Cluster-size optimization within a cloud-based ETL framework for Big Data

Conference Paper · December 2019

DOI: 10.1109/BigData47090.2019.9006547

Eftim Zdravevski
Ss. Cyril and Methodius University in Skopje
157 PUBLICATIONS **1,432** CITATIONS
SEE PROFILE

Petre Lameski
Ss. Cyril and Methodius University in Skopje
102 PUBLICATIONS **929** CITATIONS
SEE PROFILE

Ace Dimitrievski
Ss. Cyril and Methodius University in Skopje
19 PUBLICATIONS **132** CITATIONS
SEE PROFILE

Marek Grzegorowski
University of Warsaw
18 PUBLICATIONS **148** CITATIONS
SEE PROFILE

Cluster-size optimization within a cloud-based ETL framework for Big Data

Eftim Zdravevski, Petre Lameski, Ace Dimitrievski
Faculty of Computer Science and Engineering, Ss Cyril and Methodius University, Skopje, North Macedonia
Email: eftim.zdravevski@finki.ukim.mk, petre.lameski@finki.ukim.mk ace.dimitrievski@gmail.com

Marek Grzegorowski
Institute of Informatics, University of Warsaw, Poland,
m.grzegorowski@mimuw.edu.pl

Cas Apanowicz
CogniTrek Corp., Toronto, Canada,
Email: cas.apanowicz@cognitrek.com

Abstract—The ability to analyze the available data is a valuable asset for any successful business, especially when the analysis yields meaningful knowledge. One of the key processes for acquiring such ability is the Extract-Transform-Load (ETL) process. For Big Data, ETL requires a significant effort and it is a very challenging task to be performed in a cost-effective way. There are quite a few examples in the literature that describe an architecture for cost-effective ETL but none of the available examples are complete enough and they are usually evaluated in narrow problem domains. The ones that are more general, require specific implementation details. In this paper we propose a cloud-based ETL framework where we use a general cluster-size optimization algorithm, while providing implementation details, and is able to perform the required job within a predefined, and thus known, time. We evaluated the algorithm by executing three scenarios regarding data aggregation during ETL: (i) ETL with no aggregation; (ii) aggregation based on predefined columns or time intervals; and (iii) aggregation within single user sessions spanning over arbitrary time intervals. The execution of the three ETL scenarios in a production setting showed that the cluster size could be optimized so it can process the required data volume within a predefined and thus, expected, latency. The scalability was evaluated on Amazon AWS Hadoop clusters by processing user logs collected with Kinesis streams with datasets ranging from 30 GB to 2.6 TB.

Index Terms—Data streams; ETL; Business analytics; Hadoop; Spark; Cluster size optimization

I. Introduction

The ubiquitous smart devices, sensors and social media result in sheer data volumes [1]. At the same time, consumers became accustomed to personalized services that are available instantaneously, while many companies, health providers and institutions have focused tremendous resources on providing this. In the past, they could decide which data to store by making compromises between available resources and capabilities to manage data.

In the era of Big Data, companies experience a growing pressure to store and analyze all data that is being collected [2] to stay competitive in the data-driven marketplace. Volume, velocity and variety are intrinsic properties of Big Data [2, 3]. Recently, variability, veracity, visualization, and value were identified as similarly significant [4]. Together, they define the 7 Vs of Big Data reflecting the enormous complexity presented to those who would process, analyze and benefit from it. Another issue the companies face is that the results of data analysis (e.g., joins, transformations and aggregations) or integration with other parts of the system (e.g., projections and models created by machine learning algorithms) further boost data generation and increase data volume [5].

In order to make the data available in a usable format, several steps need to be performed [6, 7]: analyses and modeling to identify all relationships and business context; data collection; and Extract-Transform-Load (ETL), which is usually time-consuming in terms of both development and execution time. Once the data is processed and loaded into a data warehouse, it needs to be available for reporting, visualization, analytics and decision support [2]. The typical applications of the state-of-the-art machine learning techniques also require proper data preprocessing, including the costful process of representation learning or feature extraction [8]. Cost-effective approach to scale this process for Big Data [9] is crucial for the efficiency of applications in many domains [10].

As identified in [11], there are various challenges for data warehousing (DW) and business intelligence (BI) over Big Data: data modeling; data consistency and lineage; extending applicability of traditional tools for data exploration, visualization and analytics; and integration with traditional DW/BI solutions and platforms. The usability of data at any point in time amid various processing stages relates to data consistency. Ensuring it is considerably challenging in Big Data systems. Strong consistency models introduce severe limitations to the system's scalability and performance. On the other hand, weak and eventual consistency models facilitate high levels of availability and lower latencies, but could significantly impair the value of obtained information and reduce its usability [12].

While cloud computing has emerged as an important paradigm offering a variety of low-cost hardware and software, which is particularly convenient for deploying Big Data systems, it also raises new challenges related to architecture, cost and performance optimization, providing reliability, guaranteeing security and ensuring privacy and data consistency [12]. Big Data exacerbates these concerns because of the distributed architectures, which require more advanced mechanisms for synchronization, replication, scheduling and security [3].

Even though there are technologies for efficient ETL and analytics of Big Data, there is no comprehensive cloud-based architecture offering an integrated, scalable and cost-effective solution. Most approaches are either for specific purposes in a narrow domain or only provide general definitions and lack experimental evaluation. These approaches neglect real-world development and deployment challenges [4].

We evaluated three ETL scenarios concerning data aggregation commonly encountered in real-world systems with a combination of traditional tools – for processing dimensional data, and Spark executed on on-demand clusters – for processing high-volume transactional data. We propose a Hadoop extension, inspired by the edge computing paradigm, to further process the Spark output and fully prepare it for the data warehouse. To validate the scalability of the architecture, we performed experiments on Amazon AWS clusters with datasets ranging from 30 GB to 2.6 TB. To minimize cloud costs, we also proposed and validated an algorithm for cluster-size optimization considering data volume, expected latency and historic execution metrics The evaluation of each step of the three ETL scenarios showed that the cluster size could be optimized so it can process the required data volume within the expected time.

Most importantly, the whole process is integrated from end-to-end and evaluated in a production environment on real high-velocity streaming Big Data, something that lacks in most related approaches. Production environment refers to the setting where one puts software and other products into operation for their intended end users.

