

**EGI-Engage**

Appendix to Analysis on Techniques to Manage Big Data on the EGI Accounting System

Appendix to D3.6

|  |  |
| --- | --- |
| **Date** | 21 December 2016 |
| **Activity** | WP3 |
| **Lead Partner** | STFC |
| **Document Status** | DRAFT |
| **Document Link** | https://documents.egi.eu/document/XXX |

Abstract

This report forms an appendix to deliverable D3.6 “Analysis on Techniques to Manage Big Data on the EGI Accounting System” and covers the testing performed on the techniques covered in that deliverable. The investigation focused on the accounting repository summarising process which runs daily on the repository and currently takes a significant amount of time. Following a description of the testing performed, a summary of the results is presented, alongside future work that should improve the performance of the accounting repository.

**COPYRIGHT NOTICE**



This work by Parties of the EGI-Engage Consortium is licensed under a Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/). The EGI-Engage project is co-funded by the European Union Horizon 2020 programme under grant number 654142.

**DELIVERY SLIP**

|  |  |  |  |
| --- | --- | --- | --- |
|  | ***Name*** | ***Partner/Activity*** | ***Date*** |
| **From:** | A. Coveney, G. Corbett | STFC/JRA1 | 2016-12-20 |
| **Moderated by:** |  |  |  |
| **Reviewed by** |  |  |  |
| **Approved by:** |  |  |  |

**DOCUMENT LOG**

|  |  |  |  |
| --- | --- | --- | --- |
| ***Issue*** | ***Date*** | ***Comment*** | ***Author/Partner*** |
| **v0.1** | 2016-12-20 | Document creation | A. Coveney / STFC |
| **V0.2** | 2017-04-13 | Review | D. Scardaci /EGI F. -INFN |

**TERMINOLOGY**

A complete project glossary and acronyms are provided at the following pages:

* <https://wiki.egi.eu/wiki/Glossary>
* <https://wiki.egi.eu/wiki/Acronyms>

**Contents**

[1 Introduction 5](#_Toc470083025)

[2 Testing of Techniques 6](#_Toc470083026)

[2.1 Parallel Batch Processing 6](#_Toc470083027)

[2.2 Replace APEL backend 7](#_Toc470083028)

[2.2.1 Replace with Percona 7](#_Toc470083029)

[2.3 Combining Hadoop Tools and MySQL 7](#_Toc470083030)

[2.3.1 PySpark on top of a MySQL database 7](#_Toc470083031)

[2.4 Replace with Hadoop/HDFS 8](#_Toc470083032)

[2.4.1 PySpark on top of a HDFS 8](#_Toc470083033)

[2.4.2 Phoenix and HBase 9](#_Toc470083034)

[2.5 Parallel Stream Testing 9](#_Toc470083035)

[2.5.1 Using PySpark in streaming mode 9](#_Toc470083036)

[3 Summary 11](#_Toc470083037)

**Executive summary**

This investigation focused on the accounting repository summarising process which runs daily on the repository. This process aggregates individual job records from the JobRecords table by month and groups them by various metrics. This is the most time consuming part of the summarising process. Normalising and copying summaries was excluded from tests, as in production these steps take less than a minute each. With some technologies, only the aggregation part of the summariser process was conducted, excluding the update/insert element.

Running multiple summariser processes in parallel reduced the summarising time significantly and, alongside tweaks to the partition layout, seems to be a quite promising improvement to the current summarising process that could have a positive impact for a reasonable amount of effort. Replacing the APEL MySQL backend with Percona Server did not affect the running time of the summariser and so does not seem to be of any benefit. This may be due to the structure of the current architecture and database schema, and alternative ones could bring benefits, but then this should benefit any version/flavour of MySQL. No clear benefits were seen when using Apache Spark in any of the tests performed whether that was using it on top of a MySQL database or an HDFS, or in streaming mode. This seemed to be due to the architecture of the accounting repository not matching the use cases that Spark is designed for. Testing on Phoenix and HBase had to be abandoned as after a week the test data had not finished loading, so no clear conclusion could be drawn

Overall, it seems that the technique with the clearest benefit is the parallel batch processing, whereas a lot the techniques presented as big data tools require a lot of expertise and effort to get a benefit from and that benefit is not often clear. Tools like HBase, Phoenix and storing data in the HDFS would involve porting a lot of the database to a new technology. Whereas parallelising the MySQL calls is relatively simple and effective change.

