Massive Smart Meter Data Storage and Processing on top of Hadoop

Leeley D. P. dos Santos
EDF R&D
1 av. Général de Gaulle
92141 Clamart, France
leeley.daio-pires-dos-santos@edf.fr

Alzennyr G. da Silva
EDF R&D
1 av. Général de Gaulle
92141 Clamart, France
alzennyr.gomes-da-silva@edf.fr

Bruno Jacquin
EDF R&D
1 av. Général de Gaulle
92141 Clamart, France
bruno.jacquin@edf.fr

Marie-Luce Picard
EDF R&D
1 av. Général de Gaulle
92141 Clamart, France
marie-luce.picard@edf.fr

David Worms
Adaltas
16 rue Saint Hilaire
25220 Thise, France
david@adaltas.com

Charles Bernard
EDF R&D
1 av. Général de Gaulle
92141 Clamart, France
charles-externe.bernard@edf.fr

ABSTRACT

Smart-grid projects are being motivated by social and environmental needs, regulatory aspects and economic constraints. In France, the deployment of smart meters was recently decided by the authorities. To manage the emerging amount of metering data, scalable solutions for storing and processing huge volume of data must be addressed. In this article, we describe a proactive work in progress carried by EDF R&D on designing and implementing a Hadoop based solution to store and mine massive time series from smart meters. The data processed represent the energy consumed by 35 million customers. We focus on the solution architecture defined on top of Hive and HBase, the implementation of domain-specific use cases and discuss how Hadoop can be efficiently used to manage structured data.

Categories and Subject Descriptors

G.3 [**Probability and Statistics**]: Time series analysis; H.4 [**Information Systems Applications**]: Miscellaneous; H.2.8 [**Database Management**]: Data mining

General Terms

Experimentation

Keywords

Big Data, Hadoop, time series, load curve, smart meter, Hive, HBase

1. INTRODUCTION

BigData Workshop, VLDB 2012 Instanbul, Turkey

The volume of commercial, financial and management data is constantly increasing. The energy domain is not (yet) a reference in the big data field, although more and more regularly cited by many major players in information technology, most often in reference to the smart-grid scenario. In France, a recent law implies the replacement of old electromechanic power meters by smart meters which are able to transmit data representing real-time electrical consumption. This operation has been already accomplished in two cities (Lyon and Tours) and is expected to cover the whole country (35 million customers) by 2020. From now and until then, scalable solutions must be addressed to cope with the new challenges arising from metering data management. These power meters will transmit data every 10 minutes which represent about 120 Terabyte/year of raw data for the whole country.

The EDF Group is an integrated energetic utility, and manages all aspects of the electricity business: deregulated activities (generation, supply, trading) or regulated activities (transmission, distribution). EDF has a strong footing in Europe, with a total of 40.2 million customers worldwide (including 35 millions in France). The EDF Group needs to anticipate future problems and investigate new methodologies in order to be prepared to deal with emerging challenges in the energy domain. This article describes a proof of concept (POC) carried out as a proactive work by the Research and Development Division (EDF R&D) in order to design and implement a Hadoop based solution to manage big data generated by French smart meters in the upcoming years. The data stored includes CRM data representing customer's information, power meter and network concentrator description data, meteorological data and massive time series representing load curves.

The remainder of this article is organized as follows. Section 2 presents related work. Section 3 presents the data description and the use cases implemented. Section 4 presents the Hadoop based solution developed. Section 5 details the experiments carried out and describes the lessons learned. Finally, Section 6 presents the conclusion and future work.

2. RELATED WORK

In the last few years, distributed storage frameworks have gained traction in reaching the needs of scaling storage and processing for massive data. In this context, Hadoop [13] has become known as the platform of choice to low cost scalable solutions based on commodity hardware for storing and processing big data in a wide range of fields: bioinformatics [10], social sciences [1], healthcare [12], sensor network [8], e-commerce [4][9], spatial analysis [7], semantic Web[5], etc. In the Web sphere, Hadoop technology is already widely used by some of the largest companies such as Facebook [2], eBay, Amazon, Baidu and Yahoo.

In the science research domain, the Magellan project[1], a joint research effort of the National Energy Research Scientific Computing Center (NERSC), Lawrence Berkeley National Laboratory and the Leadership Computing Facility at Argonne National Laboratory (ANL) have been using Hadoop in streaming mode for BLAST (Basic Local Alignment Search Tool) computations. They tested the Hadoop ecosystem on massive scientific problems in the cloud and present in a final report[2] both potential and current limitations of cloud computing for cutting-edge science. Hadoop showed promise for high-throughput data and very large workloads.

For the storage of time series, a distributed database called OpenTSDB[3] (Open Time Series Database) was developed on top of HBase at StumbleUpon[4]. OpenTSDB was written to store, index and serve metrics (real-time state information) collected from computer systems at a large scale.

In the energy sector, the Tennessee Valley Authority has built a data collection and analysis system called OpenPDC[5] on top of Hadoop[6] for the monitoring of data streaming from sensors attached to power generation systems.

3. DATA DESCRIPTION

3.1 Data Model

Figure 1 presents a simplified entity-relation model defining a metering data storage. In this data model, the central table is named PDL (point of electricity delivery) which represents the physical place (house, building, etc.) where a power meter is installed. In this work, the hierarchy of geographical dimension is as follows: city, department, region and country. A `PDL` is associated to one power meter at a time which is replaced in case of dysfunction. The power meter sends data to a network concentrator which is attached to a city.

