# Efficient Incremental Smart Grid Data Analytics

David Xi Cheng Wojciech Golab Paul A. S. Ward

Department of Electrical and Computer Engineering
University of Waterloo, Canada
{david.cheng, wgolab, pasward}@uwaterloo.ca

## ABSTRACT

Analytical computations over energy data are gaining popularity thanks to the growing adoption of smart electricity meters. Computations in this context range from seemingly straightforward tasks such as calculating monthly bills based on time-of-use pricing, to elaborate model building for predictions and recommendations in an effort to reduce peak demand. While research in this promising area is progressing steadily, published algorithms and prototypes have largely avoided the important practical question of how to deal efficiently with the incremental nature of energy data, for example per-hour readings produced by smart electricity meters. As a stepping stone towards a comprehensive solution to this problem, we investigate incremental techniques for disaggregating different categories of energy consumption, such as base load versus activity load, from hourly smart meter data using the popular "three-line model" of Birt et al. Our software prototype, called Insparq, exhibits speedups in excess of 2x for data sets up to tens of GB in size, compared to a naive implementation on top of a conventional scalable batch processing framework.

## Keywords

Smart meters, energy data modeling, incremental analytics, Apache Spark, cluster computing, batch processing.

## 1. INTRODUCTION

Analytical computations over energy data are gaining popularity thanks to the growing adoption of smart electricity meters, which are critical data sources in the emerging smart grid. A smart meter reports energy usage data for a given customer at regular intervals, such as every hour, and transmits it to the utility company for storage and analysis. Some of the most fundamental analytic tasks are carried out on a subset of the latest data, for example, calculating a customer's monthly bill based on time-of-day pricing, or identifying the top energy consumers in the past 24 hours. These tasks are simple enough that they can be expressed using SQL and solved using existing database tools (e.g., [5]) without the need for specialized techniques. In contrast, more elaborate analytic computations proposed in the context of predictions and recommendations consider patterns in energy data over longer timescales [7, 15–17], and therefore operate on potentially much larger data sets. For example, a model of the relationship between external temperature and energy consumption for a customer may by constructed over a year, or perhaps even several years, of data, and yet the utility company may want to recompute the model at monthly, weekly, or even daily intervals. Thus, successive repetitions of an analytic task may operate on largely overlapping data sets.

In this paper we focus on techniques for incremental computation over energy data that integrate new incoming data efficiently with existing data. Although most of the analytic tasks in this context are easy to parallelize because they deal with each customer independently, efficient solutions are hard to obtain for several reasons. First, the style of computation lies between conventional batch processing (e.g, using MapReduce [9]) and stream processing techniques (e.g., using Apache Spark or Storm [4, 26, 27]), neither of which is a good fit for incremental tasks. Batch processing accommodates massive data sets but discards intermediate state when a job runs to completion, whereas stream processing techniques operate on small windows of data and tend to maintain intermediate state continuously in main memory, making them very resource-intensive. Second, the few available systems that manage intermediate state between execution of jobs (e.g., [6, 10, 18, 20, 22]) provide a variety of novel abstractions for manipulating data but do not automate the translation of complex tasks onto those abstractions.

As a stepping stone towards a comprehensive framework for incremental energy analytics, we consider in this paper the specific task of disaggregating energy data into categories, such as base versus activity load, using the "three-line model" of Birt et al. [7]. This task, described in more detail in Section 2, involves several stages:

1. grouping energy consumption readings into bins by temperature
2. computing energy consumption percentiles for each bin
3. fitting regression lines over temperature-percentile data

The model must be computed over a long time interval, such as a year or more, to capture a range of outdoor temperatures. However, updates to the model may be based on a

©2016, Copyright is with the authors. Published in the Workshop Proceedings of the EDBT/ICDT 2016 Joint Conference (March 15, 2016, Bordeaux, France) on CEUR-WS.org (ISSN 1613-0073). Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0

day, week, or month of new data that exhibits relatively narrow temperature variations. For example, the temperature bins for a household in Waterloo may range from $-30°C$ to $35°C$ on an annual basis whereas data collected in the month of September typically varies from $10°C$ to $25°C$. As a result, more than three quarters of the bins determined in stage (1) and processed in stage (2), corresponding mostly to low-to-mid temperatures, are unaffected by the addition of new data.[1] This, in turn, means that the regression line fitted over the low-temperature data remains unaffected in stage (3), and hence the heating gradient determined from this line segment is unchanged.

