This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2772241, IEEE Transactions on Parallel and Distributed Systems

Optimised Lambda Architecture for monitoring scientific infrastructure

Uthayanath Suthakar, Luca Magnoni, David Ryan Smith, and Akram Khan

Abstract—Within scientific infrastructure scientists execute millions of computational jobs daily, resulting in the movement of petabytes of data over the heterogeneous infrastructure. Monitoring the computing and user activities over such a complex infrastructure is incredibly demanding. Whereas present solutions are traditionally based on a Relational Database Management System (RDBMS) for data storage and processing, recent developments evaluate the Lambda Architecture (LA). In particular these studies have evaluated data storage and batch processing for processing large-scale monitoring datasets using Hadoop and its MapReduce framework. Although LA performed better than the RDBMS following evaluation, it was fairly complex to implement and maintain. This paper presents an Optimised Lambda Architecture (OLA) using the Apache Spark ecosystem, which involves modelling an efficient way of joining batch computation and real-time computation transparently without the need to add complexity. A few models were explored: pure streaming, pure batch computation, and the combination of both batch and streaming. An evaluation of the OLA on the Worldwide LHC Computing Grid (WLCG) Hadoop cluster and the public Amazon cloud infrastructure for the monitoring WLCG Data acTivities (WDT) use case are both presented, demonstrating how the new architecture can offer benefits by combining both batch and real-time processing to compensate for batch-processing latency.

Index Terms—Big Data, Distributed Systems, Lambda Architecture, Low-latency Computation, Parallel Computing

1 Introduction

Monitoring a scientific experiment requires the gathering of a large volume of data that is produced at a rapid rate. This is illustrated in Figure 1 that shows events produced over various days.

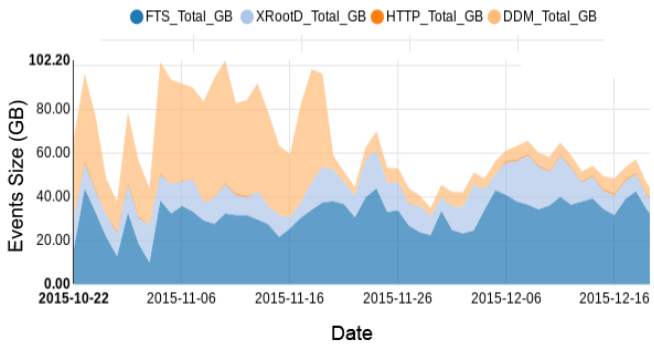

Fig. 1: WDT events size [1].

Scientific infrastructures are highly distributed and heterogeneous platforms with various middleware characteristics, job submission and execution tools, and diverse methods of transferring and accessing datasets. The high computation activity and distributed nature of the infrastructure make the system extremely complex. Monitoring is necessary as it provides a clear status of a task including job distribution by sites and over time, the basis of failure, and advanced plots providing a useful and engaging interface to the users [2]. Efficient monitoring is necessary in order to present a comprehensive strategy to recognise and resolve any potential issues within the infrastructure that may cause failures or inefficiencies. Such failures or inefficiencies may be brought about by cyberattacks, as well as work overloads in the infrastructure. To identify the root cause of a problem with a specific task, the support group requires a monitoring system capable of presenting comprehensive information detailing the task itself [2]. This is an important factor in the overall effective utilisation of resources. Knowledge obtained from the monitoring system can be utilised for automating or streamlining the infrastructure, which would include improved job allocation and increased efficiency in resource deployment.

The main objective of the scientific use case is the need to process an arbitrary set of historical data, and handle recomputation or an old backlog of data injected by producers. In the scientific realm it is normal to have jobs running for a long period of time. Old backlog injection is therefore very common when a long running job is completed. It is necessary to have both a batch layer as well as a streaming layer (real-time) as presented in [3]. However, this requires better mechanisms in place to simplify the system. In this paper an Optimised Lambda Architecture (OLA) has been presented, and evaluated.

The rest of the article is organised as follows. Section 2 of this article examines other Big Data architectures, and introduces the WLCG monitoring use case as well as chal-

- *U. Suthakar, D.R. Smith and A. Khan are with the Department of Electronic and Computer Engineering, Brunel University London, College of Engineering, Design and Physical Sciences, Kingston Lane, Uxbridge, Middlesex, UB8 3PH, UK.*
  E-mail: uthayanath.suthakar, david.smith, akram.khan@brunel.ac.uk
- *U. Suthakar and L. Magnoni are with the CERN, European Organization for Nuclear Research (CERN), Geneva, Switzerland.*
  E-mail: uthayanath.suthakar, luca.magnoni@cern.ch

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2772241, IEEE Transactions on Parallel and Distributed Systems

lenges with handling low-latency computations. Section 3 introduces three variations of Big Data architectures: pure stateless batch computation, pure stateful streaming computation, and a combination of batch and streaming computation. This section also details how the architectures may be joined together using algorithms. Section 4 introduces evaluation results of the proposed architecture. Section 5 highlights the main contributions, and how the proposed architecture has improved the overall performance of the WDT use case.

2 Background

Monitoring events, metadata of the jobs, and data transfers are collected and analysed to produce summary plots used by operators and experts to evaluate computing activities [4]. Due to the high volume and velocity of the events that are produced, the traditional methods are not optimal. The WLCG Data acTivities (WDT) dashboards are a set of monitoring tools utilised to monitor data access, and transfer data across WLCG sites through various protocols and service [3]. The monitored services using WDT include the HTTP federations, XRootD, ATLAS Distributed Data Management (DDM) system, and the File Transfer Service (FTS). The WDT use case is one of the most data intensive applications, and the WDT dashboards struggle from the restriction of the original processing infrastructure [3]. The original architecture relies on an Oracle database to store, process, and serve the monitoring data [3]. Raw monitoring events are archived in tables for several years, while periodic PL/SQL jobs run at regular intervals (10 minutes) to transform raw data into summarised time-series statistics. These are then fed into dedicated tables where they are eventually exposed to the web-framework for user visualisation [3]. For data intensive use cases, such as WDT, this approach has several limitations [3]. Scalability is difficult to achieve; PL/SQL execution time fluctuates from seconds to minutes due to input rate spikes, affecting overall user interface latency [3]. Advanced processing algorithms are too complex to implement in PL/SQL within the dashboard 10 minutes time constraint, and reprocessing of the full raw data can take days [3].

While examining the performance of various traditional and modern architectures, authors in [5][6] talk about the Lambda Architecture (LA). According to [5][6] the LA's primary objective is to fulfil requirements for any infrastructure that is fault-tolerant, robust, or prone to human as well as hardware faults. This is due to its ability to function in a variety of ways, in situations where it is critical to ensure that the system has low-latency while providing regular updates to users. Therefore, the final system developed using the LA is linearly mountable and it scales outwards. The LA has several critical layers, or stages for servicing a system. The first stage involves the entrance of raw data into the system. At this stage, the data is dispatched to two different layers, the speed and the batch layers, where it is processed. In the batch layer, the data is managed within the master data set and pre-computed into batch views. Then, it is forwarded to the serving layer where the batch views are indexed to allow for the data to be queried in low-latency. In the speed layer, only recent data is processed, and the LA is able to compensate for high-latency of updates for the data in the system. Queries entering the system are answered when the results of the batch views in the serving layer and the speed layer are merged [5][6].

In agreement with the views of [5][6], this article [7] explains that another approach to a real-time system and analytics platform is the Kappa Architecture (KA). The KA uses a software architecture approach and it avoids the implementation of relational databases. Instead, it has an immutable log for append only. According to report [8], it is from this log that data is streamed and fed into stores for serving via a computational system. Supplementing the assertions in [8], the report [9] reveals that KA is a simpler architecture in comparison to the LA. In fact, it is a simpler and easier version of LA. This is due to the fact that, excluding the batch processing part of the LA, the parts and functionality of the Kappa architecture are very similar to those of the LA.

By detailing some of the benefits associated with the use of KA, the report [9] explains that it was initially invented to avoid the issue of maintaining two different codes frameworks for the speed and batch layer respectively, as is the case with the LA. This implies that the main idea behind the KA is to ensure that real-time data processing systems, and the continuous data reprocessing systems, are integrated into one effective and efficient system. This is important as both parts are absolutely critical to analytics platforms [7].

Though the KA only has two layers, the streaming layer and the serving layer, it also has a data section that supports other basic functionalities of a real-time system including storing results and historical data. In this approach, the stream processing jobs are tackled first, then the data reprocessing jobs are carried out when some stream processing jobs require modification, alteration, or reprocessing.

The serving layer in the KA, similar to the LA, is used to forward the queries based on the results of the processing carried out. With regard to its application and use, [10] argues that where algorithms for real-time data and historical data processing systems are different from each other, the LA should be used. According to [11] the main benefit associated with KA is that it allows developers of real-time systems to operate, test, and debug on a single framework for processing.

The KA is mainly stream processing, reliable when coupled with tools offering certain guarantees such as Kafka (though Kafka is a temporary buffer, where the retention policy can be hours, days at maximum). Processing an arbitrary set of historical data, a feature which is relevant in the scientific infrastructure to handle recomputation is limited with this architecture.