Incremental improvement of the accounting repository is planned that will incorporate some of the knowledge from this research relating to parallelising the batch MySQL summarising process. Additionally, it would be possible to create a “light” summariser run that only processes job records from the latest month, or runs that only process data from sites that send summaries. This would help to produce more frequent updates of the majority of the data, which tends to be either data from recent job runs or already provided as a summary to the repository.

# Introduction

This investigation focused on the accounting repository summarising process which runs daily on the repository. This process aggregates individual job records from the JobRecords table by month and groups them by various metrics such as site the job ran at, job ownership parameters like virtual organisation (VO) and global user name, and job parameters like machine performance and number of cores used. This is the most time consuming part of the summarising process.

Normalising and copying summaries was excluded from tests, as in production these steps take less than a minute each. With some technologies, only the aggregation part of the summariser process was conducted, excluding the update/insert element.

# Testing of Techniques

## Parallel Batch Processing

The APEL JobRecords table is partitioned by the month the job ends, with each month having its own partition. The APEL development database, which has significantly less entries in the JobRecords table than the production database, was used to investigate if the summariser process could take better advantage of this partitioning. On the development infrastructure, the existing summariser process took in the order of 5 minutes[[1]](#footnote-1). Investigation of the data stored revealed that January, July and August 2013 each contributed about 9% of the total JobRecords. As such, it was expected that doing a monthly summary, which only searched one of those month’s partitions, should take 9% of the time, or around 30 seconds.

The partitions of the JobRecords table are defined using the MySQL built in function TO\_DAYS() which returns the number of days since year 0 given a date. As such, the summariser has to be written using a specific date and time format. Without this format, the monthly summary will search all partitions, even if it only actually reads from one partition, and take the same amount of time to execute as a full summary[[2]](#footnote-2). The monthly summariser would require a WHERE clause with dates in ‘YYYY-MM-DD hh:mm:ss’ format to summarise July 2013. i.e. WHERE EndTime BETWEEN '2013-07-01 00:00:00' AND '2013-07-31 23:59:59'. When passed dates in this format, MySQL can infer that it only needs to access the one partition, and this does reduce the execution time to around 30 seconds as expected.

As all processes would be using the same table to write out the summaries produced, it is feasible that this would cause a bottleneck. However, a further test was conducted, running three separate summarisers for January, July and August 2013 as different MySQL processes. It was found that all three finished within the expected 30 seconds, suggesting that the bottleneck may not be an issue with a sufficient number of processors. Of course, this will need to be re-evaluated on a full scale test as the chance of bottle necks forming will increase with the number of processes trying to write to the summaries table.

It should also be noted that because the TO\_DAYS() function is used, it is recommended that the first partition should only contain job records where TO\_DAYS(EndTime) evaluates to NULL, i.e. job records with invalid end times. This behaviour is considered a feature[[3]](#footnote-3). The first JobRecords partition currently contains job records ending before January 2013. As such, each monthly summary will try and read this partition. The partition could be changed to include this TO\_DAYS(EndTime) == NULL partition to alleviate this issue, as such a partition would be expected to be empty.

Running multiple summariser processes in parallel alongside tweaks to the partition layout seems to be a quite promising improvement to the current summarising process that could have a positive impact for a reasonable amount of effort.

## Replace APEL backend

### Replace with Percona

The data from the accounting repository test infrastructure was copied to a virtual machine with two cores and 8GB of RAM. With MySQL Server 5.5 installed, a summariser run took 1 minute and 30 seconds (54 seconds on subsequent runs, possibly due to some caching). Replacing MySQL Server 5.5 with Percona Server 5.5 did not affect the running time of the summariser and so does not seem to be of any benefit. This may be due to the structure of the current architecture and database schema, and alternative ones could bring benefits, but then this should benefit any version/flavour of MySQL.

