From Big Data to business analytics: The case study of churn prediction

Article *in* Applied Soft Computing · February 2020

DOI: 10.1016/j.asoc.2020.106164

4 authors:

Eftim Zdravevski
Ss. Cyril and Methodius University in Skopje
156 PUBLICATIONS **1,421** CITATIONS
SEE PROFILE

Petre Lameski
Ss. Cyril and Methodius University in Skopje
101 PUBLICATIONS **924** CITATIONS
SEE PROFILE

Cas Apanowicz
CogniTrek Corp.
13 PUBLICATIONS **74** CITATIONS
SEE PROFILE

Dominik Ślęzak
University of Warsaw
318 PUBLICATIONS **4,598** CITATIONS
SEE PROFILE

From Big Data to Business Analytics: The Case Study of Churn Prediction

Eftim Zdravevski[a,*], Petre Lameski[a], Cas Apanowicz[b], Dominik Ślęzak[c]

[a]*Faculty of Computer Science and Engineering, Sts Cyril and Methodius University, Skopje, North Macedonia*
[b]*CogniTrek Corp., Toronto, Canada*
[c]*Institute of Informatics, University of Warsaw, Poland*

Abstract

The success of companies hugely depends on how well they can analyze the available data and extract meaningful knowledge. The Extract-Transform-Load (ETL) process is instrumental in accomplishing these goals, but requires significant effort, especially for Big Data. Previous works have failed to formalize, integrate, and evaluate the ETL process for Big Data problems in a scalable and cost-effective way. In this paper, we propose a cloud-based ETL framework for data fusion and aggregation from a variety of sources. Next, we define 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 third scenario is very valuable in the context of feature engineering, making it possible to define features as "the time since the last occurrence of event X". 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. The business value of the architecture was demonstrated with applications in churn prediction, service-outage prediction, fraud detection, and more generally – decision support and recommendation systems. In the churn prediction case, we showed that over 98% of churners could be detected, while identifying the individual reason. This allowed support and sales teams to perform targeted retention measures.

Keywords: Data streams; ETL; Business analytics; Hadoop; Spark; Churn Prediction

1. Introduction

The ubiquitous smart devices, sensors and social media generate sheer data volumes. At the same time, consumers became accustomed to instantly-available personalized services. Many companies, health providers and institutions have focused tremendous resources on

[*]Corresponding author

Email addresses: `eftim@finki.ukim.mk` (Eftim Zdravevski), `petre.lameski@finki.ukim.mk` (Petre Lameski), `cas.apanowicz@cognitrek.com` (Cas Apanowicz), `slezak@mimuw.edu.pl` (Dominik Ślęzak) providing this. In the era of Big Data, companies experience a growing pressure to store and analyze all data that is being collected [1] to stay competitive in the data-driven marketplace. Volume, velocity and variety are intrinsic properties of Big Data [1, 2]. Recently, variability, veracity, visualization, and value were identified as similarly significant [3]. 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 [4].

In order to make the data available in a usable format, several steps need to be performed [5, 6]: 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 [1].

As identified in [7], there are various challenges for data warehousing (DW) and business intelligence (BI) over Big Data: size; complexity; design and data modeling; computing methodologies; query languages; usability; end-user performance; data consistency and lineage; extending applicability of traditional tools for data exploration, visualization and analytics; and integration with conventional DW/BI solutions and platforms. As envisioned in [7], some of the future research directions about DW and OLAP over Big Data should provide several improvements, including innovative solutions for computing aggregations, which is one of the main benefits of our work.

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 [8].

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 [8]. Big Data exacerbates these concerns because of the distributed architectures, which require more advanced mechanisms for synchronization, replication, scheduling and security [2].

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 [3].

Therefore, the primary goal of this study is to create a general and cost-effective framework for ETL of Big Data, covering several challenges: (i) coping with velocity concerns by creating a Data Lake; (ii) solving veracity concerns by defining ETL scenarios, template matching and data tagging; (iii) utilizing short-lived on-demand cloud resources for cost-effectiveness; (iv) investigation of the business value in a production deployment of the system in a churn prediction case.

This work defines three Extract-Transform-Load (ETL) scenarios concerning data aggregation commonly encountered in real-world systems. The first two are trivial and reflect ETL with no aggregation, and aggregation based on predefined columns or time intervals. The third one is about aggregation within single user sessions spanning over arbitrary time intervals. In addition to its use in traditional BI, it is very valuable in the context of feature engineering. This ETL scenario makes it possible to define features as "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. This scenario practically does not exist in the feature extraction literature, especially not as an integral part of the ETL process.

We evaluated the ETL scenarios with a combination of traditional tools – for processing dimensional data, and Spark executed on on-demand clusters – for processing high-volume transactional data. To validate the scalability of the architecture, we performed experiments on Amazon AWS clusters with datasets ranging from 30 GB to 2.6 TB.

We demonstrated the usability of the architecture with a churn prediction problem, utilizing thousands of automatically generated features during the ETL process. We integrated methods to explain the each prediction so that customer service and sales teams can perform personalized active measures to try to prevent the service cancellations. The experimental results of the proposed system particularly stand out from related works, because we evaluated the whole data-processing pipeline in a production setting with real high-velocity streaming Big Data.

The rest of the paper is structured as follows. Section 2 reviews the related work. Section 3 describes the proposed architecture for scalable ETL of Big Data, defines three common ETL scenarios and describes a churn prediction in which the output of the ETL was applied. Next, in section 4 we present the results from our experiments and then, in section 5 we discuss them. Finally, section 6 concludes the paper and identifies directions for future research.

2. Related work

Traditional BI relies on ETL tools (e.g., Informatica, IBM Infosphere Datastage, Ab Intio, Microsoft SQL Server Integration Services (SSIS), Oracle Data Integrator, Talend, Pentaho Data Integration Platform - PDI, etc.) that load data into warehouse servers [9]. Lately, Enterprise Application Integration (EAI) systems are succeeding ETL tools and perform much more functionalities. The other class of tools are the so-called "Extract-Load-Transform" (ELT). Those tools are mostly promoted by database vendors that rely on the power of database engines to perform the transformations, which are postponed to a later time. ELT is useful when business requirements change frequently, so it is convenient when the data is already loaded [10]. Authors in [11] review Traditional ETL and ELT tools, with focus on the description of their terminology and capabilities, but without significantly addressing Big Data challenges.

Apache Spark draws ideas from the MapReduce paradigm, focusing on applications that reuse a working set of data across multiple parallel operations. The work presented in [12] demonstrates the similarities between ETL and MapReduce processes and emphasizes the data partitioning as a missing aspect in the classical ETL deployments. The authors perform partitioning of the data within the Map step, and aggregate the transformed data during the Reduce step. The scalability was demonstrated on static datasets of up to 300 GB. However, this work does not address matching of dimensions from fact tables. In addition to addressing that, our approach also facilitates the processing of high-velocity data in a production setting.

Consistent data warehouses allow utilizing traditional visualization, reporting and business intelligence services, and simplify the use of data mining and machine learning libraries. The BigDimETL approach [13] aims to conserve the multidimensional structure of DW while integrating Big Data. Differently from our approach, this work is only theoretical, without any experimental evaluation. A distributed parallel architecture for ETL of Big Data is proposed in [14]. Unlike this approach, which requires the ETL to be completed before data can be aggregated, our approach performs aggregation during the ETL process.

An approach that proposes a set of rules to translate a conceptual design and map star schemas into NoSQL logical models with precomputed aggregate lattice is described in [15]. Similarly to our approach, aggregate metrics need to be defined up front so that the ETL process can compute all of them. Differently from this paper, which focuses on rules for generation of the model, we focus on systematising the ETL scenarios and creating distributed load agents to uniformly divide the load on on-demand clusters, before it even gets to the DW.

The work presented in [16] proposes the CloudETL system, which exploits MapReduce and Hive for distributed processing of data, focusing on slowly changing dimensions. Similar to this approach, we also use in-memory lookup for matching the fact tables to the corresponding dimensions. Our approach goes further utilizing lambda functions for handling high-velocity data to create data lakes, and then processes them with Spark. Finally, we demonstrate the value of the data marts by an end-to-end implementation, including an application in a churn prediction problem.

Addressing data veracity by cleansing and tagging, similar to our idea of cleansing and standardization with templates, is proposed in the GENUS system [17]. The document store used in this approach is XML, limitting the evaluation in regards to scalability, data volume, versatility and velocity. Processing historical and incoming data separately is proposed in [18]. To avoid the contention between OLAP queries and OLTP updates, this approach uses dynamic mirror replication technology. The experimental evaluation was conducted on a static dataset of only 16 GB, which is only a fraction of the dataset that we used in our experiments.

Real-world data processing and analytics projects often require the ETL to be carefully designed and implemented. One specific case of the ETL process are the stages of feature engineering, feature selection and prediction model construction [19]. However, properties of the data, such as class imbalance or missing data, aggravate parallelization of feature extraction and selection and require appropriate algorithms [20].

Customer churn is a common problem in many businesses and sophisticated techniques have been applied in churn prediction for years. However, making a breakthrough by improving prediction techniques is increasingly difficult. By using high-volume training data with a variety of features from business and operations support systems and high-velocity new data, [21] shows that prediction performance can be significantly improved. Moreover, this can be used to launch proper retention campaigns that targets potential churners. Motivated by this work in the telco industry, we applied similar techniques in the video streaming industry and came to similar conclusions. Additionally, we employed Local Interpretable Model-agnostic Explanations (LIME) [22] for explaining the predictions to help the customer sales teams in the retention campaigns. Another difference is that instead of manually generating potentially valuable features, the ETL capabilities of our architecture were leveraged to generate a variety of features automatically through various aggregations.

