Project Title	High-performance data-centric stack for Big Data applications and operations
Project Acronym	BigDataStack
Grant Agreement No	779747
Instrument	Research and Innovation action
Call	Information and Communication Technologies Call (H2020-ICT-2016-2017)
Start Date of Project	01/01/2018
Duration of Project	36 months
Project Website	http://bigdatastack.eu/

D4.3 – WP4 Scientific Report and Prototype Description – Y3

Work Package	WP4 – Data as a Service: Data Paths Services
Lead Author (Org)	Yosef Moatti (IBM)
Contributing Author(s) (Org)	Stathis Plitsos (Danaos) Paula Ta Shma, Guy Khazma (IBM), Javier López Moratalla, Jacob Roldan, Rogelio Rodriguez (LeanXcale) Luis Tomás Bolívar (RedHat) Marta Patiño, Ainhoa Azqueta (UPM) George Makridis, Christos Doulkeridis, Maria Kanakari, Dimitris Poulopoulos, Giannis Poulakis, Nikitas Sgouros (UPRC) Richard Mccreadie (Glasgow University)
Due Date	30.10.2020
Date	30.12.2020
Version	1.0

Dissemination Level

X	PU: Public (*on-line platform)
	PP: Restricted to other program participants (including the Commission)
	RE: Restricted to a group specified by the consortium (including the Commission)
	CO: Confidential, only for members of the consortium (including the Commission)

The work described in this document has been conducted within the project BigDataStack. This project has received funding from the European Union's Horizon 2020 (H2020) research and innovation programme under the Grant Agreement no 779747. This document does not represent the opinion of the European Union, and the European Union is not responsible for any use that might be made of such content.

Versioning and contribution history

Version	Date	Author	Notes
0.1	13.10.20	Yosef Moatti	First draft chapters 1 to 3 as well as 4.2
0.2	14.10.20		Pavlos' input for chapters 1 2 3 and 4.2 Other improvements from Yosef
0.3	15.10.20	Yosef Moatti	First draft chapters 1 to 4
0.4	18.10.20	Yosef Moatti	Changes from Ainhoa, Pavlos, Christos and George added
0.5	18.10.20	Ainhoa Azqueta	Changes and contributions in Section 4.2
0.6	20.10.20	Yosef Moatti	Improve in the Ingestion flow image
0.9	25.10.20	Yosef Moatti	First integrated version
0.11	26.10.20	LXS	Chapter 6 bumped from version 02 to 03
0.12	27.10.20	Yosef Moatti	Fixes for figure, tables and section numbering
0.13	27.10.20	George Makridis	Fixes regarding analytics
0.14	27.10.20	George Makridis	Additional fixes. All comments and track changes reset
0.15	28.10.20	Richard	Log Search section added
0.16	29.10.20	Chrysostomos Symvoulidis	Review of the current version
0.17	01.11.20	Yosef Moatti	Review + fixes for Chapter 6, Chrysostomos' review & fixes, Fixes of Ainhoa following Yosef' review, New CEP performance Evaluation, Amaryllis' review & fixes, Christos and Richard inputs
0.18	03.11.20	Yosef Moatti	Improvements of Dimitris for Data Quality Assess. Section Various small fixes
0.19	04.11.20	Yosef Moatti	Fix for figure 3 – Data ingestion path
0.20	12.11.20	Yosef Moatti	Handling of Dimos' review received Nov 12th
0.21	13.11.20	Christos Doulkeridis	Moved 9.4 to conclusions of Section 9, based on Dimos' review
0.22	14.11.20	LXS	Handling of Dimos' review
0.23	14.11.20	Ainhoa Azqueta	Handling of Dimos' review
0.24	15.11.20	Dimitris Poulopoulos	Handling of Dimos' review
0.25	16.11.20	LXS	Initial seamless changes
0.26	17.11.20	Yosef Moatti	Review of 0.22 to 0.24 changes
0.27	17.11.20	Yosef Moatti	All Pavlos' seamless track change accepted and then review of them with new track changes
0.27A	19.11.20	Yosef Moatti	Same as 0.27 but with all tracked changes accepted and comment removed.

0.28	23.11.20	Josh Salomon	Review comments addressed by Yosef Moatti
0.29	26.12.20	Pavlos Kranas	Seamless JOINs contributions
1.0	30.12.20	Yosef Moatti	Wide review & final version

Table of Contents

List of tables

List of figures

Acronyms

ACID	Atomicity, Consistency, Isolation, Durability
CEP	Complex Event Processing
DaaS	Data as a Service
GCP	Google Cloud Platform
JDBC	Java Data Base Connectivity
KPI	Key-Performance Indicators
MOC	Mass Open Cloud[1]
OLTP	On Line Transaction Processing
QoS	Quality of Service
SLA	Service-Level Agreement
SLO	Service-Level Objective
UDF	User Defined Function
VM	Virtual Machine

[1] https://massopen.cloud/

1 Executive Summary

The BigDataStack project was conceived as a data centric environment, integrating approaches for Data as a Service (DaaS). The data services of this environment are naturally at the core of BigDataStack and are covered in this deliverable in terms of design specification as well as in terms of integration and experimentation outcomes (i.e. prototypes evaluations). An initial demonstration of the capabilities offered by the data services has been performed during the interim review of the project, in which all the components have been integrated and interacted to demonstrate their added-value and innovations in the scope of the real-time ship management use case.

This third deliverable for WP4 is the natural evolution of the deliverables D4.1 (M11) and D4.2 (M23) and reports of the new technical status of the services. New integrations between them and new usages done in the three Use Case demonstrations previewed for the final review.

On top of the natural evolution of each of the services from M23, the main differences with D4.2 are:

1. The description of the advancements of the "Adaptable Distributed Storage" task which was not demonstrated during the interim review, as it has been mainly developed during the second part of the project, and D4.2 did not report the design of the missing functionalities.
2. The integration of the data service components with the infrastructure layer of BigDataStack (WP3) to tap on the Triple Monitoring and QoS Evaluation component as well as the Dynamic Orchestrator all this to enable the automation of the scale out/in capabilities for both the Real-time Complex Event Processing data service as well as much heavier LeanXcale data base.
3. New interactions and intercommunications of data service components. This includes the usage of the new JOIN capabilities of the Data Skipping technology by the seamless component.
4. Demonstration of new usage of data service components by Use Cases. This includes:
    a. Usage of the Data Skipping capabilities by the Predictive Maintenance stack to enhance performance.
    b. Usage of the Data Quality Assessment component in the Smart Insurance use case.
5. An added "Log Seach" section that pertains to Task 4.5 at section 10, and a detailed experimental evaluation of Process Mining algorithms for data sets with controlled level of noise, different noise types, and different size of process models at Section 9.4
6. An update of the WP4 component architectures to reflect the implementation level that was reached by this deliverable issuance.
7. Experimentation results regarding the performance of the architecture components.

2 Introduction

2.1 Relation to other deliverables

This deliverable is related to other project deliverables as follows:

- D2.3 - Requirements & State of the Art Analysis III (M34). Collected requirements have been analysed to drive the design specifications of data services (top-down approach), while technical requirements from the data services have also been collected and analysed to provide input to other components of the overall architecture (bottom-up approach).
- D2.6 - Conceptual model and Reference architecture III (M30) which extends and details the relevant part of D2.5 (M18).
- D3.3 (Data-driven infrastructure management scientific report) and D5.3 (Dimensioning Modelling and Interaction services) respectively WP3 and WP5 Scientific Reports and Prototype Descriptions (M34). Alongside D4.3, D3.3 and D5.3 present the technical status, of the BigDataStack project by M34.
- D6.2 - Use case description and implementation -Y3 (M34).
- D4.2 - the second WP4 technical deliverable (M23). D4.3 is an evolution of D4.2.

2.2 Document structure

The structure of this deliverable is similar to the structure of D4.2:

Section 3 presents the general architecture and provides an overview of the various components / mechanisms that build up the overall data services block.

Section 4 presents the implementation and experimentation at WP4 level. It provides an overview of the components and how they were and will be used during the interim and final review for the sake of the three data management scenarios. As compared with D4.2 we present in this deliverable the following enhancements:

1. An updated Real-time shipping Management use case with new capabilities and integrations.
2. Updates for each of the sections describing the added capabilities of each for each of the tasks of WP4
    a. The **big data layout and Data Skipping** (T4.2 presented in section 5) with an updated presentation of its extensible Data Skipping capabilities and performance results as well as new capabilities added in the past year
    b. The **Adaptable Distributed Storage** (T4.3 presented in section 6) which presents the design and implementation of the reconfiguration engine that allows for the automated redeployment of data regions among data nodes, under heavy transactional workload, and allows the online scalability of this component

c. **Seamless** (T4.4 presented in section 7) which presents the added SQL JOIN capabilities

d. The **data quality assessment** (T4.1 presented at section 8) demonstrated in the Real-time shipping Management use case during the interim review will be demonstrated in the context of the Smart Insurance use case during the final review. This use case is described in detail in D6.2. In the present deliverable we only report of the new assets of this technology.

e. The work on **predictive & process analytics** has been extended to include well-established process mining algorithms, which have been evaluated in a comparative study to assess their scalability and effectiveness, for different process models and event logs, of variable size, and of varying levels of noise (T4.5 presented in section 9).

f. "Log search" is presented in new section 10. This effort falls under Task 4.5 and presents a real-time tool to quickly explore the compute logs.

g. Finally, at section 11 we present T4.6: **real-time complex event processing** (CEP) task with its new dynamic elastic capabilities.

3 Solution Architecture

3.1 Vision

BigDataStack provides a complete data-driven infrastructure, (see Figure 1). The BigDataStack environment capabilities could not have been realized without a full stack that answers the requirements of Big Data operations and applications. Maybe the simplest illustration is that two critical data services (CEP and the LeanXcale data base) rely on the Data-driven Infrastructure Management layer for realizing their elasticity.

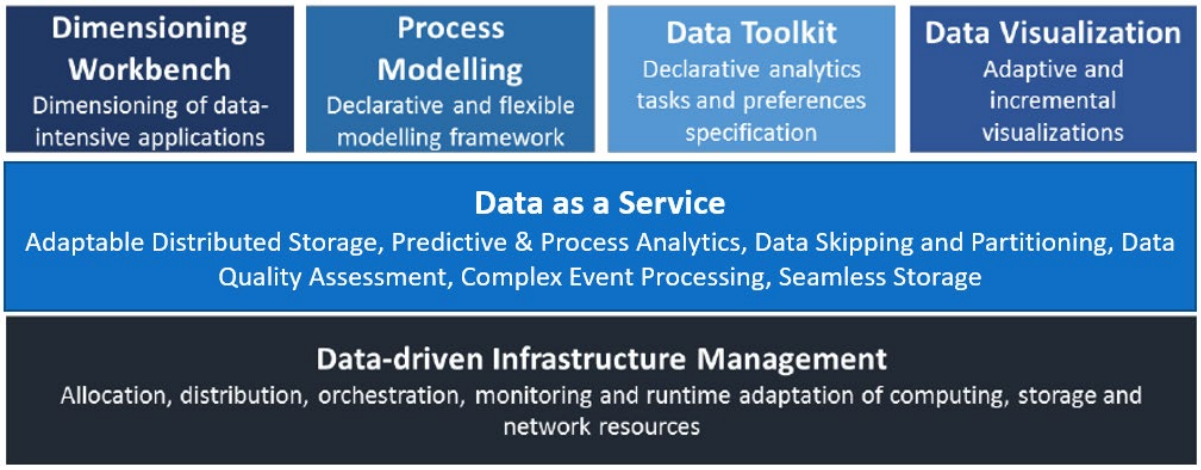


Figure 1: BigDataStack core platform capabilities

The "Data as a Service" layer is the cornerstone on which the four upper platform capabilities of BigDataStack rely. This layer offers a set of services that provide the building blocks of an efficient and modern data infrastructure covering all the major phases of data life cycle and usage, i.e., data ingestion, data storage and data analytics.

Let us now review the main components / mechanisms of the Data as a Service:

Data Quality Assessment & Improvement is an essential part to data ingestion as it offers domain-agnostic data quality assessment and enhancement services. As demonstrated during the interim review, the data quality assessment technology that was developed by UPRC under the umbrella of task T4.1 proved to be a critical prerequisite for the machine learning software that was developed for predicting mechanical faults in the Real-time shipping Management use case. Details can be found in chapter 3 of D6.2.
Furthermore, this technology was extended and will be demonstrated during the final review as part of the Smart Insurance use case demonstration which detailed in chapter 5 of the same deliverable D6.2.

In D4.2 we presented interesting capabilities of the Data Skipping yields technology: it already provided a pluggable data skipping engine with the ability to handle complex queries as well as User Defined Functions (UDF) and this technology had already been

integrated within the IBM SQL Cloud Service and GA[2]-ed in October 2020. During the past year, this technology has been further matured and has been GA-ed for three additional IBM commercial services. We present in this document new capabilities that were added in the past year.

Distributed Storage is one of the data storage approaches of BigDataStack, it is based on the LeanXcale internal key value storage layer. This component has mostly been developed during the second half of the project enhances the capabilities of this layer with:

- data fragmentation of the stored datasets;
- dynamic reconfiguration by splitting, merging and moving the data regions across the distributed data nodes in order to balance themselves against diverse loads (both in terms of incoming work and data load);
- the elastic re-deployment of the storage that will be able to horizontally scale in/out to adapt to lack/surplus of resources.
- Its integration with the Infrastructure Management building block of BigDataStack that will allow the automated provision of additional resources or the release of already reserved resources
- the ability to provide a cost-effective way for the *Seamless Analytical Framework* to drop a data slice that has been moved from the operational datastore to the object store

The second approach of BigDataStack is not Object Storage per se but rather new techniques to overcome the "data ingestion" problem that is typical of Object Storage[3]: BigDataStack presents advances both in data layout (or data partitioning) and data skipping. These advances have already been proved to greatly alleviate the data ingestion problem.

During the interim review we demonstrated the Seamless Data Analytics Framework which provides a novel storage solution that federates two very different data stores: a transactional database (LeanXcale) and an object store. The seamless component permits to store and query a dataset that has been split between these two data stores as a single logical data set. We now present a critical missing part which is the handling of SQL JOINs by the seamless component. This critical addition is nicely complemented by the advances done in the Data Skipping JOIN capabilities.

Predictive & Process Analytics: the two main scenarios of the Predictive & Process Analytics component are the discovery of insights and the prediction of future events in the context of process flows derived from event driven data. Process mining techniques have been utilized to extract knowledge from event logs which in turn can be transformed into insights and recommendations for the user in the Process Modelling phase of the project. In the second half of the project, we have applied well-established process mining algorithms on

[2] https://cloud.ibm.com/docs/sql-query?topic=sql-query-what-s-new

[3] https://www.researchgate.net/publication/317066213_Too_Big_to_Eat_Boosting_Analytics_Data_Ingestion_from_Object_Stores_with_Scoop

various synthetic datasets, in order to study their comparative performance, in a controlled experiment, allowing us to vary the size of the process model and respective event log, its complexity, as well as the level of noise that can be tolerated.
Real-time Complex Event Processing (CEP) is another essential part for data ingestion as it permits to process data being ingested to yield essential information (e.g., triggering of alarms, push notifications or alerts to end-users, etc.). This component gives the ability to process information on the fly before being stored. During the interim review, the CEP was demonstrated running in the data center. During the second part of the project, the CEP ability to run over distributed setups in devices with different resources has further been developed and analysed. During the second part of the project, the CEP ability to run over distributed setups in devices with different resources has further been developed and analyzed. In particular the CEP is running in two geographically distributed clusters (Madrid and Boston), sending aggregated data from the Madrid cluster to the Boston cluster. With this approach an early generation of alarms will be produced at Madrid cluster speeding up the analytical processing and a further data process will be placed at Boston cluster. In addition, new dynamic scale out/in capabilities have been added allowing the CEP reconfiguration in order to assure enough resources to process the incoming data as fast as possible. To achieve that, CPU, memory and input queues length that store unprocessed data, are controlled by the Dynamic Orchestrator component (WP3) sending scale in/out instructions to the CEP.

3.2 Relevant Roles

The following table lists the BigDataStack roles (as described in deliverable D2.3) that are relevant to the Data as a Service block.

Table 1 – BigDataStack Platform roles

Id	Name	Description
ROL-01	Data Owner	BigDataStack offers a unified Gateway to move (streaming) data from data owners into BigDataStack data stores layer, which support both SQL and NoSQL data stores.
ROL-02	Data Scientist	Data as a Service offers to the data scientists: a) Data Quality Assessment services such as Data Cleaning b) Complex Event Processing, which can be applied on the data streaming in, both before and after it passes though the unified Gateway. c) The possibility to store the data as a single logical data set on both transactional data bases and object store. The data set can seamlessly be queried. d) Data skipping capabilities, optimizing the analytics tasks in terms of time and resources where Big Data is utilized.

3.3 Design

The set of basic data capabilities offered by the Data as a Service block are in fact mostly independent. As it will be presented in section 4, they may be naturally used together as required of typical scenarios of data usage. However, the design of each component is quite component-centric and independent of other components / mechanisms.

Following are the exceptions:
First is for the seamless analytics framework, which takes the LeanXcale database and the object store and produces a new entity built upon these two first ones, permitting a) to define rules for automatic balancing of data sets between the two basic data storage components (e.g., data older than 3 months should be moved to the object store), b) to define and retrieve data from a dataset which may be spread over these two data storage components seamlessly. This component does not depend on Data Skipping, however Data Skipping has a nice synergy with the Seamless component because it permits to accelerate the performance of the SQL queries over the Object Store. Moreover, the seamless analytics framework takes advantage of the adaptable distributed storage in order to allow LeanXcale to efficiently drop a data slice from its storage engine, without stressing its transactional manager, while continuing to ensure data consistency during the process.

A second exception is the synergy between the data quality assessment component (see section 8) and the Real-time Complex Event Processing (CEP) (see section 11): since data quality assessment on-line processing is stateless, CEP happens to be a nice support for data quality assessment which can benefit of all the CEP advantages.

A third exception is the necessity for the integration of the components of the Data Service layer that need to scale with the infrastructure building block of BigDataStack. To make this happen, a novel mechanism has been developed during the third phase of the project that allows for the data services to request a scaling action from the infrastructure (either request additional computational resources or notify the infrastructure that it can release some of the reserved resources), or allows the infrastructure itself to force to scale out/in. This mechanism is common for all components belonging to the Data Service building block, and it is used by the Real-Time Complex Event Processing and the LeanXcale database, both have implemented the interfaces corresponding to the defined protocol.

Typical Big Data processing starts from data acquisition at the edge and then goes through a pipeline as described in Figure 2 where the extraction / cleaning may be performed at the edge or / and near the data store. Data as a Service offers data services that map to all these phases, but the last one (interpretation).

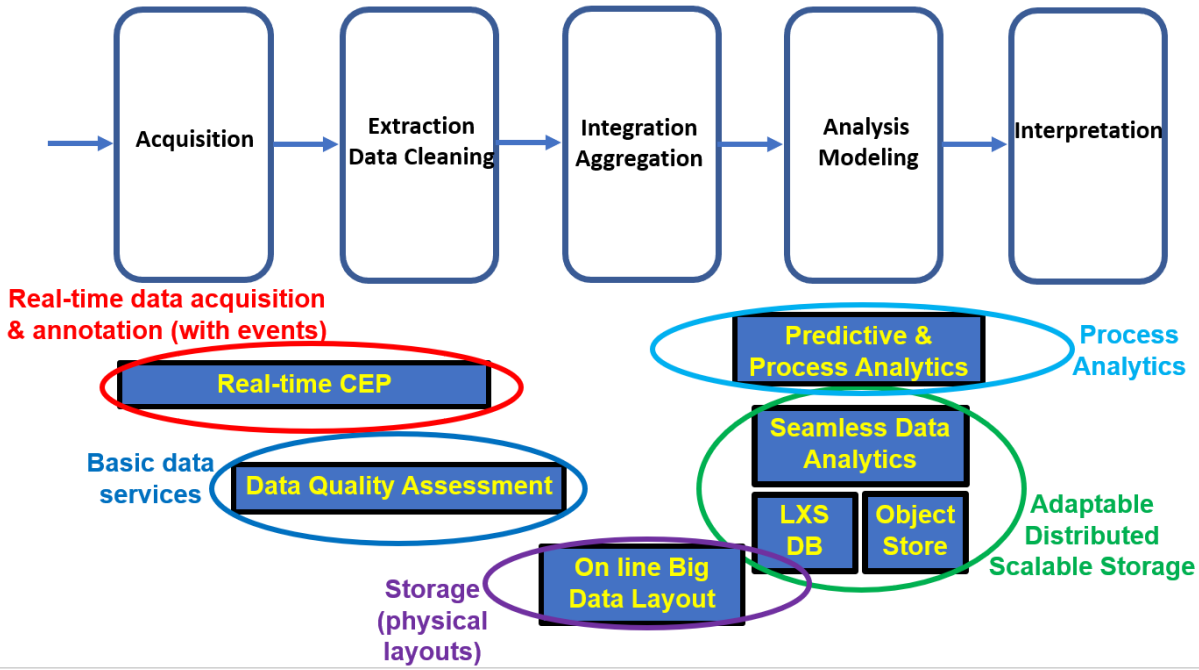


Figure 2: Data as service components mapping to Big Data pipeline

The CEP (section 11) distributes and parallelizes the queries and operators to run in distributed setups. The operators of queries can be User Defined Functions (UDF), such as the data cleaning. In this sense the CEP can be used as an infrastructure for parallelizing UDFs.

4 Implementation and Experimentation

4.1 Experimental Setting

This section introduces the use cases and scenarios that are to be demonstrated during the final review.
Data as a Service is designed to fit a broad scale of Big Data analytics use cases with demanding requirements. During the interim review most of the data services components were integrated and demonstrated for the ship management use case (see deliverables D6.1 and D2.3 section 4.1).

During the final review we extend the demonstration with the following additions:

1. The CEP component has now the ability to scale out when the request pressure causes the events entry queue to become longer than expected, this happens when the CEP doesn't have enough CPU or memory resources. Inversely scale in is triggered when the allocated resources are disproportionate with the event rate processing. This scenario demonstrates full integration with many components of the Data driven Infrastructure Management (such as metrics gathering, intelligent scale decision, etc…).
2. The LeanXcale database uses the similar services of the Data driven Infrastructure Management to scale for fitting its data servers with the data load.
3. The Predictive maintenance ML stack is now reading its data from an Object Store (it was reading directly from the LeanXcale database during the interim demonstration). In addition, this stack has been ported to Apache Spark so as to benefit from the Data Skipping component. The computed alerts are still pushed to the LeanXcale database. The Danaos Alarm detection daemon will detect them and push them to the Danaos software platform.

All the data services involved in this scenario are basically agnostic of the Use Case specificities. The sole exception is the Predictive Maintenance software which is obviously specific to the Use Case.

These general notes apply to the full deliverable:

1. Our design and solution are agnostic of what exact object store we are using. It could be the IBM COS object store[4], as well as OpenStack Swift[5], minio[6], Ceph[7], etc. Therefore, in the following text we use with interchangeability "object store" or "IBM COS". Moreover, the Object Store may either be local or remote. In the interim review of M19, we used a remote IBM COS within the seamless component. In the final review a local minio Object store is to be used.

[4] https://www.ibm.com/cloud/object-storage
[5] https://wiki.openstack.org/wiki/Swift
[6] https://www.minio.io/
[7] https://ceph.com/

2. In terms of distributed computation framework, we chose Apache Spark as the distributed SQL engine for Big Data because of the following reasons:
    1. Among all SQL engines available for Big Data (such as Presto), Apache Spark has the biggest momentum (1400 contributors, Spark SQL alone has 450)
    2. Apache Spark has a very large user base
    3. ANSI SQL 2003 support: Spark SQL has the best ANSI SQL 2003 standard support among all Big Data SQL engines
    4. Apache Spark SQL supports complex and long running queries better than competition
    5. Apache Spark is best when it comes to extensibility and modularity
    6. Apache Spark beats competition in term of SQL query performance[8]. Apache Spark 3.0 yielded an overall improvement factor of X2 for SQL queries performance.

Apache Spark is used by the following components:

- The data skipping component.
- The component incorporating machine learning algorithms for incident prediction towards predictive maintenance in the ship management use case.
- The component incorporating machine learning algorithms for providing recommendations in the connected customer use case.
- The seamless analytics component.

We will now present the ship management use case scenario to be demonstrated during the final review: first of all, we will discuss the use case data ingestion path. Figure 3 shows all the involved components. We detail them from the IoT data creation up to the object store upload phase. In the following description "[x]" will refer to the component with label "x".

[8] https://cdn2.hubspot.net/hubfs/488249/Asset%20PDFs/Benchmark_BI-on-Hadoop_Performance_Q4_2016.pdf

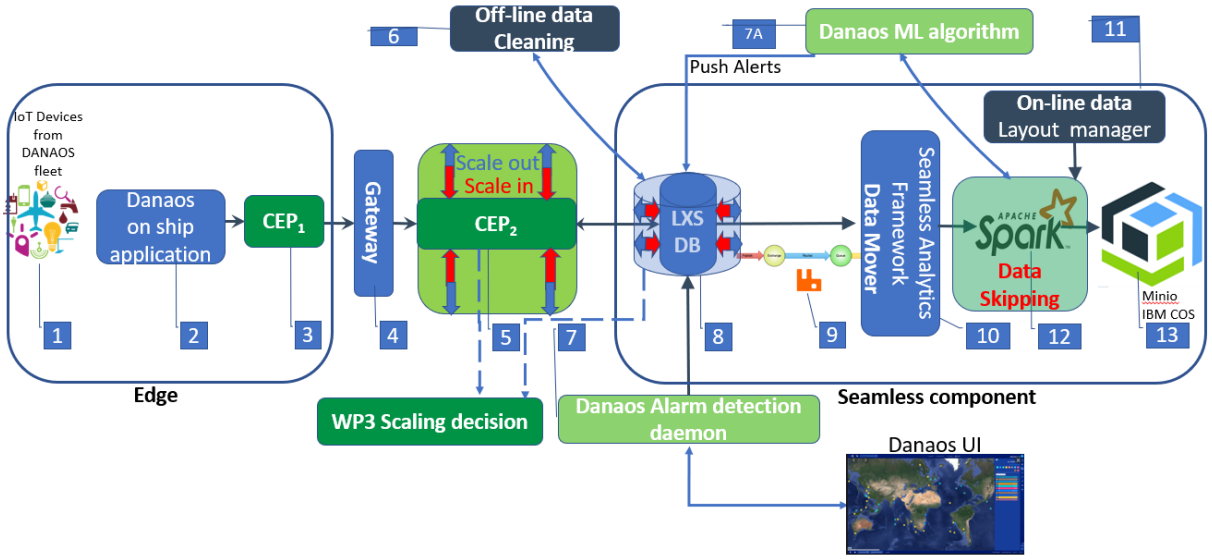


Figure 3: Data ingestion path

We have now 20 vessels whereas we had 7 during interim review. The data of the use case scenario consisted of many Internet of Things (IoT) records (approximately 125 different attributes per minute, excluding meteorological and oceanographic data) collected from several on-board components and sensors, as follows:

- The navigational system (i.e. latitude, longitude, wind speed and angle, speed over ground and speed through water).
- The Alarm Monitoring System (main engine related attributes such as the rotations per minute and the torque of the main shaft, fuel oil inlet temperature and pressure, exhaust gas out temperature, etc.).
- Key-Machinery components (fuel oil volume and temperature, etc).

We refer to the use case description (see deliverable D6.2 section 3 for details about the ship management use case). These IoT records are gathered by what we call the "ship management on board application" [2] for which the sensors of each vessel periodically output events with sensor information that gets aggregated in a row with the updated value for each of the sensors.

These IoT data is fed twice to the CEP: once on-board the ship (to CEP1 [3]) and a second time after passing the gateway, inside the platform (to CEP2 [5]). These two CEP components fulfil different goals.

First IoT data events are input at the edge (the ship) into CEP1. CEP1 has for main goal to detect malfunctioning equipment, such as damaged sensors or recording equipment. This detection is done thanks to a set of rules which, if they are satisfied, prove that either a sensor is malfunctioning (i.e. speed has a negative value) or the IoT data is erroneous (e.g., the speed through water is zero but the rotations per minute of the main shaft is not).

Upon CEP1 processing completion, the IoT data is sent to the Gateway [4]. If data inconsistency is detected by CEP1, an alarm record is created and also sent to the Gateway [4]. The sole use-case dependency is the specific set of rules which are to be checked.

The gateway [4] is the external accessible point of the BigDataStack analytics system and it receives both the IoT data and the potential alarms and forwards them to CEP2 [5]. The gateway is based on OpenShift routes which enables, under the same domain (i.e., same public IP and same URL, just different path), layer 7 load balancing of incoming requests. Therefore, making easy to differentiate the queries/request having to be redirected to CEP2 or possibly to another component. If extra requirements arise, the plan is to move to the usage of service mesh (Istio[9] in this case) and make use of the sidecar containers to handle the extra required actions and the Istio-gateway to redirect the traffic through the entry point, i.e., the Gateway. For more detailed information about the gateway, please check section 5 of Deliverable D3.3.

CEP2 [5] runs in the data center and therefore, has much more processing resources than CEP1 [3]. It will process the incoming IoT data and also reads information from the LeanXcale database to correlate incoming events with stored data (such data is needed for applying rules that CEP2 checks with regard to the Charter Party Agreements of each vessel).

Communication failures lasting hours between vessels and the Danaos ground station are not exceptional. Upon communication restoration, all the data that was not sent is very quickly transmitted to the ground station.

The size of a measurement from a sensor onboard which consists of (a) timestamp of measurement, (b) unique identifier of metric, (c) value, has a maximum size of 32 bytes. These measurements are produced on a per minute basis. A vessel has a minimum of 120 different metrics, thus in steady state the 20 vessels upload 10 Kb/s (kilo-bit per second) of data to the ground station. The minimum upload rate of a satellite connection is 64Kb/s. If the satellite connection is lost, the period of down-time can reach 2 days. In such a case, the record backlog will take up to 20 minutes (for a 2 days connection failure and for an upload rate of 64 Kb/s) to be resolved. During these 20 minutes, the ingestion rate will be approximately (10 + 64) = 74 Kb/s for the 20 vessels, that is 740% of the usual ingestion rate which obviously will demand a scale out of CEP2.

Once all backlog has been resolved, resources requirements decrease back to normal and the scale in process is triggered in order to release the unused resources. This scale up and scale in of the CEP component is demonstrated during the final review and is one of the major novelties as compared with the static CEP component that we had in the interim review.

IoT data output from CEP2 [5] is ingested in LeanXcale database [8] using the datastore API for direct ingestion to the storage layer, which improves performance as it bypasses the query engine while still ensuring transactional semantics. Here too, elasticity has been added and when it is detected that the data servers are no longer sufficient, an additional

[9] *https://istio.io/*

data server will be provisioned and used. This second elasticity capability demonstrate deep integration with the Data Driven Infrastructure Management.

Data is first stored into the LeanXcale database which ensures consistency in terms of transactional semantics and can be scaled out accordingly to handle massive ingestions.

The seamless section 7 details how historical data slices are moved from the LeanXcale database to the object store. The data slices are retrieved from the LeanXcale database by the seamless component: after being triggered by a notification received from RabbitMQ, the seamless component will retrieve the data slice by submitting a regular SQL query to the LeanXcale database. It then creates objects out of the data slices and uploads them to the object store [13]. The major novelty for the seamless component is the support of JOINs which was added during the last year of the project and which now completes the seamless support for SQL.

The data query path (depicted in Figure 4 – Data Query Path) is based on the architecture detailed in the Seamless section 7 where the object store may either be remote or on premises. This query path was implemented and demonstrated in the interim review. This path has not been changed afterwards except for the new support of JOINs.

After data has been ingested in the LeanXcale database, it is processed by the Off-line data quality assessment component [6] which adds to each row its validity probability. This validity probability is taken into account by the Danaos Machine Learning algorithm [7A]. During the interim demonstration, it was shown that [7A] does not work well when data was not assessed while, and on the contrary, it gives high quality failure predictions when the data has been processed by the Off-line data quality assessment component [6].

During the interim demonstration, the integration of the Danaos alarm detection daemon (and the full Danaos visualization application of its fleet) uses as back-end the LeanXcale database which stores the fresh IoT data as well as alarm predictions output by the machine learning algorithm [7A].

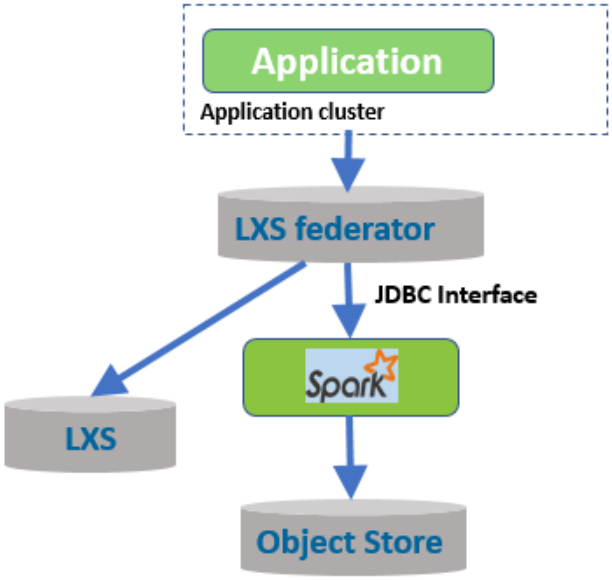


Figure 4: Data Query Path

The seamless analytics component permits to access a data set that may be stored both within the LeanXcale database and within the object store as a single logical data set, which is without having to bother with the location of the data.

The entry point of the seamless component is the LeanXcale federator which federates the two underlying data stores. The component is based on the LeanXcale query engine and can join data coming from different sources. We further refer the reader to sub-section 7.2 for details on how the seamless component design and possible usage.

As depicted in Figure 5 - Process and Predictive Analytics, the entire analytics flow takes advantage of the seamless analytics component illustrated above to gather information from both the LeanXcale distributed database and IBM's object store. The event data is processed through the Predictive and Process Analytics component (section 9) and information concerning the relationship among events is forwarded to other components, more precisely, the Process Modelling Framework part of WP5.

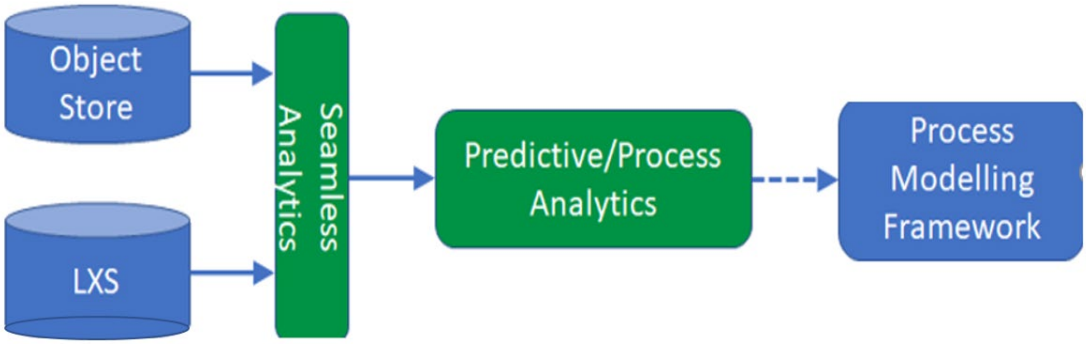


Figure 5: Process and Predictive Analytics

4.2 Past roadmap, actual results and mapping to demos

In this section we report the following:

1. the task actual results versus the roadmap of D4.2
2. additional results not mentioned in the roadmap of D4.2
3. the mapping between the tasks and the demonstrations previewed for the final review

4.3 Task results: roadmap versus actual results

The following list of components was mentioned in the roadmap table of D4.2. We revisit this list by giving the actual status of the components by start of month 35 (submission date for this scientific deliverable).

Table 2 – Actual results for the items mentioned in D4.2 implementation roadmap

Task	Component	Status at M35
T4.1	Support for categorical dataset	Working prototype
T4.1	Support for incremental learning	Working prototype
T4.1	Integration with all use cases	Integrated with shipping and Smark Insurance use cases
T4.2	Data Skipping: support Object Store as metadata store	In operational use for IBM commercial services
T4.2	Data Skipping: support new indexes for data partitioning (e.g., Bloom Filter)	In operational use for IBM commercial services
T4.2	Data Skipping: Hive Metastore support	In operational use for IBM commercial services
T4.3	Implementation of algorithm for solving the problem of resource allocation	Working prototype
T4.3	Development of the Reconfiguration Engine	Working prototype
T4.3	Development of the Elasticity Manager	Full implementation
T4.3	Integration with the WP3 infrastructure	Full implementation
T4.4	SQL Grammar Extension	Will not be implemented*
T4.4	implementation of the JOIN operator for seamless	Working prototype
T4.4	Integration with the LXS query engine	Will not be implemented*

T4.4	Seamless: experimentation using the tpc-ch benchmark	To be evaluated in the Connected Consumer use case
T4.5	Support several process mining algorithms based on ProM	Working prototype
T4.6	Support for queries running in heterogenous nodes	Working prototype
T4.6	Performance evaluation	Done and published
T4.6	Support of distributed queries among heterogeneous nodes	Working prototype

* LeanXcale was out of schedule due to the unexpected additional effort demanded by its new calcite engine (i.e. integration with the seamless components, CEP, additional implementation of a spark-connector to be used instead of JDBC etc). In addition, there was no formal requirement from the task to provide these results which will be pursued in the context of the H2020 PolicyCLOUD project

4.4 Additional results

The following table lists additional results that go beyond the roadmap that has been defined in the previous version of the deliverable (i.e. D4.2).

Table 3 - Additional results

Task	Additional Component	Status at M36
T4.2	Porting to Apache Spark 3.0	In operational use for IBM commercial services
T4.2	Data Skipping for JOINs	Working prototype
T4.3	Migration of the storage engine to the LeanXcaleDB Apache Calcite query engine	In operational use for LXS commercial product
T4.3	Split/Join/Merge of data regions	In operational use for LXS commercial product
T4.3	Online data movement of data regions across nodes	In operational use for LXS commercial product
T4.3	Automated scalability	Working prototype
T4.3	Integration with BigDataStack infrastructure	Working prototype
T4.4	No additional component	
T4.5	Comparative performance evaluation of process mining algorithm for different levels of noise	Working prototype

4.5 Mapping of tasks to final demos

While during the interim review, the WP4 tasks were demonstrated within the Real-time shipping Management use case, in the final review, the WP4 tasks will be used in all of the three Use Case demonstrations. The following table gives a mapping between the tasks and the use cases:

Table 4 - Tasks to final demos mapping

Task	Use Case demonstration	Comments
T4.1	Smart Insurance	
T4.2	Real-time ship management	Integration with the ML Predictive Maintenance stack
T4.3	Real-time ship management	
T4.4	Connected Consumer	
T4.5	Applicable to different use cases	The components developed in the context of Task 4.5 provide generic functionality in the BigDataStack infrastructure, namely search and analysis of event logs, which may be generated by any of the three use-cases in the project
T4.6	Real-time ship management	

4.6 Availability of components

The following table gives pointers of how to obtain the code of the various components as well as how to use it:

Table 5 - Availability of Components

Task	Availability
T4.1	The source code of the predictive maintenance is available at: http://bigdatastack-tasks.ds.unipi.gr/george_makridis/danaos_use_case_predictive_maintenance/-/tree/master/src
T4.2	Data Skipping is (for now still) a closed technology
T4.3	The adaptable distributed storage is now part of the LeanXcale core engine. Its distribution has been available for the consortium and can be found here: http://bigdatastack-tasks.ds.unipi.gr/pavlos_LXS/lx-store-release Regarding the integration with the infrastructure building block, the source code of this mechanism can be found here: http://bigdatastack-tasks.ds.unipi.gr/pavlos_LXS/adsdeploy-wp4
T4.4	The source code of the query federator is available here: http://bigdatastack-tasks.ds.unipi.gr/pavlos_LXS/lx-bigdatastack/-/tree/master/seamless-query-federator

	The source code of the orchestrator of the seamless analytical framework can be found here: http://bigdatastack-tasks.ds.unipi.gr/pavlos_LXS/lx-bigdatastack/-/tree/master/saf-manager
T4.5	Neither Process Mining nor Log Search are publicly available at the time of this writing.
T4.6	The sources for CEP is not currently open

5 Big Data Layout and Data Skipping

5.1 Introduction

5.1.1 General Introduction

The Data Skipping technology developed in the course of the BigDataStack project is without any doubt one the big successes of the project. Indeed, this technology was been selected by the EU "innovation radar[10]" and this innovation was indeed recognized by IBM which has harnessed the technology in no less than four products:

1. GA-ed in the IBM Analytics Engine
2. GA-ed in the IBM Watson Studio
3. GA-ed in the IBM Cloud Pack for Data
4. GA-ed in the IBM SQL Query Service

In addition a paper titled "Extensible Data Skipping" [49] has been accepted to the 2020 IEEE International Conference on Big Data.

By the time of this document submission, this technology is on its way to be Open Sourced by IBM and the open sourcing will probably have been completed by the final review of the project.

As compared with the D4.2 deliverable, the Big Data Layout and Data Skipping section has undergone a major rewriting. The main differences are:

1. The extensible data skipping section has been rewritten. In particular the Experimental Results (sub-section 5.5) which reports entirely new performance results.
2. The state of the art for this technology, already described and submitted one year ago in deliverable D2.3 had to been actualized. We introduced it in sub-section 5.6.
3. We report of an important integration between Data Skipping and the Machine Learning stack used for predictive maintenance of the Shipping use case (see component 7A in the Figure 3). Data Skipping yields an acceleration for the Spark 3.0 Machine Learning algorithms in use in this stack.
4. We added at sub-section 5.7 a short description of additional and important features that were developed during the third year of the project.

After a general introduction, 5.1.1 explains the technological context of the problem and is concluded by the requirements that pertain to Data Skipping and Data Layout (copied for completeness from deliverable D2.3).

Section 5.2 presents the developed Scala APIs to create data skipping indexes and specify how to exploit them.

[10] See https://www.innoradar.eu/innovation/35322

Section 5.3 presents the implementation of the data skipping support

Section 5.4 presents the requirements of a good metadata index type and cover indicators of skipping effectiveness

Section 5.5 presents the experimental results

Section 5.6 presents the State of the Art for Data Skipping and Data Partitioning

In Section 5.7 we present in this sub-section the features added in the second part of the project.

Finally, sub-section 5.8 is the conclusion followed by the references and the Appendixes.

5.1.2 Introduction to the Technology

According to today's best practices, cloud compute and storage services should be deployed and managed independently. This means that potentially huge datasets need to be shipped from the storage service to the compute service to analyse the data. This is problematic even when they are connected by a fast network, and highly exacerbated when connected across the WAN e.g. in hybrid cloud scenarios. To address this, minimizing the amount of data sent across the network is critical to achieve good performance and low cost. Data skipping is a technique which achieves this for SQL analytics on structured data.

Data skipping stores summary metadata for each object (or file) in a dataset. For each column in the object, the summary might include minimum and maximum values, a list or bloom filter of the appearing values, or other metadata which succinctly represents the data in that column. This metadata can then be indexed to support efficient retrieval, although since it can be orders of magnitude smaller than the data itself, this step may not be essential. The metadata can be used during query evaluation to skip over objects which have no relevant data. False positives for object relevance are acceptable since the query execution engine will ultimately filter the data at the row level. However false negatives must be avoided to ensure correctness of query results.

Unlike fully inverted database indexes, data skipping indexes are much smaller than the data itself. This property is critical in the cloud, since otherwise a full index scan could increase the amount of data sent across the network instead of reducing it. In the context of database systems, data skipping is used as an additional technique which complements classical indexes. It is referred to as synopsis in DB2 [41] and zone maps in Oracle [48], where in both cases it is limited to min/max metadata. Data skipping and the associated topic of data layout, has been addressed in recent research papers [44], [42] and is also used in cloud analytics platforms [12], [6]. Data skipping metadata is also included in specific data formats [5], [4].

Despite the important role of data skipping, almost all production ready implementations are limited to min/max indexes over numeric or string columns, with the exception of the ORC/Parquet formats which also support bloom filters. Moreover, queries with UDFs cannot be handled. For example, today's implementations do not support data skipping for the query below[11].

```
SELECT max(temp) FROM weather
WHERE ST_CONTAINS(India, lat, lon) AND city LIKE '%Pur'
```

We address this by implementing data skipping support for Apache Spark SQL[21], and making it extensible in several ways.

1) users can define their own data skipping metadata beyond min/max values and bloom filters
2) data skipping can be applied to additional column types beyond numeric and string types e.g. images, arrays, user defined types (UDTs), without changing the source data
3) users can enable data skipping for queries with UDFs by mapping them to conditions over data skipping metadata

For the query above, our framework allows defining a suffix index for text columns and mapping the LIKE predicate to exploit it for skipping, as well as mapping the ST_CONTAINS UDF to min/max metadata on geospatial attributes. This can reduce the amount of data scanned by orders of magnitude. Our implementation supports plugging in metadata stores, with connectors for Parquet and Elastic Search, and is integrated into multiple IBM products/services including IBM Cloud®SQL Query, IBM Analytics Engine and IBM Cloud Pak®for Data [12], [10], [11].

We demonstrate various use cases for extensible data skipping, show its benefits far outweigh its costs, and show that centralized metadata storage provides significant performance benefits beyond relying on data (Parquet/ORC) formats only for data skipping.

To complete this introduction, we list here the requirements pertaining to Data Skipping and Data Layout as they were expressed in deliverable D2.3:

[11] "India" denotes a polygon with India's geospatial coordinates

Table 6 - Requirement REQ-BDL-01 for Big Data Layout

	Id[12]	Level of detail[13]	Type[14]	Actor[15]	Priority[16]
	REQ-BDL-01	Software	FUNC	Developer	MAN
Name	Support data skipping for arbitrary query predicates				
Description	The query predicate could comprise UDFs and AND/OR/NOT. Example UDFs could be geospatial or temporal functions.				
Additional Information	This functionality is important for the ship management use case, which requires geospatial UDFs.				

