INVITED PAPER *Special Section on Management for the Era of Internet of Things and Big Data*

Managing the Synchronization in the Lambda Architecture for Optimized Big Data Analysis

Thomas VANHOVE[†a)], ***Student Member***, **Gregory VAN SEGHBROECK**[†], **Tim WAUTERS**[†], **Bruno VOLCKAERT**[†], ***Nonmembers***, ***and*** **Filip DE TURCK**[†], ***Member***

SUMMARY In a world of continuously expanding amounts of data, retrieving interesting information from enormous data sets becomes more complex every day. Solutions for precomputing views on these big data sets mostly follow either an offline approach, which is slow but can take into account the entire data set, or a streaming approach, which is fast but only relies on the latest data entries. A hybrid solution was introduced through the Lambda architecture concept. It combines both offline and streaming approaches by analyzing data in a fast speed layer first, and in a slower batch layer later. However, this introduces a new synchronization challenge: once the data is analyzed by the batch layer, the corresponding information needs to be removed in the speed layer without introducing redundancy or loss of data. In this paper we propose a new approach to implement the Lambda architecture concept independent of the technologies used for offline and stream computing. A universal solution is provided to manage the complex synchronization introduced by the Lambda architecture and techniques to provide fault tolerance. The proposed solution is evaluated by means of detailed experimental results.

key words: *Lambda architecture, synchronization, big data, Tengu*

1. Introduction

Our digital universe is continuously expanding and predicted to contain 40 ZB (1 Zettabyte = 1 billion Terabyte) of data by the year 2020 [1]. Retrieving valuable information from these data sets through conventional methods becomes nearly impossible if time constraints apply. Moreover, most of these data sets consist of unstructured data, making the processing even more complex. A popular approach in the big data domain is to precompute views with big data processing technologies and let applications or users query this view instead of the entire data set. An important distinction is made in semantics: the entries in the original big data set are referred to as *data*, whereas the entries in the precomputed views are referred to as *information* [2]. Information is thus derived from data through the algorithms implemented in big data processing technologies.

These technologies can be divided in two types: batch processing, and stream processing. The best known batch processing approach is Map-Reduce, originally developed by Google [3], but made popular by its open-source implementation in Apache Hadoop [4]. Other popular solutions include Spark [5] and Flink [6]. The stream processing on the other hand, satisfies the processing needs of applications that generate data streams, such as sensor networks, social media, and network monitoring tools [7]. While batch processing analyzes an entire data set, stream processing does the analysis on a message to message basis. Important streaming analysis frameworks are Storm [8], S4 [9], and Samza [10].

The power of batch processing comes from the ability to access an entire data set during the computation, e.g. creating the opportunity for the detection of relations in the data. The drawback of batch processing is that all resulting information only becomes available after the execution is complete. This process can take hours or even days during which recent data is not taken into account. While stream processing lacks the overview of batch processing, it does allow for a (near) real-time analysis of data as it arrives in the system. The Lambda architecture is built upon a hybrid concept where during a batch analysis execution, in a batch layer, newly arriving messages are analyzed by a stream analysis technology, or speed layer [2]. This effectively harnesses the power of both approaches, giving an application a complete historic informational overview through the batch layer, stored in batch views, and (near) real-time information through its speed layer, stored in speed views. As soon as data is processed in the batch layer, the information is stored in a batch view and the corresponding information is removed from the speed view.

The Lambda architecture is clearly a very powerful concept, but it does pose several implementation challenges. First, as information is stored in two different views, the synchronization between batch and speed layer is key to providing applications and/or users with the correct information. If this is overlooked or ill-handled, information could be lost or redundantly stored for a period of time. Second, storing information across different data stores leaves the system in a state of polyglot persistence, creating the need for the aggregation of information from both the batch and speed views every time a query is sent by the application or users.

This paper proposes a general implementation of the Lambda architecture concept without dependencies on the technologies used in the batch/speed layers or views. A proof of concept has been implemented as part of the Tengu platform, formerly known as Kameleo [11]. The paper focuses on providing a generic solution for the synchronization challenges that arise during the implementation of the concept, but also proposes a solution for the aggregation challenge.

Manuscript received August 20, 2015.
Manuscript revised October 21, 2015.
[†]The authors are with the Department of Information Technology (INTEC), Ghent University - iMinds, Belgium.
a) E-mail: thomas.vanhove@intec.ugent.be
DOI: 10.1587/transcom.2015ITI0001

The remainder of this paper is structured as follows: Section 2 discusses the Lambda architecture in depth. In Sect. 3 the synchronization challenge is discussed in detail and a solution is proposed. Section 4 explains how the system handles failures in the different layers. The implementation of the synchronization solution is detailed in Sect. 5. The experimental setup and results are provided in Sect. 6. In Sect. 7 initial steps are detailed towards a solution for the aggregation of polyglot persistent views. Finally, the conclusions are presented in Sect. 8.

