PhEDEx Dataset Replica Monitoring

A.Repecka
Vilnius University, Vilnius, Lithuania

Abstract
This report suggests an apporoach for analysing CMS dataset block replica storage information. It contains of four main segments: reading data, processing data, writing and visualizing results. Most of the decisions made in the approach were chosen in order to get high configurability and good performance. Created service takes advantage of Hadoop, Spark, Elasticsearch and Kibana technologies.

1 Introduction

In CMS data are recorded in files, files are grouped into datasets based on physics. Datasets are then divided in specific size blocks. The CMS data management system distributes the data to tens of sites and tracks in its Oracle database the location of every replica of every block produced by CMS. Since the PhEDEx database only contains the current status of the block replicas, to preserve historical evolution of space occupation daily snapshots are exported to HDFS. The main idea of this project is to extend PhEDEx monitoring with a system to generate statistics about the storage space occupied by different types of datasets at different sites.

2 Input data

Input data for the created service is stored in hadoop file system and is formatted in csv files. Each snapshot of data have a size ranging from 2 to 3.5GB. The schema of data includes these fields: *now, dataset_name, dataset_id, dataset_is_open, dataset_time_create, dataset_time_update, block_name, block_id, block_files, block_bytes, block_is_open, block_time_create, block_time_update, node_name, node_id, br_is_active, br_src_files, br_src_bytes, br_dest_files, br_dest_bytes, br_node_files, br_node_bytes, br_xfer_files, br_xfer_bytes, br_is_custodial, br_user_group, replica_time_create, replica_time_update*. More information about the snapshots can be found in the CERN resource[1].

3 Processing

The next step was to porcess data. As data was exported to hadoop file system and it was exported in huge amounts, it was decided to use spark to process the data in distributed manner using executors memory instead of disk. In order to get even more efficient service dataframes and sparkSQL were used.

3.1 Reading input

Before processing data it is needed to load it from hadoop file system. Service provides few options for this task. First of all, specific hadoop directory name can be specified (*–fname hdfs:///project/awg/cms/phedex/block-replicas-snapshots/csv/time=2016-07-09_03h07m28s*). It is worth mentioning that specified directory cannot contain inner directories. It should only consist of one or more partitioned files. In order to read many directories at once service provides different solution. User can specify base directory - the place where one or more hadoop directories are held (*–basedir hdfs:///project/awg/cms/phedex/block-replicas-snapshots/csv/*). With this parameter it is a must to define date range (*–fromdate 2015-08-04, –todate 2015-08-09*). As example imposes hadoop directory must contain directories with date in its name. Date is expected to be written in YYYY-MM-DD manner. Different date formats are not supported. If

[1] http://awg-virtual.cern.ch/data-sources-index/projects/#phedex-blk-replicas-snapshot

date parameters are not specified default value of current day is used. In order to make reading efficient third party package for reading CSV files was used (*spark-csv*). Detailed implementation of reading input can be found in github repository[2].

3.2 Additional data parsing

To get more sophisticated data analysis it was decided to extract additional data from given fields. For that reason python regular expressions were used.

- Dataset name - /A/B-C-...-X-Y/Z
  - Acquisition era (B). Element between second slash and the first following dash.
  - Campaign (B-C-...-X). Every element in the middle section except processing era (Y).
  - Data Tier (Z). The last segment of dataset name (after third slash).
- Node name.
  - Node tier. It is simply first two symbols of node name (ex.: T0, T1 ...).
- Now (now_sec).
  - Now. As records in one snapshot contain different timestamps(but the same date), date extraction was implemented. It allowed proper grouping by now field.

Snapshots does not provide all the necessary data for block replicas analysis. For that reason two extra csv files were stored inside project data directory. This solution was chosen because the files are relatively small and not likely to be changing. One of them (/data/phedex_groups.csv) is needed to get user group name from id. Second (phedex_node_kinds.csv) is needed to get node kind from node id.

3.3 Dynamic management

In order to analyse and visualize data properly it was decided that service should provide dynamic aggregations. For efficiency reasons dataframes and sparkSQL were used in Spark Jobs.

