Deep Dive:
Query Execution of Spark SQL

Maryann Xue, Xingbo Jiang, Kris Mok

Apr. 2019

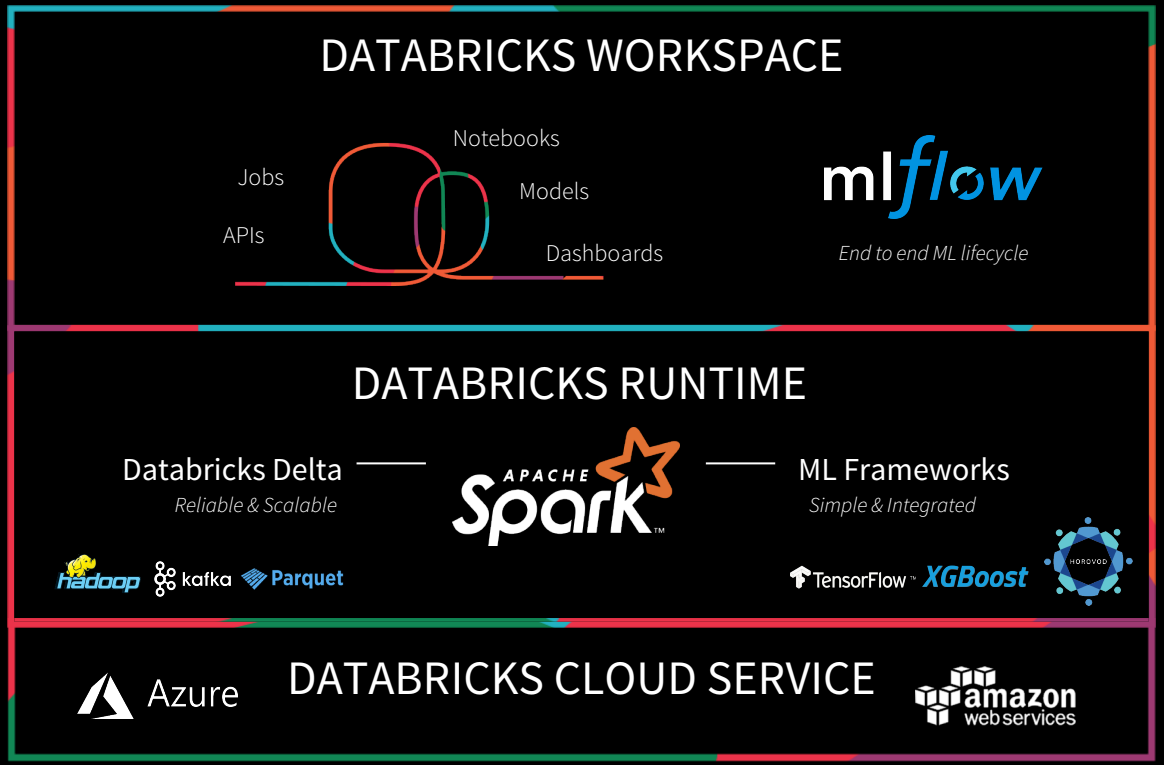
databricks

About Us

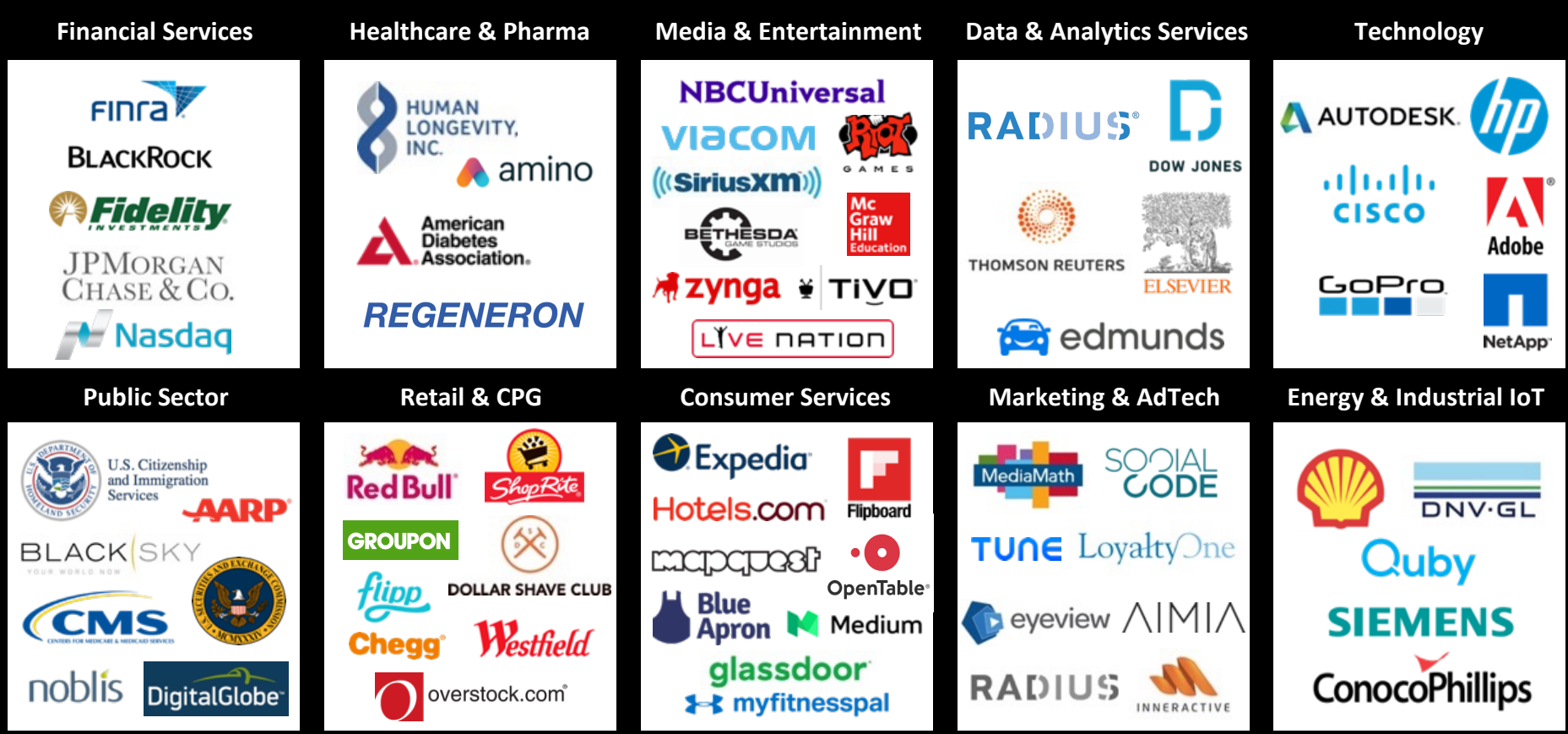
Software Engineers 

- Maryann Xue
  PMC of Apache Calcite & Apache Phoenix @maryannxue
- Xingbo Jiang
  Apache Spark Committer @jiangxb1987
- Kris Mok
  OpenJDK Committer @rednaxelafx

Databricks Unified Analytics Platform

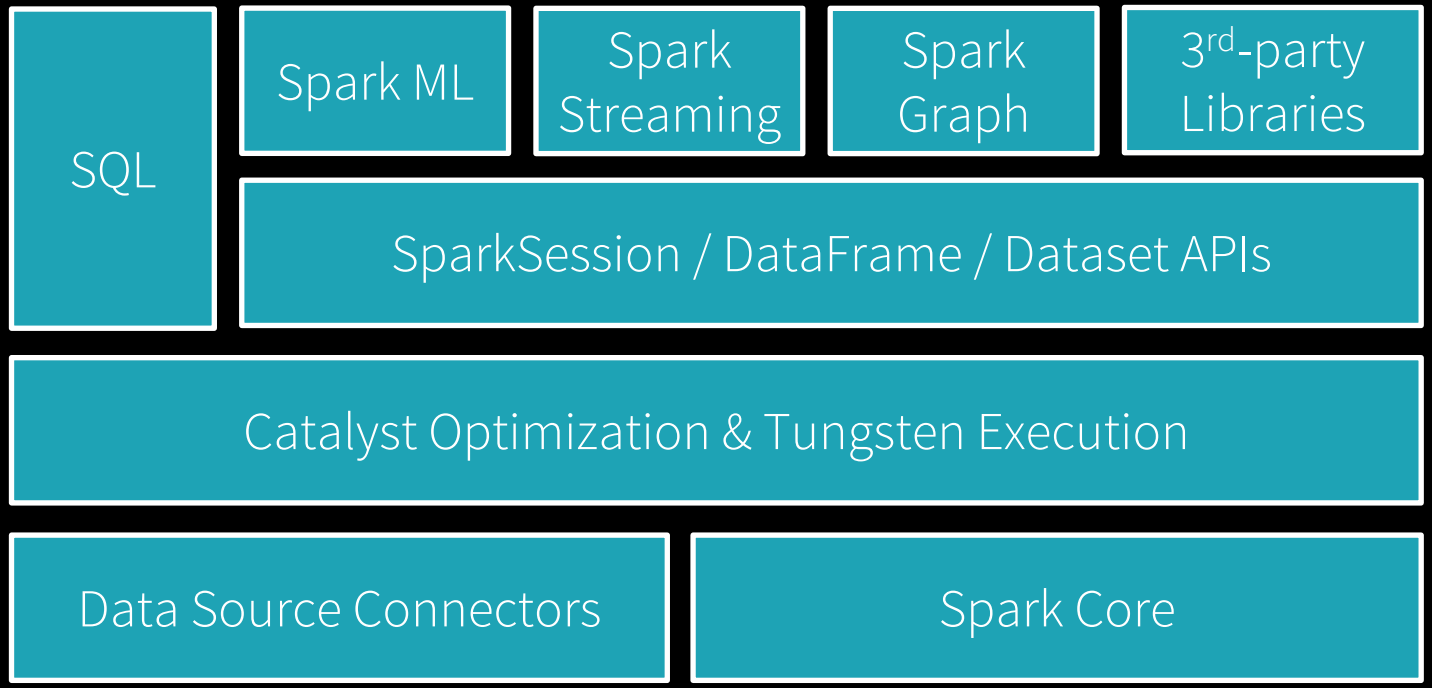


Databricks Customers Across Industries

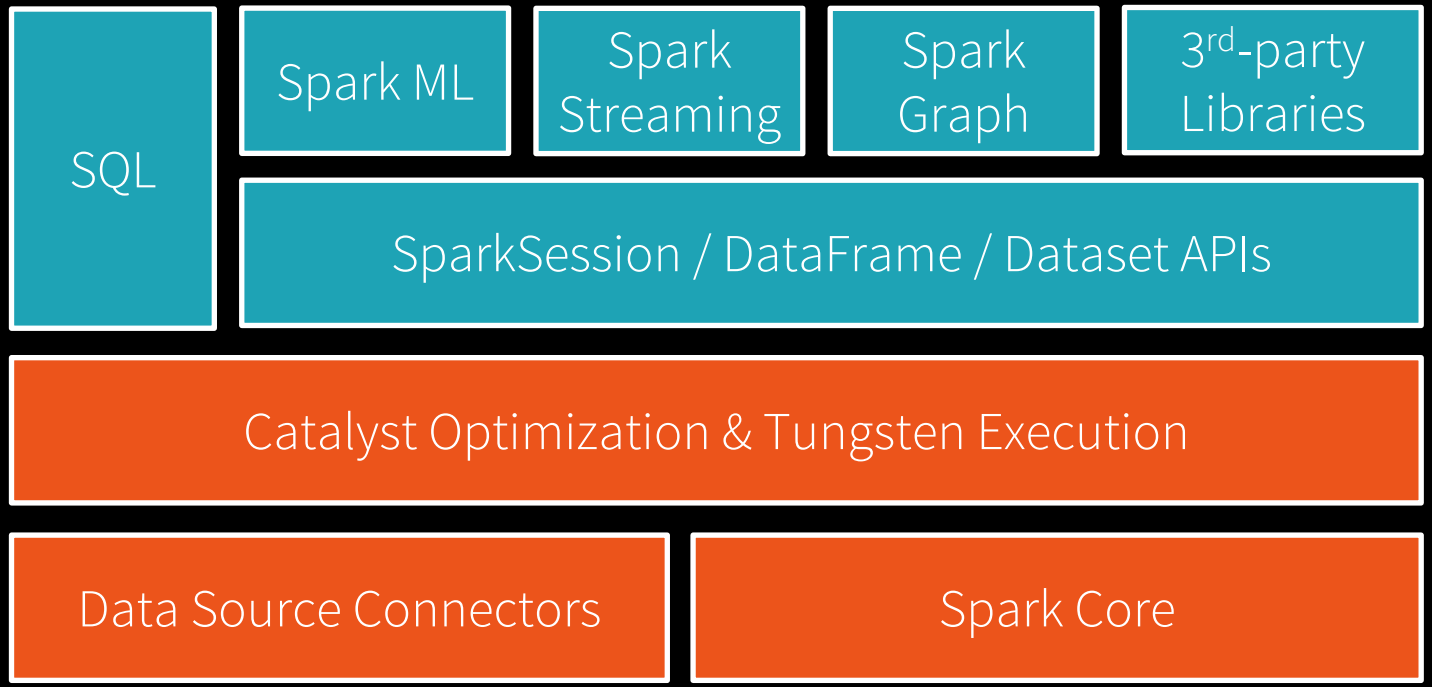


databricks

Apache Spark 3.x

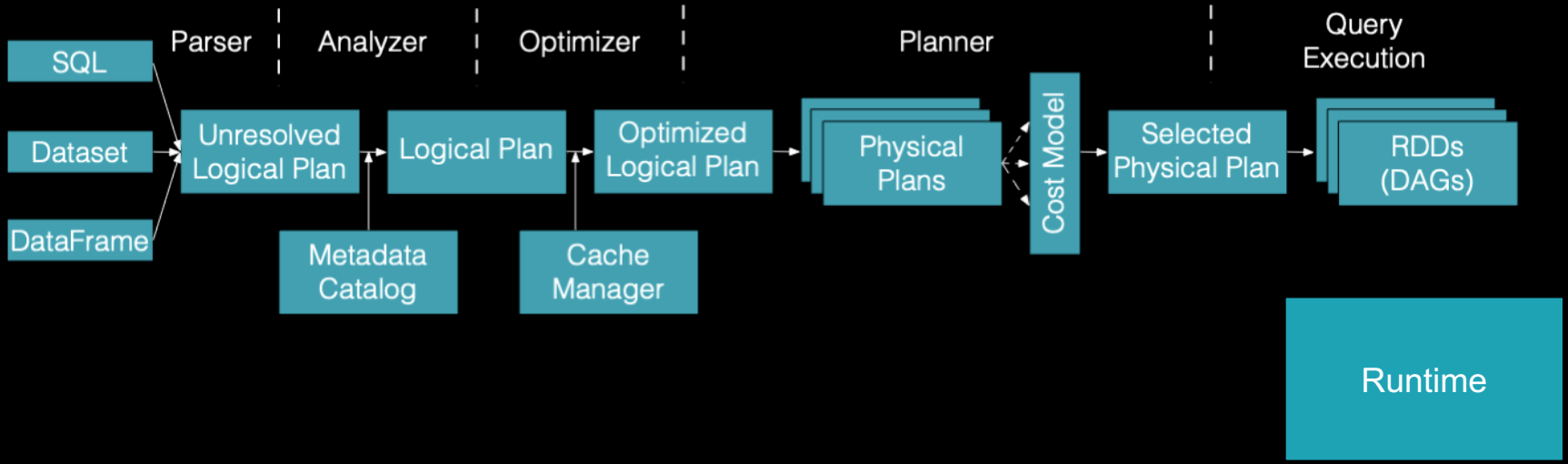


Apache Spark 3.x



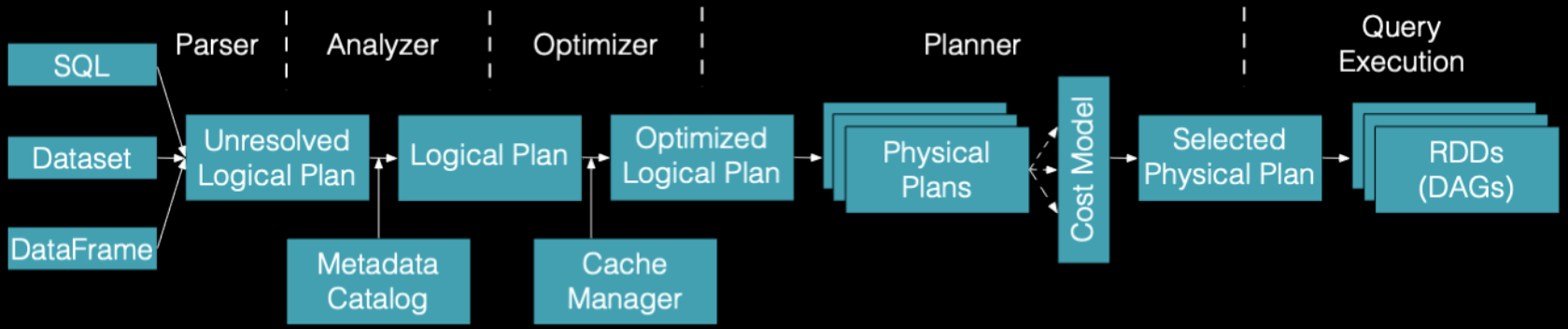
Spark SQL Engine

Analysis -> Logical Optimization -> Physical Planning -> Code Generation -> Execution



Spark SQL Engine - Front End

Analysis -> Logical Optimization -> Physical Planning -> Code Generation -> Execution

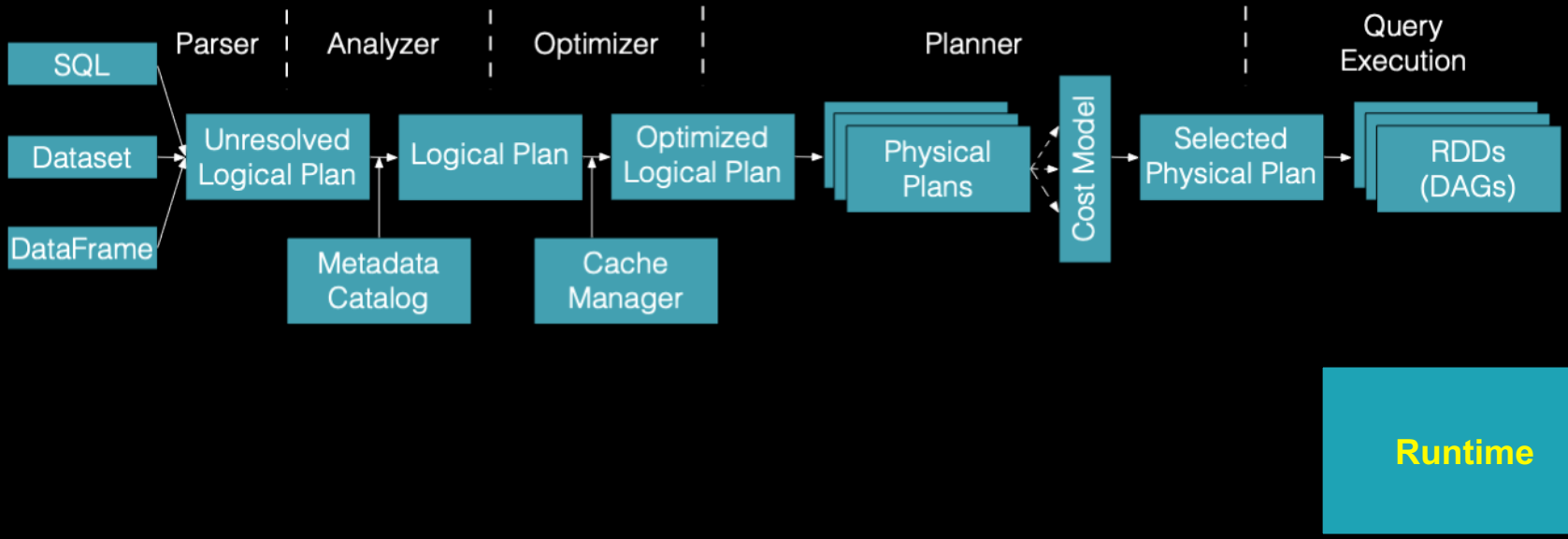


Reference: A Deep Dive into Spark SQL's Catalyst Optimizer,
Yin Huai, Spark Summit 2017

