iBig Hybrid Architecture for Energy IoT: When the power of Indexing meets Big Data Processing!

Houssem-Eddine Chihoub, Christine Collet

iBig Hybrid Architecture for Energy IoT: When the power of Indexing meets Big Data Processing!

Houssem Chihoub and Christine Collet
Univ. Grenoble Alpes, CNRS, Grenoble INP, Enedis*
LIG, F-38000 Grenoble, France
houssem.chihoub@imag.fr, christine.collet@grenoble-inp.fr

***Abstract*—Nowadays, IoT data come from multiple sources and a large number of devices. To manage them, IoT frameworks rely on Big Data ecosystems hosted in the cloud to provide scalable storage systems and to achieve scalable processing. Although these ecosystems scale well to process large sizes of data, in many cases this is done naively. Many datasets, such as IoT Energy measurement data, consist, even partially, of attributes that can be indexed to avoid unnecessary and costly data scan at these scales. In this work, we propose the *iBig* architecture that provides secondary indexing to Big Data processing for energy IoT datasets. Indexes are considered as metadata stored in a separate component integrated in the ecosystem. Subsequently, MapReduce-based and MPP (massively parallel processing)-based processing leverage indexing to handle only relevant data in a given dataset. Our experimental evaluation on the Grid5000 cloud testbed demonstrates that performance gains can exceed 98% for the MapReduce-based Spark and 81% for the MPP-based Drill for Energy IoT Data. Furthermore, we provide comparison insights about the Spark and Drill frameworks when processing the whole dataset or only with relevant data.**

I. Introduction

In the digital world we live today, the number of connected devices is exploding. Gartner says that 8.4 Billion connected *things* will be in use in 2017, which is 31% increase from 2016 [1]. A typical case of large-scale IoT application can be found within energy utilities. In their efforts to transform power grids into smart grids, utilities rely on massive deployments of sensors and smart meters at the scale of a country or even a continent. For instance, in France Enedis has an on-going plan to deploy 35 million smart meters by the year 2021 [2]. In this context, more and more IoT data are collected providing fine grain insights about client consumption profiles and the behavior of the power grid. These IoT data are critical towards more efficient management of energy resources and provides new levels of efficiency for business applications. However, the frequency in which data are generated and collected from millions of sensors and smart meters makes the task of data management and processing very complex. This complexity is due to the huge volumes of data collected and stored over long periods of time, as well as the performance requirements on accessing and processing them.

In order to deal with tremendous volumes of IoT data and cope with its challenges, energy utilities have started to rely on Big Data ecosystems for scalable storage and processing. In these ecosystems, data are stored, typically, under almost any formats, structured, semi-structured, and unstructured on a distributed file system or a cloud storage system. In contrast, data processing is based primarily on the MapReduce programming model [3]. Since early MapReduce frameworks [3] [4], remarkable efforts were carried out to provide an even more scalable and efficient data processing. Recently, a new generation of faster in-memory processing frameworks with richer programming APIs have emerged, such as Spark[5] and Flink[6]. More recently, in an attempt to lower analytical processing latency, Massively Parallel Processing (MPP) models -that favor independent processisng where computation are moved to data rather than the opposite- were revisited in this context. Subsequently, a new class of Big Data processing engines such as Apache Drill [7], Google Dremel/BigQuery [8], and Presto[9] have been rising to challenge MapReduce systems.

Most of the aforementioned Big Data processing frameworks scale out with large datasets stored on distributed file systems, specifically with exploratory analytics that need to scan the entire data. In contrast, data search and scan can be extremely costly over large datasets, in particular if it is not necessary. For instance, we consider the case of IoT measurement data. Many queries over measurement data are selective where only a subset of data is needed for processing. However, within todays' frameworks all the files or objects in a given dataset need to be scanned to filter data, hence incurring significant delays considering Big Data sizes. In this work, we propose the new *iBig* hybrid architecture. The primary goal is to provide secondary indexing to energy IoT Big Data processing. To this end, we introduce a new architecture managed exclusively by a new component where the indexing metadata is separated from storage and processing. Upon request, the indexing component provides a relevant selection of files that can be used to push down filtering prior to data processing and thus, reduce data scan cost. Furthermore, we provide an efficiency study of indexing in the proposed architecture for various distributed processing models.

The following is a summary of our contributions.

- Design of the new iBig hybrid architecture that combines Big Data processing frameworks with indexing for Energy IoT data management. Indexing is provided as metadata stored in an independent component added to the ecosystem.

*This work was carried out in collaboration with the Enedis Information Systems Division (Franck Atgie and Tarik Loiseau)

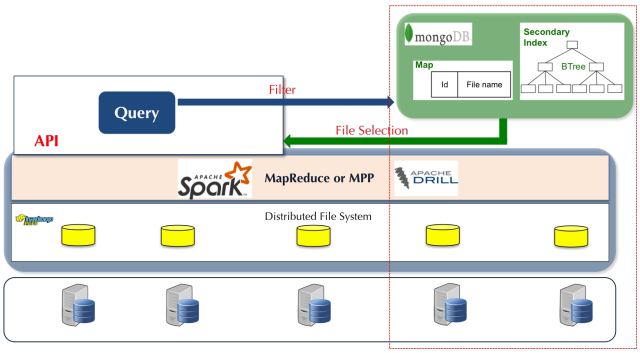


Fig. 1: iBig Hybrid Architecture

- Study of indexing efficiency in the proposed architecture. This is a very important step to understand when to use indexing and to provide scale-out properties.
- Thorough experimental evaluation to measure the new architecture gains for an energy IoT benchmark.
- Fair experimental comparison of a Map Reduce system and an MPP system where both share the same underlying storage environment.