Each city is associated to a weather station which measures weather conditions (temperature, pressure, etc.). Weather measurements, which are collected and diffused by Meteo France[7], are used in a domain-specific use case in charge of profiling electricity consumption behaviors. Tables `JDB_OC` and `JDB_EXCUR` store, respectively, data describing electrical disjunctions and tension changing registered by the power meter.

In the electricity market, a customer signs a contract with an electricity supplier (table `SUPPLIER`). The contract is associated to an electricity consumption profile (residential or professional) and a pricing policy based on daily time periods (with or without slack periods). Since the opening of French electricity market in 2007, energy regulator (table `ENERGY_REGULATOR`) entities are designated by the government to control the balance between electricity demand and offer. In data model, table `METER_READING` is the most voluminous as it stores time series from all power meters every 10 minutes during the year 2008. In this table, all the measurements are synchronized in a 10-minute time step.

Table `PDL` stores 35 millions of records and table `METER_READING` stores 1839.6 billion power meter measurements (= 35 millions x 144 measurements/day x 365 days).

3.2 CourboGen®: a Big Data Generator

As data from the French smart meters are still unavailable, we implemented a distributed data generator named CourboGen which is able to produce massive artificial data respecting the expected volume of data from real smart meters as well as the electricity consumption trends of the French population. CourboGen generates massive data to feed tables in our data model (cf. Figure 1). For each attribute of a table, values are randomly generated which are controlled by parameters calibrated with real-life values. Data generated by CourboGen are all idempotent, i.e., different executions of CourboGen based on a fixed set of parameters will generate identical data sets. The idempotent property is useful to retrieve data in case of undesired stop of the generation process.

CourboGen is based on a distributed architecture and is composed of one single master process and multiple worker processes (cf. Figure 2). Both master and worker processes are written as Node.js[8] applications and call custom C++ functions. The bi-directional communication is handled by a Redis[9] publish-subscribe channel. Redis, which is an open-source in-memory key-value data store, also store intermediate results and reporting information. Being distributed, the generation process scale proportionally to the number of registered nodes. Each worker node receives the responsibility of generating a subset of the final data set and report its progress to a master node from which it is otherwise independent. There could be multiple worker instances per server but there is a unique master orchestrating the whole process. Each worker process is programmed to optimally consume nearly 100% of a single CPU.

Each worker node produces one stream of data per table. The data stream produced is outputted in CSV or JSON format and is consumed by external processes. The communication can be achieved through multiple protocols including TCP, UDP and Unix pipes. In our experiments, data consumers are Unix scripts responsible for putting the generated data into HDFS as they are produced.

3.3 Use Cases

In our experiments, we implemented 6 business use cases. Scenario 1 represents a low-latency query which should be

[1] `http://magellan.alcf.anl.gov/`
[2] `http://science.energy.gov/~/media/ascr/pdf/program-documents/docs/Magellan_Final_Report.pdf`
[3] `http://opentsdb.net/`
[4] `www.stumbleupon.com/`
[5] `http://openpdc.codeplex.com/`
[6] `http://www.cloudera.com/blog/2009/06/smart-grid-hadoop-tennessee-valley-authority-tva/`
[7] `http://www.meteofrance.fr`
[8] `http://nodejs.org/`
[9] `http://redis.io/`

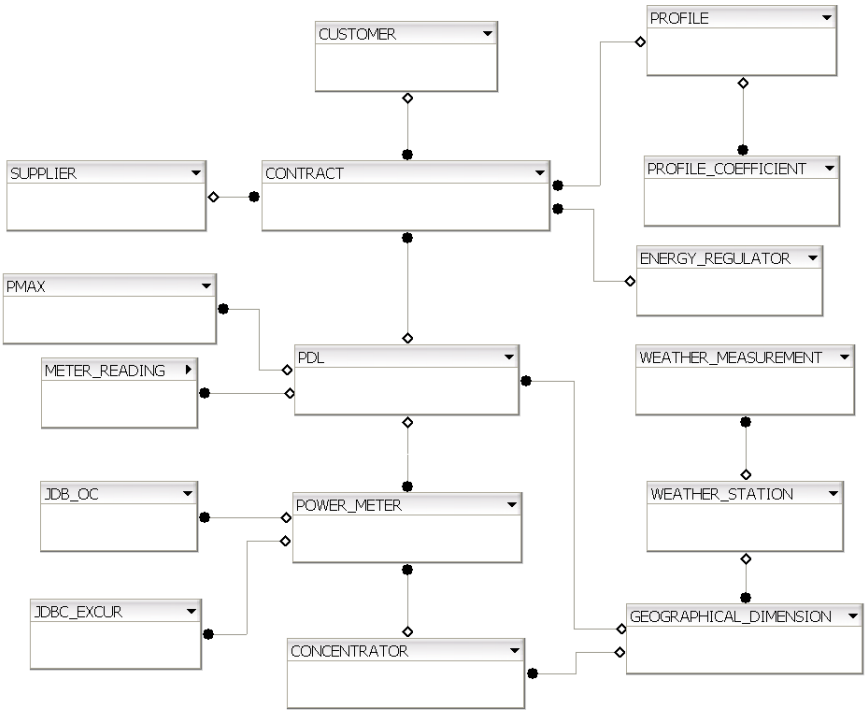

Figure 1: The data model

executed in short time (SLA constraints) due to the number of potential simultaneous requests from different customers. Scenario 2 to 6 can be executed off-line and are intended to evaluate the ability of the system to execute domain-specific use cases with no SLA constraints.