II. Related work

In classical BI, the ETL process loads data into warehouse servers [13]. For reasonable data volumes there are ETL tools that were successfully used in organizations throughout the years, such as Informatica, IBM Infosphere Datastage, Ab Intio, Microsoft SQL Server Integration Services (SSIS), Oracle Data Integrator, Talend, Pentaho Data Integration Platform (PDI), etc. Recently, Enterprise Application Integration (EAI) systems are inheriting ETL tools and now perform considerably more functionalities than just ETL. In cases when the business requirements vary often, database vendors are also promoting another set of tools called "Extract-Load-Transform" (ELT), which postpone the transformations to a later time [14]. The terminology and capabilities of ETL and ELT tools are described in [15], but without much regard to the challenges encountered with Big Data.

Apache Spark was inspired by the MapReduce concept and focuses on the class of applications that reuse a working set of data across multiple parallel operations. Prime example of such applications are the iterative machine learning algorithms and interactive data analysis tools. Resilient Distributed Datasets (RDD) are the basic abstraction in Spark and represent immutable, partitioned collections of elements that can be operated on in-memory and in parallel on different nodes in the cluster. MLLIB[16] is an example of an open-source distributed machine learning library that utilizes Spark. The parallelism of MapReduce and Spark allows all ETL processes to be distributed across multiple nodes and all transformations to be performed on distinct portions of data [17, 18].

Big Data ETL processes rely on distributed storage for efficient writing and reading. The Hadoop Distributed File System (HDFS) is similar to Google's distributed file system. HBase is database built on top of HDFS and there are several designs for efficient storage of high-volume data, including [19].

Traditional ETL deployments do not consider data partitioning [20], which is an essential aspect in ETL and MapReduce processes. This work performed MapReduce partitioning experiments on static datasets, which neglects the task of matching of dimensions from fact tables.

After we load the data into a high-performance data warehouse, such as Hive [21], HBase Amazon Redshift [22], SAP HANA [23] or Infobright [24], we can execute ad-hoc queries. It is important to note that execution of queries against databases like Hive and HBase commonly relies on MapReduce, Spark or Tez [25] jobs. Furthermore, the synchronous dataflow execution model of MapReduce and Spark limits the use of asynchrony for complex analytics. Approaches, such as [26], attempt to bridge this gap by proposing asynchronous architectures.

A framework that process historical and incoming data separately is proposed in [27]. It uses a dynamic mirror replication technology to avoid the contention between OLAP queries and OLTP updates. The main limitation of [27] is the evaluation, which only considers a static dataset of 16 GB.

There are quite a few examples of cluster size optimization for Big Data analytics. Authors in [28] propose a query like environment where developers can query for the required cluster size. The proposed approach requires implementation specific details. Another approach is presented in [29]. This work focuses on short jobs optimization. Our proposed cluster-size optimization algorithm works without the need of implementation details. Furthermore, our architecture facilitates data processing in different ETL scenarios, making it more versatile and applicable for automated feature extraction.

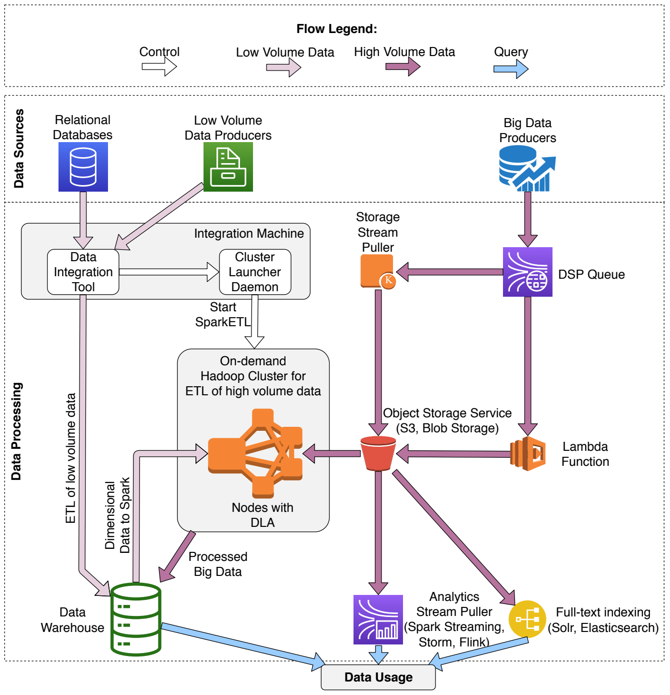

Fig. 1: Architecture of scalable Cost-optimizing cloud-based Big Data Warehouse.

III. Architecture of the framework for ETL of Big Data

Figure 1 shows the architecture of the proposed system. In organizations, commonly, there are traditional data sources, such as relational database management systems (RDBMS) and structured and semi-structured data from internal or third-party data providers, that generate reasonably-sized data. Companies usually process this kind of data using traditional Data Integration Tools. In our experiments, we utilized the Pentaho Data Integration Platform (PDI) for such ETL tasks, which process incoming low-volume data and store it in the Data Warehouse (marked with light gray arrows in Figure 1).

If there are data producers that generate Big Data with high volume, velocity or versatility, then the classical approach for ETL is not suitable. The Big Data streams can be efficiently collected and processed by Distributed Streaming Platforms (DSP), which are scalable, replicated and fault-tolerant (e.g., Apache Kafka, Amazon Kinesis, etc.) [30].

By defining a retention policy, we can configure DSPs, to retain data on the queue for a specific time after it was published, regardless if it was consumed or not. For example, for Amazon Kinesis, the maximum data retention period is one week at the time of this writing. DSPs allow the same data stream to be consumed by multiple consumers independently and simultaneously, each of them working at their own pace. Accessing data on a DSP queue can be performed by either push or pull mechanisms [31]. The pull mechanism is innate for Amazon Kinesis and Apache Kafka, so each consumer has and manages its read pointer.

