Benchmarking Smart Meter Data Analytics

Xiufeng Liu, Lukasz Golab, Wojciech Golab and Ihab F. Ilyas
University of Waterloo, Canada
{xiufeng.liu,lgolab,wgolab,ilyas}@uwaterloo.ca

ABSTRACT

Smart electricity meters have been replacing conventional meters worldwide, enabling automated collection of fine-grained (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, but the focus thus far has been on what can be done with the data rather than how to do it efficiently. In this paper, we examine smart meter analytics from a software performance perspective. First, we propose a performance benchmark that includes common data analysis tasks on smart meter data. Second, since obtaining large amounts of smart meter data is difficult due to privacy issues, we present an algorithm for generating large realistic data sets 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). We compare the five platforms in terms of application development effort and performance on a multi-core machine as well as a cluster of 16 commodity servers. We have made the proposed benchmark and data generator freely available online.

1. INTRODUCTION

Smart electricity grids, which incorporate 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 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 aims to help utilities and consumers understand electricity consumption patterns, has become an active area in research and industry. According to a recent report, utility data analytics is already a billion dollar market and is expected to grow to nearly 4 billion dollars by year 2020 [16].

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 and provide personalized feedback to consumers on how to adjust their habits and reduce their bills, and design targeted engagement programs to clusters of similar consumers. However, the research focus has been on the insights 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 data sets. Thus, despite the increasing amounts of available data and the increasing number of potential applications[1], it is not clear how to build and evaluate a practical and scalable system for smart meter analytics. This is exactly the problem we study in this paper.

1.1 Contributions

We begin with a *benchmark* for comparing the performance of smart meter analytics systems. Based on a review of the related literature (more details in Section 2), we identified four 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 typically daily habits of consumers (e.g., by extracting consumption trends that occur at different times of the day regardless of the outdoor temperature) and 4) finding similar consumers (e.g., by running times series similarity search). These tasks involve aggregation, regression and time series analysis. Our benchmark includes a representative algorithm from each of these four sets.

Second, since obtaining smart meter data for research purposes is difficult due to privacy concerns, we present a *data generator* for creating large realistic smart meter data sets from a small seed of real data. The real data set we were able to obtain consists of only 27,000 consumers, but our generator can create much larger data sets 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, accompanied by MADLib [17], an in-database machine learning toolkit;

[1]See, e.g., a recent competition sponsored by the United States Department of Energy to create new apps for smart meter data: http://appsforenergy.challengepost.com.

10.5441/002/edbt.2015.34

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 [28]: a main-memory distributed data processing platform;

5. Hive [25]: a distributed data warehouse system built on top of Hadoop, with an SQL-like interface.

We report performance results on our real data set and larger realistic data sets 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 non-standard 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 Hive easier to use than Spark and not much slower. Spark and Hive are competitive with System C in terms of efficiency (throughput per server) for several of the workloads in our benchmark.

Our benchmark (i.e., the data generator and the tested algorithms) is freely available for download at https://github.com/xiufengliu. Due to privacy issues, we are unable to share the real data set or the large synthetic data sets based upon it. However, a smart meter data set has recently become available at the Irish Social Science Data Archive[2] and may be used along with our data generator to create large publicly available data sets for benchmarking purposes.

1.2 Roadmap

The remainder of this paper 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 our experimental results; and Section 6 concludes the paper with directions for future work.

2. RELATED WORK

2.1 Smart Meter Data Analytics

There are two broad areas of research in smart meter data analytics: those which use whole-house consumption readings collected by conventional smart meters (e.g., every hour) and those which use high-frequency consumption readings (e.g., one per second) obtained using specialized load-measuring hardware. We focus on the former in this paper, 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 end-users on reducing electricity consumption and saving money (see, e.g., [10, 21, 24]). Producer-oriented applications are geared towards 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., [1, 3, 5, 8, 12, 13, 14, 15, 22, 23]).

From a technical standpoint, both of the above classes of applications perform two types of operations: extracting representative features (see, e.g., [8, 10, 13, 14]) and finding similar consumers based on the extracted features (see, e.g., [1, 12, 23, 24, 26]). Household electricity consumption can be broadly decomposed into the temperature-sensitive component (i.e., the heating and cooling load) and the temperature-insensitive component (other appliances). Thus, representative features include those which measure the effect of outdoor temperature on consumption [4, 10, 23] and those which identify consumers' daily habits regardless of temperature [1, 8, 13], as well as those which measure the overall variability (e.g., consumption histograms) [3]. Our smart meter benchmark, which will be described in Section 3, includes four representative algorithms for characterizing consumption variability, temperate sensitivity, daily activity and similarity to other consumers.

We also point out recent work on smart meter data quality (specifically, handling missing data) [18], symbolic representation of smart meter time series [27], and privacy (see, e.g., [2]). These important issues are orthogonal to smart meter analytics, which is the focus of this paper.

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, e.g., PostgreSQL/MADLib [17]. 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 the above classes (details in Section 5).

Smart meter analytics software is currently offered by several database vendors including SAP[3] and Oracle/Data Raker[4], as well as startups such as Autogrid.com, C3Energy.com and OPower.com. 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 [20], but that work focuses on simple operational queries rather than the deep analytics that we address in this paper.

2.3 Benchmarking Data Analytics

There exist several database (e.g., TPC-C, TPC-H and TPC-DS) and big data[5] benchmarks, but they focus mainly on the performance of relational queries (and/or transactions) and therefore are not suitable for smart meter applications. Benchmarking time series data mining was discussed in [19]. 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 evaluated in [19]. Additionally, [19] evaluated standalone algorithms whereas we evaluate data analytics platforms. Furthermore, [7] benchmarked data mining operations for power system analysis. However, its 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. propose a benchmark for smart meter analytics that focuses on routine computations such as finding top customers and calculating monthly bills [9]. 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 data set.

