SELIS BDA: Big Data Analytics for the Logistics Domain

Nikodimos Provatas, Evdokia Kassela, Nikolaos Chalvantzis
Anastasios Bakogiannis, Ioannis Giannakopoulos, Nectarios Koziris
National Technical University of Athens, Athens, Greece
{nprov,evie,nchalv,abk,ggian,nkoziris}@cslab.ece.ntua.gr

Ioannis Konstantinou
University of Thessaly, Lamia, Greece
ikons@uth.gr

***Abstract*—In this paper we present the SELIS Big Data Analytics and Machine Learning System (BDA), an open-source cloud-enabled elastic system that has been designed and implemented in order to address data related issues from the logistics domain. By taking into consideration real-life data analytics needs from more than 40 EU logistics providers we present the detailed SELIS BDA architecture along with the generic data and execution model devised to accommodate their diverse needs. We describe the main technologies we have utilized to realize the respective offering and justify our choices from the wider open-source Big Data systems community. We experimentally test our offering under various workloads where we prove that it can scale to serve a large number of concurrent requests while its abstraction/orchestration poses a very small overhead compared to the stand-alone Big Data systems. We believe that the SELIS BDA can be an easy-to-use entry point for the big data analytics world for any logistics company especially from the SME domain.**

***Index Terms*—Logistics, Cloud Computing, Analytics**

I. Introduction

Supply chain ecosystems consist of networks of multiple different agents such as retailers, 3PL companies, manufacturers, *etc.* with the common goal of achieving efficient and cost-effective patterns for the transfer of goods. These networks produce a vast amount of heterogeneous information which is constantly updated at extremely fast rates. IoT machinery such as RFID sensors, cameras, GPS devices, *etc.* also contribute to this ever flowing stream of information resulting in a typical high volume, high velocity and high variety "Big Data" setting. It is well known that the efficient collection, storage and analysis of this sort of information in a timely manner can often prove impossible with the use of traditional data processing techniques. At the same time, however, it can offer many advantages to organizations which have access to the resources and skills required to implement and utilize solutions that can help them gain valuable insights and understanding of how the business works through data analysis. The logistics domain is no exception to the above [1]. From supply chain visibility, to inventory management and forecasting [2], Big Data technologies are able to assist in solving complex problems and offer powerful tools to optimize the entire supply chain operation.

According to a recent Third-Party Logistics study [3], more than 98% of the participants agreed that data-driven decision making will shape the future. However, while Big Data analytics seem to be the solution for the optimization of supply chain operation, certain prerequisites need to be fulfilled before organizations are in the position to fully benefit from the power that Big Data technologies can offer [4]. In our involvement in the SELIS project, we observed that many logistics companies lack specialized IT expertise to configure and operate the massive hardware resources which are essential for Big Data analysis to be performed, especially small and medium-sized enterprises (SMEs). Moreover, the IT personnel of SMEs may lack the required data analytics and IT skills to analyze the ever increasing amount of data at their disposal. Finally, existing widely used legacy systems are often not able to interoperate and exchange data in a real-time manner, as required for state-of-the-art analytics workflows (refer to section 2 of [5] for a thorough analysis of respective adoption barriers). Thus, any necessary processing is extremely difficult to schedule and execute since the data need to be transferred in a centralized location beforehand, following the "data lake" concept [6]. Even in cases where logistics companies decide to adopt modern Big-Data-enabled approaches, commercial solutions are often offered as black-box products, without the flexibility to customize and modify the purchased services according to their own needs.

Meanwhile, open-source general purpose Big Data frameworks, systems and deployment tools have existed for close to a decade and are widely adopted by both industry and academia for large scale data processing. Specifically, solutions from the Hadoop [7] and Spark [8] ecosystem can be used for generic collection and storage of data, while distributed Big Data enabled machine learning toolkits, such as Spark MLlib, Google Tensorflow and GraphLab are able to handle data of both high speed and volume. Cloud-based deployment solutions enabling scalability and elasticity are available, either directly as a simple service (*e.g.*, Amazon's Elastic MapReduce [9], Google's MapReduce and Spark on Google Cloud Platform [10] and Azure [11]) or at higher levels of abstraction (*e.g.*, Docker containers [12] managed by Mesos [13] or Kubernetes [14]). Last but not least, open-source distributed real-time processing frameworks such as Apache Storm [15], Spark Streaming [16], Kafka [17],

978-1-7281-6251-5/20/$31.00 ©2020 IEEE

Flink [18] and Heron [19] have also reached a very mature state and are widely used in industrial settings. Although these approaches are not specifically targeted towards logistics, they do contain the necessary technology that can be tailored to the logistics community's needs.

One of the most important business requirements for a network of collaborating logistics stakeholders is the seamless information exchange between participants. Towards this direction, the multi-million EU research project SELIS[1] with more than 40 participants from the logistics and ICT domains proposed and delivered a "platform for pan-European logistics applications". This aspired offering has been described as a Shared European Logistics Information Space which can be used to link participants' existing systems and offer a collaborative environment. In the heart of this project lies the "SELIS Community Node" (SCN). The SCN is a collaboration space shared between business partners in a specific logistics ecosystem. It includes all the necessary technical components (*i.e.*, cloud-enabled software modules) which offer the data processing, storage and exchange functionalities (for a thorough description of the SCN refer to the SELIS Architecture Deliverable D4.1 [20]). In this setup, data is a first-class citizen and therefore the respective software subsystems have been designed and developed so that they can address data-related issues in a holistic, extensible and easy-to-use manner. The ultimate goal of the SELIS approach is to offer all the aforementioned functionalities either as a service through a cloud deployment of choice, or as a package that can installed on-premise with minimal effort while allowing fast integration with existing systems. Thus, a centralized "data lake" will be available where information flows from different sources in a real-time and secure manner and useful insights are extracted enabling the logistic chain stakeholders to perform accurate, fast and optimal decisions while increasing their efficiency.