The proposed architecture allows consumption of DSP queues by the three most common and widely used types of consumers. First, with Push Lambda Functions (Stream-based model), event sources publish events on DSP, which trigger the lambda function multiple times per second as data arrives on the queue, and the lambda function processes the events [32, 33]. Lambda functions are subscribed to automatically read batches of records of the DSP queue and process them if they are detected on the stream. They poll the queue periodically (up to few times per second) for new records. The lambda function should be rarely updated because it stores unprocessed raw data in the original format on Object Storage Services (OSS) (e.g., Amazon S3 or Windows Azure Blob Storage). However, if it requires updating, that usually forces some minimal downtime. The most common reason for updating the function is to improve error handling because of unparsable or illegal input provided by the event sources.

Next, Storage Stream Pullers as consumers have more control in fetching records from DSPs because they manage their read pointer independently, and therefore, can reprocess events if needed (e.g., for recovering after failures). However, their management is more involved because the uptime of the machine that executes the storage stream pullers need to be guaranteed by manual mechanisms, as Amazon AWS does not guarantee it. This is an alternative for durable and persistent storage of the raw data with a variable frequency of persisting data [32].

The first two types of consumers are redundant alternatives for reliable and permanent storage of the incoming data in raw format on different S3 buckets. Each of these alternatives is very reliable with guaranteed Service Level Agreement (SLA) and either one of them is sufficient for the proposed architecture. Using both alternatives can simplify deployment procedures and further improve the reliability of the system. Namely, if the data format changes drastically, or sources vary, the consumers can be updated without any downtime or risk of losing data. Having both alternatives also provides integration convenience with the existing infrastructure in organizations.

Third, Analytics Stream Pullers, such as Spark Streaming and Apache Storm, are typical consumers that perform stream processing and provide near real-time analytics [32]. Very complex algorithms can be implemented with these technologies to provide valuable business insights and near real-time analytics, and can also store data in the warehouse. Be that as it may, Analytics Stream Pullers can execute algorithms that use only recent data because of data retention policies.

To complement this, the proposed architecture employs on-demand Spark clusters for implementing more complex algorithms for ETL and feature engineering [34]. They can analyze dynamic trends over longer time periods (e.g. week-by-week or month-by-month comparisons of various metrics) or find the time since some particular event happened (e.g., “the time since the last occurrence of event X”, “the time since the user’s last login”, “last bought or viewed product”, “last used service”, etc.). Such metrics are not computable with Analytics Stream Pullers.

The Cluster Launcher module, located on the same instance that hosts the Data Integration Tool (DIT), facilitates the starting of on-demand Spark clusters. It can be invoked manually or based on a predefined schedule by DIT. The Cluster Launcher can start a Hadoop cluster with configurable size and can run a particular Spark job on it. After one starts the Spark cluster, it downloads the source code from a release branch of a code repository (i.e., git, mercurial, subversion or even a location on an FTP server, S3 or Azure Blob Storage) and automatically starts it. Code development and management adhere to the adopted strategy of the organization (e.g., GitFlow), which defines rules and best practices for conflict resolution, peer-review, merging to staging and production branches, etc. Each Spark cluster during its lifetime executes only a specific ETL job. If the organization requires multiple ETL processes of unrelated data, then multiple Spark jobs can be defined, and for each of them a separate workflow can be managed (i.e., separate code repositories, execution schedules, destination data warehouses, etc.).

Spark applications process Big Data stored on OSS, while also considering the dimensional data from the data warehouse. Generally, the dimensional data does not have to be processed by Spark because usually, it is with much smaller volumes compared to the transactional data; therefore, we can use traditional ETL tools can for it. However, while processing the transactional data, which we would store in fact tables, the dimensional data is still required. If the fact tables are wide, then the dimensional data is a prerequisite for the denormalization. Otherwise, we need it for setting up foreign keys to the dimensions.

In addition to the ETL process, Spark can also look in past data if required, perform complex aggregations and run machine learning algorithms. Spark outputs the results to HDFS, but some data can be stored additionally on OSS, depending on the requirements (e.g., for archival purposes). Another reason why we choose Spark is because of its ability to utilize all available resources on the cluster dynamically. This ability allows adding new nodes on the fly, so a running Spark job can recognize all resources and utilize them without restarting.

Next, we execute the Distributed Load Agent (DLA) on different nodes of the Hadoop cluster, processing distinct portions of HDFS data generated by Spark. After the DLA processes complete, data is available in the warehouse for various BI reporting and analytics tools, as well as for external data mining or machine learning algorithms.

The proposed architecture makes it possible to handle high-velocity Big Data thanks to its three components. First, DSP apply sharding techniques allowing processing of thousands of events per second. For instance, each Kinesis shard can process up to 1000 writes per second, and we could add new shards without any downtime. Second, after the data is on the DSP stream, it is reliably retained up to a predefined period (e.g., up to a week on Kinesis). During that time a Stream Puller application or an automatically triggered Lambda Function will store it in OSS. For instance, up to 1000 concurrent executions of lambda functions on AWS are allowed, with each execution being able to process thousands of records and store them to S3. Even if this is not enough and some throttling is activated, the data is still retained on the stream and would be processed automatically by the lambda function after the system handles the peak load. The stream puller applications need more manual management in terms of maintaining checkpoints of last successful read, frequency of data pulling, deciding how much data can be processed per unit time, etc.

Nevertheless, they can still accomplish a reliable stream data storing process. Therefore, the high-velocity data is transformed into static data stored in OSS. Third, OSS is reliable and persistent storage by design. Objects on OSS are usually labeled with a hierarchical prefix describing the date and time of their collection, making it straightforward to obtain the source path of the new data for a particular ETL run. A common pattern that facilitates this on AWS S3 is s3://bucket/collection/folder/year/month/day/hour/minute/. Moreover, OSS provides information about the total size of all objects with a common prefix (analogous to folder size calculation); thus, the size of the data is known before the ETL starts. Consequently, using the mechanisms proposed further in Section IV, the cluster size can be approximately set.

IV. Algorithm for cluster size cost-optimization

The proposed system utilizes on-demand clusters for processing Big Data. Therefore, it is important to optimize the cluster-size, considering the volume of data that needs to be processed, as well as the allowed latency.

