Stream Processing on Demand for Lambda Architectures

Johannes Kroß[1(✉)], Andreas Brunnert[1], Christian Prehofer[1], Thomas A. Runkler[2], and Helmut Krcmar[3]

[1] fortiss GmbH, Guerickestr. 25, 80805 Munich, Germany
{kross,brunnert,prehofer}@fortiss.org
[2] Siemens AG, Corporate Technology, Otto-Hahn-Ring 6, 81739 Munich, Germany
thomas.runkler@siemens.com
[3] Technische Universität München, Boltzmannstr. 3, 85748 Garching, Germany
krcmar@in.tum.de

Abstract. Growing amounts of data and the demand to process them within time constraints have led to the development of big data systems. A generic principle to design such systems that allows for low latency results is called the lambda architecture. It defines that data is analyzed twice by combining batch and stream processing techniques in order to provide a real time view. This redundant processing of data makes this architecture very expensive. In cases where process results are not continuously required to be low latency or time constraints lie within several minutes, a clear decision whether both processing layers are inevitable is not possible yet. Therefore, we propose stream processing on demand within the lambda architecture in order to efficiently use resources and reduce hardware investments. We use performance models as an analytical decision-making solution to predict response times of batch processes and to decide when to additionally deploy stream processes. By the example of a smart energy use case we implement and evaluate the accuracy of our proposed solution.

Keywords: Lambda architecture · Big data · Performance · Model · Evaluation

1 Introduction

With the increasing ubiquity of information and communication technology (ICT) and the emergence of the Internet of things (IoT) the available data amount is growing exponentially. Simultaneously, technologies have been developed to store, manage and analyze these diverse and high volumes of data, also known as *big data* [30]. These circumstances allow for applying analytics in order to gain knowledge and support decision-making. For more and more usage scenarios, these analytical capabilities must also meet specific time requirements such as real-time [17]. One common approach to design big data systems that can cover many use cases is the lambda architecture [26]. It mainly consists of a

M. Beltrán et al. (Eds.): EPEW 2015, LNCS 9272, pp. 243–257, 2015.
DOI: 10.1007/978-3-319-23267-6_16

batch layer and a speed layer. The former iteratively processes a set of historical data in batches while the latter processes the arriving data stream in parallel to incrementally analyze latest data. By joining the output of both layers query results always reflect current data.

Nowadays, various complementary technologies with different characteristics exist to build a big data system and there is hardly one technology solution that fits most use cases of an organization. Although the lambda architecture simply is a generic design framework which offers a solution for many use cases, nonetheless, a variety of technologies can be applied for the batch or speed layer. Examples for the batch layer are Hadoop MapReduce [5], Apache Pig [7], and Apache Spark [9] and for the speed layer Apache Storm [10], Apache Spark Streaming [9], Apache Samza [8], or Amazon Kinesis [2]. This multitude leads to the development of complex system of systems, which often results in performance issues and high resource requirements [14]. Furthermore, the lambda architecture intends to process all data twice in both layers. Batch processes also analyze data from the ground up in each iteration to ensure fault tolerance in case of hardware failures or human mistakes [26]. These fundamental ideas require costly resources. For use cases where time constraints are not continuously needed or lie between several minutes, it can be often an important question whether a speed layer is really required or not. However, this question can usually not be answered during system development nor in test systems under realistic workload. As stream processing heavily utilizes main memory, the speed layer can also become an expensive investment [24].

Therefore, we propose a speed layer or stream processing, respectively, on demand. The idea is to exclusively use batch processes as often as possible and switch on stream processing only when batch processes are likely to exceed response time constraints. In this way, computing power is utilized more efficiently and resources can be saved as well as be available for other processes. In case of virtualized environments, investments can be directly decreased by reducing cloud service resources. In order to switch on stream processing at the right time, it is inevitable to predict the response time of succeeding batch iterations. For this purpose, we use performance models. They allow to describe performance influencing factors of software systems and to predict performance metrics such as response time, throughput and utilization by means of analytical solvers or simulation engines [13]. Therefore, we integrate estimated resource demands into the model based on measurements from batch processes to simulate an accurate system behavior. This enables us to efficiently schedule stream processes.

In this paper, we first give a detailed description of our proposed approach in Section 2 and how we use performance models to support decision-making. In Section 3, we validate our approach in an experiment. We describe the selected use case, the setup and sample algorithm for the batch layer, and the prototype performance model to predict batch processes. Afterwards, we discuss the experimental results we derived for different workload scenarios. In Section 4, we reflect

related work in the area of the lambda architecture and, finally, conclude our paper with providing an outlook for future work in Section 5.