The remaining of this paper will be is organized as follows. Section II introduces the iBig architecture and Section III details its main aspects. Section IV presents the methodology we develop. The evaluation of our architectures is extensively discussed in Section V. Sections VI and VII give an overview of related work and sum up our conclusions respectively.

II. iBig Architecture Design

Figure 1 demonstrates the general design of our architecture. Datasets are, generally, stored in files under directories in a distributed file system (HDFS in this particular setup). Data processing can be carried on either Spark, a next-generation of MapReduce computation framework or Apache Drill an MPP framework. Both processing frameworks can operate on the same dataset stored in HDFS. This configuration is current within many organizations. In our iBig architecture, we introduce a new component that provides indexes as metadata about datasets. For both Spark and Drill, a new library is designed to allow users and developers to specify filters on indexed attributes in the dataset prior to loading and processing data in either framework. The library calls result in a subset of files (file selection) that contain at least an element of relevant data needed for further processing, whereas the other files are filtered out. This in turn, reduces the wasted efforts of scanning and processing the latters. The indexes on selected attributes are built in B-Trees [10] that provide a search complexity of log(N) with N being the size of the dataset. Henceforth, providing the opportunity for important gains considering that sizes in the era of Big Data are very "Big". For instance, searching measurements in log(N) time from trillions of energy IoT measurements can reduce processing times by order of magnitude for selective queries.

The indexing component hosts index data along a map that determines which data belongs to which file. Upon request the index is scanned in order to build a file selection to return to the processing component. The file selection is further used either within Spark or Drill for processing. In our architecture we do not store the split index of attributes inside files to preserve the efficiency of indexing in the architecture as demonstrated in Section III-C. The component that provides file selection based on index data is built using MongoDB [11]. This choice was motivated by the fact that MongoDB provides built-in indexes (including secondary and spatial ones), has a flexible document data model, and scales-out with data sharding supported at the system level with local indexing.

MapReduce Vs. MPP

The efficiency of indexing in the proposed architecture is reliant on the big data processing model and distribution among other factors. In recent years, two paradigms of Big Data processing have gained large merits: *MapReduce* and *Massively parallel processing (MPP)*. MapReduce was rediscovered at Google as an efficient programming model to distributed Big Data processing [3]. Inspired by divide and conquer, it consists of defining a Map phase to decompose a big problem into a set of small sub problems and a Reduce phase to address the small subproblems. In between, a Shuffle phase to redistribute data to their respective mappers is performed. The Shuffle relies, generally on a sort algorithm or a hash algorithm and can be costly in terms of performance due to random data transfer. Thanks to the Hadoop implementation [4], the MapReduce model has been widely adopted. Lately, Spark [12], a new generation of in-memory MapReduce processing with richer and friendlier programming API, has emerged to be the most popular framework.

More recently, many efforts to provide low-latency SQL-like processing of Big Data have revisited the Massively Parallel Processing (MPP) model. This latter has written the success story of shared-nothing distributed database management systems. In contrast to MapReduce, MPP consists of moving computation to data rather than data to CPU thus, enhancing data locality. This is achieved relying partly on cost-based opitmizations implemented with MPP systems. This way, many processors can process data locally, transferring their intermediate results only when necessary. Nowadays, many systems such as Apache Drill [7] and Presto [9] adopt MPP and are integrated efficiently into the Hadoop ecosystem.

In many aspects, MapReduce and MPP share the same approach consisting of divide and conquer to provide scalable processing of large and voluminous datasets. However, these two paradigms diverge on how to divide the initial problem. MapReduce frameworks fully focus on exploiting the computation and CPU power of the underlying infrastructure even if it implies redistributing data at the cost of additional delays. In contrast, MPP frameworks focus on data locality (to amortize the delay impact of redistributing data) where in many type of processing the load is not fully balanced and the CPU power is not leveraged to its fullest. As a result, the efficiency of indexing in the iBig architecture can vary according to the

processing model and the impact of either data transfer or load unbalancing, as it will be demonstrated later on.

III. iBig components and Efficiency

A. Data Processing

Our iBig architecture supports two processing models: MapReduce provided by Spark and MPP provided by Drill.

Spark: Spark is implemented in the Scala 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. To facilitate data manipulation, Spark introduces the abstraction of *resilient distributed datasets (RDDs)*, an in-memory read-only collections of objects partitioned across a set of machines that can be recovered in case of failures. Data must be loaded to RDDs prior to processing. Within Spark, the Spark SQL [13] module provides an SQL query engine that generates Spark tasks for SQL statements. Spark SQL relies on the introduced DataFrame API to store data in a columnar layout in memory and perform relational operations on both external data sources and spark built-in collections.

