SparkCLR

Contents

[Overview 2](#_Toc432104425)

[Rationale 2](#_Toc432104426)

[Design Goals 3](#_Toc432104427)

[Development Experience 3](#_Toc432104437)

[Word Count Example in Scala & C# 3](#_Toc432104438)

[Log Processing Sample 3](#_Toc432104439)

[RDD Sample 4](#_Toc432104440)

[DataFrame DSL Sample 4](#_Toc432104441)

[DataFrame TempTable Sample 5](#_Toc432104442)

[Schema Specification Sample 5](#_Toc432104443)

[Performance Considerations 5](#_Toc432104444)

[Implementation 5](#_Toc432104445)

[Driver-side Adapter 5](#_Toc432104446)

[RDD 5](#_Toc432104447)

[Executor - RDD 6](#_Toc432104448)

[DataFrame 7](#_Toc432104449)

[Executor – DataFrame 7](#_Toc432104450)

[Streaming 7](#_Toc432104451)

[Other Topics 8](#_Toc432104452)

[Reuse of code 8](#_Toc432104453)

[Serialization in C# 8](#_Toc432104454)

[Cluster Manager 8](#_Toc432104455)

[External dependencies 9](#_Toc432104456)

[License 9](#_Toc432104457)

[Background & Contributions 9](#_Toc432104458)

[Open Questions & TBD 9](#_Toc432104459)

[Links 9](#_Toc432104460)

[Samples 10](#_Toc432104461)

[Pi - RDD 10](#_Toc432104462)

[Word count - RDD 11](#_Toc432104463)

[Log processing - RDD 11](#_Toc432104464)

[Serializable object processing - RDD 11](#_Toc432104465)

[Register Temp Table - DataFrame 12](#_Toc432104466)

[Project & Filter - DataFrame 13](#_Toc432104467)

[Join – DataFrame 13](#_Toc432104468)

[Aggregate – DataFrame 14](#_Toc432104469)

# Overview

SparkCLR (pronounced like “Sparkler”) adds C# language binding to Apache Spark enabling the implementation of Spark driver code and data processing operations in C#. With SparkCLR developers can implement C#-based Spark applications for batch, interactive and stream processing making C# a first-class citizen for Spark app development.

# Rationale

SparkCLR will lower the barrier for organizations that use C# as the primary application development language to start building Spark applications without having to learn/invest in languages like Scala, Java, Python or R that are currently supported in Spark. Organizations that have invested heavily in .NET can possibly leverage their existing libraries in Spark applications built using SparkCLR. With SparkCLR it should be possible to develop apps in C# for Spark deployed to private clusters/cloud, Windows-based VMs in Azure or AWS. With Mono or .Net Core it may be possible to support SparkCLR for Spark clusters in Linux as well.

# Design Goals

# In Spark executors, CLR should be launched during execution if & only if needed

# User-implemented custom code in C# will run outside of Spark JVM -- in CLR

# Any Spark stage that does not involve user-implemented custom code in C# should just use built-in Spark functionality & the execution should be limited to JVM.

# For example, the following scenarios do not involve executing custom C# code and hence execution will not involve CLR

# Creating a RDD from a text file and performing row count

# Projection, filtering, joining in DataFrames that do not involve C# UDF/UDAF

# Re-use (design & code) from other language binding implementations in Spark (SparkR & PySpark)

# Maintain fidelity with Spark public API in Scala, Java (and R, Python)

# Build SparkCLR as an extension to Spark

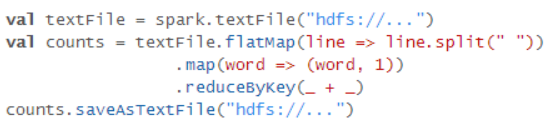
* + Spark bits should just be dependencies. If needed contribute something to Spark to stay as an extension to it.

# Development Experience

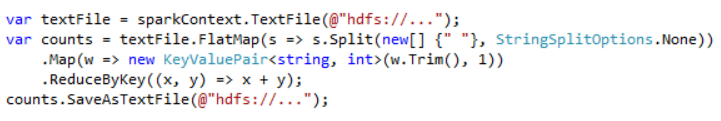
## Word Count Example in Scala & C#

SparkCLR API will be similar to that of Scala API for Spark but C# style with.NET conventions. Shown below are word count examples in Scala and C#.