Table 7 - Requirement REQ-BDL-02 for Big Data Layout

	Id	Level of detail	Type	Actor	Priority
	REQ-BDL-02	Software	FUNC	Developer	MAN
Name	Support a truly pluggable architecture for data skipping				
Description	The goal of this requirement is to enable the addition of new data skipping index types without changing the core data skipping library. This is needed for requirement REQ-BDL-01 since supporting new UDFs may require new index types.				
Additional Information	External users can also exploit this capability				

Table 8 - Requirement REQ-BDL-03 for Big Data Layout

	Id	Level of detail	Type	Actor	Priority
	REQ-BDL-03	Software	FUNC	Developer	MAN
Name	Enable layout change for (part of) a dataset				
Description	There is a strong relationship between how a dataset is laid out in the object store and the performance of data skipping against this data set. Moreover, this performance may be also very dependent on the queries. Hence the need to adapt the layout, not only for future data but also for				

[12] Identifier: To be used in D2.3 to allow for the correct traceability of requirements.
[13] Level of detail: Following the use of ISO/IEC/IEEE 29148:2011 (see section 2.1 Methodology), we use the following levels: Stakeholder, System and Software (i.e., technology details).
[14] Type: Types of requirements are functional: FUNC (function), DATA (data); and non-functional: L&F (Look and Feel Requirements), USE (Usability Requirements), PERF (Performance Requirements), ENV (Operational/Environment Requirements), and SUP (Maintainability and Support Requirements).
[15] Actor: It needs to be either one of the BigDataStack platform roles identified in section 3.2 or a system actor, e.g. another component or service.
[16] Priority: Requirements can have different priorities: MAN (mandatory requirement), DES (desirable requirement), OPT (optional requirement), ENH (possible future enhancement).

	heavily queried data already in object store.
Additional Information	N/A

Table 9 - Requirement REQ-BDL-04 for Big Data Layout

	Id	**Level of detail**	**Type**	**Actor**	**Priority**
	REQ-BDL-04	Software	FUNC	Developer	MAN
Name	Enable on-line data layout				
Description	Layout is critical for the data skipping performance. As of now data is stored as is and possibly laid out again offline. The need is to upload dataset chunks with the best-known layout as data is ingested.				
Additional Information	N/A				

The following requirements have been added after M18 of the project:

Table 10 - Requirement REQ-BDL-05 for Big Data Layout

	Id	**Level of detail**	**Type**	**Actor**	**Priority**
	REQ-BDL-05	Software	FUNC	Developer	MAN
Name	Support Object Store as metadata store				
Description	Currently an ElasticSearch cluster has to be installed to store the metadata pertaining to data skipping. This requirement adds a moving part to the Data Skipping solution deployment. In addition, ElasticSearch has some limitations such as not implementing Bloom Filters as first-class citizen. We want to move away from ElasticSearch and replace it by the object store itself.				
Additional Information	N/A				

Table 11 - Requirement REQ-BDL-06 for Big Data Layout

	Id	**Level of detail**	**Type**	**Actor**	**Priority**
	REQ-BDL-06	Software	FUNC	Developer	MAN
Name	Support new indexes				
Description	We currently support three indexes: min/max, value list and geo index (which permits to address 2 columns at once such as latitude and longitude).				

	Initial users feedback leads us to add new index the objects. For example, there are many situations where value list should be replaced by much more compact index such as a bloom filter or a gap list index
Additional Information	N/A

Table 12 - Requirement REQ-BDL-07 for Big Data Layout

	Id	**Level of detail**	**Type**	**Actor**	**Priority**
	REQ-BDL-07	Software	FUNC	Developer	DES
Name	Automatic index selection				
Description	Currently the dataset owner has to specify what column should be indexed and with what kind of index. The technology will be more user friendly if we could automatically infer which columns should be indexed and how (e.g., with a value list or with a bloom filter index).				
Additional Information	N/A				

Table 13 - Requirement REQ-BDL-08 for Big Data Layout

	Id	**Level of detail**	**Type**	**Actor**	**Priority**
	REQ-BDL-08	Software	NON-FUNC	Developer	MAN
Name	Enhance the data layout tool				
Description	The indexing of a large dataset takes a long time. Its performance should be improved. This is critical if we detect a change of query load and if we want the flexibility to re-index (parts of) existing dataset to improve the data skipping score.				
Additional Information	N/A				

Table 14 - Requirement REQ-BDL-09 for Big Data Layout

	Id	**Level of detail**	**Type**	**Actor**	**Priority**
	REQ-BDL-09	Software	FUNC	Developer	MAN
Name	Hive Metastore support				
Description	Hive Metastore is a relational database used by Apache Spark SQL to manage the metadata of the persistent relational entities such as databases, columns, etc. We need to integrate our Data Skipping technology with Hive Metastore				

Additional Information	N/A

5.2 Extensible Data Skipping

The developed Scala APIs allow the developer to (1) create data skipping indexes, including adding support for new index types, and (2) specify how to exploit data skipping indexes during query evaluation by mapping predicates to operations on summary metadata. Our framework covers compositions of predicates e.g. using AND, OR and NOT, allowing expressions of arbitrary complexity.

For simplicity, we provide a running example for min/max data skipping, but our APIs can handle arbitrary predicates/UDFs and user defined metadata (e.g. LIKE/ST_CONTAINS and suffix indexes). Useful data skipping metadata for the query below is the minimum and maximum temperature for an object (data subset[17]).

```
SELECT * FROM weather WHERE temp > 101
```

1) *Index Creation*: Users can define new metadata types which extend our MetaDataType class, such as the example below.

```
abstract class MetadataType
case class MinMaxMetaData(col: String,var min: Double, var max: Double)
extends MetadataType
```

Indexes are created explicitly and executed as a dedicated Spark job. Index creation runs in 2 phases - see Figure 6. The first phase accepts a Spark DataFrame (representing an object) and generates metadata having some MetaDataType. The second phase translates this metadata to a metadata store representation. In order to implement the first phase, the developer extends the Index class.

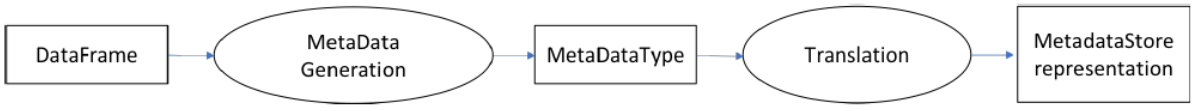
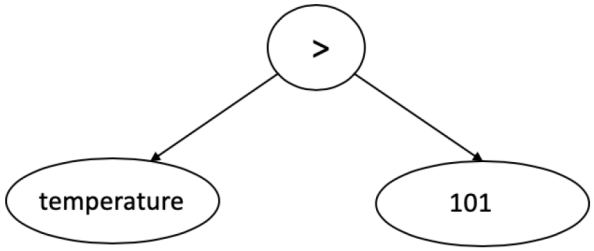


Figure 6: Index Creation Flow

[17] Other alternatives for data subsets are blocks, row groups etc. Our integration with Spark skips at the object level.

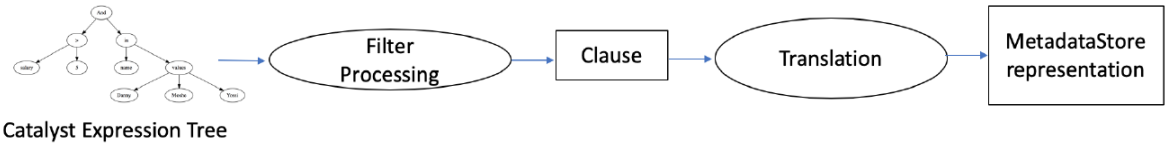
```
abstract class Index(params: Map[String, String], col: String*) {
    def collectMetaData(df: DataFrame):
        MetadataType
```

Our example MinMaxIndex extends Index, and collectMetadata returns a MinMaxMetaData instance containing minimum and maximum values for the given object column.



temperature 101

Figure 7: An Expression Tree (ET) for the example query



Catalyst Expression Tree

Figure 8: Query evaluation flow

2) *Query Evaluation*: Spark has an extensible query optimizer called Catalyst[21], which contains a library for representing query trees and applying rules to manipulate them. We focus on query predicates i.e. boolean valued expressions typically appearing in a WHERE clause, which can be represented as Expression Trees (ETs). Figure 7 shows the expression tree for our example query.

We analyse ETs and label tree nodes with *Clauses*. A Clause is a boolean condition that can be applied to a data subset *s*, typically by inspecting its metadata. Note that for a query ET *e*, for every vertex *v* in *e*, we denote the set of Clauses associated with *v* by *CS(v).*

Definition 1. *Denote the universe of possible data subsets (i.e., objects) by U. A Clause c is a boolean function U → {0 , 1}.*

Definition 2. *For a Clause c and a (boolean) query expression e, we say that c* **represents** *e (denoted by c ≀ e), if for every data subset S, whenever there exists a row r ∈ S that satisfies e, then S satisfies c.*

This means that if S does not satisfy *c*, then S can be safely skipped when evaluating the query expression *e*. For example, let *e* be temp > 101. Given a data subset *S*, let *c* be the

Clause $\max_{r \in S}$ temp(r) > 101. Then *c* represents *e*. Therefore, objects where $\max_{r \in S}$ temp(r) <= 101 can be safely skipped.
Query evaluation is done in 2 phases as shown in Figure 8. In the first phase, a query's ET *e* is labelled using a set of clauses and the clauses are combined to provide a single clause which represents *e*. The labelling process is extensible, allowing for new index types and for new ways of using metadata. In the second phase, this clause is translated to a form that can be applied at the metadata store to filter out the set of objects which can be skipped during query evaluation.
The labelling process is done using *filters*. Typically, there will be one or more filters for each metadata index type. For example, we will define a MinMaxFilter to correspond to our MinMaxIndex.

Definition 3. *An algorithm A is a* **filter** *if it performs the following action: When given an expression tree e as input, for every (boolean valued) vertex v in e, it adds a set of clauses C s.t.* $\forall c \in C$: *c ≀ v to the existing set of clauses.*[18]

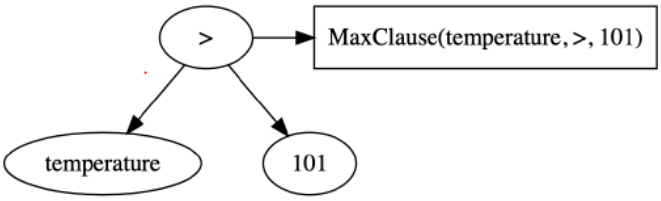


Figure 9: The result of a filter on an ET

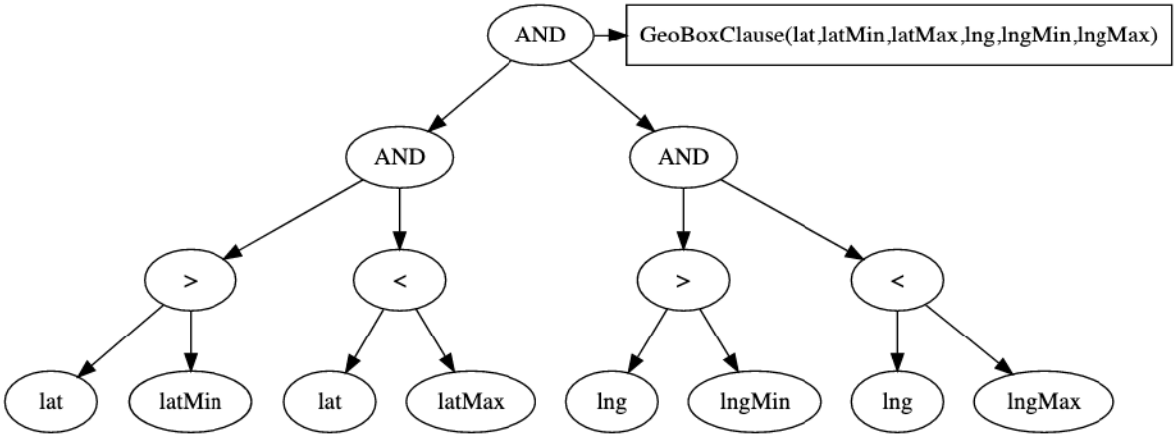


Figure 10: The result of a geobox filter on a complex query

[18] Note that for a particular node, a filter might not add any clauses (this is the special case of adding the empty set).

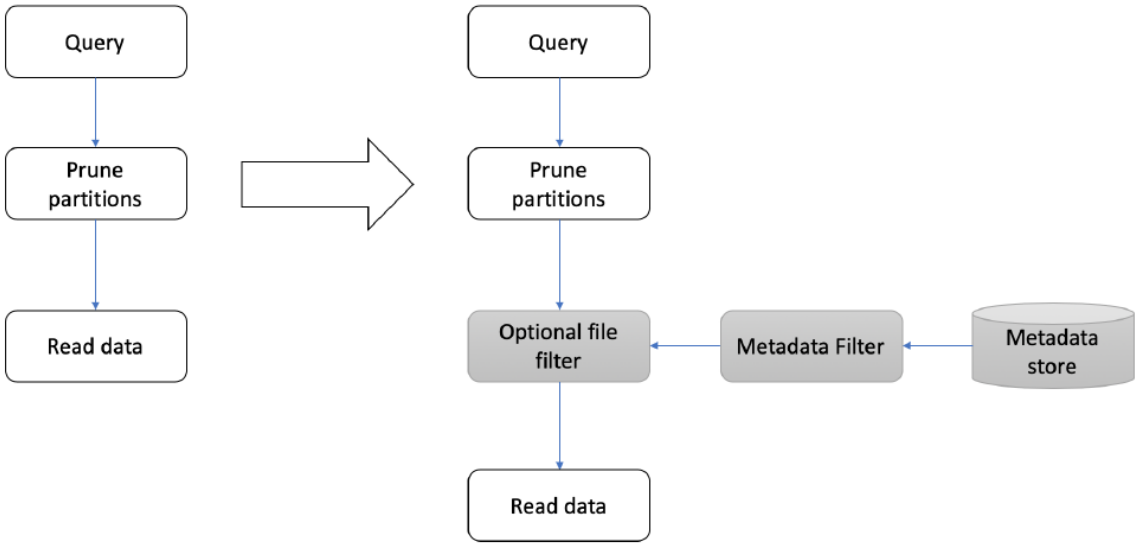


Figure 11: Modified Spark query execution flow after integration

For example a filter *f* might label our ET using MaxClause, as shown in Figure 9, where for a column name *c* and a value *v*, MaxClause(c,>,v) is defined as $\max_{r \in S} c(r) > v$. Since MaxClause(temperature,>,101) represents the node to which it was applied, *f* acted as a filter. Since $\max_{r \in S} c(r)$ is stored as metadata in MinMaxMetaData, MaxClause can be evaluated using this metadata only.

We provide the user with APIs to define clauses and filters. A Clause is a trait which can be extended. A Filter needs to define the labelNode method.

```
case class MaxClause(col:String, op:opType, value:Literal) extends Clause
case class MaxFilter(col:String) extends BaseMetadataFilter {
  def labelNode(node:LabelledExpressionTree):
    Option[Clause] = {
        node.expr match {
          case GreaterThan(attr: Attribute, v: Literal)
            if attr.name == col => Some(MaxClause(col, GT, v))
          case _ => None
  }}
```

Filters typically use pattern matching on the ET structure[19]. Similarly, we can define a MinFilter which can label a tree with MinClauses. Patterns can also match against UDFs in expression trees e.g. ST_CONTAINS - see also section 5.4.3 for queries using UDFs.

In some cases a filter's patterns may need to match against complex predicates using AND/OR/NOT. For example, the GeoBox index (section 5.5.3) stores a 2-dimensional bounding
box for each object and the corresponding filter needs to match against an AND with child constraints on both lat and lng. Figure 10 illustrates this.

Each MetaDataFilter needs to be registered in our system, and during query optimization we inspect the types of metadata that were collected and run the relevant filters on the query's ET. Running the complete set of registered filters will generate an ET where each node can be labelled by multiple Clauses. For every vertex *v* in *e*, we denote the
set of Clauses associated with *v* by *CS(v)*. We recursively merge all of an ET's Clauses to form a unified Clause which represents it. This Clause is then applied to the metadata to make a final skipping decision. For a full formal description of the algorithm used and proof of correctness, see 1.1

5.3 Implementation

We implemented data skipping support for Apache Spark SQL[21] as an add-on Scala library which can be added to the classpath and used in Spark applications. Our work applies to storage systems which implement the Hadoop FileSystem API, which includes various object storage systems as well as HDFS. We tested our work using IBM Cloud Object Storage (COS) and the Stocator connector [19], [46]. Metadata is stored via a pluggable API which we describe in section 5.3.2. The library supports multiple levels of extensibility: code which implements any of our extensible APIs such as metadata types, and clause and filter definitions, as well as additional metadata stores, can be added as plugin libraries.

5.3.1 Spark Integration

Spark uses a partition pruning technique to filter the list of objects to be read if the dataset is appropriately partitioned. Our approach further prunes this list according to data skipping metadata as shown in Figure 11.

Our technique applies to all Spark supported native formats e.g. JSON, CSV, Avro, Parquet, ORC, and can benefit from optimizations built into those formats in Spark. Unlike approaches which embed data skipping metadata inside the data format which require reading footers of every object, our approach avoids touching irrelevant objects altogether. It also avoids wasteful resource allocation because when relying on a format's data skipping, Spark allocates resources to handle entire objects, even when only object footers need to be processed. We provide an API for users to retrieve how much data was skipped for each query.

We used APIs provided by Spark's Catalyst query optimizer to achieve this without changing core Spark. In particular, we added a new optimization rule using the Spark session

[19] For simplicity we left out the cases of _ and _ for MaxFilter above.

extensions API[34]. Spark SQL maintains an InMemoryFileIndex which tracks the objects to be read for the current query and their properties. Our rule wraps the InMemoryFileIndex with a new class extending it by adding the additional filtering step from Figure 11.
We refrain from skipping objects when our metadata about them is stale. This can happen if objects are added, deleted or overwritten in a dataset after indexing it. We keep track of freshness using last modified timestamps, which are retrieved during file listing by the InMemoryFileIndex. We also provide a refresh operation, which updates stale metadata.

5.3.2 Metadata Stores

We support a pluggable API for metadata stores including the specification of how metadata and clauses should be translated for a particular store. This includes the indexing time translation API for Figure 6 and the query time translation API for Figure 8. The key property is that these translations should preserve the correctness of our skipping algorithm. We used this API to implement both Parquet and Elastic Search [8] metadata stores.
It is now widely accepted practice to use the same storage system for both data and metadata[7], [3], [2], avoiding deployment of an additional metadata service. This is achieved using our Parquet metadata store, and all storage systems implementing the Hadoop FS API are supported. Relevant metadata indexes are scanned prior to query execution, but this cost is not significant, since metadata is typically considerably smaller than data. By leveraging Parquet's column wise compression and projection pushdown for metadata, we minimize the amount of metadata that needs to be read per query, ensuring low overhead. Use of Parquet also allows generating and storing metadata for multiple columns together, resulting in better indexing and refresh performance, compared with storing indexes on each column separately.

5.3.3 Protecting Sensitive Data and Metadata

Security and privacy protection for sensitive data are essential for today's cloud services. Parquet supports column wise encryption of sensitive columns in a modular and efficient fashion [16], [17], and being format-agnostic, our library supports skipping over encrypted parquet data transparently. However, an end-to-end solution needs to encrypt metadata, since it can also leak sensitive information. To prevent leakage, when storing metadata in Parquet, we implemented an option to encrypt indexes on sensitive columns, by assigning a key to each index. A user can choose the same key used to encrypt the column the index originates from, choose another key, or leave the index as plaintext. This scheme enables scenarios such as storing data and metadata at a shared location, where each user can only access a subset of the columns and indexes according to their keys.
Spark's partition pruning capability relies on either (1) a widely accepted naming convention which names appropriately partitioned data objects according to their partitioning column name and column value or (2) a Hive metastore. The first option leaks metadata into object names, and therefore partitioning according to sensitive columns is problematic. Use of a Hive metastore in a multi-tenant cloud service pushes the problem of managing sensitive multi-tenant metadata to the underlying database. An alternative is to rely on our data

skipping framework for partition pruning, thereby ensuring end-to-end data and metadata protection, without sacrificing performance.

5.4 Metadata Index Design

In this section we explain the requirements of a good metadata index type and cover indicators of skipping effectiveness. We show that in theory both selecting and designing optimal indexes are hard problems. However, we demonstrate practical choices that work well in this and the following section. We survey various index types implemented using our APIs with a summary in Table 15.

Our goal is to minimize the total number of bytes scanned, because there is a close correlation between this and query completion time (e.g. see section 5.5). Moreover, users of serverless SQL services are typically billed in proportion to the number of bytes scanned [1], [13].

For each query, prior to reading the data, the relevant metadata is scanned and analyzed. As long as the metadata is much smaller than the data, this approach can significantly reduce the amount of data scanned overall. For big datasets the overhead of scanning metadata is usually insignificant compared to the benefits of skipping data (see Figure 13), and in some cases metadata can also be cached in memory or on SSDs. When using our Parquet metadata store, we read only the relevant metadata indexes by using Spark and Parquet column projection capabilities.

5.4.1 Indicator of Skipping Effectiveness

Given a dataset (set of rows) D and a query Q, a row r in D is relevant to Q if r must be read in order to compute Q on D. Let D_r denote the set of relevant rows in D. Assuming D is stored as objects[20], let O denote the set of all objects for D, let O_r denote the set of objects relevant to Q (i.e. having at least one relevant row), and let O_m be the set of objects deemed relevant according to the metadata associated with D. Note that $O_r \subseteq O_m$. Note that O_s = O - O_m is the set of objects that can be skipped.

Denote the number of rows in object o (or dataset D) as $|o|$ ($|D|$).
All definitions below are w.r.t. a dataset D and a query Q.

Definition 4. *The* **selectivity** σ *of a query is the proportion of relevant rows* σ Dr V $\frac{}{DV}$

Data skipping can potentially reduce bytes scanned for selective[21] queries. The definitions use relevant rows rather than rows in the result set to account for queries which perform further computations such as aggregation.

Definition 5. *The layout* **factor** λ *of a query is the proportion of relevant rows in relevant objects*

$$\lambda = \frac{|D_r|}{\sum_{o \in 0_r} |o|}$$

[20] Alternatively other units can be considered such as blocks, row groups etc.
[21] Selectivity ranges between 0 and 1.“Highly selective” queries have close to 0 selectivity

Mixing relevant and irrelevant rows in the same object decreases the layout factor. A high layout factor (grouping relevant rows together) increases the potential for data skipping. To realize this potential we need effective metadata.

Definition 6. *The layout* **metadata factor** μ *of a query is*

$$\mu = \frac{\sum_{o \in 0_r} |o|}{\sum_{o \in 0_m} |o|}$$

The metadata factor is closely related to the metadata's false positive ratio - a low false positive ratio gives rise to a high metadata factor. In addition, the metadata factor takes into account the relative size of each object. A high metadata factor denotes that the metadata is close to optimal given the data layout.

Definition 7. *The* **scanning factor** ψ *of a query is the proportion of rows actually scanned (using metadata)*

$$\psi = \frac{\sum_{o \in 0_m} |o|}{|D|}$$

Our aim is to achieve the lowest possible scanning factor. According to our definitions:

$$\psi = \frac{\sigma}{\lambda\mu} \quad (1)$$

To achieve this for a selective query we need $\lambda\mu$ to be high, and we are equally dependent on good layout and effective metadata[22].

We focus here on metadata effectiveness for any given data layout, and refer the reader to previous work regarding data layout optimization [44], [42]. In practice, often data layout is given and cannot be changed e.g. legacy requirements, compliance, encryption of one or more sensitive columns. In other cases, re-layout of the data is too costly, or it might be difficult to meet the needs of multiple conflicting workloads without duplicating the entire dataset.
Our approach is to enable an extensible range of metadata types, which cater to data within a reasonable range of layout factors. Generating data skipping metadata is typically significantly cheaper than changing the data layout, since no shuffling of the data is needed. Moreover, unlike data layout, it can be done without write access to the dataset and only requires read access to the column(s) at hand. Each user can potentially store metadata corresponding to their particular workload.
On the other hand, using equation (1), we can identify cases where the layout factor is prohibitively low and good skipping is unachievable without re-layout.
To take averages of skipping indicators when considering multiple queries, we use the geometric mean, following [22].

[22] Note that the scanning factor is not defined for queries with 0 selectivity

Let G(X) denote $(\Pi_{i=1}^{n} x_i)^{\frac{1}{n}}$ Given a dataset D and a workload with queries q1…. qn, where for each qi we have $\psi_i = \frac{\sigma_i}{\lambda_i \mu_i}$, then we also have

$$G(\psi) = \frac{G(\sigma)}{G(\lambda)G(\mu)} \quad (2)$$

We apply this approach to measuring the skipping indicators on real world datasets and workloads in the Experimental Results (1.5).

5.4.2 The Index Selection Optimization Problem

Given a dataset D and query workload (set of queries) Q, it is natural to ask what the optimal set of metadata indexes is we can store to achieve the lowest possible scanning factor. Since the workload and data layout are given,σ and λ are given, and to achieve low ψ we need to achieve high μ. We assume that every metadata index *i* has a cost c_i, and that we need to stay within a given metadata budget *K*. A natural cost definition is the size of the metadata in object storage. We also assume that each index $i \in I$ provides a benefit v_i which in our case corresponds to the increase in μ as a result of *i*. Ideally, given *K* and a set of candidate metadata indexes *I*, one could choose an optimal subset $I' \subseteq I$ which gives maximal μ while staying within budget. We show that this problem is NP-hard using a reduction from the knapsack problem. Previous work showed that the problem of finding a data layout providing optimal skipping is also NP-hard [44].

Problem 8. Given dataset *D*, workload *Q*, a set of indexes *I*, and a metadata budget *K*, find $I' \subseteq I$ that maximizes $\sum_{i \in I'} v_i$ subject to $\sum_{i \in I'} c_i \leq K$

Claim 9. *Problem 8 is NP-hard*

Proof: By reduction from {0,1}-knapsack. Knapsack item weight and value correspond to the cost and benefit of an index respectively, and knapsack capacity corresponds to the metadata budget. Clearly, maximizing the value of items in the knapsack within capacity is equivalent to maximizing index benefit within a metadata budget.

Remark 10. *This formulation shows that even in the special case where the benefit of an index is independent from other indexes, the problem is hard. In the general case, the benefit of indexes is relative since, for example, an index which achieves maximalμ renders further addition of indexes obsolete*.

Given a fixed metadata budget, choosing optimal indexes is a hard problem. However, for many index types[23] we store a fixed #bytes per object, thereby bounding the index size to a small fraction of the data size. Using such index types, it is reasonable to index all data columns, assuming the metadata is stored in the same storage system as the data (i.e. with the same storage/access cost).

[23] all index types in table I except for value list and prefix/suffix indexes

5.4.3 An Index Design Optimization Problem

Choosing an optimal set of indexes is hard. What about designing a single optimal index? We show that this is hard even for a range query workload on a single column. Consider a single column c with a linear order e.g. integers, and a workload *Q* with **range queries** over *c* i.e. queries of the form

SELECT * FROM D

WHERE c between c1 and c2

Storing min/max metadata only for *c* may not achieve maximal μ, for example, when an object's rows have gaps in column *c* between the min and max values. In this case if c1 and c2 both fit inside the gap then min/max metadata will give a false positive for the query above. A gap list metadata index could store a list of such gaps per object and be used to skip objects having gaps covering the intervals used in queries. Given a dataset *D*, a workload *Q* and metadata budget of *k* gaps, which gaps should be stored to give optimal μ? (We assume the cost of each gap is equal). An algorithm which achieves this is provided in [31]. However, we show that allowing queries with disjunction turns this into a hard problem.

Problem 11. *Given a dataset D with a column c having a linear order, a workload Q comprising of disjunctive range queries over c, and a metadata budget of K gaps, find a set of k gaps where* $k \leq K$ *such that* μ *is maximized.*

Problem 12. *(Densest k-Subhypergraph problem) Given a hypergraph G = (V , E) and a parameter k, find a set of k vertices with maximum number of hyperedges in the subgraph induced by this set[29].*

Claim 13. *Problem 11 is NP-hard.*

Proof. By reduction from the densest k-Subhypergraph problem. Given *G = (V , E)* and *k*, we construct an input to problem 11 as follows. We create a dataset with one object *O* such that its column *c* induces |V| gaps – $\{g_1, g_2, \ldots, g_{|V|}\}$, and use the function $f(v_i) = g_i$ to map each vertex to a gap. Each hyperedge e ∈ E is mapped to a query with a WHERE clause comprised of a predicate of the form:$v \epsilon ec \in f(v)$. In order to skip *O* for this query we need exactly those gaps in {f(v) | $v \ \epsilon e$ }. In this setting maximizing μ (the number of queries where *O* is skipped) is equivalent to finding the densest k-Subhypergraph.

5.4.4 Metadata Index Types

Table 15 contains a summary of common index types (Min-Max, BloomFilter) as well as novel ones we found useful for our use cases and implemented for our Parquet metadata store. All metadata enjoys Parquet columnar compression and efficient encoding - therefore the Bytes/Object values in the table can be considered an upper bound.

The **MetricDist** index enables similarity search queries using UDFs based on any metric distance e.g. Euclidean, Manhattan, Levenshtein. Applications include document and genetic similarity queries. Recently semantic similarity queries have been applied to databases[26], where values are considered similar based on their context, allowing queries such as "which employee is most similar to Mary?". Assuming a metric function for similarity, extensible data skipping can be successfully applied.
Additional index types can be easily integrated by implementing our APIs - example candidates include SuRF[47], HOT[24], HTM[45]. Recent work demonstrated the use of range sets (similar to our gap lists) to optimize queries with JOINs[35]. Adding a new index type via our APIs requires roughly 30 lines of new code.

5.4.5 A Hybrid Index

When a column typically has low cardinality per object, a value list is both more space efficient than a bloom filter and avoids false positives. However, for high cardinality, value list metadata size can approach that of the data. In order to achieve the best of both worlds, we implemented a hybrid index, which uses a value list up to a certain cardinality threshold, and a bloom filter otherwise. We now explain how we determined an appropriate threshold.
Assuming equality predicates only, we compare value list and bloom filter indexes using the formulas presented in Table 15. Our aim is to minimize the total bytes scanned for data and metadata. Given an object of size |o|, a column with v distinct values each one of size b bits, and a workload Q = $\{q_i\}_{i=1}^{n}$ of exact match queries, let Ei ∈ {0 , 1} be the event in which o must be read for q_i. It follows that the average data to be scanned for the workload using a bloom filter index is approximately $\frac{1}{n}\Sigma_{i=1}^{n}\left(\frac{-vlnf}{ln^2 2}+E_i|o|+(1-E_i)f|o|\right)$.The average data to be scanned for the workload using value list is exactly $\frac{1}{n}\Sigma_{i=1}^{n}\left(-v\bar{b}E_i|o|\right)$
Therefore, a value list index is preferable when:

$$v\left(\bar{b}+\frac{lnf}{ln^2 2}\right)<f|o \vee \left(1-\frac{1}{n}\sum_{i=1}^{n}E_i\right)$$

The term $\frac{1}{n}\sum_{i=1}^{n}E_i$ can be approximated using the expected scanning factor when using a value list index, which can be derived from the workload mean layout and selectivity factors using equation 2.
For example, given an object of size 64MB with a string column of up to 64 characters ($\bar{b}$ = 512) and a target scanning factor of 0.01, a value list up to 10,088 elements is preferable over a bloom filter with f = 0.01. We implemented a **hybrid index** which creates a bloom filter or value list per object according to the column cardinality. By default, we use a threshold of 10K elements based on the above example, but this threshold can be changed according to dataset properties.

5.5 Experimental Results

We focus on use cases where data is born in the cloud at a high, often accelerating, rate so highly scalable and low-cost solutions are critical. We demonstrate our library for geospatial analytics (representing IoT workloads in general) and log analytics on 3 proprietary datasets. We collect skipping effectiveness indicators and discuss their effect on the scanning factor (hence data scanned). All experiments were conducted using Spark 2.3.2 on a 3 node IBM Analytics Engine cluster, each with 128GB of RAM, 32 vCPU, except where mentioned otherwise. The datasets are stored in IBM COS. All experiments are run with cold caches.

Table 15 - Data Skipping Index Types

Index Type	Description	Column Types	Handles Predicates[1]	Bytes/Object[2]
MinMax	Stores minimum/maximum values for a column	ordered	$p(n, c)$	$2b$
GapList	Stores a set of k gaps indicating ranges where there are no data points in an object	ordered	$p(n, c)$	kb
GeoBox	Applies to geospatial column types e.g. Polygon, Point. Stores a set of x bounding boxes covering data points	geospatial	geo UDFs	$2xb$
BloomFilter	Bloom filter is a well known technique[25]	hashable	$n = c, n \in C$	$\frac{-v \ln f}{\ln^2 2}$ (in bits)
ValueList	Stores the list of unique values for the column	has =, text	$n = c, n \in C$, LIKE	vb
Prefix	Stores a list of the unique prefixes having b_1 characters	text	LIKE 'pattern%'	$v_1 b_1$
Suffix	Stores a list of the unique suffixes having b_2 characters	text	LIKE '%pattern'	$v_2 b_2$
Formatted	Handles formatted strings. There are many uses cases.	text	template based UDFs	varies
MetricDist	Stores an origin, max and min distance per object	has metric dist	metric distance UDFs	$2m + b$

[1] $p \in \{<, \leq, >, \geq, =\}$. n is a column name and c is a literal, C is a set of literals.
[2] b is the (average) number of bytes needed to store a single column element. v is the number of distinct values in a column for the given object. k is the number of gaps (configurable). x is the number of boxes per object. v_1 (v_2) is the number of distinct values with prefix (suffix) of size b_1 (b_2). m is the number of bytes needed to store a distance value. f is the false positive rate ($f \in (0, 1)$).

A proprietary (1) **Weather Dataset** contains a 4K grid of hourly weather measurements. The data consists of a single table with 33 columns such as latitude, longitude, temperature and wind speed. The data was geospatially partitioned using a KD-Tree partitioner[42]. One month of weather data was stored in 8192 Parquet objects using snappy compression with a total size of 191GB.
The two proprietary http server log datasets below are samples of much larger datasets and use Parquet with snappy compression:

(2) A **Cloud Database Logs dataset**, consisting of a single table with 62 columns such as db name, account name, http request. The data was partitioned daily with layout according to the account name for each day, resulting in 4K objects with a total size of 682GB.

(3) A **Cloud Storage Logs dataset**, consisting of a single table with 99 columns such as container name, account name, user agent. The data was partitioned hourly, resulting in 46K objects with a total size of 2.47TB.

5.5.1 Indexing

Use of our APIs allows adding new index types achieving similar performance to native index types with little programmer effort. Table 61 in 1.21 reports statistics for indexing a single column using various index types on our datasets. In addition, we implemented an

optimization which reads min/max statistics from Parquet footers, which gives significant speedups when only MinMax indexes are used on Parquet data[24].
Figure 12 shows that indexing multiple columns using the Hybrid index is significantly faster than indexing each column separately[25], even for Parquet data where columns are scanned individually. For MinMax the indexing time remains low (benefiting from our MinMax optimization) and flat when varying the number of columns.

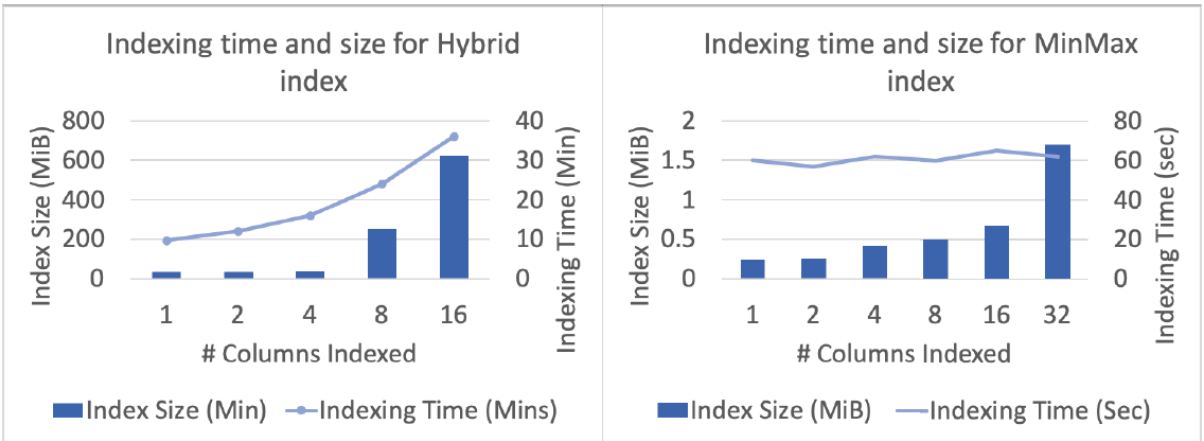


Figure 12: Indexing time/size vs #columns (log scale) with Hybrid

We note that indexing can be done per object at data generation or ingestion time, and can alternatively be done using highly scalable serverless cloud frameworks e.g. [33].

5.5.2 Metadata versus Data Processing

Figure 13 shows time and bytes scanned for 4 queries searching for different values of the DB name column (cloud database logs dataset). The queries retrieve 8 columns, and we compare ValueList, BloomFilter and Hybrid indexes, and in all cases either ValueList or the Hybrid index outperforms BloomFilter (whereas BloomFilter is the index most widely adopted in practice). There is a clear correspondence between bytes scanned and query completion times, and data skipping reduces query times roughly between x3 and x20. In all cases, the time spent on metadata processing is a small fraction of the overall time. For all queries, the Hybrid index requires the least metadata processing time because of its smaller size. For Q4, when almost all data is skipped, the Hybrid index is superior for this reason. For Q2 and Q3, Hybrid and BloomFilter incur false positives and so retrieve more data than ValueList, resulting in longer query times.

We point out that for this scenario it only takes around 3 queries to save the 10 mins that were spent on indexing the db name column. On the other hand, the overhead for all queries (selective and non-selective) with all indexes is less than 20 seconds per query. In terms of bytes scanned, we scanned 6.73 GB to index the DB name column, whereas each

[24] If additional index types are used it provides no benefit since the Parquet row groups need to be accessed in any case
[25] Other index types behave similarly

query saves over 100GB (because of the additional columns retrieved). Therefore, in terms of cost, a user can achieve payback after a single query.

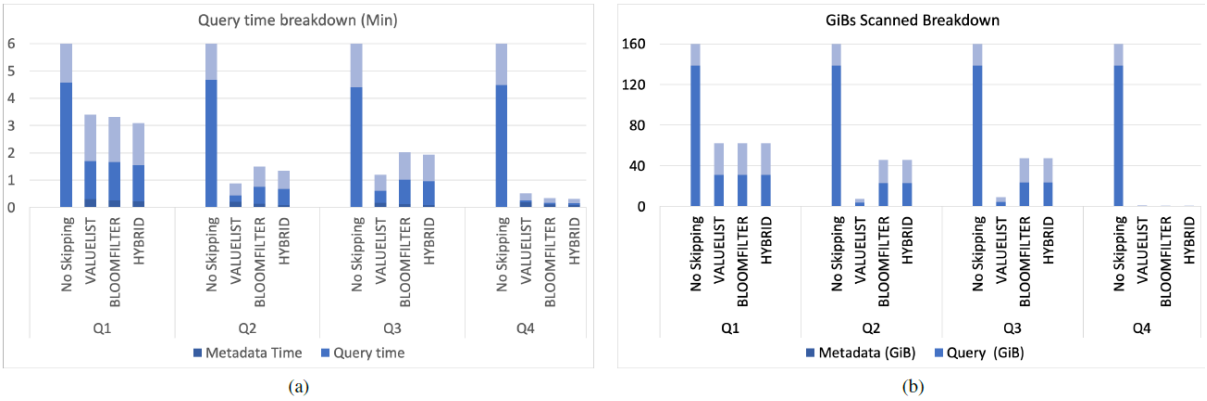


Figure 13: Breakdown of time spent on data and metadata processing for 4 queries on cloud database logs and corresponding bytes scanned

5.5.3 Data Skipping for Geospatial UDFs

The real-time ship management use case was an excellent use case to demonstrate data skipping for queries with predicates containing UDFs. We present in this section a more compact report of the results. For more ample details, we refer the reader to section 5.8.1 of D4.2.

To our knowledge, no other SQL engine supports this, since query optimizers typically know very little about UDFs. We used our extensible APIs to create filters that identify UDFs from IBM's geospatial toolkit[14] and map them to MinMax and GeoBox index types. Supported predicates include containment, intersection, distance and many more [9].

For example, the following query retrieves all data whose location is in the Bermuda Triangle. Without data skipping, the entire dataset needs to be scanned.

```
SELECT * FROM weather WHERE ST_CONTAINS(
        ST_WKTToSQL('POLYGON((-64.73 32.31,...))'),
        ST_POINT(lat, lng)
)
```

In order to support skipping we can either use the GeoBox index on the pair of lat/lng columns, or use independent MinMax indexes on both lat and lng. For each case we map the relevant UDFs to the corresponding Clauses. The GeoBox index has the advantage that it can handle lower layout factors by using multiple boxes per object. Since we partitioned the dataset according to lat/lng, the MinMax approach is also effective.

Figure 14 compares running ST Contains queries with and without data skipping[26]. The queries were run on an extrapolation of the weather dataset to a 5-year period. We used MinMax indexes resulting in 11MB of metadata for close to 12TB of data. The specific query we ran has the same form as our example query and selects data with location in the Research Triangle area of North Carolina, with time windows ranging between 1 to 12

[26] The results for ST Distance are similar

months. We achieved a cost and performance gap which is over 2 orders of magnitude – the gap increases in proportion to the size of the time window. For a 5-month window we achieved a x240 speedup. The cost gaps reflected by amount of data scanned are similar. We conclude that even with a high layout factor, running queries with UDFs directly on big datasets is clearly not feasible without extensible data skipping.

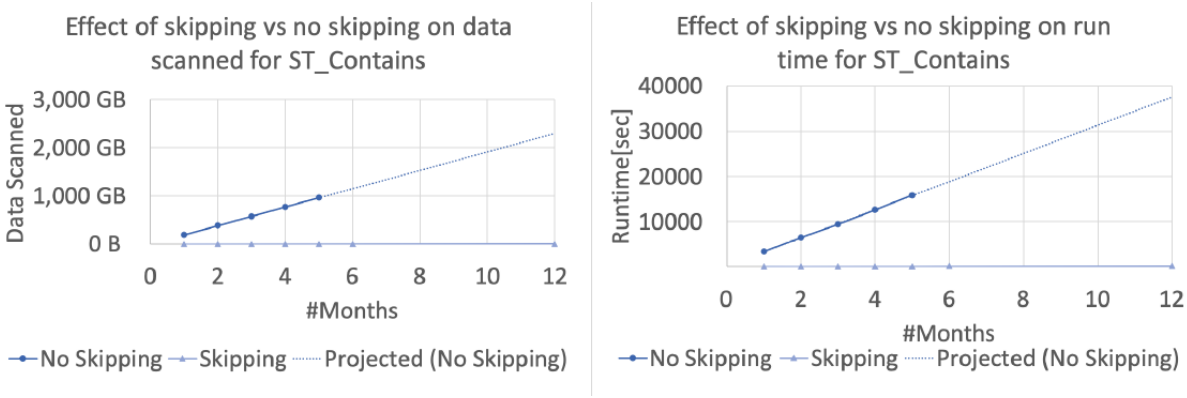


Figure 14: Effects of skipping versus no skipping for ST_Contains applied to the weather dataset with varying time window sizes

5.5.4 Benefits of Centralized Metadata

An alternative approach is to apply geospatial data layout and rewrite queries to exploit min/max metadata, if available in the storage format. This approach requires users to rewrite queries manually, or else query rewrite needs to be implemented for each query template. For example, the previous query could be rewritten to the one below

```
SELECT * FROM weather WHERE
ST_CONTAINS(ST_WKTToSQL('POLYGON((-64.73 32.31,...))'),ST_POINT(lat,lng))
AND lat BETWEEN 18.43 AND 32.31
AND lng BETWEEN -80.19 AND -64.73
```

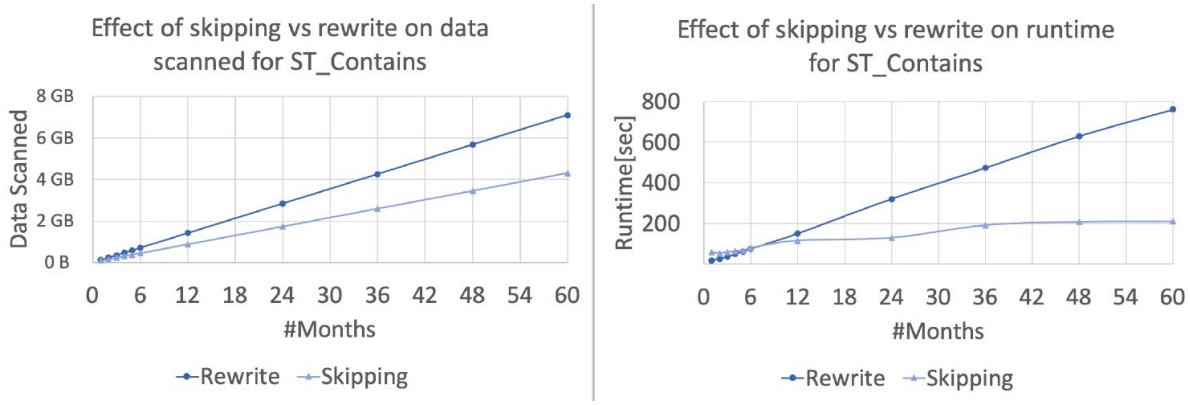


Figure 15: Effects of skipping versus rewrite approach for ST_Contains applied to the weather dataset with varying time window sizes

Our approach uses centralized metadata which avoids reading the footers of irrelevant Parquet/ORC objects altogether. This achieves a performance boost for 2 main reasons: overheads for each GET requests are relatively high for object storage, and Spark cluster resources are used more uniformly and effectively. The bytes scanned are reduced both by avoiding reading irrelevant footers and by metadata compression, which lowers cost. Figure 15 compares the cost and performance of extensible data skipping to a query rewrite approach. Since the data is partitioned geospatially, both identify the same objects as irrelevant. However, our centralized metadata approach performs x3.6 better at run time at x1.6 lower cost for 5 year time windows, demonstrating significant benefit.