Drill: Apache Drill [7] is an open-source Big Data SQL query engine that is open-source and can be integrated in the Hadoop ecosystem. It was partly inspired by Google Dremel [8]. It implements an MPP model to process data coming from different datastores including file systems (with various file formats), NoSQL datastores, and relational systems. Drill supports nested document data model, and like Google Dremel converts nested data into flatten columnar formats for in-memory internal execution. Moreover, Drill relies on cost-based and rule-based optimizations to generate optimal locality-aware execution plans. Although, Drill provides SQL query processing as the only language support, it allows developers to write User Defined Functions (UDFs) in order to customize and develop their own specific functions and computations.

B. Data Indexing

Our indexing component is hosted entirely on MongoDB.

MongoDB: MongoDB [11] is an open-source document datastore. It stores data in a BSON (JSON-Like) document data model to provide flexibility and easier evolution of data structure. MongoDB is a distributed system by design and supports built-in data sharding to provide scale-out properties. Moreover, MongoDB supports a wide range of indexes for a faster data search. The achieved performance and scalability together with secondary indexes support makes MongoDB the ideal candidate to host our indexing component. Indexed attributes are stored in a document collection with their mapped file names and if necessary the row key from the dataset. Secondary indexes are then built within MongoDB on corresponding attributes (fields in MongoDB). With sharding in MongoDB, indexes are local to shards. This allows a parallel scan of indexes, which can improve efficiency as shown in the next section. This latter property is key to provide efficient indexing in a scale-out architecture.

When new files are to be ingested into HDFS, updating the index data on them is necessary before it is possible to use indexing. Data ingestion processes rely generally on either streaming or loading batches that put data in immutable files in a data lake or an OLAP system (unlike transnational systems). Until new data files are indexed, queries must scan all data.

C. Indexing Efficiency

Indexing data in our architecture can potentially provide large gains, as with IoT data for instance. However, relying on indexing has an overhead. The overhead for some types of processing can be so important and results in worse performance. A typical case would be for queries that need to process the whole data. For instance, aggregations on the entire dataset. In such a case, the execution time would consist of the indexing overhead in addition to the original execution time of the query (without indexing). To study the efficiency of indexing in the iBig architecture, we model its potential gains. To this end, we consider a dataset S with a size $|S|$. Since the same type of processing is applied with or without indexing, the main difference relies within scanning and filtering data. Accordingly, we define the efficiency in Formula 1 as proportional to the fraction of the response time of index scan added to the resulting filtered data scan, and the response time of scanning of all data.

$$Eff = 1 - \frac{T_{scan}(Index) + T_{scan}(FilteredData)}{T_{scan}(AllData)} \quad (1)$$

The scan time is proportional to both the data size and the number of nodes that data are distributed across. In the following, we assume an uniform distribution and partitioning of data across nodes in the cluster. Subsequently, we define the efficiency in Formula 2 where N_{idx} is the number of nodes on which index data are distributed (a.k.a MongoDB). Similarly, N_{proc} is the number of nodes on which the dataset is distributed that is the number of datanodes in HDFS. Moreover, S_{sel} is the set of selected data after indexing, i.e. the data to be scanned in the indexing component in order to build the file selection. Finally, F_{sel} is the file selection that results from indexing and $|f|$ is the average size of files. It is noteworthy to observe that $|F_{sel}| \times |f| \geq |S_{sel}|$ and that these two sizes that determine the most the efficiency of indexing.

$$Eff = 1 - \frac{log(\left|\frac{S}{N_{idx}}\right|) + \frac{|S_{sel}|}{N_{idx}} + \frac{|F_{sel}| \times |f|}{N_{proc}}}{\frac{|S|}{N_{proc}}} \quad (2)$$

After simplification, the result is shown in Formula 3.

$$Eff = 1 - \frac{N_{proc} \times log(\left|\frac{S}{N_{idx}}\right|) + \frac{N_{proc} \times |S_{sel}|}{N_{idx}} + |F_{sel}| \times |f|}{|S|} \quad (3)$$

The Efficiency metric Eff can determine whether the query would benefit from indexing or not. If $Eff < 0$, then indexing should not be used because it would result in a slower

$$Eff = 1 - \frac{T_{scan}(Index + FilteredData) + T_{partition}(FilteredData) + T_{mscan}(FilteredData)}{T_{scan}(AllData) + T_{partition}(AllData) + T_{mscan}(AllData)} \quad (4)$$

$$Eff = 1 - \frac{T_{scan}(log(\left|\frac{S}{N_{idx}}\right|) + \frac{|S_{sel}|}{N_{idx}} + \frac{|F_{sel}|\times|f|}{N_{proc}}) + T_{partition}(|F_{sel}| \times |f|) + T_{mscan}(\frac{|F_{sel}|\times|f|}{N_{proc}})}{T_{scan}(\frac{|S|}{N_{proc}}) + T_{partition}(|S|) + T_{mscan}(\frac{|S|}{N_{proc}})} \quad (5)$$

execution time. Efficiency estimation can be computed relying on collected statistics about the dataset. These statistics can be stored in histograms to provide estimated values of the parameters described above for a given query at runtime.

