© INTERNATIONAL JOURNAL FOR RESEARCH PUBLICATION & SEMINAR

ISSN: 2278-6848 | Volume: 06 Issue: 02 | May-June 2015

Lambda Architecture : Mixing Real-time processing with Batch Processing

Manisha Sethi

Department of Computer Science and Applications,RNCET, Kurukshetra Univerisity, sethi.mani89@gmail.com

Abstract— ***Lambda Architecture is a design principle in BigData systems where dealing with throughput and latency in real-time is the most important. This mixture of batch processing with real-time streaming process provides the benefits of both the approaches, thus making the system having the precomputed views which enables high throughput and fresh calculations are done on online data to provide end result most accurate with high throughput, decent accuracy and low latency. The lambda architecture is inspired by the rise in bigdata architectures striving for accuracy as well as speed.***

Keywords— Real-time, hadoop, storm, batch, lambda, bigdata

I. INTRODUCTION

Today fast processing systems need the high throughput of batch processing architectures and low latency of real-time processing systems. The need for both the low latency , high throughput and accuracy in results inspires the mix of both technology that uses batch layer and speed layer as batch processing framework for historical data and real-time computation systems for on-line data respectively.

Reasons for evolution of this mixed architecture can be stated as below:

i. To serve a wide variety of work flows where low latency is most important but support for ad-hoc queries is also a requirement.

ii. To have a fault tolerant and reliable system

iii. To build a horizontally scalable system

iv. To create an extensible system if new features are required.

v. To develop a system where minimal maintenance is required.

vi. To develop a system which is easily debug-gable.

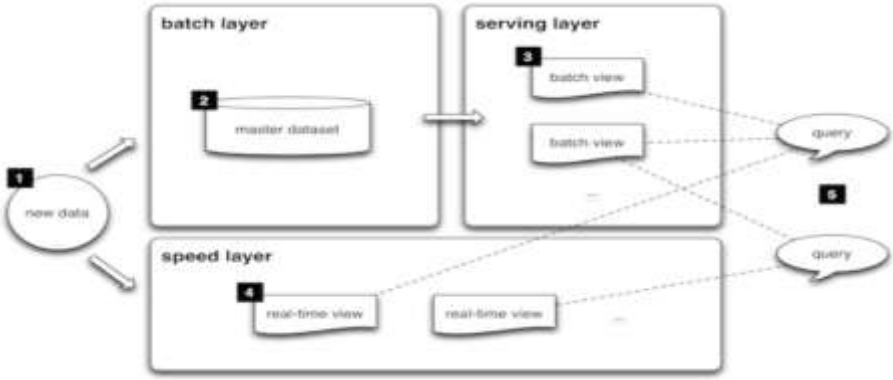


Figure 1: Lambda Architecture layers

II. RESEARCH BACKGROUND AND RELATED WORK

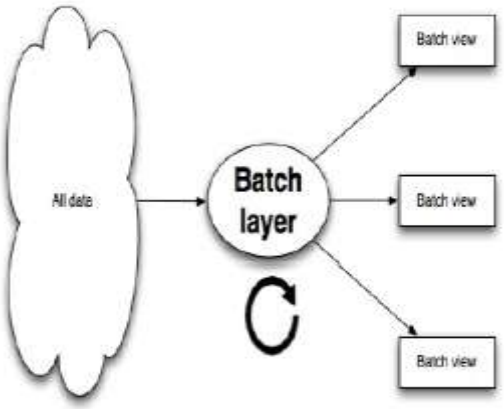
A. BATCH LAYER

Every work starts with a "query = function(all data)" equation. And this query function is called as batch view. Whenever a query is executed , then despite the calculations of output results on the fly at run-time, results are used from this already calculated precomputed view . This precomputed batched view of data is already indexed ahead of time of querying so that it can be accessed quickly with ad-hoc random reads.

Batch layer looks like :

ISSN: 2276-6848 | Volume: 06 Issue: 02 | May-June 2015

Figure 2 : Batch layer



Here, all historical data is executed with a predefined function to have a precomputed batched view of results/outputs , which helps in getting result values when you need faster access as data is already indexed with batch layer. Considering the example :

A web analytics application, if we want to query number of page views(visitors) for a URL within a range of dates. Then if we compute this count on fly by scanning all days and URLs it will cause high latency in result.