2 Stream Processing On Demand

In order to make decisions about when to switch on stream processing, we use performance models as an analytical solution. As illustrated in Figure 1, the iterative process is divided into two main steps in which the following Sections 2.1 and 2.2 are structured. First, one batch iteration and, potentially, a concurrent stream process are started within the lambda architecture. Second, after the batch process has ended, a decision-making model is used to decide whether stream processing is required in the next batch process iteration or not. Basis of decision-making is a performance model which is used to predict the response time of a batch process. Afterwards, the procedure is repeated.

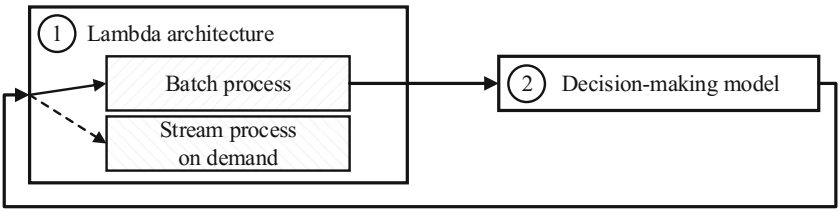


Fig. 1. Stream Processing On Demand Process

2.1 Data Processing in the Lambda Architecture

As already mentioned, our focus is on data processing, namely batch and stream processing, within lambda architecture and not storing data sets or results. Figure 2 illustrates the data flow and structure of batch and speed layer that differ from each other. Starting point is a shared data source which either streams the same data into each processing layer or gets accessed by each layer to retrieve data. Within the batch layer, all data are stored in a data set. A special characteristic of the data set is that it is append-only and data are not updated or removed [26]. Batch processes use the data set to operate on. In doing so, each batch process usually analyzes a huge set of historical data which leads to response times of minutes or hours for one batch job. The results are written to separate views, which is also considered as serving layer by Marz and Warren [26] for batch results. Batch processes constantly run iteratively and start from the beginning once a batch job has finished. If a batch process starts, only data that have been created before are included. Consequently, data that arrive during the current batch process are only included in the next new batch process.

Since all data are analyzed in each cycle, each new result view can replace its predecessor. As the batch layer does not rely on incremental processing, it has the advantage of being a robust system where everything can be recomputed and reconstructed in case of hardware or software failures or human mistakes [26].

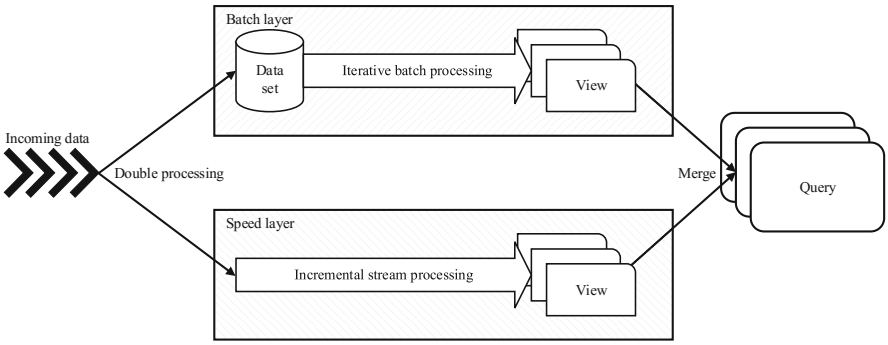


Fig. 2. Composition and data flow of batch and speed layer of the lambda architecture (adapted from Marz and Warren [26])

In contrast to the batch layer, the speed layer does not keep a record of historical data and solely uses main memory. As of today, stream processes run permanently and analyze each incoming message. They incrementally calculate and immediately update their result views. Thus, both layers include separated views and, in practice, usually different technologies are used as underlying databases because of their distinct requirements regarding read and write operations. In order to receive a holistic result, the view of both layers have to be merged in a query.

Although both layers process the same data, the results of queries that merge views only reflect data that are processed once at the time of the query. The purpose of the speed layer is to analyze the data prior to the batch layer and enable low latency by incremental updated result views. As a result, a past view of the speed layer can be discarded as soon as a subsequent batch job has finished.

A typical implementation of the lambda architecture as illustrated in Figure 2 would be to use Apache Kafka [6] - a publish-subscribe messaging system - as shared source for incoming data. For the batch layer, HDFS can be used as data set and Hadoop MapReduce for batch processing. For storing batch results, which Marz and Warren [26] also describe as serving layer, ElephantDB[1] represents a specialized database for this purpose. For the speed layer Apache Storm [10] is an example of an appropriate technology and Apache Cassandra [4] of a database.

