Smart Meter Data Analytics: Systems, Algorithms, and Benchmarking

XIUFENG LIU, Technical University of Denmark
LUKASZ GOLAB, WOJCIECH GOLAB, IHAB F. ILYAS, and SHICHAO JIN,
University of Waterloo

Smart electricity meters have been replacing conventional meters worldwide, enabling automated collection of fine-grained (e.g., every 15 minutes or hourly) consumption data. A variety of smart meter analytics algorithms and applications have been proposed, mainly in the smart grid literature. However, the focus has been on what can be done with the data rather than how to do it efficiently. In this article, we examine smart meter analytics from a software performance perspective. First, we design a performance benchmark that includes common smart meter analytics tasks. These include offline feature extraction and model building as well as a framework for online anomaly detection that we propose. Second, since obtaining real smart meter data is difficult due to privacy issues, we present an algorithm for generating large realistic datasets from a small seed of real data. Third, we implement the proposed benchmark using five representative platforms: a traditional numeric computing platform (Matlab), a relational DBMS with a built-in machine learning toolkit (PostgreSQL/MADlib), a main-memory column store ("System C"), and two distributed data processing platforms (Hive and Spark/Spark Streaming). We compare the five platforms in terms of application development effort and performance on a multicore machine as well as a cluster of 16 commodity servers.

CCS Concepts: • **Information systems** → *Database performance evaluation*; *Data mining*

Additional Key Words and Phrases: Smart meters, data analytics, performance benchmarking, Hadoop, Spark

ACM Reference Format:
Xiufeng Liu, Lukasz Golab, Wojciech Golab, Ihab F. Ilyas, and Shichao Jin. 2016. Smart meter data analytics: Systems, algorithms, and benchmarking. ACM Trans. Database Syst. 42, 1, Article 2 (November 2016), 39 pages.
DOI: http://dx.doi.org/10.1145/3004295

1. INTRODUCTION

Smart electricity grids, which include renewable energy sources such as solar and wind and allow information sharing among producers and consumers, are beginning to replace conventional power grids worldwide. Smart electricity meters are a fundamental component of the smart grid, enabling automated collection of fine-grained (usually every 15 minutes or hourly) consumption data. This enables, among other things, dynamic electricity pricing strategies, in which consumers are charged higher prices during peak times to help reduce peak demand. Additionally, smart meter data analytics, which aim to help utilities and consumers understand electricity consumption patterns, has become an active area in research and industry. According to a recent

Authors' addresses: X. Liu, Technical University of Denmark, 2800 Kgs. Lyngby, Denmark; email: xiuli@dtu.dk; L. Golab, W. Golab, I. F. Ilyas, and S. Jin, University of Waterloo, 200 University Avenue West, Waterloo, Ontario, Canada N2L 3G1; emails: {lgolab, wgolab, ilyas, s37jin}@uwaterloo.ca.

DOI: http://dx.doi.org/10.1145/3004295

report, utility data analytics is already a $1 billion market and is expected to grow to nearly $4 billion by the year 2020.[1]

A variety of smart meter analytics algorithms have been proposed, mainly in the smart grid literature, to predict electricity consumption and enable accurate planning and forecasting, extract consumption profiles to provide personalized energy-saving tips to consumers, and design targeted engagement programs to clusters of similar consumers. However, the research focus has been on the insight that can be obtained from the data rather than performance and programmer effort. Implementation details were omitted, and the proposed algorithms were tested on small datasets. Moreover, a recent industry survey found that smart grid analytics are severely lacking [EPRI 2013]. Thus, despite the increasing amounts of available data and the increasing number of applications,[2] it is not clear how to build and evaluate a practical system for smart meter analytics. This is exactly the problem we study in this article.

1.1. Contributions

We begin with a *benchmark* for comparing the performance of smart meter analytics systems. Based on a review of prior work (details in Section 2), we identified five common tasks: (1) understanding the variability of consumers (e.g., by building histograms of their hourly consumption), (2) understanding the thermal sensitivity of buildings and households (e.g., by building regression models of consumption as a function of outdoor temperature), (3) understanding the typical daily habits of consumers (e.g., by extracting consumption trends that occur at different times of the day regardless of the outdoor temperature), (4) finding similar consumers (e.g., by running times series similarity search), and (5) detecting anomalies. These tasks involve aggregation, regression, time series analysis, and cross-checking new data against historical data; they include a mix of operators (aggregation, time series analysis, machine learning) and workloads (historical and online) that have been studied in isolation, but not in a single benchmark. Our benchmark includes representative algorithms from each of these sets, including an online anomaly detection framework that we propose.

Second, since obtaining smart meter data is difficult due to privacy concerns, we present a *data generator* for creating large realistic smart meter datasets from a small seed of real data. The generator includes several user-controlled parameters. For instance, it can create new datasets corresponding to consumers who are less or more "peaky" than those in the original sample. The real dataset we were able to obtain consists of over a year of data from 27,000 consumers, but our generator can create much larger datasets and allows us to stress-test the candidate systems.

Third, we implement the proposed benchmark using five state-of-the-art platforms that represent recent data management trends, including in-database machine learning, main-memory column stores, and distributed analytics. The five platforms are:

(1) Matlab: a numeric computing platform with a high-level language
(2) PostgreSQL: a traditional relational DBMS, plus MADlib [Hellerstein et al. 2012], an in-database machine-learning toolkit
(3) "System C": a main-memory column-store commercial system (the licensing agreement does not allow us to reveal the name of this system)
(4) Spark/Spark Streaming [Zaharia et al. 2010, 2012]: a main-memory distributed data processing platform
(5) Hive [Thusoo et al. 2009]: a Hadoop-based distributed data warehouse system

[1] http://www.greentechmedia.com/research/report/the-soft-grid-2013.
[2] See, for example, a recent competition sponsored by the United States Department of Energy to create new apps for smart meter data: http://appsforenergy.challengepost.com.

We report performance results on the real dataset and larger realistic datasets created by our data generator. Our main finding is that System C performs extremely well on our benchmark at the cost of the highest programmer effort: System C does not come with built-in statistical and machine-learning operators, which we had to implement from scratch in a nonstandard language. On the other hand, MADlib and Matlab make it easy to develop smart meter analytics applications, but they do not perform as well as System C. In cluster environments with very large data sizes, we found that Hive was only slightly slower than Spark for offline tasks; for online anomaly detection, Spark was faster than Hive when all the data it needed had been cached in memory and slower otherwise. Additionally, Spark and Hive are competitive with System C in terms of efficiency (throughput per server) for several of the workloads in our benchmark.

To summarize, we make four novel contributions in this article: (1) a benchmark for smart meter analytics, (2) a technique for generating very large realistic smart meter datasets, (3) a smart meter anomaly detection framework, and (4) an experimental evaluation of five data processing platforms using the proposed benchmark.

Our benchmark (i.e., the data generator and the tested algorithms) is freely available at https://github.com/xiufengliu. Due to privacy issues, we are unable to share the real dataset or the large synthetic datasets based on it. However, a smart meter dataset has recently become available at the Irish Social Science Data Archive[3] and may be used along with our data generator to create large publicly available datasets for benchmarking purposes.

1.2. Roadmap

The remainder of this article is organized as follows. Section 2 summarizes the related work; Section 3 presents the smart meter analytics benchmark; Section 4 discusses the data generator; Section 5 presents experimental results; and Section 6 concludes the article with directions for future work.

2. RELATED WORK

An earlier version of this article appeared at the EDBT 2015 conference [Liu et al. 2015a], where we proposed a benchmark for offline smart meter data analytics. In this submission, we added a new component of the benchmark to test online analytics. In particular, we proposed a framework for online anomaly detection in smart meter data and implemented and tested the framework using three different platforms. This new material is in the new Section 3.5 (description of the proposed framework), Section 5.2 (implementation details), and Sections 5.5, 5.6, and 5.7 (new benchmarking results). We also extended the data generator with the ability to control the "peakiness" of the generated time series (Section 4.1) and experimentally verified that it produces realistic data (new Section 4.2). Furthermore, a system that implements the algorithms in the proposed benchmark was demonstrated at the ICDE 2015 conference [Liu et al. 2015b].

2.1. Smart Meter Data Analytics

Two types of smart meter datasets have been studied: whole-house consumption readings collected by conventional smart meters (e.g., every hour) and high-frequency consumption readings (e.g., one per second), coming from the whole house or an individual circuit, obtained using specialized load-measuring hardware. We focus on the former, as these are the data that are currently collected by utilities.

For whole-house smart meter data feeds, there are two classes of applications: consumer and producer oriented. Consumer-oriented applications provide feedback to

[3]http://www.ucd.ie/issda/data/commissionforenergyregulationcer/.

end-users about reducing electricity consumption and saving money (see, e.g., Birt et al. [2012], Mattern et al. [2010], and Smith et al. [2012]). Producer-oriented applications are geared toward utilities, system operators, and governments and provide information about consumers such as their daily habits for the purposes of load forecasting and clustering/segmentation (see, e.g., Abreu et al. [2012], Albert et al. [2013], Albert and Rajagopal [2013b], Ardakanian et al. [2014], Chicco et al. [2006], Espinoza et al. [2005], Figueiredo et al. [2005], Ghofrani et al. [2011], Nezhad et al. [2014], and Rasanen et al. [2010]).

From a technical standpoint, both of these classes of applications perform two types of operations: extracting representative features (see, e.g., Ardakanian et al. [2014], Birt et al. [2012], Espinoza et al. [2005], and Figueiredo et al. [2005]) and finding similar consumers based on the extracted features (see, e.g., Abreu et al. [2012], Chicco et al. [2006], Rasanen et al. [2010], Smith et al. [2012], and Tsekouras et al. [2007]). Household electricity consumption can be broadly decomposed into the temperature-sensitive component (i.e., heating and air conditioning) and the temperature-insensitive component (other loads and appliances). Thus, representative features include those which measure the effect of outdoor temperature on consumption [Albert and Rajagopal 2013a; Birt et al. 2012; Rasanen et al. 2010] and those which identify consumers' daily habits regardless of temperature [Abreu et al. 2012; Ardakanian et al. 2014; Espinoza et al. 2005], as well as those which measure the overall variability (e.g., consumption histograms) [Albert et al. 2013]. Our smart meter benchmark, which will be described in Section 3, includes representative algorithms for characterizing consumption variability, temperate sensitivity, daily activity, and similarity to other consumers.

Furthermore, we distinguish between offline and online analytics. The algorithms described previously build prediction models and extract useful features offline. There are also online algorithms for identifying anomalies in smart meter data [Chen and Cook 2011; Mashima and Cardenas 2012], which may run every hour as new data arrive or perhaps once a day. Again, these algorithms can be divided into consumer and utility oriented. Consumer-oriented anomaly detection includes daily alerts if a household's monthly bill is on track to be much higher than average. Utility-oriented anomaly detection may include detecting electricity theft [Mashima and Cardenas 2012]. In Section 3, we propose a distance-based outlier detection framework for smart meter data that generalizes existing techniques and supports additional functionalities such as explaining why a certain measurement was determined to be an outlier.

We also point out recent work on smart meter data quality (specifically, handling missing data) [Jeng et al. 2013], symbolic representation of smart meter time series [Eichinger et al. 2015; Wijaya et al. 2013], and smart meter data privacy (see, e.g., Acs and Castelluccia [2011], Buchmann et al. [2013], and Kessler et al. [2015]). These important issues are orthogonal to smart meter analytics, which is the focus of this article.

2.2. Systems and Platforms for Smart Meter Data Analytics

Traditional options for implementing smart meter analytics include statistical and numeric computing platforms such as R and Matlab. As for relational database systems, two important technologies are main-memory databases, such as "System C" in our experiments, and in-database machine learning, for example, PostgreSQL/MADlib [Hellerstein et al. 2012]. Finally, a parallel data processing platform such as Hadoop or Spark is an interesting option for cluster environments. We have implemented the proposed benchmark in systems from each of these classes (details in Section 5).

Smart meter analytics software is currently offered by several database vendors including SAP[4] and Oracle/Data Raker,[5] as well as startups such as Autogrid.com, C3Energy.com, and OPower.com. Additionally, SQLstream[6] describes a solution for real-time analytics and alerting. However, it is not clear what algorithms are implemented by these systems and how.

There has also been some recent work on efficient retrieval of smart meter data stored in Hive [Liu et al. 2014], but that work focuses on simple operational queries rather than the deep analytics that we address in this article.

2.3. Benchmarking Data Analytics

There exist several databases (e.g., TPC-C, TPC-H, and TPC-DS) and big data[7] benchmarks, but they focus mainly on the performance of relational queries (and/or transactions) and therefore are not suitable for smart meter data mining. Benchmarking time series data mining was discussed in Keogh and Kasetty [2003]. Different implementations of time series similarity search, clustering, classification, and segmentation were evaluated. While some of these operations are relevant to smart meter analytics, there are other important tasks such as extracting consumption profiles that were not considered in Keogh and Kasetty [2003]. Additionally, Keogh and Kasetty [2003] evaluated standalone algorithms, whereas we evaluate data analytics platforms. Furthermore, Anil et al. [2013] benchmarked data mining operations for power system analysis. However, their focus was on analyzing voltage measurements from power transmission lines, not smart meter data, and therefore the tested algorithms were different from ours. Finally, Arlitt et al. [2015] propose a benchmark for smart meters that focuses on routine computations such as finding top customers and calculating monthly bills. In contrast, our work aims to discover more complex patterns in energy data. Their workload generator uses a Markov chain model that must be trained using a real dataset.

We also note that TPC benchmarks include the ability to generate very large synthetic databases, and there has been research on synthetic database and data stream generation (see, e.g., Bruno and Chaudhuri [2005] and Gu et al. [2015]), but we are not aware of any previous work on generating realistic smart meter data.

3. THE BENCHMARK

In this section, we propose a performance benchmark for smart meter analytics. The main goal of the benchmark is to measure the running time of a set of tasks that will be defined shortly. We consider two types of tasks: offline and online. For offline tasks, the input consists of n time series, each corresponding to one electricity consumer, in one or n text files. We assume that each time series contains hourly electricity consumption measurements (in kilowatt-hours, kWh) for a year, that is, $365 \times 24 = 8,760$ data points. For each consumption time series, we require an accompanying external temperature time series, also with hourly measurements. For online tasks, the input consists of a sequence of 30 files, each containing 1 day of data, that is, 24 hourly consumption measurements and 24 hourly external temperature measurements, for all n consumers. Additionally, online tasks are assumed to have access to the offline (historical) dataset as well as the models built by offline tasks (we will explain shortly why this is required).

For offline tasks, we measure the running time on the input dataset, both with a cold start (data on disk) and a warm start (data loaded into physical memory). For online

[4]http://www.sap.com/pc/tech/in-memory-computing-hana/software/smart-meter-analytics/index.html.
[5]http://www.oracle.com/us/products/applications/utilities/meter-data-analytics/index.html.
[6]http://www.sqlstream.com/solutions/smart-grid.
[7]https://amplab.cs.berkeley.edu/benchmark.

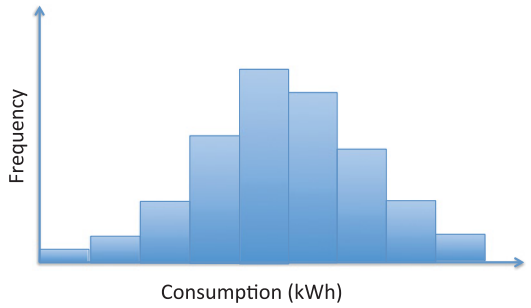


Fig. 1. Example of a consumption histogram.

tasks, we measure the time it takes to load new data and the running time of the task itself.