The batch view approach runs a predefined computation function on all the page views to pre-compute an index for the data from a key of [url, day] in order to get count of the number of page views for particular URL for specific day. Then, to resolve our query, we retrieve all of the values from that view for all of the days within that time range and sum up the counts to get the result. The precomputed batch view indexes our data by URL, so we can quickly retrieve all of the data points as and when we need to complete the query. And obtain the faster results.

The Batch layer stores the main copy of dataset and pre-computes views on that master data set. Master data set is a huge dataset with high number of records. Sometime this master dataset is called to be the historical data where the processing is done ahead of the run-time when actual query will be executed. This fastens the response time and gets the results faster as processing is already done and what query wants in results is already there. But batch layer suffers from the limitation of real-time response for the latest upcoming data processing which might change the results and hence batch layer needs results needs to be accurate and corrected with the real time data process layer.

The Batch Layer needs to do the following things :

i .Storing of the master data set which is growing and immutable.

ii. Computing arbitrary functions on the master dataset.

Hadoop is the basic example of a batch processing system, and we will use Hadoop to demonstrate the concepts of the batch layer.

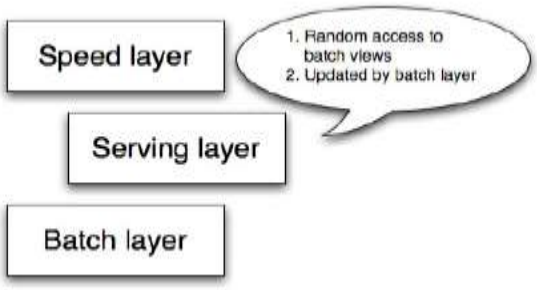
This code computes the number of pageviews for every URL, given an input dataset of raw pageviews. What's interesting about this code is that all of the concurrency challenges of scheduling work, merging results, and dealing with runtime failures (such as machines going down) are done for you. Because the algorithm is written in this way, it can be automatically distributed on a MapReduce cluster, scaling to however many nodes you have available. So, if you have 10 nodes in your MapReduce cluster, the computation will finish about 10 times faster than if you only had one node.At the end of the computation, the output directory will contain a number of files with the results.

So all the distribution and co-ordination of data across machines is handled by the framework itself, batch layer helps developer focus only on the code while doing the logic computation , no need is there in architecture for developer focusing on other framework issues.

B. SERVING LAYER

Batch layer emits the views, so next action is to load the views somewhere so that data can be queried. This is where the serving layer comes in. For example, your batch layer may precompute a batch view containing the pageview count for every [url, hour] pair. That batch view is essentially just a set of flat files though: there's no way to quickly get the value for a particular URL out of that output.

Figure 3 :Serving Layer



A serving layer database only requires batch updates and random reads. Most notably, it does not need to support random writes. This is a very important point because random writes cause most of the complexity in databases. By not supporting random writes, serving layer databases can be very simple. That simplicity makes them robust, predictable, easy to configure, and easy to operate. ElephantDB, a serving layer database, is only a few thousand lines of code.

Batch and serving layers satisfy almost all properties

The long update latency is due to the fact that new pieces of data take a few hours to propagate through the batch layer into the serving layer where it can be queried.

The important thing to notice is that, other than low latency updates, the batch and serving layers satisfy every property desired in a Big Data system. Let's go through them one by one:

i. Robust and fault tolerant: The batch layer handles failover when machines go down using replication and restarting computation tasks on other machines. The serving layer uses replication under the hood to ensure availability when servers go down. The batch and serving layers are also human fault tolerant, since, when a mistake is made, you can fix your algorithm or remove the bad data and recompute the views from scratch.

ii. Scalable—Both the batch layer and serving layers are easily scalable. They can both be implemented as fully distributed systems, whereupon scaling them is as easy as just adding new machines.

iii. General—The architecture described is as general as it gets. You can compute and update arbitrary views.