A. Apriori cluster size

The apriori algorithm for cost-optimization considers two parameters: the size of data that needs to be processed in one run, and the maximum time in which this data needs to be fully processed and loaded into the warehouse. The main output of the algorithm is the number of nodes in the cluster, while also providing auditing information for the duration of the distinct steps of the ETL process and the associated cost, depending on the cluster size. The algorithm consists of two phases: one, which identifies a cluster size that can perform the ETL within the required time; and a second phase, which attempts to downsize the cluster, so it still performs the ETL within the required time but uses fewer resources.

In Listing 1, some global constants and variables are defined, as well as the function that performs the ETL steps. Line 1 in Listing 1 defines constants used later in the other functions. The global list info, which contains tuples representing various parameters computed upon completion of the ETL process. The type of node instance offered by the cloud provider is denoted by inst_type, and the unit_time constant denotes the smallest billing interval in seconds.

The last constant is node granularity (i.e., node_gran), which bounds the error during optimization of cluster size. So, if the globally best number of nodes is nodes_best, and the estimated optimal number of nodes by the algorithm is nodes_optimal, then the algorithm guarantees that nodes_optimal - nodes_best < node_gran and nodes_optimal >= nodes_best stand. If the globally best number of nodes needs to be determined, then node_gran should be set to one. However, in enterprise applications, this is not required because the cluster needs to be able to cope with a sudden increase in data volume or node failure without incurring delays in the processing time. Therefore, lines 9 to 10 in Listing 1 define the function round_up, which rounds up the number of nodes to the nearest multitude of node_gran. So, if unit_time is set to 5, then all cluster sizes determined by the algorithm will be a multitude of 5 (e.g., 5, 10, 15, etc.).

Function perform_ETL (lines 4 to 14 in Listing 1) has two input parameters: the number of nodes in the cluster (i.e. nodes) and the maximum time allowed for the ETL process to finish (i.e. max_time_allowed). This function first starts a new cluster (line 5) and then executes the steps of the ETL process: running the Spark job (line 6) that performs parsing, data cleaning, transformations and various aggregations if needed (Extract and Transform steps); and running distributed data load in the warehouse (line 7), which is the Load step. This function also terminates the cluster after everything finishes (line 8), computes the execution time of each of these steps, the total estimated cost and eventually stores them in the info list (lines 9 to 13). Clusters for which the value of ratio is less than one, are considered as valid because the total execution time of the ETL process is shorter than the maximum allowed time (i.e., max_time_allowed). In contrast, invalid clusters are not able to complete the ETL process within the required time.

Next, Listing 2 shows the Python code of three auxiliary functions. First, the function clusters_valid_min (lines 1 to 3 in Listing 2) returns the number of nodes of the smallest valid cluster. Similarly, the function clusters_invalid_max (lines 4 to 7 in Listing 2) returns the number of nodes of the largest invalid cluster or None if there is no invalid cluster. Subsequently, function cluster_optimal (lines 8 to 11 in Listing 2) returns the number of nodes of the optimal cluster. If multiple clusters can perform the ETL in the required time, then the one with the lowest cost and shortest time is considered as the optimal cluster. Accordingly, if two clusters cost the same, the one that performed the ETL process more quickly is preferred.

Finally, function optimize_cluster, shown in Listing 3, defines the actual algorithm for cluster size optimization. It accepts two input parameters: the expected volume of data that needs to be processed (i.e., data_size) and the maximum allowed time (i.e., max_time_allowed) in which the ETL process has to be completed. The first parameter can be estimated by investigating the highest data volumes that need to be processed. The second one depends on the business rules and relates to the time after which the processed data needs to be available for applications such as reporting, decision making, machine learning, etc. The variable nodes (line 2 in Listing 3) is set to a default number of nodes, which is heuristically computed depending on the data_size parameter and the available RAM on the used instance type. The rationale is that all data needs to fit in the RAM for Spark to be efficient. However, there is some overhead (e.g., operating system occupies some of the RAM, replicated data across nodes, type conversions, etc.) and not all RAM is available for Spark. Therefore, the formula of the heuristic attempts to address this. Even if the default value is very unsuitable (i.e., is either too high or too low), it will be tuned by the remainder of the algorithm. The variable nodes_tested (line 2 in Listing 3) keeps track of the number of nodes of evaluated clusters.

Subsequently, the first loop (lines 3 to 9 in Listing 3) performs scaling up until the cluster size is enough to execute the ETL process in the expected time (i.e. ratio < 1). Because the ratio parameter represents the number of times the max_time_allowed parameter is contained in the total time needed for the ETL process, it is used as a reasonable multiplier for scaling up the cluster (line 7 in Listing 3). Line 5 checks if the cluster is inefficient, meaning that it executes longer than unit time (one hour in this case) and that in the last period it is active less than two thirds (2400 out of 3600 seconds in this case). This heuristic forces clusters to utilize the most of the time units for which they are billed by scaling the cluster up, even if the ETL was performed in the allowed time (line 8 in Listing 3).

The following loop (lines 10 to 16 in Listing 3) performs scaling down, striving to find a smaller cluster that can still execute the ETL process in the required time. Based on the execution times collected during the previous executions of the ETL process, the nodes of the largest invalid cluster and the smallest valid cluster are obtained (line 11 in Listing 3). Next, the number of nodes of a smaller cluster is calculated. If the default cluster size was enough to perform the ETL in the required time, then there would be no invalid clusters tested so far. Therefore the next cluster to be tested will be twice as smaller (line 12 in Listing 3). Otherwise, the next cluster size would be the arithmetic mean of the nodes of the largest invalid and the smallest valid cluster max_time_allowed (line 13 in Listing 3). If the difference between the nodes of the smallest valid cluster and the next cluster to be tested is less than node_gran, then the optimal cluster size is returned (line 14 in Listing 3). If not, the ETL is performed with the smaller cluster, and another iteration starts.

```
info, inst_type, unit_time, node_gran = [], 'r3.2xlarge', 3600, 5
def round_up(x):
return ceil(x / node_gran) * node_gran
def perform_ETL(nodes, max_time_allowed):
t0, cluster = time(), start_cluster(nodes, inst_type)
t1, _ = time(), run_Spark(cluster) # Extract and Transform steps
t2, _ = time(), run_DDL(cluster) # Load step
t3, _ = time(), terminate_cluster_async(cluster)
t_boot, t_spark, t_DDL, t_total, t_charged = t1 - t0, t2 - t1,
        t3 - t2, t3 - t0, t3 - t1
t_units = ceil(t_charged / unit_time)
cost = t_units * nodes * get_cost(inst_type)
ratio = t_total / max_time_allowed
info.append((cost, t_total, nodes, ratio, t_charged, t_spark,
        t_DDL))
return ratio, t_total
```