In Table 1 we summarize the differences between the related works that propose different approaches related to ETL of Big Data. Our proposed approach is in the highlighted row, clearly offering significant improvements in terms of capabilities and experimental evaluation over the related approaches. The last two rows for reference about directly applying Spark Streaming, or Spark, Pig or Map Reduce, without the proposed framework.

Table 1: Comparison of related works.

		Considerations			**Dataset**			**Dimensions**					
Approach	**Paral.**	**Velocity**	**Veracity**	**Destination**	**Type**	**Size**	**Format**	**Surrogate keys**	**Key matching**	**SCD**	**Load Model**	**Aggregations**	**Application**
[12]	MR	No	N/A	HDFS	Static	100 GB	CSV	No	No	No	Full, Diff.	Fixed, During	No
[13]	MR	No	N/A	N/A	N/A	N/A	CSV, JSON	No	Yes	Type 1	N/A	Fixed, During	No
[14]	MR	No	N/A	HBase	N/A	N/A	CSV	No	No	No	N/A	Fixed, After	No
[15]	MR	No	N/A	HBase, Mongo DB	Static	375 GB	CSV, JSON	No	Yes	No	N/A	Fixed, During	No
[16]	MR, Pig	No	N/A	Hive	Static	320 GB	CSV	Yes	Yes	Type 2	Full, Diff	During	No
[17]	N/A	No	Templates, tagging	N/A	Static	N/A	XML	No	No	No	N/A	None	No
[18]	MR	Separates processing of historic and real-time data.	N/A	Hive	Dynamic	32 GB	CSV	No	No	No	Full, Diff.	Fixed, During	No
Proposed	Spark, Pig, MR	Maintaining a Data Lake with Lambda functions.	Templates, tagging	S3, Blob Storage, HDFS, Hive, HBase, Infobright	Dynamic	2.6 TB	CSV, JSON	Yes	Yes	Type 2	Full, Diff.	Arbitrary, During	Churn prediction
Spark Streaming	Spark	Limited to recent data.	N/A	Any	Dynamic	Any	CSV, JSON	No	No	No	Full	Fixed, During	N/A
Spark, Pig, MR	Spark, Pig, MR	No	N/A	Any	Static	Any	CSV, JSON	No	No	No	Full	Fixed, During	N/A

Paral. denotes the framework for parallelizing the computation. MR means MapReduce. Velocity refers to considerations about data collected at high velocity. Veracity denotes any effort in the standardization of data. The Static and Dynamic dataset types denote whether the data was static and at rest, or dynamically changing, respectively. Yes in surrogate keys means surrogate keys for dimensions are being generated, and Yes in Key matching means that they are maintained by setting foreign keys in the fact tables. SCD denotes Slowly Changing Dimensions support. Diff. denotes differential or incremental load. Aggregations: Fixed means they are performed on fixed time periods, Arbitrary means on arbitrary time periods ranges, During means during the ETL and After means after the ETL.

3. Methods

3.1. Architecture

3.1.1. 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).

We chose PDI because it enables users to ingest, blend, cleanse and prepare diverse data from any source. Its visual tools eliminate coding and complexity for creation of data pipelines. It offers data agnostic connectivity spanning from flat files and RDBMS to Hadoop, powerful orchestration and scheduling capabilities (including notifications and alerts), agile views for modeling and visualizing data on the fly during the data preparation process, support for Hadoop distributions, Spark, NoSQL data stores and analytic databases, etc.

On the other hand, 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.) [23].

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 [24]. 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:

Push Lambda Functions (Stream-based model) Herein, 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 [25, 26]. 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.

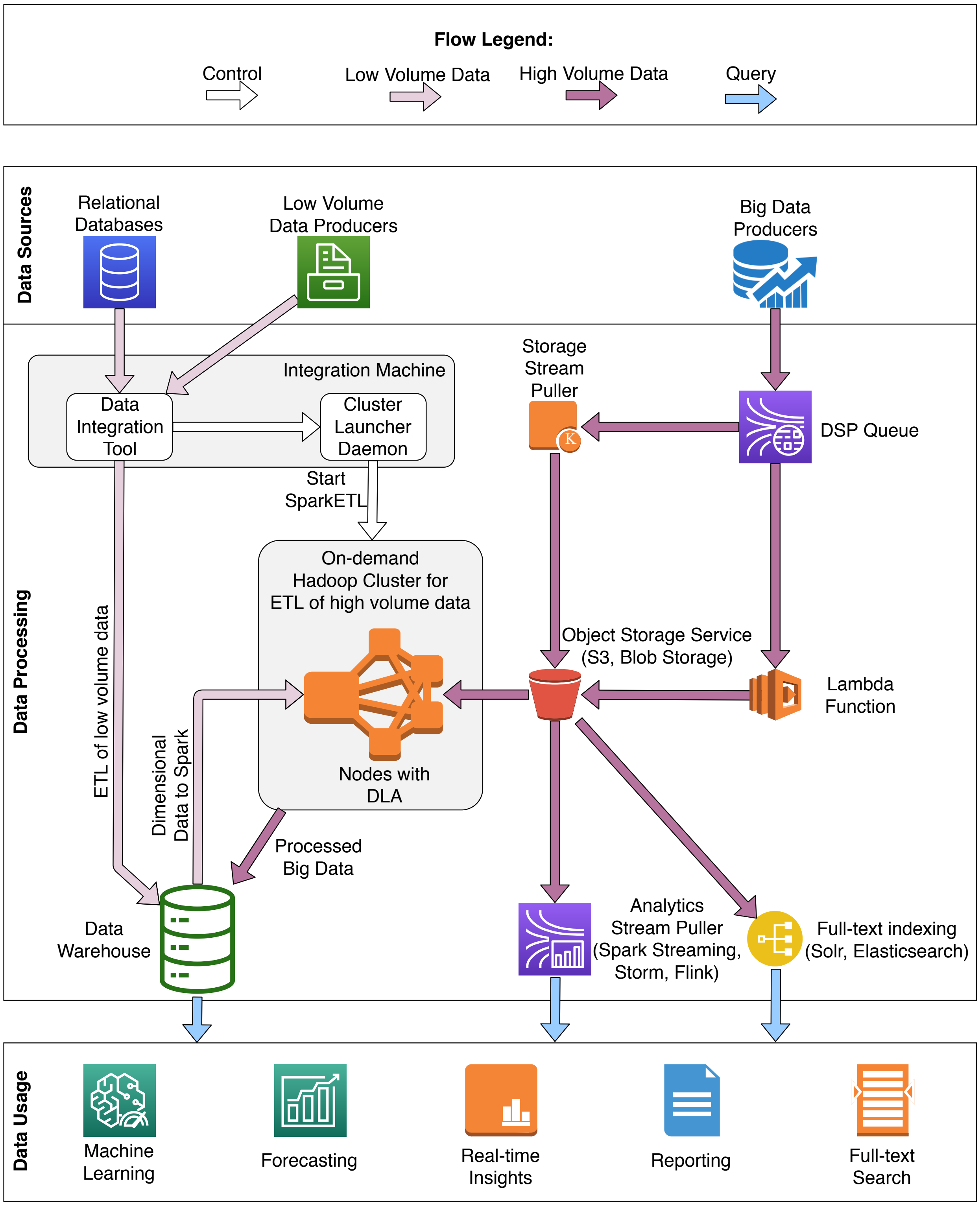


Figure 1: Architecture of scalable Cost-optimizing cloud-based Big Data Warehouse.

Storage Stream Pullers These 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 mechanism provides an alternative for durable and persistent storage of the raw data with a variable frequency of persisting data [25].

Analytics Stream Pullers Spark Streaming and Apache Storm are typical consumers that provide stream processing and real-time analytics [25].

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.

Ability to perform full-text searches could provide some robustness to data veracity and complement the analytical capabilities of data warehouses. Services such as Elasticsearch or Solr [27] could cleanse the raw data stored in OSS (e.g., extract plain text from HTML files) and then index it to provide full-text searching functionalities, as shown in Figure 1.

The Analytics Stream Pullers (e.g., Apache Spark Streaming and Apache Storm) are a different kind of consumers that process the streaming data to provide near real-time insights and analytics. 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. 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 (more examples are provided in Subsections 3.2.1, 3.2.2 and 3.2.3). 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 [28]), 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.

3.1.2. Distributed Load Agents

Traditionally, data ingestion is a huge burden on database servers and often is a bottleneck. After we complete the extraction and transformation steps, and we set the primary and foreign keys, we only need to perform the load process. The idea for Distributed Load Agent (DLA) comes from the principles of edge computing, and the goal is to offload most of the load process to remote machines at the edge and away from the data warehouse. These edge nodes would compress data and prepare an output, which, at least theoretically, could be copied to the end of the database. Thus, the overall impact on the database server would be minimal. The work presented in [29] demonstrates that the load process can indeed work so efficiently. In the proposed architecture, the whole on-demand Hadoop cluster is considered as being on the edge from the data warehouse perspective (see Figure 1).

