Assignment on Spark

One of the most common uses of Spark is analyzing and processing log files. In this assignment, we will put Spark to good use for an OSS project that retrieves and downloads data from GitHub, called GHTorrent (http://ghtorrent.org). GHTorrent works by following the Github event timeline (https://api.github.com/events) and then retrieving all items linked from each event recursively and exhaustively. To make monitoring and debugging easier, the GHTorrent maintainers use extensive runtime logging for the downloader scripts.

Here is an extract of what the GHTorrent log looks like:

```
DEBUG, 2017-03-23T10:02:27+00:00, ghtorrent-40 -- ghtorrent.rb: Repo EFForg/https-everywhere exists
DEBUG, 2017-03-24T12:06:23+00:00, ghtorrent-49 -- ghtorrent.rb: Repo Shikanime/print exists
INFO, 2017-03-23T13:00:55+00:00, ghtorrent-42 -- api_client.rb: Successful request. URL: https://api.github.com/repos/CanonicalLtd/maas-docs/issues/365/events?per_page=100, Remaining: 4943, Total: 88 ms
WARN, 2017-03-23T20:04:28+00:00, ghtorrent-13 -- api_client.rb: Failed request. URL: https://api.github.com/repos/greatfakeman/Tabchi/commits?sha=Tabchi&per_page=100, Status code: 404, Status: Not Found, Access: ac6168f8776, IP: 0.0.0.0, Remaining: 3031
DEBUG, 2017-03-23T09:06:09+00:00, ghtorrent-2 -- ghtorrent.rb: Transaction committed (11 ms)
```

Each log line comprises of a standard part (up to `.rb:`) and an operation-specific part. The standard part fields are like so:

1. Logging level, one of `DEBUG`, `INFO`, `WARN`, `ERROR` (separated by `,`)
2. A timestamp (separated by `,`)
3. The downloader id, denoting the downloader instance (separated by `--`)
4. The retrieval stage, denoted by the Ruby class name, one of:
    - `event_processing`
    - `ght_data_retrieval`
    - `api_client`
    - `retriever`
    - `ghtorrent`

Grade: This assignment consists of 130 points. You need to collect 100 to get a 10!

Loading and parsing the file

For the remaining of the assignement, you need to use this file (https://drive.google.com/file/d/0B9Rx0uhucsroYWJxdEpPd2JYcjg/view?usp=sharing) (~300MB compressed).

T (5 points): Download the log file and write a function to load it in an RDD. If you are doing this in Scala, make sure you use a case class to map the file fields.

In [2]:
```
import java.text.SimpleDateFormat
import java.util.Date

case class LogLine(debug_level: String, timestamp: Date, download_id: Integer,
                   retrieval_stage: String, rest: String);

val dateFormat = "yyyy-MM-dd:HH:mm:ss"
val regex = """([^\s]+), ([^\s]+)\+00:00, ghtorrent-([^\s]+) -- ([^\s]+).rb:
 (.*$)""".r

val rdd = sc.
    textFile("ghtorrent-logs.txt").
    flatMap ( x => x match {
      case regex(debug_level, dateTime, downloadId, retrievalStage, rest) =>
        val df = new SimpleDateFormat(dateFormat)
        new Some(LogLine(debug_level, df.parse(dateTime.replace("T", ":")), do
wnloadId.toInt, retrievalStage, rest))
      case _ => None;
      })
```

T (5 points): How many lines does the RDD contain?

In [4]:
```
rdd.count;
```

Out[4]:
```
9669634
```

Basic counting and filtering

T (5 points): Count the number of WARNing messages

In [3]:
```
rdd.filter(x => x.debug_level == "WARN")
    .count
```

Out[3]:
```
132158
```

T (10 points): How many repositories where processed in total? Use the `api_client`

In [7]:

```
val repos = rdd.filter(_.retrieval_stage == "api_client").
    map(_.rest.split("/").slice(4,6).mkString("/").takeWhile(_ != '?'))
```

In [15]:

```
repos.distinct.count
```

```
[Stage 15:====================================================>      (8 + 1) / 
9]
```

Out[15]:

```
105296
```

Analytics

T (5 points): Which client did most HTTP requests?

