DATA+AI
SUMMIT EUROPE
FORMERLY SPARK+AI SUMMIT

Building a SIMD supported vectorized native engine for Spark SQL

Chendi Xue(chendi.xue@intel.com), Software Engineer
Yuan Zhou(yuan.zhou@intel.com), Software Engineer
Intel Corp

Agenda

- Native SQL Engine Introduction
- Native SQL Engine Design
  - Columnar Data Source
  - Columnar Shuffle
  - Columnar Compute
  - Memory Management
- Summary

Motivations for Native SQL Engine

- Issue of current Spark SQL Engine:
  - Internal row based, difficult to use SIMD optimizations
  - High GC Overhead under low memory
  - JIT code quality relies on JVM, hard to tune
  - High overhead of integration with other native library

Proposed Solution

Issue of current Spark SQL Engine:

- Internal row based, not possible to use SIMD

→ Columnar-based Arrow Format

- High GC Overhead under low memory

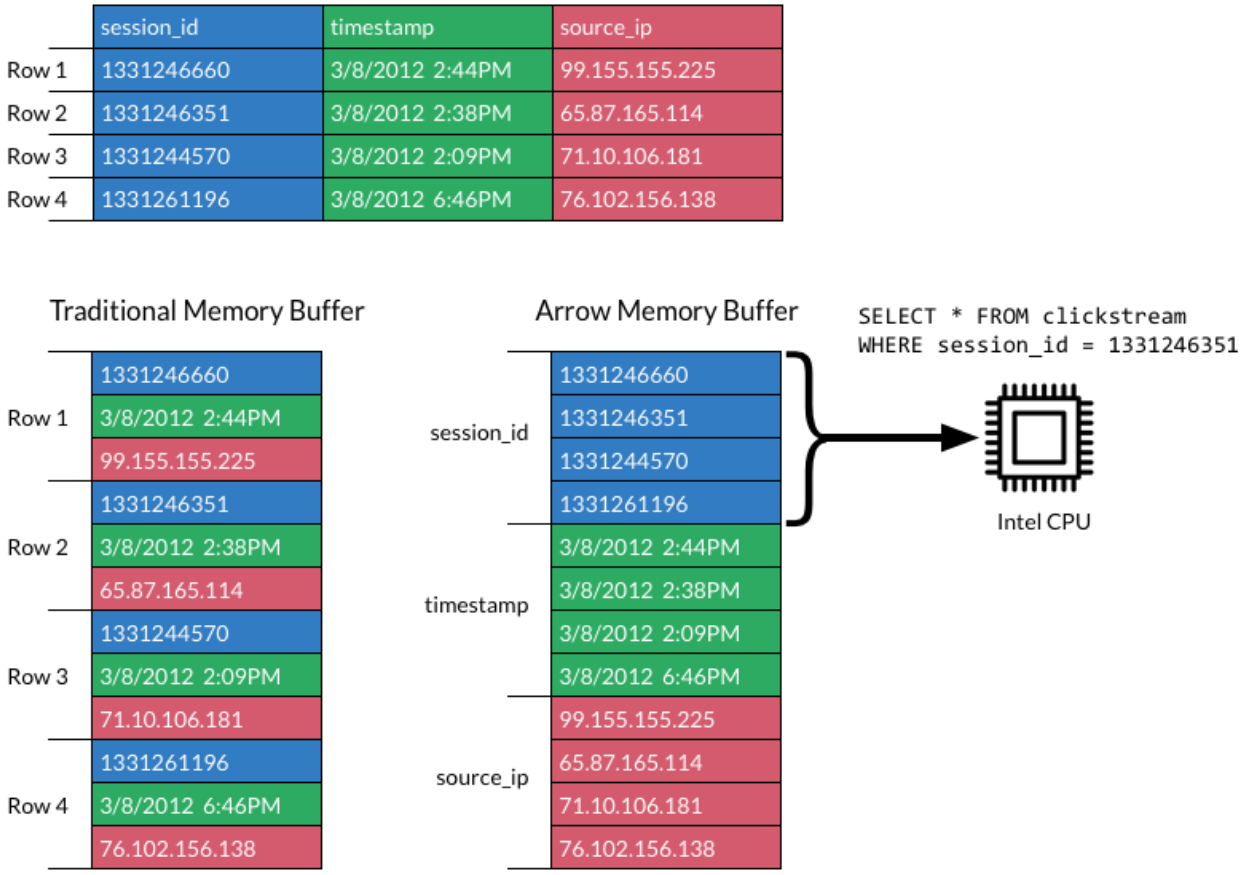
→ native codes for core compute instead of java

- JIT code quality relies on JVM, hard to tune

→ cpp / llvm / assembly code generation

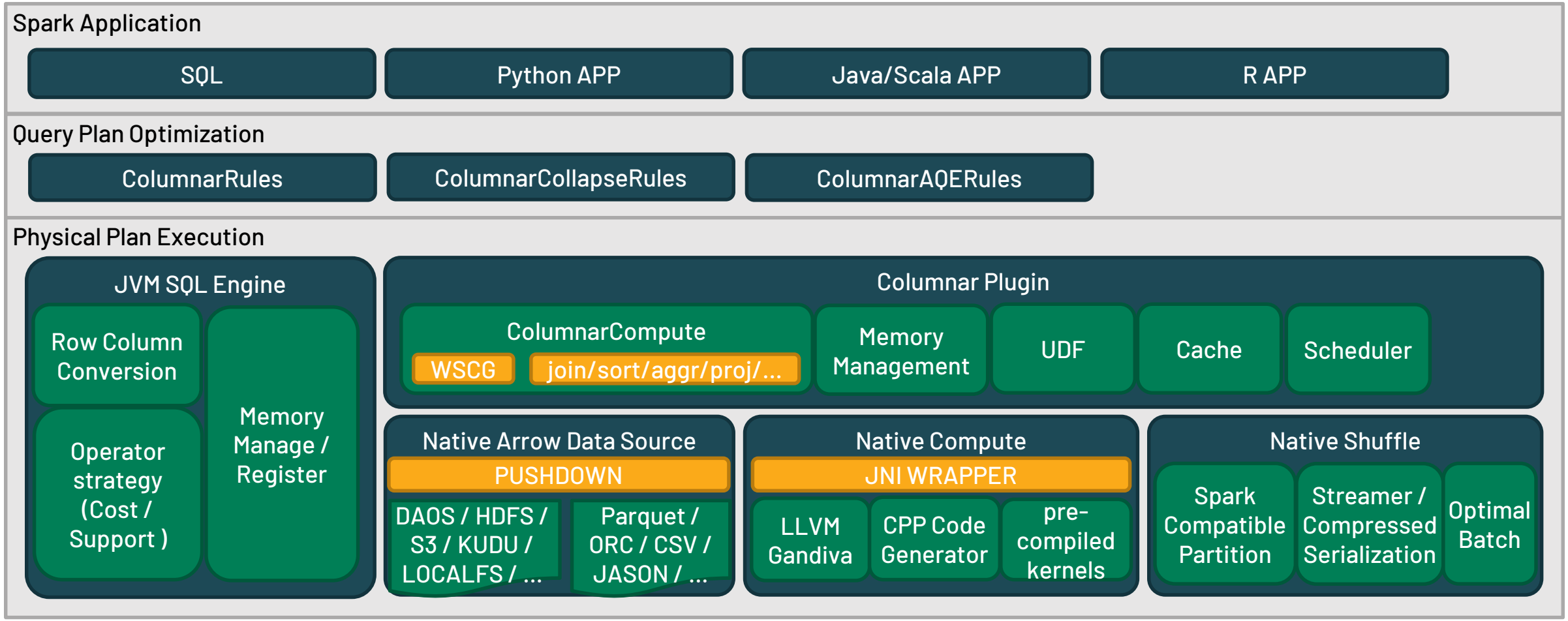
- High overhead of integration with other native library