iv. Extensible—Adding a new view is as easy as adding a new function of the master dataset. Since the master dataset can contain arbitrary data, new types of data can be easily added. If you want to tweak a view, you don't have to worry about supporting multiple versions of the view in the application. You can simply recompute the entire view from scratch.

v. Allows ad hoc queries—The batch layer supports ad-hoc queries innately. All of the data is conveniently available in one location and you're able to run any function you want on that data.

vi. Minimal maintenance—The batch and serving layers consist of very few pieces, yet they generalize arbitrarily. So, you only have to maintain a few pieces for a huge number of applications. As explained before, the serving layer databases are simple because they don't do random writes. Since a serving layer database has so few moving parts, there's lots less that can go wrong. As a consequence, it's much less likely that anything will go wrong with a serving layer database, so they are easier to maintain.

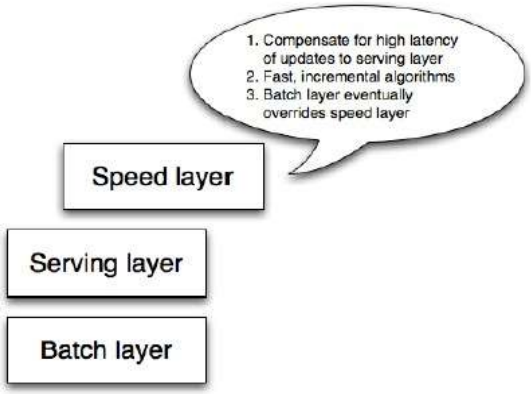
vii. Debuggable—You will always have the inputs and outputs of computations run on the batch layer. In a traditional database, an output can replace the original input—for example, when incrementing a value. In the batch and serving layers, the input is the master dataset and the output is the views. Likewise, one have the inputs and outputs for all of the intermediate steps. Having the inputs and outputs gives one all the information one need to debug when something goes wrong.

The batch and serving layers satisfy almost all of the properties you want with a simple and easy to understand approach. There are no concurrency issues to deal with, and it scales trivially. The only property missing is low latency updates. The final layer, the speed layer, fixes this problem.

C. SPEED LAYER

The serving layer updates whenever the batch layer finishes precomputing a batch view. This means that the only data not represented in the batch views is the data that came in while the precomputation was running. All that's left to do to have a fully real-time data system—that is, arbitrary functions computed on arbitrary data in real time—is to compensate for those last few hours of data. This is the purpose of the speed layer.

Figure 4 : Speed layer



one can think of the speed layer as similar to the batch layer in that it produces views based on data it receives. There are some key differences, though. One big difference is that, in order to achieve the fastest latencies possible, the speed layer doesn't look at all the new data at once. Instead, it updates the realtime view as it receives new data instead of recomputing them like the batch layer does. This is called incremental updates as opposed to recomputation updates. Another big difference is that the speed layer only produces views on recent data, whereas the batch layer produces views on the entire dataset.

Let's continue the example of computing the number of pageviews for a URL over a range of time. The speed layer needs to compensate for pageviews that haven't been incorporated in the batch views, which will be a few hours of pageviews. Like the batch layer, the speed layer maintains a view from a key [url, hour] to a pageview count. Unlike the batch layer, which recomputes that mapping from scratch each time, the speed layer modifies its view as it receives new data.

When it receives a new pageview, it increments the count for the corresponding [url, hour] in the database.

The beauty of the lambda architecture is that, once data makes it through the batch layer into the serving layer, the corresponding results in the realtime views. This means you can discard pieces of the are no longer needed realtime view as they're no longer needed. This is a wonderful result, since the speed layer is way more complex than the batch and serving layers. This property of the lambda architecture is called complexity isolation, meaning that complexity is pushed into a layer whose results are only temporary. If anything ever goes wrong, you can discard the state for entire speed layer and everything will be back to normal within a few hours. This property greatly limits the potential negative impact of the complexity of the speed layer.

The last piece of the lambda architecture is merging the results from the batch and realtime views to quickly compute query functions.

