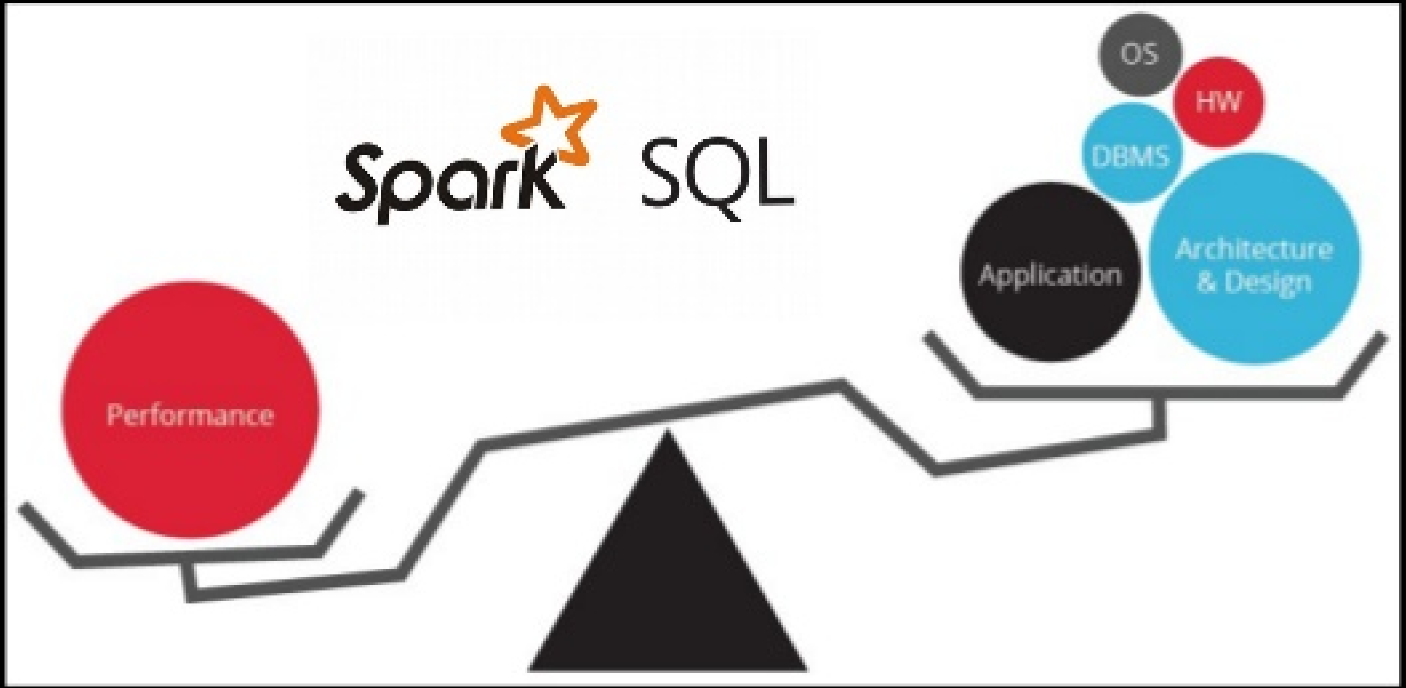
Spark SQL Catalyst Code Optimization Using Function Outlining



Kavana N Bhat, Senior Systems Developer &
Madhusudanan Kandasamy, STSM, PowerAI
Power Systems, IBM