In [38]:

```
rdd.filter(_.retrieval_stage == "api_client").
    keyBy(_.download_id).
    mapValues(l => 1).
    reduceByKey((a,b) => a + b).
    sortBy(x => x._2, false).
    take(3)
```

Out[38]:

```
[(13,135978), (21,100906), (20,31401)]
```

T (5 points): Which client did most FAILED HTTP requests?

In [39]:

```
rdd.filter(_.retrieval_stage == "api_client").
    filter(_.rest.startsWith("Failed")).
    keyBy(_.download_id).
    mapValues(l => 1).
    reduceByKey((a,b) => a + b).
    sortBy(x => x._2, false).
    take(3)
```

Out[39]:

```
[(13,79623), (21,1378), (40,1134)]
```

T (5 points): What is the most active hour of day?

In [42]:

```
rdd.keyBy(_.timestamp.getHours).
    mapValues(l => 1).
    reduceByKey((a,b) => a + b).
    sortBy(x => x._2, false).
    take(3)
```

```
[Stage 40:==============================================>            (7 + 2) / 
9]

warning: there was one deprecation warning; re-run with -deprecation for deta
ils
```

Out[42]:

```
[(10,2662487), (9,2411930), (11,2367051)]
```

T (5 points): What is the most active repository?

```
In [41]: repos.
             filter(_.nonEmpty).
             map(x => (x, 1)).
             reduceByKey((a,b) => a + b).
             sortBy(x => x._2, false).
             take(3)
```

```
[Stage 35:==============================================>            (7 + 2) /
9]
```

```
Out[41]: [(greatfakeman/Tabchi,79524), (mithro/chromium-infra,4084), (shuhongwu/hockey
app,2575)]
```

T (5 points): Which access keys are failing most often?

Hint:: extract the `Access: ...` part from failing requests

```
In [43]: rdd.filter(_.rest.startsWith("Failed")).
             filter(_.rest.contains("Access: ")).
             map(_.rest.split("Access: ", 2)(1).split(",", 2)(0)).
             map(x => (x, 1)).
             reduceByKey((a,b) => a + b).
             sortBy(x => x._2, false).
             take(3)
```

```
[Stage 48:====================================================>      (8 + 1) /
9]
```

```
Out[43]: [(ac6168f8776,79623), (46f11b5791b,1340), (9115020fb01,1134)]
```

Indexing

Typical operations on RDDs require grouping on a specific part of each record and then calculating specific counts given the groups. While this operation can be achieved with the `group_by` family of funcions, it is often useful to create a structure called an *inverted index*. An inverted index creates an `1..n` mapping from the record part to all occurencies of the record in the dataset. For example, if the dataset looks like the following:

```
col1,col2,col3
A,1,foo
B,1,bar
C,2,foo
D,3,baz
E,1,foobar
```

an inverted index on `col2` would look like

```
1 -> [(A,1,foo), (B,1,bar), (E,1,foobar)]
2 -> [(C,2,foo)]
3 -> [(D,3,baz)]
```

Inverted indexes enable us to quickly access precalculated partitions of the dataset. To see their effect on large datasets, lets compute an inverted index on the `downloader id` part.

T (10 points): Create a function that given an `RDD[Seq[T]]` and an index position (denotes which field to index on), it computes an inverted index on the RDD.

In [4]:

```
// Create inverted index for rdd on column idx_id
def inverted_index(rdd : org.apache.spark.rdd.RDD[LogLine], idx_id : Int):
    org.apache.spark.rdd.RDD[(Any, Iterable[LogLine])] = {
    return rdd.groupBy((x : LogLine) => x.productElement(idx_id));
}
```

T (5 points): Compute the number of different *repositories* accessed by the client `ghtorrent-22` (without using the inverted index).

In [3]:

```
// get unique repos count for user 22 without inverted index
val start_time = System.currentTimeMillis();

val client22 = rdd.filter(_.download_id == 22).
                   map(_.rest.split("/").slice(4,6).mkString("/").takeWhile(_
!= '?'))

println(client22.distinct.count)
println("Took " + (System.currentTimeMillis() - start_time) + "ms."); //get ti
me in seconds
```

```
3973
Took 32945ms.
```

T (5 points): Compute the number of different *repositories* accessed by the client `ghtorrent-22` (using the inverted index you calculated above). Remember that Spark computations are lazy, so you need to run the inverted index generation before you actually use the index.

