HAL
open science

A Scalability Comparison Study of Data Management Approaches for Smart Metering Systems

Houssem Chihoub, Christine Collet

To cite this version:

Houssem Chihoub, Christine Collet. A Scalability Comparison Study of Data Management Approaches for Smart Metering Systems. The International Conference on Parallel Processing (ICPP 2016), Aug 2016, Philadelphia, United States. pp.474 - 483, 10.1109/ICPP.2016.61 . hal-01585352

HAL Id: hal-01585352

https://hal.science/hal-01585352v1

Submitted on 11 Sep 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A scalability comparison study of data management approaches for smart metering systems

Houssem Chihoub and Christine Collet
Grenoble Institute of Technology (Grenoble INP)
Grenoble, France
houssem.chihoub@imag.fr, christine.collet@grenoble-inp.fr

***Abstract*—Nowadays, more and more data are being generated and collected in electrical smart grids. Most of these data are coming from smart meters and sensors deployed massively throughout the power grid. As the generation of data is becoming ever more frequent and with the constantly increasing volumes, it is becoming harder and harder to manage and process these data at the scale of a smart grid within legacy systems. In this work, we focus on investigating the scalability and performance of different data management approaches for meter data processing. To this end, we conduct a thorough experimental study of various systems including a parallel relational database system, MapReduce based systems including Hadoop and Spark, and a NoSQL datastore system. Our experiment sets were conducted on up to 140 nodes on Grid5000 and up to 1.4 TB of meter data. Our results demonstrate that parallel relational systems are more suited for most processing types on smart meter data in the smart grid. In contrast, we show that with the appropriate distribution model, data partitioning and modeling choices we achieve very fast and scalable bill computations, the main complex processing for utilities providers.**

I. Introduction

Smart grids are electrical grids that combine conventional and renewable energy resources, and rely on digital advances in smart metering, sensoring technologies and two-way communication means in order to provide efficient management of energy. Recent years have seen many energy utilities invest heavily to transform their power grid infrastructures. For instance, in France ERDF is planning to deploy 35 million smart meters by the year 2021 [1]. With massive deployments of millions of smart meters and sensors throughout the power grid, more and more data are collected providing fine grain insights about client consumption profiles and the behavior of the power grid. Such data are critical towards more efficient management of energy resources and provides new level of efficiency for analytic-based applications such as predictive maintenance and demand forecasting. However, the frequency in which data are generated and collected from millions of sensors and smart meters at the scale of a country makes the task of data management and processing very complex. This complexity is due to the huge volumes of data collected over long periods of time to be stored, as well as the performance requirements on accessing and processing these data.

Legacy data management systems in smart grids are typically based on relational database management systems. As a result, many of these systems have started to exhibit major difficulties to cope with the data deluge of smart metering and sensoring and meet the scalability requirements of these systems. Fortunately, dealing with huge amounts of data is a common issue in the era of Big Data. In recent years, many data management and processing systems have emerged to cope with various challenges (volumes, velocity, variety, load variability, performance, etc.). These systems vary in their models and architectures, with many being specific to use cases such as graph processing and documents management. In this work, we investigate four approaches with their emerging models and systems for scalable data management and processing. We further evaluate their capacities to deal with massive meters data and compare their parallelism and distribution architectures, their data partitioning schemes, and their data models and query languages support. Within the first approach, we investigate MapReduce based processing frameworks [2]. The second approach is based on next-generation of MapReduce computations with acyclic graph execution engines and in-memory processing. The third approach relies on NoSQL datastores with peer-to-peer architectures, which are known for their scale-out properties. The last approach consists of parallel relational database management systems (RDBMS) with massively parallel processing (MPP) models. Lately, parallel RDBMS models have been revisited to provide solutions to many Big Data problems at scale. As illustrative implementations, we rely on the following systems respectively: Hadoop [3], Spark [4], Cassandra [5], and Postgres-XL [6]. To evaluate these approaches for our smart meter data management case, we introduce a set of queries that exhibit the different processing types on smart meter data. These are queries that are mainly of three types: queries based on aggregation functions (such as the sum of measurements), selection and filtering queries based on some criteria (e.g. a measurement threshold), and bill computation queries that are based on some complex rules (e.g. tarif vert of EDF). Bill computation is considered to be the most complex type of processing for utilities providers and is a bottleneck for many meter data management systems [7].

We conduct a thorough experimental evaluation on GRID5000 [8], the French cloud and grid testbed. First, following the methodology in [9], we generate data for more than 4 million meters over a period of one year with a measurement every hour and a total of more than 35 billion measurements. Thereafter, we conduct several experimental sets on these data on up to 140 nodes to evaluate and compare data management

approaches for smart meter data processing queries. Our extensive evaluation demonstrates that Postgres-XL relying on massively parallel processing (MPP) outperforms all other systems for aggregation queries over meter data. Furthermore, we show that due to data partitioning and layout scheme in Cassandra, Spark on top of Cassandra provides an extremely fast response time for bill computation queries with massive data sizes and scales out with the increasing number of nodes. Furthermore, our study reveals that overall, MapReduce based systems (Hadoop and Spark) tend to generate more data transfer throughout the processing cluster, which in turn delays response times for many queries. In addition, Spark processing performance is bound to the size of live memory.

The remaining of this paper is organized as follows. Section II gives an overview of meter data processing. In Section III, we introduce the data management approaches. We further provide a comparison of these approaches in Section IV. Section V shows our experimental methodology while Section VI zooms into our experimental evaluation. Related work is presented in Section VII and our conclusions in Section VIII.

II. Meter data processing in the smart grid

A. Smart meter systems

Many utility providers (electricity, water, etc.) nowadays rely on smart metering and deploy Advanced Metering Infrastructures (AMIs) to guarantee an efficient management of their resources. The Advanced Metering Infrastructure (AMI) is the system that integrates smart meters, network protocols, and the meter data management system (MDM). In smart grids, data are generally sent from meters to head-ends (such as data concentrators) using low-throughput communication means such as the G3-CPL protocol. Meter data is further sent to the MDM where it should be stored. The MDM system insures data cleaning, storage, access, and processing. Many of the MDM systems rely on legacy relational database systems adapted to manage meter data. Recent years however have seen meter data volumes grow very fast due to massive deployment of smart meters and the increase of the number of meter measurements by households. As a result, many of the current MDM systems will have major difficulties to meet the data deluge of future smart grids. According to Navigant Research, smart grid IT software and services expenditures are expected to reach 20 billion $ by 2022 [7], [10].