#SAISDD15

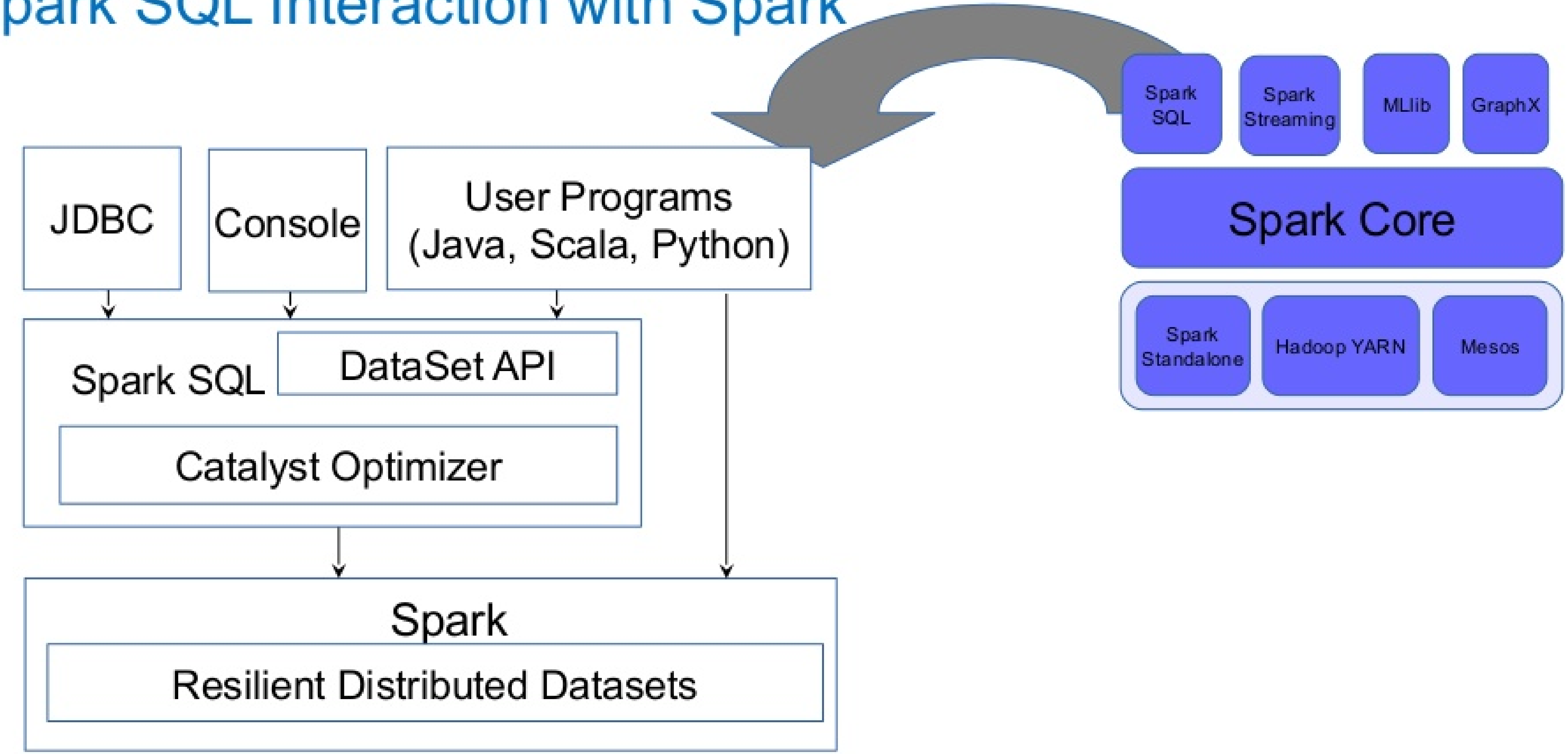
About Presenters:

- Kavana N Bhat
  - Senior Systems Developer at IBM
  - Working for IBM Power Systems over 14 years
    - AIX OS Development
    - Apache Spark & Alluxio Optimization for Power Systems
    - ML/DL Frameworks for Enterprise Systems
  - Has 5 Patents, 5 Disclosure Publications
  - Email: kavana.bhat@in.ibm.com
- Madhusudanan Kandasamy
  - PowerAI STSM at IBM Systems
  - Working for IBM Power Systems over 15 years
    - AIX & Linux OS Development
    - Apache Spark Optimization for Power Systems
    - Distributed ML/DL Framework with GPU & NVLink
  - IBM Master Inventor (20+ Patents, 18 Disclosure Publications)
  - Github: https://github.com/kmadhugit Email: madhusudanan@in.ibm.com

Outline

- Spark SQL Overview
- Catalyst Optimizer in Depth
- Java JIT Compiler Overview
- Spark Performance Profile of TPC-DS Benchmark Query
- Function Outlining
- Catalyst Code Generation Optimizations
- Performance Statistics
- Conclusion
- References