## Combining Hadoop Tools and MySQL

### PySpark on top of a MySQL database

Apache Spark was installed on a virtual machine and the data from the test infrastructure was copied over. This virtual machine was given 4 cores and 12GB of RAM. A MySQL query was run, without taking advantage of partitioning, to select all job records from 2013. This query requires a full table scan and took 1 minute 40 seconds[[4]](#footnote-4).

A similar query using the Spark Python API (PySpark) was attempted, but could not be executed on this infrastructure, with the process running out of Java heap space. This is due to PySpark having to load the MySQL data into a Spark ‘context’ before running the query. For even larger data sets, such as the production job records, this would be infeasible.

Instead, the test data set was reduced to contain only job records from 2014, approximately 3% of the total data from the test infrastructure. A MySQL query[[5]](#footnote-5) was executed to select the data and group by the same attributes as the summariser would. This query took 0.42 seconds[[6]](#footnote-6). The corresponding query[[7]](#footnote-7) using PySpark and four local threads took twelve seconds. Although only seven seconds was spent actually executing the query, the other five seconds were spent loading data into the Spark context.

Spark does not seem to be suited for use on top of a MySQL database as, while you can use Spark in this way, it is not the main use case for it. This is mainly due to not being able to store enough data in the context and so the whole JobRecords table would need to be loaded into memory every time. Additionally, Spark seems to not be designed for aggregating multiple fields at once and is possibly more suited to tasks like filtering log output using simpler aggregations on a single field at a time. The documentation for Spark was sufficient, when combined with information from question and answer sites, to set up this test and the following ones that used Spark.

## Replace with Hadoop/HDFS

### PySpark on top of an HDFS

A strength of Spark is that it can create a ‘context’ from many different types of datastore. As well as reading from a MySQL database, PySpark can also read CSV files into a ‘context’ and run the same queries and aggregations on that data.

Therefore, the 2014 job records were converted into CSV format. Initially they were stored directly on the local file system. This did not offer a speed up when compared to reading from the MySQL database. Nor did storing the CSV file in a one node HDFS give a noticeable speed up.

Using CSV does allow for larger data sets to be worked on when using PySPark on top of MySQL. The entirety of the data from the accounting repository test infrastructure could be queried7 in 1 minute and 23 seconds. Nearly all of that time was spent processing the data, rather than loading it into the context. A similar query5 took 43 seconds with MySQL. Again, storing the CSV file in a one node HDFS did not give a noticeable speed up. It is expected that a multinode cluster would make the processing quicker. So this type of architecture could be beneficial, however this is hard to confirm due to the lack of an available multi-node cluster, and the effort and expertise required in setting one up.

### Phoenix and HBase

Converting the current JobRecords schema to create a table using Phoenix required a few minimal changes, mainly to data types, i.e. INT to INTEGER, BIGINT to BIGINTEGER, DATETIME to DATE. A larger issue is that only fields in the primary key can be set as not nullable. The current JobRecords schema has several mandatory fields that are not part of the primary key. Neither making these fields non-mandatory nor making them part of the primary key is an attractive solution.

There was also some accounting test data with a single quote in strings, e.g. “SiteName. This broke the parsing of the CSV into HBase/Phoenix. The loading of the data from the test infrastructure was abandoned as after a week of loading it had still not finished. The documentation for Phoenix and HBase was very good and probably the best when compared to the other tools investigated. Unfortunately this technique does not seem to be applicable to the accounting repository.

## Parallel Stream Testing

### Using PySpark in streaming mode

Streaming in PySpark works by defining a Python function that contains the filtering, grouping and aggregation commands. A time window is also defined and once per window this function is executed. All the job records received during that window will be aggregated together.