Hadoop YARN (Yet Another Resource Navigator), which is responsible for job scheduling, monitoring and resource management, guarantees reliable execution of Spark jobs even if some nodes of the Hadoop cluster die. Likewise, HDFS can detect and automatically recover after node failures. DLAs use the underlying Hadoop infrastructure, so each of them executes on a slave node and loads some portion of the data to the data warehouse. However, if a node fails, the DLA dies along with it. Because DLAs are an extension to Hadoop and are not part of the standard distributions (i.e., Cloudera, Hortonworks, MapR, Amazon EMR, and Azure HDInsight), we have to manage them separately. To provide reliability of this component of our framework, we propose a distributed system for managing DLAs, which monitors and uniformly distributes their tasks across the Hadoop cluster.

In Figure 2, we present the sequence diagram of the proposed DLA management. The initialization process determines the number of available nodes, the number of tables that need to be loaded, and the number of horizontal partitions on HDFS generated by Spark. Mapping of data to tables happens on the edge-nodes by the agents with predefined rules and templates. For instance, by using a set of regular expressions, a variety of fields could be correctly standardized (e.g., date/time fields, URLs, e-mail addresses, phone numbers, IP addresses, amounts and currencies, floating point numbers, campaign identifiers, referrers, postal addresses and other spatial data, device IDs, or other domain-relevant entities).

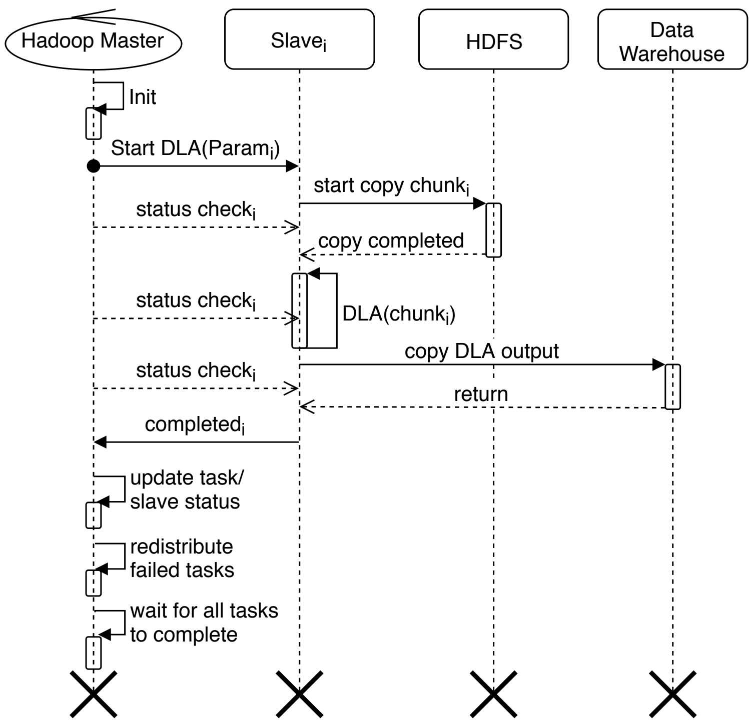


Figure 2: Sequence diagram of the Distributed Load Agents

Then DLA task parameters are configured (i.e., list of files that need to be processed on a particular node), so each node in the cluster executes DLA on approximately the same amount of data. When a node completes the DLA process, it notifies the master. Meanwhile, the master pings the nodes and checks their status, to continuously monitor the overall progress of each task. If it detects a DLA failure or a stale process, it reassigns the task to the first idle node. We store all metadata related to the DLAs (i.e., node and task statuses) on HDFS for reliability. This storage policy provides the ability to overcome any failure of the master because then the secondary master would be able to continue the orchestration of the DLAs. If during the lifetime of the Hadoop cluster (while executing a Spark or DLA task) a corruption the cluster is detected, this information is propagated to the Master and/or back to the Cluster Launcher. When DLA completes successfully on all nodes, the master collects and estimates the duration of each step, the volume of processed data, the number of rows per table, cost, etc. We upload this metadata to OSS and the data warehouse, so it can be further used for self-calibration of cluster-size and for estimating future task duration. Finally, the master gracefully self-terminates the cluster.

The DLA experiments were performed using the distributed data load processor layer developed by Infobright from 2005 through 2016, which is now available as a part of Infobright DB product (`https://www.ignitetech.com/index.php/download_file/view_inline/221/240/`). We chose this particular solution because it has been already well adopted within the organization wherein the framework proposed in this paper was evaluated, owing to its analytical performance for ad-hoc queries and the ingestion rates at the level of 2 TB of data per hour. For more information about the characteristics of Infobright's load processes, let us refer to [30], where so-called knowledge capture agent - the new generation of the aforementioned distributed load processor - is considered.

3.2. ETL data-flow scenarios

In this section, we describe three ETL data-flow scenarios commonly needed in organizations. These scenarios relate to the major portion of data to be stored in data warehouses, i.e., data contained in fact tables. The volume of dimensional data is considerably smaller; therefore, it does not necessarily require processing based on Big Data technologies, rather traditional ETL tools for this type of data are sufficient. The proposed architecture assumes that traditional ETL tools already process the dimensional data and that it only needs to process data to be stored in fact tables. We present the steps for implementing the ETL process of the three scenarios in Figure 3.

Business (i.e., natural) keys denote unique record identifiers that could have some business meaning, but most importantly, they are managed by operational data stores (ODS). In a data warehouse, a surrogate key is a necessary generalization of the ODS business key and is one of the basic elements of data warehouse design. Every join between dimension tables and fact tables in a data warehouse environment is based on surrogate keys, not business keys. It is up to the data extract logic to systematically look up and replace every incoming business key with a data warehouse surrogate key each time either a dimension record or a fact record is brought into the data warehouse environment. The surrogate keys provide independence from ODS business keys, which could be subject to deletion, updating, or recycling.

Populating foreign keys in fact tables in a traditional way with joins between the fact and dimension tables would be inefficient and unfeasible for large datasets because it requires shuffling and redistributing data across the cluster nodes. On the other hand, if the dimensions' business and surrogate keys are distributed across nodes in advance, populating foreign surrogate keys is a simple dictionary lookup operation based on the business key with $O(1)$ complexity. This method does not require reshuffling and adheres to the data locality principle.

In Figure 3 the proposed data flow for the three scenarios is shown, and each of them is described in more detail in the following subsections. All scenarios share steps one and two. Spark in the *first step* loads the business and surrogate keys of all processed dimensions, and then distributes them to each node. Also, the maximum values of the surrogate keys for each table are calculated in step 1, because they define the starting values of the new surrogate keys that would be generated during a subsequent run of the ETL process. For non-existing business keys, new surrogate keys are generated as a sequence of increasing integers, starting from the current maximum key for the table. Gaps in the generated sequence of numbers are allowed by design (for computational efficiency). During the surrogate key generation, their density is calculated (defined as the ratio of the total number of surrogate keys and the maximum surrogate key), which shows how efficiently they are used. If the density is very low and the maximum value of surrogate keys increases rapidly, so this may be used as a recommendation to redesign the key generation approach.