Authors in [12][13] proposed solutions for tackling the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2772241, IEEE Transactions on Parallel and Distributed Systems

limitation with Big Data architectures that only supports general purpose applications. A computational model called distributed stream processing is presented in [12] to characterise it as a real-time infrastructure, which would work well for a pure streaming use case. This architecture is similar to the KA, due to its inability to process historical data that is necessary for monitoring the scientific infrastructure. The WDT use case also requires a time-critical analytics, an architecture has been proposed in [13], which adopts the idea of the LA using various technologies. This architecture forces code duplication for batch and real-time computations.

The LA was employed to support the WDT use case in [3]. However, by combining and synchronising many technologies, the issue of high complexity became a significant concern. The LA that is presented in [3] demonstrated that it has the ability to work well for monitoring. Most notably, the WDT use case has shown that it outperforms the traditional architecture. However, with the complexity of a three-layer structure that includes various technologies, comes a price when integrating all three layers together to serve several main goals (monitoring infrastructure in real-time, supporting scalability, ease of implementation, maintenance and migration). Having different technologies for each layer would be difficult to integrate, implement, and maintain. There is a pressing need to identify a single solution that can accommodate and integrate the batch layer as well as the streaming layer for monitoring events seamlessly. Apache Spark [14] is a new parallel processing paradigm similar to MapReduce [15], but with improved analytical performance. By exercising in-memory computation, it has the ability to support iterative computation [16][17]. It can also support data streaming, which is useful in optimising the LA to limit code differences between the batch and streaming layers. It can also support SQL-like commands, interactive command line, machine learning, and Graphx [18]. Having Spark batch and streaming under a stack is useful in optimising the LA. The Spark streaming and batch computations adapt the Resilient Distributed Dataset (RDD), an abstract data collection that is distributed across nodes for parallel processing [18][19]. Transformation and computation logics can therefore be reused between batch and streaming layers.

Spark processes are 'lazy' [18], and no action is carried out until it is required. An example would be the RDD, which does not physically hold data. It contains instructions on what to do when an action is called. The RDDs support two types of operations: transformations, which create a new dataset from an existing one, and actions, which return values to the driver program after running a computation on the dataset. For example; the *mapPartition* is a transformation that passes each dataset element to a partition level through a function and returns a new RDD representing the results. Counter to this, *reduceByKey* is an action that aggregates all the items of the RDD using a function and returns the final result to the driver program.

By default, each transformed RDD may be recomputed every time it is put into action. It is also possible to persist an RDD in memory using the persist (or cache) method, in which case Spark would keep the computed data in memory for expedited access the next time it is queried. There is also support for persisting RDDs on disk, or replicated across multiple nodes [19]. When monitoring a scientific infrastructure it is typical that various statistics are derived from the same monitoring events. In-memory storage and computations are profitable as multiple yet distinct computations can be carried out by a job on the cached data. This will make it easier to maintain the job as well. The LA evaluated in [3] employs MapReduce framework which does not support in-memory persistence [15], so data cannot be shared. In order to implement a complex algorithm in MapReduce framework it requires the creation of chained MapReduce jobs. Essentially, the output of a job will need to be directly connected to the input of the next job. Spark does not require this due to in-memory processing. Spark can also support global data sharing using Tachyon (licensed under Apache), which is a memory-centric distributed storage system that can be used for data sharing.

Spark Streaming supports three notable functions:

1. **Cumulative Computations**, which supports cumulative statistics computation while streaming in new data (incremental calculations). Spark Streaming supports maintenance of the state (which is stored information at a given instant in time) for those statistics. The Spark Streaming library has a function called *updateStateByKey* for maintaining and manipulating the state [18].
2. **Windowed Computations**, which is useful when the data received in the last n amount of time is non-trivial. Spark Streaming readily splits the input data into the desired time windows for easy processing, using the window function of the streaming library [18]. A function such as *forEachRDD* allows access to the RDDs created at each time interval.
3. **Transformation**, which returns a new DStream (stream of events) by applying an RDD to RDD function for every RDD of the source DStream [18]. This is where the code can be reused between batch and streaming layers using the *transform()* function as both frameworks support RDD as the core component. This feature also supports merging (i.e. joining) the batch RDDs with the streaming RDDs, which optimises the LA.

3 Architecture and Design

The core part of the OLA inherits the technologies and approaches from [3][20] such as a message broker, data pipeline (Flume), storage (HDFS), and serving layer (Elasticsearch). This is outlined in Figures 2 and 3.

The main requirement of any monitoring architecture is that it be able to provide information about the infrastructure in near-real-time so appropriate action can be taken. Therefore, the following approaches were designed and implemented:

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2772241, IEEE Transactions on Parallel and Distributed Systems

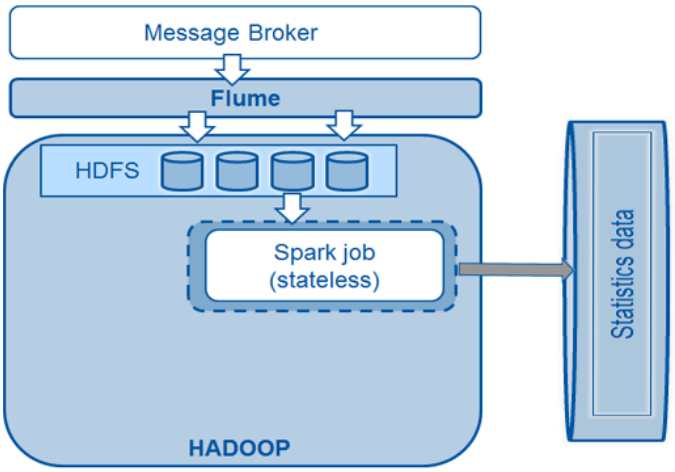

Fig. 2: Pure stateless batch computation. Monitoring events were sent to the Hadoop Distributed File System (HDFS) for batch computation, which can be scheduled to run at any preferred time interval.

1. Pure stateless batch computation as seen in Figure 2, which can be scheduled to run at a preferred interval. The system will not have any knowledge of the previous jobs. This does not support real-time computation, but Spark framework provides in-memory computations. Therefore, the execution time can be compared with the MapReduce framework that was used in [3]. The batch computation can also be used for historical computation (i.e. high-latency).
2. Pure stateful streaming computation as seen in Figure 3, will carry out incremental computation on continuously streaming data 24 hours/7 days a week. From this, it can maintain the state of the computed statistics. It also has a checkpoint mechanism to dump the state to the disk; in case of job failure it can pick up from where it stopped. This method on it is own is enough for real-time computation. This allows the complexities of merging multiple technologies, as in the LA, to be eliminated.
3. A combination of batch and streaming computation is also shown in Figure 3. Pure streaming is enough, but the potential of getting duplicate events from the message brokers due to failure is prevalent. Having pure streaming computation cannot address this issue, as the raw events are dropped once they are processed. The state of the streaming job cannot keep the unique ID of the events once they are aggregated by a key (e.g. sites). Incorporating batch computation can correct the inaccurate statistics as it will recompute whole datasets from the storage layer, eliminating duplicate events. Having a streaming layer do continuous calculation, while scheduling the batch layer to run at specified intervals in order to override the results ultimately validating the statistics seems most appropriate. As pointed out previously, historical computation is necessary in scientific domains, so it is important to incorporate a batch layer in the architecture. To support this approach, the monitoring events were duplicated with one being sent to the HDFS for batch computation, while the other was streamed straight into the streaming receiver. However, there are a variety of complexities that need to be addressed in synchronising these approaches together including: informing the streaming job about newly available data (computed by batch job) so that it may utilise it to override the streaming state as well as the serving layer that is used for storing computed statistics for serving the UI, and a mechanism to eliminate the network communication bottleneck at the serving layer to make sure only the newly streamed data are updated/inserted into the serving layer. This is discussed further in Section 3.1.

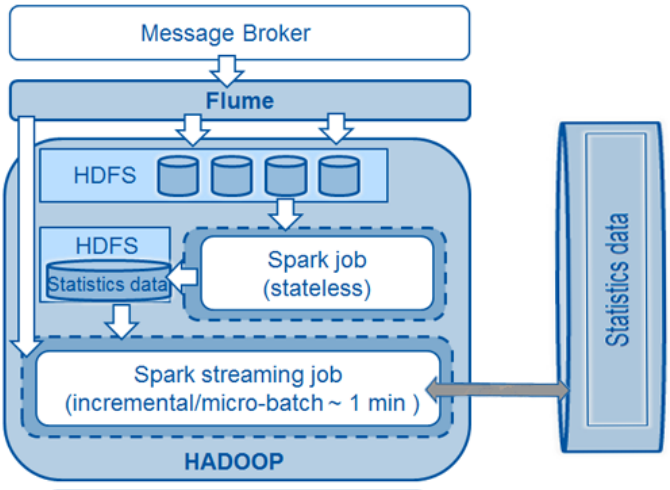

Fig. 3: Pure stateful streaming and combination of both batch and streaming computations. Monitoring events were duplicated with one sent to the HDFS for batch computation, while the other streamed straight into the streaming receiver for incremental computation.