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

[3]http://www.sap.com/pc/tech/in-memory-computing-hana/software/smart-meter-analytics/index.html

[4]http://www.oracle.com/us/products/applications/utilities/meter-data-analytics/index.html

[5]https://amplab.cs.berkeley.edu/benchmark

We also note that the TCP benchmarks include the ability to generate very large synthetic databases, and there has been some research on synthetic database generation (see, e.g., [11]), 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 primary goal of the benchmark is to measure the running time of a set of tasks that will be defined shortly. The input consists of n time series, each corresponding to one electricity consumer, in one or more text files. We assume that each time series contains hourly electricity consumption measurements (in kilowatt-hours, kWh) for a year, i.e., $365 \times 24 = 8760$ data points. For each consumption time series, we require an accompanying external temperature time series, also with hourly measurements.

For each task, we measure the running time on the input data set, both with a cold start (working directly from the raw files) and a warm start (working with data loaded into physical memory). In this version of the benchmark, we do not consider the cost of updates, e.g., adding a day's worth of new points to each time series. However, adding updates to the benchmark is an important direction for future work as read-optimized data structures that help improve running time may be expensive to update.

Utility companies may have access to additional data about their customers, e.g., 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 the smart meter time series and publicly-available weather data.

We now discuss the four analysis tasks included in the proposed benchmark.

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. The x-axis in the histogram denotes various hourly consumption ranges and the y-axis is the frequency, i.e., 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 ten 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 the recently-proposed algorithm from [10] for the benchmark, to which we refer as the 3-line algorithm. Consider a consumption-temperature scatter plot for a single consumer shown in Figure 1 (the actual 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.

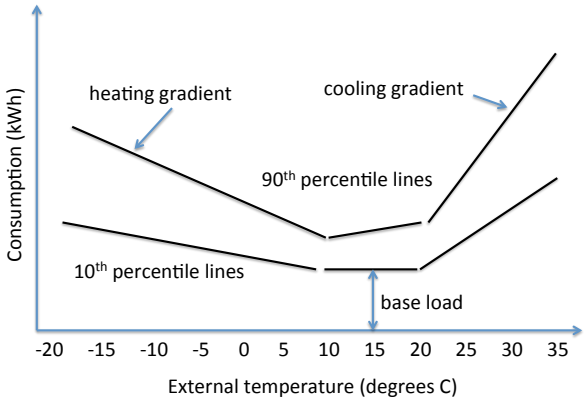


Figure 1: Example of the 3-line regression model.

As shown in Figure 1, the 3-line algorithm extracts useful information for customer feedback. For instance, the slopes (gradients) of the left and right 90th percentile lines correspond to the 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 [8, 13]. The idea behind this algorithm is illustrated in Figure 2. 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 2. Thus, for each consumer, the output consists of a vector of 24 numbers, denoting the expected consumption at different hours of the day due solely to the occupants' daily habits and not affected by temperature.

For each consumer and each hour of the day, the PAR algorithm fits an auto-regressive model, which assumes that the electricity consumption at that hour of the day is a linear combination of the consumption at the same hour over the previous p days (we use $p = 3$, as in [8]) and the outdoor temperature. Thus, it groups the input data set by consumer and by hour, and computes the coefficients of the auto-regressive model for each group.

3.4 Similarity Search

The final task is to find groups of similar consumers. Customer segmentation is important to utilities so they can determine how many distinct groups of customers there are and design targeted energy-saving campaigns for each group. Rather than choosing

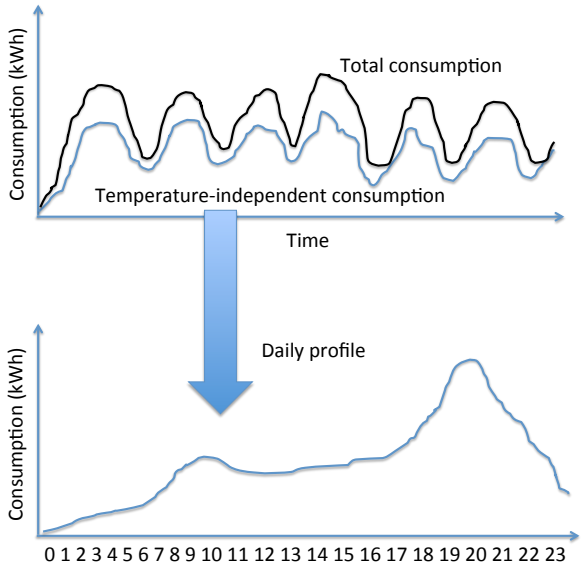


Figure 2: Example of a daily profile.

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, i.e., $\frac{X \cdot Y}{||X|| * ||Y||}$.

3.5 Discussion

To recap, the proposed benchmark consists of 1) the consumption histogram, 2) the 3-line algorithm for understanding the effect of external temperature on consumption, 3) the periodic autoregression (PAR) algorithm to extract typical daily profiles and 4) the time series similarity search to find similar consumers. 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. While many more smart meter analytics algorithms have been proposed, we believe the four tasks we have chosen accurately represent a variety of fundamental computations that might be used to extract insights from smart meter data.

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 3-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. Thus, the proposed benchmark tests the ability to extract different segments of the data and run various statistical and time series operations.

4. THE DATA GENERATOR

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 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 2) and temperature regression lines (recall Figure 1) for each consumer therein. To generate a new time series, we take the daily activity pattern from a randomly-selected consumer in the real data set, 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 data set, and we then re-aggregate 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 3 illustrates the proposed data generator. As a preprocessing step, we use the PAR algorithm from [13] to generate daily profiles for each consumer in the seed data set. 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. We also run the 3-line algorithm and record the heating and cooling gradients for each consumer.

Now, creating a new time series 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[6]. 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.

5. EXPERIMENTAL RESULTS

This section presents our experimental results. We start with an overview of the five platforms in which we implemented the proposed benchmark (Section 5.1) and a description of our experimental environment (Section 5.2). Section 5.3 then discusses our experimental findings using a single multi-core 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 multi-threaded 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. We conclude with a summary of lessons learned in Section 5.5.