B. Data processing types

Meter data are used for multiple purposes in the smart grid: Bill computations, consumption analysis, power generation forecasting, fraud detection etc. Most of these services and applications exhibit three types of meter data processing queries as shown in [7]. First, the aggregation queries that consist of applying a computation function on the dataset as a whole or a subset of data. The computation functions can be of various natures such as the sum of consumption in a given region or for a given set of clients, the count of total measurements, or the max measurement to infer the highest consuming client in a given time of the year. The second type of processing queries consists of selection and filtering based on some predicates. A typical application for this type of queries is the selection of all meter ids (clients) that exceeds a given threshold consumption at a given bill period to determine the highest consuming homes during this period. The last type of processing queries are the bill computation queries. These are complex queries that consist of sub queries that can be of different nature in order to compute the bill based on very specific tarification rules.

III. Large scale data management and processing

In this section, we introduce four approaches to large-sale data management and processing that are widely adopted nowadays. Their illustrative systems are depicted in Figure 1.

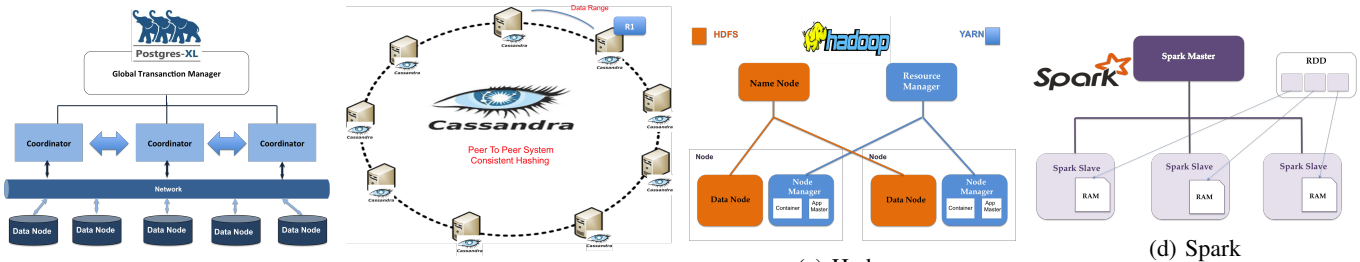
A. MapReduce based systems

The MapReduce programming model for data-intensive large-scale applications was first introduced by Google [2]. Inspired by divide and conquer principle, it consists of defining a Map phase to decompose big problem into a set of small sub problems and a Reduce phase to address the small sub-problems. The success Goolge had with their implementation of the MapReduce paradigm in solving their large-scale data processing problems has resulted in a big enthusiasm around the model, in particular with the exponential grow of data volumes. Furthermore, open sourcing Hadoop [3] implementation of MapReduce has resulted in a wide adoption of this paradigm. Although, Hadoop remains the most popular implementation of MapReduce, many other implementations have been introduced such as Disco [11], and MARIANE [12].

Hadoop: Hadoop [3], shown in Figure 1c, is the most popular implementation of MapReduce. The Hadoop ecosystem consists of multiple projects including the MapReduce framework and the Yarn resource manager, the Hadoop distributed file system (HDFS) designed after Google File System [13], and the HBase data store [14] designed after Google Big Table [15]. An important component of the Hadoop ecosystem is Hive SQL query engine [16]. Hive provides SQL on top of MapReduce where its query engine generates MapReduce jobs for SQL statements.

B. Next-generation MapReduce in-memory processing

In recent years, many efforts have been dedicated to enhance the performance of MapReduce systems. One of the main issues with Hadoop was its inflexible programming API (in particular with iterative computations) and its disk bound high latency. As a result, a new generation of MapReduce processing frameworks has emerged. They mainly provide richer APIs that provide various types of Map and Reduce functions and acyclic graph execution engines for multiple Map and Reduce steps and thus multiple intermediate phases (compared to only one with early MapReduce implementations). Furthermore, many rely on in-memory processing to provide fast computations compared to disk-bound computations. The most popular solution in use today is Spark [4]. Other solutions such as Tez [17], or Flink [18] are also being widely used.



(a) Postgres-XL (b) Cassandra (c) Hadoop (d) Spark

Fig. 1: Data management and processing systems investigated in our study: Postgres-XL a distributed RDBMS; Hadoop ecosystem with MapReduce framework and HDFS; Spark with in-memory processing; Cassandra NoSQL data store with its peer-to-peer consistent hashing based architecture

Spark: Spark framework [4], depicted in Figure 1d, is implemented in the Scala programming language and provides a rich programming API to users. Within this API, operations are divided into a set of *transformations*, which are specific Map functions and a set of *actions* that are specific Reduce functions. Spark API in addition, is very powerful to express iterative operations. In order to facilitate data manipulation, Spark introduces the abstraction of *resilient distributed datasets (RDDs)*, a read-only collections of objects partitioned across a set of machines that can be recovered in case of failures. RDDs can be cached in memory prior to processing to avoid disk high latency. Within Spark framework, the Spark SQL [19] module provides an SQL query engine that generates Spark jobs for SQL statements. Spark SQL relies on the introduced DataFrame API to perform relational operations on both external data sources and spark built-in collections, which facilitates the integration with native Spark.