[1] https://github.com/nathanmarz/elephantdb

2.2 Decision-Making Model

To decide when to switch on stream processing, we predict the response time of succeeding batch processes and build a decision-making model. To comprehend why it is necessary to predict the succeeding batch processes, the chronological sequence of batch and stream processes as intended by the lambda architecture is illustrated in Figure 3. As already mentioned, results of batch processes are not available until they finish, while results of stream processes are incremental and can be queried at any time. Supposing one *batch process i* has ended and a decision must be made at time y on whether additional stream processes are needed afterwards or not, the earliest point in time where results of stream processes can be reasonably used is at time z. *Stream process j* considers only data newer than time y. Therefore, a batch process is required that has analyzed data before time y. However, the corresponding *batch process j* will only start after time y and end at a given time z. Thus, a decision must already be made at time y, if *batch process k* violates time-constraints so stream processes are switched on at time y. Consequently, query results after time z will have consistently incorporated all data.

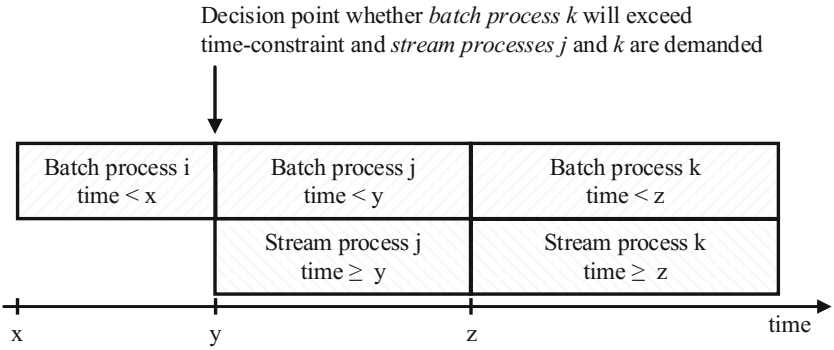


Fig. 3. Chronological sequence of batch and stream processes

The above mentioned response time prediction is part of our decision-making model. Its procedure is depicted in Figure 4. Starting point is a finished batch process iteration. The response time of the second next batch iteration is predicted by using a performance model, which takes two inputs - the time constraint for the duration of a batch process and the load intensity. The latter means information about the incoming data of the batch layer. For instance, this can be in the form of a variable distribution as modeled by the LIMBO tool [22]. The prediction can be accomplished by means of simulation or analytical solving. If the predicted response time does not lie within the specified

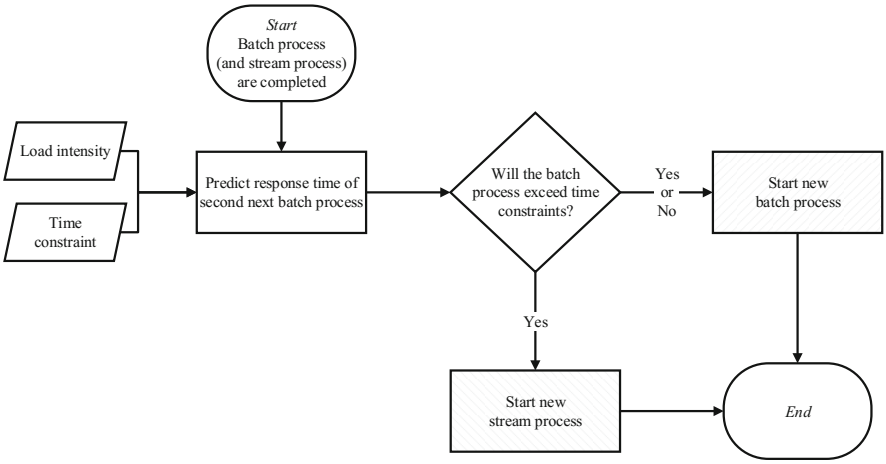


Fig. 4. Decision-making model

time limitation, the model tries to start batch processing in parallel with stream processing, otherwise the model considers batch processing only as sufficient.

3 Experimental Validation

For the evaluation of our proposed approach, we conduct a controlled experiment which is described in the following Subsections. First, we discuss the selected use case. Second, we list the used setup and technologies of our exemplary batch layer as well as the sample algorithm for data processing. Afterwards, the performance model prototype to support decision-making is presented. Finally, we evaluate the accuracy of the inferred decision-making on the basis of three selected scenarios and discuss results from our observed measures.

3.1 Use Case and Design Options

To represent incoming data and their distribution, we pick the example of a common smart energy use case as illustrated in Figure 5.