3.3.1 Grouping keys

User can specify one or more fields as grouping keys. Keys are expected to be written in csv manner. If not - user gets an error. If parameter is not set - default value is set (dataset_name, node_name). Possible values for keys: *now_sec, now, dataset_name, block_name, node_name, br_is_custiodial, br_user_group, data_tier, acquisition_era, node_kind, node_tier, campaign*

3.3.2 Result values

Results are used for specifying result fields for group operation. Results are expected to be written in csv manner. If not - user gets an error. If parameter is not set - default value is set (block_files, block_bytes). Possible values for results: *block_files, block_bytes, br_src_files, br_src_bytes, br_dest_files, br_dest_bytes, br_node_files, br_node_bytes, br_xfer_files, br_xfer_bytes*

3.3.3 Aggregations

Aggregation functions are defined for each result field. If the same aggregation function should be used fo all results columns then it is enough to specify one aggregation function and service automatically apply aggregation for all fields. If user wants to specify different aggregation functions for different columns then aggregations is expected to be written in csv manner and in the exact order as results were specified. If parameter is not set - default value is set (sum) for all results elements. Possible values for aggregations: *sum, count, min, max, first, last, mean, avg-day, delta*.

[2]https://github.com/aurimasrep/PhedexReplicaMonitoring/blob/master/src/python/ReplicaMonitoring/pbr.py#L162

3.3.3.1 Day avarage

All aggregaions is self explanatory except last two (avg-day, delta). Avg-day function simply sums result fields and divides it by distinct number of dates in given data range.

3.3.3.2 Delta

Developed delta function for calculating block transfers between different time intervals. The operation is not dynamic. It has fixed group keys: node_name, date. User can only specify one result field (most often it will be br_node_bytes) and interval in days (–interval 1). The basic algorithm:

1. For all dates in the given period generate interval group.
2. Group data by block, node, interval and retrieve the newest result value in the interval. If newest value in the interval does not match the end in the interval - newest result value 0.
3. Generate records for blocks that disappeared from node (in csv there is no row because block is missing but the period has minus delta).
4. Join existing and generated records.
5. Calculate deltas between intervals.
6. Divide delta_plus and delta_minus columns and aggregate by date and node.

Algorithm was implemented having data shuffled through the network as litlle as possible. Shuffling large amounts of data results in performance issues. Detailed information about delta aggregation with code examples[3] and the mplementation itself[4] can be found on github repository.

3.4 Data ordering

For better data analysis data ordering was implemented. The process use two main arguments (order, asc):

- Parameter order is expected to be written in csv manner and should contain fields only from keys and results parameters. If not - user gets an error. This parameter goes along with parameter asc.
- Parameter asc is expected to be written in csv manner and should contain only 1,0 (1 - ascending, 0 -descending). Symbols 1,0 should appear in the exact same order as columns in order parameter. This parameter goes along with parameter ord. If parameter is not set - all columns will be sorted ascending.

3.5 Data filtering

For better data analysis data filtering was implemented. The process uses one argument "filt". It has a form of *field:regex*. Parameter is expected to be written in CSV manner. Field section of parameter can contain any field from data schema. The second part of parameter can contain regular expressions. This was implemented due to a need to remove unnecesary data from aggregation process (*–filt acquisition_era:Run2012.*,data_tier:RAW*).

3.6 Additional features

This service is expected to be used with variuos aggregations. For that reason it was decided that service should provide a possibility to write header of columns for each partitioned hadoop file. This can be done by using *header* parameter. Also, as for personal usage script produces too many logs, so logs level choice was implemented. User can define *logs* parameter with one of available options: *ALL, DEBUG,*

[3] https://github.com/aurimasrep/PhedexReplicaMonitoring/blob/master/doc/reports/Week_5.md
[4] https://github.com/aurimasrep/PhedexReplicaMonitoring/blob/master/src/python/ReplicaMonitoring/pbr.py#L462

ERROR, FATAL, INFO, OFF, TRACE, WARN. Parameter is not case sensitive. Also, service provides a possibility to be run in local or yarn mode (parameter *–yarn*).