In this paper we present the SELIS Big Data Analytics and Machine Learning subsystem (SELIS *BDA*), an open-source[2], innovative, cloud-enabled, elastic Big Data framework aiming to tackle challenges stemming from the logistics landscape. The *BDA* is the driving force of the SELIS technical offering, which implements the concept of the SCN as introduced in [20] and includes, besides the *BDA* itself, tools and subsystems that enable external system integration, message exchange, administration and monitoring, decision support services *etc.* The main contributions we make in this paper are the following:

- We introduce business agnostic data and execution models which can fit the needs of the majority of the encountered cases within the logistics domain. The proposed models are the result of a thorough study and analysis of existing use cases within the SELIS project covering more than 40 partners from the logistics and ICT domains.
- We present the architectural design of a system that supports both streaming and batch data ingestion and task execution and implements the proposed models. Additionally, we thoroughly discuss the rationale behind our design choices.
- We implement the proposed system using distributed, open-source, big-data-enabled, state-of-the-art software. Our solution is easily deployable to the cloud using containerization technologies while it offers an easy to use REST API. Although our implementation prototype is based on specific widely adopted frameworks and technologies (*e.g.*, Spark, HBase, PostgreSQL), it remains modular and is written in a way which makes it extensible with little programming effort.
- We conduct experiments to show that i) the SELIS *BDA* induces very low overhead to the tasks it is required to execute and ii) that it can horizontally scale, thus increasing its overall throughput when its infrastructural resources increase.

The structure of the rest of this paper is as follows. In Section II we discuss some of the use cases we worked with in the scope of the SELIS project, highlighting their common characteristics which lead us to a common design. In Section III we present business agnostic data and execution models based on the use cases. In Section IV we discuss technical details on the implementation of our prototype system as well as on its deployment. Section V presents and discusses the experimental evaluation of the SELIS *BDA*. Section VI briefly presents existing commercial solutions and discusses the differences between their approach. Section VII concludes this work with a summary where we expose our final remarks and discuss some of the most interesting challenges to be addressed in the future.

II. Use Cases

In this section, we present actual business use cases from the logistics domain that we faced within the scope of the SELIS project. While our system was designed taking into account several more use cases, some of the most representative are presented in this section. In the next few paragraphs we will highlight the common patterns and properties which multiple use cases share and have guided our system design choices.

- In one of the use cases, we worked with incoming data flows from railroad networks provided by Adria Kombi[3], an intermodal logistics operator in South Eastern Europe. Events related to trains, wagons and containers (*i.e.*, arrivals to or departures from specific stations, GPS location updates *etc.*) are modelled as an incoming flow of messages. Additionally, a static dataset including station locations, train, wagon and container properties *etc.* has been provided. Among the calculation of various statistical metrics, the main requirement here has been to accurately estimate the time of arrival of a train, wagon or container to their terminal stations, based on historical data and the incoming data stream.
- A similar use case demanded the estimation of the time of arrival for trucks delivering goods for a Greek logistics

[1] http://www.selisproject.eu

[2] https://github.com/selisproject/bda

[3] https://www.adriakombi.si/

company SARMED[4]. The Estimated Time of Arrival (ETA) is calculated using an incoming data stream of events declaring the truck's GPS location and the prediction was updated every time an update on the truck's location was received. Static datasets containing information on the trucks, the various warehouses as well as drop off points were separately provided. Moreover, for the same use case, a recommendation service was built upon arrival of order delivery requests, automatically offering the task to the most suitable freelance transportation agents available, taking into account proximity, shipment properties (*e.g.*, frozen/temperature sensitive cargo) and service quality.

- A third use case involves container transportation through a canal network using barges in the Netherlands sailing to and from the Port of Rotterdam[5] – the largest port in Europe. In this case, static data including information about the containers, barges, deep sea vessels and terminals involved were provided. The movement of vessels which are involved in the use case, captured through the dedicated automatic identification system (AIS) which every vessel is equipped with, is reported periodically (every few seconds) to *BDA*. Using the data available, the execution of analytical workloads and the calculation of reliability metrics were requested by the SELIS business partners.
- In the final use case we encountered the problem of automating the ordering forecast process of SONAE[6] – one of the largest retailers in Portugal. In this case, incoming streams of sales forecasts arrive, produced by SONAE's proprietary legacy systems. Additionally, the *BDA* has access to the stock levels of all warehouses. Combining the two data sources and taking into consideration specific rules and restrictions which apply to the transportation of goods from the suppliers' premises to SONAE's warehouses according to an agreement between the two parties, the *BDA* produced order forecasts for the following days.

Anonymized versions of the datasets that were provided by business partners Adria Kombi and SONAE have been made available through the *zenodo.org* platform[7] per the EU directives and initiatives to promote research via Open Access to Data. A collection of AIS data is available seperately[8].

The vision of the SELIS project is that for each use case, a dedicated SCN is set up, configured and made available to the business stakeholders. From a technical point of view, the consolidated modelling of such diverse problems using a common framework has been a great design challenge. To address it, we exploited the observation that the logistics business operation is largely based on message exchanges. These messages could possibly describe events, status updates or requests involving entities which are active in their respective ecosystems. Exchanged messages modify the *current state* of these ecosystems and the transient properties of their entities (such as the location of vessels or vehicles). Additionally, such messages typically arrive at a very fast rate (GPS location updates can be produced as often as once every two or three seconds for every individual vehicle or vessel). On the contrary, most of the properties of the entities which appear as actors in our use cases are static or change rarely, if ever at all. This kind of static data is typically accessed for read purposes only. Taking this top-down approach has allowed us to envision, design and implement a system with very specific characteristics which can fit to most of the problems that can be encountered in the business environment of the logistics domain. Based on these observations, we introduce the *BDA* by describing its architecture in detail in Section III.

[4]http://www.sarmed.gr/
[5]https://www.portofrotterdam.com/en
[6]https://www.sonae.pt/en/
[7]https://zenodo.org/record/3248918
[8]https://zenodo.org/record/3342090

III. Architecture

In this section, we present the architecture of our system. Initially, we present the various modules which the *BDA* consists of. The next subsection introduces our data and execution models which can fit any single use case we have encountered in a consolidated manner. Finally, we briefly discuss how our system can support multiple use cases by using multiple SCN instances and managing the respective metadata securely and persistently.

A. System Overview

The *BDA* is organized in modules which are responsible for its different operations, namely the *Datastore*, *Analytics/ML*, *Connector* and *Controller*. These modules are wrappers which manage internal sub-systems operating as data warehouses, execution engines, etc. The *BDA* architecture is depicted in Figure 1 and is organized in the following modules:

- ***Datastore:*** This module is a wrapper for Storage Engines. Specifically, it is used to interact with them for i) configuring SCNs, ii) data ingestion of streaming incoming messages and iii) data retrieval. Each of the following submodules contacts the *Datastore*, when it needs to access data.