Here, several hundred wind turbines are positioned in several wind farms in different geographic locations with long distances onshore or offshore. In order to operate efficiently, they measure several thousand parameters per turbine such as pressure, temperature or vibrations of rotor blades. As they are subject to various influences, wind turbines are not always in operation and do not measure data, for instance, if they are defect or are maintained. While onshore wind turbines and wind farms, respectively, tend to have a time-based availability between 95-99%, the values for offshore wind farms with distance less than 12km range from

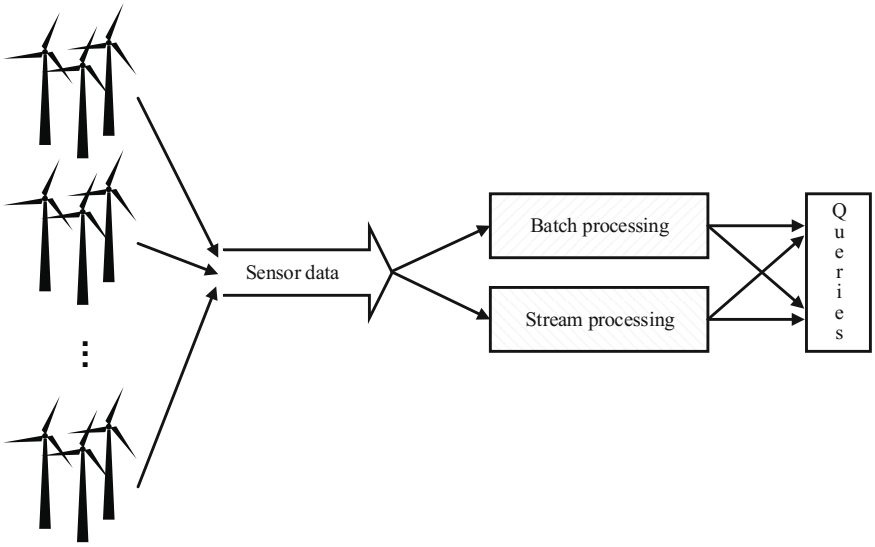


Fig. 5. Data processing of wind power facilities

67.4% to 90.4% [19]. However, wind turbines include also downtimes, if wind is too strong or too weak which is described by the metric energy-based availability. Faulstich et al. [20] compared time-based and energy-based availability of wind turbines. In an extreme case where the downtime due to defects and the downtime due to wind speed does not overlap, the energy-based availability lies within 90.4-95,2%.

Dependent on a wind turbine's availability, we assume it either produces a set of measurement data with constant volume or does not produce any output data. As a result, wind turbines generate not only immense amount of heterogeneous data, but also variable load which makes it difficult to predict the production rate of data. As soon as data are generated, they flow into a central data center where they are processed. Dependent on the use case, data are handled in different ways. They can be gathered and stored in a central repository where batch processing can be used to extract, transform, and load (ETL) data and to apply complex analytics. This procedure usually lies in the range of minutes or hours and is not suitable for real-time requirements. For this purpose, stream processing can be used to directly process data as they stream in. Here, analytical algorithms may be designed in a simpler and less complex way than at batch processing as well as implemented in slightly different way as they produce incremental results.

In scenarios where low latency results are required and normally stream processing is chosen, but also analysis of historical data by batch processing need to

be incorporated for conclusive results, the lambda architecture is an appropriate solution that allows for serving such use cases. Therefore, on both processing layers, stream and batch, the same kind of algorithm is implemented and results are joined.

Sensor data can be used for a variety of analytical scenarios such as for condition monitoring, diagnostics, predictive analytics or maintenance, and load forecasting. For our experiment, we concentrate on the latter example. Since the introduction of energy exchange such as the continuous intraday spot market of the European power exchange (EPEX), power can be bargained in 15-minute intervals up to 45 minutes before delivery which enables providers as well as consumers to efficiently act on short notice. In this case, the time-constraint is within 15 minutes. Typical forecast methods for short-term load forecasting include different exponential smoothing methods such as an autoregressive integrated moving average (ARIMA) model [33] or recurrent neural networks [29]. Furthermore, these algorithms are often applied on a sliding window of historical data.

Therefore, we will use this smart energy scenario as an example for our proposed approach and generate sensor data that are processed by one central system in similarly way as we have modeled it in a previous work [23]. The generator produces comma-separated values (CSV) files that represent measurements from wind turbines of one wind farm. Listing 1 shows the file structure and syntax.

Listing 1. Example of generated monitoring data from wind turbines

```
id,   timestamp,                  power,   param1, ...  paramN
12,   2015-04-01 08:23:04.125,    12.67,   value1, ...  value1
15,   2015-04-01 08:23:03.973,    13.49,   value2, ...  value2
13,   2015-04-01 08:23:04.096,    12.59,   value3, ...  value3
...
```

Each line represents a measurement of one wind turbine consisting of a *id*, *timestamp*, a *power* value and several hundred more parameters which we generated randomly and do not include in our succeeding analytic algorithms.

3.2 Implementation of the Batch Layer

To examine the accuracy of response time prediction for batch processes, we setup the batch layer using HDFS to store data sets and Hadoop MapReduce for batch processing. For simplicity, we installed a single node cluster in pseudo-distributed mode so Apache Hadoop runs only on one machine, but their daemons have their own Java processes. In order to do load forecasting and apply the data generator as mentioned in Section 3.1, we implemented a simple moving average algorithm in a Hadoop MapReduce job. It is based on an example algorithm[2].