- Scenario 1 (select an individual load curve): retrieve the individual load curve of a single customer;
- Scenario 2 (calculate the monthly energy consumption): for each PDL, compute the difference between two index values representing the cumulative energy consumed by a customer. This value is necessary to compute the electricity-specific usage factor (UF) of a client;
- Scenario 3 (profile the load curve of a customer): approximate the load curve of a customer based on a public consumption profile and his particular UF;
- Scenario 4 (estimate the aggregated load curve from a group of customers): compute the sum of estimated consumption measurements from a subset of customers;
- Scenario 5 (aggregate UF values): compute the sum of UF values from a subset of customers;
- Scenario 6 (profile the load curve from aggregated UF values): approximate the aggregated load curve from a group of customers based on the aggregated UF and the public profile subscribed by these customers.

4. SOLUTION DESCRIPTION

The objective of our POC is to define a solution on top of Hadoop able to store and process massive smart meter measurements. Our challenge here is to deal with a large volume of data (≈120TB uncompressed data) in Hadoop which was not originally intended for the treatment of highly structured data. Our solution must be able to perform different kinds of treatments: from point queries on raw data till deep analytical processing on the whole data set.

4.1 Physical Architecture

The cluster we use in our experiments is composed of 2 racks and a total of 20 nodes with 2 kinds of commodity servers (7 are 1U and 13 are 2U). The 1U servers have 4x1 TB disks while the 2U servers have 8x1 TB disks. Most of the servers have 2 CPUs AMD Opteron(tm) with 8 cores each (2 have 2 x 12 cores). In total, the cluster is composed of 132 TB of storage and 336 cores. The Hadoop distribution we installed is the open source CDH3u3 from Cloudera[10].

The cluster is installed and configured with the Chef deployment tool[11]. We wrote our own Chef scripts because the existing one targeted different versions of Hadoop and were half complete. We also developed a set of distributed commands with Node.js to administer the cluster.

4.2 Logical Architecture

Our solution is composed of the building blocks described in Table 1. The injector also computes statistics to evaluate different implementation choices concerning the current data storage (format, compression, partitioning, etc.) and Hadoop tuning settings.

[10] `http://www.cloudera.com/`
[11] `http://www.opscode.com/chef/`

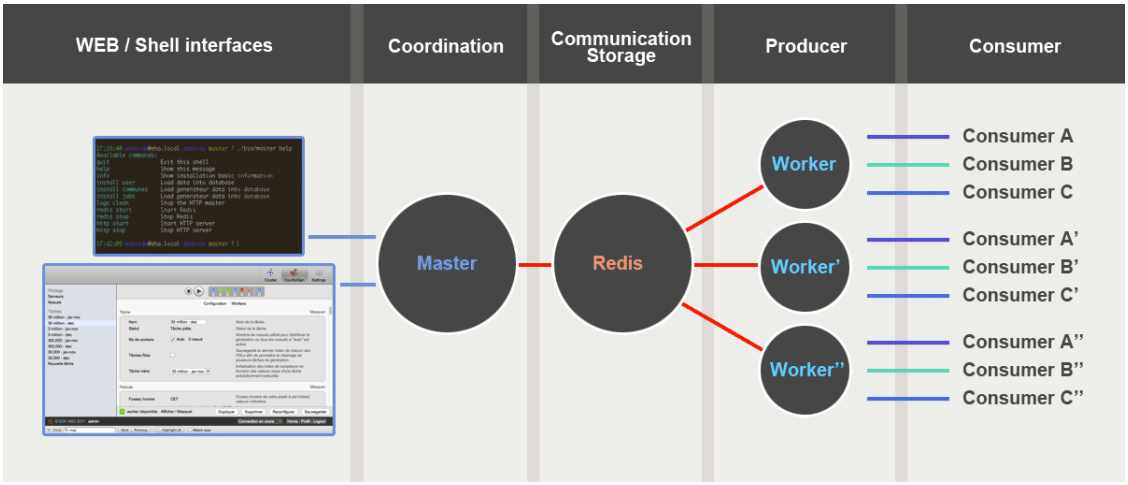

Figure 2: Courbogen architecture

Table 1: Solution building blocks

Loader	Loads data from CourboGen into HDFS and computes data quality validation
Aggregator	Summarizes smart meter measurements
Publisher	Puts raw data and aggregates into HBase for low-latency queries
API	Data access layer using HBase and Hive drivers
Injector	Runs the use case queries via scripts for Hive analytical queries and Java applications using the API to execute low-latency queries.

The logical architecture of our solution is presented in Figure 3. In a real-life scenario, metering data would come from geographically distributed network concentrators via an internet protocol (e.g. FTP). Other information currently stored in classical RDBMSs could be made available to our Hadoop environment with a tool like Sqoop[12]. In our case, we anticipate to leverage the tight integration between Sqoop and Hive. In the current experiments, input data comes directly from CourboGen.

As part of the Hadoop ecosystem, we use Hive and HBase. Hive and HBase manage tables stored in the Hadoop Distributed File System (HDFS). Those two data storage systems complement each other and allow different kinds of treatments. Hive is a data warehouse infrastructure suitable for ETL and analytical treatments. HBase is a distributed column-oriented database suitable for low-latency queries.