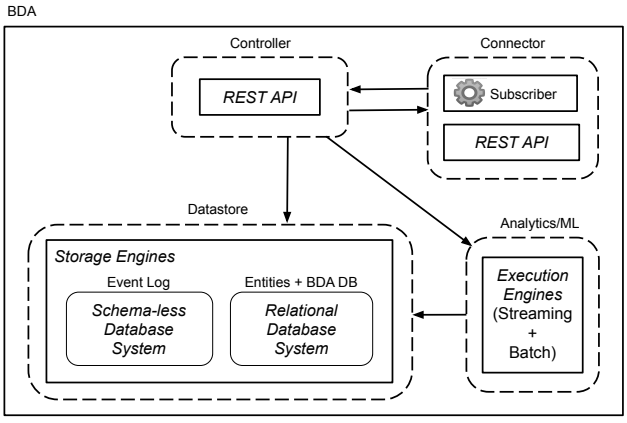


Fig. 1: *BDA* architecture

- ***Analytics/ML:*** This module is a wrapper around execution engines. It is responsible for the execution of any type of computations, either batch or streaming, on the execution engines, by serving them with the relevant data retrieved from the *Datastore* module.
- ***Connector:*** This module acts as the data "gateway" for the *BDA* as it receives messages from message exchange pub/sub systems that the *BDA* may be connected to. It is able to parse and validate any incoming messages, and forward them to the *Controller*. Note that it is an optional part of our system, since data can be send to the *BDA* using the *Controller*'s *REST API*.
- ***Controller:*** This module is responsible for handling the three aforementioned modules. Specifically, it informs the Connector about the messages that it must listen to. It then handles any incoming messages passing them to the *Datastore* module for ingestion in the Storage Engines. Moreover, it triggers the execution of any computations related to the incoming message by calling the *Analytics/ML* module and it is also able to schedule computations to be performed periodically. Finally, it provides a *REST API* which can be used to access any of the *BDA*'s functionalities.

B. Data & Execution Model

Data Model. According to the SELIS vision, each distinct logistics ecosystem is served by a dedicated SCN instance. As discussed in Section II, the sample use cases stated show us that the data which the *BDA* handles can be categorized in i) those describing entities in a use case/SCN – mainly static, or ii) those describing dynamic events which arrive in the form of messages in real-time. The data schema we have derived to model logistics use cases has a business agnostic representation of a star schema [21]. Therefore, it consists of a fact table that we will refer to as "Event Log". It can be considered as an append-only log that captures messages flowing into the *BDA*. Additionally, our star schema includes smaller dimension tables that contain further static information about participating "Entities" in the ecosystem. The "Entity" tables contain what is essentially the static master data of a supply-chain. Figure 2 provides a visual representation of the star schema as employed in the *BDA* Datastore.

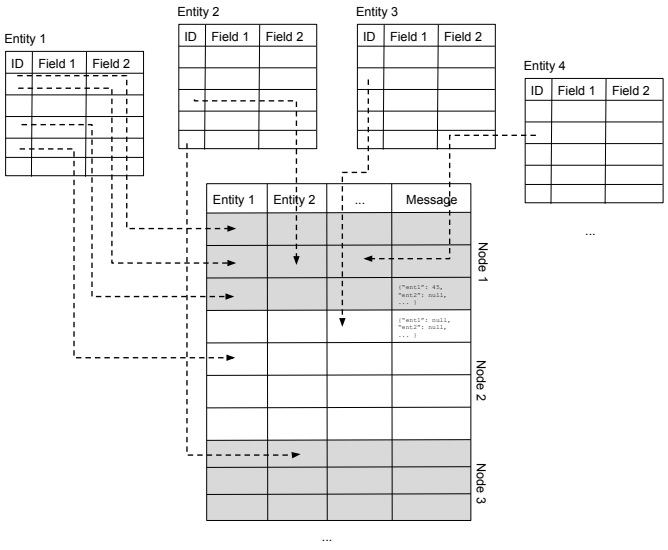


Fig. 2: Event Log and Entites storage using star schema.

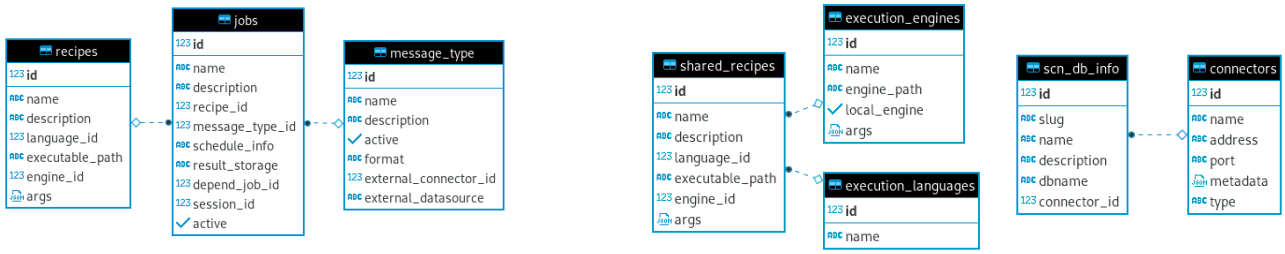
While this schema is business agnostic, it should constantly be aware of the participating Entities as they are considered critical information for the function of the SCN, since the messages flowing into the system and describing events in a real-world logistics business often refer to these entities. For this purpose, the Entities need to be first of all populated during the *SCN configuration* process, that we will explain in Section III-C. This information rarely changes and should be easily referenced and queried.

While the Entity related data are considered static, every message (*i.e.*, event) that the *BDA* receives is stored in the Event Log table. Messages are appended to the Event Log as a new row and might refer to one or more existing Entities. The messages that the BDA must subscribe to and ingest are also defined during the *SCN configuration*. In particular, each message is defined as a `message_type` resource, which contains the message name (topic) and its format. This information is used by the BDA as a message identifier.