**Driver program in Scala**



**Driver program in C#**



## Log Processing Sample

Scenario – This sections covers a sample scenario involving join of data in two log files in csv format using guid column and computing max and average latency metrics grouped by data center

The columns in the first log file “Requests.csv” is as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| guid | Datacenter | abtestid | Traffictype |

The columns in the first log file “Metrics.csv” is as follows:

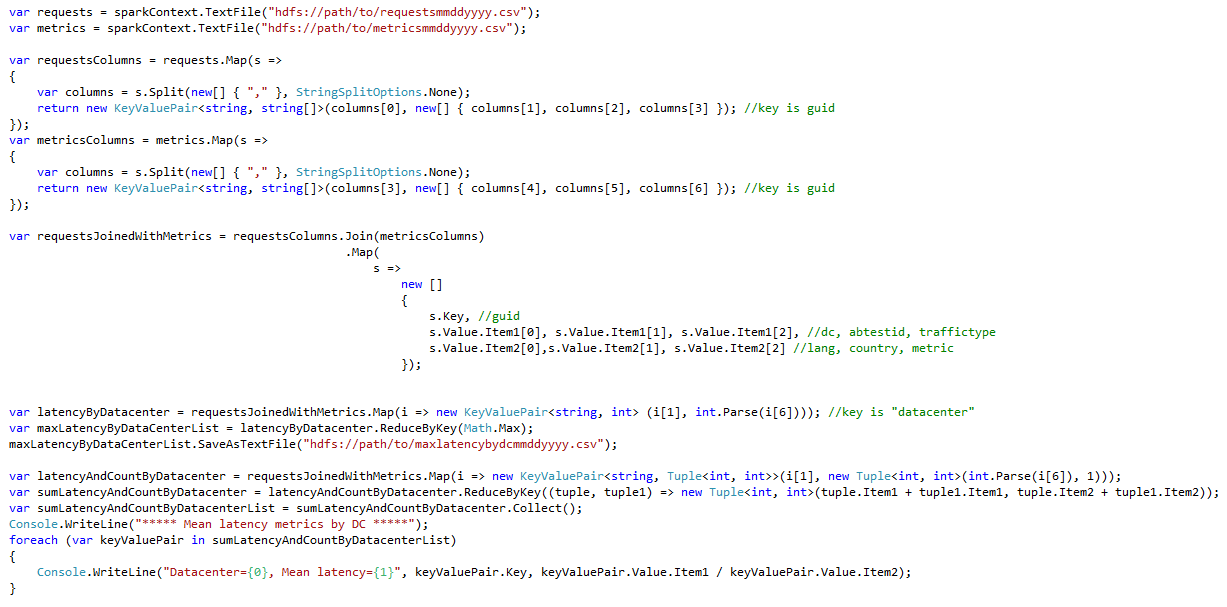
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Unused | date | Time | guid | lang | country | latency |

The RDD-based implementation is shown below and it is more verbose than the DataFrame-based implementations shown below. DataFrame implementation has two categories

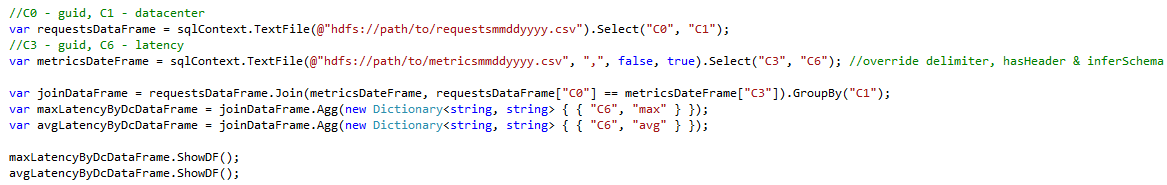
1. DataFrame DSL based (using methods like Select, Where, Join, Agg on DataFrame)
2. DataFrame TempTable based (register the DataFrame as TempTable for SQL queries)

Note that RDD sample involves C# implementation in Map, Join and ReduceByKey methods on C# RDD object. At executors CLR is launched to execute these operations. DataFrame samples do not involve C# UDFs. So CLR will not be launched at the executors and execution behavior will be exactly same as that of Scala-based DataFrame execution.