III. PROBLEM DESIGN

The two fundamental requirements of big data projects commonly are one-dimensional to begin with. The (near) real-time information extraction from a continuous inflow of data and the analysis of an immense volume of data recurrently are each singular problems. Two solutions, each addressing one of the cases, have become popular in the recent years. Storm, an open source, distributed, real-time computation platform is being used by companies like Twitter and Groupon, and the project has been recently adopted in the Apache foundations incubator. The other, Hadoop, is so well known and widely adopted that it has become synonymous with batch processing.

The problem with these approaches is that business requirements are both historic and real-time—simultaneously. Many organizations find the two challenges of extracting real-time data and analyzing immense volumes of data converge with time. Real-time data accumulates and the inevitable demand for an aggregated historic view requires batch processing. And the batch processing solution is slow, which eventually leads to business users or customers asking to get immediate or near real-time insight, such as the most recent data updates to react faster to market changes.

The idea of the Lambda Architecture is simple and two-fold. First, a batch layer computes views on your collected data and repeats the process when it is done to infinity. Its output is always outdated by the time it is available since new data has been received in the meantime. Second, a parallel speed processing layer closes this gap by constantly processing the most recent data in near real-time. Any query against the data is answered by querying both the speed and the batch layers' serving stores, and the result is merged to give a near real-time view on the complete dataset. The Lambda Architecture itself provides only a paradigm. The technologies with which the different parts of a Lambda Architecture are implemented are independent from the general idea. Due to their popularity, Hadoop and Storm are strong contenders for the speed and batch layer of new data architectures. However, any technology, such as a company's legacy software exhibiting a speed or batch layer characteristic, could fulfill either function.

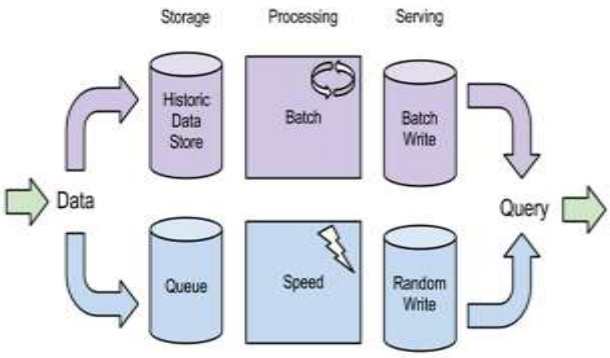


Figure 5: Data pipeline in Lambda Architecure

A. METHODOLOGY/ PLANNING OF WORK :

Steps:

i. All new data is sent to both the batch layer and the speed layer. In the batch layer, new data is appended to the master dataset. In the speed layer, the new data is consumed to do incremental updates of the realtime views.

ii.—The master dataset is an immutable, append-only set of data. The master dataset only contains the rawest information that is not derived from any other information you have.

iii.—The batch layer precomputes query functions from scratch. The results of the batch layer are called batch views. The batch layer runs in a while(true) loop and continuously recomputes the batch views from scratch. The strength of the batch layer is its ability to compute arbitrary functions on arbitrary data. This gives it the power to support any application.

iv. The serving layer indexes the batch views produced by the batch layer and makes it possible to get particular values out of a batch view very quickly. The serving layer is a scalable database that swaps in new batch views as they're made available. Because of the latency of the batch layer, the results available from the serving layer are always out of date by a few hours.

PROPOSED WORK

We want to implement counting the number of appearances of hashtags in tweets, grouped by date, and serve the data as a remote service, for example to be able to populate timelines in a website / mobile app (e.g. give me the evolution of mentions for hashtag "california" for the past 10 days).

The requirements for the solution are:

- It must scale.
- It must be able to serve low-latency requests to potentially a lot of concurrent users asking for timelines.

