THE UNIVERSITY OF SYDNEY

School of Information Technologies
A/Prof Uwe Roehm

COMP5349: Cloud Computing **1.Sem./2017**

Week 10: Iterative Processing with Spark/Flink

18.05.2017

Introduction to Iterative Processing

Iterative algorithms occur in many domains of data analysis, such as machine learning or graph analysis. Such algorithms are crucial in order to realize the promise of Big Data to extract meaningful information out of your data. With increasing interest to run these kinds of algorithms on very large data sets, there is a need to execute iterations in a massively parallel fashion.

Source: `https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/batch/iterations.html`

As an introduction to iterative processing in Apache Spark and Flink, we'll be looking at the example of Pi estimation using the Monte Carlo method. A great example and illustration of the method can be found here: `https://academo.org/demos/estimating-pi-monte-carlo/`

Let's look at implementing this Monte Carlo Pi estimation using iteration in Apache Spark and Apache Flink. Note that these implementations will not be very highly efficient; think of them more as a simple introduction to iteration.

Iterative Processing in Apache Spark

Iteration in Spark is considerably basic. It involves using traditional looping mechanisms in the driver, and performing the iterative operations within that loop, for example:

```
for(int i = 0; i < numIters; i++) {
    data = data.map(...);
}
```

This method is very convenient; the driver can easily collect values from previous computations and use those values to iterate in whichever way may be necessary. This might involve broadcasting a variable to the iteration steps for them to use, or performing a variable number of iterations based on previous computations. If the operations within the iteration don't involve shuffling then Spark can perform the calculations in-memory, very quickly.

To achieve similar results in Hadoop MapReduce would involve the driver class reading in the output from a completed job, and then launching new jobs accordingly. Such would incur a far greater overhead.

Spark (Java)

```
package comp5349;

import java.util.Arrays;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.SparkConf;

public class PiIterativeProcessing {
    public static void main(String[] args) {
        final SparkConf conf      = new SparkConf()
                                        .setAppName("Pi Iterative Processing");
        final JavaSparkContext sc = new JavaSparkContext(conf);
        final int numIters        = 100;

        JavaRDD<Integer> count = sc.parallelize(Arrays.asList(0));

        for(int i = 0; i < numIters; i++) {
            count = count.map(c -> {
                double x = Math.random();
                double y = Math.random();

                return c + ((x * x + y * y < 1) ? 1 : 0);
            });
        }

        System.out.println(count.collect().get(0) / (double) numIters * 4);
    }
}
```

Spark (Python)

```
from pyspark import SparkContext, SparkConf
from random import random

conf = SparkConf().setAppName("Pi Iterative Processing")
sc = SparkContext(conf=conf)

num_iters = 100

count = sc.parallelize([0])

def throwDart(count):
    x = random()
    y = random()

    return count + (1 if x * x + y * y < 1 else 0)

for i in range(num_iters):
    count = count.map(throwDart)

print(count.collect()[0] / float(num_iters) * 4)
```

Limitations

Notice that the number of iterations in the code above is set to 100. If you look below in the Flink example, it's set to 100,000; substantially higher than in Spark. This is because the two examples are using different mechanisms to achieve iteration - Spark however is limited only to this one.

Spark's limit is low because this form of iteration actually involves iteratively adding new computation steps to the job's execution:

```
for(int i = 0; i < numIters; i++) {
    data = data.map(...);
}
```

The above code is practically translated into:

```
data.map(...).map(...).map(...).map(...).map(...)...
```

Spark evaluates a chain like this recursively, and recursion always has its limits. In this case the limit is quite low, leading to an exception being raised event with 1000 iterations. It wouldn't be wise to simply increase the limit as it's likely that each recursion will use some memory, and having a large limit would lead to memory exhaustion.

Whilst this is indeed a limitation for use-cases like this Pi estimation example, in reality it's not too bad. Most machine learning algorithms would use a lower number of iterations, where each iteration would perform a far more expensive task and take advantage of distribution and parallelism. That's why this Pi example isn't particularly great: it doesn't take advantage of distribution or parallelism. Instead we've used it just to illustrate the concept of iteration in Apache Spark and Apache Flink.

Iterative Processing in Apache Flink

Iteration can be achieved in Flink in the same way it has been in Spark above; by chaining operations within a traditional loop structure. This will have it work in a similar manner, with similar limitations.

However, Apache Flink has additionally introduced a specialised data structure to handle iteration; namely the `IterativeDataSet`. It takes a little more effort to reason about the structure, however the use of it allows Flink to recognise and optimise the execution of the iteration, for great benefit. For instance where we were unable to set the number of iterations even to 1000 in Spark's example, Flink was comfortable even with 100,000.

You should read more about how Flink handles iteration here: `https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/batch/iterations.html`.

Flink (Java)

```
package comp5349;

import org.apache.flink.api.java.*;
import org.apache.flink.api.java.operators.*;

public class PiIterativeProcessing {
    public static void main(String[] args) throws Exception {
        final ExecutionEnvironment env = ExecutionEnvironment
                                            .getExecutionEnvironment();
        final int numIters             = 100000;

        IterativeDataSet<Integer> initial = env.fromElements(0).iterate(numIters);

        DataSet<Integer> iteration = initial.map(i -> {
            double x = Math.random();
            double y = Math.random();

            return i + ((x * x + y * y < 1) ? 1 : 0);
        });

        DataSet<Integer> count = initial.closeWith(iteration);

        count.map(c -> c / (double) numIters * 4).print();
    }
}
```

Iteration in Assignment 2

Assignment 2 has been released now, and a key concept within it is iteration. The k-means clustering algorithm that tasks 2 and 3 require you to work with is an iterative algorithm. To implement it yourself you will need a solid understanding of iterative algorithms in Apache Spark and Flink and how they can be used.

Use the remaining time in this tutorial in comprehending the usage of iteration, and starting to think about how iteration can be combined with the other features of these systems (like map, reduce, broadcast variables, etc.) to solve the k-means clustering problem.

If you're having trouble understanding the k-means clustering algorithm, or have any points for discussion, do raise them while you have the chance.