5.1 Benchmark Implementation

We first introduce the five platforms in which we implemented the proposed benchmark. Whenever possible, we use native statistical functions or third-party libraries. Table 1 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

[6]In our experiments, we used the temperature time series corresponding to the southern-Ontario city from which we obtained the real data set.

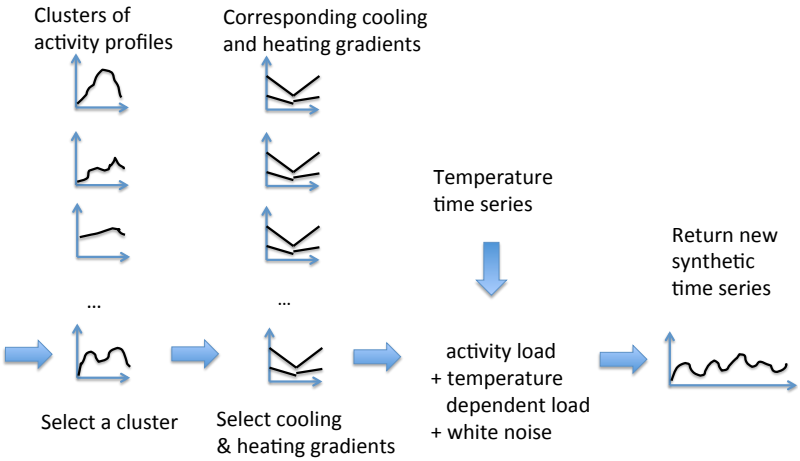


Figure 3: Illustration of the proposed data generator.

Table 1: 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

the built-in histogram, quantile, regression and PAR functions, and we implemented our own (very simple) 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 [17], which is an open-source platform for in-database machine learning. As we will explain later in this section, we tested two ways of storing the data: one measurement per row, and one customer per row with all the measurements for this customer stored in an array. Similarly to Matlab, everything we need except cosine similarity 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 use the default settings for PostgreSQL[7].

Next, we use System C as an example of a state-of-the-art commercial system. It is a main-memory column store geared towards time series data. System C maps tables to main memory to improve I/O efficiency. In particular, at loading time, all the files 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 [28] as an example of an open-source distributed data processing platform. Spark reports improved performance on machine learning tasks over standard Hadoop/MapReduce due to better use of main memory [28]. 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.

Finally, we test another distributed platform, Hive [25], which is built on top of Hadoop and includes a declarative SQL-like interface. 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 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. However, since we did not conduct a user study, these programmer effort observations should be treated as anecdotal.

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

5.2 Experimental Environment

We run each of the four algorithms in the benchmark using each of the five platforms discussed above, and measure the running times and memory consumption. We use the following two testing environments.

- Our server has an Intel Core i7-4770 processor (3.40GHz, 4 Cores, hyper-threading is enabled, two hyper-threads per core), 16GB RAM, and a Seagate hard drive (1TB, 6 GB/s, 32 MB Cache and 7200 RPM), running Ubuntu 12.04 LTS with 64bit Linux 3.11.0 kernel. PostgreSQL 9.1 is installed with the settings "shared _buffers= 3072MB, temp_ buffers= 256MB, work_ mem=1024MB, checkpoint_segments =64" and default values for other configuration parameters.

- 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, 6 cores per socket, and two hyper-threads per core), 60GB RAM, running 64bit Linux with kernel version 2.6.32. The nodes

[7]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.

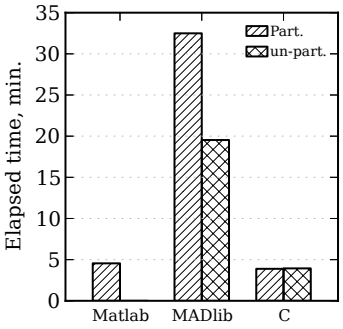


Figure 4: Data loading times, 10GB real dataset.

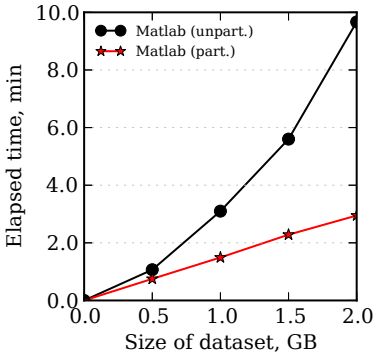


Figure 5: Impact of data partitioning on analytics, 3-line algorithm.

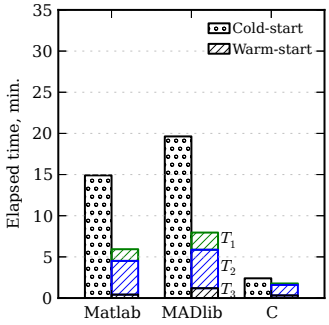


Figure 6: Cold-start vs. warm-start, 3-line algorithm, 10GB real dataset.

are connected via gigabit Ethernet, and a working directory is NFS-mounted on all the nodes.

Our real data set consists of $n = 27,300$ electricity consumption time series, each with hourly readings for over a year. We also obtained the corresponding temperature time series. The total data size is roughly 10 GB. We also use the proposed data generator to create larger synthetic data sets of size up to one Terabyte (which corresponds to over two million time series), and experiment with them in Section 5.4.

5.3 Single-Server Results

We begin by comparing Matlab, MADLib and System C running on a single multi-core server, using the real 10GB dataset.

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 4 shows the time it took to load our 10-GB real data set into the three systems tested in this section, both in a partitioned (one file per consumer, abbreviated "part.") and non-partitioned (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. 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 data set into small files.