Listing 1: Definition of global constants and variables and Python function for performing the steps of the ETL process

```
def cluster_valid_min():
valid = [nodes for (_, _, nodes, ratio, _, _, _) in info if ratio
        < 1]
return min(valid)
def cluster_invalid_max():
invalid = [nodes for (_, _, nodes, ratio, _, _, _) in info if
        ratio >= 1]
if len(invalid) > 0: return max(invalid)
else: return None
def cluster_optimal():
info.sort()
valid = [nodes for (cost, t_total, nodes, ratio, _, _, _) in info
        if ratio < 1]
return valid[0]
```

Listing 2: Auxiliary Python functions used by the algorithm for cluster size cost-optimization

```
1   def optimize_cluster(data_size, max_time_allowed):
2   nodes, nodes_tested = cluster_default(data_size), set()
3   while True: # scaling up
4   ratio, t_total = perform_ETL(nodes, max_time_allowed)
5    inefficient = t_total > unit_time and t_total % unit_time < (2
            / 3) * unit_time
6   nodes_tested.add(nodes)
7    if ratio > 1: nodes = round_up(nodes * ratio)
8    elif inefficient : nodes = round_up(nodes * (1 + (t_total %
            unit_time) / unit_time))
9    else: break
10  while True: # scaling down
11  nodes_invalid_max, nodes_valid_min = cluster_invalid_max(),
            cluster_valid_min()
12   if nodes_invalid_max is None: nodes =
            round_up(nodes_valid_min / 2)
13   else: nodes = round_up((nodes_valid_min +
            nodes_invalid_max) / 2)
14   if nodes_valid_min - nodes < node_gran or nodes in
            nodes_tested: return cluster_optimal()
15  perform_ETL(nodes, max_time_allowed)
16  nodes_tested.add(nodes)
```

Listing 3: Python function implementing the algorithm for cluster size cost-optimization

B. Dynamic adjustment of cluster size

The main limitation of the apriori algorithm for cluster size optimization is that the data size needs to be known before the actual execution of the ETL. The collected execution times, cluster size and data size, during the initial runs required for optimization, as well as the subsequent production runs, can allow dynamic adjustment of cluster size. Namely, we can represent the execution time (t) as a function (F) of cluster size (c) and data size (d), i.e $t = F(c, d)$. The function F is not known, but with enough samples (c, d, t), it can be approximated. Likewise, we can define a function G, which estimates the cluster size, depending on the required execution time and current data size, i.e., $c = G(d, t)$. Again, having information about previous ETL executions in the form of (c, d, t) samples, we can use bilinear interpolation to estimate the appropriate cluster size before the cluster is launched. This model allows us to dynamically set the cluster size depending on data size, without the requirement of frequent re-running the cluster size optimization algorithm described in the previous subsection.

Owing to data distribution skew, in some cases, some steps of the ETL could take longer than during other runs with similar data size and equal cluster size. Therefore, the algorithm can be further enhanced by an option of adding more nodes while a Spark job is executing. This requires comparing the execution time of each consecutive Extract and Transform step to previous executions while considering the data size and cluster size. If there is a significant deviation from the expected duration of a particular step, then new nodes can be added. Herein, several things should be considered when performing this kind of addition to a live cluster. First, the provisioning and booting time of a new node (i.e., 10-15 minutes) needs to be taken into account. The dynamics of the system need to be considered too. By the time the new nodes are available, the current Spark job continues with the processing and may have already scheduled and started all of its tasks at a given step. In such a case, the new nodes will be idle, waiting for the current step to finish so that they could be utilized in the next step. Detailed analysis of such dynamic behavior is out of the scope of this work, although it is undoubtedly one of our future research goals. Scaling-down the clusters while executing Spark jobs, is not recommended for two reasons. First, decommissioning instances while Spark is using them would imply that some tasks can fail, causing Spark to redistribute them to other nodes, thus prolonging the execution of the job. Moreover, the cloud provider would charge the terminated instances regardless due to the hourly pricing model of AWS, so choosing the right moment in time to terminate them without incurring costs for the next hour is another challenge.

V. Results

A. Cluster hardware

Contributing to the popularity in industry and research community of the Amazon Web Services (AWS), and the hardware heterogeneity offered in various instance types

[32], it was used for the experimental evaluation of the proposed architecture. From all available instance families, only the "R3" and "R4" are optimized for memory-intensive applications and offer the best price per GB of RAM https://aws.amazon.com/ec2/instance-types/. "R4" does not have local storage, thus require separate instantiation of EBS (Elastic Block Store) volumes, so to simplify the provisioning of hardware, we decided to use the "R3" instances. The smaller instance types from the "R3" family were not appropriate for Spark: "r3.large" has only two cores, and "r3.xlarge" offers lower network performance compared to the larger instance types. So, we used the "r3.2xlarge" instance type for our experiments because it has high network performance while offering better granularity in controlling the cluster resources compared to larger instances. These instances have 61 GB RAM, 8 CPUs, 160 GB SSD and cost 0.665 USD per hour on demand, as of this writing.

We used Infobright DB [24] as data warehouse, installed on a "r3.8xlarge" instance which has 244 GB RAM, 32 CPUs, 640 GB HDD, a 10 Gigabit network and costs 2.66 USD per hour on demand, as of this writing.

Throughout this paper, we illustrate the architecture with services provided by Amazon AWS, as it is currently the cloud provider with the highest market share of about 32% (https://www.canalys.com/newsroom/cloud-market-share-q4-2018-and-full-year-2018). Nonetheless, other providers, such as Windows Azure or Google Cloud, offer services with similar functionalities to the AWS services used in the paper. Therefore, it is relatively straightforward to replicate the proposed architecture in another cloud environment. Even on-premises infrastructures can provide suitable replacements for the utilized AWS services. In particular, instead of Amazon S3, one can use Microsoft Azure Blob Storage or HDFS of a Hadoop cluster which runs permanently (different than the on-demand short-lived cluster used for the ETL). Likewise, Microsoft Azure Event Hubs or an on-premises Apache Kafka deployment could substitute Amazon Kinesis. Similarly, Elasticsearch or Solr can be replaced by another text indexing service.