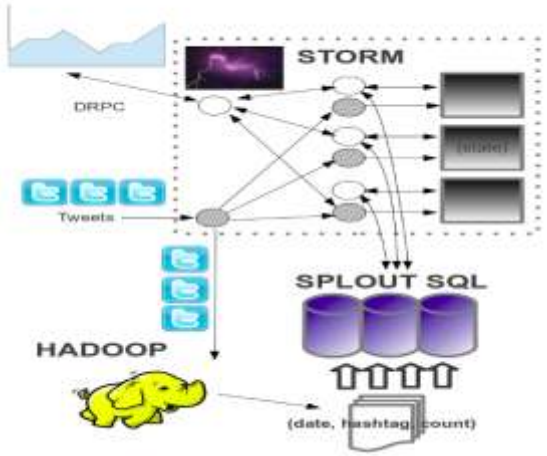


Figure 6: Archtitecture

Using Hadoop to store the tweets and a simple Hive query for grouping by hashtag and date seems good enough for calculating

the counts. However, we also want to add real-time to the system: we want to have the actual number of appearances for hashtags updated for today in seconds time. And we need to put the Hadoop counts in some really fast datastore for being able to query them.

A. Trident

Trident is an API on top of Storm. Trident provides higher-level constructs on top of Storm, similar to those that Cascading provides to Hadoop (each(), groupBy()). It also provides wrappers and primitives for saving/retrieving state in a topology, be it in memory or in a persistent datastore.

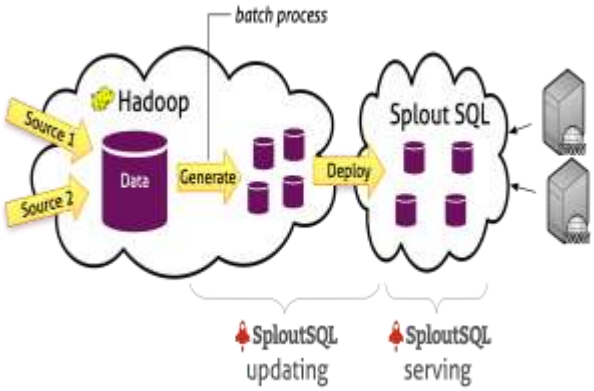
B. Splout SQL

Splout SQL is a database developed by us that can pull data from Hadoop very efficiently. You can think of it as an ElephantDB with SQL. It is a partitioned, read-only, highly performant SQL database for Hadoop-scale datasets. Slout allows serving an arbitrarily big dataset with high QPS rates and at the same time provides full SQL query syntax. Splout is appropriated for web page serving, many low latency lookups, scenarios such as mobile or web applications with demanding performance.

Splout uses Pangool for decoupling database creation from database serving. The data is always optimally indexed and evenly distributed among partitions, plus updating all the data at once doesn't affect the serving of the datasets. Splout can import data directly from Hive, Cascading or Pig.

Splout is horizontally scalable. You can increase throughput linearly by just adding more machines. Splout is replicated for failover. Splout transparently replicates your dataset for properly handling failover scenarios such as network splits or hardware corruption.

Figure 7: Architecture in Tweet Hashtag flow



Putting All Layers together

Tweets are fed into the system (1), for example through a queue from where we can pull them (Storm has connectors for kafka , JMS and Kestrel, but you can easily develop your own one too). A Trident stream (2) saves them into Hadoop (HDFS) and processes them in real-time for creating an in-memory state with the counts by date. In Hadoop (3), where we have all the historical data available, we can run a batch process that aggregates the tweets by hashtag and date and generates a big file from it. After that, we can use Splout SQL's command-line or API tools for indexing the file and deploying it to a Splout SQL cluster (4), which will be able to serve all the statistics pretty fast. Then, a second Trident stream (DRPC) can be used to serve timeline queries (5), and this stream will query both the batch layer (through Splout SQL) and the real-time layer (through the first stream's memory state), and mix the results into a single timeline response. We will see each of these things in more detail.

