

Big Data Platform for Smart Grids Power Consumption Anomaly Detection

Peter Lipčák², Martin Macak^{1,2} and Bruno Rossi^{1,2}

¹*Institute of Computer Science, Masaryk University*

²*Faculty of Informatics, Masaryk University*

Brno, Czech Republic

Email: {plipcak,macak,rossi}@mail.muni.cz

Abstract—Big data processing in the Smart Grid context has many large-scale applications that require real-time data analysis (e.g., intrusion and data injection attacks detection, electric device health monitoring). In this paper, we present a big data platform for anomaly detection of power consumption data. The platform is based on an ingestion layer with data densification options, Apache Flink as part of the speed layer and HDFS/KairosDB as data storage layers. We showcase the application of the platform to a scenario of power consumption anomaly detection, benchmarking different alternative frameworks used at the speed layer level (Flink, Storm, Spark).

I. INTRODUCTION

BIG data architectures are designed to handle ingestion, processing and analysis of data that possesses the five V's properties: Volume, Velocity, Value, Variety, Veracity [1], [2]. In the context of Smart Grids (SG), utilities have to deal with an increasing volume of data leading to typical big data problems. According to Zhang et al. [3], the five V's in the SG domain are represented by several needs: to analyze large amounts of data in real-time, like from smart meter readings (Volume), to deal with quick generation of records (Velocity), and diversity of data structures (Variety), with multiplicity of use case, such as anomaly detection or load balancing that bring value to the customers (Value), and inherent problematics of data in terms of possible measurement errors (Veracity). To exemplify, with a sampling rate of 15 minutes, a sample of 1 Million Smart Meter devices installed results in around 3 Petabytes of data in one year (3000TB, ~35Billion records at a size of 5KB each record) [4].

There are a plethora of use cases for the application of big data analysis in the context of SGs [5], [6], like anomaly detection methods to detect power consumption anomalous behaviours [7], [8], the analysis of false data injection attacks [9], load forecasting for efficient energy management [10], among others. Such data analysis requirements create needs to define architectures and platforms to support large scale data analysis.

In this paper, we focus on power consumption data anomaly as the application scenario: dealing with the identification of anomalous patterns from energy consumption traces collected from smart meters, that can have several benefits for utilities, such as load optimizations based on determined patterns of energy usage [5], [7] or clustering of customers [11]. The final

goal is the definition and evaluation of a big data platform for power consumption anomaly detection.

We have two main contributions in this paper:

- the provision of a big data platform for power consumption anomaly detection with the main components mapped to the reference architecture proposed by Pääkkönen and Pakkala [12].
- the results of a scenario run with public datasets to assess the applicability of batch-oriented (Apache Spark), stream-oriented (Apache Storm), or hybrid (Apache Flink) frameworks in the speed layer of the platform.

The paper is structured as follows. In Section II, we discuss the background of big data analysis in the context of Smart Grids. In Section III, we discuss big data energy management platforms that can be comparable to our proposal. In Section IV, we propose a platform for big data power anomaly detection with components mapped to the reference architecture in Pääkkönen and Pakkala [12]. In Section V, we propose a power consumption scenario aimed at showcasing the application of the platform and the evaluation at the speed layer level of three frameworks from the Apache Software Foundation (Spark, Flink, Storm). The conclusions are presented in Section VI.

II. BACKGROUND - BIG DATA ANALYSIS ARCHITECTURES

Big data processing is assuming more and more relevance in many fields of modern society. For energy utilities, the needs to manage energy resources based on the vast amount of information collected from sensors and the ICT infrastructure is nowadays of paramount importance [3].

In the context of big data processing, we can have a first distinction between batch and stream processing. Batch processing is a type of processing executed on large blocks (batches) of data stored over a period of time. These data blocks are appended to highly scalable data stores and periodically analyzed in batches by big data processing frameworks. This approach to data processing is very effective in case of large datasets for use cases that are not time-critical, as the main drawbacks are higher latencies in processing requests [13]. On the other hand, stream processing allows dealing with data in real-time, getting approximate results that can be complemented, if needed, from the analysis of batch processing. Managing streams of data usually implies the capabilities of online learning. Some authors also consider

an intermediate category: micro-batch processing [13], that overcomes some of the issues of batch and stream processing: near real-time performance is granted by considering streams of data in micro-batches sent to the batch processing engine.

There are two popular architectures that were proposed over time, Lambda and Kappa [13], [14]. They were mainly based on the relevance that is given to batch and stream processing. **Lambda architecture** is a processing architecture designed to handle massive amount of data efficiently by taking advantage of both batch and stream-processing methods. Efficiency in this context means high-throughput, fault-tolerance and low latency [13]. The rise of the Lambda architecture is correlated with the growth of data and the speed at which they are being generated, real-time analytics and the drive to mitigate big latencies of map-reduce [15].

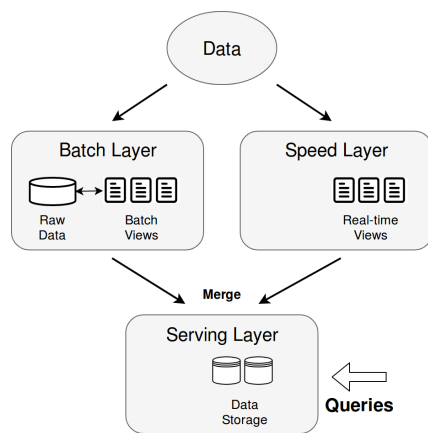


Fig. 1. Lambda Architecture

Generally, a Lambda architecture consists of three distinct layers (Fig. 1): a batch layer (batch processing), a speed layer (stream processing) and a serving layer (data storage). The batch layer is responsible for bringing comprehensive and accurate views of batch data while simultaneously, the speed layer provides near-real-time data views. Stream processing can take advantage of batch views and may be joined before presentation. Data streams entering the system are dual fed into both batch and speed layer.

The batch layer stores raw data as it arrives and computes the batch views in intervals. When the data gets stored in the data store using different data storage systems, the batch layer processes the data using one of the big data processing frameworks that implement the map-reduce programming paradigm. Popular frameworks that support batch processing are Apache Hadoop, Apache Spark, and Apache Flink.

The speed layer processes data streams in real-time with the focus on minimal latencies. Usually latencies vary from milliseconds to several seconds. This layer often takes advantage of pre-computed batch views. Popular frameworks that support stream processing are Apache Flink, Apache Storm, Apache Spark, and Apache Samza.