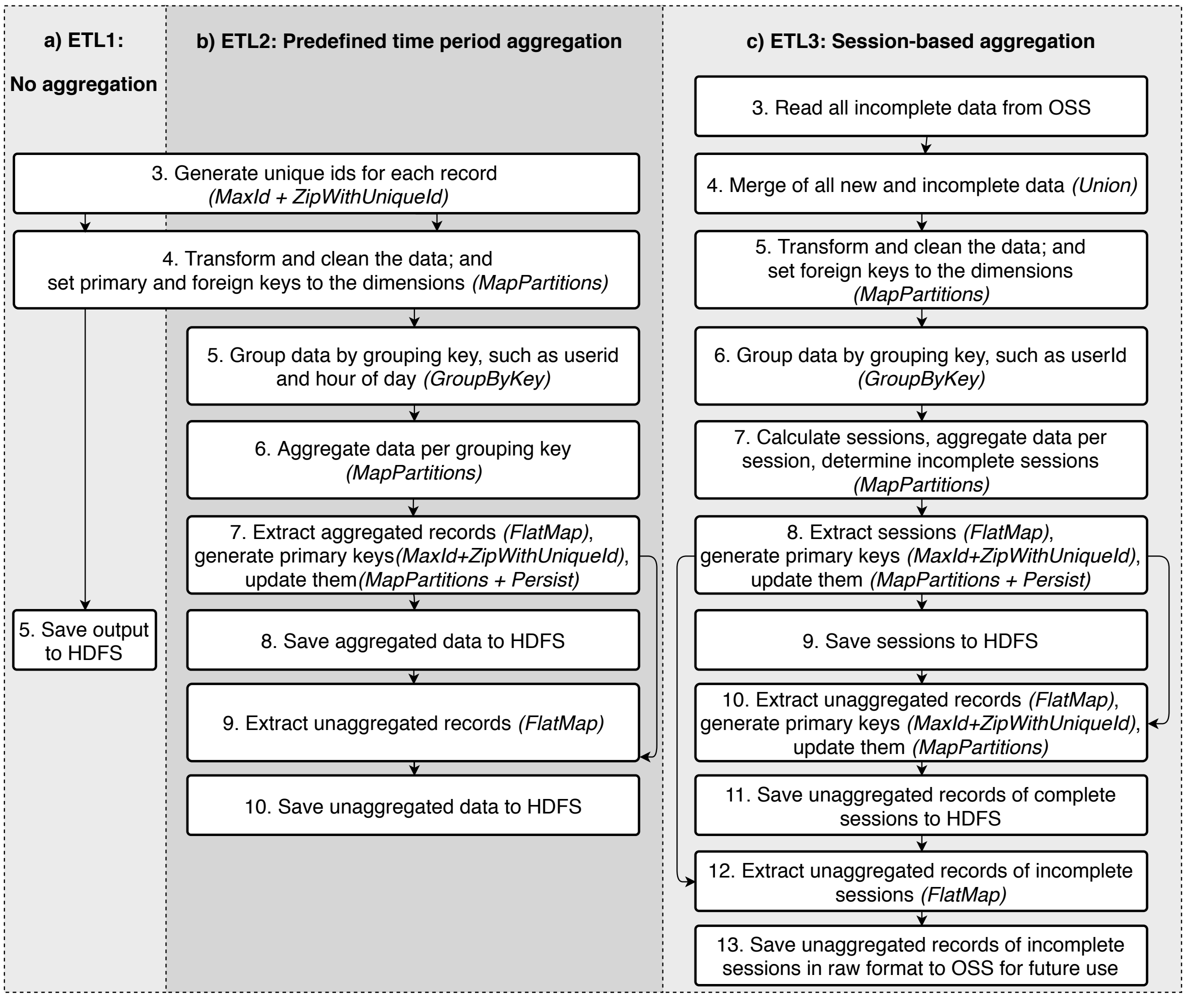

a) ETL1: No aggregation
b) ETL2: Predefined time period aggregation
c) ETL3: Session-based aggregation
3. Generate unique ids for each record (MaxId + ZipWithUniqueId)
4. Transform and clean the data; and set primary and foreign keys to the dimensions (MapPartitions)
5. Save output to HDFS
5. Group data by grouping key, such as userid and hour of day (GroupByKey)
6. Aggregate data per grouping key (MapPartitions)
7. Extract aggregated records (FlatMap), generate primary keys(MaxId+ZipWithUniqueId), update them(MapPartitions + Persist)
8. Save aggregated data to HDFS
9. Extract unaggregated records (FlatMap)
10. Save unaggregated data to HDFS
3. Read all incomplete data from OSS
4. Merge of all new and incomplete data (Union)
5. Transform and clean the data; and set foreign keys to the dimensions (MapPartitions)
6. Group data by grouping key, such as userId (GroupByKey)
7. Calculate sessions, aggregate data per session, determine incomplete sessions (MapPartitions)
8. Extract sessions (FlatMap), generate primary keys (MaxId+ZipWithUniqueId), update them (MapPartitions + Persist)
9. Save sessions to HDFS
10. Extract unaggregated records (FlatMap), generate primary keys (MaxId+ZipWithUniqueId), update them (MapPartitions)
11. Save unaggregated records of complete sessions to HDFS
12. Extract unaggregated records of incomplete sessions (FlatMap)
13. Save unaggregated records of incomplete sessions in raw format to OSS for future use

Figure 3: Steps for completing the three ETL scenarios: (a) ETL1: No aggregation; (b) ETL2: Predefined time period aggregation, (c) ETL3: Session-based aggregation

Next, *step 2* reads all new data from OSS (with the Spark operations TextFiles or WholeTextFiles). Because OSS storage is distributed, different nodes of the cluster read distinct portions of data, thus achieving high parallelism. Usually, files on OSS are stored with a hierarchical prefix describing the date and time when that data was collected (e.g., on AWS S3 `s3://bucket/collection/folder/2018/01/12/23/34/`), therefore it is trivial to determine the source path of the new data for a particular run.

Even though the steps in between differ between scenarios, after the data is stored on HDFS, the Data Load to the data warehouse are executed in a similar manner using the proposed algorithm for a distributed load, processing each table in a parallel way, one at a time. Ultimately, all metadata collected during the cluster lifetime (i.e., various metrics such as duration of each step, number of processed rows per table, etc.) is also loaded into the data warehouse, and then the cluster self-terminates.

3.2.1. ETL scenario 1: No aggregation

The first scenario of ETL requires parsing, data type conversion, setting foreign keys (Extract and Transform steps) and only loading into the fact tables of a data warehouse. This scenario is the simplest of the three and does not require any aggregations in the fact tables. Such use-cases require that all generated data be available at the lowest level of granularity (after performing proper cleansing), as well as their associations with other entities in the system. Some typical use-cases of this work-flow refer to the log analysis in the following application fields:

Resource management: Monitoring across systems to detect particular log events and patterns in the data; Identifying performance or configuration issues; Meeting operational objectives and SLAs; etc.

Application troubleshooting: Diagnosing and identifying root causes of application installation and run-time errors; Pinpointing areas of poor performance; Assessing application health and troubleshooting.

Marketing insights: Gathering insights and analyzing their impact on visibility, traffic, conversions, and sales; Revealing potential improvements of Search Engine Optimization (SEO); Understanding which pages are useful and useless; etc.

Regulatory Compliance and Security: Maintaining compliance with industry regulations often requires all data to be preserved in source format, while tagging and identifying certain events in data.

What is common about these use-cases is that the original data needs to be completely preserved without any level of aggregations so that one could pinpoint particular events. From this perspective, it is also worth mentioning one more application aspect - regulatory compliance and security. Indeed, maintaining compliance with industry regulations often requires all data to be preserved in source format, while tagging and identifying certain events in the data.

The steps for completing the ETL of this scenario are shown in Figure 3a. *Step 3* generates unique numeric identifiers for each row with the ZipWithUniqueId transformation. Even though there can be gaps in the generated numbers by this operation, it does not require data shuffling, making it very efficient. When the maximum value of the surrogate key (MaxId) is added to the generated unique number, a unique surrogate key of each row is obtained.

In *step 4* the transform phase of ETL is performed, consisting of data transformations, data cleaning, data type casting, setting primary surrogate keys (using the unique ids generated in the previous step), and setting foreign keys to the dimensions (by performing lookups in the dictionary already distributed to each node in step 1). This step uses the MapPartitions Spark operation, which guarantees that the transformations will not cause shuffling, thus adhering to the data locality principle.

Subsequently, *step 5* stores the output of the transformations to HDFS in text format.

3.2.2. ETL scenario 2: Predefined time period aggregation

The second ETL scenario refers to a predefined time period aggregation. This scenario is commonly present in data aggregation concerning nominal or dynamically quantized column domain (e.g., user, campaign, asset) in conjunction with some predefined time period. Therefore, it is an extension of the first scenario because it analyzes the granular data and picks out important pieces of information (e.g., response times, customer plan dollar values, performance resource usage info, etc.), then rolls them up into aggregated records, which can be displayed in metrics dashboards. Associating the aggregated records with the actual records that comprise them, allows drilling down. For example, if suddenly a spike in the number of signups on a daily basis happens, the change can be quickly validated by checking the logs to see who signed up and when. This is not always possible with the traditional metrics dashboards, as they do not maintain the source of data that is used to calculate the metrics. Another use of the aggregated data is for concept drift detection, for trend analysis over more extended periods, or engineering features for machine learning algorithms [6, 19].

Determining which aggregate metrics would be needed requires significant manual effort by data warehouse architects [5, 6]. Therefore, a more conventional approach is to generate a variety of potentially useful metrics and rely on statistical and feature selection methods to discover the more relevant ones afterward. Another reason to include more metrics is that data changes over time so the importance of the metrics could change. For those reasons, this ETL scenario generates a variety of aggregate metrics upfront, comparable to [15].

The steps of the second ETL scenario are shown in Figure 3b. Steps 1 to 4 are identical to the previous scenario and are related to data transformations, data cleaning, and setting primary and foreign keys.

The records with the same grouping key are grouped with the GroupByKey operation in *step 5*. Even though the grouping key could be any column, we recommend using time dimension columns (e.g., hourly or daily aggregations). The reason for that is because the proposed architecture uses the Infobright data warehouse, which already facilitates swift ad-hoc queries, including aggregations on any dimensional data, owing to its knowledge grids.

Step 6 aggregates the records in the same group, thus producing an aggregate record with one or more aggregate values (e.g., count, sum, min, max, or any user-defined aggregate function). Additionally, in each aggregate record, all records that comprise it are also preserved (i.e., are not lost because of the aggregation).

Next, *step 7* extracts the aggregated records (with the Map or FlatMap operations), generates primary keys for them with the same method as applied in step 3, and finally updates the aggregated records to reflect the generated primary keys. Additionally, it sets the foreign key to the aggregate record in all records that comprise it, which is the sole reason why the comprising records were preserved in the previous step. This step also uses the Persist operator at the end, which stores the result in memory. By doing this, the next time the result of this step is needed (in step 9), no re-computation would be required.

Step 8 stores the aggregated records (without the comprising records) on HDFS.

Subsequently, *step 9* extracts the comprising records of each aggregate record with the FlatMap operation in one set of unaggregated records.

Then these records are stored into HDFS in *step 10*.

3.2.3. ETL scenario 3: Session-based aggregation

Aggregation on predefined time periods still does not cover all use-cases. As an example, let us go back to the feature engineering task mentioned previously. As discussed in [19], focused on predictive maintenance and risk monitoring, the derived features of the form, e.g. "the time since the last occurrence of event X" can be of special importance. Likewise, in churn prediction problems, the "the time since the user's last login", "last used service", or "last bought product", can be useful features [31]. Even though such features are logically easy to understand, their calculation requires to look in variable and practically unlimited periods of the past data. On the other hand, in typical streaming scenarios, only the recent data is available.