Execution Model. *BDA*'s execution model supports two computation modes, streaming and batch. Its streaming execution engines can execute fast computations, such as machine learning inference tasks upon message arrival. On the other hand, batch execution engines can periodically execute heavier computational workloads, such as machine learning model training or statistical workloads on huge chunks of historical data. To support this functionality, the Execution Model of the *BDA*, introduces the following resources:

- `recipe`: This resource refers to an executable program written in a language supported by the *BDA*'s execution engines. A `recipe` can be applied on specific types of data as an *operator*. The `recipe` resource contains meta-information regarding the location of the executable, the execution engine it will use, the type of data it can consume *etc.* which all need to be defined when the `recipe` resource is created.
- `job`: According to our execution model, `recipe` resources need to be linked with an *execution trigger* in order to be used. There exist two different sorts of `jobs`: i) Streaming `jobs`, which are launched upon the arrival of a message of specific `message_type` on one of the supported streaming execution engines (*e.g.*, launching the ETA calculation for a vehicle every time an update for its location is received). ii) Batch `jobs`, which are launched periodically with a user-defined period on one of the supported batch execution engines (*e.g.*, estimating the greenhouse gas emissions for a fleet of vehicles for the past 10 days every night).

Additionally, the *BDA* supports "workflows" consisting of sequential tasks. A workflow in general is considered as a Directed Acyclic Graph of tasks that exchange data between them. For example, let `job` X be defined as dependent on `job`



(a) SCN Metadata.

(b) BDA metadata.

Fig. 3: Metadata Schemas

Y. In such case, `job` X uses the output of `job` Y as input and these two `jobs` are part of a workflow. This feature can be used to create workflows which include many `recipes` that must be executed in sequence.

C. The SCN in the BDA

The SELIS technical offering is a suite of software tools and sub-systems which implement the conceptual idea of the SCN. After careful thought and consideration of all available options and impacting parameters – from technical aspects such as deployment to commercialization strategies, it was decided that the *BDA* would be implemented as an underlying framework upon which multiple SCN instances can be deployed and accessed following the Software-as-a-Service multi-tenant paradigm. This decision has had a heavy impact on some the design of the *BDA*. More specifically, to support multiple SCN instances, a separate star schema is created for each SCN in an isolated database. Dedicated database tables are created storing SCN-specific meta-information regarding the resources that are defined during the *SCN configuration* process[9] which are required for the operation of the *BDA*. Those resources are stored at respective tables per SCN. We will refer to these resources as *SCN metadata*. *SCN metadata* tables are only accessible as a resource from within the respective SCN. Finally, a few more tables are used, persistently storing information regarding the state and contents of *BDA* itself. Metadata on common resources such as `shared_recipes` (*i.e.*, `recipes` which are offered out-of-the-box by the *BDA* and are available for all SCN instances to use), supported `execution_engines` and `execution_languages` as well as instantiated Connector and SCNs are stored in these tables. The schema of these *BDA metadata* tables is presented in Figure 3. The *BDA metadata* database is accessible by all SCNs.

IV. BDA Prototype

A. Technologies

In this section we describe in detail the prototype implementation of the *BDA* and its storage and execution engines. Our implementation choices are based on the architectural design presented in Section III. We also provide an overview of the open-source technologies we use for storage and execution. The code developed for the BDA modules will also be open-sourced[10] by the end of the SELIS project.

1) Controller: Taking into consideration its important role inside the *BDA*, the Controller is implemented as a Java interface that is deployed as a stateless Java-based REST service that runs in a Jetty HTTP Web Server [22]. We use multiple server instances which are accessible through a proxy to achieve load balancing, scalability as well as high-availability. We note here that the Controller's REST service also includes the interfaces of the Datastore and Analytics/ML modules and is therefore the main "gateway" for interacting with the *BDA*. Since multiple users with different roles from various organizations can have access to the *BDA*'s *REST API*, its endpoints are secured using Keycloak [23], a state-of-the-art Identity and Access Management (IAM) tool. Keycloak supports the OpenId-Connect Protocol [24], which is a layer over the OAuth 2.0 [25] Protocol for authorization. Thus, user authentication and authorization is performed before service access.

2) Connector: The Connector module is an optional part of our system that can be used to connect the BDA with an external pub/sub system. It is also implemented as a Java interface that is deployed as a separate stateless Java-based REST service offered through a Jetty HTTP Web Server. The Controller interacts with the Connector through the REST service to define the message types (or topics) it must subscribe to. This module is initially configured to launch a predefined standalone Pub/Sub subscriber instance (see Fig. 1). Taking into account the requirements for load balancing and scalability in terms of the message insertion throughput, we do not use a single Connector instance but instead multiple Connectors can be created and each SCN can have its own. In Figure 4 we can see how multiple Connector instances are used by the *BDA*. Each SCN has its own subscriber which is connected with a specific Pub/Sub server, but generally the same Connector (and Pub/Sub server) can be used by multiple SCNs or the same Pub/Sub server can be used by multiple Connectors.

3) Datastore: The Datastore module is implemented as a Java interface that is exposed through the Controller's service. The Datastore uses predefined system connectors to interact

[9] *SCN configuration* is the process where a user, guided through a user interface defines and initializes important SCN resources such as `message_types`, `recipes`, `jobs` and populates *Entity tables*

[10] https://github.com/selisproject/bda

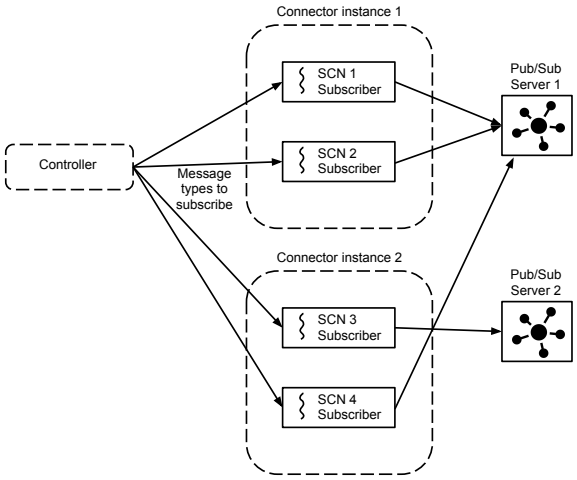


Fig. 4: Connectors inside the BDA

with the underlying storage engines that include the EventLog, the Entities and the BDA database, as shown in Figure 1. The Event Log, as an append-only structure that constantly grows can be stored in a distributed database or a distributed file system. The Entity tables on the other hand, contain limited and static information which may be accessed very often so a replicated RDBMS solution or an in-memory database are more suitable systems. Taking into consideration the aforementioned, we use a distributed NoSQL Apache HBase [26] database to store the Event Log table which offers out-of-the-box scalability and high-availability features. In HBase data is stored in a semi-structured and sparse format over HDFS [27] thus allowing us to adopt any data schema that a use case requires in an efficient way. User access can be controlled on a database, table or even file system folder level. A different Namespace is created for each SCN.