Once data are loaded into tables in PostgreSQL or System C, the number of input files no longer matters. However, Matlab reads data directly from files, so the goal of our next experiment is to investigate the performance of analytics in Matlab given the two partitioning strategies discussed above. Figure 5 shows the running time of the 3-line algorithm using Matlab on (partitioned and non-partitioned) subsets of our real data sets sized from 0.5 to 2 GB. (We observed similar trends when running the other algorithms in the 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 3-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 vs. Warm Start

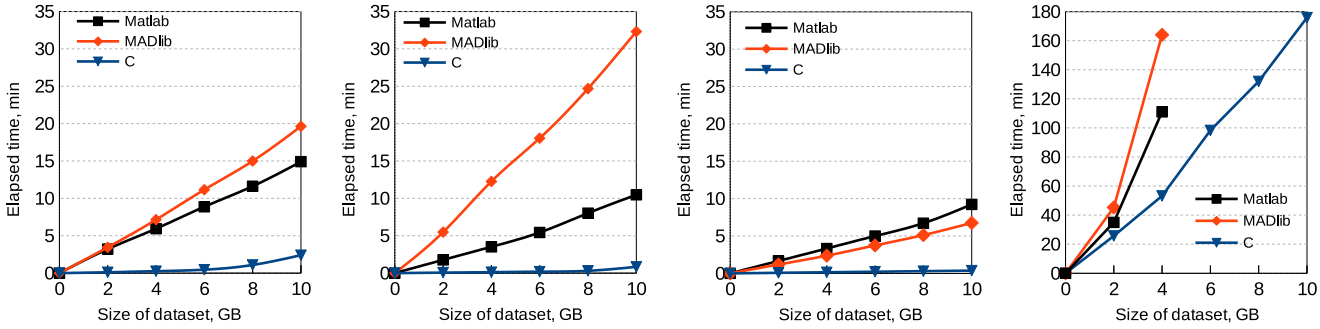
Next, we measure the time it takes each system to load data into main memory before executing the 3-line algorithm (we saw similar trends when testing other algorithms from the benchmark). In *cold-start*, we record the time to read the data from the underlying database or filesystem and run the algorithm. 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 6 shows the results on the real data set. 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, i.e., the time to run least-squares linear regression, is the most costly component of the 3-line algorithm.

Figure 6 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 confirm this hypothesis, we measured the running time of multiplying two randomly-generated 4000x4000 floating-point matrices in Matlab and System C. Indeed, Matlab took under a second, while System C took over 5 seconds.

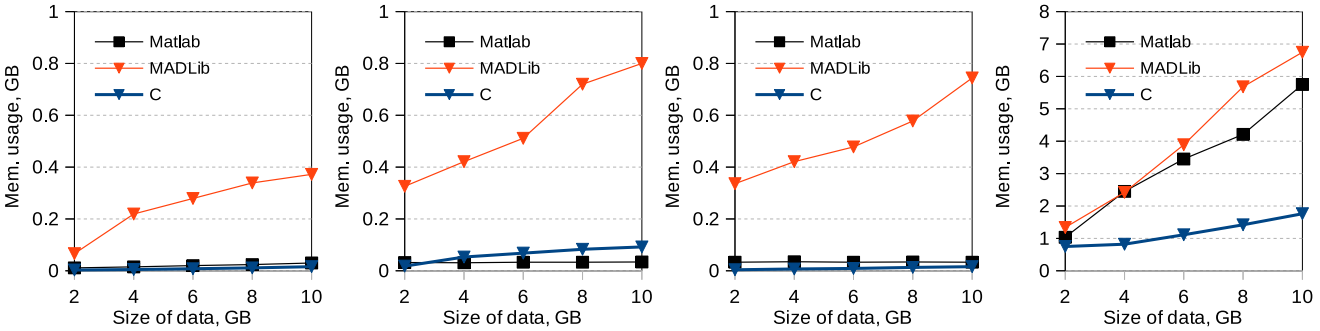
5.3.3 Single-Threaded Results

We now measure the cold-start running times of each algorithm in single-threaded mode (i.e., no parallelism). System C has configuration parameters that govern the level of parallelism, while for Matlab, we start a single instance, and for MADLib, we establish a single database connection. We use subsets of our real data sets with sizes between 2 and 10 GB for this experiment. The running time results are shown in Figure 7 for 3-line, PAR, his-



(a) 3-line (b) PAR (c) Histogram (d) Similarity

Figure 7: Single-threaded execution times of each algorithm using each system.



(a) 3-line (b) PAR (c) Histogram (d) Similarity

Figure 8: Memory consumption of each algorithm using each system.

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,...]
...		

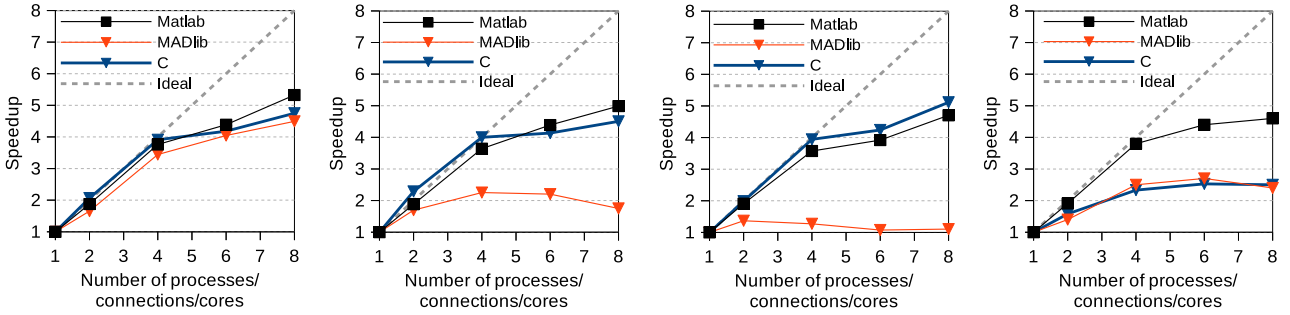
Figure 9: Two table layouts for storing smart meter data in PostgreSQL.

togram 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 data sets 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 for 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 3-line, PAR and similarity search.

Figure 8 shows the corresponding memory consumption of each algorithm for each platform; the plots correspond to running the "free -m" command every five 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.