The serving layer aggregates the outputs from batch and speed layers, storing the data in a datastore. As storage, highly

scalable and distributed data lakes are often used. Among popular big data datastores belong Apache Cassandra, Apache HBase, Hadoop Distributed File System, OpenTSDB, and KairosDB.

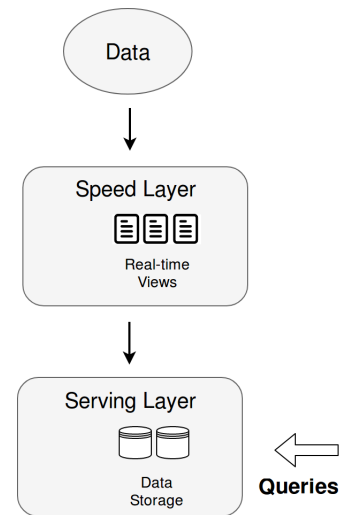


Fig. 2. Kappa Architecture

Kappa architecture is an alternative to the Lambda architecture proposed to overcome some of the limitations, like maintaining two code bases for batch and speed layers and the general complexity of the platform [16]. The Kappa architecture consists of two distinct layers (Fig. 2): a speed layer and a serving layer. The speed layer processes streams of data in the same way as Lambda architecture. The only main difference is that when the code changes, data needs to be reprocessed again. This is because parts of the Kappa architecture act as an online learner.

Big data analysis frameworks often do not support both batch and stream processing, thus a hybrid combination of frameworks is the choice, e.g., Apache Hadoop and Apache Storm can be used to fully support a Lambda architecture [17].

In the context of SGs, we can find some examples of the main constituting layers of both architectures. A batch layer in Spark, distributed in-memory computing framework, was used to pre-compute a statistical-based model using linear regression for anomaly predictions of energy consumption data [7]. A stream layer was used for the detection of defective smart meters, solution implemented using the Flink stream processing engine [18]. As a serving layer, KairosDB, time-series database built on top of Apache Cassandra, could handle the workload of a large city with around six million smart meters. During this experiment, KairosDB was installed in a cluster of 24 nodes [19].

The next section will discuss more in detail about big data energy management platforms that have been proposed so far.

III. RELATED WORKS—BIG DATA ENERGY MANAGEMENT PLATFORMS

We have discovered several proposed big data architectures in the SG domain that we mapped to either Lambda or Kappa based on the available information. We found that the majority of the architectures are cloud-oriented and that several energy management architectures do not specify the applicability to the big data context. Therefore, they might use different architectural structure than pure Lambda or Kappa.

A. Big Data energy management architectures

Mayilvaganan and Sabitha [20] proposed a SG architecture which uses HDFS and Cassandra to store historical data for the prediction of energy supply and demand. In this work, only MapReduce processing was used.

Munshi and Mohamed [21] presented a SG big data ecosystem based on the Lambda architecture. Smart meter data are being ingested to a cloud with Flume. For the batch layer of the Lambda architecture Hadoop is used, and Spark for the speed layer. Authors also performed data mining and visualization applications on top of this ecosystem with real data.

Liu and Nielsen [22] designed a smart meter analytic system. The architecture is divided into data ingestion, processing, and analytics layer. It can process both batch and stream data. In the processing module, they list several tools which can be used, like Spark, Hive, or Python. In the end, the data are sent to the analytics layer, which contains a PostgreSQL database, analytics libraries, and applications for users. This architecture can be viewed as Lambda-based because usage of Hive can be considered as a batch layer and the usage of Spark as a speed layer. The analytics layer of this architecture can be mapped to a serving layer.

Fernández et al. [?] proposed an architecture that improves energy efficiency management in a smart home. It consists of four modules: data collection, data storage, data visualization, and a machine learning module. It is designed to work with both batch and real-time processing. The data storage module consists of three blocks: acquisition, real-time, and batch block. From those processing blocks, the data are available to blocks that can be mapped to a service layer of a Lambda architecture. Therefore this architecture is also considered as Lambda-based.

Balac et al. [23] proposed an architecture for real-time predictions in energy management. The data streams are collected to the server, which provides management functionality like dashboards, alerting, and basic reporting. From this server, data are transferred to their high-performance file system where the real-time analysis is performed. The analysis results are then passed back to the server but are also archived in the cloud storage for some later batch processing task. Based on this description, this architecture can be viewed more as Kappa-based.

Al-Ali et al. [24] proposed a system for energy management in a smart home. They specify both hardware and software architecture. The software architecture contains three modules: data acquisition, a middleware, and a client application

module. In this architecture, data are stored and then used by several services. Therefore it does not represent neither a Lambda nor a Kappa architecture.

Daki et al. [25] presented an architecture which is composed of five parts: data sources, integration, storage, analysis, and visualization that can be used for the analysis of customer data. They provide a set of technologies which might be used and can be considered as either a Lambda or a Kappa architecture, depending on the use cases.

B. Other energy management architectures

Yang et al. [26] proposed an energy management system that uses a service-oriented architecture in the cloud. For storage, they use distributed software as MySQL Cluster and HDFS. However, authors do not specifically mention big data. Ali et al. [27] proposed a computing grid based framework for the analysis of system reliability and security. The architecture consists of three layers: application, grid middleware, and resource layer, with focus on high-performance computing. Rajeev and Ashok [28] presented a cloud computing architecture for power management of microgrids, consisting of four modules: infrastructure, monitoring, power management, and a cloud service module.

IV. PROPOSED PLATFORM

In this section, we propose our big data platform for smart meters power consumption anomaly detection, based on our previous research ([29], [8], [30], [31]). The goal of such platform is to process large amounts of data from smart meters and weather information sources to detect anomalous behaviours from the side of customers. Such analysis can allow to further create customer profiles that can be used to cluster users according to their power consumption behaviours [11]. The five V's in this area derive from several aspects, namely volume: the large amount of information traces generated by smart meters multiplied by the number of users [4], velocity: the needs for real-time analysis of such traces that are constantly generated [4], variety: multiple data sources involved, either structured or unstructured, mainly about power consumption and weather data [3], value: the added value that analyses can have for utilities, that can create customer profiles to optimize power production and balancing of the whole grid [11], and veracity: the many issues that measurements errors might pose, such as corrupted or missing data from smart meters [32].