→ Lightweighted JNI based call framework



Source: https://www.slideshare.net/dremio/apache-arrow-an-overview

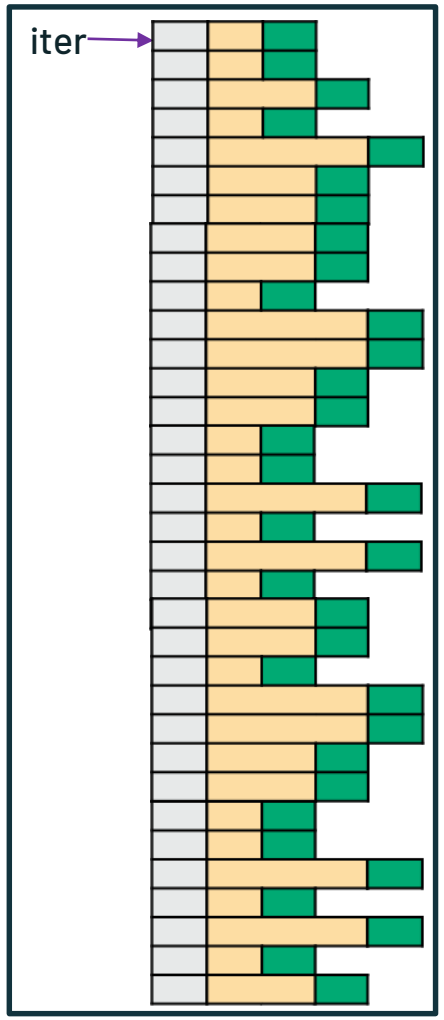
Native SQL Engine Layers



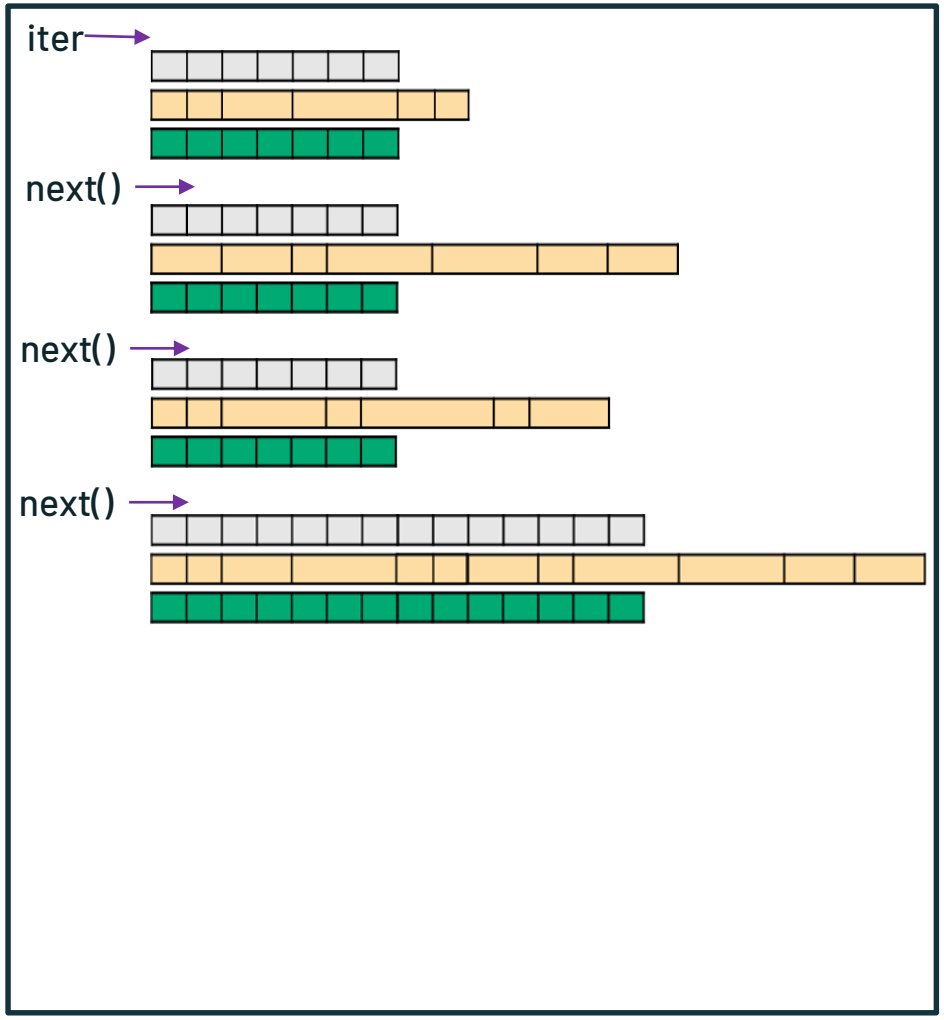
- A standard columnar data format as basic data format
- Data keeps on off-heap, data operations offload to highly optimized native library