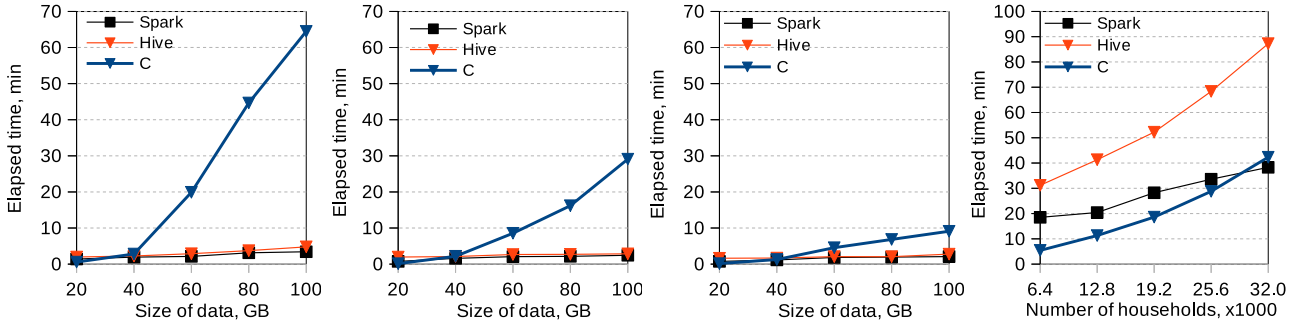
In terms of the tested algorithms, 3-line has the lowest memory usage since it only requires the 10th and 90th percentile data points to compute the regression lines, not the whole time series. The memory footprint of PAR and histogram construction is higher because they both require the whole time series. 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 9 shows the conventional row-oriented schema for smart meter data which 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 time time series is stored as a separate row, and a B-tree index is built on the household ID to speed up the extraction of all the data for a given consumer. Table 2 in Figure 9 stores one row for each consumer (household) 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 3-line on the whole 10 GB data set went down from 19.6 minutes to 11.3 minutes, which is faster than Matlab and Spark but still much slower than System C (recall the leftmost plot in Figure 7). 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 6400



(a) 3-line (b) PAR (c) Histogram (d) Similarity

Figure 10: Speedup of execution time on a single multi-core server using the 10GB real dataset.



(a) 3-line algorithm (b) PAR algorithm (c) Histogram construction (d) Similarity search

Figure 11: Execution times using large synthetic data sets.

households, which works out to about 2 GB) went down from 58.3 to 40.5 minutes. Finally, 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 *Multi-Threaded Results*

We now evaluate the ability of the tested platforms to take advantage of parallelism. Our server has 4 cores with two hyper-threads per core and so we vary the number of processes from 1 to 8. Again, we can do so directly in System C, but we need to manually run multiple instances of Matlab and start up multiple database connections in MADLib. The histogram, 3-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.

Figures 10(a)–10(d) show the speedup obtained by increasing the number of threads from 1 to 8 for each algorithm. Again, we continue to use the 10-GB real data set. 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 hyper-threads. Matlab appears to scale better than MADLib, but this may be an artifact of how we simulate parallelism for these two platforms: Matlab instances effectively 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 a single table.

5.4 Cluster Results

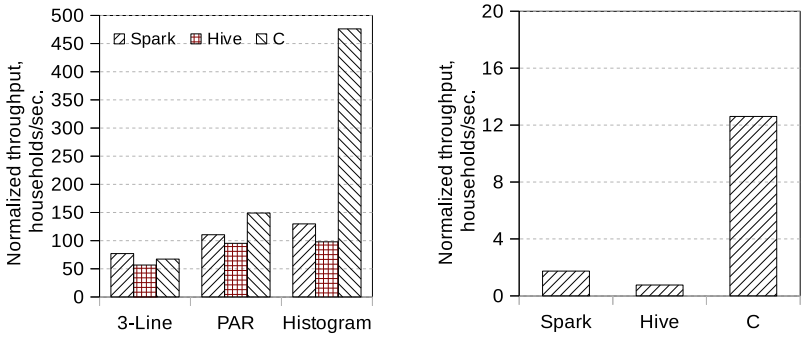
We now focus on the performance of Spark and Hive on a cluster using large synthetic data sets. We set the number of parallel executors for Spark and the number of MapReduce tasks for Hive to be up to 12 per node, which is the number of physical cores[8].

5.4.1 *System C vs. 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 data sets 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 hyper-threads) but we run Spark and Hive on the cluster. Nevertheless, the results are interesting.

Figure 11 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

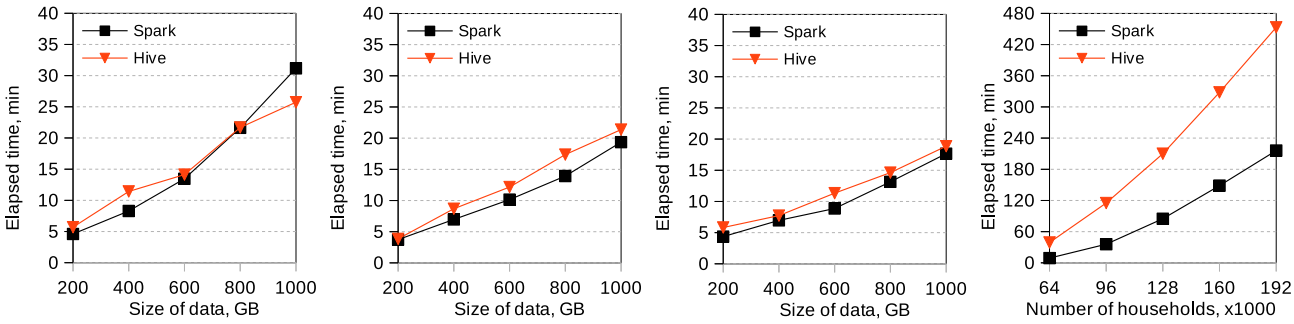
[8]We experimented with different values of these parameters and found that Spark was not sensitive to the number of parallel executors while Hive generally performed better with more MapReduce tasks up to a certain point.