Utility companies may have access to additional data about their customers, for example, location, square footage of the home, or family size. However, this information is usually not available to third-party applications. Thus, the input to our benchmark is limited to smart meter time series and publicly available weather data.

We now discuss the five analysis tasks included in the proposed benchmark. The first four are offline: consumption histograms, thermal sensitivity, daily profiles, and similarity search. The last one, anomaly detection, is online.

3.1. Consumption Histograms

The first task is to understand the variability of each consumer. To do this, we compute the distribution of hourly consumption for each consumer via a histogram, an example of which is shown in Figure 1. The x-axis in the histogram denotes various hourly consumption ranges, and the y-axis is the frequency, that is, the number of hours in the year whose electricity consumption falls in the given range. For concreteness, in the proposed benchmark, we specify the histograms to be equi-width (rather than equi-depth) and we always use 10 buckets.

3.2. Thermal Sensitivity

The second task is to understand the effect of outdoor temperature on the electricity consumption of each household. The simplest approach is to fit a least-squares regression line to the consumption-temperature scatter plot. However, in climates with a cold winter and warm summer, electricity consumption rises when the temperature drops in the winter (due to heating) and also rises when the temperature rises in the summer (due to air conditioning). Thus, a piecewise linear regression model is more appropriate.

We selected a recent algorithm from Birt et al. [2012] for the benchmark, which we refer to as the three-line algorithm. Consider a consumption-temperature scatter plot for a single consumer shown in Figure 2. The actual data points are not shown, but a point on this plot would correspond to a particular hourly consumption value and the outdoor temperature at that hour. The upper three lines correspond to the piecewise regression lines computed only for the points in the 90th percentile for each temperature value, and the lower three lines are computed from the points in the 10th percentile for each temperature value. Thus, for each time series, the algorithm starts by computing the 10th and 90th percentiles for each temperature value and then computes the two sets of regression lines. In the final step, the algorithm ensures that the three lines are not discontinuous and therefore it may need to adjust the lines slightly.

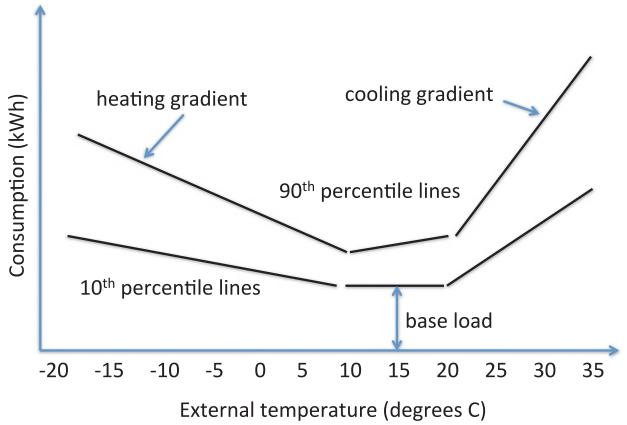


Fig. 2. Example of the three-line regression model.

As shown in Figure 2, the three-line algorithm extracts useful information for customer feedback. For instance, the slopes (gradients) of the left and right 90th percentile lines correspond to heating and cooling sensitivity, respectively. A high cooling gradient might indicate an inefficient air conditioning system or a low air conditioning set point. Additionally, the height at the lowest point on the 10th percentile lines indicates *base load*, which is the electricity consumption of appliances and devices that are always on regardless of the temperature (e.g., a refrigerator, a dehumidifier, or a home security system).

3.3. Daily Profiles

The third task is to extract daily consumption trends that occur regardless of the outdoor temperature. For this, we use the periodic autoregression (PAR) algorithm for time series data from Ardakanian et al. [2014] and Espinoza et al. [2005]. The idea behind this algorithm is illustrated in Figure 3. At the top, we show a fragment of the hourly consumption time series for some consumer over a period of several days. We are only given the total hourly consumption, but the goal of the algorithm is to determine, for each hour, how much load is temperature independent and how much additional load is due to temperature (i.e., heating or cooling). Once this is determined, the algorithm computes the average temperature-independent consumption at each hour of the day, illustrated at the bottom of Figure 3. Thus, for each consumer, the daily profile consists of a vector of 24 numbers, denoting the expected consumption at different hours of the day due to the occupants' daily habits and not affected by temperature.

Formally, let y_d^h and t_d^h be the electricity consumption of a particular household and external temperature, respectively, during hour h on day d. The PAR model assumes that the electricity consumption of a household at a particular hour of the day is a linear combination of its consumption at that hour over the previous p days (we use $p = 3$, as in Ardakanian et al. [2014]) and the outdoor temperature. That is, for each h from 1 to 24, $y_d^h = \alpha_1 y_{d-1}^h + \alpha_2 y_{d-2}^h + \alpha_3 y_{d-3}^h + \alpha_4 t_d^h$, where the α_is are coefficients determined by the PAR algorithm.[8] The algorithm groups the input dataset by household and by hour of the day, and, for each household-hour group, computes the α_i coefficients using least-squares auto-regression. Thus, each household is associated with 24 sets

[8]We have shown a simple form of this equation with only one temperature term. The model we use has three temperature variables to model the piecewise linear relationship between temperature and consumption shown in Figure 2.

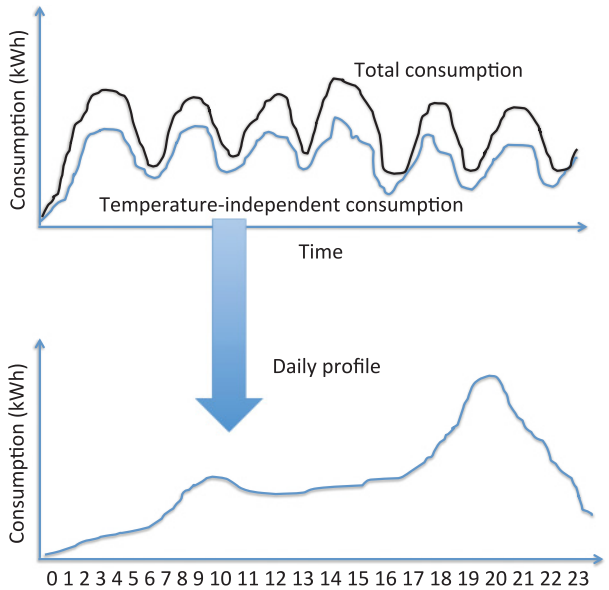


Fig. 3. Example of a daily profile.

of α_i coefficients, one set for each hour of the day. Finally, to compute the daily profile for a given household, we scan its consumption and temperature time series again, and, for each consumption data point, we subtract the $\alpha_4 t_d^h$ term from it to obtain the temperature-independent consumption component (i.e., the blue curve at the top of Figure 3). We then group the temperature-independent consumption numbers by hour of the day and take the average, obtaining 24 data points that can be plotted as in Figure 3.

We include both the PAR algorithm and the three-line algorithm (see Section 3.2) in the benchmark despite the fact that both algorithms compute, to some extent, a household's thermal sensitivity. The primary rationale behind this decision is that PAR and three-line achieve this at different levels of detail. PAR is designed to extract daily profiles, but it also captures a household's thermal sensitivity through the α_4 coefficient and, more generally, through the autoregression coefficients associated with exogenous temperature variables. On the other hand, three-line provides a finer level of detail by computing suitable switching points between the individual regression lines, which correspond to heating and cooling setpoints, and which PAR assumes to be part of the input (recall footnote 8). Furthermore, since the cooling and heating gradients in three-line (recall Figure 2) are only based on electricity consumption values above the 90th percentile, they provide a better indication of the occupants' thermal sensitivity: datapoints with high consumption values correspond to times when the occupants are at home and active. Another reason to include both PAR and three-line in the benchmark is that they represent different workloads, one using autoregression and the other using ordinary linear regression.

3.4. Similarity Search

The next task is to find groups of similar consumers. Customer segmentation is important to utilities: for instance, they can determine how many distinct groups of customers there are and design targeted energy-saving campaigns for each group. Rather than choosing a specific clustering algorithm for the benchmark, we include a more general

task: for each of the n time series given as input, we compute the top-k most similar time series (we use $k = 10$). The similarity metric we use is *cosine similarity*. Let X and Y be two time series. The cosine similarity between them is defined as their dot product divided by the product of their vector lengths, that is, $\frac{X \cdot Y}{||X|| * ||Y||}$.

3.5. Online Anomaly Detection

The final algorithm, and the only online algorithm in our benchmark, performs anomaly detection. The previous four algorithms build consumption models and extract consumption patterns from a historical dataset. In contrast, anomaly detection is performed on 1 day of data at a time. Next, we present an anomaly detection framework for smart meter data and discuss how we implemented the framework in the proposed benchmark.

3.5.1. Anomaly Detection Framework. We focus on two categories of anomalies. If the electricity consumption of a particular household today is significantly different from its typical consumption in the past, we flag a *self-anomaly*. If the consumption today is significantly different from the average consumption within the household's group today, we flag a *group anomaly*. In this article, we define a group in a spatial sense as the neighborhood of the given household. However, there are other reasonable definitions, such as clustering the households based on their daily profiles, with each cluster forming a group of households that have similar daily habits. From the point of view of a performance benchmark, the precise definition of a group is not relevant: we assume that we are given a dataset containing each household ID and its group number.

The idea is illustrated in Algorithm 1, which runs at the end of each day. In line 2, $\vec{y}$ contains the 24 hourly consumption measurements for a particular household, call it s, for the current day. In line 3, we compute 24 predicted consumption values for s, one for each hour (we will discuss how to do this shortly). In line 4, we compute the average consumption for each hour of the current day for all the households belonging to the same group as s. Then, in lines 5 and 8, we check for self- and group anomalies, respectively, by computing the distance (i.e., the norm of the difference vector) between the current day's consumption vector $\vec{y}$ and the predicted consumption vector and current day's average group consumption, respectively. We will discuss how to compute the anomaly thresholds τ_s and τ'_s shortly.

In this article, we assume that anomaly detection runs every day over the last 24 hourly measurements. However, Algorithm 1 can easily be adjusted to run at higher frequencies if higher-frequency data become available, or to use a sliding window of the most recent values rather than running at the end of each hour or day. If using a sliding window, it may help to precompute and store $y^i - \hat{y}^i$ and $y^i - g^i$ for each time point i within the current window. This will allow $\|\vec{y} - \vec{\hat{y}}\|$ and $\|\vec{y} - \vec{g}\|$ to be recomputed incrementally when the window slides.

3.5.2. Instantiation of Anomaly Detection Framework. We implemented line 3 of Algorithm 1 using the PAR model from Section 3.3. To predict the hourly electricity consumption for a day for a given household, we need the following information: the α_i coefficients of the PAR model for this household, the most recent 3 days of consumption data for this household (i.e., the y^h_{d-j} terms in the PAR model), and the temperature time series during the 24 hours being predicted. For a particular hour h of day d, we simply plug in the temperature and the consumption at that hour over the past 3 days into the PAR equation. We get $\hat{y}^h_d = \alpha_1 y^h_{d-1} + \alpha_2 y^h_{d-2} + \alpha_3 y^h_{d-3} + \alpha_4 t^h_d$. Repeating for each of the 24 hours of the day, we assemble the predicted consumption vector $\vec{y} = <\hat{y}^1_d, \hat{y}^2_d, \ldots, \hat{y}^{24}_d>$.

Figure 4 summarizes our anomaly detection algorithm. At the end of each day, new consumption and temperature data arrive. For each household, we compare its actual

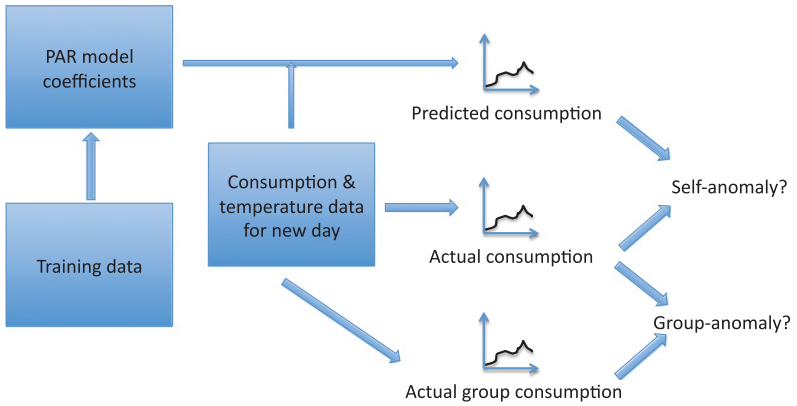


Fig. 4. Overview of anomaly detection.

ALGORITHM 1: Anomaly Detection Framework

1 **foreach** household s **do**
2 Let $\vec{y} = <y^1, y^2, \ldots, y^{24}>$ be the actual hourly consumption of s for the current day
3 Compute $\vec{\hat{y}} = <\hat{y}^1, \hat{y}^2, \ldots, \hat{y}^{24}>$, the predicted hourly consumption of s for the current day
4 Compute $\vec{g} = <g^1, g^2, \ldots, g^{24}>$, the average hourly consumption of s's group for the current day
5 **if** $\|\vec{y} - \vec{\hat{y}}\| > \tau_s$ **then**
6 Output "Self-anomaly for household s"
7 **end**
8 **if** $\|\vec{y} - \vec{g}\| > \tau'_s$ **then**
9 Output "Group anomaly for household s"
10 **end**
11 **end**

consumption over the current day with (1) its predicted consumption based on its PAR model and today's temperature, and (2) the average consumption in its group over the current day.

We now explain how we obtain the thresholds τ_s and τ'_s for each household s. For τ_s, that is, the self-anomaly threshold, we go back to the historical consumption dataset that was used to train the PAR model, and, for each day in this dataset, we compute the predicted consumption as earlier. For each day in the historical dataset, we then compute the distance between the actual consumption and predicted consumption as in line 5 of the algorithm. Finally, we set τ_s to be the mean plus three standard deviations of these differences. Thus, a self-anomaly is flagged only if the current day's predicted consumption deviates from the actual consumption by much more than the natural variation in the training dataset.

We use a similar methodology for τ'_s, but here we do not need to use the PAR model. We go back to the historical consumption dataset, and, for each day in this dataset, we compute the distance between the actual consumption of a given household s and the average consumption of s's group, as in line 8 of the algorithm. Then, we set τ'_s to be the mean plus three standard deviations of these differences. Thus, a group anomaly is flagged only if the current day's actual consumption of s deviates from its group's average consumption much more than it did in the training dataset.

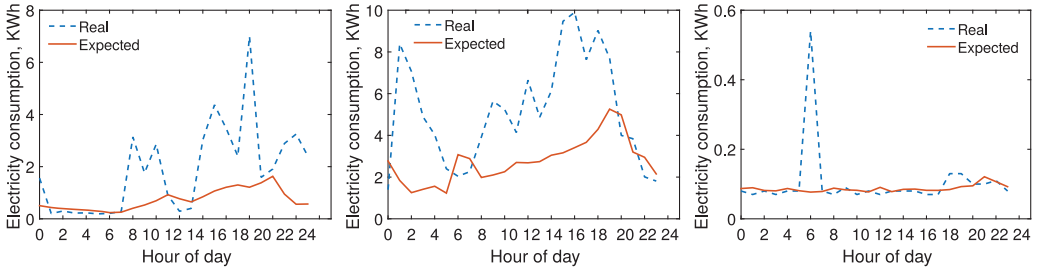


Fig. 5. Three examples of anomalies detected by our self-anomaly algorithm.