2. Lambda Architecture: Overview and Challenges

The aim of each data system is to answer queries for applications or users on the entire data set. Mathematically, this can be represented as follows [2]:

$$query = function(all\ data)$$

While in the era of Relational Database Management Systems (RDBMS) it was still possible to query the entire data set in real time, this is no longer the case with big data sets [12]. Therefore, in big data analysis systems queries are already partially precomputed and stored in views to limit the applications' query latency. Expressed in terms of functions, this gives us the following:

$$view = function(all\ data)$$
$$query = function(view)$$

It is here that Marz also makes a distinction between data and information [2]. Data is the rawest information from which all other information is derived and is perceived to be true within the system, in this case the main big data set. A big data system thus becomes the function that analyzes data through a programmed algorithm and stores the resulting information in a view. Queries thus no longer access data, but information stored inside views. According to Marz, these big data systems need to achieve several properties:

- **Robustness and fault tolerance:** a data system needs to behave correctly even in the event of software, machine, but also human failures.
- **Low latency reads and updates:** data or information needs to be available when an application or user needs it.
- **Scalability:** a data system needs to maintain a stable performance with increasing or decreasing load.
- **Generalization:** a data system needs to be applicable to a wide range of applications.
- **Extensibility:** the potential to add functionality with a minimal cost.
- **Ad hoc queries:** unanticipated information in a data set needs to be accessible.
- **Minimal maintenance:** limit the implementation complexity of the components.
- **Debuggability:** a data system needs to provide information allowing to trace how output was construed.

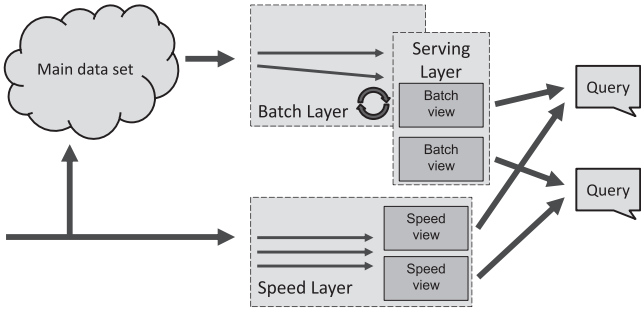

Fig. 1 Conceptual overview of the Lambda architecture.

The Lambda architecture is built in layers each satisfying a subset of these properties.

As stated before, a big data system precomputes views on a big data set to reach reasonable latency query times. This is achieved by the first layer of the Lambda architecture: the batch layer. The results of the batch layer are stored in batch views, managed by the serving layer. Most of the above-stated properties are already fullfilled by these two layers. The final property, concerning the low-latency reads and updates, is accomplished with the final layer: the speed layer[†]. It provides the analysis of data as soon as it enters the system and stores it in a speed view. Queries by applications or users then combine the information that is stored in the batch and speed view. A query on a big data set, analyzed by the Lambda architecture, can thus be described as follows:

$$batch\ view = function(all\ data)$$
$$speed\ view = function(speed\ view, new\ data)$$
$$query = function(batch\ view, speed\ view)$$

Figure 1 gives a conceptual overview of all the above discussed layers of the Lambda architecture.

The batch layer thus continuously recomputes the main big data set, which in time grows, causing the execution time to increase accordingly. This execution time can be limited by using an incremental function to compute the batch view:

$$batch\ view = function(batch\ view, new\ data)$$

However, in order to guarantee the robustness and fault tolerance, a recomputational algorithm needs to always exist.

As soon as data is processed by the batch layer, the derived information that will be stored in a batch view has a duplicate in the speed view. The corresponding information in the speed view thus needs to be removed to make sure no redundant information is present in the system. While this keeps the data store for the speed views relatively small, i.e. it only contains the most recent information of the system, it does expose a critical part of the system. If the synchronization between batch and speed layer is incorrect, the entire system is vulnerable to missing or redundant information.

[†]This layer is called the real time layer by Marz, but in practice it is often more near real time than true real time. To avoid confusion, in this paper it is referred to as the speed layer.

Marz suggests to maintain two sets of speed views and alternately clearing them, which introduces redundancy. This paper proposes a general solution in Sect. 3 without information redundancy.

A second challenge arises with the final function to answer a query:

$$query = function(batch\ view, speed\ view)$$

To answer a query, information from both the batch and speed view is needed. The idea where applications store their information in a mix of data stores to take advantage of the fact that different data stores are suitable for storing different information, is referred to as polyglot persistence [13]. Support for polyglot persistent applications is still a very active research topic [14], [15]. Initial steps towards a general solution for the aggregation challenge in the Tengu platform are disclosed in Sect. 7.

While the Lambda architecture is regarded as a promising concept in both academia [16], [17] and industry [18], [19], some critique is expressed as well [20]. Kreps points out that maintaining two code bases (for batch and speed layer) is a complex and painful issue. While this is true in some form, their proposed alternative, the Kappa architecture, limits the information that can be retrieved from the big data set. This new proposal eliminates the batch layer and only uses the speed layer to analyze the entire data set message by message. However, this way an algorithm can no longer benefit from an overview of the entire data set. For example, suppose an application analyzes the chat messages between social network users for the detection of cyber bullying [21]. In the speed layer a message is analyzed on its own, but in the batch layer a more accurate analysis is possible because the algorithm has the context of the entire chat history. In the next sections a solution for the synchronization challenge in the Lambda architecture is given without compromising on the information stored in the views.

3. Synchronization

The most important aspect of the synchronization between the batch and speed layer happens when the batch layer finishes its computation. A delicate operation follows where the soon to be redundant data needs to be removed from the speed view before it is entered in the batch view. If too much information is removed from the speed view, the system enters a temporary state with missing information. If too little information is removed, the system enters a temporary state where redundant data is processed in the queries. Both states are temporary, because it is fixed after another execution of the batch layer algorithm, although other information might then be missing or redundant.

Nathan Marz proposes a solution where two parallel speed views are used to store the most recent information [2]. As he points out, this leaves the system in a redundant state, but it is considered to be an acceptable price for a general solution. The goal of this paper is to design a general solution without redundancy or information loss. In order to do so, a precise answer is needed to the following question: which information needs to be deleted once a batch layer run has finished? The system thus needs to know which data was processed by the batch layer and what the corresponding information is in the speed view.