4 Output

The service provides few possibilities to store results. Every of them has to be used for specific purposes.

4.1 Hadoop file system

The service provides a possibility to write data back to hadoop file system at specific path (*–fout hdfs:///user /arepecka/ReplicaMonitoring*). User cannot control number of partitions it is going to get. It depends on input file (how many partiions it has) and the aggregation type. This kind of output should be used for data that should be stored long-term.

4.2 Local file system

The service also lets user to get data to local file system. As this action requires data collection to master node it should be used with caution. It might fill all available memory on master node. As this depends on the aggregation result size it cannot be controlled programatically (different aggregations produces different size results). User is responsible for having enough space on executing environment. User should use the same *fout* parameter with the extra parameter *collect* specified. For convenience purposes data is written in JSON format. This option should be used only with basic aggregations and small files.

4.3 Elasticsearch resource

Service also provides a possibility to output results into elasticsearch data resource (parameter *es*). This should be done with data that needs to be visualized. As elasticsearch resource might be changing configuration file (*/etc/pbr.cfg*) was specified. The file must have section called "ElasticSearch" and elements: node, port, resource (index/type). Service writes data to elasticsearch using "org.elasticsearch.spark.sql" package which is used with the mode "append". For efficiency reasons data is repartitioned before. As service can be used in both: cronjob or custom mode it was decided that is important to distinct this data. It is also important for kibana visualization not to get duplicate results. To solve this problem new field (*origin*) was added to data schema. When running as a cronjob it should have value of "cronjob". In any other use cases user should provide his own value (*–esorigin run2*). Failing to do that service would write "custom" value to origin field. Field origin is only written to elasticsearch data resource (results in hdfs and local fs are not affected).

4.3.1 Hadoop-elasticsearch connection

When implementing this service it was decided that elasticsearch/kibana should be considered as detached sub-system. Elasticsearch might not be available all the time or users might want to export already aggregated data from hdfs to elasticsearch. It showed a need to have direct hdfs-elasticsearch connection. For this reason another service[5] was created. It was also built on top of spark making it possible to use advantages of distributed computing. Service allows user choose one (parameter *fname*) or more (parameters *basedir, fromdate, todate*) directories in hadoop file system. It also reads user defined data schema (specified in environment variable *HDFSES_SCHEMA*). Schema should be defined it json format and with the structure defined in example[6]. As mentioned earlier elasticsearch resource should have a configuration file with section "ElasticSearch" and elements: node, port and resource (index/type). It should be provided in environment variable *HDFSES_CONFIG*. Moreover, it provides

[5]https://github.com/aurimasrep/HdfsES

[6]https://github.com/aurimasrep/HdfsES/blob/master/data/schema.json

a possibility to define origin of data (parameter *esorigin*). Service results in having data exported to specified elasticsearch resource.

5 Performance

Having good performance on data processing was one of the most important goals of this project. Using different measures to achieve this goal allowed service to reduce processing time to a minimum. Measures include:

- Using sparkSQL instead of map/reduce
- Using dataframes instead of rdds
- Avoiding automatic data schema interference
- Using third party packages
- Using spark udfs
- Creating aggregation algorithms which minimize data shuffling through the network

As different aggregations have different performance results we provide two performance figures on different aggregations. It is worth mentioning that both aggregations was processed in yarn mode. Given resources are written in the performance table.