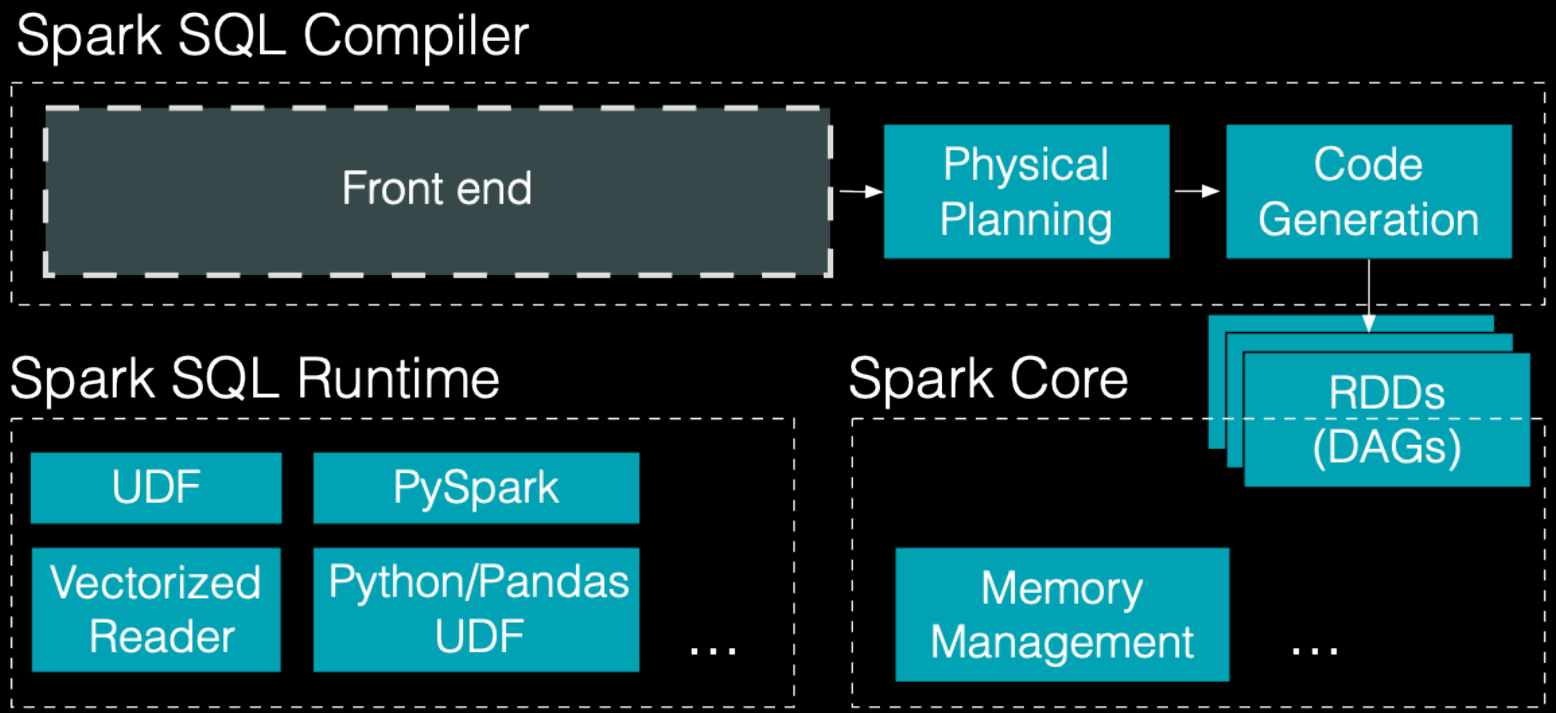
Runtime

Spark SQL Engine - Back End

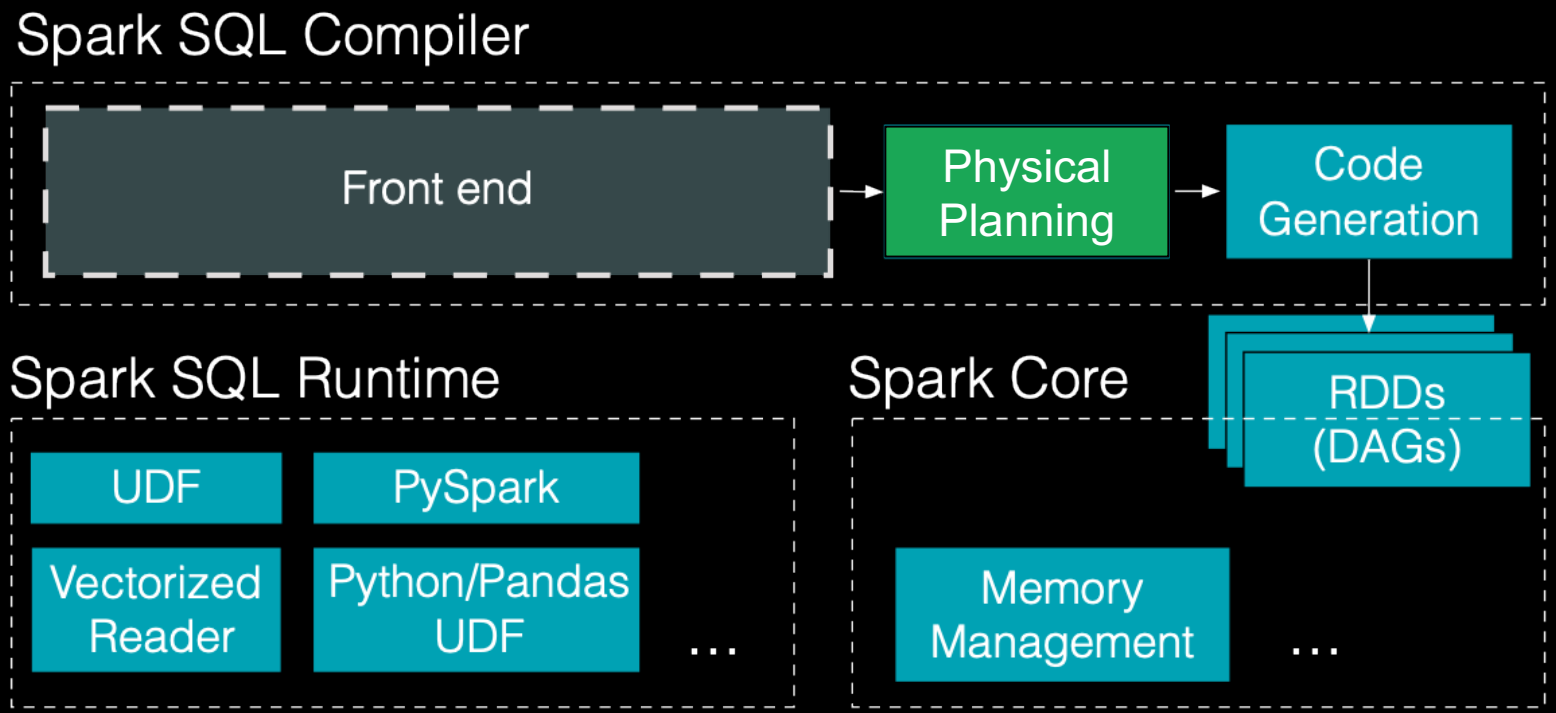
Analysis -> Logical Optimization -> Physical Planning -> Code Generation -> Execution



Agenda


Spark SQL Compiler
Front end
Physical Planning
Code Generation
RDDs (DAGs)
Spark SQL Runtime
UDF
PySpark
Vectorized Reader
Python/Pandas UDF
…
Spark Core
Memory Management
…

Agenda


Spark SQL Compiler
Front end
Physical Planning
Code Generation
RDDs (DAGs)
Spark SQL Runtime
UDF
PySpark
Vectorized Reader
Python/Pandas UDF
…
Spark Core
Memory Management
…

Physical Planning

- Transform logical operators into physical operators

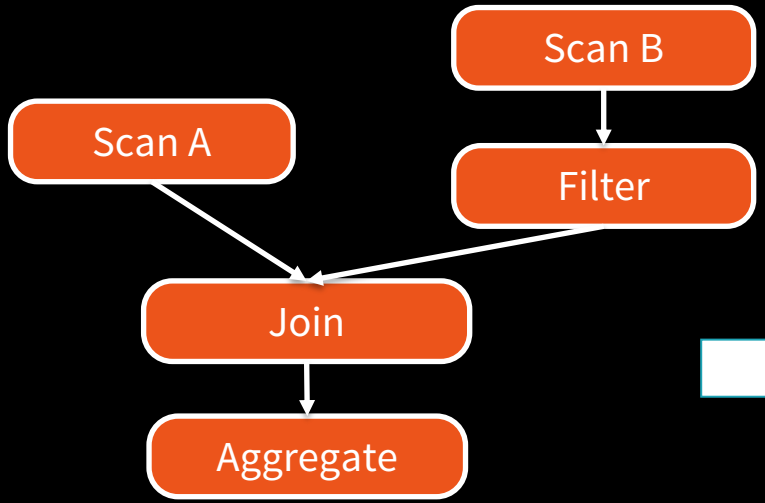
- Choose between different physical alternatives
  - e.g., broadcast-hash-join vs. sort-merge-join

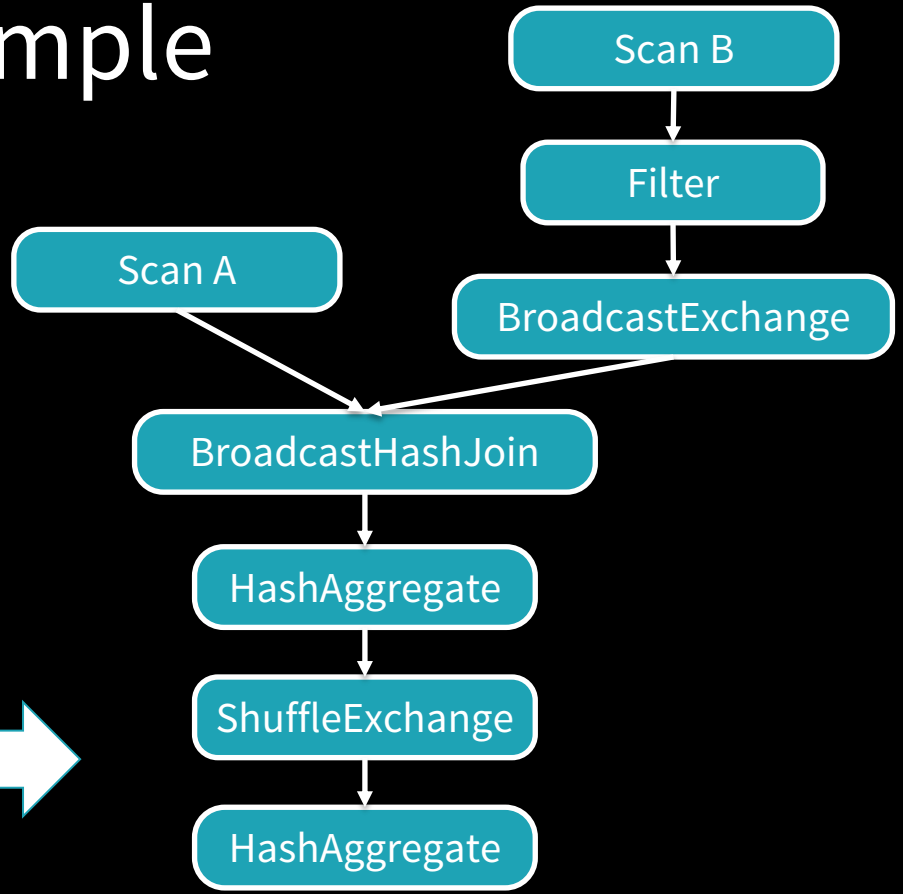
- Includes physical traits of the execution engine
  - e.g., partitioning & ordering.

- Some ops may be mapped into multiple physical nodes
  - e.g., partial agg —> shuffle —> final agg

A Physical Plan Example

```
SELECT a1, sum(b1)FROM A
JOIN B ON A.key = B.key
WHERE b1 < 1000 GROUP BY a1
```





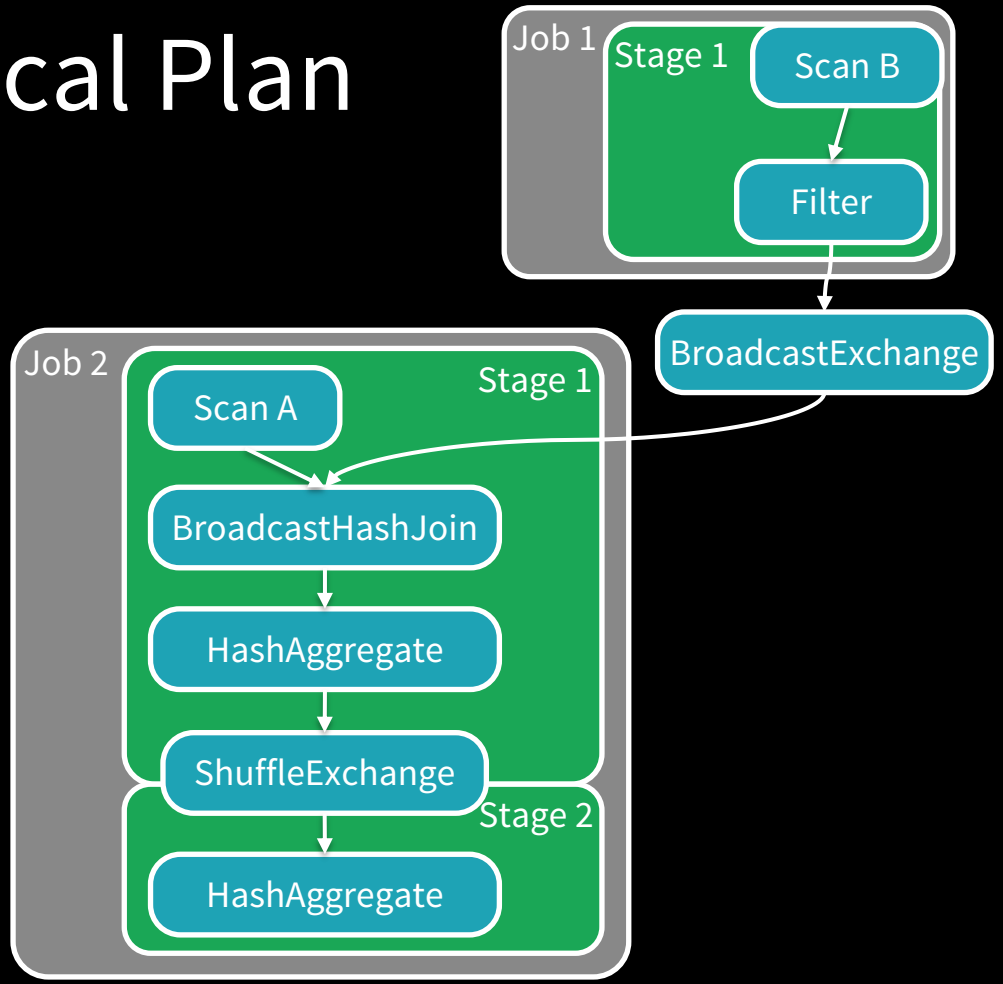
Scheduling a Physical Plan

- Scalar subquery

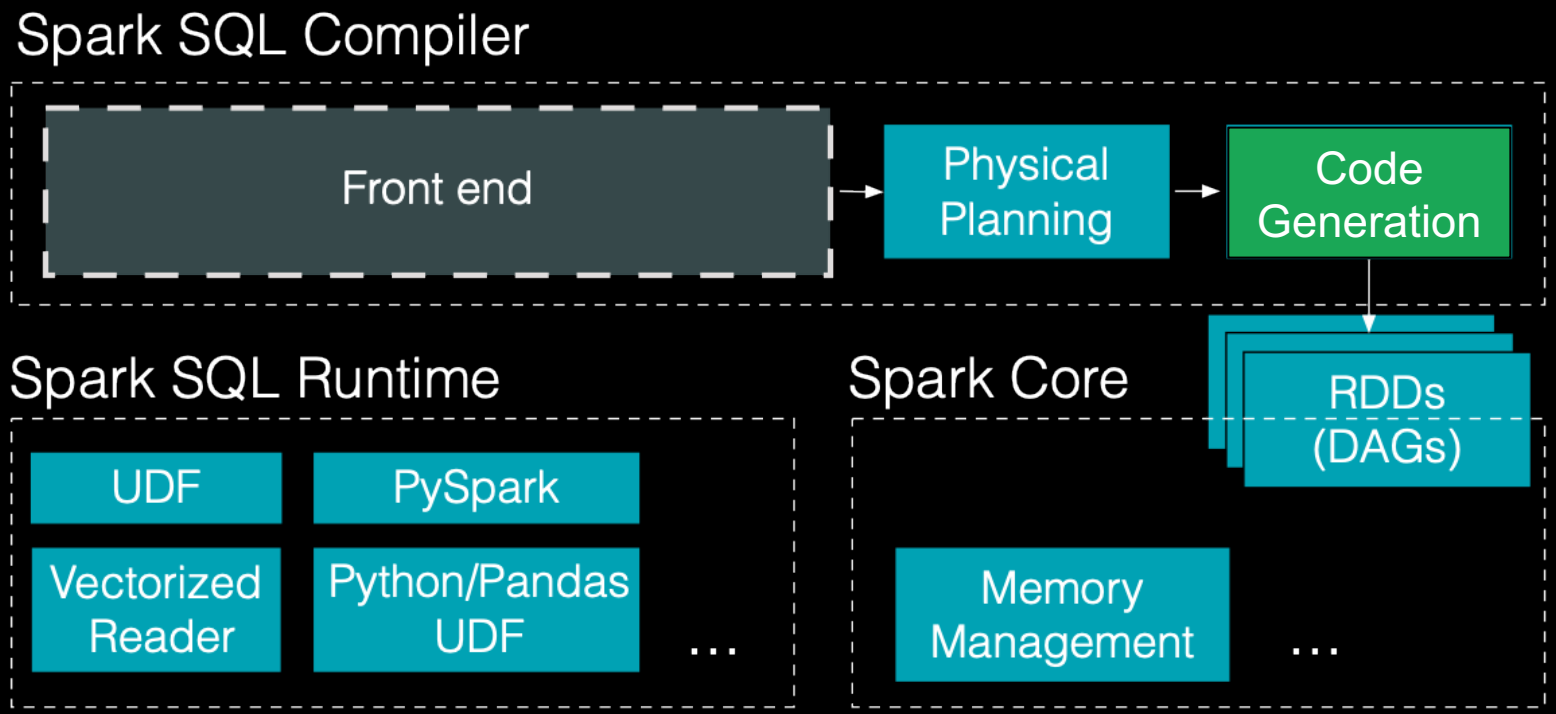
  Broadcast exchange:
  - Executed as separate jobs

- Partition-local ops:
  - Executed in the same stage

- Shuffle:
  - The stage boundary
  - A sync barrier across all nodes



Agenda


Spark SQL Compiler
Front end
Physical Planning
Code Generation
RDDs (DAGs)
Spark SQL Runtime
UDF
PySpark
Vectorized Reader
Python/Pandas UDF
…
Spark Core
Memory Management
…

Execution, Old: Volcano Iterator Model

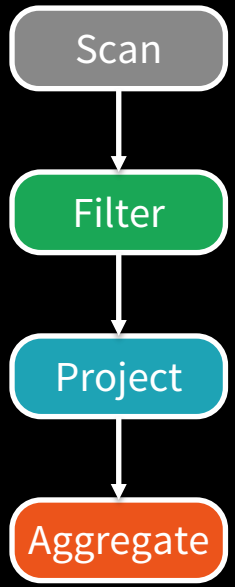
- Volcano iterator model
  - All ops implement the same interface, e.g., next()
  - next() on final op -> pull input from child op by calling child.next() -> goes on and on, ending up with a propagation of next() calls
- Pros: Good abstraction; Easy to implement
- Cons: Virtual function calls —> less efficient

Execution, New: Whole-Stage Code Generation

- Inspired by Thomas Neumann's paper

- Fuse a string of operators (oftentimes the entire stage) into one WSCG op that runs the generated code.