B. Experimental scenarios and datasets

Our experimental evaluation is performed on three ETL scenarios using the proposed architecture as described in [35]. The first evaluated scenario ETL1 is scenario where we don't perform data aggregation. The second scenario ETL2 is a scenario where we perform data aggregation for predefined time intervals, and the third scenario ETL3 is session based aggregation.

Table I shows the information about the datasets used for evaluating the three scenarios. The data was provided from a service that collects user logs very frequently and the result of the processing was timely and actionable information. All three proposed scenarios were used to populate several data marts for different purposes in the company. First, decision support systems leveraged aggregated data for evaluating investment opportunities and tracking historical performance. Next, the second and third ETL scenarios were applied for feature engineering for building two machine learning systems: one for churn prediction, and another one for fraud detection (i.e., account sharing against the terms of use). Finally, with the proposed architecture, we preprocessed the log data in order to infer the implicit feedback of users, in order to build a recommendation system.

The first scenario was evaluated with structured data that only required cleansing, transformation, surrogate keys generation, and setting up foreign keys to dimensions. The second and third scenario additionally required aggregations, and they were evaluated on user logs data collected during a typical workday. The data was published from a variety of devices (i.e., smartphones, web sites, and desktop applications) on an Amazon Kinesis stream with five shards in the "us-east-1" region. Events on the stream triggered a python lambda function up to 5 times per second per shard, which stored the raw JSON records, one record per line in a plain text file, in S3. The producers were able to generate up to 1000 records per second per Kinesis shard. With this configuration (5 shards), we could process a maximum of $1000 writes \times 5 shards = 5000$ events (source rows) per second ($24 hours \times 3600 seconds \times 5K = 432M$ events per day), that would be stored in up to $24 hours \times 3600 seconds \times 5 reads \times 5 shards = 2.16M$ S3 objects (files).

TABLE I: Information about the datasets and generated rows in each ETL scenario

	ETL scenario		
	ETL1	ETL2	ETL3
Source type	CSV	JSON	JSON
Source columns	31	17	17
Dest. aggr. columns	-	86	86
Dest. unaggr. columns	86	26	26
Source S3 objects	550	410K	410K
Source size (GB)	53	30	30
Source rows	137M	44M	46M
Dest. unaggr. rows	137M	44M	44M
Dest. unaggr. size (GB)	94	28	28
Dest. aggr. rows	-	2M	1M
Dest. aggr. size (GB)	-	2	1

Acronyms: Dest. - destination, aggr. - aggregated, unaggr. - unaggregated (i.e. not aggregated).

C. Cluster optimization results

The algorithm for cluster size cost-optimization depends on two input parameters: the source data size and the maximum time allowed for the ETL process to finish. Since all three scenarios have the first parameter predefined (53 GB for the first, and 30 GB for the second and third scenario), only the second parameter impacts the selection of the cluster size. Table II displays the selected

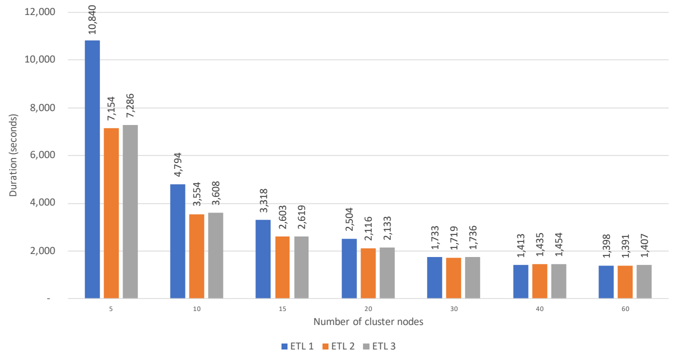

Fig. 2: Duration of ETL for each scenario per cluster size

TABLE II: Selected cluster size depending on maximum allowed time for the ETL process to finish

Max. time	ETL1		ETL2 and ETL3	
allowed (hours)	Nodes	Cost (USD)	Nodes	Cost (USD)
1	20	13.3	15	9.98
≥ 2	15	9.98	15	9.98

cluster sizes for each scenario, depending on the maximum time allowed for completion of the ETL process. When comparing the selected cluster sizes to the execution times and cost (Fig. 2), it evident that indeed, the most optimal clusters were selected. When compared to the clusters that would be selected with a brute force approach that tests all cluster sizes, starting from a small cluster, for these experiments, the same cluster sizes are obtained. For the first scenario and the maximum time of one hour, the algorithm needed one iteration less to find the optimal cluster. However, the true benefit from this algorithm would be apparent when the volume of data is much greater (e.g., terabytes of data) and the required clusters are much larger (hundreds of nodes). To confirm this, we performed simulations with interpolated execution times (linear interpolation for the Load and Booting times and logarithmic interpolation for the Spark time) and data volumes and compared it to the brute force approach. The proposed algorithm could find the optimal cluster size in significantly fewer iterations (i.e., evaluated clusters). Because the data used for the simulations was synthetic and based on the recorded data during the real experiments, it could be easily replicated. For conciseness of the manuscript, we are not providing the results of the simulations.

VI. Discussion

This study proposed architecture for efficient and scalable ETL of Big Data, consisting of distributed processing of data (extract and transform steps) with Apache Spark and distributed load into a data warehouse. We evaluated three common ETL scenarios: no aggregation, predefined time period aggregation and session-based aggregation.

For all cluster configurations and all ETL scenarios, Amazon does not charge for the booting time needed to provision the cluster. Nonetheless, this has an impact on the overall duration of the whole ETL process, so it needs to be considered. The booting time was very similar for smaller and larger clusters.

One limitation of our experimental evaluation is using relatively static volumes of data. Creating a realistic environment where data is generated with rapidly changing pace and volume, while it is being processed was out of the scope of this work, but is a prospective idea for future work. Nonetheless, the proposed system was deployed in a production system, where high streams of real data were generated and processed. The data volumes varied daily within a 30 percent margin, so this did not actually require using clusters with a different size than the one that was already configured at the start. The change in the data volume was not soliciting addition or removal of at least five nodes to get to the optimal cluster size.

