SPARK NATIVE SQL ENGINE

Binwei Yang, Chendi Xue, Yuan Zhou, Hongze Zhang, Ke Jia
Weiting Chen, Carson Wang, Jian Zhang
IAGS/MLP/DAS/BDF
Contact: weiting.chen@intel.com **Last update: May, 2021**

AGENDA

Motivation

Core Design

- Architecture
- Arrow Data Source
- Native SQL Engine
- Columnar Shuffle

Getting Started

Performance

Summary

AGENDA

Motivation

Core Design

- Architecture
- Arrow Data Source
- Native SQL Engine
- Columnar Shuffle

Getting Started

Performance

Summary

MOTIVATION

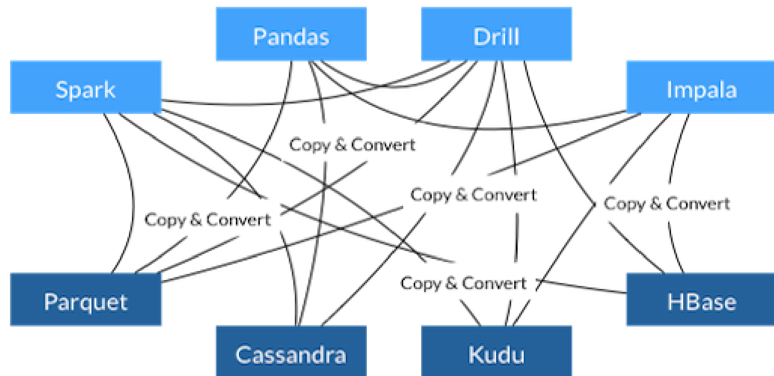
Issues of current Spark SQL Engine

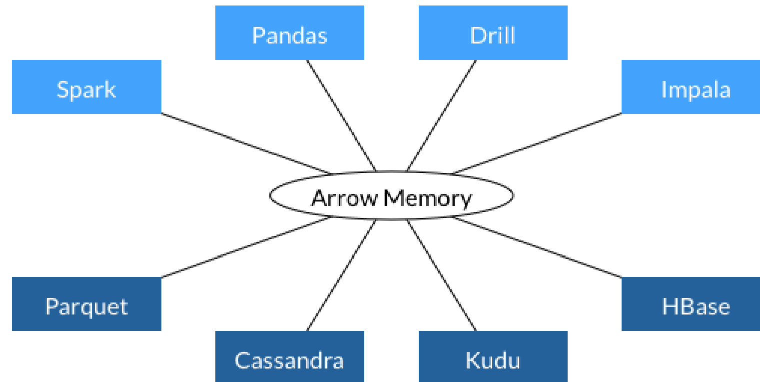
- Row based processing, not friendly for SIMD instructions with Intel technologies such as AVX, GPU, FPGA, …etc.
- Java GC Overhead
- JIT code quality relies on JVM, hard to tune
- High overhead of integration with other native libraries

Goals

- Move the key SQL operations to highly optimized native code with Intel technologies
- Easily add accelerators support with Intel products in Spark
- Building a friendly end-to-end AI workloads

APACHE ARROW – UNIFIED DATA FORMAT





Without Arrow

- Each system has its own internal memory format
- 70-80% computation wasted on serialization and deserialization
- Similar functionality implemented in multiple projects

With Arrow

- All systems utilize the same memory format
- No overhead for cross-system communication
- Projects can share functionality(eg, Parquet-to-Arrow reader)

Reference: https://arrow.apache.org

APACHE ARROW – HIGHLY OPTIMIZED LIB.

Columnar format -> The opportunity for Intel AVX Support

Native C++ implementation –> No JVM Overhead

Dataframe functions -> SQL Operators such as Filter, Join, Aggregate …

Implement Java, python, etc. as interface

Share to other projects with a unified interface -> Pandas, Spark, Flink…

Using the same interface to offload to other accelerators such as Intel GPU, Intel FPGA

SPARK-27396 PUBLIC APIS FOR EXTENED COLUMNAR PROCESSING

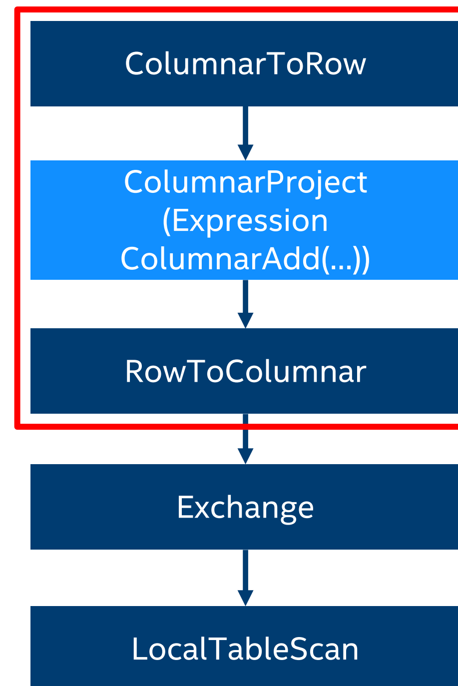
https://issues.apache.org/jira/browse/SPARK-27396

Existing APIs use RDD[InternalRow]

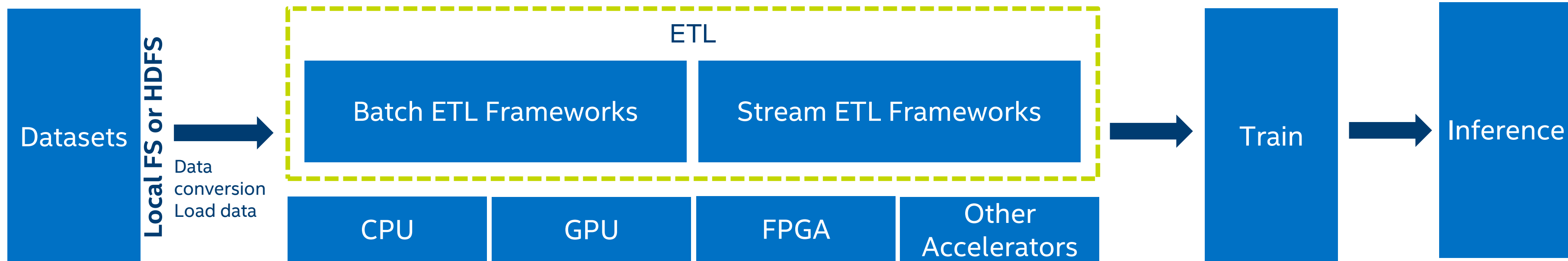
New APIs use RDD[ColumnarBatch]
- def executeColumnar(): RDD[ColumnarBatch]
- def columnarEval(batch: ColumnarBatch): Any
- class ColumnarRule

ColumnarToRow/RowToColumnar
- Translate RDD[InternalRow] <-> RDD[ColumnarBatch]

The opportunity to support accelerators in Spark



END TO END WORKLOAD WITH ARROW



The vision is

- Use Apache Arrow as unified in memory format
- Use Arrow between different frameworks
- Use Arrow between different accelerators

AGENDA

Motivation

Core Design
- Architecture
- Arrow Data Source
- Native SQL Engine
- Columnar Shuffle