The proposed approach is as follows: tagging data as soon as it enters the system allows for this traceability of when the data entered the system, and thus what corresponding information can be removed. As soon as data arrives, it is tagged by a current tag T_n. The data is stored with the big data set, but still marked with the tag T_n. It is also analyzed by the speed layer, which stores the resulting information in a view specifically for all information with the tag T_n, $(speed\ view)_{Tn}$. As soon as the batch layer finishes its current execution, the following happens: the system switches to a new tag $T_{(n+1)}$ for all new incoming data. The information, resulting from the batch layer execution, is pushed into the batch view. The corresponding information in the speed view can be easily cleared with the tag that came before T_n, $(speed\ view)_{T(n-1)}$. Then the new batch data set becomes the union of all data with the T_n tag, $data_{Tn}$, and the previous batch data set:

$$batch\ data = data_{Tn} \cup batch\ data \tag{1}$$

At this point, the batch layer starts a new execution and the entire walkthrough described above is repeated. Similar to the solution proposed by Marz, parallel speed views are used, but now clearly marked with a tag that marks the information that is contained within them as to avoid redundant or missing information. A query now becomes:

$$query = function(batch\ view, (speed\ view)_{T(n)}, ..., (speed\ view)_{T(n-i)}) \tag{2}$$

Figure 2 depicts the lifetime of different events and services in relation to each other in a normal running Lambda architecture implementation. The directional line on top represents the time moving from left to right. The batch layer execution time is portrayed by the dashed line. The dotted tagger line shows which tag is given to a new message that enters the system at a given time. Finally, the lifetime of the speed views is represented by solid lines and the name of the tag it stores. The sequence clearly shows how a speed view exists for two batch runs before being cleared.

Figure 2 also shows two atomic points that will need to be addressed in the implementation:

1. **Batch view update - speed view clearance:** during this operation the system is vulnerable for responding to queries with redundant or missing information. If a query were to enter the system between the update of the batch view and the clearance of the corresponding information in the speed view, the response of the query will contain redundant or missing information, depending on the order of the previously mentioned operations.

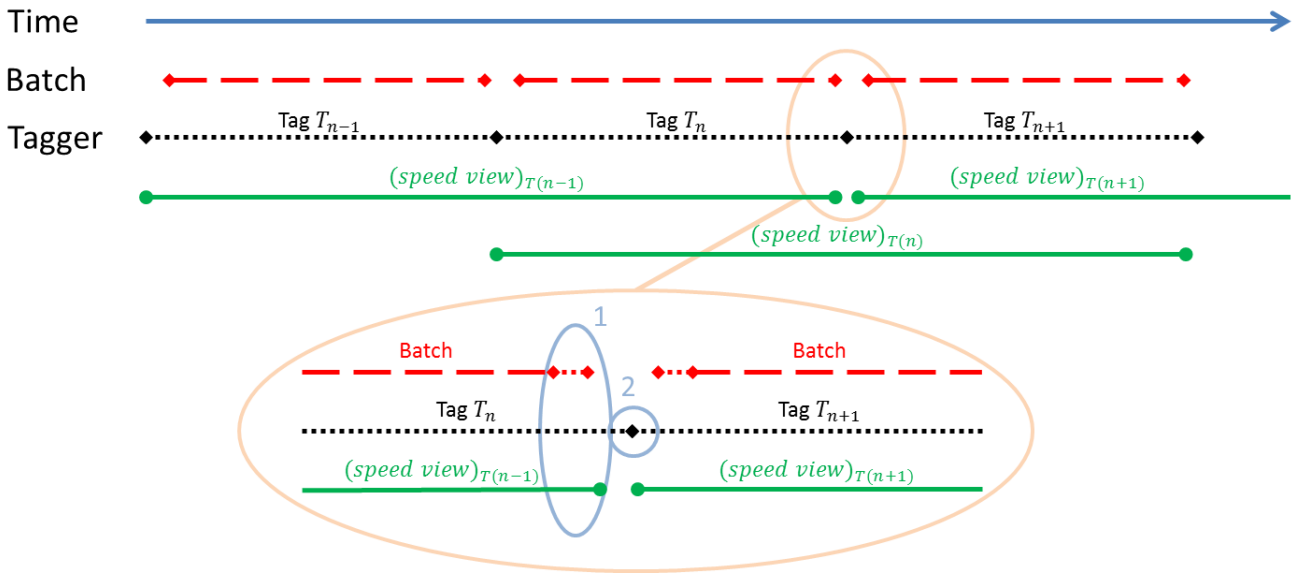

Fig. 2 Synchronization timeline of the different layers. Two important atomic points are identified: 1) batch view update - speed view clearance 2) tag switching.

2. **Tag switch:** a message cannot enter the system while no or multiple tags are active. If a message is not tagged, the system will ignore it and data is lost. If a message is tagged multiple times with different tags, redundant data is introduced into the system.

Important to note is the difference in impact both points have: the tag switch concerns data, while the update/clearance works in the context of information. Recovering a system from faulty information is possible through a complete recomputation of the data set. However, recovering from faulty data is a whole lot more complex since all derived information is false as well.

Note that in this section no assumptions have been made as to which technologies are used to implement the proposed tagging solution. Tagging can be implemented in different ways: a tag can be directly inserted into a message or it can be indirectly associated with the message. The proof of concept of the tagging solution for the synchronization challenge uses the indirect approach and is presented in Sect. 5.