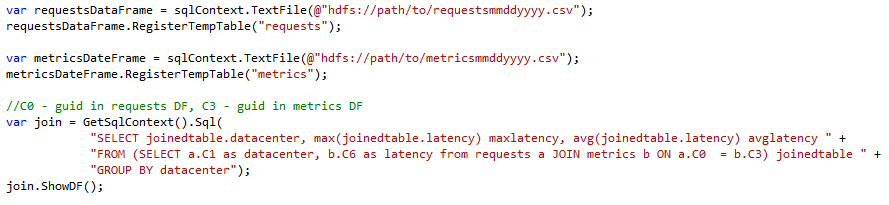
### RDD Sample



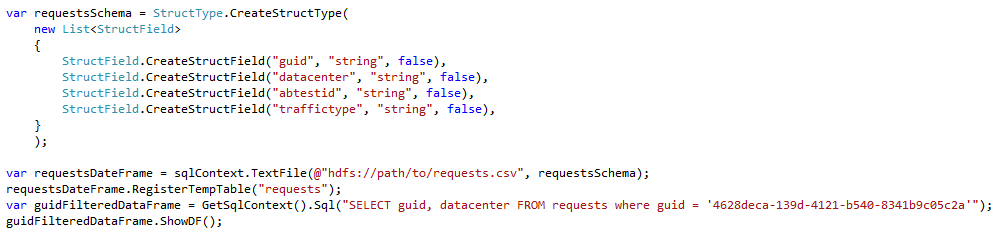
### DataFrame DSL Sample



### DataFrame TempTable Sample



### Schema Specification Sample



# Performance Considerations

* Map & Filter RDD operations implemented in C# require serialization & deserialization across JVM & CLR and it will have an impact on the performance. However, C# operations are pipelined within a stage to minimize unnecessary Ser/De
* DataFrame operations that do not use C# UDFs will take advantage of execution plan optimization & code generation perf improvements built into Spark and there will not be any perf impact relative to Scala apps

# Implementation

SparkCLR is built on top of Spark’s Scala API. Data and processing stays in JVM when no C# user-implemented code is involved (like in Map or Filter methods in RDD API or UDFs in DataFrame).

## Driver-side Adapter

The C# driver program uses Netty-based server for invoking calls in JVM from C# similar to how a driver program in SparkR uses RBackend. The adapter code implemented in SparkCLR binds C# API to the existing Spark API in Java/Scala.

## RDD

RDD API is SparkCLR will allow standard operations like creating RDDs, Map & Filter support in C# and other operators like Distinct, Collect etc. SparkCLR pipelines transformations by composing the functions similar to how pipelining is done in PySpark. With that each SparkCLR stage corresponds to a CSharpRDD instance. Figure 1 below shows the interaction between the components in Java/Scala and C# and how the user implemented driver code in C# is invoked when submitting a C#-driver based Spark job.