Getting Started

Performance

Summary

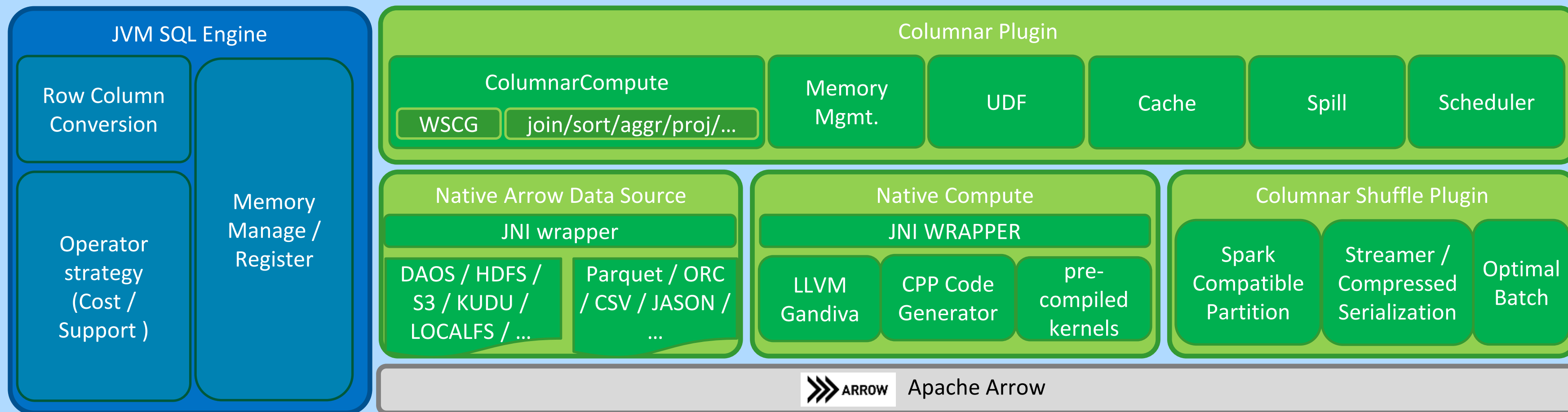
NATIVE SQL ENGINE ARCHITECTURE

Spark Application

SQL | Python APP | Java/Scala APP | R APP

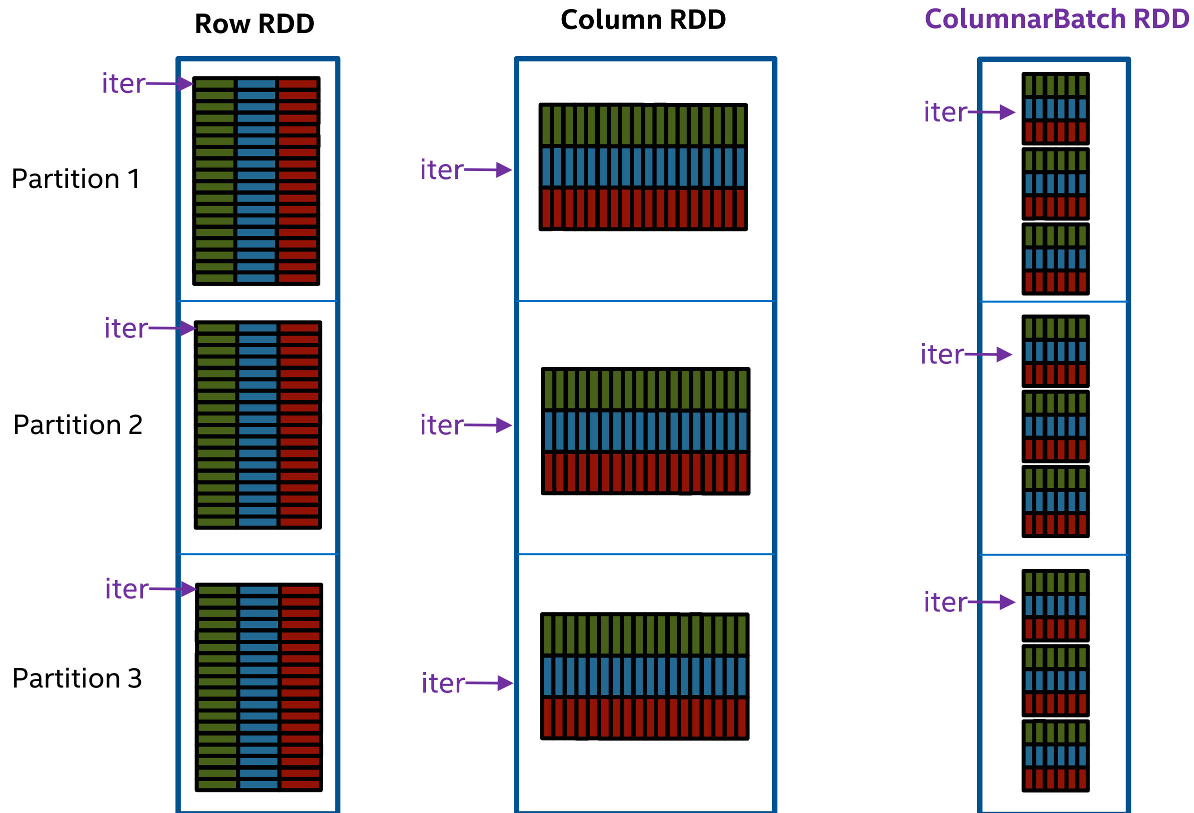
Query Plan Optimization

ColumnarRules | ColumnarCollapseRules | ColumnarAQERules | FallbackRules



Github Repository: https://github.com/oap-project/native-sql-engine

BASIC DATA FORMAT: COLUMNAR BATCH



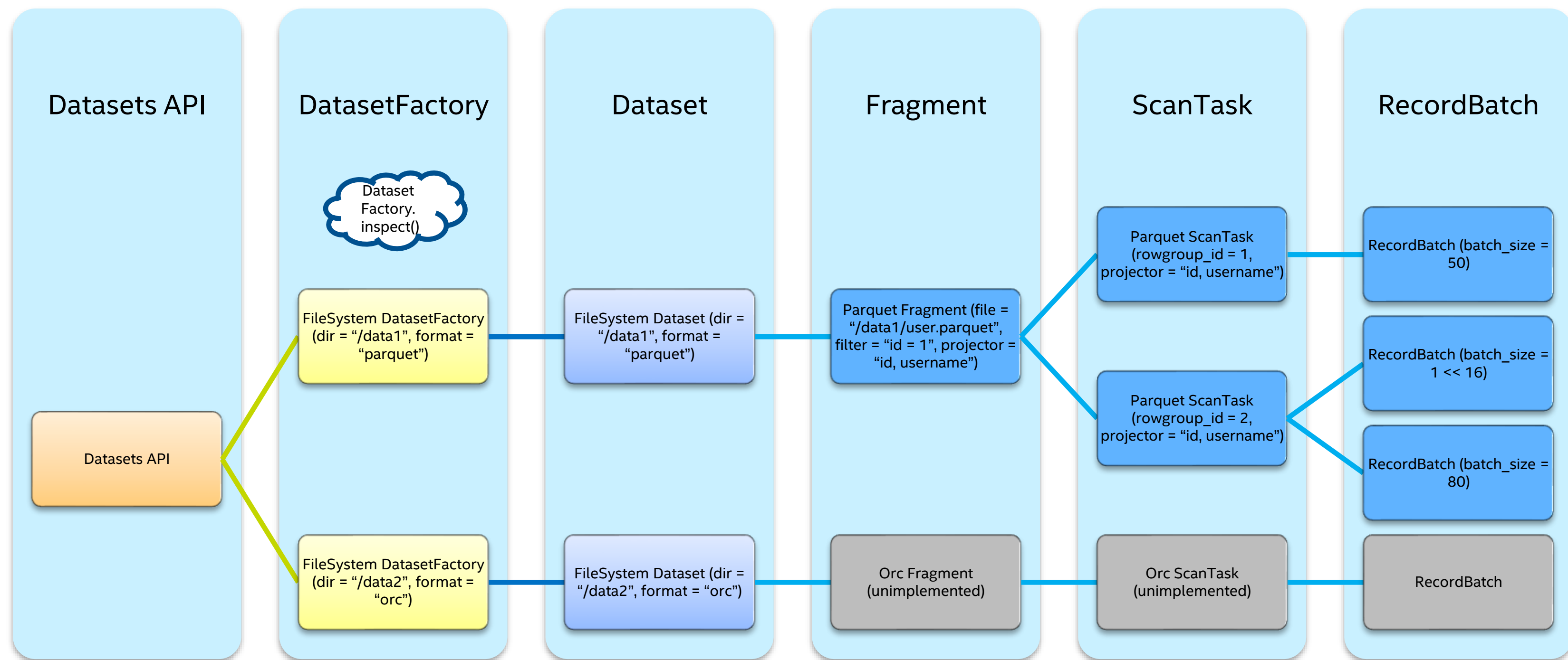
Batch size is configurable

- =#row => Column RDD
- =1 => Row RDD

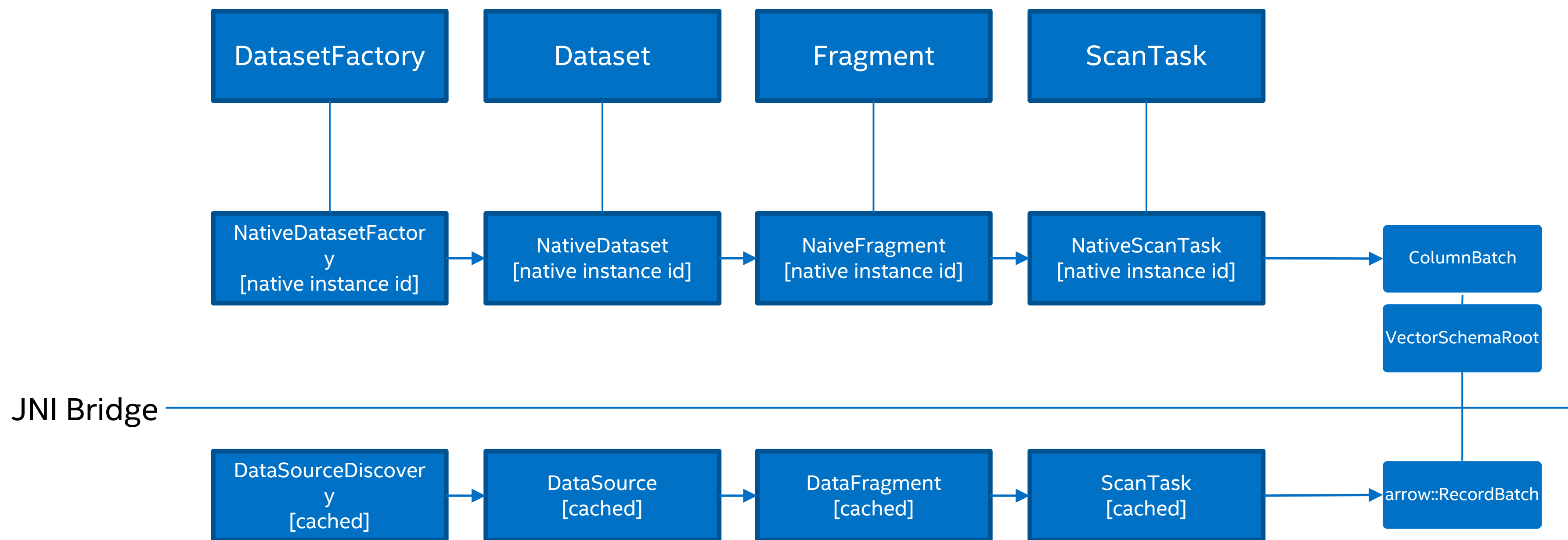
Some parameters are related to the Batch size, need to be tuned carefully:

- spark.sql.parquet.columnarReaderBatchSize
- spark.sql.inMemoryColumnarStorage.batchSize
- spark.sql.execution.arrow.maxRecordsPerBatch

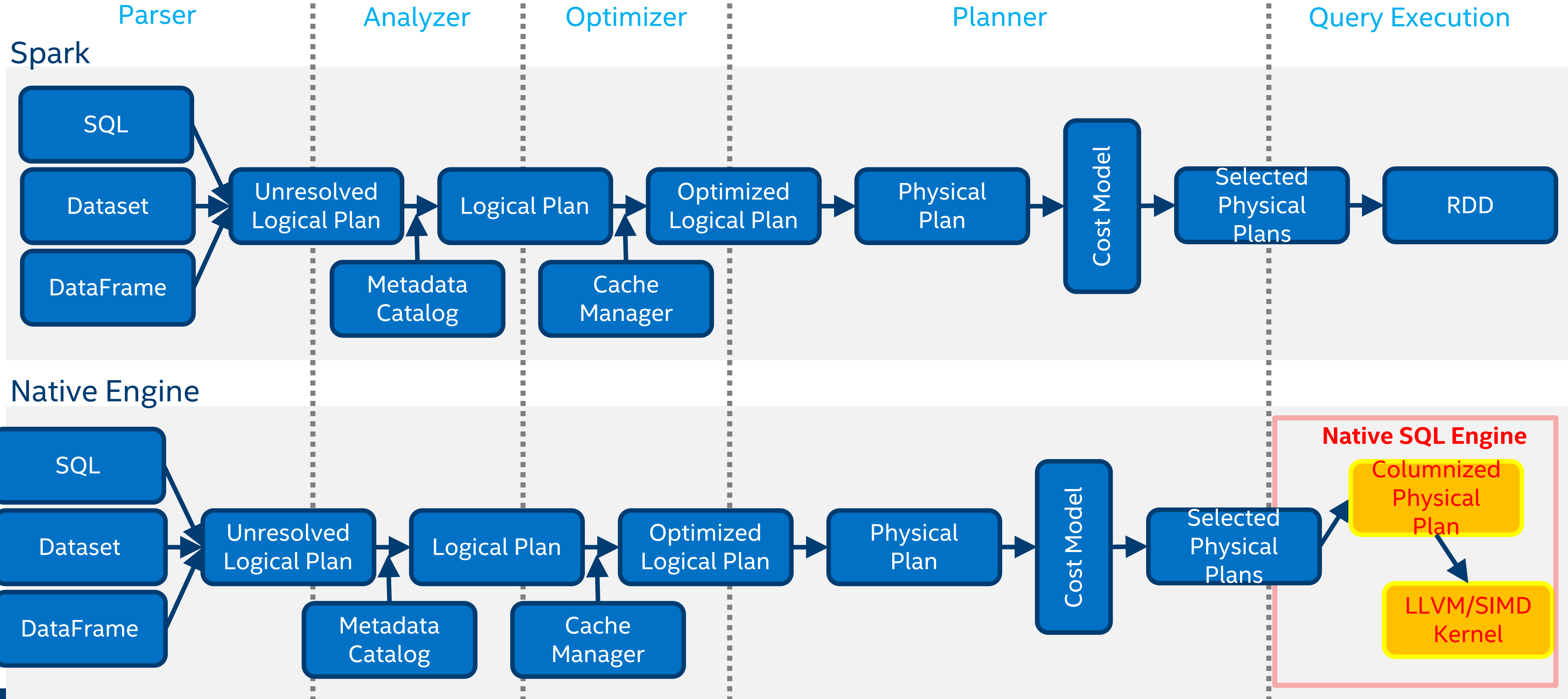
LEVERAGING C++ ARROW DATASETS API



DATASETS JAVA/SCALA API FOR SPARK



SPARK SQL NATIVE ENGINE



NATIVE SQL OPERATOR

What SparkColumnarPlugin does:

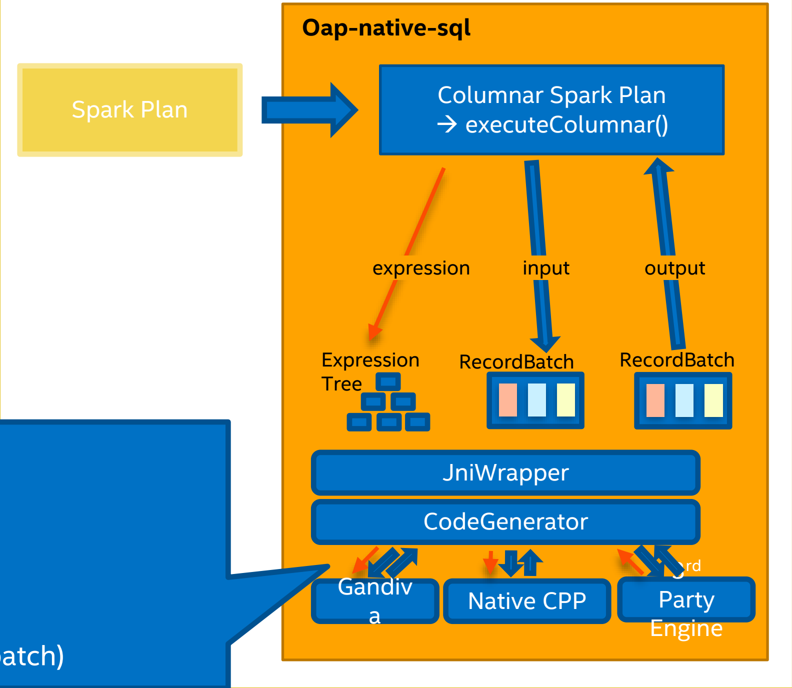
Plug into spark as an extra jar

Convert Spark Plan to a columnar supported spark plan.

Handle data using ColumnarBatch format

Use Apache Arrow Compute and Gandiva modules to do ColumnarBased Data Process.

key	value
spark.sql.extensions	com.intel.sparkColumnarPlugin.ColumnarPlugin



Unified API

Build(schema, expr_tree, return_types)

Evaluate(input_record_batch, *vector<output_record_batch>)

Finish(*ResultIterator<record_batch>)

ResultIterator->Next()

ResultIterator->Process(input_record_batch)

JAVA SIDE WORKFLOW

Expression Build (CodeGen):

- Convert a list of expressions into one columnar based function

ColumnarBased Evaluation

- Pass a columnarBatch to this operator, then return processed columnarBatch using ‘CodeGen’ed function.

Expression codegen:

Expression:
(c_0 + c_1 + c_2 + ...c_9)

1

Spark Expression Tree:
Alias
Add
Add c_9
Add c_8
Add c_7
......
c_0 c_1

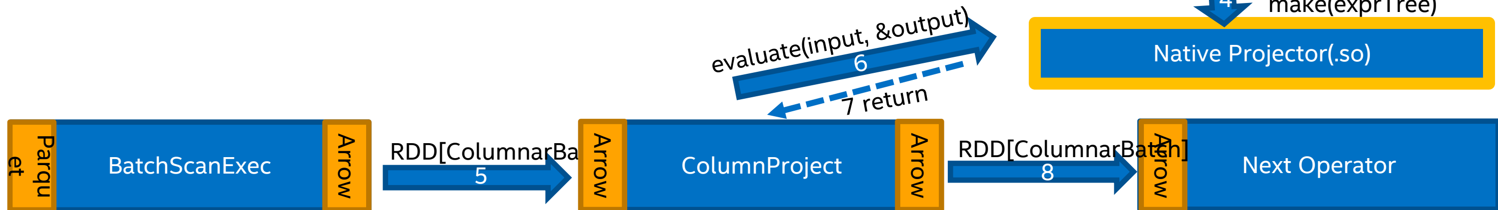
2

Spark Expression Tree:
Alias
ColumnarAdd
ColumnarAdd c_9
ColumnarAdd c_8
ColumnarAdd c_7
......
c_0 c_1

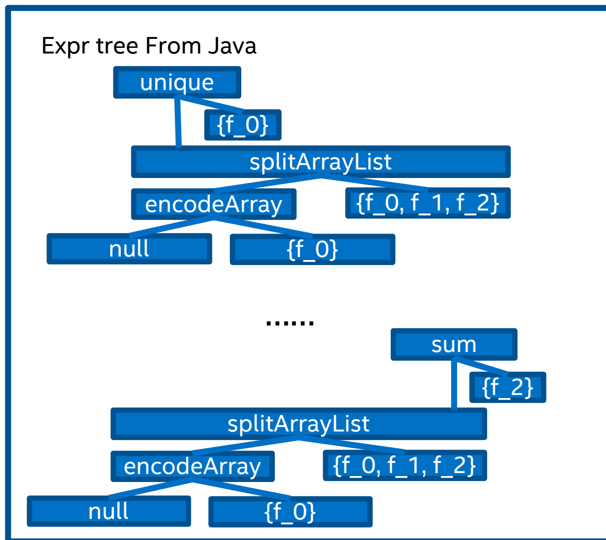
3

Gandiva Expression Tree:
ResultTreeNode
AddTreeNode
AddTreeNode int
AddTreeNode int
AddTreeNode int
......
int int