Different households may have different anomaly thresholds. For instance, a household s with regular daily habits can have its consumption easily predicted by the PAR model and therefore τ_s will be small. On the other hand, a household s without regular habits will have large differences between actual and predicted consumption in the training data and therefore a larger τ_s. Similarly, if a particular household s has similar habits to the others within its group/neighborhood, its τ'_s will be low. However, if the occupants of s work nightly shifts but those in the other households in this group work regular hours, then we will likely see large differences between s's consumption and its group's average consumption in the training dataset, leading to a larger value of τ'_s. In fact, if the value of τ'_s ends up being very large, we may turn off group anomalies for this household to avoid false alarms: this household is naturally different from the others in its group.

It is important to note that we will not measure the time it takes to determine the anomaly thresholds. We assume that the PAR models and the thresholds have already been computed and are stored in a file or a table. We will only measure the loading time of new data, 1 day at a time, and the running time of the anomaly detection algorithm.

Finally, we note that the proposed anomaly detection framework provides a way to impute missing data. If a particular time series is missing a consumption value for a particular hour of the day, we can estimate this value from the average consumption at that hour of the day over the past 3 days (and the corresponding temperature time series to account for changes in temperature).

3.5.3. Explanations of Anomalies. In addition to returning binary decisions in lines 6 and 9, our framework can be extended to provide *explanations* of the detected anomalies. In the benchmark, we implemented the following simple explanation algorithm for each detected anomaly. First, we divide a day into three 8-hour periods: night (midnight till 8 a.m.), day (8 a.m. till 4 p.m.), and evening (4 p.m. till midnight). Then, we check whether the actual consumption of a given household was lower or higher than predicted consumption (for self-anomalies) or average group consumption (for group anomalies) by at least 10% during each of the three periods. Based on this, we report explanations of the form "higher than normal consumption throughout the day," "higher than normal consumption in the evening," "higher than normal consumption at night but lower than normal consumption during the day," and so forth.

3.5.4. Examples of Anomalies. In Figure 5, we show three examples of self-anomalies that our algorithm found in the real dataset. Dotted lines correspond to actual consumption throughout a particular day and solid lines correspond to predicted consumption for that household and that day. In the first example, nightly consumption is normal, but day and evening consumption is higher. In the second example, night and day consumption is higher but evening consumption is not. The last example is

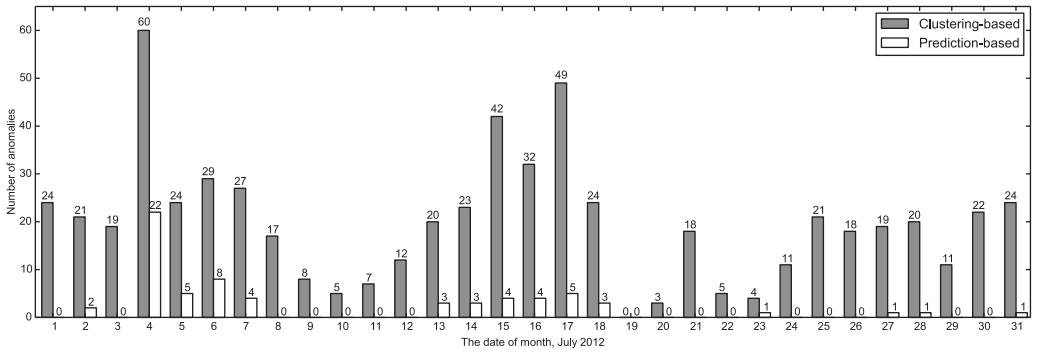


Fig. 6. Anomalies flagged by our algorithm (Prediction-based) and the Clustering-based algorithm in the month of July.

interesting: only the 6 a.m. consumption is much higher than predicted. Since the focus of this article is on performance benchmarking rather than anomaly detection itself, we leave a further investigation of anomaly/outlier detection and explanation in smart meter data for future work.

3.5.5. Comparison to Existing Approaches. We now compare our anomaly detection framework to a clustering-based approach for outlier detection in energy consumption from Chen and Cook [2011]. In their approach, the first training step is to cluster all the input vectors $\vec{y}$ provided in the training dataset, each corresponding to 24 hourly consumption values of a particular household for a particular day. Next, for each cluster, the algorithm computes the mean and the standard deviation of the distances between the cluster centroid and each vector belonging to this cluster. Assuming that the distances are normally distributed, anomaly detection proceeds as follows. When the data for a new day arrive, the closest cluster centroid to the new day's consumption vector is identified. Then, the algorithm computes the distance to this centroid, and, using the probability density function for a normal distribution, it computes the probability that this distance value comes from the distribution empirically obtained from the training dataset. If this probability is below some threshold, then an anomaly is flagged.

A labeled dataset containing true anomalies is required to evaluate the accuracy of any anomaly detection technique. In the absence of such a dataset, we trained our algorithm and the clustering-based algorithm from Chen and Cook [2011] on our real dataset, from January till June, and used both algorithms to identify self-anomalies in July. Figure 6 plots the number of anomalies found by both algorithms (ours is labeled "Prediction-based") and shows that the clustering-based algorithm labels many more consumption vectors as anomalous. For this experiment, the threshold value for the clustering algorithm was extremely small: 10^{-20} compared to 1.5×10^{-3} for our algorithm, which corresponds to three standard deviations for a normal distribution, and larger threshold values produced orders of magnitude more anomalies for the clustering algorithm. Upon further inspection, we found that several days in July had very high temperature, and the clustering algorithm considered many households during those days as anomalies. On the other hand, our algorithm, which is based on the PAR model, which focuses on daily habits independently of temperature, did not. Thus, we believe that many anomalies found by the clustering algorithms were false alarms indicating anomalous weather patterns rather than anomalous customer behavior.

3.6. Discussion

The proposed benchmark consists of (1) consumption histograms, (2) the three-line algorithm for understanding the effect of external temperature on consumption, (3) the periodic autoregression (PAR) algorithm to extract typical daily profiles, (4) time series similarity search to find similar consumers, and (5) anomaly detection. The first three algorithms analyze the electricity consumption of each household in terms of its distribution, its temperature sensitivity, and its daily patterns. The fourth algorithm finds similarities among different consumers. The fifth algorithm examines 1 day of data at a time and compares each household's consumption patterns from that day to those in the past and to those of its group/neigborhood. While many more smart meter analytics algorithms have been proposed, we believe the five tasks we have chosen accurately represent a variety of fundamental computations that might be used to extract insight from smart meter data, both offline and online.

In terms of computational complexity, the first three algorithms perform the same task for each consumption time series and therefore can be parallelized easily, while similarity search has quadratic complexity with respect to the number of time series. Computing histograms requires grouping the time series according to consumption values. The three-line algorithm additionally requires grouping the data by temperature and requires statistical operators such as quantiles and least-squares regression lines. The PAR and similarity algorithms require time series operations. Finally, anomaly detection requires vector and matrix operations, and lookups of historical data and model parameters. Thus, the proposed benchmark tests the ability to sequentially scan the data or extract different subsets of the data, and run various statistical and time series operations.

4. THE DATA GENERATOR

4.1. Algorithm Description

Recall from Section 3 that the proposed benchmark requires n time series as input, each corresponding to an electricity consumer. Testing the scalability of a system therefore requires running the benchmark with increasing values of n. Since it is difficult to obtain large amounts of smart meter data due to privacy issues, and since using randomly generated time series may not give accurate results, we propose a data generator for realistic smart meter data.

The intuition behind the data generator is as follows. Since electricity consumption mainly depends on external temperature and daily activity, we start with a small seed of real data and we generate the daily activity profiles (recall Figure 3) and temperature regression lines (recall Figure 2) for each consumer therein. To generate a new time series, we take the daily activity pattern from a randomly selected consumer in the real dataset and the temperature dependency from another randomly selected consumer, and we add some white noise. Thus, we first disaggregate the consumption time series of existing consumers in the seed dataset, and we then reaggregate the different pieces in a new way to create a new consumer. This gives us a realistic new consumer whose electricity usage combines the characteristics of multiple existing consumers.

Figure 7 illustrates the proposed data generator. As a preprocessing step, we use the PAR algorithm from Espinoza et al. [2005] to generate daily profiles for each consumer in the seed dataset. We then run the k-means clustering algorithm (for some specified value of k, the number of clusters) to group consumers with similar daily profiles. This gives us k cluster centroids, one for each type of daily profile. We also run the three-line algorithm and record the heating and cooling gradients for each consumer. The preprocessing step only needs to run once.

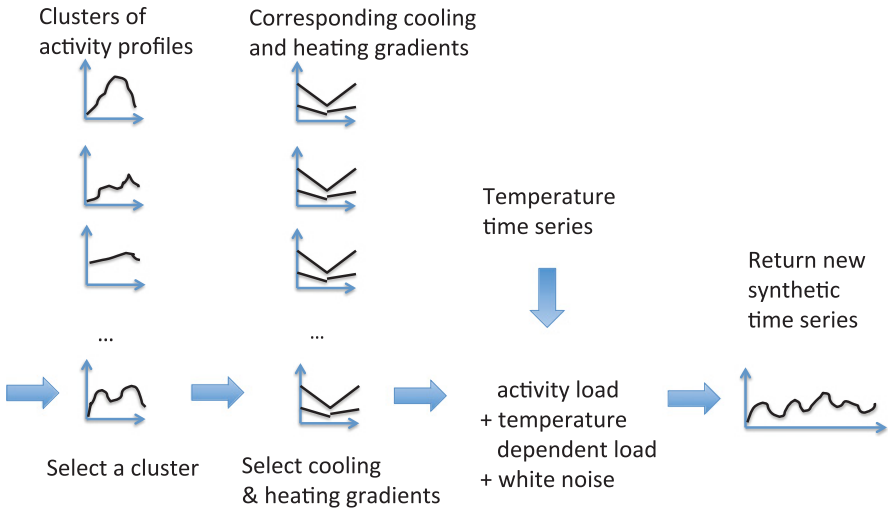


Fig. 7. Illustration of the proposed data generator.

Now, creating a new time series based on the clusters computed in the preprocessing step proceeds as follows. We randomly select an activity profile cluster and use the cluster centroid to obtain the hourly consumption values corresponding to daily activity load. Next, we randomly select an individual consumer from the chosen cluster and we obtain its cooling and heating gradients. We then need to input a temperature time series for the new consumer and we have all the information we need to create a new consumption time series.[9] Each hourly consumption measurement of the new time series is generated by adding together (1) the daily activity load for the given hour, (2) the temperature-dependent load computed by multiplying the heating or cooling gradient by the given temperature value at that hour, and (3) a Gaussian white noise component with some specified standard deviation σ.

The data generator also includes a user-controlled *peak factor* parameter that determines the sharpness of the peaks in the generated consumption time series. Given a generated time series, we identify each peak, defined as at least three consecutive consumption values that are higher than the consumption immediately before and immediately after. Then, for each such peak, we increase the peak-to-average ratio by the peak factor. Thus, a peak factor of one does not change anything, whereas a peak factor of, say, five, creates significantly more pronounced peaks, as shown in Figure 8 (with hour of the day on the x-axis). The user can also specify a peak factor range, and the data generator will randomly select a peak factor from this range for each generated time series. This allows us to produce new datasets in which customers have similar habits to those in the underlying real dataset, but some have significantly higher activity load during peak times.

We implemented the data generator in Spark. The preprocessing step runs first, and its results are written to HDFS and replicated to each node in the cluster. Afterward, data generation proceeds in parallel without any communication among nodes. For load balancing, we configure the Spark job that generates new data such that each processing node produces the same number of new time series.

[9]In our experiments, we used the temperature time series corresponding to the southern Ontario city from which we obtained the real dataset.

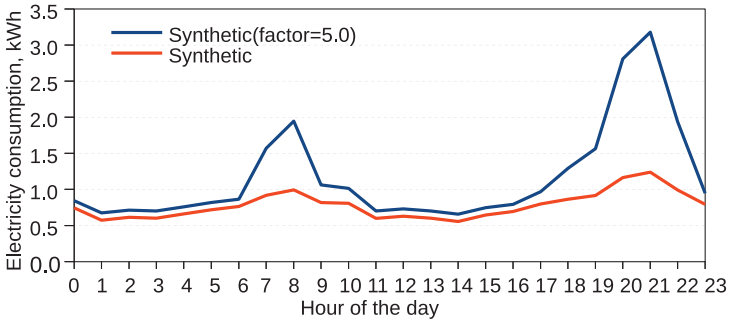
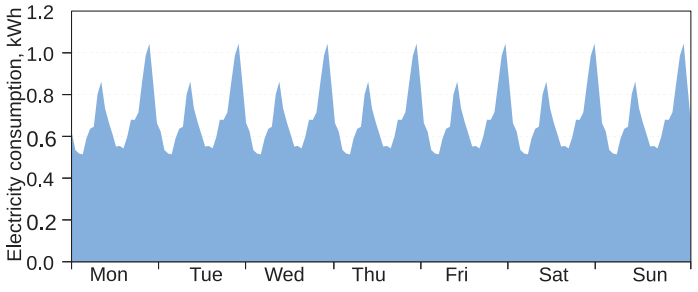
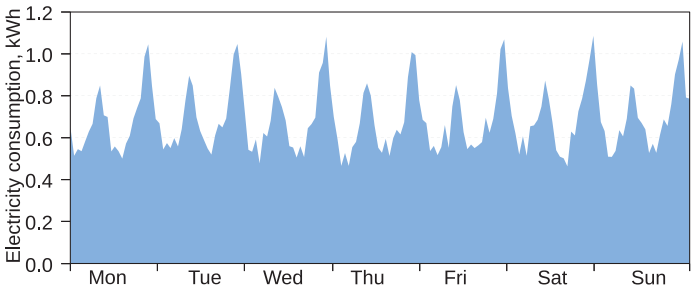


Fig. 8. Effect of setting the peak factor to five.



(a) Real



(b) Generated

Fig. 9. Daily and weekly consumption patterns in the real (a) and generated (b) datasets.

4.2. Validation and Efficiency

We now show that the generator produces realistic data and comment on its performance. We use $k = 4$ as the number of clusters of activity profiles, and we use a randomly selected sample of 10% of our real dataset, or about 2,700 customers. We also generate the same number, or about 2,700, of synthetic customers. First, Figure 9(a) shows the aggregate consumption within our real dataset as a function of day of the week, and Figure 9(b) shows the same for a dataset produced by our generator. Consumption patterns, such as morning and afternoon peaks occurring every day, are preserved.

Next, we randomly select a new time series produced using our generator, record the activity profile that was used to generate it (recall Figure 7), and compare its consumption histogram to that of the average consumption of (real) customers from this activity profile cluster. Figure 10 shows the results, with real consumption on the

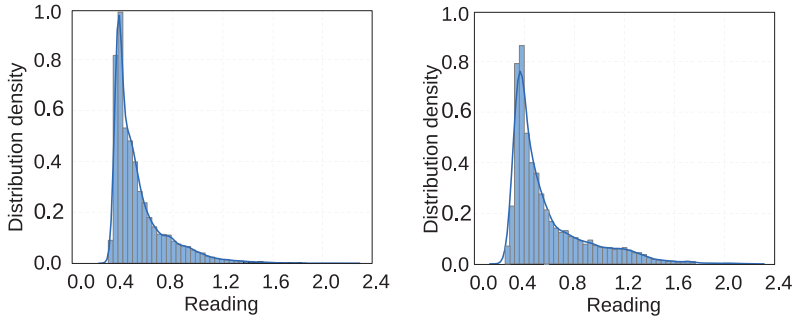


Fig. 10. Distribution of hourly consumption of real customers (a) and a corresponding synthetically generated customer (b).