Efficiency with Spark. The previous efficiency estimation is applicable with general architectures where data are fetched from the filesystem, then filtered before in-memory internal execution. This is true for Apache Drill as well. However, with Spark the approach is quite different and the efficiency estimation is more complicated. In fact, in Spark processing, the dataset is first uploaded to an RDD, a DataFrame, or DataSet, which are all in-memory collections of data. Upon loading time, data are partitioned across the Spark nodes, then processing starts from this point further, including particularly data filtering. For this reason, data indexing has even greater potential. With indexing in our architecture, much of data filtering is done prior to loading data to spark. Therefore, the effort needed to partition data in spark is reduced accordingly along with data scan from memory. As a result, the Efficiency is given by Formula 4 where T_{scan} is the data scan time from the filesystem, $T_{partition}$ the time needed to partition data in spark, and T_{mscan} is the scan time from memory.

Furthermore, the efficiency metric, given by Formula 5, can be computed based on estimating sizes of filtered data.

To further simplify the previous formula, an estimation of the cost of partitioning data in spark should be developed. It should include the implied data transfer and data scan in distributed memory as opposed to disks from the filesystem. The main difficulty resides in the fact it is difficult to provide a relative cost between scanning data from disk, scanning data from memory, and transferring data.

IV. Experimental Methodology

A. iBig Implementation

We have designed a Python-based tool to integrate our indexing component with the data processing frameworks Spark and Drill. In this context, a library that allows users to specify filters on indexed attributes is available. Based on this filter, MongoDB is solicited relying on the PyMongo driver to provide the file selection of relevant data to be loaded to a Spark DataFrame. For this purpose, we use PySpark to execute queries. With Drill, our approach is different. Drill SQL-based syntax considers a file name or a file directory as a table name. Hence, providing multiple file names does not comply with its syntax. To overcome this issue, we use Drill RESTful API to generate physical plans for our queries. We thereafter, inject the indexed file selection into the JSON representation of the physical plan. The plan is then submitted for execution.

B. Data Generation

For privacy reasons, it has not been possible for us to get real smart meter and IoT data from energy utilities. Fortunately, it was possible to generate realistic datasets that illustrate meter data as described in [14]. In a previous work [15], we have used the authors approach to create our meter dataset. Accordingly, We generate meter data for 50000 meters over a period of one year with a measurement every one hour and a total of 438 million measurements.

C. Benchmarking

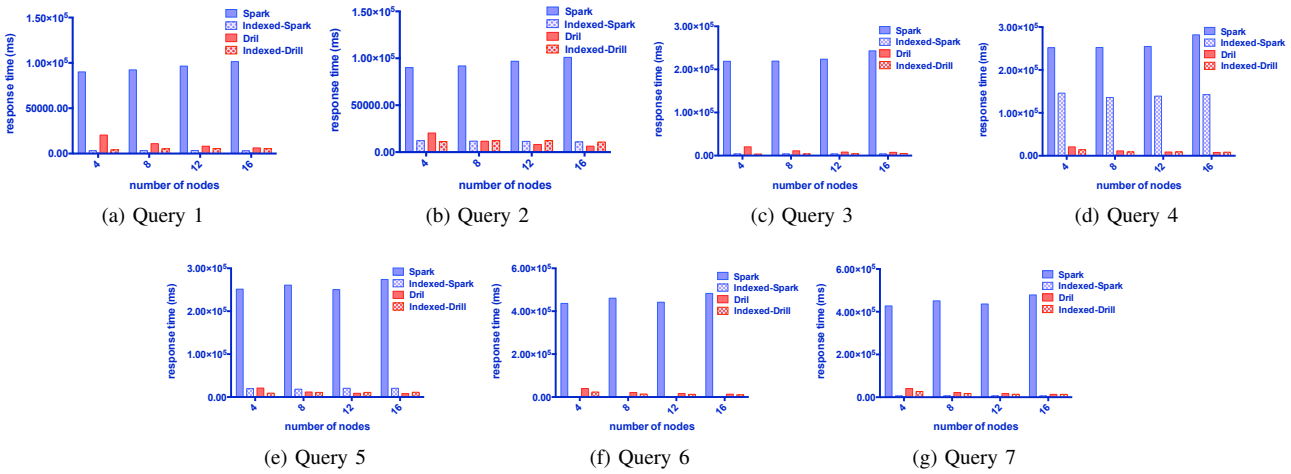
Meter data are used for multiple purposes in the smart (power) grid: bill computations, consumption analysis, power generation forecasting, fraud detection etc. Most of these services and applications exhibit three types of meter data processing queries as shown in [16]: aggregation queries, selection and filtering based on some predicates (e.g. to filter based on a consumption threshold), and bill computation queries that exhibit the most complex processing for meter data management systems [16]. In this context, we introduce seven (7) queries that illustrate processing types on meter data. The queries are summarized in Table I.