5.5.5 Prefix/Suffix Matching

SQL supports pattern matching using the LIKE operator, supporting single and multi-character wildcards. We added prefix and suffix indexes to support predicates of the form LIKE 'pattern%' and LIKE '%pattern' respectively. The indexes accept a length as a parameter and store a list of distinct prefixes (suffixes) appearing in each object. This is more efficient and results in smaller indexes compared to value list when a column's prefixes/suffixes are repetitive[27].

In Figure 16 we present the skipping effectiveness indicators for prefix/suffix matching on the DB name column and prefix matching on the http request column of the cloud database logs dataset. For the DB name column we stored prefixes and suffixes of length 15, and for the http request column we stored prefixes of length 20. Note the average column lengths for these columns are much higher. We generated a workload for each index consisting of 50 queries. For the prefix workloads, each query has a LIKE 'pattern%' predicate, where the pattern is a random column value in the dataset with prefix of random size up to the column value length. The suffix workload is generated similarly.

Overall, the aim is to bring the scanning factor as close as possible to the selectivity. The extent to which this is possible depends on how close we can bring the layout and metadata factors to 1 (equation 2). Despite relatively low layout factors (layout was not done according to the queried columns), good skipping is achievable. All indexes shown here achieve metadata factor close to 1, despite storing only prefixes/suffixes, and give a range of beneficial scanning factors.

[27] A trie based implementation is a topic for further work

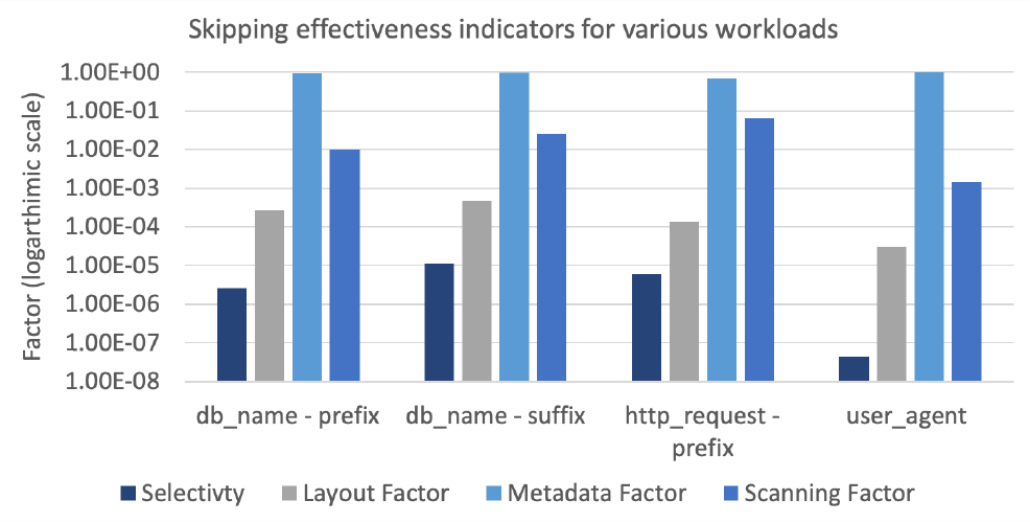


Figure 16: Skipping effectiveness indicators for prefix/suffix and format specific user agent indexes.

For selectivity and scanning factors lower is better, for metadata and layout factors higher is better. With approximately equal metadata factors, a highly selective (selectivity closer to 0) user agent workload makes up for a significantly lower layout factor, achieving the best scanning factor overall. All indexes are beneficial, achieving between 1/1000 and 1/10 scanning factors.

In Figure 17 we show the effects of increasing the prefix length in terms of skipping indicators as well as metadata size. In this case we generated a different random workload for the db name column with 20 queries13. Here the selectivity and layout factors are fixed so the scanning factor is inversely proportional to the metadata factor. According to equation 1 the lowest possible scanning factor is around 10^{-2}. We achieve this for prefix length 15 with an order of magnitude smaller metadata compared to a value list index.

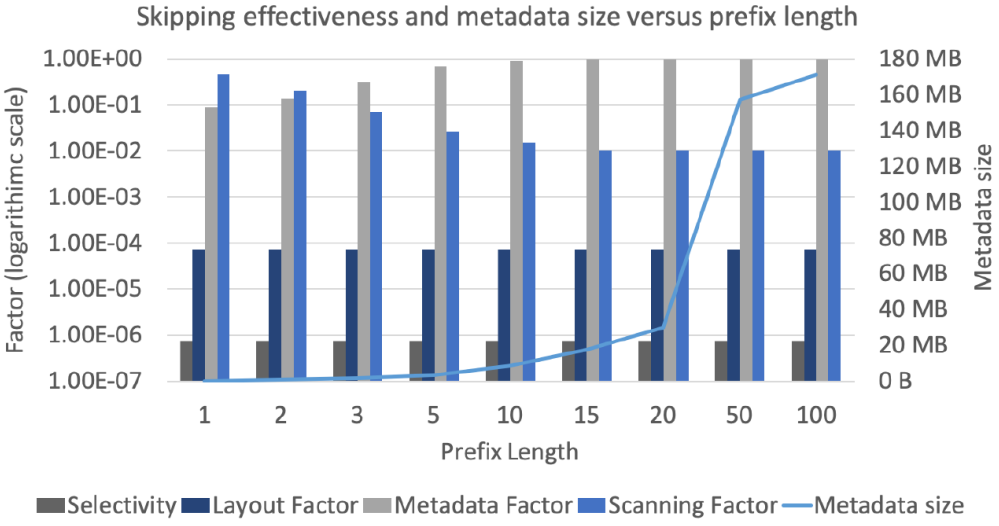


Figure 17: Skipping effectiveness indicators and metadata size for a prefix index on the DB name column w.r.t. prefix length

5.5.6 Format Specific Indexing

As is typical for log analytics, many columns in our logs datasets e.g. DB name, http request have additional application specific (nested) structure not captured by prefix/suffix indexes, such as hierarchical paths and parameter lists. We show how to index such columns, avoiding the need to add new data columns, which is often not feasible for large and fast growing data.
We indexed the user agent column[40] of both datasets to track the history of malicious http requests. Our extensible framework enables easy integration with open source tools. We used the Yauaa library[18], benefitting from its accurate client identification[23], its handling of idiosyncrasies in the format, and its keeping up to date with frequent client changes. The library parses a user agent string into a set of field name-value pairs. To generate the metadata, we parsed out the agent name field, and stored a list of names per object. We also implemented the getAgentName UDF. The query below retrieves all malicious http requests in the log.

```
SELECT * FROM storagelogs
WHERE getAgentName(user_agent)='Hacker'
```

In Figure 16 we show the skipping effectiveness indicators for this index, using a workload consisting of 50 queries, where for each query we chose a random agent name appearing in the dataset. This highly selective workload enables very good skipping even with low layout factor.

5.5.7 Data Skipping for ML stack acceleration

A short description of the ML Predictive Maintenance component which was presented in interim review. The historical data from 20 vessels was incorporated as described in section 4. We approach the Predicting Maintenance problem in two steps. Firstly, we trained various models, and calculated Permutation Entropy to perform either regression or classification and predict the expected value of each model of the next timestamp (a.k.a time step). Secondly, we created anomaly detection rules according to each model objective.
Considering the data-skipping necessity in the specific use-case, we have applied it to the ML Predictive Maintenance pipeline. It is worthwhile mentioning that the Model is a complementary model consisted of 4 ML algorithms (models including LSTM, one Class SVM, some Tree-based Boosting models) and some statistical terms as and presented in the 23rd IEEE International Conference on Intelligent Transportation Systems. Regarding the "training" part of the process, the process is depicted in Figure 18.

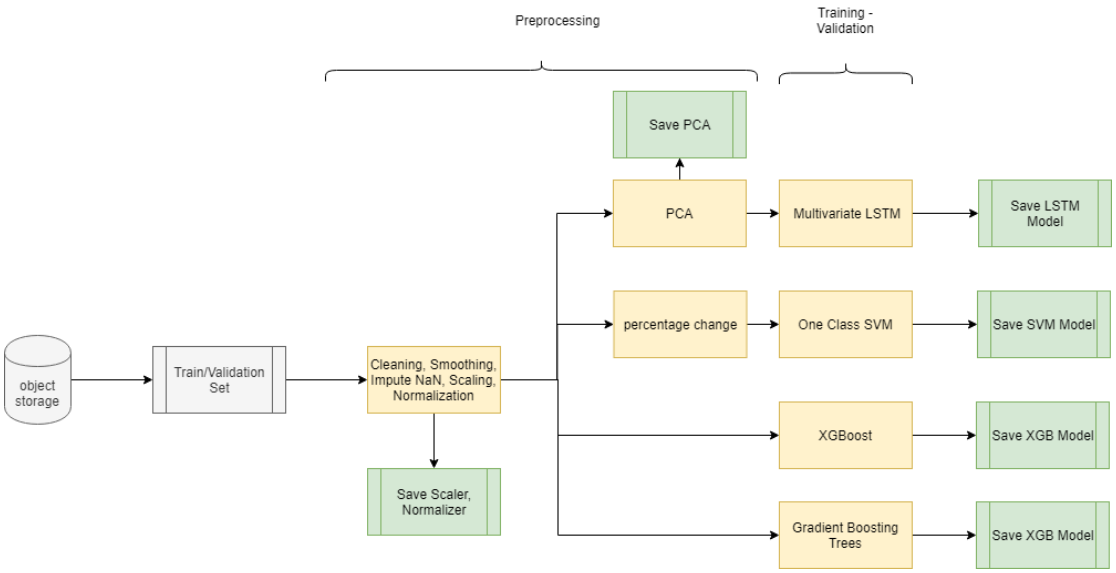


Figure 18: Training process

While the data collection may include inconsistent and/or faulty data (e.g. due to recording inconsistencies, human errors, or sensor faults). Subsequently specific preprocessing steps were applied as depicted in Figure 19.

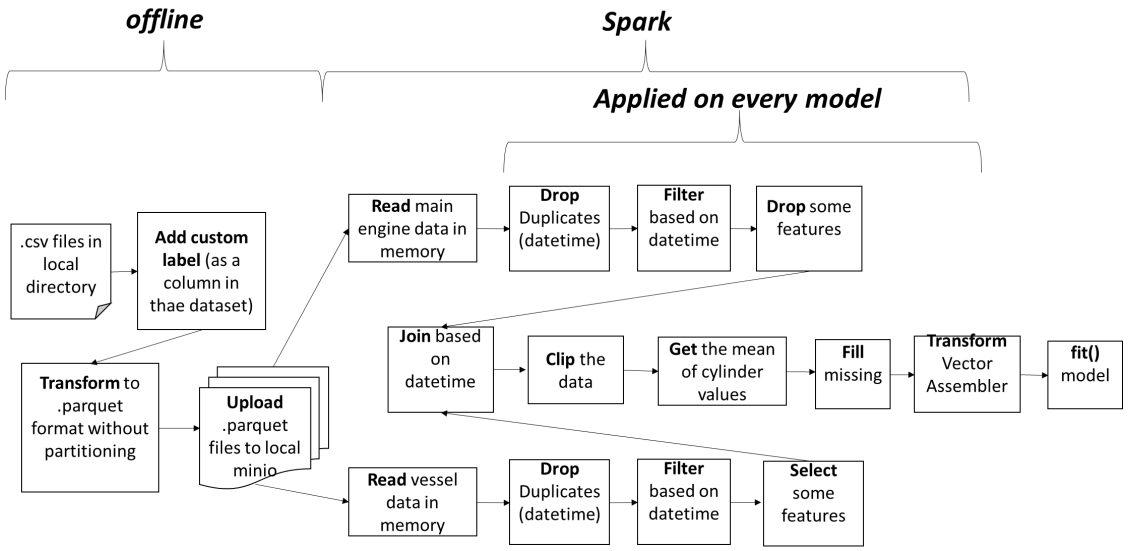


Figure 19: preprocessing steps

The key features of the used component, yielding the results in this section, were:

- We used Spark 3.0
- Local Minio as object store
- 2 parquet files for each vessel (one for the main-engine data, and one for the generic vessel data).

- Each parquet file is partitioned into 200 partitions and sorted by “datetime”. This yields a total of 20x2x200 = 8000 objects
- Both files have the “datetime” as “feature”, so the data-skipping JOIN capability was not tested on the datetime filtering.
- Not all the files were utilized for every model, according to the model objective.
- MinMaxIndex indexing was used.

The maritime industry has been confronted with some issues since 2016 due to increased market prices, and the reduction of chartered ship rates, the growing concern for suppressing emissions from shipping has set additional challenges. So, based on the fact that the “operational-behavior” of the vessels have started to change after approximately 2016. Which is aligned with the fact that the utilized dataset has many missing values before 2016.

Based on the assumptions we filter out the records taken before 2016. The results of the data-skipping usage for each of the model, regarding the “training” process are presented in Table 16. The results came from the data skipping analytics tool.

Table 16 - The performance results for Danaos use case

Model	#vessel data utilized	skipped Bytes	skipped Objs
Full Dataset	20/20	21%	22.7%
XGB Classifier	7/20	20%	22%
SVM	12/20	21%	23%
LSTM	12/20	21%	23%
Gradient Boosting Regressor	12/20	21%	23%

The same results were also verified using the spark measure framework as presented in Table 17:

Table 17 - The performance results for Ship Management use case (sparkmeasure)

Model	#vessel data utilized	elapsed time (min)	Records read	Bytes read (GB)	elapsed time (min)	Records read	Bytes read (GB)
XGB Classifier	7/20	11	98%	1159	11	$3.36\ 10^9$	1161
SVM	12/20	6.4	92%	297	5.6	$9.20\ 10^8$	299
LSTM	12/20	7.2	76%	12	335020	$3.68\ 10^8$	16
Gradient Boosting Regressor	12/20	17	99%	1376	17	$4.93\ 10^{10}$	1378

Considering the data-skipping necessity in the specific use-case, it can be approached from the perspective of a Data Scientist. Concerning data analysis in a real-life problem, many challenges should be addressed. A well-established approach is the evaluation of different hypotheses - scenarios trying to prove or extract significant insights from the data.

Given that, when a Dataset consists of Big Data, not being feasible to load it in memory, there is a point where data skipping is handy and useful. Towards this perspective data-skipping will be incorporated utilizing the JOIN feature in a complementary data analysis leveraging some 300 GB of weather data in the Danaos use case to be presented in the final review.
As expected, Data skipping did not affect the results (predictions) of the model. In the following we detail elapsed time and needed resources.
The used the "MinMax" indexing, which stores the minimum and the maximum values for a column as metadata. The MinMax indexer was applied on the "datetime" feature. This was applicable as the files (.parquet format) were the data set was stored was partitioned and sorted based on the "datetime" column.
Note that the metadata is typically much smaller in volume than the data. Furthermore, it is obvious that the application of different sparkML models may have an impact on the records read, this task is under investigation.
The summary of the result of the data-skipping for the examined scenario, is that, 23% of the total objects and 21% of the total size were skipped and 21%. These results are slightly different with spark-measure, presented in the second table. As a small overhead is introduced possibly due to the process of meta-index to be read. Note that the metadata is typically much smaller in volume than the data. Furthermore, it is obvious that the application of different sparkML models may have an impact on the number of read records, this task is under investigation.

5.6 Baseline

Hive style partitioning partitions data according to certain attributes, encoded as metadata in filenames. Spark/Hadoop use this metadata for partition pruning. Using this technique alone is inflexible since only one hierarchy is possible, changing the partitioning scheme requires rewriting the entire dataset when using object storage (which has no rename operation), and range partitioning is not supported. Our framework for extensible data skipping is complementary to this technique.

Parquet and ORC support min/max metadata stored in file footers, as well as bloom filters[5], [4]. Both support dictionary encodings which provide some of the benefit of our value list indexes. Note that these encodings are primarily designed to achieve compression, so in some cases other encodings are used instead, compromising skipping [27]. Both formats require all objects to be partially read to process a query, and footer processing is not read optimized. Neither format allows adding metadata to an existing file, whereas our approach allows dynamic indexing choices. Parquet allows user defined predicates as part of a Filter API, however this is designed to work with existing metadata only. Since query engines have not exposed similar APIs this does not achieve extensible skipping.

Data skipping Min/max metadata, also known as synopsis and zone maps, is commonly used in commercial DBMSs [41], [48] and some data lakes[6]. Other index types have been explored in research papers e.g. storing small materialized aggregates (SMAs) per object column such as min, max, count, sum and histograms[37]. Brighthouse[43] defines a data

skipping index similar to Gap List. Their Character Map index could be easily defined using our APIs. Recently range sets (similar to our gap lists) have been proposed to apply data skipping to queries with joins[35].

Data layout research Many efforts optimize data layout to achieve optimal skipping e.g.[44], [20], [42], [38]. We survey those most relevant. The fine grained approach [44] adopts bit vectors as the only supported metadata type, where 1 bit is stored per workload feature. To obtain a list of features one needs to analyze the workload, inferring subsumption relationships between predicates and applying frequent itemset mining. This approach does not work well when the workload changes. To handle a UDF, the user needs to implement a subsumption algorithm for it, although this aspect is not explained in the paper. On the other hand, our framework enables defining a feature based (bit vector) metadata index, allowing feature based data skipping when applicable.
Both AQWA[20] and the robust approach[42] address changing workloads by building an adaptive kd-tree based partitioner which exploits existing workload knowledge and is updated when as the workload changes. AQWA focuses on geospatial workloads only whereas the robust approach handles the more general case. Data layout changes are made when beneficial according to a cost benefit analysis. Kdtrees apply to ordered column types, and generate min/max metadata only. Other layout techniques are needed to handle categorical data and application specific data types such as server logs and images.

Extensible Indexing Hyperspace defines itself as an extensible indexing framework for Apache Spark[15], although at the time of this writing it only supports covering indexes which require duplicating the entire dataset, and does not include any data skipping (chunk elimination) indexes. The Generalized Search Tree (GiST) [32], [36] focused on generalizing inverted index access methods with APIs such that new access methods can be easily integrated into the core DBMS supporting efficient query processing, concurrency control and recovery. Our work focuses on data skipping for big data where classical inverted indexes are not appropriate, and a different set of extensible APIs is needed.

Applications Prior work addressed specific applications such as geospatial analytics[39], [20], and range and k nearest neighbour (kNN) queries for metric functions e.g.[30], [28] without providing general frameworks.

5.7 Added features

In the third year of the project, the technology has reached the GA status of 3 (by the time of this writing) IBM Services. All of them include the extensible Data Skipping technology.

Additional and important features have been added, we succinctly present the most important ones

5.7.1 Database Catalogue

Independently from the Data Skipping technology, SQL queries against Object Stores through Apache Spark, suffer from a serious performance problem: the schema of the

dataset is discovered for each of the sent queries. Since schema discovery spawns a full dataset read, this can be a very serious performance problem.

This problem is addressed by the "Database Catalogue" feature also named "Hive Metastore".

For a given dataset, the database catalogue discovers once its schema and stores it. When other SQK queries are directed against the dataset, they address the database catalogue which stores the schema, thus additional schema discoveries are avoided. Further details concerning this feature are reported in [50].

5.7.2 Metadata store possible for Object Storage

As of D4.2 submission, we used ElasticSearch as sole possible metadata store. This solution, however very handy for a first prototype raised some questions for a product level solution. For instance, we should take care of the situations where the ElasticSearch data base is down or disconnected etc.

For all these reasons the metadata store was moved to be just in the Object Storage. We implemented this change.

5.7.3 Spark 3.0 and Dynamic File Pruning

In June 2020, Apache Spark 3.0 was released. This new version globally improved the performance of SQL queries by a factor of 2.

One of the major improvements was Data Skipping for Queries with JOINs as Apache Spark 3.0 introduced "dynamic partition pruning". As explained in [51], this technique permits to avoid partition scanning based on intermediate JOIN query results. However, this is only applicable when the JOIN key is on the partition column which covers probably a minority of the cases.

The term "Dynamic File Pruning" has been coined by Databrics and is a technique which improves Dynamic Partition Pruning by covering also the cases where the JOIN key is on a non partition column. This is a proprietary feature of Databrics when using DeltaLake format [52].

IBM is extending this capability to Spark native formats and is has started performance tests.

5.8 Conclusions

The described technology is the first extensible data skipping framework, allowing developers to define new metadata types and supporting data skipping for queries with arbitrary UDFs. This technology has been extended in the last year of the project to support and even go further than Apache Spark3.0 most advanced features.
Moreover, this technology enjoys the performance advantages of consolidated metadata, is data format agnostic, and has been integrated with Spark in several IBM products/services.

We demonstrated that our framework can provide significant performance and cost gains while adding relatively modest overheads and can be applied to a diverse class of applications, including geospatial and server log analytics. Our work is not inherently tied to Spark and could be integrated in any system with the ability to intercept the list of objects to be retrieved. Further work includes integration into additional SQL engines and automatic index selection.

.

6 Adaptable Distributed Storage

The purpose of the Adaptable Distributed Storage is to deliver a fully distributed storage layer across nodes that can be adaptable on the runtime in order to serve diverse workloads that can change during the deployment. It is based on the storage engine of the LeanXcale database. Its main functionalities are the ability for data fragmentation at runtime and the re-deployment of those fragments among the various nodes in order to balance the served workload. Moreover, the main target objective for this component is to provide fully elastic capabilities, and thus, being able to automatically self-adapt as the workload is being changed, without the need for intervention by a database administrator, while ensuring transaction semantics at the same time. During the first phase of the project (M18) the main focus was given on the implementation of the basic tools that will allow the Adaptable Distributed Storage to partition the stored data, and provide the ability for re-deployment in order to balance the incoming workload. Due to this, an internal experimentation took place in order to validate its functionality that has been already reported in the previous versions of that deliverable. In the beginning of the second phase of the project, we used the TPC-C standard benchmark to generate transactional workload and we demonstrated that this component can efficiently scale in/out, without affecting the overall responsiveness (in terms of average latency of incoming transactions) of the deployment, while on the meantime, ensuring transactional semantics and data consistency.

During the third and final phase of the project, we implemented all remaining functionality, that is summarized as follows:

- the ability to split datasets into data regions, and move those regions into other data nodes in order to balance the incoming workload
- implement an algorithm that solves the non-polynomial problem of optimal distribution of resources, that can decide on how to move those regions
- integrate this component with the infrastructure building block of BigDataStack, so that it can automatically request for the provision of additional resources, and thus, let the storage component scale out automatically.

In this section, we firstly briefly list the overall requirements, as were modified in the various phases of the project. We updated the design of the overall solution, by including the mechanism and protocol that is being used for letting the Data Services component interact with the Infrastructure building blocks in order to request for a scaling action, and vice versa. We document the additional functionalities that have been implemented in order to achieve the elastic scalability and we give details regarding the algorithm that we rely on to automate the suggestion of the deployments of the data fragments. Finally, we verify the functionality by moving regions across data nodes under heavy transactional workload.

6.1 Requirements Specification

The adaptable distributed storage is a fully distributed storage layer relying on the data nodes of the LeanXcale relational datastore, and its main purpose is to be able to

dynamically reconfigure both its resources and its data fragments in order to serve diverse workloads in runtime. Towards this, this component should be able to split the existing datasets in different fragments, move them across the data nodes in order to reduce the resource consumption in nodes that are over-consuming the available resources and under-performing, and request additional resources from the infrastructure, in order to scale out appropriately. It is important to be noted that all these operations must take place with no downtime and in a fully operational workload introducing a minimum overhead. As it has strong dependencies on the components of WP3, several requirements have been identified that both concern its internal functionality, as well as the interactions with these aforementioned components.

The following tables present the final list of those requirements, as identified during the last iteration of this deliverable where an initial version of the prototype has been developed but only tested locally and independently. These requirements are categorized both as mandatory for the delivery of the prototype, while others can be considered optional.

Table 18 - Requirement REQ-ADS-01 for Adaptable Distributed Storage

	Id	Level of detail	Type	Actor	Priority
	REQ-ADS-01	System	DATA	Developer	MAN
Name	Being able to fragment a dataset and move the data fragments across different nodes.				
Description	The adaptable distributed storage should be able to split a dataset into different regions, and move these regions to different data nodes, in order to adapt in cases of increased load (both in terms of user workload or data load) so as to achieve efficient consumption, based on the provided resources.				
Additional Information	When a movement (move, split, join) of a data fragment occurs, the storage must not suffer from a down-time. On the contrary, it must remain operational with minimum overhead on the overall performance.				

Table 19 - Requirement REQ-ADS-02 for Adaptable Distributed Storage

	Id	Level of detail	Type	Actor	Priority
	REQ-ADS-02	System	ENV	Developer	MAN
Name	Identify data nodes that are overprovisioning.				
Description	The adaptable storage must be able to identify data nodes that are overprovisioning their available resources and send internal alerts to trigger a dynamic reconfiguration of the deployment of the data fragments.				
Additional Information					

Table 20 - Requirement REQ-ADS-03 for Adaptable Distributed Storage

	Id	Level of detail	Type	Actor	Priority
	REQ-ADS-03	System	FUNC	Developer	DES
Name	Solve the non-linear resource allocation problem to suggest alternative deployment of the data fragments.				
Description	According to the available resources for the deployment of the data nodes and the stored data set, along with its split points that define data fragments, there is a non-linear resource allocation problem for the optimal deployment of the data fragments.				
Additional Information	As a non-linear, the solution of the resource allocation problem requires exponential time to be solved, which is not acceptable for run-time requirements. The provided solution should take into account possible acceptable solutions that can solve the problem and improve the resource consumption, under a minimum time interval.				

Table 21 - Requirement REQ-ADS-04 for Adaptable Distributed Storage

	Id	Level of detail	Type	Actor	Priority
	REQ-ADS-04	System	ENV	Developer	DES
Name	Be able to request additional resources from the infrastructure layer.				
Description	In case of overprovisioning of the resources, the adaptable distributed storage should be able to request additional resources from the infrastructure of BigDataStack.				
Additional Information	As noted in REQ-ADS-02, the adaptable storage must identify data nodes that are overprovisioning, and using REQ-ADS-03, it can suggest different distribution of the data fragments. However, there might be cases that this is not possible due to the overprovision of the whole system, and in such case, a horizontal scale out must take place. The adaptable storage should request additional resources, and grant them, if they are available. The communication should be as follows: 1. The adaptable storage requests an additional node with the specific requirements for resources. 2. The infrastructure responds if it can allocate additional resources for the storage. 3. The infrastructure informs the storage that the additional resources are now available. This requirement also includes the need from the adaptable storage to inform the infrastructure that it can release resources that are not needed.				

Table 22 - Requirement REQ-ADS-05 for Adaptable Distributed Storage

	Id	Level of detail	Type	Actor	Priority
	REQ-ADS-05	System	ENV	Developer	OPT
Name	Being able to release resources and adapt if resources are deallocated from the infrastructure.				
Description	There might be cases where the whole infrastructure is overprovisioning there are no more resources to be allocated to tasks. Then, the infrastructure might decide to reduce the overall resources of specific components, in favour of others that might execute some critical operations, or they have biggest priority at that point. The adaptable storage engine should be listening to the infrastructure for such cases and adapt accordingly.				
Additional Information	Once the adaptable distributed storage receives a request to release some of its nodes, then it should inform if it can do so: releasing some the data nodes, might result in not having the required amount of storage available for the dataset. In such cases, the adaptable distributed storage should respond to the infrastructure that this is not permitted, as this would lead to data loss. In case that this is permitted, then it should re-distribute its data load, and inform the infrastructure that the node is ready to be released.				

Table 23 - Requirement REQ-ADS-06 for Adaptable Distributed Storage

	Id	Level of detail	Type	Actor	Priority
	REQ-ADS-06	System	ENV	Developer	DES
Name	Inform the re-deployment component regarding reconfigurations of the data fragments.				
Description	As it is up to the storage itself to decide its optimal configuration of its data load, the re-deployment component cannot be aware of possible reconfigurations, which might affect the overall deployment of an application. Therefore, the storage should inform the re-deployment component about these actions.				
Additional Information	A message should be sent just before the re-configuration takes place, along with the setup, so that the re-deployment component can be notified and not take into account possible outliner monitoring information coming from this subcomponent. During this time, the re-deployment component should not modify any deployments that rely on the data set that is being re-configured. When the reconfiguration is finished, the adaptable storage should notify the redeployment component again, in order for the latter to start looking on the new monitoring information and decide upon possible redeployment of existed applications as well.				

Table 24 - Requirement REQ-ADS-07 for Adaptable Distributed Storage

	Id	Level of detail	Type	Actor	Priority
	REQ-ADS-07	System	ENV	Developer	MAN
Name	Re-establish connectivity with the monitoring subcomponent when a horizontal scaling action takes place				
Description	The adaptable storage engine exports its monitoring data to a specific place where the Prometheus, part of the monitoring subcomponent of BigDataStack can periodically pull and gather this information. Prometheus can be configured on where to pull this information upon its initialization. However, in cases of a runtime redeployment that takes place after a horizontal scaling action, information regarding the newly deployed nodes should also reach the monitoring component.				
Additional Information	There should be a monitoring proxy of the adaptable storage that will take the responsibility to send monitoring information to the target component. This proxy should encapsulate the details of the underlying deployment. It should gather all information of the data nodes, reconfigure itself to take into account newly deployed data nodes, and send everything to the Prometheus.				

Table 25 - Requirement REQ-ADS-08 for Adaptable Distributed Storage

	Id	Level of detail	Type	Actor	Priority
	REQ-ADS-08	System	ENV	Developer	MAN
Name	Enable a deployment of the data node component using Kubernetes				
Description	As the infrastructure of BigDataStack uses Kubernetes for deploying the various application/platform components, the adaptable distributed engine must be able to deploy and configure additional data nodes via this technology.				
Additional Information	N/A				

6.2 User Story

Today, companies are storing more data compared to years ago, which creates a need for systems capable of storing and processing so much information. The data generated and stored by companies has been exponentially growing during the last years. It is foreseen that by 2025 450 exabytes of data will be generated every day globally. The best-known technology to store and process data at the same time is a database. However, traditional databases cannot manage that huge amount of data, and as a result, an alternative

appeared during the last decade, called NoSQL. One key difference between traditional database systems and NoSQL datastores is that the latter can easily scale out their resources by adding additional nodes and redistribute their load. However, even if the ability for a storage system to dynamically adapt to diverse workloads by scaling horizontally the resources needed has been already implemented, scaling in/out a database introduces additional challenges: fragment a dataset to smaller portions and move these portions to different nodes for load balancing. Typically, NoSQL database systems can scale in/out sufficiently and move their regions across the available nodes, but they compromise the consistency of the data, that they never promised to offer. On the contrary, fragmenting tables of a relational data store and move the corresponding regions across the nodes will introduce significant concerns regarding data consistency, under OLTP workloads. Due to this, transactional datastores either never support elasticity, or when they do support, they suffer from long periods of downtime or significant decrease of their performance, and as a result, they cannot be considered as elastic, as they cannot scale efficiently at runtime.

In the scope of the Adaptable Distributed Storage of the BigDataStack, a novel mechanism is being implemented that allows the storage system of LeanXcale, which is relational operational datastore, to be provided with elastic scalable capabilities, thus being able to adapt to diverse workloads in run-time. This component will allow LeanXcale to partition its datasets to smaller portions (fragments) of data by splitting (or later merging them), and move those partitions effectively among the available data nodes, in order to achieve the balance of the load, both in terms of incoming workload, and in terms of the stored data load. In order to achieve this balance, this component will rely on the monitoring information that is being generated by the storage subsystem, and which provides useful insights regarding the resource consumption of each data region inside a node. Given that, a novel algorithm is being implemented, that solves the non-linear resource allocation problem, taking into account the multi-dimensional aspects of the problem to be solved: CPU, memory, storage, network consumptions. The proposed configurations of the algorithm and the ability of LeanXcale to distribute its load at runtime with no downtime or decrease of its performance, now allows to dynamically adapt and reconfigure the data regions to deal with increased workloads. Finally, based on that, the process of scaling in/out the data nodes of LeanXcale transparently triggers the reconfiguration process, making this component truly elastic.

6.3 User Perspective

In the final phase of the project, we decided not to restrict the use of the adaptable distributed storage to a specific use case, as its purpose is general and it is not targeting a demonstrator rather it is suitable for all scenarios that require from the storage to be adaptable. However, the three use cases of BigDataStack do have diverse variations of their workloads. For instance, the connected consumer scenario has quick and frequent variations during the day or the week, but it overall maintains the average level of expected traffic, , which is not the typical case for scaling out the data management component, as this will require the move of data elements that is slow, as it involves I/O access and actual move of data from one storage medium to another, in order to balance the load. In these

types of cases, scaling stateless components might be more beneficial. On the other hand, the ship management use case relies on the ship fleet that produces IoT data from numerous sensors deployed on each of its vessel. This data is pre-processed by different installations of CEP subsystems, cleaned and finally they arrive to the relational data store that has the responsibility to store them to the underlying *Adaptable Distributed Storage* component. In this scenario, changes in the workload are rather infrequent, but more severe, as the addition of just another vessel in the feel is rather permanent and will increase the overall incoming workload significantly and for a long period. Being inspired by the current pandemic crisis, we identified that this scenario has real value in real life situations: During the first wave of the pandemic, the DANAOS fleet had to stay in port due to global restrictions for traveling between countries. That has a a sequence that only a small part of the fleet was able to travel, and thus, the overall generated load had dropped significantly, having as a result the consumption of unused resources that were previously deployed. After the first wave of the pandemic crisis, given the steep increase in number of supported vessels and for each vessel the number of sensors as a result of the temporal removal of those restrictions, the continuous data ingestion started to impose again significant requirements in order for this component to be able to store all data. Currently, at the second wave of the pandemic, it is still unclear the level of restrictions that will be imposed by governments and how they will affect the number of vessels that will be able to travel, making extremely difficult to predict the requirements in terms of storage and computational resources for the data management layer. Moreover, the size of the required storage is constantly increasing, as new data are continuously ingested. Due to this, the *Adaptable Distributed Storage* component should have the ability to automatically scale in and out.

The cause for a scaling action can be either due to storage or due to the computational resources. Regarding the requirements for storage, IoT data collected through sensors will continue to grow, and thus, eventually the allocated storage resources will not be adequate to support the load. As a result, a new data node will be requested, and when the resources are available, the Adaptable Distributed Storage will split the regions and move some fragments to the newly added node, in order to balance the load. However, a scaling action might also be performed to acquire additional computational resources. In the scenario described above, the ship management can decide to change the number of its overall fleet. For instance, during the economic crisis of the previous decade, it was common for many shipping companies to liquefy their portfolio by selling part of their fleet (as global trades were also decreased) at that point, and invest in new vessels afterwards, in an attempt to get a bigger part of the global market. Selling and re-invest afterwards part of the fleet means that lesser or bigger amount of IoT data coming from the sensors on the vessels will be generated, which would need to be further ingested and processed in the data storage level. Thus, different needs for computational resources are required for the processing of the incoming workload, according to the number of vessels. In the case that additional vessels have been acquired by the shipping company, the computational resources allocated in the data storage level might not be enough to process the incoming load with the desired latency, so it might need to scale out. When additional computational resources are acquired by the data storage, in terms of additional data nodes, the Adaptable Distributed Storage will also split the data regions and move some fragments to the newly added node

for overall load balancing. It is important to highlight at this point that the Adaptable Distributed Storage can identify by its own when there is an evident saturation of resources (by monitoring the overall resource consumption on each node), and when this happens, it can now dynamically request additional data nodes from the infrastructure. The provisioning of additional data nodes makes its re-configuration engine to split, move and finally balance the ship management dataset among the available resources. This allows to be dynamically adapted to increase loads in terms of data.

Moreover, as described in the use case, data ingested to LeanXcale relational datastore will be periodically transferred to the IBM object store, making this information now outdated, so that it can be later used for analytical queries over historical data. As mentioned in more details in the Seamless section, IBM object store eventually imports this data and informs the LeanXcale data base about the successful ingestion. Then LeanXcale data base can safely discard this information from the storage, as it had been already available in the object Store. As a result, the LeanXcale data base will periodically perform a *vacuum* process, which is resource consuming (in terms of memory and computation usage) but which eventually frees storage resource. The ability of the Adaptable Distributed Storage to dynamically split regions and move them to different nodes allows this *vacuum* process to efficiently discard the data that has been now stored in the IBM object store, without downgrading the overall performance of the Seamless Analytical Framework.

6.4 Detailed Design

Figure 20 depicts the main architectural pillars of the adaptable distributable storage, along with the main components of BigDataStack that interacts. The adaptable distributable storage consists of the adaptable storage driver, the reconfiguration engine and the elastic manager. It mainly interacts with the components of the infrastructure management system. Main interactions are a) deployment of storage data nodes b) allocation of additional resources requests c) being triggered for forced release of already available resources d) accessing the monitoring information of the storage and e) updates to/from the re-deployment component concerning new configurations. Moreover, the figure depicts some internal built-in components of the LeanXcale relational data store that are involved in the functionalities of the adaptable storage. Finally, it is important to note that LeanXcale relies on its own distributed key value store (KiVi) which is used to persistently store data. KiVi consists of a metadata node, which contains all meta-information that describes the operational status of the storage, and various instances of data nodes. The key value store offers its own API for data access, allowing not only simple get/put operations, but also complex SQL-alike queries and aggregation operations. The metadata node of KiVi is part of the configuration component of the LeanXcale, while on the other hand, each KiVi data node co-exists with an instance of the Query Engine, in order for the latter to exploit the locality of the data in each node. As a result, the LeanXcale datanodes consist of both a KiVi data instance, which contains a fragment of the dataset, along with a Query Engine instance. The adaptable storage manages the scalability of these datanodes.

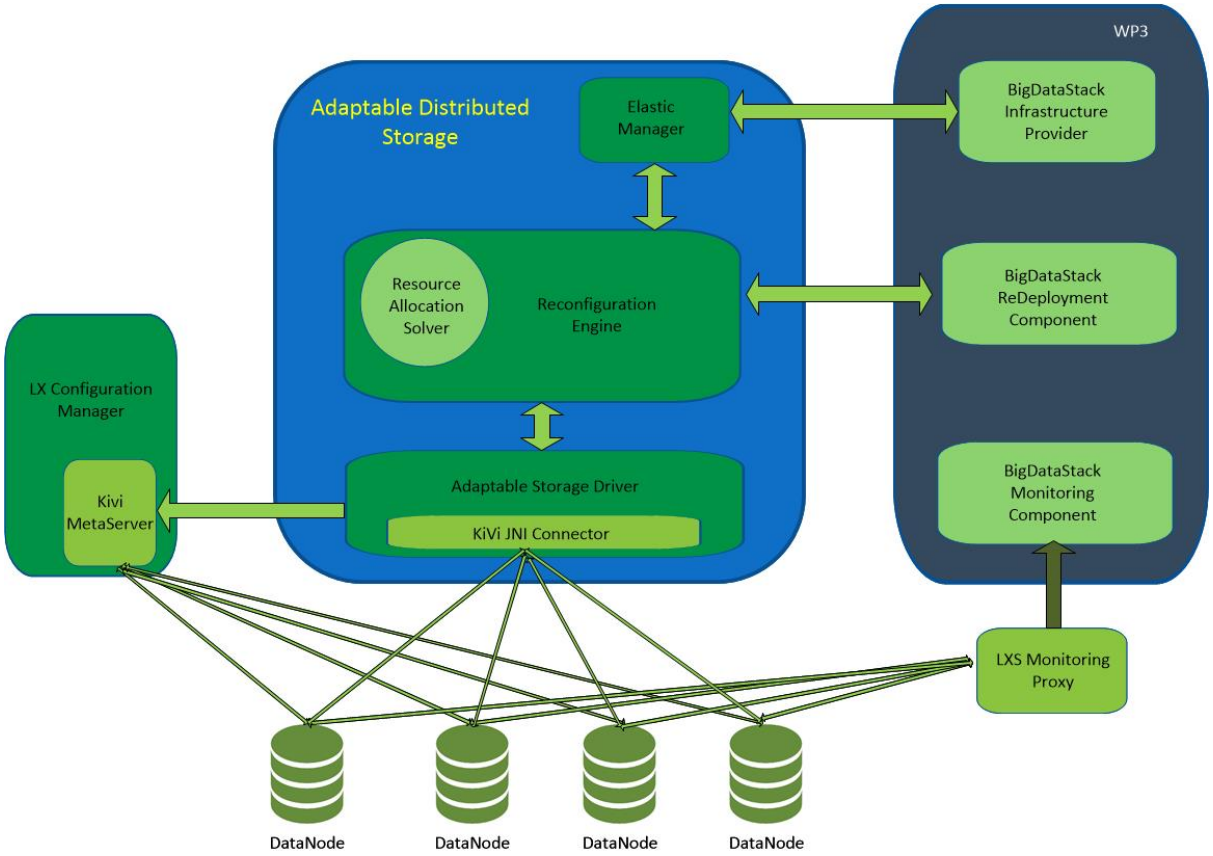


Figure 20: Adaptable Distributed Storage architectural design

The main purpose of the adaptable distributed storage is to efficiently manage the cluster of the underlying data nodes. The information regarding the current configuration of the nodes is managed by the LeanXcale internal Configuration manager, which contains the metadata server of its internal storage engine. The tools that provide the basic pillars of the adaptable storage, which are splitting a dataset into different data regions, move the regions across the data nodes or merge two data regions, are included in the Adaptable Storage Driver. The latter provides Utility methods to the upper layers that implement the business logic of this component, thus dynamically reconfigure the deployment of the data fragment, by splitting/merging/moving the existed datasets among the data nodes of the LeanXcale at runtime and provide elasticity capabilities by scaling in/out the available resources of the storage when necessary. Its implementation is written in Java, and therefore, internally makes use of JNI[28] calls to allow this subcomponent to call the required bindings which are written in C and handle internally the details of the data movement.

The reconfiguration engine implements the business logic regarding the steps and decisions that should be taken in order to maintain the efficient consumption of the available resources. It can detect hot spots, meaning data nodes require more than their available resources. When it identifies such situations, taking into account the overall available resources, it submits to its internal Resource Allocation Solver the corresponding non-linear

28 https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html

problem. The latter suggests different configurations of the data fragments, and if possible, the Reconfiguration Engine executes all necessary steps.

In case that the whole system exceeds the overall available resources, and no alternative configurations can balance the system according to the available resources, the Reconfiguration Engine requests additional resources for the system from the Elastic Manager. The latter will ask from the tools of the WP3, additional resources, and will reply back when these resources are available (or if the request was rejected), while additionally, it will deploy the data node software elements to the new allocated resources. When this process is finished, the Reconfiguration Engine will consult again the Resource Allocation Solver in order to get updated configurations, and once a new configuration is received, it will drive process to dynamically redistribute the fragments of the dataset, following the same flow as already described. It is important to note that the Elastic Manager can also receive requests from the infrastructure component of WP3, in order to release already allocated resources, dealing with cases that the whole BigDataStack platform is lacking of resources and other tasks requirements are estimated to be more important. The Elastic Manager will reply if the storage can release some of its resources (i.e. the available storage should be bigger than the overall size of the datasets, otherwise it would lead to data loss), and if yes, it will force the Reconfiguration Engine to redistribute the load.

Finally, the LeanXcale monitoring proxy can receive monitoring information from all the underlying data nodes, taken into account scenarios when a re-deployment takes place which leads to a horizontal scalability action. As additional data nodes are being deployed, the monitoring proxy takes into account all the connectivity details with the new nodes and will serve as the central point for pulling data from, by the monitoring subcomponent of WP3. During a dynamic reconfiguration, the Engine will inform the deployment component of the WP3 that such a process is taking place, so that it can ignore potential outliner monitoring information, and prohibit any redeployment of an application, and when the reconfiguration finishes, it will be informed again so that it can continue consuming monitoring data and check for candidate redeployments on the application level.

Integration with Infrastructure building block of BigDataStack

This sub-section describes the mechanism and protocol for the communication of the Adaptable Distributed Storage with the Infrastructure building block of BigDataStack. Even if this section focuses on how the former has been integrated, this mechanism follows a generic approach and can plug in any of the Data Services that are presented in this document. For instance, the Real-Time Complex Event Processing also interacts with that. As it is depicted in the following figures, there have been identified three different scenarios, all of them have the same 4 actors:

- The Infrastructure: As the Data Services are agnostic of the underlying infrastructure and vice versa, with the term *infrastructure* we define the overall underlying building block as a whole, without getting into details on which components interact with each other, as this is out of the scope in this document

- WP3 End: This is the endpoint of the infrastructure level, which is exposed by the deployment component of WP3. It encapsulates the details and communication flow that happens in that level and it is unique point of interaction with the Data Services.
- WP4 End: In the same sense, this is the endpoint that the Data Services are exposing to the infrastructure building block for communication. It has been implemented by the *Reconfiguration Engine*, but its scope is general, and can be also used by other components of the Data Service Layer.
- Data Service: We define with the term *Data Service* each of the components of this layer that provides the ability to scale out, either independently, or by the demand of the infrastructure. It can be either the Adaptable Distributed Storage component, or the Complex Event Processing. The implementation is generic and provides a set of interfaces that each data service needs to implement according to its specific characteristics. However, the set of methods are the same. In this sub-section, we will describe what these methods are doing, and in the corresponding sub-sections, there will be described how these methods have been implemented by each of the different components.

The three different scenarios that have been identified are the following: Infrastructure identifies a bottleneck and a saturation of resources and gives an order for a Data Service to scale out, by creating additional resources and provide them to the latter. Secondly, the Data Service itself identifies the need to scale out and requests the infrastructure for the provision of additional resources. Finally, the Infrastructure needs to withdraw some resources of a Data Service, because they are more crucial to be provided to another layer of the software stack. Therefore, it informs the data service to scale in, and release its resources. A fourth scenario would imply the data service to inform the infrastructure that it can be safely scale in, however, this is a combination of the aforementioned three and it has not been described here.