For the Entity tables, on the other hand, we used a more appropriate replicated setup of a PostgreSQL [28] database. Essentially, we create multiple instances of the same database using master-slave streaming data replication. One instance is considered as the master server that handles write requests and the slaves only allow read access to the data that they contain. Moreover, by using a proxy layer for the slave instances we achieve load-balancing and scalability for the read throughput when accessing the Entity tables. A slave will be turned into a master in case the primary master server fails providing high-availability. As a classic RDMB system, PostgreSQL also offers adoptable database schema and user roles and permissions. Each SCN has its own database created inside this replicated setup, where the metadata of each SCN are also stored using a separate database schema.

Regarding the BDA database, since the metadata it stores contain limited amount of information, we place it in the same PostgreSQL setup.

4) Analytics/ML: Similarly, the Analytics/ML module is implemented as a Java interface that is exposed through the Controller's service. The module uses predefined system connectors to interact with the underlying execution engines. Given that the batch calculations can be performed on a large part of the historical data, the batch execution engine is selected as a scalable distributed Big Data processing system. The fast streaming calculations on the latest data require at the same time a streaming engine with the ability to store data in memory in order to process them instantly with a scalable and distributed in-memory processing system. Taking into account the aforementioned, we deploy a Apache Spark cluster [8] that offers both offline and online processing (using interactive sessions) and out-of-the-box scalability and high-availability features. Spark supports many programming interfaces such as Python, Java and Scala so users can easily create their own analytics `recipes`, and it also provides a standard SQL interface where even simpler analytics can be programmed. Spark also provides a built-in Machine Learning library that contains most of the state-of-the-art machine learning algorithms that can be used in our use cases as the black-box executables in order to create and train a model.

In order to exploit Spark's interactive sessions for fast streaming calculations we use Apache Livy [29] that offers a REST interface to facilitate programmatic and fault-tolerant submission of both interactive and batch Spark jobs. Multiple long-running Spark Contexts can be managed by Livy, offering concurrency and each Context is identified using its own session id. The same running Spark Context can be used by multiple Spark jobs. In this way RDDs and Dataframes can be shared across multiple jobs (RDDs are described in Sec. 2.1 of [30] whereas Dataframes in Sec. 3 of [31]). Fault-tolerance is also provided as session persistence and recovery is supported in a state store like Zookeeper [32]. Finally, user authentication is supported. By configuring the Analytics/ML module to use the Livy service in order to communicate with Spark's execution engine, we are able to easily create a different 'live' session for every streaming (message-triggered) job. Moreover, the jobs that are part of a workflow are launched on the same 'live' session one after the other. Finally, we use Hadoop YARN [33] as a cluster manager, which Livy uses in order to commit physical resources for the Spark jobs and then run them.

It is also important to notice that when a `recipe` is being executed it accesses only the Datastore's data that the Analytics/ML module allows. In order to ensure this, we use Livy's capability to run interactive sessions and we create for each job an interactive Spark session where we pre-load the required input data and models from the Datastore as DataFrames using our own Spark-Storage Engine connectors that are provided by the module as a library (depending on the Execution Engine and the Execution Language). Then, the `recipe` is launched on this session and receives as input arguments the predefined DataFrames, so in this way it has no access to other data of the Datastore or directly to the Datastore itself. Similarly, the output data of a `recipe` are saved before closing the Spark session using the appropriate connector that the Analytics/ML module defines (depending also on the selected output location for the job). The Analytics/ML module employs code generation for the aforementioned procedure. In the case of workflows of jobs that are executed

on the same Spark session, the output data of one job are saved as we previously explained and the session remains open so that the output data are available for use from the next job without the need to transfer them from the Datastore.

Currently, for the prototype that we have created, Python is the only supported Execution Language for `recipes` and the Analytics/ML module uses code generation with a library of Spark connectors implemented in Python. In particular, the library contains a Spark-HBase connector, a Spark-PostgreSQL connector, a simple PostgreSQL JDBC connector and a Publisher implementation. In the future other languages like Java and R could also be added depending on the use case requirements. The Execution Engines catalog of the BDA subsystem contains Apache Spark for executing PySpark code, and also a local engine which is offered for executing simple (Python) code with small input datasets locally on the Controller. A second Python library is available for the local Execution Engine which contains connectors to retrieve data from a storage engine, save them locally in text files and use them as input for the `recipe`. Similarly, output data are saved in a file which is then handled by the library.

B. API

The BDA offers a REST API which is implemented to allow easy interaction with its storage and execution layers. The implemented interfaces can connect to different storage, execution and Pub/Sub systems using the appropriate internal connector defined during its initial configuration. From the user's point of view however, one can access BDA data or process them without requiring prior knowledge of the underlying systems themselves to be able to communicate with them. The procedure in order to define a new SCN, initialize its data model, and configure it in order to receive a message and execute a `recipe` is performed with the following simple REST calls (we enumerate the API calls made for the SONAE use-case described in Sec II):

1) *POST /api/datastore*: register a new SCN with the specified scnSlug, create its database with the specified name, create the Metadata schema and connect with the specified Pub/Sub server.
2) *POST /api/datastore/scnSlug/boot*: create Entities tables and populate them with data. The posted JSON has the following format:

```
{"tables":[{
   "name": "items",
   "schema":{
     "columnNames":["sku","supplierpacksize","
         businessunit"],
     "columnTypes":[{
       "key":"sku",
       "value":"character varying(100) NOT NULL"
           },{
       "key":"supplierpacksize",
       "value":"integer"},{
       "key":"businessunit",
       "value":"integer"}],
     "primaryKey":"sku"},
   "data":[{
   "tuple": [{
     "key": "sku",
       "value": "2296118"},{
       "key": "supplierpacksize",
       "value": "4"},{
       "key": "businessunit",
       "value": "1203"}]},{
    "tuple": ..}]},
    ...
]}
```

Listing 1: POST data in JSON format for inserting the Master Data.

