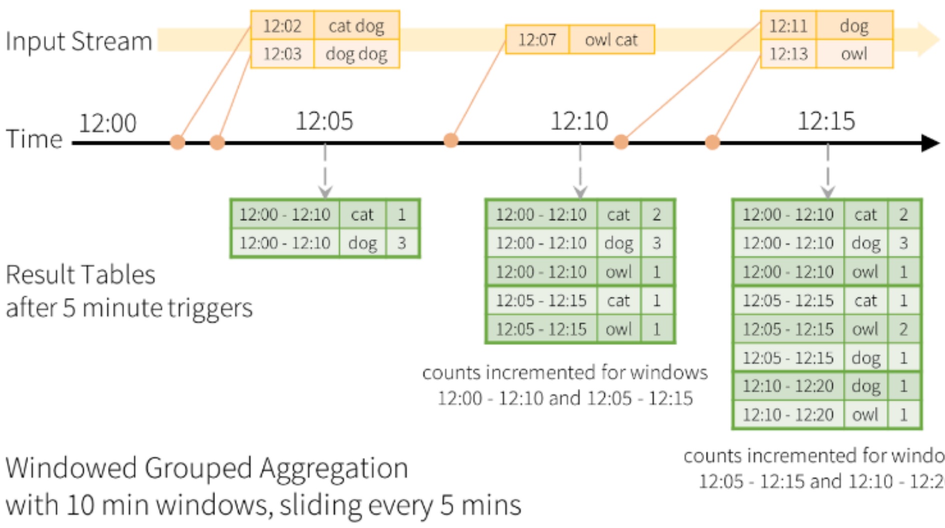
```
Dataset[Row] socketDF = spark
  .readStream()
  .format("socket")
  .option("host", "localhost")
  .option("port", 9999)
  .load();

socketDF.isStreaming();    // Returns True for DataFrames that
have streaming sources

socketDF.printSchema();

// Read all the csv files written atomically in a directory
StructType userSchema = new StructType().add("name",
"string").add("age", "integer");
Dataset[Row] csvDF = spark
  .readStream()
  .option("sep", ";")
  .schema(userSchema)      // Specify schema of the csv files
  .csv("/path/to/directory");    // Equivalent to
format("csv").load("/path/to/directory")
```

In a grouped aggregation, aggregate values (e.g. counts) are maintained for each unique value in the user-specified grouping column. In case of window-based aggregations, aggregate values are maintained for each window the event-time of a row falls into.



```
Dataset<Row> windowedCounts = words.groupBy(
  functions.window(words.col("timestamp"), "10 minutes", "5
minutes"),
  words.col("word")
).count();
```

watermarking, which let's the engine automatically track the current event time in the data and and attempt to clean up old state accordingly. You can define the watermark of a query by specifying the event time column and the threshold on how late the data is expected be in terms of event time.

```
Dataset<Row> windowedCounts = words
    .withWatermark("timestamp", "10 minutes")
    .groupBy(
        functions.window(words.col("timestamp"), "10 minutes", 
"5 minutes"),
        words.col("word"))
    .count();
```

- `withWatermark` must be called on the same column as the timestamp column used in the aggregate. For example, `df.withWatermark("time", "1 min").groupBy("time2").count()` is invalid in Append output mode, as watermark is defined on a different column as the aggregation column.

- `withWatermark` must be called before the aggregation for the watermark details to be used. For example, `df.groupBy("time").count().withWatermark("time", "1 min")` is invalid in Append output mode.

Output mode must be Append. Complete mode requires all aggregate data to be preserved, and hence cannot use watermarking to drop intermediate state

```
sorting is not supported on the input streaming
Dataset, as it requires keeping track of all the data
received in the stream.
```

`show()` - Instead use the console sink

- Multiple streaming aggregations (i.e. a chain of aggregations on a streaming DF) are not yet supported on streaming Datasets.

- Limit and take first N rows are not supported on streaming Datasets.

- Distinct operations on streaming Datasets are not supported.

- Sorting operations are supported on streaming Datasets only after an aggregation and in Complete Output Mode.