Moreover, dynamically adding nodes while a cluster is executing was not appropriate considering: the overall duration of the spark job (less than one hour), booting time for newly added nodes (about 10 minutes) and considering at which step was already the Spark job. For such short jobs, by the time the newly added nodes are ready, the Spark job could have already distributed the tasks to existing nodes and they could be already processing them. Therefore, the new nodes would be idle because there are no tasks in the queue. Such an approach is more appropriate for long running jobs, something that we could not evaluate in a production environment with dynamic volumes.

The main limitation of the proposed architecture is that it cannot be used for real-time ETL, rather with some predefined latency. Another drawback of the proposed distributed data load architecture is that it was evaluated with only one data warehouse engine.

Let us also add that we are fully aware that our study refers just to a subset of Big Data V's. Nevertheless, the considered scenarios referring to large and dynamically increasing relational data sets occur very frequently in practice, with a number of research challenges that still need to be solved (and which are not fully solved by purely Hadoop-style systems with regards to velocity). Moreover, we believe that the proposed solutions addressing a combination of volume and velocity challenges can be adapted in the future also for handling the unstructured data.

VII. Conclusions

In this paper, we have proposed a cloud-based architecture for efficient ETL of Big Data. The extract and transform phases are performed by Spark, and then the results are loaded into a data warehouse using distributed load agents (DLAs) that utilize the processing resources of the cluster slaves (edge nodes), instead of the database server. To that end, the ETL process utilizes on-demand

Hadoop clusters with a variable size that run for a limited duration on Amazon AWS. By defining and evaluating three ETL scenarios that cover a variety of use-cases, we showed that with the proposed algorithm for clusters-size optimization, the number of nodes can be tuned so the ETL process can complete within the required time while minimizing the cost.

The proposed approach allows using already established ad-hoc, analytical and integration capabilities of traditional data warehouses. For some cases, such as processing of computer generated logs, the number of rows can be significantly reduced by some aggregations, while still preserving enough information for a variety of ad-hoc queries. For such cases, the pure Big Data solutions raise several challenges related to development time, compatibility with visualization and reporting tools, and massive overhead when executing MapReduce or Spark jobs. This is the real benefit of the proposed method that combines Big Data technologies for the heavy lifting (e.g., ETL and aggregation) and traditional data warehousing technologies for data exploration (analytics, reporting, visualization, etc.).

The proposed management system of DLAs can be integrated with YARN in a future work. Also, the response time for ad-hoc queries in Big Data systems could be reduced by eliminating, or at least lowering, the overhead for starting MapReduce and Spark jobs; designing keys that provide uniform distribution, and thus processing load across nodes; improving data segmentation on secondary indexes; or providing optimization suggestions for Spark jobs, just like traditional databases offer.

Acknowledgment

This work was partially financed by the Faculty of Computer Science and Engineering at the Ss. Cyril and Methodius University, Skopje, North Macedonia, and partially by CogniTrek Corp., Toronto, Canada.

Likewise, it was supported by the COST action CA15110 - Harmonising standardisation strategies to increase efficiency and competitiveness of European life-science research (CHARME) supported by COST (European Cooperation in Science and Technology).

References

[1] C. Dobre, F. Xhafa, Parallel Programming Paradigms and Frameworks in Big Data Era, International Journal of Parallel Programming 42 (5) (2014) 710–738. doi:10.1007/s10766-013-0272-7.

[2] H. Chen, R. Chiang, V. Storey, Business intelligence and analytics: From big data to big impact, MIS Quarterly: Management Information Systems 36 (4) (2012) 1165–1188.

[3] I. A. T. Hashem, I. Yaqoob, N. B. Anuar, S. Mokhtar, A. Gani, S. Ullah Khan, The rise of "big data" on cloud computing: Review and open research issues, Information Systems 47 (2015) 98–115. doi:10.1016/j.is.2014.07.006.

[4] M. D. Assunção, R. N. Calheiros, S. Bianchi, M. A. Netto, R. Buyya, Big data computing and clouds: Trends and future directions, Journal of Parallel and Distributed Computing 79 (2015) 3–15.

[5] C. P. Chen, C.-Y. Zhang, Data-intensive applications, challenges, techniques and technologies: A survey on big data, Information Sciences 275 (2014) 314 – 347. doi:10.1016/j.ins.2014.01.015.

[6] U. Dayal, M. Castellanos, A. Simitsis, K. Wilkinson, Data integration flows for business intelligence, in: Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology, EDBT '09, ACM, New York, NY, USA, 2009, pp. 1–11. doi:10.1145/1516360.1516362.

[7] S. Zhang, C. Zhang, Q. Yang, Data preparation for data mining, Applied Artificial Intelligence 17 (5-6) (2003) 375–381. doi:10.1080/713827180.

[8] D. Ślęzak, M. Grzegorowski, A. Janusz, M. Kozielski, S. H. Nguyen, M. Sikora, S. Stawicki, Ł. Wróbel, A framework for learning and embedding multi-sensor forecasting models into a decision support system: A case study of methane concentration in coal mines, Information Sciences 451-452 (2018) 112 – 133. doi:j.ins.2018.04.026.

[9] M. Grzegorowski, A. Janusz, D. Ślęzak, M. S. Szczuka, On the role of feature space granulation in feature selection processes, in: J. Nie, Z. Obradovic, T. Suzumura, R. Ghosh, R. Nambiar, C. Wang, H. Zang, R. Baeza-Yates, X. Hu, J. Kepner, A. Cuzzocrea, J. Tang, M. Toyoda (Eds.), International Conference on Big Data, BigData 2017, Boston, MA, USA, December 11-14, 2017, IEEE, 2017, pp. 1806–1815. doi:10.1109/BigData.2017.8258124.

[10] A. Janusz, M. Grzegorowski, M. Michalak, L. Wróbel, M. Sikora, D. Ślęzak, Predicting seismic events in coal mines based on underground sensor measurements, Eng. Appl. of AI 64 (2017) 83–94. doi:10.1016/j.engappai.2017.06.002.