3.1 Merging and synchronising Optimised Lambda Architecture layers

This section explores how the batch, serving, and streaming layers are merged and synchronised.

Batch Layer The batch layer is a high-latency mechanism, so computing an enormous volume of data would result in a delay which would be reflected in monitoring statistics. It is important that the batch should discard some statistics, a certain amount of data which is linked to how often the batch process is executed and how big the dataset is. These 'missing statistics' can be accommodated by the streaming layer.

$$D_{filtered} = F_{discard}\left(D_{raw} \exists \left(B_{time} \ominus B_{interval}\right)\right) \quad (1)$$

In Equation 1, how the monitoring events should be discarded from the computation is represented by a formula expression. In this equation $F_{discard}()$ is the function for discarding events, D_{raw} is the number of raw events prior

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2772241, IEEE Transactions on Parallel and Distributed Systems

to event selection (filter), B_{time} is the batch execution time, $B_{interval}$ is the time interval for discarding events from the batch and $(B_{time} \ominus B_{interval})$ calculates the time frame for selecting events and for emitting all existing, $\exists$, events that match the condition. Assuming a batch job runs at B_{time}, specified $B_{interval}$ 1 hour, a batch should discard all the events in a time $> (B_{time} \ominus B_{interval})$. This will prevent having partially computed results, which will be compensated by the streaming layer.

$$D_{batch} = D_{filtered} \overset{F_{map}(K,V)}{\rightarrow} \overset{F_{reduce}(K,V)}{\rightarrow} S^{data}_{batch} \tag{2}$$

Equation 2 describes how the selected (filtered) events, $D_{filtered}$, will go through a mapping process, $F_{map}()$, to generate (K)ey (a unique ID for the statistics)/(V)alues (matrices values associated with the key) pairs. Subsequently, it will go through the reduce process, $F_{reduce}()$, to aggregate the values by the key from all distributed nodes, which are then stored in a designated storage, S^{data}_{batch}, folder. The batch process will write the result (i.e. a new file) into a known folder on HDFS (this can be replaced by any storage layer, e.g. Tachyon).

Streaming Layer In the consolidated streaming and batch layers, previously computed statistics (if they exist) need to be loaded from the serving layer, which can be represented by Equation 3.

Serving Layer process

$$\begin{aligned} D^{storage}_{stats} &= F^{storage}_{load}(T_{current}, T_{from}) \\ &= \left\{ D^{storage}_{filtered} = \left(D^{storage}_{input} > (T_{current} \ominus T_{from}) \right) \right. \end{aligned} \tag{3}$$

where $D^{storage}_{stats}$ are the loaded pre-computed monitoring statistics from the serving layer, $T_{current}$ is the current timestamp, T_{from} is the timestamp that statistics will be loaded from and $F^{storage}_{load}()$ is the loading function for loading data from the serving layer. If input data, $D^{storage}_{input}$, which are all statistics from the serving layer to the DB load function, are greater than $(T_{current} \ominus T_{from})$ then select, and return the statistics $D^{storage}_{filtered}$.

$$D^{storage}_{processed} = D^{storage}_{stats} \overset{F_{map}(K,V)}{\rightarrow} M^{storage}_{stats} \tag{4}$$

Equation 4 is an expression for $D^{storage}_{processed}$, the mapped and stored statistics selected from the serving layer $D^{storage}_{stats}$ into the memory, which goes through a mapping process, $F_{map}()$, generates (K)ey/(V)alues pairs, which are then stored into memory and/or disk, $M^{storage}_{stats}$ for later usage (i.e. for merging with other layers).

Streaming Layer process

In the streaming layer, the computation is defined as:

$$D^{stream}_{processed} = F^{data}_{transformation}(D_{stream}) \overset{F_{map}(K,V)}{\rightarrow} \overset{F_{reduce}(K,V)}{\rightarrow} \tag{5}$$

where $D^{stream}_{processed}$ is the mapped, aggregated, and computed statistics from streaming monitoring events, D_{stream} is the number of streaming monitoring events, $F^{data}_{transformation}()$ is the function filtering and transforming events, wich then go through the mapping process, $F_{map}()$, which generates (K)ey/(V)alues pairs. Finally, it goes through the reduce process, $F_{reduce}()$, to aggregate the values by the key.

Batch Layer process

The batch reading implementation is defined as:

$$D^{batch}_{loaded} = F^{load}_{batch}\left(D_{batch}\right) \overset{F_{map}(K,V)}{\rightarrow} \tag{6}$$

where D^{batch}_{loaded} are the statistics read from storage and mapped, D_{batch} is the pre-computed statistics from Equations 1 and 2, $F^{load}_{batch}()$ is a function to load only the "new" pre-computed batch statistics and flag the file as "old" once it is loaded successfully which then goes through mapping process, $F_{map}()$. The mapping process does not require any reduction in the statistics as it has already been done by the batch process.

Synchronise and update

The implementation of joining, merging and synchronising statistics from all three layers is defined as:

$$D_{joined} = \left(D^{storage}_{processed} \bigcup D^{batch}_{loaded} \bigcup D^{stream}_{processed} \right) \tag{7}$$

where $D^{storage}_{processed}$ are the statistics loaded from the serving layer, D^{batch}_{loaded} are the data loaded from batch computations, $D^{stream}_{processed}$ are the data computed from streaming data, which are unioned (joined) and returned as a new dataset D_{joined}.

The implementation of the statistics state is defined as:

$$D^{memory}_{state} = F^{update}_{state}(D_{joined}) = \begin{cases} insert, & if \;\; storage = 1 \wedge \;\; state' \\ overwrite, & if \;\; batch = 1 \\ update \vee insert, & if \;\; storage' \vee \;\; batch' \end{cases} \tag{8}$$

where D^{memory}_{state} is where the state of new and old statistics are kept in the memory for incremental calculation, D_{joined} are the joined $D^{storage}_{processed}$, D^{batch}_{loaded}, and $D^{stream}_{processed}$ statistics. The $F^{update}_{state}()$ is the state update function for

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2772241, IEEE Transactions on Parallel and Distributed Systems

updating the statistics and keeping them in the memory. If the statistics are from the serving layer, *storage*, and if it is not already in the state, *state*′, then it should insert the statistics into state memory. If the data are from the batch layer, *batch*, then it should over-write the state memory with the batch statistics. If the statistics are not from serving layer, *storage*′, or batch layer, *batch*′, then they are from the streaming layer (relatively new statistics) so they should be aggregated with the statistics in the state memory, and updated if they already exists (or it should insert the statistics into state memory if they do not exist (totally fresh statistics)).

Update serving layer
Only the new and altered statistics are inserted/updated into the serving layer which is defined as:

$$\left(D^{stream}_{processed} \bigcup D^{batch}_{loaded}\right) \bowtie D^{memory}_{state} \;\; \forall \;\; F^{upsert}_{serving\_layer} \tag{9}$$

The expression in Equation 9 says union (join) the $D^{stream}_{processed}$ and D^{batch}_{loaded} and then leftjoin, $\bowtie$, with the D^{memory}_{state}, to insert/update only the new and updated statistics from the batch (if D_{batch} exists in the spooling location) and streamed statistics into the serving layer. Statistics from $D^{storage}_{processed}$ are not required because they are already in the serving layer. For each, $\forall$, statistics partition, establish a connection to the serving layer and bulk upsert, update the records if it already exists in the serving layer, otherwise insert new records. Finally, set up a checkpoint at a specified interval for recovery in case of any failure.

*- **Summary***
In short, the functions explained above are:

1. The batch layer will write the result in a known folder on HDFS (this can be replaced by any storage layer e.g. Tachyon). The streaming layer will initially load specified data from the serving layer to start incremental calculations from old statistics. Then, at each micro-batch loop, and at the end of the statistics computation, it will check if there are any data in the "batch" folder. If yes, load the computed data, join with its last computed results from history, and insert the newly computed results into the serving layer while updating the history.
2. The batch layer should discard some statistics, certain data which are linked to how often the batch process is executed and the expected delay in processing the ever growing dataset. Assuming a batch run at time *t*, discard the last hour of data, the batch should discard all the statistics referring to time interval $> t - 1$ hour, this will prevent having a partially computed result.
3. The streaming layer will run forever, and the batch process can be executed regularly, or on-demand. Broker queues can be used for ingesting messages from the data pipeline so that if the streaming fails, the data will be retained on the broker indefinitely.
4. The serving layer insertion time will be reasonably short due to micro-batch computation. When the streaming iteration reads the full batch, and inserts it into the serving layer, it will stop processing new data. This has the potential of being noticeable on the UI. This would only be a short-lived temporary glitch. Data would still be present and it would quickly (scaling nodes and paralleling the tasks would improve performance) recover when insertion is over.

4 Performance evaluation of the Optimised Lambda Architecture

4.1 Experiment setup

For the evaluation of the OLA, the same WLCG Hadoop cluster that was used in [3] for evaluating LA was used. The cluster consisted of 15 nodes of Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60 GHz (8 nodes: 32 cores/64 GB, 7 nodes: 4 cores/8 GB). Hadoop-2.6 and Spark-1.6.0 were configured on all machines. The OLA was also evaluated on the Amazon cloud infrastructure [21], as described in section 4.5.