4. Failure Handling

An important property of a big data system is its robustness and fault tolerance as outlined in Sect. 2 above. In the following subsections failure scenarios of the different parts of the platform are discussed and how they can be handled.

4.1 Batch Layer Failures

If the execution of the batch layer fails, there are several possibilities to handle the failure. First, a simple restart of the execution can be done with the same data set as before.

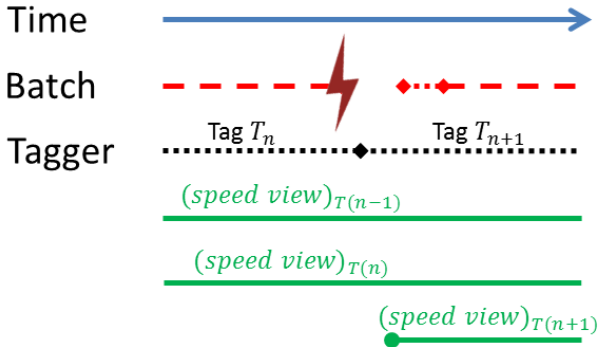

Fig. 3 Batch layer failure handling.

The batch and speed view still contain the correct information for applications and users, and the current tag needn't change. A repeatedly failing algorithm does require human intervention as the cause might be a faulty implementation.

A second possibility is to handle the failure similar to a correct end of the batch layer: a new tag is used to tag future incoming messages, but the previous tags are not wiped from the speed view as they were not yet analyzed by the batch layer. Otherwise this would cause temporary information loss. Data tagged with the previous tags is added to the data set that will be analyzed by the batch layer. In other words, while the batch layer needs to restart, the data set is expanded to take into account more recent data. This method is limited in the number of failures it can handle due to the increasing number of concurrent tags and the possibility of an overflow of the tag value. As with the previous method, the information in the batch and speed views remains available for applications and users. Figure 3 depicts this method of failure handling for the batch layer. The proof of concept, detailed in Sect. 5, handles a batch layer failure with a simple restart.

4.2 Speed Layer Failures

A failure of the speed layer has less impact on the entire data system compared to a batch layer failure because the information displayed in the speed view is only a fraction of the total data set. That being said, the goal is to eliminate redundant and missing information completely.

Failure handling is mostly dependent on how a streaming big data analysis platform handles the failures. If the analysis of one message fails, it is important the chosen technology has guaranteed message processing or checkpointing, i.e. each data message is fully processed without fault. If an entire machine or cluster fails, data in transit should be recovered or re-analyzed. For example, in an implementation with Kafka and Storm, Storm provides guaranteed message processing, but it also needs to keep an offset of messages it already consumed from Kafka. Both technologies combined can therefore recover from a variety of failures.

4.3 View Failures

A view failure results in partial information not being available for applications and users. A failure of the speed view has a limited impact as it only contains the most recent information of the system, while a failure of the batch view would cause most of the historical information to be unavailable. Therefore, it is important to use distributed and replicated data stores for the views of both layers. In the NoSQL (Not Only SQL) domain most data stores are of a distributed nature and support some form of replication. The amount of replicas depends on the critical nature of the application. A careful consideration is required in this trade-off between storage cost and availability.

While a view failure can cause a temporary unavailability or redundancy of information, the layered approach of the Lambda architecture allows the system to recover without human intervention. A recomputational algorithm in the batch layer always starts with the original main data set, meaning errors in a batch or speed view are overruled in the next iteration. This property is shown extensively in the results in Sect. 6.

4.4 Data and Communication Failures

Query latency In Sect. 3 it was mentioned that the operation updating the batch view and deleting the corresponding information in the speed views needs to be atomic. During this time a read lock needs to be enforced on the different views as to insure no missing or redundant information is used to answer the query. If an error occurs during one of the steps in the operation, a rollback can make sure the views are not corrupted.

Equation (2) also defines a query in the Lambda architecture as a function that aggregates data from different views. Both the read lock and the aggregation will cause a certain query latency.

Tagging The impact of missing or redundant data compared to information was already briefly discussed in Sect. 3. An error in the tagging or switch between tags could cause this missing or redundant data. Recovery from such a failure entails much more than an information failure and the system will be unable to recover from this without manual intervention.

Data persistence Finally, data persistence is an important feature to make sure no data or information is lost. For example, assume a message is the last message to be tagged with tag T_n. All the $data_{Tn}$ needs to be merged with the previous batch data set, as defined in Eq. (1). There needs to be a guarantee that all data with tag T_n is present in $data_{Tn}$, i.e. even the last message to be tagged with T_n needs to be present and not get lost in the network. This is closely related to guaranteed message processing discussed in Sect. 4.2.

4.5 Human Failure

A final important failure is the realistic possibility that a human error will occur in the system. Here the importance of the main data set is again featured. The main data set contains unaltered data and is expected to be true, within the Lambda architecture system. This assumption allows the system to recover from any human error in the different layers. For example, if a faulty implementation in any layer causes faulty information to be stored in the views, a fix of the faulty code allows the entire system to recover after a couple of iterations. This emphasizes the need for a re-computational algorithm in the batch layer. While an incremental batch algorithm can be used to limit the execution time of the batch layer, a re-computational algorithm needs to exist to recover from human-introduced errors, such as faulty implementations.

5. Implementation Details

The proposed Lambda architecture implementation is implemented as part of the Tengu platform, previously known as Kameleo [11]. The Tengu platform was originally developed for the automated setup of big data technologies on experimental testbeds. Figure 4 shows an overview of all used technologies in the proof of concept implementation and how they are chained together.