4 make(exprTree)



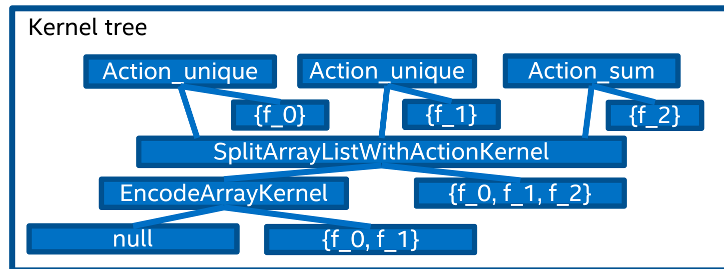
NATIVE SIDE WORKFLOW



build

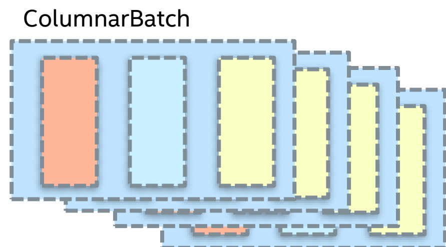
Expr visitor Cache

Func_id	Expr_vistor_ptr	Dependency_ptr
encodeArrayf_0	Visitor_0	Nullptr
splitArrayListf_0f_1f_2	Visitor_1	Visitor_0
uniquef_0	Visitor_2	Visitor_1
uniquef_1	Visitor_3	Visitor_1
sumf_2	Visitor_4	Visitor_1



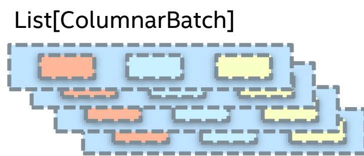
Evaluate

ColumnarBatch

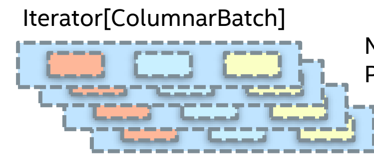


or

List[ColumnarBatch]

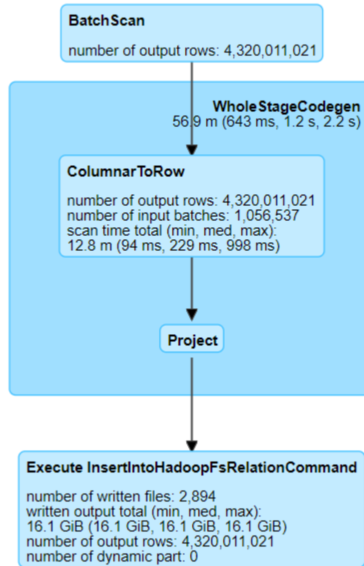


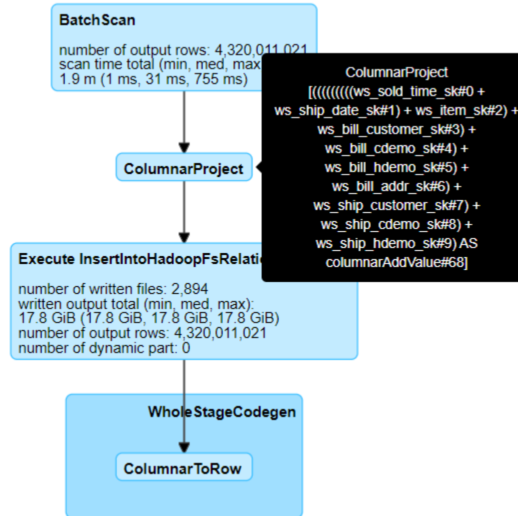
Iterator[ColumnarBatch]



Next()
Process(columnarBatch)

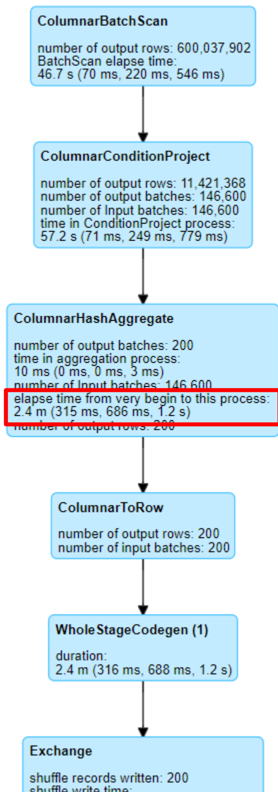
EXAMPLE 1: COLUMNAR CONDITIONED PROJECT

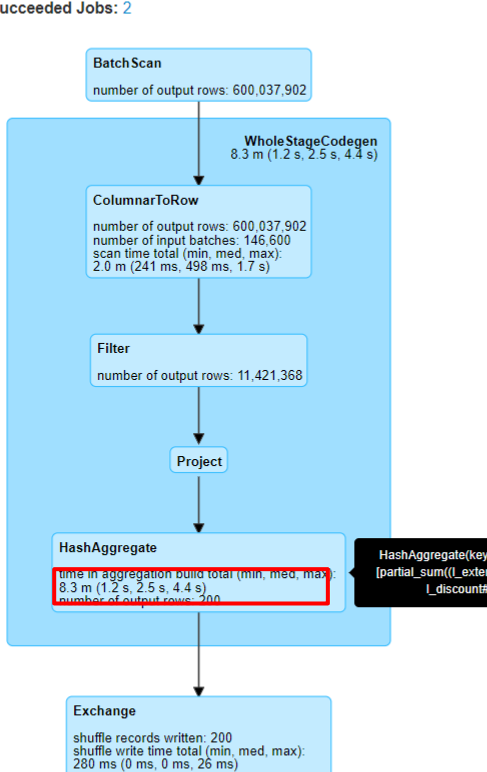


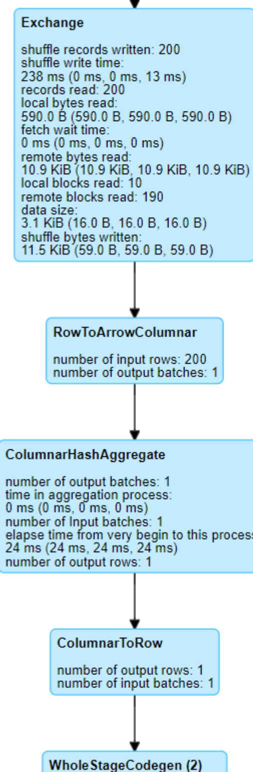


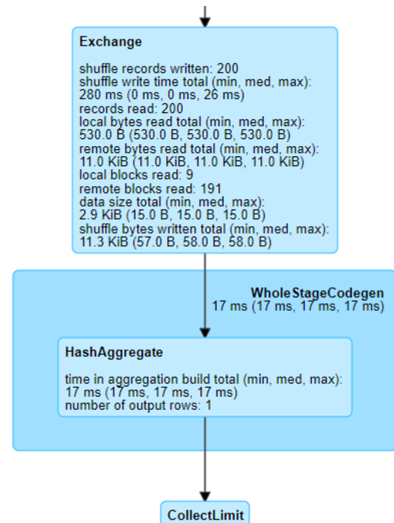
ColumnarBased project took 28 sec, and RowBased project took 50 sec.
Project added 10 columns into one.

EXAMPLE 2: COLUMNAR GROUPBY AGGREGATE IN TPCH Q6









Details

```
== Parsed Logical Plan ==
GlobalLimit 21
+- LocalLimit 21
   +- Project [cast(revenue#762 as string) AS revenue#7
      +- Aggregate [sum((l_extendedprice#735 * l_discou
         +- Filter (((cast(l_shipdate#740 as date) >= 8
(l_discount#736 <= cast(0.07 as double))) AND (l_quanti
            +- SubqueryAlias `lineitem`
               +- RelationV2[l_orderkey#730L, l_partkey
l_tax#737, l_returnflag#738, l_linestatus#739, l_shipda
```

ColumnarBased project took 3 sec, and RowBased project took 7 sec for TPCH Q6
HashAggregate took most of exec time, ColumnarBased spent avg 686ms, and Rowbased spend avg 2.5s