During a window, APEL would be sent job records. When this window is processed these records would be summarised, or grouped. This would result in “incomplete summaries” as it would contain only the information processed in this window. These incomplete summaries would then need to be combined and the number of jobs aggregated with the incomplete summaries of previous windows. It is not clear what the best way to do this is. One could load the existing summaries into PySpark and then do an aggregation, or keep the summaries in a Spark context, do the aggregation and then write the summaries to a more permanent data store.

If the processing takes longer than the window, PySpark will wait until the processing of the current window is over before starting the next. This means there is no guarantee that a window is processed immediately after the window has closed making it difficult to manually determine when data has been processed. It is unclear if this could be done programmatically so that deletion of files could be performed once they have been processed, although in-order processing is guaranteed with a single stream.

It is also unclear how republishing or deleting data would work if summaries are being continually write-locked and loaded into, and out of, a data store. Currently this is only an issue when the summariser is running as the tables are only write-locked for part of the day.

The processing of an individual window is comparable to the other methods, with the main benefit coming from summarising less data. However, eventually this window’s incomplete summaries would have to be aggregated with summarises in the main summary store resulting in the summaries being run twice per window. Once to aggregate the job records received during that window, and a second time to aggregate the summaries generated this window with previous summaries.

# Summary

Overall, it seems that the technique with the clearest benefit is the parallel batch processing, whereas a lot the techniques presented as big data tools require a lot of expertise and effort to get a benefit from and that benefit is not often clear. They require significant re-architecting of the accounting repository to get an improvement. Tools like HBase, Phoenix and storing data in the HDFS would involve porting a lot of the database to a new technology. Whereas parallelising the MySQL calls is relatively simple and effective change. With the current state of big data tools and techniques, it is hard to improve on a well-optimised SQL solution. Further work is required to investigate different potential architectures of the accounting repository and their relative benefits.

Incremental improvement of the accounting repository is planned that will incorporate some of the knowledge from this research relating to parallelising the batch MySQL summarising process. As part of this, a further test to ascertain the optimum number of processes should be carried out by increasing the number of processes until there is no extra speed up in the overall processing. The testing reported did not look at time-series databases like Influx as they were not covered in the original analysis, but this could be a useful area of future research. Additionally, it would be possible to create a “light” summariser run that only processes job records from the latest month, or runs that only process data from sites that send summaries. This would help to produce more frequent updates of the majority of the data, which tends to be either data from recent job runs or already provided as a summary to the repository.

1. The production summariser process takes in the order of 10 hours. [↑](#footnote-ref-1)
2. <https://dev.mysql.com/doc/mysql-partitioning-excerpt/5.5/en/partitioning-pruning.html> [↑](#footnote-ref-2)
3. <http://bugs.mysql.com/bug.php?id=49754> [↑](#footnote-ref-3)
4. A summariser run on the same virtual machine took 2 minutes 15 seconds, but a Spark version of this query was not performed to simplify the analysis. [↑](#footnote-ref-4)
5. SELECT SiteID, VOID, GlobalUserNameID, VOGroupID, VORoleID, EndYear, EndMonth, InfrastructureType, SubmitHostID, ServiceLevelType, ServiceLevel, NodeCount, Processors, COUNT(\*) as count from JobRecords GROUP BY SiteID, VOID, GlobalUserNameID, VOGroupID, VORoleID, EndYear, EndMonth, InfrastructureType, SubmitHostID, ServiceLevelType, ServiceLevel, NodeCount, Processors ORDER BY EndMonth, GlobalUserNameID, count [↑](#footnote-ref-5)
6. For comparison, a full summariser run of just the 2014 job records on this infrastructure completed in 1 second. [↑](#footnote-ref-6)
7. result = df.groupBy('SiteID', 'VOID', 'GlobalUserNameID', 'VOGroupID', 'VORoleID', 'EndYear', 'EndMonth', 'InfrastructureType', 'SubmitHostID', 'ServiceLevelType', 'ServiceLevel', 'NodeCount', 'Processors').count().orderBy('EndMonth', 'GlobalUserNameID','count').show(668) [↑](#footnote-ref-7)