C. NoSQL and Peer-to-peer based systems

Historically, data storage and management was provided mainly by filesystems or relational database systems. However, the last two decades have seen a major shift with the appearance of new systems due to many reasons. For instance, the emergence of web applications such as e-commerce and social media sharing has introduced new challenges and requirements to data management and storage. Nowadays, applications vary in their requirements for data availability, data consistency, data formats and models, scalability, etc. As a result various NoSQL (Not only SQL) datastore systems have emerged. Many of them introduce data models specific to some use-cases such as document stores or graph database systems. Other NoSQL systems target high availability and scalability at scale such for Amazon Dynamo [20], Google Big Table [15], Google Spanner [21], and Cassandra [5]. To achieve their goals, these systems either rely on Master/Slave architectures or peer-to-peer architectures. These latter have been widely used to build datastores mainly due to their horizontal scalability capacities where data nodes are equals, host equal shares of data, and fairly participate in the cluster management with no single point of failure.

Apache Cassandra: Cassandra [5] was first designed and implemented at Facebook. It was designed for managing large objects of structured data spread over a large amount of commodity hardware located in different datacenters worldwide. Cassandra design was highly inspired by that of two other distributed storage systems: Amazon Dynamo [20] and Google BigTable [15]. It inherited the peer-to-peer architecture with the consistent hashing [22] based data partitioning (as shown in Figure 1b) from the first and the column family based data model from the second. In Cassandra, data is stored in structures called column families where columns can be created dynamically. Data access is usually key-based. However, with the latest versions of Apache Cassandra [23], a Cassandra Query Language (CQL) has been introduced. CQL allows users to write declarative queries in an SQL-like manner. Nevertheless, CQL still lacks many features such as join operations and aggregation functions.

D. Parallel relational database management systems

In the 1980s and early 1990s, relational database management systems were already facing scalability problems. Many applications, including scientific and engineering ones, have started to deal with large volumes of data that overwhelmed the single-machine database systems at the time. In this context, parallel database systems have emerged. Many architectural models have been proposed but the most viable architecture that provided the needed scalability was the shared nothing architecture where processors have their own private memory and disk resources. With the explosion of Big Data in recent years, NoSQL and MapReduce based systems have emerged to the rescue. However, lately parallel RDBMS (relational database management systems) models have been revisited to provide an efficient data processing at scale. The massive parallel processing (MPP) of these systems that allows processors to be as independent as possible and to have minimal communications together with the power of optimized query engines can provide solutions to many complex data processing at scale in many cases. Many parallel RDBMS have been deployed within many organizations for many years such as IBM DB2 Parallel Edition [24] and Teradata [25]. More recently, emerging systems with novel designs, such as open source solution Postgres-XL [6], have been introduced.

1) Postgres-XL: Postgres-XL [6] is an open source parallel relational database management system. Its architecture is shared nothing master/slave based, as depicted in Figure 1a. Postgres-XL provides massively parallel processing (MPP)

and a strong SQL support. Additionally, in a Postgres-XL database, ACID semantics (atomicity, Consistency, Isolation, and Durability) are guaranteed. To provide such features, Postgres-XL relies on a cluster wide Multi-Versioning Concurrency Control (MVCC) that allows it to be faster in comparison to traditional systems with 2-phase locking (2PL). Furthermore, Postgres-XL is designed to support both OLAP (online analytical processing) and OLTP (online transaction processing) workloads.

IV. Comparison of large-scale data management and processing systems

In this section, we zoom on parallelism and distribution models, data partitioning schemes, and data models of each data management system. Table I summarizes the characteristics and models for each system. In this study, we do not consider optimizations that can be specific to each solution such as indexing, columnar file formats, compression, etc.

A. Parallelism and Distribution architectures

The data management and processing systems in our study (Postgres-XL, Hadoop, Spark, and Cassandra) have different and various schemes of distributed architectures and parallel processing. Postgres-XL relies on a Master/Slave architecture where a GTM master manages the global coordination to provide ACID semantics of transactions relying on MVCC. Every slave has, typically, two main components : the coordinator that handles clients request and manages communication with other slaves and its datanodes. The datanodes are responsible of hosting their fraction of data. Both components are implemented using a postgreSQL database. Similarly HDFS consists of a Master/Slave architecture. A master called NameNode, hosts the metadata server that typically keeps track of global information on the filesystem and the location of file chunks. Slaves called DataNodes host their share of file chunks. An HDFS client typically request information from the NameNode about which datanodes are involved in its operations before sending requests to datanodes. In contrast, Cassandra relies on a peer-to-peer architecture. All nodes are equals, where they handle equal ranges of the data space and share similar tasks in the cluster management. Every node can handle client requests and locate data as well as respond to data access operations for its data. Whereas, Hadoop and Spark rely on MapReduce parallel processing, Postgres-XL implements an MPP (massively parallel processing) where processors are loosely coupled and should be as independent as possible with only minimal communications. Furthermore, it relies on a SQL query engine that computes the best execution plan prior to processing favoring computation transfer and minimizing data movement. MapReduce computations however, rely on completely independent Map operations that can be executed in parallel without any communication. However, to perform Reduce operations processors need their input data (intermediate data from the Map phase) to be moved to them, which is done by the shuffle-sort phase. Spark further provides various implementations to be specified for the shuffle phase including shuffle-sort and shuffle-hash. With its programming API and acyclic graph execution engine, Spark applications tend to have more intermediate phases and thus more shuffle operations. In many cases and for many types of applications, the shuffle phase results in the transfer of huge amounts of data throughout the cluster resulting in a performance bottleneck.

B. Data Partitioning

The data partitioning schemes within the aforementioned systems are quite different. Postgres-XL relies on sharding where tables are partitioned horizontally. Every shard is assigned to a datanode based on either range hashing or in a round robin manner. As a result, operations that involve only a subset of shards can be handled by only the datanodes hosting them, which in turn reduce data transfer compared to cases where data is located on larger number of nodes. In contrast, Cassandra relies on consistent hashing [22] where every data row is assigned to a node based on the hash value of its key. Every node in Cassandra is responsible of an equal range of the data space. A direct result of this scheme is the fast response time of accessing data based on keys (eg. read a specific row). Furthermore, at scale, this approach has potential for high load balancing compared to sharding when row keys are carefully defined. Additionally, recent versions of Cassandra integrated the concept of virtual nodes in peer-to-peer systems where data ranges are assigned to virtual nodes instead of physical nodes and a physical node can host one or more virtual nodes. This approach allows even better load balancing in many cases. However, range queries can be much slower since data can be potentially spread over all the nodes in the cluster even for small ranges. In the Hadoop ecosystem, the Hadoop distributed file system (HDFS) is the component responsible for storing data. Within HDFS, data is stored, typically in big files. A file is split into a set of chunks (with a default size of 64MB) where every chunk is assigned by the metadata server to a datanode. As a result, data partitioning depends greatly on the file format and layout used to store data.