Since an RDMB system is used for the Entities tables, the *"columnTypes"* field can contain as values any SQL compliant data types.

3) *POST /api/messages/scnSlug*: define a new Pub/Sub message format and the subscription fields and subscribe to it.
4) *POST /api/recipes/scnSlug*: define a new `recipe`, its language, arguments and execution engine.
5) *POST /api/recipes/scnSlug/recipeId/executable*: Upload the `recipe` executable which is written in the following form:

```
def run(sparkContext, items_dataframe):
   result = items_dataframe
            .groupBy("businessunit")
            .agg({"sku":"count"})
            .toJSON().collect()
   return result
```

Listing 2: PySpark code to compute the number of skus per businessunit.

6) *POST /api/jobs/scnSlug*: create a new job either periodic or message-triggered that launches the specified `recipe` and specify the output location.

It is clear in the example that we presented, that the user does not need to use database-specific drivers to load data to the Entities tables or to fetch data inside the `recipe`. In particular, the `recipe` contains a simple run() method that takes as arguments the Spark Context and the dataframes that contain the required input data and returns the final result as a JSON. The input dataframes are created by the BDA taking into account the `recipe` arguments, where the user defines, among others, the EventLog message types and the Entities tables that contain the required input data. The BDA uses code generation to load data from the storage engines to the Spark execution engine before running the user's `recipe`. Similarly, code generation is used to either save the job result to the KPI db or publish it to the Pub/Sub according to the job configuration. The code generated in the previous example is the following (assuming the job result is selected to be published as a message):

```
import RecipeDataLoader
import userRecipe
items = RecipeDataLoader.fetch_from_master_data(
    spark, "jdbc://hostname:5432/scnDBname", "
    scnDBusername", "scnDBpassword", "items")
result = userRecipe.run(spark, items)
RecipeDataLoader.publish_result("SCNpubSubAddress",
    "SCNpubSubPort", "PubSubCert", "scn_slug", "
    userRecipe_jobId", result)
```

Listing 3: PySpark code generated to run the `recipe`.

In this case the *fetch_from_master_data()* and *publish_result()* functions contain: the PySpark code in order to load data from a PostgreSQL table into a Spark dataframe, and the Pub/Sub client Python code to publish the result respectively.

The user can therefore perform any action with the BDA using simple REST calls and with basic knowledge of Python and PySpark commands for our current implementation.

Finally, in case different execution languages or storage and execution engines are integrated into the BDA, the BDA code can be extended with little programming effort as we previously mentioned to support them. In particular, when a new engine is integrated, either for storage or execution, it is required to create the relevant engine connector to be used by the BDA. In case of changes in the execution environment, either the language or the engine, it is required to create or extend the library (RecipeDataLoader in the previous example) that will be used to handle data and extend the code generation logic to create the appropriate data models using this library. In any case the user interface remains the same as it is agnostic to the underlying system/technologies, and the `recipe` generic format will remain the same too. Of course, when changing the execution environment, the appropriate execution language and engine-specific operators must be used inside the `recipe`.

C. Cloud Deploymnet

The implementation of the *BDA* is based on established Big Data technologies that provide us with many desired properties like scalability, fault-tolerance, etc. However, deploying such technologies on the cloud as part of a bigger system, such as the *BDA*, and configuring their inter-connections is a challenging task. To this end we choose to i) containerize all the *BDA*'s components and deploy them on the cloud as containers and ii) use a container orchestrator to handle all the aspects related to deploying container based micro-service applications.

We use Docker [12] as container technology and create Docker images based on Dockerfiles for each one of the *BDA*'s components. As container orchestrator, we choose Kubernetes [14] since it is one of the most widely used and feature rich orchestrators available. Through the use of Helm charts [34] we can automatically deploy the SELIS *BDA* on a given Kubernetes cluster, manage the connections between different containers, handle container failures, easily scale different components of the *BDA* as well as upgrade containers individually.

While in a Kubernetes cluster containers are ephemeral, some of the *BDA*'s components should keep state. Consider, as an example, the Postgresql container that stores the "Master Data" as mentioned in Section III-B. If the container crashes it will be restarted by Kubernetes, the data however will be lost. To achieve persistent storage for the components that require it, we use Kubernetes Persistent Volume Claims (PVC). In our prototype implementation we issue PVCs through Kubernetes to a Ceph storage cluster in order to provide persistent storage to containers.

V. Experiments

In this section we present the results of our experimental evaluation of the SELIS *BDA*. Through the experiments conducted we are aiming at proving that i) while our system orchestrates the operation of various interconnected sub-components, it adds minimal overhead to the core operations of data ingestion and processing workload execution, and ii) our system's components can individually scale horizontally, thus also making it scalable as a whole.

A. BDA Overhead

In the first of our experiments, we aim to prove that the overhead of the orchestration of the various sub-components which are abstracted by the *BDA* is minimal. We set up a SELIS Community Node and simulate the workflow of messages triggering a data processing task upon being published (*i.e.*, pushed) into the *BDA*– it should also be taken into account that all incoming messages are by default persistently stored. We execute this experiment twice, modifying the message size: i) the first time, sending a minimal message ii) the second time, using real messages which correspond to one of the use cases we came across in the SELIS project and record the overhead which the *BDA* adds to the total workflow execution time.

According to the specifications of our implementation, JSON formatted incoming messages are supported. To prove that our implementation adds negligible overhead to the system, we start by dispatching a message containing only a minimal payload. In total, this message takes up about 100 bytes in size. We continue the experimentation useing real messages. In this case, messages contain sales forecast information provided for the SONAE use case and include reports of the retailer's expected sales of products of a specific producer/supplier stored in a specific warehouse. Some of the suppliers have a wide range of products to their name, while others have only a few. Additionally some of the retailers supply more than a single warehouse. The messages we have used for this experiment include two *indexed attributes* – acting as foreign keys to the supplier and product Entity tables – and a deserialized JSON array type *payload*. We conduct the experiment using 3 different message sizes ($\approx$1 kilobyte, $\approx$10 kilobyte, $\approx$100 kilobyte). In all cases, we repeat the experiment several hundred times and use the average metrics. To quantify the overhead of the *BDA*, we record timestamps at 5 different progress checkpoints of the workflow: i) upon message arrival, ii) before it is permanently stored into the Event Log, iii) after it is successfully stored, iv) before the processing task it is associated with is executed and finally, v) after task execution has finished. As described in Section IV, the permanent storage where the Event Log resides is an HBase NoSQL database, whereas the default execution engine employed is Apache Spark. For the scope of this experiment we are not interested in the time the HBase insertion or Spark streaming execution require but only on the overhead of *BDA*.