It is important to mention that when a scale out action is taking place, the infrastructure creates a number of additional Kubernetes *pods*, interacting with the OpenShift orchestration framework. All Data Service components have been configured as *Stateful Sets* according to the Kubernetes terminology, The newly created *pod* contains the binaries of the Data Service, which usually contains several components of the service, however, this is totally agnostic to the infrastructure that does not know how to configure the new deployment. Therefore, when a new *pod* is available, there is the need for the Data Services themselves to proceed to additional actions and install and start the corresponding components that are needed, while additionally there might the need to move data fragments from one *pod* to the other, in order to balance the overall workload.

The three scenarios are described in more details as follows:

Infrastructure requests a Data Service to scale out

The sequence diagram of the flow of interactions between the components are the depicted in Figure 21.

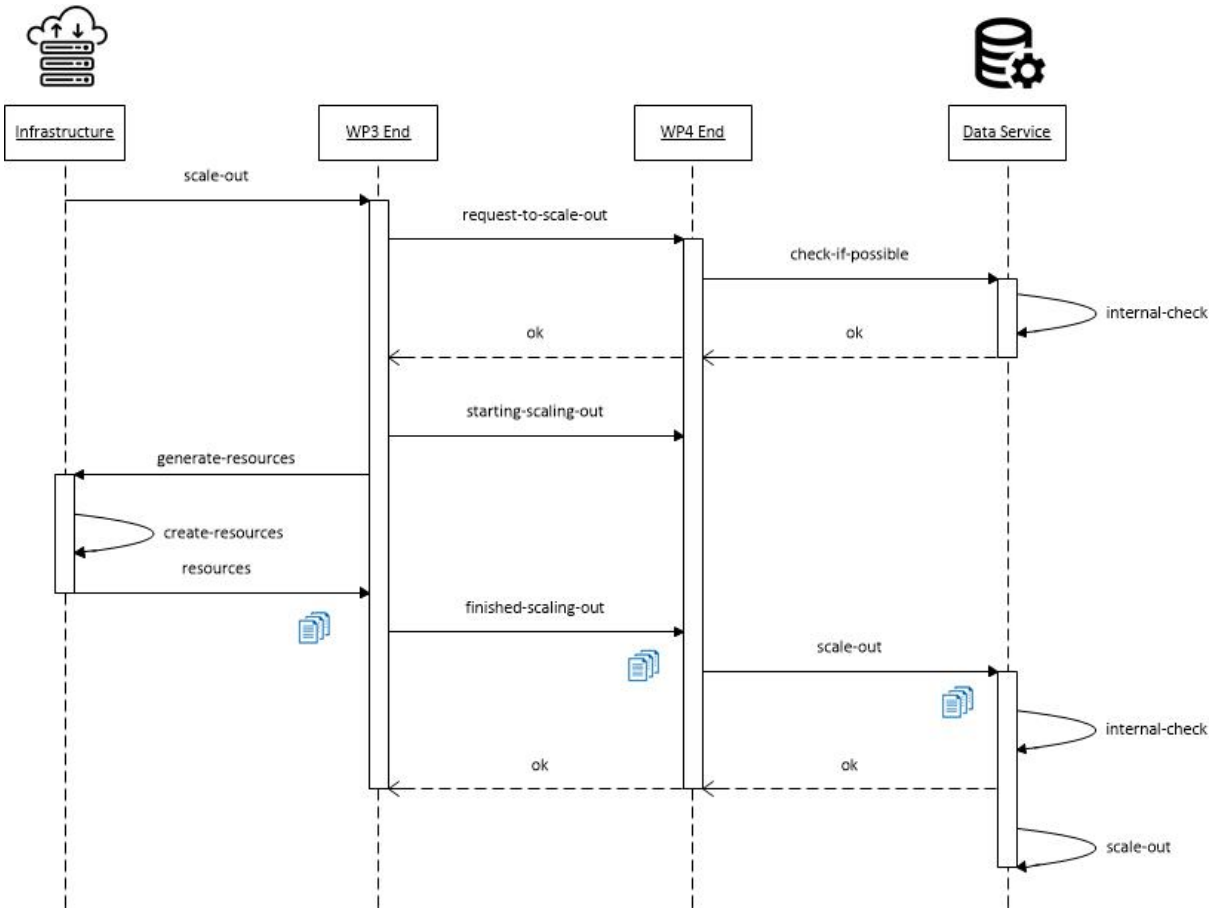


Figure 21: Infrastructure requests a Data Service to scale out

The infrastructure identifies the need for a Data Service to scale out. This decision was made after examining the monitoring information gathered by all components of the application, and it was noticed that some KPIs have been violated, and according to the scalability rules, the scale out of a Data Service was the best option. It informs the WP3 End who maintains the business logic of this mechanism from the infrastructure building block. Then the WP3-End starts a *session* for this activity and sends a request for scaling out in the WP4-End. From the Data Service point of view, it must be verified if the latter is able to perform a scaling action at that point. In fact, the WP4-End firstly notifies the data service itself, that has to make an internal check. If for whatever reason decides that this is not possible at the given time (i.e. the end-user might decide to drop a data table, which will imply the movement of other data regions in order to equally distribute the data load and no other movement is allowed until this action finishes), then replies with a no, and the process ends there. The infrastructure will check other alternatives like scaling out another component of the application, and if none achieves the goal, it can retry at a later phase. If the data service agrees that it is able to scale, this is communicated to the WP3-End, and after this

handshake, it starts the process of scaling out and informs the WP4-End. In parallel, it communicates with the internal components of the infrastructure, which interacts with the OpenShift in order to create a number of additional resources, in terms of *pods* with possible *persistent volume claims*. The details of this process are out of scope of this deliverable, and there have been reported in the corresponding reports of WP3. When the new *pod* is created, the infrastructure sends this information to the WP3-End which communicates with the WP4-End part. This information contains the IP of the new *pod,* its name which is also the hostname of the *pod,* the id of the created *persistent volume claim* etc. If there is a failure from the infrastructure and the additional resources cannot be created, WP3-End notifies the WP4-End about this failure and the process terminates. This might imply that the Data Service might need to release any possible *locks* that it might have issued to prevent administrative actions. In case that the new resources have been created successfully, due to the distributed design of this mechanism, the WP4-End should re-check with the Data Service that it is capable to perform the scaling action. Even if the latter had blocked administrative actions (like creating/deleting data tables) to happen, as it was previously informed for a pending scalability action, other unpredicted issues might have been raised up that prevents the data service to scale at that point. For instance, a possible failure in one of its nodes might have triggered the recovery process, which blocks all other actions until the data service is fully recovered. If this is the case, the WP4-End replies to the WP3-End with a message informing that the scalability action has been rejected. The WP3-End then communicates with the infrastructure to release the newly created resource (to destroy the pod). If, however the Data Service is capable to scale out, the WP4-End returns a successful message to the WP3-End, and the whole process terminates. The Data Service however, needs to start installing and starting the necessary components inside the *pod,* and to perform whatever additional action is necessary, like moving data regions etc. The details of this process vary according to the specifics of each of the Data Services and will be explained at the corresponding sub-sections.

Infrastructure requests a Data Service to scale in

The sequence diagram of the flow of interactions between the components are the depicted in Figure 22:

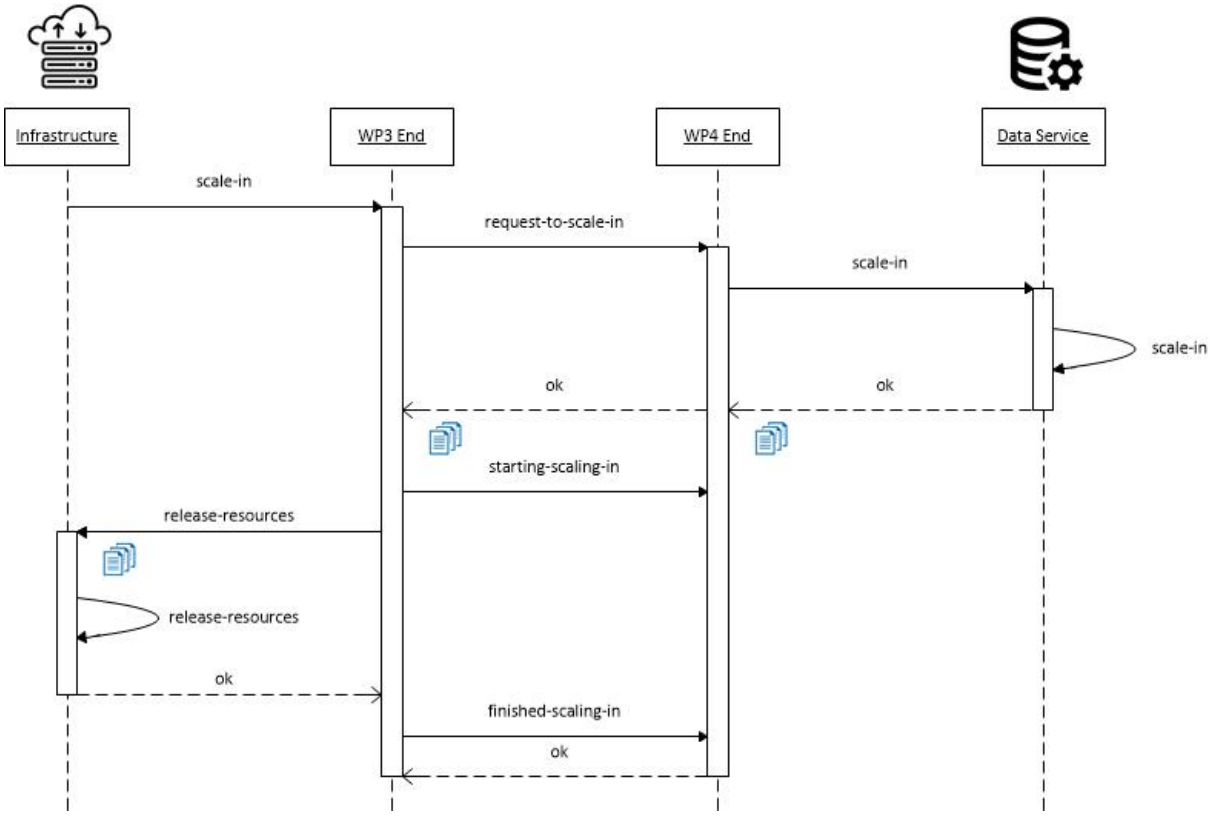


Figure 22: Infrastructure requests a Data Service to scale in

In this scenario, the infrastructure identifies the need to scale in the resources of a Data Service for a variety of reasons: Either it has identified via its integrated monitoring subsystem that it is overprovisioning resources to the data store, meaning the data store can also perform with no problems with lesser resources, which in fact indicates that the platform is overspending at that point, or the infrastructure is getting saturated of resources and it needs to reduce the resources of some components in order to be able to host or scale out others. If for whatever reason decides to reduce the resources of a data service, it communicates with the WP3-End that starts a new *session* to handle this request. The request is sent to the WP4-End that asks for the Data Service to scale in. The latter might decide that it cannot due to numerous reasons (i.e. an administrative action is taking place at that point, a recovery process is under execution, or simply there might be not enough storage space if it releases a data node, which will lead to data loss that simply cannot happen), so this rejection is forwarded back to the WP3-End, and the process terminates there. The infrastructure should ask for other components to reduce their resources. If however the data service accepts the request to scale in, it firstly redistributes the data load in order to clean storage elements from holding data, and then it sends the *pods* that can be released by the infrastructure. When this information reaches the WP3-End, the latter starts the same process as in the previous scenario. It informs the WP4-End first that the scaling action is about to begin, and then it communicate with the infrastructure components, that are interacting with the OpenShift, in order for the latter to terminate and destroy these pods. When this is finished, the WP3-End sends a notification that the process has been

succeeded and it terminates. From the Data Service point of view, there is no additional action that needs to perform. In case that the release of the resources by the infrastructure fails, this is indifferent from the Data Service point of view. The infrastructure can retry as many times as it is required.

Data Service requests the Infrastruture to scale-out

The sequence diagram of the flow of interactions between the components are the depicted in

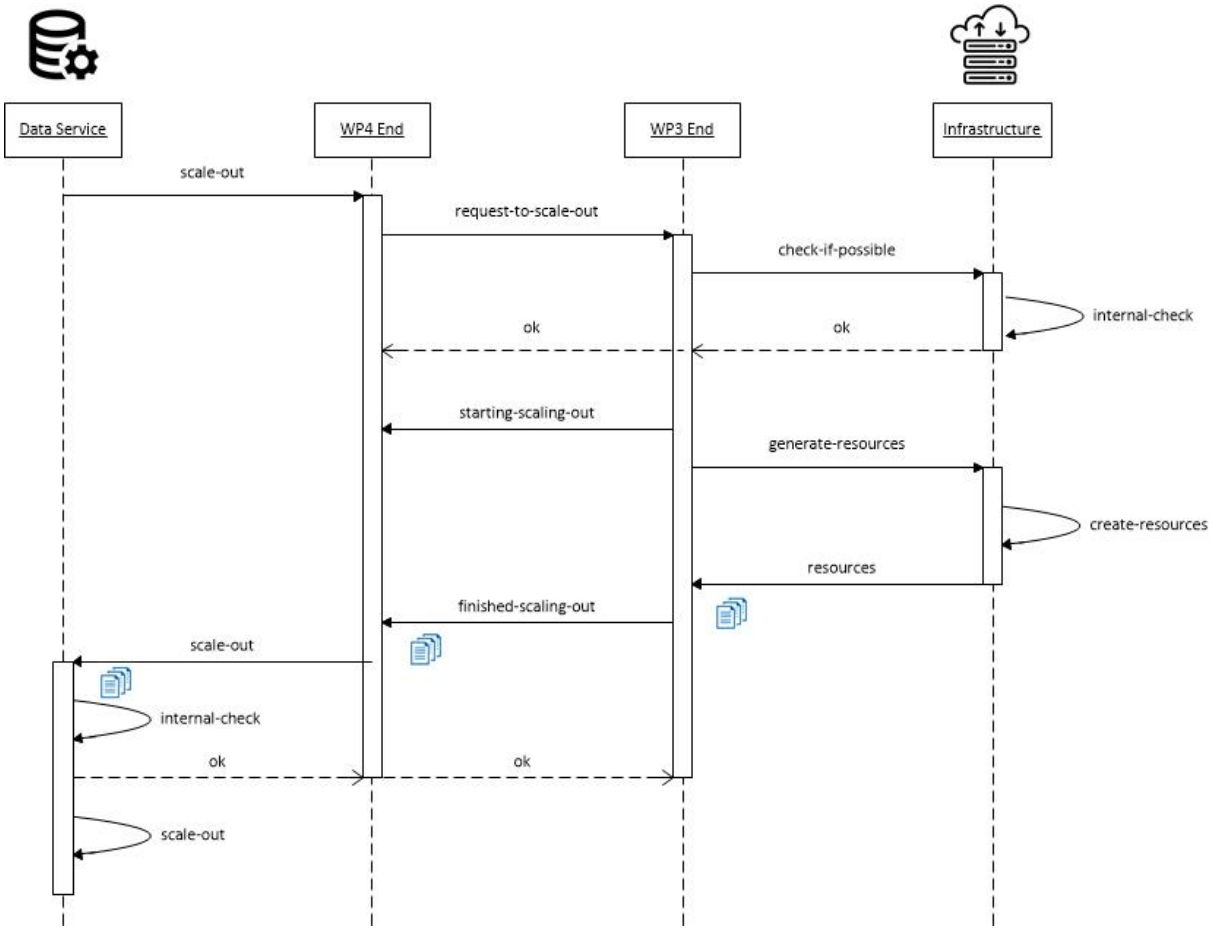


Figure 23: Data Service requests the Infrastructure to scale-out

The Adaptable Distributed Storage component has in an internal monitoring mechanism and its reconfiguration engine is capable to identify *hot spots*, that is data region that are being frequently accessed and demands increasing needs for computational resources, or they are becoming big enough that soon will not fit into the persistent storage. For this, the provision of an additional data node is required, which implies the need to scale out. The reconfiguration engine sends this request via the WP4-End to the infrastructure, which firstly checks if the scale out is possible. The request might be rejected for instance due to the lack of resources. As the request from a data service to scale out is of highest priority due to potential data loss, the infrastructure should start checking and requesting other components to scale in and release their resources, so that they can be available to the data

service. The latter, in case of such a rejection, it continuously sends these requests until they are eventually accepted. When this happens, the WP3-End starts the *session* by informing the WP4-End that a scaling action has been started, and then, this scenario is the similar to scenario 1.

6.5 Prototype

The goal for the first phase of the project was the implementation and experimentation of the core functionality of the Adaptable Distributed Storage, mainly it is fundamental pillars that are its ability to split/join/move data regions in different data nodes, under intensive operational workload. Additionally, the *adaptable storage driver* had been implemented, delivered as a library written in Java, that allows for the programmatically control of the behaviour of the storage engine and instruct it on how to proceed to the corresponding actions. As a result, in M18 the basic functionality had been delivered =which allowed the development of the *Reconfiguration Engine* during the second phase that has implemented the business logic of the component, along with the integration with the WP3 components in order to allocate/de-allocate resources when needed.

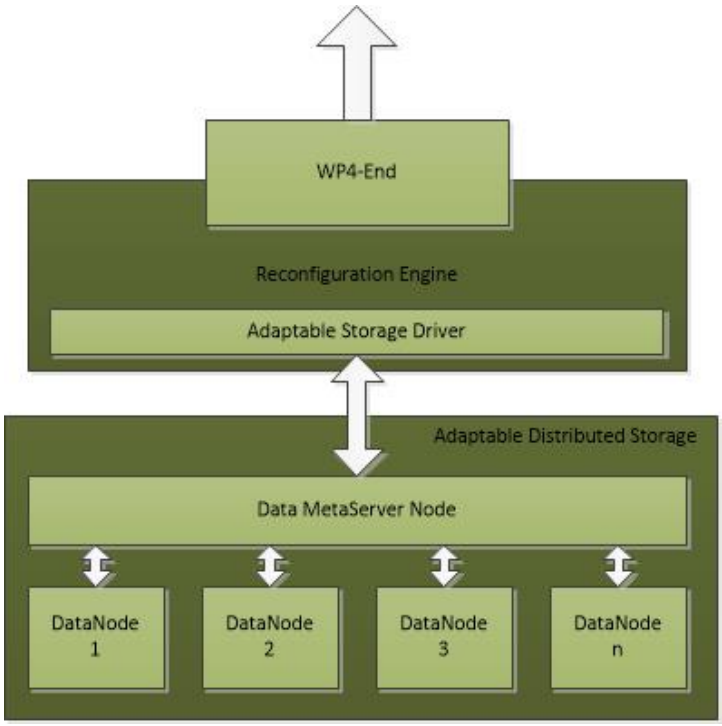


Figure 24: Main Building Blocks of Adaptable Distributed Storage

As it is depicted from Figure 24, the Adaptable Distributed Storage, which is part of the LeanXcale datastore solution, consists of a list of data nodes, where the data itself is persistently stored into data regions, and a data meta-server node, that holds information about the distribution and deployment of those regions among the data nodes. The internal details of the communication between those components is out of scope of this document. However, the metaserver node is providing an API via RPC so that can be programmatically

configured. This code has been written in *C* programming language, and there is a JNI binding written in Java that can be used instead. This bridge is the role of the *Adaptable Storage Driver,* as it can be depicted in Figure 20. As a result, the *Reconfiguration Engine* makes use of this driver in order to ask the status of the storage (i.e. in case it needs to verify if a scalability action is permitted), and to move regions from one data node to the other, in case of a scale out, so as to balance the overall data load. Moreover, the WP4-End component, as explained in the description of the previous scenarios, is the endpoint for the communication with the Infrastructure building block of BigDataStack. It exposes a REST API and accepts requests from the infrastructure for scaling in or out the data services. It implements the business logic of this mechanism that allows the automatic scalability of the Data Service components and provides specific interfaces that each of one of those must implement. By doing so, it hides the complexity of each individual component, and provides an abstraction over this layer. Therefore, it can be considered as part of the Reconfiguration Engine, which makes use of it and implements that interface. Finally, part of the Reconfiguration Engine is the Resource Allocator Solver, which is an algorithm that decides over the optimal deployment of data regions when a new node has been provided.

The following sub-sections describes in more detail the methods and interfaces that have been implemented by each of those subcomponents of the Adaptable Distributed Storage.

6.5.1 Data services-end

This subcomponent provides a REST API for communicating with the Infrastructure building block. The web methods are the following:

The infrastructure should send a request for a scalability action. It awaits for a POST request at the following URL:

http://ads-deploy-route-wp4.apps.moc.bigdatastack.com/wp4.request

An example of the input message of the body of the request can be the following:

```
{
  "sessionID": "UUID",
  "application": "lxs-store-eroski",
  "applicationType": "LXS",
  "newReplicas": 1,
  "existingReplicas": 2,
  "queryName": null,
  "subQueryName": null,
  "subQueryInstanceId": null
}
```

The infrastructure opens a new session with a given uuid, telling that it needs to scale the application ‘lxs-store-eroski’ that is an instance of type LXS, which refers to the adaptable distributed storage component. The number of existing nodes of the storage now are two, and it will be scaled out by adding one more node. If the number of the parameter *newReplicas* was negative, that would have implied that a scale in action is requested and

the storage should release 1 instance. The additional parameters are ignored by the Adaptable Distributed Storage, as they are meaningful to the Complex Event Processing only.

In case that the data service accepts this request, the WP4-End will respond with the following message:

```
{
  "sessionID": "UUID",
  "application": "lxs-store-eroski",
  "applicationType": "LXS",
  "answer": "YES"
}
```

The *answer* attribute can have values: YES, NO, WAIT. In case of WAIT, then additional attributes will be returned, defining the number of time (in period units) that the infrastructure should wait before retrying.

Inform the data services that the infrastructure has started creating the additional resources to be provided. It should send a POST request to the following URL:

http://ads-deploy-route-wp4.apps.moc.bigdatastack.com/wp4.action

An example of the input message of the body of the request, with respect to the session that has been already opened before, will be the following:

```
{
  "sessionID": "UUID",
  "application": "lxs-store-eroski",
  "applicationType": "LXS",
  "action": "START_SCALING"
}
```

The WP4-End accepts the message, and returns with the following:

```
{
  "sessionID": "UUID",
  "application": "lxs-store-eroski",
  "applicationType": "LXS",
  "type": "OK"
}
```

The attribute *type* can have one of the following values: OK, WAIT, INTERNAL_ERROR.

The same web method can be used to inform the Data Service that the scaling has been finished and gives the parameters of the newly created pod. The input message should be the following:

```
{
  "sessionID": "UUID",
  "application": "lxs-store-eroski",
  "applicationType": "LXS",
  "action": "FINISH_SCALING",
  "deploymentInformation": {
    "ip": "102.36.2.214",
    "pod": "lxs-store-eroski-3",
    "pvc": "pvc-lxs-store-4fcab3",
    "queryName": null,
    "subQueryName": null,
    "subQueryInstanceId": 0
  }
}
```


Now the Data Service has all the information it needs. It has been provided with an additional pod named *lxs-store-eroski-3* with the given IP, which has the given persistent storage attached. The additional attributes are irrelevant for the Adaptable Distributed Storage, as they are meaningful only to the Complex Event Processing component. After receiving this information, the Data Service can start its internal process for scaling out internally to the new node. More details will be given in the following sub-section.

It is important to mention that the *action* attribute of the input message can have one of the following values: START_SCALING, FINISH_SCALING, FAILED_SCALING, START_SCALING_DOWN, FINISH_SCALING_DOWN, FAILED_SCALING_DOWN.

Finally, there is an additional web method that can be invoked periodically for each of the components to check the status of the other. This might be useful in terms the Data Service needs to perform an immediate action and therefore, it cannot be scaled out at that moment. Therefore, the Infrastructure can continuously check the status, during the different phases that are required internally to create a new resource. It should send a POST request at the following url:

http://ads-deploy-route-wp4.apps.moc.bigdatastack.com/wp4.status

The following message can be given as input:


```
{
  "sessionID": "UUID",
  "application": "lxs-store-eroski",
  "applicationType": "LXS"
}
```


And an example response can be the following:


```
 {
  "sessionID": "UUID",
  "application": "lxs-store-eroski",
```

```
  "applicationType": "LXS",
  "status": "TEST",
  "message": "this is a mock message"
}
```

Those methods have been documented via the swagger[29] tool, which autogenerates the web methods description in JSON files. Those files can be used by software developers to generate the stub classes/code that implements the façade of the web services. It can be used by the UI of the tool which builds a testing framework for the data service developers to quickly interact with the Infrastructure and see the behavior of their implementation. Figure 25 shows the list of web methods exposed using the swagger user interface.

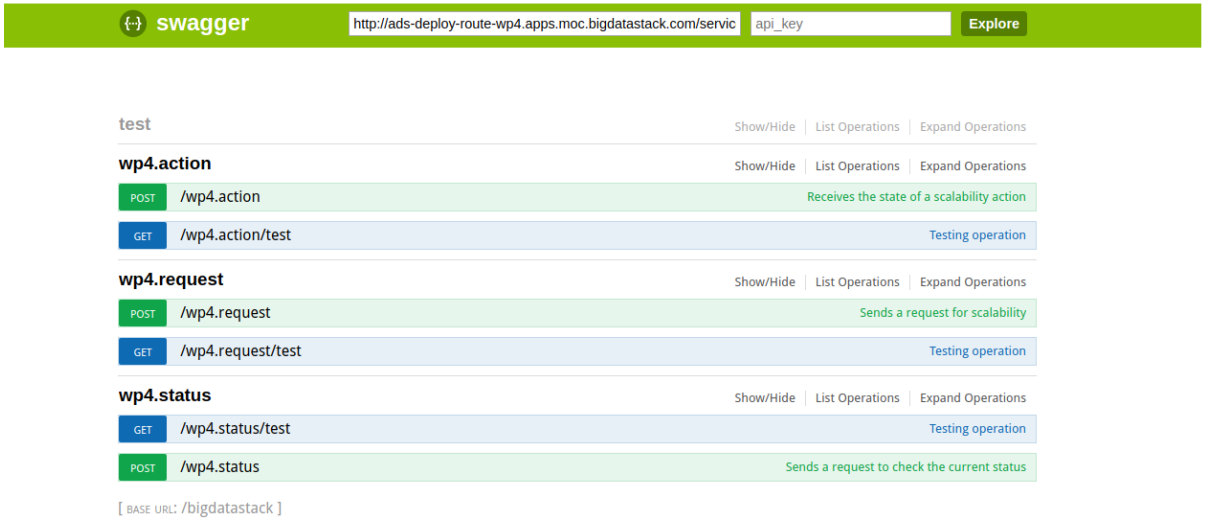


Figure 25: An Overview of the implemented web methods using swagger

Figure 26 also shows the usage of the swagger tool to invoke a specific web method. The data service developer can insert the input message in the corresponding textbox, thus mocking the behavior of the Infrastructure building block. By clicking the button *'try now'* she can directly send this message in the body of a POST request, which will invoke the deployed web service of the component. The swagger will show the return message of the request, while the implementation of the reconfiguration engine will execute its corresponding code for scaling out.

[29] https://swagger.io/

e

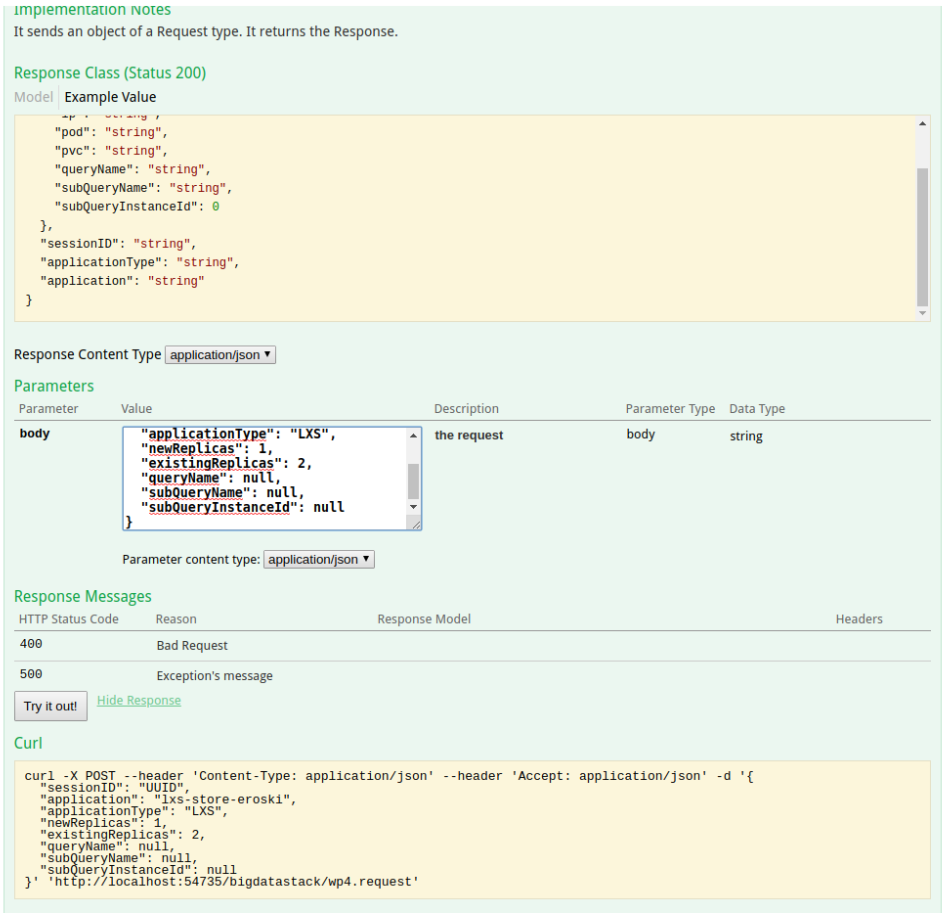


Figure 26: Example of using swagger for experimentation

6.5.2 Reconfiguration Engine

When the WP4-End component receives the aforementioned messages from the Infrastructure building block, it invokes the corresponding interfaces in order for the layer above to proceed with the details of each specific Data Service. The *Reconfiguration Engine* of the Adaptable Distributed Storage implements the specific logic that is targeting this component. The methods that are defined in the interface and implemented, along with a short description of the logic, are the following:

- boolean canYouScale(String sessiongID, String application, int newReplicas, int existingReplicas, String queryName, String subQueryName) throws DriverException: This method checks if there is any process ongoing in the system that prevents us from scaling. If the system is already doing another scale operation or a recovery process the response will be false, otherwise, it will be true.
- boolean infrastructureStartsScaling(String session, String application) : This method will launch the scaling procedure that prepares the database to scale out. This procedure will analyze all the data regions in the database and select the most appropriate data regions to move. This procedure may also split a region if none of

the regions will fit in the new node. Additionally, this method acquires a lock that will prevent us from accepting any other scaling request. When the procedure that decides the data region that will be moved has finished analyzing all the data regions, the method returns true. It is important to highlight here, that in case that none of the existing regions are capable to move and a split operation needs to take place, this operations starts asynchronously at the moment that the method returns, in order to take advantage of the time the infrastructure needs to provide the additional resources and prepare the datastore to do the movement, when the resources are ready.

- boolean infrastructureFailedScaling(String session, String application) : This method will release the lock that prevents us from new scaling requests. In case of a previous data split might have occurred, this method will also notify the corresponding procedure that has prepared the data regions that should have been moved to undo the changes this procedure may have done.
- boolean infrastructureFinishedScaling(String session, String application, String ip, String pod, String pvc, String queryName, String subQueryName, int subQueryInstanceId) throws DriverException: This method starts the instances of LeanXcale in the new pod and registers them in the system. When all the instances of this new pod have been started, the move of the data regions decided previously is done. After moving the data regions and balancing the data load, the global lock that was preventing us from any other scaling operation is released. This method returns synchronously in case the actual movement of the regions can take place (i.e. a data node has not been crashed in the meantime that would have triggered a recovery process) and performs the actual movement asynchronously.
- boolean infrastructureFinishedScalingDown(String session, String application, String ip, String pod, String pvc, String queryName, String subQueryName, int subQueryInstanceId) throws DriverException: This method checks if there is already an ongoing scaling operation or a recovery process. If this is the case, this method will return false. Otherwise, it will check if scaling down is possible by analyzing the free resources of the system. The main resources that have a direct impact on this decision are CPU usage, memory usage, network usage, and disk space free. If the system does not have enough resources, the method will return false. This method starts the scaling in operation in case that the system has resources enough to scale down. It then acquires a global lock that prevents from running any other scaling operation concurrently and decides the pod that will be stopped. Then, an internal procedure is called. This procedure decides where to move the regions depending on the resources that remain in the system. Finally, all the data regions movement is done ensuring that all the regions will keep in the remaining pods and stop all the LeanXcale instances of the pod that will be removed. This method returns true after the movement of the data and cannot be done in an asynchronous manner, as the infrastructure might release the resources, before the data have been moved, leading to a data loss. However, as the movement process might take long, it is impossible for the web method to be blocked until the process finishes, as this can lead to a timeout of the requested. Therefore, it returns a WAIT response, with an

indicative time to wait. The infrastructure will need to invoke again this method with the same UUID, and the method checks if the process has already been finished. If not, it returns again a WAIT response with the indicative time, and if yes, it returns a YES response, with the information of the pod that can be dropped, which will allow the infrastructure to continue release this resource.

- boolean infrastructureStartsScalingDown(String session, String application) throws DriverException: This method does not implement any specific logic on the storage level, rather than holding the information that the resource will be eventually released.
- boolean infrastructureFinishedScalingDown(String session, String application) throws DriverException: This method will release the global lock that prevents the system from running concurrent scaling processes.
- boolean infrastructureFailedScalingDown(String session, String application) throws DriverException: This method will undo the data movements that may have been already done.
- String getStatus(String sessionID, String application) throws DriverException: This method analyses the current status of the database and returns if the database is already doing a scaling process or doing another process that blocks a new scaling process.

6.5.3 Adaptable Storage Driver

Regarding the *Adaptable Storage Driver*, the following methods are indicative on functionalities that have implemented and are used by the reconfiguration engine to split, move, merge regions and decide on which data regions need to be moved:

- public void addKvds(String tableName, String kvds, String max, String min, boolean redistribute): Adds a data node to a table, generating a new region and assigning to it. This method cannot operate with replicated tables. Params:
  - tableName: Full name of the table
  - kvds: ServiceIp of the KVDS server to be added
  - max: (Only needed if redistribute = true) an estimation of the maximum key value. It is used to estimate the size of the regions.
  - min: (Only needed if redistribute = true) an estimation of the minimum key value. It is used to estimate the size of the regions.
  - redistribute: Indicates if the size of the existing regions needs to be distributed to assure a uniform distribution of all the region
- public void addKvdsList(String table, String min, List<String> kvdsList): Adds a list of kvds to a replication group for a region, and replicates it in all of them. If there is no replication group for this region, a new one is created. If some kvds is already in this replication group, it's ignored. Params:

  - table: The full name of the table which owns the region
  - min: Lower limit of the region
  - kvdsList: The list of kvds to be added to the replication group.
- public void deleteReplicationGroup(String table, String min, String lastKvds): Deletes a replication group, deleting also the region of all its kvds, except the one passed as a parameter. This method converts a replicated region into a single one. Params:
  - table: Identifier of the region owner table
  - min: Represents de the lower limit of the region.
  - lastKvds: The kvds in which the region won't be deleted.
- public void cloneKvds(String source, String dest): Clones a data node from a source to the destination. Params:
  - source: The serviceIP of the source of the data node to be cloned
  - dest: The serviceIP of the destination that the data node will be cloned
- public List<ReplicationGroup> getRegionsByServer(String kvds): Retrieves the list of replication groups in which a server participates. Returns he list of Replication group objects defined for this server. Params:
  - kvds: ServiceIp corresponding to the server
- public List<ReplicationGroup> getRegionsByTable(String table): Retrieves the list of replication groups for all the regions in a table. Returns the list of Replication group objects defined for this table. Params:
  - table: Full qualified name of the table.
  - public List<String> getReplicas(String table, String min): Returns the list of kvds in where a region is replicated. Params:
  - table: Identifier of the owner table
  - min: Lower limit of the region
- public void joinRegion(String table, String point): Joins two adjacent regions of a table, using a point to identify the first of them. The parameter represents the min key of the upper region to be joined. In case of be a composed key, the point parameter must be a sequence of field values separated by a ','. In case of the two regions are replicated, the resultant region is replicated in the kvds belonging to the replication group for the first region, and all the replicas belonging to the second region are removed. If the first region is not replicated, the resultant region won't be replicated independently of the second region replication. Params:
  - table: Full name of the table to be joined.
  - point: table's primary key fields.
- public void moveRegion(String table, String min, String source, String dest): Moves a region from one server to another. Params:

  - table: Full qualified name of the table.
  - min: Min key value for the region
  - source: Source server for the movement
  - dest: Destination server of the movement.

- public void moveRegion(String table, String source, long fileId, String dest): Moves a region, specified by a server and its local file id, from one server to another. Params:
  - table: Full qualified name of the table.
  - source: Source server for the movement
  - fileId: File identifier of the replica in the server passed as a parameter
  - dest: Destination server of the movement.
- public void splitRegion(String table, String splitPoint, List<String> destKvds): Split the given table using a splitpoint that represents a possible key of the table, and move the new region to a destKvds list passed as parameter. The parameter *splitPoint* should contain the values for the fields in the primary key separated by the key separator character. For example, given a table with a composite primary key made up of an integer and a string, an example input can be *15,Madrid*. This method will check if the input matches the key of the table (i.e., all the fields of the primary key have to be provided). If the destKvds list is empty, the split regions are not moved from the source server, but a replication group of the same arity that the source region replication is created. If the destKvds list has only one server, a replication group is not created. In any other cases, the upper regions resultant of the split are moved to the destKvds list, and a replication group is created. Params:
  - table: name of the table in which to make the split.
  - splitPoint: table's primary key fields.
  - destKvds: List of ServiceIp which represents the kvds where the new region will be placed after the split.

The above functionalities have been exposed to the end user who can manually invoke the aforementioned operations as shown in following console image:

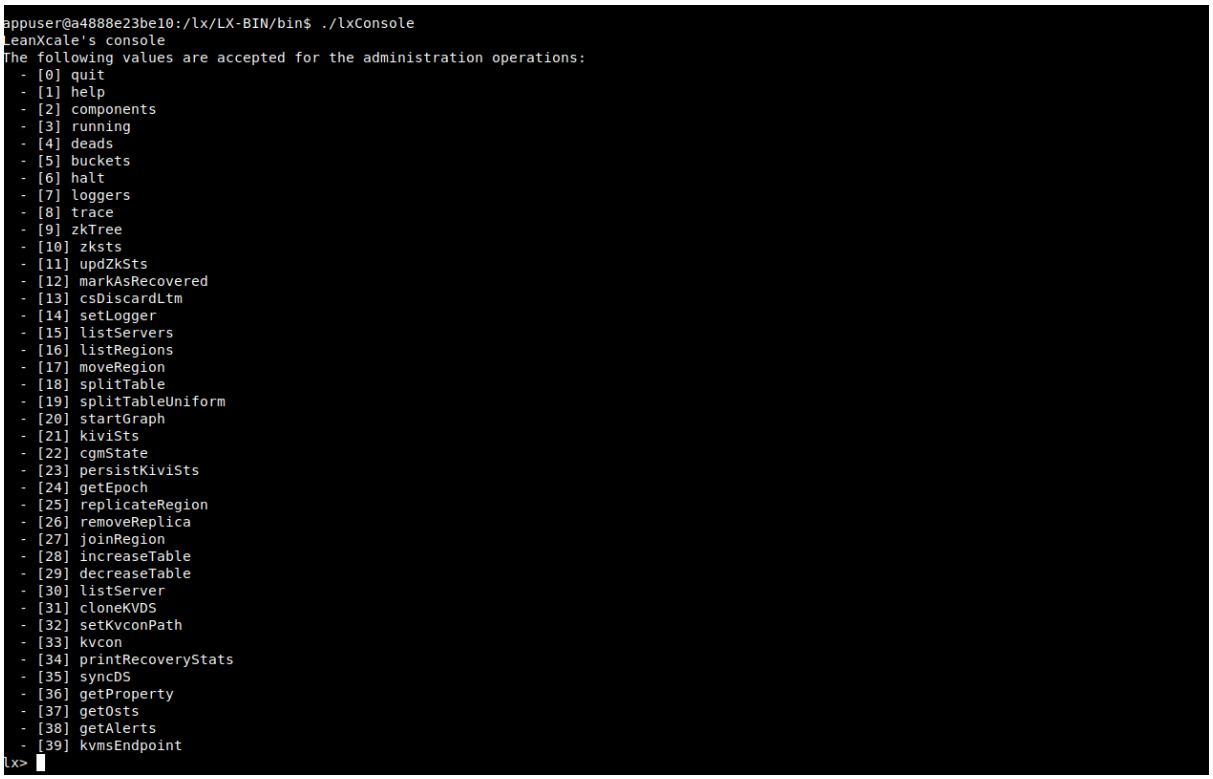


Figure 27: LeanXcale administration console

6.5.4 Resource Allocator Solver

The Adaptable Distributed Storage is based on the storage engine of LeanXcale, which has the ability to keep all table structures that can be further divided into one or more regions. Each region is a horizontal partition of the table containing a fraction of the data of a table and may contain some associated data, such as secondary indexes, dictionaries, and blobs. Regions on the other hand, are the units that can be distributed among the available data nodes of the storage engine of this component. The adaptable distributed storage allows to move, split or merge those data regions online, without any stop on the current workload, neither downgrading the overall performance of the system.

In order to decide which regions should be moved to a newly created and now available datanode, the Adaptable Distributed Storage makes use of the *Resource Allocator Solver.* The latter is a process which relies on an algorithm whose purpose is to solve the problem of balancing the data regions among the available data nodes. This component is periodically acquiring monitoring information regarding the resource consumption of the machines for each region. These resources are CPU and memory usage, network I/O bandwidth, and disk I/O bandwidth utilizations.

The imbalance of the system is computed with an aggregation of the usage of each of those resources, taking into account all available metrics of a given data node. Consequently, the algorithm associates each server with a vector of four values containing the usage of each resource during the last period. The typical deviation is computed for each resource giving a

metric of imbalance. If the system is perfectly balanced and all the servers are consuming the same for all the resources, the typical deviation is 0%. If a server is consuming 100% of a resource and the rest of the servers consume 0% of that resource, the typical deviation takes the maximum value. If any of the resources exceeds a threshold, the system is imbalanced and needs a balancing process to assign more resources to the regions of the server imbalanced.

In order to multidimensionally compare the load, the multi-resource vector, which contains the usage of each server, is aggregated in a unique value with the weighted sum. This represents a valid metric to compare the imbalance of the servers.

It is important to highlight here that this process is designed for homogeneous servers, this is, with the same CPU, the same amount of memory per core, and the same I/O and network bandwidth. Moreover, it assumes that each available *pod* or server contains the same amount of data node instances. This is important, as the algorithm makes use of no weights that would have been required in case there are different sizes of available servers, with varying number of data nodes instances in each one of them. As a result, it requires an homogeneous provision of machines and data nodes installed in each of the machines.

The process containing the resource allocator solver can be used by the Adaptable Distributed Storage to request for the provision of additional resources. When it identifies and over consumption of the overall available resources, such as it cannot move data regions to other data nodes, it will have to request for a scaling out process. The Data *MetaServer Node* notifies the *reconfiguration engine*, via the *Adaptable Distributed Storage Driver* and the *reconfiguration engine* makes use of the *WP4-End* to trigger a scaling out process requesting a new pod to allocate some imbalanced resources. This process is also used to decide the regions that will be moved to the new pod when the Infrastructure Management layer of BigDataStack also requests the storage layer to scale out. Additionally, the *Resource Allocator Process* is also used to detect the most underloaded servers where to move the regions back to scale in.

6.6 Experimentation Results

In order to validate the Adaptable Distributed Storage, we have decided to deploy the full stack of the LeanXcale datastore, where this component lies in its core as its storage system, and test its performance using a benchmark. We relied on an implementation of the standardized TPC-C benchmark[30], which is an OLTP benchmark proposed by the Transaction Processing Performance Council that emulates sales in a large company represented by different *warehouses* and it is considered the standard to validate performance among the database industry. The number of *warehouses* has a direct relation to the number of *clients* of this virtual company that place *orders* for buying various *items* from the various *stocks* of the *warehouse*. This benchmark can run with different data and transaction sizes by increasing the number of *warehouses* and *clients* to simulate different workloads both in terms of data and of requests per data. The TPC-C defines 5 different types of transactions,

30 http://www.tpc.org/tpcc/

each one with specific characteristic: a first one is a long running read-only transaction that needs to scan the whole table, a second one involves a multi-statement operational transaction which inserts and updates attributes on the tables, etc.

For the purposes of experimentation, we deployed LeanXcale with only one data node. This implies that the Adaptable Distributed Storage was initially deployed using a single data node. The deployment components can be shown in Figure 28, where we can identify the most important ones such as the LXQE which is using the storage and TPCC clients that are deployed in a separate in order to isolate the performance of the storage with the performance of the load generator. This component is using the LXMETA, where the LX Configuration Manager is deployed and contains all meta-information regarding the deployment of LeanXcale, the distributed of data among the data regions etc.