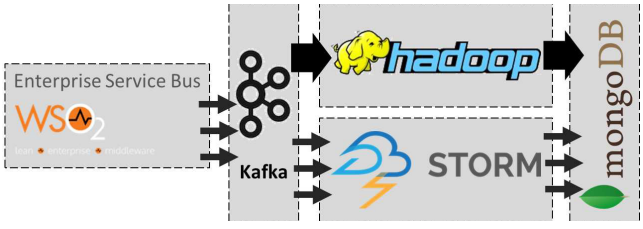

Fig. 4 Technology overview of the implemented Lambda architecture proof of concept.

The first technology a message encounters when it enters the system is the WSO2 Enterprise Service Bus (ESB). It allows for advanced communication between services by routing messages in a bus architecture using a vast array of protocols. For this reason the ESB was favored over a Message Broker (MB) or a Complex Event Processor (CEP) as those would limit the amount of control the system had over the messages and services. The WSO2 ESB was chosen over other candidates, such as UltraESB, Mule, and Talend, for its performance and maturity [22]–[24]. It is the intelligent controller-like component that coordinates the execution of the different services, i.e. the batch and speed layer, and their views. The ESB also maintains the current active tag corresponding to an active topic in Apache Kafka [25].

After retrieving the tag in the ESB, the message is sent to a Kafka topic corresponding to the received tag. The tag is hence never attached to the incoming message, but indirectly associated with the message through a topic in Kafka. From this topic the message is ingested by a speed technology, analyzed and stored in a speed view. In the proof of concept Storm [8] is used as a speed technology, while the speed views are stored in MongoDB [26]. Storm contains a topology that is responsible for a specific tag, i.e. a Kafka topic. This topology analyzes the messages and stores them in the MongoDB collection related to the tag. The union defined in Eq. (1) is performed using all data in the Kafka topic as $data_{Tn}$. The batch layer, implemented with Hadoop [4] in this proof of concept, performs an analysis and stores the information in a batch view, a specific collection in MongoDB.

Important to note is that the implementation of the tagging system is done by the WSO2 ESB and Kafka. While Hadoop, Storm and MongoDB are used in this proof of concept, they are merely services of the ESB through which the messages are analyzed and stored. As a consequence they can be replaced by similar technologies such as Spark, Samza and Cassandra. Additionally, many technologies can already act as a consumer of Kafka messages, but if not, an extension of the WSO2 ESB can still provide the necessary communication.

In Fig. 2 two critical points were also identified concerning the update of the batch view and simultaneous removal of the corresponding information in the speed view, and the switch between active tags. Both operations are required to be atomic to prevent data/information loss or redundancy.

The tag is stored local to and managed by the ESB, making every operation transactional. For each message the ESB reads the value of the tag and sends the message to the corresponding topic. If a call is made to change the tag, the value is updated with an atomic operation. A message can therefore never continue without a tag or with multiple tags.

The switch between views after a completed batch layer iteration is handled by inserting a read lock on the views. This can cause somewhat of a query latency if a query is on hold during the switch. A solution for this latency can be to cache the information during the transition, but this is outside the scope of this paper and considered part of future work.

6. Evaluation Results

The Tengu platform is deployed on the iLab.t Virtual Wall infrastructure [27]. These experimental testbeds consist of over 300 nodes spanning different generations of hardware setups. For the tests in this paper generation 3 nodes were used: 2x Hexacore Intel *E*5645 (2.4 GHz) CPU, 24 GB RAM, 1x250 GB harddisk, 1-5 gigabit nics. Eight nodes were used in the following setup interconnected with a 1 Gigabit connection:

- 2 hadoop nodes
- 2 storm nodes
- 1 WSO2 Enterprise Service Bus node
- 1 MongoDB node
- 1 Zookeeper node
- 1 Kafka node

In the following subsection the results are detailed to show the correctness and regenerative capabilities of the Lambda architecture implementation, especially in the context of information redundancy and information loss. Next, insight is given as to where information is stored among the different views in a normal run of the system.

6.1 View Failure

The most important part of the synchronization challenge consists of eliminating redundant information and information loss. The first results in Fig. 5 show the normal progress of data sizes in the Lambda architecture. For each tag 20 messages were injected into the system through a REST API, one every second, where each message had a specific value. The WSO2 ESB supports a variety of message formats but for this test JSON messages were used:

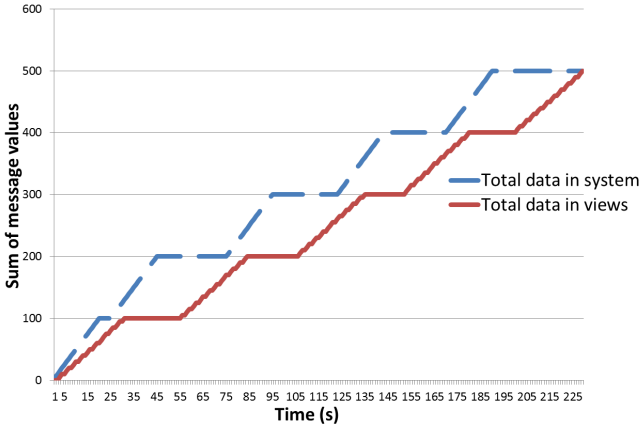

Fig. 5 Normal progress of the active Lambda architecture implementation.