1) Experimental Setup: For this experiment we deployed an instance of the *BDA* on Docker containers (a total of 10 containers) running on a single Openstack VM with 4 virtual

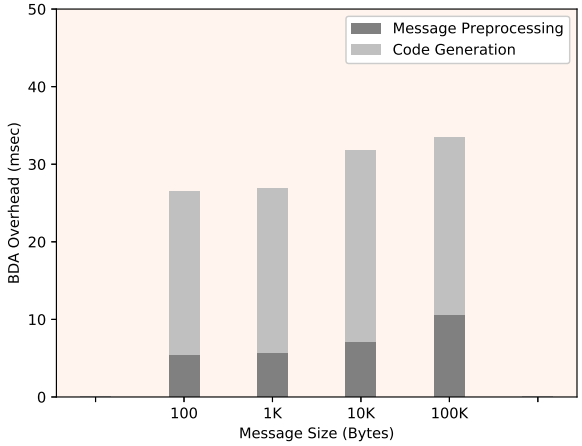


Fig. 5: *BDA* Overhead - Overhead per message size

cores and 8GB of RAM. We perform HTTP requests which post identical messages to the *BDA*, one at a time, and wait until the workflow execution has finished before we repeat to ensure there is no interference between repetitions. As stated in the previous paragraph, messages are saved in the Event Log and consequently trigger a processing task emulating a typical *BDA* workflow, meanwhile recording the checkpoint timestamps and report on average time spent per execution stage where the BDA is involved excluding the impact of HBase data insertion and Spark data processing.

2) Results: We present the findings of our first experiment in Figure 5. We have split the time spent in the *BDA* in two phases, *message preprocessing*, which happens before the insertion, and *code generation*, which happens before the processing task execution, as described in IV. It is noticeable that despite the varying in message sizes, the *BDA* is able to handle the data ingestion and task execution workflow by only adding sub-second overhead to set up and execute the distinct sub-tasks which involve various sub-components and underlying systems.

B. BDA *Scalability*

In this section we present experimental results which show that the SELIS *BDA* can scale horizontally. Since the *BDA* is comprised of a number of components it suffices to show that each of our system's components can scale individually. If so by consequence the entire system can be considered scalable.

As we mention in Section III-B the *BDA* has two main functions: data storage and computation. For storage we use an HBase cluster and a PostgreSQL database. We refer to existing literature [35] that verifies that an HBase cluster can support massive volumes of data and high insertion rates by scaling linearly with the number of region-servers added. HBase is highly available through the use of Zookeeper and fault-tolerant being based on HDFS. As a result we argue that basing our storage layer in HBase allows us to handle massive volumes of data and scale horizontally if required. PostgreSQL on the other hand follows the master-slave architecture and therefore scales vertically. Although, in theory, that could be considered a bottleneck of our system, we use PostgreSQL to store mostly metadata which we expect to change in very slow rates. Consequently we consider sufficient the horizontal scaling we achieve, only for reads, by adding more slave servers and enabling reads from the slaves. Considering computation, the *BDA* uses a Spark cluster to execute both streaming and batch jobs. Again, we refer to existing literature [30] that supports that Spark can handle batch jobs at any scale as well as streaming computations based on micro-batching.

Consequently, to prove that the SELIS *BDA* is scalable horizontally we show that the *BDA* controller, the component that acts as the control hub of the whole system, can also scale. By doing so we prove that we can achieve horizontal scalability for all the system's components and as a result for the system as a whole. Since the *BDA* controller is a stateless server it can scale seamlessly by introducing a load balancer and distributing requests to a configurable number of *BDA* controller servers.

1) Experimental Setup: We demonstrate the *BDA* controller's scalability by running a data ingestion workload. We perform HTTP requests that post messages to the controller, the messages are processed and then saved in the HBase cluster emulating a typical data ingestion workflow. We conduct this experiment on a SELIS *BDA* deployment on a Kubernetes cluster. Kubernetes is deployed on 8 Openstack VMs spanning 4 physical hosts. Each VM has 4 virtual CPUs running at 2.1GHz and 8Gb RAM. Since we choose a data ingestion workload to measure throughput we use 12 HBase region-server containers to ensure that HBase does not become a bottleneck. We use an Nginx based container as the load balancer of a Kubernetes *BDA* controller Service. We vary the *BDA* controller containers from 1 to 16. We create a constant load to the service using *wrk* [36] running on a different container. We spawn 10 threads maintaining a total of 500 open connections and perform requests continuously. We measure requests-per-second for different numbers of *BDA* controller containers.

2) Results: In Figure 6 we present results for a data ingestion workload. As we can see the SELIS *BDA* service can scale horizontally with the number of containers achieving thousands of requests per second.

