Processing Java UDFs in a C++ Environment

Viktor Rosenfeld, Rene Mueller, Pınar Tözün, Fatma Özcan
IBM Research – Almaden

SoCC '17, Santa Clara – September 27, 2017
Programming languages split

JVM-based data analytics ecosystem

liberal use of UDFs

Software components that do not use the JVM

“native code”

**Libraries and Tools:**
- Apache Hadoop
- Apache Spark
- Apache Flink
- VERTICA
Wildfire

- HTAP prototype from IBM Research [CIDR2017]
- C++ columnar engine
- Spark as user-facing front end
- data analytics through SparkSQL queries
- SparkSQL queries can contain Scala UDFs

**Goal:** Execute Scala UDFs inside SparkSQL queries on the Wildfire C++ engine.
Java Native Interface
JNI overheads

- Simple Java function called in a loop
- 1 billion iterations
- JNI overhead: 2 orders of magnitude
- Splitting the loop (strided execution) hides the overhead

```java
int udf(int i) {
    return i;
}
```

- JNI calls have significant overhead
- Execute tuple-based UDF in a strided fashion
SparkSQL UDFs

Usage in Spark

```scala
var offset = 10
sqlContext.udf.register("add_offset", (i: Int) => i + offset)
sqlContext.sql("SELECT add_offset(i) FROM table").show()
```

Java class representation

```java
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);
    ...
}
```

SparkSQL UDFs represented both as Java class and instance.
Embedded JVM

- Instantiate UDF in embedded JVM
- JIT-compile strided execution wrapper
- Wrap engine buffers as Java direct ByteBuffers
Embedded JVM performance

- Comparable performance to execution in Spark
- Strided execution is key to fast performance
Can we do better?
JIT compilation to machine code

- Compile bytecode to LLVM IR
- Generate wrapper LLVM IR
- Link both to object code
- Deserialize instance
- Dynamically load and execute object code
Compilation to machine code is beneficial if UDF is computationally heavy and does not create objects.
Word length UDF optimizations

UDF wrapper

for $i \leftarrow 1$ to size of input do
  javaString $\leftarrow$ CreateJavaString($input_i$)
  output$_i$ $\leftarrow$ WordLengthUdf(javaString)
  CheckForJavaException()
  ReleaseJavaObject(javaString)
end

Optimizations

1. Reuse javaString object
2. Eliminate data copies when creating javaString
3. Remove memory fence
4. Relax exception handling

These optimizations violate Java language guarantees (e.g., immutability of Strings)! We can apply them because we know that the UDF does not leak or retain the String object.
These optimizations violate Java language guarantees!
Summary

- Transparent strided execution of tuple-based SparkSQL UDFs inside a C++ engine
- Comparable performance to execution in Spark
- Java direct ByteBuffers simplify passing data between C++ code and the embedded JVM
- Compilation to machine code is beneficial if UDF is computationally heavy and does not create objects
Backup
Comparison with SQL

Range query with UDF predicate

``` scala
-- def filter(a: Int, low: Int, high: Int)
--       = low < a && a < high
SELECT sum(a) FROM R
WHERE filter(a, min, max)
```

Range query with SQL predicate

``` sql
SELECT sum(a) FROM R
WHERE min < a AND a < max
```
Embedded JVM performance

- 2.5B integers (10 GB)
  - Stride size 4096
- 64M points (1 GB)
  - Stride size 512
- 80M words (1 GB)
  - Stride size 1024
  - Word length follows poisson distribution

- Intel Xeon X7560, 2.27 GHz, 512 GB RAM
- Oracle JDK 112
- Data stored as uncompressed Parquet files with RLE/PLAIN encoding

Wall-clock time [relative to Spark-only]

Range query (bandwidth-bound)

Vincenty formulae (compute-bound)

Word length (object creation)
State of the art

- Impala: supports C++ UDFs (fast) and unmodified Hive UDFs (tuple-based, slow, discouraged)
- Vertica: *User-defined Transform Functions* that process multiple rows via an iterator
- Tupleware: UDFs in LLVM-based languages
Wildfire system architecture

Analytics
- tolerate slightly stale data

Analytics
- require most recent data

Transactions
- high volume

Spark applications

Spark executor
- Wildfire engine

Spark executor
- Wildfire engine

Spark executor
- Wildfire engine

Spark executor
- Wildfire engine

Spark executor
- Wildfire engine

Shared file system
Java direct ByteBuffer

- Wraps memory that is not managed by the JVM
- Access in Java through typed getter and setter methods
- JVM will make “best effort” to avoid unnecessary data copies
- Primitive types: equivalent to typed array
- String objects: data must be copied into temporary array
public class StridedExecutionWrapper {
    public static void wrapUdf(
        UdfClass udfInstance,
        int numRows,
        ByteBuffer output,
        ByteBuffer auxiliaryOutput,
        ByteBuffer[] inputs,
        ByteBuffer[] auxiliaryInputs) {
        for (int i = 0; i < numRows; ++i) {
            output.put(udfInstance.apply(
                inputs[0].get(), … ,inputs[N].get()));
        }
    }
}
Supported types

**Supported**

- Primitive types (no extra copies)
- Strings (copy into temporary array)

**Not supported in prototype**

- User-defined types require object creation (straightforward if type can be decomposed into primitives)
- Complex nested types cannot be used with scalar UDFs
Security considerations

• UDFs must not crash database
• typically executed in separate process
• JVM offers similar separation
• errors conditions are signalled as exceptions
• UDFs can further be restricted through Java SecurityManagers
Thread scaling

- SQL predicate
- UDF predicate
Modified Java String implementation

**Default implementation**

- **Engine**
  - Auxiliary buffer

- **BugVM JVM**
  - String Object
    - Length
    - Char Array

  - set
  - copy

  2 Object allocations

**Reuse string buffer**

- **Engine**
  - Auxiliary buffer

- **BugVM JVM**
  - Reused String Object
    - Length
    - Value

  - set
  - copy

  1 Object allocation

**Wrap engine buffer**

- **Engine**
  - Auxiliary buffer

- **BugVM JVM**
  - Custom String Object
    - Length
    - Pointer

  - set
  - wrap