Interactive Audience Analytics With Apache Spark and HyperLogLog

October 13, 2015 by Eugene Zhulenev in **COMPANY BLOG**

This is a guest blog from Eugene Zhulenev on his experiences with Engineering Machine Learning and Audience Modeling at Collective.

At Collective, we are working not only on cool things like Machine Learning and Predictive Modeling but also on reporting that can be tedious and boring. However at our scale even simple reporting application can become a challenging engineering problem. This post is based on a talk that I gave at NY-Scala Meetup. Slides are available here.

> Example application is available on github: https://github.com/collectivemedia/spark-hyperloglog

Impression Log

We are building reporting application that is based on an impression log. It's not exactly the way how we get data from out partners, it's pre-aggregated by Ad, Site, Cookie. And even in this pre-aggregated format it takes hundreds of gigabytes per day on HDFS.

```
Ad            | Site          | Cookie          | Impressions | Clicks | Segments
------------- |-------------- | --------------- | ----------- | ------ | --------------
bmw_X5        | forbes.com    | 13e835610ff0d95 | 10          | 1      | [a.m, b.rk, c.
mercedes_2015 | forbes.com    | 13e8360c8e1233d | 5           | 0      | [a.f, b.rk, c.
nokia         | gizmodo.com   | 13e3c97d526839c | 8           | 0      | [a.m, b.tk, c.
apple_music   | reddit.com    | 1357a253f00c0ac | 3           | 1      | [a.m, b.rk, d.
nokia         | cnn.com       | 13b23555294aced | 2           | 1      | [a.f, b.tk, c.
apple_music   | facebook.com  | 13e8333d16d723d | 9           | 1      | [a.m, d.sn, g.
```

Each cookie id has assigned segments which are just 4-6 letters code, that represents some information about the cookie, that we get from 3rd party data providers such as Blukai.

```
- a.m  : Male
- a.f  : Female
- b.tk : $75k-$100k annual income
- b.rk : $100k-$150k annual income
- c.hs : High School
- c.rh : College
```

```
- d.sn : Single
- d.mr : Married
```

For example if a cookie has been assigned `a.m` segment, it means that we think (actually the data provider thinks) that this cookie belongs to a male. The same thing for annual income level and other demographics information.

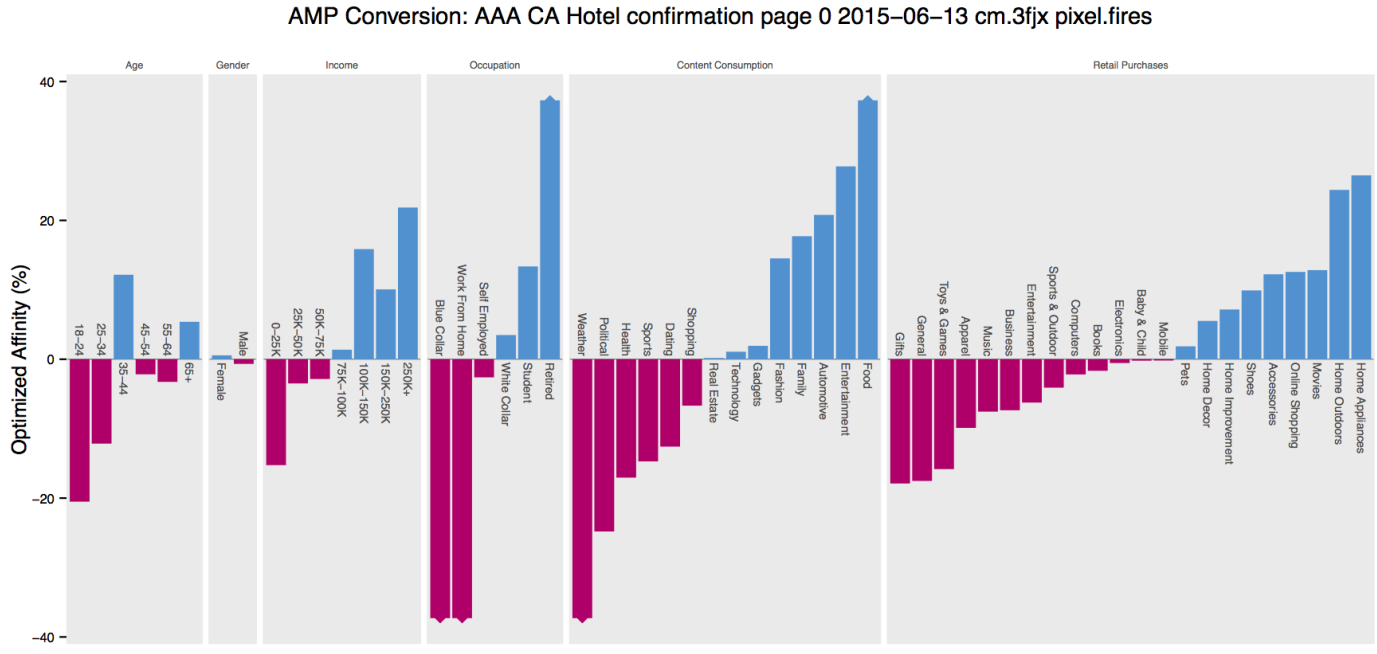
We don't have precise information, to whom exactly a particular cookie belongs nor what is their real annual income level. These segments are essentially probabilistic, nevertheless we can get very interesting insights from this data.