(a) 3-Line, PAR and Histogram construction; 100GB data set

(b) Similarity search, 32k households

Figure 12: A comparison of throughput per server of System C, Spark and Hive.



(a) 3-Line (b) PAR (c) Histogram (d) Similarity

Figure 13: Execution times using the first data format in Spark and Hive.

is also very good.

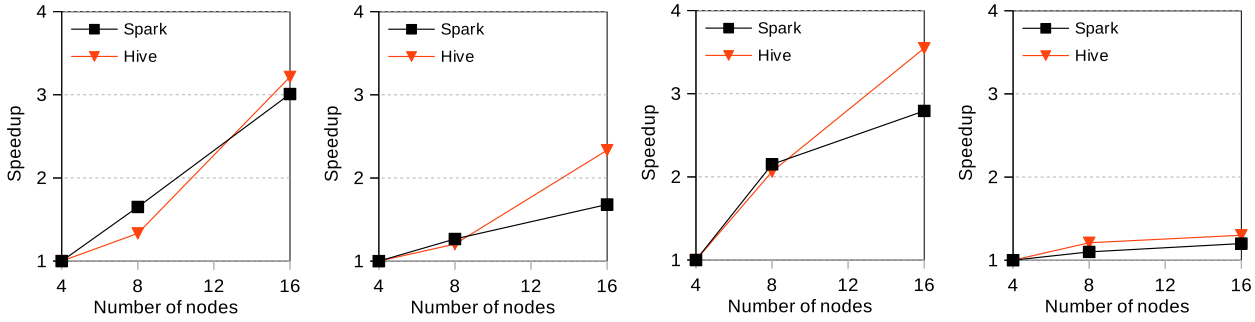
Figure 12 illustrates another way of comparing the three systems that is more fair. Part (a) shows the throughput, for 3-Line, PAR and histogram construction, in terms of how many households can be handled per second *per server* when using the 100GB synthetic data set. That is, we divide the total throughput 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 3-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 vs. 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 data sets up to a Terabyte. 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) 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.

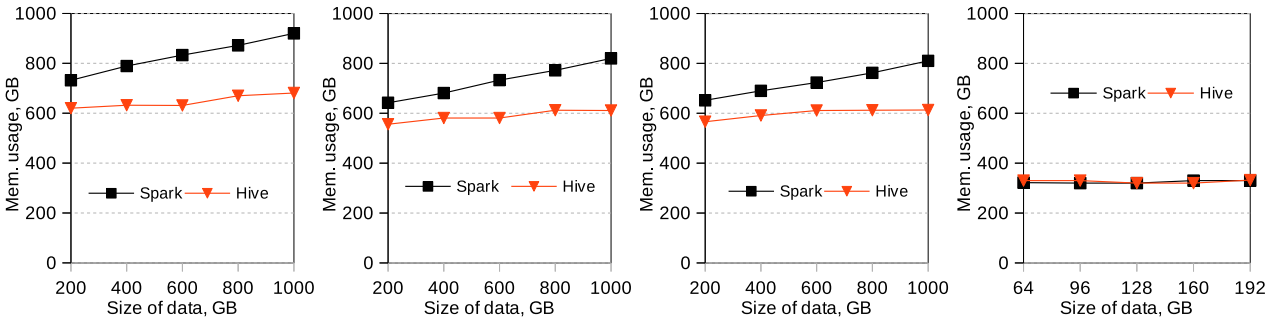
In Hive, we use three types of user-defined functions with the three file formats: generic UDF (user defined function), UDAF (user defined aggregation function) and UDTF (user defined table function). UDF and UDTF typically run at the map side for the scalar operations on a row, while UDAF runs at the 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 13 shows the execution time of the four tested algorithms on various data set sizes up to a Terabyte. 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 3-Line construction as the data size grows. Figure 14 shows the speedup relative to using only 4 out of 16 worker nodes for the Terabyte data set, 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 15 shows the memory usage as a function of the data set size, computed the same way as in Figure 8. Spark uses more memory than Hive, especially as the data size increases. As for the different algorithms, 3-Line is



(a) 3-Line (b) PAR (c) Histogram (d) Similarity, 64k households

Figure 14: Speedup obtained using the first data format in Spark and Hive.



(a) 3-Line (b) PAR (c) Histogram (d) Similarity

Figure 15: Memory consumption of each algorithm in Spark and Hive.

the most memory-intensive because it requires temperature data in addition to smart meter data.

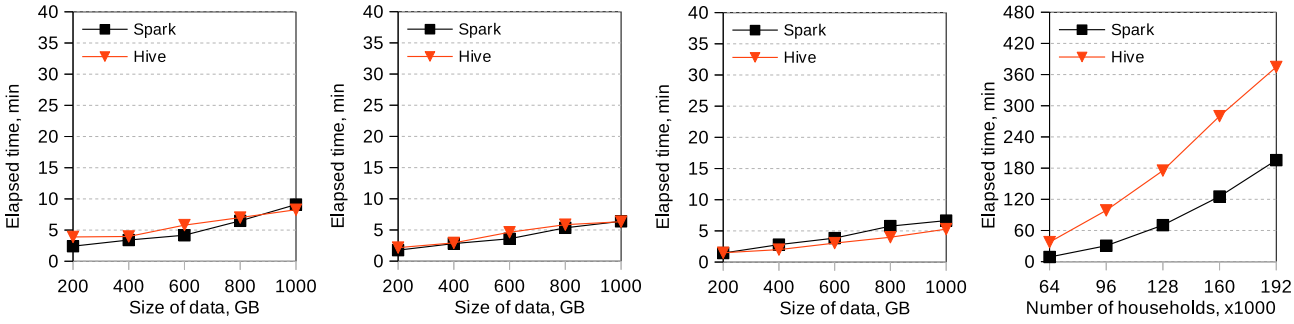
Second data format. Figure 16 and 17 show the execution times and the speedup, respectively, with one time series per line. For 3-Line, PAR and histogram construction, we do not require a reduce step. Therefore the running times are lower than for the first data format, in which a single time series 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 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 pair-wise 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 households and find the top-k most similar consumers.