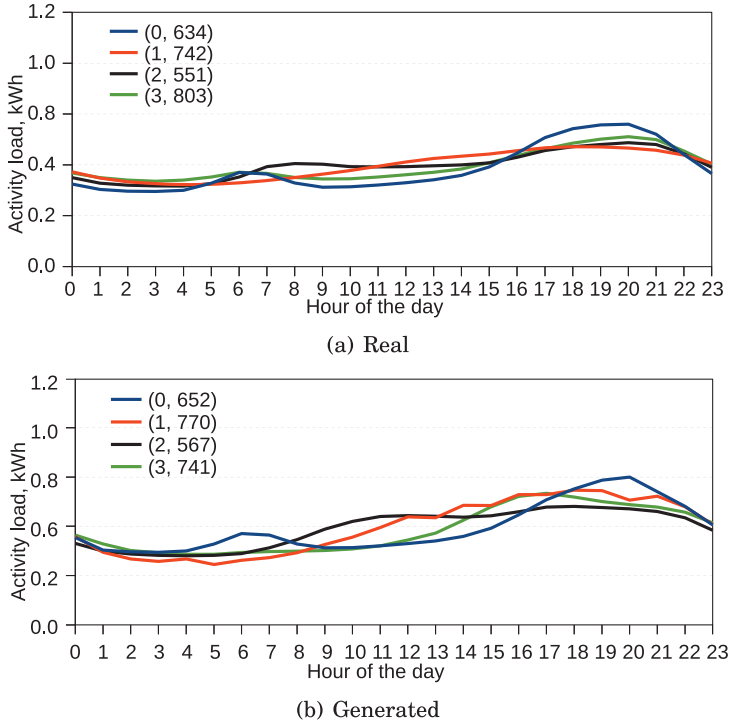
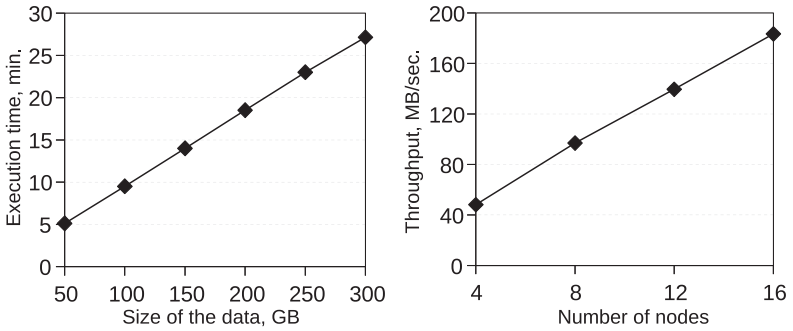


Fig. 11. Cluster centroids of daily activity load profiles in the real (a) and generated (b) datasets.

left and generated consumption on the right. Hourly consumption ranges are shown on the x-axis and frequency on the y-axis. The distributions are very similar.

In the final experiment demonstrating that our generator produces realistic data, we compare the centroids of activity clusters in the real dataset (which we have computed in the preprocessing step of data generation) with those in the generated dataset (which we have computed by rerunning k-means clustering on the generated time series). Figure 11(a) shows the four activity profile centroids in the real data, with hour of the day on the x-axis. The clusters are numbered 0 through 3, with the legend showing how many customers belong to each cluster. The corresponding centroids in



(a) Using all 16 nodes to generate datasets with various sizes

(b) Generating a 300GB dataset using 4 to 16 nodes

Fig. 12. Scalability of the data generator.

the generated dataset, along with the number of customers per cluster, are shown in Figure 11(b). We conclude that the generated dataset contains similar activity profile clusters to the real dataset.

Finally, we demonstrate linear scalability of our Spark implementation of the data generator. We use a 16-node cluster (full details in Section 5.1) to generate datasets sized between 50 and 300GB. Figure 12(a) shows that execution time increases linearly as we increase the size of the dataset to be generated (using all 16 nodes), and that it takes roughly 10 minutes to generate 100GB of data. Figure 12(b) shows that throughput increases linearly as we increase the number of nodes (to generate a 300GB dataset).

5. BENCHMARKING RESULTS

This section presents our benchmarking results. We start with a description of the experimental environment (Section 5.1) and an overview of the five platforms in which we implemented the proposed benchmark (Section 5.2). We then evaluate the offline algorithms. Section 5.3 discusses our findings using a single multicore server, including the effect of data layout and partitioning (Section 5.3.1), the relative cost of data loading versus query execution (Section 5.3.2), and the performance of single-threaded and multithreaded execution (Section 5.3.3 and 5.3.4, respectively). In Section 5.4, we investigate the performance of Spark and Hive on a cluster of 16 worker nodes. Next, we experiment with online anomaly detection, with data loading times presented in Section 5.5, single-server results in Section 5.6, and cluster results in Section 5.7. We conclude with a summary of lessons learned in Section 5.8.

5.1. Experimental Environment

We use the following two testing environments.

—Our server has an Intel Core i7-4770 processor (3.40GHz, four cores, hyperthreading is enabled, two hyperthreads per core), 16GB RAM, and a Seagate hard drive (1TB, 6GB/s, 32MB Cache, and 7200 RPM), running Ubuntu 12.04 LTS with 64-bit Linux 3.11.0 kernel.

—We also use a dedicated cluster with one administration node and 16 worker nodes. The administration node is the master node of Hadoop and HDFS, and clients submit jobs there. All the nodes have the same configuration: dual-socket Intel(R) Xeon(R) CPU E5-2620 (2.10GHz, six cores per socket, and two hyperthreads per core), 60GB RAM, running 64-bit Linux with kernel version 2.6.32. The nodes are connected via gigabit Ethernet, and a working directory is NFS mounted on all the nodes.

Table I. Statistical Functions Built into the Five Tested Platforms

Function	Matlab	MADlib	System C	Spark	Hive
Histogram	yes	yes	no	no	yes
Quantiles	yes	yes	no	no	no
Regression and PAR	yes	yes	no	third-party library	third-party library
Cosine similarity	no	no	no	no	no
Matrix/vector operations	yes	no	no	third-party library	third-party library

Our real dataset consists of $n = 27,300$ electricity consumption time series, each with hourly readings for over a year. We will use the terms consumption time series, consumers, and businesses interchangeably. We also obtained the corresponding temperature time series. The total data size is roughly 10GB.

We also use the proposed data generator to create larger synthetic datasets of size up to 1TB (which corresponds to over 2 million time series) and experiment with them in Sections 5.4 and 5.7. For anomaly detection, we use the first 6 months of data for training (i.e., to build the PAR models and obtain the anomaly thresholds) and the next 30 days of data for detecting anomalies. In anomaly detection, the algorithm receives 1 day of data (hourly consumption and temperature) at a time and must process that day before moving to the next one.

5.2. Benchmark Implementation and Evaluation Methodology

We now introduce the five platforms in which we implemented the proposed benchmark. Whenever possible, we use native statistical functions or third-party libraries. Table I shows which functions were included in each platform and which we had to implement ourselves.

The baseline system is Matlab, a traditional numeric and statistical computing platform that reads data directly from files. We use the built-in histogram, quantile, regression, PAR, and matrix/vector manipulation functions. For similarity search, we implemented our own cosine similarity function by looping through each time series, computing its similarity to every other time series, and, for each time series, returning the top 10 most similar matches.

We also evaluate PostgreSQL 9.1 and MADlib version 1.4 [Hellerstein et al. 2012], which is an open-source platform for in-database machine learning. As we will explain later in this section, we test two ways of storing the data: one measurement per row with a clustered index on consumer ID, and one customer per row with all the measurements for this customer stored in a Postgres array. Everything we need except cosine similarity and vector distance is built in. We implemented the benchmark in PL/PG/SQL with embedded SQL, and we call the statistical functions directly from within SQL queries. We set "shared buffers = 3,072MB, temp buffers = 256MB, work mem = 1,024MB, checkpoint segments = 64" and we use default values for other configuration parameters.[10]

Next, we use System C as an example of a state-of-the-art commercial system. It is a main-memory column store geared toward time series data. System C maps tables to main memory to improve I/O efficiency. In particular, at loading time, all the files

[10]We also experimented with turning off concurrency control and write-ahead logging, which are not needed in our application, but the performance improvement was not significant.

are memory mapped to speed up subsequent data access. However, System C does not include a machine-learning toolkit, and therefore we implemented all the required statistical operators as user-defined functions in the procedural language supported by it.

We also use Spark 1.5.2 [Zaharia et al. 2010] and Spark Streaming [Zaharia et al. 2012] as examples of open-source distributed data processing platforms. Spark reports improved performance on machine-learning tasks over standard Hadoop/MapReduce due to better use of main memory [Zaharia et al. 2010]. We use the *Apache Math* library for regression, but we had to implement our own histogram, quantile, and cosine similarity functions. We use the Hadoop Distributed File System (HDFS) as the underlying file system for Spark, and we experiment with several different file formats.

Finally, we test another distributed platform, Hive 1.2.1 [Thusoo et al. 2009], which is built on top of Hadoop and includes a declarative SQL-like interface. We use Hadoop/Yarn 2.6.2 in this article. Hive has a built-in histogram function, and we use *Apache Math* for regression. We implemented the remaining functions (quantiles and cosine similarity) in Java as user-defined functions (UDFs). The data are stored in Hive external tables.

In the remainder of this section, we will refer to the five tested platforms as Matlab, MADlib, C (or System C), Spark, and Hive.

For the four offline algorithms, we measure running time and memory consumption using different dataset sizes (i.e., different number of households). For anomaly detection, we first compute the PAR model coefficients and anomaly thresholds for each consumer but do not include the time it took to do this. The coefficients and thresholds are then stored in files (Matlab, C, Spark, Hive) or tables indexed by consumer ID (MADlib). Recall that anomaly detection requires the last 3 days of consumption data to compute predictions (Section 3.5.2). Thus, to speed up the process, each platform stores a sliding window of the last 3 days plus the current day as a separate table or as separate files before adding new data to the table storing the entire history. To compute predictions, it suffices to access these smaller tables/files rather than the entire history. We then process the next 30 days of data, 1 day at a time, and measure the time it takes to load the data and run anomaly detection. Loading new data means different things in different platforms:

—In Matlab, we copy 1 day of data at a time from disk to memory as a single file. The file is then loaded into two n-by-24 matrices, with each row corresponding to the 24 hourly consumption or temperature measurements for a particular household. We use Matlab's *find* operator to retrieve data from the matrix when computing predictions and distances between vectors.
—In MADlib, new data are loaded into the sliding-window table, which stores one measurement per row.
—In C, we copy new data into memory and maintain the sliding window there.
—In Spark/Spark Streaming, we copy new data from the local file system into HDFS and then into memory. We use the Spark Streaming API to consume batches of new data 1 day at a time and to maintain a sliding window of recent data. Furthermore, we define the model parameters and thresholds as broadcast variables (accessible via hashing) so that each processing node can compute predictions and detect anomalies at the map side.
—In Hive, we also copy new data from the local file system into HDFS, and the sliding window of recent data is accessible as a Hive external table.

Note that the online part of our benchmark does not take into account lagged reporting and delayed data (e.g., due to data feed outages); instead, it "replays" 30 days of data, 1 day at a time. This means that each batch of new data has the same size

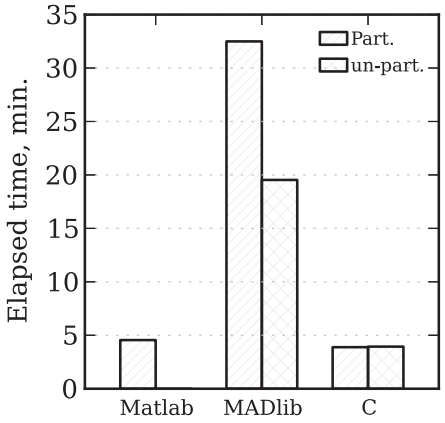


Fig. 13. Data loading times, 10GB real dataset.

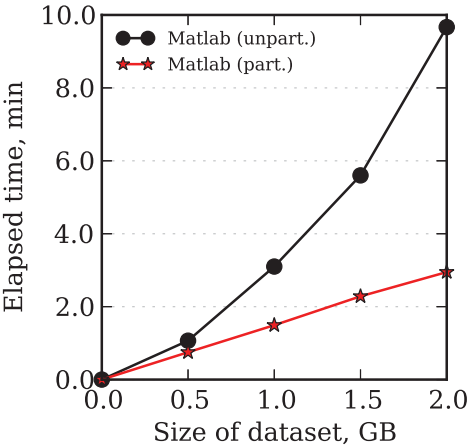


Fig. 14. Impact of data partitioning on analytics, three-line algorithm.

and takes roughly the same amount of time to process. In practice, some batches may contain more data than others, meaning that the system must be overprovisioned to process all the data that can possibly arrive in one batch before the next batch arrives.

In terms of programming time to implement our benchmark, PostgreSQL/MADlib required the least effort, followed by Matlab and Hive, while Spark and especially System C required by far the most effort. In particular, we found Hive UDFs easier to write than Spark programs.[11] However, since we did not conduct a user study, these programmer effort observations should be treated as anecdotal.

5.3. Offline Algorithms: Single-Server Results

We begin by comparing Matlab, MADlib, and System C running on a single multicore server, using the real 10GB dataset. Again, we start with offline tasks only.

5.3.1. Data Loading and File Partitioning. First, we investigate the effect of loading and processing one large file containing all the data versus one file per consumer. Figure 13 shows the time it took to load our 10GB real dataset into the three systems tested in this section, both in a partitioned (one file per consumer, abbreviated "part.") and nonpartitioned (one big file, abbreviated "un-part.") format. The partitioned data load also includes the cost of splitting the data into small files. The loading time into PostgreSQL is the slowest of the three systems, but it is more efficient to bulk-load one large CSV file than many smaller files. In terms of data loading time, System C is not significantly affected by the number of files. Matlab does not actually load any data and instead reads from files directly. The single bar reported for Matlab, of roughly 4.5 minutes, simply corresponds to the time it took to split the dataset into small files.

Since Matlab reads data directly from files, the goal of our next experiment is to investigate the performance of analytics in Matlab given the two partitioning strategies discussed earlier. Figure 14 shows the running time of the three-line algorithm using Matlab on (partitioned and nonpartitioned) subsets of our real datasets sized from 0.5 to 2GB. (We observed similar trends when running the other algorithms in the

[11]However, had we used the recently proposed SparkSQL rather than writing Spark programs directly, the programming effort would likely be on par with that of Hive.

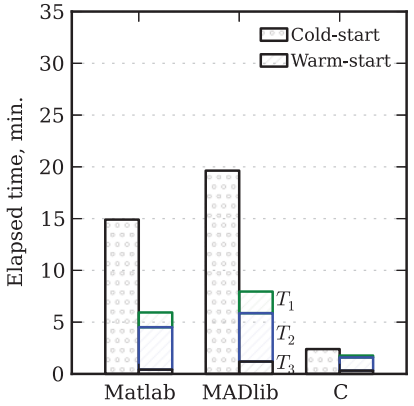


Fig. 15. Cold start versus warm start, three-line algorithm, 10GB real dataset.

benchmark.) The impact on Matlab is significant: it operates much more efficiently if each consumer's data are in a separate file. Upon further investigation, we noticed that Matlab reads the entire large file into an index, which is then used to extract individual consumers' data; this is slower than reading small files one by one and running the three-line algorithm on each file directly.

Based on the results of this experiment, in the remainder of this section, we always run Matlab with one file per consumer.