Another challenging case is when aggregations on time periods are of arbitrary length. These situations are common when there is an interaction between users and services, and the user sessions could last hours or even days. Such cases are browsing sessions which end either by logging out or by some period of inactivity; viewing sessions of video services; gaming sessions; etc. Historically, such sessions were related to the user's activity on a single device, and therefore, were easily distinguished by the corresponding device id. However, nowadays the user can start a given activity on one device and continue on another (e.g., one can start watching a movie on a tablet or smartphone and then, shortly after, resume on a smart TV). From a business logic perspective, these seemingly distinct sessions should be often considered as single sessions, thus making the traditional method for distinguishing them unsuitable. One might try to cope with this by updating session records in real time, but in case of millions of concurrent users with hundreds of millions of updates per hour, this is simply unfeasible. A more suitable approach is thus to generate and maintain atomic user activity log files, where records are only inserted, and then process them in an offline fashion to determine the unique user activities that we are interested in.

Using technologies such as Spark streaming cannot easily cope with this type of scenario because the time period in which the user session is not predefined and can be very long, requiring very long sliding windows, which in turn, require a lot of memory. Moreover, Spark streaming requires a cluster to be running permanently, which incurs higher costs, even though real-time analytics may not always be required (e.g., if a delay of a couple of hours is acceptable, or if the reports are daily, weekly or monthly).

The third ETL scenario calculates user sessions, then performs aggregation on session level and finally loads data in an aggregated and unaggregated format. We propose to address the ETL process of this scenario by using the algorithm shown in Figure 3c. The algorithm processes all records that comprise complete sessions and loads the result in the warehouse. Incomplete sessions are defined as sessions that at the end of the period that is being processed (e.g., a day's worth of data) were still active. Therefore, the records of incomplete sessions are stored separately in raw format at the end of the algorithm, so they could be processed in the next run.

Steps 1 and 2 are the same as in the previous two scenarios. *Step 3* reads data that was part of incomplete sessions from a location on OSS where the previous run of the algorithm stored them. *Step 4* merges the two RDDs of the new and incomplete data with the Union operator. Next, in *step 5* data is cleansed and transformed, and the foreign keys to the dimensions are also set.

In *step 6* the records are grouped by a more coarse grouping key, such as user id. This enables implementation of complex business rules in *step 7* for determining user sessions because all recent user records are available in one logical and physical location. During *step 7*, the complete sessions are determined, along with the records that comprise them, and some aggregations are performed per session. Additionally, step 7 identifies incomplete sessions, which are also included in the aggregated data but are marked as incomplete. The records that comprise incomplete sessions are kept in the original format so they can be stored on OSS for re-processing in the next run.

Step 8 extracts complete sessions with the FlatMap operator, generates primary keys for them and updates the records to reflect the generated keys: aggregated to have proper primary keys and comprising records to have proper foreign keys to the corresponding aggregate (session) records. The result of this step is preserved in memory because it will be needed three times in the following steps.

Step 9 stores the complete sessions to HDFS.

Using the result from step 8, *Step 10* combines the unaggregated records that comprised completed sessions, and generates and sets their primary keys. Subsequently, in *step 11* these records are stored on HDFS.

Next, *step 12* extracts the unaggregated records of incomplete sessions into one set and then in *step 13* stores them in the original format (without any data cleansing and transformations) in OSS so they can be used in the next run in step 3.

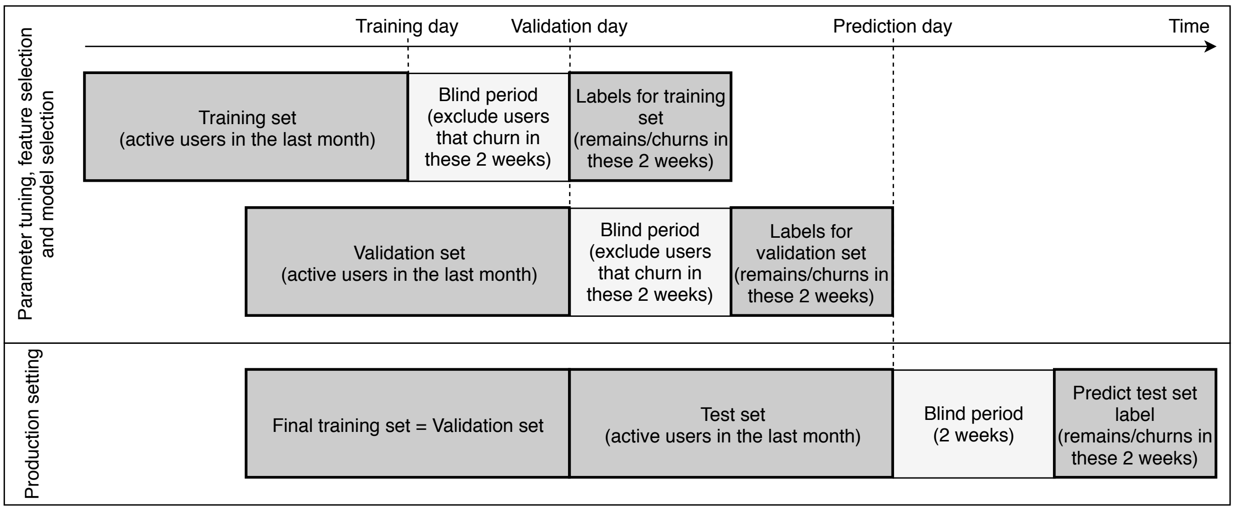
3.3. Business analytics application: Customer churn prediction

3.3.1. Problem description

The ultimate goal of any ETL process is to provide support of various business applications. The proposed architecture was applied for timely and actionable information for a recommendation system, service-outage prediction, fraud detection (i.e., account sharing against the terms of use) and historical performance tracking and decision support. For brevity, here we only describe the use-case for a customer churn prediction problem for a video streaming service operating in the Canadian market. The goal is to analyze usage patterns and discover which customers will cease the business relationship with the company in the near future, and identify the reasons for that.

Three types of data were available for the churn prediction. *First*, demographic and personal data from the customer profile (e.g. number of avatars, account age, number of profiles, number of kids profiles, payment history, account type,etc.). *Second*, results from customer surveys were also available. For these two types of data, ETL scenario 1 was utilized. *Third*, a lot of dynamic data originating from the user interactions with the service. This data, in its raw format, is unusable for churn prediction, or any analytics scenario for that matter, as it only contains logs generated hundreds of times for each minute the user streams some content. At the lowest level of granularity was a record called heartbeat, contains a timestamp and various information describing the user, profile, content, device, operating system, state, and other meta information. As expected, it sometimes contains missing or malformed data.

Utilizing the three proposed ETL scenarios, the raw data was associated and aggregated with other information available from the user profiles (e.g., gender, age, subscription date, region, etc.), content description (e.g., genre, category, tags, etc.), service-quality metrics for the same user session (e.g., streaming speed, dropped packages, latency, etc.). Some of the resulting aggregated metrics were defined manually and had business meaning (e.g., the total number of times or total time some content id was watched). Others were calculated automatically and did not have any business meaning (e.g., mean, minimum, maximum, sum and standard deviation of the number of heartbeats per watching session), but could be potentially valuable for some analytics applications.



Parameter tuning, feature selection and model selection
Training day
Validation day
Prediction day
Time
Training set (active users in the last month)
Blind period (exclude users that churn in these 2 weeks)
Labels for training set (remains/churns in these 2 weeks)
Validation set (active users in the last month)
Blind period (exclude users that churn in these 2 weeks)
Labels for validation set (remains/churns in these 2 weeks)
Production setting
Final training set = Validation set
Test set (active users in the last month)
Blind period (2 weeks)
Predict test set label (remains/churns in these 2 weeks)

Figure 4: Dataset creation for the churn prediction problem

The requirements for customer churn prediction specify that there should be enough time before the predicted churn time. This would allow the company to take active measures to prevent the customer from churning (e.g., a retention campaign offering some special promotion, discount, bonus content, etc.). For this particular use-case, we selected active customers in the last month and predicted whether they would churn between two and four weeks of the churn prediction time. For example, data from July subscription period were used to predict whether the customer will churn between August 15 and August 31. The setting is shown in Figure 4, corresponding to the "no-memory approach" described in [32]. This setup provides at least two weeks (e.g., the first half of August) for the customer support and sales teams to reach out to the customer and try to prevent the service cancellation. Please note that even though the limitation was that the predictions are made for active users within a certain month, the features describing a particular customer requires processing of unlimited passed data (i.e., since the customer joined).

A condition for users to be considered in the analysis was that they were active for the whole month, meaning they registered before the training period started. The reason was two-fold. First, to overcome the cold-start problem some data is required to build the model. Second, newly subscribed users get the first month for free, so they are not relevant for the churn prediction problem due to business reasons. However, they are of primary interest for the similar task of predicting which newly subscribed customers will choose a paid plan after the trial period. Even though we built a model for this problem, this is out of the scope of this paper.

Considering that in real systems concept drift occurs over time, we also implemented the strategy described in [33] to deal with it. Namely, by detecting value distribution variations of features in the training and validation sets, we keep ones that are more stable throughout time. The same training/validation scheme is also used for hyper-parameter tuning of classification algorithms, as well as to select the final algorithm that will be used in the production setting to make the classification.

3.3.2. Feature extraction

Using the three proposed ETL scenarios, data was already processed and aggregated on various levels. Initially, heartbeats records were processed using ETL scenario 3 scenario providing aggregate metrics about a watching session (e.g., the total time in player, watched percentage of the content, how many devices were used within it, number of fast forwards, number of pauses, rewatched time, count of missing heartbeat records, etc.).