C. Data Model and Query Language

Similar to their different data partitioning schemes, these data management and processing systems vary in their data models as well. Postgres-XL is a relational system with a relational data model. All data have to be well structured and stored in well-defined relations and supports relational algebra. Postgres-XL provides a full ANSI SQL 2008 standard support. It supports however only declarative queries. In contrast, Cassandra relies on the column families model introduced by Google for their storage system BigTable [15]. In this model, data are stored in dynamic tables called column families. In column families, columns can be defined on the fly specifically for every row. Furthermore, an additional abstraction in a column family is introduced: a super column. A super column can group columns together but not other super columns. This data model was introduced to provide flexibility to both data layout in physical storage (memory and disk) and to allow better modeling for unstructured and semi structured data

System	Distribution & Parallelism	Partitioning	Data model	SQL Support
Postgres-XL	Master/slave, MPP (massive parallel processing)	Range hashing, round robin	Relational	ANSI SQL:2008
Hadoop	MapReduce, master/slave	file chunks based	files	Hive SQL query engine
Spark and HDFS	MapReduce based, master/slave, acyclic graph execution engine	file chunks based	files and RDDs	SparkSQL query engine
Cassandra	peer-to-peer	consistent hashing (key-based)	column families	CQL (restricted number of operations)
Spark and Cassandra	peer-to-peer, MapReduce, acyclic graph execution engine	consistent hashing (key-based)	column families and RDDs	SparkSQL query engine

TABLE I: Data management and processing systems

while making it possible to store structured data. Cassandra however does not support transactions nor a full SQL-like and linear algebra operations natively. With earlier versions, data were accessed based on key specification. Lately, Cassandra Query Language (CQL), a declarative query language have been introduced. However, this language does not support aggregation queries and time functions. Instead, users have to implement their own solutions within their applications or rely on a third party data processing that integrates with Cassandra such as Spark. In MapReduce based systems (Hadoop and Spark), data are usually stored in a filesystem (HDFS) with no specification about the data model. Data are however put in key/value model for Hadoop MapReduce and RDDs Spark native data collections. Natively, Both Hadoop and Spark rely on their procedural programming to process data. Nevertheless, declarative queries are supported by SQL query engines Hive and SparkSQL that generate native MapReduce code for Hadoop MapReduce and Spark respectively.

V. Benchmarking for meter data management

A. Methodology

In this section, we aim to compare and analyze the performance of the aforementioned systems for meter data processing that can be encountered in a smart grid. In order to achieve this goal, we first generate data for more than 4 million meters with a measurement every hour in a period of one year. Then, we conduct a set of three experiments. The first set consists of deploying the aforementioned data management and processing systems on 110 nodes and increase the dataset size from 0.55 million meters to 4 million meters to measure the response time for every query. In the second set of experiments, we evaluate the horizontal scalability for each system with the different queries increasing the number of nodes from 70 to 140. In the last experiment set, we vary the number of nodes from 5 to 30 nodes hosting only 10 thousands meters data in order to guarantee that initial data can be fit in memory. This allows us to evaluate the performance in a fair manner for in-memory processing systems.

Experimental Setup: We run our experiments on Grid5000 cloud and grid testbed [8] in France. For this purpose we use nodes in the Griffon and Graphene clusters located in the Nancy site in north east of France. All nodes in these two clusters are equipped with 298GB HDDs, 16GB of Ram and two 2.5GHz CPUs with 4 cores/CPU. We use from 5 nodes up to 140 nodes to run our experiments. Furthermore, we rely on Storage5K [1] that provides the necessary storage space to store 1.4TB of meter data prior to the experiment and in order to load them to data management systems. Hadoop-2.5 [3] is deployed within Cloudera CDH-5.2.3 distribution [26] including Hive-0.13 [16]. As for Spark we deploy Spark-1.5 [27] with its built-in Spark SQL. Additionally, we deploy Apache Cassandra-2.2 [23] as well as Postgres-XL-9.2 [6]. In order to implement processing queries, we rely on Spark on top of Cassandra using the Spark-Cassandra-1.5 connector.

B. Data processing queries

We introduce 7 queries that illustrate processing types on meter data presented in Section II. The queries are summarized in Table II. Queries Query1, Query2, and Query3 are three aggregation queries that compute the sum of measurements. Query2 computes the sum for a subset of clients whereas Query3 computes the sum of measurements in just 1-month period of time. Query4, Query5, and Query6 are selection and filtering queries. Query4 in addition sorts the results by their measurements value whereas Query6 filters the results on a 2-months period of time for a given list of meter ids. The last query (Query7) is a bill computation query that is based on the *tarif vert* of EDF the electricity provider utility in France [2]. Within this tarification scheme, the electricity price varies according to power provided and the time in the day of consumption.

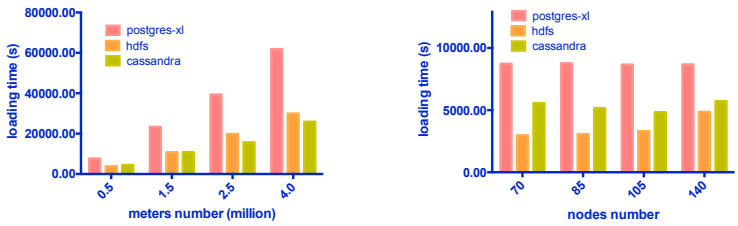
C. Data generation and loading

For data privacy reasons, it has not been possible for us to get real meter data from energy utilities. Fortunately, it

[1] https://www.grid5000.fr/mediawiki/index.php/Storage
[2] http://www.fournisseurs-electricite.com/tarif-vert