- A general-purpose execution engine just like Volcano model but without Volcano's performance downsides:
  - No virtual function calls
  - Data in CPU registers
  - Loop unrolling & SIMD



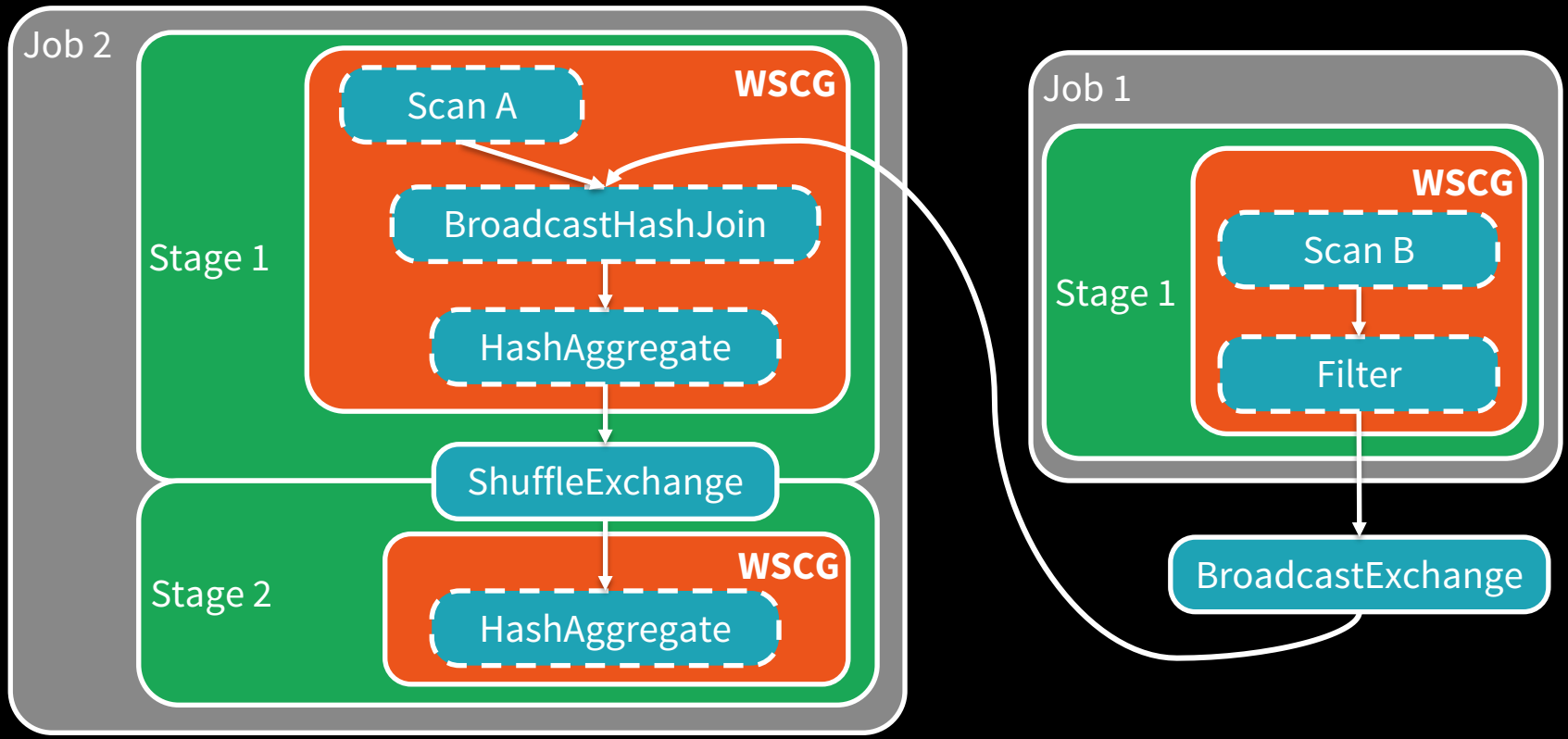

```
long count = 0;
for (item in sales) {
  if (price < 100) {
    count += 1;
  }
}
```

Execution Models: Old vs. New

- Volcano iterator model: Pull model; Driven by the final operator

- WSCG model: Push model; Driven by the head/source operator

A Physical Plan Example - WSCG



Implementation

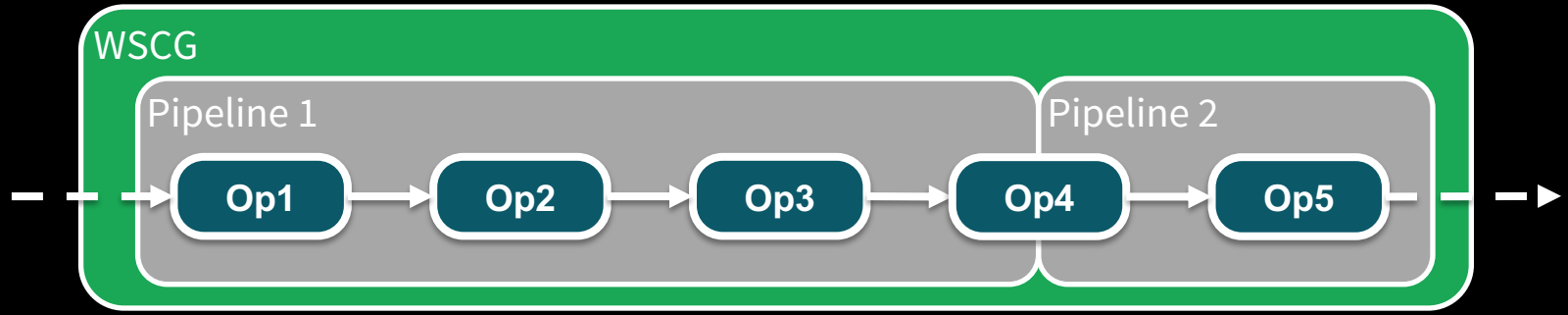
- The top node WholeStageCodegenExec implements the iterator interface to interop with other code-gen or non-code-gen physical ops.

- All underlying operators implement a code-generation interface: **doProduce()** & **doConsume()**

- Dump the generated code: `df.queryExecution.debug.codegen`

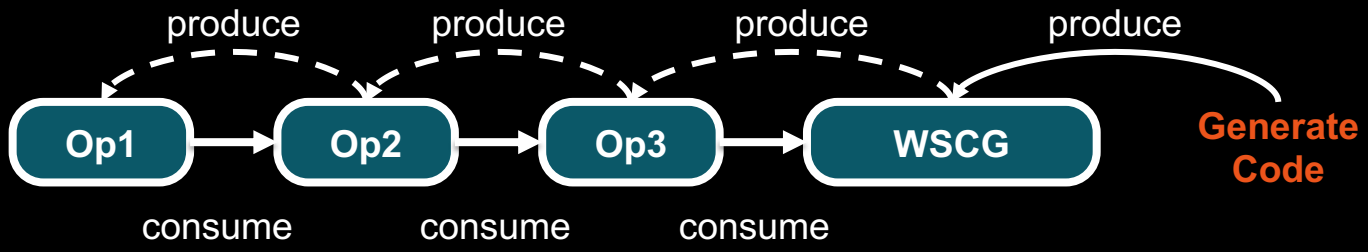
Single dependency

- A WSCG node contains a linear list of physical operators that support code generation.
- No multi dependency between enclosed ops.
- A WSCG node may consist of one or more pipelines.

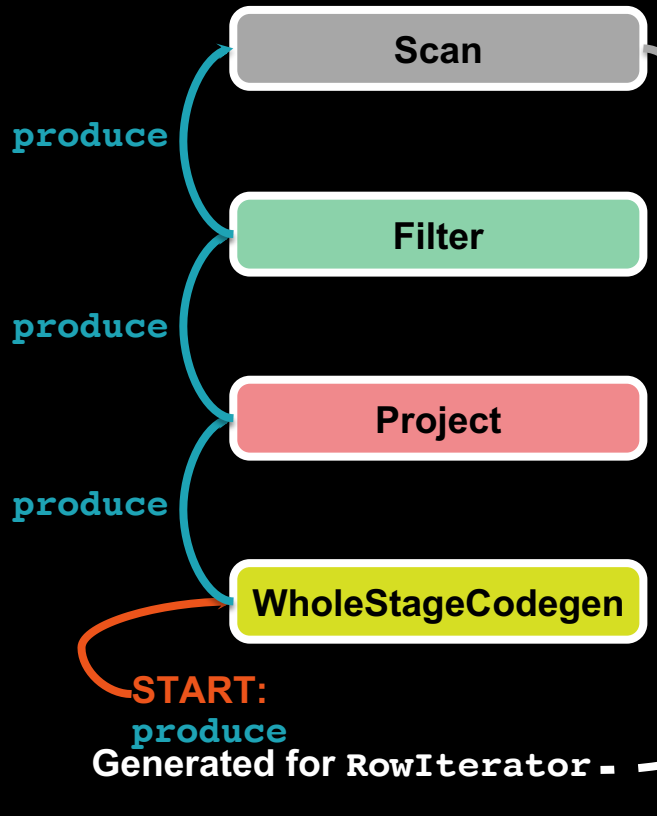


A Single Pipeline in WSCG

- A string of non-blocking operators form a pipeline in WSCG
- The head/source:
  - Implement `doProduce()` - the driving loop producing source data.
- The rest:
  - `doProduce()` - fall through to head of the pipeline.
  - Implement `doConsume()` for its own processing logic.



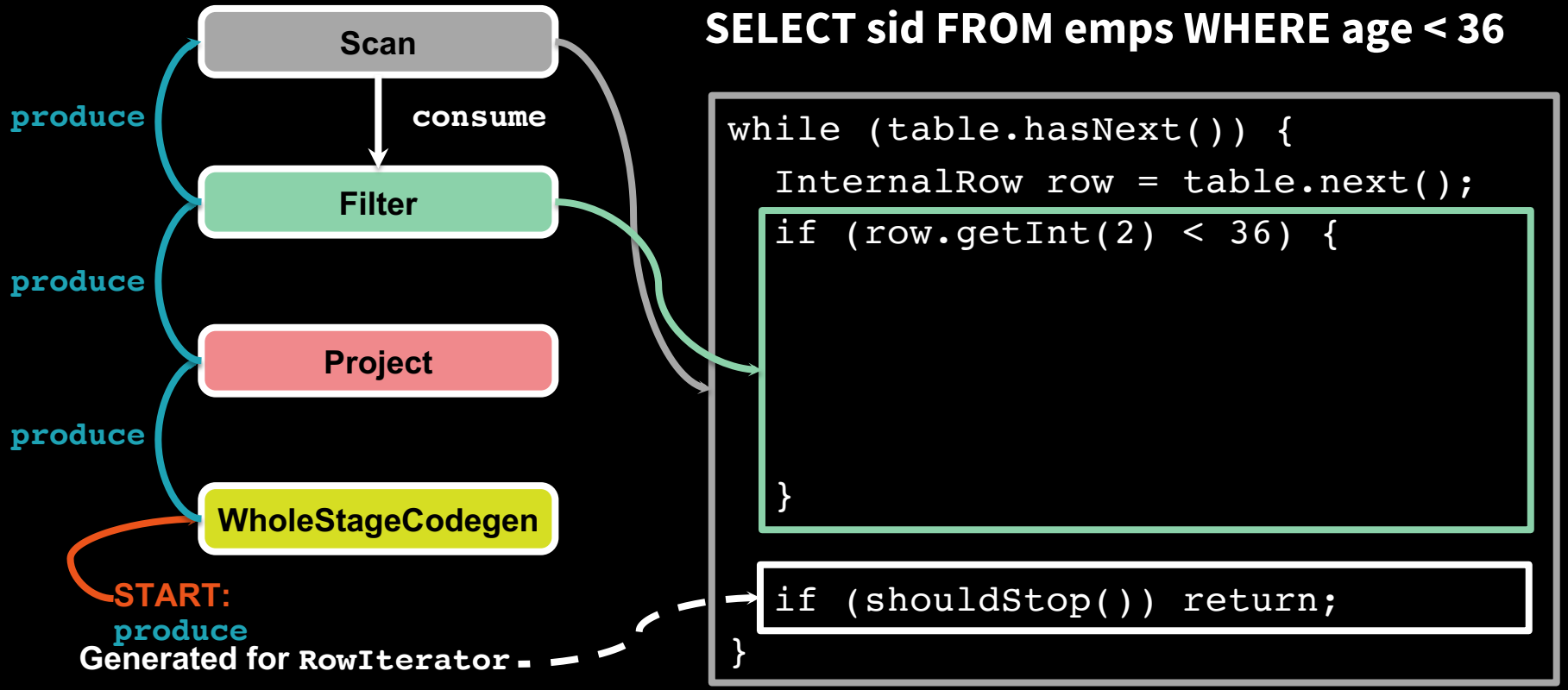
A Single Pipeline Example



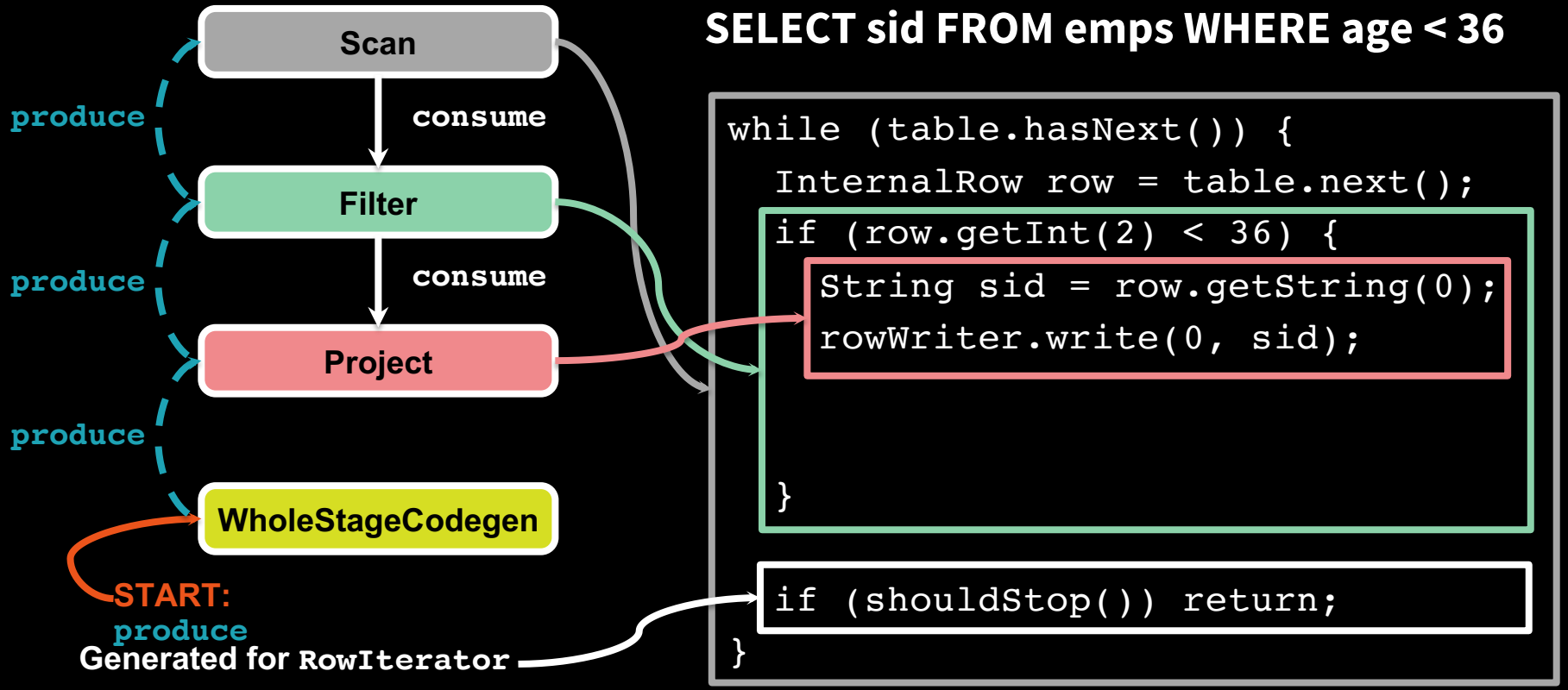
SELECT sid FROM emps WHERE age < 36

```
while (table.hasNext()) {
  InternalRow row = table.next();

  if (shouldStop()) return;
}
```

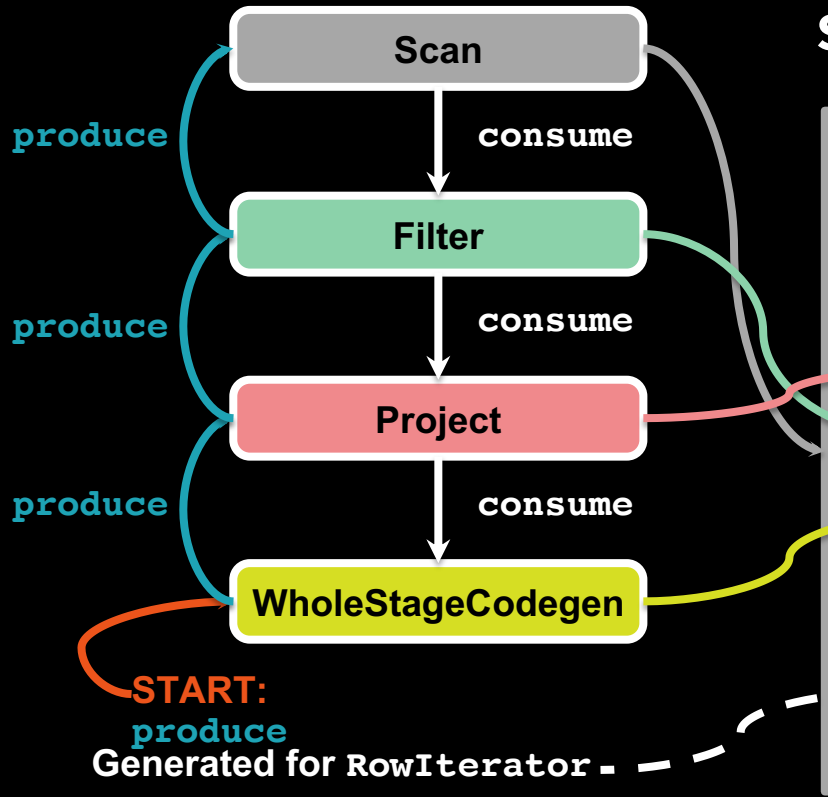
A Single Pipeline Example



A Single Pipeline Example



A Single Pipeline Example

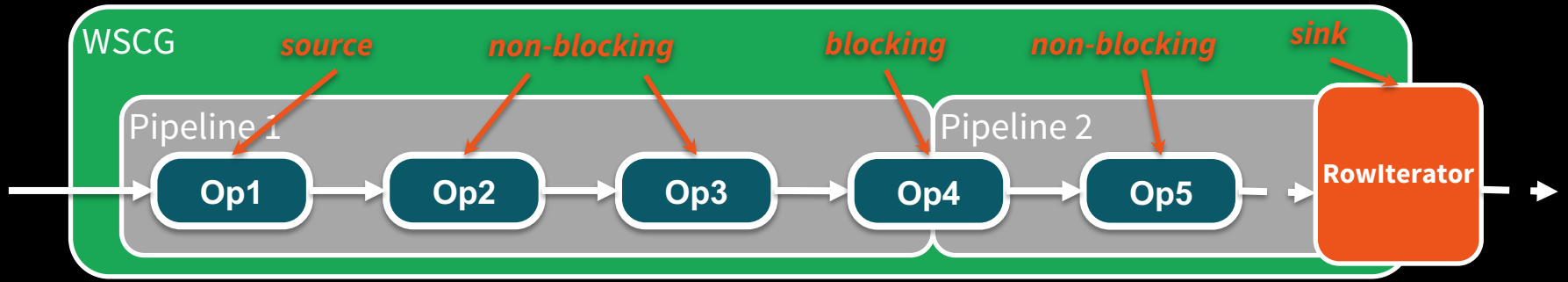


SELECT sid FROM emps WHERE age < 36

```
while (table.hasNext()) {
  InternalRow row = table.next();
  if (row.getInt(2) < 36) {
    String sid = row.getString(0);
    rowWriter.write(0, sid);

    ret = rowWriter.getRow();
  }

  if (shouldStop()) return;
}
```