Then, the session data was further aggregated on bi-monthly, monthly, bi-weekly, weekly, and daily level, per user account and profile, genre, etc. Meanwhile, other data (e.g., number of distinct contents watched, number of heartbeats, distinct devices used, number of operating systems, etc.) was processed per ETL scenario 2. All aggregated data were later combined and further aggregated calculating various statistical metrics (i.e., count, mean, mode, median, minimum, maximum, standard deviation, different percentiles, kurtosis, and skewness).

Let us again stress the importance of ETL scenario 3 for these kinds of applications with another example that utilized the previously processed data. Namely, events, such as "time since last answered survey", "time since last fully watched content", "time since last fully watched content with tag X" could be valuable predictors. The same goes for features like "number of technical issues per day", that require proccessing of various lower level metrics (e.g. a watching session with more than 5% missed heartbeats, number of restarts within the same session, number of dropped packets, etc.). With typical streaming applications, where only recent data is available, these types of features could not be calculated because they require processing of potentially unlimited periods of data.

In order to end up with tabular data for the classification algorithms where each row represents a customer and all colums are cetrain features, dimensional data was pivoted and all metrics were available for each value of the dimension as a separate feature. For example, for the "family" and "comedy" genres, there is a different feature originating from the same metric (e.g., total number of heartbeats, total watch time, etc.). We additionally computed differences of the same metric and aggregation type between consecutive intervals (e.g., the variation of the total number of videos watched in the first half and second half of the month). As a result, thousands of features were generated. The whole process was performed every two weeks, facilitating churn predictions with the most recent data, but also stressing the importance of a cost-effective and efficient solution.

4. Results

4.1. Experimental setup

Contributing to the popularity in industry and research community of the Amazon Web Services (AWS), and the hardware heterogeneity offered in various instance types [25], it was used for the experimental evaluation of the proposed architecture. Nonetheless, it would be relatively straightforward to replicate the proposed architecture on other cloud providers, such as Windows Azure or Google Cloud, considering that they already offer similar services. Even on-premises infrastructures can provide suitable replacements for the utilized AWS components.

From all available instance families on AWS, we used the "r3.2xlarge" instance type (61 GB RAM, 8 CPUs, 160 GB SSD) for our experiments because it has high network performance while offering better granularity in controlling the cluster resources compared to larger instances.

We used Infobright DB as data warehouse because of the following reasons: fast ad-hoc queries; no need for physical database design (indexing, partitioning, tablespaces design, etc.); extremely fast load process exceeding terabytes per hour with high scalability; ability to run multiple parallel loads to a single table; and very small footprint of the database. The data warehouse was installed on a "r3.8xlarge" instance which has 244 GB RAM, 32 CPUs, 640 GB HDD, a 10 Gigabit network.

The data used for evaluating the framework was collected from a variety of devices (i.e., smart phones, web sites, and desktop applications) which published events 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).

The scalability experiments with each of the scenarios were repeated ten times and all presented times represent the average of the repetitions. All scenarios were evaluated on clusters with 5, 10, 15, 20, 30, 40 and 60 nodes, so we could investigate the impact of cluster size on the speedup.

Table 2 shows the information about the datasets used for evaluating the three ETL scenarios. The data was provided from a video-streaming service that collects user logs hundreds of times each minute.

Table 2: Information about the datasets and generated rows in each ETL scenario

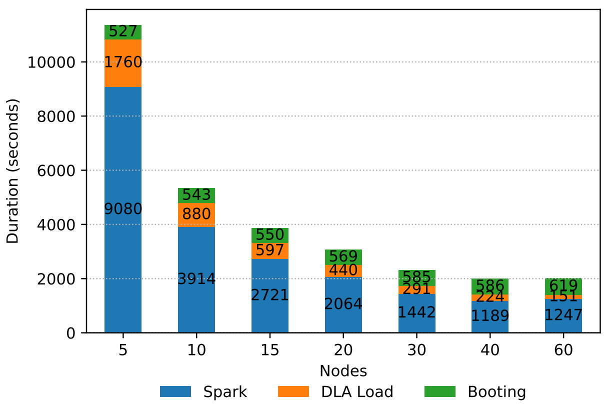
	ETL scenario			
	ETL1	ETL2	ETL3	ETL3 (100 days)
Source type	CSV	JSON	JSON	JSON
Source columns	31	17	17	17
Dest. aggr. columns	-	86	86	86
Dest. unaggr. columns	86	26	26	26
Source S3 objects	550	410K	410K	36M
Source size (GB)	53	30	30	2603
Source rows	137M	44M	46M	3987M
Dest. unaggr. rows	137M	44M	44M	3985M
Dest. unaggr. size (GB)	94	28	28	2427
Dest. aggr. rows	-	2M	1M	108M
Dest. aggr. size (GB)	-	2	1	70

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

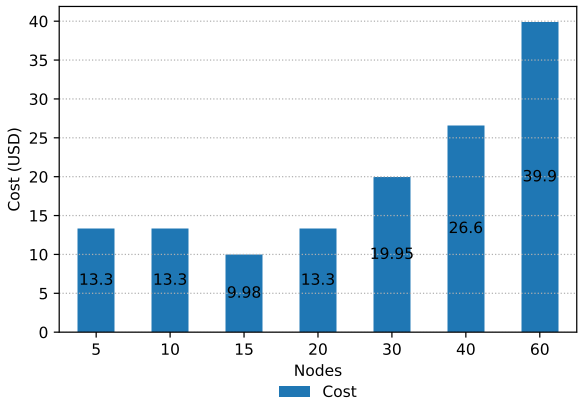
4.2. Evaluation of the ETL scenarios

All three proposed scenarios were used to populate several data marts for different purposes, most notably a churn prediction system, a recommendation system and a fraud-detection system. The first ETL scenario is evaluated with structured data that only requires cleansing, transformation, surrogate keys generation, and setting up foreign keys to dimensions. The second and third ETL scenarios additionally required aggregations, and they were evaluated on user logs data collected during a typical workday.

For the *ETL scenario 1*, which does not require aggregation, we experimented when all data was stored in one large text file, as well as in 550 smaller text files (see Table 2, column ETL1). The size of the source files did not influence the performance of the system, which is understandable, considering that S3 is a distributed storage system. The performance of each step (Spark duration and DDL duration) depending on cluster size is shown in Figure 5a. Note that Amazon does not bill the booting duration. Similarly, the cost depending on cluster size is shown in Figure 5b. From these two figures, it is evident that the 15-node cluster was the most cost-effective because its chargeable duration is just under one hour. It is also notable that when more than 30 nodes were used, the overall duration did not improve significantly. For the *ETL scenario 2*, which performs aggregation on predefined

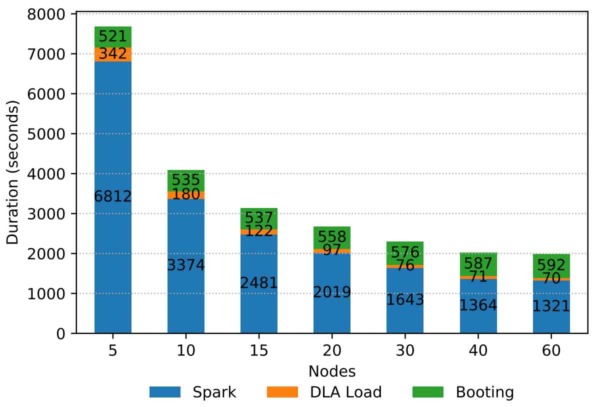


(a) Duration

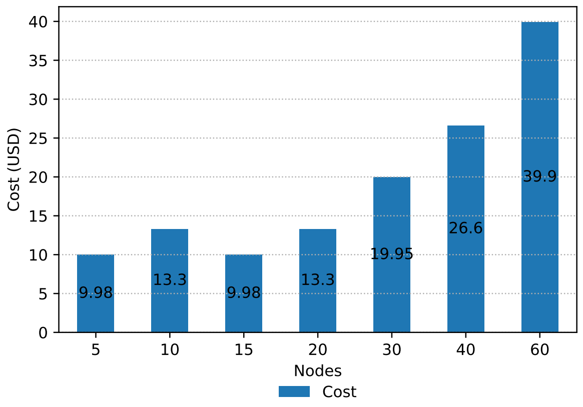


(b) Cost

Figure 5: Duration of each step (a) / cost (b) for ETL scenario 1 per cluster size

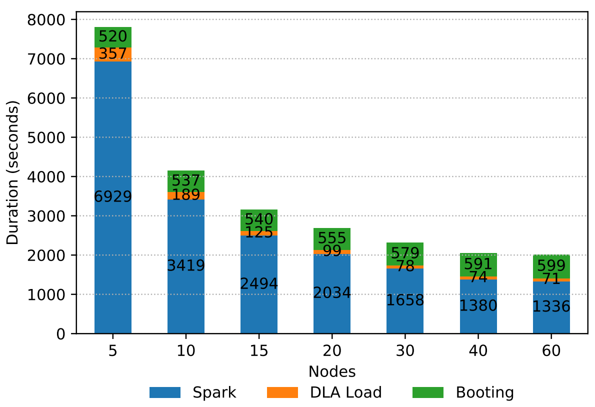


(a) Duration

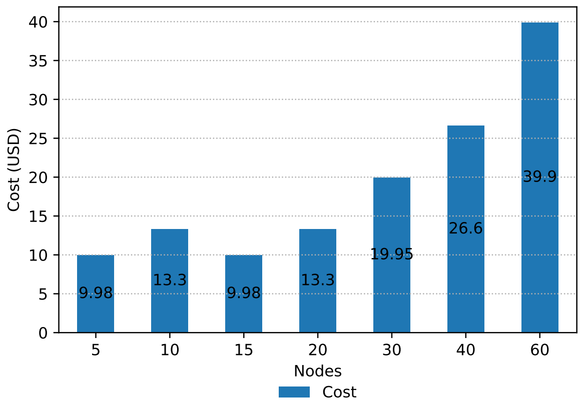


(b) Cost

Figure 6: Duration of each step (a) / cost (b) for ETL scenario 2 per cluster size



(a) Duration



(b) Cost

Figure 7: Duration of each step (a) / cost (b) for ETL scenario 3 per cluster size

time periods, the duration of each step and the cost depending on cluster size is shown in Figures 6a and 6b, respectively. Obviously, the 5-node and 15-nodes cluster are the cheapest, yet the latter completes the job considerably faster for the same cost. Finally, the results of the experiments with the *ETL scenario 3*, which performs session-based aggregation, are shown in Figures 7a and 7b. Similar to the second scenario, the most cost-effective is the 15-node cluster.