CURRENT FLOW OPERATOR BY OPERATOR

Stage 1

Scan B

Filter

BoradcastExchange

Stage 2

Scan A

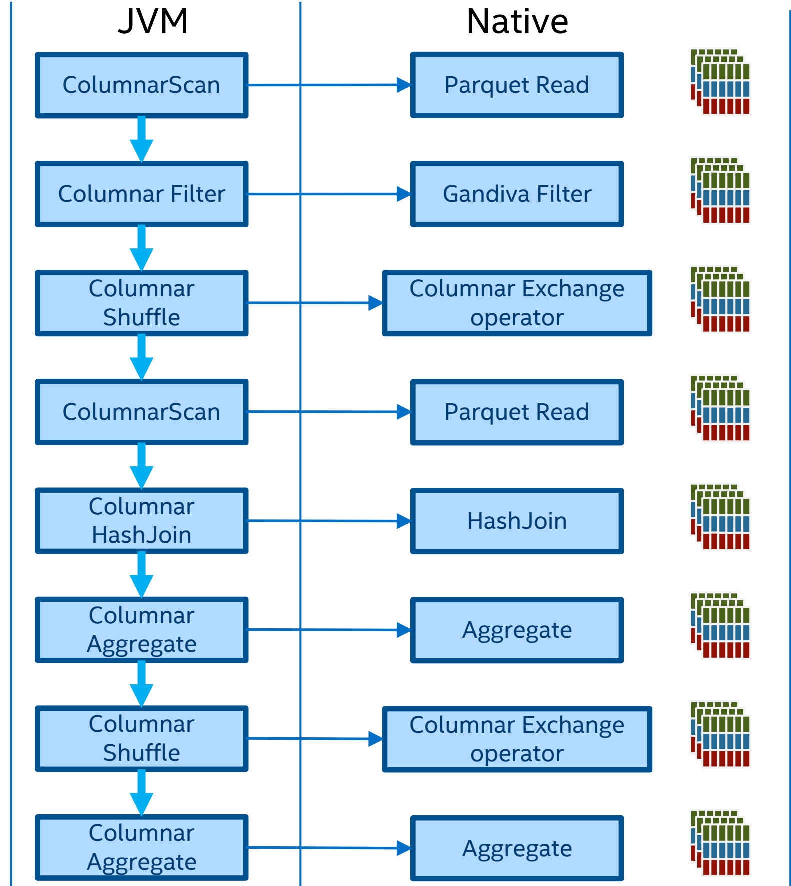
BoradcastHashJoin

HashAggregate

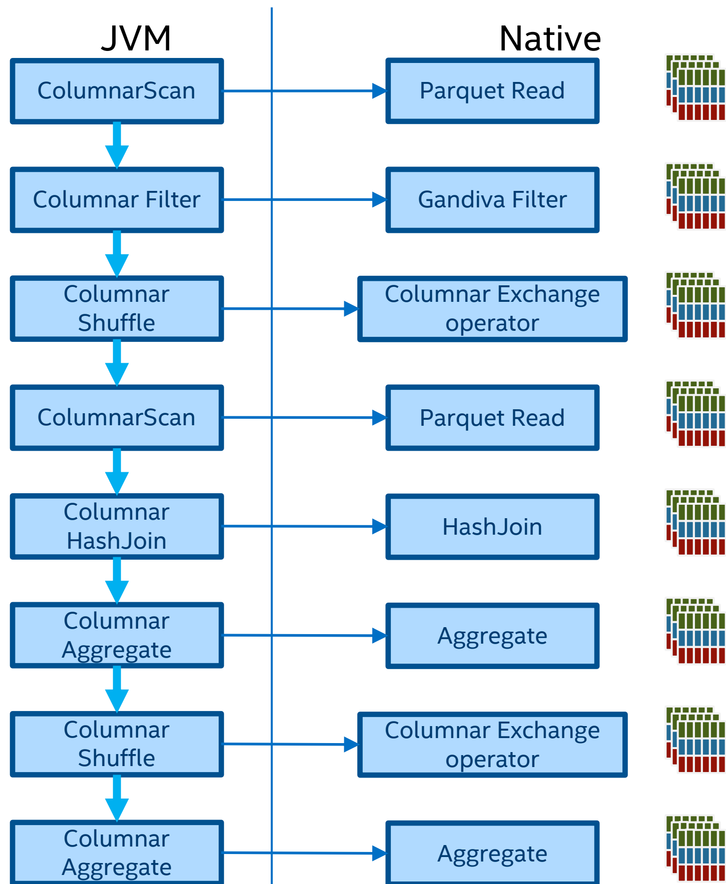
ShuffleExchange

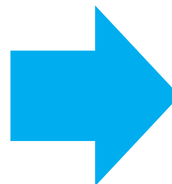
Stage 3

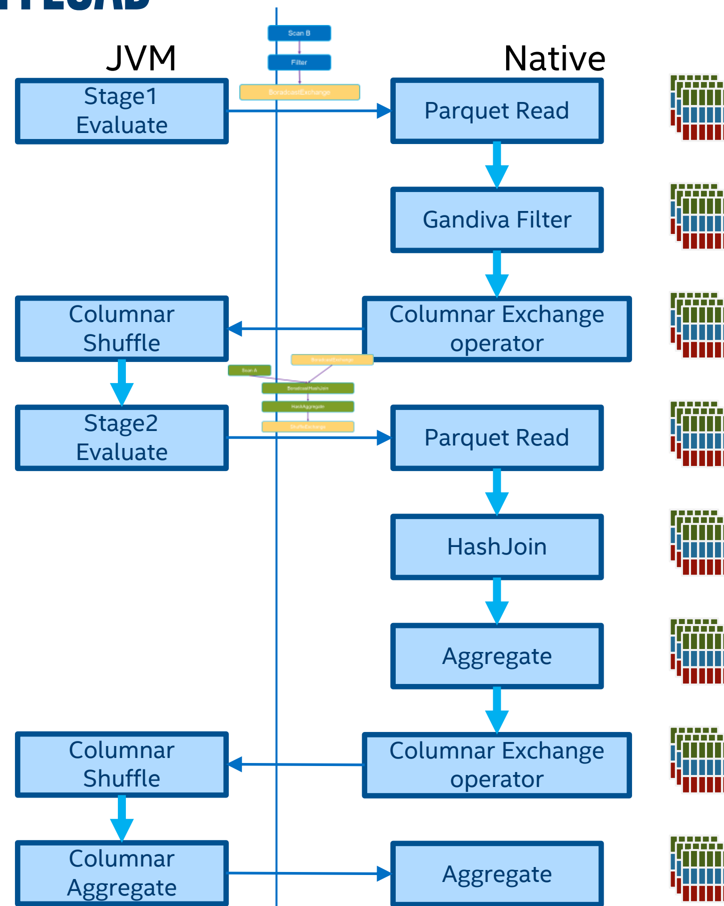
HashAggregate



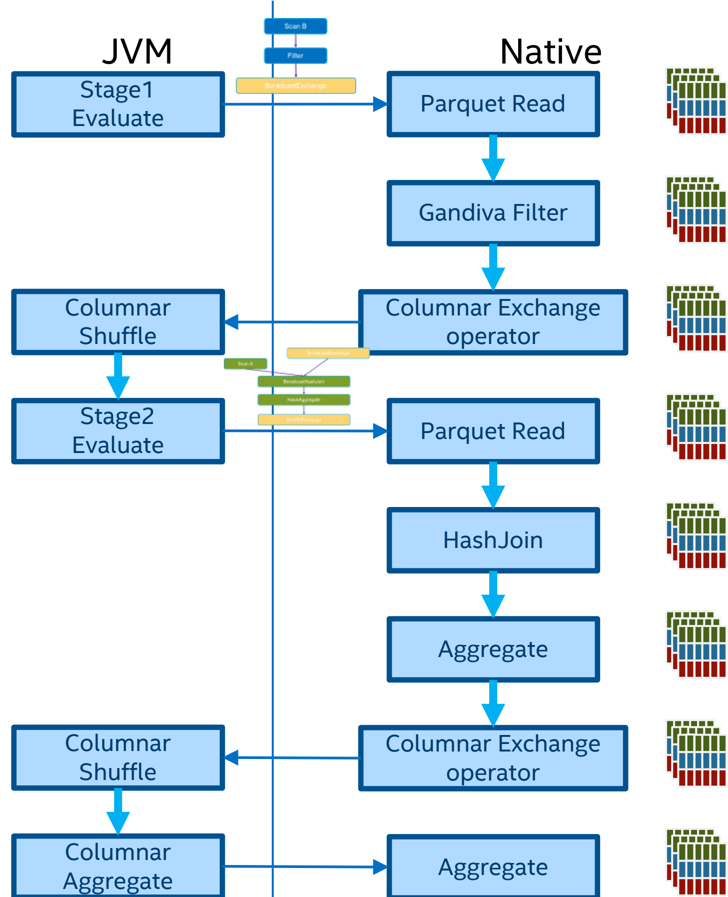
WHOLE STAGE OFFLOAD

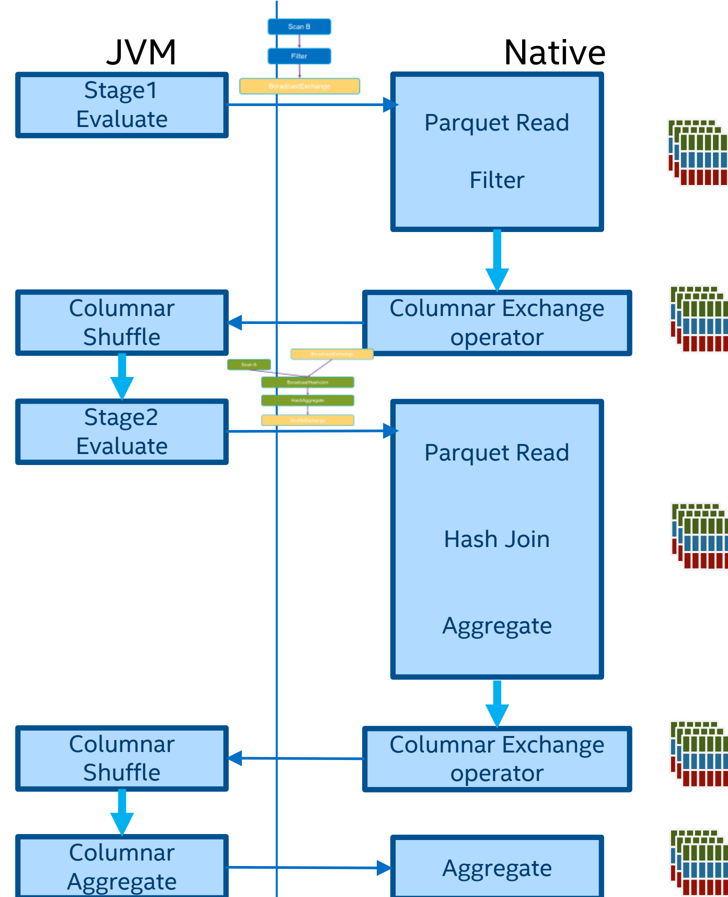




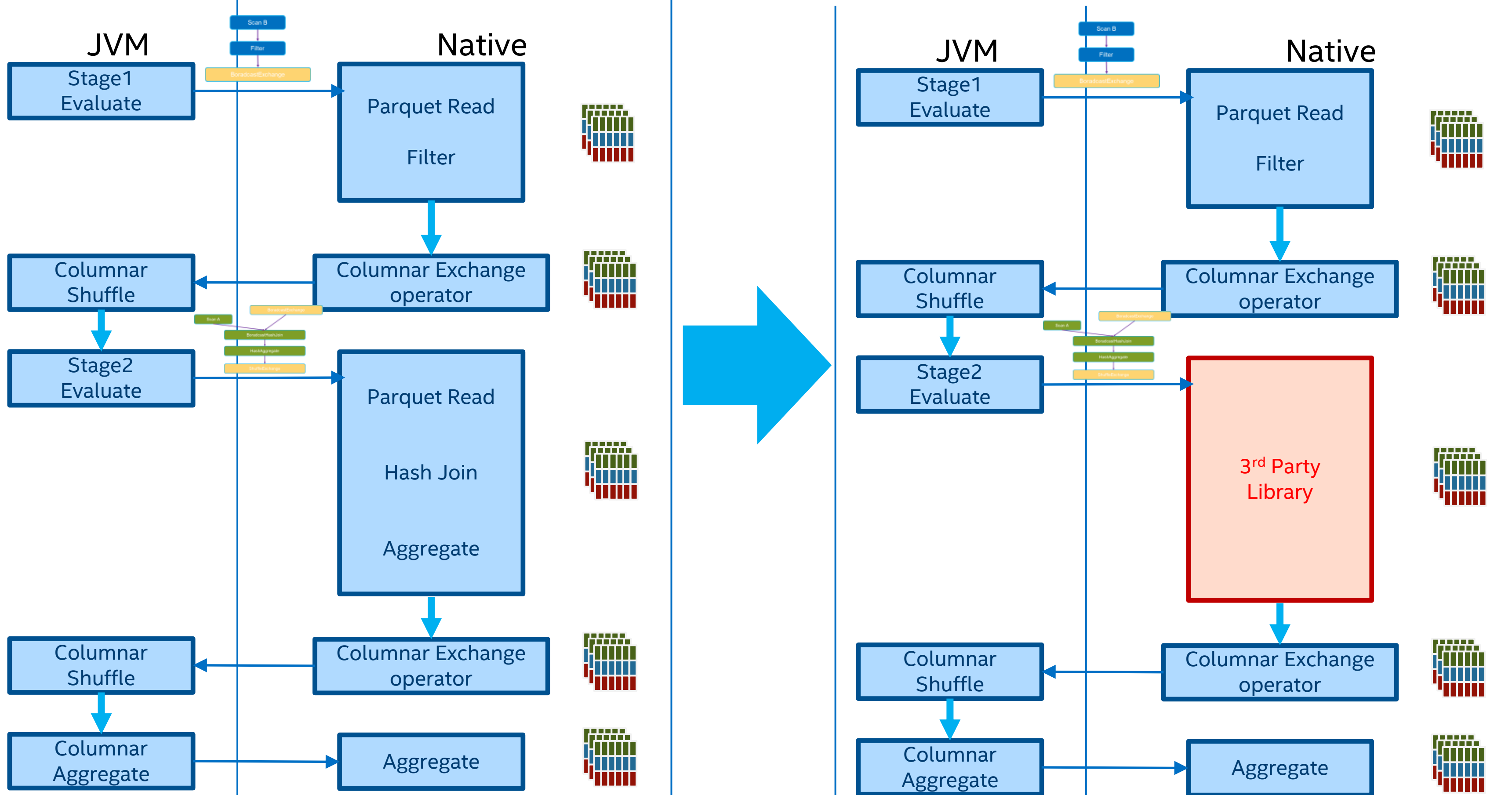


WHOLE STAGE OFFLOAD + NATIVE WHOLE STAGE CODE GEN





WHOLE STAGE OFFLOAD + THIRD PARTY LIBRARY



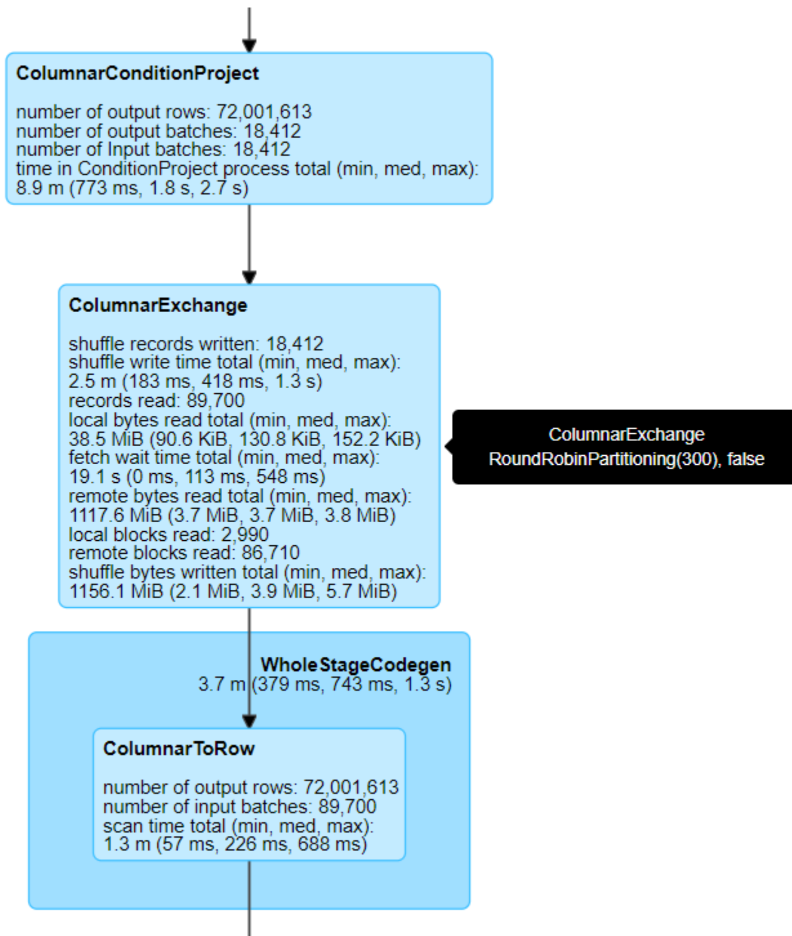
SUPPORTED SQL OPERATORS OVERVIEW

https://github.com/oap-project/native-sql-engine/blob/master/docs/operators.md

Operators(X of Y supported)	SQL functions (X of Y supported)	Data Types (X of Y supported)
WindowExec	NormalizeNaNAndZero	
UnionExec	Subtract	
ExpandExec	Substring	
SortExec	ShiftRight	
ScalarSubquery	Round	
ProjectExec	PromotePrecision	
ShuffledHashJoin	Multiply	
BroadcastJoinExec	Literal	
FilterExec	LessThanOrEqual	
ShuffleExchangeExec	LessThan	
BroadcastExchangeExec	KnownFloatingPointNormalized	
datasources.v2.BatchScanExec	IsNull	
datasources.v1.FileScanExec	And	
HashAggregateExec	Add	
SortMergeJoinExec …..	….	….