- Outer joins between a streaming and a static Datasets are conditionally supported.
  - Full outer join with a streaming Dataset is not supported

  - Left outer join with a streaming Dataset on the right is not supported

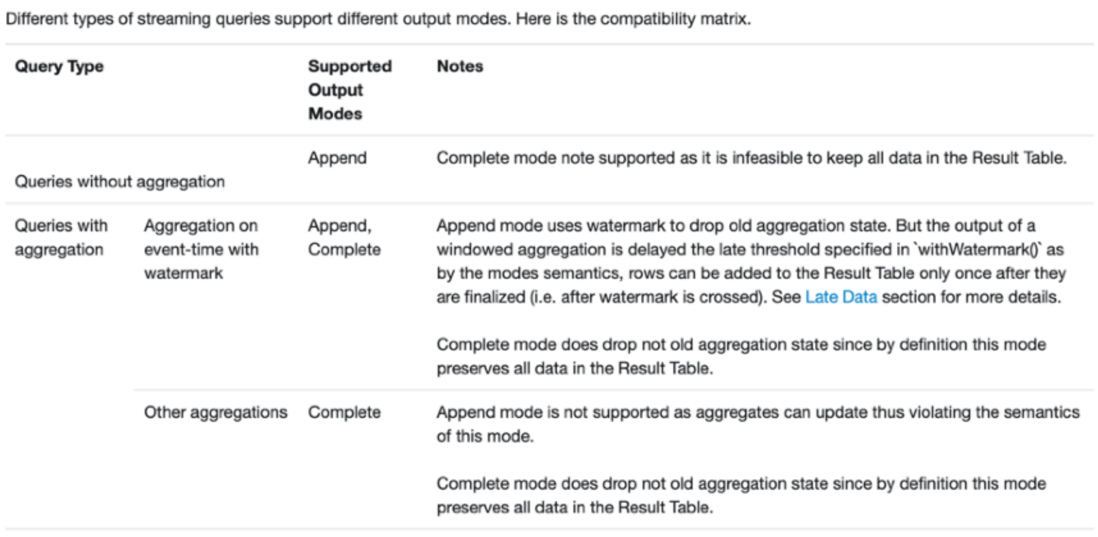
  - Right outer join with a streaming Dataset on the left is not supported

- Any kind of joins between two streaming Datasets are not yet supported.

- **Append mode (default)** - This is the default mode, where only the new rows added to the Result Table since the last trigger will be outputted to the sink. This is supported for only those queries where rows added to the Result Table is never going to change. Hence, this mode guarantees that each row will be output only once (assuming fault-tolerant sink). For example, queries with only `select`, `where`, `map`, `flatMap`, `filter`, `join`, etc. will support Append mode.

- **Complete mode** - The whole Result Table will be outputted to the sink after every trigger. This is supported for aggregation queries.

- **Update mode** - (*not available in Spark 2.1*) Only the rows in the Result Table that were updated since the last trigger will be outputted to the sink. More information to be added in future releases.



Different types of streaming queries support different output modes. Here is the compatibility matrix.

Query Type		**Supported Output Modes**	**Notes**
Queries without aggregation		Append	Complete mode note supported as it is infeasible to keep all data in the Result Table.
Queries with aggregation	Aggregation on event-time with watermark	Append, Complete	Append mode uses watermark to drop old aggregation state. But the output of a windowed aggregation is delayed the late threshold specified in `withWatermark()` as by the modes semantics, rows can be added to the Result Table only once after they are finalized (i.e. after watermark is crossed). See Late Data section for more details. Complete mode does drop not old aggregation state since by definition this mode preserves all data in the Result Table.
	Other aggregations	Complete	Append mode is not supported as aggregates can update thus violating the semantics of this mode. Complete mode does drop not old aggregation state since by definition this mode preserves all data in the Result Table.

```
query.awaitTermination();   // block until query is terminated, with stop() or with error
```

to prevent the process from exiting while the query is active.

we want to count words within 10 minute windows, updating every 5 minutes.

```
words.groupBy(
  functions.window(words.col("timestamp"), "10 minutes", "5 
minutes"),
  words.col("word")
).count();
```

grouped aggregation, aggregate values

Watermarking

```
Dataset<Row> windowedCounts = words
    .withWatermark("timestamp", "10 minutes")
    .groupBy(
        window(col("timestamp"), "10 minutes", "5 minutes"),
        col("word"))
    .count();
```

```
Num/10^num.length-1
```