Data Format

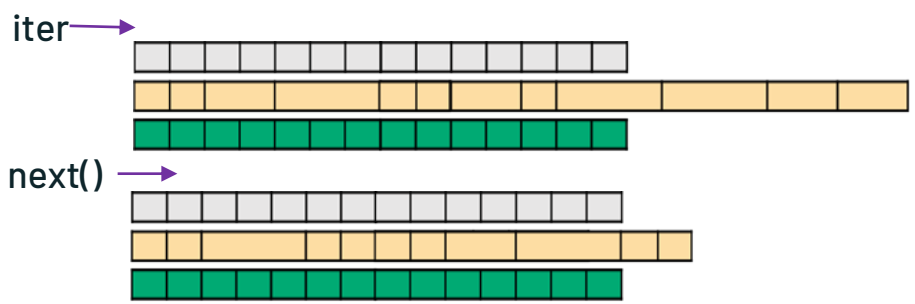
Row RDD



Column RDD

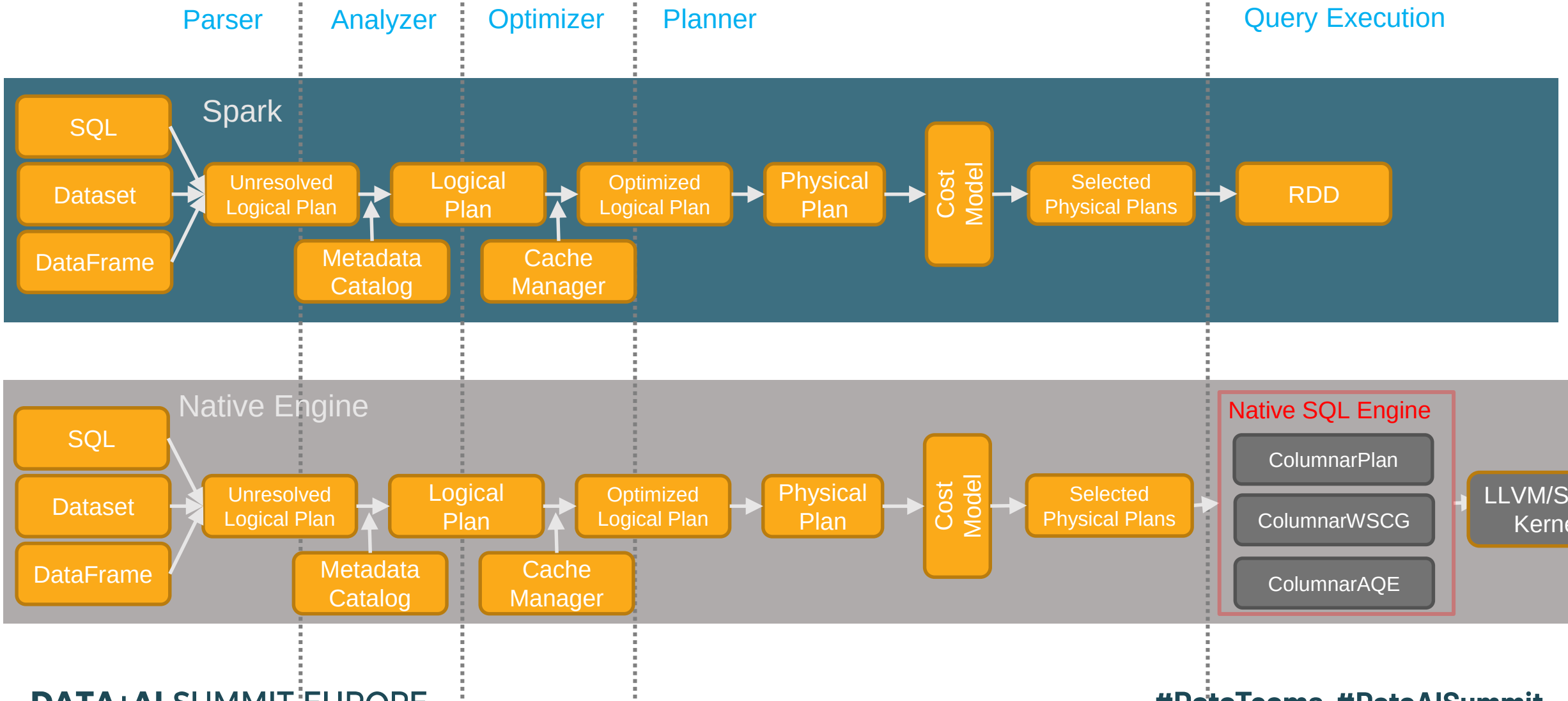


Optimal Column RDD

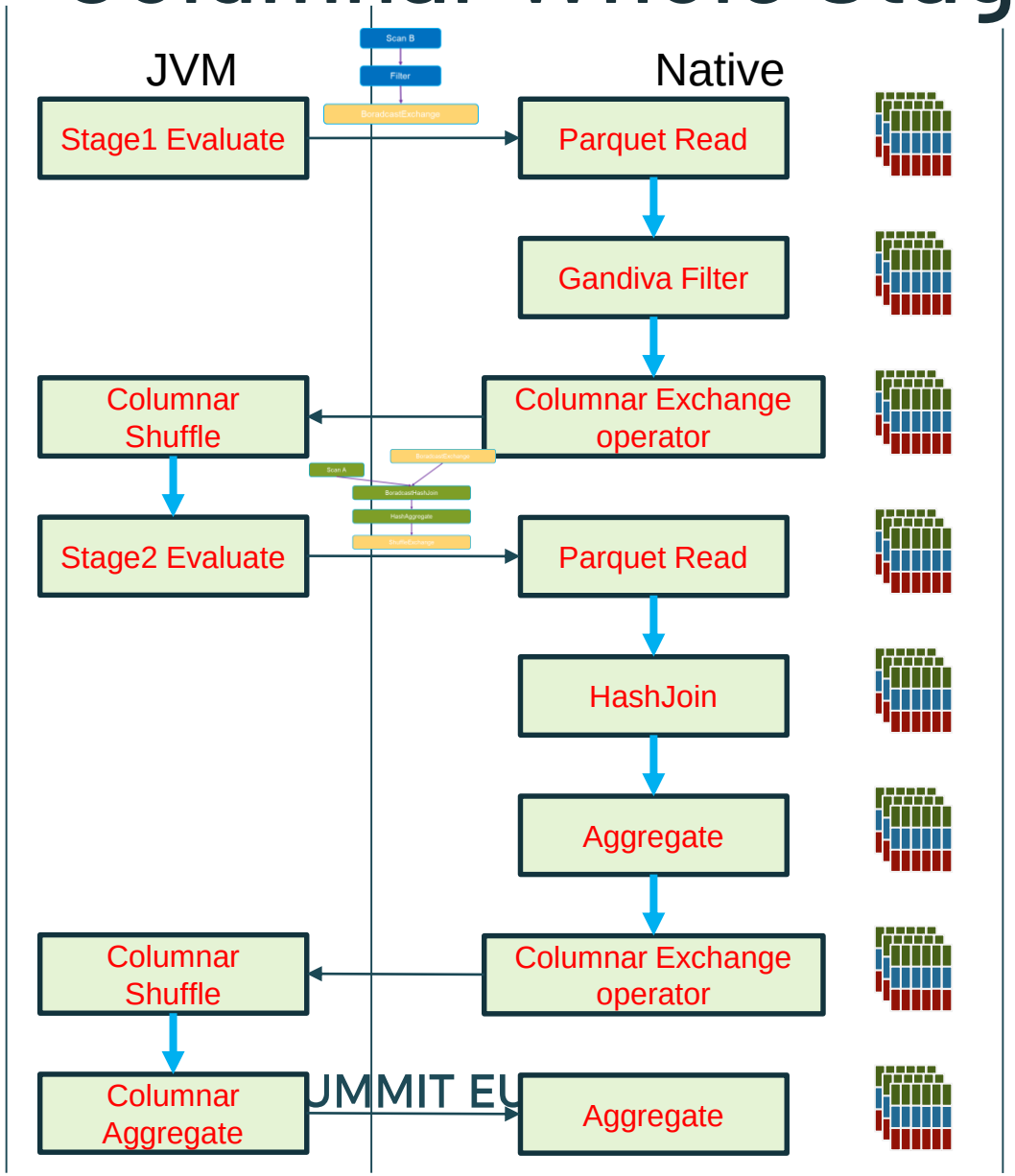


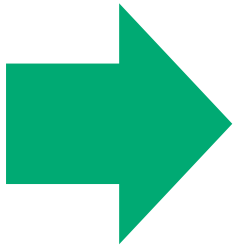
- Auto split and coalesce
- Tunable batch size