100% TPC-DS **performance critical operators** supported
Automatically fallback to row-based execution if there are unsupported operators/expressions

COLUMNAR SHUFFLE



COLUMNAR SHUFFLE

Shuffle Write

NativeSql::ColumnarShuffleExchange

1. Insert Partition Id

NativeSql::ColumnarShuffleWriter

2. Native split

3. Native compress & serialize & write to disk

Temp file

Temp file

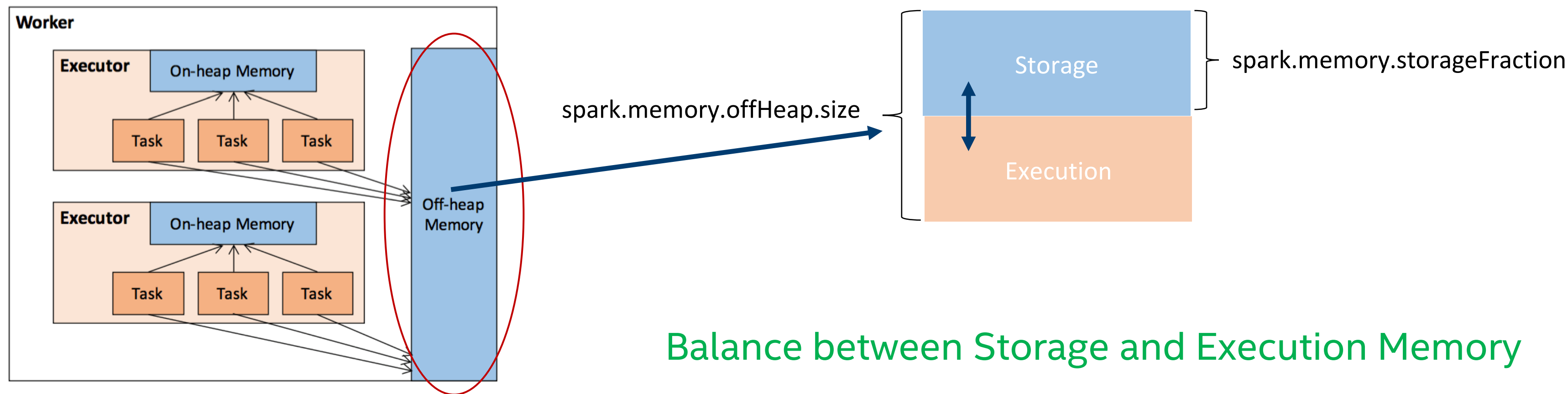
Temp file

Shuffle Read

InputDataStream

deserialize & decompress

MEMORYMANAGEMENT



- If none of its space is insufficient but the other is free, then it will borrow the other's space
- If both parties doesn't have enough space, evict storage memory using LRU mechanism

AGENDA

Motivation

Core Design

- Architecture
- Arrow Data Source
- Native SQL Engine
- Columnar Shuffle

Getting Started

Performance

Summary

METHOD1: USE FAT JAR

Download Fat Jars from https://mvnrepository.com/artifact/com.intel.oap

Use fat jar is a quick way to run Native SQL Engine on your environment. You only need to download below two jar files and add them when running spark-shell, spark-submit, …etc:

1. Arrow Data Source Jar:
https://repo1.maven.org/maven2/com/intel/oap/spark-arrow-datasource-standard/1.1.0/spark-arrow-datasource-standard-1.1.0-jar-with-dependencies.jar

2. Native SQL Jar:
https://repo1.maven.org/maven2/com/intel/oap/spark-columnar-core/1.1.0/spark-columnar-core-1.1.0-jar-with-dependencies.jar

Please notices to use the fat jar files, you must ensure your environment is fulfilled below requirements:

- Use GCC9.3.0
- Use LLVM 7.0.1

Reference: https://github.com/oap-project/native-sql-engine/blob/master/README.md

METHOD2: BUILDING FROM CONDA

https://github.com/oap-project/native-sql-engine/blob/master/docs/OAP-Installation-Guide.md

Use Conda to build the jar files for all the OAP projects.

Step0: Please make sure your environment has installed the prerequisites(link)

Step1:

```
# conda create -n oapenv -y python=3.7
# conda activate oapenv
# conda install -c conda-forge -c intel -y oap=1.1.0
```

Step2: You can find all the jar files in below directory.

```
$HOME/miniconda2/envs/oapenv/oap_jars
```

Please notices to use conda to build the jars will install all the required dependency libraries and all the jars for every single OAP projects including Native SQL Engine, SQL Data Source Cache, Pmem, ...etc.

METHOD3: BUILDING BY YOURSELF

https://github.com/oap-project/native-sql-engine/blob/master/docs/OAP-Installation-Guide.md

Step0: Please make sure your environment has installed the prerequisites(link)

Step1: Please follow the installation guide to build Native SQL Engine(link)

```
# git clone -b ${version} https://github.com/oap-project/native-sql-engine.git
# cd oap-native-sql
# mvn clean package -DskipTests -Dcpp_tests=OFF -Dbuild_arrow=ON -Dcheckstyle.skip
```

Please notices Native SQL Engine use a custom version of Apache Arrow(link), the parameter "-Dbuild_arrow" can help to build arrow from source.

You can also modify the parameters in Arrow from build_arrow.sh

HOW TO USE THE JARS

```
${SPARK_HOME}/bin/spark-shell \
    --verbose \
    --master yarn \
    --driver-memory 10G \
    --conf spark.driver.extraClassPath=$PATH_TO_JAR/spark-arrow-datasource-standard-<version>-jar-with-dependencies.jar:$PATH_TO_JAR/spark-columnar-core-<version>-jar-with-dependencies.jar \
    --conf spark.executor.extraClassPath=$PATH_TO_JAR/spark-arrow-datasource-standard-<version>-jar-with-dependencies.jar:$PATH_TO_JAR/spark-columnar-core-<version>-jar-with-dependencies.jar \
    --conf spark.driver.cores=1 \
    --conf spark.executor.instances=12 \
    --conf spark.executor.cores=6 \
    --conf spark.executor.memory=20G \
    --conf spark.memory.offHeap.size=80G \
    --conf spark.task.cpus=1 \
    --conf spark.locality.wait=0s \
    --conf spark.sql.shuffle.partitions=72 \
    --conf spark.sql.extensions=com.intel.oap.ColumnarPlugin \
    --conf --conf spark.shuffle.manager=org.apache.spark.shuffle.sort.ColumnarShuffleManager \
    --conf spark.executorEnv.ARROW_LIBHDFS3_DIR="$PATH_TO_LIBHDFS3_DIR/" \
    --conf spark.executorEnv.LD_LIBRARY_PATH="$PATH_TO_LIBHDFS3_DEPENDENCIES_DIR"
    --jars $PATH_TO_JAR/spark-arrow-datasource-standard-<version>-jar-with-dependencies.jar,$PATH_TO_JAR/spark-columnar-core-<version>-jar-with-dependencies.jar
```

HOW TO CONFIGURE THE PARAMETERS

https://github.com/oap-project/native-sql-engine/blob/master/docs/Configuration.md

The users can use the parameters to enable or disable the columnar based operators.

For example, spark.oap.sql.columnar.sortmergejoin can help to turn on/off columnar sort merge join.

We expose most of the parameters based on the columnar operators to help the users to fine-tune the performance by individual queries.

Spark Configurations for Native SQL Engine

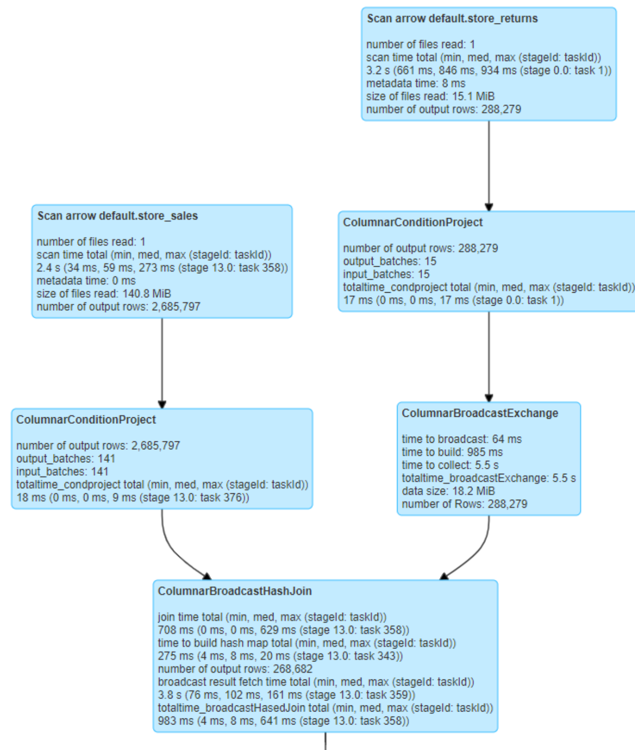
There are many configuration could impact the Native SQL Engine performance and can be fine tune in Spark. You can add these configuration into spark-defaults.conf to enable or disable the setting.

spark.sql.extensions	To turn on Native SQL Engine Plugin	com.intel.oap.ColumnarPlugin
spark.shuffle.manager	To turn on Native SQL Engine Columnar Shuffle Plugin	org.apache.spark.shuffle.sort.ColumnarShuffleMan
spark.oap.sql.columnar.batchscan	Enable or Disable Columnar Batchscan, default is true	true
spark.oap.sql.columnar.hashagg	Enable or Disable Columnar Hash Aggregate, default is true	true
spark.oap.sql.columnar.projfilter	Enable or Disable Columnar Project and Filter, default is true	true
spark.oap.sql.columnar.codegen.sort	Enable or Disable Columnar Sort, default is true	true
spark.oap.sql.columnar.window	Enable or Disable Columnar Window, default is true	true
spark.oap.sql.columnar.shuffledhashjoin	Enable or Disable ShffuledHashJoin, default is true	true
spark.oap.sql.columnar.sortmergejoin	Enable or Disable Columnar Sort Merge Join, default is true	true

HOW TO CHECK IF NATIVE SQL ENGINE IS ENABLED

If the query can run successfully, you can check the DAG in history server. The chart on the right is an example for your reference.