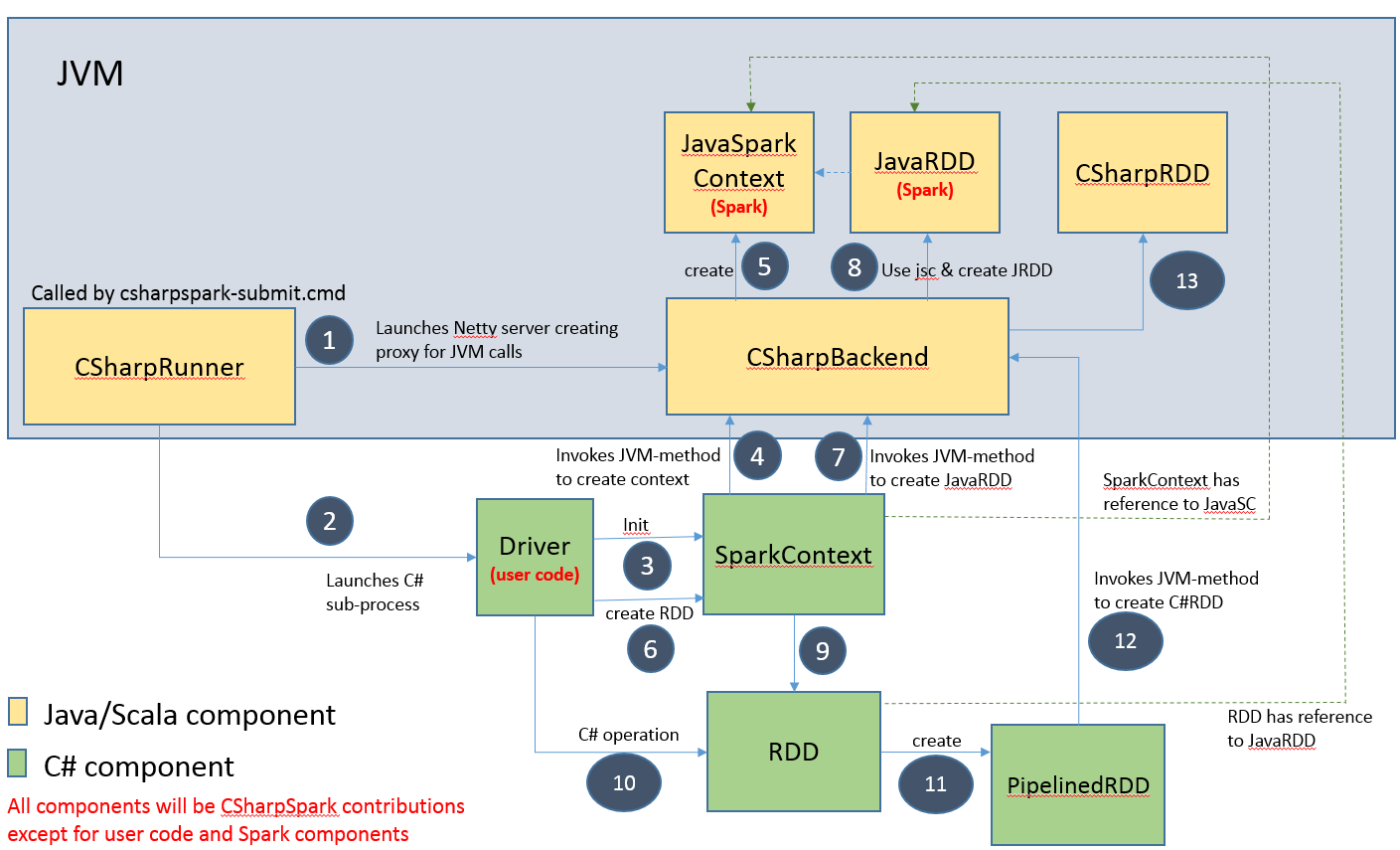


Figure 1 – Interaction between JVM and CLR components in SparkCLR driver

## Executor - RDD

This section covers worker-side details for RDD. PySpark has already implemented PythonRDD that launches sub-processes and communicate with them using pipes, sending the user's code and the data to be processed. PythonRDD implementation is not limited to launching Python sub-processes. Hence we have extended that implementation as CSharpRDD and used it in SparkCLR to process the data using the user code in C#. Figure 2 below shows the interaction between the components in Java/Scala and C# and how the user implemented code in C# is invoked in the Spark workers.