Our main contribution with respect to incremental computation of the three-line model is two-fold:

- We present Insparq, a distributed computation framework that provides a flexible incremental computation model. We designed and implemented Insparq on top of Spark [26], a state of the art data-parallel processing framework, along with a number of other software components including Parquet [2], Zookeeper [12] and Avro [1].
- We compare the performance of an incremental three-line model implementation in Insparq with a naive batch-oriented implementation that runs directly on top of Spark, and show performance gains of up to 5x on data sets tens of GB in size.

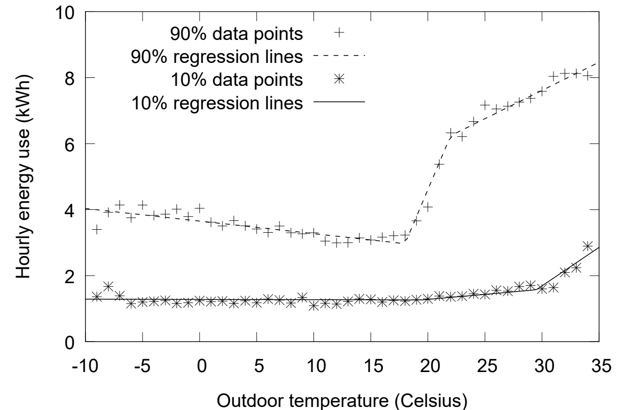
## 2. BACKGROUND: THREE-LINE MODELING & CHALLENGES

In this section we review in greater detail the three-line model of Birt et al. [7], which is used to disaggregate energy data into different categories, as well as to determine the effect of external temperature on energy consumption. The model is computed independently for each customer using fine-grained (e.g., hourly) smart-meter data coupled with external temperature readings. The computation comprises the three stages outlined earlier in Section 1:

1. Energy-temperature pairs are grouped into bins according to the temperature value rounded to the nearest integer. Only bins with sufficient data (at least 20 points) are used in subsequent stages.
2. The 10th and 90th percentile energy consumption is calculated for each bin. The 10th percentile represents the *base load*, which is the energy demand of appliances that operate continuously, such as a fridge, water heater, furnace, air conditioner, or security alarm. The difference between the 90th and 10th percentile energy consumption is the *activity load*, which is the additional demand due to human activity, for example from lighting, cooking, showering, or watching TV.
3. Regression lines are computed separately for the 10th and 90th percentile data. In each case the bins are partitioned into three contiguous sections by temperature, and one line is fitted per region. The first section starts at $-15°C$ and continues up to at least $10°C$ and at most $20°C$. The second (respectively, third) section starts where the first (respectively, second) one ends, encompasses at least five temperature bins, and spans at most $15°C$ in total. Regression lines are then fitted for all possible combinations of section boundaries, and the combination that minimizes the total root-mean-square error (RMSE) is chosen. In the event that the piecewise-linear fit is discontinuous, a second stage of line fitting is applied that pins the regression lines to specific points chosen from a $9 \times 9$ grid around the discontinuity at each section boundary. A total of $(9^2)^2 = 6561$ combinations of such points are tested, and the final three-line model is chosen by once again minimizing the RMSE. An example of the final fit is shown in Figure 1.

[1] A similar argument applies to the deletion of old data, which is outside the scope of this paper.

**Figure 1: Example of the three-line model.**




From an implementation standpoint, the first stage of the model computation is the most I/O-intensive as it requires scanning over the entire dataset, whereas the second and third stage involve mostly CPU. In particular, the amount of data involved in the regression computations is fairly small, with only one point per temperature bin at each percentile, and thousands of regression lines. An incremental implementation of the algorithm stands to improve performance in terms of both I/O and CPU by addressing the following points:

- **Redundant I/O.** Energy-temperature pairs for a customer need to be loaded from storage only when an update occurs to a particular bin on arrival of new data. For the remaining bins, it suffices to load the percentile data directly, which is much more compact.
- **Redundant arithmetic operations.** When new energy readings are added to a particular temperature bin, the percentiles can be recomputed efficiently if the data in each bin are sorted, which makes it possible to combine new data with old data using merge sort. Furthermore, regression lines can be recomputed efficiently by retaining intermediate data, such as sums of squares, over the data points.