The OLA was evaluated from the aspects of scalability, low-latency processing time, jobs workflow such as parallel processing versus sequential with various data caching mechanism, and execution time over various data size, data format, and data compression.

Three different computing and data intensive algorithms from the WDT use case were used for evaluating the OLA:

1. **Access Pattern** - This algorithm works out what are the hot (popular) and cold (unpopular) data. Hot data is very popular among physicists, so they need to be replicated and distributed across many nodes for load balancing, and better accessibility. An example of the access pattern algorithm:
    1) *Inject Log Message event into Map statement.*
    2) *Split the Log Message into several Log Map Events according to the time bins the initial event belongs.*
    3) *Inject each of the Log Map Events into a single Log Statistic Event and compute the following:*
        a) *If (client domain == server domain) then remote access = 1 else 0*
        b) *If (read bytes + write bytes == file size) then is transfer = 1 else 0*
    4) *Aggregate all the single log statistics:*
        a) *If user domain == null then replace it with server username. In case that server username is also null replace it with "n/a"*
        b) *If file name == null then replace it with "n/a"*
        c) *AVG(file size)*
        d) *If (read bytes $>$ 0) then number of read = 1 else is 0*
        e) *SUM(read bytes)*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2772241, IEEE Transactions on Parallel and Distributed Systems

f) *If (read bytes > 0) then sum(end time - start time) else read time = 0*
g) *If (write bytes > 0) then number of write = 1 else is 0*
h) *SUM(write bytes)*
i) *If (write bytes > 0) then sum(end time - start time) else write time = 0.*

5) *Aggregate and Reduce all the log statistics:*

a) *If there is not already a time bin for the injected log statistic events then create it and insert (establish connection to Elasticsearch and Bulk insert).*
b) *Else update the existing bin (update the Elasticsearch document version).*

2. **Transfer Statistics** - This algorithm has already been used in LA evaluation and works out the average data transfer rate from site A to B. A completed file transfer lasting several hours from site A to site B, also contributes to several time bins in the past. Information about the average traffic from site A to site B has to be updated. An example of the transfer statistics algorithm:

1) *Inject Log Message event into Map statement.*
2) *Split the Log Message into several Log Map Events according to the time bins the initial event belongs.*
3) *Inject each of the Log Map Events into a Single Log Statistic Event and compute the following:*

a) *If writes bytes at close > 0 then we have a client domain else we have a server domain.*
b) *If read bytes at close > 0 then we have a server domain else we have a client domain.*
c) *If client domain = server domain set remote access 0 else set is as 1.*
d) *If writes bytes at close + read bytes at close = file size set is transfer = 1 else is transfer = 0.*
e) *If read bytes at close > 0 then setactivity = 'r'.*
f) *if write bytes at close > 0 then set activity = 'w'.*
g) *if write bytes at close <= 0 and read bytes at close =< 0 then set activity = 'u'.*

4) *Aggregate all the single log statistics:*

a) *If there is not already a time bin for the injected Single log statistic event then create it and insert.*
b) *Else update the existing bin.*

i) *active = active + new.active*
ii) *bytes = bytes + new.bytes*
iii) *activeTime = activeTime + new.activeTime*

3. **User Activities** - This algorithm works out the number of active users, and how much data they have downloaded within a specified time interval. An example of the user activities algorithm:

1) *Inject Log Message event into Map statement.*
2) *Split the Log Message into several Log Map Events according to the time bins the initial event belongs.*
3) *Inject each of the Log Map Events into a Single Log Statistic Event and compute the following:*

a) *If (client domain == server domain) then remote access = 1 else 0*
b) *If (read bytes + write bytes == file size) then is transfer = 1 else 0*
c) *If if user domain == null then replace it with server username*

4) *Aggregate all the single log statistics:*

a) *SUM(read single bytes)*
b) *SUM(read vector bytes)*
c) *SUM(file size)*

5) *Aggregate and Reduce all the log statistics:*

a) *If there is not already a time bin for the injected log statistic events then create it and insert (establish connection to Elasticsearch and Bulk insert).*
b) *Else update the existing bin (update the Elasticsearch document version).*

4.2 Evaluation of the workflow

An evaluation of the workflow in the OLA is presented in this section using those algorithms described in Section 4.1. Job 1, Job 2, and Job 3 in Figure 4 and Figure 5 denote Transform Statistics, Access Pattern, and User Activities algorithm respectively.

The timeline in Figure 4 shows sequential job execution, where jobs were performed one at a time. The next job will only be initiated once the previous job has been completed. This workflow is useful when the later job is dependent on the previous job, e.g. when the second job relies on the results computed by the previous (similar to the MapReduce framework).

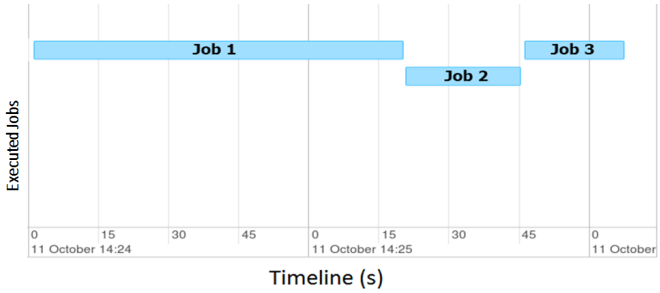

Fig. 4: Sequential jobs execution time. Jobs were executed one at a time.

Figure 5 shows parallel job execution, where multiple concurrent jobs (i.e. Transfer Statistics, Access Pattern, and User Activities) are executed at the same time. This is not achievable with the MapReduce framework presented in [3]. This is the workflow that is beneficial for carrying out in-memory computation, meaning data can be loaded into

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2772241, IEEE Transactions on Parallel and Distributed Systems

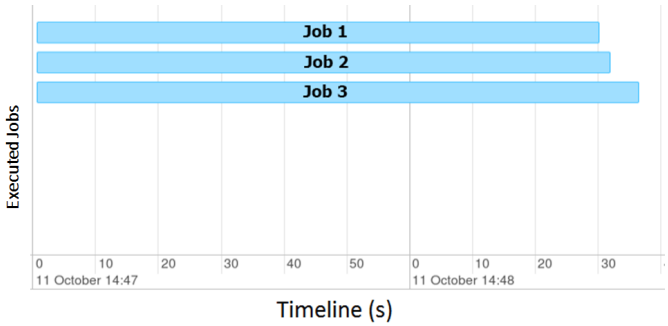

Fig. 5: Parallel jobs execution time. Multiple jobs were executed at a time.

memory, and used by concurrent jobs rather than having each job load data from the storage layer.

After all executors required for a job have been registered, the job commences execution. The executors were removed when the job was complete, in order to make resources available for other jobs.

In Spark, a job is joined with a chain of RDD dependencies arranged in a direct acyclic graph (DAG) as can be seen in Figure 6. From the DAG, it can be seen that the evaluated WDT use case (i.e. Transfer Statistics) first executed a textFile operation to read data from the HDFS, then called the mapPartitions operation to transform the data into Java Objects, calling another mapPartitions operation to extract the required data and to carry out an initial transformation. Subsequently, it then called a reduceByKey function (in the second stage, which is dependent on the first stage) to aggregate the final results, and finally the saveAsTextFile operation was used to save the data into HDFS. It can be seen that each executor immediately applied the subsequent mapPartitions action to the dataset partition after reading it from HDFS in the same task, minimising the data shuffles between nodes. The black dots in the boxes represent RDDs created by the corresponding actions [22].

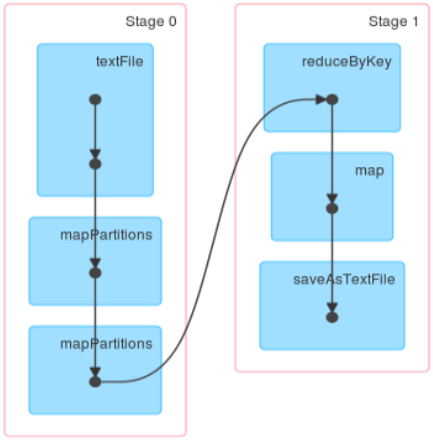

Fig. 6: Overview of job stages.

Spark supports caching stages into memory so that data may be reused, rather than recomputed. Figure 7 illustrates that Stage 3 reads data from HDFS and carries out initial transformation, caching into memory (shown greyed out). The subsequent jobs can easily recover the stage from memory, therefore, reducing re-computation time.

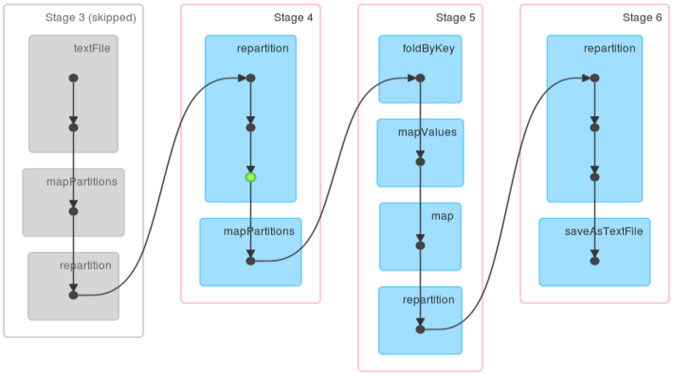

Fig. 7: Cached stages were reused by parallel jobs. The green circle denotes that an RDD is cached from the previous stage. The greyed stage (cached) was skipped by the following concurrent jobs.