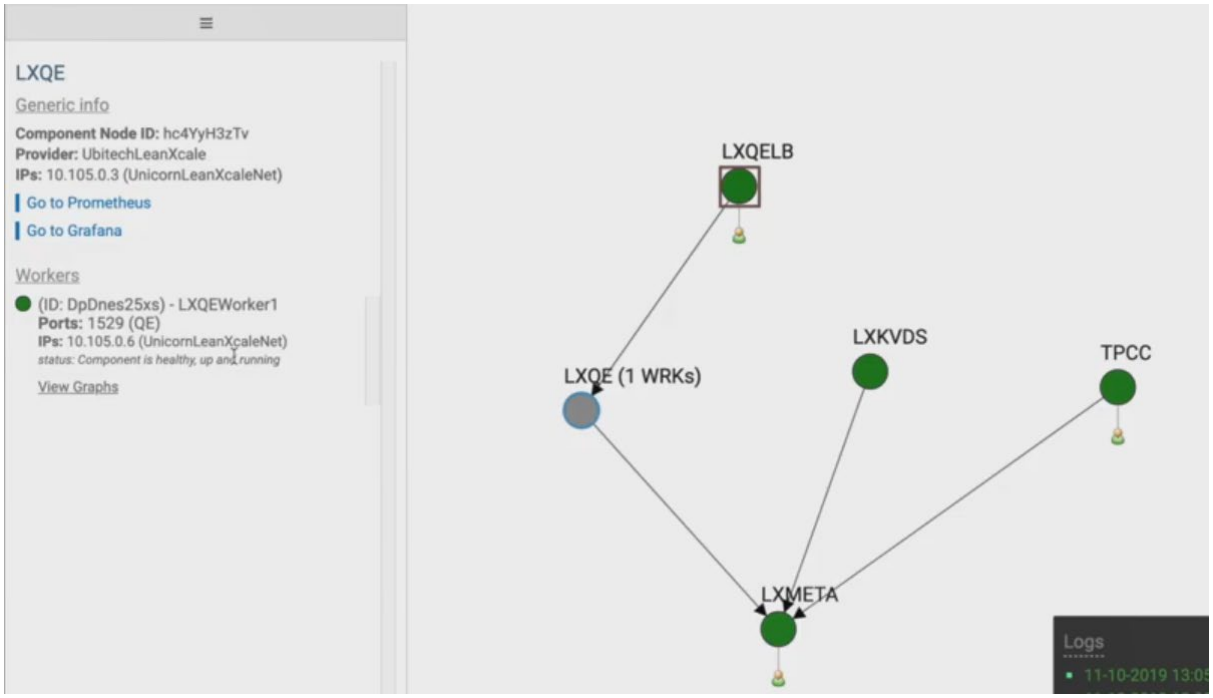


Figure 28: Initial Deployment of the Adaptable Distributed Storage using 1 node

After the initial deployment, we started the TPC-C benchmark in order for the latter to generate load and stress the Adaptable Distributed Storage. At this point the database administrator can check from the Prometheus monitoring system that the usage of the CPU is high and should apply an action.

Figure 29: Monitoring information showing CPU is fully consumed

Due to the tools that have been deployed in the scope of BigDataStack, the database administrator can use the administration console of LeanXcale and decide to create a new data node, split and move the data regions in order to balance the load. After these corrective actions, we can see now a second node in the deployment.

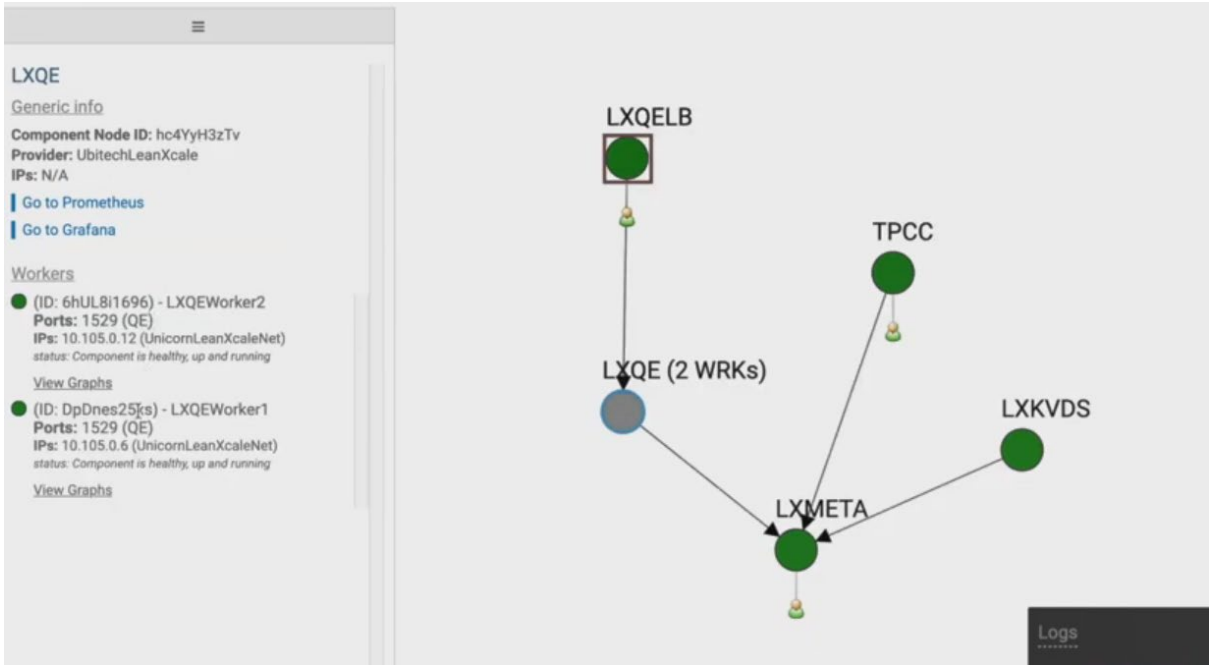


Figure 30: Deployment after the manual scale-out action

The result of these actions is that load has finally been distributed and balanced among the two nodes, as it is shown in Figure 30.

Figure 31: Monitoring information showing CPU load balanced

It is important to notice that all these actions took place while the TPC-C benchmark was running to insure a continuous operational workload, and that no downtime occurred. As explained in 1.9, database scalability actions that leave the database in an operational state and do not require from it to be shut down have already implemented by various NoSQL technologies. However, the important point is to notice that we are not aware of any implementation that also guarantees transaction semantics. So , the key differentiation of our implementation is that the scale-out action reaches all of the following: a) load balancing between the two nodes, , b) no downtime nor downgrading of the transactional semantics even under an operational workload downgrading the , c) ensured data consistency.

6.7 Conclusions

At the third phase of the project, all envisioned functionalities and features of the Adaptable Distributed Storage component of the platform have been now implemented and delivered. During the first two iterations, the fundamental technological pillars were put in place, they allow to effectively split data tables to data fragments called regions, and later to move and merge those regions among the different data nodes of the distributed storage.

This allows the distributed storage to redistribute the data load among those nodes in order to balance the overall resource consumption needed to serve the incoming workload. Resource can be considered as both computational resources (i.e. amount of memory and CPU cycles) or data related resources (i.e. volume of a data table, number of I/O disk accesses etc). For instance, a specific data table may be identified as the source of the consumption of more than 90% of the overall computational resources in a data node. Subsequently, this table shall be split to smaller regions that will be moved to other data nodes, so as to balance the overall consumption.

Another possible example: an application or a data analyst which commonly sends query statements that require a full scan of a data table. This generates an I/O intensive load that cannot be parallelized as long as the table is in a single region. Therefore, a decision might be taken to split the table into smaller regions and move the regions to other data nodes, so that the scan can be parallelized, and the I/O access can be shared now between the nodes.

As highlighted in this section, the innovation of this component is on its ability to perform the data movements at runtime, under heave operational workload, while ensuring transactional semantics and ACID properties of the ongoing transactions. This is where other vendors fail to achieve, as they either sacrifice the provision of transactions for the benefits of scalability, or they have to suffer from downtime, when the data movement takes place. The Adaptable Distributed Storage can achieve both, while during this last phase of the project, it has been integrated with the Infrastructure layer of BigDataStack, in order to request the provision of additional resources in terms of Kubernetes pods, that makes this component truly elastic.

The development of this component was based on the background technology of LeanXcale and is now part of its product, as it has been already up streamed into the release version, which is currently 1.6. As a result, it is under proprietary rights of LeanXcale. It is available for the partners of the consortium and can be found at http://bigdatastack-tasks.ds.unipi.gr/pavlos_LXS/lx-store-release Moreover, the code that has been developed for the integration of stateful data components with the infrastructure layer of BigDataStack has been released with open license and can be found here http://bigdatastack-tasks.ds.unipi.gr/pavlos_LXS/adsdeploy-wp4 Finally, LeanXcale has already submitted a request for a patent for the Adaptable Distributed Storage.

7 Seamless Data Analytics Framework

7.1 Introduction

The Seamless technology was demonstrated during the interim review in the context of the **Real-time Ship Management** use case developed in the course of the BigDataStack project. This demonstration encompassed all of the targeted features except for the SQL JOINs which were not supported.

In the last year of the project, the core of the LeanXcale data base has been heavily improved by incorporating the Apache Calcite query processing framework within the core of its internal query engine. This significantly impacted the integration of query federator with Object Storage, which had been achieved during the first two years of the project. A lot of efforts were poured at porting the seamless component to the new LeanXcale code as well as having seamless running over OpenShift in the MOC testbed.

This caused a delay in the implementation of the support for the SQL JOINs. JOINs have been supported during the last phase of the project and will be demonstrated during the final demo. The main achievements during this third phase is the integration of the query federator with LeanXcale's novel query engine and the newly delivered functionality that allows the execution of a JOIN operator over tables split between the two datastores. In addition the seamless JOIN support takes advantage of the *Dynamic File Prunning* (see sub-section 5.7.3) that permits data skipping to boost the performance of JOIN statements in the object store.

7.2 Requirements Specification

The requirements for the *Seamless Data Analytics Framework*, were redefined after the interim review (M19) and appear in the following tables. They now include improvements and additional functionalities along with updates and modifications that were identified during the implementation of the initial prototype.

Table 26 – Requirement REQ-SDAF-01 for Seamless Data Analytics

	Id	Level of detail	Type	Actor	Priority	
	REQ-SDAF-01	Software	FUNC	Developer	MAN	
Name	Provide access to data stores via a single and common interface.					
Description	BigDataStack includes two different data stores: the LeanXcale relational data store and IBM object store. The dataset can be fragmented and distributed over the two data stores (historic data being moved to object store). However, the application should be kept unaware of these internal data transfers. The application needs a common interface to submit queries, without having to specify where the data is stored.					
Additional Information	A federation mechanism is required that will encapsulate the process of data retrieval from the two data stores. The LeanXcale access point will act					

	as the federator between the relational and the Object Storage. The LeanXcale data base already provides a common JDBC interface for data connectivity. The federator will receive the query and execute it in both data stores. For the object store, the access would be via Spark SQL, with the assistance of Apache Hive for storing the metadata of the schema catalogue, which can also be transparently accessible via a JDBC interface. The federator will take into consideration the operations that can be supported in order to push down the operations accordingly. Regarding the relational store, all operations will be pushed down to the store. At the very end, the federator will merge the results and return back the result set. It shouldn't count data that appears in both data stores twice.

Table 27 - Requirement REQ-SDAF-02 for Seamless Data Analytics

	Id	Level of detail	Type	Actor	Priority
	REQ-SDAF-02	System	DATA	Developer	MAN
Name	Move historical data from the relational data store to the object store.				
Description	Data ingested by the use cases will be stored into the relational datastore, as they are operational, in order to ensure data consistency in terms of ACID properties. After a configurable period of time, called the *freshness window* (which depends on the data set), the data becomes outdated and is no longer used by operational workloads. However, this historical data is still valuable and can be exploited by Big Data analytics algorithms. This data should be moved from the LeanXcale data base to the IBM object store.				
Additional Information	A mechanism should be implemented that monitors the *freshness window* and decides whether or not a data movement should take place. The mechanism must allow the data pulling of the data slice from the operational datastore and the persistently storage on the object store. During the data movement, the mechanism should allow the continuous execution of data retrieval from the data federator, so that no down time should be observed, while ensuring the data consistency.				

Table 28 - Requirement REQ-SDAF-03 for Seamless Data Analytics

	Id	Level of detail	Type	Actor	Priority
	REQ-SDAF-03	Software	DATA	Developer	MAN
Name	Inform the LeanXcale data store when data are imported to the object store.				
Description	When data is pulled from the operational datastore, the LeanXcale data base can drop them. However, due to the asynchronous design, the LeanXcale data base cannot know when the data has been made available				

	to the object store. As a result, the object store must inform the LeanXcale data base regarding the successful insertion of the data, so that the LeanXcale data base can safely drop these data.
Additional Information	One possible solution to deal with this requirement will be the introduction of marking the data to be transferred to the object store by additional timestamps. Data that is being flushed and exported to the object store can be marked that way, so that later on, the object store can inform the LeanXcale data base that this bunch of data has been successfully imported. By doing so, the *federator* component can push down operations accordingly, and only request specific data from the underlying data stores. Data that is known to the LeanXcale data base that has been previously uploaded to the object store, will not be retrieved by the *federator* and can be safely discarded by the *vacuum* process of the LeanXcale database.

Table 29 - Requirement REQ-SDAF-04 for Seamless Data Analytics

	Id	**Level of detail**	**Type**	**Actor**	**Priority**	
	REQ-SDAF-04	Software	DATA	Developer	OPT	
Name	Optimize query execution					
Description	The federator receives a query and executes it into the different stores. The federator will be based on the LeanXcale query engine. The latter provides a query optimizer, which allows it to examine the different execution plans that can be produced in order to execute a query. However, it has been implemented to evaluate plans to be executed locally. It should be extended in order to take into consideration the operations that can be pushed down to the object store, and whether or not it is worth for an operator to be pushed down, according to the response time of the execution from Spark SQL, the amount of data that will be retrieved to the federator etc.					
Additional Information	As every operation that can be supported by the object store will be pushed down to be executed locally, in order to avoid transferring a big amount of data through the network and process them in the query engine level, the implementation of this requirement corresponds to the following two aspects: the choice of the optimal strategy for executing the JOIN operation concerning data tables that are distributed and split to the two stores, and the redefinition of the query execution plan, in order for the query federator to exploit data locality and reduce the number of rows that will be retrieved and transferred from the object store via the network.					

Table 30 - Requirement REQ-SDAF-05 for Seamless Data Analytics

	Id	Level of detail	Type	Actor	Priority
	REQ-SDAF-05	Software	DATA	Developer	OPT
Name	Optimize access to Object Storage.				
Description	In order to perform analytics efficiently on Object Storage, a client-side caching/acceleration layer is needed. This is critical for a hybrid cloud scenario, where some of the customer data is on premise (potentially the LeanXcale data base and Spark) and some is in the cloud (potentially IBM COS). In such a scenario, when performing analytics, data needs to move from COS to Spark across the WAN, therefore minimizing the amount of data movement when part of the data is retrieved multiple times is of utmost importance. A similar scenario involves multi-cloud, where a dataset may be distributed among more than one cloud, also requiring data transfer across the WAN for the purposes of analytics.				
Additional Information	This complements data skipping and data layout techniques to further reduce the KPI measuring the number of bytes sent from Object Storage to Spark.				

Table 31 - Requirement REQ-SDAF-06 for Seamless Data Analytics

	Id	Level of detail	Type	Actor	Priority
	REQ-SDAF-06	Stakeholder	FUNC	Developer	MAN
Name	SQL Grammar extension				
Description	In order to better support the seamless, an extension of the SQL grammar is needed				
Additional Information	The grammar extensions will allow the database administrator to define that a data table can be split across the two datastores, and will allow him to provide additional information like the time window of the data slice, along with other configuration attributes like the minimum size of a data slice that is allowed to be moved, time frequency of the moving action etc.				

Table 32 - Requirement REQ-SDAF-07 for Seamless Data Analytics

	Id	Level of detail	Type	Actor	Priority
	REQ-SDAF-07	Stakeholder	DATA	Developer	MAN
Name	Better decision of datasets partition				
Description	Currently slices of a dataset will start to be moved to the Object Store on a time basis (e.g., data older than 3 months). However, this is not flexible				

	enough when the growth of the dataset is not known in advance. We often will prefer to leave the full dataset within the LeanXscale database if it will be under a given size.
Additional Information	This requirement is very useful for the implementation of JOINs. Indeed, it is a typical case that one of the 2 tables being joined is small. In this case, the JOIN can be implemented both in a simpler and more efficient way when the (small) table fully stored at LeanXcale.

Table 33 - Requirement REQ-SDAF-08 for Seamless Data Analytics

	Id	Level of detail	Type	Actor	Priority
	REQ-SDAF-08	System	DATA	Developer	OPT
Name	Handle JOINs for datasets distributed within both data stores				
Description	Currently, the seamless technology does not handle JOINs which limits the applicability of the seamless technology. The updated version of the query federator should support the JOIN operator thus being fully SQL compliant. The strategy for implementing this operator should take into account that data has been split between two non-homogeneous datastores, and should exploit their locality, in order to avoid moving huge amounts of data over the network.				
Additional Information	As of the current release, JOINS are being done in the query engine level of the LeanXcale datastore. This is not efficient as it requires the data from both sides to be fetched in memory, and the JOIN must be performed at the level. The new JOIN operator must be able to push down the operator itself to the two datastores, joining data locally where possible, and only send data concerning the smaller datable to the object store, applying strategies like bind join and get the result. The JOIN operator must then perform a union over the intermediate results.				

Table 34 - Requirement REQ-SDAF-09 for Seamless Data Analytics

	Id	Level of detail	Type	Actor	Priority
	REQ-SDAF-09	System	PER	Developer	MAN
Name	Ensure data consistency when a moving action is taking place				
Description	When data is moving from the operational store to the object store, data might either co-exist in both stores, or are non-existent in any store. The framework must be able to serve requests for data retrieval with no downtimes during this process, and the data should be consistent, meaning that the result of the execution of a query should be the same, no matter if the data are being moved.				
Additional Information	The operational datastore must not withdraw a data slice, until an acknowledgement of a persistence storage is being notified by the object store. In this case, data can co-exist in both stores. The Query Federator of				

	the framework must take this into account and re-write the queries to be executed in both stores accordingly in order to scan records on the visible data set in each store. In order to ensure data consistency when parallel transactions are being executed, before, during and after the data moving process, it will rely on the transactional manager of the operational datastore.

Table 35 - Requirement REQ-SDAF-10 for Seamless Data Analytics

	Id	Level of detail	Type	Actor	Priority
	REQ-SDAF-10	System	PERF	Developer	OPT
Name	Efficiently drop a data slice from the operational datastore				
Description	Dropping data slice from the operational datastore will usually involve the deletion of numerous tuples that might affect the operational behavior of the datastore, as it will push a lot of workload to its transactional engine. A most efficient way should be implemented.				
Additional Information	When the moving action should be performed, the Seamless Analytical Framework should inform the operational datastore to be prepared to drop that slice afterwards. In correspondence with the REQ-ADS-01, it will move this slice to specific data region, and additionally it will mark it as read-only. By doing so, it won't be allowed for any data modification operation to be performed when the data is being moved, which is also aligned with the REQ-SDAF-09. When the acknowledge of the persistent storage of the slice in the object store is being received, the data slice can be dropped from the operational datastore by removing the data region, with no further action from the transactional manager, as this data region had been already defined as read-only.				

7.3 User Story

Data coming from different sources at high rate, is being ingested into the operational datastore. Due to its ultra-scalable transactional manager and its API to direct ingest data on the storage layer bypassing the SQL query engine while forcing transactional semantics, LXS can handle this highly intensive workload, while at the same time ensuring online analytical processing (OLAP). However, it is typical to expect with Big Data, that "old" data becomes historical and are no longer subject to modifications. Moreover, their volume is continuously increasing, and as a fact, an operational datastore may not be considered anymore the best solution to store data. On the other hand, an object store designed to perform heavy analytics workloads on large volume of data might be more suitable. Due to this, historical data slices are carved out from the operational datastore and sent to the IBM Object Store[31]. The distribution of data across two datastores is problematic since: a) Data

[31] In this section the IBM Cloud Object Store (COS) is intended as an Object Store example. It was chosen since this is the object store that was used during the demonstrations of the M18 interim review. The IBM

must be retrieved from both stores, b) both data results must be merged at the application level, which is a non-trivial task, both in terms of interoperability between the different datastores and ~~of~~ efficiency. That is, this demands from the application level to be able to perform operations that are not natural to be executed at that layer and which a database can do more efficiently. Moreover, moving data from one datastore to the other introduces significant data consistency concerns, that would typically require operation freeze during the data migrations. The Seamless Analytical Framework is designed to solve these problems: it provides a common interface for the application developer to query data, without having to know neither where the data are actually stored nor the query semantics of the different datastores. The SAF provides a common JDBC implementation that supports standard SQL statements for the end-user to retrieve information and hides all data stores specificities. Moreover, it ensures data consistency when moving data from the operational datastore to the Object Store, without requiring any freeze. The SAF relies on the LXS transactional management component to ensure that data records will not be retrieved from both LXS or Object Store even when data is being migrated.
Finally, it supports all standard SQL commands, regardless if they are targeting scan operators on a simple data table stored locally in LXS, or split between the datastores, and regardless of the complexity introduced by the need for a distributed execution of the command. The SAF hides all this lower level details about data management and retrieval, thus providing a unique interface where the data analyst or application developer can submit standard SQL statements, whose execution will be done seamlessly by the framework itself, giving the impression that the full dataset is stored locally into this logical data base that consists of the operational and the analytical stores.

7.4 User Perspective

Figure 32 provides a high-level view of the main components of the Seamless Data Analytical Framework, from an application perspective. This framework can be considered as a black box from an application point of view, which includes the two data stores with both distinct types and usage purposes: Data can either exist on the LeanXcale data base, or the object store, or co-exist in both. As a result, data can be fragmented across the data stores, however, from an application point of view this must be totally encapsulated by the framework itself.

COS may be replaced by any object store that presents an interface compatible with the S3 API such as Ceph or minio.

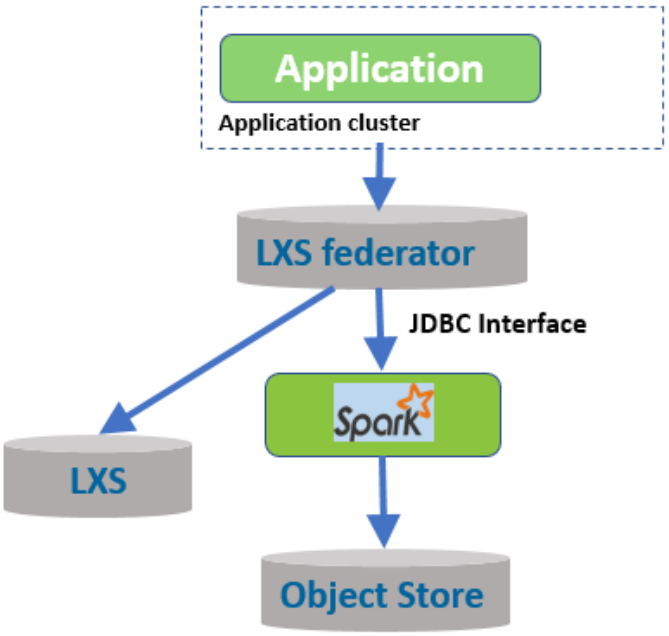


Figure 32: Federation of the two data stores in the scope of the Seamless Data Analytical Framework

The federation between the two data stores is based on the LeanXcale internal Query Engine which has polyglot capabilities and can join data coming from different sources. Having LeanXcale Query Engine (QE in the following) as the *federator* of the framework, we addressed the REQ-SDAF-01 requirement: Data connectivity with the *federator* is done via the JDBC implementation, thus offering a uniform access to all the use cases and platform components. The data modification operations are directly forwarded to the LeanXcale relational data store for execution since, by assumption, only data stored within the LXS DB can be modified. However, data retrieval operations generally involve both stores since datasets are distributed and can be stored within any of these stores. In order to retrieve data from the LeanXcale database, the QE gets the requests from JDBC and executes the query in its distributed storage. Exploiting its capabilities for intra-operation parallelism and the ability of its key-value storage to accept push downs and execute locally various operations, the query engine splits the execution and distributes it in parallel, by pushing down the majority of operations, and then, it merges the intermediate results and returns them back to the user. The LeanXcale QE can push down simple selections and aggregation operators. Constructing the results for the latter takes into account the following: the sum/count operations can accumulate the intermediate results, the max operation will require a logical operation to keep the maximum value across the nodes per granular row, and the average operation can be split to sum divided by count. Therefore, the LeanXcale Query Engine can be distributed across various nodes and uses a random instance to coordinate the distributed execution.

A different approach has been followed though in order to support the execution of JOIN operators: LeanXcale's data storage cannot support JOINs on the data storage level, therefore, it is its query engine itself that implements this operator. However, when having to retrieve data from Object Storage, special considerations had to be taken into account in order to exploit data locality and move as few data items as possible between the two

stores. In fact, the JOIN operator over fragmented tables had to be implemented from scratch in order to support JOINs over tables distributed into the 2 data stores and refine the statements to be forwarded to the Object Storage side for efficient data retrieval.

Object stores are typically accessed through Spark SQL. The query federator is integrated with Spark in order to push to the Object Storage side SQL statements. Moreover, Spark provides a JDBC interface to enable data connectivity. Thus, by exploiting the polyglot capabilities of the LeanXcale data base, its Query Engine has been used to push down operations both to the LeanXcale data base and to Spark, via the JDBC, and merge the intermediate results as previously described. The use of Apache Hive provides a common shared metadata catalogue of the data schema. This permits data to be retrieved from the object store via Spark using the same table and column names, as within LeanXcale.

Figure 33 shows the pipeline designed to move historical data from the LeanXcale relational data store to the IBM object store. The LeanXcale data base is informed periodically to get ready to export a dump of the oldest updates, which now fall outside the freshness window (depicted in the diagram as cold data slices). Data are pulled by the *Data Mover* component, which injects this data into the object store. As a result, eventually the data exported by the LeanXcale database is now available in the latter. After the successful ingestion of the historical data, the LeanXcale database needs to be informed that the data has been persisted in the object store, so that it can be discarded from the relational database. During this phase, data co-exist in both stores, however the *Query Federator* ensures that they will be scanned and taken into account in the result only once.

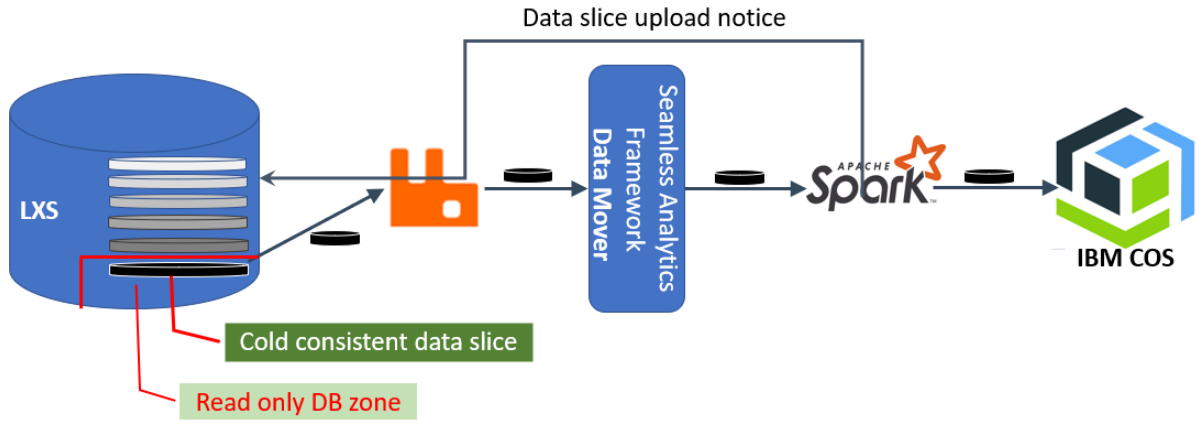


Figure 33: LeanXcale data base to IBM Object Storage pipeline for historical data

7.5 Detailed Design

The main architectural components of the Seamless Analytical Framework appear in Figure 34, which also shows the sequence of interactions, when a data slice is moved:

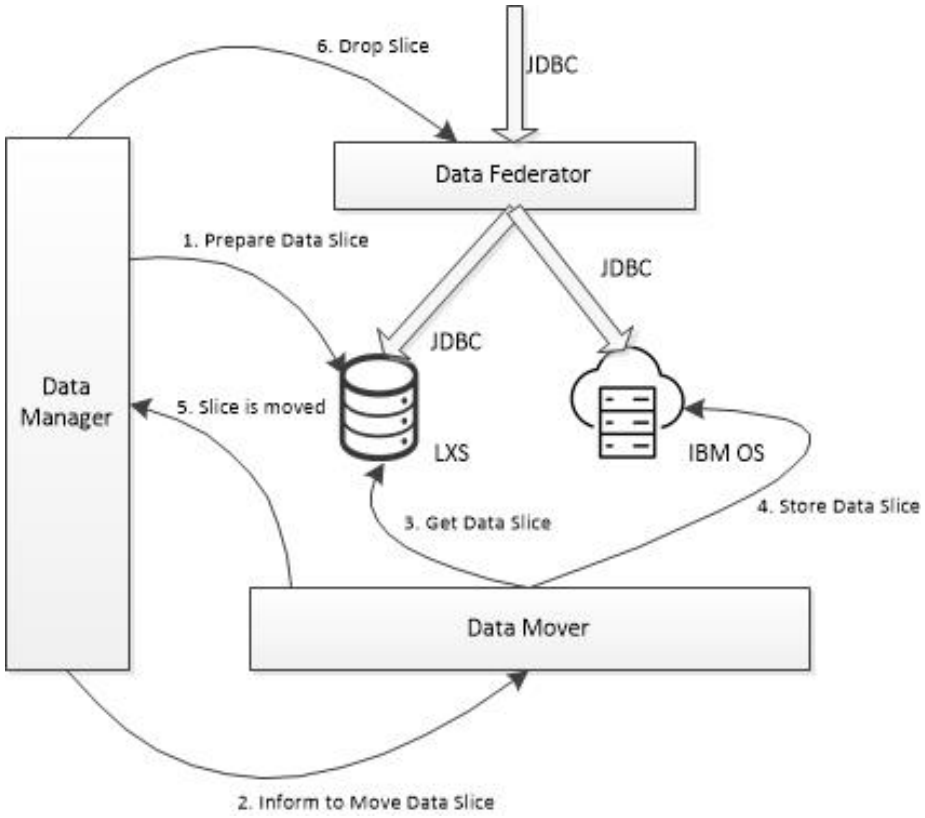


Figure 34: Interactions among SAF components

As the architecture shows, the main components of the Seamless Analytical Framework are the following:

1. **The LXS datastore**: an operational relational database that ensures transactional semantics over intensive write workloads, while allowing at the same time to run complex analytical queries. It provides a standard JDBC interface which allows to store and retrieve data using standard SQL statements.
2. **The IBM Object Store**: IBM Cloud™ Object Storage (COS) makes it possible to store large amounts of data, simply and cost effectively as it follows the REST s3a API. It is commonly used for data archiving and backup, for web and mobile applications, and as scalable, persistent storage for analytics. In this section the IBM Cloud Object Store (COS) is intended as possible Object Store example. It was chosen since it is the object store that was used during the demonstrations of the M18 interim review. The IBM COS may be replaced by any object store such as Ceph or minio that follows the s3a API.
3. **The Data Mover**: which is responsible for moving historical data slices from LXS to IBM COS. It listens for notifications that are meant to trigger the process. A notification message contains:
    a. The system timestamp of the youngest data record to be moved from LXS to OS
        i. The start system time stamp of the data that should be moved – this time stamp should be on the hour, month, year, meaning, for

example, if during table creation the user chose that the granularity is hourly – the time stamp will be for example 2017-01-01 15:00.

ii. This timestamp represents the start time of the slice.

b. The period of time that represents a slice/the end system time stamp for this slice – according to the configuration for this table, if for example the user chose granularity of hour the end time stamp will be 2017-01-01 16:00, or alternatively the parameter will represent a time span (60 minutes for this example).
c. All the needed information for building one (or many) queries to retrieve the data slice from LXS. Note that this gives the flexibility of retrieving, for example, only certain columns.
d. The schema of the result.
e. Statistics, which for instance may give hints concerning the data distribution. They are meant to help the Data Mover with partitioning of the data slice. This feature was not implemented by the interim review and has been left for possible future optimizations.

It is important to highlight the fact that the *notifications* must contain non overlapping data slices, aside from duplicate notifications for the same slice which can happen for example when LXS misses an acknowledge message. Upon message reception, the Data Mover retrieves from LXS the appropriate data slice by doing the following operations:

1. Upon reception of a data slice move request, the Data Mover checks whether this data slice has already been successfully stored in the OS, and in case this is true, the request will be ignored.
2. Otherwise, it establishes a JDBC direct connection to LXS and submits one or multiple common SQL statements to retrieve the data slice.
3. After the full data slice has been received, the Data mover eventually stores it to the OS, and performs all optimizations needed by OS side (i.e. building indexes for data skipping etc). Note that since, as further detailed, data is laid out with Hive, layout decisions can only be made at the slice level. That prevents any scenario where an object would contain rows from two different slices. For efficient usage of the OS a data slice size should therefore be larger than the targeted minimum size of the objects to be uploaded to the object store (typically 30MB). If the data slice is bigger, it is up to the Data Mover to decide how to split it into objects.
4. When the process succeeds and makes the data slice visible for any direct client of the Object Store, the Data Mover notifies the Data Manager by sending an acknowledge message. The visibility will be done by executing "ALTER TABLE ADD PARTITIONS" to add the relevant partition to the hive metastore (see below for more information). In cases of failures, the Data Mover will retry the store request for the data slice.
It is important to mention that once the data slice or part of it has been uploaded to the object store, it is visible and thus retrievable by any client

that would directly connect to the OS. However, the Data Federator ensures that this newly uploaded data will stay inaccessible by SAF clients until the whole process succeeds and the LXS transactional manager transfers the data slice visibility from LXS to object store as an atomic operation.

4. **The Data Manager**: This component keeps track of the data movement process, triggers the corresponding actions of the Data Mover and orchestrates the whole data movement process. It persistently stores the most recent timestamp of data that was confirmed to have been uploaded to Object Store and schedules the next data slice move per table. When a new data movement is to be done the Data Manager:
   a. Firstly, informs the Query Federator to be prepared for the movement. The Federator informs the KiVi storage system of LXS to mark the corresponding data slice as Read-Only and move the data in a separate region. The purpose of this procedure is to ease the process of dropping this region at the end of the data movement workflow.
   b. It then notifies the Data Mover to trigger the data movement by sending to it the message displayed in Figure 34 which informs that the data slice is now ready to be moved from LXS. The message contains various information and the SQL statement among others that should be used from the Data Mover in order to retrieve data from the operational datastore. Therefore, the Data Mover opens a direct JDBC connection through Spark to LXS and submits one or many SQL statements. In cases of failures/restarts etc, the Data Manager checks if there is a need to repeat the notification to the Data Mover for a given slice in which case it notifies again the Data Mover. Duplication of the same message is allowed but different data slices must be non-overlapping. Each slice corresponds to the time interval specified in the table creation.
   c. The Data *Manager* listens for the Data Mover to acknowledge that the data slice has been persistently stored with success in the object store. Once the notification is received, it informs the Query Federator to drop the corresponding data slice. It is important to note that, due to the distributed and asynchronous nature of the architecture, the Data Manager might get out of order success notification (for example, receiving success notification for the period of 15:00 - 16:00 while yet to receive success notification for 14:00 – 15:00). In such cases, the Data Manager waits for acknowledgment notifications from the Data Mover so as to be sure that all data slices have been persistently stored in the object store, and that there is no missing gap in between. It then informs the Query Federator with the timestamp that latest data slice that has been stored.
   d. When the Query Federator receives a request to drop a slice, it firstly moves its splitting point to the value of the latest timestamp that has been stored in the OS. The splitting point is being taken into account by the Query Federator in order to rewrite the input query so as to scan the correct amount of data from each table. It updates the splitting point by opening an internal transaction using the transactional manager of LeanXcale. Taking into

account that all access to the Seamless Analytical Framework is being done by opening a JDBC connection to its internal Query Engine, the data user always opens a transaction via this component. This implies that concurrent read queries will be executed in a serialized order with the transaction that updates the splitting point, therefore, data consistency is ensured when having multiple requests for data retrieval while the finalization of the data movement is taking place. After the splitting point has been updated, it is certain that no additional transaction will access the corresponding data slice. Therefore, the system waits until all pending transactions finish, and afterwards it sends a request for the data slice to be dropped. This is carried out by the KiVi storage system of LeanXcale. Given the fact that the data slice has been already moved to a specific data region that is additionally marked as read-only, the process of dropping the data slice is translated in the removal of the corresponding files that contains the data of the data region. As the latter is marked as read-only, the process does not have to interfere with the transactional manager, as there can be no further conflicts and the whole process is being performed much more efficiently.
It is important to notice that all communications between the *Data Manager* and the *Query Federator* for preparing a dropping a data slice are performed by executing a JDBC statement to LeanXcale containing a table function whose implementation executes the corresponding procedure in the Query Federator. The parameters of this table function is the type of the operation (DROP, SPLIT), the timestamp and the data table. An example of a statement for preparing the data slice for movement will be the following:

```
select *
from table("dataslicemanager"(
'SPLIT:BigDataStackDB:DANAOS:vessel_engine_weather_data:1362092400000:Europe/Madrid'))
```

This will invoke the *dataslicemanager* table function, that will trigger the Query Federator to prepare the data slice, by splitting the dataset and move the corresponding data into a separate data region. The parameters are the type of the action (i.e. SPLIT in this case), the logical database name, the schema, the data table, the timestamp of the data slice to be moved and the time zone.
Moreover, the communication between the *Data Manager* and the *Data Mover* is via the use of RabbitMQ, in order to ensure the durability of the exchanged messages. It is not important for the protocol to receive requests or acknowledgement notifications out of order, but it is important that all messages finally are received and that's why we included such a queue in the design.

5. **The Data Federator**: This component a) ensures data consistency when a data slice is being moved from the operational data store to the object store b) handles data retrieval among the federated data sources c) provides a single point of access to the framework via a JDBC interface for data retrieval that allows for Read-Only queries using standard SQL statements and for CREATE TABLE DDLs to enable the creation of seamless tables.

   Table Creation – In order to create a seamless table, the user has to run a "CREATE TABLE" DDL against the Data Federator. This query ensures the creation of the table both in LXS and in the OS. The OS will use Apache Hive metastore as its catalogue and this will ensure that every table created in LXS will have a corresponding table in the OS (and the same database). Apache Hive metastore can be based on a standard OLTP database such as LeanXcale, which offers a JDBC driver, and can be used as the backend for Apache Hive metastore. During the table creation the user specifies the following 2 parameters: the minimal age of a data slice to be moved to the OS and the periodicity of the slice moves.
   Note: a more complex retiring rule can be set (such as size based with time constraints) as long as periodicity is kept. The periodicity of slice moves can be hourly, monthly, yearly and must be a on the time (i.e hourly means for instance data between 15:00 – 16:00 and not some arbitrary 1 hour period say between 15:23 – 16:23). This timestamp represents the system timestamp which is the timestamp added by LXS to the data when it is ingested to LXS and/or can related to a possible timestamp figuring within the data with the constraint that new inserted records will always have the most recent value. Moreover, upon receiving the DDL statement the Query Federator will additionally inform the *Data Manager* component to monitor the data table so that it can periodically orchestrate the data movement process.

   Data Querying - At any given point in time, The Data Federator is aware of the latest timestamp of the newest data that was declared as successfully uploaded to Object Store, per data table. Having this information locally, when a query statement is submitted, it internally submits it to both LXS and OS while adding to the query sent to the OS an additional time constraint for data older than this timestamp, and inversely from LXS, data which is newer than this timestamp. It then merges the results and returns them via the opened JDBC connection. Note that the added timestamp field that is being used to query the appropriate data subsets both in the OS and LXS is a system timestamp and it will not be part of the result set.

   If we designate by *dataset_lxs* and *dataset_os* the respective subsets: of data strictly older than the timestamp (in the OS) and of data newer than the timestamp (in the LXS DB), then the presented scheme ensures that *dataset_lxs* and *dataset_os* have an empty intersection and that their union is the full dataset.

   After the Data mover notifies that the data having been successfully uploaded, the

Data Manager informs the Data Federator of the new timestamp and the Data Federator will start dropping data older than this timestamp.

However, as it has been already mentioned, the Data Federator cannot drop data if there are on-going transactions which accessing this data. When a data retrieval query is received by the Query Federator, the latter uses an internal compiler to firstly validate that the input is valid in terms of syntax and constructs the tree of the operational plan. Then it pushes down the filter operation based on the timestamp of the splitting point in order for the federated datastores to take into account the correct space while scanning with respect to the current transaction (i.e. two concurrent transactions might need to scan different space in each store). Finally, it generates the SQL statement to be executed in each one of the datastores, taking into account the specific supported dialects of each one (i.e. the syntax of the LIMIT operation in LeanXcale is different than the one supported by Spark). According to the type of the query operators that can be pushed down to both stores, a specific implementation is instantiated that will submit the queries and will merge the intermediate results to the final *ResultSet.* All of them open two JDBC connections to LeanXcale and to the Object Store, and retrieve data concurrently, however according to the type of the operation, each instance implements different strategy to merge the results. Until M23 when the second version of this deliverable was published, the supported operations were the following:

1. **Simple Scan**: This operator will return rows randomly, when a new row is being retrieved from each datastore, it is directly added to the result.
2. **Ordered Scan**: As the implementation pushes down ordered scans as well, it collects data from both stores, and for each one, it returns back the tuple that is lesser/greater according to the fields that are involved in the ORDER BY operation.
3. **Simple Aggregations**: This operation waits until data have been received by both stores. It then merges the results according to the aggregation operator that is involved: if it is minimum, it returns the minimum of the two values, if it is maximum, it returns the maximum of the two values, if it is a summary, it summaries the two values and if it is a count, it summarizes the two values as well. However, all these operations can be executed distributed. However, the average operation cannot be executed distributed. In such a case, the Query Federator re-constructs the SQL statement asking for both the count and summary operations to be executed. Then, it merges these results as explained, and returns the result of the summary operation divided by the count.
4. **Group by Aggregations**: This operation uses a HashMap structure to store the intermediate results. The hash is performed taking into account the fields involved in the GROUP BY operator. Data are retrieved in parallel from both stores. As a row is being received, the implementation checks if the row is already contained in the HashMap structure. If not, it adds it. If yes, it calculates the values of the rows that need to be merged as explained above, removes the row from the structure and adds the row to the result set. One

important thing to be highlighted here is that this operation needs to wait until all data has been retrieved by both stores, in order to finally return the rows that have been added in the HashMap. The reason for that is that unless there is a GROUP BY on a primary key, the Query Federator cannot know if the hash of the group by operator is contained in both stores, therefore, it has to wait until the data retrieval from both stores is complete.

5. **Ordered Group by Aggregations**: This operation combines the logic of the previous.

The final version adds the JOINs operation and Section 7.6 contains its design.

Note: The Data Federator should submit to the OS appropriate queries – i.e. queries using a syntax compatible with the OS. This includes:

- Querying the right table and database in the OS – the use of hive metastore ensures that this "translation" is straight forward as the convention is that the same logical database name and table are to be used in the OS and in LXS. (Note: In order for the Data framework to work well, LeanXcale and the OS should hold a consistent catalogue. This catalogue can either be federated or unified. We use Apache Hive metastore mainly for having a consistent (federated) catalogue. Federated catalogue - since the OS has its own catalogue (when using Apache Hive metastore) and LeanXcale has its own catalogue , one possibility is to maintain the "unified" catalogue using both by processing the TABLE CREATE DDLs through the seamless API and updating both catalogues each one with the relevant information. Maintaining federated yet consistent catalogues will also ensure that at any time a user can query only LXS or only the OS.
  Note: updates will only be processed by LXS as the assumption is that historic slices that were moved to the OS no longer need to be updated. This assumption was derived from the Shipping Management use case where IoT data received from vessels may need to be modified but only up to 3 months after its reception. In the second part of the project, we plan to loosen this assumption to broaden the seamless design to use cases where after a first historic period where a data slice may be modified and after this first period, and this is the change, we assume that it still can be modified, however with a very low probability. This means that a data slice may be brought back from OS to the operational datastore. Since by assumption, these events are very rare, the design may lead to a costly implementation.
- Selecting only the necessary columns – the timestamp being used to determine the period is a system time stamp and is not related to the table data, so for example "select * from db.table" should either be translated to "select <column1>,…,<columnN> from db.table where sys_timestamp < …" to discard the system timestamp column, another option is for the data federator to discard the unnecessary column when joining the results with the results from LeanXcale.

- Making sure the syntax of the query being submitted to the OS is compatible with OS syntax.


```
{
"dbSchema": "DANAOS",
"database": "BigDataStackDB",
"table": "vessel_engine_weather_data",
"slice_start": 1362092400000,
"slice_end": 1362092400000,
"sql_statetment": "SELECT * FROM DANAOS.VESSEL_ENGINE_WEATHER_DATA WHERE
dateadded <= {ts '2013-03-01 00:00:00'} AND dateAdded > {ts '2013-03-01 00:00:00'}",
"uuid": "25ee18b6-6021-4e42-9c69-e95ca68cf38f"
}
```


Figure 35: Data Manager notifies Data Mover to start moving data slices


```
{
 "dbSchema": "DANAOS",
 "database": "BigDataStackDB",
 "table": "vessel_engine_weather_data",
 "slice_start": 1362092400000,
 "slice_end": 1362092400000,
 "uuid": "25ee18b6-6021-4e42-9c69-e95ca68cf38f"
}
```


Figure 36: Data Mover notifies the Manager that the movement succeeded

7.6 JOIN Operator

JOINing two different tables is a non-trivial operation and in fact, is one of the most complex operations that a database can perform. It requires scanning the two tables while keeping the retrieved data items in memory, in order to validate if they can be joined with records retrieved by the second table. The execution of this operation takes into account various factors like the overall size of target tables, whether or not the join will be performed over columns that are indexed (and therefore, the corresponding data items have been stored with a predefined order), the number of records expected to be retrieved from an intermediate scan, just to name a few.

As a result, different techniques can be applied in each case and it is up to the query optimizer of the query engine to validate these parameters and propose the expected most efficient algorithm to be executed to retrieve the results.

Implementations vary from

Nested-loop join, where data returned from one table are stored in memory, and then, an iteration over the whole dataset is being performed (inside a nested loop) to ask if they are data items that match the join condition in the second table. This requires the movement of a whole table in memory, which is expensive, and therefore, this technique is favourable when the size of one of the two operands is expected to be small enough.