- Group keys: node, user group, acquisition era, data tier, node kind
- Result fields: node bytes, destination bytes
- Aggregations: sum, sum
- Output: hadoop file system

Table 1: Performance of basic aggregations

Interval	Input	Cores	Memory	Output	Duration
1 day	3GB	65	361472MB	600KB	1.6min
1 month	100GB	65	361472MB	18MB	4.3min
3 months	310GB	65	361472MB	52MB	9.7min
1 year	1.1TB	65	361472MB	186MB	28min

- Result fields: node bytes
- Aggregations: delta
- Interval: 1
- Output: hadoop file system

Table 2: Performance of delta aggregations

Interval	Input	Cores	Memory	Output	Duration
2 days	5GB	65	361472MB	12.7KB	2.5min
7 days	20GB	65	361472MB	41.6KB	4min
1 month	90GB	65	361472MB	189.2KB	8.5min
6 months	500GB	65	361472MB	1MB	35min

Delta operation is more commputationally expensive, so for year's data processing it is a must to have more recources (either more executors or more dedicated RAM). Given resources is not enough for

delta operation to process 1.1TB of data. When executors do not have enough memory Spark causes spills to the disk and it produces an error as user does not have enough disk qouta.

6 Visualization

When we have data in elasticsearch resource visualization becomes a formality. First of all, user should configure kibana to point to elasticsearch resource that was used in results storage. Moving forward, index shhould be configured properly (Ex.: now should have time format of YYYY-MM-DD; node, destination bytes should have SI formating, etc...). Then searches, visaulizations and dashboards are created. Example kibana objects are stored in github repository[7]. For clearing out the purposes that service might be used for two examples were given.

6.1 Current data analysis

1. Define search that filters records which have an origin of cronjob and node kind of disk
2. Define search that filters records which have an origin of cronjob and node kind of mss
3. For each search create pie charts:
   - Define aggregation as sum of node bytes
   - Split charts by top 5 storage consuming user groups
   - Split slices by;
     - Top 20 storage consuming acquisition eras
     - Top 20 storage consuming data tiers
4. Combine all 4 visualizations in one dashboard and save it with time "yesterday"

Capture of the dashboard can be seen in 1

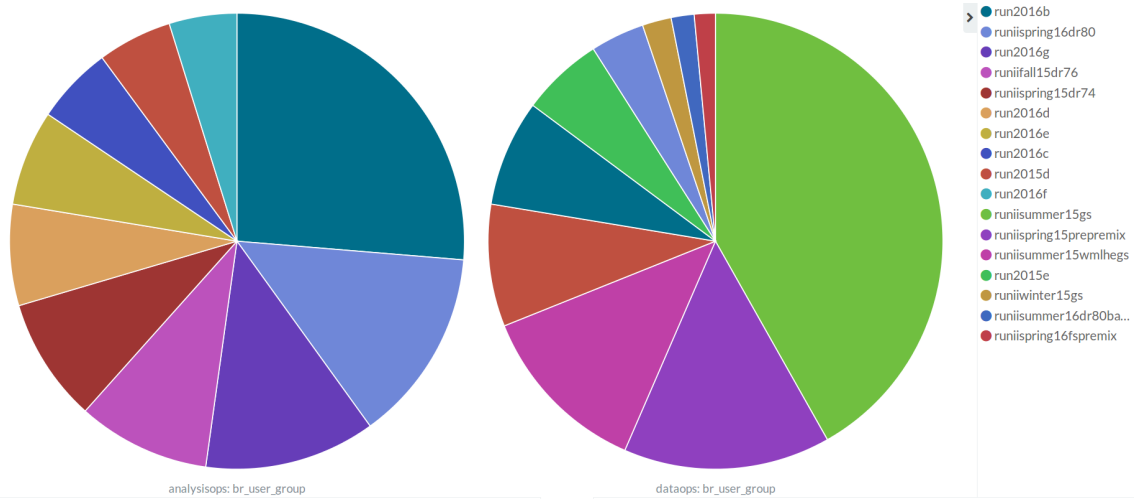

Fig. 1: Pie charts with kibana

6.2 Historical data analysis

1. Define search that filters records which have an origin of cronjob and node kind of disk
2. Define search that filters records which have an origin of cronjob and node kind of mss

[7] https://github.com/aurimasrep/PhedexReplicaMonitoring/blob/master/data/kibana_objects.json

3. For each search create bar charts:
   - Define Y axis as sum of node bytes
   - Define X axis as field now
   - Split charts by top 3 storage consuming user groups
   - Split bars by;
     - Top 20 storage consuming acquisition eras
     - Top 20 storage consuming data tiers
4. Combine all 4 visualizations in one dashboard and save it with time "Last 7 days"