What we can do with this data

Using this impression log we can answer some interesting questions

- We can calculate a given group's prevalence in a campaign's audience, eg. what role do **males** play in the optimized audience for a **Goodyear Tires** campaign?
- What is **male/female** ratio for people who have seen **bmw_X5** ad on **forbes.com**
- Income distribution for people who have seen an Apple Music ad
- Nokia clicks distribution across different education levels

Using these basic questions we can create an "Audience Profile" that describes what type of audience is prevailing in an optimized campaign or partner website.



Blue bars mean that this particular segment tends to view ad/visit website more than on average, and red bar mean less. For example for **Goodyear Tires** we expect to see more **male** audience than **female**.

Solving problems with SQL

SQL looks like an easy choice for this problem, however as I already mentioned we have hundreds of gigabytes of data every day, and we need to get numbers based on 1-year history in seconds. Hive/Impala simply cannot solve this problem.

```
select count(distinct cookie_id) from impressions

    where site = ‘forbes.com’

    and ad = ‘bmw_X5’

    and segment contains ‘a.m’
```

Unfortunately, we have almost infinite combinations of filters that users can define, so it’s not feasible to pre-generate all possible reports. Users can use any arbitrary ad, site, campaign, order filter combinations, and may want to know audience intersection with any segment.

Audience cardinality approximation with HyperLogLog

We came up with a different solution; instead of providing precise results for every query, we are providing approximated numbers with very high precision. Usually, the error rate is around 2% which for this particular application is really good. We don’t need to know an exact number of male/female cookies in the audience. To be able to say what audience is prevailing, approximated numbers are more than enough.

We use HyperLogLog, which is an algorithm for the count-distinct problem, approximating the number of distinct elements (cardinality). It uses finite space and has configurable precision. It able to estimate cardinalities of >10^9 with a typical accuracy of 2%, using 1.5kB of memory.

```
trait HyperLogLog {

    def add(cookieId: String): Unit

    //   |A|

    def cardinality(): Long

    //   |A ∪ B|

    def merge(other: HyperLogLog): HyperLogLog

    //   |A ∩ B| = |A| + |B| - |A ∪ B|,

    def intersect(other: HyperLogLog): Long

}
```

Here is a rough API that is provided by `HyperLogLog`. You can add a new cookieId to it, get cardinality estimation of unique cookies that were already added to it, merge it with

another `HyperLogLog`, and finally get an intersection. It's important to notice that after the `intersect` operation, the `HyperLogLog` object is lost, and you only have approximated intersection cardinality. Therefore, usually `HyperLogLog` intersection is the last step in the computation.