Merge joins which takes advantage of the fact that the data items have been stored in both tables ordered according to the columns on which the join will be performed. As a result, two internal pointers are being used at each table, and the scan can be performed over the two while the condition is kept, returning data items if the condition is satisfied, and skipping items if it is not.

Hash joins can be considered as a special case of the *nested-loop* which keeps data in memory and allows for a most efficient execution when there is the case of *equity joins*, while more advanced techniques like *bind-join* can be also applied, which transforms the execution of the statement into an IN clause, in order to reduce the amount of data that need to be moved from the storage layer to the query engine.

In our implementation of the JOIN operator over federated datastores, we take advantage of some of these techniques in order to execute the statement in a more efficient manner.

Furthermore, this operation requires the involvement of two different data tables, it cannot be implemented in a distributed fashion in the storage level. Data tables may be stored in different data nodes and therefore, pushing down this operator to the storage is not feasible. In fact, there are two techniques that can be used in order to parallelize the execution and return the results in a more efficient manner. First, the implementation in the query engine level can be done in a distributed way, so that the operation itself can be parallelized and its overall response time can be further improved. However, this will still require the movement of data from the storage layer to the query engine level, while at the same time, due to its stateful nature, data retrieved from one of the executors to the query engine, must be broadcasted to all other execution nodes. When having to deal with BigData, sending gigabytes of data over the network and continuously transmitting them among the different nodes of the query engine is not an option. Another technique is to try to filter data at the storage level, as much as possible. Which means sending over the network only a small part of the dataset and internally transforming the query in such a way that the processing will take place in the storage, which will eventually filter out irrelevant records. In our implementation, we heavily took advantage of this technique which exploits data locality and allows us to push part of the processing of the federator to the storage side.

In order to demonstrate the JOIN operator over the fragmented table work, let’s take a look at Figure 37.

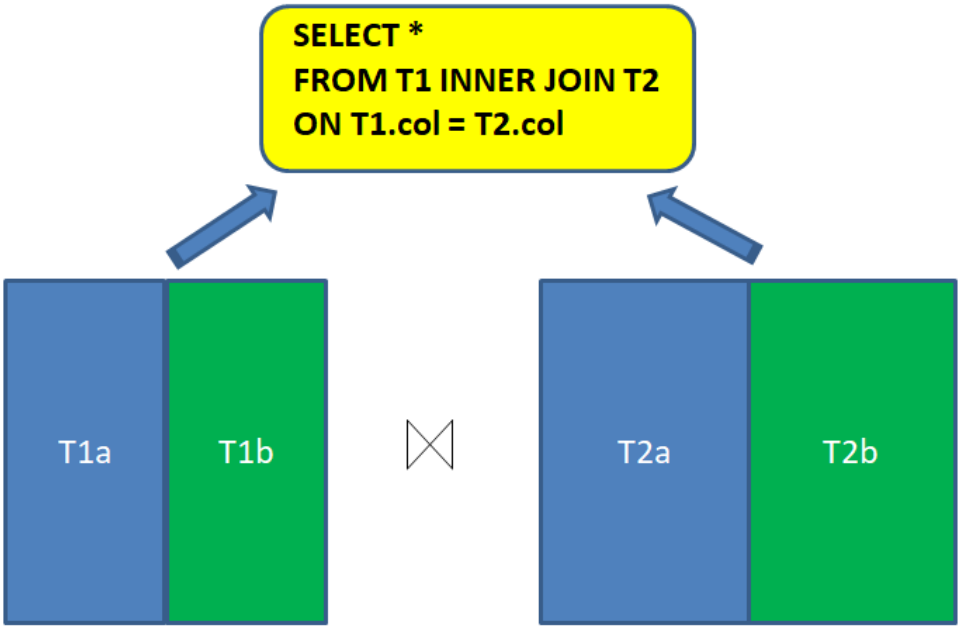


Figure 37: Example of a join operation over two fragmented tables

The user wants to perform a join operation over two tables (T1 and T2) using an equity condition. We assume that both tables are fragmented and exist both in the operational datastore of LeanXcale and the Object Store. T1a and T2a will be the part of the data tables that are stored in LeanXcale while T1b and T2b are the part of the tables stored in the Object Store. As both parts are stored in different nodes, by default, the query federator should request all the tables data from both stores, and then apply the operation in the query engine level. As the tables are split into the Object Store, since the operation is stateful and data items might need to be kept in memory during the whole execution of the JOIN it is very likely that the data size will be well over the available amount of memory,

In order to overcome this barrier, we take advantage of the properties of the JOIN operation itself. The result of the execution of the operator in memory, as just described, can be equivalent to the union of the results of the execution of the algebraic product of the initial operation, as depicted in Figure 38.

Figure 38: The join operation as the algebraic product

Here, we can see that the JOIN is being split into 4 separate ones which do not require any shared state and therefore can be executed in parallel.

Moreover, they can take advantage of data locality to boost the overall performance:

The T1a|X|T2a part, is the join of two data fragments both stored in LeanXcale. This will be translated into a single join that will be pushed down locally and will be performed by the LeanXcale query engine.

Same applies to the T1b|X|T2b part which can be pushed down for a local execution in the Object Store, while the query federator will only take the final results that will be returned back to the user. It is very important to highlight at this point that both those data segments are expected to contain a significant amount of data volume, as they correspond to tables that need to be split with an object store and, as a result, they involve BigData with possibly petabytes of data items. Taking advantage of the *Dynamic File Prunning* of IBM's Object Store, that enhances its data skipping capabilities with join support, this part of the operation can be also executed efficiently when predicates are applied in the JOIN query.

For the two parts that require joining data items from records located in different datastores, we apply a technique that is known as *bind-join*, which was introduced first by IBM back in 1992. The purpose of this technique is to send and move as little data as possible while executing the operator. It relies on the transformation of the JOIN into an *IN* clause and it works as follows.

1. Firstly, identify which of the two operands of the JOIN is expected to retrieve less results. In our case, it will almost always the table that is stored in LeanXcale, as the Object store is expected to hold 90-95% of the data table. In the general case, let's assume the left operand is smaller.
2. Push down to the left side as much operators as possible (in our case we push down filter conditions and projections) and return the data items, that can be kept now in memory. A hashmap is used, having as key the columns

involved in the JOIN, while the value contains a list of rows that share the same columns.

3. Now filter out and retrieve data items from the right operand, by pushing down as much as possible (again projections and filter conditions) and by applying the IN clause, whose value will be the list of the columns retrieved in the previous step.
4. After retrieving data items from the right operand, get the corresponding rows from the hashmap, construct the final result by merging both rows, and return it back to the caller.

Figure 39 illustrates this algorithm.

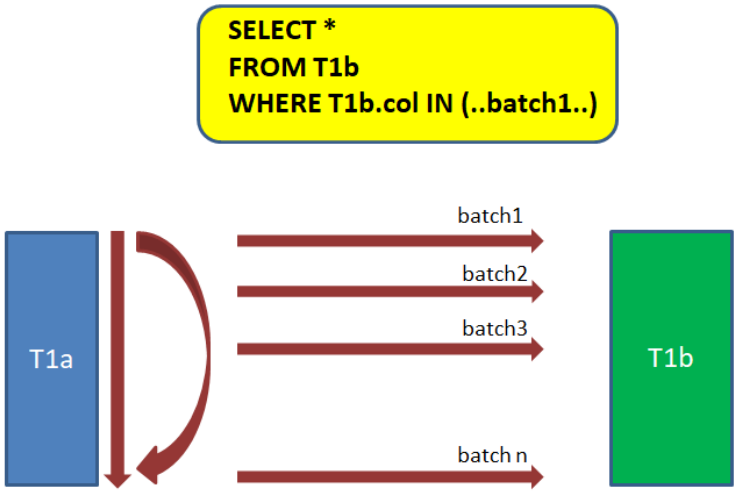


Figure 39: Bind Join execution

It is important to mention that by applying this technique, we only send the values of the columns that need to be part of the JOIN in the right operand (the Object Storage side), thus minimizing up to the absolute necessary the amount of data transmitted over the network and kept in memory.

Moreover, we parallelize the execution of the bind join, which allows us to keep the records we retrieved from the left operand (the LeanXcale part) in memory only for as long as necessary, and not during the whole execution of the operation. In fact, as we retrieve data from the left part, we send batches of results to the right part for execution, in parallel. The thread which executes the bind-join for a specific batch on the right side, will only need to keep the data items related with the columns that were sent to the object store, and not the whole hashmap. Thus, by the time that the thread which has received the data items from the right part, has merged the rows and has returned the result, it can free its internal hash map and stop, without having to keep in memory the whole table T1.a.

Last but not least, depending on the batch size and the number of rows returned by the left part, we might end up having thousands of threads trying to execute in parallel the bind-join

on the right side. This would require maintaining thousands of connections, which is not feasible. Due to this, we use the producer-worker design pattern, where the thread that retrieves data from the left part is the producer, and creates *tasks* for the workers to take them from an interim *blocking queue* used for orchestration, and the workers execute the bind join in parallel in the right side. Both the batch size and allowed number of workers are configurable.

To conclude, we showed in this sub-section how we have implemented the JOIN operator over fragmented datatables that are split across the two datastores. We take benefit from the equivalent execution of the algebraic product of the JOIN operator, in order to break the latter into 4 different parts: two of them can be executed locally in both stores, and the two other needs a combination of techniques to be executed in an efficient manner. We relied on the bind join for the last two, which allows us to only send the absolute minimum data over the network and filter as much as possible at the storage level. We also parallelized their execution in order to avoid holding in memory more data items than what is absolutely required. As the executions of the 4 parts of the JOIN are totally independent, as they do not share any state, we also execute them in parallel in separate threads. As the Federator JOIN operator returns a ResultSet, it implements an Iterator. This iterator orchestrates the execution of the 4 sub-elements and share a common data queue as a data pipeline with those threads. When each of the thread has a row to return (either by executing a standard join statement locally, or by transforming to a bind join) it puts it in the queue so that the iterator can retrieve it instantly. When a thread has no more records to retrieve, it sends a 'FINISHED' message to the queue and set up a flag so that the central orchestrator can know its status. The flag for the executors of the local joins is a Boolean attribute, whose value should be set to true when they finish. In the case of the bind joins, it is an Integer, whose value should be set to 0 when everything has finished. When a worker thread starts for the execution of a bind join for a given batch, the value of the flag is incremented by 1, and when this thread finishes, it is decremented by 1. In that sense, it will only have value 0 when all threads finish. Finally, as the central iterator makes use of this data queue, it immediately returns the data from the queue, when data is available. Therefore, it never keeps data items in memory for further processing. Even if the result of the right part of the local join in Object Store is expected to be very big, it is being retrieved immediately when a new records arrives, so it is up to the data analyst and the application layer to further manage it.

7.7 Prototype

In the interim review we successfully demonstrated a functional prototype of the full seamless component showing that all the SAF sub-components were implemented and integrated.

In more details the following were demonstrated:

1. The definition of the data schema used for the Ship Management use-case, including the necessary extensions for data movement.

2. Loading data for a specific data table of the use case firstly to the OS until the timestamp 2018-01-01 and then to LeanXcale the rest of them. Also, we loaded the entire dataset to an instance of vanilla LeanXcale.
3. Querying a dataset in four ways: through the seamless component, directly from the federated DB, directly from the Object Store and then to the vanilla LeanXcale instance. This demonstrated that the dataset was stored in both data stores and that the seamless component could merge the intermediate results, filtering out possible records that co-exist in both stores. The results that were gathered from the Seamless Analytical Framework were the same as the ones retrieved by the vanilla LeanXcale, proving the correctness of the Query Federator.
4. We triggered the data movement process and then execute the same queries as in the previous step. The results were the same proving that the Seamless Analytical Framework as whole works transparently from the user when moving data from one store to the other, and the retrieved results were the same in all cases as with the vanilla instance of LeanXcale, thus our prototype can allow to retrieve data truly seamlessly.

The queries that were used to validate the prototype were focused on different types of cases in order to cover all different operations that have been implemented by the Query Federator. They are summarized as follows:

```
SELECT COUNT(*)
FROM DANAOS.VESSEL_ENGINE_WEATHER_DATA
```

This query performs a full scan applying the count aggregation operation. It was used to show the number or rows in each federated datastore, and that this changed when a data movement takes places, in order to validate that data has been actually moved from one store to the other. Moreover, it validates the Aggregation operation of the Query Federator.

```
SELECT VESSEL_CODE, DATETIME, WIND_SPEED
FROM DANAOS.VESSEL_ENGINE_WEATHER_DATA
ORDER BY DATETIME
LIMIT 10
```

This query performs a scan, ordered by the timestamp in order to validate that data are stored in both federated stored until/from the specific splitting point. It also validates the Ordered Scan operation of the Query Federation.

```
SELECT VESSEL_CODE, max(WIND_SPEED) AS MaxWindSpead,
avg(FOVOLCONSUMPTION) AS AvgVesselPower,
sum(FOVOLCONSUMPTION) AS SumFOVOLConsumption
FROM DANAOS.VESSEL_ENGINE_WEATHER_DATA
WHERE DATETIME > TIMESTAMP('2017-12-01 00:00:00')
GROUP BY VESSEL_CODE
```

This query validates the most complex operation of the Query Federator, which is the Group by Aggregator.

7.8 Use Case Mapping

The *Seamless Data Analytical Framework* is based on two main pillars: The *Federator* mechanism which lies on top of the two involved data stores and is the central point of access to the data of the platform, and the deployed *pipeline* that is being used for transferring historical information from the LeanXcale relational data store to the object store. Regarding the latter, the real-time ship management use case was mainly used to validate its functionality as demonstrated in the mid review. This use case produces IoT data coming from various sensors deployed on its fleet, which are ingested in the store on a per-minute basis for each of the vessels of the fleet. This information can be considered historical after a certain point in time, and will be transferred to the object store, through this pipeline.

The *federator* which includes the introduction of the JOIN operator in the last phase of the project can be also considered as the main access point to the data stored in the platform. it is used by many components of BigDataStack that require the storing or retrieval of data: the integral components of the platform (i.e. data toolkit, and other WP5 tools, the cleaning process and the CEP engine of Data as a Service block etc.) and the use case applications.

Update operations are pushed down by the *federator* directly to the LeanXcale relational data store which provides transactional semantics.

Read operations, are pushed down both to the LeanXcale data store and to the Object Store.

For demonstrator purposes of the JOIN operation, we relied on the *Connected Consumer* use case, where there is a need to join data tables, some of those maintaining data that are being ingested and rarely modified, which can be considered historical at some point and can be moved to the Object Store.

7.9 Experimental Results

During the interim review, we successfully demonstrated the seamless component. However, after that, LeanXcale has upgraded its internal query engine in order to improve the performance of the query optimizer, and therefore by the time we are writing this deliverable, the migration of the query federator component of the seamless analytical framework to the new query engine is in progress. For this reason, we defer the experimental results to D4.3.
However, the experimentation plan is to use the CH-benCHmark, that combines the (OLTP-based) TPC-C with the (OLAP-based) TPC-H standards. We'll take results running the vanilla TPC-H on the object store, and then the CH-benCHmark on the seamless. Then, we'll run the TPC-C on vanilla LeanXcale, having already the results of the CH-benCHmark and we'll try to prove that the performance overhead is be very low.

The overhead of the federator in terms of latency should be zero, when requesting data that are stored in the LeanXcale data store only.

1. The overhead of the federator in terms of latency should be zero, when requesting data that are stored in the object store only.
2. The overhead of the federator in terms of latency should be less than defined value when joining data coming from both data stores.
3. The responsiveness of the framework should be the same when executing heavy analytical queries, while at the same time, the LeanXcale data store exports the historical data and pushes them to the SAF Mover.
4. The responsiveness of the framework should be the same when the LeanXcale data store exports the historical data under heavy operational workloads coming from the IoT data ingestion that happens in parallel.
5. The framework can handle Big Data analytics, when the result is of a significant size.

7.10 Next Steps

The delivery of the first version of the prototype of the Seamless Analytical Framework not only covered all requirements and target objectives that were defined and agreed during the first phase of the project, but also has already implemented some additional complex SQL query operators that were initially planned for the second period of the project. To be more precise, the first prototype was planned to be able to query federated data that is stored in both stores and to be able to move data from the operational datastore to the object store. However, data retrieval was intended to involve only scan operations while most complex operators were foreseen to be implemented for the second version. However, in M18 the Aggregation operators including GROUP BYs were successfully demonstrated, while the use of the Apache Hive metastore for sharing the common catalogue accelerated the integration process of the Query Federator. This revealed the true potentials of our solution in a pragmatic and real-life scenario, and therefore the next steps have already been planned in order to widen its scope and make it possible to be used in real life scenarios, overcoming the research assumptions that triggered its first version implementation. The following important steps are now planned to be delivered at the end of the project.

Firstly, a target objective until the end of the project is the support of the JOIN operator, involving tables that are split among the federated stores. The operator will be based on the ability of both the operational datastore and the object store via Apache Spark to execute JOINs locally. Therefore, the Query Federator is in position to push down these operators to both stores and merge the results. This will allow investigating efficient strategies that will take into account the data locality in order to reduce the amount of data that needs to be sent over the network. To be more precise, the implementation of the JOIN operator typically requires data to be sent to the query engine of a datastore so as the latter can be

able to perform the *join*. Optimization techniques can reduce the data to be moved across the distributed nodes, however the ability of both nodes (the two federated datastores in our case) to *join* data can increase the efficient even more. The operation can be translated in the union of the 4 separate joins where two of those can be executed locally, and the rest will require a minimum amount of data to be sent to one of the two stores, potentially exploiting strategies like the *bind join*. The support of the JOIN operation by the Query Federator will make possible for the Seamless Analytical Framework to cover most of the use cases.

Secondly, given the fact that the Query Federator is extending the Query Engine of LeanXcale and taking into account the support of JOIN operations, the Query Optimizer of the operational datastore can be further improved in order to identify tables split among the two stores and generate different query execution plans. Until now, the Query Federator directly returns the result to the end user. However, the next plan is to provide the ability to be used in a wider SQL query, where it will be part of one of the nodes of the tree of the execution plan. Hence, it will be an integral operator that will be part of the data pipeline of the operators that execute the query and will have to return the result to the upper layers. In order to do so, the Query Engine will have to generate different query execution plans taking into account the unique nature of the Query Federator and the latter to be able to accept operators than can be pushed down by the Query Engine, and re-write the queries to be executed in the federated datastores accordingly. Having these two functionalities developed, the Query Optimizer of LeanXcale could then further improve its plan so that the overall data retrieval can be more efficient.

8 Data Quality Assessment

8.1 Introduction

The data quality assessment mechanism aims at evaluating the quality of the data prior to any analysis on them, to ensure that analytics outcomes are based on datasets of high quality. To this end, BigDataStack architecture includes a component to assess the data quality. The component incorporates a set of algorithms to enable domain-agnostic error detection, in a tabular dataset.

8.2 Requirements Specification

	Id	Level of detail	Type	Actor	Priority
	REQ-DQAI-01	Software	DATA	Developer	MAN
Name	Data store connection				
Description	The Data Quality Assessment and Improvement module should be able to connect to the Data Source, to retrieve, update and validate the available tables.				
Additional Information	The connection is established via a JDBC driver.				

Table 36 - Requirement REQ-DQAI-01 for Data Quality Assessment & Improvement

	Id	Level of detail	Type	Actor	Priority
	REQ-DQAI-02	Software	DATA	Developer	OPT
Name	Data Augmentation				
Description	The Data Quality Assessment and Improvement module should be able to augment the original dataset, to design a new set that alleviates the problem of class imbalance, for modelling purposes. This problem exists because there are only a few corrupted examples in any given dataset, assuming that most of the corpus is valid.				
Additional Information	The data augmentation module can ingest random typos, format errors and value swaps and is an optional component in the DQAI pipeline.				

Table 37 - Requirement REQ-DQAI-02 for Data Quality Assessment & Improvement

	Id	Level of detail	Type	Actor	Priority
	REQ-DQAI-03	Software	DATA	Developer	MAN
Name	Correlation				
Description	The Data Quality Assessment and Improvement module should be able to compute the correlation between several attribute columns in a dataset. This way we can discard non-informative columns, such as primary keys,				

	that deteriorate the model's performance and accuracy.
Additional Information	The correlation metric that we use is the "uncertainty coefficient" or Theil's U, which is a non-symmetric correlation metric. This can uncover information that otherwise would be lost, if for example the metric used was the "Pearson correlation coefficient" or "Crammer's V".

Table 38 - Requirement REQ-DQAI-03 for Data Quality Assessment & Improvement

	Id	**Level of detail**	**Type**	**Actor**	**Priority**
	REQ-DQAI-04	Software	DATA	Developer	MAN
Name	Model storage				
Description	The Data Quality Assessment and Improvement module should be able to serialize and store the artefacts of the model.				
Additional Information	The artefacts contain the model itself as well as several other entities, such as tokenizers and parameter configuration.				

Table 39 - Requirement REQ-DQAI-04 for Data Quality Assessment & Improvement

	Id	**Level of detail**	**Type**	**Actor**	**Priority**
	REQ-DQAI-05	Software	DATA	Developer	MAN
Name	Scheduler				
Description	The Data Quality Assessment and Improvement module should be able to request new data that were inserted in the data store repeatedly. Thus, a scheduler is implemented to make those requests periodically.				
Additional Information	The scheduler is implemented as a simple python script.				

Table 40 - Requirement REQ-DQAI-05 for Data Quality Assessment & Improvement

	Id	**Level of detail**	**Type**	**Actor**	**Priority**
	REQ-DQAI-06	Software	DATA	Developer	MAN
Name	Database ingestion module				
Description	The Data Quality Assessment and Improvement module should be able to upsert the assessed data back in the data store.				
Additional Information	The ingestion module is inserting assessed data directly via the transaction manager, bypassing the SQL layer. This makes the process significantly faster and more efficient.				

Table 41 - Requirement REQ-DQAI-06 for Data Quality Assessment & Improvement

8.3 User Story

The analysis of data and, consequently, Big Data passes through several district phases, with each of them posing different challenges. In contrast to the much more researched

modeling phase, the cleaning step is often seen as a sore point, even though without it the modeling phase is of little use. During this phase, domain experts either manually check the incoming signal from every sensor, or systemically clean the arriving data using predefined criteria.

Our view on the problem takes advantage of the recent developments in artificial neural networks (ANN) and deep learning (DL), to extract latent features that correlate different fields, and identify possible defects in their measurements. By passing each measurement through a deep ANN, we get a distributed representation of each concept, which is a much more comprehensible notion for the machine. This process helps us gain knowledge about the connections between fields and diagnose irregularities in the input data.

Moreover, in the case that data correlation is not possible, due to many missing values for example, we can detect errors within a single column by applying Language techniques like the n-gram models.

8.4 Detailed Design

8.4.1 Approach

While current solutions in data cleaning are quite efficient when considering domain knowledge (e.g., when functional dependencies are given), they provide limited results regarding data volatility, if such knowledge is not utilized. BigDataStack provided a data quality assessment service that exploits Artificial Neural Networks (ANN) and Deep Learning (DL) techniques, to extract latent features that correlate pairs of attributes of a given dataset and identify possible defects in it. Furthermore, if discovering dependencies between pairs of attributes is not possible (e.g., due to heavy anonymization), Data Quality Assessment component employs techniques that can detect errors within a single column, using n-gram models.

The key issues that need to be handled by the Data Quality Assessment service are:

- Work in a context-aware but domain-agnostic fashion: The process should be adaptable to any dataset, learn the relationships between the data points and discover possible inconsistencies.
- Model the relationships between data points and reuse the learned patterns: The system should store the models learned by the machine learning algorithms, and reuse them through an optimization component, which checks if the raw data have similar patterns, dataset structure or sources. In that case, already existing models should be activated, to complete the process in an efficient manner.
- Fall back to within-column error detection if discovering relationships between the data points is not possible for any reason.

8.4.2 Detecting Errors via Pairs of Attributes

To discover possible deviations in data, we need to learn and predict the relationships between data points. To this end, we strive to exploit the recent breakthroughs in Deep Learning, and the idea of an embedding space. Figure 40 depicts a serial architecture, which tries to predict if two entities are related to each other.

Given the learned distributed encodings of each entity x, y or, in our case any data point, we can discover if these two candidate entities or data points are related. Thus, considering the DANAOS Real-Time Shipping Management use case, if the temperature sensor emits a value that is illogical given other rpm sensor readings, the relationship between these two data points would be associated with a low score (or probability). This could provide significant improvements in the results of an analytical task, which the data scientist wants to execute and is part of a general business process.

To optimize the data quality assessment process, we introduce a subcomponent that retrieves previously learned models, when a similar dataset structure arrives in the system, or the same data source sends new data.

Data quality assessment component inputs:

- The raw data ingested by the data owner through the Gateway & Unified API
- The data model provided by the optimizer if exists
- User preferences and specifications, ingested through the Data Toolkit

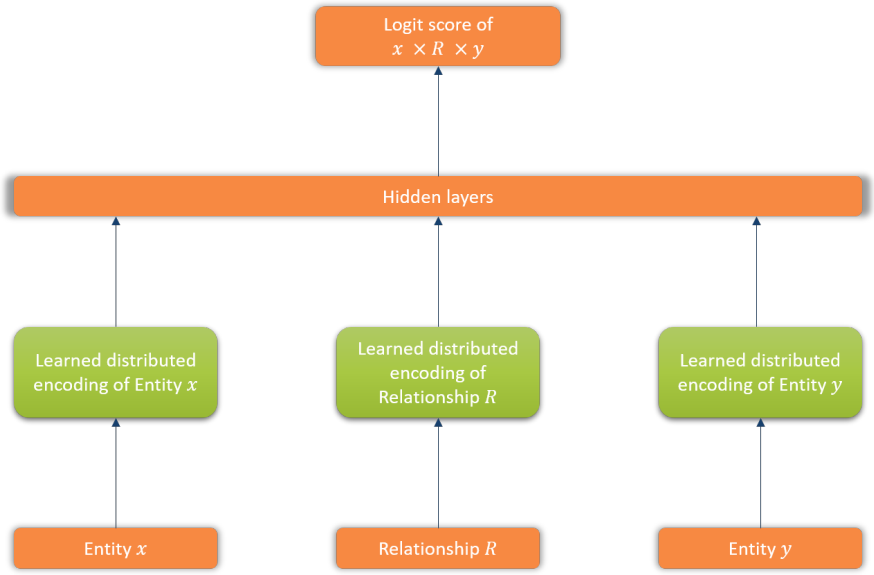


Figure 40: Domain agnostic data cleaning model architecture

Data cleaning component outputs:

- Assessed data, establishing data veracity
  - A probability score for each tuple in the database column
- Trained, reusable ML models, stored in a repository for later use

The main structure of the Data Quality Assessment component is depicted in Figure 41. Based on this figure the flow is as follows:

- The Data Pre-processing unit takes raw data and converts them in a form that the machine learning algorithms can work with
- The main pillar of the service is the data cleaning component, which takes the pre-processed data as input, trains a new model and stores it in the model repository
- During the assessment phase, a scheduler pulls newly ingested data to be assessed
- The data quality assessment module retrieves the learned model from the repository and makes the necessary predictions
- The assessed data are updated into the distributed storage

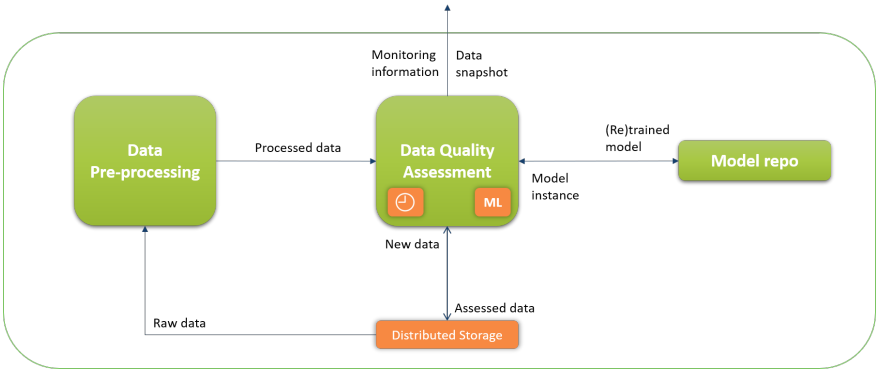


Figure 41: Data Cleaning Module Architecture

8.4.3 Detecting Errors via n-gram Models

To use n-grams to detect format errors in tables, we need a way to generalize raw values to patterns. We could also train an n-gram model with the raw values, but the complexity of that approach introduces too many degrees of freedom, leading to high sparsity.

For example, let v_1 be a raw sequence of numbers expressing a zip code: 15122. Then, let v_2, v_3 signify two different codes: 345a7, 47592. Clearly, v_2 is erroneous, but if we pass the raw values in an n-gram model we will not get valuable information; see that $p(a|5) = p(1|5)$, thus we either miss the error or produce a false positive. But if we could generalize a raw value into a pattern, we could get more meaningful representations.

To this end, we use the notion of generalization trees and generalization languages. A generalization tree is just a hierarchy like the one in Figure 42, mapping raw values to a different representation.

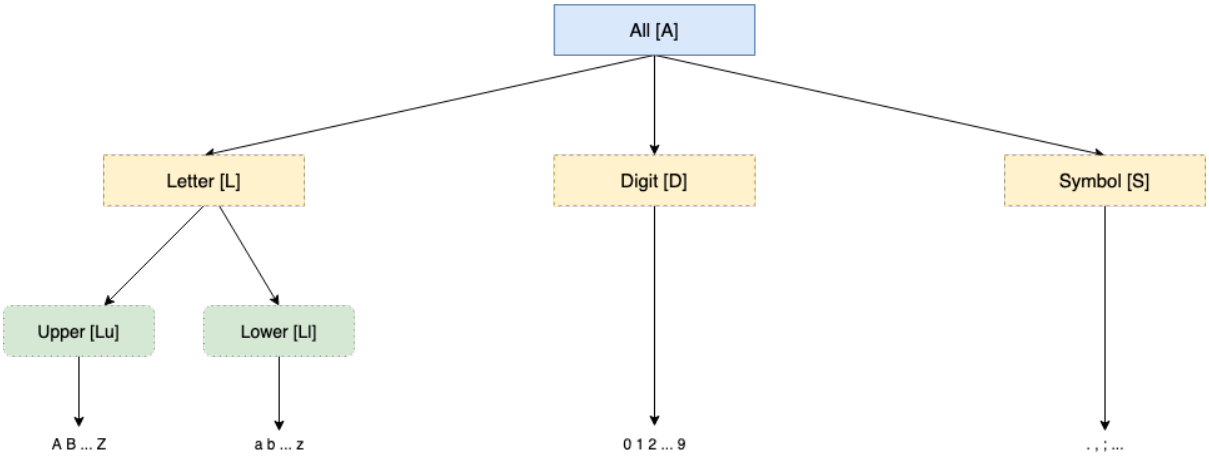


Figure 42: A Generalization Tree

From that tree, we can derive many languages. An example is given in Figure 43.

$$L_1(x) = \begin{cases} \text{L} & \text{if x is a letter} \\ \text{D} & \text{if x is a digit} \\ \text{S} & \text{if x is a symbol} \end{cases}$$

Figure 43: A Generalization Language

Using L_1 we can transform v_1, v_2, v_3 into DDDDD, DDDLD, DDDDD. We could also compress that pattern to v_1 = D(5), v_2 = D(3)L(1)D(1), v_3 = D(5). Having this representation and a big enough dataset to train an n-gram model, we can get that the probability of having a letter L in a zip-code format is extremely low.

8.5 Prototype

The main structure of the Data Quality Assessment component is divided into two steps. The training phase and the assessment phase.

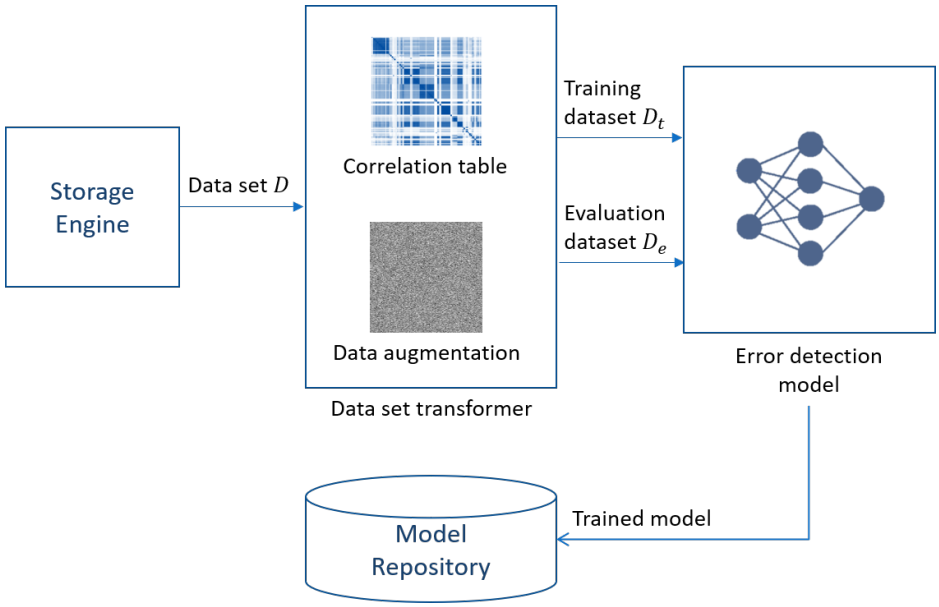


Figure 44: Training phase

Figure 44 depicts the training phase of the DQA component. Initially, the module requests the dataset from the storage engine via a simple SQL query. Our experiments have shown that a small fraction of the dataset is enough for our model to learn. This is because we assume that most of the dataset is correct and the data augmentation model that we have implemented can create most of the errors that we would encounter.

Then the Data Set transformer module extracts the top correlated features and introduces new errors via the data augmentation submodule to the original data set. These can be random typos, format errors and value swaps. The augmented dataset is then split into two datasets for training and evaluation which are fed to the model (i.e. neural network). After the training process, the trained version of the model is serialized and stored in the model repository.

As depicted in Figure 45, during the assessment phase a scheduler, which is a simple python script, requests the newly added tuples in the dataset. This is done via a simple SQL query that limits the results with a *WHERE* clause, retrieving only the tuples in the dataset that has not yet been assessed. Then, the module loads the trained serialized model in memory and assigns a probability to each new tuple. This probability number is then stored back in the data store, describing each tuple by its primary key.

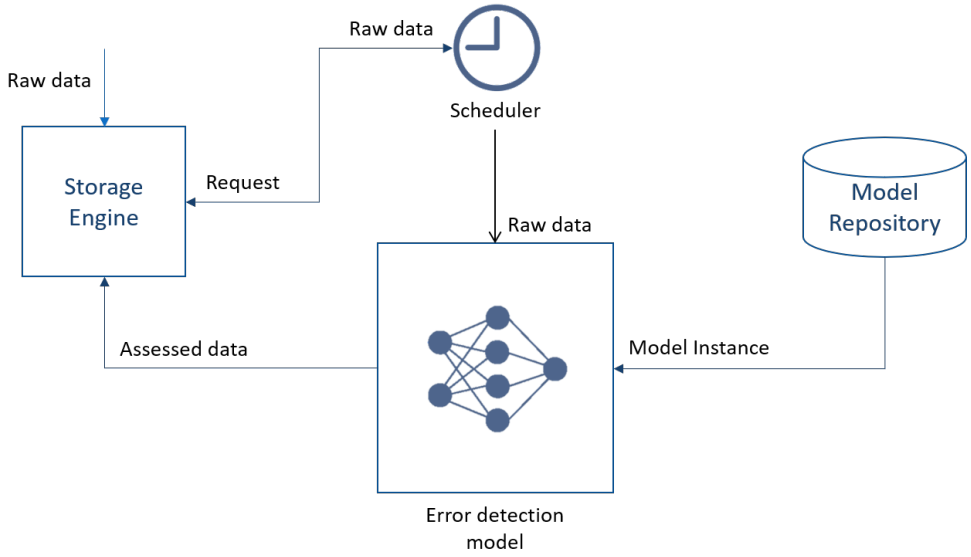


Figure 45: Assessment phase

8.6 Experimental Results

We compare our approach against several, well established error detection methods. We also analyze how the performance changes when we alter the number of the top-attribute correlations we consider (i.e., hyper-parameter value k) for the pairs of attributes approach, and what our augmentation methodology offers in our attempt to alleviate the problem of class imbalance. Also, we report the results of out n-gram models approach for the Smart Insurance use case.

8.6.1 Pairs of Attributes

In this section we describe the data sets we use, how the errors were introduced, the other methods that we compare against, as well as the metrics and configuration we used.

For the evaluation of the *Pairs of Attributes* approach, we used the two datasets from diverse areas of domains. Table 42 presents a summary of each data set and their attributes. As shown the data sets vary significantly in size and number of attributes.

Table 42 - Datasets

Dataset	Size (rows)	Attributes
Hospital	1K	19
DANAOS main engine	29M	100

- The hospital data set is the baseline benchmark for almost every error detection process. It is a small data set and is treated as an easy challenge and more like a proof of concept. The errors that we come across this data set are artificially generated typos.

- The DANAOS dataset refers to the shipping management use case of BigDataStack. For a detailed view of the dataset see D2.5 – Conceptual model and Reference architecture – II.

We consider several different methods as our baselines. Some are searching from specific side-effects, such as outlier detection, functional dependencies, etc., and others are more holistic and automated. These are:

- Logistic Regression (LR): The first baseline is a simple supervised classifier that checks pairwise co-occurrence statistics of attribute values and constraint violations to detect corrupted cells.
- Constraint Violations (CV): A classical rule-base error detection method.
- Outlier Detection (OD): This approach utilizes correlations to identify possible outliers. For instance, consider a set of attributes A and a specific attribute A_i taken from that set. To detect possible outliers, this method looks at all correlated attributes in $A \setminus A_i$ with A_i and examines their pair-wise conditional distributions.
- Forbidden Item Sets (FBI): This approach detects pairs of values that are implausible to co-exist in a data set. To achieve this, it uses the lift measure from association rule mining.
- HoloDetect (AUG): HoloDetect uses several representation models that feed to a classifier to capture the inherent syntactic and semantic heterogeneity of errors, as well as a data augmentation learning technique to address the problems of class imbalance and data scarcity.

We measure the accuracy of our approach in two ways. First the binary classifier that detects unlikely co-existent pairs of attributes is evaluated by computing the approximate AUC (Area under the curve) via a Riemann sum. Then, to assess the accuracy of our model at cell level, we use Precision (P) defined as the fraction of correct error predictions and Recall defined as the fraction of true errors that are predict as such. We also give the F1 score, defined as the harmonic mean of Precision and Recall:

$$F1 = 2\frac{precision \cdot recall}{precision + recall}$$

For training we take a clean subset of the data set and create a corrupted one by introducing errors using our data augmentation algorithm. This algorithm ingests random spelling mistakes, format errors and value swaps (random permutations). During evaluations we assess our approach using the original erroneous data set or a new one where errors are introduced using BART.

For the optimization algorithm we use the ADAM optimizer and train our model for 200 epochs with a batch size of 64. For each data set, the value of hyper-parameter k, which defines how many pairs of attributes to consider, is not constant and is reported in the results.

In Table 43, we present the results of the algorithm for different values of the hyperparameter k (the number of pairs of attributes,i.e. correlations, to consider) for the hospital data set. The metrics we report are Precision, Recall, F_1 score and AUC score. As expected, we get the best results for the values of k that are in between the two extremes. This is because when we have a small number of correlations, we lose valuable information, while for a large value of k we are starting to introduce noise in the training set.

Table 43 - Results for different values of k

k	Precision	Recall	F1	AUC	Examples
50	0.7696	1.0000	0.8698	0.9729	72590
60	0.7212	1.0000	0.8380	0.9627	92544
70	0.6907	0.9975	0.8162	0.9754	109475
80	0.8997	1.0000	0.9472	0.9965	117358
90	0.8675	1.0000	0.9290	0.9914	131666
100	0.8693	1.0000	0.9301	0.9950	145873
110	0.9531	1.0000	0.9760	0.9989	162496
120	0.7067	1.0000	0.8281	0.9770	178035

Finally, in Table 44 we see that DQA outperforms all other benchmark methods in the hospital dataset. We also see the experimental results on the DANAOS dataset that we presented in sub-section 7.8.

Table 44 - Precision, Recall and F_1 score

Dataset	M	DQA	AUG	LR	CV	OD	FBI
Hospital	P	0.953	0.903	0.0	0.030	0.640	0.008
	R	1.000	0.989	0.0	0.372	0.667	0.001
	F_1	0.976	0.944	0.0	0.055	0.653	0.003
DANAOS	P	0.870	n/a	n/a	n/a	n/a	n/a
	R	1.000	n/a	n/a	n/a	n/a	n/a
	F_1	0.931	n/a	n/a	n/a	n/a	n/a

8.6.2 N-Gram Error Detectors

For the Smart Insurance use case we introduced a new feature in the Data Quality Assessment component that can detect errors within a single column. When applying this technique to the GFT insurance data, we discovered several errors, which could cause problems to downstream analytical tasks. For example, consider Figure 46.

marca	is_dirty
HAMAMATSU	0.499692118226601
YHAMAA	0.49892912264211...
YAHAMA	0.49846245612893...
AMBULANZA	0.49723856010995...
ATLANTIS	0.4968141118926348
RENAUTL	0.4966112770724421
AMERICAN INTERN...	0.4966080112802509
TIAG	0.4965233878433648
MAGGIOLINO	0.49599105163129...
HAMANATSU	0.4957990433475348
HIUNDAY	0.4956521739130435
PHOON GILERA	0.4955027494108405
KNAUS	0.4952388686763687
KNAUS E 450	0.4952388686763687
FERCAM	0.49494142408516...

Figure 46: GFT Insurance Dataset

In this table we store the insurance vehicles brand and model. In the top-10 results with the highest probability of being "dirty" we come across errors like YHAMAA and YAHAMA instead of YAMAHA, or RENAUTL and HIUNDAY instead of RENAULT and HYUNDAI.

Thus, a user could set a threshold and retrieve only the rows that have probability of being dirty below 0.49 and drop those rows. Alternatively, an expert could inspect those rows and correct the errors. This is now possible because the user will not have to go through the entire data set, which could be millions of rows.

8.7 Next steps

This component has been utilized already and evaluated by the Real-Time Shipping Management use case during the first half of the project, while we turned our attention to the Smart Insurance use case during the second half.

During the course of the project we implemented two different algorithms to address separate challenges; one algorithm that detects the interdependencies of the features in a dataset and tries to identify inconsistencies, and one algorithm to detect errors within a single column. The second algorithm is really useful when we have a heavily anonymized dataset or with many missing data, where the information we have is scarce and we cannot identify correlations.

Our goal is to unify these algorithms and provide a Data Quality and Assessment framework that can work under any circumstances. We have released part of the Data Quality Assessment component as an open-source Python library available through PyPi (https://pypi.org/project/forma/).

9 Predictive & Process Analytics

This section describes the BigDataStack components that implement the functionality of Predictive and Process Analytics, as summarized below. **It should be noted that additional mechanisms that have been implemented in the scope of process analytics are described in Section 10, Log Search.**

Predictive analytics: Machine learning algorithms such as collaborative filtering or k-means run over large data sets. The parallelization of those algorithms is needed in order to produce results with low latency. However, the parallelization of these algorithms or other workflows for data analytics is not trivial. Moreover, each framework (e.g., map-reduce, Spark …) uses different approach. The predictive analytics component of BigDataStack provides a framework for the parallelization of data workflows and uses the infrastructure provided by the CEP. Workflows are expressed as a dataflow graph in which nodes are custom operators (arbitrary Java code) and edges represent the flow of data. The workflow can be deployed on a distributed system where sets of operators run on different nodes. Operators can also be parallelized in a single node taking advantage of the multi-core architectures available nowadays. Finding the best configuration for workflow or machine learning algorithm may require repeated executions each time with different parameters (for instance, to train a model). The predictive analytics allow to store and share partial results of a workflow for other executions in order to achieve low latency results by avoiding the repetition of the computation. Many of these analytics algorithms need to share state which sometimes may be partitioned. The integration of the CEP operators with LeanXcale database allows the efficient distribution and access to the shared state from the CEP (Figure 47).

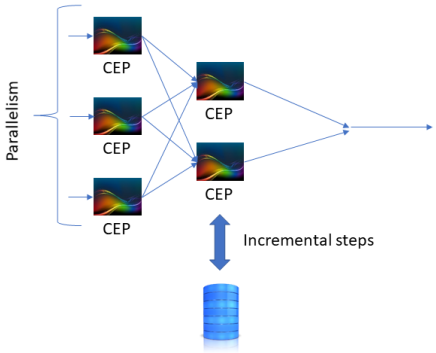


Figure 47: Parallelization and distribution of analytics workflow

Process analytics, which is able to analyze event logs of BigDataStack, discover processes, extract useful knowledge, and exploit this knowledge in order to assist the Process Modelling phase. Process discovery is performed using appropriate process mining algorithms, applied on the event log, as explained in the following. For example, Figure 48 illustrates the concept of process recommendation. Given a process trace that can be the outcome of Process Modelling, the Process Analytics component can provide feedback to the modelling phase in the form of recommending the next state that has the highest

probability of occurrence from a set of k possible next states, based on the discovered process models.

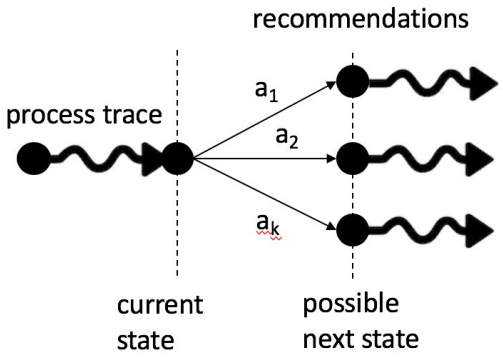


Figure 48: The recommendation of next possible state during process modelling

In addition, task T4.5 has been re-scoped in order to provide additional functionality. This consists of a *Catalogue of Analytics Algorithms*, which is perceived as a datastore that stores candidate algorithms that can be selected by the Process Mapping mechanism (WP5/Task 5.2), when passing from the process modelling to the data toolkit (in the flow). This Catalogue also holds the descriptors of the candidate algorithms, so as to make them searchable by an external application. It should be highlighted that this additional functionality is provided on top of the prescribed functionality of Task 4.5.