Third data format. Here, we only use the 100GB data set 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. Figure 18 and Figure 19 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.

5.5 Lessons Learned

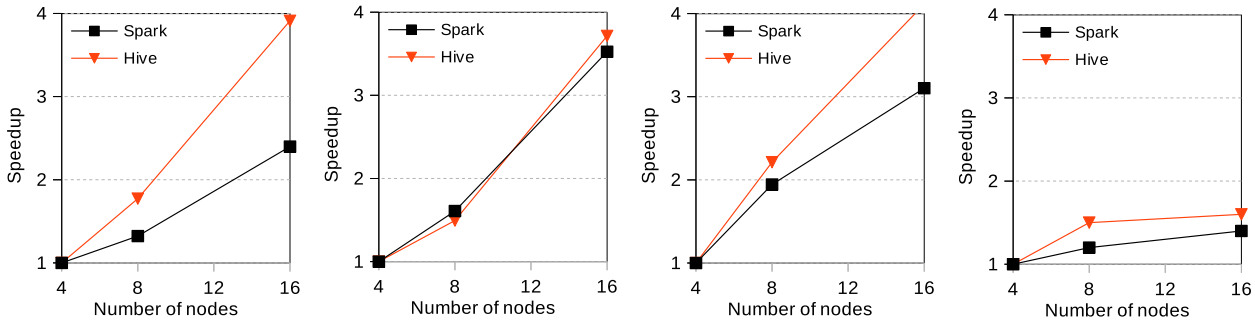
Our main finding is that System C, which is a commercial main-memory column store, is the best choice for smart meter analytics in terms of performance, provided that the resources of a single machine are sufficient. However, System C lacks a built-in machine learning toolkit and therefore we had to invest significant programming effort to build efficient analytics applications on top of it. On the other hand, Matlab and MADLib are likely to be more programmer-friendly but slower. Furthermore, 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.

As for the two distributed solutions, Spark was slightly faster but Hive scaled slightly better as we increased the number of worker nodes. Moreover, we found Hive easier to use due to its DBMS-like features and a declarative language. Furthermore, we showed that the choice of data format matters; we obtained 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 3-line and PAR, whereas cluster computing frameworks in general are known to suffer from poor efficiency compared to centralized systems [6].



(a) 3-Line (b) PAR (c) Histogram (d) Similarity

Figure 16: Execution times using the second data format in Spark and Hive.



(a) 3-Line (b) PAR (c) Histogram (d) Similarity, 64k households

Figure 17: Speedup obtained using the second data format in Spark and Hive.

6. CONCLUSION AND FUTURE WORK

Smart meter data analytics is an important new area of research and practice. In this paper, we studied smart meter analytics from a software performance perspective. We proposed a performance benchmark for smart meter analytics consisting of four common tasks, and presented a data generator for creating very large smart meter data sets. We implemented the proposed benchmark using five state-of-the-art data processing platforms and found that a 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 not much slower. Compared to centralized solutions, we found Hive and Spark competitive in terms of efficiency for CPU-intensive data-parallel workloads (3-line and PAR).

We are currently building a smart meter analytics system that includes the four algorithms from the proposed benchmark and many more. As part of this ongoing project, we are investigating new ways of improving the efficiency and effectiveness of smart meter data mining algorithms, including parallel implementation. Another interesting direction for future work is to investigate real-time applications using high-frequency smart meters (which are not yet widely available, but are likely to become cheaper and more common in the future), such as alerts due to unusual consumption readings, using data stream processing technologies. Finally, we are interested in developing a general time series analytics benchmark for a wider range of applications.

7. ACKNOWLEDGEMENTS

We are grateful to Matt Cheah for developing a preliminary implementation of the 3-line algorithm using Apache Spark and evaluating its performance on a multi-core server.

8. REFERENCES

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

[2] G. Acs and C. Castelluccia, I have a DREAM (DiffeRentially privatE smArt Metering), in Conf. on Information Hiding, 118-132, 2011.

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

[4] A. Albert and R. Rajagopal, Building dynamic thermal profiles of energy consumption for individuals and neighborhoods, in IEEE Big Data Conf., 723-728, 2013.

[5] A. Albert and R. Rajagopal, Smart meter driven segmentation: what your consumption says about you. IEEE Transactions on Power Systems, 4(28), 2013.

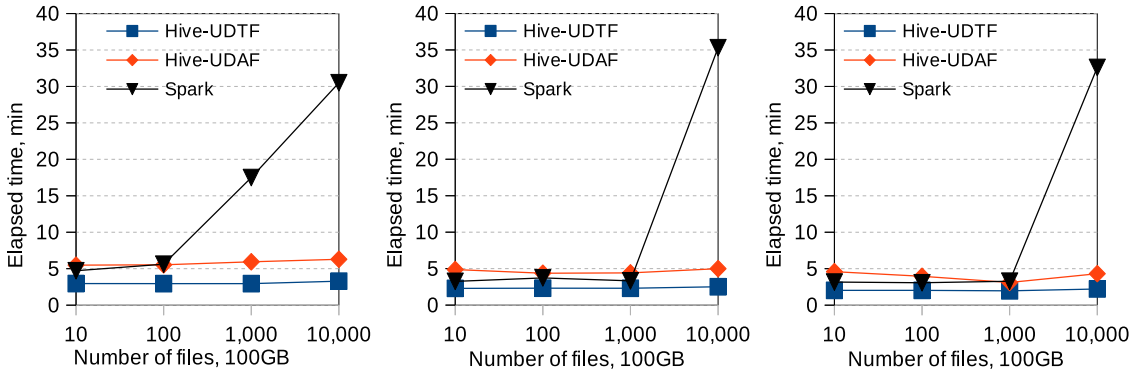
[6] E. Anderson and J. Tucek, Efficiency Matters!, SIGOPS Operating Systems Review, 44(1):40-45, 2010.