I suggest you to watch the awesome talk by Avi Bryant where he discusses not only HyperLogLog but lot's of other approximation data structures that can be useful for big-data analytics: http://www.infoq.com/presentations/abstract-algebra-analytics.

From cookies to HyperLogLog

We split out original impression log into two tables.

For the ad impressions table, we remove segment information and aggregate cookies, impressions and clicks by Ad and Site. `HyperLogLog` can be used in an aggregation function similar to how use the `sum` operation. Zero is an empty `HyperLogLog` while the plus operation is `merge` (btw it's exactly properties required by `Monoid`)

```
Ad            | Site          | Cookies HLL        | Impressions | Clicks
------------- | ------------- | ------------------ | ----------- | ------
bmw_X5        | forbes.com    | HyperLogLog@23sdg4 | 5468        | 35
bmw_X5        | cnn.com       | HyperLogLog@84jdg4 | 8943        | 29
```

For the segments table, we remove ad and site information, and aggregate data by segment.

```
Segment       | Cookies HLL        | Impressions | Clicks
------------- | ------------------ | ----------- | ------
Male          | HyperLogLog@85sdg4 | 235468      | 335
$100k-$150k   | HyperLogLog@35jdg4 | 569473      | 194
```

Percent of college and high school education in the BMW campaign

If you can imagine that we can load these tables into `Seq`, then audience intersection becomes a really straightforward task, that can be solved by a couple lines of functional scala operations.

```
case class Audience(ad: String, site: String, hll: HyperLogLog, imp: Long, clk: Long)
case class Segment(name: String, hll: HyperLogLog, imp: Long, clk: Long)

val adImpressions: Seq[Audience] = …

val segmentImpressions: Seq[Segment] = …

val bmwCookies: HyperLogLog = adImpressions
    .filter(_.ad = "bmw_X5")
    .map(_.hll).reduce(_ merge _)
```

```
val educatedCookies: HyperLogLog = segmentImpressions
    .filter(_.segment in Seq("College", "High School"))
    .map(_.hll).reduce( _ merge _)

val p = (bmwCookies intersect educatedCookies) / bmwCookies.count()
```

Apache Spark DataFrames with HyperLogLog

Obviously we can't load all the data into a scala `Seq` on single machine, because it's huge. Even after removing cookie level data and transforming it into `HyperLogLog` objects, it's around 1-2 gigabytes of data for a single day.

So we have to use some distributed data processing framework to solve this problem, and we chose Spark.

What are Spark DataFrames

- Inspired by R data.frame and Python/Pandas DataFrame
- Distributed collection of rows organized into named columns
- Used to be SchemaRDD in Spark < 1.3.0

High-Level DataFrame Operations

- Selecting required columns
- Filtering
- Joining different data sets
- Aggregation (count, sum, average, etc)

You can start by referring to the Spark DataFrame guide or DataBricks blog post.

Ad impressions and segments in DataFrames

We store all of our data on HDFS using Parquet data format, and that's how it looks after it's loaded into Spark DataFrames.

```
val adImpressions: DataFrame = sqlContext.parquetFile("/aa/audience")
adImpressions.printSchema()

// root
//   | – ad: string (nullable = true)
//   | – site: string (nullable = true)
```

```
//   | – hll: binary (nullable = true)
//   | – impressions: long (nullable = true)
//   | – clicks: long (nullable = true)

val segmentImpressions: DataFrame = sqlContext.parquetFile("/aa/segments")

segmentImpressions.printSchema()
// root
//   | – segment: string (nullable = true)
//   | – hll: binary (nullable = true)
//   | – impressions: long (nullable = true)
//   | – clicks: long (nullable = true)
```

`HyperLogLog` is essentially a huge `Array[Byte]` with some clever hashing and math, so it's straightforward to store it on HDFS in serialized form.