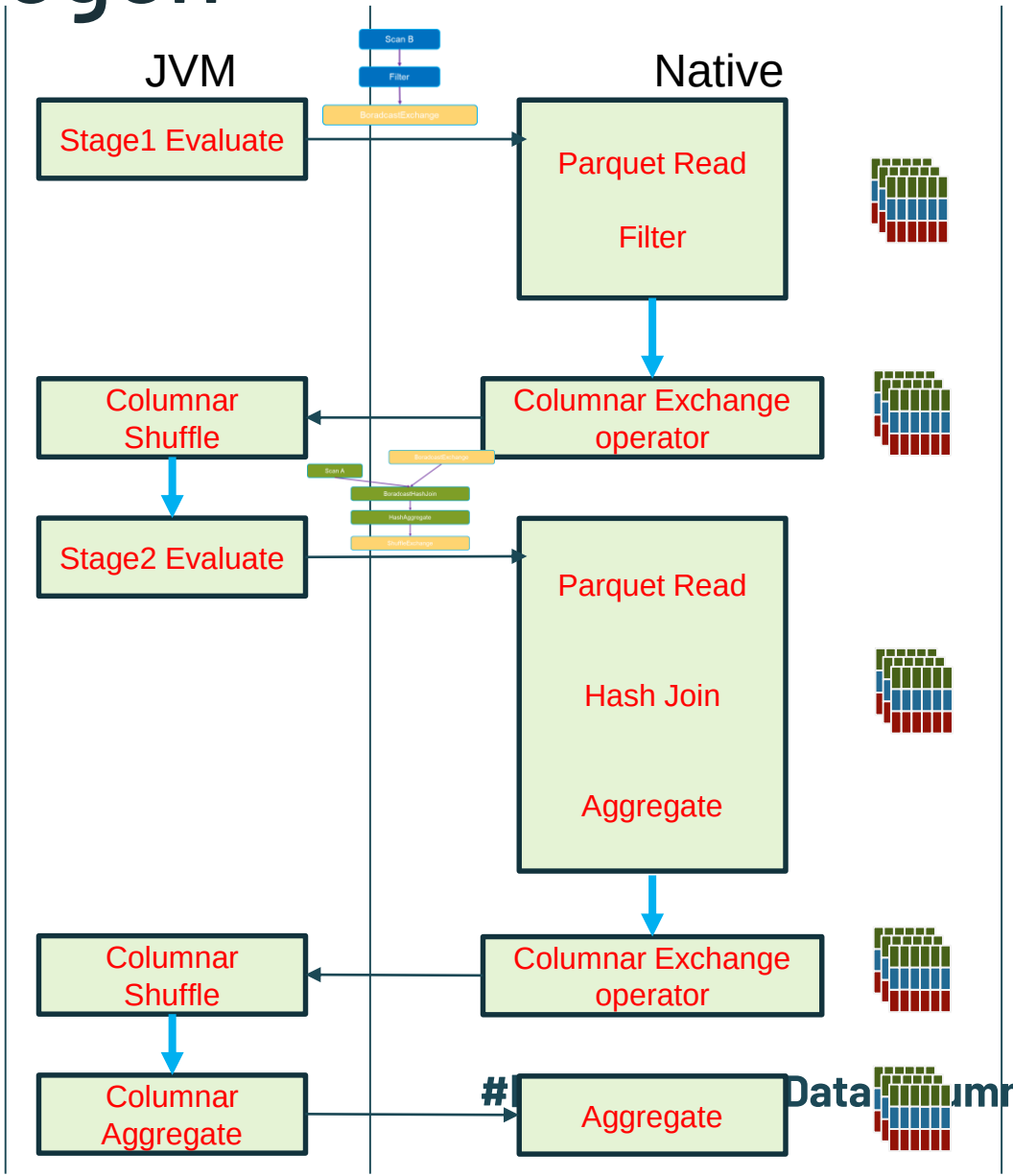
Columnar Spark Plan Rules



Columnar Whole Stage Codegen





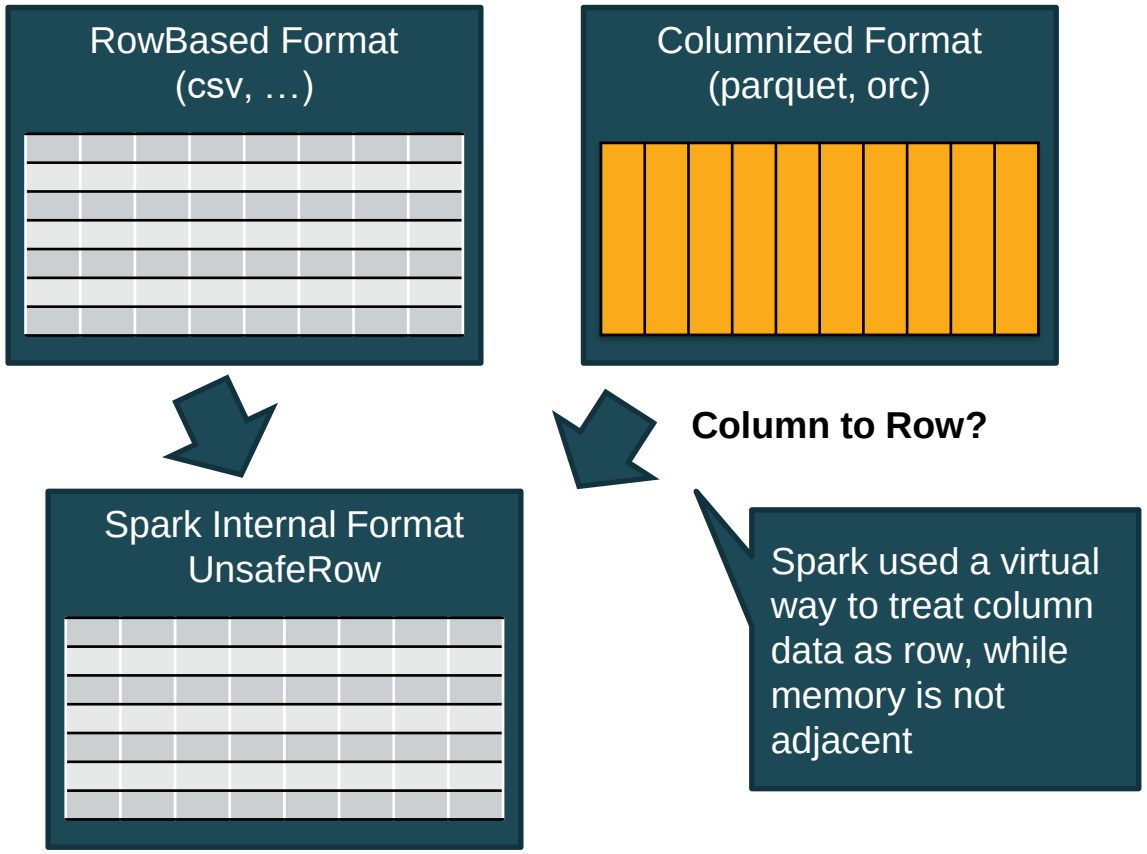


Native SQL Engine Design

- Columnar Data Source
- Columnar Shuffle
- Columnar Compute
- Memory Management

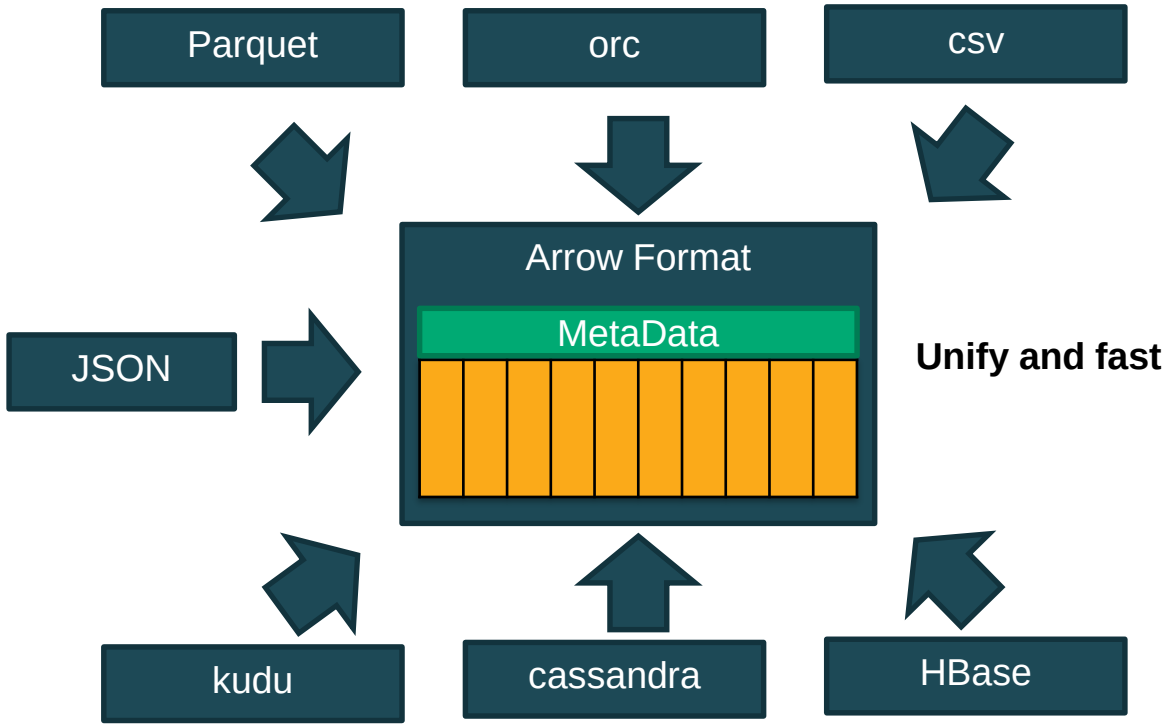
Columnar Data Source

Row-based Data Source

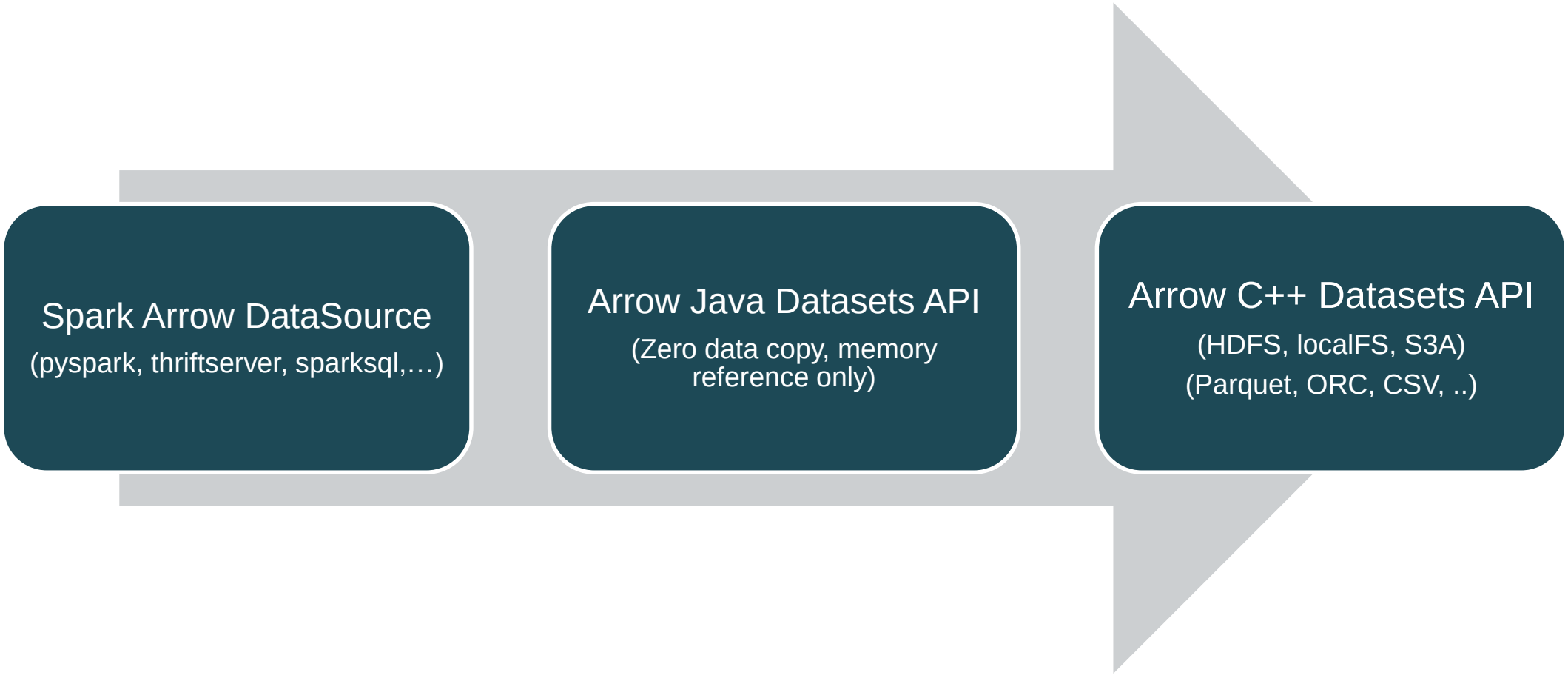


V.S.

Arrow based Data Source

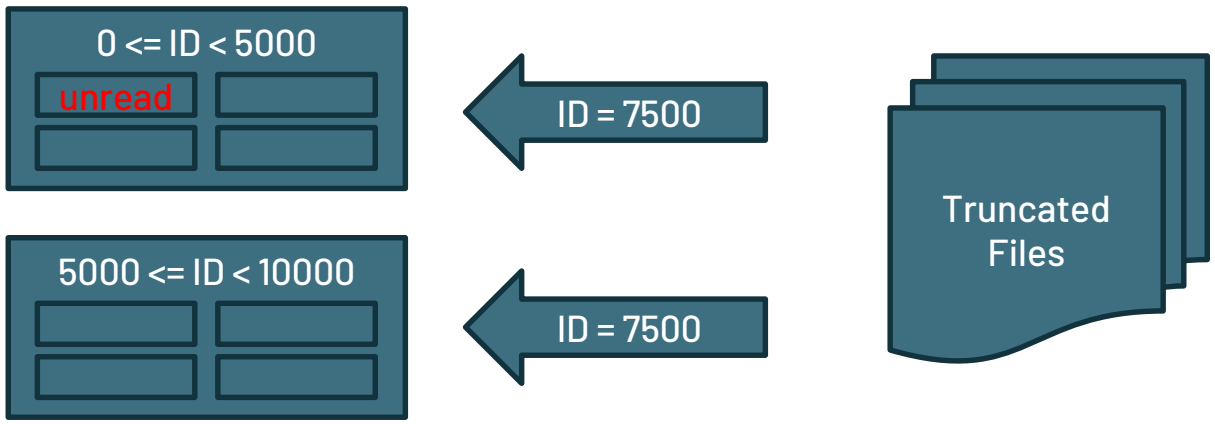


Columnar Data Source



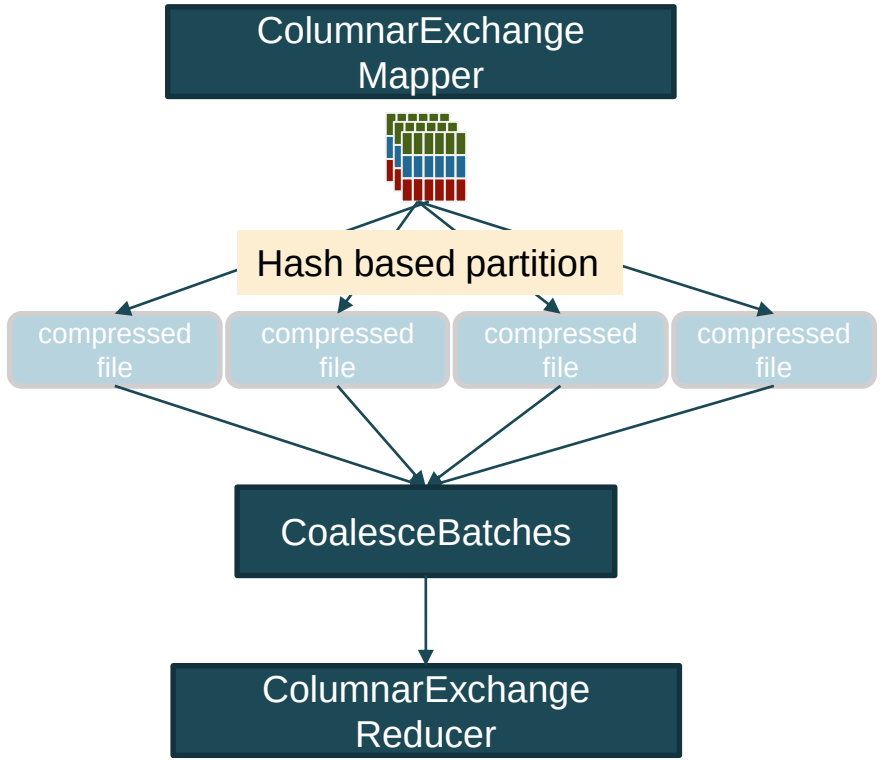
Columnar Data Source

- Features
  - Fast / Parallel / Auth supported Native Libraries for HDFS / S3A / Local FS
  - PushDown supported Pre-executed statistics/metadata filters
  - Partitioned File and DPP enabled



Columnar Shuffle

- Hash-based partitioning(split) with LLVM optimized execution
- Ser/de-ser based on arrow record batch
- Efficient data compression for different data format
- Coalesce batches during shuffle read
- Supports Adaptive Query Execution(AQE)



Supported SQL Operators Overview

Operators
WindowExec
UnionExec
ExpandExec
SortExec
ScalarSubquery
ProjectExec
ShuffledHashJoin
BroadcastJoinExec
FilterExec
ShuffleExchangeExec
BroadcastExchangeExec
datasources.v2.BatchScanExec
datasources.v1.FileScanExec
HashAggregateExec
.....