Multiple Pipelines in WSCG

- Head (source) operator:
  - The source, w/ or w/o input RDDs
  - e.g., Scan, SortMergeJoin
- Non-blocking operators:
  - In the middle of the pipeline
  - e.g., Filter, Project
- End (sink): `RowIterator`
  - Pulls result from the last pipeline
- Blocking operators:
  - End of the previous pipeline
  - Start of a new pipeline
  - e.g., HashAggregate, Sort

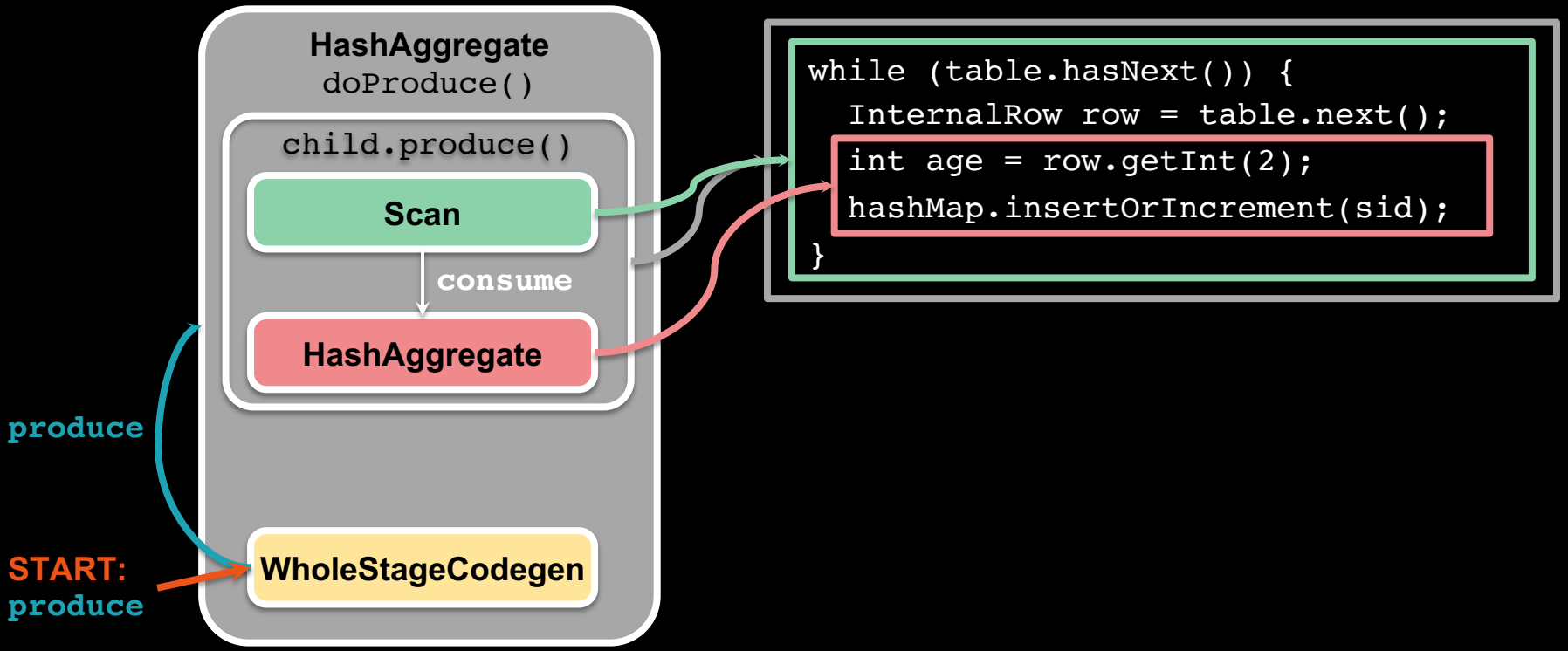


Blocking Operators in WSCG

- A Blocking operator, e.g., HashAggregateExec, SortExec, break pipelines, so there may be multiple pipelines in one WSCG node.
- A Blocking operator's `doConsume()`:
  - Implement the callback to build intermediate result.
- A Blocking operator's `doProduce()`:
  - Consume the entire output from upstream to finish building the intermediate result.
  - Start a new loop and produce output for downstream based on the intermediate result.

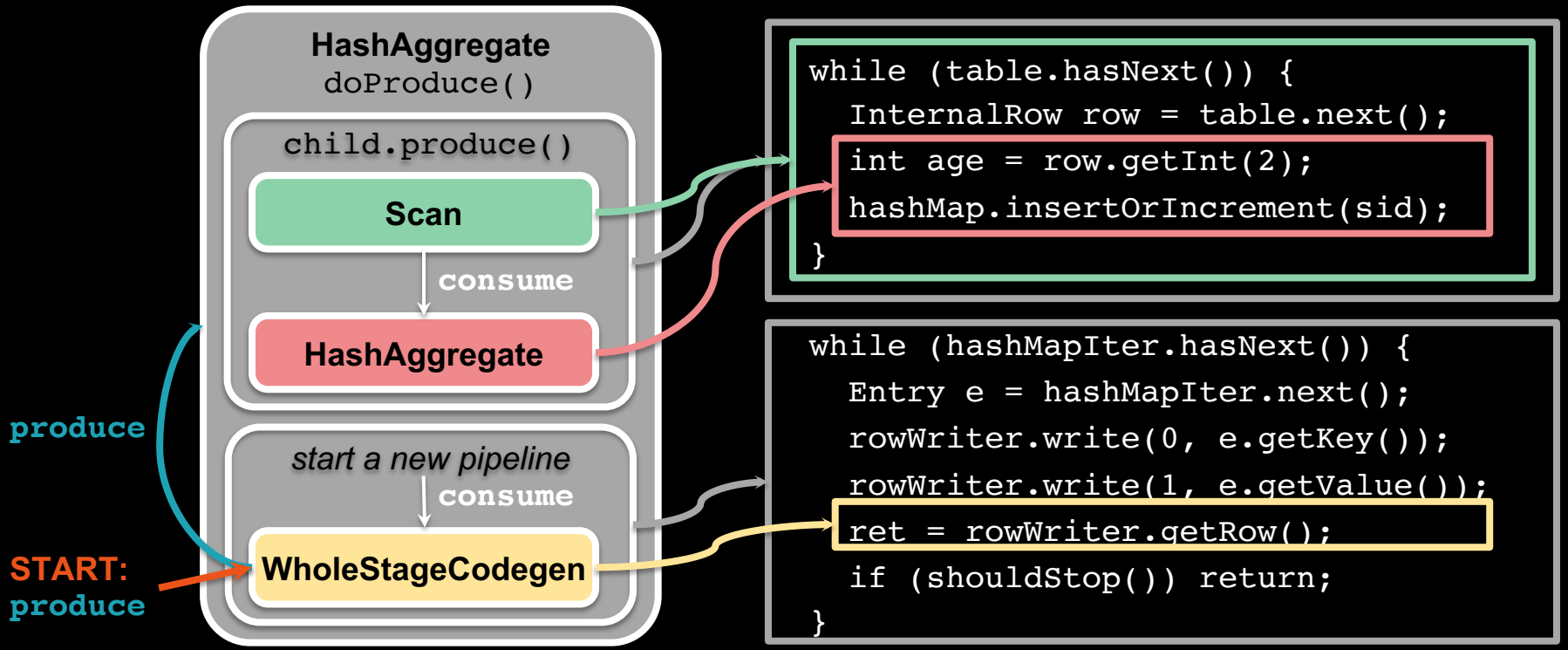
A Blocking Operator Example - HashAgg

SELECT age, count(*) FROM emps GROUP BY age



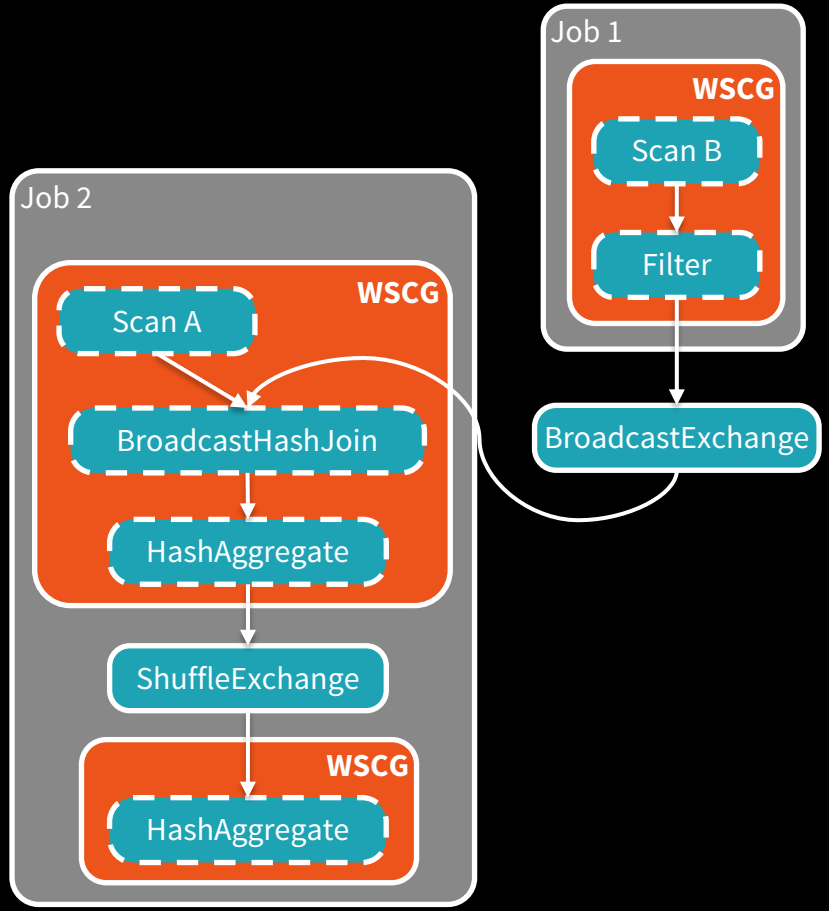
A Blocking Operator Example - HashAgg

SELECT age, count(*) FROM emps GROUP BY age



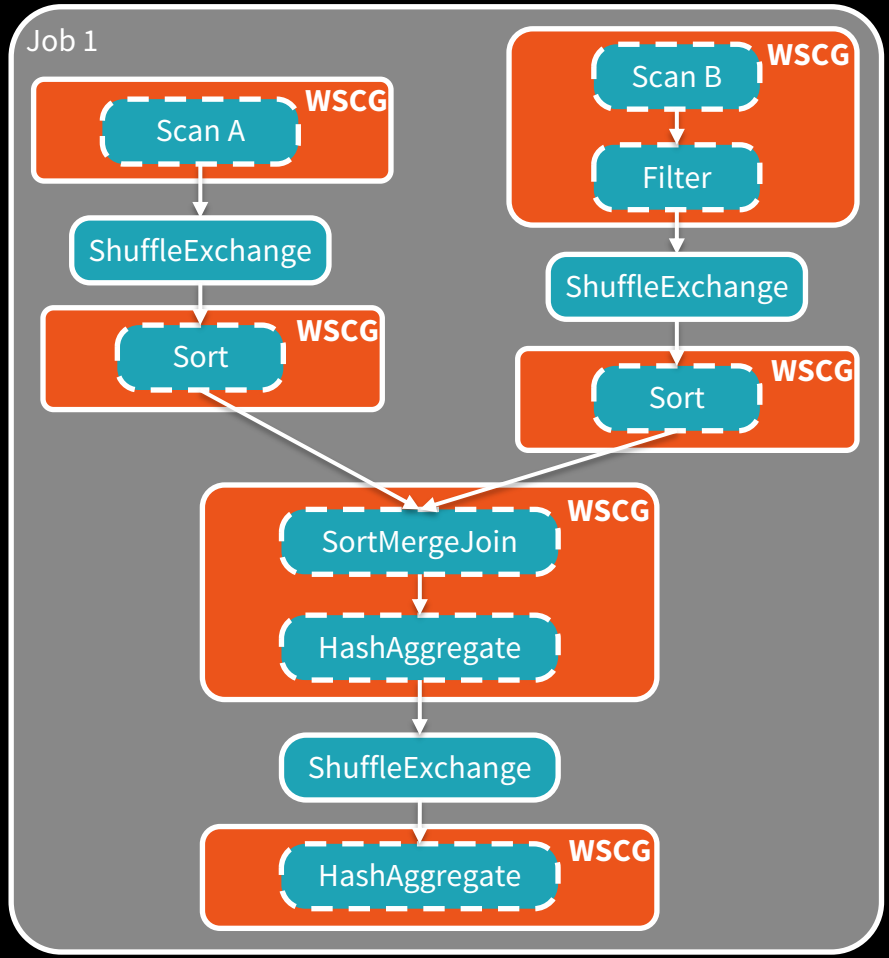
WSCG: BHJ vs. SMJ

- BHJ (broadcast-hash-join) is a pipelined operator.
- BHJ executes the build side job first, the same way as in non-WSCG.
- BHJ is fused together with the probe side plan (i.e., streaming plan) in WSCG.



WSCG: BHJ vs. SMJ

- SMJ (sort-merge-join) is NOT fused with either child plan for WSCG. Child plans are separate WSCG nodes.
- Thus, SMJ must be the head operator of a WSCG node.



WSCG Limitations

- Problems:
  - No JIT compilation for bytecode size over 8000 bytes (*).
  - Over 64KB methods NOT allowed by Java Class format.

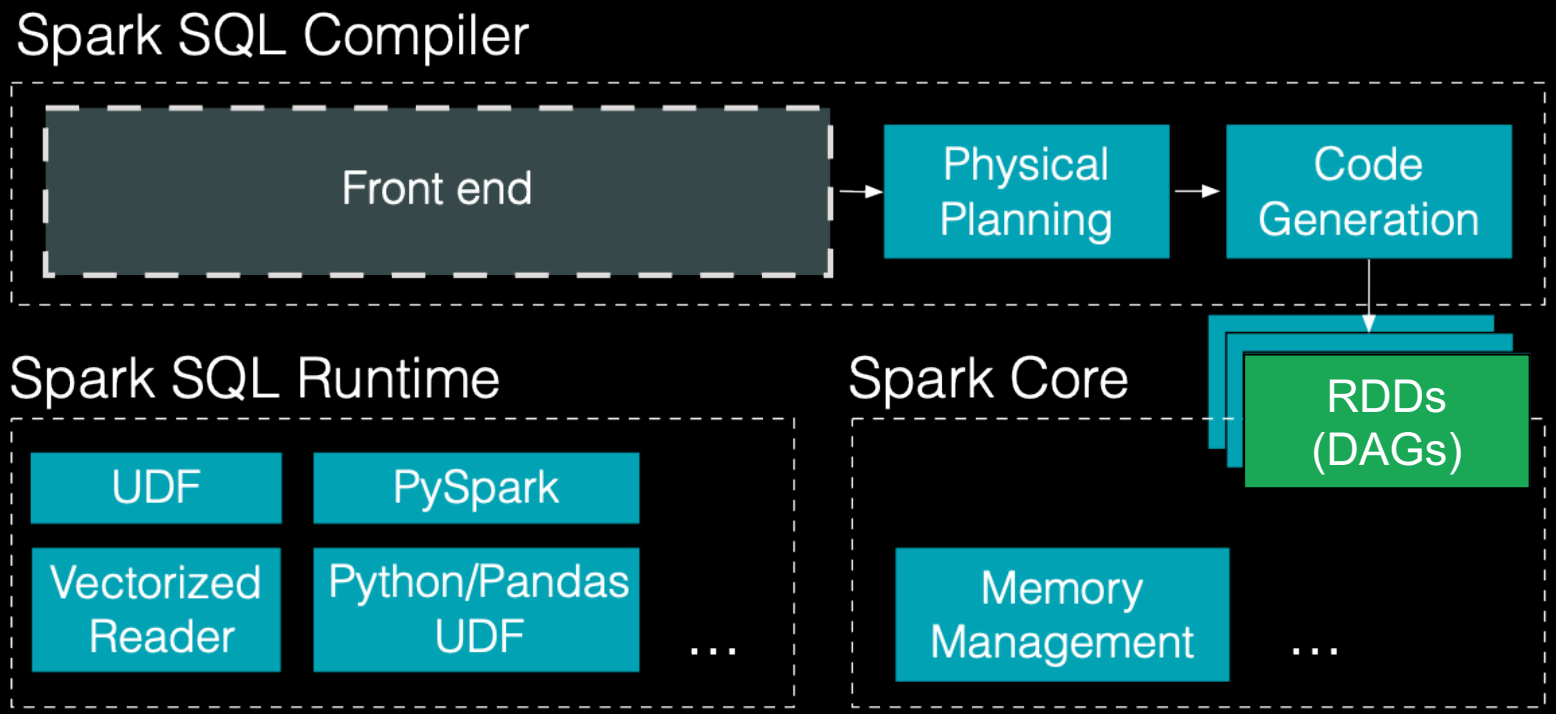
- Solutions:
  - Fallback - `spark.sql.codegen.fallback; spark.sql.codegen.hugeMethodLimit`
  - Move blocking loops into separate methods, e.g. hash-map building in HashAgg and sort buffer building in Sort.
  - Split consume() into individual methods for each operator - `spark.sql.codegen.splitConsumeFuncByOperator`