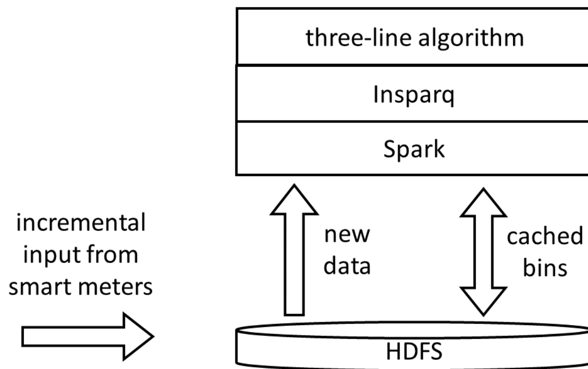
## 3. DESIGN AND IMPLEMENTATION

Insparq is an API layer implemented on top of Apache Spark to facilitate incremental computations, as illustrated

in the architectural diagram presented in Figure 2. In this section we describe the details of how the framework loads and stores intermediate state in a computation, as well as how this state is merged with new inputs. In the context of three-line modeling, the framework provides the following features:

- An abstraction of incremental storage that supports the automated loading and saving of intermediate values, on which developers may implement any data combination and transformation logics. Insparq maintains the intermediate state using an off-the-shelf distributed file system.
- An abstraction of linear regression, and an implementation of the incremental version of the least squares technique.
- An abstraction of dataset representation inside of Insparq that encapsulates a plain list or Spark Resilient Distributed Dataset (RDD) in an incremental data structure. Multiple datasets may be represented in the form of *IncrementalDataProvider* that distinguishes between the base and incremented data of a particular list.

**Figure 2: Architectural diagram of Insparq.**




## 3.1 Design Details

Before we dive into the implementation details, we first discuss various open source projects incorporated in our system, as well as the rationale for their use.

- Zookeeper [12] is a distributed coordination service. In our design, Zookeeper is used to implement a Read-Write lock for file operations that ensures only one file is written to the file system among distributed nodes, and that files are read only in the absence of writers.
- Parquet [2] is a columnar storage format for Hadoop that supports *schema projection* and *column filtering*. It has the flexibility to read a particular column of data without touching the rest of the file stored in a distributed file system, which can be used to reduce redundant I/O operations by reading only the relevant portion of the data. Parquet also provides native support for data compression, which is helpful for large data sets.
- Spark [26] is a scalable fault-tolerant cluster computing framework that augments the capabilities of the MapReduce framework by introducing distributed data sets that can be used for caching intermediate results in iterative computations. It also supports a richer set of data operators and a powerful pipelining mechanism. Insparq uses Spark as the underlying computation framework, and is naturally capable of a wide variety of computations.
- Avro [1] is a data serialization system that supports the automatic generation of schema from structured text files and materialized class files in different languages. It is used in Insparq to integrate with Parquet by defining the underlying data representation.

### 3.1.1 Caching

Caching is the most important part of the system, referring in this context to the storage of intermediate state in the form of files. We adopt HDFS as the underlying file system, but other distributed file systems should work in theory as long as proper APIs are provided.

In the Insparq storage layer, we use simple hierarchies to organize data. Intermediate results of different computations are stored in different directories, under the root directory */cache/*. Each intermediate file is stored with the hash code of the corresponding dataset as its filename. Such a design eliminates the requirement for an additional lookup operation in a separate data structure. Instead, Insparq directly issues a query to the filesystem that checks for the existence of the file, which speeds up the lookup process. As a concrete example, if Insparq would like to get the result of computation A on dataset B, it directly queries the file system for a path of the form */cache/name(A)/hashcode(B)*.

Intermediate results are persisted to the file system whenever a calculation successfully finishes, and are loaded back to facilitate incremental computation whenever an intermediate value exists for the particular dataset.

### 3.1.2 Parquet Columnar Format

All files are saved in the Parquet file format due to the fact that some calculations only require a small fraction of the entire dataset. Parquet provides native support for schema projection and column filtering, which makes data access extremely flexible.

Parquet files are essentially HDFS files, with a particular binary format to define the data as tables with columns of various names and types. The *schema* of a parquet file can be determined by using a data serialization framework, in our case, *Avro*. Parquet supports schema projection and column filters when reading a file. One can specify a set of columns of interest to read a portion of the file, or define a filter to only read in data that satisfy a set of predicates.

### 3.1.3 Incremental Computation

We model incremental computation as follows. Assume that we have two datasets, A and B, where A is the base dataset, and B is the increment. An incremental computation is such that one only needs to compute over the increment, i.e., dataset B, without recomputing the base dataset

A, but using intermediate state generated in the processing of A, and produces the same results as if the computation was carried out from scratch over $A \cup B$.