In [5]:

```
// create inverted index on ID
val invertedIndex = inverted_index(rdd, 2);

// dummy lookup here to create the index
val look21 = invertedIndex.lookup(21);
```

In [8]:

```
// lookup user 22 and check unique repos
val start_time = System.currentTimeMillis();
val lookedUp22 = invertedIndex.lookup(22)

val it = Iterator(lookedUp22).next();
var uniqueRepos = List[String]();

for (x <- it){
    for (y <- x) {
        if (!uniqueRepos.contains(y.rest)) {
            uniqueRepos = uniqueRepos :+ y.rest;
        }
    }
}

println(uniqueRepos.size);
println("Took " + (System.currentTimeMillis() - start_time) + "ms."); //get ti
me in seconds
```

T (5 points): You should have noticed some difference in performance. Why is the indexed version faster?

79.28 seconds vs 1.31 seconds Because after creating an inverted index on the ID key retrieving the data using an ID goes a lot faster, it only has to find 1 row instead of multiple rows containing the user.

T (5 points): Read up about `groupByKey`. Explain in 3 lines why it the worst function in the Spark API and what you can use instead.

In [ ]:

Joining

We now need to monitor the behaviour of interesting repositories. Use [this link (https://drive.google.com/open?id=0B9Rx0uhucsroRHNVTFpzMV9OUGs)](https://drive.google.com/open?id=0B9Rx0uhucsroRHNVTFpzMV9OUGs) to download a list of repos into which we are interested to. This list was generated on Oct 10, 2017, more than 7 months after the log file was created. The format of the file is CSV, and the meaning of the fields can be found on the GHTorrent project web site [documentation (http://ghtorrent.org/relational.html)](http://ghtorrent.org/relational.html).

T (5 points): Read in the CSV file to an *RDD* (let's call it *interesting*). How many records are there?

In [75]:

```scala
import java.text.SimpleDateFormat
import java.util.Date

case class Repo(id: Integer, url: String, owner_id: Integer,
                   name: String, language: String, created_at: Date, forked_fr
om: String, deleted: Integer, updated_at: Date)

val dateFormat = "yyyy-MM-dd HH:mm:ss"
val regex = """([^,]+),([^,]+),([^,]+),([^,]+),([^,]+),([^,]+),([^,]+),([^,]
+),([^,]+)""".r

val interesting = sc.
    textFile("hdfs://bdp1:8020/important-repos.csv").
    mapPartitionsWithIndex((idx, iter) => if (idx == 0) iter.drop(1) else iter
). //remove header from CSV file
    flatMap ( x => x match {
      case regex(id, url, owner_id, name, language, created_at, forked_from, d
eleted, updated_at) => {
        val df = new SimpleDateFormat(dateFormat)
        new Some(Repo(id.toInt, url, owner_id.toInt, name, language, df.parse(
created_at), forked_from, deleted.toInt, df.parse(updated_at)))
      }
      case _ => print(x); None;
      }).cache()
```

```
defined class Repo
dateFormat = yyyy-MM-dd HH:mm:ss
regex = ([^,]+),([^,]+),([^,]+),([^,]+),([^,]+),([^,]+),([^,]+),([^,]+),([^,]
+)
interesting = MapPartitionsRDD[147] at flatMap at <console>:32
```

Out[75]:

```
MapPartitionsRDD[147] at flatMap at <console>:32
```

```
In [48]:  interesting.count();

Out[48]:  1435
```

T (10 points): How many records in the log file refer to entries in the *interesting* file?

Hint: Yes, you need to join :) First, you need to key both RDDs by the repository name to do such a join.

```
In [135]:  val interestingRepo = interesting.keyBy(_.name);
           val logLineRepo = rdd.keyBy(_.rest).
                                 map(x => x.copy(_1 = x._1.split("/").slice(4,6).mkString
           ("/").takeWhile(_ != '?').split("/", 2).last)).
                                 filter(_._1.nonEmpty); //delete all empty repos

           val joinedRepo = interestingRepo.join(logLineRepo);

           interestingRepo = MapPartitionsRDD[333] at keyBy at <console>:76
           logLineRepo = MapPartitionsRDD[336] at filter at <console>:79
           joinedRepo = MapPartitionsRDD[339] at join at <console>:81

Out[135]:  MapPartitionsRDD[339] at join at <console>:81
```

```
In [137]:  joinedRepo.count;

           [Stage 152:===================================================>      (8 + 1) /
           9]

Out[137]:  87930
```