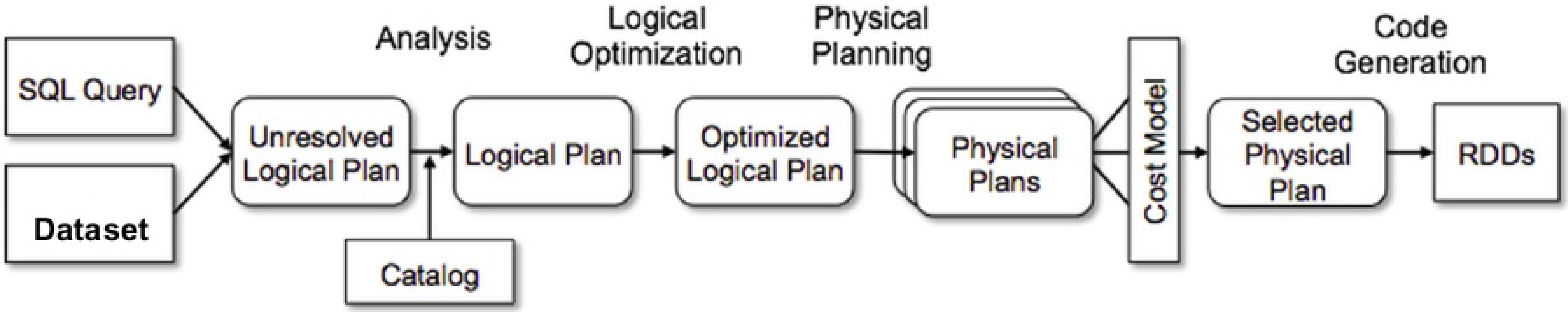
Spark SQL Interaction with Spark



Spark SQL Architecture

Terminology:

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



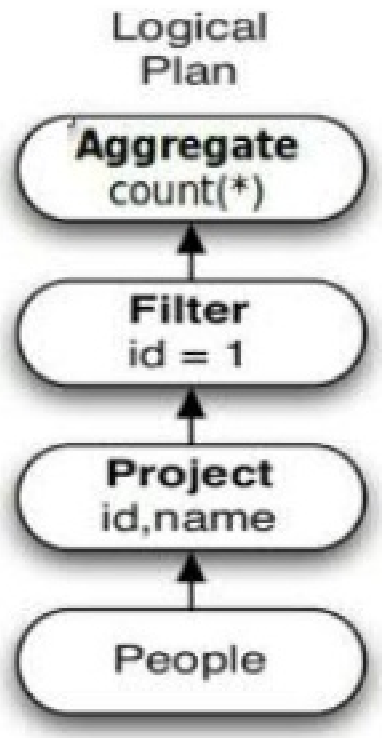
Catalyst Optimizer Phases

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

Logical Plan Optimization Example

- An example query

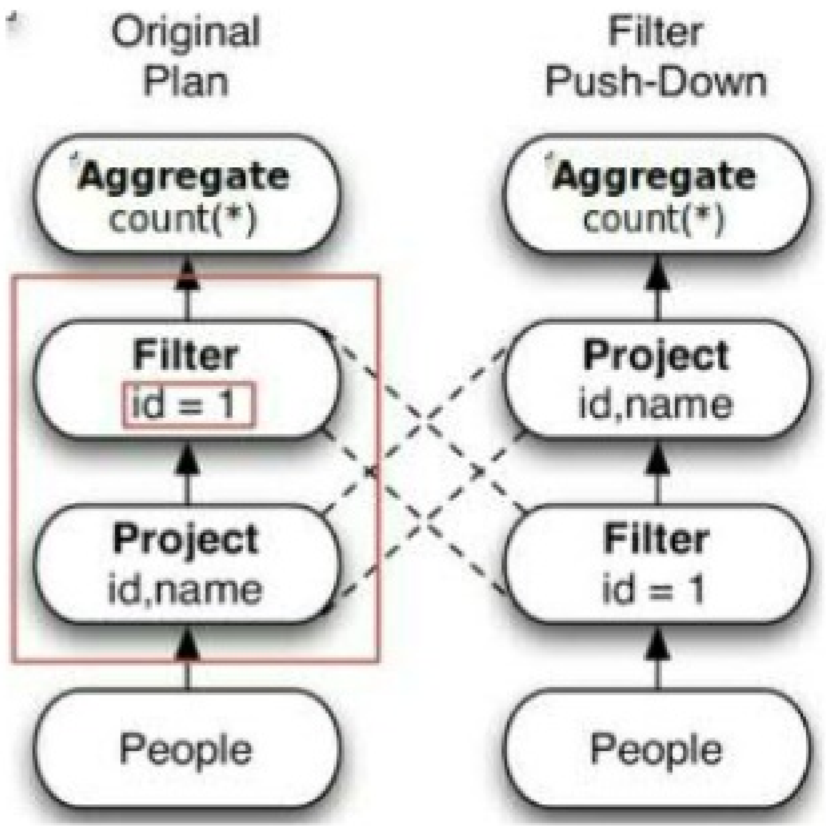
```
SELECT count(*)
FROM (
    SELECT id, name
    FROM People ) p
WHERE p.id = 1
```



Logical Plan Optimization Example (Contd..)

- Optimization Rules example

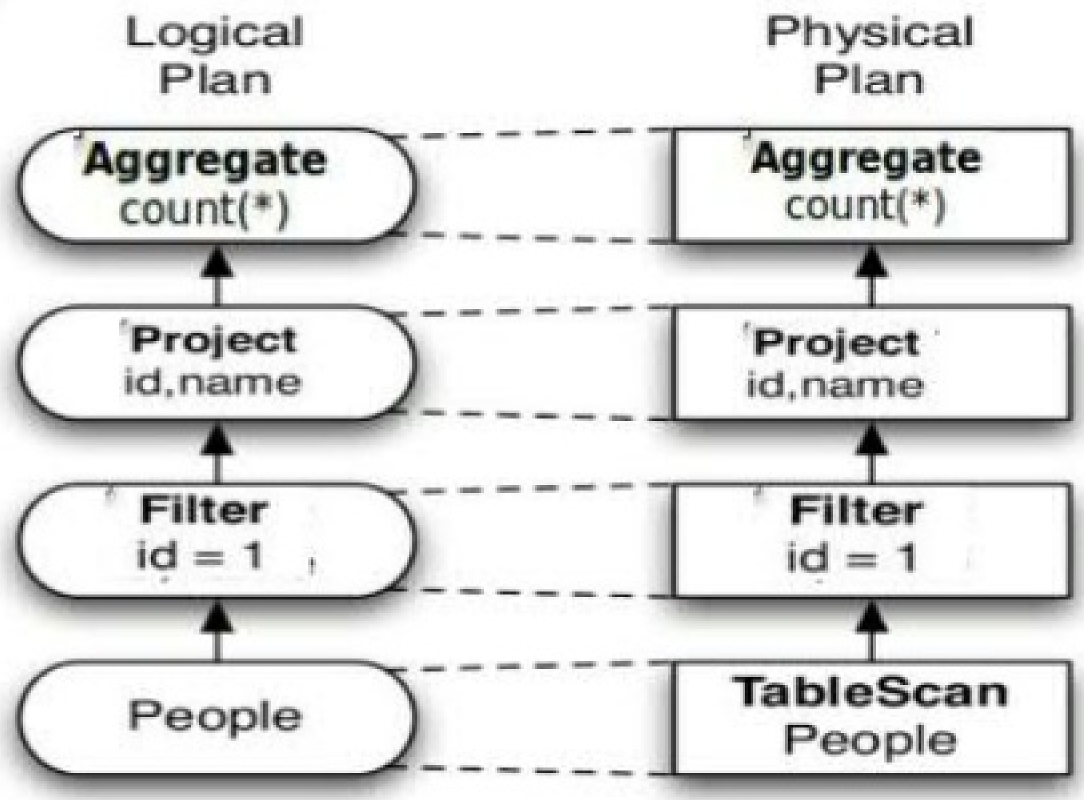
1. Find Filters on top of projections.
2. Check that the filter can be evaluated without the result of the project.
3. If so, switch the operators.