The batch layer:

he batch layer is the simplest one. One just need to append all tweets in the HDFS and periodically run some simple process that aggregates them by hour date. The dataset looks like this :

```
hashtag 2009081313      1       calirfonia
hashtag 2009101713      1       caliroadtrip
hashtag 2009101815      2       caliroadtrip
hashtag 2009080813      1       caliroots
hashtag 2009092807      1       caliroots
```

Figure 8: Input Data Set for processing

There is plenty of literature and examples on the Internet on how to do such simple tasks with (from lower to higher level): Pangool, Cascading, Pig or Hive. The idea is that the output of the batch layer should look like above: a tabulated text file with hashtags counts. Because you save all the tweets in the HDFS, you can run a batch process that calculates many other things, and which recalculates everything from scratch every time. You have complete freedom and fault-tolerance here

The serving layer

A serving layer database only requires batch updates and random reads. Most notably, it does not need to support random writes. This is a very important point because random writes cause most of the complexity in databases. By not

supporting random writes, serving layer databases can be very simple. That simplicity makes them robust, predictable, easy to configure, and easy to operate. ElephantDB, a serving layer database, is only a few thousand lines of code.

Splout SQL's command-line tools for indexing and deploying look like this:

>hadoop jar splout-hadoop-*-hadoop.jar simple-generate -i sample-hashtags -o out-hashtags -pby hashtag -p 2 -s "label:string,date:string,count:int,hashtag:string" --index "hashtag,date" -t hashtags -tb hashtags

>hadoop jar splout-hadoop-*-hadoop.jar deploy -q http://localhost:4412 -root out-hashtags -ts hashtags

The first line generates the indexed data structures (SQL files) needed for serving queries fast and the second line launches a deploy process that moves the files from the Hadoop cluster to the Splout SQL serving cluster.

Because Splout is a partitioned SQL, you need to specify partitioning schema and fields, as well as number of partitions to generate. You can also specify things like "fields to be indexed" if you know what kind of queries you will do. In this case we created a compound index on "hashtag" and "date" which would allow us to extend the application, being able to query the data between arbitrary time periods.

The Speed layer

The real-time layer is implemented using a Trident stream that saves state into a memory map. The code can be found in the topology class (LambdaHashTagsTopology) and it looks like this:

```
TridentState hashTagCounts = topology
  .newStream("spout1", spout)
  // note how we carry the date around
  .each(new Fields("tweet", "date"), new Split(), new Fields("word"))
  .each(new Fields("word", "date"), new HashTagFilter(), new Fields("hashtag"))
  .groupBy(new Fields("hashtag"))
  .persistentAggregate(mapState, new Fields("hashtag", "date"), new CountByDate(), new Fields("datecount"));
```

When developing Trident streams you have to keep in mind two things. One is the way Tuples are mutated around the stream: each() functions process a set of input Fields and emit a Tuple with these input fields together with the output Fields that the function emits. On the other hand, aggregate() functions only emit the fields that are derived from the function. If you want to emit a subset of the Fields of an each() function you can use project(). The other thing is that, because of this, you can't emit output Fields that are named like an input Field (they would collide in the result Tuple).

The DRPC service

One of Storm's goodness is its ability to execute distributed RPC calls, and parallelize them. We can use this service for populating a website that will show timelines for hashtags. Now that we have a batch layer that computes all historical counts and feeds them into Splout SQL, and a real-time layer that can update hashtag counts happening right now in seconds time, how do we put all this together?

The DRPC service we add to the topology looks like this:

```
topology
  .newDRPCStream("hashtags", drpc)
  .each(new Fields("args"), new Split(), new Fields("hashtag"))
  .groupBy(new Fields("hashtag"))
  .stateQuery(hashTagCounts, new Fields("hashtag"), new MapGet(), new Fields("resultrt"))
  .stateQuery(sploutState, new Fields("hashtag", "resultrt"), new HashTagsSploutQuery(), new Fields("resultbatch"))
  .each(new Fields("hashtag", "resultrt", "resultbatch"), new LambdaMerge(), new Fields("result"))
  .project(new Fields("result"));
```

Queries are parallelized by "hashtag". Two queries are executed in sequence, one to the real-time layer (hashTagCounts) and the other one to the batch serving-layer (sploutState). Note how we use Trident's built-in MapGet() for querying the real-time layer and HashTagsSploutQuery() for querying Splout SQL. Note how the Tuple evolves to have three fields: the hashtag, and the result of each layer. Finally, we use a function called LambdaMerge() that merges the result into a new Tuple field called "result" and project the result to a one-field Tuple that will be returned to the DRPC user.