D. Setup

We run our experiments on Grid5000 cloud and grid testbed [17] in France. We use up to 16 nodes in the Parasilo cluster located in the Rennes site. All nodes are equipped with 2 Intel Xeon E5 CPUs having 8 cores each and 2.4GHz speed. Every node has memory size of 128GB and a Hard Drive Disk of 600GB used in our experiments. The nodes are interconnected relying on two 10 Gigabit Ethernet links. Our deployments rely on HDFS within Cloudera CDH-5.2.3 distribution, Spark-2.1.0, Apache Drill-1.9, and MongoDB-3.2.11. MongoDB is deployed on only one node co-located with HDFS datanode, and either a Spark Slave or a Drillbit (drill processor). In our dataset, two attributes are indexed: the *measurement* attribute, a measurement value about client consumption read from a meter, and the *meter_id* attribute, which is a unique id for a given smart meter. Beforehand, indexes data are loaded to MongoDB. Thereafter, the BTree-based secondary indexes on both *measurement* and *meter_id* are built in approximately 1350 seconds for the first, and 880 seconds for the second.

Query	Description
Query1	Sum of all measurements (consumption of all meters) for a range of 300 meters (over 1 year data)
Query2	same as Query1 but for a range of 1200 meters
Query3	Sum of measurements for a range of 300 meters in a 2-month time interval
Query4	Selection of rows based on a highly restrictive consumption threshold, sorted by their meter ids
Query5	Selection of rows based on a wider consumption threshold for a range of 2000 meters, all sorted by their meter ids
Query6	Compute the bill for a given client (meter) following the tarif blue billing rules of EDF (power vendor)
Query7	Compute the bill for 300 clients (meters) following the tarif blue billing rules of EDF(power vendor)

TABLE I: Processing Queries on meter data



(a) Query 1 (b) Query 2 (c) Query 3 (d) Query 4

(e) Query 5 (f) Query 6 (g) Query 7

Fig. 2: Evaluation of queries in iBig

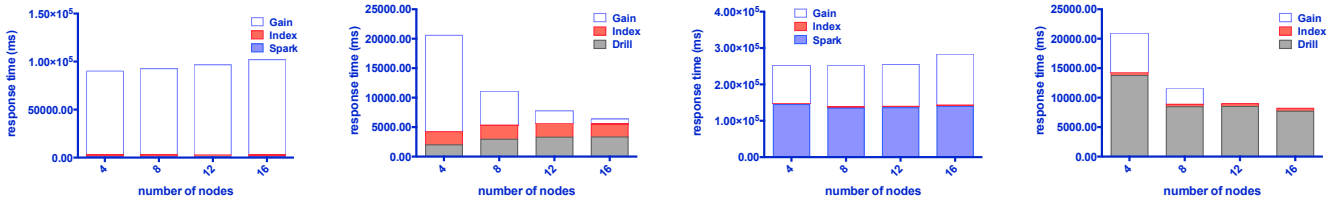
V. Experimental Evaluation

A. Evaluation of the iBig Architecture

In Figure 2, we show the response time for both Spark and Drill with and without indexing in the iBig architecture. As depicted, our indexing solution reduces considerably the response time achieving high levels of efficiency, in particular for Spark. In fact, Spark with indexing exhibits comparable latency to Drill (with and without indexing), which is due to large gains in our architecture. Furthermore, with bill queries, Query6 and Query7 (Figures 2f and 2f respectively) indexing makes Spark extremely fast. However, with Query4 (depicted in Figure 2d), even indexing can not make Spark at the same level as Drill. In fact, data transfer both at partitioning time and in intermediate phases, in particular with the sort clause, introduces important delays with regard to the file selection of relevant data at input. Another observation is that Drill can rapidly become faster than indexing with added nodes in the cluster, and that when queries have wide filters on the indexed attribute. This is true for instance, with Query2 (in Figure 2b) when the number of nodes increases beyond 8 nodes because of the wide data selection of its filter. Obviously, one solution is to increase the number of nodes for the indexing component for higher efficiency as shown in Section III-C.

An additional important result, is the exhibited performance of Spark in comparison to Drill. For all queries, Drill is order of magnitude faster (e.g. 97% faster with Query5). Both systems are written in Java and implement an optimized columnar in-memory execution (with DataFrames in Spark) and vectorization. Drill is much faster, since it is data locality-aware with its MPP model, which in turn, reduces significantly data transfer in intermediate phases. Furthermore, data transfer delay dominates the gained CPU power in Spark. Prior to processing, data are first uploaded to a DataFrame, an in-memory distributed and structured collection of data. When reading data from files, the entire dataset is uploaded and then partitioned across Spark nodes, which can incur significant and unnecessary data transfer along with the useless data scan.