About Us

Software Engineers 

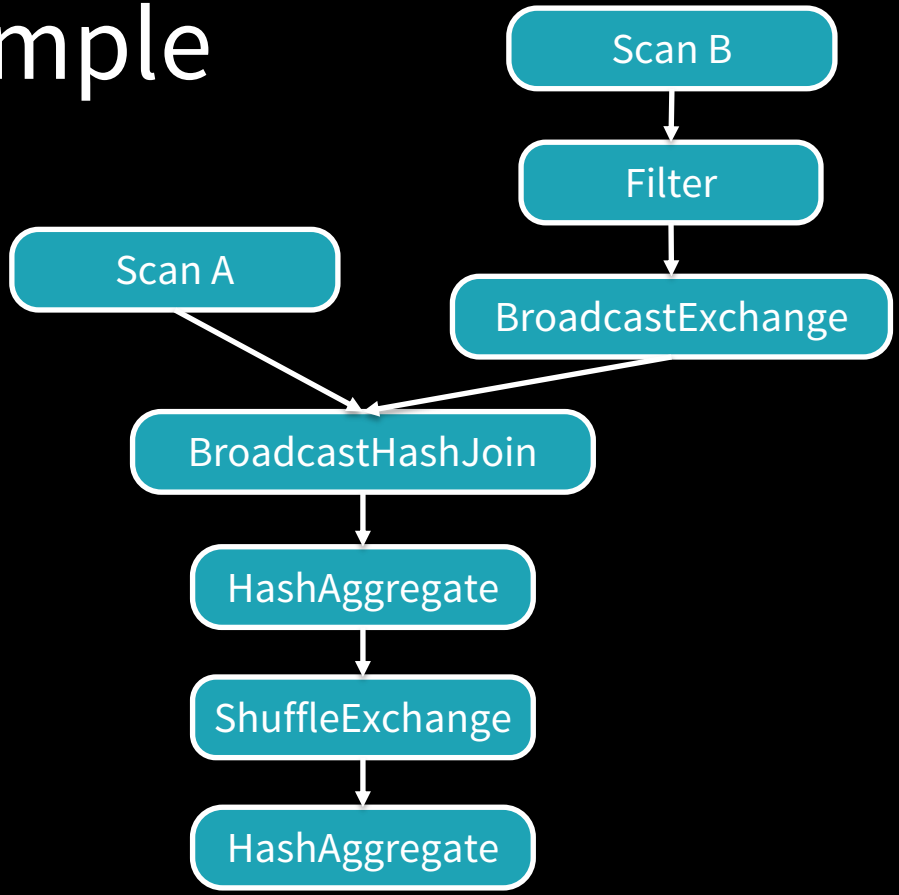
- Maryann Xue
  PMC of Apache Calcite & Apache Phoenix @maryannxue
- **Xingbo Jiang**
  Apache Spark Committer @jiangxb1987
- Kris Mok
  OpenJDK Committer @rednaxelafx

Agenda


Spark SQL Compiler
Front end
Physical Planning
Code Generation
RDDs (DAGs)
Spark SQL Runtime
UDF
PySpark
Vectorized Reader
Python/Pandas UDF
…
Spark Core
Memory Management
…

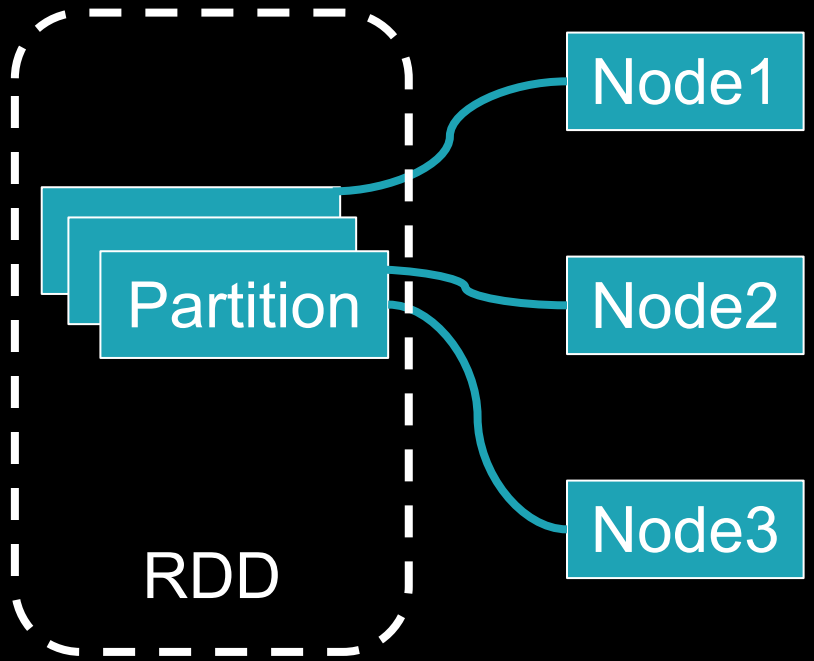
A Physical Plan Example

```
SELECT a1, sum(b1)
FROM A JOIN B
ON A.key = B.key
WHERE b1 < 1000
GROUP BY a1
```

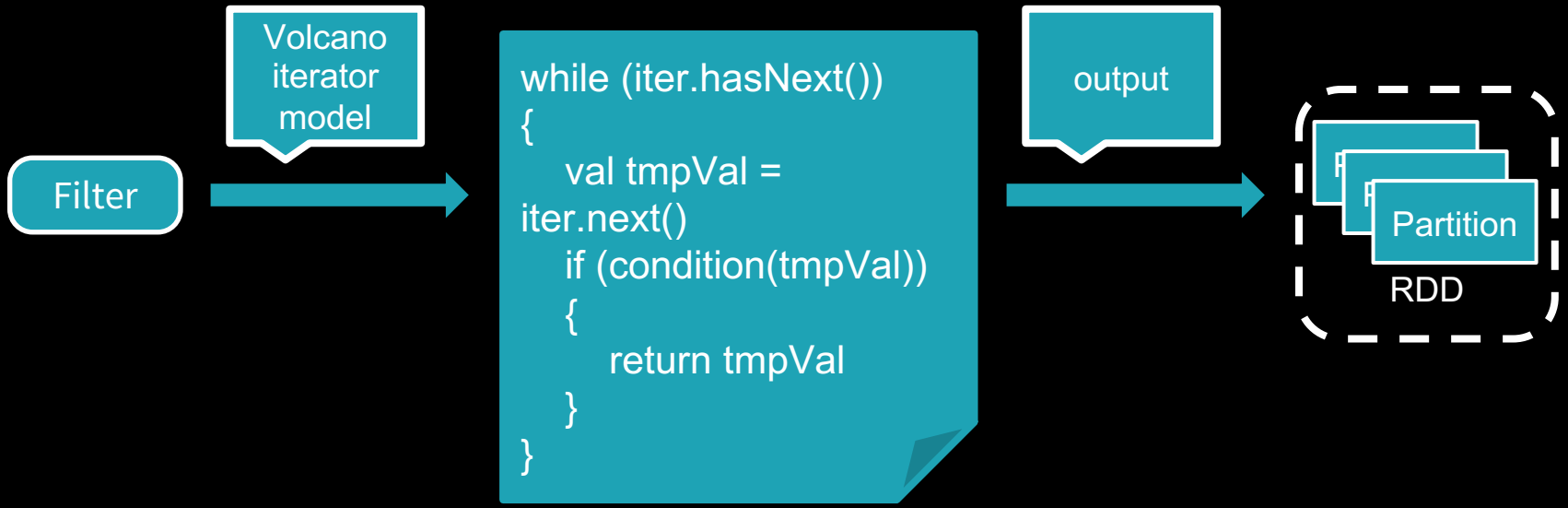


RDD and Partitions

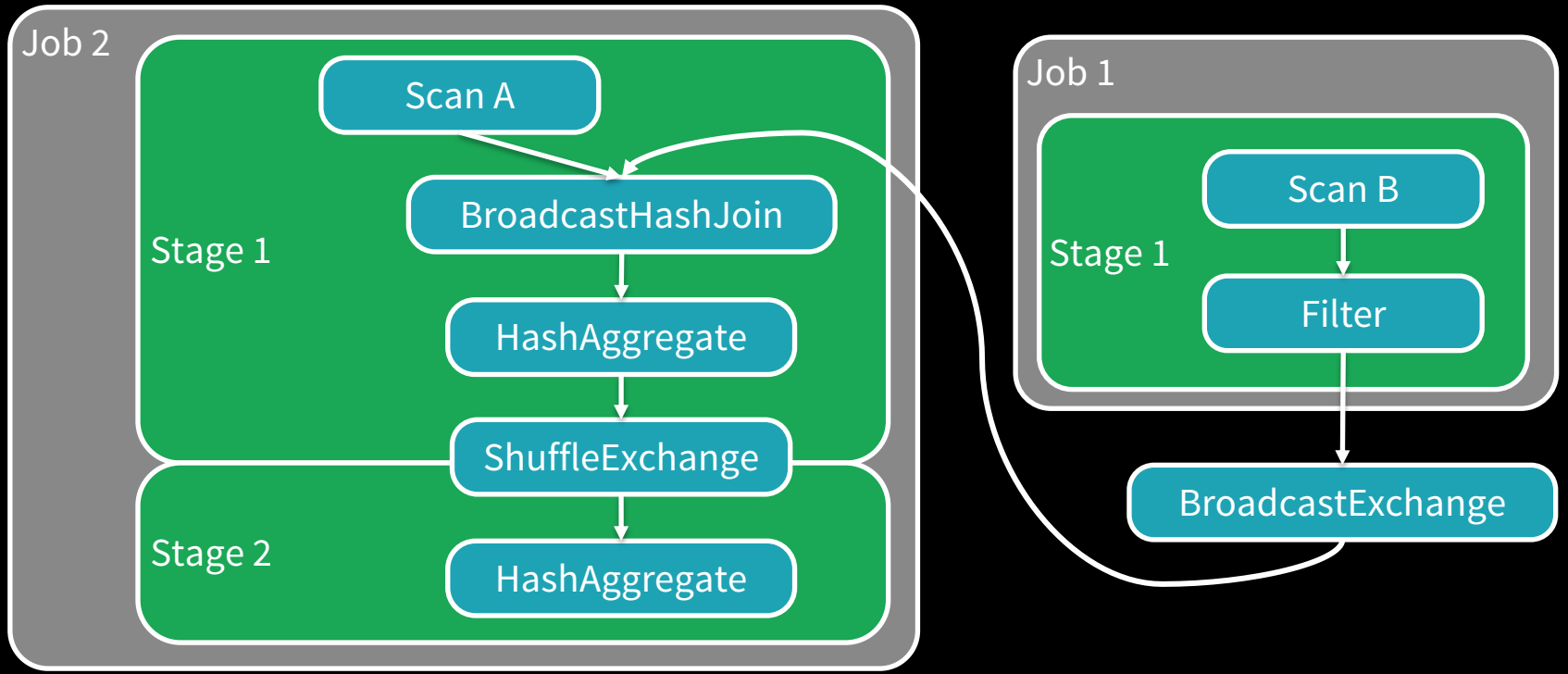
RDD(Resilient Distributed Dataset) represents an immutable, partitioned collection of elements that can be operated in parallel.



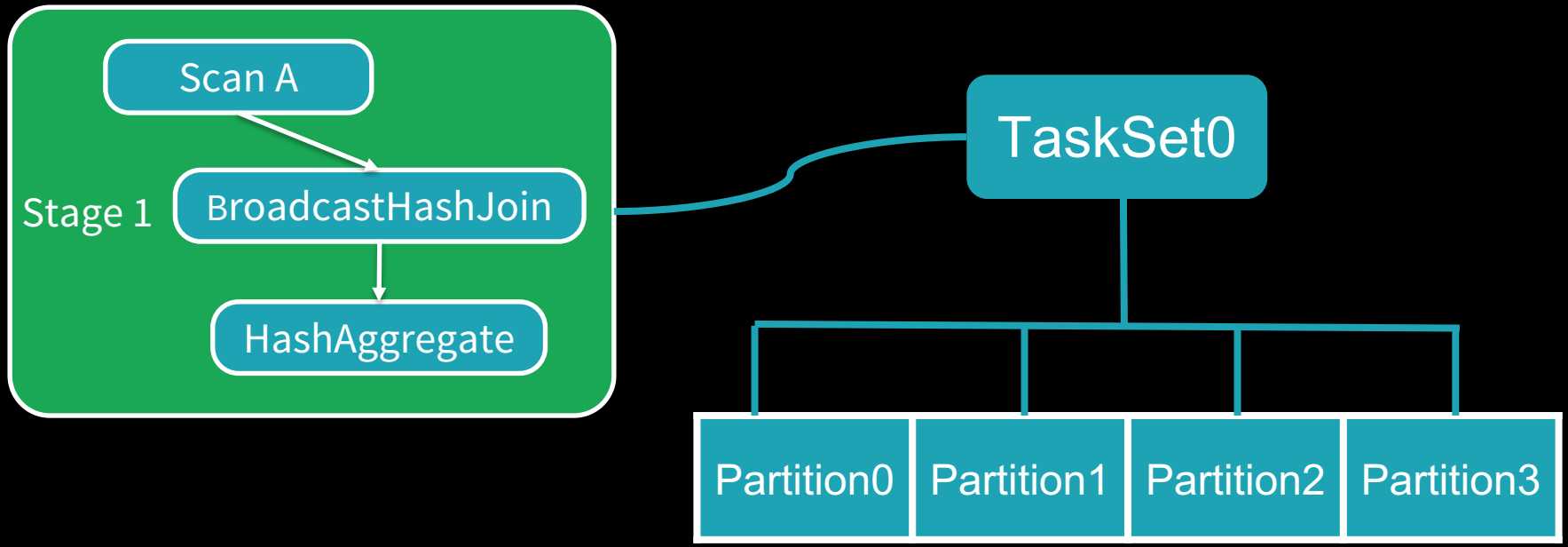
Physical Operator



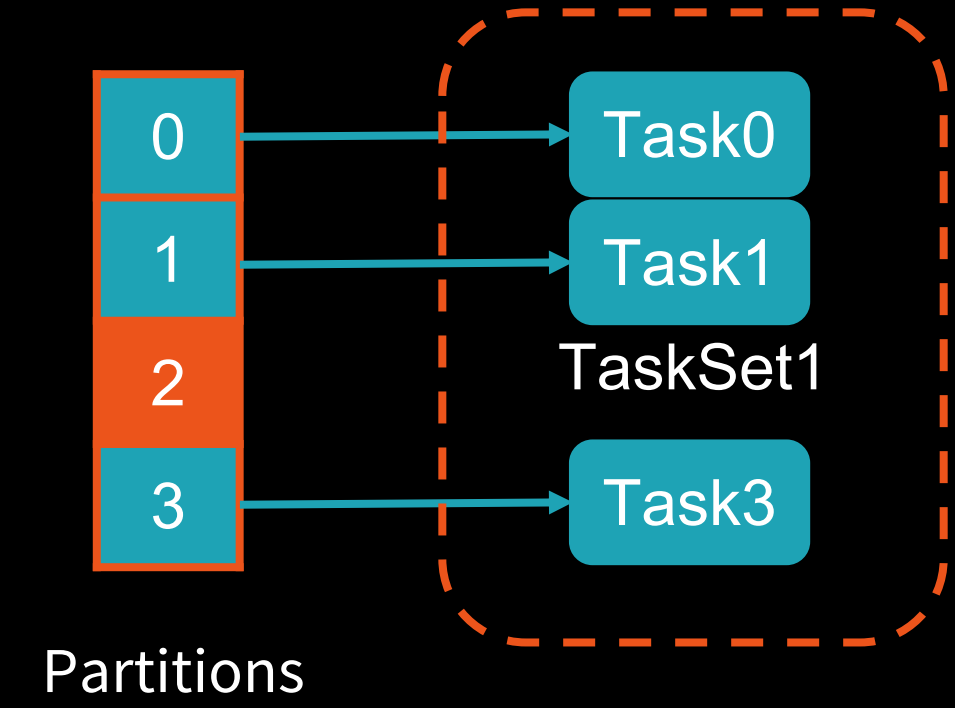
A Physical Plan Example - Scheduling



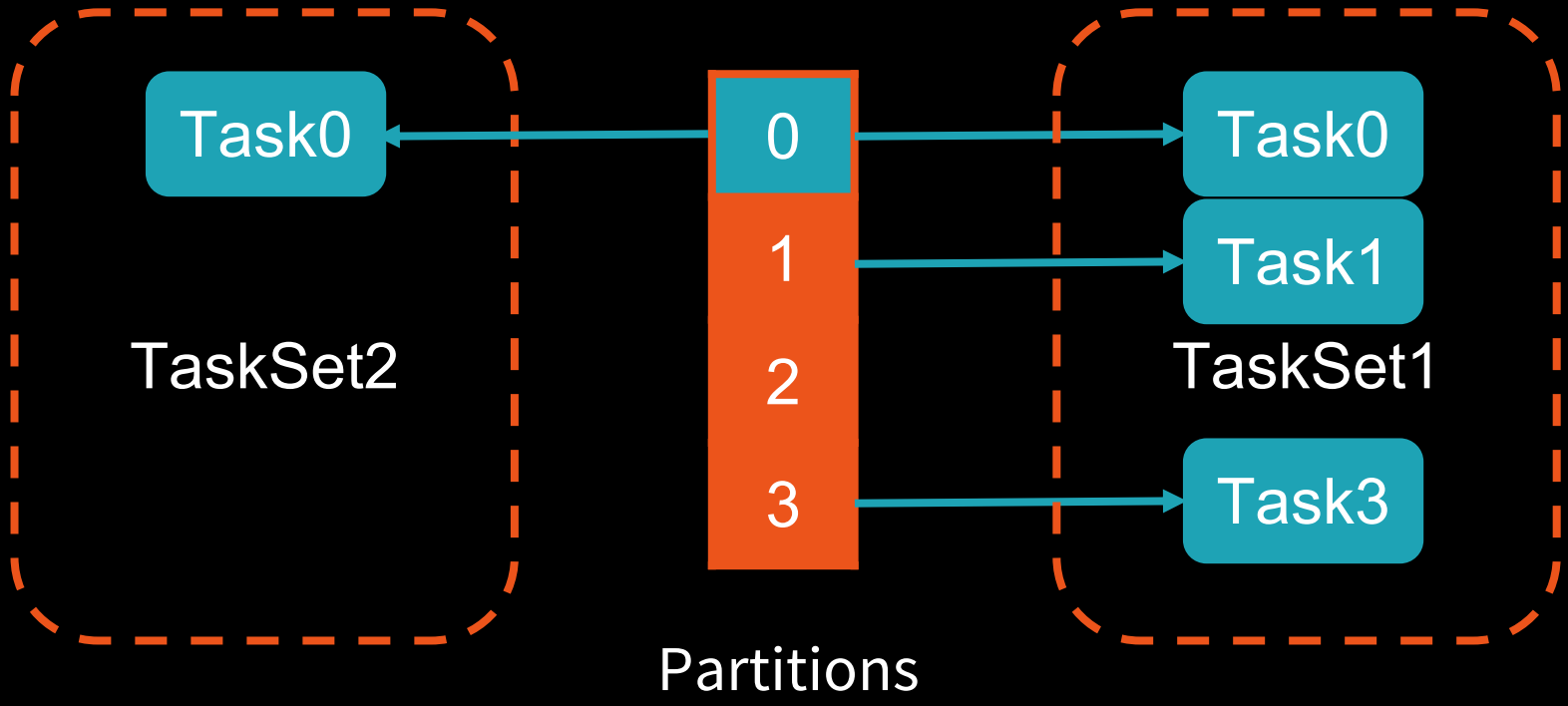
Stage Execution



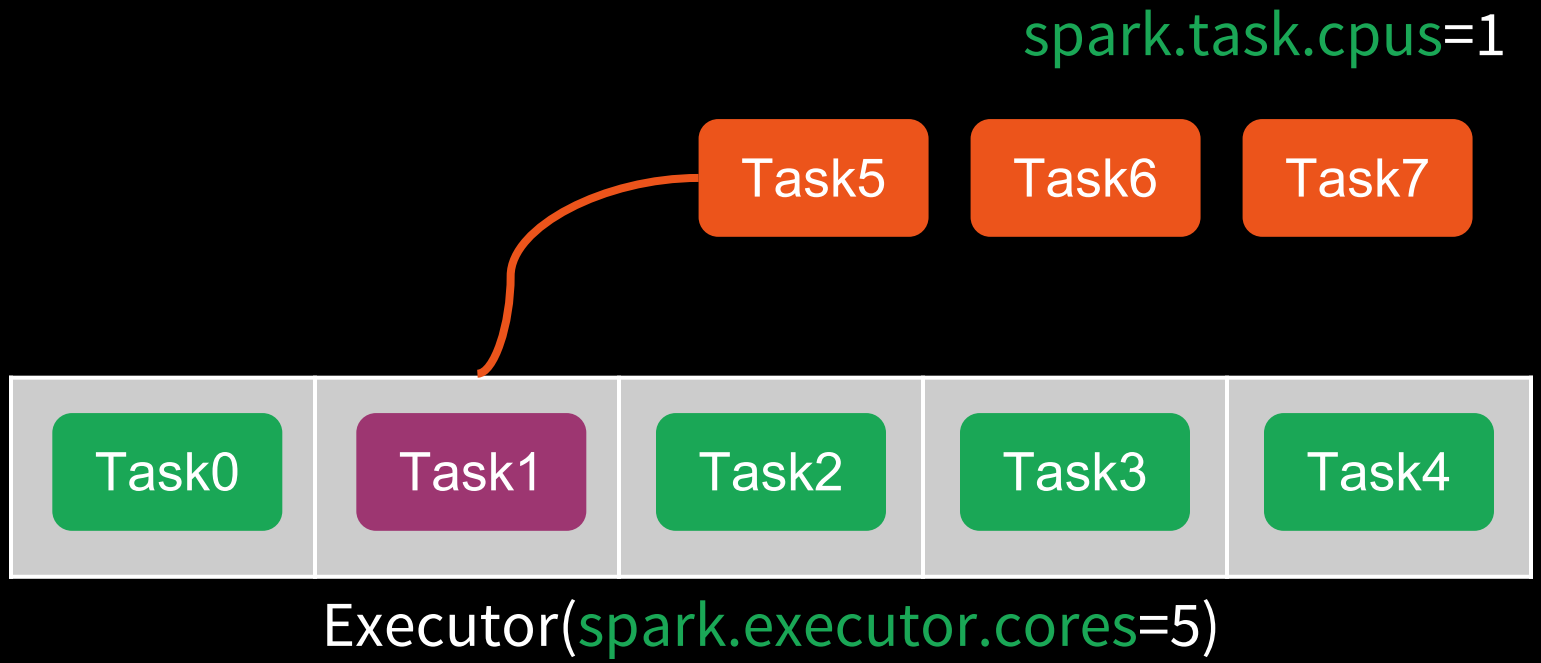
Stage Execution



Stage Execution



How to run a Task



Fault Tolerance

- MPP-like analytics engines(e.g., Teradata, Presto, Impala):
  - Coarser-grained recovery model
  - Retry an entire query if any machine fails
  - Short/simple queries
- Spark SQL:
  - Mid-query recovery model
  - RDDs track the series of transformations used to build them (the lineage) to recompute lost partitions.
  - Long/complex queries [e.g., complex UDFs]