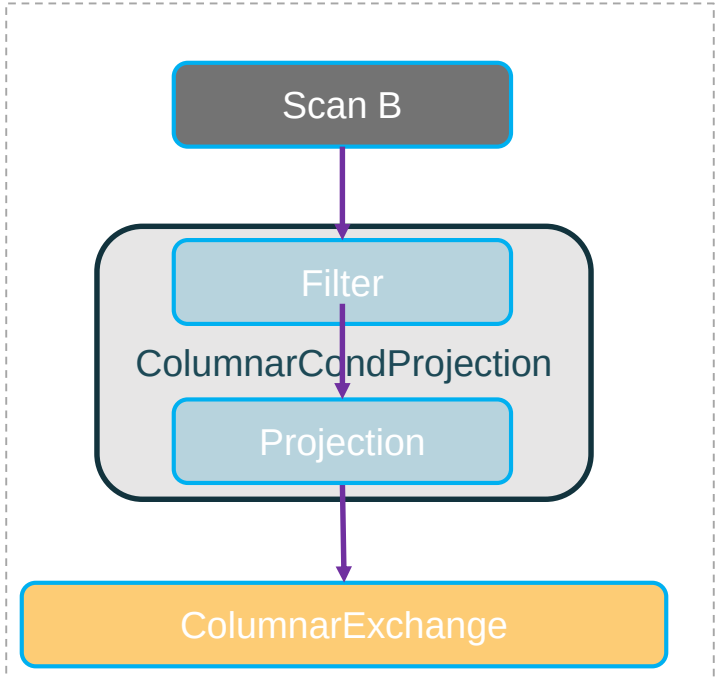
Expression
NormalizeNaNAndZero
Subtract
Substring
ShiftRight
Round
PromotePrecision
Multiply
Literal
LessThanOrEqual
LessThan
KnownFloatingPointNormalized
IsNull
And
Add
....

Expression
IsNotNull
GreaterThanOrEqual
GreaterThan
EqualTo
ExtractYear
Divide
Concat
Coalesce
CheckOverflow
Cast
CaseWhen
BitwiseAnd
AttributeReference
Alias
....

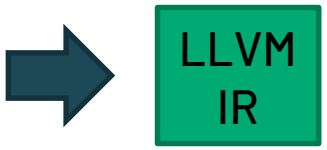
Automatically fallback to row-based execution if there are unsupported operators/expressions

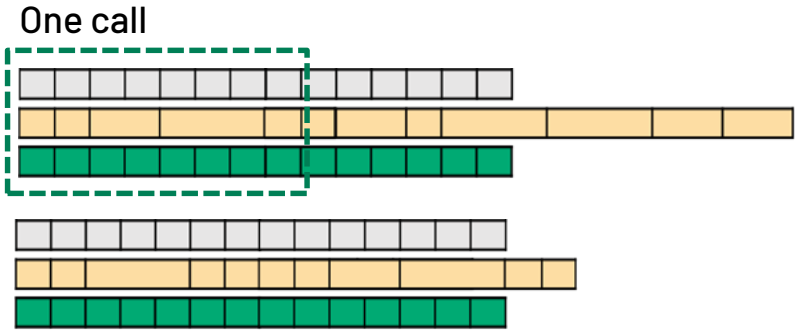
Columnar Projector & Filter

- LLVM IR based execution w/ AVX optimized code
- Based on Arrow Gandiva, extended with more functions
- Combined filter & projection into CondProjector
- ColumnarBatch based execution



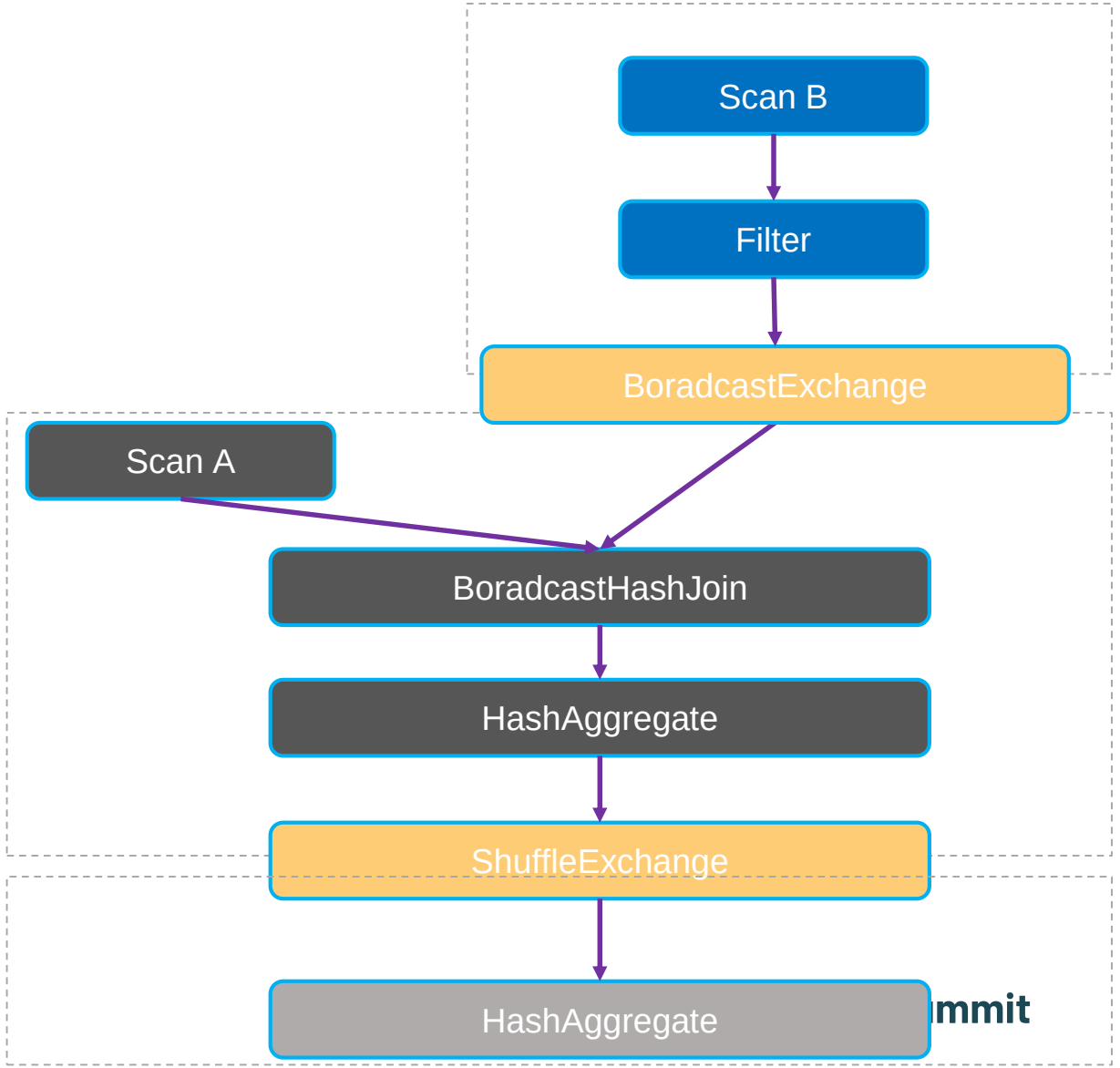
Example:
If (Field_A + Field_B + Field_C) as Field_new > Field_D
output [Field_new, Field_A, Field_B, Field_C, Field_D]