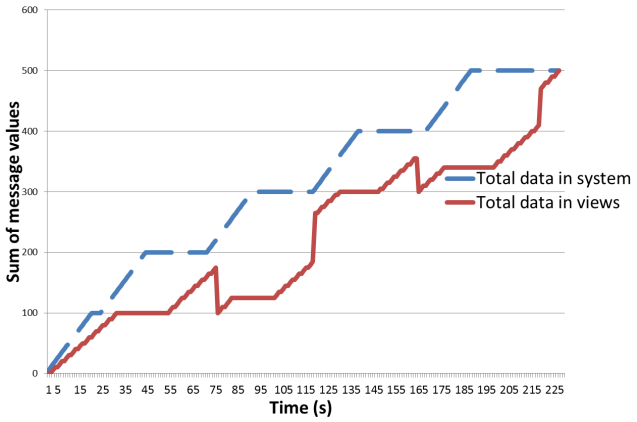

Fig. 6 Regenerative progress of the active Lambda architecture implementation with data loss in views.

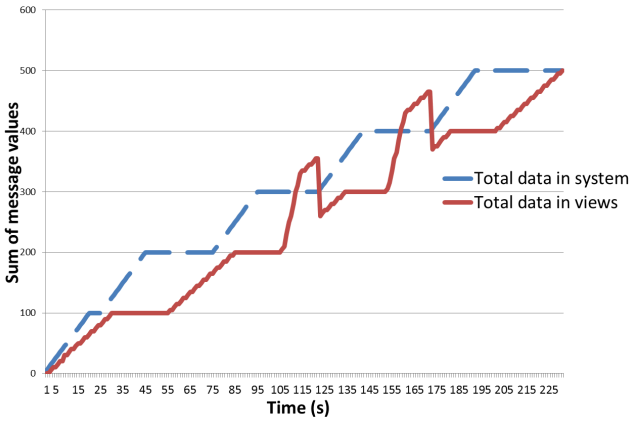

Fig. 7 Regenerative progress of the active Lambda architecture implementation with data redundancy in views.

```
{
  'value': '5'
}
```

The algorithm in the batch and speed layer were tasked with calculating the total sum of the message values. The dashed line shows the sum of all message values injected in the system at any given point. The solid line shows the aggregated sum that is available in all the views, both batch and speed. The sum calculation in the speed layer is slowed as to clearly differentiate the two graph-lines from each other. As can be seen in Fig. 5 the solid line can never drop down, as this would indicate information loss, or be higher than the dashed line, as this would indicate information redundancy.

Information loss in the views is introduced in the second graph, depicted in Fig. 6. Loss is introduced twice in the speed view at around 65 and 165 seconds. The regenerative property of the Lambda architecture is shown at around 115 seconds and 215 seconds. This is when the batch layer has recomputed the main data set and the lost information is restored in the batch view.

Figure 7 shows the regenerative measures of the implementation after redundancy is introduced to the speed views. The solid line clearly surpasses the dashed line in the graph, indicating the presence of information redundancy. The redundancy is however not present in the main data set, meaning that after a batch iteration the redundant information is deleted from the views, again displaying the correct total sum.

Both graphs clearly show the regenerative capabilities of the implemented Lambda architecture in situations with varying information inconsistencies. The time in which the system returns to a consistent state depends on the execution time of the batch layer. In Sect. 3, Fig. 2 illustrates that speed views exist for two batch layer runs before being cleared, meaning that in a worst case scenario an inconsistent state is maintained during two batch layer runs before being resolved. The batch layer execution time can be shortened through use of an incremental algorithm, but as mentioned in Sect. 2 a re-computational algorithm is still required to achieve fault tolerance and robustness. An inconsistent state in the batch view can be resolved after one batch run, but only with a re-computational algorithm.

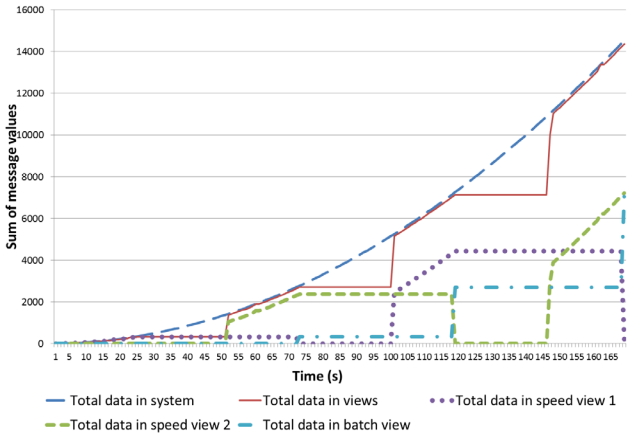

Fig. 8 The total data in the Lambda architecture in time with respect to the different views.

6.2 Information Transition from Speed to Batch Views

As information is moved between different views a lot in the Lambda architecture, the graph displayed in Fig. 8 shares some insight as to where information is stored during a normal run of the Lambda architecture implementation. Important to note is that messages are now continuously sent to the system and have ever increasing values, hence the exponential curve of the total data sum. The speed layer is also no longer slowed down in these tests. First, speed view 1, marked by the dotted line, is filled with information until it reaches a plateau at around 25 seconds. This plateau occurs as the Storm topology is swapped for a new topology to start processing the new tag, i.e. ingest the new topic from Kafka. Once the new topology is active at around 50 seconds, it quickly catches up to the total expected sum by

filling up speed view 2, indicated by the small dashed line, until it reaches the next plateau. Again the Storm topologies are switched, but speed view 1 is also cleared as the information is now contained within the batch view, marked by the dashed-dotted line. Now speed view 1 can again be used to store information and the entire above described process repeats itself. A maximum of two concurrent tags are thus active at any given time.