Handling Task Failures

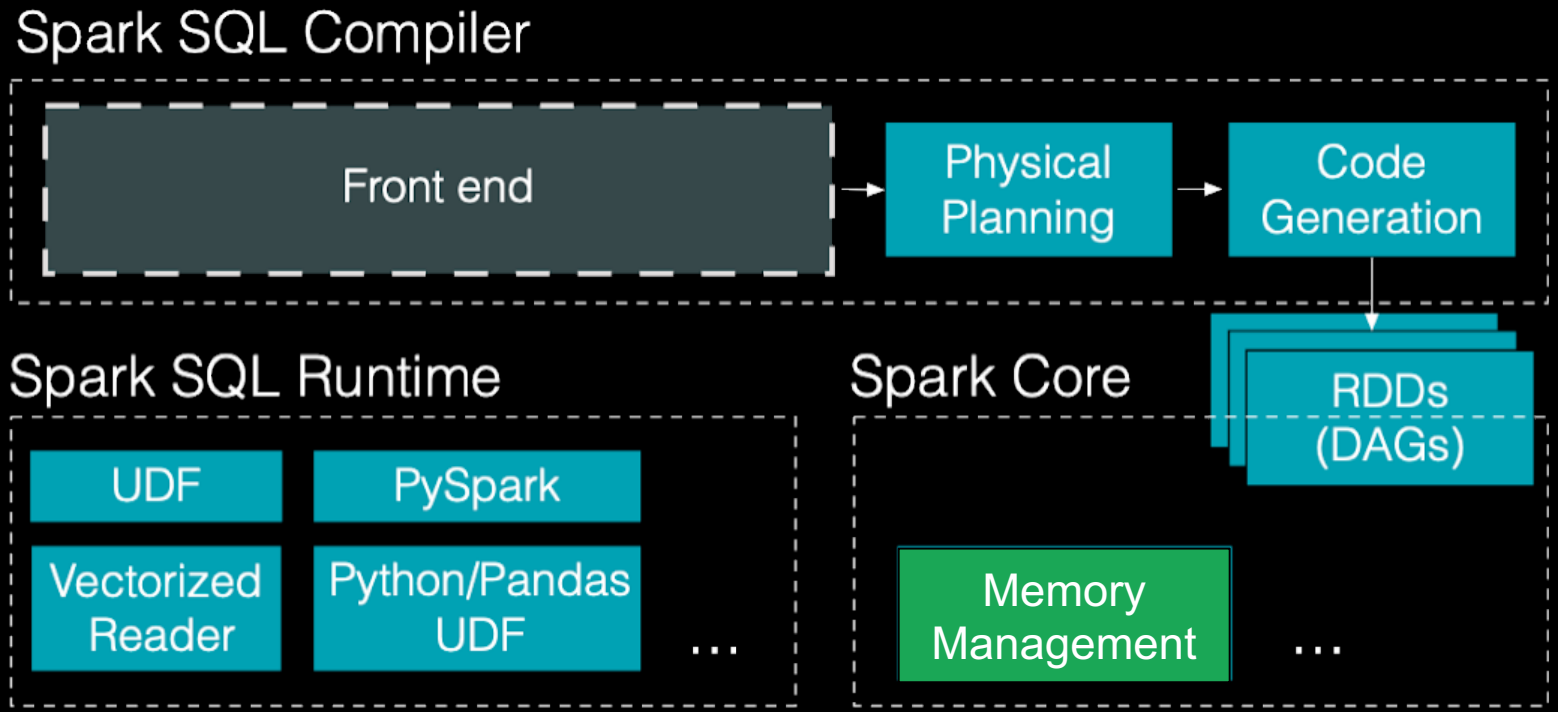
Task Failure

- Record the failure count of the task
- Retry the task if failure count < maxTaskFailures
- Abort the stage and corresponding jobs if count >= maxTaskFailures

Fetch Failure

- Don't count the failure into task failure count
- Retry the stage if stage failure < maxStageFailures
- Abort the stage and corresponding jobs if stage failure >= maxStageFailures
- Mark executor/host as lost (optional)

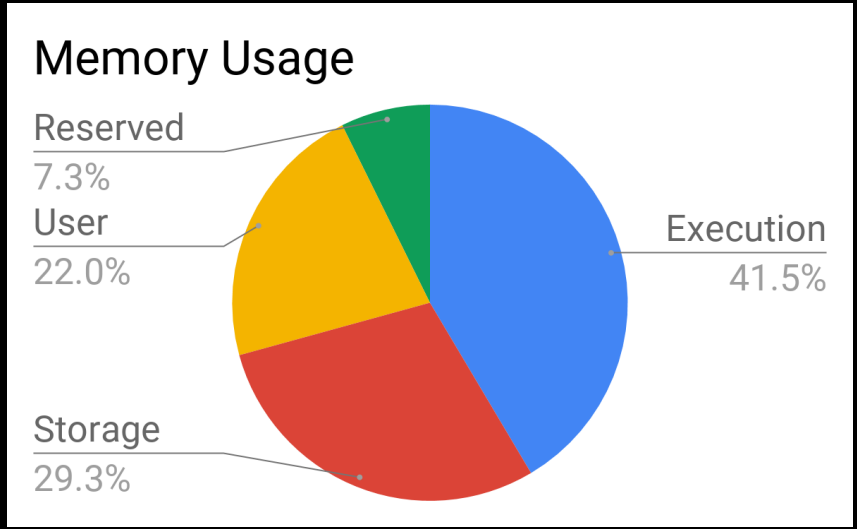
Agenda


Spark SQL Compiler
Front end
Physical Planning
Code Generation
RDDs (DAGs)
Spark SQL Runtime
UDF
PySpark
Vectorized Reader
Python/Pandas UDF
...
Spark Core
Memory Management
...

Memory Consumption in Executor JVM

Spark uses memory for:

- RDD Storage [e.g., call cache()].
- Execution memory [e.g., Shuffle and aggregation buffers]
- User code [e.g., allocate large arrays]

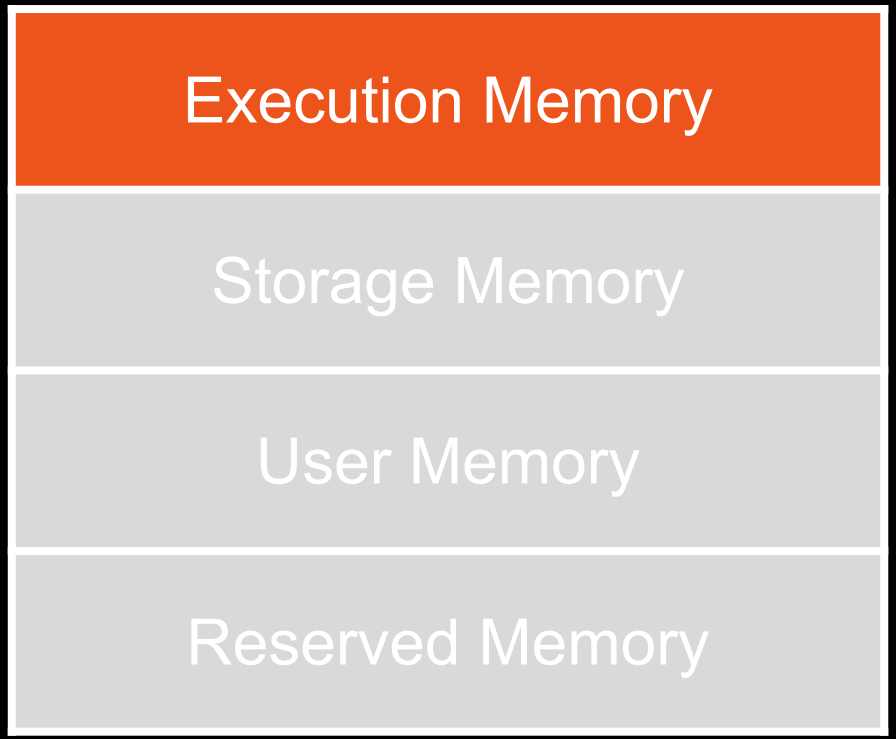


Challenges:

- Task run in a shared-memory environment.
- Memory resource is not enough!

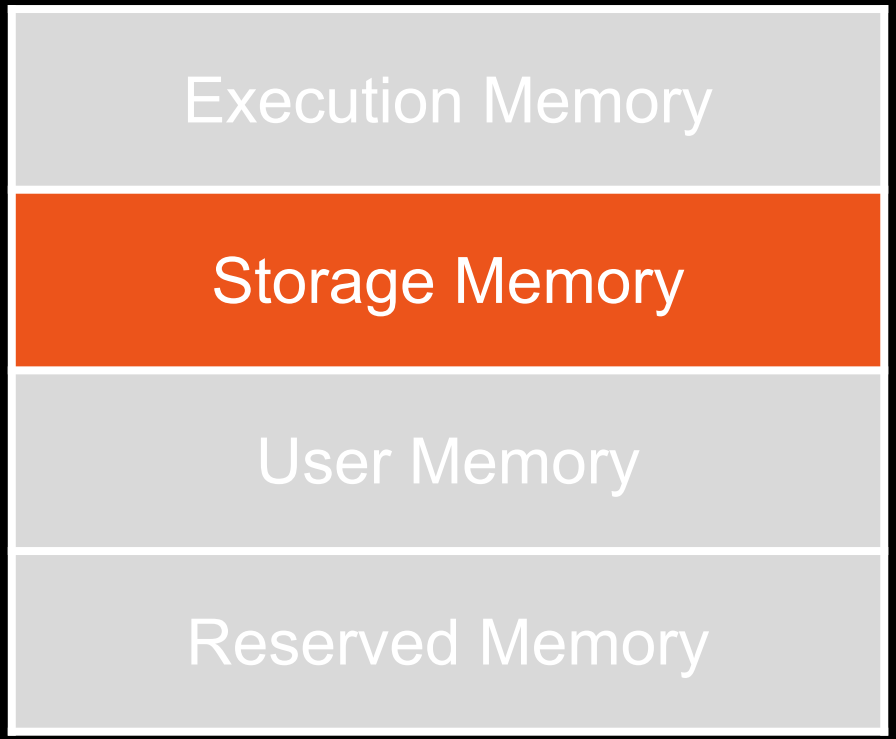
Execution Memory

- Buffer intermediate results
- Normally short lived



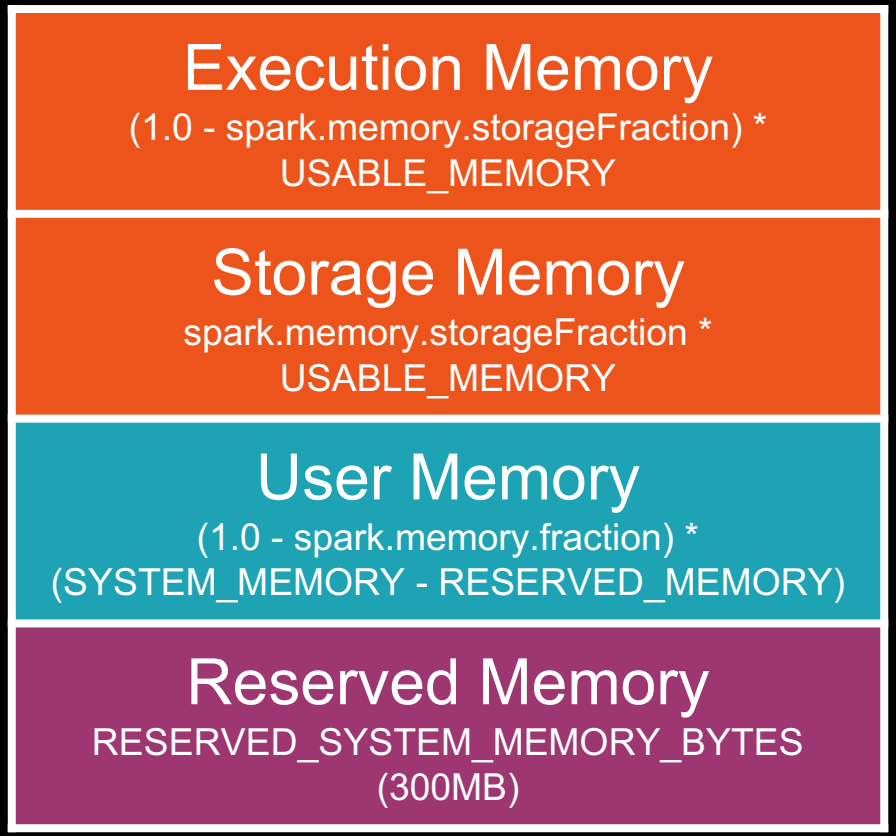
Storage Memory

- Reuse data for future computation
- Cached data can be long-lived
- LRU eviction for spill data



Unified Memory Manager

- Express execution and storage memory as one single unified region
- Keep acquiring execution memory and evict storage as you need more execution memory



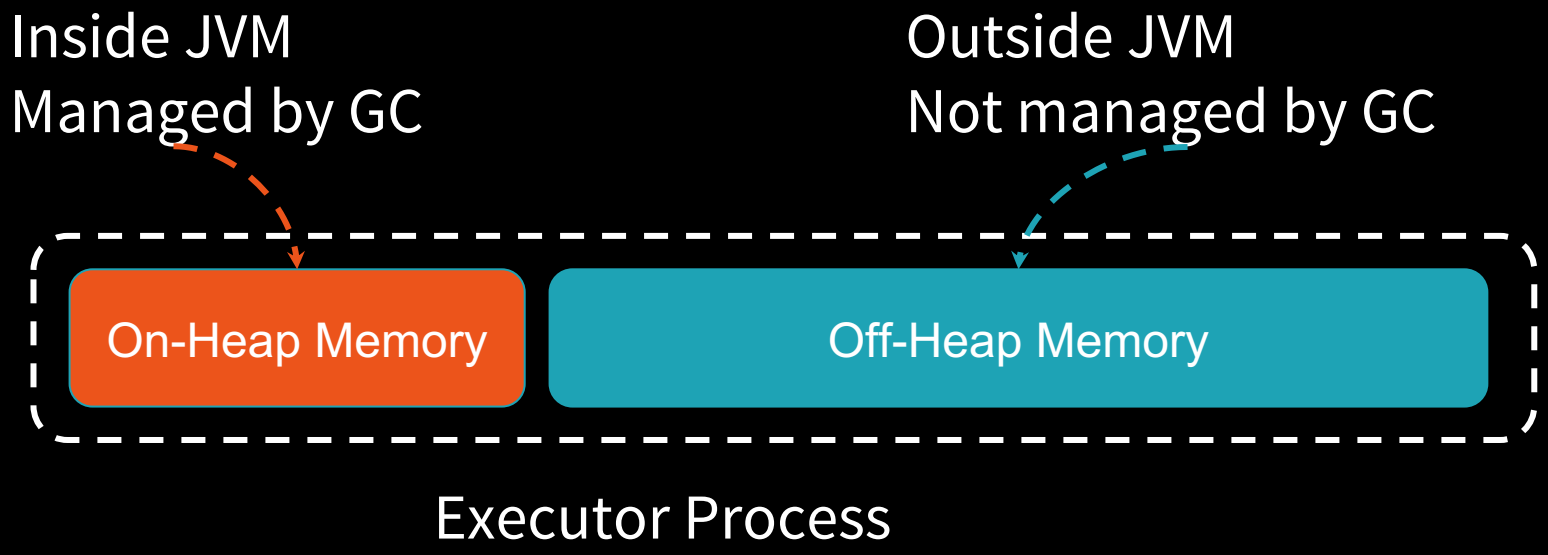
Dynamic occupancy mechanism

spark.memory.storageFraction

- If one of its space is insufficient but the other is free, then it will borrow the other's space.
- If both parties don't have enough space, evict storage memory using LRU mechanism.

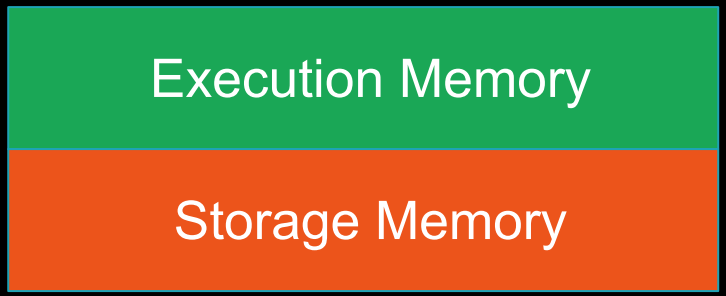
One problem remains...

- The memory resource is not enough!