Figure 49: The interaction between Process Analytics and Global Decision Tracker and Process Modelling Framework

9.1 Requirements Specification

The following set of functional requirements have been identified regarding the operation of the Predictive and Process Analytics component. The first two requirements refer to system integration with two other components of BigDataStack, namely the Global Event Tracker and the Process Modelling Framework respectively. This is also exemplified by means of Figure 49.

The third requirement refers to the necessary pre-processing and data preparation operations, including data transformation in order to bring the input data to the appropriate input format. The last requirement dictates the use of ProM (http://www.promtools.org/), a widely used tool for Process Mining, by extending the functionality of ProM in order to become applicable to the event logs generated in BigDataStack.

Table 45 - Requirement REQ-RD-01 for Predictive & Process Analytics

	Id	Level of detail	Type	Actor	Priority
	REQ-RD-01	Stakeholder	FUNC	Developer	ENH
Name	Global event tracker connection				
Description	A connection to the Global Event Tracker (GET) is needed for the Predictive & Process Analytics component.				
Additional Information	The information stored in GET is crucial to the implementation of this module.				

Table 46 - Requirement REQ-RD-02 for Predictive & Process Analytics

	Id	Level of detail	Type	Actor	Priority
	REQ-RD-02	Stakeholder	FUNC	Developer	ENH
Name	Connection to the Process Modelling Framework				
Description	A connection between this component and the Process Modelling Framework needs to be established, so information can be sent and received.				
Additional Information	The recommendations made by this component will be in real time, as the Business Analyst – Data engineer is modelling the process.				

Table 47 - Requirement REQ-RD-03 for Predictive & Process Analytics

	Id	Level of detail	Type	Actor	Priority
	REQ-RD-03	Stakeholder	FUNC	Developer	ENH
Name	Data pre-processing				
Description	The data ingested by this component needs to be in an eXtensible Event Stream[32] (XES) file format. A tool is created, depending on the format of the Global Event Tracker.				
Additional Information	The XES standard defines a grammar for a tag-based language whose aim is to provide designers of information systems with a unified and extensible methodology for capturing systems behaviours by means of event logs and event streams is defined in the XES standard. An XML Schema describing the structure of an XES event log/stream and an XML Schema describing the structure of an extension of such a log/stream are included in this standard. Moreover, a basic collection of so-called XES extension prototypes that provide semantics to certain attributes as recorded in the event log/stream				

[32] http://www.xes-standard.org/

	is included in this standard.

Table 48 - Requirement REQ-RD-04 for Predictive & Process Analytics

	Id	Level of detail	Type	Actor	Priority
	REQ-RD-04	Stakeholder	FUNC	Developer	ENH
Name	ProM framework				
Description	ProM is an extensible framework that supports a wide variety of process mining techniques in the form of plug-ins.				
Additional Information	The process mining techniques used will be utilized to derive metrics of the event log, to create the semantics needed between events for the recommendation process.				

9.2 Design

Process Analytics

Figure 50 depicts the high-level design of the Process Analytics component, focusing mainly on its constituent sub-components and their inputs/outputs. As already mentioned, the basic input for this component is an event log. Since the event log may come in different formats, depending on the way data is captured and the application-specific logging mechanism, the first step is to transform the log in a format suitable for further analysis. This format is pre-specified and defined based on the input requirements of the next sub-component (the process analytics engine) in the processing flow.

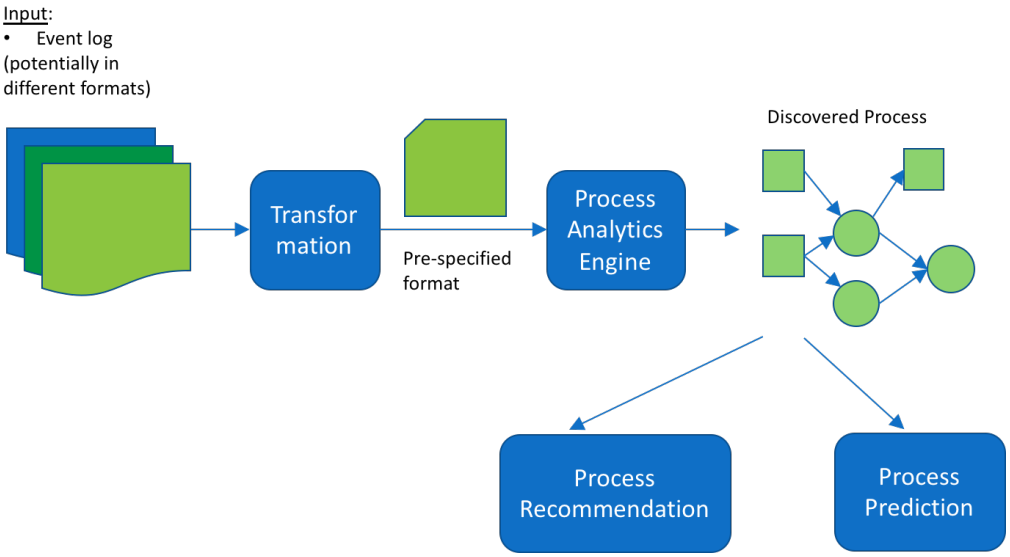


Figure 50: Design of Process Analytics component

After the transformation of event log data, the process analytics engine takes as input the transformed event log and performs the main step of analysis. As a result, a process model is generated capturing the steps of the underlying process and the possible paths between steps, which can typically be represented using a graph-like structure. Additionally, it is possible for the graph to have edges annotated with weights, which correspond to the frequency that a specific transition between steps (an edge) has been observed. Put differently, in a weighted graph representation, the weight corresponds to transition probabilities between steps of the process.

The output of the process analytics engine can be exploited during the execution of a process, in order to perform prediction of the next step. Furthermore, the same output can be exploited during process modelling, by providing recommendations to the process designer/modeler, based on the discovered patterns of process execution.

9.3 Prototype

At the time of this writing (month M35), the prototype for Process Analytics works as follows:

- **[Phase A]:** It receives as input event log data and builds a weighted graph representation that corresponds to the discovered processes. Weights represent the transition frequency between two nodes of a process.
- **[Phase B]:** A community detection algorithm is executed on the weighted graph, in order to find process states that are often used together. As will be demonstrated, this step may help in the case of complex processes, by discovering communities (i.e., process states that are often used together).

- **[Phase C]:** Then, given a specific process, we find the community in which it belongs, and we seek the next state within this community. In this way, we drastically narrow down the search space for the next process state.

In more technical details, the prototype of Process Analytics is implemented in Python. Phase A practically consists of parsing and extracting data from the event log. During the extraction, the weights of the underlying graph are computed. Internally, the graph is represented as a n×n matrix, where n corresponds to the process states in the event log. Each cell contains the frequency of transition between the corresponding process states (row and column).

In Phase B, the community detection algorithm is executed on the weighted graph. Obviously, different community detection algorithms are available in the literature and are readily applicable in our case. We choose to use the *Louvain method*[33] which is an efficient way to extract communities from large graphs. The method follows a greedy optimization approach with observed run in time O(nlogn). The Louvain method is widely used (in social networks, mobile phone networks, etc.) due to its salient features, namely its runtime efficiency as well as the extraction of communities of high quality.

Finally, in Phase C, the community that contains the given process trace is found. Then, this small subgraph is searched in order to find the candidate next states, and eventually select the most probable next state based on the weights.

In parallel with this working prototype of Process Analytics, another ongoing activity is the exploitation of the functionality of the ProM framework, as an integrated subcomponent of Process Analytics. In more detail, the research and development activities that have been carried out include:

- Applying process discovery algorithms on event log data. So far, our prototype includes the following algorithms from ProM: *Alpha Miner*, *Heuristic Miner*, and *Inductive Miner*.
- Exploiting PLG2, a software for random process generation, with various parameters that control the complexity of the underlying processes. The use of this software is helpful because it allows the evaluation of different process discovery algorithms over synthetic event log data of variable complexity, and also supports the inclusion of noise, which is typical in real-life event log data.

Preliminary results from this second prototype have been obtained from synthetically generated data. The integration of the working prototype with ProM/PLG2 together with a detailed experimental evaluation will be reported in the next version of this deliverable.

[33] Vincent D. Blondel, Jean-Loup Guillaume, Renaud Lambiotte, Etienne Lefebvre: Fast unfolding of community hierarchies in large networks. CoRR abs/0803.0476 (2008).

9.4 Experimental Evaluation

In this section, we report results obtained from the working prototype of Process Analytics. The prototype comprises two sub-modules: (a) a module that relies on community detection algorithm to discover process states that are often used together, and (b) a set of process mining algorithms from the open-source process mining tool ProM (http://www.processmining.org/prom/start).

9.4.1 Evaluation based on Community Detection

We used real event log data that resemble one of the use cases of the project. We focus on the connected consumer use case, which entails retail data, and we use the Online Retail dataset from the UCI Machine Learning Repository (https://archive.ics.uci.edu/ml/datasets/online+retail). This is a transnational dataset which contains all the transactions occurring between 01/12/2010 and 09/12/2011 for a UK-based and registered non-store online retail. The dataset consists of 541,909 records described by 8 attributes.

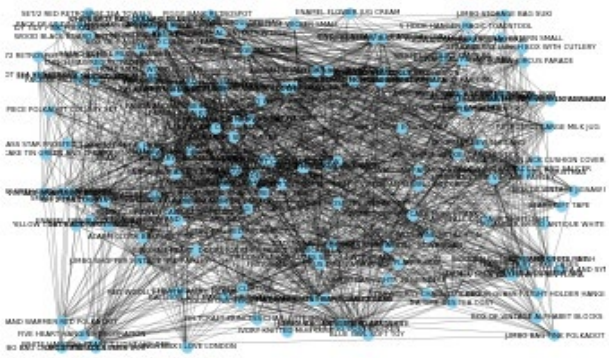


Figure 51: Graph produced from the analysis of the Online Retail dataset.

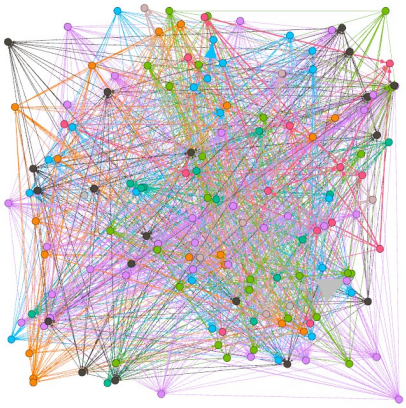


Figure 52: The same graph colored for better visualization.

Figure 51 depicts the graph produced when Phase A is applied on the Online Retail dataset. The nodes of the graph correspond to products, and the edge between two products relays the fact that they were bought together by the same customer. The weights of the edges are computed based on the frequency of purchases for a given pair of products. As shown in the figure, the resulting graph is highly connected, which complicates the task of identifying the next node. Figure 52 depicts the same graph with colored nodes, simply for improved visualization.

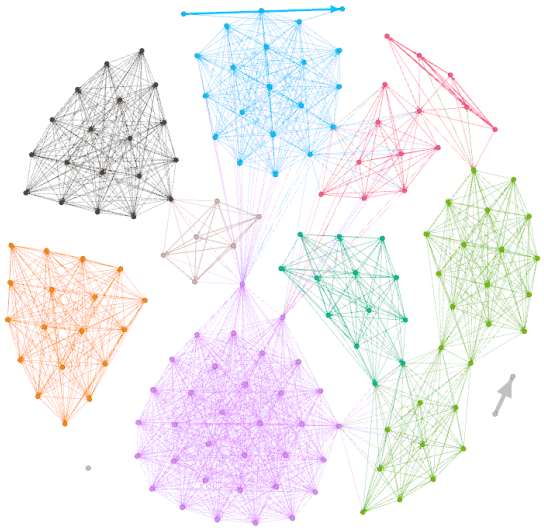


Figure 53: The result of the community detection algorithm when applied on the afore-described graph (colors represent different communities).

Figure 53 depicts the same graph after having applied Phase B, which corresponds to the execution of the Louvain community detection algorithm. Clearly, some communities have been identified and are represented with different colors. Apparently, this step simplifies the selection of the next node, given an already selected node or set of nodes. The reason is that the search space is reduced and also that the candidate nodes belong to the same community of the already selected node, which also helps in producing/recommending a node that is semantically close.

9.4.2 Evaluation based on Process Mining Algorithms

In addition, we perform a comparative sensitivity analysis of several process mining algorithms with respect to their performance in process discovery, for process models of variable complexity, size, and noise.

Algorithms. In the evaluation, we use notable process mining algorithms from the open-source software ProM, namely: *Alpha Miner*, *Inductive Miner*, *Evolutionary Tree Miner* και *Heuristic Miner*.

Datasets. For the generation of process models, we used the PLG2 tool[34] (https://plg.processmining.it/). This tool is flexible enough to produce synthetic process models, vary their complexity using appropriate input parameters in a controllable way. In this way, we obtain appropriate inputs for the process mining algorithms, thus allowing the study of their performance when varying a specific input parameter.

[34] Andrea Burattin. "PLG2: Multiperspective Process Randomization with Online and Offline Simulations". In Online Proceedings of the BPM Demo Track 2016; Rio de Janeiro, Brasil; September 18, 2016; CEUR-WS.org 2016.

Metrics. We use a variety of evaluation metrics that are suitable for process mining: Simplicity, Generalization, Precision και Fitness[35]. All these metrics refer to the discovered process model in comparison with the event log from which it is generated. *Simplicity* is used to capture the notion that the simplest model that can explain the behavior seen in the log, is the best model (also referred to as "Occam's razor"). *Generalization* refers to the ability of the process model to generalize, rather than restrict its behavior to data found in the event log. A model that does not generalize resembles the concept of overfitting in machine learning. *Precision* refers to not allowing too much room for diversity, and a model with low precision can be thought of as underfitting. Finally, *Fitness* refers to the ability of the process model to replay the event log.

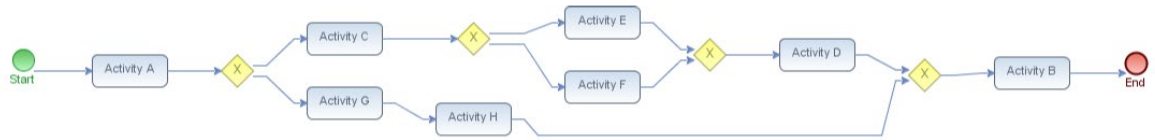


Figure 54: Process model (in BPMN) used to generate event logs for the experimental evaluation.

Methodology. Figure 54 depicts the process model generated using PLG2, which comprises eight (8) activities. We generate event logs of process models with different number of executions/traces: 500, 1.000, and 1.500. In addition, we vary the level of noise, which is typically present in real-life event logs. We use no noise, complete noise, noise on the control flow, and noise in the activity names. For all cases, we use ProM to generate a Petri Net from the event log, which corresponds to the discovered process. Then, we evaluate the discovered process based on the metrics mentioned earlier. We use the following notation: {traces,noise}, where traces={500,1.000,1.500} and noise={none,compl,flow,names}. For instance, {500,compl} refers to the experiment with 500 traces and complete noise. Also, notice that in the case that an algorithm did not complete its execution, either due to extremely long-running time or due to limitations of the different plug-ins of ProM, we report a value of zero (which explains the missing bars in the charts).

Results. Figure 55 depicts the obtained results in terms of the fitness metric. The first observation is that in the presence of large quantities of noise, denoted {*,compl}, the fitness is lower. In particular, this is evident for Inductive Miner whose values drop to 0,8 and then to 0,2. Instead, Heuristics Miner is not significantly affected. Regarding the other types of noise, we observe that fitness is negatively affected mostly in the case of noise in activity names.

[35] Joos C. A. M. Buijs, Boudewijn F. van Dongen, Wil M. P. van der Aalst: On the Role of Fitness, Precision, Generalization and Simplicity in Process Discovery. OTM Conferences (1) 2012: 305-322

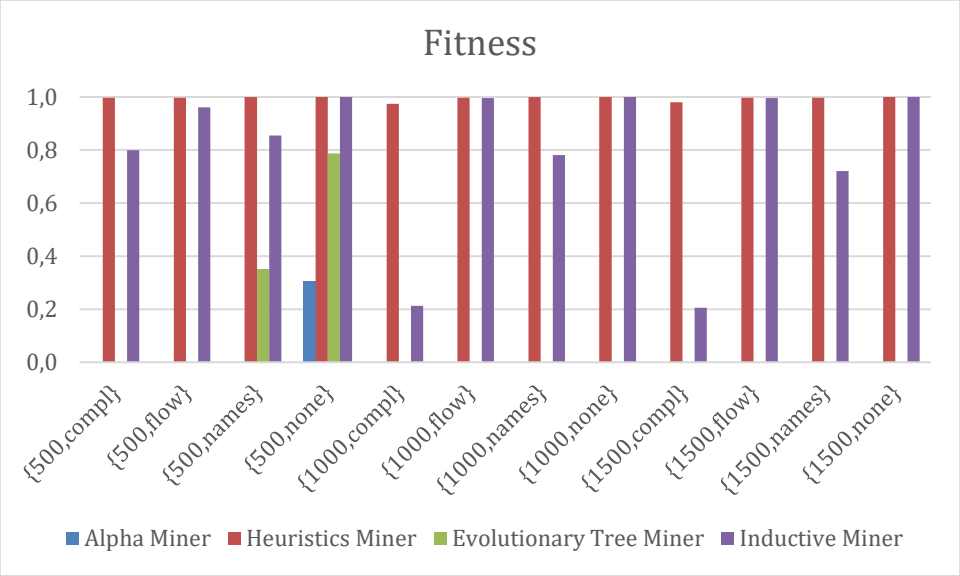


Figure 55: Evaluation results: Fitness

Figure 56 depicts the results of the same experiment, but for the generalization metric. In this case, we observe that the Evolutionary Tree Miner achieves the best performance in terms of generalization. This means that the discovered process model is better able to cope with unseen traces. Also, Inductive Miner performs quite well in the different setups. In the case of complete noise, all algorithms are affected negatively, but probably Alpha Miner is the one most affected by noise.

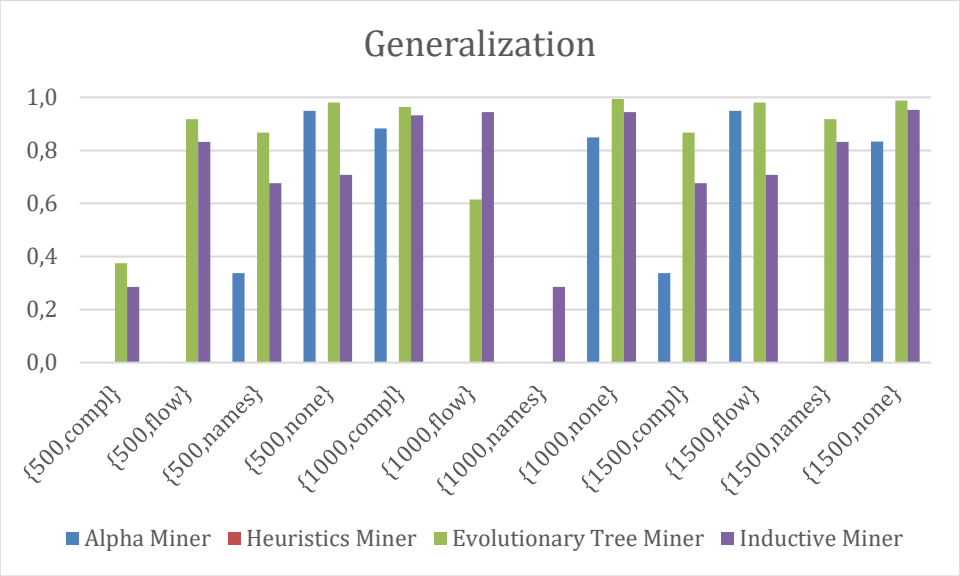


Figure 56: Evaluation results: Generalization

State of the art shows the results for the precision metric. In this experiment, the best performing algorithms are the Inductive Miner and the Evolutionary Tree Miner. Clearly, the different types of noise have a more significant effect on precision, because all algorithms have difficulties in discovering accurate process models from a noisy event log.

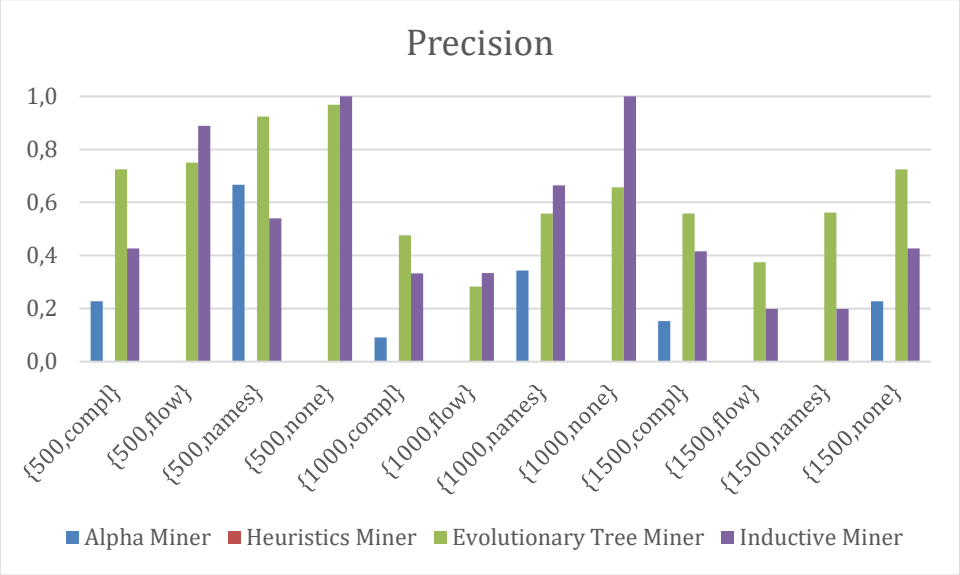


Figure 57: Evaluation results: Precision

In summary, we conclude that different types of noise may have a significant effect on process discovery algorithms, and this may result in process models of poor quality. In particular, we visualized the obtained process models and we observed that when the noise level was very high, the discovered processes were “spaghetti-like”, indicating the limitations of state-of-the-art algorithms.

9.5 Conclusions

The Process Analytics component is applicable on event logs that have been produced by any application executed in BigDataStack, regardless of the actual use-case scenario. However, it should be clarified that its use is meaningful only when many applications consists of steps that are being re-used in different executions. This is the main use-case scenario targeted by the Process Analytics component. The code of this component is not currently publicly available; this is subject to change in the near future, after our research will have been published.

10 Log Search

The Log Search component of BigDataStack is a containerized service that **falls under Task 4.5 (Predictive & Process Analytics)** and developed in Y3, but is separate from the main process mining work-flow. In particular, the goal of the log search component is to provide the application engineer real-time tools to quickly explore the compute logs produced by their application components.

10.1 Motivation and Technology Gap

During application development and maintenance of software applications, it is critical that information regarding failures are easily accessible to the responsible engineer. This information typically can come in two forms. First, as notifications of failures, which in BigDataStack are exposed as Events by the Realization Engine. However, notifications by their nature will only specify *what* failure has happened, not *why* that failure has occurred. Hence, the engineer will need to explore the log data generated by the failing application to understand exactly where the failure occurred and why. Currently, if using OpenShift alone, then the engineer can use the OpenShift console to view the logs produced by a container. This is simply provided as a complete dump of all logs produced in a single window. However, this can be completely inadequate depending on the properties of the application and its previous lifetime prior to the failure. First, once the volume of log data exceeds around 200 to 500 lines, reading the log data becomes cumbersome and it is easy to miss key information. Indeed, 'chatty' application components can generate 10's of thousands of log lines in only a few hours (e.g. a user-facing application generates a log entry on each request). Furthermore, there is a second issue with how OpenShift exposes log data. When a container within a pod fails, the default action for most deployments is to kill the surrounding pod and restart it. When this occurs, the logs from the failed container become un-retrievable from the OpenShift console, as only the most recent Pod is shown. Hence, there is a need for a better solution to enable engineers to interrogate the log data produced by their application components.

Building off the success of modern search engines as highly effective information access tools for textual data, the Log Search service provides a search engine for use with application log data that integrates with the Realization Engine (see D3.3 Section 6).

10.2 Requirements Specification

In this section we summarize the requirements identified for a log search system within BigDataStack.

Table 49 - Requirement REQ-LS-01

	Id	Level of detail	Type	Actor	Priority
	REQ-LS-01	Stakeholder	FUNC	Application Engineer	MUST
Name	Trigger Indexing of Application Logs				
Description	When a new BigDataStack Object representing an application component is started on the cloud/cluster, the application engineer needs a means to mark that application as a target for log indexing. Such marked components will have their log data collected by the Log Search service, where the logs are transformed into inverted index structures ready for access.				
Additional Information	This triggering will be handled within the Realization Engine via a BigDataStack Operation.				

Table 50 - Requirement REQ-LS-02

	Id	Level of detail	Type	Actor	Priority
	REQ-LS-02	Stakeholder	FUNC	Application Engineer	MUST
Name	Log Text Search				
Description	Given a selected BigDataStack Object that has an associated index, the application engineer should be able to enter text queries and retrieve snippets of the application component logs that are relevant to their query.				
Additional Information					

Table 51 - Requirement REQ-LS-03

	Id	Level of detail	Type	Actor	Priority
	REQ-LS-03	Stakeholder	FUNC	Application Engineer	MUST
Name	Grouped Results				
Description	One of the notable aspects of the log search scenario is that often simply retrieving a relevant line in a log file is not sufficient to understand the failure, hence, results retrieved should have customisable snippets that provide contextual logs from before and after each retrieved snippets.				

Additional Information	

Table 52 - Requirement REQ-LS-04

	Id	Level of detail	Type	Actor	Priority
	REQ-LS-04	Stakeholder	FUNC	Application Engineer	DES
Name	Fast Response Time				
Description	The query response time should be fast, in this case results should be returned in under 1 second.				
Additional Information					

10.3 Terrier Information Retrieval Platform

The Log Search service is built on-top of a state-of-the-art open source search platform called Terrier (http://terrier.org) developed and maintained by the GLA partner. In this section we provide an overview of the Terrier search engine and core concepts used by the Log Search service. Terrier at its core is a highly flexible, efficient, and effective open source search engine, readily deployable on large-scale collections of text documents. Terrier's first non-beta release (v1) was 16 years ago in 2004, and since then it has received continuous updates, with the most recent release at the time of writing being v5.3 released in June 2020. Terrier has had over 40,000 downloads since its first open source release and is used by both industry and academia.

10.3.1 Basic Terrier Components

Traditional search within Terrier is based on four main components. The Collection class interprets specific collections and is responsible for determining the discrete unit of a document, i.e. a container log entry in this case. The Document class is responsible for parsing the log entry and extracting the terms within that entry. Terms within that document are extracted by a tokeniser. An Indexer process takes these terms and stores them in an inverted file and other index data structures. An overview of the indexing process within Terrier is illustrated in Figure 58.

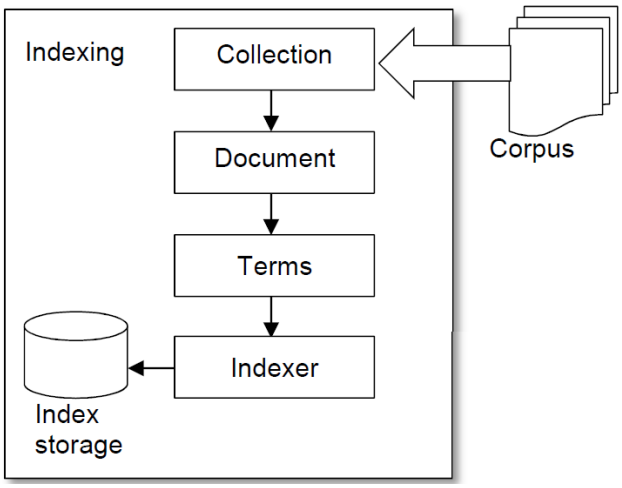


Figure 58: Indexing Process within Terrier

If using a classical index structure, then the index will be comprised of a lexicon and inverted index at minimum. The lexicon (vocabulary or dictionary) contains a list of all the terms in the collection sorted alphabetically (typically along with their frequencies). The inverted index contains a posting list for each document in which that term occurs, with a posting giving the document identifier in which that term appears and how often it occurs in that document (term frequency). Thus, the classical inverted index structure allows the documents that contain a term to be far more quickly retrieved than when using a database reliant on performing string matching over rows in a table.

An indexing service within Terrier facilitates the creation of the aforementioned index structures. It also determines the types of search operations that are subsequently available. During initialization, the user specifies the types of query language functionality that they want supported, in addition to free text search. For instance, the user may specify that they want to be able to perform error-stream-only search over the log entries instead of over all log entries produced. At this point, the user also needs to specify the rules that govern the maintenance of the index structures. For instance, after how long should old entries be deleted (if ever). Once initialized, the indexing service provides an index method that takes a document (log entry) in a format that Terrier understands and adds it to the search index.

Indices produced by Terrier can be of a range of types depending on the needs of the user. However, we can break indices into two main families, namely on-disk indices and memory indices. As their name suggests on-disk indices are designed to be written onto slow physical media (like spinning disks hard drives) and use compression techniques to achieve fast look-up performance. In contrast, memory-based indices are designed to be stored live in memory and are optimized for low latency response times. Indices can be converted from one type to the other. A special index type called a multi-index also exists, which allows indices of different types to be grouped and used as a single index.

10.3.2 Incremental Indexing

Traditional index structures are built from a static set of documents in a single batch operation. However, when processing document streams in real-time such as indexing log entries as they are produced, indexing needs to be performed incrementally. In this case, each new document is indexed as soon as it arrives and is made available for searching immediately. Incremental indexing permits new documents to be appended to an online in-memory index without re-building the index or re-indexing the collection. Postings may be retrieved for the processing of a query as soon as the document has been indexed.

Terrier builds an incremental index that consists of a single in-memory index partition, which is periodically flushed to disk using on-disk index partition(s) as required. The incremental online in-memory index partition and the static read-only on-disk index partitions are wrapped up in a multi index structure to make them appear as one index for retrieval purposes.

10.3.3 Retrieval

The retrieval service within Terrier provides the means to query a target search index to find relevant content. In particular, the user issues a text query (using the supported Terrier query language). The retrieval service then runs the query against the search index, returning documents, ranked by their estimated relevance to that query. Terrier supports two advanced query languages that enable complex matching and subsequent filtering of the indexed documents, one for use by basic users and one for use by programmatic services and advanced users. The Log Search service will have the advanced (Matching Op) query language enabled by default, which supports the following operations:

- **term1** -- scores documents containing this single query term.
- **term1.title** -- scores documents containing this single query term in the title.
- **#band(op1 op2)** -- scores a term containing both operators. The frequency of each matching document is 1.
- **#syn(op1 op2)** -- scores documents containing either op1 or op2. The frequency of each matching document is the sum of the frequencies of the constituent words.
- **#prefix(term1)** -- scores documents containing terms prefixed by term1.
- **#fuzzy(term1)** -- scores documents containing terms that fuzzily match term1.
- **#uw8(op1 op2)** -- the #uwN operator scores documents op1 or op2 within unordered windows of N tokens -- in this case windows of 8 tokens in size.
- **#1(op1 op2)** -- the #1 operator scores documents op1 or op2 appearing adjacently.
- **#band(op1 op2)** -- the #band operator scores documents that contain both op1 and op2.

10.3.4 Terrier-as-a-Service

Since v5.0, Terrier has supported querying of remote index structures through a common manager interface, enabling Terrier indices to be hosted as services that can be used concurrently by multiple applications. In this scenario, a Terrier instance is started in remote

mode, which hosts a REST API end-point that can be used for querying. Both Java and Python client libraries are provided with Terrier, enabling integration with user applications needing these remote search capabilities.

10.4 Log Search Service Architecture

Having summarized the underlying Terrier search engine that is used as the basis for the Log Search service, we next discuss how the service is integrated into the larger BigDataStack platform. The overall architecture of the Log Search service ecosystem is shown in Figure 59. As can be seen from Figure 59, the log search component integrates with both the Realizatione Engine and Kubernetes to achieve its needed functionality. In particular, when the application engineer triggers an operation sequence for launching their application, they can optionally include the BigDataStack specific IndexLogs operation, which in turn passes information about that user application component (in the form of a BigDataStack Object instance) to the Log Search service via its indexing API (triggering log indexing for that application component). Internally, the Log Search service extends the Terrier search engine with a Kubernetes client and container logs formatter. The former enables Terrier to obtain the real-time logs about the target application, while the latter converts those logs into individual documents that Terrier can understand and index. Once a BigDataStack Object is registered via the Indexing API, a new thread is started on the Log Search service to handle the indexing process for that object. This thread first produces a new Terrier incremental index structure, spanning both memory and a connected physical volume for persistent storage. Once the index has been initialized, the thread connects to the container logs end-point of Kubernetes and requests all past logs associated to the target object, as well as listens to the websocket end-point to access new log entries as they are generated. Each log entry is passed through the container logs formatter to convert each ‘line’ in the log into a Terrier Document. These contain the raw (tokensed) text of each log entry as well as metadata about those logs (such as whether they are from the ‘out’ or ‘err’ stream and the log timestamps). These documents are then added to the index (meeting REQ-LS-01).

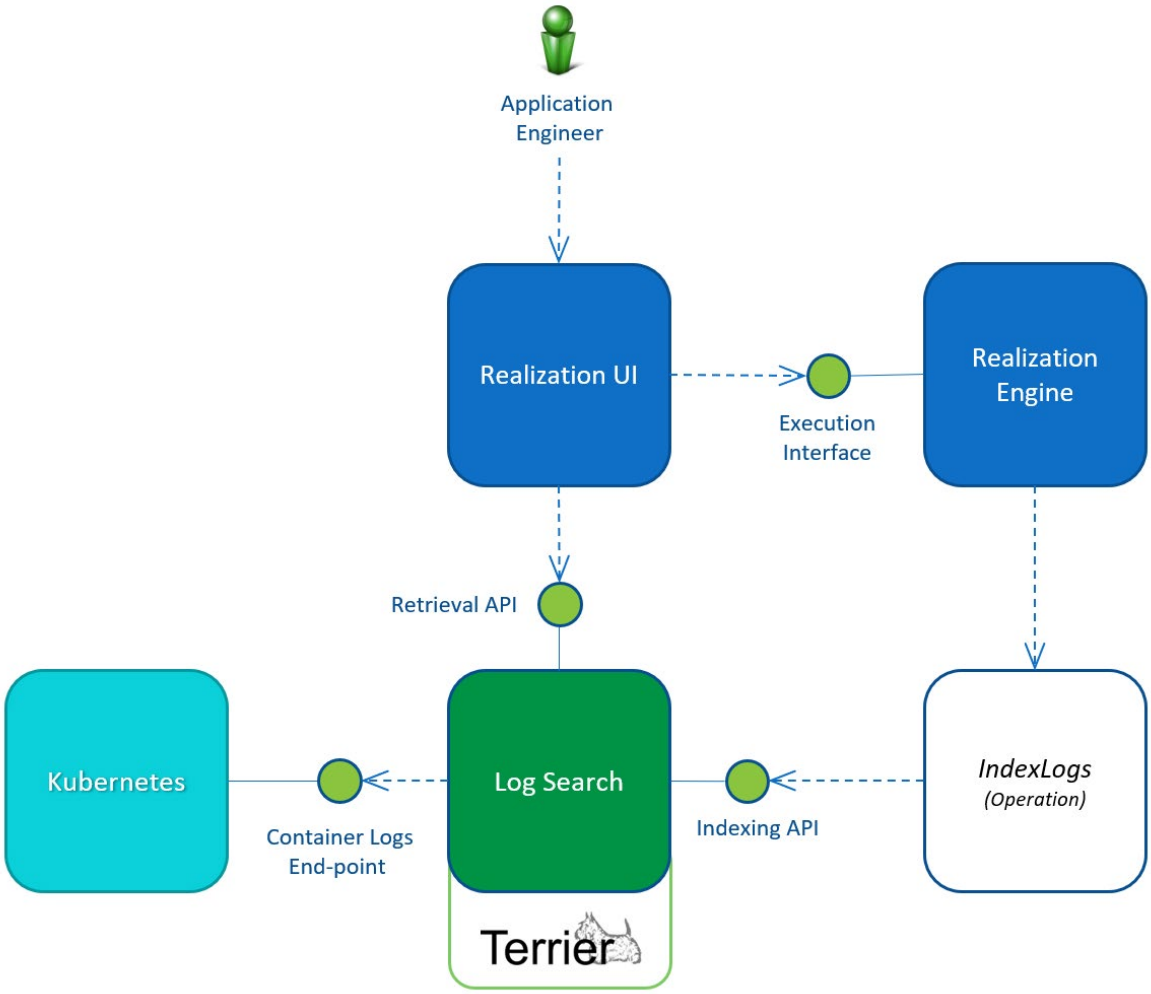


Figure 59: Log Search Service Architecture

For any BigDataStack Object, the Realization UI can query the Log Search service to check whether there is an associated index structure. Within the Realization UI, the Search tab allows the application engineer to select a BigDataStack Object, which performs this check. If an associated index is detected, a search bar and options are rendered, otherwise the user is notified that log indexing was not enabled. Where log indexing had not been triggered, in scenarios where Kubernetes still has the underlying container logs stored, the IndexLogs operation can be triggered post-hoc to produce the index structure.

If an index is available, the user then may enter a search query using the query language summarized earlier (REQ-LS-02), as well as enable the following filters on the search results (REQ-LS-03):

- **Time-range**: The time period within which to return results (based on the log entry timestamps)
- **Stream**: The type of the log entry, i.e. either ‘out’, ‘err’ or both.
- **Context**: The amount of context information that is provided ‘around’ each search result retrieved (specified in a number of lines before or after the result).

10.5 IndexLogs Operation

Following the updated application modelling described in D3.3 Section 6, actions to be performed with respect to user applications are modeled either by atomic operations, or groups of such operations, known as operation sequences. The Log Search component integrates with the Realization Engine to enable the indexing of logs with respect to an application component (BigDataStack Object) via its own operation 'IndexLogs'. This operation takes as input an 'instanceRef' as shown in its configuration below, which is a key referring to a particular BigDataStack Object instance. This key may have been generated by a previous call of the Instantiate operation when deploying the application component, or can be generated via a call of the GetParameterFromObjectLookup for an arbitrary BigDataStack Object.

```
className: IndexLogs # Generate an Index for the named component
instanceRef: "recommendationServiceInstance"
persistIndex: "true"
```

Figure 60: Index Logs Operation Configuration

The IndexLogs operation also takes as input a 'persistIndex' parameter that specifies the type of index that will be generated and how it is internally managed. If set to false, the Log Search service will generate a memory-only index. This will provide the fastest retrieval performance, but if the Log Search service is restarted or its memory becomes full the index may be deleted (all memory-only indices are lost on container exit, meanwhile upon memory exhaustion the oldest index is deleted to free space). If set to true, then an incremental index is produced, which will combine a memory index with one or more on-disk shards. As documents are indexed, they are periodically pushed to disk to free memory and enable persistence across service restarts.

When the IndexLogs operation starts, it performs the following stages:

- **Object Retrieval**: The instanceRef is first used to get the objectID and instance values for the target BigDataStack Object. This object is then retrieved from the Application State Database via the Realization Engine API.
- **Pod Retrieval**: Assuming the BigDataStack Object is found, a second query to the Application State Database is made to retrieve the list of all Pods associated to that object.
- **Registration with the Log Search Service**: The BigDataStack Object and the list of pod identifiers are sent to the Log Search service via its indexing API.

10.6 Log Search Performance

As summarized in REQ-LS-04, the key performance indicator for the Log Search component is search latency, i.e. the amount of time it takes to retrieve results from a search operation. In this section we summarize a series of experiments designed to evaluate the performance of the Log Search service when responding to different types of queries and when searching logs of different sizes.

10.6.1 Experimental Setup

To evaluate the performance of the Log Search service, we require a range of applications with different logging characteristics. To this end, we use the logs of all applications deployed on the GLA OpenShift testbed over a 1-month period for testing. This comprises 682 unique pods deployed on the cluster with an aggregate 32,518,410 lines of logging data. The statistics of the indices produced from these applications is summarized in Table 53. The average and maximum values are calculated across the indices.

Table 53 - Statistics of the Log Search Dataset

Namespaces/Projects	43		
Indices	682		
Value	**Average**	**Maximum**	**Total**
Log Lines/Documents	47,680	7,881,380	32,518,410
Tokens	659,878	70,861,321	450,037,286

For each application, we also require a set of queries for each application to use for testing. As we cannot manually generate realistic queries for the large number of applications tested, we instead generate queries through term sampling from the lexicon for each application index. In particular, we generate three query sets per index as follows:

- **Single Term Queries**: The first set of queries that we generate are single term queries. As query response time is dependent on the length of the posting lists (document frequency) for each query term, we ideally want to test query latency when searching on terms with different posting list lengths. As such, we randomly select one term from each lexicon for each unique document frequency observed in the lexicon. Across all 682 indices, this totals 28,586 single term queries.
- **Two-Term Queries (bigrams)**: The second set of queries that we generate are bigram queries. Here we randomly combine single term queries selected previously into unique pairings. We generate 100 such two-term queries per index, producing 21,920 bigram queries across all 682 indices (not all indices have sufficient terms to form 100 unique combinations).
- **Three-term Queries (trigrams)**: We generate another 100 three term queries per index in the same manner as the two term queries. This produced 22,280 bigram queries across all 682 indices (again not all indices have sufficient terms to form 100 unique combinations).

As the performance of the Log Search service will vary depending upon the type of index (i.e. memory or incremental) we track query response time for the above query sets when querying indices of both types. In the case of the incremental indices (where documents can be spread over both memory and on-disk shards), we transition each index to a state where all documents have been written to disk.

The experimental process is then as follows. A list of all unique pods with logging data is retrieved from Kubernetes. For each unique pod, the Log Search Service connects to the Kubernetes container logs endpoint to retrieve the log data and then indexes it into a memory index. Next the three query sets are generated from the resultant index. Once the queries are ready, a new process is started that sends each query to the Log Search service and the response latency is timed. Once all queries have been completed, the memory index is converted to an incremental index and written to disk. The memory index is closed and the incremental index is loaded. The same query set is then run over the incremental index, with latency timed as before. This process is repeated for all pods on the testbed.

10.6.2 Metrics

During testing we collect metrics regarding both the time to produce the indices and the query response times for the different query sets as follows. Note that query latencies do not contain additional network latencies, i.e. reported latency is only the querying time within the Log Search service, not round-trip time (as that will vary depending on network bandwidth and congestion between the cluster and user).

Indexing Latency:

- **Memory Index Generation Time**: This is the time taken to produce each index structure from start to finish. It includes network latency when retrieving the logging data from Kubernetes and the time to produce the memory index.
- **Conversion and Write Time**: This is the time it took to convert the memory index to an incremental index and write to disk. This time is dominated with the physical write operation, as well as the time taken to optimize the on-disk index once the main write is complete.

Querying Latency:

- **Single Term Query Latency**: Latency for running the single term queries for an index.
- **Bigram Latency**: Latency for running the two-term queries for an index.
- **Trigram Latency**: Latency for running the three-term queries for an index.

10.6.3 Environment

The Log Search engine was deployed on the OpenShift testbed as a container with a limit of 4 CPU-cores (Intel Xeon Gold 6244) and 10 GB of system memory. For writing the container mounted a dynamic volume orchestrated via GlusterFS and comprised of Samsung PM883 SATA Enterprise SSDs. The compute nodes are connected on a via a single 10Gb ethernet switch in a single rack.

10.6.4 Performance Results

Prior to examining query latency, it is first valuable to investigate how long the indexing and write process took using the Log Search service, as if indexing performance is slow then the index may not be up to date when the application engineer issues there queries. Figure 61

reports the time taken to index the logs of the 682 applications contrasted against the size of the final index, where each datapoint is a different application. Both axes are scaled logarithmically. Ideally, we should aim for linear scaling of indexing time with the number of log lines needing indexed and that is what we observe from Figure 61. In particular, for applications with small amounts of logging data (less than 500 lines), memory index generation time is very fast (around 100ms), and the conversion and write times are similarly quick (under 100ms). For applications with more logging data, we see that the time taken transitions to almost exactly linear scaling with number of log lines needing indexed. This indicates that the system should be scalable to applications with very large numbers of log lines. Indeed, the farthest right datapoint in both graphs represents an application with over 7.8 million lines of log data.

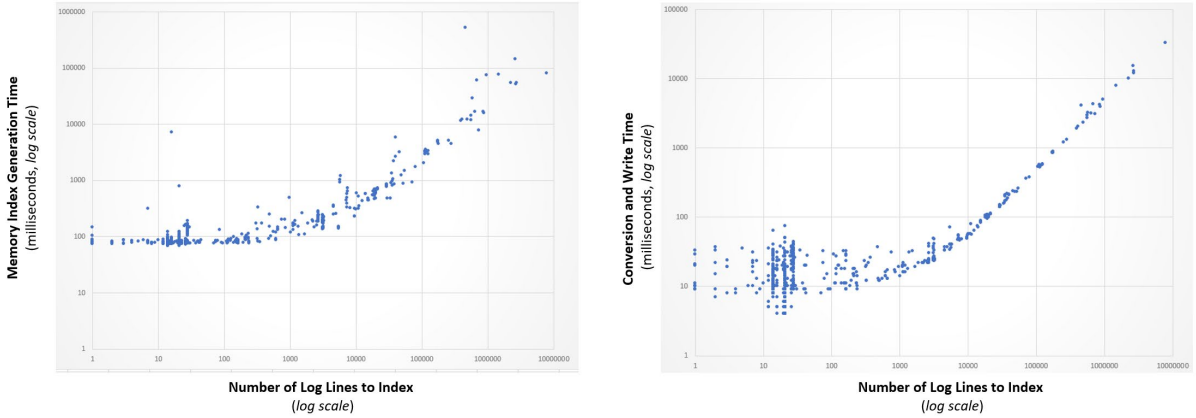


Figure 61: Memory index Generation Time (left) and Conversion and Write Time (right) for the Log Search service

Having shown that indexing is fast, we next need to evaluate the response time when it is queried. As a general rule of thumb, query latency should not exceed 1000ms (1 second), although lower is better (REQ-LS-04) and ideally it should be significantly below that to provide time for the results to get sent over the network to the user. We begin by examining the query latency for the single-term query set comprised of 28,586 queries. As we know the posting list size for each of these queries, in Figure 62 we plot the query latency observed for all queries against the size of the posting list that needed to be traversed to produce the results. The left chart shows latencies when querying the memory-only index, while the right chart reports latency for the equivalent incremental index with all log entries stored on disk. We would like to observe two main outcomes here: 1) we want query latencies to remain below 1000ms and 2) we would want retrieval time to scale linearly (or better) with posting list size. From Figure 62 we can see that for all queries latency remained below the 1000ms target, indeed for the memory index latency never exceeds 300ms, while for the incremental index, latency is higher as expected, but never exceeds 450ms. We also observe that for the vast majority of queries, the response time for the system scales approximately linearly with the length of the posting list.

