Selective Python UDF evaluation inside the JVM

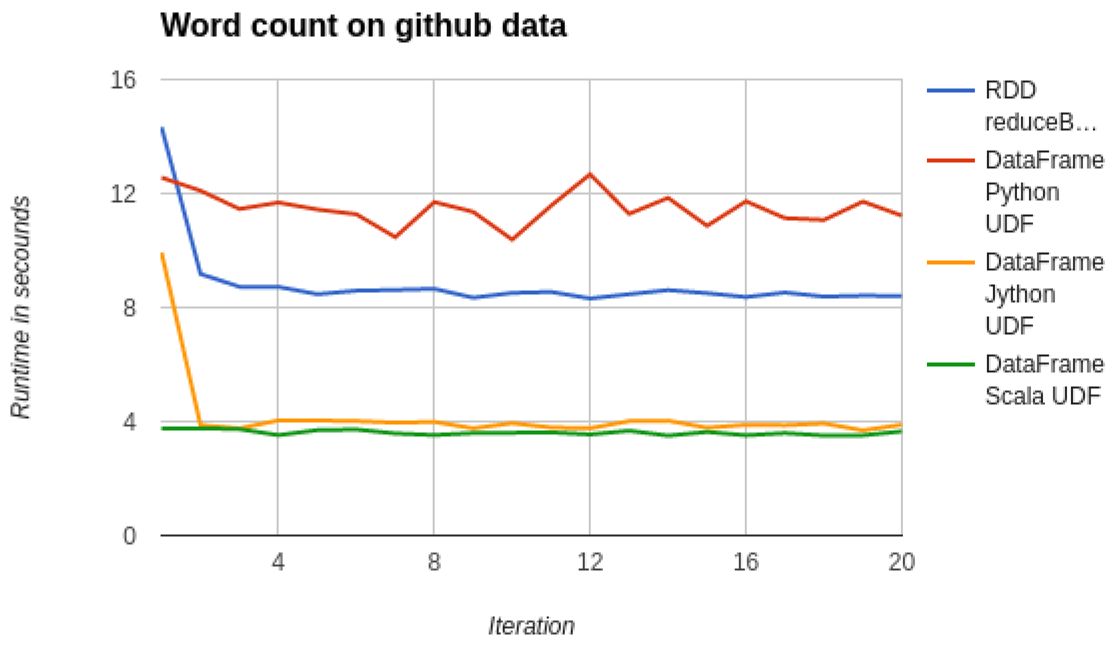
Linked JIRA: SPARK-15369 & PR

Background

PySpark has historically lagged behind Scala & Java Spark in terms of execution speed due in a large part to the need of copying data from the JVM on the executor to the Python worker process. The DataFrame API allows many transformations in Python to be executed directly inside of the JVM, e.g. when the result is specified using SQL expressions, however for expressions using arbitrary Python code (e.g. UDFs) the performance bottleneck remains. While not appropriate for all Python code, or even all Python UDF code, there is a 2.7 Python VM available to be used inside of the JVM which avoids the serialization and copying costs associated with the traditional PySpark RDD model.

Performance Comparison

Since this is a proposed change to improve performance, debating the API details doesn't matter much if the performance difference isn't worth the overhead of maintaining the changes. Using an early draft (from holden/spark/SPARK-15369-investigate-selectively-using-jython) & a simple wordcount benchmark found Jython UDF to be 3x faster than Python UDF and ~2% slower than a native Scala UDF for multiple runs. The first run with a Jython UDF involves starting the Jython interpreter on the workers, but even in those cases it outperforms regular PySpark UDFs by ~20%.



Proposed API

Since Jython will not be a suitable drop in replacement for all Python UDFs we need to either automatically detect the situations where is appropriate or have the user specify when they want their UDF to run in Jython.

Detecting which code can run in Jython could be done by trying to run the code Jython and then falling back to regular Python if the Jython code throws an exception. This approach would slow down UDFs which can't be run in Jython. Alternatively we could use introspect or similar to attempt to determine if the users function can be run in Jython, but this would require a substantial investment in creating automated rules for which functions are suitable for running in Jython.

A simpler option is having a separate entry point (e.g. registerJythonFunction) which the user can call explicitly to signal that their UDF should be run in Jython. This approach will not automatically speed up old code, but also benefits from not breaking any existing functionality if we fail to create the correct rules for what should be run in Jython versus regular Python.

Implementation Options

In the driver program, serialization of UDFs serialize the Python bytecode and frames across the wire, requiring the same version of Python on the workers as the driver. Since the driver will be running in regular Python any UDFs running in Jython must find a different way to serialize the closure representing the UDF. For running the UDF itself there are multiple options for how to package/compile our Python code - ranging from the Jython specific interfaces to the generic javax.scripting interface from JRS 223 as well how to expose the Jython UDF to Spark (subclass of ScalaUDF or separate UDF class).

Starting with the driver side challenge, there are a few options for how to solve the serialization problem. The existing serialization serializes the Python byte code and frames which doesn't work across different Python versions, and since the driver program runs in regular Python this will not work for Jython UDFs. One option is to add an optional dependency on the dill library to automatically determine the source code associated with the lambda. The other existing serialization tools to determine what closure elements can be reused. However dill does not work with all source code, another option would be having the user specify a string with the provided lambda function.

Creating the class to evaluate the Python code has a few options. One of the simpler more general approaches is to use javax.scripting interface, which is already being used in the EclairJS project. Early versions of Jython provided the option of compiling Python code to a native Java class, but this interface is no longer supported. We can also use Jython's specific scripting interface in place of the generic Java class.

Implementing the UDF can be done in a few ways. The simplest way is subclassing the Scala UDF as it already has conversions from InternalRow to the types expected by the Jython API. Alternatively we could implement our own UDF class which would give us more control over codegen.

In the Python side chaining UDFs is very important as round tripping can be quite expensive. Similar chaining could be implemented in Jython with the potential to save some boxing/unboxing cost - however since Jython performance is already so close to Scala performance it is unlikely that this would bring a substantial benefit.

Dependencies

Python UDF evaluation inside of the JVM depends on Jython (PSF license[1]), however not all deployments will be utilizing this functionality. The Jython standalone JAR is ~36mb (of 207mb total for Spark with Jython). We could have this turned on with a flag or included by default. Another (possible) dependency is dill (BSD licensed), however dill is only required on the driver side and can already be installed with pip easily. Depending on system dill could result in differences between instances, however if we wish to avoid this we could package it in the same way we package py4j.

Draft Implementation

A simple draft implementation exists at holden/spark/SPARK-15369-investigate-selectively-using-jython and is implemented with a separate “registerJythonFunction” for users to call as well as supporting both dill based and explicit string passing for lambda functions with the Scala side UDF existing as a subclass of ScalaUDF.

Related Projects

- Jython
- Eclair JS
- Dill
- JyNI (Jython Native interface)

[1] Note: allowed to be included with Apache projects - http://www.apache.org/legal/resolved.html :)