Query	Type	Description
Query1	Aggregation	Sum of all measurements (consumption of all meters) for 1-year period (2013)
Query2	Aggregation	Sum of all measurements for a given range of meter ids (clients)
Query3	Aggregation	Sum of measurements in a 1-month period (march 2013)
Query4	Selection & filtering	Selection of the first 20k meter ids and their measurements over a 2-month time interval where the consumption exceeds a given threshold, then sort the result by their consumption values (order by clause)
Query5	Selection & filtering	Selection of meter ids and their measurements where the consumption exceeds a given threshold
Query6	Selection & filtering	Selection of measurements given the list of meter ids over a 2-months period of time
Query7	Bill	Compute the bill for a given client following the tarif vert billing rules of ERDF

TABLE II: Processing Queries on meter data



(a) increasing number of meters (b) increasing number of nodes

Fig. 2: Data loading

was possible to generate realistic datasets that illustrate meter data as described in [9]. In [9], the authors rely on a small client dataset in order to create data for new clients. First, a temperature-independent profiles from existing clients are extracted. These profiles are further used in combination with randomly selected weather data, while adding some noise, in order to create consumption data for new clients. In this work, we follow a similar methodology and use the authors approach to create our meter dataset. In this context, we rely on 50 nodes in Grid5000 to distribute the generation process of more than 4 millions meters data in a reasonably short time.

The generated data can be loaded to Storage5K in Grid5000 prior to experimentation. This allows us to load data to the target data management system directly from within Grid5000 without encountering wide areas network latencies at the time of experimentation.

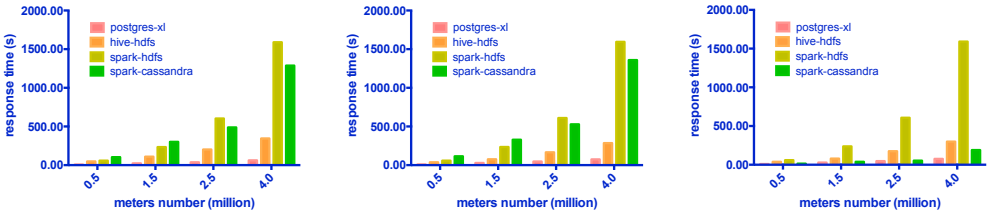
The data loading process consists of creating clients that fetch data from storage5K and load it to each storage system (Postgres-XL, HDFS, Cassandra) in the target data model (relational tables, big CSV files, and column families respectively). The number of clients equals the number of data node in each system. Figures 2a and 2b depicts the data loading total time for every storage solution when increasing the data size and when increasing the number of nodes respectively. In both figures, we can observe that data loading is much slower with Postgres-XL. This is mainly because Postgres-XL is transaction-based and data are stored in relational tables, which makes it slower to load compared to raw files with HDFS. Figure 2a shows that HDFS is faster than Cassandra with small data sizes (0.5 and 1.5 millions of meter data) but Cassandra slightly outperforms HDFS when data sizes grow further. This can be explained by the fact that with high loads at longer running time Cassandra with its peer-to-peer architecture exploits better the capacities of all its nodes compared to HDFS that direct data to nodes in a round robin manner based on chunks. Furthermore, Cassandra has an additional operational node that can host data compared to HDFS that has to dedicate at least one node to be master. In Figure 2b, the main observation is that increasing the number of nodes does not decrease the loading time. The main cause for this behavior is that with an increasing number of concurrent clients fetching data from Storage5K (which is provided by the NFS file system), copying data from source becomes even slower than loading it in the destination storage.

VI. Experimental Evaluation

A. Increasing data size

The first experiment set consists of evaluating data management and processing systems on 110 nodes with increasing volumes. The data size is increased gradually to have 4 different sizes: 0.55 million meters (4.82 billion measurements), 1.5 M meters (13.14 B measurements) 2.5 M meters (21.9 B measurements), and 4 M meters (35.04 B measurements).

Aggregation queries: Figure 3 depicts the response time with the 3 aggregation queries (Query1, Query2, and Query3 shown in Table II). For the 3 queries Postgres-XL is outperforming all the other system no matter the size of the data size. Such behavior is explained by the fact that for aggregation



(a) Query1: sum, all (b) Query2: sum, meter id range (c) Query3: sum, time range

Fig. 3: Aggregation queries evaluation: increasing number of meters

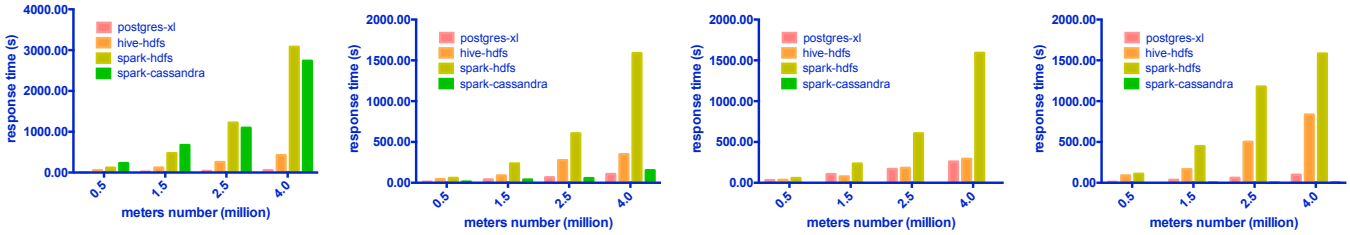
queries computation is moved to nodes according to the MPP architecture rather than moving data around. This results in a faster computation and minimal data movement. A main observation in Figures 3a and 3b is that Spark with both Cassandra and HDFS are the slowest systems. Furthermore, response time gets increasingly worse with these solutions as data size increases. For instance in Figure 3a, Spark-HDFS is 7.5 times slower than Postgres-XL with 0.55 million meters, but is almost 26 times slower when the number of meters reach 4 millions. This is mainly caused by the fact that when data and intermediate data grow by order of magnitude bigger than the size of memory, processing aggregation queries becomes very slow due to the need to swap data from disk every time. Moreover, the programming API of Spark and its acyclic graph execution engine results in many intermediate stages where data get shuffled and moved around, which in turn has an impact on performance. Figure 3c demonstrates however, that the response time of Cassandra with Spark is close to that of Postgres-XL (only 1.14 times slower when the number of meters is 2.5 millions). With its column family data model, and even with its consistent hashing model, Cassandra allows data storage ordered on the time column on disk in each node. This allows faster filtering on the time column for Query3 when using CQL (Cassandra Query Language), which in turn results in a smaller dataset to be loaded in-memory for Spark. As a result, response time is much faster since data can be fit in memory.