Zoom on indexing gain: In this section, we further zoom on the efficiency of indexing in our architecture and the gains obtained with both Spark and Drill. Figure 3 depicts the gains with the aggregation queries Query1 and Query4. Spark gains for Query1 exceed 98% no matter the number of nodes in the cluster. Such extraordinary efficiency is due to the fact that with indexing, most of irrelevant data are filtered out before loading data to memory. As a result, the data scan and partitioning time in Spark is negligible with selected data compared to the entire dataset. The data size to be scanned in memory is also significantly reduced. Gains are less important with Drill because it was already fast and filters data at source before loading them to memory. Despite this, indexing with Drill in our architecture, has allowed Query1 to save from 12%



(a) Query1 Spark (b) Query1 Drill (c) Query4 Spark (d) Query4 Drill

Fig. 3: Gains with Aggregation Queries

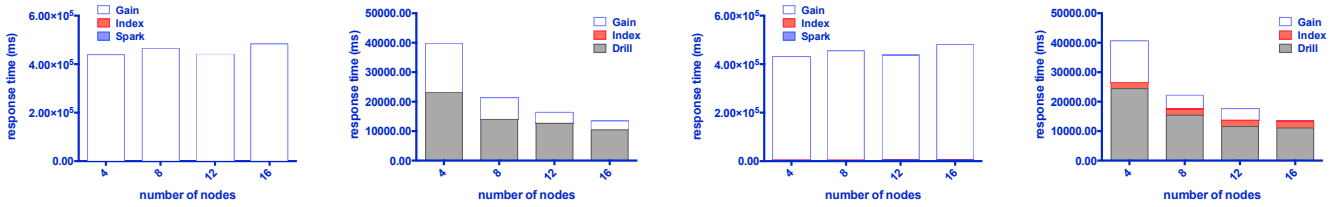
of processing time when the number of nodes is 16 to 79% when the number of nodes is 4 as demonstrated in Figure 3a.

The gains exhibited with Query4 are quite different. Query4 is a selection and filtering query with a sort clause that leverages indexing in our architecture on the *measurement* attribute. Indexing with Spark achieves between 41% and 49% gains. With Drill however, the gains are 31% and 21% when the number of nodes is 4 and 8 respectively. Indexing with Drill becomes inefficient beyond this number of nodes in this setup. The cause for Spark Gains drop in this case, lies within the *measurement* attribute. Measurement values are not unique and are not ordered. This implies that values in a given range have a great probability to be contained in a large number of files. Therefore, the file selection (files that contains at least one element necessary for processing) can be bigger than other attributes such as *meter_id*. This in turn, reduces the gains. Within iBig, we can make Drill (that is already fast) more efficient with 12 and 16 nodes by adding more nodes to the indexing component hosted on MongoDB.

For Bill queries Query6 and Query7 shown in Figure 4, the gains are huge with Spark mainly because the selected data after indexing are very small compared to the entire dataset. Furthermore, Bill queries are scheduled as two spark jobs executed sequentially. This can be very costly when scanning the entire dataset. In turn, with indexing, the gains are multiplied since only the small selected data are scanned and processed. It is noteworthy also, to observe that index time is quite small for Query6 (shown in Figures 4a and 4b) since the query computes the bill for only one given client indexed on the *meter_id* attribute. In contrast, the gains with Drill are less important, in particular when the number of nodes in the cluster increases. For instance, the gains are 41% and 43% when nodes number is 4 for Query6 and Query7, whereas only 11% gains are observed with Query6. Indexing becomes inefficient with Query7 beyond 16 nodes. The exhibited behavior of these queries is because Drill scales out well in this setup with the entire data, in particular, with the current distribution of the dataset across HDFS datanodes but not with the filtered data. Bill queries are CPU-intensive and require minimal data transfer. Drill however processes data locally, and therefore will use CPUs only in the nodes that contains data. This results of load unbalancing with filtered data that are not usually uniformly distributed across the cluster depending on filtering conditions.