We map our proposed architecture to the reference architecture by Pääkkönen and Pakkala [12]. The reference architecture for big data systems is technology independent and is based on the analysis of published implementation architectures of several big data use cases. We mapped all the functionalities, data stores and data flows contained within our platform proposal to the reference architecture diagram to allow for easier comparison with other platforms (Fig. 3):

Data sources. Our platform supports two possible data sources. First, a stream of semi-structured data is being collected from smart meters datasets. These can either be live data

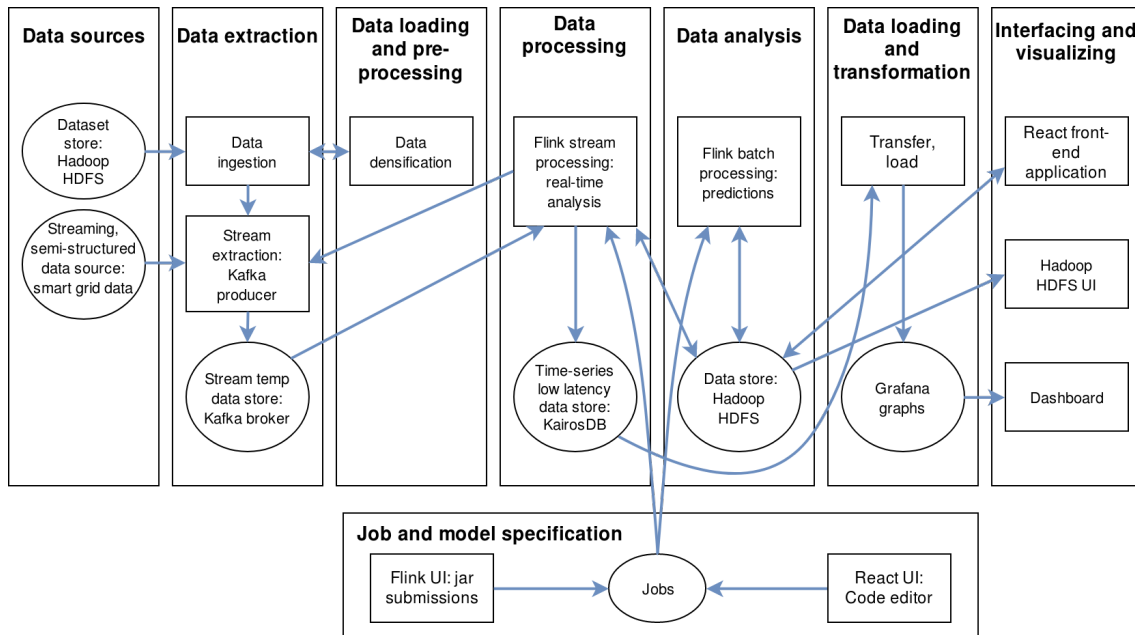


Fig. 3. Architecture Mapping to Pääkkönen and Pakkala [12] Reference Architecture

from smart meters or private/public available datasets. Another way for data to enter the platform is reading data available in the Hadoop Distributed File System (HDFS).

Data extraction. A Kafka producer corresponds to the stream extraction functionality and a Kafka broker serves as temporary data store. An ingestion manager extracts data from HDFS and sends data to the Kafka broker.

Data loading and pre-processing. Data extracted from HDFS using the ingestion manager can be further densified. Densification can increase the itemset for smaller datasets (e.g., for benchmarking reasons). Currently, two densification methods are available in the platform – multiplication (replicating n times the dataset) and interpolation (constructing new data points based on interpolating previous intervals n times). However, more advanced methods can be implemented, such as regression-based and probability-based methods [33].

Data processing. Flink's stream-processing jobs read incoming data streams from Kafka. These jobs can also access HDFS to read pre-computed models or datasets and merge them before producing results. The output of stream-processing jobs can be sent back to Kafka, HDFS or to the time-series datastore KairosDB.

Data analysis. Flink's batch-processing jobs can read data from HDFS and perform data analytics. The output can be further stored in HDFS.

Data loading and transformation. Results of stream-processing jobs stored in KairosDB can be loaded into a Grafana server. In this context, Grafana server serves as a temporary data store until the graphs are generated.

Data storage. The following data storage technologies (temporary or persistent) are supported by the platform: Kafka broker, HDFS, KairosDB and Grafana.

Interfacing and visualization. There are several user interfaces that allow interacting with the platform. HDFS provides an UI with information regarding storage and the file system. Flink UI gathers statistics about running jobs, e.g. records processed per second by an operator. A dashboard component displays graphs produced by Grafana. A React front-end application allows users to upload or delete datasets and shows datasets previews. It is also possible to select the dataset for ingestion, which invokes the ingestion Manager to read the dataset from HDFS and ingests the data into Kafka.

Job and model specification. Submitting stream and batch processing jobs can be done either by uploading a JAR file with all dependencies using the Flink UI or by submitting the code using an ad-hoc code-editor contained within a React front-end application. The JAR file has to include the source code for the processing job (e.g., an anomaly detection algorithm like in [7]), all the dependencies the job requires and the path to the entry Java file to be executed.

Discussing the platform's architecture (Fig. 4), data itemsets generated by smart meter devices are forwarded to the platform using a publish/subscribe messaging system. Kafka can be used as a data source and a data sink for Flink's jobs and the Kafka Connector provides access to event streams without manual implementation needs — making it a good choice for the platform. Each of the technologies is capable of running in clusters, providing scalability and fault-tolerance. Each technology can be scaled independently based on the use cases, making the platform flexible in terms of configuration.

Apart from the integration of existing big data tools, we implemented three separate applications to support the needs of smart meters data analysis (Fig. 4). These applications are: i) an ingestion manager, responsible for densification and