Selection and Filtering, and Bill queries: Figure 4 shows the response time of the selection and filtering queries, Query5, and Query6 as well as the bill query Query7. Postgres-XL is order of magnitude faster than all other systems with Query4 as shown in Figure 4a. Similar to the case with aggregation queries, Postgres-XL execution engine compute the optimal plan and sends computation to nodes whereas Spark and Hadoop are delayed with data movement in intermediate phases, in particular, with additional *order by* clause (in Query4) that starts new MapReduce jobs. Furthermore, with Spark, this introduces additional data that cannot be fit in memory for fast processing on our setup. With Query5 in Figure 4b however, Spark with Cassandra is much faster with a response time comparable to that of Postgres-XL because there is no *order by* clause while filtering on a time interval, which reduces the dataset size that can be fit in memory and results in faster processing. However, moving data around in intermediate phases is still less efficient than sending computation to nodes. In contrast, Spark on top of Cassandra is outperforming all other systems with less than 1s response time for Query6 and Query7 (Figures 4c and 4d respectively). The main reason for this extremely fast response time is that Cassandra relies on peer-to-peer architecture with consistent hashing. Therefore, when the meter ids (that are part of the row keys) are specified as query input, accessing data is straightforward based on the hash function. Moreover, with even data distribution there is a high level of parallelism to get data from different nodes simultaneously. As demonstrated in [7], The bill queries are the most complex queries with very slow response times and are considered, in general, as a bottleneck for meter data management within energy utilities. However, with the right data model and a suitable architecture such as exhibited by Apache Cassandra, these queries can become the fastest in the data management ecosystem as demonstrated in Figure 4d.

B. Horizontal scalability

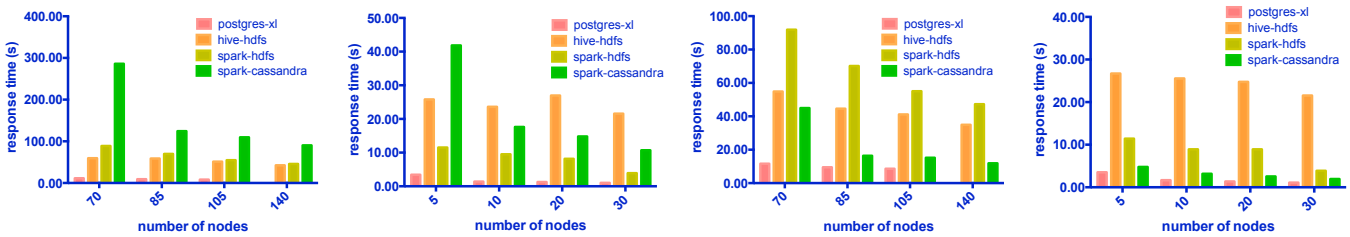
In these experiment sets, we evaluate the aforementioned systems with increasing number of nodes. In a first set, that we refer to as the big set or the big experiments, we increase the number of nodes gradually to have the following number of nodes: 70, 85, 105, and 140 nodes where we load 500 thousand meters data (4.38 B measurements) to each data management system. Since not all data and intermediate data can be fit in memory, we introduce another experiment set, that we refer to as the small experiments, where we load only 10 thousands meters to the data management systems. In this experiment set, the various configurations have the following number of nodes: 5, 10, 20, and 30.

Aggregation queries: Figure 5 depicts the scale-out properties of the various data management and processing systems with the aggregation queries Query2 and Query3. It clearly demonstrates that Postgres-XL is the most efficient solution for this type of processing outperforming all other solutions for both the big and small experiments while scaling out with increasing number of nodes. However, with this system, our experimentation have failed when the number of nodes reached 140 where tables creation was no longer possible without any exception or error message. We can observe clearly that all other solutions exhibit a performance speedup with increasing number of nodes. For instance Hadoop with



(a) Query4: time range, measurement threshold, orderby (b) Query5: measurement threshold (c) Query6: meter ids input, time range (d) Query7: Bill

Fig. 4: Bill, Selection and filtering queries evaluation: increasing number of meters



(a) Big set - Query2 (b) Small set - Query2 (c) Big set - Query3 (d) Small set - Query3

Fig. 5: Aggregation queries evaluation: increasing number of nodes

Hive is 1.41 times faster when the cluster size grows from 70 nodes to 140 nodes. Moreover, with the big experiment set, Hadoop-Hive response times is faster than Spark with both Cassandra and HDFS because Hadoop does not rely on in-memory processing (which is more efficient when data and intermediate data do not fit in memory). For instance, it is 4.8 times faster than Spark-Cassandra when the number of nodes is 70 with Query2 and 1.67 times faster than Spark-HDFS with Query3 on 70 nodes as well. However, with the small experiment set when data are fit in memory, both Spark-HDFS and Spark-Cassandra are faster then Hadoop-Hive exceeding its response time by up to 5.55 times for Spark HDFS (when number of nodes is 30 with Query2) and up to 11.10 times for Spark-Cassandra (when the number of nodes is 30 with Query3). Spark-HDFS outperforms Spark-Cassandra with Query2 because Cassandra with its architecture behaves badly with range queries. However, when filtering on time interval, Cassandra outperforms HDFS and loads a smaller data set to Spark memory resulting in a faster response time.