Off-Heap Memory

- Enabled by spark.memory.offHeap.enabled
- Memory size controlled by spark.memory.offHeap.size



Off-Heap Memory

- Pros
  - Speed: Off-Heap Memory > Disk
  - Not bound by GC
- Cons
  - Manually manage memory allocation/release

Tuning Data Structures

In Spark applications:

- Prefer arrays of objects instead of collection classes (e.g., HashMap)
- Avoid nested structures with a lot of small objects and pointers when possible
- Use numeric IDs or enumeration objects instead of strings for keys

Tuning Memory Config

spark.memory.fraction

- More execution and storage memory
- Higher risk of OOM

spark.memory.storageFraction

- Increase storage memory to cache more data
- Less execution memory may lead to tasks spill more often

Tuning Memory Config

spark.memory.offHeap.enabled

spark.memory.offHeap.size

- Off-Heap memory not bound by GC
- On-Heap + Off-Heap memory must fit in total executor memory (spark.executor.memory)

spark.shuffle.file.buffer

spark.unsafe.sorter.spill.reader.buffer.size

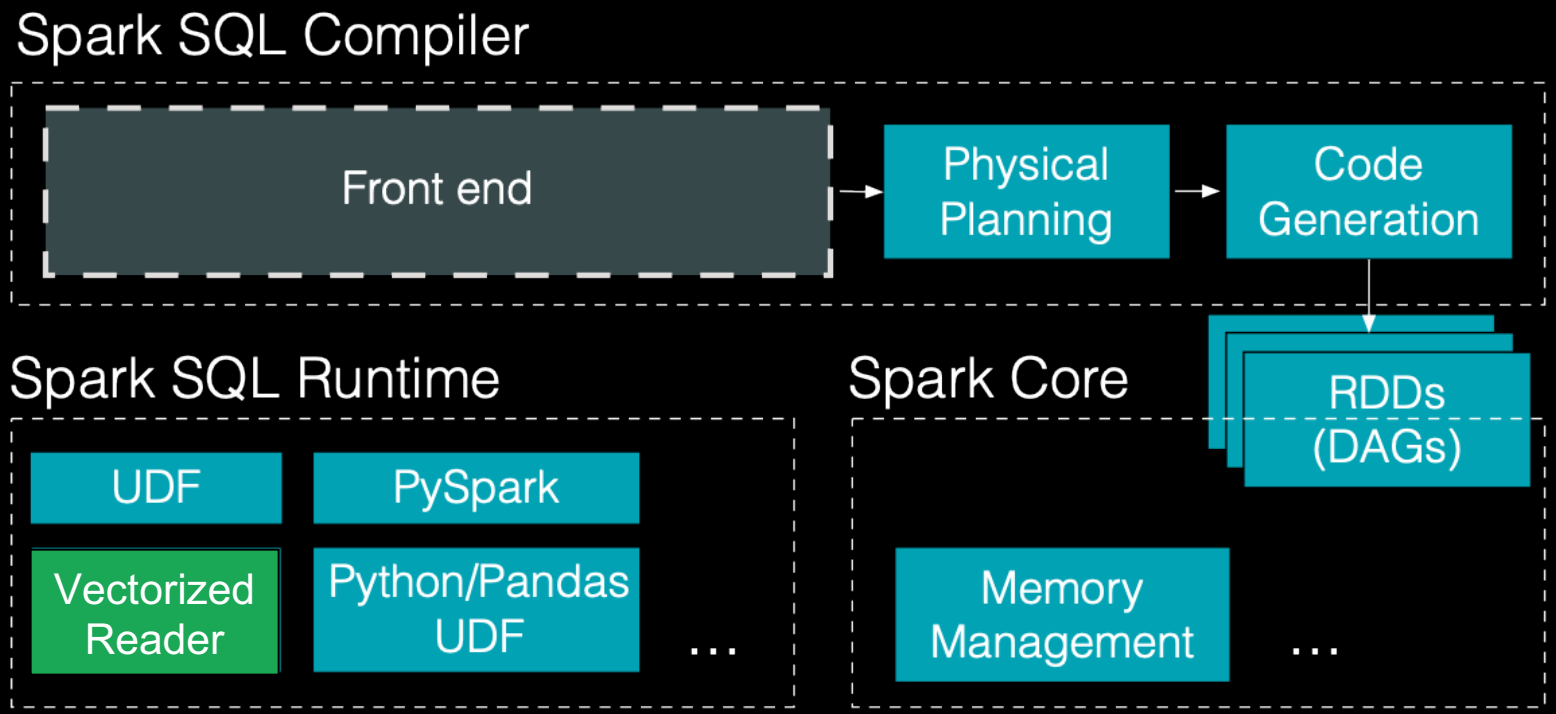
- Buffer shuffle file to amortize disk I/O
- More execution memory consumption

About Us

Software Engineers 

- Maryann Xue
  PMC of Apache Calcite & Apache Phoenix @maryannxue
- Xingbo Jiang
  Apache Spark Committer @jiangxb1987
- **Kris Mok**
  OpenJDK Committer @rednaxelafx

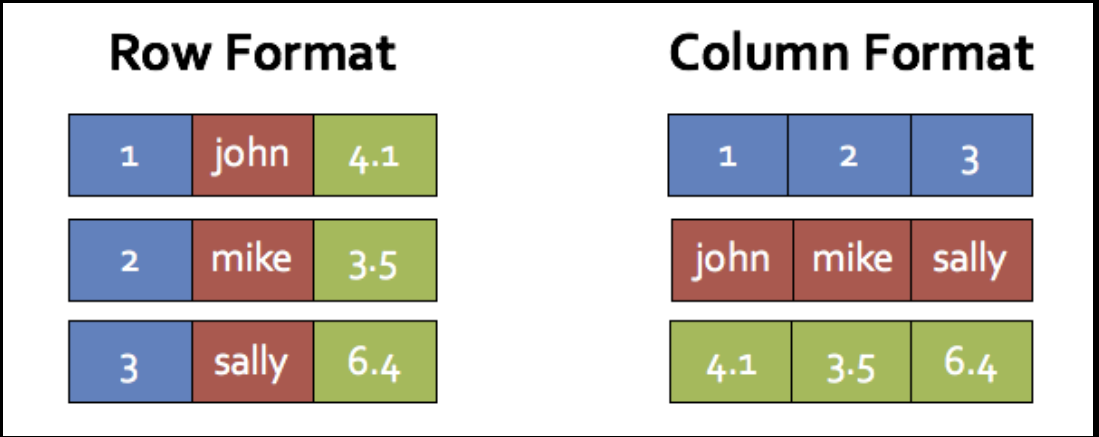
Agenda


Spark SQL Compiler
Front end
Physical Planning
Code Generation
RDDs (DAGs)
Spark SQL Runtime
UDF
PySpark
Vectorized Reader
Python/Pandas UDF
...
Spark Core
Memory Management
...

Vectorized Readers

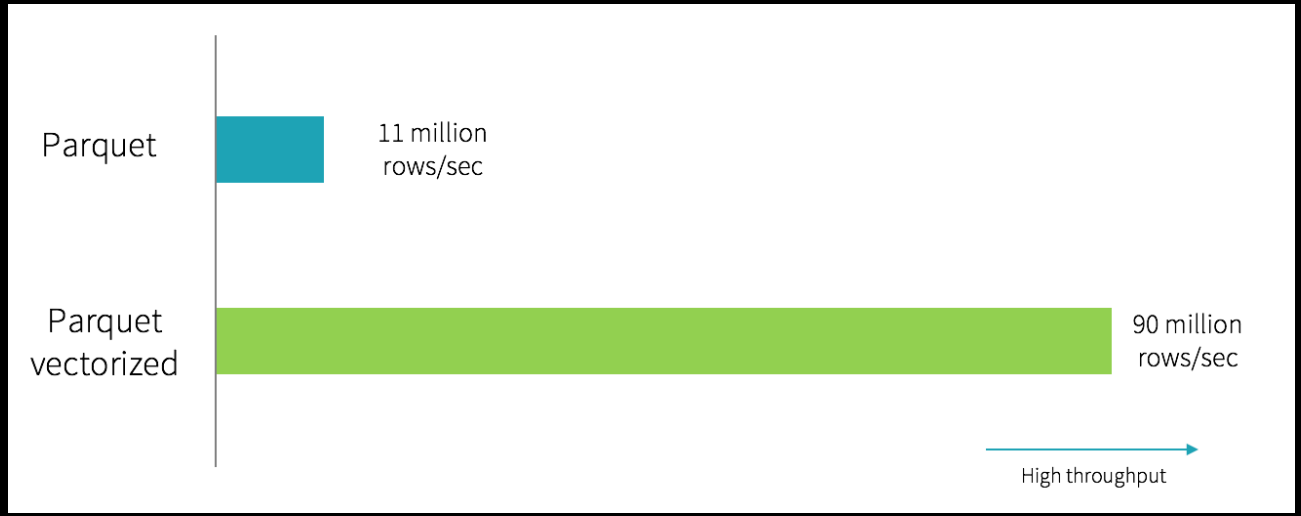
Read columnar format data as-is without converting to row format.

- Apache Parquet
- Apache ORC
- Apache Arrow
- ...



Vectorized Readers

Parquet vectorized reader is 9 times faster than the non-vectorized one.



See blog post

Vectorized Readers

Supported built-in data sources:

- Parquet
- ORC

Arrow is used for intermediate data in PySpark.

Implement DataSource

DataSource v2 API provides the way to implement your own vectorized reader.

- `PartitionReaderFactory`
  - `supportColumnarReads(...)` to return `true`
  - `createColumnarReader(...)` to return `PartitionReader[ColumnarBatch]`

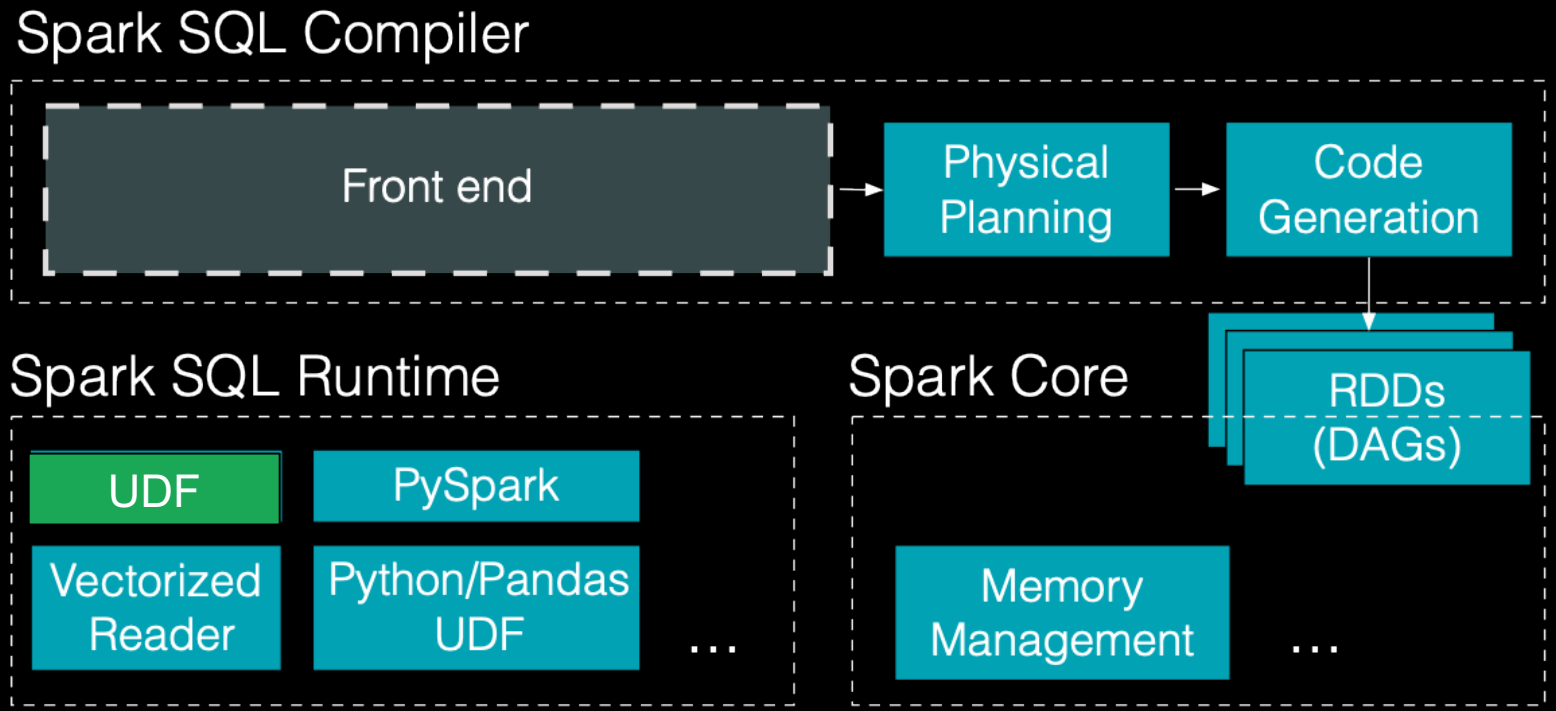
- [SPARK-25186] Stabilize Data Source V2 API

Delta Lake

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

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

Agenda


Spark SQL Compiler
Front end
Physical Planning
Code Generation
RDDs (DAGs)
Spark SQL Runtime
UDF
PySpark
Vectorized Reader
Python/Pandas UDF
...
Spark Core
Memory Management
...

What's behind `foo(x)` in Spark SQL?

What looks like a function call can be a lot of things:

- `upper(str)`: Built-in function
- `max(val)`: Aggregate function
- `max(val) over ...`: Window function
- `explode(arr)`: Generator
- `myudf(x)`: User-defined function
- `myudaf(x)`: User-defined aggregate function
- `transform(arr, x -> x + 1)`: Higher-order function
- `range(10)`: Table-value function

Functions in Spark SQL

	Builtin Scalar Function	Java / Scala UDF	Python UDF (*)	Aggregate / Window	Higher-order Function
Scope	1 Row	1 Row	1 Row	Whole table	1 Row
Data Feed	Scalar expressions	Scalar expressions	Batch of data	Scalar expressions + aggregate buffer	Expression of complex type
Process	Same JVM	Same JVM	Python Worker process	Same JVM	Same JVM
Impl. Level	Expression	Expression	Physical Operator	Physical Operator	Expression
Data Type	Internal	External	External	Internal	Internal

(*): and all other non-Java user-defined functions

UDF execution

User Defined Functions:

- Java/Scala UDFs
- Hive UDFs
  - when Hive support enabled

Also we have:

- Python/Pandas UDFs
  - will talk later in PySpark execution

Java/Scala UDFs

- UDF: User Defined Function
  - Java/Scala lambdas or method references can be used.

- UDAF: User Defined Aggregate Function
  - Need to implement `UserDefinedAggregateFunction`.

UDAF

Implement `UserDefinedAggregateFunction`

- `def initialize(...)`
- `def update(...)`
- `def merge(...)`
- `def evaluate(...)`
- `...`

Hive UDFs

Available when Hive support enabled.

- Register using create function command
- Use in HiveQL

Hive UDFs

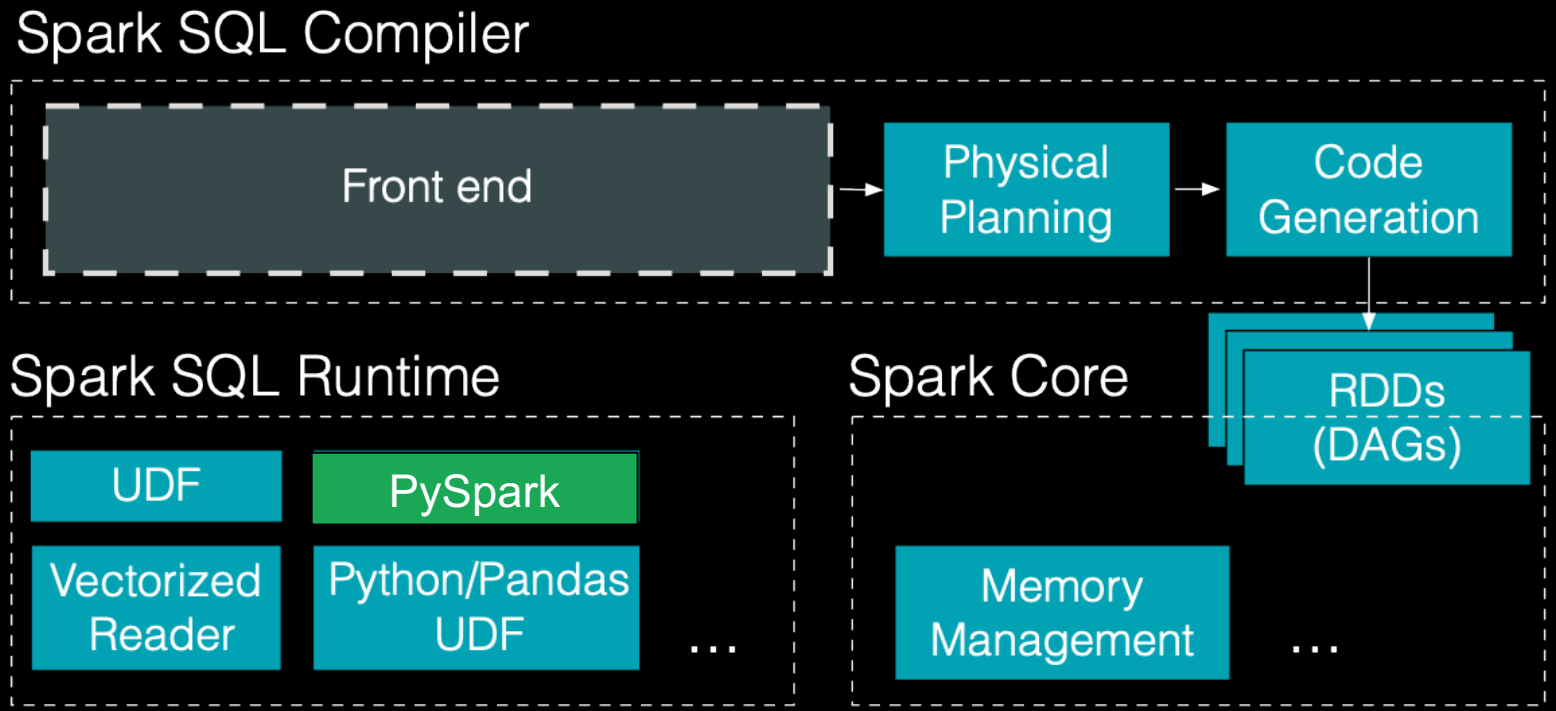
Provides wrapper expressions for each UDF type:

- HiveSimpleUDF: UDF
- HiveGenericUDF: GenericUDF
- HiveUDAFFunction: UDAF
- HiveGenericUDTF: GenericUDTF

UDF execution

1. Before invoking UDFs, convert arguments from internal data format to objects suitable for each UDF types.
   - Java/Scala UDF: Java/Scala objects
   - Hive UDF: ObjectInspector
2. Invoke the UDF.
3. After invocation, convert the returned values back to internal data format.

Agenda


Spark SQL Compiler
Front end
Physical Planning
Code Generation
RDDs (DAGs)
Spark SQL Runtime
UDF
PySpark
Vectorized Reader
Python/Pandas UDF
...
Spark Core
Memory Management
...

PySpark

PySpark is a set of Python bindings for Spark APIs.

- RDD
- DataFrame
- other libraries based on RDDs, DataFrames.
  - MLlib, Structured Streaming, ...