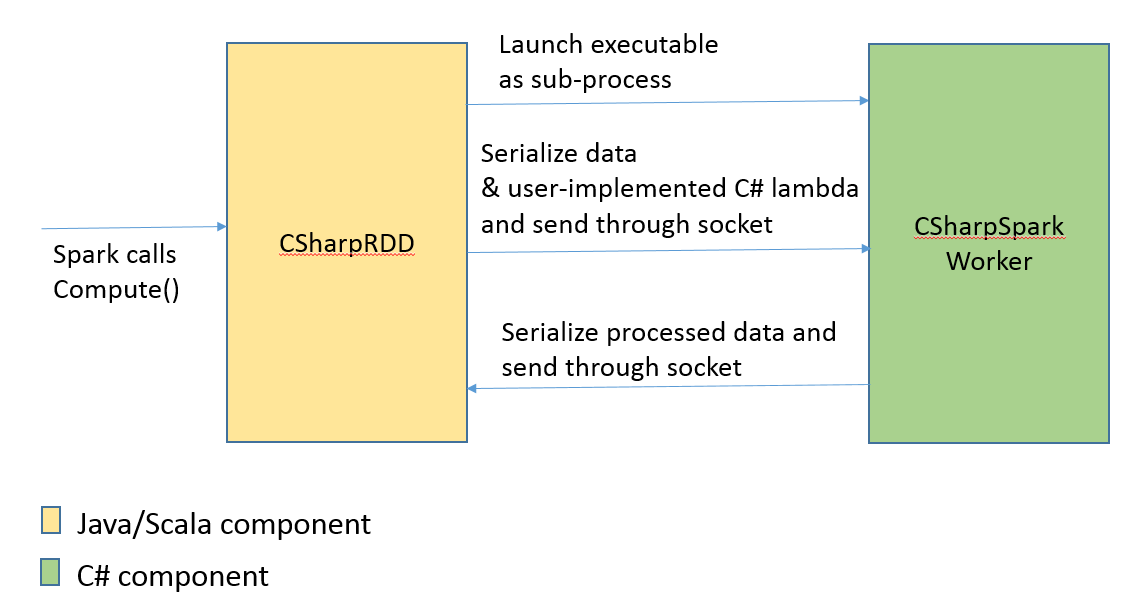
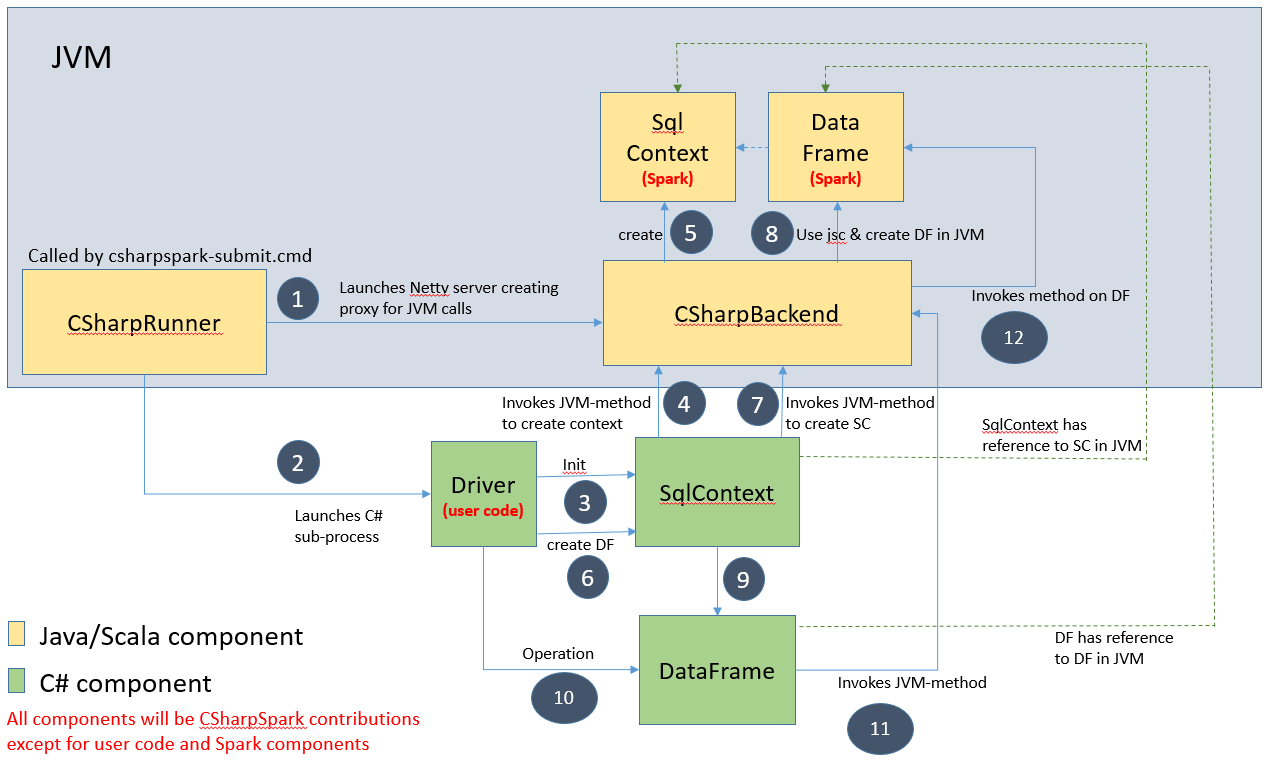


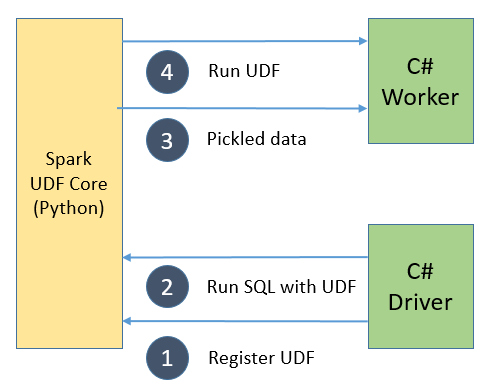
Figure 2 – Interaction between JVM and CLR components in SparkCLR worker