Physical Planning

- Native query planning

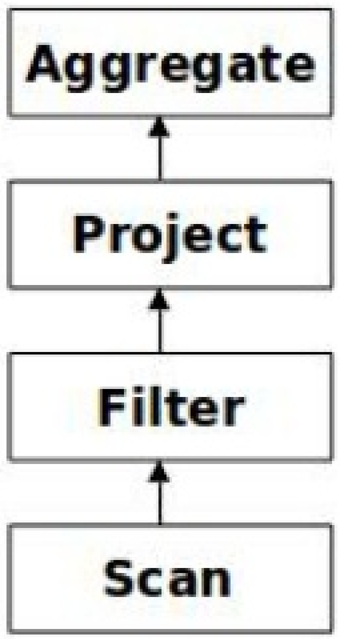
```
SELECT  count(*)
FROM  (
      SELECT  id,  name
      FROM  People )  p
WHERE  p . id  =  1
```



Catalyst Code Generation

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

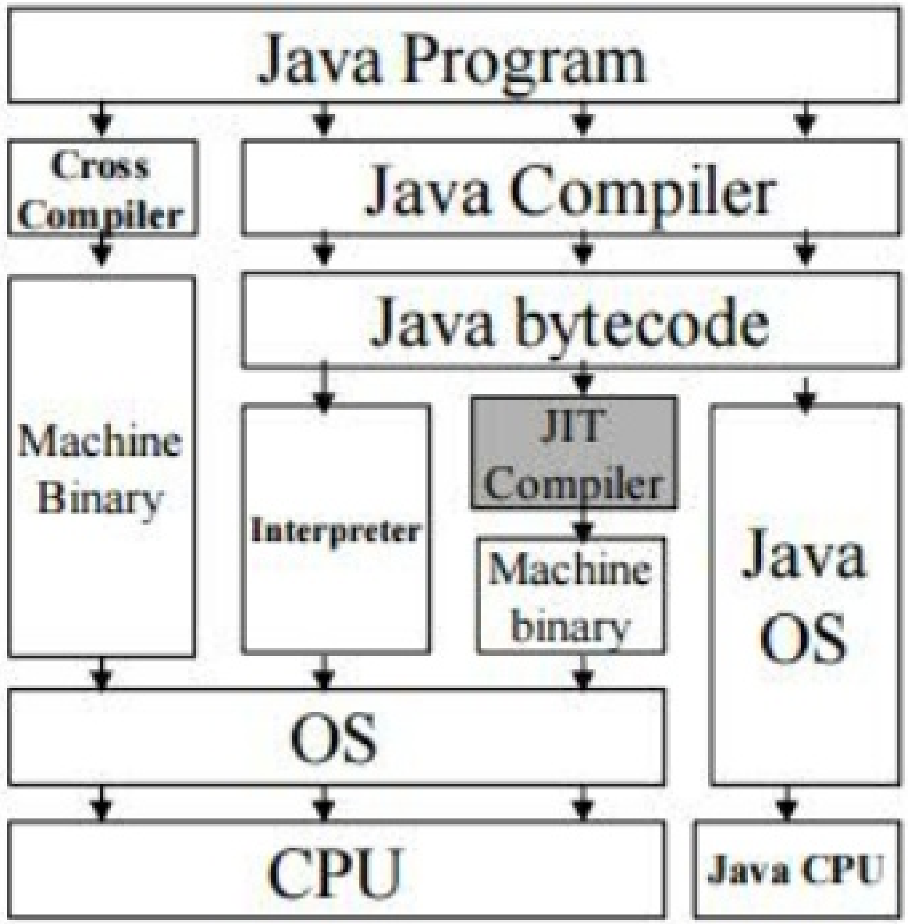
```
int count = 0;
for (Row r : rows) {
   if (<filter>) {
       count++;
   }
}
return new Row(count)
```



Traditional Java Compilation and Execution

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

Java JIT Compiler Overview:



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

Spark Performance Profile of TPC-DS Benchmark Queries

- TPC-DS is the de-facto industry standard benchmark for measuring the performance of SQL queries
- Performance profile of TPC-DS query #72 showed one of the catalyst generated routine to be hot.
- Logs from JVM PrintCompilation flag for the workload were analyzed
  - This flag shows basic information on when Hotspot compiles methods
- The identified hot method was reported as ‘COMPILE SKIPPED’
  - Indicates the method was identified by JVM as hot, but JIT failed to compile.
  - One reason could be the code cache is full
- The job-stage-task which took the longest time was identified
- The catalyst code modified to log the dynamically generated java code with file names tagged with job/stage/task Id.
- Detailed look at the catalyst generated code for the hot method indicated that it was a huge routine (>300 java LOC)


```
207669 12478 %    3    org.apache.spark.sql.catalyst.expressions.GeneratedClass$GeneratedIterator::processNext @ 59 (1605 bytes)
207681 12478 %    3    org.apache.spark.sql.catalyst.expressions.GeneratedClass$GeneratedIterator::processNext @ 59 (1605 bytes)   COMPILE SKIPPED: CodeBuffer overflow (retry at different tier)
207697 12481 %    4    org.apache.spark.sql.catalyst.expressions.GeneratedClass$GeneratedIterator::processNext @ 59 (1605 bytes)
207720 12482      4    org.apache.spark.sql.catalyst.expressions.GeneratedClass$GeneratedIterator::processNext (1605 bytes)
207758 12483 %    4    org.apache.spark.sql.catalyst.expressions.GeneratedClass$GeneratedIterator::processNext @ 59 (1605 bytes)
207949 12486 %    4    org.apache.spark.sql.catalyst.expressions.GeneratedClass$GeneratedIterator::processNext @ 59 (1605 bytes)
208282 12497 %    4    org.apache.spark.sql.catalyst.expressions.GeneratedClass$GeneratedIterator::processNext @ 59 (1605 bytes)
208886 12513 %    4    org.apache.spark.sql.catalyst.expressions.GeneratedClass$GeneratedIterator::processNext @ 59 (1605 bytes)
209538 12516      4    org.apache.spark.sql.catalyst.expressions.GeneratedClass$GeneratedIterator::processNext (1605 bytes)
1485428 12926     4    org.apache.spark.sql.catalyst.expressions.GeneratedClass$GeneratedIterator::processNext (1605 bytes)
197821 12391 %    3    org.apache.spark.sql.catalyst.expressions.GeneratedClass$GeneratedIterator::processNext @ 59 (1605 bytes)
197842 12391 %    3    org.apache.spark.sql.catalyst.expressions.GeneratedClass$GeneratedIterator::processNext @ 59 (1605 bytes)   COMPILE SKIPPED: CodeBuffer overflow (retry at different tier)
```