Figure 8 shows an insight into Stage 3 of the event timeline from Figure 6. It shows that tasks are distributed to two worker nodes. Most of the execution time was spent on computing the statistics rather than scheduler delay, network or I/O overheads. This is not unexpected since the job involves shuffling very little data. Each executor is performing three tasks concurrently, due to the CPU cores which are explicitly configured with the job submission. The parallelism can be increased or decreased in direct relation to the number of cores, which would have an effect on performance.

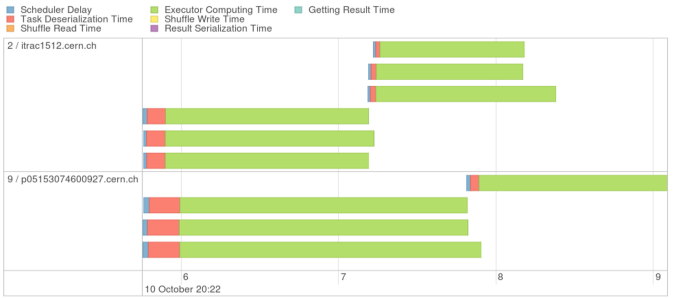

Fig. 8: Concurrent tasks execution. A job was split into multiple tasks and executed in each executor CPU core concurrently.

4.3 Performance evaluation on WLCG environment and WDT use case

In this section the batch computation, as well as the real-time computation of the OLA, were evaluated.

Evaluation of Spark's batch computation

An evaluation of Spark batch computation over increasing dataset size was carried out that was similar to the evaluation detailed in [3]. This evaluation was carried out on the same dataset so that it could be compared with the MapReduce framework computation. Although Spark supports in-memory computation, it was not used in this evaluation as the job consisted of a single algorithm (transfer

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2772241, IEEE Transactions on Parallel and Distributed Systems

statistics). It was unnecessary to persist the dataset into memory as there were no follow-up jobs that could benefit from it. The evaluation results are shown in Figure 9. It can be seen that computing 30 days of the dataset overall was completed in ∼2 minutes by the Avro, CSV and JSON jobs. It can also be seen that execution time linearly increases as the dataset size is increased. Nevertheless, the performance was better when compared with the current approach used by the WLCG. Again the performance pattern of the data types are similar to the MapReduce job, as Avro performed better overall compared to the other jobs. On the other hand, JSON performed poorly when compared with the other jobs. In total, the JSON and CSV jobs took an average of 64% and 14% more execution time compared with Avro, respectively.

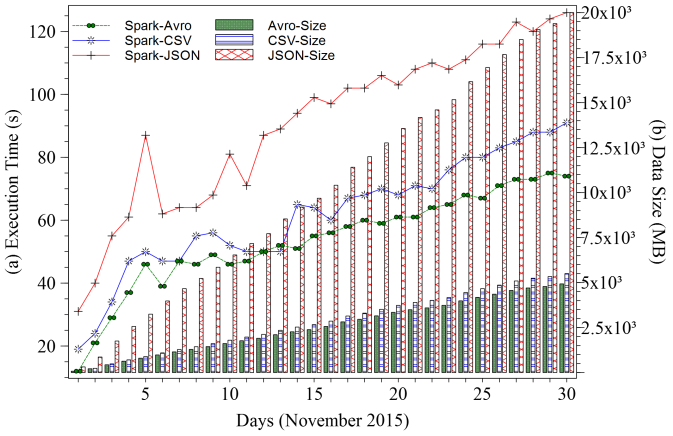

Fig. 9: Computation of Avro, CSV and JSON files over augmented dataset (day 1 to 30 days). The primary axis (a) shows the execution time that is being represented by lines, whereas the secondary axis (b) represents the input data size in Megabytes (MB) which is represented by bars.

A comparison of the job execution time using MapReduce framework from [3] and using Spark is presented in Figure 10. It can be observed that Spark jobs performed much better when compared with the MapReduce jobs, although data persistence was not used in both frameworks. The first day dataset (smallest) took a lot less time for computing using Spark, whereas computing using MapReduce took significantly more. From this observation it can be concluded that Spark took less overhead time in allocating resources when compared with the MapReduce. Nevertheless, MapReduce job execution time stablised as the dataset size was were increased further as only minor oscillations were seen. Comparatively the Spark execution time increased linearly when the dataset size was increased. The Spark-Avro job total execution time on average was 43% less than the MR-Avro job, whereas the Spark-CSV performance improved by 38% when compared to the counterpart, and the Spark-JSON improved 23% compared to its counterpart. All Spark jobs appeared to have performed better than their counterpart, but the most improvement can be seen with the Spark-Avro computation.

Figure 11 shows the execution time over various data partition sizes (i.e. parallelisation, which is the process of splitting the dataset into a number of partitions, and allo-

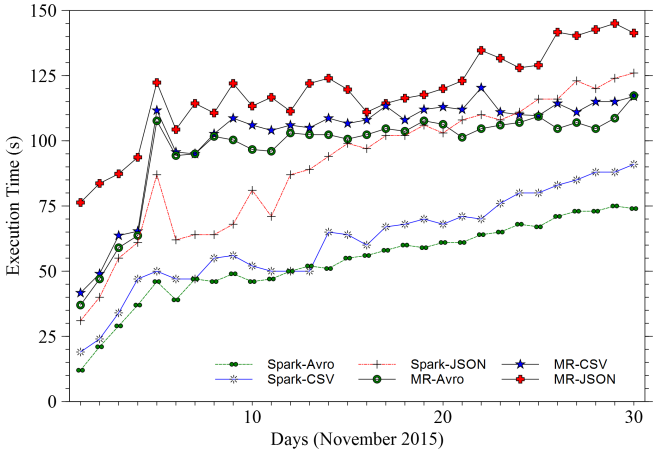

Fig. 10: Comparison of the MapReduce versus the Spark framework against various data types.

cating tasks to process each of those split portions). It can be seen that the execution time improved as the number of partitions was increased (execution time decreased). It can also be observed that after the job met a certain number of partitions, the execution time stabilised. This can be explained by the fact that more partitions would require an equal share of tasks, requiring finding resources, allocating, and garbage collections. This would also require shuffling data over the network. It can be observed that the Avro job performed better compared to the other two jobs.

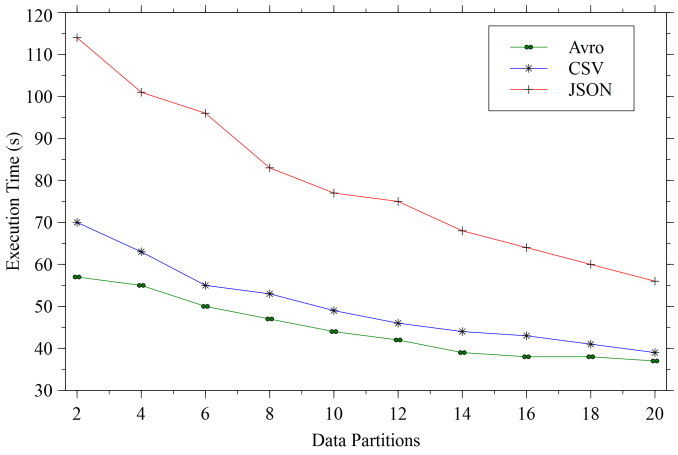

Fig. 11: Execution time versus the number of partitions of various data types.

In the previous evaluation, only a single job (algorithm) was evaluated, so persisting the dataset into memory was not trivial. In order to benefit from the in-memory computation, it was necessary to evaluate multiple jobs (multiple statistics algorithms) on the same dataset (i.e. derive various statistics from the same dataset). The single job assessed previously only parallelises the tasks, but as multiple jobs may be deployed to profit from in-memory computation, it was essential to evaluate parallel job execution versus sequential job execution. Figure 12 illustrates parallel jobs performed better than the sequential jobs; in particular, the cached job performed exceptionally well. However, when comparing the uncached parallel job with the cached sequential job,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2772241, IEEE Transactions on Parallel and Distributed Systems

it is evident that the parallel job performed better, which could only be explained by the simultaneous job execution. The sequential job requires submitting one job at a time so that the next one in the queue can only be submitted when the previous job has been completed. This is not the case for the parallel job as it would submit all jobs at one time. This should not be a problem in the OLA, as it can scale dynamically when there are more demands for resources.