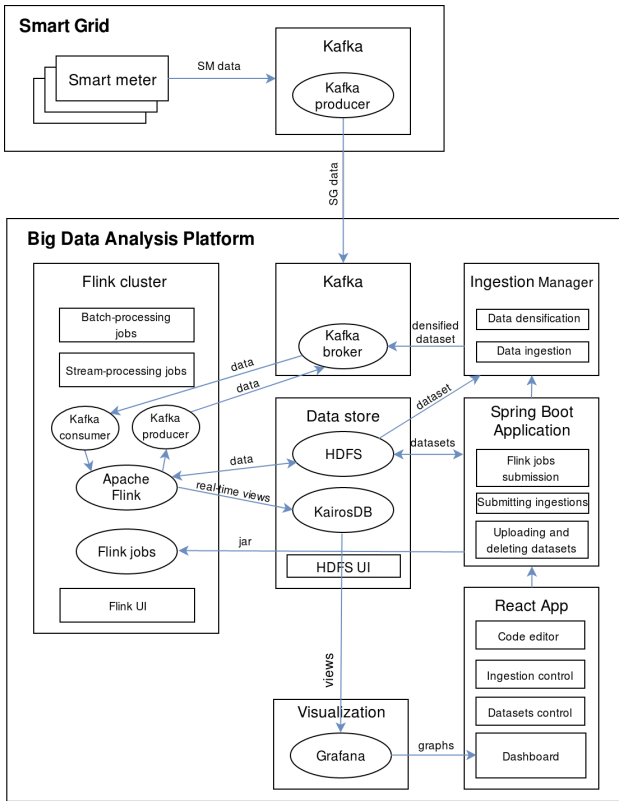


Fig. 4. Platform Architecture

ingestion of datasets into Kafka, ii) a Spring Boot Application, i.e. a back-end implementation with a RESTful API which enables execution of ingestions, Apache Flink job submissions, and uploading and deleting datasets to/from HDFS, iii) a React front-end application that allows users to submit ingestions, upload and delete datasets. Apache Flink provides both batch and stream processing of the data, thus a Lambda architecture is fully supported, as discussed in the background section. As the main data storage, HDFS was chosen and KairosDB is used to provide real-time views on the data. Grafana can be integrated with KairosDB and produce real-time graphs of data. Generated graphs can be further shown in the dashboard of the application.

V. POWER CONSUMPTION ANOMALY DETECTION SCENARIO

We showcase the use of the platform in the context of power consumption anomaly detection. Studying unusual consumption behaviors of customers and discovering unexpected patterns is an important topic related to the use of smart metering devices in the smart grids domain, as discussed in the previous section and in related research ([7], [8]).

In this context, we propose a scenario to showcase the platform and to look into the performance of three different frameworks for the streaming part: batch-based (Spark), stream-based (Storm), and hybrid (Flink). As the speed layer

is a key part of the performance of the platform, the selection of the best framework is an important decision.

A. Compared Frameworks

Apache Spark. Created in 2009, is a general purpose processing engine suitable for a wide range of use cases. There are four libraries built on top of Spark processing engine: Spark SQL for SQL language support, Spark Streaming for stream processing support, MLlib for machine learning algorithms, GraphX for graph computations. Languages supported by Spark include: Java, Python, Scala, and R.

Spark applications consist of a driver program that runs the main function and executes various operations in parallel. The main abstraction that Spark provides is a resilient distributed dataset (RDD), which is a collection of elements partitioned across the cluster nodes. Operations such as map or filter executed on RDDs are executed in parallel. Spark Streaming discretizes the streaming data into micro-batches, then latency-optimized Spark engine runs short tasks to process the batches and outputs the results. Each batch of data is an RDD [34].

Apache Flink. Created in 2009, is a framework and distributed processing engine for computations on both batch and streams of data. Stream processing is supported natively and provides excellent performance with very low latencies. It also provides a machine learning library called FlinkML as well as a graph computation library Gelly. Supported programming languages are Java, Scala, Python, and SQL.

Flink provides different levels of abstraction that can be used to develop batch or stream processing applications. Abstractions from low-level to high-level respectively are as follows: Stateful Streaming Processing, DataStream / DataSet API, Table API and SQL. In practice, most applications would not need the lowest-level abstraction, Stateful Streaming Processing, but would rather program against the Core APIs like the DataStream API (bounded/unbounded streams) and the DataSet API (bounded datasets). These APIs provide very similar operations as Spark's RDDs such as map, filter, aggregation and other transformations [35].

Apache Storm. Created in 2011, is a distributed real-time processing engine. Storm is a pure stream processing framework without batch processing ability. Storm provides great throughput with very low latencies. It was designed to be usable with any programming language thanks to its Thrift definition for defining and submitting topologies. Zookeeper is also required to be installed because Storm uses it for cluster coordination.

The overall logic of Storm applications is packaged into a topology. A Storm topology is analogous to a MapReduce job with the difference that Storm applications run forever. A topology is an acyclic directed graph (DAG) composed of spouts and bolts connected with stream groupings. The stream is a core abstraction in a Storm application. A stream is an unbounded sequence of tuples that are generated and processed in parallel. Spouts serve as a source of streams in a topology. Analogically to Flinks DataStream/DataSet or Sparks RDDs operations, bolts are responsible for all the

distributed processing. Bolts can implement operations such as map, filter or aggregation [36].

We could not find any specific benchmarking study of the three frameworks based on smart grids related datasets, but previous studies found contrasting results in other domains.

Karimov et al. [37] found Flink to have more than three times faster throughput than Spark and Storm for aggregations. Joins were more than two times faster for Flink than Spark. Flink outperformed Storm and Spark in six out of seven benchmark categories including throughput and latency. However, Spark was found to perform better than the two other frameworks in case of skewed data, as well to improve the performance more than the other frameworks in presence of more than three nodes.

Wang et al. [38] developed a full benchmarking system to test the performance of Storm, Flink, and Spark. The results of the application of the benchmark showed that Flink is three times faster than Spark and six times faster than Storm in processing advertisement clicks. The final conclusion is that Storm would be the best choice if very low latency is requested, while Spark would be a good option if throughput is a key aspect of the use case. Flink gives a more balanced performance with low latency and high throughput.

Lopez et al. [39] found that Storm and Flink were consistently better than Spark in terms of throughput. Storm had in general better throughput than the other frameworks, while Spark had the worse performance due to the application of micro-batches, as each batch is grouped before processing. However, Spark was found to be more reliable in terms of node failures and recoverability of the functionality. The conclusion is that the lower performance of Spark Streaming might be justified in use cases in which absolute reliability is necessary, considering no messages loss in case of nodes failures.

Chintapalli et al. [40] performed a benchmarking of Spark, Flink, Storm Streaming focusing on latency. Storm performed the best with Flink, both having less than 2 seconds latency at high throughput. The latencies of Spark, on the other hand, were rising with higher throughput, resulting in latencies of over 8 seconds.

B. Scenario definition

Our scenario contains three implementations of the same algorithm using the three different big data processing frameworks (Spark, Storm, Flink). Implementing the same use case we can easily compare the performance of each framework by measuring the processing time of the same dataset.