Function Outlining

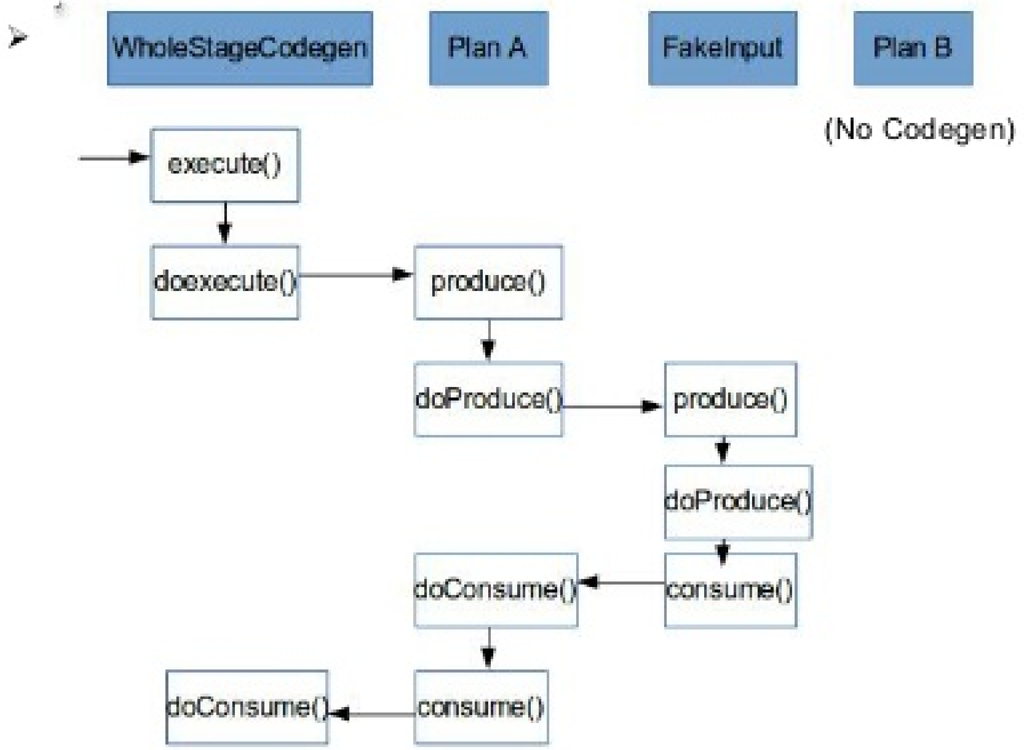
- It is a technique that splits a code region into a new, independent function and replaces it with a function call to new function.
- Benefits :
  - JVM loves small methods– can improve performance as they might enable more inlining
  - Improve code locality.
- Drawbacks :
  - Extra function calls are introduced to transfer control between the outlined region and the other parts of the program unit.

Catalyst Code Generation Optimizations

- Query #72 used Broadcast Hash Join/Sort Merge Join
- The Spark catalyst code generation for the identified hot routine was modified to generate the same logic using multiple small routines.
- Some of the logic pulled into smaller routines referenced some locals of the initial routine.
- The issue was resolved by making such local variables to be class private variables.
- The performance was analyzed with TieredCompilation enabled IBM JDK

Catalyst Code Generation Optimizations – High Level Code Modifications

- WholeStageCodegen compiles a subtree of plans that support codegen together into single java function
- Three code generation paths:
  - Non-whole-stage-codegen path
  - Whole-stage-codegen "produce" path - Java code that reads the rows from the input RDDs
  - Whole-stage-codegen "consume" path - Java source code to consume the generated columns or a row from a physical operator.



Call Graph to generate Java Code

sql/core/src/main/scala/org/apache/spark/sql/execution/joins/SortMergeJoinExec.scala

```
override def doProduce(ctx: CodegenContext): String = {
………………
   val forLoop_sub = ctx.freshName("forLoop")
   ctx.addNewFunction(forLoop_sub,
    s"""
       |private void $forLoop_sub(scala.collection.Iterator<UnsafeRow> iterator)
throws java.io.IOException {
       | while ($iterator.hasNext()) {
       |   InternalRow $rightRow = (InternalRow) $iterator.next();
       |   ${condCheck.trim}
       |   $numOutput.add(1);
       |   ${consume(ctx, leftVars ++ rightVars)}
       | }
       |}""".sstripMargin)
   s"""
      |while (findNextInnerJoinRows($leftInput, $rightInput)) {
      | ${beforeLoop.trim}
      | scala.collection.Iterator<UnsafeRow> $iterator =
$matches.generateIterator();
      | $forLoop_sub($iterator);
      | if (shouldStop()) return;
      |}
    """.stripMargin
```