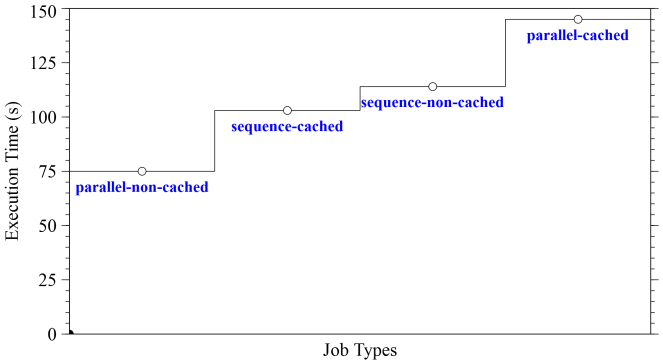

Fig. 12: Comparison of parallel and sequential jobs with cached and uncached datasets. Execution time versus parallel, sequential cached and uncached jobs.

Figure 13 shows the evaluation of execution time over various types of data persistence used in parallel jobs submission. The persistence options are:

- Memory only (MEMORY_ONLY), which only uses the memory for caching the dataset. In the case of a dataset being larger than the memory capacity, it will use the disk for dumping the remaining dataset.
- Memory only with two replications (MEMORY_ONLY_2), which is similar to MEMORY_ONLY but it replicates the dataset two times for improved data availability.
- Memory only with serialisation (MEMORY_ONLY_SER), which is similar to MEMORY_ONLY, but it uses serialisation to compact the data so that more information can be stored into memory as memory spaces are very limited. However, serialising and deserialisation will add computation overhead to the job.
- Memory only with two serialised replications (MEMORY_ONLY_SER_2), which is similar to MEMORY_ONLY_SER but replicates the dataset two times.
- Disk only (DISK_ONLY), which spills the dataset onto the disk.
- Disk only with two replications (DISK_ONLY_2), which is similar to DISK_ONLY but it replicate the dataset two times.
- Memory and disk (MEMORY_AND_DISK), which uses both memory and disk for storage, but some data that need to be persisted into memory are configurable at execution time.
- Memory and disk with two replications (MEMORY_AND_DISK_2), which is similar to the MEMORY_AND_DISK but with two replications of the dataset.
- Memory and disk with serialisation (MEMORY_AND_DISK_SER), which is analogous to the MEMORY_AND_DISK but it uses serialisation to compact the data so that more data can be stored in memory.
- Memory and disk with two serialised replications (MEMORY_AND_DISK_SER_2), which is similar to MEMORY_AND_DISK_SER but with two replications of the dataset.

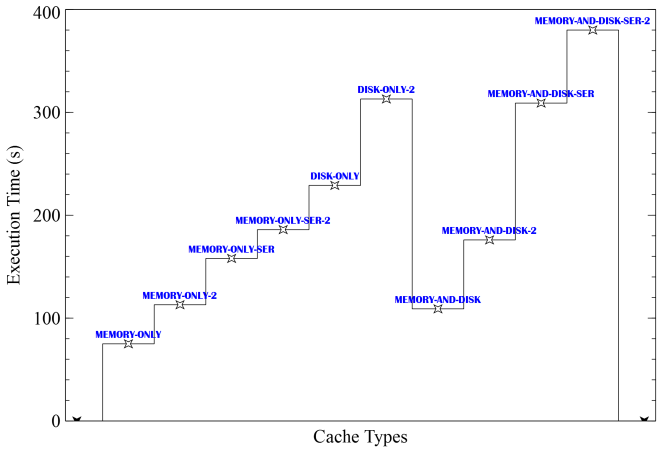

Fig. 13: Comparison of various cache types. Execution time versus computation of data cached in memory, disk, and memory and disk (also a combination of replicated and serialised dataset).

As shown in Figure 13, it is evident that in-memory persistence outperformed the other methods. Having two replications of the dataset into memory did not improve the performance compared to the single dataset. In general, serialisation did not perform well, which is understandable as extra overhead is required for serialising and deserialising the dataset. Using memory and disk performed better than the pure disk option. It was better to recompute from the source rather than reading the cached data from disk. When clustered and compared the execution time of all memory only, disk only, and disk and memory, the disk only options took 104% more execution time than the memory only options, whereas the memory and disk options took 83% more execution time than the memory only options.

The scalability was evaluated by incrementing the number of executor nodes one at a time. The memory size was fixed to 1024 MB. The evaluated dataset size was 7.5 GB, which was used for the following evaluations unless otherwise stated. The total amount of memory allocated for the jobs can be calculated by multiplying the number of executors by amount of allocated memory for each executor (i.e. 1024 MB). The previous evaluations showed that the Avro job performed better compared with the CSV and JSON jobs. Therefore, it was used for the node scalability analysis. Figure 14 shows that execution time improved as the number of executors was increased. However, there was a dramatic improvement in performance in increasing up to three executors. With more than three executors, there was not a significant further improvement. The execution

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2772241, IEEE Transactions on Parallel and Distributed Systems

time decreased by 64% when three executors were used compared to the initial single executor execution time. However, when nine more executors were used, compared with the single executor, the performance improved by 84%. This shows an only 20% improvement using nine executors over three executors. Over allocating resources (in this case executors) can be wasteful, displacing resources that could have been used for other jobs.

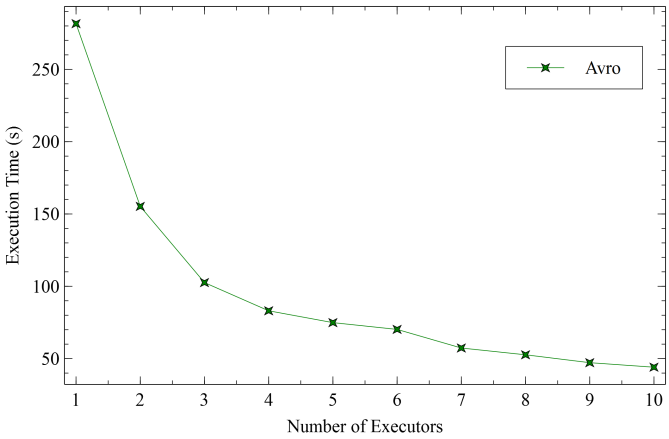

Fig. 14: Execution time versus the number of executors.

For the evaluation of memory usage, the number of executors was fixed at four (for comparison to the former analysis), the number of CPU cores was fixed at one, and memory was increased by 1024 MB at each evaluation. The total amount of memory allocated for the jobs can be calculated by multiplying the amount of allocated memory for each executor by the number of executors (i.e. 4). The allocated memory for each core would be the same as the executors as the core was fixed at 1. Therefore, it does not need to share the memory. The dataset size was the same as in the previous evaluation, which was 7.5 GB. The performance improved rapidly as the memory was increased, as seen in Figure 15. What was indisputable from the results, was that memory plays a significant role in performance. With four executors a better result was achieved by just increasing the memory, rather than using ten executors as can be seen from the previous analysis.

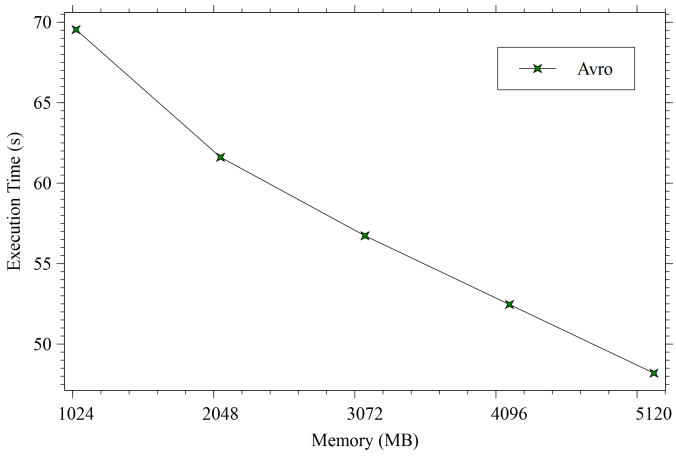

Fig. 15: Execution time versus the amount of memory size.

In order to evaluate the CPU core utilisation in optimising the performance, the number of executors was fixed at four, and the memory was fixed at 2048 MB. The number of cores was increased by one at each execution. Each executor was allocated 2048 MB memory, but there were four executors so the total amount of memory allocated to these jobs was 8192 MB. The amount of memory allocated to each core was calculated by dividing the memory allocated to each executor (i.e. 2048 MB), by the total number of cores allocated to each executor. Again, the performance was improved as the number of cores was increased as seen in Figure 16. Initially, the performance improved steadily as the number of cores increased. However, there was a sharp improvement when 4 cores were allocated. The observed improvement in the performance was caused by the parallelisation of the tasks.

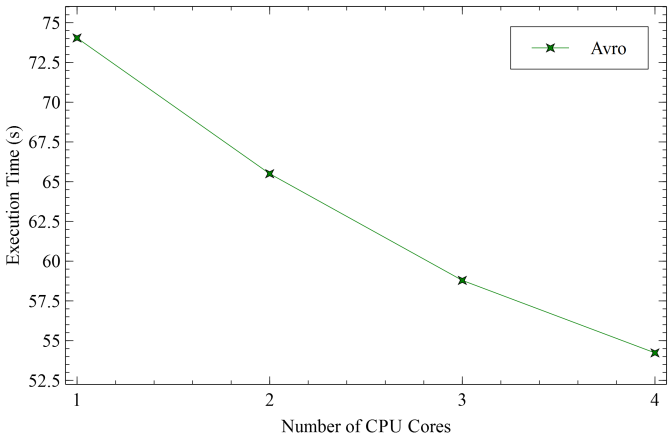

Fig. 16: Execution time versus the number of CPU cores.