When a computation is first initiated on dataset A, Insparq calculates a unique hash code for A, and queries the global file system for the existence of cached intermediate results. If the global filesystem returns *false*, indicating that such a calculation has not been performed, Insparq will process A and save intermediate results to the distributed file system using the naming scheme described earlier. When Insparq subsequently performs a computation over dataset $A \cup B$, it first queries the file system to see if the cached result exists for dataset A, then loads this intermediate result back, and queries the filesystem for the cached result of B, for which the filesystem returns false. At this point Insparq does a complete computation on dataset B and combines it with the cached result of A.

This mechanism works perfectly for our purpose except when there exist race conditions in writing and reading the files. For example, such conditions occur if two machines are working on the same dataset and try to query a cached file, fail to find it, then both perform a computation and try to save the newly created intermediate state. In this case an exception will be thrown in the absence of appropriate synchronization, indicating that the file already exists. Or, if one process is trying to access a cached file while it is being saved by another process, it may observe the contexts of the file as malformed (a parquet file is either healthy or malformed). Although such exceptions may be properly handled in application code, in our experience this greatly stalls the execution of the entire program. As a remedy, we implement a distributed file ReadWrite lock using ZooKeeper to coordinate concurrent accesses to cached files, ensuring that only one writer is permitted, and that a file is only read in the absence of writers.

## 3.2 Implementation

We implemented the three-line modeling process with both Spark and Insparq to enable a comparative evaluation of the benefits of incremental processing over conventional batch processing. In both cases, the input is stored as one or more comma-delimited text files, with each record indicating one smart meter reading for a particular household. The output is a collection of three-line models, one for each household provided in the input.

### 3.2.1 Basic Three-line Modeling on Spark

The basic or non-incremental three-line modeling algorithm is implemented in several steps, which are presented as pseudo-code in Figure 3. We group these steps conceptually into two phases: Phase 1 comprises steps 1-3, which are responsible for grouping the data points. This entails a series of map and reduce operations Spark. Phase 2 comprises step 4 and involves arithmetic followed by output.

In step 1, the input text file is read from HDFS, and each record is mapped to a KeyValue pair, with *houseId* as the key and the record as the value. A *reduceByKey* operation is then applied in step 2 to group the records by *houseId* so that energy data for each household can be modeled independently, which enables parallelism at scale. This operation involves a shuffle of the data set across the network, which is known to be a major performance overhead [8, 15].

Step 3 marks the beginning of the model computation for each household. The RDD obtained in step 2 is transformed using the *reduceByKey* operator, which maps each KeyValue pair in the input to another KeyValue pair with the same key (*houseId*) by applying a function to the value. Specifically, the value is transformed by grouping the energy-temperature readings according to temperature into corresponding *bins*, as well as by computing the required percentiles for energy consumption. This step does not require a costly shuffle because the transformation preserves the key of each KeyValue pair and hence the partitioning of data among Spark workers.

In step 4, the three-line model for each household is finally constructed by applying regression calculations to the energy-temperature data prepared in earlier steps. Although the linear regression and adjustments are implemented sequentially for each household, Spark parallelizes the execution automatically at the granularity of different households.

```
// 1. Map record to houseId-record pair
rdd1 <- record => [{houseId, record}]

// 2. reduce by houseId
rdd2 <- rdd1.reduceBykey()

// 3. map to bins
rdd3 <- rdd2 => [{houseId, [bins]}]

// 4. compute 3-line models for each household
rdd3 => [3-line models]
```

**Figure 3: High-level structure of three-line model implementation in Spark.**

### 3.2.2 Incremental Three-line Modeling

We recognize that the most time-consuming parts of the three-line modeling algorithm are the I/O-intensive operations of reading data from HDFS and shuffling RDD partitions across the network, as a result of the loading and binning process. Therefore, our incremental version keeps track of the history of each binning computation, and persists a set of intermediate representations to HDFS that can be loaded for Phase 2 computation to reduce the cost. We store this intermediate representation, which is a collection of temperature bins for each household, as parquet files in HDFS, which were discussed earlier in Section 2. We use an Avro schema to define the structure of a Parquet table, which records *houseId*, temperature of the bin, a list of readings in ascending order, along with the percentiles. The schema is presented in detail in Figure 4.