Dataset. Our first dataset consists of power consumption data collected from apartments in one building with a sampling rate of 15 minutes [41]. The apartment dataset $(id, timestamp, consumption(kW))$ contains data for 114 single family apartments for the period 2014-2016. The second dataset contains weather data $(timestamp, temperature, humidity, pressure, windspeed, \dots)$ with a sampling rate of one hour. The size of the apartment dataset is 2.1 GBs and contains 64 million records. For the scenario to better

represent a big data context, we replicated each of these records eight times. The ingestion manager part of the platform with multiplication densification method was used for this purpose. The result size of the testing dataset is 512 million records in CSV format.

Scenario Setup. The scenario was run with three server nodes: a Master node and two worker nodes (Fig. 5). Apart from Kafka, each technology runs in a cluster on all three nodes. Kafka was only running in the Master node, as we considered it could serve all other nodes without delays. For Kafka to operate, there is the need to have Apache Zookeeper to manage the cluster. Because Apache Storm uses Zookeeper for cluster management as well, we decided to install Zookeeper in all three nodes. Each of these nodes was configured with an Intel Xeon E3 (2.4GHz, 4 cores), 8GB RAM running on Ubuntu 16.04 LTS 64bit Linux 4.4.0 kernel.

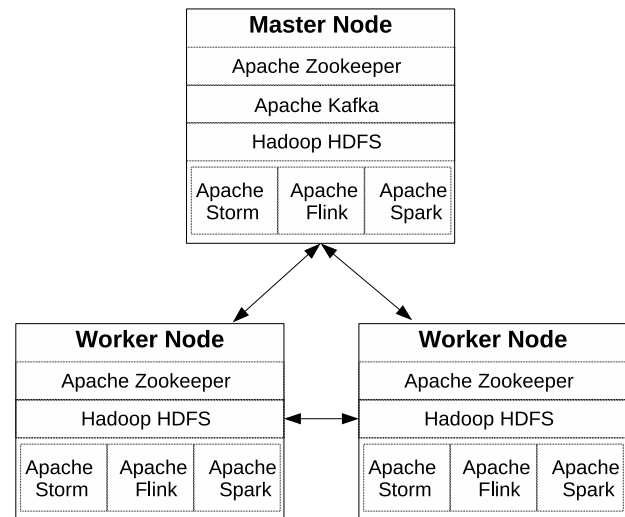


Fig. 5. Nodes involved in the experimental setup

Anomaly Detection Algorithm Implementation. We implemented a simple algorithm for finding consumption anomalies using a pre-computed model of consumption predictions (pseudocode, algorithm 1). For the computation of anomaly detection, we implemented a Spark batch processing program in the Java programming language. We did not implement this algorithm using other big data processing frameworks because our main focus was on measuring performance of stream processing. To decide whether a current power consumption item is an anomaly, we analyse power consumptions of three previous days. We take each hour of a day as a season, i.e., $t=[0, 23]$, and use previous three days consumptions at the time t to compute our predictions. We also take into consideration outside weather because it is highly correlated with power consumption: during winter power consumption is higher due to heating, as well as in summer due to the cooling equipment in function. For each apartment, predictions are made individually because customers have different living habits and power consumption predictions cannot be

generalized to all of them.

```

Input:  $C, T$            ▷ Consumption and Temperature Datasets
Output:  $P$              ▷ Anomaly Detection Model
function CreateAnomalyDetectionModel( $C, T$ ):
     $P \leftarrow \emptyset$            ▷ Initialize Anomaly Detection Model
    foreach  $d \in \text{days}(2014 - 2016)$  do
        foreach  $s \in 0..23$  do
            foreach  $id \in \text{apartmentIds}$  do
                 $C1 \leftarrow \text{MakeAverage}(\text{GetConsumptions}(d - 1, s, id))$ 
                 $XT1 \leftarrow \text{Compute.XT}(\text{GetOutsideTemperature}(d - 1, s))$ 
                 $C2 \leftarrow \text{MakeAverage}(\text{GetConsumptions}(d - 2, s, id))$ 
                 $XT2 \leftarrow \text{Compute.XT}(\text{GetOutsideTemperature}(d - 2, s))$ 
                 $C3 \leftarrow \text{MakeAverage}(\text{GetConsumptions}(d - 3, s, id))$ 
                 $XT3 \leftarrow \text{Compute.XT}(\text{GetOutsideTemperature}(d - 3, s))$ 
                 $\text{Prediction} \leftarrow (C1 * XT1 + C2 * XT2 + C3 * XT3) / 3 + 7$ 
                 $P.\text{insert}(d, s, id, \text{Prediction})$ 
            end
        end
    end
    return  $P$ 

```

Algorithm 1: Anomaly Detection Model Pseudocode: s =season, n =day, C =avg power consumption, XT =outside temp variables

The evaluation of the performance of the frameworks happens at the speed layer. The speed layer of our Lambda implementation is taking advantage of the pre-computed model (algorithm 1), based on which we can detect anomalies of incoming power consumption data. First, we load the anomaly detection model from the datastore and then we compare smart meter readings against this model. If the new consumption value exceeds the value from the model, we consider it as an anomaly (pseudocode, algorithm 2).

We implemented the stream processing part in Java using each framework (Spark, Flink, Storm). Each implementation contains framework specific distributed operations such as map, filter, foreach.

```

Input:  $M, C$            ▷ Anomaly Detection Model and New
                        Consumptions
Output:  $A$              ▷ Anomalies
function AnomalyDetection( $M, C$ ):
     $A \leftarrow \emptyset$            ▷ Initialize Anomalies
    foreach  $c \in C$  do
         $P \leftarrow M.\text{get}(c.\text{day}, c.\text{season}, c.\text{id})$ 
        if  $c.\text{value} > P$  then
             $A.\text{insert}(c.\text{day}, c.\text{season}, c.\text{id}, P)$ 
        end
    end
    return  $A$ 

```

Algorithm 2: Anomaly Detection Streaming Process

Process Flow of Anomaly Detection. The flow of the process of anomaly detection in this scenario is as follows. The power consumption dataset, as well as the weather dataset, were both stored in HDFS. First, Spark loads the datasets into

memory and performs operations in a distributed environment to generate the prediction model using Algorithm 1 (Fig. 6, steps 1-2). The output of Spark is stored back into HDFS (Fig. 6, step 3).