Evaluation of Spark's Streaming computation

In [3], Esper was used for carrying out real-time computation, which did not support scalability. However, Spark Streaming supports scalability just as it supports batch computation. The performance observed with Esper was reasonable for the WDT as it computed the events as soon as they were received. Nevertheless, to support the foreseen explosion of volume and speed of the data, scalability is required, so Spark Streaming was investigated. Despite this, it needs to be evaluated to see how it performs on a real life scientific application, which was the same algorithm that was used for evaluating batch computation (i.e. transfer statistics). A few metrics are important in evaluating the streaming layer. One such is the event input rate at which data is being received, while the other is the processing time of each micro-batch. The streaming layer was deployed with three executors, each with 2048 MB memory and three cores. The streaming layer was evaluated on the last 1000 batches of streamed data. The streaming layer was run for ∼15 hours at a two seconds batch interval prior to the evaluation. At the time of the evaluation, the streaming layer had completed ∼27 thousand batches and computed ∼5 million records.

Figure 17 shows that the streaming layer was receiving data at a rate of about 116 events/second on average across all its sources. The streaming layer is capable of handling

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2772241, IEEE Transactions on Parallel and Distributed Systems

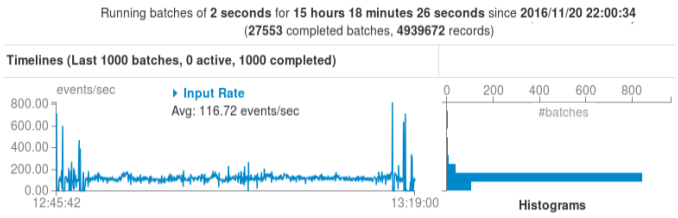

Fig. 17: Streaming data input rate. Streaming job receiving data at a rate of 116 events/second on average.

much larger events than the one shown in Figure 17. However, the source was sending a relatively low load of events at the time of evaluation.

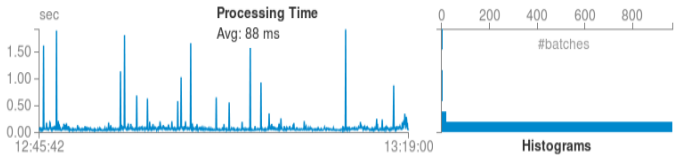

Fig. 18: Streaming data processing time. Processing time shows that these batches have been processed within 88 ms on average.

Figure 18 presents processing time which shows that these micro-batches were processed within 88 ms of being received on average. Displaying a reduced processing time compared to the batch interval means that the scheduling delay (which is the time a batch waits for previous batches to complete [22]) was almost zero as seen in Figure 19. It can also be noted that there were a few spikes on the schedule delay, including when there was a sudden peak in data input rate which increased the schedule delay by 16 ms. The scheduling delay is the key indicator of whether the streaming layer is stable or not [22]. In this particular evaluation it indicated the streaming layer was very steady.

Fig. 19: Schedule delay in processing next batch.

Figure 20 shows that the total delay in scheduling and processing the batches was 105 ms on average. This means the transfer statistics can be presented to the end user within a second.

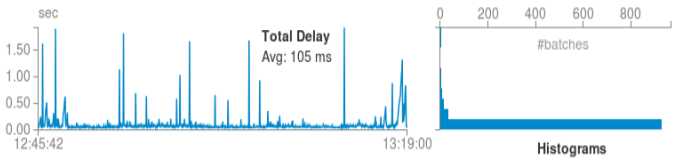

Fig. 20: Total delay in scheduling and processing streaming data.

4.4 Evaluation of monitoring events computation accuracy

To evaluate how accurately the architecture was able to compute the WLCG site throughput in time-series, all three OLA approaches were tested. As shown in Figure 21, the stateless batch job was scheduled to run every five minutes and carry out batch computations on the data stored in HDFS. However, the plot highlighted in Figure 21 shows that some data is missing. This is due to the latency of the batch computation and the unavailability of the data when the job started.

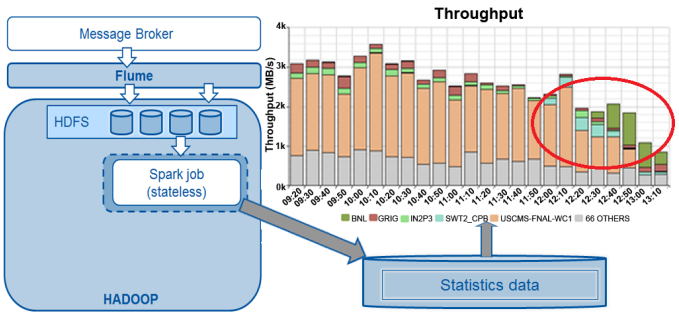

Fig. 21: The Spark batch computations for WLCG monitoring (some statistics are missing as highlighted).

Figure 22 represents both the pure streaming, and the combination of batch and streaming approaches. Both approaches show the computation in real-time as highlighted in the plot. This shows that both of these approaches are capable of providing up-to-date statistics that are beneficial to the users in comparison with the pure batch computation approach. The Spark batch computation performed better than the MapReduce job presented in [3] due to the use of in-memory processing. The intermediate results were cached into memory in comparison with the former approach, which utilises the disk for reading and writing.

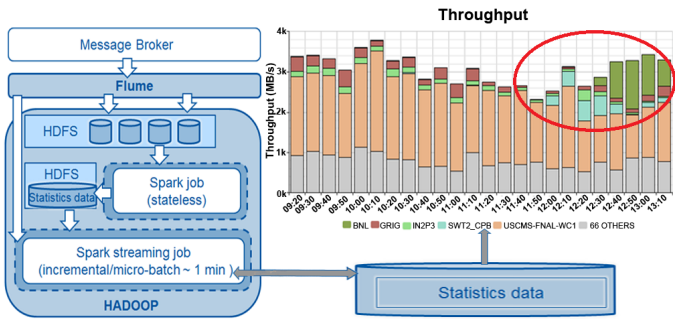

Fig. 22: The Spark batch and streaming computations for WLCG monitoring (statistical data are in near real-time as highlighted).

4.5 Evaluation of scalability, on the Amazon EC2 cloud cluster

In the previous section, the OLA was evaluated on the WLCG infrastructure (shared). The architecture was also evaluated on a public cloud infrastructure to understand how portable the architecture is. The purpose of the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2772241, IEEE Transactions on Parallel and Distributed Systems

evaluation was not to compare the performance of the WLCG and the cloud infrastructure, but solely to understand the flexibility of the OLA model. The metadata from the ATLAS experiment were used for the evaluation of the WLCG infrastructure, whereas metadata from the CMS experiment were used for the evaluation of cloud infrastructure. In this section, various scalability properties were evaluated on the Amazon cloud cluster such as the number of cores, memory size, and the number of executors. All three algorithms discussed in the previous section of this paper were used to evaluate the parallelism. Taking this into account, the performance on the cloud may vary in comparison with the WLCG.

A virtual cluster was created in the Amazon Elastic Cloud [21] using a general purpose instance "m4.2xlarge" that has eight virtual CPUs, 32 GB of memory, and 20 GB of storage per instance. The cluster was configured with four nodes, one name node, and three data nodes. The nodes were distributed with 24 GB of data (seven days log data). For conducting the tests, the job was submitted with various scalability properties. At each execution, a parameter was changed, and the rest remained fixed.

Executor memory

Although there were 32 GB of memory available in each node, it is possible to limit how much memory should be allocated to a job. For this evaluation, the number of executors was fixed at four. Then, the jobs were submitted with varying memory sizes, such as 2 GB, 4 GB, 6 GB, 8 GB, and 20 GB for each executor. Since there were four executors running, the total memory used for each test was 8 GB, 16 GB, 24 GB, 32 GB and 80 GB. In general, the performance was improved as the memory was increased. In particular, the performance from 2 GB to 8 GB in the execution time was improved by 48%. What is evident from the Figure 23 was that the difference in execution time is not substantial when increasing from 8 GB to 20 GB; in fact, it varies by just 10 seconds. This difference can be explained by the fact that when 8 GB per node is used, the total available memory for the jobs is 32 GB; more than enough to accommodate 24 GB of data that is required to be processed. Any additional memory would not have a huge impact on job performance, as it would mostly remain unused.

Executor instances

The evaluation of the executors was created in order to to measure how performance would be impacted when changing the number of executors. For this test, the amount of memory used for each executor was fixed at 4 GB. The number of cores per executors was fixed at two. As seen in Figure 24, the execution time was improved by 76% using four executors, when compared with just one. The execution time was seven seconds slower when five executors were used in comparison to four. This was in part due to there only being four virtual nodes available in the cluster. When there were five executors, one of the nodes would run more than one executor, contributing to an uneven distribution of the job. Ultimately, this would cause an overloaded node.