IV. RESULTS AND ANALYSIS

Batch Layer

1) Prepare data in HDFS

Use commands below :

>hadoop fs -put $TRIDENT_LAMBDA_SPLOUT_HOME/sample-hashtags sample- hashtags

2) Index , partition and load Data in SploutSQL

>hadoop jar splout-hadoop-*-hadoop.jar simple-generate -i sample-hashtags -o out- hashtags -pby hashtag -p 2 -s "label:string,date:string,count:int,hashtag:string" --index "hashtag,date" -t hashtags -tb hashtags

>hadoop jar splout-hadoop-*-hadoop.jar deploy -q http://localhost:4412 -root out-hashtags -ts hashtags

You can check it by looking for Tablespace "hashtags" at the management webapp: http://localhost:4412

Real-Time Layer

Execute the class in this repo called "LambdaHashTagsTopology" from your favorite IDE. This class will:

- Start a dummy cyclic input Spout that emits always the same two tweets.
- Start a Trident topology that counts hashtags by date in real-time.
- Start a DRPC server that accepts a hashtag as argument and queries both Splout (batch-layer) and Trident (real-time layer) and merges the results.

If you want to execute it from command line you can use maven as follows:

```
mvn clean install
mvn dependency:copy-dependencies
mvn exec:exec -Dexec.executable="java" -Dexec.args="-cp target/classes:target/dependency/*
com.datasalt.trident.LambdaHashTagsTopology"
```

When developing Trident streams you have to keep in mind two things. One is the way Tuples are mutated around the stream: each() functions process a set of input Fields and emit a Tuple with these input fields together with the output Fields that the function emits. On the other hand, aggregate() functions only emit the fields that are derived from the function. If you want to emit a subset of the Fields of an each() function you can use project(). The other thing is that, because of this, you can't emit output Fields that are named like an input Field (they would collide in the result Tuple).

V. RESULTS

The example will use two fake tweets for the real-time layer (#california is cool, I like #california) and query Splout where you would have loaded the example dataset – which also contains counts for california. So after running everything you should see something like this:

Result for hashtag 'california' →

[[{"20091022":115,"20091023":115,"20091024":158,"20091025":19}]][[{"20091022":115,"20091023":115,"20091024":158,"20091025":19}]]
Result for hashtag 'california' -> [[{"20091022":115,"20091023":115,"20091024":158,"20091025":19,"20130123":76}]]
Result for hashtag 'california' ->
[[{"20091022":115,"20091023":115,"20091024":158,"20091025":19,"20130123":136}]]
Result for hashtag 'california' ->
[[{"20091022":115,"20091023":115,"20091024":158,"20091025":19,"20130123":192}]]
Result for hashtag 'california' ->
[[{"20091022":115,"20091023":115,"20091024":158,"20091025":19,"20130123":232}]]
Result for hashtag 'california' ->
[[{"20091022":115,"20091023":115,"20091024":158,"20091025":19,"20130123":286}]]

VI. CONCLUSION AND FUTURE WORK

As you see, the last date (which should match today's date) is increased in real-time while the other historical dates are appended – they come from Splout.

The Lambda Architecture is a powerful big data analytics framework that serves queries from both fast and historical data. However, the architecture emerged from a need to execute OLAP-type processing faster, without considering a new class of applications that require real-time, per-event decision-making. In its current form, Lambda is limited: immutable data flows in one direction, into the system, for analytics harvesting.

Using a fast in-memory scalable relational database in the Lambda Architecture greatly simplifies the speed layer by reducing the number of components needed.

Through this example we have shown Trident, an interesting and useful higher-level API on top of Storm, and Splout SQL, a fast, partitioned, read-only SQL for Hadoop. We have also shown a real example of a fully scalable "lambda architecture", omitting certain parts that were not so relevant for the example.

What seemed once impossible, is starting to be achieved: developing Lambda Architectures using a "unified" framework which makes the same code available to both the real-time and batch layer, and combines the results of both layers transparently. Two tools are in the scope for achieving that:

- **SummingBird:** Based on "monoids", supporting Hadoop and Storm as backing infrastructure. Used at Twitter with "Manhattan" as read-only store and Memcached as "real-time" store.
- **Lambdoop:** Based on user-defined "operations", Avro schemas, supports Hadoop and Storm as backing infrastructure and uses HBase as batch store and Redis as real-time store.