5.3.2. Cold Start Versus Warm Start. Next, we measure the time it takes each system to load data into main memory before executing the three-line algorithm (we saw similar trends when testing other algorithms from the benchmark, so we omit those results for brevity). In *cold start*, we record the time to read the data from the underlying database or filesystem and run the algorithm. In *warm start*, we first read the data into memory (e.g., into a Matlab array, or in PostgreSQL, we first run SELECT queries to extract the data we need) and then we run the algorithm. Thus, the difference between the cold-start and warm-start running times corresponds to the time it takes to load the data into memory.

Figure 15 shows the results on the real dataset. The left bars indicate cold-start running times, whereas the right bars represent warm-start running times and are divided into three parts: T_1 is the time to compute the 10th and 90th quantiles, T_2 is the time to compute the regression lines, and T_3 is the time to adjust the lines in case of any discontinuities in the piecewise regression model. Cold-start times are higher for all platforms, but Matlab and MADlib spend the most time loading data into their respective data structures, followed by System C. Overall, System C is easily the fastest and the most efficient at data loading—most likely due to efficient memory-mapped I/O. Also note that for each system, T_2, that is, the time to run least-squares linear regression, is the most costly component of the three-line algorithm.

Figure 15 suggests that System C is noticeably more efficient than Matlab even in the case of warm start, when Matlab has all the data it needs in memory. There are at least two possible explanations for this: Matlab's data structures are not as efficient as System C's, especially at the data sizes we are dealing with, or Matlab's implementation of linear regression and other statistical operators is not as efficient as our hand-crafted implementations within System C. We suspect it is the former. To investigate this further, we measured the running time of multiplying two randomly

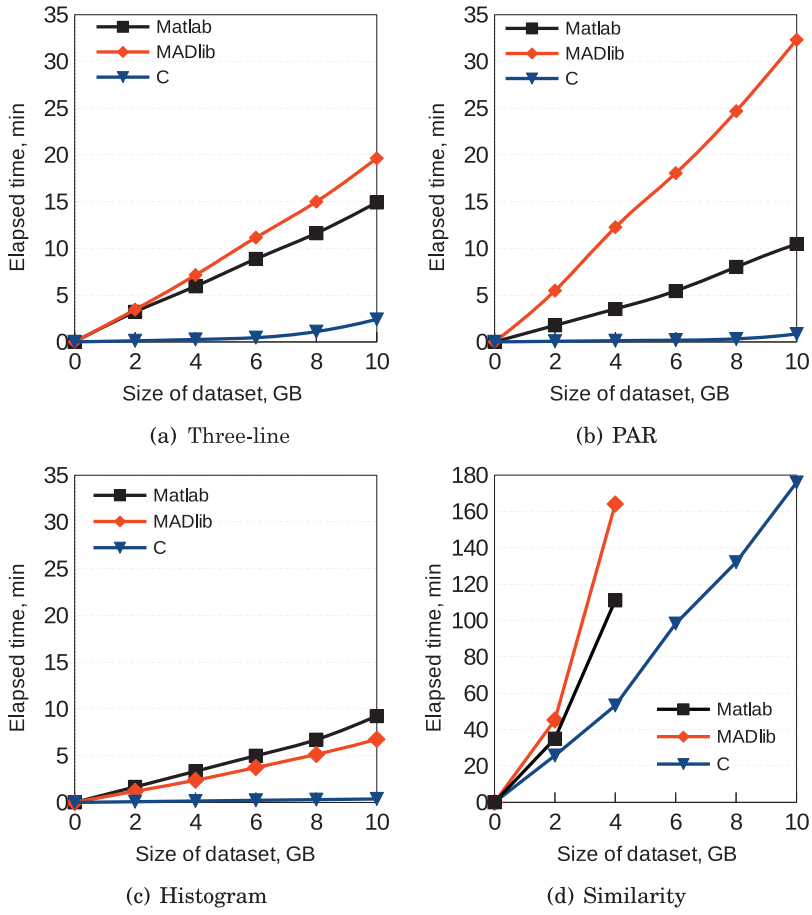


Fig. 16. Single-threaded execution times of each algorithm using each system.

generated 4,000 × 4,000 floating-point matrices in Matlab and System C. Indeed, Matlab took under a second, while System C took over 5 seconds.

5.3.3. Single-Threaded Results. We now measure the cold-start running times of each offline algorithm in single-threaded mode (i.e., no parallelism). System C has a configuration parameter that governs the level of parallelism (the number of cores to use), while for Matlab, we start with a single instance, and for MADlib, we establish a single database connection. We use subsets of our real datasets with sizes between 2 and 10GB for this experiment. The running time results are shown in Figure 16 for three-line, PAR, histogram construction, and similarity search, from left to right. Note that the y-axis of the rightmost plot is different: similarity search is slower than the other three tasks, and the Matlab and MADlib curves end at 4GB because the running time on larger datasets was prohibitively high. System C is the clear winner: it is a commercial system that is fast at data loading thanks to memory-mapped I/O, and fast at query execution since we implemented the required statistical operators in a low-level language. Matlab is the runner-up in most cases except histogram construction, which is simpler than the other tasks and can be done efficiently in a database system without optimized vector and matrix operations. MADlib has the worst performance for three-line, PAR, and similarity search.

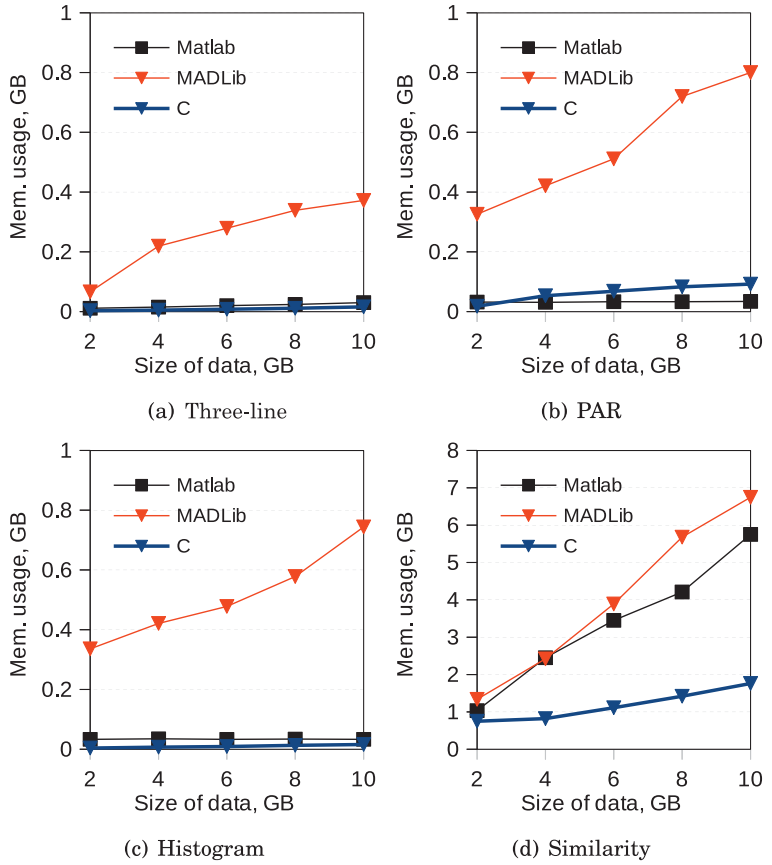


Fig. 17. Memory consumption of each algorithm using each system.

Figure 17 shows the corresponding memory consumption of each algorithm for each platform; the plots correspond to running the "free -m" command every 5 seconds throughout the runtime of the algorithms and taking the average. Matlab and System C have the lowest memory consumption; recall that for Matlab, we use separate files for different consumers' data and therefore the number of files that need to be in memory at any given time is limited.

In terms of the tested algorithms, three-line has the lowest memory usage since it only requires the 10th and 90th percentile data points to compute the regression lines, not the whole time series. The memory footprint of PAR and histogram construction is higher because they both require the whole time series. The memory usage of similarity search is higher still, especially for Matlab and MADlib, both of which keep all the data in memory for this task. On the other hand, since System C employs memory-mapped files, it only loads what is required.

The relatively poor performance of MADlib may be related to its internal storage format. In the next experiment, we check if using the PostgreSQL *array* data type improves performance. Table 1 in Figure 18 shows the conventional row-oriented schema for smart meter data that we have used in all the experiments so far, with a household ID, the outdoor temperature, and the electricity consumption reading (plus the timestamp, which is not shown). That is, each data point of the time series is stored as a separate row, and a clustered B-tree index is built on the household ID to speed up

Table 1

householdID *int*	temperature *double*	reading *double*
1000	-4	0.35
1000	-3	0.26
...		
2000	-2	2.0
2000	3	1.1
...		

Table 2

householdID *int*	temperature *double[]*	reading *double[]*
1000	[-4,-3,...]	[0.35,0.26,...]
...		
2000	[-2,3,...]	[2.0,1.1,...]
...		

Fig. 18. Two table layouts for storing smart meter data in PostgreSQL.

the extraction of all the data for a given consumer (household). Table 2 in Figure 18 stores one row for each consumer and uses arrays to store all the temperature and consumption readings for the given consumer using the same positional encoding. Using arrays, the running time of three-line on the whole 10GB dataset went down from 19.6 minutes to 11.3 minutes, which is faster than Matlab but still much slower than System C (recall the leftmost plot in Figure 16). The other algorithms also ran slightly faster but not nearly as fast as in System C: the PAR running time went down from 34.9 to 30 minutes, the histogram running time went down from 7.8 to 6.8 minutes, and the running time of similarity search (using 6,400 households, which works out to about 2GB) went down from 58.3 to 40.5 minutes. However, the performance gains of the array data type come with the overhead of converting raw data, containing one measurement per line, to an array-based layout. In our experiments, it took 28 minutes to transform the 10GB real-world dataset. In addition, a PostgreSQL array cannot exceed 1GB. We also experimented with a table layout in between those in Table 1 and Table 2, namely, one row per consumer per day, which resulted in running times in between those obtained from Table 1 and Table 2.

5.3.4. Multithreaded Results. We now evaluate the ability of the tested platforms to take advantage of parallelism. Our server has four cores with two hyperthreads per core, but we cannot control the use of these resources directly in all of the systems tested. As a result, we define the *parallelism level* differently for each system: for System C we run one process and vary the number of threads; for Matlab we vary the number of separate processes, which we run manually; and for MADlib we vary the number of database connections.

In general, the histogram, three-line, and PAR algorithms are easy to parallelize as each thread can run on a subset of the consumers without communicating with the other threads. Similarity search is harder to parallelize because for each time series, we need to compute the cosine similarity to every other time series. We do this by running parallel tasks in which each task is allocated a fraction of the time series and computes the similarity of its time series with every other time series. The results are then merged and the top 10 results are returned for each consumer.

Figures 19(a) through 19(d) show the speedup obtained by increasing the parallelism level from one to eight for each algorithm. Again, we continue to use the 10GB real dataset. Each plot includes a diagonal line indicating ideal speedup (i.e., using two connections or cores would be twice as fast as using one). The results show that Matlab and System C can obtain nearly linear speedup when the degree of parallelism is no greater than four. This makes sense since our server has four physical cores, and increasing the level of parallelism beyond four brings diminishing returns due to increasing resource contention (e.g., for floating-point units) among hyperthreads. Matlab and C appear to scale better than MADlib, but this may be an artifact of how we simulate parallelism: Matlab instances run in a shared-nothing fashion because each consumer's data are in a separate file, while MADlib uses multiple connections to the same database server, with each connection reading data from the same table.

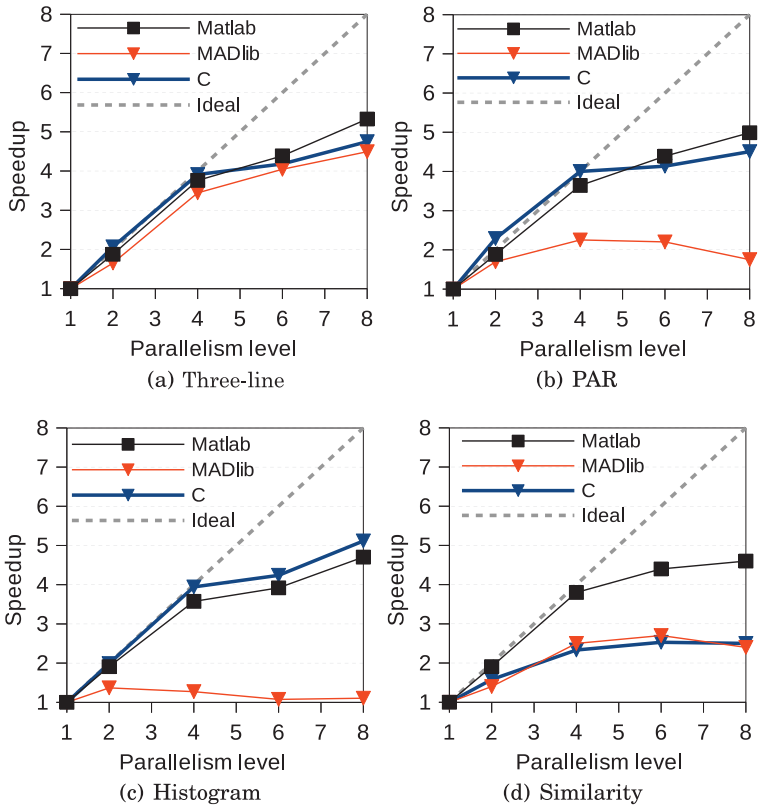


Fig. 19. Speedup of execution time on a single multicore server using the 10GB real dataset.

5.4. Offline Algorithms: Cluster Results

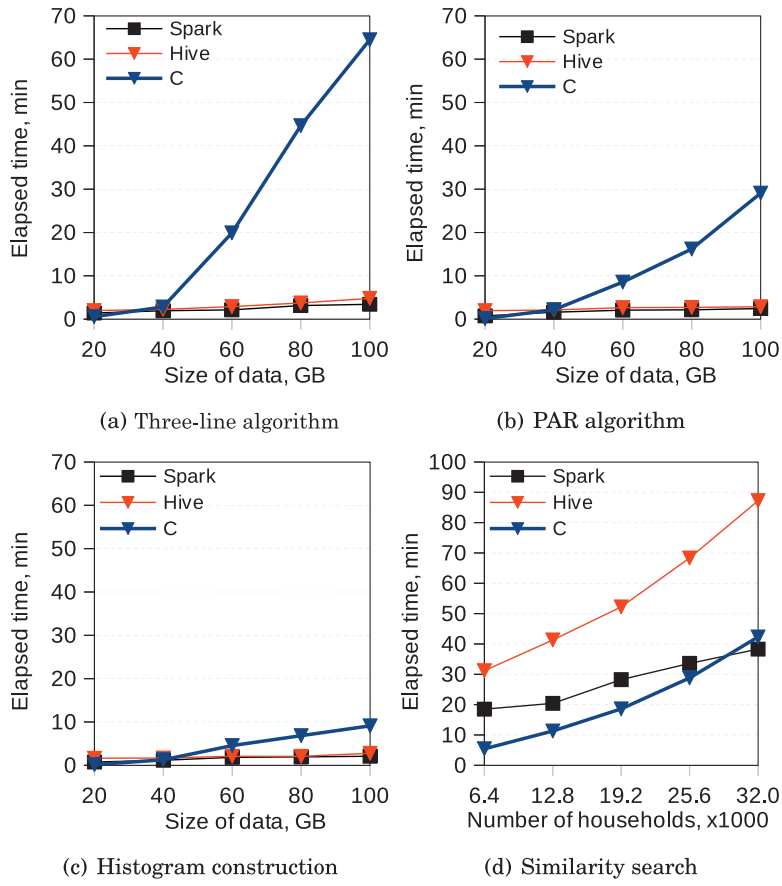
We now focus on the performance of offline analytics in Spark and Hive on a cluster using large synthetic datasets. We set the number of parallel executors for Spark and the number of MapReduce tasks for Hive to be up to 12 per node, which is the number of physical cores.[12]

5.4.1. System C Versus Spark and Hive. In the previous batch of experiments, System C was the clear performance winner in a single-server scenario. We now compare System C against the two distributed platforms, Spark and Hive, on large synthetic datasets of up to 100GB (for similarity search, we use 6,000 up to 32,000 time series). This experiment is unfair in the sense that we run System C on the server (with maximum parallelism level of eight hyperthreads) but we run Spark and Hive on the cluster. Nevertheless, the results are interesting.