If the jar cannot be load in spark CLI such as spark-shell, there will be error message in CLI.



AGENDA

Motivation

Core Design
- Architecture
- Arrow Data Source
- Native SQL Engine
- Columnar Shuffle

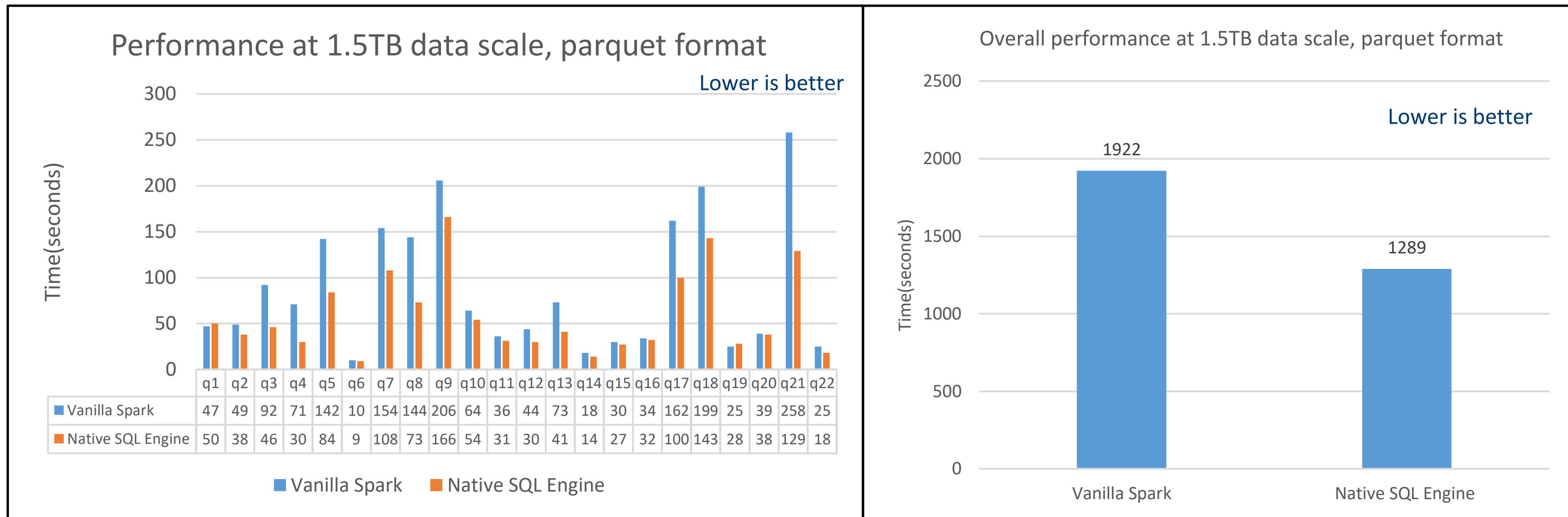
Getting Started

Performance

Summary

NATIVE SQL ENGINE VS VANILLA SPARK

DECISION SUPPORT BENCHMARK 1 DERIVED FROM TPC-H, 22 QUERIES

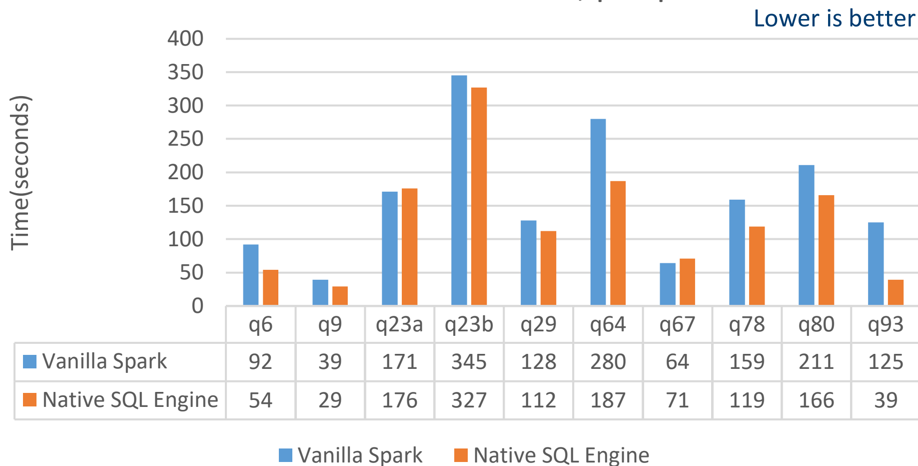


	q1	q2	q3	q4	q5	q6	q7	q8	q9	q10	q11	q12	q13	q14	q15	q16	q17	q18	q19	q20	q21	q22
Vanilla Spark	47	49	92	71	142	10	154	144	206	64	36	44	73	18	30	34	162	199	25	39	258	25
Native SQL Engine	50	38	46	30	84	9	108	73	166	54	31	30	41	14	27	32	100	143	28	38	129	18

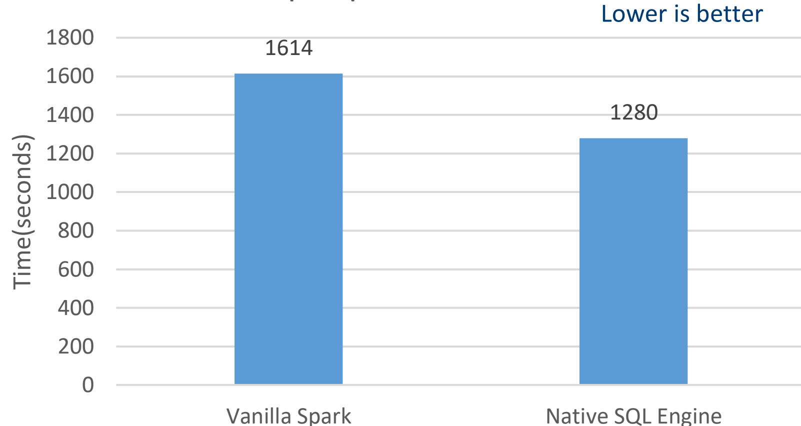
- 49% performance speedup over Vanilla Spark @ 1.5TB dataset

NATIVE SQL ENGINE VS VANILLA SPARK

DECISION SUPPORT BENCHMARK 2 DERIVED FROM TPC-DS, 10 QUERIES



	q6	q9	q23a	q23b	q29	q64	q67	q78	q80	q93
Vanilla Spark	92	39	171	345	128	280	64	159	211	125
Native SQL Engine	54	29	176	327	112	187	71	119	166	39



- 26.1% performance speedup over Vanilla Spark @ 1.5TB dataset

HOW TO TUNE THE PERFORMANCE

Run a profiling by individual query, check how much operators can run with columnar based processing.

If most of the operators are using columnar based processing, you can
1. set spark.executor.memory as smaller as possible
2. keep spark.memory.offHeap.size & -XX:MaxDirectMemorySize as larger as possible.
3. keep spark.sql.shuffle.partitions equal to the cores usually can get better performance, but may raise the risk for out of memory issue.

If there are lots of fallback for row based processing, you may have to reserve some memory space for spark.executor.memory.

In some cases, use row based operators may run faster than columnar based operators.
One example is when run a 6+ continuous joins in a query, users can use spark.oap.sql.columnar.joinOptimizationLevel to keep this case using row based processing.

LIMITATIONS & OVERHEAD

- Only pass TPC-H and TPC-DS testing.
- Not all the operators & data types are supported, please check the operator support list.
- Still not support some SPARK built-in features such as RDD Cache, Spilling, UDF, ...etc.
- Codegen overhead when running at 1st run.

AGENDA

Motivation

Core Design
- Architecture
- Arrow Data Source
- Native SQL Engine
- Columnar Shuffle

Getting Started

Performance

Summary

CURRENT STATUS

Decision Support Benchmark1 (TPC-H Like)

- Pass all 22 queries
- No fallback to row-based processing
- 1.49X performance boost in total time under 1.5TB data scale

Decision Support Benchmark2 (TPC-DS Like)

- Pass all 99 queries
- Partially fallback to row-based processing
- Up to 3.2X performance boost in a single query under 1.5TB data scale

CURRENT STATUS

OAP v1.0 has been released on January 2021

- Pass all 22 TPC-H Like Queries(not support Decimal)

OAP v1.1 is planning to be released on April 2021

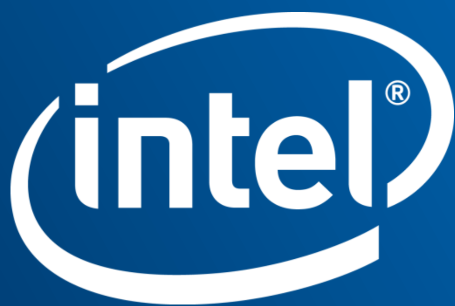
- Pass all 99 TPC-DS Like Queries
- Support Decimal
- Reduce the times to fallback to row-based processing
- Arrow 3.0 Integration

FUTURE PLAN

OAP v1.2 is under developing(target on Q3, 2021)

- Pass more Real World workloads
- RDD Cache Support
- Spilling for Sort
- UDF
- Arrow 4.0 Integration

Q&A


intel