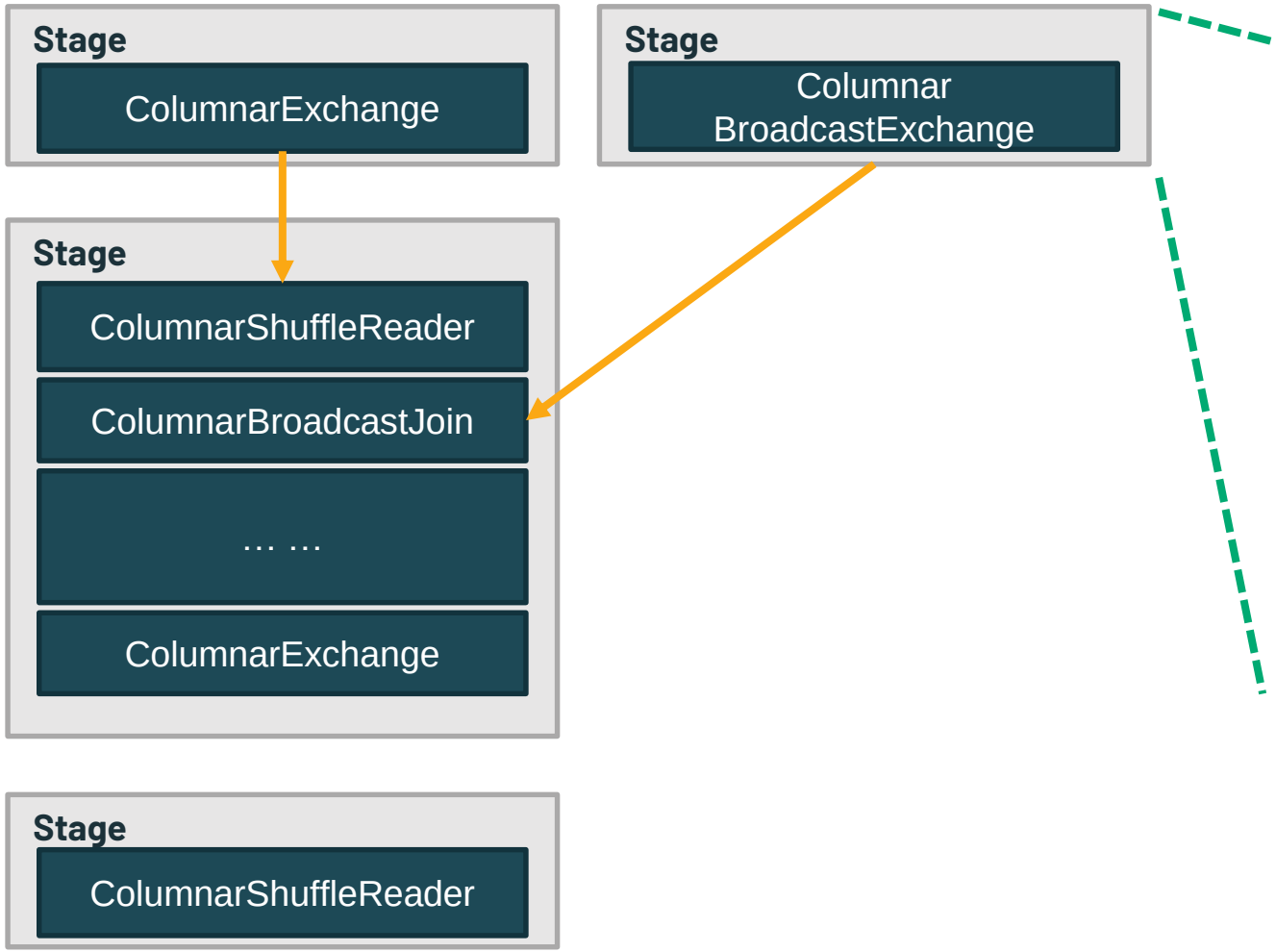


Native Hashmap

- Faster Hashmap building and lookup w/ AVX optimizations
- Compatible with Spark's *murmurhash* if configured
- Performance benefits for HashAggregation and HashJoins



ColumnarBroadcastHashJoin

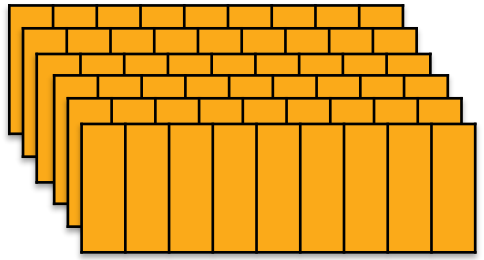


Broadcast data consists of

1. HashMap (key -> indices)
2. Arrow RecordBatch

hash	payload
0x8198	<0, 0>, key1
0x7723	<0, 1>,key2
…	…
0x6388	<10240, 1076> Key1076
0x9944	<10240, 1077> Key1077
0x8761	<10240, 1078> key1078