For each input file, we save the bins that have been computed to HDFS, along with a list for each bin of the corresponding energy readings in increasing order, and mark the input file as processed. If the program input contains previously processed files, we skip the loading and binning computations of those particular files, and instead load those bins from HDFS instead. All bins, whether computed from scratch or loaded from HDFS, are later combined using the *union* operator in Spark, to update the energy consumption percentiles. For each temperature bin, if multiple instances exist from different input files (e.g., past and new), we merge the sorted lists of energy readings efficiently in linear time and recompute the percentiles. The pseudo-code for the entire computation is presented in Figure 5.

```
{
  "type": "record",
  "name": "Bin",
  "fields":[
     {"name": "houseId", "type": "string"},
     {"name": "temp", "type": "double"},
     {"name": "tenthPercentile", "type": "
         double"},
     {"name": "median", "type": "double"},
     {"name": "nintiethPercentile", "type": "
         double"},
     {"name": "readings", "type": {"type": "
         array", "items": "double"}}
  ]
}
```

**Figure 4: Avro schema for a temperature bin.**

To further reduce the I/O overhead associated with reading cached bins from HDFS, we store them using the Parquet [2] columnar format. This not only avoids storing duplicated fields, as in the original plain text format, but it also enables compression using the *Snappy* codec [3]. Snappy a portable library developed at Google for fast lossless compression and decompression.

```
// Phase 1. for each file to process
for each file:
  // 1.1. if file already processed
  if file_processed:
    bins <- load from cache

  // 1.2. if not processed
  else:
    record <- load from file
    // 1.2.1. Map record to houseId-record pair
    rdd1 <- record => [{houseId, record}]
    // 1.2.2. reduce by houseId
    rdd2 <- rdd1.reduceBykey()
    // 1.2.3. map to bins
    bins <- rdd2 => [{houseId, [bins]}]

  // 2.3. union bins
  all_bins <- all_bins.union(bins)

// Phase 2. compute 3-line models for each
    household
all_bins => [3-line models]
```

**Figure 5: Pseudo-code for incremental computation of the three-line model.**

# 4. EVALUATION

## 4.1 Dataset Synthesis

We used synthetic data to evaluate our system. The dataset consists of multiple comma separated text files, with each line representing a tuple in the form of (*Date, Hour, Reading, Temp, HouseId*). There are 32,000 households in total, each providing a year's worth of hourly energy readings. In plain text format, the dataset occupies 8 GB. We split the dataset conceptually into four quarters, referred to as Q1-Q4 in later sections, and perform incremental computation at intervals of one or more quarters. As we explain later on, the trends in our experimental results suggest that shorter increments tend to yield greater performance benefits compared to non-incremental batch processing.

## 4.2 Hardware and Software Environment

We evaluate Insparq with three-line modeling on a cluster of seven commodity machines. Each machine is equipped with dual 2.40GHz Intel Xeon E5-2620 CPUs, supporting a total of 24 hyperthreads per server, as well as 64 GB of main memory and a 1Gbps NIC. All servers run Ubuntu Server 14.04.3 LTS 64-bit with kernel version 3.13.0-65-generic. In our experiments we use Spark 1.5.1 and Hadoop 2.6. We use a dedicated server for the Spark master and HDFS name node, while having one Spark slave and HDFS data node on each of the other six servers.

We set up each Spark slave to use 8 cores and 32 GB of memory on each of the nodes. Spark has a default behavior that only uses a small fraction of memory allocations for shuffle storage, and spills exceeding shuffling data to hard disk, greatly stalling the execution with growing size of dataset. We manually configure Spark to disable shuffle spilling and use as much memory for shuffling as possible. HDFS is set up to have a replication factor of 3.

## 4.3 Experiments & Results

We evaluated our incremental three-line modeling on Insparq against a non-incremental version on Spark in terms of runtime performance and multi-core scalability. Due to the excessive amount of memory on the servers, aggressive file system caching prevents our experimental results from reflecting the actual performance. To avoid the interference of OS-level caching mechanisms, we restart Spark and HDFS processes and drop all pagecache, dentries and inodes (using "`echo 3 > /proc/sys/vm/drop_caches`") before each set of experiments.