Hive [11] is a data warehouse framework built on top of Hadoop, used for ad hoc querying with an SQL-like language called HiveQL. It is possible to extend the language with customized functions called user defined functions (UDF). Hive queries are transparently translated to MapReduce jobs.

HBase[13] is a column-oriented database modeled after Google Bigtable [3] and built on top of Hadoop. HBase provides random real-time read/write access to data and is designed for online transaction. A table is made up of regions. Each region is defined by a startKey and EndKey, which may live on different nodes, and is made up of several HDFS files and blocks, each of which is replicated by Hadoop. A cell value is uniquely identified by `(Table, Row, Column-Family:Column, Timestamp) -> Value`. All table accesses are made through the primary key. HBase has its own Java client API instead of an SQL-like language.

In our solution, we adopted the following Hive and HBase integration approach. Ad-hoc queries are directly submitted to Hive. Low-latency queries are submitted to HBase. Hive is placed at the center of our solution. In the data loading stage, the metering measurements from CourboGen are first compressed and than placed into a staging area made of several HDFS folders. The daily volume corresponds to 50 GB with bz2 compression (instead of 327 GB whith no compression). After that, the raw data is inserted into the Hive data model.

We then apply some data type transformations in order to optmize the overall storage space of the most voluminous table `METER_READING`. Per day, each power meter sends 144 mensurements concerning PDL power consumption. The identifier field of table `PDL` is a domain-specific code stored in 8 bytes. This identifier is mapped into an internal identifier (integer - 4 bytes) used to reference metearing measurements stored in `METER_READING` table. The mapping table is stored in Hive as well as in HBase (in memory enforced). This allows to save 18 GB of daily storage space.

Since table `METER_READING` is partitionned by day (cf. Section 4.3) and metering data follow a fixed 10-minute calendar (144 measurements per day), another efficient optimization consists in rewritting the time dimension field. Instead of storing an epoch timestamp (4 bytes), we opted for storing the daily position of the measurement (in range [1,144]) into a 1 byte integer (in range [-127,15]). This allows to save 15 GB of daily storage space.

Finally, this Hive table is stored as RCFILE (cf. Section 5.1) with default compression and has a daily volume of 25 GB. Thanks to these optmizations, the final volume of one-year data stored in Hive is around 10 TB (replicated 3 times in HDFS). That represents a very important gain in terms of storage disk space compared to the original 120 TB of raw data we have in the output of CourboGen.

In order to send data from Hive to HBase, we use the build-in HBase bridge present in Hive. For each HBase table,

[12] `http://sqoop.apache.org/`
[13] `http://hbase.apache.org/`

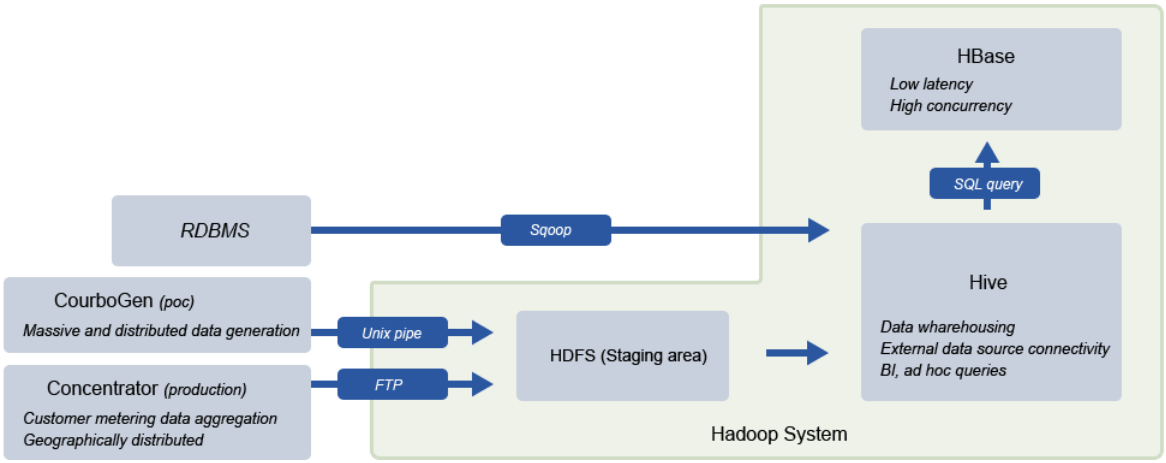

Figure 3: Logical architecture

one single Hive query selects the final data and inserts it into HBase (cf. Section 5.2).

Choosing HBase over similar NoSQL database like Cassandra[14] speeds up the deployment since it relies on Hadoop components which are already present and simplifies the data loading process by using the existing Hive connector.

4.3 Data Partitioning

Among the different data access optimization offered by Hive, the table partitioning is one for which we concentrated a lot of efforts. This optimization is very useful for data pruning and well known in database systems. Hive allows table partitioning based on hierarchical HDFS directories. The directory names contain the table partitioning column values. A good solution for the choice of the partitioning strategy is a trade-off between the top-n filtering query columns, the columns used in join conditions and the data loading policies. Thus, after reviewing our conditions, we decided to partition our metering data tables using one daily temporal dimension and two customer contract characteristics. For illustration, we ended up whith 366 x 4 x 6 = 8784 partitions. The average partition size is 1 GB. However, because the customer contract characteristics are not evenly distributed accross our population, we had a sparse distribution (from 150MB to 2GB).