1) Selection and Filtering, and Bill queries: Similar to the results observed with aggregation queries, Figures 6a and 6b demonstrates that Postgres-XL is the fastest for Query4 because of the presence of the additional *order by* clause that tends to generate more intermediate phases, and thus more data movement for MapReduce based processing. Furthermore, Postgres-XL exhibits a small speedup with increasing number of nodes. For instance, it is 1.05 times faster on 105 nodes compared to 85 nodes. For Query4, Hadoop-Hive outperforms Spark-HDFS because data size exceeds the memory size, in particular with the *order by clause*. However with the small experiments Spark-HDFS outperforms Hadoop-Hive while speeding up performance by up to 2.43 times when going from 5 nodes to 30 nodes. For this type of processing Cassandra with Spark is completely inefficient exhibiting the worst performance in most cases. This is because of range queries in addition to the exploding size of intermediate data that exceeds the memory capacity. However, with small size experiment set, Cassandra outperforms Hadoop-Hive when the number of nodes increases beyond 20 nodes. This is mainly because of increasing size of available memory in the cluster that speeds up performance. In contrast, Spark-Cassandra response time is quite unmatched with Query6 and Query7. As depicted in Figures 6c, 6d, 7a, and 7b, no matter the size of the cluster, the response time does not exceed 1s, which is impressive at this scale. We can observe that Postgres-XL provides almost the same response time as Hadoop-Hive (36 times longer than Spark-Cassandra) being less suitable for Ad Hoc queries. Nevertheless, Postgres-XL is still outperforming Hadoop-Hive and Spark-HDFS with both Query6 and Query7.

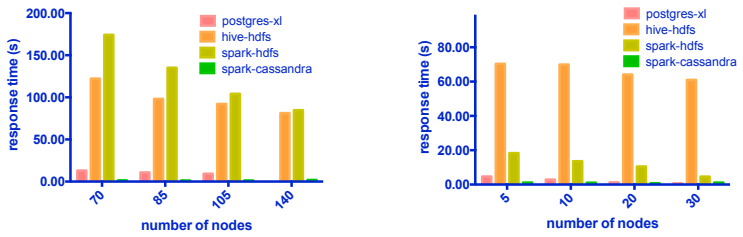
C. Data transfer

At scale with massive data sizes, every fraction of data transferred can be very penalizing for performance. In order to measure the network traffic in the cluster for the studied data processing and management systems, we rely on the *vnstat* tool [3]. We monitor and store the size of data transferred from every node in the cluster and that for every deployed system including the data loading phase and the experimentation phase with the small experiment set (from 5 to 30 nodes with 10 thousand meters data). Figure 8 depicts the obtained results. The main observation is that Postgres-XL is by far the solution that moves data around the least no matter the number of

[3]http://humdi.net/vnstat/

(a) Big set - Query4 (b) Small set - Query4 (c) Big set - Query6 (d) Small set - Query6

Fig. 6: Selection and filtering queries evaluation: increasing number of nodes



(a) Big set - Query7 (b) Small set - Query7

Fig. 7: Bill query evaluation: increasing number of nodes

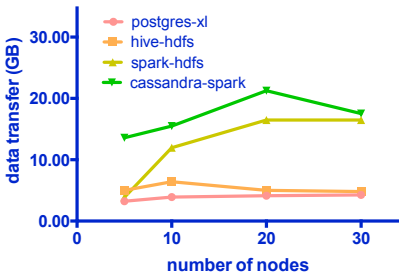


Fig. 8: Total data transfer

nodes. Such a result is expected because of the MPP model of Postgres-XL where computation is moved to nodes rather than data. This explains why it outperforms other solutions with aggregation queries and those with many intermediate phases. In addition, it is noteworthy to observe that Postgres-XL moves less data than Hadoop and Spark-Hdfs even if we consider that the data loading process does not generate data transfer with HDFS. Within Hadoop Master/Slave architecture, the master provides HDFS clients with the address of the datanode that will host the file chunk (with a replication factor of 1). As a result data is transferred from external source to nodes with no internal traffic. With Postgres-XL and Cassandra however, data transfer occurs even at loading time because of table range hashing scheme for the first and consistent hashing for the second. In both schemes, there is always a strong probability that loaded data does not fall in the hash range of a given coordinator, and thus data is transferred to another location in the cluster. Figure 8 demonstrates further that Hadoop-Hive moves less data than Spark (with Spark SQL on top of HDFS and Cassandra). The main cause is that Spark SQL and Spark tend to create more transformations and thus more intermediate phases (where data is shuffled and moved) than Hive on top of Hadoop MapReduce. In fact, Spark with its rich programming API and acyclic graph engines allow users to make many Map and Reduce steps then Hadoop Map Reduce.

VII. Related Work

In recent years, many efforts were dedicated to compare MapReduce based processing to parallel RDBMS [28], [29], [30]. In [28], Stonebraker argues that both RDBMS and MapReduce systems should be considered as complementary rather then competing systems. Furthermore, Hadoop, with its MapReduce processing and Hive query engine, was compared to RDBMS for many SQL queries in [29]. However, these studies only compare to early models of MapReduce systems and did not address the impact of data storage and layout schemes on data processing performance. In addition, data processing types similar to those of our case of meter systems were not targeted.

Data management in power grid systems is an issue that is widely investigated. Many energy utilities providers rely on legacy relational systems in order to manage their data because of their level of reliability and familiarity, such as with Teradata [31]. However in recent years, novel approaches to provide scalable data management approaches for smart grids data have been introduced [32], [33], [34]. In [32], the authors propose to leverage Cloud Computing paradigm to manage smart grid data. Public cloud infrastructures provide the necessary distributed data management and a high level of reliability to handle smart grid data. Similarly, the authors in [33], propose a distributed data management architecture to

deal with the explosion of meter and sensor data in residential distribution systems. Their architecture relies on Hadoop MapReduce processing and a distributed filesystem. More recently, a few efforts were focused more on realtime analysis of power grid data [35], [36]. Most of these aforementioned studies however, lack a thorough analysis of performance and scalability evaluation of data management architectures for smargrid data processing. In [9], a benchmarking study of meter data was introduced. In their study, the authors compare several analytic algorithms with a wide range of platforms including Madlib, Matlab, Hive and Spark.

