Processing Java UDFs in a C++ Environment

Viktor Rosenfeld, Rene Mueller, Pinar Tözün, Fatma Özcan

IBM Research – Almaden

Wildfire
- Distributed HTAP system
- Columnar, pipelined query execution engine
- Written in C++
- Spark as user-facing front end
- Data analytics with SparkSQL

Java Native Interface
- Standard way to connect JVM with native code
- JNI calls have significant overhead
- Strided execution hides overheads

Class
Execute Scala UDFs

public sqlContext.sql(sqlContext.udf.register("add_offset", (i: Int) => i + offset)).show()

Class representation (simplified)

public final class SparkProgramm$$anonfun$run$1 extends scala.runtime.AbstractFunction1$mcII$sp implements scala.Serializable {
  public SparkProgramm$$anonfun$run$1(scala.runtime.IntRef);
  public final int apply(int);
}

Execution in Embedded JVM
- Strided execution wrapper compiled transparently
- Engine buffers wrapped as Java direct ByteBuffers
- Comparable performance to execution in Spark and as SQL statement

Spark SQL UDFs
- Represented as Java classes
- SparkSQL UDFs are closures
- Free variables captured in class instance

Usage

var offset = 10
sqlContext.sql("SELECT add_offset(i) FROM table").show()

JIT compilation to machine code
- UDF bytecode translated to LLVM IR with BugVM
- Object code dynamically loaded and executed
- Beneficial for computationally heavy UDFs that do not create objects
- Optimizations to speed up UDFs that create objects violate Java language guarantees

Contributions
- We transparently enable strided execution of tuple-based Java UDFs in a C++ query engine.
- The performance of our solution is comparable to execution in Spark and UDFs hand-written in C++.
- Our analysis shows that compiling UDFs to machine code has only marginal benefits.