Based on the graph in Fig. 8 some improvements can be made: the plateau could be reduced by having two parallel Storm topologies, as with the speed views. This has the additional benefit that the old topology can continue generating information next to the new one. The single topology setup of this proof of concept can cause additional delay because the system waits for the topology to be entirely finished before swapping. For a simple task, like calculating a sum, Storm is fast enough and no additional delay is caused, but with more complex algorithms the time for data to be processed by the topology increases, heightening the possibility of additional delay. In a production environment it is therefore highly recommended to work with two parallel Storm topologies.

7. Aggregation

In Sect. 2 a query in the Lambda architecture is defined as a function over the different views. An application that stores data or information in a mix of data stores to take advantage of the fact that different data stores are suitable for storing different data is referred to as a polyglot persistent application [13]. While the work of Sadalage and Fowler focuses on dividing the data set based on data type and/or model, the polyglot persistence in the Lambda architecture splits information based on time, derived from the tag the data got when it entered the system. Both Eq. (2) and Fig. 1 show the need for aggregation, as an answer to a query consists of multiple queries to different data stores. The nature of the aggregation depends on the nature of the information stored in the views and the nature of the query. For example, two integers can be added in a sum, but could equally well be concatenated.

Data abstraction layers, such as Hibernate OGM [28], Kundera [29], and DataNucleus [30], help applications with polyglot persistence by providing general access to their data stores, usually through a unified querying language. Although these data abstraction layers shield applications from underlying data storage technologies, they lack the ability to intelligently combine information from several data stores and return it. The application is thus still responsible for combining information from the different views and not effectively protected from data model changes.

If this responsibility is to be moved away from the application, it needs to be re-introduced in a new layer between the application and the data stores. As mentioned before, the nature of the aggregation is specific to the query the application sends, so user input is required. However, users often also lack the insight into the different technologies to correctly write the code for information retrieval. A definition of the aggregation through a technology independent data flow could prove to be a solution in this case.

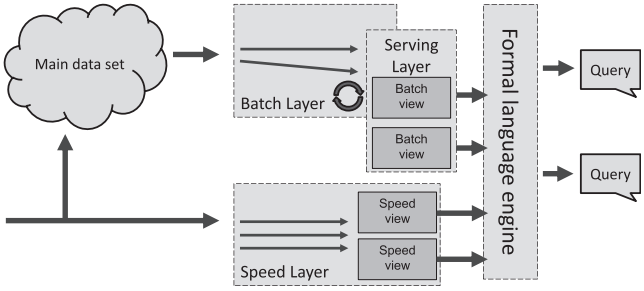

Fig. 9 Lambda architecture with a formal language for the aggregation of information.

A proposed approach is to define this data flow through a formal language. The formal language would allow users, lacking any programming skills or technology specific knowledge, to define an algorithm answering their query through a flow of operations and other queries on the different underlying data stores. Once an aggregation is created through the formal language, an engine can translate it into code and technology-specific queries for different data stores. Figure 9 shows how the formal language fits in with the Lambda architecture. Initial steps towards a definition and implementation of this formal language are ongoing and will be reported on in future work.

8. Conclusion and Future Work

The Lambda architecture is a powerful concept for big data systems. However, it does pose several implementation challenges. This paper proposes a general implementation of the concept, independent of the technologies used for different layers and views. It focuses on a solution for the synchronization challenge between the batch and speed layer through a tagging system. A solution is proposed, tagging messages when they enter the implemented Lambda architecture system, and a proof of concept is implemented in the Tengu platform. Results show that the proof of concept works correctly in regard to eliminating information loss and redundancy, and that when manually introduced, it is able to recover automatically. The information transition between batch and speed view also indicated a delay where no new information was posted in the views during the transition of topologies. A solution is suggested where two parallel topologies exist in the Storm cluster.

Another challenge was identified as the aggregation of information from batch and speed views to answer queries from applications or users. This paper discusses the initial steps that have already been taken towards a general solution in the Tengu platform. The implementation itself will be reported on in future publications.

Acknowledgement

This work was partly carried out with the support of the AMiCA (Automatic Monitoring for Cyberspace Applications) project, funded by IWT (Institute for the Promotion of Innovation through Science and Technology in Flanders) (120007).