Figure 20 shows the running time of each algorithm. Up to 40GB data size, System C is keeping up with Spark and Hive despite running on a single server. Similarity search performance of System C is also very good.

Figure 21 illustrates another way of comparing the three systems that is more fair. Part (a) shows the throughput, for three-line, PAR, and histogram construction, in

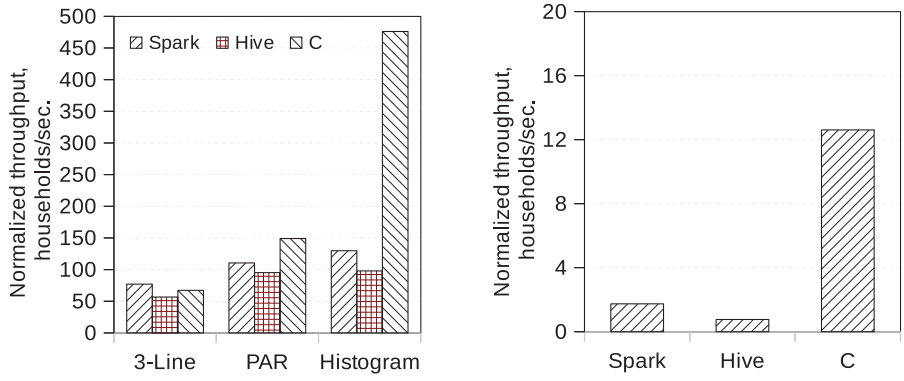
[12]We experimented with different values of these parameters and found that Spark was not sensitive to the number of parallel executors, while Hive generally performed better with more MapReduce tasks up to a certain point.



(a) Three-line algorithm (b) PAR algorithm

(c) Histogram construction (d) Similarity search

Fig. 20. Execution times using large synthetic datasets.



(a) Three-line, PAR, and histogram construction; 100GB dataset

(b) Similarity search, 32K households

Fig. 21. A comparison of throughput per server of System C, Spark, and Hive.

terms of how many households can be handled per second *per server* when using the 100GB synthetic dataset. That is, we divide the total throughput of Spark and Hive by 16, the number of worker nodes in the cluster. Using this metric, even at 100GB, System C is competitive with Spark and Hive on three-line and PAR, and better on the simple algorithm of histogram construction. Similarly, part (b) shows that the throughput per server for similarity search is higher for System C at 32K households.

5.4.2. Spark Versus Hive Using Different Data Formats. In this experiment, we take a closer look at the relative performance of Spark and Hive and the impact of the file format, using synthetic datasets up to 1TB. We use the default HDFS text file format, with default serialization, and without compression. The three options we test are (1) one file (that may be partitioned arbitrarily by HDFS) with one smart meter reading per line, (2) one file with one household per line (i.e., all the readings from a single household on a single line), and (3) many files, with one or more households per file (but no household scattered among many files) and one smart meter reading per line. Note that while the first format is the most flexible in terms of storage, it may require a *reduce* step for the tested algorithms since we cannot guarantee that all the data for a given household will be on the same server. The second and third options do not require a reduce step.

In Hive, we use three types of user-defined functions with the three file formats: generic UDF (user-defined function), UDAF (user-defined aggregation function), and UDTF (user-defined table function). UDF and UDTF typically run at the map side for the scalar operations on a row, while UDAF runs at the reduce side for an aggregation operations on many rows. We use a UDAF for the first format since we need to collate the numbers for each household to compute the tested algorithms. We use a generic UDF for the second format, for which map-only jobs suffice. We use a UDTF for the third format since UDTFs can process a single row and do the aggregation at the map side, which functions as a *combiner*. For the third format, we also need to customize the file input format, which takes a single file as an input split. We overwrite the `isSplitable()` method in the `TextInputFormat` class by returning a `false` value, which ensures that any given time series is processed in a self-contained manner by a single mapper.

First data format. Figure 22 shows the execution time of the four tested algorithms on various dataset sizes up to 1TB. Spark is noticeably faster for similarity search (in Hive, we implemented this as a self-join, which resulted in a query plan that did not exploit map-side joins, whereas in Spark we directly implemented similarity search as a MapReduce job with broadcast variables and map-side joins), slightly faster for PAR and histogram construction, and slower for three-line construction as the data size grows. Figure 23 shows the speedup relative to using only four out of 16 worker nodes for the terabyte dataset, with the number of worker nodes on the x-axis. Hive appears to scale slightly better as we increase the number of nodes in the cluster. Finally, Figure 24 shows the memory usage as a function of the dataset size, computed the same way as in Figure 17. Spark uses more memory than Hive, especially as the data size increases. As for the different algorithms, three-line is the most memory intensive because it requires temperature data in addition to smart meter data.

Second data format. Figures 25 and 26 show the execution times and the speedup, respectively, with one time series per line. For three-line, PAR, and histogram construction, we do not require a reduce step. Therefore, the running times are lower than for the first data format, in which a single time series may be scattered among nodes in the cluster. Spark and Hive are very close in terms of running time because they perform the same HDFS I/O. We also see a higher speedup than with the first data format thanks to map-only jobs, which avoid an I/O-intensive data shuffle among

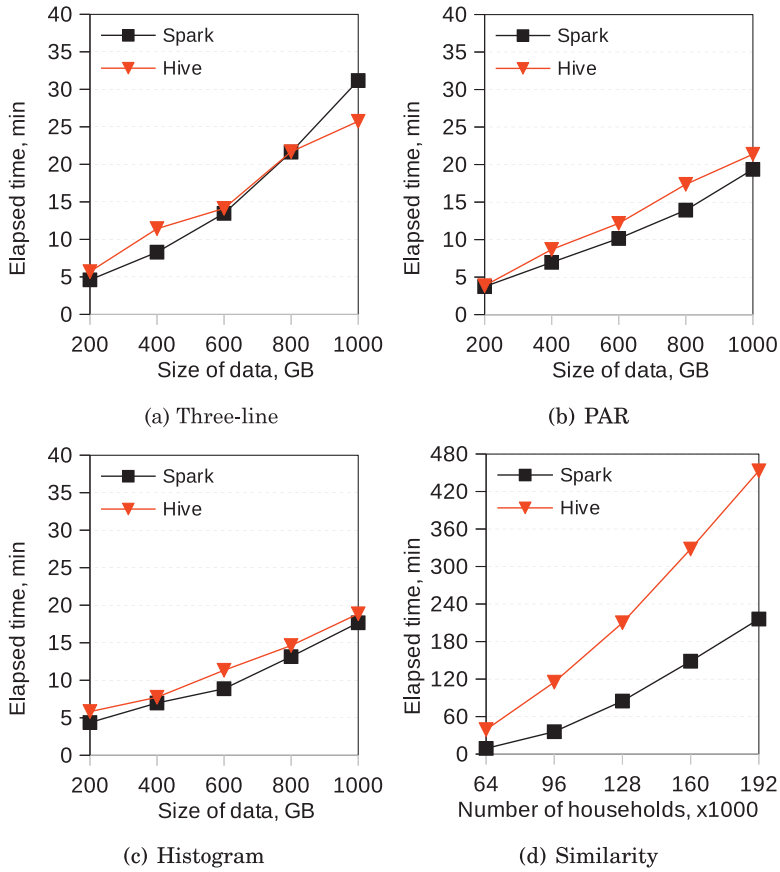


Fig. 22. Execution times using the first data format in Spark and Hive.

servers compared to jobs that include both map and reduce phases. Similarity search is slightly faster than with the first data format; most of the time is spent on computing the pairwise similarities, and the only time savings in the second data format are due to not having to group together the readings from the same households. Note that similarity search still requires a reduce step to sort the similarity scores for each household and find the top-k most similar consumers.

Third data format. Here, we only use the 100GB dataset with a total of 260,000 households and we vary the number of files from 10 to 10,000; recall that in the third data format, the readings from a given time series are guaranteed to be in the same file. We test two options in Hive: a UDTF with the customized file input format described earlier, and a UDAF in which a reduce step is required. We do not test similarity search since the distance calculations between pairs of time series cannot be done in one UDTF operation. Figures 27 and 28 show the execution times and the speedup, respectively. Hive with UDTF wins in this format since it does not have to perform a reduce step. Furthermore, while Hive does not seem to be affected by the number of files, at least between 10 and 10,000, Spark's performance deteriorates as the number of files increases. In fact, we also experimented with more files, up to 100,000, and found that Spark was not even runnable due to "too many files open" exceptions.

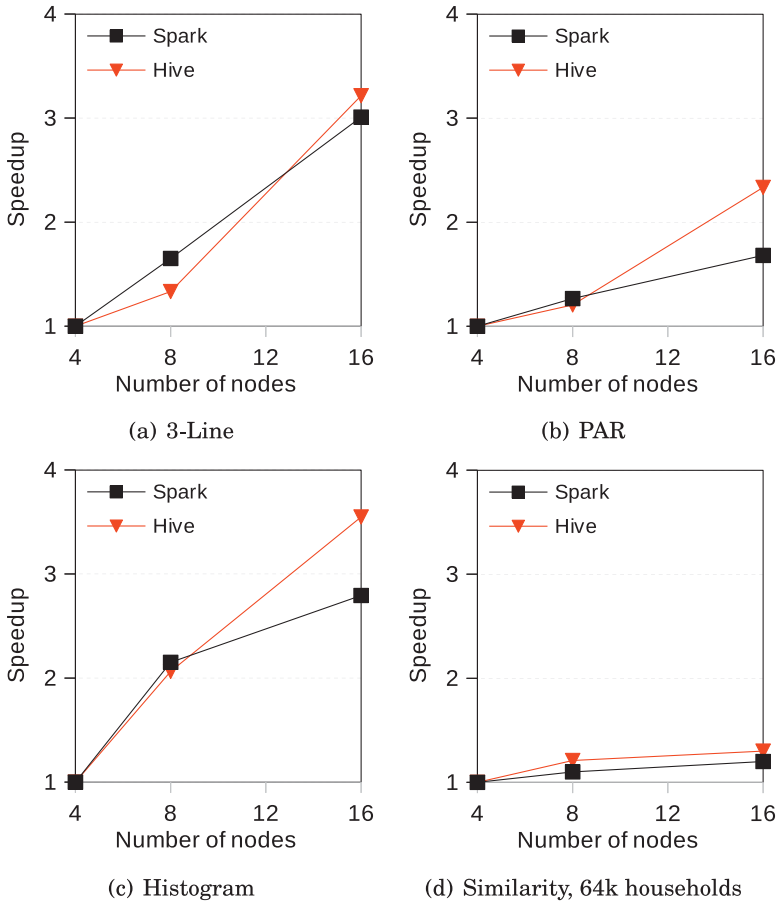


Fig. 23. Speedup obtained using the first data format in Spark and Hive.

5.5. Online Algorithms: Data Loading

We now turn to anomaly detection, starting with how long it takes, on average (over the 30 tested days of data), to load 1 day of data in each platform. Figure 29 summarizes the results. The plot on the left compares Matlab, MADlib, and C using our server and using subsets of our real dataset, from 5,000 to 25,000 time series (x-axis). System C loads data (i.e., performs the memory mapping) quickly, whereas MADlib is the slowest due to the overhead of inserting data into tables and building indices. The middle plot shows the data loading time into HDFS for Hive/Spark using our 16-machine cluster and the larger synthetic datasets, with 500,000 up to 2.5 million time series. The loading time scales roughly linearly, and notice that it is similar to that in the left plot despite the data size (x-axis) being two orders of magnitude larger. The plot on the right compares the data loading throughput per second, in thousands of time series, for each platform. Parallel data loading, using the copy utility from the local file system to HDFS, clearly gives much higher throughput. However, even Matlab and MADlib can load several thousand time series (1 day, i.e., 24 measurements each) per second.

5.6. Online Algorithms: Single-Server Results

We now investigate the running time of our anomaly detection algorithms, starting with a single-server comparison of Matlab, MADlib, and C.

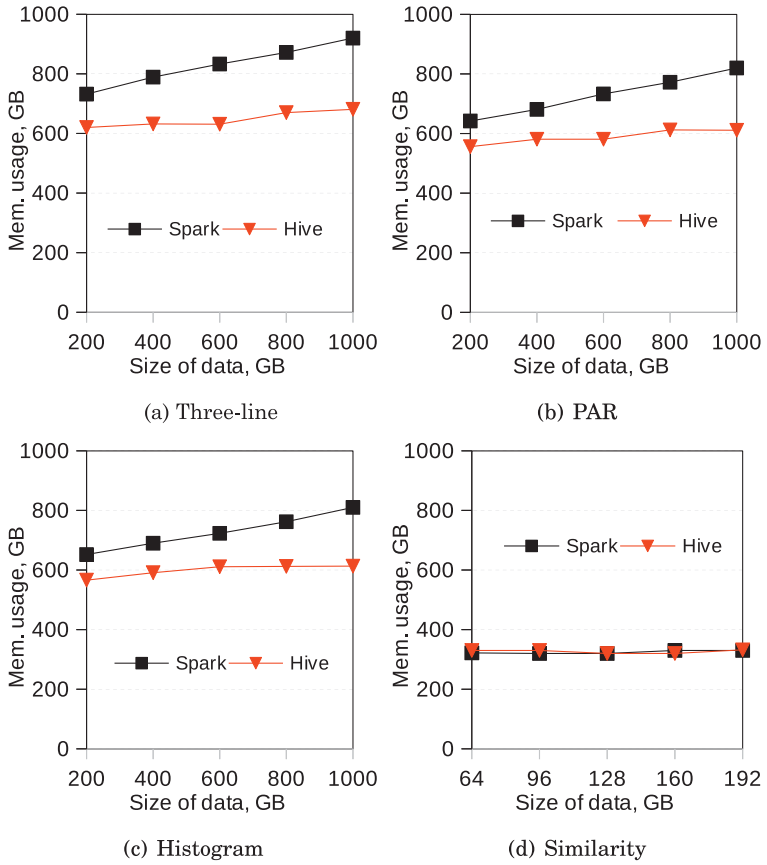


Fig. 24. Memory consumption of each algorithm in Spark and Hive.

5.6.1. Single-Threaded Results. Figures 30 and 31 illustrate the single-threaded running time and memory consumption, respectively, of our self-anomaly (left) and group anomaly (right) algorithm. We use subsets of the real dataset between 5,000 and 25,000 households. We run the anomaly detection algorithms right after loading the data, 1 day at a time, so the running times reported here correspond to warm starts. The reported runtimes correspond to processing 1 day of data, averaged over 30 days.

For all platforms, the anomaly detection time increases as the data size increases. As with the offline algorithms, System C is the fastest: it can run anomaly detection on 1 day's data for the full 25,000 time series in 8 seconds (4 seconds each for self- and group anomalies). However, in contrast to our experiments with offline algorithms, System C is not the most memory-efficient platform for self-anomaly detection. Upon further investigation, we noticed spikes in memory consumption during the processing of self-anomalies, which suggests that System C required more memory for intermediate results of this task.