In HDFS, only data insertion is allowed (and eventually appending in the latest HDFS and Hive versions). Data modification does not exist in the Hadoop ecosystem, which represents a considerable limitation compared to a traditional database system. However, this limitation has a low impact in our context because metering data are mostly append-only. The few updates concerns customer data in our operational database. Because of the relative small size of the customer data, we can consider a full import. For larger datasets, the update mechanism is achieved by merging the targeted Hive table/partition with a modified data set. This can be done via HiveQL by pointing the merged result to a new table/partition. For a partition, this can be achieved with the following HiveQL command: `ALTER TABLE <tb> SET PARTITION (partition_field=value) LOCATION '/path/to/new/folder/'`.

[14] `http://cassandra.apache.org/`

4.4 Representation Model for a Load Curve

The storage of load curves is a first-class citizen in the electricity domain. A load curve is a time series composed of measurements sent by the meters and representing the power consumption at a time for a particular customer. French smart meters can be programmed to send a power measurement every 10 minutes. Considering our daily partion scheme, a customer load curve is composed of 144 power measurements. In our data model (cf. Figure 1), these measurements are stored in the table `METER_READING` which is the most voluminous.

Our solution should be able to manage 35 million load curves and rapidly retrieve a load curve from a particular customer. In order to deal with this issue, we explored 3 different representation models to be used for the storage of a load curve in Hive: (1) *tuple*, (2) *column* and (3) *array*. For experiment purposes, we used a subset of columns in table `METER_READING`.

The tuple representation model is the most classical and consists in storing one measurement per row. Using this representation model, the system should pick-up 144 records in order to retrieve a daily load curve for a specific customer. The HiveQL code used to define the tuple representation model is detailed in listing 1. In this code, the field $index \in [1, 144]$ represents the position of the measurement in the load curve.

Listing 1: HiveQL code for load curve definition in tuple format

```
1 CREATE TABLE load_curve_tuple
2 (
3     id INT,
4     index TINYINT,
5     power INT
6 )
7 PARTITIONED BY(day STRING)
8 ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.
  columnar.ColumnarSerde'
9 STORED AS RCFILE;
```

The column representation model consists in defining a database column per measurement. In this case, we define a table containing 144 columns as detailed in listing 2. Using this representation model, the application has the advantage

of retrieving a daily load curve by reading a single record.

Listing 2: HiveQL code for load curve definition in column format

```
CREATE TABLE load_curve_column
(
id INT,
power_1 INT,
power_2 INT,
power_3 INT,
.
.
.
power_144 INT
)
PARTITIONED BY(day STRING)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.columnar.ColumnarSerde'
STORED AS RCFILE;
```

Finally, the array representation model takes advantage of a primitive type defined in Hive that is able to store multiple values of a same type as detailed in listing 3. As for the previous representation model, the array model is also able to retrieve a load curve by reading a single record. The three representation models apply a partitioning by day, i.e., measurements concerning a same day are stored in a same folder in HDFS.

Listing 3: HiveQL code for load curve definition in array format

```
CREATE TABLE load_curve_array
(
    id INT,
    power ARRAY<ARRAY<INT>>
)
PARTITIONED BY(day STRING)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.columnar.ColumnarSerde'
STORED AS RCFILE;
```

In order to compare the performance of the application, we executed a query that computes a daily aggregated load curve based on the different representation models. The daily aggregated load curve is a single curve of 144 points representing the sum up of 35 million daily individual load curves. The results obtained are detailed in Table 2.

Table 2: Representation model comparison

Format	Data volume (1 day)	Query execution time
tuple	10.1 GB (x 3 replicas)	142 s
column	8.8 GB (x 3 replicas)	77 s
array	16 GB (x 3 replicas)	78 s

The data volume column represents the volume of the daily aggregated load curve. As we can notice, the most voluminous aggregated load curve is the one using the array representation model. That can be explained by the fact that values stored in an array type need to be referenced by an index position in the array which can be space consuming. The smallest data volume is obtained by the column representation model which takes advantage of the compression model used by the RCFILE format (cf. Section 5.1). In terms of query execution time, the best value is also obtained by the column representation model, but it is only one second faster than the array representation model. The query execution time obtained in the tuple representation model is almost 2 times greater than the other formats. In conclusion, our column representation model for Hive represents a good choice both in terms of disk storage space and query execution time. However, if the focus is on the flexibility of the model, the array format is the best choice as the number of values that can be stored in an array type is not fixed in advance. Again, if the array type is used and the application of summary functions (e.g. SUM, AVG, etc.) is needed, then the devellopment of specific UDF's (user defined functions) able to manage the array type is necessary.