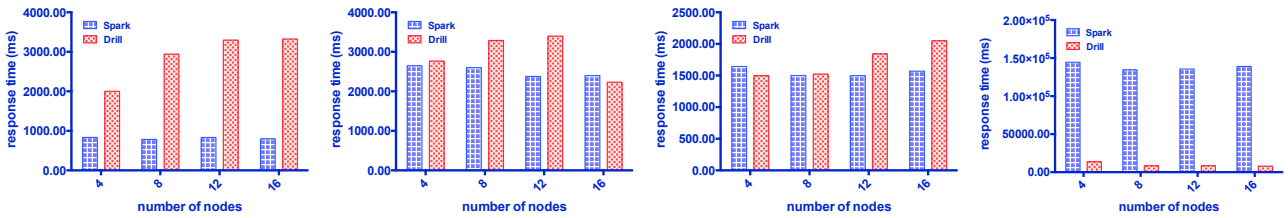
B. *Indexed Spark vs. Indexed Drill*

In Figure 5, we focus on the response time exhibited by both Spark and Drill and this only for the set of files that contain relevant data for the respective processing (i.e. after index-based filtering but without accounting for index scan). Compared to the result demonstrated in the previous Section, the first observation is that spark response time is close to that of Drill and in many times even faster. This is due to minimizing the impact of scanning the whole data and partitioning them in-memory. For aggregation queries Query1, Query2, and Query3 (Figures 5a, 5b, and 5c respectively), Spark outperforms Drill in most cases, but not by much. For instance, Spark is 76% faster with Query1 when the number of nodes is 16, but only 1% faster with Query3 for 8 nodes-sized cluster and even 6 % slower with Query2 for 16 nodes-sized cluster. Two reasons cause this behavior: First, Drill relies on cost-based optimization, which can take hundreds to thousands milliseconds. With such fast response time with filtered data, the overhead introduces noticeable overweight. Secondly, unlike Spark that partitions data before processing to achieve load balancing, Drill is locality-aware and processes data on the node it stores it. Subsequently, this can unbalance the processing load between nodes because filtered data at input is located on a subset of nodes (since data are ordered and partitioned into files on the *meter_id* attribute in this case). This latter reason explains as well why Drill does not scale out with filtered data as it did with the entire dataset. In fact, increasing the number of Drill processors increases the coordination overhead while the added processing power is not really needed. The most interesting results however, are demonstrated by the selection and filtering queries with a sort clause (Query4 and Query5 shown in Figures 5d and 5e respectively), which are queries that incur heavy data transfer, and the Bill queries (Query6 and Query7 shown in Figures 5f and 5g respectively), which are CPU-intensive queries with very little need for data transfer. The results clearly show that spark excels with the CPU-intensive Bill computation whereas Drill is similarly suited for data transfer-heavy processing. Drill is up to 90% faster than Spark with Query4 and up to 76% faster with Query5. In contrast, Spark can outperform Drill by 98% with Query6 and 84% with Query7. Spark is more suited to CPU-intensive queries because it partitions data in memory across the cluster nodes. This would allow better load balancing, which in turn exploits the computing power in

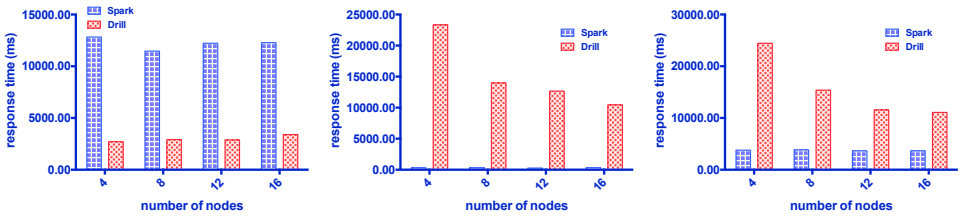


(a) Query6 Spark (b) Query6 Drill (c) Query7 Spark (d) Query7 Drill

Fig. 4: Gains with Bill Queries



(a) Query 1 (b) Query 2 (c) Query 3 (d) Query 4



(e) Query 5 (f) Query 6 (g) Query 7

Fig. 5: indexed Spark Vs indexed Drill

the cluster to its fullest, but at the cost of data transfer, thus exposing data processing to the network latency. In contrast, Drill, with its MPP model, favors locality-aware processing where computations are sent to data, rather than the opposite. When data does not span across the entire cluster however, computing power is under used, which can prove costly with CPU-intensive processing. We argue that a lot of improvement can be achieved in this area to provide the most relevant model according to the nature of processing and queries.

VI. Related Work

Although indexing structures have been around for many years, in particular to speed up data access in relational database systems, it is only recently they started to be integrated in Big Data Ecosystems. Hadoop vendors have integrated search services within their Hadoop distributions mostly to search and index text. Their solutions are based on either Elasticsearch [18] or Solr [19]. Additional file text and split indexing approaches for Hadoop are proposed in [20] and [21]. In [22], HadoopDB a hybrid architecture that consists of Hive with its SQL-like query execution engine over Hadoop MapReduce and Relational DBMS was introduced. Although indexing is supported within RDBMS, It is used for data within relational tables. In [23], the authors propose a hybrid architecture that consists of an RDBMS in addition to Hadoop. They use split indexes per file in HDFS, which might be costly in terms of scan, thus impacting the efficiency of indexing, in particular when the number of processors is large. The architecture is also reliant on the job tracker of the former version of Hadoop MapReduce and is not implemented with Spark. In [24] and [25], indexing approaches are proposed for IoT sensors data. The work in [24], provides novel indexes for key/value stores in the cloud while the work in [25] proposes an index structure for Apache Hive. The indexing efficiency of most of the aforementioned solutions is limited and hardly discussed. In fact, the amount of information stored for a given index (split indexes) is not negligible. Therefore, index scan can result in important delays for large datasets, which jeopardizes the overall efficiency. Additionally, in these studies horizontal scalability is not considered although it is a very important aspect of Big Data processing. In a different direction, studies in [26] and [15], demonstrate the impact of different underlying storage systems, that adopt different approaches to storing, partitioning, and searching data, on Big Data processing.

Many efforts were dedicated to compare MapReduce against parallel database management systems (PDBMS) that adopt MPP type of processing [27], [28], [29]. However, those efforts

focused mostly on early MapReduce implementations and can be considered as unfair since in MapReduce systems, data storage and data processing are separated, unlike PDBMS. Henceforth, more often than not it is storage optimization that favors PDBMS. In our study however, MapReduce and MPP frameworks are evaluated in the same ecosystem with the same underlying storage architecture, which makes them on par for a fair comparison.