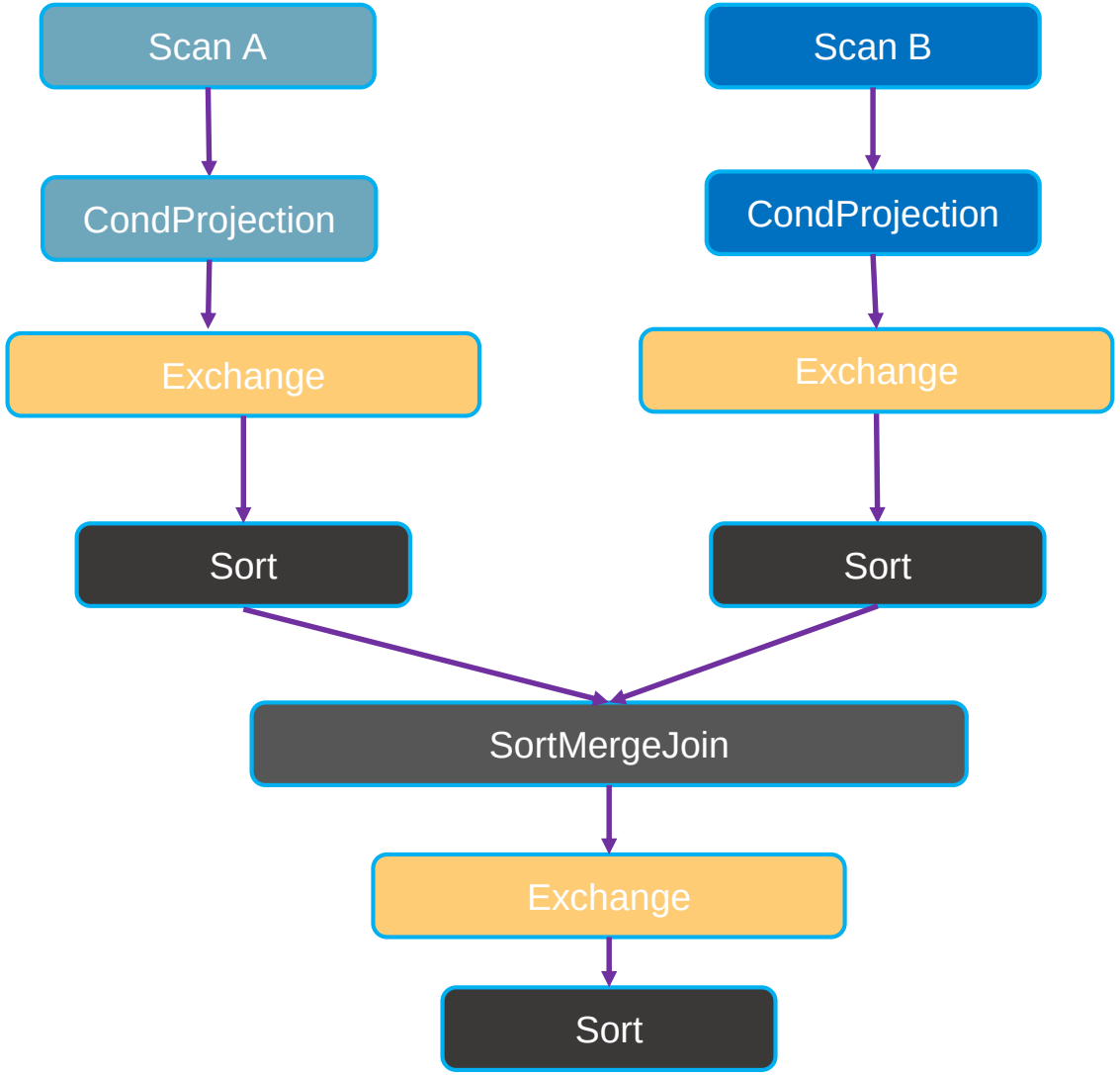
+



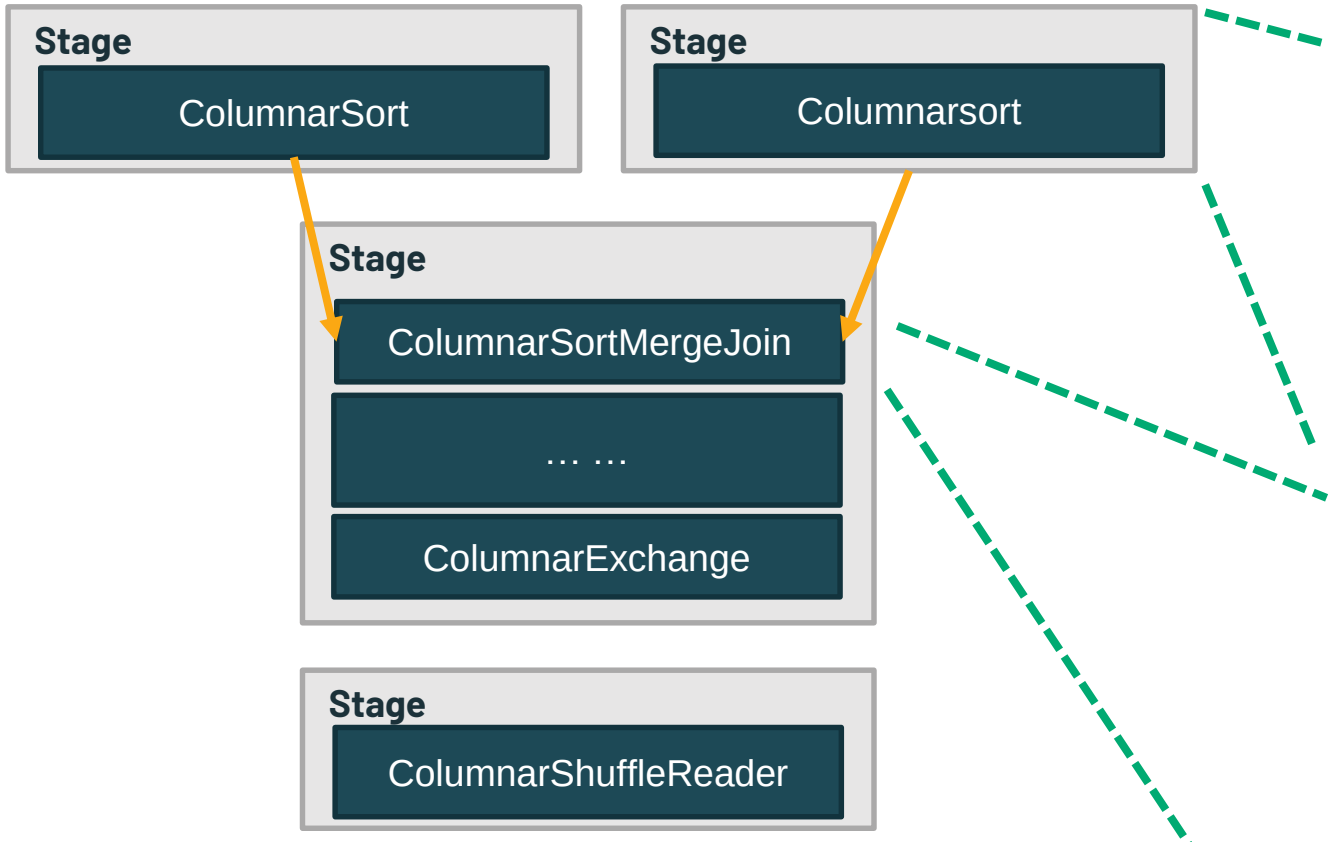
using ColumnarBroadcastExchange,
data size is reduced by 80%

Native Sort

- Faster sort implementation w/ AVX optimizations
- Most powerful algorithms used for different data structures
- Performance benefits for sort and sort merge joins



ColumnarSort and ColumnarSortMergeJoin

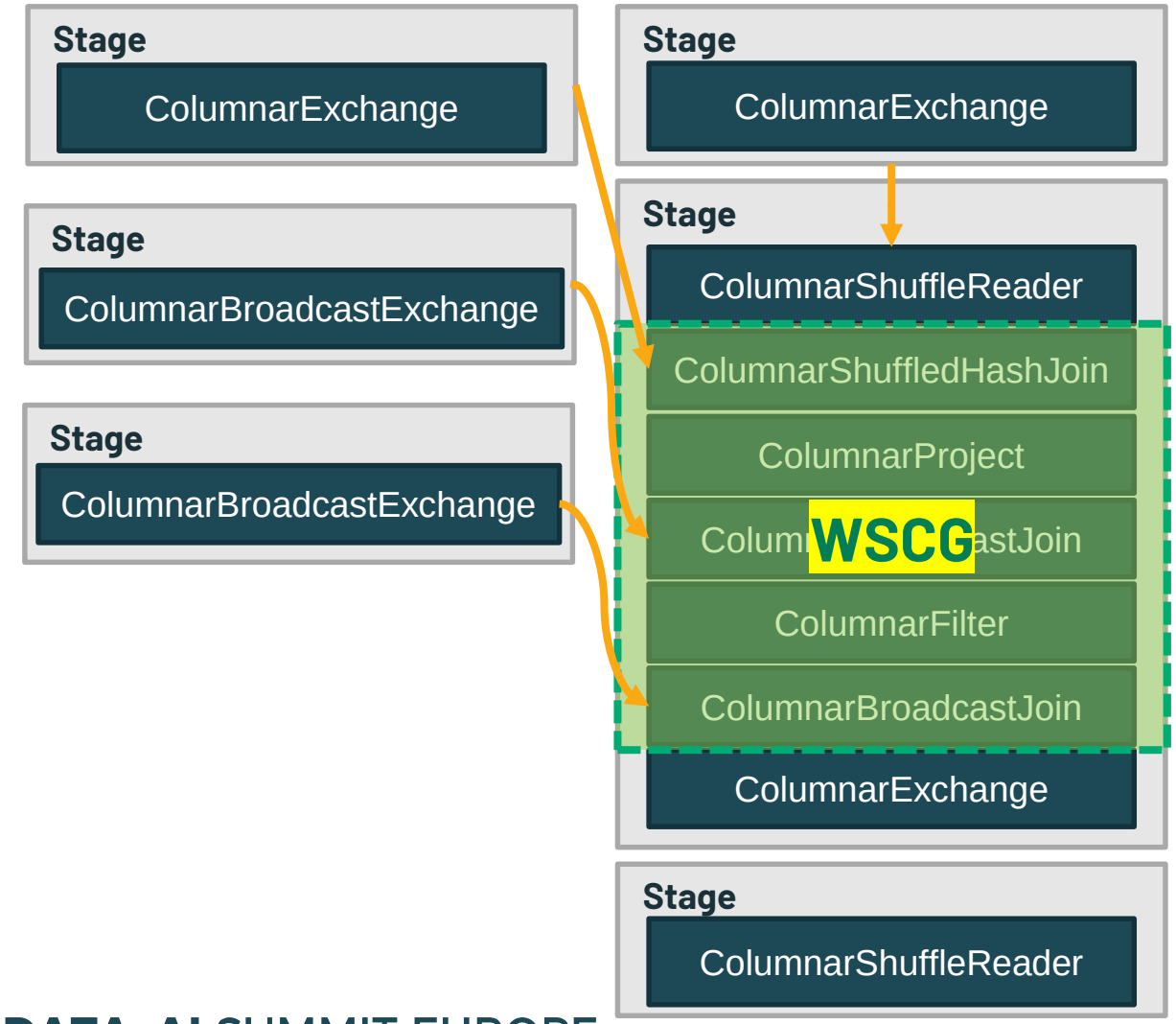


3 implementation of sort

1. One column data sort(used by semi or anti)
   -> inplace radix sort (AVX optimizable)
2. One column of key with multiple columns of payload
   -> radix sort to indices (AVX optimizable)
   -> lazy materialization
3. Multiple columns of keys with payloads
   -> codegened quick sort to indices(AVX optimizable)
   -> lazy materialization

```
AVX optimized project {
  chain key_0 in left table #1
  apply project
  compare  key_0 and key_1 in right table #2
  apply condition
materialize one line to arrow
}
```

Columnar WholeStageCodeGen



Code generated Cpp codes

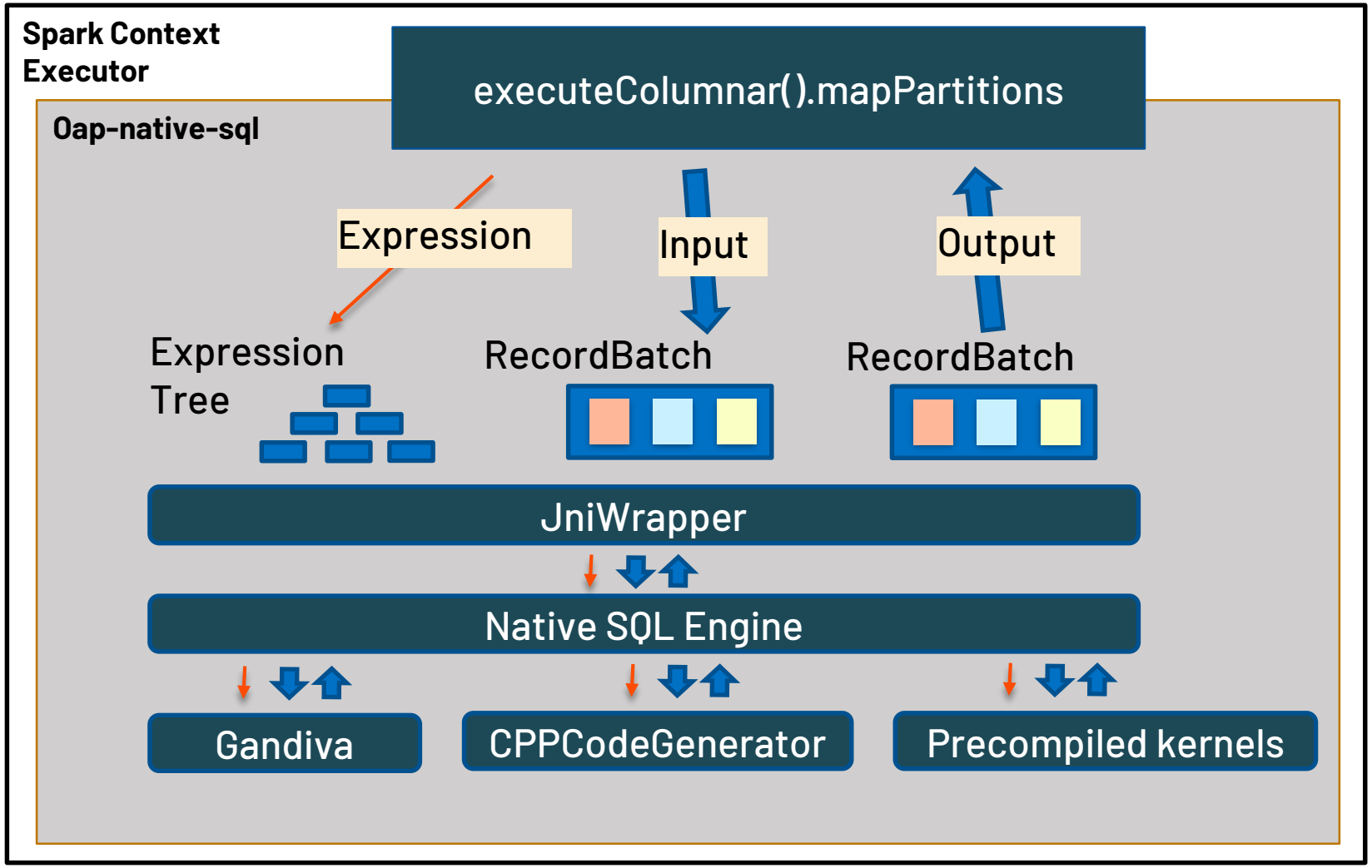
1. Build HashRelation #0
2. Build HashRelation #1
3. Build HashRelation #2

```
Loop keys_arrays {
  probe key_0 in HashRelation #0
  apply Project
  probe  key_1 in HashRelation #1
  apply condition
  probe key_2 in HashRelation #2
  materialize one line to arrow
}
```