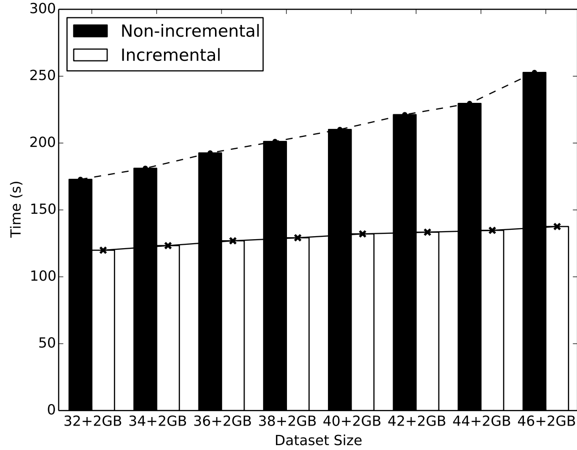
### *4.3.1 Incremental Execution*

We evaluate the performance of three-line modeling with our synthetic dataset to assess the practical benefits of incremental processing. Although it is clear that incremental computation can avoid redundant I/O and arithmetic computations, the benefit comes at the cost of maintaining intermediate state in the distributed file system. Thus, a careful analysis is necessary to determine the range of parameters for which performance gains are feasible.

In our experiment, we first execute the program with the base dataset of 32 GB, comprising four years of data for 32,000 households sampled every hour. On top of that, we increase the dataset by one quarter, which corresponds to 2 GB of data, and compute the three-line models over the union. The result of this incremental computation is shown in Figure 6. The x-axis indicates the size of datasets as A+B GB, with A being the size of a base dataset, and B being the size of the increments. For an incremental run the framework uses intermediate results from previous executions, while for the non-incremental case a batch computation is conducted from scratch over A+B. The y-axis shows the running time in seconds. Each run is repeated five times starting with a clean cache to ensure that results are reproducible.

The running time for the non-incremental version on Spark is up to roughly 250 seconds on data sets up to 48 GB, while the incremental version always finishes in less than 150 sec-

**Figure 6: Running time comparison of three-line modeling with and without incremental execution, using 2 GB increments.**

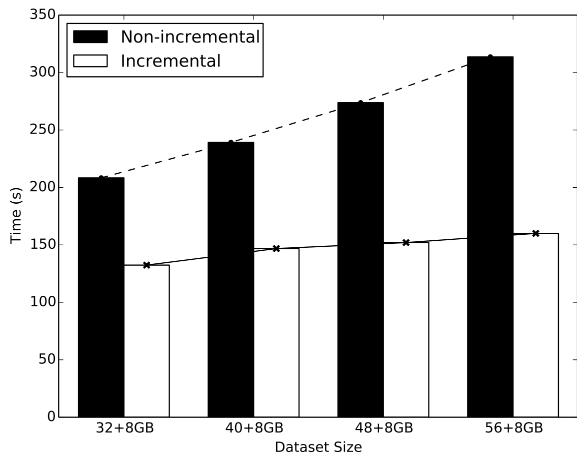



onds to process a 2 GB increment. The running time of the non-incremental implementation grows with the size of the data set, whereas the incremental algorithm performs steadily with a slight upward trend.

We further show the benefit of incremental three-line modeling with one year's worth of data as the increment, corresponding to 8 GB of data, as shown in Figure 7. While the average run time of non-incremental runs increases by roughly 30 seconds per 8 GB increment, the incremental version finishes in 160 seconds with a dataset of 64 GB in total, 2x faster than the non-incremental version.

Thus, the results demonstrate a clear advantage of incremental execution when the increment is small relative to the base data set.

**Figure 7: Running time comparison of three-line modeling with and without incremental execution, using 8 GB increments.**




### 4.3.2 Multi-core scalability

To show the multi-core scalability of incremental and non-incremental three-line modeling on Spark, we evaluate running time speedups with a varying number of cores used by each node, ranging from 1 to 5. We use the *–executor-cores* option in Spark to control the number of cores actually used by the workers. Speedups are calculated assuming the actual runtime is $T_1$ with 1 core per node, and $T_n$ with $n$ cores per node, using the following formula:

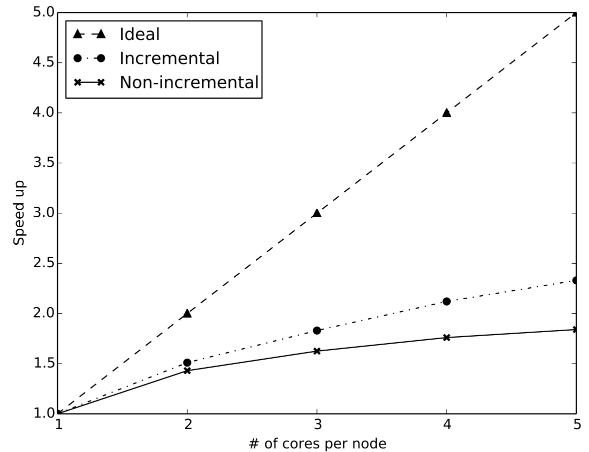
$$\text{Speedup with } n \text{ cores} = \frac{T_1}{T_n} \quad (1)$$

For this experiment we use two 1-year datasets sized to 8 GB each as input. The non-incremental version computes over the entire 16 GB dataset, while the incremental version takes one as base, and the other as increment.