Capture of the dashboard can be seen in 2

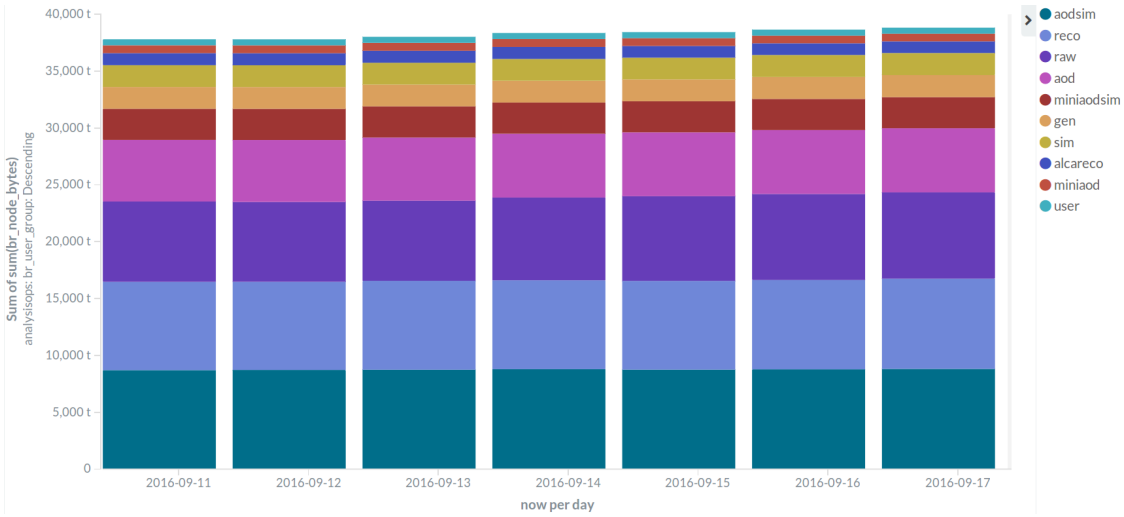

Fig. 2: Bar charts with kibana

7 Deployment

Service was implemented in development environment which consisted of personl VM (where elasticsearch/kibana were installed) and analytix cluster (where spark jobs were submitted). Later it was moved into more accesible environment - WMArchive node. Migration process included:

- Creating rpm packages for the service
- Writing deployment and management scripts[8]
- Solving kerberos authentication issues while accesing hdfs
- Deploying elasticsearch and opening port 9200 iptables
- Deploying kibana and opening port 5601 iptables
- Making rpm packeges from third-party resources needed for service

Future perspecitive is to migrate service under DMWM umbrella nad use elasticsearch/kibana from central CERN IT service. In order to use service in personal environment user should specify some environment variables (in the common usage variables will be set in manage script by programmer)

- SPARK_CSV_ASSEMBLY_JAR - spark-csv package path

[8] https://github.com/vkuznet/deployment/tree/master/phedexreplicamonitoring

- ES_HADOOP_JAR - elasticsearch-hadoop package path
- PYTHONPATH - path to projects python folder. Ex.: /src/python
- PBR_DATA - path to external data (phedex_groups.csv, phedex_node_kinds.csv files). Ex.: /data
- PBR_CONFIG - path to elasticsearch configuration (pbr.cfg). Ex.: /etc

8 Conclusion

Service really helps to analyze data by reducing its dimensions (aggregation) and visualizing it. Proposed approach has three main advantages. First of all, it is very efficient as it is build on top of Spark which allows distributed computing using executors memory instead of disk. Moreover, it is fully-covered. By using this service user can get everything from raw data in hadoop file system to visualizations and dashboards in kibana. Lastly, it is highly configurable. If a user wants to look at data from different perspective all he need to do is to change service parameters. By doing that he gets different agregations, different results written in hdfs and elasticsearch, different visualizations and dashborads in kibana. All of this without a need to change the service itself.