With Lambdoop not yet released and Summingbird being a very recent release, the usefulness of these frameworks is yet to be seen. **It might be the case that they are too complex or restrictive**, or that they evolve naturally into something very usable. By now the kind of operations that one can implement using these frameworks seem somehow restricted by the nature of the frameworks themselves, and the final data serving one can expect is restricted to pure **key/value serving**.

Lambda's shortcoming is the inability to build responsive, event-oriented applications. In addition to simplifying the architecture, an in-memory, scale-out relational database lets organizations execute transactions and per-event decisions on fast data as it arrives. In contrast to the one-way streaming system feeding events into the speed layer, using a fast database as an ingestion engine provides developers with the ability to place applications in front of the event stream. This lets applications capture value the moment the event arrives, rather than capturing value at some point after the event arrives on an aggregate-basis.

This approach improves the Lambda Architecture by:

- Reducing the number of moving pieces — the products and components. Specifically, major components of the speed layer can be replaced by a single component. Further, the database can be used as a data store for the serving layer.
- Letting an application make per-event decision-making and transactional behavior.
- Providing the traditional relational database interaction model, with both *ad hoc* SQL capabilities and Java on fast data. Applications can use familiar standard SQL, providing agility to their query needs without requiring complex programming logic. Applications can also use standard analytics tooling, such as Tableau, MicroStrategy, and Actuate BIRT, on top of fast data.

VII. REFERENCES

[1] Schuster, Werner. "Nathan Marz on Storm, Immutability in the Lambda Architecture, Clojure". *www.infoq.com*. Interview with Nathan Marz, 6 April 2014
[2] Bijnens, Nathan. "A real-time architecture using Hadoop and Storm". 11 December 2013.
[3] Marz, Nathan; Warren, James. *Big Data: Principles and best practices of scalable realtime data systems*. Manning Publications, 2013.
[4] Kar, Saroj. "Hadoop Sector will Have Annual Growth of 58% for 2013-2020", 28 May 2014. *Cloud Times*.
[5] Kinley, James. "The Lambda architecture: principles for architecting realtime Big Data systems", retrieved 26 August 2014.
[6] Ferrera Bertran, Pere. "Lambda Architecture: A state-of-the-art". 17 January 2014, Datasalt.
[7] Yang, Fangjin, and Merlino, Gian. "Real-time Analytics with Open Source Technologies". 30 July 2014.
[8] Ray, Nelson. "The Art of Approximating Distributions: Histograms and Quantiles at Scale". 12 September 2013. Metamarkets.
[9] Rao, Supreeth; Gupta, Sunil. "Interactive Analytics in Human Time". 17 June 2014
[10] Bae, Jae Hyeon; Yuan, Danny; Tonse, Sudhir. "Announcing Suro: Backbone of Netflix's Data Pipeline", *Netflix*, 9 December 2013
[11] Kreps, Jay. "Questioning the Lambda Architecure". *radar.oreilly.com*. Oreilly. Retrieved 15 August 2014.
[12] "Altior's AltraSTAR – Hadoop Storage Accelerator and Optimizer Now Certified on CDH4 (Cloudera's Distribution Including Apache Hadoop Version 4)" (Press release). Eatontown, NJ: Altior Inc. 2012-12-18. Retrieved 2013-10-30.
[13] "Hadoop". Azure.microsoft.com. Retrieved 2014-07-22.
[14] "HDInsight | Cloud Hadoop". Azure.microsoft.com. Retrieved 2014-07-22.
[15] Varia, Jinesh (@jinman). "Taking Massive Distributed Computing to the Common Man – Hadoop on Amazon EC2/S3". *Amazon Web Services Blog*. Amazon.com. Retrieved 9 June 2012.
[16] Gottfrid, Derek (1 November 2007). "Self-service, Prorated Super Computing Fun!". *The New York Times*. Retrieved 4 May 2010.