Table 3: Data formats and compression methods comparison

Format_Compression	Time	Size
tf	1 min 45s	6.44 GB
tf_bz	2 min 14s	1.12 GB
tf_df	2 min 16s	1.12 GB
tf_gz	48 s	1.34 GB
tf_lzo	1 min 28s	2.41 GB
tf_snappy	1 min 2s	2.55 GB
sf	2 min 3s	7.91 GB
sf_df	2 min 12s	7.32 GB
sf_bz	2 h 43 min 24s	9.9 GB
sf_gz	2 min 29 s	8.72 GB
sf_lzo	2mn 36s	8.80 GB
sf_snappy	3mn 55s	8.23 GB
rc	1mn 30s	5.78 GB
rc_df	5mn 15s	917.68 MB
rc_gz	4mn 36s	917.80 MB
rc_snappy	52s	1.85 GB
rc_lzo	38s	1.67 GB

5. EXPERIMENT DESCRIPTION

5.1 Comparison of File Formats and Compression Methods in HDFS and Hive

In our experiments, we have conducted an exploratory test in order to compare different file formats and compression methods in HDFS and Hive. The test suite is composed of similar Hive queries which create a table, eventually set a compression type and load the same dataset into a new table. Among all file formats available in Hive, we tested the text file (tf, plain text format), sequence file (sf, binary format) and RCFILE (rc, column-based format) formats and the default (df), bz, gz, lzo[15] and snappy[16] compression codecs.

Table 3 shows the results obtained. The `tf` format acts as a reference since it stores data in an uncompressed text format. It is kind of awkward to see that all the sequence file queries generate larger file size. This is maybe due to our high usage of the integer type. In terms of file size, the RCFILE formats with the default or gz compression achieve the best results. In terms of time, the RCFILE format with the lzo or snappy compression are very fast to be executed while preserving a high compression rate.

Previous work dealed with column-oriented storage techniques in Hadoop. In [6], the authors propose a binary column-oriented storage that boosts the performance of Hadoop by an order of magnitude. In our experiments, we adopted the native RCFILE format of Hadoop.

[15] `http://www.oberhumer.com/opensource/lzo/`
[16] `http://code.google.com/p/hadoop-snappy/`

5.2 Low-latency Query Execution in HBase

It is not possible to query data stored in native Hive table from HBase. HBase has its own file format on top of HDFS. On the other hand, Hive is able to query data stored in HBase tables through the HBase storage handler for Hive. However, the performance of this mechanism is not yet acceptable. Hive launches systematically map-reduce jobs for any query implying workspace load time what is prohibitive for low-latency response time (< 1sec). Interrogating directly HBase for the same query should provide results significantly faster (10 to 20 orders of magnitude).

It's not conceivable to duplicate data in both Hive and HBase. Moreover, this is not necessary to cover our functional requirements. Low-latency queries designed to turn in HBase concerns recent (one-week) data stored in the data warehouse.

In scenario 1 of the use cases analyzed (cf. Section 3.3), HBase retrieves the load curve of a single customer among 35 million customers. Data is stored in HBase in two different formats:

- *Column format*: A table with 1 simple key (power meter identifier) and 1 column familly containing all meter measurements during 1 week (with 1008 columns = 144 measurements/day x 7 days). The HiveQL code used to prepare data for HBase in the column format is detailed in listing 4. In line 10, the OMAP function[17] is used to group the daily 144 measurements of each power meter in a single map structure where timestamp are the keys and power measurements are the values.

Listing 4: HiveQL code for HBase data preparation in column format

```
1  CREATE TABLE hbase_column_format
2  (
3          id INT,
4          value MAP<STRING, INT>
5  )
6  STORED BY 'org.apache.hadoop.hive.hbase.
   HBaseStorageHandler'
7  WITH SERDEPROPERTIES ("hbase.columns.mapping"
   = ":key,cf:");
8
9  INSERT OVERWRITE TABLE hbase_column_format
10 SELECT id, OMAP(CAST(unix_timestamp(day, '
   yyyy-MM-dd') + (time_step_index+128) * 600 as
   STRING), power)
11 FROM load_curve_tuple
12 WHERE DATEDIFF(day, '2008-01-01') < 7 AND
   DATEDIFF(day, '2008-01-01') >= 0
13 GROUP BY id;
```

- *Array format*: A table with 1 composite key (power meter identifier + day identifier) and 1 column familly containing 1 column with 1 array field (storing 144 measurements). The HiveQL code used to prepare data for HBase in the array format is detailed in listing 5.

Listing 5: HiveQL code for HBase data preparation in array format

```
1  CREATE TABLE hbase_array_format (
2      id STRING,
3      values ARRAY< ARRAY< INT > >
4  )
5  STORED AS INPUTFORMAT 'org.apache.hadoop.
   mapred.TextInputFormat' OUTPUTFORMAT 'org.
   apache.hadoop.hive.hbase.
   HiveHFileOutputFormat';
6
7  INSERT OVERWRITE TABLE hbase_array_format
8  SELECT CONCAT(CAST(id as STRING), '_', day)
   composite_key,
9          values
10 FROM load_curve_array
11 WHERE ( DATEDIFF(day, '2008-01-01') >=0 )
   and ( DATEDIFF(day, '2008-01-01') < 7 )
12 CLUSTER BY composite_key;
```

[17] http://www.adaltas.com/blog/2012/03/06/hive-udaf-map-conversion/

Table 4: Execution details for scenario 1

Format	Run time	Number of concurrent sessions	Number of queries per second	Query mean time
column	1 min	100	470	0.21 s
array	1 min	100	495	0.20 s
column	5 min	100	524	0.19 s
array	5 min	100	530	0.18 s

Table 4 shows the execution details of scenario 1 in parallel mode. As we can notice, throughput access to data in *column format* and *array format* are quite similar. The main difference between these two formats relies on HBase table sizes. Table in *column format* contains 310.62 GB and table in *array format* contains 123.18 GB. This is due to the fact that the *column format* contains more cells than in the *array format*. For this last, there are far less values to be referenced which leads to disk space saving. This corroborates HBase recommendation concerning trade-off between the number of columns, the number of rows and the cell size.