## DataFrame

DataFrame API is SparkCLR will allow standard operations like creating data frames, registering table and running SQL queries, DSL for operations etc. Since the data and processing will stay within JVM except in the case of UDF/UDAF in C# all the optimizations available for DataFrame will be applicable to SparkCLR DataFrame. At the driver-side interaction between CLR and JVM for DataFrame is similar to that of RDD.

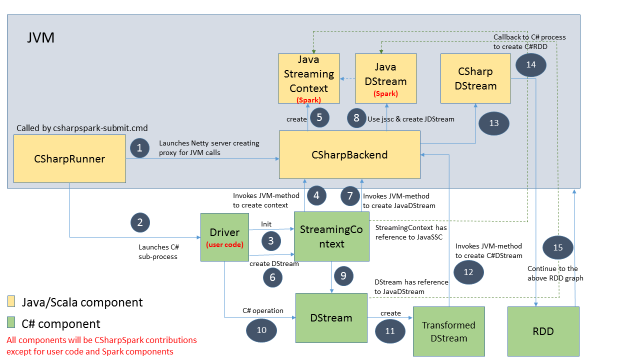


## Executor – DataFrame



# Streaming

Spark streaming is built on top of Spark core. Driver is responsible for dividing input DStream into batches (sequence of RDDs) and feeds into underlying Spark’s batch system. DStream API written in developer's languages other than Scala or Java need to be called from user process. One approach is to let Driver's Java side code call back into C# process through IPC. We borrowed SparkR's design for the callback.



# Other Topics

## Reuse of code

It is possible to refactor existing SparkR and PySpark implementations and share them with SparkCLR. For example, RBackend can be use used for SparkCLR if SerDe can be plugged in specific to C#. Based on the feedback from the project mentors in Databricks (Reynold Xin and Josh Rosen) we have decided to keep SparkCLR a standalone entity that just takes Spark core and other libraries as dependencies and does not depend on any modifications to the existing Spark codebase (note that such modifications are welcome and may take time to get integrated). As a result, we have components that are identical to that of SparkR or PySpark (for example, CSharpBackend). We have tried to ensure that code is reused as much as possible. For example, CSharpRDD just extends from PythonRDD without overriding any behavior for now and as our requirements/design change we have the flexibility to override certain parts while leveraging rest of parts from PythonRDD.

## Serialization in C#

In SparkCLR, user provided operations in Map or Filter methods in RDD API are implemented in C# and they need to be serialized to be shipped over across the Spark cluster to get executed on the worker side. So any types that are used in these implementation need to marked as serializable. During compilation the user provided implementation of anonymous method may result in compiler generated types which are not serializable by default. In those cases, user-implemented code need to explicitly create classes and mark them as serializable using [Serializable] attribute (see Pi - RDD sample below).

## Cluster Manager

Currently SparkCLR support local and standalone modes. YARN integration is being looked into. Mesos integration will probably follow that.

## External dependencies

SparkCLR has components implemented in Scala and C#. Scala components are simply extensions to functionality available in Spark and currently do not explicitly depend on any third party components other than the implicit dependency on what Spark codebase itself depends on. C# components currently have just one dependency (log4net) that has Apache license. C# UDFs in DataFrame reuse the implementation in PySpark and hence there is a dependency on Pyrolite library.

## License

SparkCLR code is released under MIT license.

## Background & Contributions

SparkCLR is currently being developed by a handful of developers in Microsoft Shared Data team to meet the demand for Apache Spark for data processing in Microsoft while leveraging existing investments in C#. We hope the community will see the value in SparkCLR and make contributions to the project. The contributions from other organizations will help expand the surface area in Spark covered by SparkCLR and cater to a wide variety of scenarios not limited to the ones the current SparkCLR development team is most familiar with.

## Open Questions & TBD

* Can explicit initialization of SparkCLREnvironment be avoided in driver code?
  + It exists right now to ensure Dispose method is called explicitly in driver before the app exists. If calling Dispose is optional on the socket used by CSharpBackend, then Init can be made implicit. Note – implicit initialization can be hard to debug in the case of issues with the creation of the environment/CSharpBackend.