Catalyst Code Generation Optimizations – A peek into catalyst generated code

Original Catalyst Generated Code

```
/* 001 */ public Object generate(Object[] references) {
/* 002 */   return new GeneratedIterator(references);
/* 003 */ }
/* 004 */
/* 005 */ final class GeneratedIterator extends
org.apache.spark.sql.execution.BufferedRowIterator {
......
/* 317 */   protected void processNext() throws java.io.IOException {
/* 318 */     if (scan_batch == null) {
/* 319 */       scan_nextBatch();
/* 320 */     }
/* 321 */     while (scan_batch != null) {
/* 322 */       int numRows = scan_batch.numRows();
/* 323 */       while (scan_batchIdx < numRows) {
........
/* 360 */         scala.collection.Iterator bhj_matches = bhj_isNull ? null :
(scala.collection.Iterator)bhj_relation.get(bhj_value);
/* 361 */         if (bhj_matches == null) continue;
/* 362 */         while (bhj_matches.hasNext()) {
/* 363 */           UnsafeRow bhj_matched = (UnsafeRow)
bhj_matches.next();
........
/* 633 */         }
/* 634 */         if (shouldStop()) return;
/* 635 */       }
/* 636 */       scan_batch = null;
/* 637 */       scan_nextBatch();
/* 638 */     }
/* 639 */     scan_scanTime.add(scan_scanTime1 / (1000 * 1000));
```

Modified Catalyst Generated Code

```
/* 001 */ public Object generate(Object[] references) {
/* 002 */   return new GeneratedIterator(references);
/* 003 */ }
/* 004 */
/* 005 */ final class GeneratedIterator extends
org.apache.spark.sql.execution.BufferedRowIterator {
......
/* 273 */   private void scan_processNextSub() throws java.io.IOException {
/* 274 */     int numRows = scan_batch.numRows();
......
/* 305 */       bhj_genJK();
/* 306 */       if (bhj_matches == null) continue;
/* 307 */       bhj_wloop();
/* 308 */       if (shouldStop()) return;
/* 309 */     }
/* 310 */   }
....
/* 873 */   protected void processNext() throws java.io.IOException {
/* 874 */     if (scan_batch == null) {
/* 875 */       scan_nextBatch();
/* 876 */     }
/* 877 */     while (scan_batch != null) {
/* 878 */       scan_processNextSub();
/* 879 */       scan_batch = null;
/* 880 */       scan_nextBatch();
/* 881 */     }
/* 882 */     scan_scanTime.add(scan_scanTime1 / (1000 * 1000));
/* 883 */     scan_scanTime1 = 0;
/* 884 */   }
```

Performance on Power8 cluster with IBM JDK

IBM JDK version	Time Taken in min - Default Spark	Time Taken in min – Catalyst AutoSplit Code	Improvement in %
Version SR3	56	45	19
Version SR5	42	38	9.5

Conclusion

- Function outlining of huge routines proves to be beneficial when
  - It is very heavily used
  - JVM is unable to accommodate the compiled code into code cache.
- Automatic split of huge routines at source may/may not lead to improvements in all cases due to additional branches
- JIT compilers need to evaluate automatic use of function outlining as needed for hot functions.
- Monitoring and tuning JVM code cache and JIT compiler warm-up period can help tune Spark application performance
  - Use of Tiered Compilation. Option : -XX:+TieredCompilation
  - Maximum code cache size (in bytes) for JIT-compiled code. Option : -XX:ReservedCodeCacheSize=<size>
  - Number of interpreted method invocations before compilation.
    - In Server JVM, default is 10,000 and Client JVM, its 1,500. Option: -XX:CompileThreshold=<invocations
  - Compiler threads to use for compilation
    - By default, is set based on the number of cores. Option: -XX:CICompilerCount=threads
  - Useful Logging Options - Option: -XX:+LogCompilation or -XX:+PrintCompilation

References

- Good Reads for Spark SQL's Catalyst Optimizer
  - Spark SQL: Relational Data Processing in Spark by Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, Matei Zahari
  - Deep Dive into Spark SQL's Catalyst Optimizer by Michael Armbrust, Yin Huai, Cheng Liang, Reynold Xin and Matei Zaharia
  - Deep dive into the new Tungsten execution engine by Sameer Agarwal, Davies Liu and Reynold Xin
  - Deep Dive Into Catalyst: Apache Spark's Optimizer by Yin Huai
  - Cost-Based Optimizer in Apache Spark 2.2 By Ron Hu, Zhenhua Wang, Sameer Agarwal, Wenchen Fan
- Github link to log all dynamically generated catalyst code:
  https://github.com/kmadhugit/spark/tree/catalyst_debug_v2.1.0-rc5
- Github link to Automatic Catalyst Code Split : https://github.com/kavanabhat/spark/tree/catalyst-autosplit

Questions?

Thank You