VIII. Conclusion

With the deployment of smart meters throughout the power grid and the generation of data in small time intervals, legacy meter data management systems are overwhelmed with the data deluge. In our study, we investigated large-scale data management and processing systems and compared their scalability capacities over storing, managing and processing massive amounts of meter data. To achieve this goal, we have identified three types of processing on meter data that were implemented in four systems: Postgres-XL a parallel RDBMS, Hadoop MapReduce, Spark, a next generation MapReduce with in-memory processing framework, and Cassandra, a peer-to-peer NoSQL datastore. Our results have demonstrated that Postgres-XL with its massively parallel processing (MPP) outperforms the other systems for many queries because it reduces data transfer. Furthermore, we have showed that Cassandra with its consistent hashing scheme, is more suitable providing extremely fast response times to the billing queries and many selection and filtering queries. In contrast, we have showed that Spark computations are strongly bound to the size of available memory. Future designs of meter data management systems (MDM) should focus on the minimization of data transfer. With massive volumes of data, any fraction of data to be moved introduces important delays in data processing. Furthermore, future efforts should focus on hybrid systems and models to achieve efficient data processing at scale given no current model is suitable for all types of processing.

References

[1] "Linky, le compteur communicant d'erdf," March 2016. [Online]. Available: http://www.erdf.fr/linky-le-compteur-communicant-derdf

[2] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation*, ser. OSDI'04, 2004.

[3] "Apache hadoop," March 2016. [Online]. Available: http://hadoop.apache.org/

[4] M. Zaharia, M. Chowdhury *et al.*, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10, 2010.

[5] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, 2010.

[6] "Postgres-xl: Scalable open source postgresql-based database cluster," March 2016. [Online]. Available: http://www.postgres-xl.org/

[7] "Using the hp vertica analytics platform to manage massive volumes of smart meter data," HP Technical white paper, Tech. Rep., 2014. [Online]. Available: http://www.odbms.org/wp-content/uploads/2014/06/SmartMetering_WP.pdf

[8] Y. Jégou, S. Lantéri *et al.*, "Grid'5000: a large scale and highly reconfigurable experimental grid testbed." *Intl. Journal of High Performance Comp. Applications*, 2006.

[9] X. Liu, L. Golab *et al.*, "Benchmarking smart meter data analytics," in *EDBT: 18th International Conference on Extending Database Technology, Brussels, Belgium, March-23-27, 2015, Online Proceedings*.

[10] "Executive summary: Smart grid it systems mdm, cis, gis, scada, ems, dms, ams, mwms, drms, derms, and data analytics: Global market analysis and forecasts," Navigant Research, Tech. Rep., 4Q, 2013.

[11] "Disco," March 2016. [Online]. Available: http://discoproject.org/

[12] Z. Fadika, E. Dede *et al.*, "Mariane: Mapreduce implementation adapted for hpc environments," in *Grid Computing (GRID), 2011 12th IEEE/ACM International Conference on*, 2011, pp. 82–89.

[13] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03. ACM, 2003.

[14] "Apache hbase," March 2016. [Online]. Available: http://hbase.apache.org/

[15] F. Chang, J. Dean *et al.*, "Bigtable: A distributed storage system for structured data," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI'06, 2006.

[16] "Apache hive," March 2016. [Online]. Available: https://hive.apache.org/

[17] "Apache tez," March 2016. [Online]. Available: http://tez.apache.org/

[18] S. Schelter, S. Ewen *et al.*, ""all roads lead to rome": Optimistic recovery for distributed iterative data processing," in *Proceedings of the 22Nd ACM International Conference on Information & Knowledge Management*, ser. CIKM '13. ACM, 2013.

[19] M. Armbrust, R. S. Xin *et al.*, "Spark sql: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15, 2015.

[20] G. DeCandia, D. Hastorun *et al.*, "Dynamo: amazon's highly available key-value store," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, ser. SOSP'07, NY, USA, 2007.

[21] J. C. Corbett, J. Dean *et al.*, "Spanner: Google's globally-distributed database," in *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, ser. OSDI'12. USENIX Association, 2012.

[22] D. Karger, E. Lehman *et al.*, "Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web," in *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, ser. STOC '97. ACM, 1997.

[23] "Apache cassandra," March 2016. [Online]. Available: http://cassandra.apache.org/

[24] C. K. Baru, G. Fecteau *et al.*, "Db2 parallel edition," *IBM Syst. J.*, 1995.

[25] "Teradata," March 2016. [Online]. Available: www.teradata.com/

[26] "Cloudera cdh," March 2016. [Online]. Available: http://www.cloudera.com/downloads/cdh/5-2-3.html

[27] "Apache spark," March 2016. [Online]. Available: http://spark.apache.org/

[28] M. Stonebraker, D. Abadi *et al.*, "Mapreduce and parallel dbmss: Friends or foes?" *Commun. ACM*, 2010.

[29] A. Pavlo, E. Paulson *et al.*, "A comparison of approaches to large-scale data analysis," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '09. ACM, 2009.

[30] A. McClean, R. Conceicao, and M. O'Halloran, "A comparison of mapreduce and parallel database management systems," in *Proceedings of ICONS 2013, The Eighth International Conference on Systems*. IARIA, 2013.

[31] "How teradata makes the smart grid smarter," Teradata, Tech. Rep.

[32] S. Rusitschka, K. Eger, and C. Gerdes, "Smart grid data cloud: A model for utilizing cloud computing in the smart grid domain," in *Smart Grid Communications (SmartGridComm), 2010 First IEEE International Conference on*, 2010.

[33] N. Zhang, Y. Yan *et al.*, "A distributed data storage and processing framework for next-generation residential distribution systems," *Electric Power Systems Research*, 2014.

[34] J. Zhou, R. Q. Hu, and Y. Qian, "Scalable distributed communication architectures to support advanced metering infrastructure in smart grid," *IEEE Transactions on Parallel and Distributed Systems*, 2012.

[35] J. Yin, I. Gorton, and S. Poorva, "Toward real time data analysis for smart grids," in *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, 2012.

[36] X. Liu and P. Nielsen, *Streamlining Smart Meter Data Analytics*. International Centre for Sustainable Development of Energy, Water and Environment Systems, 2015.