Figure 8 shows the scalability of three-line modeling on Spark with and without incremental execution. The x-axis indicates the number of cores used by each Spark worker, and the y-axis indicates its corresponding speedup compared to using only 1 core per worker.

Results show that Spark scales poorly. For non-incremental computations it achieves a speedup of only 1.8x with five cores per node compared to one core, while incremental modeling achieves 2.3x speedup. We attribute this to the expensive I/O overhead caused by a large amount of file system read/write operations and network shuffles. Incremental modeling achieves slightly better scalability since we reduced the amount of HDFS read/write operations and the size of RDDs to shuffle across the network.

**Figure 8: Speedup comparisons for incremental and non-incremental modeling with varying number of cores.**




## 5. RELATED WORK

Large-scale data processing techniques related to smart grid analytics roughly fall into four categories, which we discuss briefly below.

**Classical data-parallel abstractions.** MapReduce [9] and Dryad [13] are massively scalable fault-tolerant frameworks for parallelizing and distributing task execution across commodity servers. MapReduce divides each task into a

map phase, which transforms a set of key/value pairs to a set of intermediate outputs, and a reduce phase that merges or summarizes the intermediate outputs. Dryad supports more general fine-grained data flows represented using a directed acyclic graph (DAG), and is extended with support for language-integrated queries [24].

**Abstractions for iterative computations.** Despite their wide adoption in industry, implementations of classic abstractions are fundamentally inefficient in terms of I/O for iterative computations, such as in ranking the importance of web pages, or in machine learning. Spark [26] addresses this problem by introducing a new data-parallel abstraction, resilient distributed datasets (RDDs) [25], which enables efficient pipelining and caching intermediate results between iterations. Spark is used for non-iterative map-reduce computations over energy data in the benchmarking study of Liu et al. [15], which we discuss later on in this section, as well as in an earlier study by Cheah [8]. Naiad [19] introduces a new computation abstraction, timely data flow, that represents data flows using directed graphs similarly to Dryad but with cycles representing data reuse in iterative computations.

**Abstractions for incremental computations.** In an effort to accommodate dynamic data sets, data-parallel systems are starting to provide abstractions for incremental computations. DryadInc [21] extends Dryad [13] to automatically identify redundant computations within the same job by caching previously executed tasks. Percolator [20] is a system that provides distributed transactions and notifications for processing small, independent updates to a large data set such as Google's web indexing system. Continuous bulk processing (CBP) [18] is an architecture for stateful computation over disk-resident data using a novel groupwise operator. Incoop [6] extends the Hadoop framework—an open source implementation of MapReduce—with an incremental distributed file system, an additional contraction phase, and a memoization-aware scheduler. Hourglass [10] provides an accumulator-based interface on top of Hadoop for programmers to store and reuse state across successive runs. Itchy [22] is a framework that executes incremental jobs by processing only relevant parts of the original input, together with the incremented input. It uses novel techniques to store provenance information, intermediate data, and output data of a MapReduce job.

**Analytic databases and data warehouses.** Analytic computations can be implemented inside a database or data warehousing system using a combination of SQL and user-defined functions. HP Vertica [14] is a scalable analytics platform that was evaluated on 727 TB of energy data in the benchmarking study of Arlitt et al. [5] for fundamental tasks such as computing peak power consumption, consumption time series, top consumers, and time of usage billing. Hive [23] is a scalable data warehousing solution on top of Hadoop, and is compared on data sets of up to 1TB against Spark in the benchmarking study of Liu et al. [15] on four workloads: disaggregating energy data using the 3-line model [7], extracting daily energy consumption trends independently of outdoor temperature using periodic autoregression (PAR), building histograms to profile hourly energy consumption, and searching for similarity between customers in terms of their energy usage profiles. Both Spark and Hive are shown to scale well and perform comparably for the first three workloads, in which each customer is processed independently of others. Spark outperforms Hive for similarity search, which requires comparisons between pairs of customers. In the latter case the performance difference is attributed to Hive generating a sub-optimal query plan compared to the one coded manually in the Spark implementation, which leverages efficient map-side joins. SMAS [16] is a system for analyzing and visualizing smart meter data, and is implemented on top of PostgreSQL using the MADlib [11] analytics library, which provides multi-core parallelism but does not scale automatically across servers. The system implements three of the analytic computations discussed in [15]: 3-line model, periodic autoregression, and histograms. Real-time stream ingestion for SMAS is discussed in a follow-up paper [17].