[7] C. Anil, Benchmarking of Data Mining Techniques as Applied to Power System Analysis, Master's Thesis, Uppsala University, 2013.

[8] O. Ardakanian, N. Koochakzadeh, R. P. Singh, L. Golab, and S.Keshav, Computing Electricity Consumption Profiles from Household Smart Meter Data, in EnDM Workshop on Energy Data Management, 140-147, 2014.

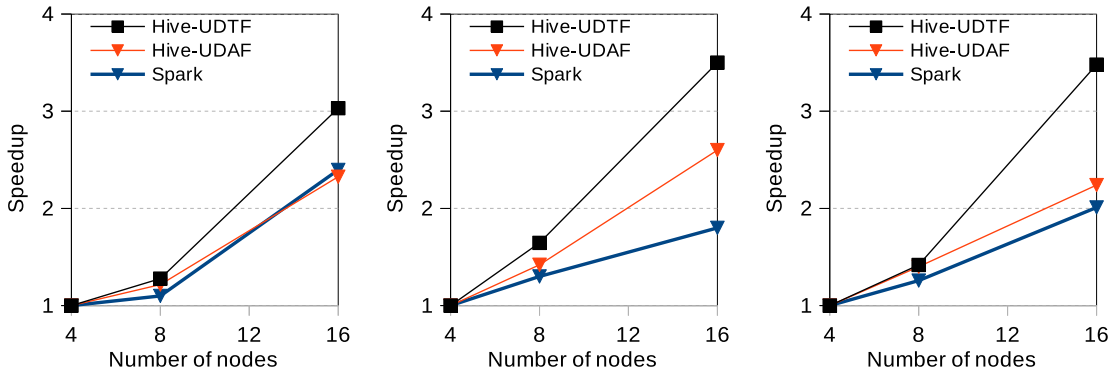
[9] M. Arlitt, M. Marwah, G. Bellala, A. Shah, J. Healey, B. Vandiver, IoTA bench: an Internet of Things Analytics benchmark, HP Laboratories Technical Report, HPL-2014-75.

[10] B. J. Birt, G. R. Newsham, I. Beausoleil-Morrison, M. M. Armstrong, N. Saldanha, and I. H. Rowlands, Disaggregating



(a) 3-Line (b) PAR (c) Histogram

Figure 18: Execution times using the third data format in Spark and Hive.



(a) 3-Line (b) PAR (c) Histogram

Figure 19: Speedup obtained using the third data format in Spark and Hive, 100 files, 1GB per file.

Categories of Electrical Energy End-use from Whole-house Hourly Data, Energy and Buildings, 50:93-102, 2012.

[11] N. Bruno and S. Chaudhuri, Flexible database generators, in VLDB, 1097-1107, 2005

[12] G. Chicco, R. Napoli, and F. Piglione, Comparisons among Clustering Techniques for Electricity Customer Classification, IEEE Trans. on Power Systems, 21(2):933-940, 2006.

[13] M. Espinoza, C. Joye, R. Belmans, and B. DeMoor, 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, 2005.

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

[15] M. Ghofrani, M. Hassanzadeh, M. Etezadi-Amoli and M. Fadali, Smart Meter Based Short-Term Load Forecasting for Residential Customers, North American Power Symposium (NAPS), 2011.

[16] Greentech Media Research, The Soft Grid 2013-2020: Big Data & Utility Analytics for Smart Grid, http://www.greentechmedia.com/research/report/the-soft-grid-2013.

[17] J. M. Hellerstein, C. Re, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, and A. Kumar, The MADlib Analytics Library: or MAD Skills, the SQL, PVLDB, 5(12):1700-1711, 2012.

[18] R.-S. Jeng, C.-Y. Kuo, Y.-H. Ho, M.-F. Lee, L.-W. Tseng, C.-L. Fu, P.-F. Liang, L.-J. Chen, Missing Data Handling for Meter Data Management System, in e-Energy Conf., 275-276, 2013.

[19] E. Keogh, and S. Kasetty, On the Need for Time Series Data Mining Benchmarks: A Survey and Empirical Demonstration, Data Mining and Know. Disc. (DMKD), 7(4):349-371, 2003.

[20] Y. Liu, S. Hu, T. Rabl, W. Liu, H.-A. Jacobsen, K. Wu, J. Chen, J. Li, DGFIndex for Smart Grid: Enhancing Hive with a Cost-Effective Multidimensional Range Index. PVLDB 7(13): 1496-1507, 2014.

[21] F. Mattern, T. Staake, and M. Weiss, ICT for green - How computers can help us to conserve energy, in e-Energy Conf., 1-10, 2010.

[22] A. J. Nezhad, T. K. Wijaya, M. Vasirani, K. Aberer, SmartD: smart meter data analytics dashboard, in e-Energy Conf., 213-214, 2014.

[23] T. Rasanen, D. Voukantsis, H. Niska, K. Karatzas and M. Kolehmainen, 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, 2010.

[24] B. A. Smith, J. Wong, and R. Rajagopal, 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, 2012.

[25] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, R. Murthy, Hive - A Warehousing Solution Over a Map-Reduce Framework. PVLDB 2(2): 1626-1629, 2009.

[26] G. Tsekouras, N. Hatziargyriou, and E. Dialynas, Two-stage Pattern Recognition of Load Curves for Classification of Electricity Customers, IEEE Trans. on Power Systems, 22(3):1120-1128, 2007.

[27] T. K. Wijaya, J. Eberle and K. Aberer, Symbolic representation of smart meter data, in EnDM Workshop on Energy Data Management, 242-248, 2013.

[28] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, Spark: Cluster Computing with Working Sets, in USENIX Conf., 10, 2010.