[2] https://github.com/jpatanooga/Caduceus/

The MapReduce programming model intends to implement one map and one reduce function. The former takes a key/value pair as input and produces a set of key/value pairs, whereas the latter takes a key and set of associated values and combines the values to another smaller set [18]. In our case the map function is implemented as

Listing 2. Map function pseudo code

```
map(Object key1, String value1):
   // key1:   file name
   // value1: measurements of wind turbines of one farm
   for each line l in value:
      kv = parse(l)
      emit({kv.id, kv.timestamp}, {kv.timestamp, kv.power})
```

The function is called for each file within a given folder. It receives one CSV file and its value, which are multiple rows of measurement data of wind turbines. The algorithm reads every line and parses it in order to filter the *id* of a wind turbine, the *timestamp* of the measurement and the *power* value that describes the generated power to that time. Afterwards it releases a composite key containing the *id* and *timestamp*, and the values *timestamp* and *power*. By using a composite key Hadoop sorts the ids of wind turbines and, in a secondary sort, the timestamp for each id. Subsequently, the reduce method results in a simpler design as displayed in Listing 3.

Listing 3. Reduce function pseudo code

```
reduce(Object key, Iterator<object> values):
   // key:    an object containing id and timestamp
   // values: power values ordered by timestamp
   result = simpleMovingAverage(values)
   emit(id, result)
```

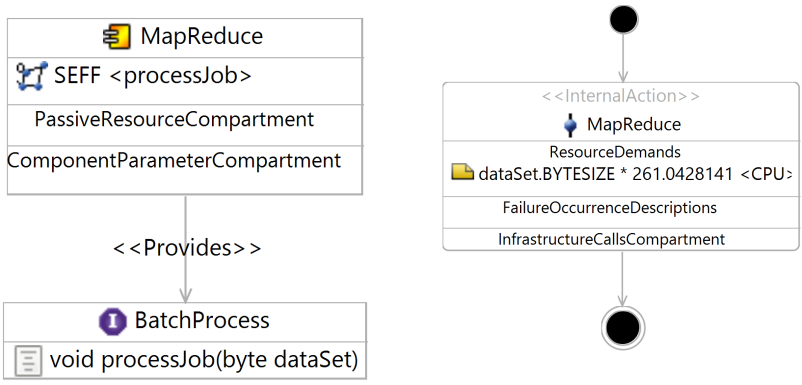
The reduce function is called for each different wind turbine and calculates the actual simple moving average. It receives the key object and a list of values as input which contains timestamps and power values sorted by the former. The function itself calculates the *result* and emits it with the corresponding wind turbine *id*.

3.3 Performance Model Prototype

We use the Palladio component model (PCM) [12] for our performance model. PCM is an annotated software architecture model that allows for describing performance relevant factors of software architecture, execution environment and usage profile [13]. Such performance models enable software architects and performance engineers to predict performance metrics such as response time, utilization or throughput by means of simulation or analytical solving.

PCM is divided into several sub-models. In the repository model, we specify a batch process as a software component with its service effect specification (SEFF) to describe the resource demands of the provided service. In the resource

environment model, we describe the hardware resources and processing rates on which a batch process will be executed. The concrete assignment of modeled batch processes to resources is determined in the allocation model. Finally, we specify the load intensity from wind turbine measurements in the usage model.



(a) Repository model

(b) Service effect specification (SEFF) <processJob>

Fig. 6. Modeling a batch process with the Palladio component model

Figure 6 shows the substantials of modeling the batch process in our performance model. As shown in Figure 6a, we specify one interface *BatchProcess* with the method *processJob* to analyze an input data set. The implementation of the interface and its method is modeled by the component *MapReduce* with the corresponding SEFF. As illustrated in Figure 6b the SEFF itself solely consists of a CPU resource demand in dependence on an incoming data set size. The data set size is specified in the usage model, in our case, in gigabyte.

In order to define the CPU resource demand and simulate a realistic system behavior we integrated measurements into our performance model. Therefore, we measured response times of the MapReduce job described in Section 3.2 while running it. Afterwards, we used an approximation with response times, which is also implemented by the LibReDe library [32], to estimate the required CPU time each process takes per transaction. One transaction means exactly one batch process that analyzes a set of messages. In our case, the resulting resource demand we estimated is 261 as represented in Figure 6b.

In order to predict results, PCM instances must be first transferred to be either simulated or solved analytically. Available model transformations are a model-to-text transformation like SimuCom [12], queuing Petri nets (QPN) transformations as well as a transformation to layered queuing networks (LQN). Brosig et al. [13] evaluated these model transformations with regards to their efficiency and accuracy. In our application scenario, time is critical and the model need to be solved as efficiently as possible so resulting predictions are available

at an early opportunity and the next batch process can be initiated. Therefore, we recommend the use of a model transformation to LQNs. It showed to be the most efficient solution as it is an analytical solver [13].