To verify that the proposed architecture is reliable and sustainable, we have executed the third scenario (as it is the most complex of the three) on a considerably increased workload using data collected during 100 days (see Table 2, column ETL3 (100 days)). We used a 20-node cluster with "r3.2xlarge" instances. Considering that the volume of the source data (2.6 TB) exceeds the storage capacity of the cluster ($20 \times 160 = 3.2$ TB) total hard drive space, of which less than 1 TB is available for HDFS), the Spark and DDL jobs were executed interchangeably one day at a time (i.e., the flow shown in Figure 3c was executed 100 times on the same cluster). This also enabled the results of the ETL to be available even though the whole process is not fully completed. The Spark jobs completed in 174,481 seconds in total, or on average, about 1,745 seconds per day's data. This is considerably less than when a cluster of the same size processed the daily data (2,034 seconds, see Figure 7a). We attribute these savings to the overhead of starting a Spark job on a new cluster and initialization of dimensional data key lookup dictionaries, and to the variance in daily data volumes. DDL completed in 8,514 seconds, an increase which is linearly proportional to the processed data volume. Figure 8 shows the obtained speedup of the Extract and Transform steps in Spark when comparing different cluster sizes for the three ETL scenarios, which is based on the results provided in Figures 5a, 6a and 7a. Obviously, as the number of nodes increases, the speedup decreases.

4.3. Business Analytics application: Customer churn prediction

Using the approach described in subsection 3.3.1, every two weeks the training validation and test datasets were regenerated, selecting only active users in the appropriate time period. During the retrospectively evaluated period for which we present the results, each of the training, validation and test datasets contained about 200,000 subscribers, of which about 10% churned. As a result of the automated feature extraction process during the ETL, for each of them there were available about 2,000 features, which were reduced to about 350 during the validation phase. The test results presented in Table 3 show the five best performing algorithms using those 350 features. The validation experiments showed that SVMs require significant time for hyper-parameter tuning without improvement of the scores, so it was not considered for the test experiments.

The information gain of the top 10 most informative features is shown in Table 4. Based on this analysis, we found that the activity of the users within the last two weeks before the prediction day was the most informative. Understandable, users who did not use the service in the last 2 weeks had a higher probability to churn than people who watched one or more videos. Similarly, users that subscribed more than two months ago and are still active, are less likely to cancel the subscription.

Table 3: Classifier performance on the test dataset with 10% churners (positive class).

Classifier	Accuracy	AUC	Precision	Recall	NPV	Specificity	F1p	F1n
XGBoost	0.9761	0.9850	0.9727	0.9975	0.9899	0.8982	0.9849	0.9418
AdaBoost	0.9705	0.9820	0.9695	0.9936	0.9746	0.8864	0.9814	0.9284
ExtraTrees	0.9735	0.9813	0.9706	0.9965	0.9858	0.8902	0.9833	0.9355
RandomForest	0.9744	0.9806	0.9726	0.9954	0.9819	0.8980	0.9839	0.9381
DecisionTrees	0.9737	0.9756	0.9718	0.9953	0.9814	0.8952	0.9834	0.9363

Table 4: Top 10 most informative features for churn prediction

Feature description	Information gain
Days since subscribed	0.014712
Number of distinct assets watched in last 2 weeks	0.004214
Mean time in player in last 2 weeks	0.004168
Std of watched seconds in last 2 weeks	0.004051
Std of time in player in last 2 weeks	0.003882
Mean of heartbeats in last 2 weeks	0.003789
Mean seconds watched in last 2 weeks	0.003711
Count of heartbeats in last 2 weeks	0.003677
Std of heartbeats in last 2 weeks	0.003671
Diff of seconds watched between first and second half of last month	0.003572

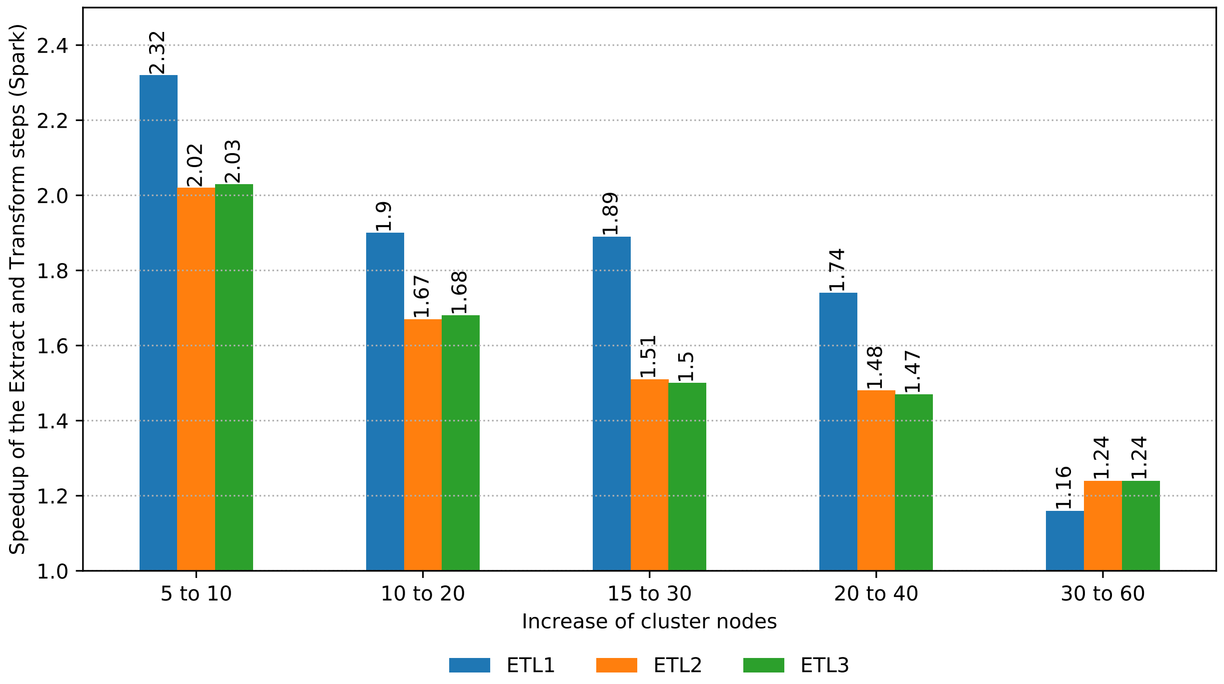


Figure 8: Speedup of the Extract and Transform steps (Spark) of the three ETL scenarios: ETL1 - no aggregation; ETL2 - predefined time period aggregation; and ETL3 - session-based aggregation.

Even though the black-box ensemble models had superior predictive performance, the more-easily interpretable Decision Trees model provided easier interpretation. However, it was still over-whelming for production use, hence the LIME approximation was used. Figure 9 shows an exemplary LIME explanation of a “churns” and “remains” predictions made by XGBoost. In this example, only 5 features are shown.

5. 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 requiring 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 (see Figures 5a, 6a and 7a). Because all experiments for each scenario were repeated at least ten times, we could detect cases when the booting time of 5-node clusters was even longer than the booting time of 60-node clusters, which we attribute to the varying resource availability in Amazon AWS. Nonetheless, in the hundreds of experiments, clusters up to 60 nodes with “r3.2xlarge” instances, this variance was negligible and the clusters were always booted in less than 15 minutes. For clusters larger than

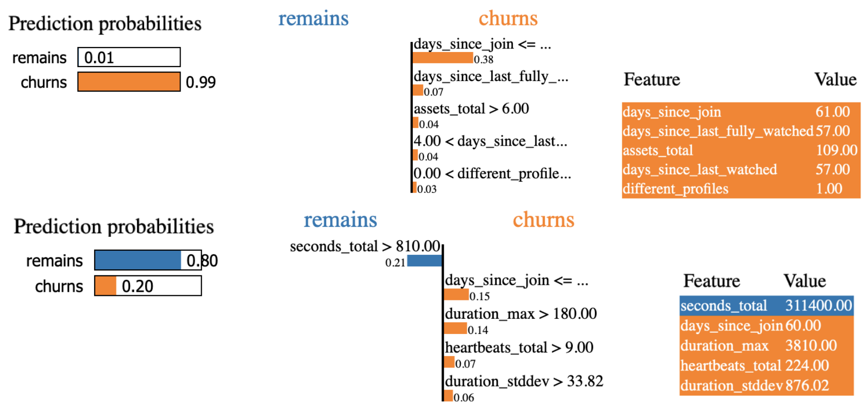


Figure 9: LIME explanations of predictions made by XGBoost: churns (top), remains (bottom)

100 nodes or ones requiring more powerful instance types than "r3.2xlarge", the booting time was more variable, which needs to be taken into account if on-demand clusters are used.