Working with a Spark DataFrame

We wanted to know the answer for the question: "Percent of college and high school education in the BMW campaign".

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

import org.apache.spark.sql.HLLFunctions._
val bmwCookies: HyperLogLog = adImpressions
    .filter(col("ad") === "bmw_X5")
    .select(mergeHll(col("hll")).first() // – sum(clicks)

val educatedCookies: HyperLogLog = hllSegments
    .filter(col("segment") in Seq("College", "High School"))
    .select(mergeHll(col("hll")).first()

val p = (bmwCookies intersect educatedCookies) / bmwCookies.count()
```

It looks pretty familiar, not too far from example based on scala `Seq`. Only one unusual operation that you might notice if you have some experience with Spark is `mergeHLL`. It's not available in Spark by default, it is a custom `PartialAggregate` function that can compute aggregates for serialized `HyperLogLog` objects.

Writing your own Spark aggregation function

To write you own aggregation function you need to define a function that will be applied to each row in RDD partition, in this example it's called `MergeHLLPartition`. Then you need to define the function that will take results from different partitions and merge them together, for `HyperLogLog` it's called `MergeHLLMerge`. And finally you need to tell Spark how you want it to split your computation across `RDD` (DataFrame is backed by `RDD[Row]`)

```
case class MergeHLLPartition(child: Expression)

  extends AggregateExpression with trees.UnaryNode[Expression] { … }
case class MergeHLLMerge(child: Expression)

  extends AggregateExpression with trees.UnaryNode[Expression] { … }


case class MergeHLL(child: Expression)

  extends PartialAggregate with trees.UnaryNode[Expression] {

  override def asPartial: SplitEvaluation = {

    val partial = Alias(MergeHLLPartition(child), "PartialMergeHLL")()

    SplitEvaluation(

      MergeHLLMerge(partial.toAttribute),

      partial :: Nil

    )

  }

}

def mergeHLL(e: Column): Column = MergeHLL(e.expr)
```

After that, writing aggregations becomes a really easy task, and your expressions will look like "native" DataFrame code, which is really nice, and super easy to read and reason about.

Also it works much faster than solving this problem with scala transformations on top of `RDD[Row]`, as Spark catalyst optimizer can execute an optimized plan and reduce the amount of data that needs to be shuffled between spark nodes.

And finally, it's so much easier to manage mutable state. Spark encourage you to use immutable transformations, and it's really cool until you need extreme performance from your code. For example, if you are using something like `reduce` or `aggregateByKey` you don't really know when and where your function instantiated, when it's done with `RDD` partition, nor when the results are transferred to another Spark node for a merge operation. With `AggregateExpression` you have explicit control over mutable state, and it's totally safe to accumulate mutable state during execution for a single partition. At the end when you'll need to send data to another node where you can create immutable copy.

In this particular case, using a mutable `HyperLogLog` merge implementation helped to speed up computation times by almost 10x. For each partition `HyperLogLog` state accumulated in single mutable `Array[Byte]` and at the end when data needs to be transferred somewhere else for merging with another partition, an immutable copy is created.

Some fancy aggregates with DataFrame API

You can write much more complicated aggregation functions, for example, to compute aggregate based on multiple columns. Here is a code sample from our audience analytics project.

```
case class SegmentEstimate(cookieHLL: HyperLogLog, clickHLL: HyperLogLog)
type SegmentName = String

val dailyEstimates: RDD[(SegmentName, Map[LocalDate, SegmentEstimate])] =

    segments.groupBy(segment_name).agg(


      segment_name,

      mergeDailySegmentEstimates(

        mkDailySegmentEstimate(      // — Map[LocalDate, SegmentEstimate]

          dt,

          mkSegmentEstimate(         // — SegmentEstimate(cookieHLL, clickHLL)

            cookie_hll,

            click_hll)

        )