The performance model prototype has the limitation that is does not reflect the scheduling of processes itself within a cluster, for instance, as accomplished by Apache Hadoop YARN. Therefore, we assume sufficient available resources so batch and stream processes always run without interference.

3.4 Controlled Experiment

To conduct our experiments we run the mentioned data generator to produce CSV files for 10 wind farms with 100 wind turbines each, whereas one wind turbine approximately produces one measurement every second. Afterwards, we run the implemented Hadoop MapReduce job which reads only data measured within a sliding window of 24 hours. While the batch process is running, meanwhile we determine the incoming data volume. After the batch process is finished, we predict the response time of the second next batch process using our performance model. For the immediate succeeding batch process, we exactly know the data volume it will process as we know the historical data distribution and tracked new arrived data. For the batch process to be predicted, the data volume must be estimated. Therefore, a variety of specialized tools and algorithms exist to classify and forecast workload such as the approach by Herbst et al. [21]. As we target an efficient solution and a short-term forecast is required, namely, only the next point, we only use a naïve forecast in this study. It does not involve any computational overhead and simply takes the value of the latest observation as next forecast point in contrast to other methods such as cubic smoothing splines or ARIMA 101 that are more appropriate for scenarios with strong trends or noises [21]. In our case, the next forecast point equals the arrived data volume which has not been absorbed by the last batch process yet. Afterwards, we trigger the performance models with the predicted load intensity as input, and compare the predicted response time with the eventual measured response time.

As already mentioned, the aim is to minimize the usage of the speed layer. The level of potential resource reductions and costs savings that can be achieved depends on the characteristics of the underlying workload and variations in data distributions. The effectiveness of our solution itself, however, depends on how well the data volume is predicted and, especially, how accurate batch processes are predicted. Therefore, we concentrate on the latter in this controlled experiment and perform three selected scenarios with different load intensities by assuming different availabilities of wind turbines based on Faulstich et al. [19,20] to evaluate the accuracy of our solution.

In the first scenario, we assume the wind turbine availability (WTA) is constant during two following batch iterations. Consequently, the measurement data wind turbines produce do also not fluctuate so the predicted load intensity using a naïve forecast is very close to the actual measured load intensity. In the second scenario, we assume an increase of the WTA of 5 % for the subsequent batch process and, vice versa, we assume a decrease in a final third scenario. For each

Table 1. Measured and predicted results of batch processes

Scenario	WTA	Fluctuation	PRT	MRT	RE
1	85 %	± 0 %	12.78 minutes	12.17 minutes	5.01 %
	90 %	± 0 %	13.53 minutes	13.60 minutes	0.51 %
	95 %	± 0 %	14.28 minutes	15.47 minutes	7.69 %
2	85 %	+ 5 %	12.78 minutes	13.82 minutes	7.53 %
	90 %	+ 5 %	13.53 minutes	15.03 minutes	9.98 %
3	90 %	− 5 %	13.53 minutes	12.58 minutes	7.55 %
	95 %	− 5 %	14.28 minutes	13.17 minutes	8.43 %

scenario, we conduct several experiments with different WTA to also validate the prediction accuracy under different load intensities. Afterwards we compare predicted response times (PRT) with eventual measured response times (MRT) of the batch process and calculate the relative error (RE) of the PRT. The results are listed in Table 1.

For a WTA of 85% and no fluctuation during the following batch process, we predict the response time for the batch process to be 12.78 minutes. We measured a MRT of 12.17 minutes which leads to a RE of 5.01%. For a WTA of 90%, the RE of the predicted response time is only 0.51 % and 7.69% for a WTA of 95%.

In the second scenario, for a 85% WTA and a 5% increase of available wind turbines during the following batch iteration, the PRT is 12.78 minutes and the MRT 13.82 minutes with a 7.53% RE. Here, the PRT equals the same PRT as in the experiment for first scenario with a 85% WTA since the naïve forecast, as already mentioned, uses the last observation point, namely 85%, as next prediction point. The same occurrence also applies for the following experiments. The highest RE with 9.98% appeared for a WTA of 90% with +5% fluctuation at which the PRT is 13.53 minutes and the MRT 15.03 minutes.

For a decrease of the 5% WTA in the last scenario, we measured REs in the range similar to the former scenario. With a starting point of 90% WTA, the PRT is 13.53 minutes and the MRT 12.58 minutes. For 95% WTA, the PRT equals 14.28 minutes and MRT 13.17 minutes.