Also, SparkR: R bindings for Spark APIs

PySpark

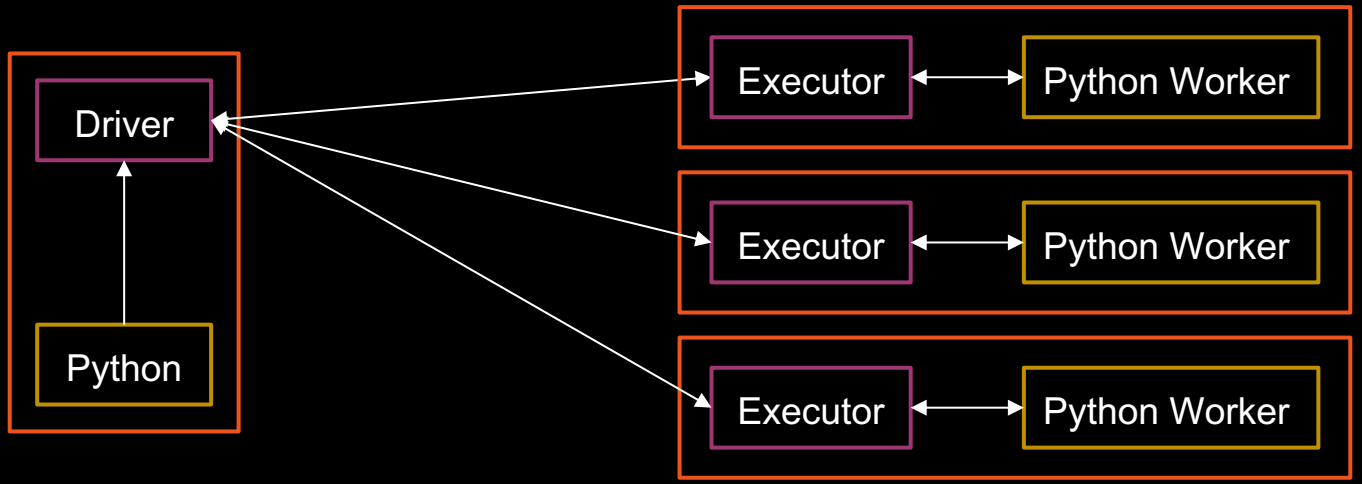
RDD vs. DataFrame:

- RDD invokes Python functions on Python worker
- DataFrame just constructs queries, and executes it on the JVM.
  - except for Python/Pandas UDFs

PySpark execution

Python script drives Spark on JVM via Py4J.

Executors run Python worker.



PySpark and Pandas

Ease of interop: PySpark can convert data between PySpark DataFrame and Pandas DataFrame.

- `pdf = df.toPandas()`
- `df = spark.createDataFrame(pdf)`

Note: `df.toPandas()` triggers the execution of the PySpark DataFrame, similar to `df.collect()`

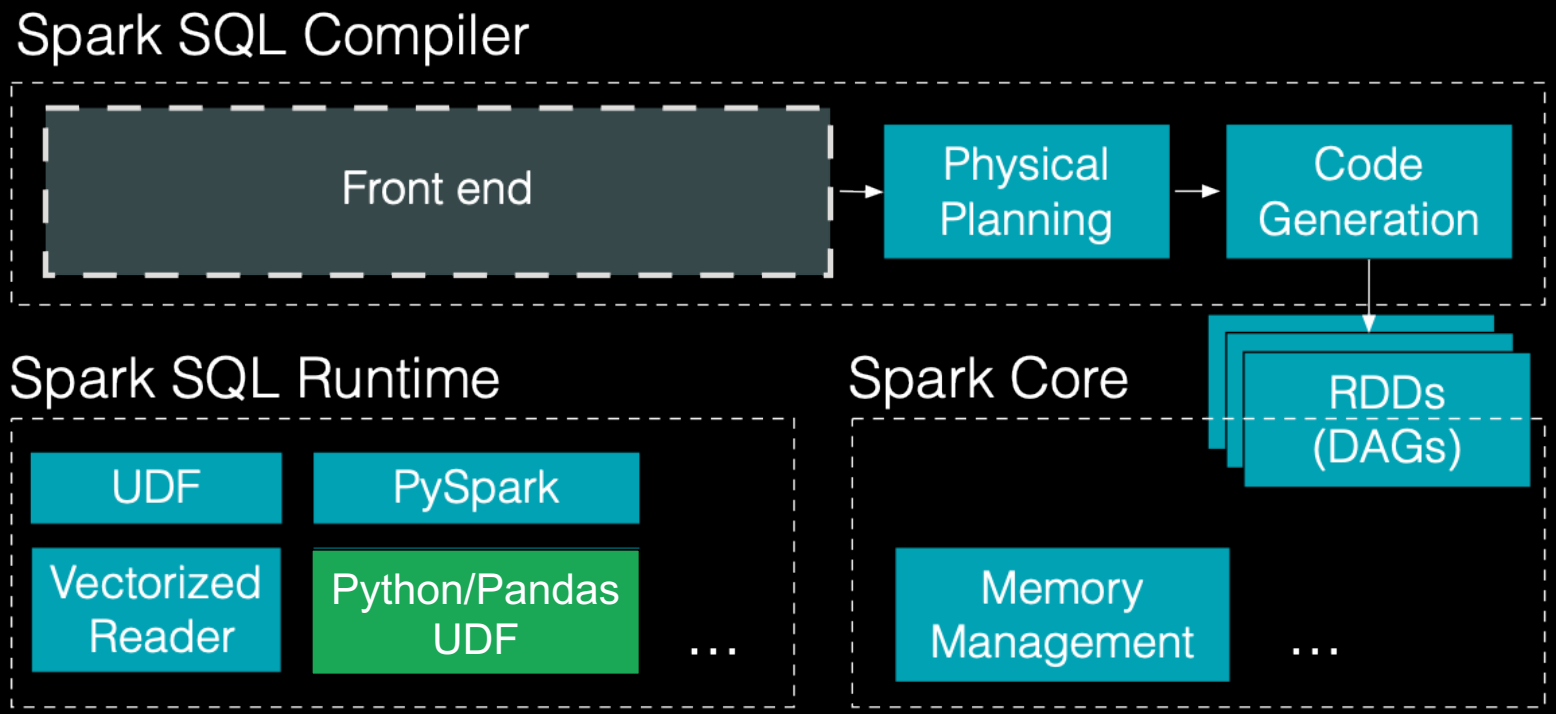
PySpark and Pandas (cont'd)

New way of interop: Koalas brings the Pandas API to Apache Spark

```
import databricks.koalas as ks

import pandas as pd
pdf = pd.DataFrame({'x':range(3), 'y':['a','b','b'], 'z':['a','b','b']})

# Create a Koalas DataFrame from pandas DataFrame
df = ks.from_pandas(pdf)

# Rename the columns
df.columns = ['x', 'y', 'z1']

# Do some operations in place:
df['x2'] = df.x * df.x
```

https://github.com/databricks/koalas

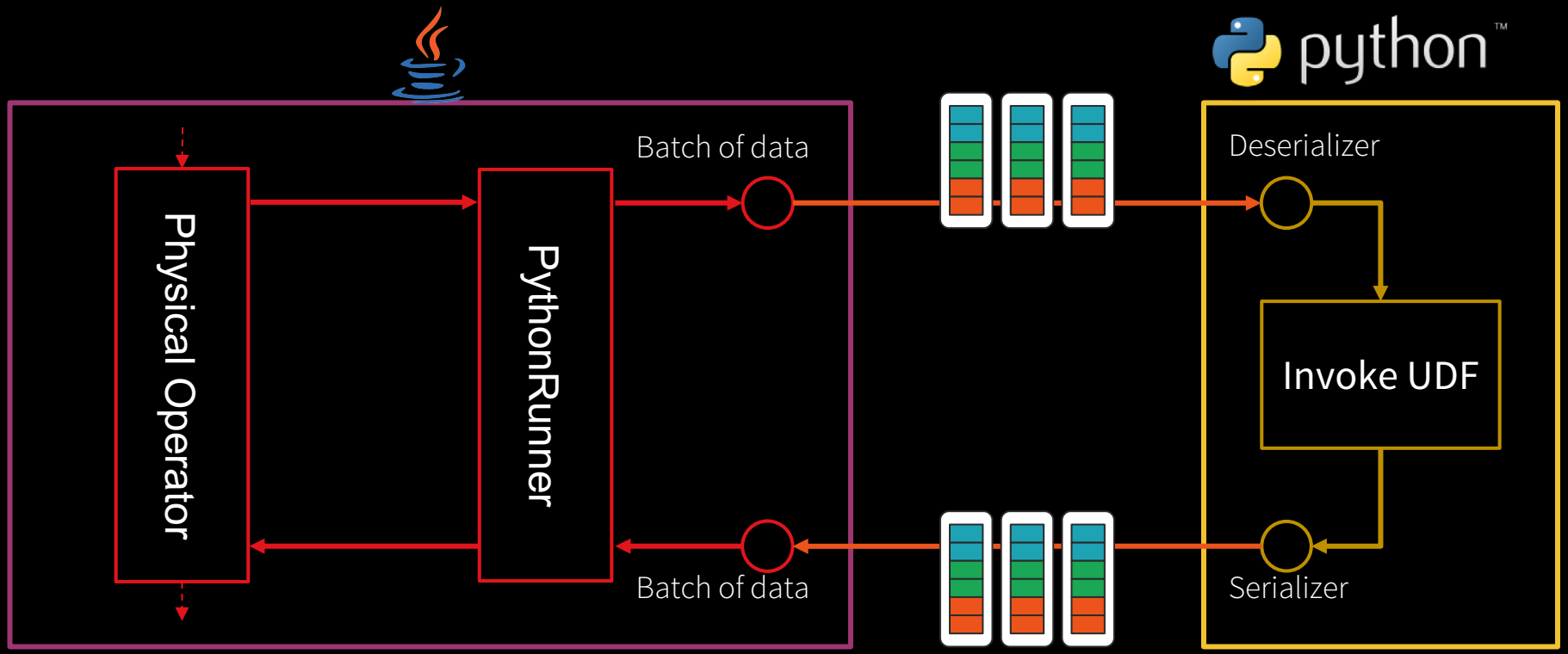
Agenda


Spark SQL Compiler
Front end
Physical Planning
Code Generation
RDDs (DAGs)
Spark SQL Runtime
UDF
PySpark
Vectorized Reader
Python/Pandas UDF
...
Spark Core
Memory Management
...

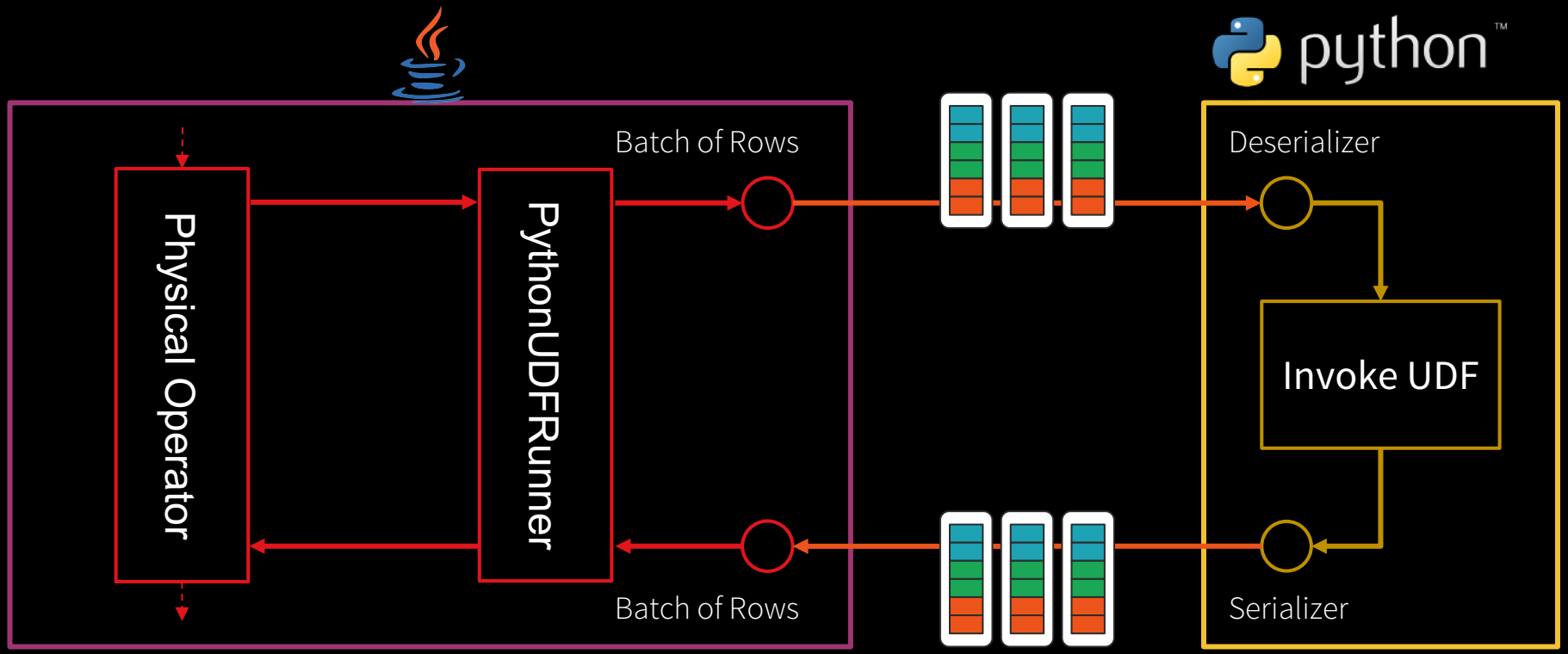
Python UDF and Pandas UDF

```
@udf('double')
def plus_one(v):
      return v + 1

@pandas_udf('double', PandasUDFType.SCALAR)
def pandas_plus_one(vs):
      return vs + 1
```

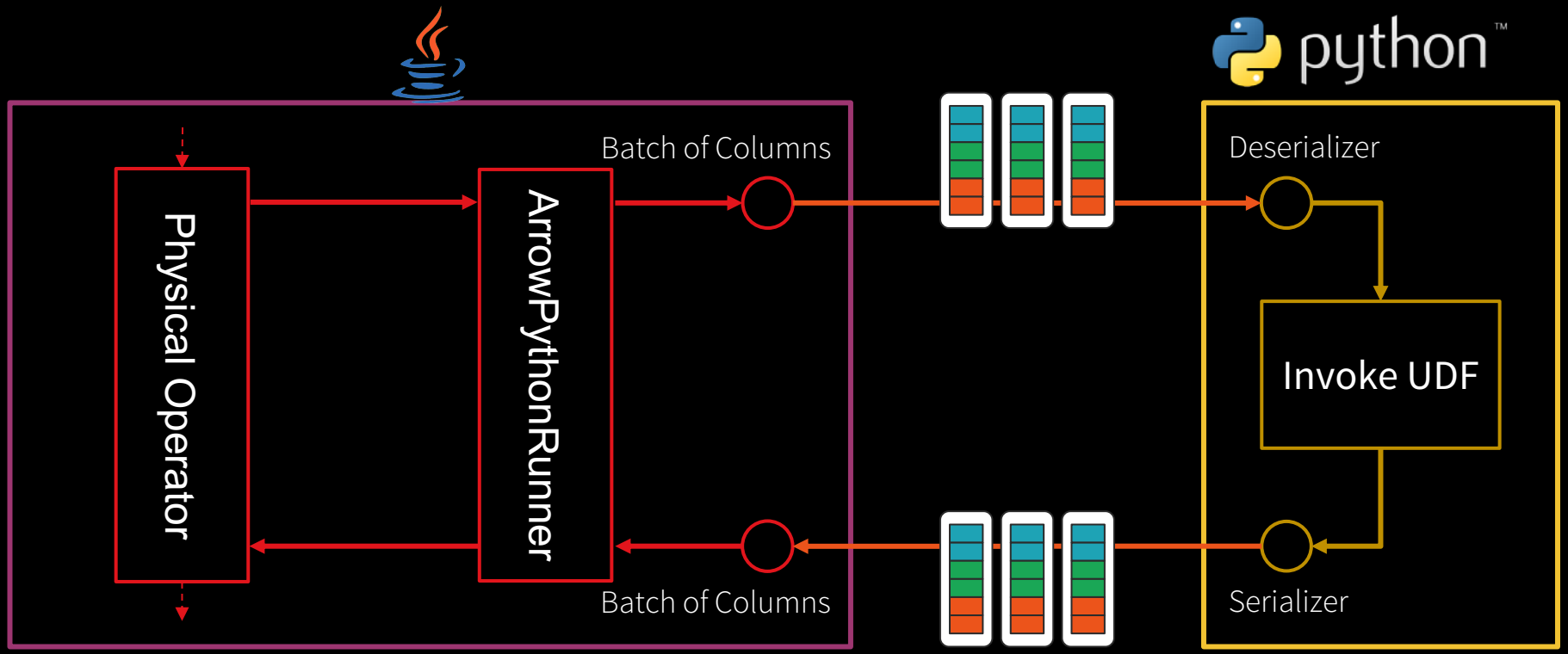
Python/Pandas UDF execution



Python UDF execution



Pandas UDF execution



Python/Pandas UDFs

Python UDF

- Serialize/Deserialize data with Pickle
- Fetch data in blocks, but invoke UDF row by row

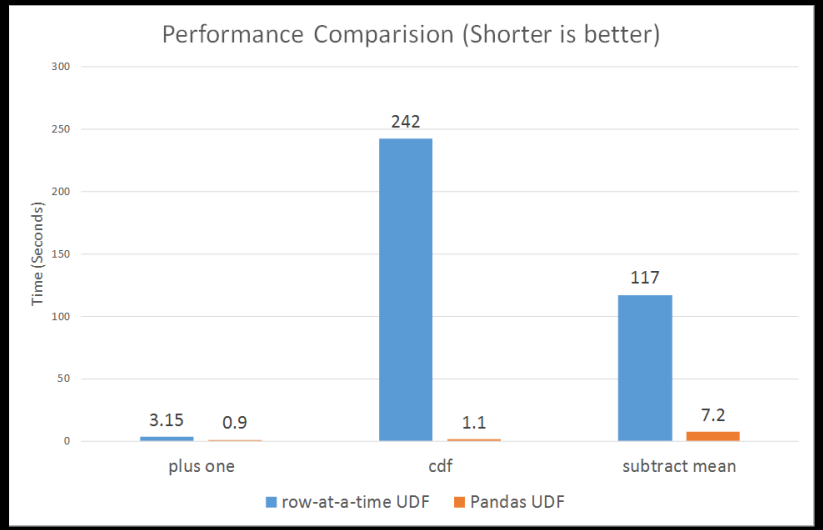
Pandas UDF

- Serialize/Deserialize data with Arrow
- Fetch data in blocks, and invoke UDF block by block

Python/Pandas UDFs

Pandas UDF perform much better than row-at-a-time Python UDFs.

- 3x to over 100x



See blog post

Further Reading

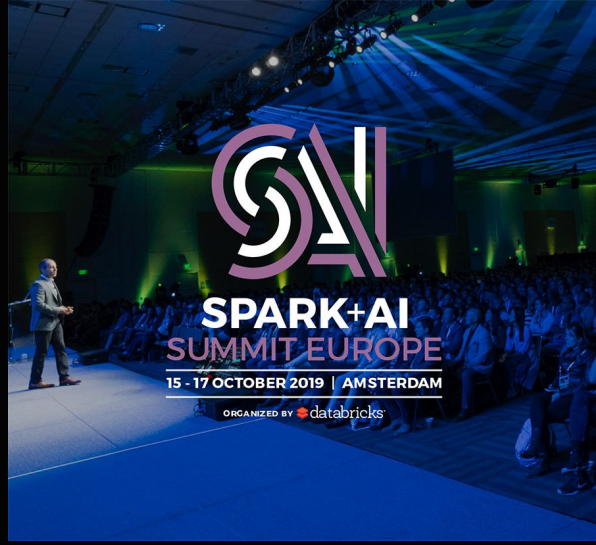
This Spark+AI Summit:

- Understanding Query Plans and Spark

Previous Spark Summits:

- A Deep Dive into Spark SQL's Catalyst Optimizer
- Deep Dive into Project Tungsten: Bringing Spark Closer to Bare Metal
- Improving Python and Spark Performance and Interoperability with Apache Arrow

Thank you



Maryann Xue (maryann.xue@databricks.com)
Xingbo Jiang (xingbo.jiang@databricks.com)
Kris Mok (kris.mok@databricks.com)

databricks