[11] A. Cuzzocrea, L. Bellatreche, I.-Y. Song, Data warehousing and olap over big data: current challenges and future research directions, in: Proceedings of the sixteenth international workshop on Data warehousing and OLAP, ACM, 2013, pp. 67–70.

[12] H. Wada, A. Fekete, L. Zhao, K. Lee, A. Liu, Data consistency properties and the trade-offs in commercial cloud storage: the consumers' perspective., in: CIDR, Vol. 11, 2011, pp. 134–143.

[13] S. Chaudhuri, U. Dayal, V. Narasayya, An overview of business intelligence technology, Commun. ACM 54 (8) (2011) 88–98. doi:10.1145/1978542.1978562.

[14] P. M. Marín-Ortega, V. Dmitriyev, M. Abilov, J. M. Gómez, Elta: New approach in designing business intelligence solutions in era of big data, Procedia

technology 16 (2014) 667–674.

[15] R. Mukherjee, P. Kar, A comparative review of data warehousing etl tools with new trends and industry insight, in: 2017 IEEE 7th International Advance Computing Conference (IACC), 2017, pp. 943–948. doi:10.1109/IACC.2017.0192.

[16] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, A. Talwalkar, Mllib: Machine learning in apache spark, J. Mach. Learn. Res. 17 (1) (2016) 1235–1241.

[17] E. Zdravevski, P. Lameski, A. Kulakov, B. Jakimovski, S. Filiposka, D. Trajanov, Feature ranking based on information gain for large classification problems with mapreduce, in: Proceedings of the 9th IEEE International Conference on Big Data Science and Engineering, IEEE Computer Society Conference Publishing, 2015, pp. 186–191. doi:10.1109/Trustcom-BigDataSe-ISPA.2015.580.

[18] E. Zdravevski, P. Lameski, A. Kulakov, S. Filiposka, D. Trajanov, B. Jakimovski, Parallel computation of information gain using hadoop and mapreduce, in: M. P. M. Ganzha, L. Maciaszek (Ed.), Proceedings of the 2015 Federated Conference on Computer Science and Information Systems, Vol. 5 of Annals of Computer Science and Information Systems, IEEE, 2015, pp. 181–192. doi:10.15439/2015F89.

[19] E. Zdravevski, P. Lameski, A. Kulakov, Row Key Designs of NoSQL Database Tables and Their Impact on Write Performance, in: Proceedings - 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2016, 2016, pp. 10–17. doi:10.1109/PDP.2016.84.

[20] M. Bala, O. Boussaid, Z. Alimazighi, P-etl: Parallel-etl based on the mapreduce paradigm, in: 2014 IEEE/ACS 11th International Conference on Computer Systems and Applications (AICCSA), 2014, pp. 42–49. doi:10.1109/AICCSA.2014.7073177.

[21] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, R. Murthy, Hive - a petabyte scale data warehouse using hadoop (March 2010). doi:10.1109/ICDE.2010.5447738.

[22] A. Gupta, D. Agarwal, D. Tan, J. Kulesza, R. Pathak, S. Stefani, V. Srinivasan, Amazon Redshift and the Case for Simpler Data Warehouses, Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data - SIGMOD '15 2015-May (2015) 1917–1923. doi:10.1145/2723372.2742795.

[23] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, W. Lehner, SAP HANA Database: Data Management for Modern Business Applications, SIGMOD Rec. 40 (4) (2012) 45–51. doi:10.1145/2094114.2094126.

[24] D. Ślęzak, R. Glick, P. Betlinski, P. Synak, A new approximate query engine based on intelligent capture and fast transformations of granulated data summaries, Journal of Intelligent Information Systems.

[25] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, C. Curino, Apache tez: A unifying framework for modeling and building data processing applications, in: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, SIGMOD '15, ACM, New York, NY, USA, 2015, pp. 1357–1369. doi:10.1145/2723372.2742790.

[26] J. E. Gonzalez, P. Bailis, M. I. Jordan, M. J. Franklin, J. M. Hellerstein, A. Ghodsi, I. Stoica, Asynchronous complex analytics in a distributed dataflow architecture, arXiv preprint arXiv:1510.07092.

[27] X. Li, Y. Mao, Real-time data etl framework for big real-time data analysis, in: 2015 IEEE International Conference on Information and Automation, 2015, pp. 1289–1294. doi:10.1109/ICInfA.2015.7279485.

[28] H. Herodotou, F. Dong, S. Babu, No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics, in: Proceedings of the 2nd ACM Symposium on Cloud Computing, ACM, 2011, p. 18.

[29] K. Elmeleegy, Piranha: Optimizing short jobs in hadoop, Proceedings of the VLDB Endowment 6 (11) (2013) 985–996.

[30] R. Ranjan, Streaming big data processing in datacenter clouds, IEEE Cloud Computing 1 (1) (2014) 78–83. doi:10.1109/MCC.2014.22.

[31] H. Hu, Y. Wen, T. S. Chua, X. Li, Toward scalable systems for big data analytics: A technology tutorial, IEEE Access 2 (2014) 652–687. doi:10.1109/ACCESS.2014.2332453.

[32] S. Mathew, Overview of Amazon Web Services, accessed: 2019-06-04 (april 2017).

[33] M. Kiran, P. Murphy, I. Monga, J. Dugan, S. S. Baveja, Lambda architecture for cost-effective batch and speed big data processing, in: 2015 IEEE International Conference on Big Data (Big Data), 2015, pp. 2785–2792. doi:10.1109/BigData.2015.7364082.

[34] E. Zdravevski, P. Lameski, A. Kulakov, S. Kalajdziski, Transformation of nominal features into numeric in supervised multi-class problems based on the weight of evidence parameter, in: Proceedings of the 2015 Federated Conference on Computer Science and Information Systems, Vol. 5 of Annals of Computer Science and Information Systems, IEEE, 2015, pp. 169–179. doi:10.15439/2015F90.

[35] E. Zdravevski, C. Apanowicz, K. Stencel, D. Ślęzak, Scalable cloud-based etl for self-serving analytics, in: P. Perner (Ed.), Advances in Data Mining: Applications and Theoretical Aspects. 19th Industrial Conference, ICDM 2019, Springer International Publishing, Cham, 2019, pp. 387–394.