Another interesting result is that MADlib scales better than Matlab, whereas the opposite was true for some of the offline algorithms (recall Figure 16). One reason for this is that anomaly detection requires lookup of model parameters, which is more efficiently done as a query against an indexed table rather than a Matlab matrix lookup via the *find* operator. On the other hand, the offline algorithms sequentially scanned the entire dataset, making indices less relevant. In particular, note that Matlab becomes

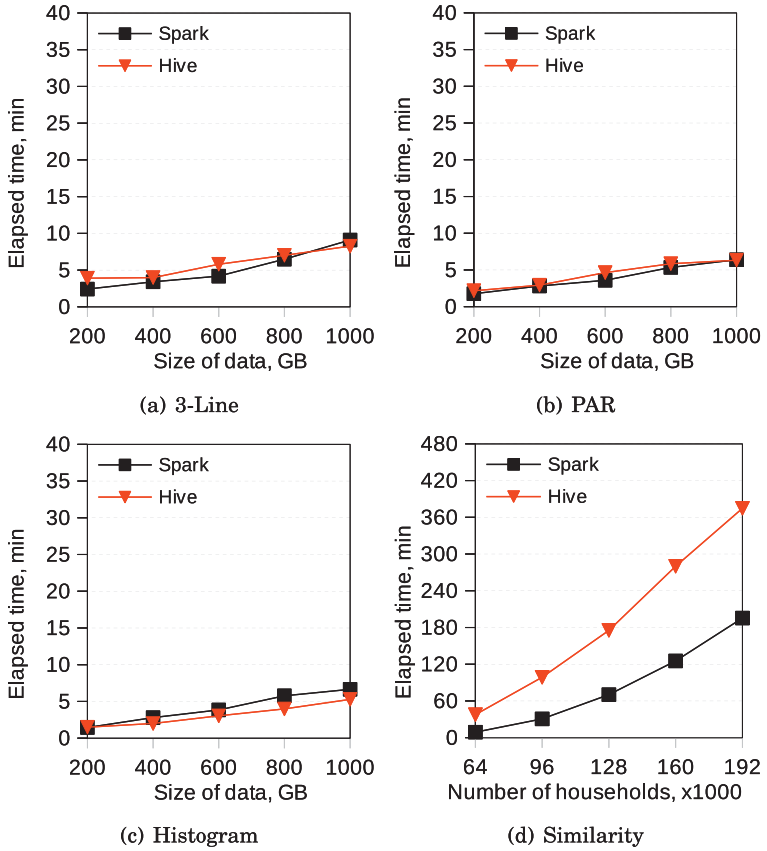
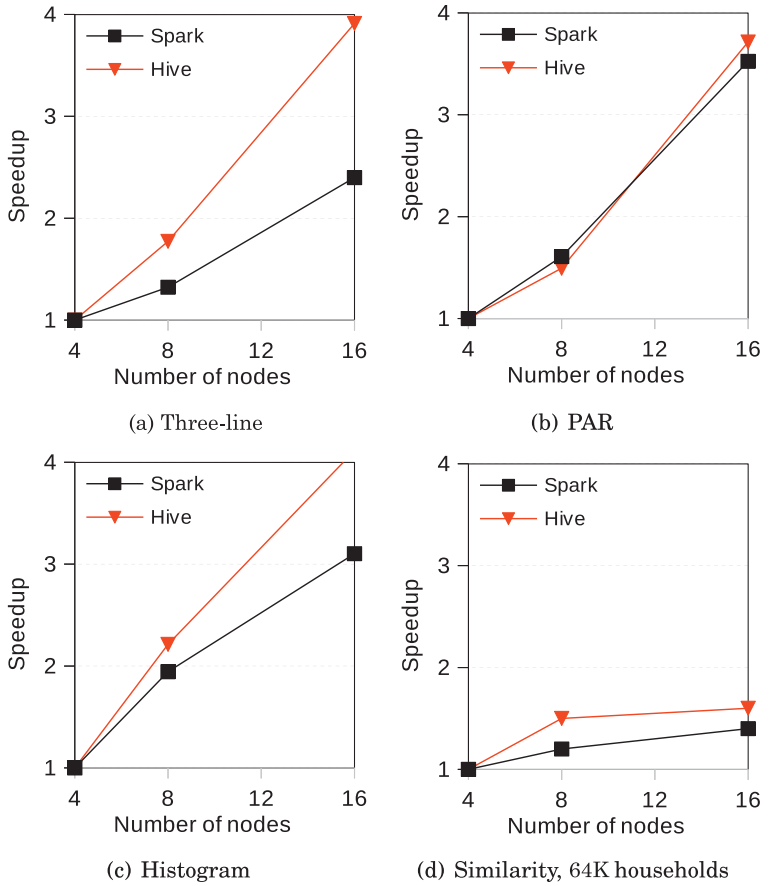


Fig. 25. Execution times using the second data format in Spark and Hive.

slower than MADlib at self-anomalies beyond 20,000 time series, and at the same time, its memory consumption also becomes greater than that of MADlib. This suggests another reason MADlib scales better: rather than loading all the data into matrices, it suffices to select the relevant historical data, model parameters, and temperature data for one household at a time.

Finally, note that MADlib's memory usage was noticeably higher than the other platforms' memory usage in group anomaly detection. This is because we were able to write a single SQL query to compute all the group anomalies, which required data for all households. On the other hand, in Matlab and System C, we implemented group anomaly detection one household at a time.

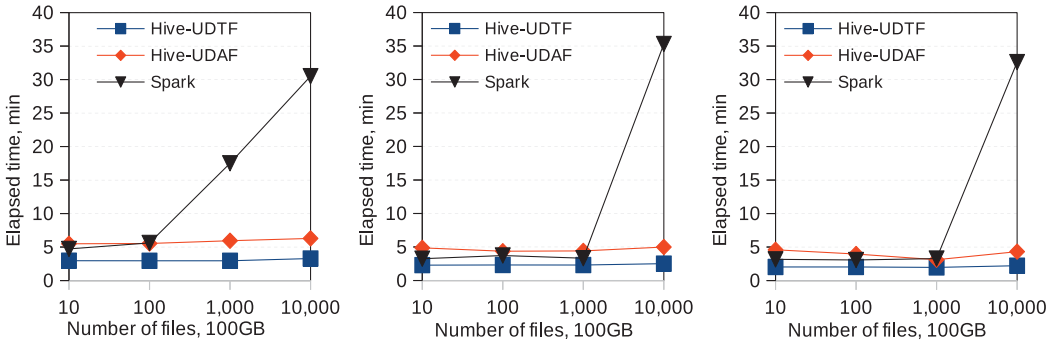
5.6.2. Multithreaded Results. We now measure the effect of parallelism using the same methodology as in Section 5.3.4. That is, we directly specify the number of threads in System C, we set up multiple database connections in MADlib, and we run multiple instances of Matlab. As with offline tasks, each PostgreSQL database connection accesses the same table and is responsible for an equal fraction of the time series. However, in contrast to Section 5.3.4, in which Matlab was working with separate files per household, here we use a single file to store recent data for all households. Rather than copying this file to each Matlab instance, we manually partition the file so that each instance receives an equal number of time series, and we pass each partition to a single instance.



(a) Three-line (b) PAR

(c) Histogram (d) Similarity, 64K households

Fig. 26. Speedup obtained using the second data format in Spark and Hive.



(a) Three-line (b) PAR (c) Histogram

Fig. 27. Execution times using the third data format in Spark and Hive.

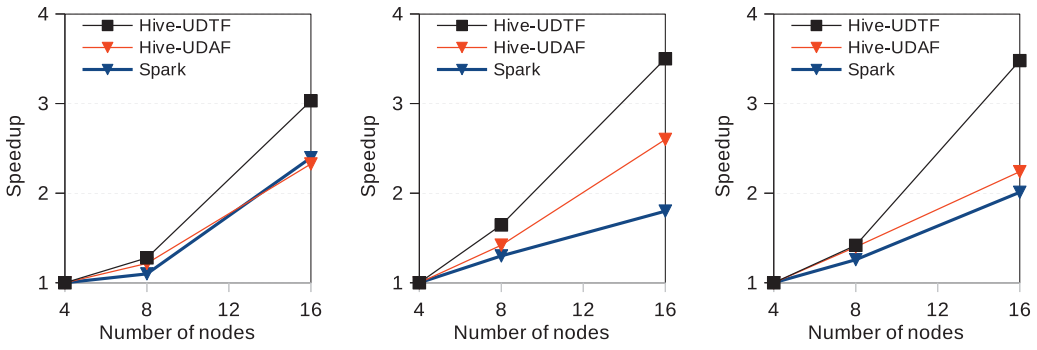


Fig. 28. Speedup obtained using the third data format in Spark and Hive, 100 files, 1GB per file.

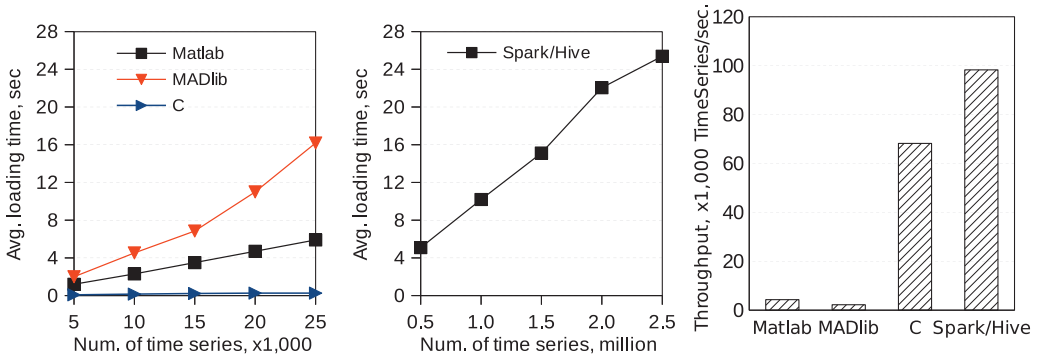


Fig. 29. Average data loading time per day for single server (left) and cluster (middle), and loading throughput comparison (right).

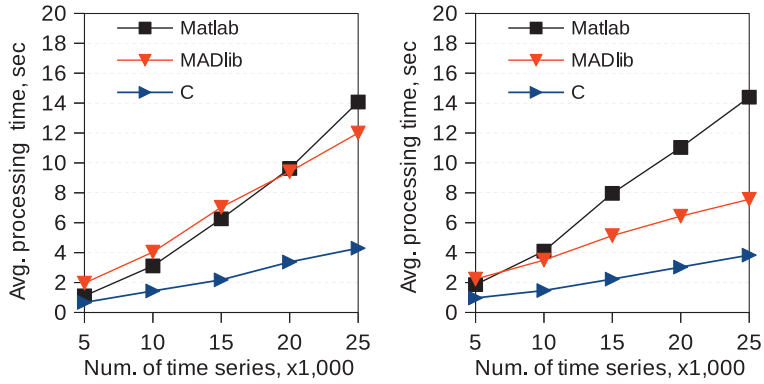


Fig. 30. Running time of self- (left) and group (right) anomaly detection, 30 days of real data.

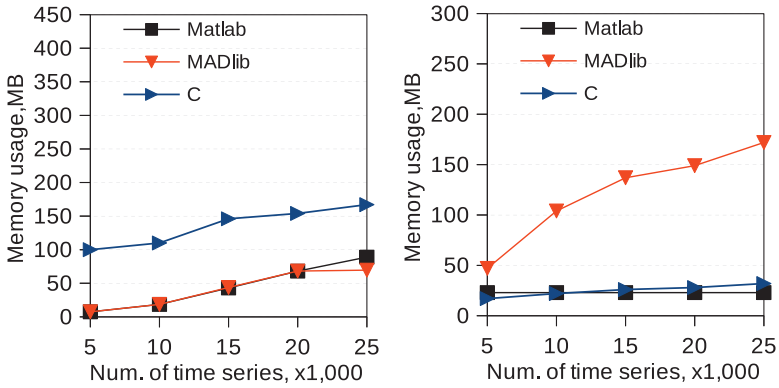


Fig. 31. Memory usage of self- (left) and group (right) anomaly detection, 30 days of real data.

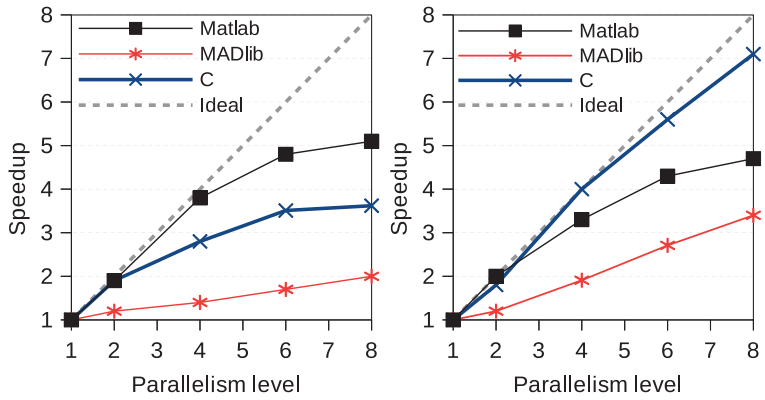


Fig. 32. Speedup of execution time on a single multicore server using the real dataset for self- (left) and group (right) anomalies.

Figure 32 shows the speedup for self- (left) and group (right) anomalies, along with a diagonal line indicating ideal speedup. We use the full real dataset (25,000 time series) and we vary the level of parallelism on the x-axis. In general, the results are similar to those in Figures 19(a) through 19(d) for offline tasks: speedup decreases beyond parallelism level four because our server only has four physical cores, and MADlib appears to scale the worst, likely due to our way of simulating parallelism, which incurs the overhead of setting up multiple database connections. Additionally, group anomaly detection in System C appears easier to parallelize than self-anomaly detection. A possible explanation is that, as we saw in Figure 31, memory consumption during intermediate stages of processing self-anomalies is higher in this platform.

5.7. Online Algorithms: Cluster Results

In the final set of experiments, we compare the running time and memory consumption of anomaly detection in Hive and Spark Streaming. We generate synthetic datasets with 500,000 up to 2.5 million time series. The largest dataset contains over 2.5GB of data per day and is the largest dataset whose history (needed for computing the model parameters) fits in the main memory of the cluster.

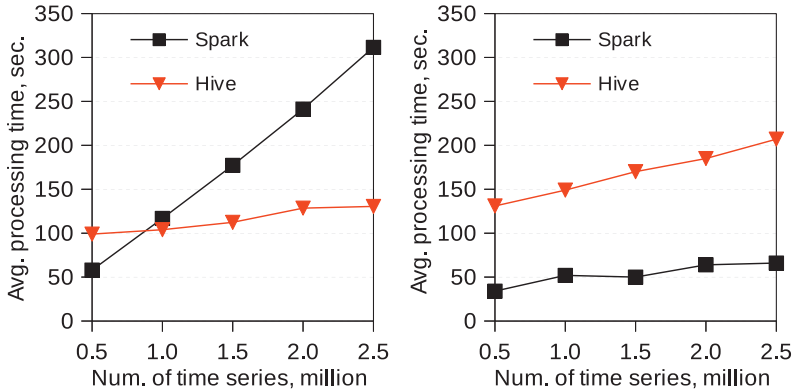


Fig. 33. Execution times in Spark and Hive for self- (left) and group (right) anomalies.