5.3 Analytical Query Execution in Hive

Scenarios 2 to 6 concern analytical queries and are executed in Hive. As an example of the code used in these scenarios, listing 6 details the HiveQL used in scenario 2. In this scenario, we calculate the monthly energy consumed by each customer. It corresponds to the difference between two index values representing the cumulative energy measured by a customer power meter. Here, the monthly energy consumed are computed in the first day of the month. The example shows the computation for January 2008. Table `energy` is partitioned by day, energy regulator entity and eletric power supplier. This partition schema was chosen according to the queries executed in this table which mostly use these fields in a `GROUP BY` clause.

Listing 6: HiveQL code used in scenario 2

```
1  INSERT OVERWRITE TABLE energy PARTITION (day
   ='2008-01-01', fk_energy_regulator, fk_supplier)
2  SELECT fk_pdl, energy, fk_energy_regulator,
   fk_supplier
3  FROM
4  (
5      SELECT a.fk_pdl, (b.max_energy_index - a.
   max_energy_index) as monthly_energy, c.fk_re, c.
   fk_fourn
6      FROM
7      (
8          SELECT fk_pdl, max(energy_index)
   max_energy_index
9          FROM meter_reading
10         WHERE day = '2008-01-01'
```

```
11          GROUP BY fk_pdl
12      ) a JOIN
13      (
14          SELECT fk_pdl, max(energy_index)
    max_energy_index
15          FROM meter_reading
16          WHERE day = '2008-02-01'
17          GROUP BY fk_pdl
18      ) b ON (a.fk_pdl = b.fk_pdl)
19  ) tb;
```

Once the monthly energy consumed by a customer is computed, we use this value to calculate the usage factor (UF) of each particular client which is stored in a table nammed `energy_fu`. This UF value is used to approximate the client load curve referenced in scenario 3. The corresponding HiveQL code is detailed in listing 7. In this listing, field `coefficient_value` corresponds to the public consumption profile stored in table `profile_coefficient`. For the same reasons of the previous scenario, table `profiled_load_curve` is also partitioned by day, energy regulator entity and eletric power provider.

Listing 7: HiveQL code used in scenario 3

```
1  INSERT OVERWRITE TABLE profiled_load_curve
   PARTITION (day='2008-01-01', fk_energy_regulator,
   fk_supplier)
2  SELECT b.fk_pdl, coefficient_date, fu*
   coefficient_value as power, fk_energy_regulator,
   fk_supplier
3  FROM
4  (
5      SELECT fk_pdl, fk_profile, fk_energy_regulator
   , fk_supplier, fu
6      FROM energy_fu
7      WHERE day='2008-01-01'
8  ) b JOIN
9  (
10     SELECT fk_profile, coefficient_date,
   coefficient_value
11     FROM profile_coefficient
12     WHERE coefficient_date='2008-01-02'
13 ) a ON (b.fk_profile = a.fk_profile);
```

Scenario 4 computes the aggregated load curve from a group of users. Its HiveQL code is detailed in listing 8.

Listing 8: HiveQL code used in scenario 4

```
1  INSERT OVERWRITE TABLE
   aggregated_profiled_load_curve PARTITION (day
   ='2008-01-01', fk_energy_regulator, fk_supplier)
2  SELECT coefficient_date, sum(power) as
   aggregated_power, fk_energy_regulator, fk_supplier
3  FROM profiled_load_curve
4  WHERE day = '2008-01-01'
5  GROUP BY fk_energy_regulator, fk_supplier,
   coefficient_date;
```

Scenario 5 computes the daily sum of aggregated UF values for a group of users. Its HiveQL code is detailed in listing 9.

Listing 9: HiveQL code used in scenario 5

```
1  INSERT OVERWRITE TABLE aggregated_fu PARTITION (
   day='2008-01-01', fk_energy_regulator, fk_supplier
   )
2  SELECT sum(fu) as sum_fu, fk_energy_regulator,
   fk_supplier
3  FROM energy_fu
4  WHERE day='2008-01-01'
5  GROUP BY fk_energy_regulator, fk_supplier, day;
```

Scenario 6 approximates the aggregated load curve of a group of customers based on their aggregated UF values computed in scenario 5 and the public consumption profile stored in table `profile_coefficient`. Its HiveQL code is detailed in listing 10.

Listing 10: HiveQL code used in scenario 6

```
1  INSERT OVERWRITE TABLE profilage_fu PARTITION (day
   ='2008-01-01', fk_energy_regulator, fk_supplier)
2  SELECT sum_fu*coefficient_value as power,
   fk_energy_regulator, fk_supplier
3  FROM profile_coefficient a
4  JOIN aggregated_fu b ON (a.day = '2008-01-01' AND
   b.day = '2008-01-01')
5  GROUP BY fk_energy_regulator, fk_supplier, day;
```

Table 5 shows the mean execution time of these scenarios both in sequential mode as well as in parallel mode. For sequential mode, queries are executed in a dedicated fashion without any concurrent task. In the parallel mode, concurrent mapreduce tasks are executed within a same scenario. The volume of data treated by each scenario is 1 million customers.