# Links

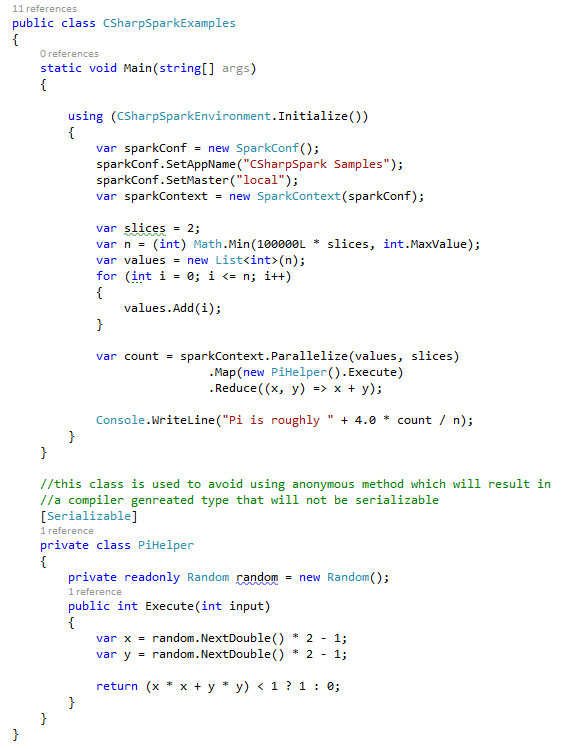
Code location: <https://github.com/Microsoft/SparkCLR>

For more details on the SparkR and PySpark implementations referenced in the doc, refer to the following links:

* <https://github.com/apache/spark/tree/master/R>
* <https://github.com/apache/spark/tree/master/python>

# Samples

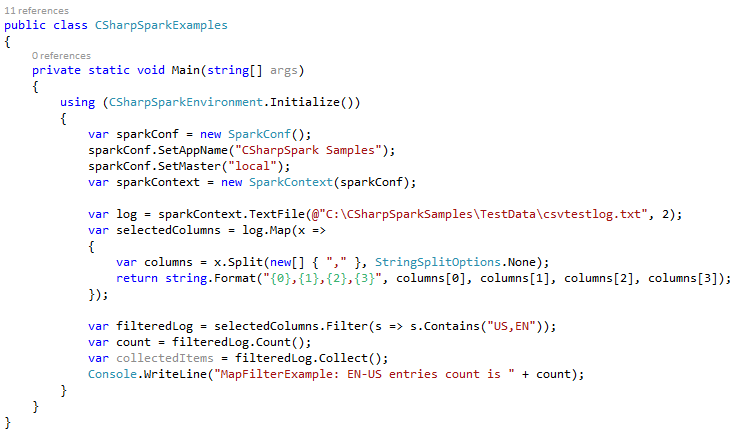
## Pi - RDD



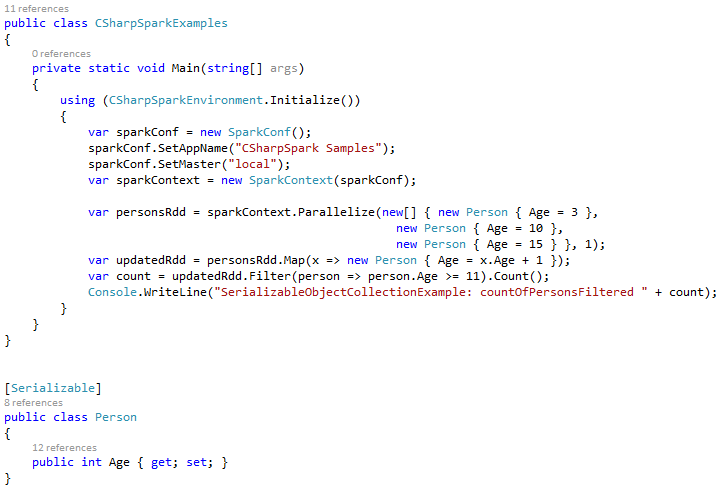
## Word count - RDD



## Log processing - RDD



## Serializable object processing - RDD



## Register Temp Table - DataFrame



## Project & Filter - DataFrame



## Join – DataFrame



## Aggregate – DataFrame