VII. Conclusion

Today's Energy IoT platforms rely on Big Data ecosystems to manage and process large datasets of multiple origins. These ecosystems scale very well to large volumes but rely on blind data search and scan from underlying file systems. As a result, all elements in a given dataset are scanned no matter how useful they are. This can be very costly when scanning irrelevant data for selective data processing. In this work, we have introduced the new *iBig* hybrid Big Data architecture to provide attribute indexing capabilities to IoT Big Data processing through an additional component in the ecosystem. We have demonstrated, through our experimental evaluation on the Grid5000 cloud and grid testbed, the potential of indexing in our architecture where performance gains for Energy IoT data processing can exceed 98% for few of the selective queries. In addition, in this paper, we have provided an extensive comparison between MapReduce and MPP models illustrated by Spark and Drill respectively. Unlike many related work that attempted this comparison in the past, we have relied on fair comparison conditions where both processing models rely on the same storage layout and environment. In a future work, we plan to enrich the indexing component with a wider selection of indexing structures and other kind of datastores and data processing models.

Acknowledgment

This work has been funded through the Industrial Research Chair Enedis (*) in Smart Grids

References

[1] "Gartner says 8.4 billion connected "things" will be in use in 2017, up 31 percent from 2016," April 2017. [Online]. Available: http://www.gartner.com/newsroom/id/3598917

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

[3] 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.

[4] "Apache hadoop," April 2017. [Online]. Available: http://hadoop.apache.org/

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

[6] "Apache flink," March 2016. [Online]. Available: http://flink.apache.org/

[7] "Apache drill," April 2017. [Online]. Available: https://drill.apache.org

(*) Enedis is a french public-service company, managing the electricity-distribution grid. It develops, operates and modernizes the electricity grid and manages the associated data. It performs customer connections, 24-hour emergency interventions, meter reading and all technical interventions. It is independent from the energy providers, which are responsible for the sale of electricity and the management of the supply contract.

[8] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, "Dremel: Interactive analysis of web-scale datasets," in *Proc. of the 36th Int'l Conf on Very Large Data Bases*, 2010.

[9] "Presto: Distributed sql query engine for big data," April 2017. [Online]. Available: https://prestodb.io

[10] R. Bayer and E. M. McCreight, "Organization and maintenance of large ordered indexes," *Acta Informatica*, vol. 1, no. 3, pp. 173–189, 1972.

[11] "Mongodb," April 2017. [Online]. Available: http://www.mongodb.org/

[12] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10, 2010.

[13] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark sql: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15, 2015.

[14] X. Liu, L. Golab, W. Golab, and I. F. Ilyas, "Benchmarking smart meter data analytics," in *EDBT: 18th International Conference on Extending Database Technology, March 2015, Online Proceedings*.

[15] H. Chihoub and C. Collet, "A scalability comparison study of data management approaches for smart metering systems," in *2016 45th International Conference on Parallel Processing (ICPP)*, Aug 2016.

[16] "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

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

[18] "Elasticsearch," April 2017. [Online]. Available: https://www.elastic.co

[19] "Apache solr," April 2017. [Online]. Available: http://lucene.apache.org/solr/

[20] N. Li, J. Rao, E. Shekita, and S. Tata, "Leveraging a scalable row store to build a distributed text index," in *Proceedings of the First International Workshop on Cloud Data Management*, ser. CloudDB '09, 2009.

[21] M. Y. Eltabakh, F. Özcan, Y. Sismanis, P. J. Haas, H. Pirahesh, and J. Vondrak, "Eagle-eyed elephant: Split-oriented indexing in hadoop," in *Proceedings of the 16th International Conference on Extending Database Technology*, ser. EDBT '13, 2013.

[22] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin, "Hadoopdb: An architectural hybrid of mapreduce and dbms technologies for analytical workloads," *Proc. VLDB Endow.*, vol. 2, no. 1, Aug. 2009.

[23] V. R. Gankidi, N. Teletia, J. M. Patel, A. Halverson, and D. J. DeWitt, "Indexing hdfs data in pdw: Splitting the data from the index," *Proc. VLDB Endow.*, Aug. 2014.

[24] T. Guo, T. G. Papaioannou, and K. Aberer, "Efficient indexing and query processing of model-view sensor data in the cloud," *Big Data Res.*, Aug. 2014.

[25] Y. Liu, W. Liu, Z. Xiao, W. Qiu, Y. Li, and Y. Liang, "Dgfindex: A hive multidimensional range index for smart meter big data," in *Proceedings Demo & Poster Track of ACM/IFIP/USENIX International Middleware Conference*, ser. MiddlewareDPT '13, 2013.

[26] K. Doan, A. O. Oloso, K. S. Kuo, T. L. Clune, H. Yu, B. Nelson, and J. Zhang, "Evaluating the impact of data placement to spark and scidb with an earth science use case," in *2016 IEEE International Conference on Big Data (Big Data)*, Dec 2016.

[27] M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin, "Mapreduce and parallel dbmss: Friends or foes?" *Commun. ACM*, 2010.

[28] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, "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.

[29] 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.