In our experiments, we showed that we are able to predict the response times of a batch process or MapReduce job, respectively, with RE between 0.51% and 9.98%. With regards to our exemplary use case, power can be traded every quarter of an hour in the intraday spot market. Assuming a fluctuating workload and a maximum acceptable response time of 14 minutes remaining one minute buffer, we would be able to accurately schedule stream processing in the second scenario, namely, not to switch on in the first experiment and to switch on stream processing in the second experiment as the MRT exceeds the time-constraint with 15.03 minutes. For a decreasing fluctuation, we would proper schedule stream processing for a starting WTA of 90%. However, for the last experiment in Table 1, we would have left the speed layer switched on as the PRT lies over 14 minutes in contrast to the MRT which is mainly caused by the naïve forecast.

4 Related Work

Similar to our use case, Sequeira et al. [31] propose a system based on the lambda architecture to analyze energy consumption. Martnez-Prieto et al. [25] adapted the architecture for semantic data and Casado and Younas [15] give an extensive review about technologies for the lambda architecture. Regarding optimization or efficient resource usage of the architecture, however, related research mainly focuses on the processing layers itself. For instance, Aniello et al. [3] and Rychl et al. [28] specify on scheduling stream processes, while Alrokayan et al. [1] concentrate on scheduling batch processes.

Regarding predicting batch processes, there is comprehensive research available, for instance, specialized for MapReduce jobs [11], [34], [35] as well as for big data applications in cloud infrastructures [16].

To overcome redundancy regarding software development and infrastructure complexity, approaches such as storm-yarn[3] or by Nabi et al. [27] exist to integrate stream processing in the Apache Hadoop environment. Summingbird[4] is an open source library that allows to write algorithms that can be used for batch as well as stream processing.

5 Conclusion and Future Work

This paper introduced a novel approach to use resources more efficiently when implementing the lambda architecture. It is applicable for usage scenarios where time constraints of queries are not permanently required to be low or lie within several minutes. To reduce processing power, we propose to switch on stream processing on demand in cases where batch processes are likely to exceed time requirements. By using historical information of incoming data and naïve forecasting to classify workload, we predicted the response time of succeeding batch iterations. Therefore, we used performance models in which we integrated estimated resource demands based on measurements. The results allow us to make decisions when additional stream processes are required or, vice versa, can be saved to reduce resource usage. If hardware provision is used in a as-a-service manner, it allows for reducing costs directly.

For future work we plan to automate the process illustrated in Figure 1. This involves to automatically measure incoming data during each batch iteration, apply workload forecasting techniques and trigger solving the performance model. Another challenge is to also integrate the speed layer into our test environment. This will enable us to examine our approach and its efficiency for successive batch iterations for a lengthy period of time. Furthermore, we will integrate other workload forecasting techniques besides the naïve forecast to evaluate possible prediction enhancements and scheduling optimizations.

[3] https://github.com/yahoo/storm-yarn
[4] https://github.com/twitter/summingbird

References

1. Alrokayan, M., Vahid Dastjerdi, A., Buyya, R.: Sla-aware provisioning and scheduling of cloud resources for big data analytics. In: Proceedings of the 2014 IEEE International Conference on Cloud Computing in Emerging Markets, pp. 1–8. IEEE (2014)
2. Amazon Web Services: Amazon Kinesis (2015). http://aws.amazon.com/kinesis/ (accessed: April 28, 2015)
3. Aniello, L., Baldoni, R., Querzoni, L.: Adaptive online scheduling in storm. In: Proceedings of the 7th ACM International Conference on Distributed Event-based Systems, pp. 207–218. ACM, New York (2013)
4. Apache Cassandra: The Apache Cassandra project (2015). http://cassandra.apache.org/ (accessed April 28, 2015)
5. Apache Hadoop: Welcome to Apache Hadoop! (2015). http://hadoop.apache.org/ (accessed April 28, 2015)
6. Kafka, A.: A high-throughput distributed messaging system (2015). http://kafka.apache.org/ (accessed April 28, 2015)
7. Apache Pig: Welcomt to Apache Pig! (2014). https://pig.apache.org/ (accessed April 28, 2015)
8. Apache Samza: Samza (2015). http://samza.apache.org/ (accessed April 28, 2015)
9. Apache Spark: Lightning-fast cluster computing (2015). https://spark.apache.org/ (accessed April 28, 2015)
10. Apache Storm: Storm, distributed and fault-tolerant realtime computation (2015). http://storm.apache.org/ (accessed April 28, 2015)
11. Barbierato, E., Gribaudo, M., Iacono, M.: Performance evaluation of nosql big-data applications using multi-formalism models. Future Generation Computer Systems **37**, 345–353 (2014)
12. Becker, S., Koziolek, H., Reussner, R.: The palladio component model for model-driven performance prediction. The Journal of Systems and Software **82**(1), 3–22 (2009)
13. Brosig, F., Meier, P., Becker, S., Koziolek, A., Koziolek, H., Kounev, S.: Quantitative evaluation of model-driven performance analysis and simulation of component-based architectures. IEEE Transactions on Software Engineering **41**(2), 157–175 (2015)
14. Brunnert, A., Vögele, C., Danciu, A., Pfaff, M., Mayer, M., Krcmar, H.: Performance management work. Business & Information Systems Engineering **6**(3), 177–179 (2014)
15. Casado, R., Younas, M.: Emerging trends and technologies in big data processing. Concurrency and Computation: Practice and Experience **27**(8), 2078–2091 (2015)
16. Castiglione, A., Gribaudo, M., Iacono, M., Palmieri, F.: Modeling performances of concurrent big data applications. Practice and Experience, Software (2014)
17. Chen, C.L.P., Zhang, C.Y.: Data-intensive applications, challenges, techniques and technologies: a survey on big data. Information Sciences **275**, 314–347 (2014)
18. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. Communications of the ACM **51**(1), 107–113 (2008)
19. Faulstich, S., Hahn, B., Tavner, P.J.: Wind turbine downtime and its importance for offshore deployment. Wind Energy **14**(3), 327–337 (2011)
20. Faulstich, S., Lyding, P., Tavner, P.: Effects of wind speed on wind turbine availability (2011)