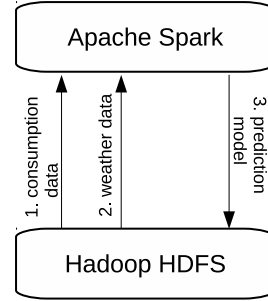


Fig. 6. Anomaly Detection Model Data Flow

Real-time Anomaly Detection Data Flow. The data flow of real-time anomaly detection is shown in Fig. 7. The prediction model is first loaded into memory from HDFS (Fig. 7, step 1). After the big data processing framework in use (Spark, Storm, Flink) is initialized and ready, we can start streaming data into Kafka. This is done using the ingestion manager. The ingestion manager reads consumption data from HDFS and sends each record multiple times to Kafka, in this scenario eight times (Fig. 7, steps 2-3) using the multiplication densification method. Big data processing frameworks subscribe to consumption topic and right after new data arrive, they start processing (Fig. 7, step 4). All found anomalies are sent to the Kafka topic "anomalies" (Fig. 7, step 5).

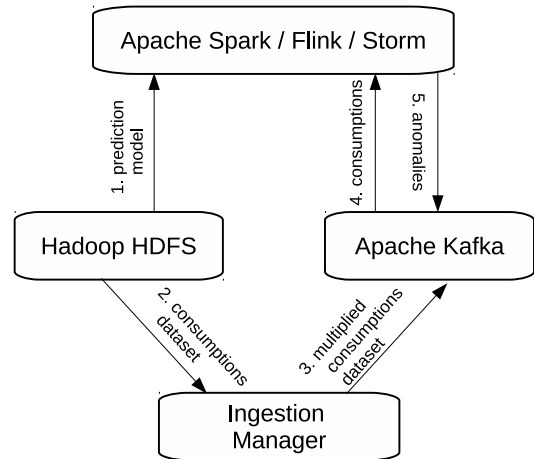


Fig. 7. Real-time Anomaly Detection Data Flow

C. Results

We can get several insights about running the platform with each of the three frameworks as the speed layer (we summarize them in Table I).

Throughput. Records per seconds processed (Fig. 8) showed that Storm (~378k records per second) and Flink

(~438k records per second) were significantly faster than Spark Streaming (~168k records per second). For the power consumption dataset used in the scenario (512 million records), this means average times of ~20min. (Flink), ~25min. (Storm) and ~50min. (Spark). If throughput is important, Flink seems to give the best results.

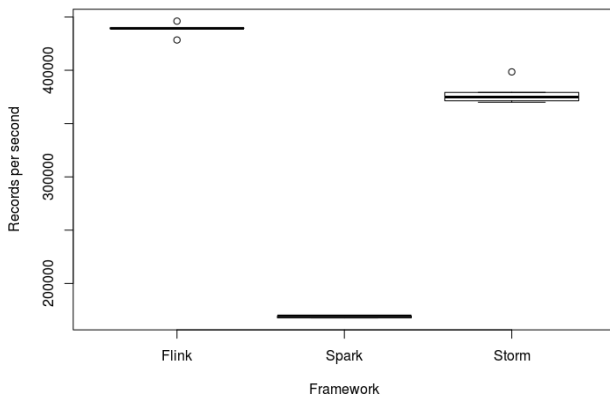


Fig. 8. Throughput. Records per seconds processed (5 runs per framework)

Latency. Internally, Spark Streaming receives live data from various sources and divides them into batches (micro-batches), which are then processed by the Spark engine to generate a stream of results. Thus, it is not considered as native streaming, but this way it can also efficiently support processing of big streams of data. However, Spark Streaming strongly depends on the batch intervals which can range from hundreds of milliseconds. During the scenario runs, both Flink and Storm had lower latencies, in terms of milliseconds, while Spark had latencies in terms of seconds. However, the selection depends on how important the batch layer (or micro-batch) is for the specific scenario. For the current scenario using a pre-computed (batch-level) model or an online learning algorithm at the speed layer, both Flink and Storm can be better options than Spark, considering latency.

Support for batch processing. In the scenario, the batch layer is only used for a pre-computed model, if updating the model is necessary, Spark natively supports batch processing and provides a very efficient batch processing engine. Batch processing in Flink is dealt with as a special case of stream processing. Apache Flink provides streaming API that can do both bounded and unbounded use cases, but also offers different DataSet API and runtime stack that is faster for batch processing use cases, so it is also possible to process batches very efficiently. Storm does not support batch processing in the current version. In this scenario, the pre-computed model was managed by Spark, so a hybrid combination of frameworks is necessary for a Lambda architecture, if adopting Storm.

Effort to set-up and configure each framework. Setting up the cluster nodes for the scenario with the default configuration requires very little time, several minutes to one hour for

TABLE I
SUMMARY OF THE SPEED LAYER FRAMEWORKS COMPARISON

	Spark	Flink	Storm
Performance	Medium	Very good	Good
Latency	Medium	Very low	Very low
Batch processing support	Yes	Yes	No
Cluster configuration effort	High	Medium	Medium
Scale-up effort	Low	Low	Low
Machine learning support	Very good	Good	Medium

all the three frameworks, Spark, Flink, and Storm. However, fine-tuning each framework brings different considerations. Spark is very flexible and allows to fine-tune many aspects that require a deep knowledge of the framework's architecture. Such fine-tuning can require a large amount of effort. Also Flink requires some effort to tune up the configuration for better performance. This process can take some considerable amount of time, although we found to be simpler in some configuration aspects, as the flexibility of Spark can have drawbacks for the many parameters that can be configured. For Storm there are similar considerations to Flink in terms of configuration, understanding the architecture of the framework is essential for optimization, but also running some tests can give an evaluation of which parameters can give better results.

Effort required to scale-up the framework. Each of the three frameworks is relatively easy to scale-up to more nodes. For each framework is a matter of changes in the configuration and propagating the changes to the newly added nodes. The represented scenario can be scaled-up to use more nodes.

Machine learning libraries and algorithms supported. While in the scenario we used a simple anomaly detection algorithm, if advanced machine learning operations are necessary, Spark is the framework with the best support. Spark comes with a machine learning library called MLlib that provides common machine learning functionalities and multiple types of machine learning algorithms, such as classification, regression, clustering, etc. All of this is designed to distribute the computing across the cluster. FlinkML is a machine learning library for Flink. It provides algorithms for supervised and unsupervised learning, recommendations and more. The list of supported algorithms is still growing and there is an ongoing work in this area. Storm does not come with any machine learning library, but there is an ongoing work on third-party library called SAMOA, that adds machine learning support to Storm. SAMOA is currently undergoing incubation process in Apache Software Foundation and provides a collection of algorithms for most common data mining and machine learning tasks such as regression, classification, and clustering.