For all three scenarios, the distributed load into the warehouse scaled very well, having almost linear scalability (see Figures 5a, 6a and 7a). Be that as it may, we noticed that in some rare cases when the smallest cluster (5 nodes) was used, one of the nodes would die. This was no issue for losing data that was already stored on HDFS because of its replication. However, for the DLAs, it would have been a problem if it was not the algorithm proposed in subsection 3.1.2 (see Figure 2). In such cases, the load of a particular chunk of data, which is atomic operation from the warehouse point of view, would be repeated by the first available node. Node failures happened very rarely during Spark's execution. Despite that, no manual effort was needed for recovery, because Spark distributed the failed tasks to other nodes using Hadoop and YARN facilities.

Even though the predefined time period aggregation ETL scenario logically differs from the session-based aggregation scenario, from Spark's data movement perspective, they are very similar. The latter has only two extra steps (steps 12 and 13 in Figure 3c), which store the unaggregated data of incomplete sessions to S3. In the experimental dataset, about 5% of the sessions were incomplete, resulting in storing about 1.5 GB to S3 for processing on the next run. Depending on the cluster size, it took 50 to 70 seconds for Spark to store this data to S3 step, making its duration insignificant compared to the duration of the whole ETL process. Likewise, the number of rows that were loaded into the warehouse in these two scenarios were very similar (see Table 2), with a difference of about one million aggregated rows, which was negligible (see Figures 6a and 7a).

The Spark process within all three scenarios considerably benefited when increasing the cluster from 5 to 10 nodes, yielding a super-linear speedup (see Figure 8). For the no aggregation scenario, which was evaluated on a 53 GB dataset, the duration dropped from 9,080 to 3,914 seconds (see Figure 5a). For the other two scenarios, which were evaluated on a 30 GB dataset, the duration dropped from about 6,900 to about 3,400 seconds (see Figures 6a and 7a). The reason for these super-linear speedups, particularly notable in the first scenario, was the insufficient RAM on the 5-node clusters to store all data into memory, which caused data spills on the hard drive. For clusters with at least ten nodes, this was not a problem for all three scenarios. When the node number increased, the speedup started to decline (see Figure 8). The no aggregation scenario had better speedup than the aggregation scenarios when using up to 40 nodes, which we attribute to the simpler data flow. However, when the nodes increased to 60, the performance was worse than when using 40 nodes (see Figure 5a). To confirm that this was not a peculiarity of this cluster size, we experimented with the first ETL scenario with a 100-node and even a 200-node cluster. The results were even worse because of the great communication and synchronization overhead. This confirms that cluster size should be optimized for data load and that Big Data approaches are not suitable for problems with small amounts of data. From all experiments and results, we can also conclude that there is a physical limit of how fast the ETL process can be completed. We could reduce the execution time under one hour for all ETL scenarios, but further increasing the nodes did not additionally lower the execution times.

Although some metrics calculated during the ETL turned out to be uninformative for the churn prediction problem, they were precious for other business analytics. Features related to the number of dropped packets per operating system and user session were very insightful for detecting production issues with mobile applications, that lead to the discovery of memory management issues and the sequence of actions that lead to them. Likewise, they were useful for predicting cancellation reasons, performed within a next phase that classified the predicted churners in a multi-label multi-class setting. This, in combination with the explained predictions by LIME, was particularly useful to sales teams.

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

6. 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) reducing the workload on 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 the ETL process could be effectively used for feature extraction of non-trivial features that require processing of variable periods of past data.

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.). We showed that the proposed system brings real benefits to business applications, such as churn prediction by performing the feature extraction within the ETL. Experiments showed that churners can be identified with over 0.98 AUC score while providing explanations of the classification that help sales teams to launch personalized retention campaigns.

In future work, the proposed management system of DLAs can be integrated with YARN. Likewise, the algorithms could be expanded to consider hybrid aggregation scenarios during ETL. Further research could be focused, e.g., on approximate database engines and their information capturing capabilities for providing quick and dirty answers quickly, thus allowing users to visualize and find data segments of interest very fast. They could facilitate navigation through the data with efficient roll-ups and drilling down, while still enabling to execute exact queries when needed. Another promising direction is to integrate machine learning algorithms within the ETL in a non-blocking manner. This would allow for online re-calibration of models, aiming to cope with concept drift, while complementing traditional analytics with instant predictions.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

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

[7] A. Cuzzocrea, Data warehousing and olap over big data: a survey of the state-of-the-art, open problems and future challenges, International Journal of Business Process Integration and Management 7 (4) (2015) 372–377.

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

[13] H. Mallek, F. Ghozzi, O. Teste, F. Gargouri, Bigdimetl: Etl for multidimensional big data, in: A. M. Madureira, A. Abraham, D. Gamboa, P. Novais (Eds.), Intelligent Systems Design and Applications: 16th International Conference on Intelligent Systems Design and Applications (ISDA 2016) held in Porto, Portugal, December 16-18, 2016, Springer International Publishing, Cham, 2017, pp. 935–944.

[14] C. Boja, A. Pocovnicu, L. Batagan, Distributed parallel architecture for "big data", Informatica Economica 16 (2) (2012) 116.

[15] M. Chevalier, M. El Malki, A. Kopliku, O. Teste, R. Tournier, How can we implement a multidimensional data warehouse using nosql?, in: S. Hammoudi, L. Maciaszek, E. Teniente, O. Camp, J. Cordeiro (Eds.), Enterprise Information Systems: 17th International Conference, ICEIS 2015, Barcelona, Spain, April 27-30, 2015, Revised Selected Papers, Springer International Publishing, Cham, Switzerland, 2015, pp. 108–130.

[16] X. Liu, C. Thomsen, T. B. Pedersen, Cloudetl: Scalable dimensional etl for hive, in: Proceedings of the 18th International Database Engineering Applications Symposium, IDEAS '14, ACM, New York, NY, USA, 2014, pp. 195–206. doi:10.1145/2628194.2628249.

[17] S. Souissi, M. BenAyed, Genus: An etl tool treating the big data variety, in: 2016 IEEE/ACS 13th International Conference of Computer Systems and Applications (AICCSA), 2016, pp. 1–8. doi:10.1109/AICCSA.2016.7945615.

[18] 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.

[19] 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.

[20] 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.

[21] Y. Huang, F. Zhu, M. Yuan, K. Deng, Y. Li, B. Ni, W. Dai, Q. Yang, J. Zeng, Telco churn prediction with big data, in: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, SIGMOD '15, ACM, New York, NY, USA, 2015, pp. 607–618. doi:10.1145/2723372.2742794. URL `http://doi.acm.org/10.1145/2723372.2742794`

[22] M. T. Ribeiro, S. Singh, C. Guestrin, "why should i trust you?": Explaining the predictions of any classifier, in: Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '16, ACM, New York, NY, USA, 2016, pp. 1135–1144. doi:10.1145/2939672.2939778. URL `http://doi.acm.org/10.1145/2939672.2939778`

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

[24] 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.

[25] S. Mathew, Overview of Amazon Web Services, accessed: 2019-06-04 (april 2017).
URL `https://d0.awsstatic.com/whitepapers/aws-overview.pdf`

[26] 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.

[27] J. Bai, Feasibility analysis of big log data real time search based on hbase and elasticsearch, in: 2013 Ninth International Conference on Natural Computation (ICNC), 2013, pp. 1166–1170. doi:10.1109/ICNC.2013.6818154.

[28] A. Dwaraki, S. Seetharaman, S. Natarajan, T. Wolf, Gitflow: Flow revision management for software-defined networks, in: Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research, SOSR '15, ACM, New York, NY, USA, 2015, pp. 6:1–6:6. doi:10.1145/2774993.2775064.

[29] J. Borkowski, Performance debugging of parallel compression on multicore machines, in: R. Wyrzykowski, J. Dongarra, K. Karczewski, J. Wasniewski (Eds.), Parallel Processing and Applied Mathematics: 8th International Conference, PPAM 2009, Wroclaw, Poland, September 13-16, 2009, Revised Selected Papers, Part II, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 82–91.

[30] D. Ślezak, A. Chadzyńska-Krasowska, J. Holland, P. Synak, R. Glick, M. Perkowski, Scalable cybersecurity analytics with a new summary-based approximate query engine, in: 2017 IEEE International Conference on Big Data (Big Data), 2017, pp. 1840–1849. doi:10.1109/BigData.2017.8258128.

[31] B. Huang, M. T. Kechadi, B. Buckley, Customer churn prediction in telecommunications, Expert Systems with Applications 39 (1) (2012) 1414 – 1425. doi:10.1016/j.eswa.2011.08.024.

[32] A. S. Iwashita, V. H. C. de Albuquerque, J. P. Papa, Learning concept drift with ensembles of optimum-path forest-based classifiers, Future Generation Computer Systems 95 (2019) 198 – 211. doi:https://doi.org/10.1016/j.future.2019.01.005.
URL http://www.sciencedirect.com/science/article/pii/S0167739X1832257X

[33] E. Zdravevski, P. Lameski, V. Trajkovik, A. Kulakov, I. Chorbev, R. Goleva, N. Pombo, N. Garcia, Improving activity recognition accuracy in ambient-assisted living systems by automated feature engineering, IEEE Access 5 (2017) 5262–5280. doi:10.1109/ACCESS.2017.2684913.