Table 5: Execution time (in minutes) for scenarios 2-6

Use Case	Sequential Mode		Parallel Mode	
	1 day	1 week	1 day	1 week
scenario 2	1.44	10.10	1.56	3.00
scenario 3	27.87	195.09	28.50	31.01
scenario 4	10.71	74.99	15.97	19.58
scenario 5	6.10	42.70	7.45	8.39
scenario 6	0.86	6.08	0.92	2.43

As we can notice in sequential mode, the execution time for one day is not linear to 1 week (not equal to 7 x one-day execution time). This is due to the fact that data is distributed and replicated in different nodes of the cluster. As a consequence, the execution of a same query on data from different weekdays will present different execution times. For queries on one day, the execution time in parallel mode is greater than the execution time in sequential mode. This is due to the fact that physical resources are shared by different mapreduce tasks. On the other hand, for the one week executions, queries in parallel mode are much faster than queries in sequential mode. These results clearly show the potential of Hadoop to execute representative use cases involving massive data from the electric power domain.

6. CONCLUSION

In this article we presented a proactive work in progress at EDF R&D. The goal is to evaluate the strengths and limits of Hadoop for storing and processing massive time series representing metering data. The description of our experiments gives insights on how to ingeniously take advantage of Hadoop potentials. Our experiments allow us to drill down the following observations:

- An open source solution as Hadoop can be used to store and process data from massive smart metering using commodity hardware and can efficiently and successfully answer the needs of representative use cases in the electricity domain.
- The choice of partitioning fields plays a key role in the system performance.

- An efficient combinaison of good choices for data modeling, partionning, file format and compression dramatically reduce the data volume while improving the overall performance.

- Combining Hive and HBase is a promising scalable solution for the storage and processing of time series such as metering data. Hive allows the execution of complex analytical queries and HBase cope with low-latency queries.

In this work, we have tested different representation models for a load curve both for Hive and HBase. In Hive, The array type offers a flexible representation model compared to the column representation model which is based on a fixed number of columns. Despite the storage volume required, the array representation model presents a good response time. The use of customized representation models requires the development of new UDF's to process data. We have already started working on it[18]. The potential of the array representation model is corroborated by the small storage space needed in HBase.

Future work will focus on the development of custom time series storage formats both on Hive (serializer/deserializer) and HDFS (file format) levels, the evaluation of other Hadoop-based solution to perform distributed data mining techniques (e.g. Apache Mahout[19]) and statistical analysis on big data (e.g. rhipe[20], RHadoop[21], etc.).

7. REFERENCES

[1] M. Bautin, C. B. Ward, A. Patil, and S. S. Skiena. Access: news and blog analysis for the social sciences. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 1229–1232, New York, NY, USA, 2010. ACM.

[2] D. Borthakur, J. Gray, J. S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, R. Schmidt, and A. Aiyer. Apache hadoop goes realtime at facebook. In *Proceedings of the 2011 international conference on Management of data*, SIGMOD '11, pages 1071–1080, New York, NY, USA, 2011. ACM.

[3] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[4] Y. Chen, D. Pavlov, and J. F. Canny. Behavioral targeting: The art of scaling up simple algorithms. *ACM Trans. Knowl. Discov. Data*, 4:17:1–17:31, October 2010.

[5] M. Farhan Husain, P. Doshi, L. Khan, and B. Thuraisingham. Storage and retrieval of large rdf graph using hadoop and mapreduce. In *Proceedings of the 1st International Conference on Cloud Computing*, CloudCom '09, pages 680–686, Berlin, Heidelberg, 2009. Springer-Verlag.

[6] A. Floratou, J. M. Patel, E. J. Shekita, and S. Tata. Column-oriented storage techniques for mapreduce. *Proc. VLDB Endow.*, 4(7):419–429, Apr. 2011.

[7] D. Guo, K. Wu, J. Li, and Y. Wang. Spatial scene similarity assessment on hadoop. In *Proceedings of the ACM SIGSPATIAL International Workshop on High Performance and Distributed Geographic Information Systems*, HPDGIS '10, pages 39–42, New York, NY, USA, 2010. ACM.

[8] C. Jardak, J. Riihijärvi, F. Oldewurtel, and P. Mähönen. Parallel processing of data from very large-scale wireless sensor networks. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 787–794, New York, NY, USA, 2010. ACM.

[9] J. Jiang, J. Lu, G. Zhang, and G. Long. Scaling-up item-based collaborative filtering recommendation algorithm based on hadoop. In *Proceedings of the 2011 IEEE World Congress on Services*, SERVICES '11, pages 490–497, Washington, DC, USA, 2011. IEEE Computer Society.

[10] R. C. Taylor. An overview of the hadoop/mapreduce/hbase framework and its current applications in bioinformatics. *BMC Bioinformatics*, 11(Suppl 12):S1, 2010.

[11] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive - a petabyte scale data warehouse using Hadoop. pages 996–1005, Mar. 2010.

[12] F. Wang, V. Ercegovac, T. Syeda-Mahmood, A. Holder, E. Shekita, D. Beymer, and L. H. Xu. Large-scale multimodal mining for healthcare with mapreduce. In *Proceedings of the 1st ACM International Health Informatics Symposium*, IHI '10, pages 479–483, New York, NY, USA, 2010. ACM.

[13] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, 1 edition, June 2009.

[18] `http://www.adaltas.com/blog/2012/03/06/hive-udaf-map-conversion/`

[19] `http://mahout.apache.org/`

[20] `http://www.rhipe.org/`

[21] `http://www.revolutionanalytics.com/products/r-for-apache-hadoop.php`