Threats to Validity. There are several threats to validity that we need to report. For internal validity, the configuration of the frameworks can have an impact on the results. We attempted to configure each framework for best efficiency based on the nodes configuration and resources available (mainly at the level of memory management, parallelism and processor setup), but an exhaustive search of all best

configurations would be unfeasible. Our scenario was more exploratory with respect to power consumption anomaly detection. A full experiment would need to take into account changes to parameters and the impact on the performance. For example, the number of deployed nodes alone can have a different impact on each of the considered frameworks. In our case, we kept a rather simple nodes topology, but for more complex topologies the results can be different (as previous research has shown, e.g., [37]). Another internal threat to validity is given by the implementation differences of the algorithm on each platform. Each framework provides different abstractions to develop applications and it is not possible to implement the algorithm equally, although we believe this threat is limited due to the simple anomaly detection algorithm we applied for benchmarking. Another threat is related to construct validity, the scenario was meant to compare the frameworks in a common data processing context and not to be a full experiment in which many factors are varied, like the number of nodes to which each framework is distributed. Another threat is related to generalization, the results apply to the specific scenario discussed to showcase the platform, other scenarios might have other needs and lead to different results.

Frameworks versions. In running the power consumption anomaly detection scenario, the following frameworks versions have been used:

Apache Zookeeper 3.4.12, Apache Kafka 2.0.0,
Apache Hadoop 2.8.5, Apache Flink 1.7.1,
Apache Spark 2.4.1, Apache Storm 1.2.2,
Scala 2.12, Java JDK 1.8.0201

VI. CONCLUSION

Big data processing in the Smart Grids context has many applications that require real-time operations and stream processing. In this paper, we presented a big data platform for anomaly detection from power consumption data. The platform is based on an ingestion layer with data densification, Apache Flink as part of the speed layer and HDFS/KairosDB as the data store. We mapped the main components to the reference architecture proposed by Pääkkönen and Pakkala [12], and provided the results of a scenario based on power consumption anomaly detection to assess the applicability of different frameworks: batch-based (Spark), stream-based (Storm), or hybrid (Flink). Overall, we adopted Flink in the platform's speed layer, as it provided the best performance for stream processing and met the requirements for power consumption datasets anomaly detection in our scenario.

Currently, we are planning to deploy the platform to analyze large-scale power consumption datasets in our running projects, by comparing several anomaly detection algorithms to help in better identifying clusters of customers based on smart metering data traces.

ACKNOWLEDGMENT

The work was supported from European Regional Development Fund Project *CERIT Scientific Cloud* (No.

CZ.02.1.01/0.0/0.0/16_013/0001802). Access to the CERIT-SC computing and storage facilities provided by the CERIT-SC Center, provided under the programme "Projects of Large Research, Development, and Innovations Infrastructures" (CERIT Scientific Cloud LM2015085), is greatly appreciated.

REFERENCES

- [1] Y. Demchenko, P. Grosso, C. De Laat, and P. Membrey, "Addressing big data issues in scientific data infrastructure," in *2013 International Conference on Collaboration Technologies and Systems (CTS)*. IEEE, 2013. doi: 10.1109/CTS.2013.6567203 pp. 48–55.
- [2] A. Radenski, T. Gurov, K. Kaloyanova, N. Kirov, M. Nisheva, P. Stanchev, and E. Stoimenova, "Big data techniques, systems, applications, and platforms: Case studies from academia," in *2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*. IEEE, 2016. doi: 10.15439/2016F91 pp. 883–888.
- [3] Y. Zhang, T. Huang, and E. F. Bompard, "Big data analytics in smart grids: A review," *Energy Informatics*, vol. 1, no. 1, p. 8, 2018. doi: 10.1186/s42162-018-0007-5
- [4] K. Zhou, C. Fu, and S. Yang, "Big data driven smart energy management: From big data to big insights," *Renewable and Sustainable Energy Reviews*, vol. 56, pp. 215–225, 2016. doi: 10.1016/j.rser.2015.11.050
- [5] B. Rossi and S. Chren, "Smart grids data analysis: A systematic mapping study," *arXiv preprint arXiv:1808.00156*, 2018. [Online]. Available: <https://arxiv.org/abs/1808.00156>
- [6] M. Ge, H. Bangui, and B. Buhnova, "Big data for internet of things: a survey," *Future Generation Computer Systems*, vol. 87, pp. 601–614, 2018. doi: 10.1016/j.future.2018.04.053
- [7] X. Liu and P. S. Nielsen, "Regression-based online anomaly detection for smart grid data," *arXiv preprint arXiv:1606.05781*, 2016.
- [8] B. Rossi, S. Chren, B. Buhnova, and T. Pitner, "Anomaly detection in smart grid data: An experience report," in *2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. IEEE, 2016. doi: 10.1109/SMC.2016.7844583 pp. 2313–2318.
- [9] Z.-H. Yu and W.-L. Chin, "Blind False Data Injection Attack Using PCA Approximation Method in Smart Grid," *IEEE Transactions on Smart Grid*, vol. 6, no. 3, pp. 1219–1226, 2015. doi: 10.1109/TSG.2014.2382714
- [10] J. Lee, Y.-c. Kim, and G.-L. Park, "An Analysis of Smart Meter Readings Using Artificial Neural Networks," in *Convergence and Hybrid Information Technology*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, vol. 7425, pp. 182–188.
- [11] F. McLoughlin, A. Duffy, and M. Conlon, "A clustering approach to domestic electricity load profile characterisation using smart metering data," *Applied energy*, vol. 141, pp. 190–199, 2015. doi: 10.1016/j.apenergy.2014.12.039
- [12] P. Pääkkönen and D. Pakkala, "Reference architecture and classification of technologies, products and services for big data systems," *Big data research*, vol. 2, no. 4, pp. 166–186, 2015. doi: 10.1016/j.bdr.2015.01.001
- [13] N. Marz and J. Warren, *Big Data: Principles and best practices of scalable real-time data systems*. New York; Manning Publications Co., 2015.
- [14] J. Lin, "The lambda and the kappa," *IEEE Internet Computing*, vol. 21, no. 5, pp. 60–66, 2017. doi: 10.1109/MIC.2017.3481351
- [15] S. Shahrivari, "Beyond batch processing: towards real-time and streaming big data," *Computers*, vol. 3, no. 4, pp. 117–129, 2014. doi: 10.3390/computers3040117
- [16] J. Kreps, "Questioning the lambda architecture," *Online article, July*, 2014. [Online]. Available: <https://www.oreilly.com/ideas/questioning-the-lambda-architecture>
- [17] M. Kiran, P. Murphy, I. Monga, J. Dugan, and S. S. Baveja, "Lambda architecture for cost-effective batch and speed big data processing," in *2015 IEEE International Conference on Big Data (Big Data)*. IEEE, 2015. doi: 10.1109/BigData.2015.7364082 pp. 2785–2792.
- [18] J. van Rooij, V. Gulisano, and M. Papatriantafyllou, "Locovolt: Distributed detection of broken meters in smart grids through stream processing," in *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems*. ACM, 2018. doi: 10.1145/3210284.3210298 pp. 171–182.