T (5 points): Which of the *interesting* repositories has the most failed API calls?

```
In [139]:  joinedRepo.filter(x => x._2._2.rest.startsWith("Failed")).
                      map(x => (x._1, 1)).
                      reduceByKey((a,b) => a + b).
                      sortBy(x => x._2, false).
                      take(3)

           [Stage 158:===================================================>      (8 + 1) /
           9]

Out[139]:  [(hello-world,740), (test,309), (demo,166)]
```

Dataframes

T (10 points) Read in the *interesting* repos file using Spark's CSV parser. Convert the log RDD to a Dataframe.

In [56]:

```
import org.apache.spark.sql.types.{StructType, StructField, StringType, IntegerType, DateType};
import org.apache.spark.sql.Row;


val interesting_df = spark.read.
                        format("csv").
                        option("header", "true").
                        option("inferSchema", "true").
                        load("hdfs://bdp1:8020/important-repos.csv");

//SQL doesn't support Date classes, so I can't .toDF() the rdd.
val logSchema = StructType(Array(
    StructField("debug_level", StringType, nullable=true),
    StructField("timestamp", DateType, nullable=true),
    StructField("download_id", IntegerType, nullable=true),
    StructField("retrieval_stage", StringType, nullable=true),
    StructField("repo", StringType, nullable=true), //add repo name to the dataframe, to make usage easier
    StructField("rest", StringType, nullable=true)
));

val log_rdd = rdd.map(x => Row(x.debug_level, new java.sql.Date(x.timestamp.getTime()), x.download_id, x.retrieval_stage, x.rest.split("/").slice(4,6).mkString("/").takeWhile(_ != '?').split("/", 2).last, x.rest));
val log_df = spark.createDataFrame(log_rdd, logSchema);

log_df.printSchema;
interesting_df.printSchema;
```

```
root
 |-- debug_level: string (nullable = true)
 |-- timestamp: date (nullable = true)
 |-- download_id: integer (nullable = true)
 |-- retrieval_stage: string (nullable = true)
 |-- repo: string (nullable = true)
 |-- rest: string (nullable = true)

root
 |-- id: integer (nullable = true)
 |-- url: string (nullable = true)
 |-- owner_id: integer (nullable = true)
 |-- name: string (nullable = true)
 |-- language: string (nullable = true)
 |-- created_at: timestamp (nullable = true)
 |-- forked_from: string (nullable = true)
 |-- deleted: integer (nullable = true)
 |-- updated_at: string (nullable = true)


interesting_df = [id: int, url: string ... 7 more fields]
logSchema = StructType(StructField(debug_level,StringType,true), StructField
(timestamp,DateType,true), StructField(download_id,IntegerType,true), StructF
ield(retrieval_stage,StringType,true), StructField(repo,StringType,true), Str
uctField(rest,StringType,true))
log_rdd = MapPartitionsRDD[120] at map at <console>:72
log_df = [debug_level: string, timestamp: date ... 4 more fields]

lastException: Throwable = null
```

Out[56]:
```
[debug_level: string, timestamp: date ... 4 more fields]
```

T (15 points) Repeat all 3 queries in the "Joining" section above using either SQL or the Dataframe API. Measure the time it takes to execute them.

In [57]:
```scala
interesting_df.count();
```

Out[57]:
```
1435
```

In [59]:
```scala
val joined_df = interesting_df.join(log_df, interesting_df("name") === log_df(
"repo"));

joined_df.count();
```

```
[Stage 34:===================>                                        (3 + 6) /
9]

joined_df = [id: int, url: string ... 13 more fields]
```

Out[59]:
```
87930
```

In [87]:

```
import org.apache.spark.sql.functions._

joined_df.filter(joined_df("rest").startsWith("Failed")).
          groupBy(joined_df("name")).
          count().
          orderBy(desc("count")).
          take(3);
```

```
[Stage 83:================================>                         (5 + 4) /
9]
```

Out[87]:

```
hello-world   740
       test   309
       demo   166
```

T (5 points) Select one of the queries and compare the execution plans between the RDD version and your version. (you can see them by going to localhost:4040 in your VM). What differences do you see?

```
scala
```