21. Herbst, N.R., Huber, N., Kounev, S., Amrehn, E.: Self-adaptive workload classification and forecasting for proactive resource provisioning. Concurrency and Computation: Practice and Experience **26**(12), 2053–2078 (2014)
22. von Kistowski, J., Herbst, N.R., Kounev, S.: LIMBO: A tool for modeling variable load intensities. In: Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering, pp. 225–226. ACM, New York (2014)
23. Kroß, J., Brunnert, A., Prehofer, C., Runkler, T.A., Krcmar, H.: Model-based performance evaluation of large-scale smart metering architectures. In: Proceedings of the 4th International Workshop on Large-Scale Testing, pp. 9–12. ACM, New York (2015)
24. Liu, X., Iftikhar, N., Xie, X.: Survey of real-time processing systems for big data. In: Proceedings of the 18th International Database Engineering & Applications Symposium, pp. 356–361. ACM, New York (2014)
25. Martnez-Prieto, M.A., Cuesta, C.E., Arias, M., Fernnde, J.D.: The solid architecture for real-time management of big semantic data. Future Generation Computer Systems 47, 62–79 (2015), special Section: Advanced Architectures for the Future Generation of Software-Intensive Systems
26. Marz, N., Warren, J.: Big data: principles and best practices of scalable real-time data systems. Manning Publications Co. (2015)
27. Nabi, Z., Wagle, R., Bouillet, E.: The best of two worlds: integrating ibm infosphere streams with apache yarn. In: Proceedings of the 2014 IEEE International Conference on Big Data, pp. 47–51. IEEE (2014)
28. Rychlý, M., Škoda, P., Smrž, P.: Heterogeneity-aware scheduler for stream processing frameworks. International Journal of Big Data Intelligence **2**(2), 70–80 (2015)
29. Schäfer, A.M., Zimmermann, H.-G.: Recurrent Neural Networks Are Universal Approximators. In: Kollias, S.D., Stafylopatis, A., Duch, W., Oja, E. (eds.) ICANN 2006. LNCS, vol. 4131, pp. 632–640. Springer, Heidelberg (2006)
30. Schermann, M., Hemsen, H.: Buchmller, C., Bitter, T., Krcmar, H., Markl, V., Hoeren, T.: Big data - an interdisciplinary opportunity for information systems research. Business & Information. Systems Engineering **6**(5), 261–266 (2014)
31. Sequeira, H., Carreira, P., Goldschmidt, T., Vorst, P.: Energy cloud: Real-time cloud-native energy management system to monitor and analyze energy consumption in multiple industrial sites. In: Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing, pp. 529–534. IEEE (2014)
32. Spinner, S., Casale, G., Zhu, X., Kounev, S.: LibReDE: a library for resource demand estimation. In: Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering (ICPE 2014), pp. 227–228. ACM, New York (2014)
33. Taylor, J.W.: An evaluation of methods for very short-term load forecasting using minute-by-minute british data. International Journal of Forecasting **24**(4), 645–658 (2008)
34. Verma, A., Cherkasova, L., Campbell, R.H.: Aria: automatic resource inference and allocation for mapreduce environments. In: Proceedings of the 8th ACM International Conference on Autonomic Computing, pp. 235–244. ACM, New York (2011)
35. Vianna, E., Comarela, G., Pontes, T., Almeida, J., Almeida, V., Wilkinson, K., Kuno, H., Dayal, U.: Analytical performance models for mapreduce workloads. International Journal of Parallel Programming **41**(4), 495–525 (2013)