# 6. CONCLUSION

In this paper, we present Insparq, a framework for efficient incremental computation over large data sets. Our experimental evaluation highlights the performance benefits of Insparq compared to Spark, on which our framework is built, in the context of analyzing energy data from smart electricity meters. Specifically, we demonstrate that Insparq improves runtime performance substantially through incremental execution, and outperforms Spark even in the absence of intermediate data prior to an execution, due to the fact that Insparq makes use of intermediate values for each computation step within an execution. In future work we plan to implement and evaluate additional analytic tasks using our framework, such as the computation of energy consumption histograms and periodic auto-regression, as well as to define a benchmark for incremental energy analytics.

# Acknowledgment

This research is supported in part by the Waterloo Institute for Sustainable Energy (WISE) – Cisco Systems Smart Grid Research Fund, as well as the Natural Sciences and Engineering Research Council (NSERC) of Canada.

# 7. REFERENCES

[1] Avro data serialization system. `http://avro.apache.org/`.

[2] Parquet: Columnar storage for hadoop. `http://parquet.io/`.

[3] Snappy: A fast compressor/decompressor. `http://google.github.io/snappy/`.

[4] Storm, distributed and fault-tolerant realtime computation. `https://storm.incubator.apache.org`.

[5] M. Arlitt, M. Marwah, G. Bellala, A. Shah, J. Healey, and B. Vandiver. Iotabench: An internet of things analytics benchmark. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 133–144, 2015.

[6] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin. Incoop: Mapreduce for incremental computations. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SoCC)*, pages 7:1–7:14, 2011.

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

[8] M. Cheah. Spark and smart meter analytics: An experience report. Technical report, University of Waterloo, 2014.

[9] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, Jan. 2008.

[10] M. Hayes and S. Shah. Hourglass: A library for incremental processing on hadoop. In *Proceedings of the IEEE International Conference on Big Data*, pages 742–752, 2013.

[11] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar. The madlib analytics library: Or mad skills, the sql. *Proceedings of the VLDB Endowment*, 5(12):1700–1711, Aug. 2012.

[12] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference*, 2010.

[13] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*, pages 59–72, 2007.

[14] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear. The vertica analytic database: C-store 7 years later. *Proceedings of the VLDB Endowment*, 5(12):1790–1801, Aug. 2012.

[15] X. Liu, L. Golab, W. Golab, and I. F. Ilyas. Benchmarking smart meter data analytics. In *Proceedings of the 18th International Conference on Extending Database Technology (EDBT)*, pages 385–396, 2015.

[16] X. Liu, L. Golab, and I. F. Ilyas. SMAS: A smart meter data analytics system. In *Proceedings of the 31st IEEE International Conference on Data Engineering (ICDE)*, pages 1476–1479, 2015.

[17] X. Liu and P. S. Nielsen. Streamlining smart meter data analytics. In *Proceedings of the 10th Conference on Sustainable Development of Energy, Water and Environment Systems (SDEWES)*, pages 1–14, 2015.

[18] D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum. Stateful bulk processing for incremental analytics. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, pages 51–62, 2010.

[19] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 439–455, 2013.

[20] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 1–15, 2010.

[21] L. Popa, M. Budiu, Y. Yu, and M. Isard. Dryadinc: Reusing work in large-scale computations. In *Proceedings of the 1st USENIX workshop on Hot Topics in Cloud Computing (HotCloud)*, 2009.

[22] J. Schad, J.-A. Quianee-Ruiz, and J. Dittrich. Elephant, do not forget everything! efficient processing of growing datasets. In *Proceedings of the 6th IEEE International Conference on Cloud Computing (CLOUD)*, pages 252–259, 2013.

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

[24] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 1–14, 2008.

[25] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2012.

[26] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics In Cloud Computing (HotCloud)*, 2010.

[27] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. In *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*, 2012.