- [19] H. Sequeira, P. Carreira, T. Goldschmidt, and P. Vorst, "Energy cloud: Real-time cloud-native energy management system to monitor and analyze energy consumption in multiple industrial sites," in *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*. IEEE, 2014. doi: 10.1109/UCC.2014.79 pp. 529–534.
- [20] M. Mayilvaganan and M. Sabitha, "A cloud-based architecture for big-data analytics in smart grid: A proposal," in *2013 IEEE International Conference on Computational Intelligence and Computing Research*, Dec 2013. doi: 10.1109/ICCIC.2013.6724168 pp. 1–4.
- [21] A. A. Munshi and Y. A. I. Mohamed, "Data lake lambda architecture for smart grids big data analytics," *IEEE Access*, vol. 6, pp. 40463–40471, 2018. doi: 10.1109/ACCESS.2018.2858256
- [22] X. Liu and P. Nielsen, "Streamlining smart meter data analytics," in *Proceedings of the 10th Conference on Sustainable Development of Energy, Water and Environment Systems*. International Centre for Sustainable Development of Energy, Water and Environment Systems, 2015.
- [23] N. Balac, T. Sipes, N. Wolter, K. Nunes, B. Sinkovits, and H. Karimabadi, "Large scale predictive analytics for real-time energy management," in *2013 IEEE International Conference on Big Data*, Oct 2013. doi: 10.1109/BigData.2013.6691635 pp. 657–664.
- [24] A. R. Al-Ali, I. A. Zualkernan, M. Rashid, R. Gupta, and M. Alikarar, "A smart home energy management system using iot and big data analytics approach," *IEEE Transactions on Consumer Electronics*, vol. 63, no. 4, pp. 426–434, November 2017. doi: 10.1109/TCE.2017.015014
- [25] H. Daki, A. El Hannani, A. Aqqal, A. Haidine, and A. Dahbi, "Big data management in smart grid: concepts, requirements and implementation," *Journal of Big Data*, vol. 4, 12 2017. doi: 10.1186/s40537-017-0070-y
- [26] C. Yang, W. Chen, K. Huang, J. Liu, W. Hsu, and C. Hsu, "Implementation of smart power management and service system on cloud computing," in *2012 9th International Conference on Ubiquitous Intelligence and Computing and 9th International Conference on Autonomic and Trusted Computing*, Sep. 2012. doi: 10.1109/UIC-ATC.2012.160 pp. 924–929.
- [27] M. Ali, Z. Y. Dong, X. Li, and P. Zhang, "Rsa-grid: a grid computing based framework for power system reliability and security analysis," in *2006 IEEE Power Engineering Society General Meeting*, June 2006. doi: 10.1109/PES.2006.1709374. ISSN 1932-5517
- [28] T. Rajeev and S. Ashok, "A cloud computing approach for power management of microgrids," in *ISGT2011-India*, Dec 2011. doi: 10.1109/ISET-India.2011.6145354 pp. 49–52.
- [29] S. Chren, B. Rossi, and T. Pitner, "Smart grids deployments within eu projects: The role of smart meters," in *2016 Smart cities symposium Prague (SCSP)*. IEEE, 2016. doi: 10.1109/SCSP.2016.7501033 pp. 1–5.
- [30] M. Schvarcbacher, K. Hrabovská, B. Rossi, and T. Pitner, "Smart grid testing management platform (sgtmp)," *Applied Sciences*, vol. 8, no. 11, p. 2278, 2018. doi: 10.3390/app8112278
- [31] K. Hrabovská, N. Šimková, B. Rossi, and T. Pitner, "Smart grids and software testing process models," in *2019 Smart cities symposium Prague (SCSP)*. IEEE, 2019, pp. 1–5.
- [32] J. Peppanen, X. Zhang, S. Grijalva, and M. J. Reno, "Handling bad or missing smart meter data through advanced data imputation," in *IEEE Innovative Smart Grid Technologies Conference*. IEEE, 2016. doi: 10.1109/ISGT.2016.7781213 pp. 1–5.
- [33] X. Liu, N. Iftikhar, H. Huo, R. Li, and P. S. Nielsen, "Two approaches for synthesizing scalable residential energy consumption data," *Future Generation Computer Systems*, vol. 95, pp. 586–600, 2019. doi: 10.1016/j.future.2019.01.045
- [34] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin *et al.*, "Apache spark: a unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016. doi: 10.1145/2934664
- [35] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.
- [36] S. T. Allen, M. Jankowski, and P. Pathirana, *Storm Applied: Strategies for real-time event processing*. Manning Publications Co., 2015.
- [37] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl, "Benchmarking distributed stream processing engines," *arXiv preprint arXiv:1802.08496*, 2018.
- [38] Y. Wang, "Stream processing systems benchmark: Streambench," G2 Pro gradu, diplomityö, Aalto University, Finland, 2016-06-13. [Online]. Available: <http://urn.fi/URN:NBN:fi:aalto-201606172599>
- [39] M. A. Lopez, A. G. P. Lobato, and O. C. M. Duarte, "A performance comparison of open-source stream processing platforms," in *2016 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 2016. doi: 10.1109/GLOCOM.2016.7841533 pp. 1–6.
- [40] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng *et al.*, "Benchmarking streaming computation engines: Storm, flink and spark streaming," in *2016 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*. IEEE, 2016. doi: 10.1109/IPDPSW.2016.138 pp. 1789–1792.
- [41] Smart* data set for sustainability. [Online]. Available: <http://traces.cs.umass.edu/index.php/Smart/Smart>