VI. Existing Approaches and State of the Art

In this section we will briefly present some of the most popular and successful Supply Chain Analytics software suites which are used in the industry. It is important to mention that while all the following are proprietary systems, our prototype will be available open-source by the time the project finishes. Some of the most widely used products in the market targetting the logistics domain are the Supply Chain Management Software – developed by SAP[11], the JDA Supply Cain Analytics platform[12], IBM Cognos[13] *etc.* These products, similar to

[11]https://www.sap.com/products/digital-supply-chain/scm.html
[12]https://jda.com/solutions
[13]https://www.ibm.com/products/cognos-analytics

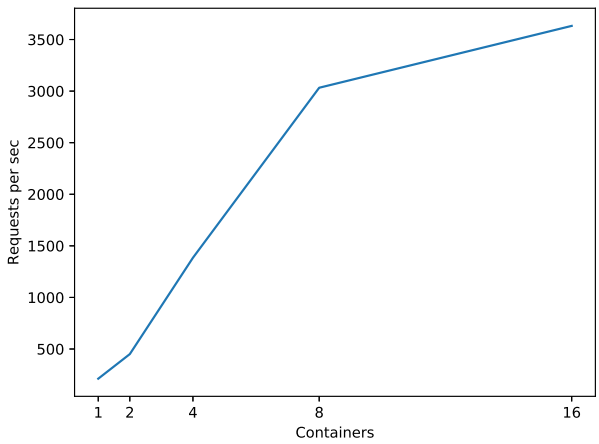


Fig. 6: Data Ingestion - Requests per second

our solution, support cloud deployments. However, with them being offered as closed-source software, we didn't have the opportunity to study their internal operation and architecture. Part of the success these systems enjoy is due to the wide range of features they are packaged with. Our approach, on the contrary, introduces a very solid framework with well-defined architecture that can be used to build functionality as required per use case. The tools we offer can be utilized to cover a very wide area of different use cases in the logistics domain. Indeed, within the scope of the various SELIS use cases, a variety of supportive tools ranging from administration dashboards to custom `recipes` have been developed to support the *BDA* and provide a high quality service suite.

Overall, our suggested solution offers an easily deployable platform supporting storage and processing at scale which is both open-source itself and based on other open-source tools and technologies. Unlike existing commercial products, it is modular – allowing expert users to exchange any of its subcomponents with different systems offering similar functionality (*e.g.*, mySQL instead of PostgreSQL). *BDA*'s functionality is also extensible, as users can contribute to the existing, out-of-the-box `recipe` pool with code of their own.

VII. Conclusions

In this paper we presented the SELIS BDA, the open-source, cloud enabled technical offering for Big-Data analytics of SELIS, an EU funded mega-research project in the area of logistics. We have briefly described the main use-cases that we examined in order to design its architecture. We have presented its architectural components, the generic data and execution models design and the specific open-source technologies that we utilized for its implementation. We experimentally measured its scalability and overhead in a large cloud-enabled cluster where we showed that it can scale to a large number of concurrent requests while posing a minimal overhead.

Acknowledgment

This paper is supported by European Union's Horizon 2020 RIA programme under GA No 690588, project SELIS.

References

[1] M. A. Waller and S. E. Fawcett, "Big data, analytics, and the changing face of supply chain management - smartdata collective." *J. Bus. Logist.*, vol. 34, pp. 77–84, 2013.
[2] "Big data, analytics, and the changing face of supply chain management - smartdata collective." [Online]. Available: http://www.smartdatacollective.com/big-data-analytics-and-changing-face-supply-chain-management/
[3] "Third-party logistics study." [Online]. Available: http://3plstudy.com/3pls5.php
[4] "Digital supply networks." [Online]. Available: https://www2.deloitte.com/us/en/pages/operations/solutions/digital-supply-networks.html
[5] S. Coleman, R. Göb, G. Manco, A. Pievatolo, X. Tort-Martorell, and M. S. Reis, "How can SMEs Benefit from Big Data? Challenges and a Path Forward," *Quality and Reliability Engineering International*, vol. 32, no. 6, pp. 2151–2164, 2016.
[6] "Data lake vs data warehouse." [Online]. Available: https://www.kdnuggets.com/2015/09/data-lake-vs-data-warehouse-key-differences.html
[7] "Welcome to apache™ hadoop." [Online]. Available: http://hadoop.apache.org/
[8] "Apache sparktm - lightning-fast cluster computing." [Online]. Available: https://spark.apache.org/
[9] "Amazon emr - amazon web services." [Online]. Available: https://aws.amazon.com/emr/
[10] "Apache spark and apache hadoop on google cloud platform documentation — apache hadoop on google cloud platform." [Online]. Available: https://cloud.google.com/hadoop/
[11] "Hadoop — microsoft azure." [Online]. Available: https://azure.microsoft.com/en-us/solutions/hadoop/
[12] "Docker: Enterprise container platform for high-velocity innovation." [Online]. Available: https://www.docker.com/
[13] "Apache mesos." [Online]. Available: http://mesos.apache.org/
[14] "Kubernetes: Production-grade container orchestration." [Online]. Available: https://kubernetes.io/
[15] "Apache storm." [Online]. Available: http://storm.apache.org/
[16] "Spark streaming — apache spark." [Online]. Available: https://spark.apache.org/streaming/
[17] "Apache kafka." [Online]. Available: http://kafka.apache.org/
[18] P. Carbone *et al.*, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.
[19] "Heron." [Online]. Available: https://twitter.github.io/heron
[20] "D4.1 open selis platform architecture design." [Online]. Available: https://www.selisproject.eu/uploadfiles/Deliverables/D4.1_Open-SELIS-Platform-Architecture.pdf
[21] "Understanding star schemas." [Online]. Available: http://gkmc.utah.edu/ebis_class/2003s/Oracle/DMB26/A73318/schemas.htm
[22] "Eclipse jetty." [Online]. Available: https://www.eclipse.org/jetty/
[23] "Keycloak." [Online]. Available: https://www.keycloak.org/
[24] "Openid-connect — openid." [Online]. Available: https://openid.net/connect/
[25] "Oauth 2.0 — oauth." [Online]. Available: https://oauth.net/2/
[26] "Apache hbase." [Online]. Available: https://hbase.apache.org/
[27] "Apache hadoop dfs." [Online]. Available: https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html
[28] "Postgresql." [Online]. Available: https://www.postgresql.org/
[29] "Apache livy." [Online]. Available: https://livy.incubator.apache.org/
[30] M. Zaharia *et al.*, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.
[31] M. Armbrust *et al.*, "Spark sql: Relational data processing in spark," in *SIGMOD*. ACM, 2015, pp. 1383–1394.
[32] "Apache zookeeper." [Online]. Available: https://zookeeper.apache.org/
[33] "Apache hadoop yarn." [Online]. Available: https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html
[34] "Helm: The package manager for kubernetes." [Online]. Available: https://helm.sh/
[35] I. Konstantinou *et al.*, "On the elasticity of nosql databases over cloud management platforms," in *CIKM*, 2011, pp. 2385–2388.
[36] "wrk: Modern http benchmarking tool." [Online]. Available: https://github.com/wg/wrk