      )
```

```
)
```

This code calculates daily audience aggregated by segment. Using Spark `PartialAggregate` function saves a lot of network traffic and minimizes the distributed shuffle size.

This aggregation is possible because of nice properties of `Monoid`

- `HyperLogLog` is a `Monoid` (has `zero` and `plus` operations)
- `SegmentEstimate` is a `Monoid` (tuple of two monoids)
- `Map[K, SegmentEstimate]` is a `Monoid` (map with value monoid value type is monoid itself)

Problems with custom aggregation functions

- Right now, it is a closed API so you need to place all of your code under the `org.apache.spark.sql` package.
- It is not guaranteed that it will work in next Spark release.
- If you want to try, I suggest you to start with `org.apache.spark.sql.catalyst.expressions.Sum`as example.

Spark as an in-memory SQL database

We use Spark as an in-memory database that serves SQL (composed with DataFrame API) queries.

People tend to think about Spark with a very batch oriented mindset. Start a Spark cluster in YARN, do the computation, kill the cluster. Submit you application to standalone Spark cluster (Mesos), kill it. The biggest problem with this approach is that after your application is done, the JVM is killed, `SparkContext` is lost, and even if you are running Spark in standalone mode, all data cached by your application is lost.

We use Spark in a totally different way. We start Spark cluster in YARN, load data to it from HDFS, cache it in memory, and **do not shut it down**. We keep JVM running, it holds a reference to `SparkContext`and keeps all the data in memory on worker nodes.

Our backend application is essentially very simple REST/JSON server built with Spray, that holds the `SparkContext` reference, receive requests via URL parameters, runs queries in Spark, and return responses in JSON.

Right now (July 2015) we have data starting from April, and it's around 100g cached in 40 nodes. We need to keep 1-year history, so we don't expect more than 500g. And we are very confident that we can scale horizontally without seriously affecting performance. Right now average request response time is 1-2 seconds which is really good for our use case.

Spark Best practices

Here are configuration options that I found really useful for our specific task. You can find more details about each of them in Spark guide.

```
- spark.scheduler.mode=FAIR
- spark.yarn.executor.memoryOverhead=4000
- spark.sql.autoBroadcastJoinThreshold=300000000 // ~300mb
- spark.serializer=org.apache.spark.serializer.KryoSerializer
- spark.speculation=true
```

Also, I found that it's really important to repartition your dataset if you are going to cache it and use for queries. The optimalmal number of partitions is around 4-6 for each executor core, with 40 nodes and 6 executor cores we use 1000 partitions for best performance.

If you have too many partitions Spark will spend too much time for coordination, and receiving results from all partitions. If too small, you might have problems with too big block during shuffle that can kill not only performance but all your cluster: SPARK-1476

Other Options

Before starting this project, we were evaluating some other options

Hive

Obviously it's too slow for interactive UI backend, but we found it really useful for batch data processing. We use it to process raw logs and build aggregated tables with `HyperLogLog` inside.

Impala

To get good performance out of Impala, you are required to write C++ user defined functions, and it's was not the task that I wanted to do. Also, I'm not confident that even with custom C++ function Impala can show performance that we need.

Druid

Druid is a really interesting project, and it's used in another project at Collective for a slightly different problem, but it's not in production yet.

- Managing separate Druid cluster – it's not the task that I want to do
- We have batch-oriented process – and druid data ingestion is stream based
- Bad support for some of type of queries that we need – if I need to know intersection of some particular ad with all segments, in case of druid it will be 10k (number of segments) queries, and it will obviously fail to complete in 1-2 seconds

- It was not clear how to get data back from Druid – it's hard to get data back from Druid later, if it will turn out that it doesn't solve out problems well

Conclusion

Spark is Awesome! I didn't have any major issues with it, and it just works! The new DataFrame API is amazing, and we are going to build lots of new cool projects at Collective with Spark MLLib and GraphX, and I'm pretty sure they will all be successful.