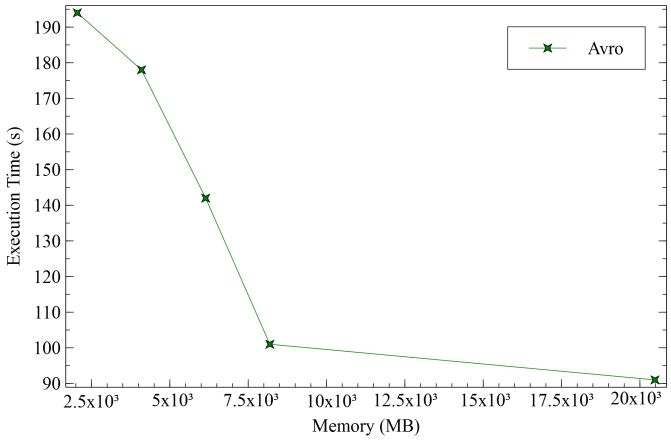

Fig. 23: Execution time versus the memory size on the cloud infrastructure.

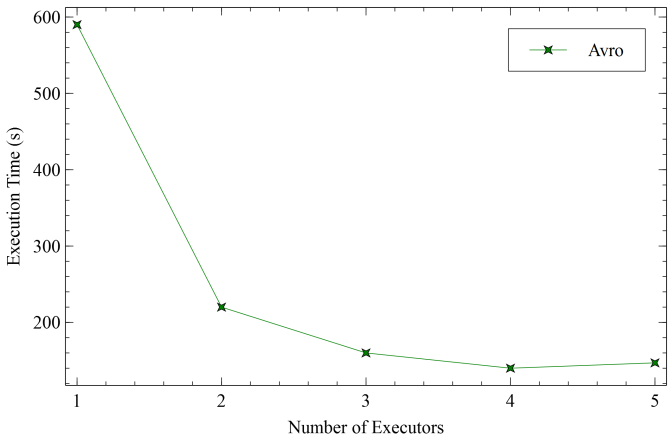

Fig. 24: Execution time versus the number of executors on the cloud infrastructure.

Executor cores

The executor cores parameter defines the number of tasks that each executor can run concurrently. In this test, the number of cores per executor was analysed as shown in Figure 25. The amount of memory used for each executor was fixed at 4 GB, and the number of executors was fixed at four, so all nodes would be utilised. The performance improvement of using eight cores over 2 cores was 69%. No difference was observed between using eight or ten cores. This is due to the fact that the maximum number of virtual CPU cores available in each node is eight.

5 Conclusion

The three data monitoring approaches presented in this paper outperform the RDBMS based system and the Lambda Architecture that is used by the WLCG in terms of execution time, low-latency, maintenance, and scalability. In particular, the streaming approach provides the up-to-date state of the infrastructure. The evaluation also shows that Optimised Lambda Architecture can be ported into other computing infrastructures with ease, as it was demonstrated in the WLCG and a cloud infrastructure. On completion of the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2772241, IEEE Transactions on Parallel and Distributed Systems

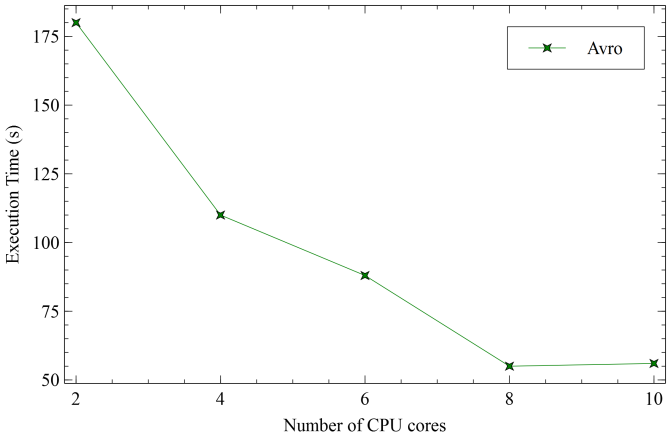

Fig. 25: time versus the number of cores on the cloud infrastructure.

work described in this paper the WLCG group has adopted the Optimised Lambda Architecture, a combined batch and streaming approach, and has been using this approach for monitoring the WLCG Data acTivities Dashboard since October 2015 [23]. Since the deployment of the Optimised Lambda Architecture, the WLCG has been able to monitor infrastructures (e.g. EOS data storage) that it was once assumed would have been impractical. It has also saved operational time as well as computation time in comparison to the traditional architecture formerly used by the WLCG. With the traditional workflow consisting of local filesystems (dirq) and local collectors, PL/SQL computations had several hours of operational time per week dedicated to cleaning up the partition of the machines and maintaining services. With the Optimised Lambda Architecture now implemented the corresponding operational time has reduced to almost zero. An estimated 0.5 days/week is saved through use of the Optimised Lambda Architecture. In terms of computation time, the Optimised Lambda Architecture utilises real-time computation, whereas the traditional architecture required recomputation at a regular intervals. The Optimised Lambda Architecture batch layer reduced computation time by a factor of five when compared with the traditional PL/SQL system, and by a factor of two when compared with the original Lambda architecture.

ACKNOWLEDGMENTS

The work by Uthayanath Suthakar was supported by a Brunel University London College of Engineering, Design and Physical Sciences Thomas Gerald Gray postgraduate research scholarship.

REFERENCES

[1] WLCG, *Available: http://wlcg.web.cern.ch [Online; accessed 20-12-2016].*
[2] E. Karavakis, A distributed analysis and monitoring framework for the compact Muon solenoid experiment and a pedestrian simulation *Doctoral dissertation, Brunel University School of Engineering and Design PhD Theses (2010).*
[3] L. Magnoni, U. Suthakar, C. Cordeiro, M. Georgiou, J. Andreeva, A. Khan and D.R. Smith, Monitoring WLCG with lambda-architecture: a new scalable data store and analytics platform for monitoring at petabyte scale, *Journal of Physics: Conference Series 664 (5) (2015).*
[4] J. Andreeva, B. Gaidioz, J. Herrala, G. Maier, R. Rocha, P. Saiz and I. Sidorova, International Symposium on Grid Computing, *ISGC 2007 pp 131139 ISBN 9780387784168 ISSN 1742-6596.*
[5] J. Hausenblas, N. Bijnens and N. Marz, Lambda Architecture, *Available: http://lambda-architecture.net/.Luettu [Online; accessed 25-05-2016] (2014).*
[6] N. Marz and J. Warren, Big Data: Principles and best practices of scalable realtime data systems, *Manning Publications Co., (2015).*
[7] J. Forgeat, Data processing architectures Lambda and Kappa, *Available: http://www.ericsson.com/research-blog/data-knowledge/data-processing-architectures-lambda-and-kappa [Online; accessed 25-05-2016] (2015).*
[8] S. Uesugi, What is Kappa Architecture?, *Available: http://milinda.pathirage.org/kappa-architecture.com [Online; accessed 25-05-2016] (2015).*
[9] C. Woodbridge, Who's Who in Technology, *Research Publications (2015).*
[10] J.M. Hellerstein, M. Stonebraker and J. Hamilton, Architecture of a database system, *Foundations and Trends in Databases, 1(2), 141-259 (2007).*
[11] B. Ellis, Real-time analytics: Techniques to analyze and visualize streaming data, *John Wiley Sons (2014).*
[12] P. Basanta-Val, N. Fernndez-Garca, A.J. Wellings and N.C. Audsley, Improving the predictability of distributed stream processors, *Future Generation Computer Systems, 52, 22-36 (2015).*
[13] P. Basanta-Val, N.C. Audsley, A.J. Wellings, I. Gray and N. Fernndez-Garca, Architecting time-critical big-data systems, *IEEE Transactions on Big Data, 2(4), 310-324 (2016).*
[14] Apache Spark, *Available: http://spark.apache.org [Online; accessed 20-04-2016].*
[15] T. White, Hadoop: The Definitive Guide, *3rd ed. Yahoo press (2012).*
[16] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker and I. Stoica, Spark: Cluster Computing with Working Sets, *HotCloud 10 (2010).*
[17] M. Zaharia, M. Chowdhury, T. Das and A. Dave, Fast and Interactive Analytics over Hadoop Data with Spark, *Usenix pp 45-51 37, 4 (2012).*
[18] H. Karau, A. Konwinski, P. Wendell, and M. Zaharia, Learning spark: lightning-fast big data analysis *O'Reilly Media, Inc. (2015).*
[19] M. Zaharia et al, Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing, *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation - USENIX Association (2012).*
[20] U. Suthakar, L. Magnoni, D.R. Smith and A. Khan, An efficient strategy for the collection and storage of large volumes of data for computation, *Journal of Big Data (3) (2016) ISSN 2196-1115 DOI 10.1186/s40537-016-0056-1.*
[21] Amazon cloud, *Available: https://aws.amazon.com [Online; accessed 15-10-2016].*
[22] Databricks, *Available: https://databricks.com [Online; accessed 07-12-2016].*
[23] WLCG Data acTivities dashboard, *Available: http://dashb-wdt-xrootd.cern.ch/ui [Online; accessed 27-04-2016].*