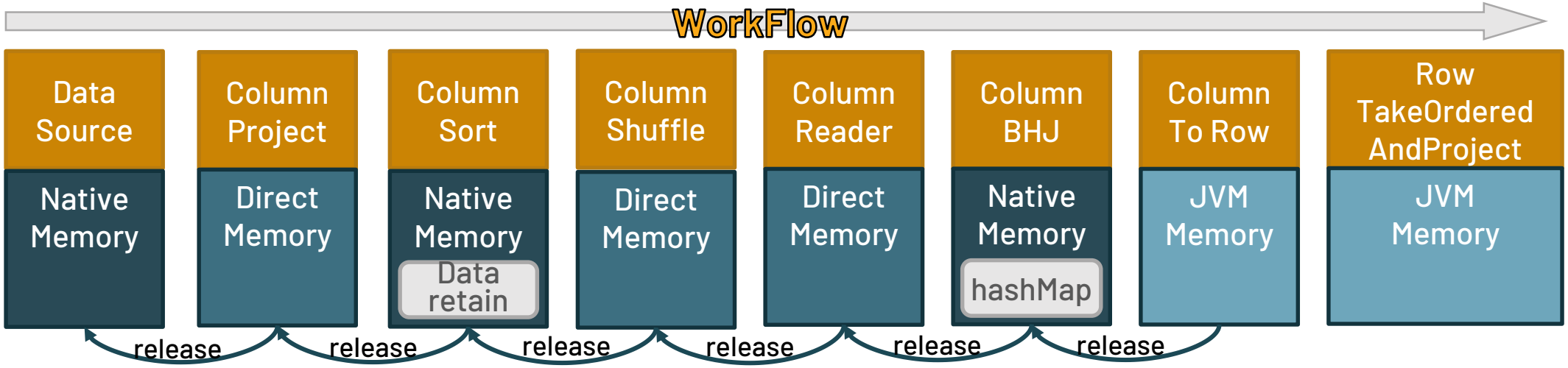
g++ compilation

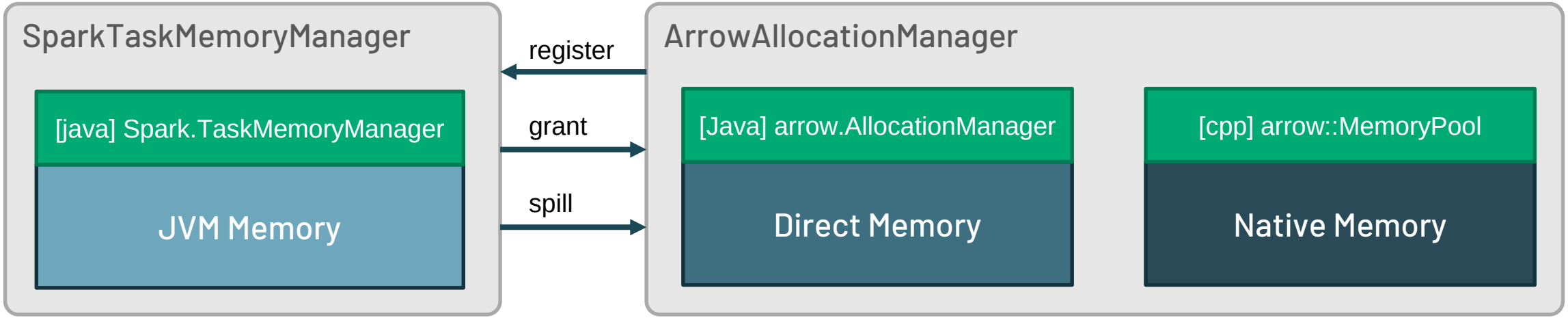
```
AVX optimized {
  probe key_0 in HashRelation #0
  apply project
  probe  key_1 in HashRelation #1
  apply condition
  probe key_2 in HashRelation #2
  materialize one line to arrow
}
```

Native SQL engine call flow

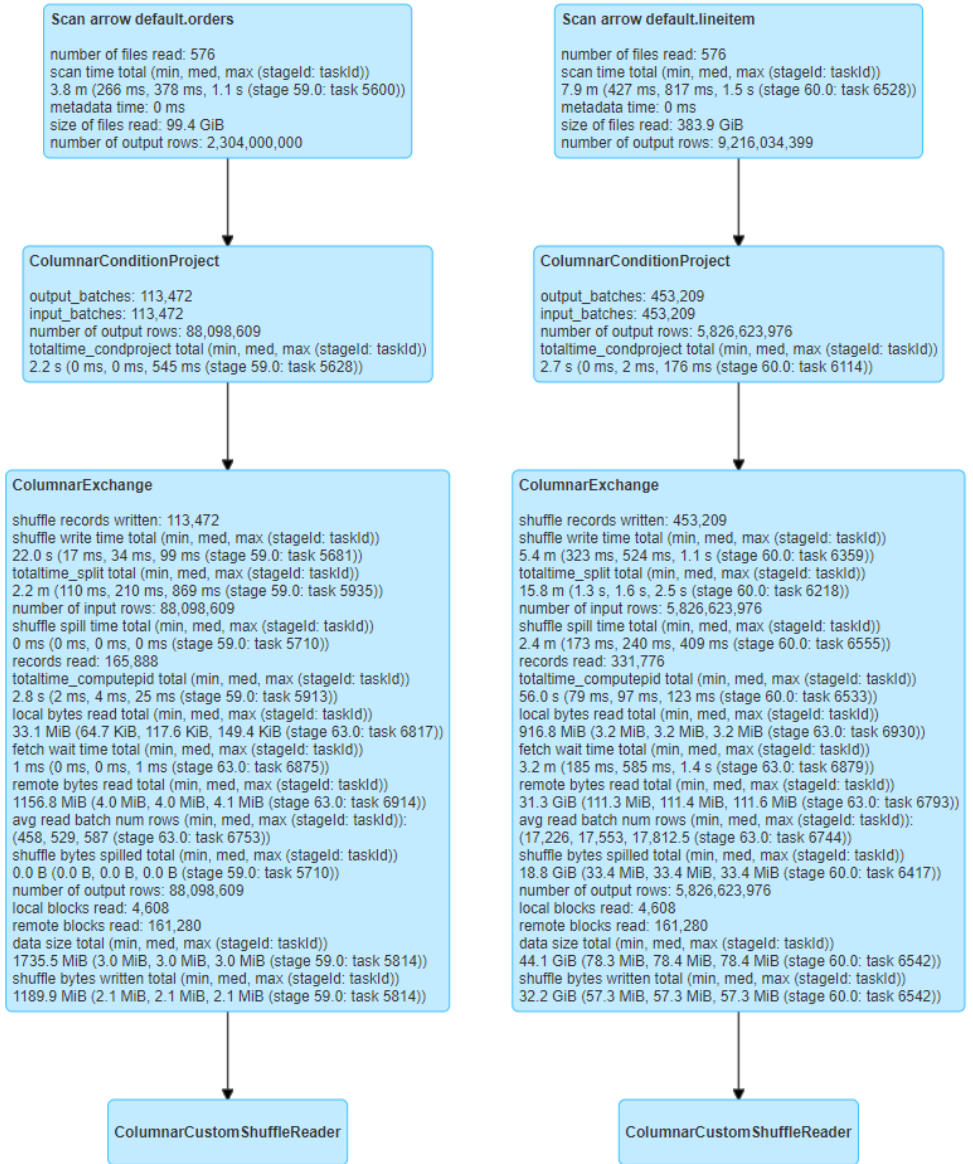


Memory Management





Example run of TPCH-Q4



Summary

- AVX instructions can greatly improve performance on SQL workloads
- Native SQL is open sourced. For more details please visit: https://github.com/Intel-bigdata/OAP
- Native SQL engine is under heavy development, works for TPC-H/TPC-DS now

Q&A

Feedback

Your feedback is important to us.

Don’t forget to rate and review the sessions.

Legal Information: Benchmark and Performance Disclaimers

- Performance results are based on testing as of Feb. 2019 & Aug 2020 and may not reflect all publicly available security updates. See configuration disclosure for details. No product can be absolutely secure.
- Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information, see Performance Benchmark Test Disclosure.
- Configurations: see performance benchmark test configurations.

Notices and Disclaimers

- No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.
- Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.
- This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.
- The products and services described may contain defects or errors known as errata which may cause deviations from published specifications. Current characterized errata are available on request.
- Intel, the Intel logo, Xeon, Optane, Optane Persistent Memory are trademarks of Intel Corporation in the U.S. and/or other countries.
- *Other names and brands may be claimed as the property of others
- © Intel Corporation.