References

[1] J. Gantz and D. Reinsel, "The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east," IDC iView: IDC Analyze the Future, vol.2007, pp.1–16, 2012.
[2] N. Marz and J. Warren, Big Data: Principles and best practices of scalable realtime data systems, Manning Publications, 2015.
[3] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," Commun. ACM, vol.51, no.1, pp.107–113, Jan. 2008.
[4] T. White, Hadoop: The definitive guide, O'Reilly Media, 2012.
[5] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," Proc. 2nd USENIX Conference on Hot Topics in Cloud Computing, HotCloud'10, Berkeley, CA, USA, p.10, USENIX Association, 2010.
[6] "Apache flink," http://flink.apache.org/ (Last Visited Aug. 13, 2015).
[7] J. Gama, Knowledge Discovery from Data Streams, Chapman & Hall/CRC Data Mining and Knowledge Discovery Series, Chapman and Hall/CRC, 2010.
[8] "Apache storm," https://storm.apache.org/ (Last Visited Aug. 13, 2015).
[9] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," 2010 IEEE International Conference on Data Mining Workshops, pp.170–177, 2010.
[10] "Apache samza," https://samza.apache.org/ (Last Visited Aug. 13, 2015).
[11] T. Vanhove, J. Vandensteen, G. Van Seghbroeck, T. Wauters, and F. De Turck, "Kameleo: Design of a new platform-as-a-service for flexible data management," Proc. 2014 IEEE Network Operations and Management Symposium (NOMS), pp.1–4, 2014.
[12] A. Jacobs, "The pathologies of big data," Commun. ACM, vol.52, no.8, pp.36–44, Aug. 2009.
[13] P.J. Sadalage and M. Fowler, NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence, Addison-Wesley, 2012.
[14] A. Maccioni, O. Cassano, Y. Luo, J. Castrejón, and G. Vargas-Solar, "NoXperanto: Crowdsourced polyglot persistence," Polibits, vol.50, pp.43–48, 2014.
[15] S. Prasad and S.B. Avinash, "Application of polyglot persistence to enhance performance of the energy data management systems," 2014 International Conference on Advances in Electronics Computers and Communications, pp.1–6, 2014.
[16] W. Fan and A. Bifet, "Mining big data: Current status, and forecast to the future," ACM SIGKDD Explorations Newsletter, vol.14, no.2, pp.1–5, 2013.
[17] S. Perera and S. Suhothayan, "Solution patterns for realtime streaming analytics," Proc. 9th ACM International Conference on Distributed Event-Based Systems, DEBS'15, pp.247–255, 2015.
[18] HPCC Systems, "Lambda architecture and HPCC systems," White Paper, Feb. 2014.
[19] "MapR," https://goo.gl/SBdQEW (Last Visited Aug. 13, 2015).
[20] J. Kreps, "Questioning the lambda architecture," Online article, July 2014. http://radar.oreilly.com/2014/07/questioning-the-lambda-architecture.html (Last Visited Aug. 7, 2015).
[21] T. Vanhove, P. Leroux, T. Wauters, and F. De Turck, "Towards the design of a platform for abuse detection in osns using multimedial data analysis," 2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013), pp.1195–1198, 2013.
[22] D. Abeyruwan, "ESB performance round 6.5," Tech. Rep., WSO2, Jan. 2013. http://wso2.com/library/articles/2013/01/esb-performance-65/
[23] A.C. Perera and R. Linton, "ESB performance round 7," Tech. Rep., AdroitLogic, Oct. 2013. http://esbperformance.org/display/comparison/ESB+Performance
[24] S. Anfar, "ESB performance round 7.5," Tech. Rep., WSO2, Feb. 2014. http://wso2.com/library/articles/2014/02/esb-performance-round-7.5/
[25] N. Garg, Apache Kafka, Packt Publishing, 2013.
[26] K. Chodorow, MongoDB: the definitive guide, O'Reilly Media, 2013.
[27] "iLab.t virtual wall," http://ilabt.iminds.be/ (Last Visited Aug. 13, 2015).
[28] "Hibernate OGM," http://hibernate.org/ogm/ (Last Visited Aug. 13, 2015).
[29] "Impetus Kundera," https://github.com/impetus-opensource/Kundera (Last Visited Aug. 13, 2015).
[30] "DataNucleus," http://www.datanucleus.org/ (Last Visited Aug. 13, 2015).

Thomas Vanhove obtained his masters degree in Computer Science from Ghent University, Belgium in July 2012. In August 2012, he started his PhD at the IBCN (Intec Broadband Communication Networks) research group, researching data management solutions in cloud environments. More specifically, he has been looking into dynamic big data stores and polyglot persistence. It was during that time he created the Tengu platform for the simplified setup of big data analysis and storage technologies on experimental testbeds.

Gregory Van Seghbroeck graduated at Ghent University in 2005. After a brief stop as an IT consultant, he joined the Department of Information Technology (INTEC) at Ghent University. On the 1st of January, 2007, he received a PhD grant from IWT, Institute for the Support of Innovation through Science and Technology, to work on theoretical aspects of advanced validation mechanism for distributed interaction protocols and service choreographies. In 2011 he received his Ph.D. in Computer Science Engineering.

Tim Wauters received his M.Sc. degree in electro-technical engineering in June 2001 from Ghent University, Belgium. In January 2007, he obtained the Ph.D. degree in electro-technical engineering at the same university. Since September 2001, he has been working in the Department of Information Technology (INTEC) at Ghent University, and is now active as a post-doctoral fellow of the F.W.O.-V. His main research interests focus on network and service architectures and management solutions for scalable multimedia delivery services. His work has been published in about 50 scientific publications in international journals and in the proceedings of international conferences.

Bruno Volckaert is a postdoctoral assistant in the INTEC Broadband Communication Networks group, which is a part of the Department of Information Technology at Ghent University. He obtained his Master of Computer Science degree in 2001 from Ghent University, after which he started work on his PhD. While doing research on data intensive scheduling and service management for Grid computing, he co-developed, together with dr. Pieter Thysebaert, NSGrid, an advanced ns-2 based Grid simulator, detailed in full in his PhD: “Architectures and Algorithms for Network and Service Aware Grid Resource Management”.

Filip De Turck leads the network and service management research group at the Department of Information Technology of the Ghent University, Belgium and iMinds (Interdisciplinary Research Institute in Flanders). He (co-) authored over 450 peer reviewed papers and his research interests include telecommunication network and service management, efficient big data processing and design of large-scale virtualized network systems. In this research area, he is involved in several research projects with industry and academia, serves as vice-chair of the IEEE Technical Committee on Network Operations and Management (CNOM), chair of the Future Internet Cluster of the European Commission, and is on the TPC of many network and service management conferences and workshops and serves in the editorial board of several network and service management journals.