Recall from Section 5.4.2 that we tested offline tasks in the cluster using three data formats: one large file for the entire dataset with one measurement per line, one large file with one household's time series per line, and one file per household with one measurement per line. The first format was the slowest for offline tasks because data for one household may be distributed among multiple servers. The second format was the fastest, and therefore we use it again in the context of anomaly detection.

Figure 33 plots the running time of self- and group anomaly detection as a function of the number of time series. For group anomaly detection, both systems scale well in the sense that processing time does not vary much with the size of the input. Given the relatively small size of the input data (the entire history is very large, but 1 day's worth of data, which is the processing unit for anomaly detection, is not), the performance difference is likely due to overheads associated with task distribution; by default, Hive launches a separate Java Virtual Machine (JVM) for each task, whereas Spark reuses task executors more intelligently. In fact, even running an "SELECT *" Hive query from a one-row table took nearly 30 seconds in our cluster. For self-anomaly detection, Spark performs substantially worse due to the overhead of accessing data from the past 3 days versus accessing the latest neighborhood data in group anomaly detection. Specifically, we noticed that Spark Streaming checkpoints the sliding window of the past 3 days to HDFS whenever new data are added.

Combining the data loading results from Figure 29 and algorithm runtime results from Figure 33, we conclude that using our cluster, Hive and Spark Streaming can detect self- and group anomalies in 2.5 million time series (24 measurements each) in roughly 6 to 7 minutes.

The corresponding memory consumption numbers are shown in Figure 34. As was the case with offline algorithms, Spark uses more memory.

In the final two experiments, we use the largest synthetic dataset (2.5 million time series) and we vary the number of nodes in the cluster from four to 16. Figure 35 illustrates the throughput, in thousands of time series per second, of the self- (left) and group (right) anomaly detection algorithms. In Figure 36, we show the speedup, relative to the running time using four nodes, obtained by increasing the number of nodes. The throughput of self-anomaly detection is much lower in Spark (recall Figure 33) but does increase as we increase the number of nodes. Spark also shows some speedup as we increase the number of compute nodes. On the other hand, the throughput and speedup of Hive increase very slowly with the number of nodes. Again, this is likely related to the overhead associated with task distribution, as discussed in connection with Figure 33, which dominate the running time in our experiments.

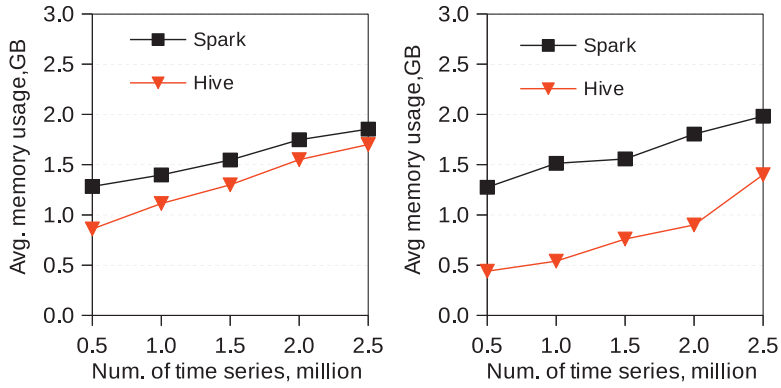


Fig. 34. Memory consumption of Spark and Hive for self- (left) and group (right) anomalies.

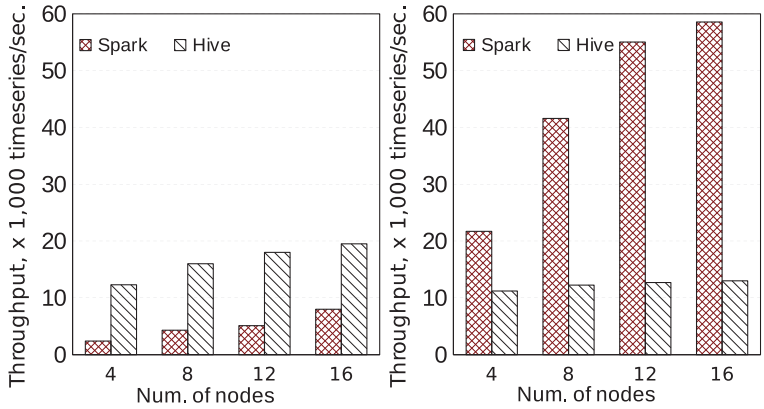


Fig. 35. Throughput of Spark and Hive for self- (left) and group (right) anomalies.

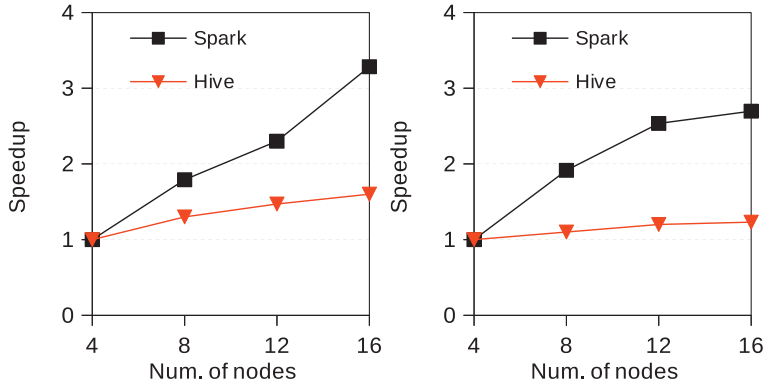


Fig. 36. Speedup of Spark and Hive for self- (left) and group (right) anomalies.

5.8. Lessons Learned

Our main finding is that out of the tested platforms, System C, which is a commercial main-memory column store, is the best choice for offline and online smart meter analytics in terms of performance, provided that the resources of a single machine are sufficient. However, System C lacks a built-in machine learning toolkit and therefore we had to invest significant programming effort to build efficient analytics applications. On the other hand, Matlab and MADlib are likely to be more programmer friendly but slower. Furthermore, for offline tasks, we found that Matlab works better if each customer's time series is stored in a separate file and that PostgreSQL/MADlib works well when the smart meter data are stored using a hybrid row/column-oriented format. For online tasks that require access to new data as well as some historical or offline data, MADlib can be more efficient due to the ability to index the data.

As for the two distributed solutions, for offline tasks Spark was slightly faster but Hive scaled slightly better as we increased the number of worker nodes. We showed that the choice of data format matters; we obtained the best performance when each time series was on a separate line, which eliminated the need to group data explicitly by household ID and thus avoided an I/O-intensive data shuffle among servers. This feature allows our implementations to remain competitive in terms of efficiency with respect to System C for three-line and PAR, whereas cluster computing frameworks in general are known to suffer from poor efficiency compared to centralized systems [Anderson and Tucek 2010]. For online anomaly detection, we found that Spark/Spark Streaming is faster than Hive only if it predominantly accesses new data. We found that historical data required for anomaly detection (i.e., the past 3 days) are not guaranteed to be in memory, in which case Spark loses its performance advantage over Hive.

6. CONCLUSIONS AND FUTURE WORK

Smart meter data analytics is an important new area of research and practice. In this article, we studied smart meter analytics from a software performance perspective. We proposed a performance benchmark for smart meter analytics consisting of five common tasks and presented a data generator for creating very large smart meter datasets. We implemented the proposed benchmark using five state-of-the-art data processing platforms and found that a commercial main-memory column-store system offers the best performance on a single machine, but systems such as MADlib/PostgreSQL and Matlab are more programmer friendly due to built-in statistical and machine-learning operators. In cluster environments, we found Hive easier to use than Spark and Spark Streaming, and not much slower or even faster for some tasks. Compared to centralized solutions, we found Hive and Spark competitive in terms of efficiency for CPU-intensive data-parallel workloads (three-line and PAR).

In future work, we plan to improve the efficiency and effectiveness of smart meter data mining algorithms using recent data management trends such as parallel processing and column storage. We are particularly interested in extending our study of smart meter anomaly detection and classification, including issues such as how to define an anomaly, how to find a proper threshold for when to flag an anomaly, and how to explain an anomaly in an interpretable and actionable way. Furthermore, as high-frequency meters become available in the future, it will be important to investigate streaming algorithms and real-time analytics for smart meter data. Finally, we are interested in developing a general time series analytics benchmark for a wider range of applications.

ACKNOWLEDGMENTS

This research was supported by the IBM Southern Ontario Smart Computing Innovation Platform (SOSCIP) and the CITIES project funded by the Innovation Fund Denmark (1035-00027B). We are grateful to Matt

Cheah for developing a preliminary implementation of the three-line algorithm using Apache Spark and evaluating its performance on a multicore server.

REFERENCES

J. M. Abreu, F. P. Camara, and P. Ferrao. 2012. Using pattern recognition to identify habitual behavior in residential electricity consumption. *Energy and Buildings*, 49:479–487.

G. Acs and C. Castelluccia. 2011. I have a DREAM (DiffeRentially privatE smArt Metering). In *Conf. on Information Hiding*, 118–132.

A. Albert, T. Gebru, J. Ku, J. Kwac, J. Leskovec, and R. Rajagopal. 2013. Drivers of variability in energy consumption. In *ECML-PKDD DARE Workshop on Energy Analytics*.

A. Albert and R. Rajagopal. 2013a. Building dynamic thermal profiles of energy consumption for individuals and neighborhoods. In *IEEE Big Data Conf.*, 723–728.

A. Albert and R. Rajagopal. 2013b. Smart meter driven segmentation: What your consumption says about you. *IEEE Transactions on Power Systems*, 4(28), 4019–4030.

E. Anderson and J. Tucek. 2010. Efficiency matters! *SIGOPS Operating Systems Review*, 44(1):40–45.

C. Anil. 2013. Benchmarking of data mining techniques as applied to power system analysis. Master's Thesis, Uppsala University.

O. Ardakanian, N. Koochakzadeh, R. P. Singh, L. Golab, and S. Keshav. 2014. Computing electricity consumption profiles from household smart meter data. In *EnDM Workshop on Energy Data Management*, 140–147.

M. Arlitt, M. Marwah, G. Bellala, A. Shah, J. Healey, and B. Vandiver. 2015. IoTA bench: An internet of things analytics benchmark. In *Proc. of the ACM/SPEC Int. Conf. on Performance Engineering.* 133–144.

B. J. Birt, G. R. Newsham, I. Beausoleil-Morrison, M. M. Armstrong, N. Saldanha, and I. H. Rowlands. 2012. Disaggregating categories of electrical energy end-use from whole-house hourly data. *Energy and Buildings* 50:93–102.

N. Bruno and S. Chaudhuri. 2005. Flexible database generators. In *Int. Conf. on Very Large Data Bases.* 1097–1107.

E. Buchmann, K. Bohm, T. Burghardt, and S. Kessler. 2013. Re-identification of smart meter data. *Pers. Ubiqit. Comput.* 17(4):653–662.

C. Chen and D. Cook. 2011. Energy outlier detection in smart environments. In *AAAI Workshop on Artificial Intelligence and Smarter Living: The Conquest of Complexity.*

G. Chicco, R. Napoli, and F. Piglione. 2006. Comparisons among clustering techniques for electricity customer classification. *IEEE Trans. on Power Systems*, 21(2):933–940.

F. Eichinger, P. Efros, S. Karnouskos, and K. Bohm. 2015. A time-series compression technique and its application to the smart grid. *VLDB Journal* 24(2):193–218.

Electric Power Research Institute (EPRI). 2013. *Big Data Survey Summary Report*

M. Espinoza, C. Joye, R. Belmans, and B. DeMoor. 2005. Short-term load forecasting, profile identification, and customer segmentation: A methodology based on periodic time series. *IEEE Trans. on Power Systems*, 20(3):1622–1630.

V. Figueiredo, F. Rodrigues, Z. Vale, and J. Gouveia. 2005. An electric energy consumer characterization framework based on data mining techniques. *IEEE Trans. on Power Systems*, 20(2):596–602.

M. Ghofrani, M. Hassanzadeh, M. Etezadi-Amoli, and M. Fadali. 2011. Smart meter based short-term load forecasting for residential customers. In *North American Power Symposium (NAPS'11).*

L. Gu, M. Zhou, Z. Zhang, M.-C. Shan, A. Zhou, and M. Winslett. 2015. Chronos: An elastic parallel framework for stream benchmark generation and simulation. In *IEEE Int. Conf. on Data Engineering.* 101–112.

J. M. Hellerstein, C. Re, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, and A. Kumar. 2012. The MADlib analytics library: Or MAD skills, the SQL. *Proc. of the VLDB Endowment*, 5(12):1700–1711.

R.-S. Jeng, C.-Y. Kuo, Y.-H. Ho, M.-F. Lee, L.-W. Tseng, C.-L. Fu, P.-F. Liang, and L.-J. Chen. 2013. Missing data handling for meter data management system. In *ACM Int. Conf. on Future Energy Systems.* 275–276.

E. Keogh and S. Kasetty. 2013. On the need for time series data mining benchmarks: A survey and empirical demonstration. *Data Mining and Knowledge Discovery (DMKD)*, 7(4):349–371.

S. Kessler, E. Buchmann, and K. Bohm. 2015. Deploying and evaluating pufferfish privacy for smart meter data. In *Proc. Int. Conf. on Ubiquitous Intelligence and Computing (UIC'15).*

X. Liu, L. Golab, W. Golab, and I. Ilyas. 2015a. Benchmarking smart meter data analytics. In *Int. Conf. on Extending Database Technology.* 285–396.

X. Liu, L. Golab, and I. Ilyas. 2015b. SMAS: A smart meter data analytics system. In *IEEE Int. Conf. on Data Engineering*. 1476–1479.

Y. Liu, S. Hu, T. Rabl, W. Liu, H.-A. Jacobsen, K. Wu, J. Chen, and J. Li. 2014. DGFIndex for smart grid: Enhancing hive with a cost-effective multidimensional range index. *Proc. of the VLDB Endowment* 7(13): 1496–1507.

D. Mashima and A. Cardenas. 2012. Evaluating electricity theft detectors in smart grid networks. In *Int. Conf. on Research in Attacks, Intrusions and Defenses (RAID'12)*, 210–229.

F. Mattern, T. Staake, and M. Weiss. 2010. ICT for green - how computers can help us to conserve energy. In *ACM Int. Conf. on Future Energy Systems*. 1–10.

A. J. Nezhad, T. K. Wijaya, M. Vasirani, and K. Aberer. 2014. SmartD: Smart meter data analytics dashboard. In *ACM Int. Conf. on Future Energy Systems*. 213–214.

T. Rasanen, D. Voukantsis, H. Niska, K. Karatzas, and M. Kolehmainen. 2010. Data-based method for creating electricity use load profiles using large amount of customer-specific hourly measured electricity use data. *Applied Energy*, 87(11):3538–3545.

B. A. Smith, J. Wong, and R. Rajagopal. 2012. A simple way to use interval data to segment residential customers for energy efficiency and demand response program targeting. In *ACEEE Summer Study on Energy Efficiency in Buildings*.

A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. 2009. Hive - A warehousing solution over a map-reduce framework. *Proc. of the VLDB Endowment* 2(2): 1626–1629.

G. Tsekouras, N. Hatziargyriou, and E. Dialynas. 2007. Two-stage pattern recognition of load curves for classification of electricity customers. *IEEE Trans. on Power Systems*, 22(3):1120–1128.

T. K. Wijaya, J. Eberle, and K. Aberer. 2013. Symbolic representation of smart meter data. In *EDBT Workshop on Energy Data Management (EnDM'13)*, 242–248.

M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. 2010. Spark: Cluster computing with working sets. In *USENIX Conf.*, 10.

M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. 2012. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. In *Proc. USENIX Conf. on Hot Topics in Cloud Computing*. 10.

Received August 2015; revised July 2016; accepted October 2016