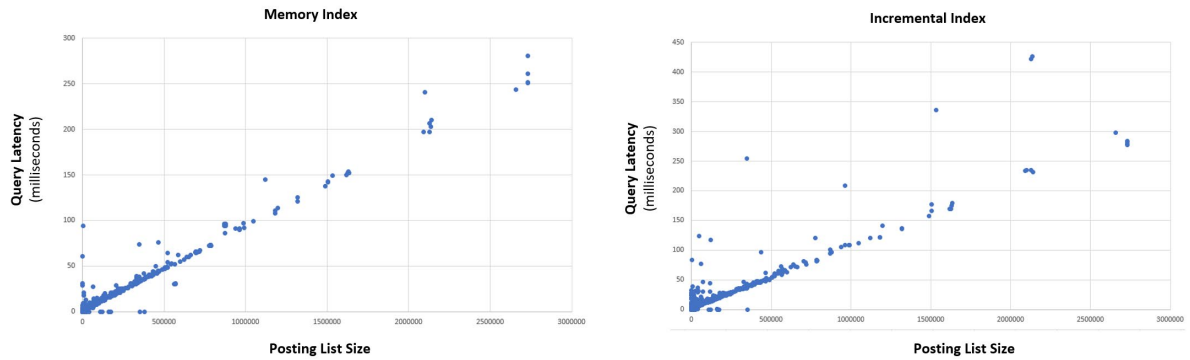


Figure 62: Query Response Latency for the Log Search Service for Single-term Queries

However, most queries that might be made to a log search engine are unlikely to be only a single term in length. Hence, we next examine the latencies observed for the two-term and three-term query sets. Table 54 reports the query response latency for the three different query sets under both the Memory and Incremental indices. As we can see from Table 54, even for these longer queries, maximum latency stays below the 1000ms target, and average latency is also very low, e.g. below 100ms for the Memory index. It is also notable that the bigram query set has a particularly low average latency. This is a result of many of those generated queries not matching any results and hence the search process being ended early.

Table 54 - Query Response Latency for the Log Search Service

Index	Query Set	Maximum Latency	Average Latency
Memory	Single Terms	281ms	2ms
	Bigrams	213ms	1ms
	Trigrams	842ms	94ms
Incremental	Single Terms	426ms	31ms
	Bigrams	250ms	1ms
	Trigrams	845ms	256ms

10.7 Summary

In this section we introduced the Log Search service within the BigDataStack environment, which provides real-time access to application log data from within the Realization Engine UI. This component enables application engineers to more quickly identify the cause of failures in their applications by reducing the time it takes to find relevant information from application log data. We also performed an efficiency evaluation of the platform over 680 applications and 35 million lines of log data, showing that the service achieves sub-second response time over all 70,000 queries tested, meeting the target KPI.

11 Real-time Complex Event Processing

The real-time Complex Event Processing (CEP) deals with the processing of events as they are produced (before they are stored). CEP processes each event independently (e.g., filtering those that do not match a predicate, adding more information to events) of grouping them on windows (e.g., calculate average speed in the last hour). The BigDataStack CEP runs on a single node or on top of a distributed system. CEP is able to run distributed and parallel queries over data streams. CEP can be deployed at the edge to run the queries as close as possible to the data. For instance, it can run at vessels to detect anomalous conditions and accelerate decisions before sending the information to a central on shore office.

During the second part of the project a performance evaluation using small devices like Raspberry Pi, and desktops (Intel i7 computer) has been conducted. A protocol for migration of queries and subqueries was developed and a performance evaluation was conducted.

During the final phase of the project, the CEP has been enriched with a dynamic elasticity. The purpose of the dynamic elasticity is to improve the system performance by distributing the queries across nodes and therefore, the load across the different nodes where the CEP runs with minimal disruption on the query execution. That is, without stopping the system. Dynamic elasticity is also used for realizing resources when they are not needed due to a low load (number of events sent to the CEP). These mechanisms have been integrated with the Infrastructure building block of BigDataStack (WP3) presented in sub-section 6.1.1.

11.1 Requirements Specification

Table 55 - Requirement REQ-CEP-01 for CEP

	Id	Level of detail	Type	Actor	Priority
	REQ-CEP-01	System	FUNC	Developer	MAN
Name	Manage data from different sources to generate alarms if required.				
Description	The CEP will process data on the fly coming from sensors. Each sensor sends events each minute. CEP will analyze the data according to a set of rules and generate the corresponding alarms.				
Additional Information	The processing will be both stateless and over windows of time and number of events.				

Table 56 - Requirement REQ-CEP-02 for CEP

	Id	Level of detail	Type	Actor	Priority
	REQ-CEP-02	System	FUNC	Developer	MAN

Name	Send alarms and data from each node of the distributed environment to the data center.
Description	Once metrics have been analyzed, the CEP will send the alarms and the data to a central location (data center).
Additional Information	The CEP may run on nodes of a geographically distributed environment.

Table 57 - Requirement REQ-CEP-03 for CEP

	Id	**Level of detail**	**Type**	**Actor**	**Priority**
	REQ-CEP-03	System	PERF	Developer	MAN
Name	Data from distributed nodes is aggregated at a central location.				
Description	Further processing over remote data will be done at a central location.				
Additional Information	The CEP processing will scale to tens streams coming from different remote sources.				

Table 58 - Requirement REQ-CEP-04 for CEP

	Id	**Level of detail**	**Type**	**Actor**	**Priority**
	REQ-CEP-04	Stakeholder	PERF	Developer	ENH
Name	Store data on the data store				
Description	The CEP will store the data at the rate is being produced.				
Additional Information	Both CEP and LX will run at the same location.				

Table 59 - Requirement REQ-CEP-05 for CEP

	Id	**Level of detail**	**Type**	**Actor**	**Priority**
	REQ-CEP-05	System	ENV	Developer	ENH
Name	Scale out process				
Description	The CEP deploys another instance of an overloaded (sub)query to distribute the load.				
Additional Information	The new instance can be deployed in the same node or in a different one. Meanwhile the rest of instances that are not involved in the scale out process continue processing events.				

Table 60 - Requirement REQ-CEP-06 for CEP

	Id	Level of detail	Type	Actor	Priority	
	REQ-CEP-06	System	ENV	Developer	ENH	
Name	Scale down process					
Description	The CEP removes one instance of the underloaded (sub)query and distribute the load of the removed instance among the active ones.					
Additional Information	The rest of instances continue processing events.					

11.2 Design

Figure 63 shows the main components of the CEP. BigDataStack CEP is a parallel and distributed data streaming engine. That is, the execution of a single query can be distributed among different nodes in a cluster. Streaming queries can also run in parallel in a single node taking advantage of multicore architectures.

Clients of CEP (Client, Sender App in the figure) connect and submit queries to the CEP using a driver, the JCEPC driver. Applications consuming the results (Client, Receiver App in the figure) of the queries also use the JCEPC driver to receive the events.

The Orchestrator is in charge of managing and monitoring the CEP internal components and deploying queries. The Orchestrator stores meta-information about the system (e.g., query deployment, number of nodes…) in Zookeeper, which is used as a reliable registry. The Instance Managers (IM) are the components in charge of query execution. The initial number of IMs is configured at deployment time. IMs can be added and removed at runtime without stopping the query processing. The Orchestrator automatically partitions queries into subqueries to distribute them among IMs. Each IM can run several subqueries. A node can run several IMs. In general, each IM will be assigned to a core. Clients can define the distribution and parallelism of the queries using the JCEPC driver.

The CEP provides a set of performance metrics (i.e., throughput, latency, CPU and memory usage) that are handled by the metric server. These metrics can be defined for individual operators, subqueries and queries.

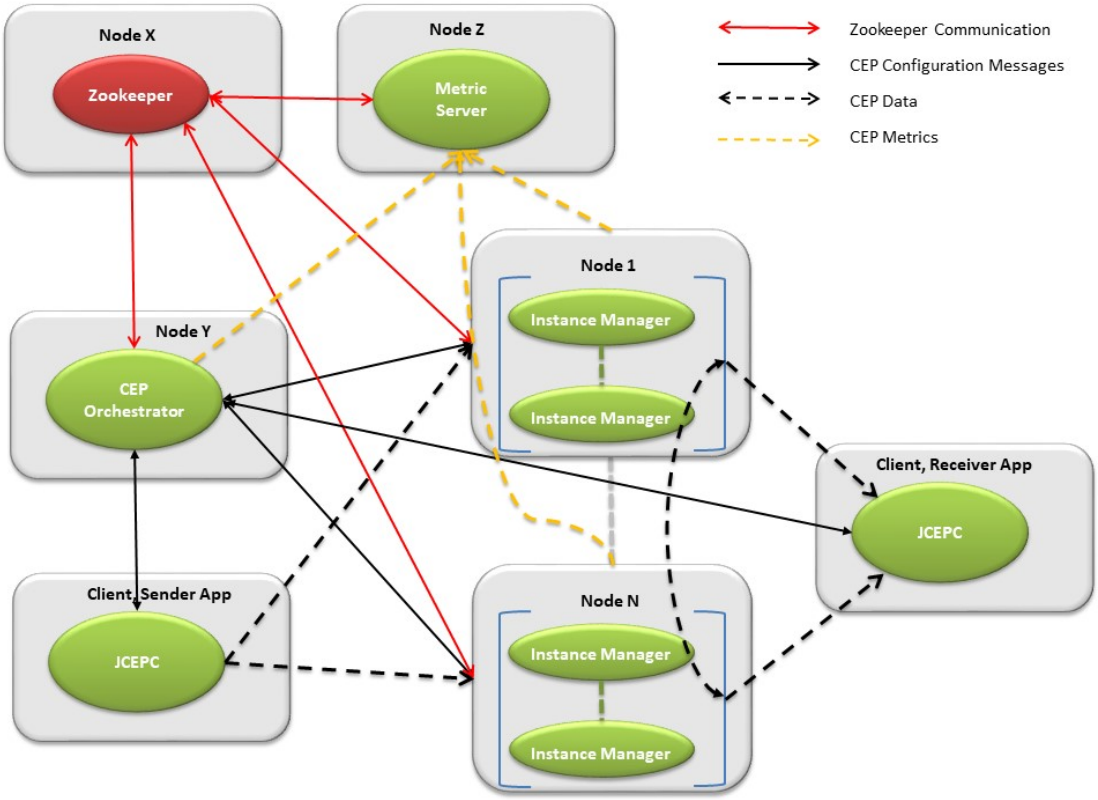


Figure 63: CEP Components

11.3 Final Prototype

At M34 there is a functional prototype that runs distributed and parallel queries in different hardware platforms, namely AMD, Intel Xeon and Raspberry Pi, with different resources in terms of memory and CPUs. Moreover, a dynamic elastic system was implemented in order to adapt to different input loads migrating, scaling-in and out (sub)queries.

11.4 Use Case Mapping

In the scope of WP4, the ship management use case is used as the main demonstrator to highlight the advantages of CEP. The sensors (up to thousand) on the vessels produce data every minute that currently is sent to a central office every three hours to be analyzed off-line. CEP now offers the possibility to process the data as it is produced triggering alarms as soon as possible. This use case requires both stateless queries (e.g., checking the current value of a sensor) and stateful queries (e.g., triggering an alarm is the average fuel consumption during the last half an hour was higher than a given value). The data is also sent to the central office for further analysis and correlation with historical data, including basic data cleaning. This data cleaning is done for each individual record/event. For instance, replacing null values, changing the format of date and time.

11.5 Experimental Evaluation

The performance evaluation for the CEP component will consider the following scenarios:

- Live scale down and out of queries without stopping the query processing.
- Wide area deployment of queries with bandwidth and load changes.

The benchmarking of the CEP will use the HiBench[36] benchmark.

11.6 Scale up / out process

The scale up / out process allows the CEP to increase the (sub)queries instances in order to be able to process all incoming data in the same node or in a different one, respectively. This process can be executed dynamically from the orchestrator or manually with the client interface. When the orchestrator detects that the CPU usage is higher than 75% or the amount of free memory is lower than 500 MB in an IM, the scale out process is executed.

If the subquery that is running in the overloaded IM is a stateless subquery the scale up / out process is easier and faster in comparison with a stateful subquery since no state transfer is involved. Figure 64 shows the different steps involved in the scale up process with an example where subquery1-1 is running at IM2 and is scaled-up producing subquery1-2 deployed at IM4. Initially, the deployment consists of 3 IMs (IM1, IM2 and IM3). all of them deployed at node 2. This node has enough resources to host more IMs. Node 1 runs the orchestrator, metric server and zookeeper services, in addition to the clients (client 1 and client 2) that inject and receive data to/from the CEP, respectively. The query is divided in one stateful subquery that is running at IM2. Data source and Data sink operator are running at IM1 and IM3, respectively.

At some point the load received by this query increases and the IM2 is overloaded due to the CPU usage has increased up to 75% or the available memory is lower than 500 MB. The orchestrator detects that IM2 is overloaded and starts the scale up / out process (step 1). First of all, a new instance of the subquery is deployed at IM4 (subquery1-2) and the output stream of this new subquery instance is configured to send the output data to the next subquery, in this case a data sink operator (steps 2 and 3). At this point, the state of the subquery1 up stream balancer that is running at the operator of the previous subquery that sends the tuples to this subquery (in this example a data source operator) is set to reconfiguration mode. This means, that the data source stops sending tuples to subquery1-1 and starts storing the tuples into an internal buffer. Next, half of the state stored by the stateful operator of the subquery1-1 is transferred to new subquery1-2, once this process finishes, the tuples and windows that have been transferred are removed from subquery1-1 (step 5). Finally, the subquery1 up stream balancer is reconfigured to the normal state, first

[36] https://github.com/Intel-bigdata/HiBench/blob/master/docs/run-streamingbench.md

sending the buffer stored tuples and at the end continuing with the normal data flow (step 6). In case of a stateless subquery, the scale process is the same but skipping step 5.

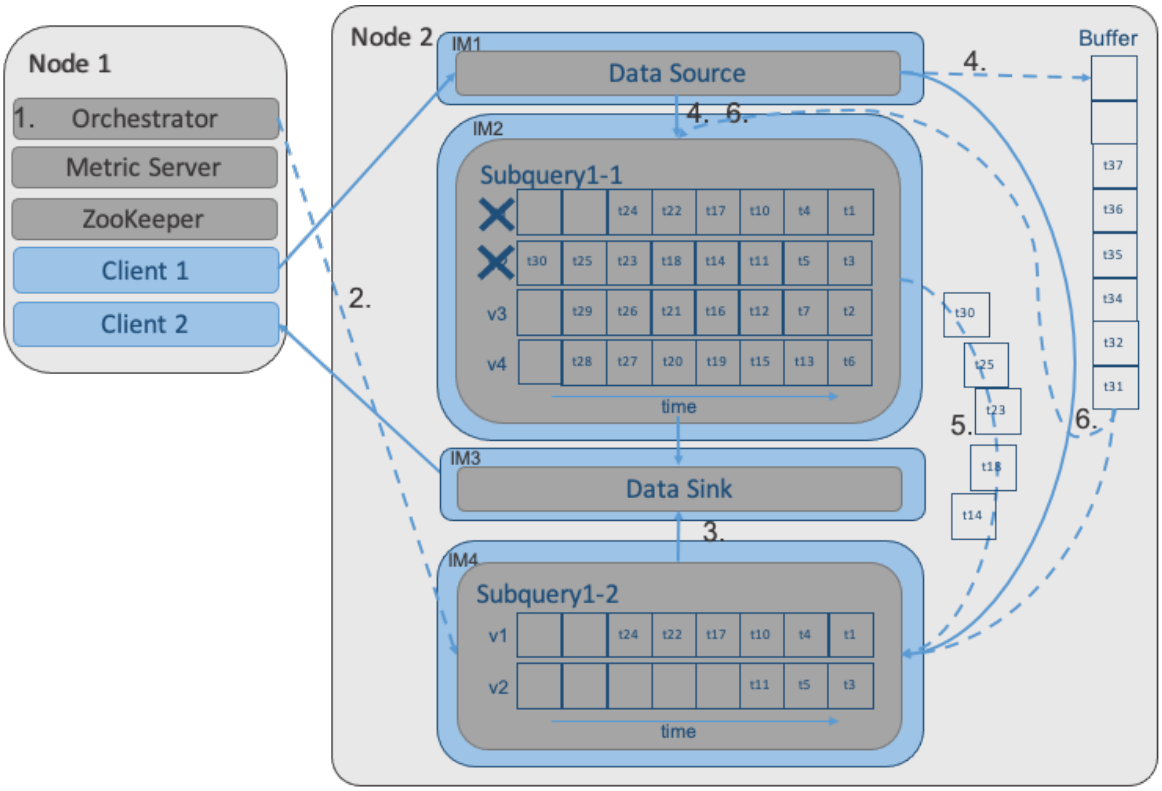


Figure 64: Scale up process

11.7 Scale down process

The CEP scale down process allows to release resources used by the system as long as it is able to process the injected load. The scale-down process also distinguishes between stateless or stateful queries. Figure 65 shows the different steps that take place during the scale down process. In this example, the query is distributed in a stateful (sub)query and there are to instances of the subquery1 deployed at IM2 and IM3 processing each one half of the input load. The goal is to remove subquery1-2, transferring the state stored by this instance to the subquery1-1. First of all, the orchestrator detects that the incoming load can be processed with only one instance of the subquery since the load of both IMs is very low (CPU lower than 25% and free memory 80% of the memory assigned to the IM) (step 1). Next, the state of the subquery1 up stream balancer that is running at the data source operator is set to reconfiguration. The balancer stops sending events to the subqueries and starts storing the tuples in a buffer (step 2). At the same time, subquery1-2 sends the state to the rest of instances of this subquery, in this case to subquery1-1 (step3). Once this process has finished the upstream balancer is reconfigured and starts sending the stored tuples to the remaining subquery instances to continue with the normal data follow (step 4),

ending the process with the removal of subquery2-2 (step 5) and releasing the resources of IM3.

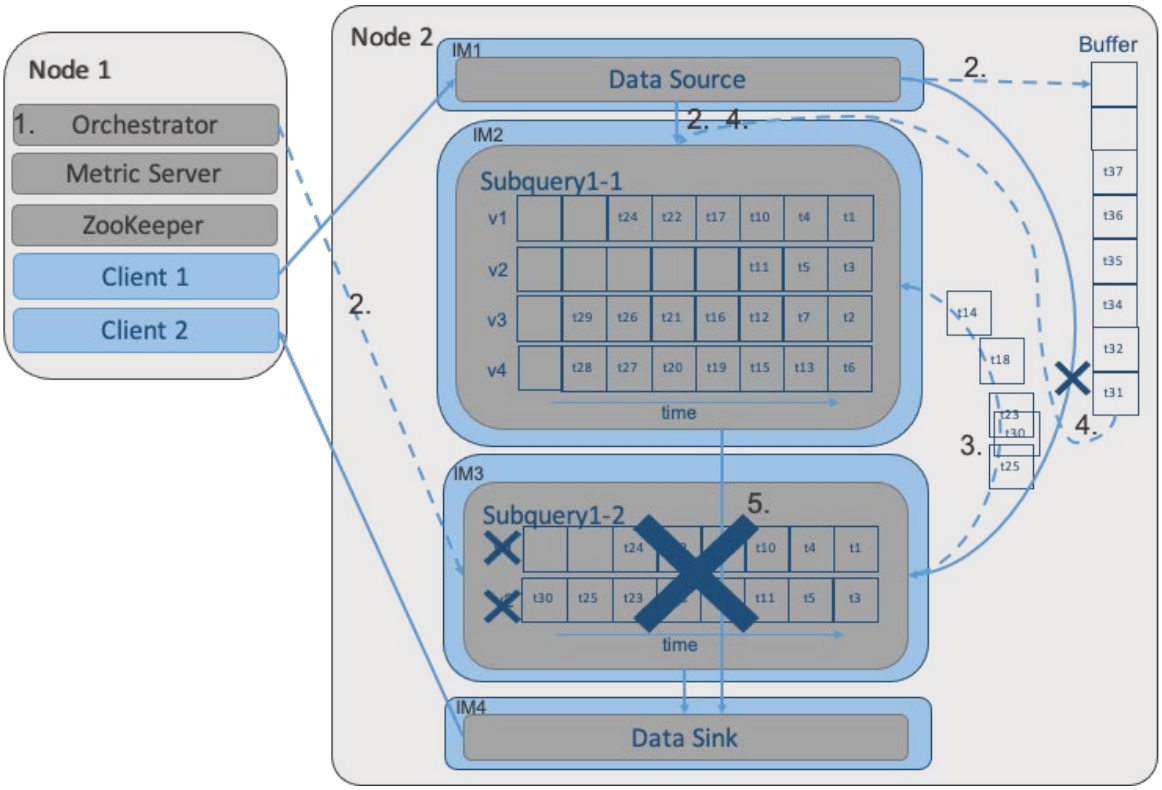


Figure 65: Scale down process

11.8 Performance Evaluation

In this evaluation three aspects of the CEP have been evaluated. The performance of the CEP scale out / in processes was evaluated in a cluster of two AMD nodes adding and removing IM instances in a distributed system using the HiBench query. Each AMD is an Opteron with 64 vcores 6376 @ 2.3GHz, 128GB of RAM, and a direct attached SSD of 480GB, hosted at Universidad Politécnica de Madrid running Ubuntu 16.04.2 LTS. The CEP has been tested in a wide-area deployment using four geo-distributed Google Cloud virtual instances.

The purpose of the first evaluation was to stress the CEP in order to increase CPU and memory usages and force the CEP execute the scale out process adding an extra IM service to distribute de load of the overloaded instance with the new deployed instance without stopping the CEP.

The second evaluation goal is the performance evaluation of the scale-down process when the current load decreases and less resources are needed. During the scaling down process the CEP is not stopped. This initial evaluation has been conducted in AMD computers.

Finally, the deployment in a geo-distributed scenario goal is to present how the CEP configures the deployment of queries according to the available load and bandwidth.

11.8.1 HiBench Benchmark.

The HiBench benchmark defines several streaming benchmarks: Identity, Repartition, Stateful wordcount and Fixwindow. The Identity benchmark reads from and writes to Kafka the data. The Repartition benchmark tests the shuffling capabilities of the streaming framework by changing the level of parallelism. The stateful wordcount counts words every few seconds. The Fixwindow aggregates the connections to a server from an IP address every five seconds and outputs for each IP address the number of connections for each IP address within the last five seconds and the time of the first connection. The query is implemented as two subqueries (Figure 66). The first one (SQ1) consists of a map operator while the second one (SQ2) has one aggregate followed by a map. We have considered 65,025 IP addresses in the performance evaluation. The throughput of the query depends on the number of IP addresses. Given that the size of the window is 5 seconds, if after five seconds a tuple for each IP address has been received, the aggregate operator will output 65,025 tuples. Since the window size is 5 seconds, the maximum throughput will be 65,025 tuples/second, if at least one tuple for each IP address was received. We use as metrics the percentage of the dynamic rate arrival of the records to be processed relative to the current maximum load that can be handled by the current configuration. This is measured as the throughput of SQ1 (the map throughput is equal to the received load) and the uniformly distributing the input IPs so that, if 13, 000 tuples are sent per second after five seconds all possible different IPs (65,000) have been generated and therefore, the maximum throughput (tuples per second) will be 65,000/5 tuples/second.

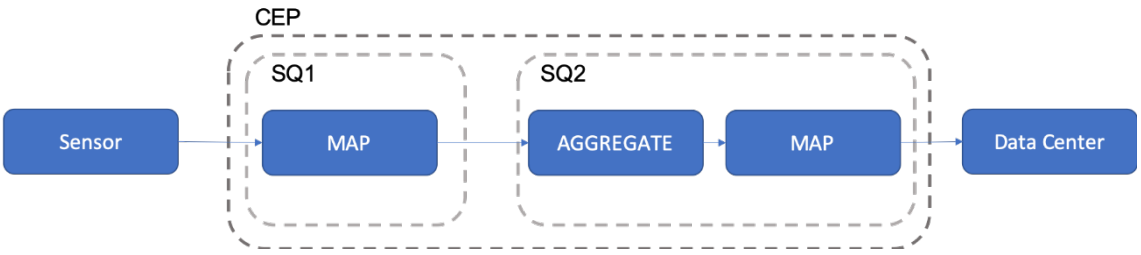


Figure 66: HiBench query and subqueries

11.8.2 Performance Evaluation of the CEP scale up / out process

During the performance evaluation subquery2 of query FixWindow will be scaled up. Figure 67 shows the deployment of the query. There is one instance of each subquery (subquery1 and subquery2) running at IM3 and IM4, respectively. The data source and data sink operators are running at IM1 and IM2, respectively. This set-up is not able to process the injected load (130,000 tuples/second) and the orchestrator decides to scale up subquery2. A new instance is deployed at IM5.

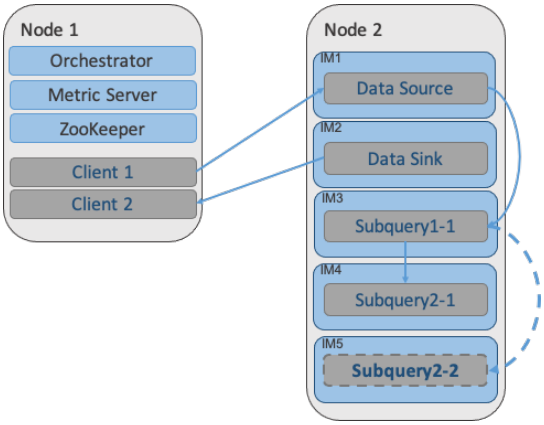


Figure 67: Scale up subquery2 FixWindow query

Figure 68 and Figure 69 show the evolution of the processed load in subquery1 and subquery2 before and after the scale up process. Both graphs have two y axis, right y axis represents the tuples/second for the load (green line) and throughput (yellow line). Left y axis represents the latency (ms) of this subquery (blue line). Initially, the CEP is not able to process the injected load (130,000 tuples/second). During the first 4 minutes of the execution the processed rate fluctuates between 90,000 and 100, 000 tuples/second (green and yellow lines). Subquery1 graph has load and throughput lines overlapped due to the outgoing rate is the same as the incoming rate (between 90,000 and 100, 000 tuples/second). Same load is being processed by subquery2-1 (green line) and the throughput is 65,025 tuples/second (yellow line). Once the scale up process of subquery2-1 takes place at minute 4 in 4.356 seconds the new instance of subquery2 is deployed and processing tuples at IM5. After the new instance of subquery 2 is running (at 16:31 in both figures), the CEP is able to process the injected load ,130, 000 tuples/second. Figure 69 shows that this load is distributed between the new instance of subquery 2 (load IM5 - M1, orange curve) and the old one (load IM4 – M1, green curve).

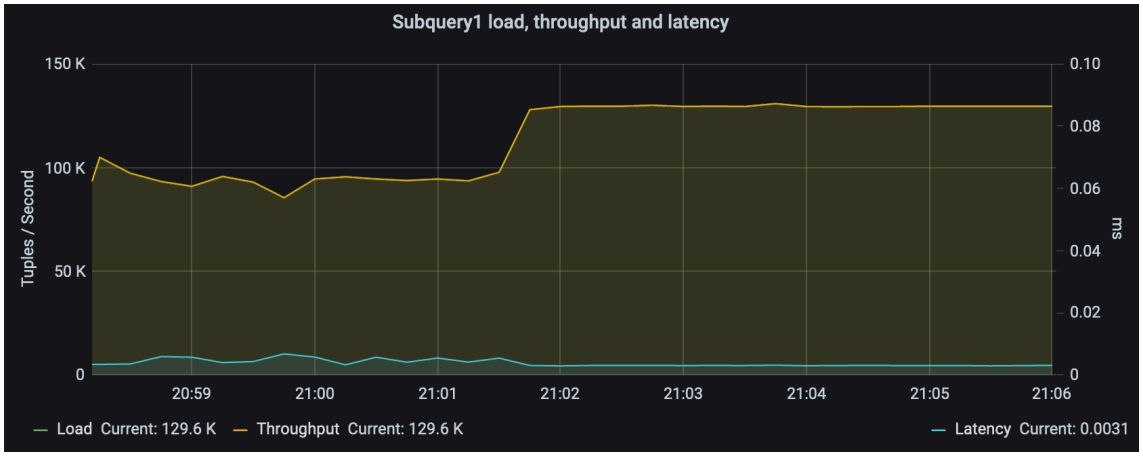


Figure 68: Scale up subquery1

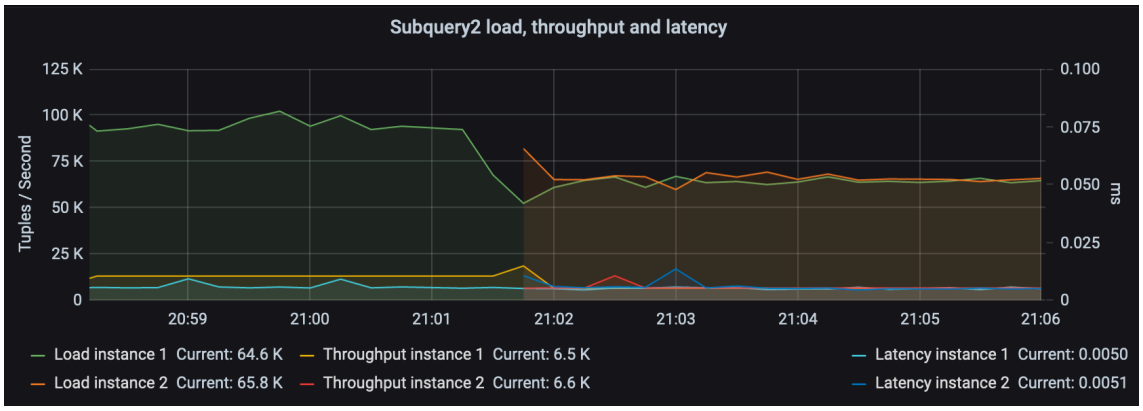


Figure 69: Scale up subquery2

11.8.3 Performance Evaluation of the CEP scale down process

The evaluation of the scale-down process has been conducted using the FixWindow and scaling in the instance 2 of subquery 2. Figure 70 shows how the FixWindow has been deployed during the evaluation. There is one instance of subquery1 and two instances of subquery2 running at IM2, IM3 and IM4, respectively. The data source and data sink operators are running at IM1 and IM5, respectively. The injected load (80,000 tuples/second) can be processed by only one subquery2 instance so, during the scale down process instance 2 of subquery 2 is going to be removed from the deployment.

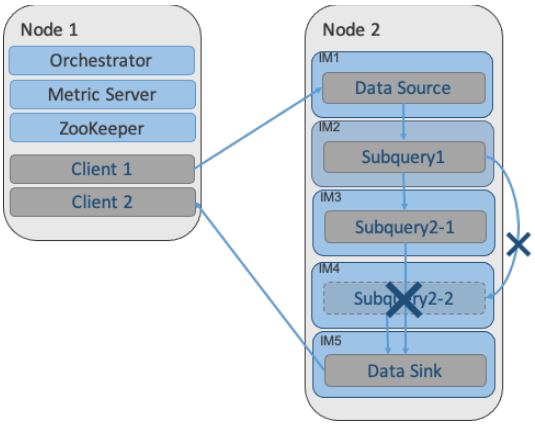


Figure 70: Scale down subquery2 FixWindow HiBench

Figure 71 and Figure 72 show the load and throughput evolution of subquery1 and subquery2 respectively, during the scale down process when the load is 80,000 tuples/second. Figure 71 shows the load (green line) and (yellow line) throughput, both overlapped, of subquery1 and latency (blue line). The processed load remains stable at 80, 000 tuples/second during all execution. This means that the system is able to process the input load.

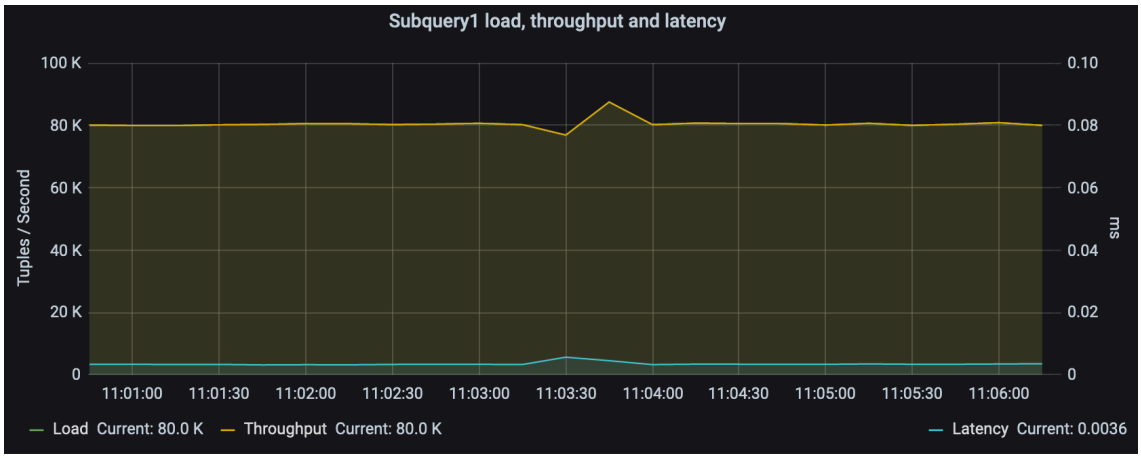


Figure 71: Scale down subquery1

Figure 72 shows the load, throughput and latency of the two instances of subquery 2 . During the first 4 minutes of the execution both subqueries process 40,000 tuples/second. Then, subquery2-2 is removed, and the load of subquery2-1 increases till 80,000 tuples/second. The resources used by that component can be released.

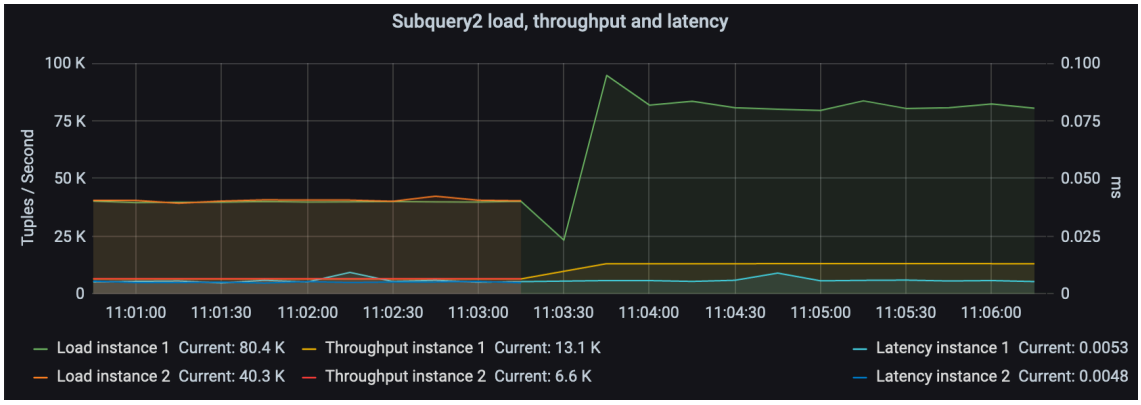


Figure 72: Scale down subquery2

11.8.4 Support for wide-area deployment

The performance of the CEP has tested in a wide area network (WAN) using geo-distributed virtual machines provided by the Google Cloud. The goal of this evaluation is to show how the CEP is able to reconfigure the query deployment due to bandwidth and load changes.

The deployment consists of a datacenter node and three edge nodes. The datacenter node is a general-purpose N1-standard-8 instance with 8 CPU cores and 30GB RAM. It was deployed in the us-central1 region (Datacenter Node). The three edge nodes are general-purpose custom N1 series virtual machines with 2 CPU cores and 4GB RAM. One edge node was deployed in the europe-west1 (Edge Node 1) and the other two nodes at the europe-west2 region within different availability zones (Edge Node 2 and Edge Node 3). All the nodes were running Ubuntu 18.04 LTS and OpenJDK8. Figure 73 shows where the instances are deployed and the available bandwidth.

Figure 73: Wide-area deployment set up overview

The HiBench query was deployed as shown in Figure 74. One instance of subquery1 (SQ1) is deployed at Edge Node 1 collocated with the Data Source operator in one CEP instancemanager. Two instances of subquery2 (SQ21 and SQ22) are deployed at Edge Node 2 and Edge Node 3 in their respective instance managers. Finally, the Orchestrator, the MetricServer, ZooKeeper and the Data Sink operator, running in one IM, are deployed at the Datacenter Node. All instance managers were configured with 1.5GB RAM and all virtual machines were able to host two instance managers.

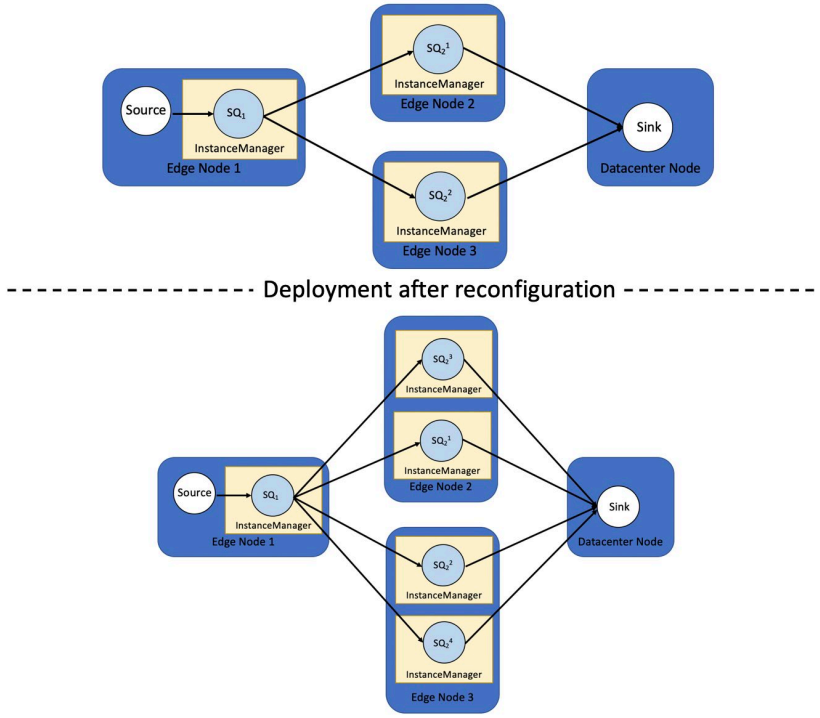


Figure 74: HiBench query deployment before and after reconfiguration

At the beginning of this evaluation, the incoming load rate is set to 20,000 tuples/second and the network bandwidth is set to 10MB. During the evaluation, both the network bandwidth and the incoming rate are modified at some points, producing a subquery reconfiguration. The performance evaluation last 33 minutes. During this time there are four different periods where the bandwidth and the input load rate varies. Period 0 (0 – 100 s) the network bandwidth is set to 15 MB and the incoming rate is 20,000 tuples/second. Period 1 (100 – 400 s) the network bandwidth is set to 9 MB and the incoming rate is increased till 30,000 tuples/second. Period 2 (400 – 1200 s) the network bandwidth is doubled (18 MB) and the input load rate is set to 20,000 tuples /second. Period 3 (1200 - 1700 s), network bandwidth is decreased, and input load rate is increased. Finally, Period 4 (1700 – 2000 s), the network bandwidth fluctuates and returns to 15MB while the input load rate is doubled to 40,000 tuples/second. Figure 75 show the bandwidth and input load rate variations.

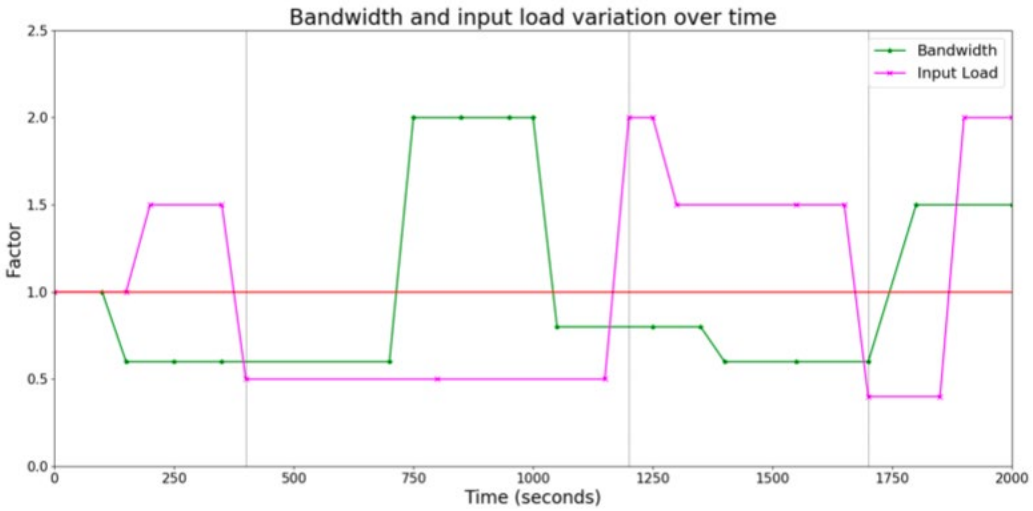


Figure 75: Bandwidth and input load rate variation

11.8.5 Query Latency

Figure 76 shows the average event latency of the HiBench query with elasticity (shown as with adaptability in the Figure) and the static system (baseline in the Figure) in a WAN deployment. During Period 1 both latencies are the same at the beginning since the system is able to process the incoming load of 20,000 tuples/second with a network bandwidth of 15 MB. When the load increases and the bandwidth decreases (Period 1), both configurations are still able to process the load however, the adaptable version is able to adapt the new configuration earlier. During Period 2, first the load decreases and later the bandwidth increases, both configurations can handle the load with the available bandwidth. During Period 3 the incoming load increases and later, the bandwidth is halved. The baseline is almost saturated and the latency increases. The adaptable version is able to reconfigure the deployment when the bandwidth decreases and process the load with a much lower latency than the static configuration. Finally, in the fourth period, first the load decreases and the static deployment is able to process the pending load from the previous period.

Then, the bandwidth increases, and the response time becomes similar in both configurations. Finally, when the load increases, the static deployment becomes saturated and the response time increases.

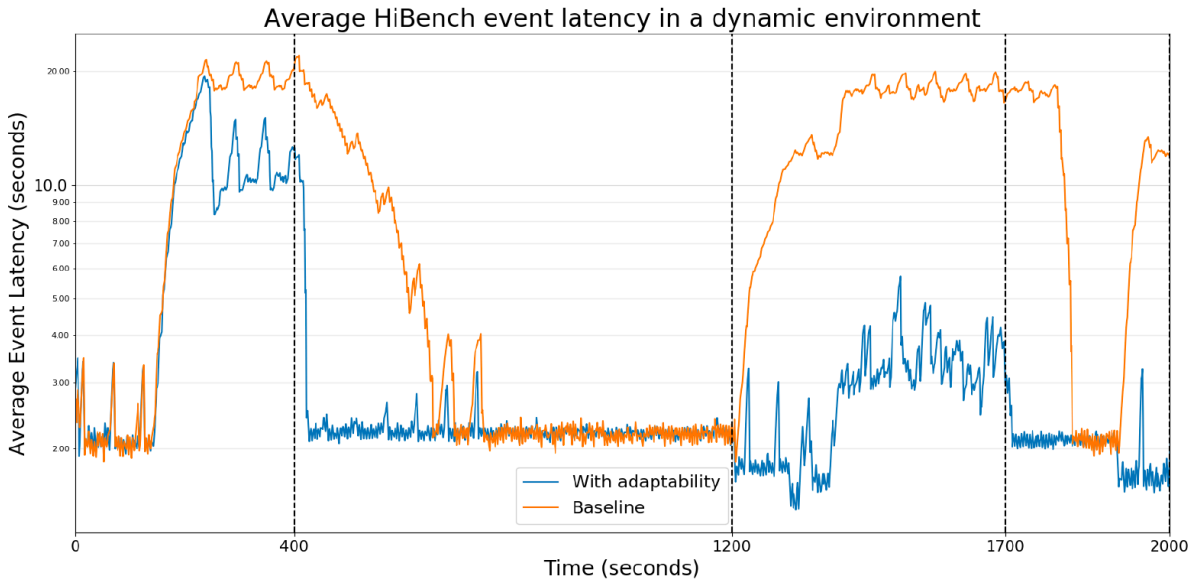


Figure 76: Latency of CEP in WAN system

11.8.6 Deployment reconfiguration

Initially the query distribution supports the incoming load. CPU usage of SQ21 and SQ22 are lower than 10% and the free memory is lower than configured free memory threshold of 750MB. This avoids the scale down process in SQ2 instances. At second 100 of the evaluation, Period 2 stars, the input load is increased to 30,000 tuples/second and the network bandwidth is 9MB. This produces links SQ1-SQ21 and SQ1-SQ22 be saturated, decreasing the query process rate and growing the size of the queue events till its maximum capacity (in this case, 80,000 tuples). Due to this scenario, the CEP decides to reconfigure the subqueries. SQ1 is not reassigned since it should run at the same node as the Data Source operator. One of the SQ2 instances is migrated to the same node where subquery SQ1 is running (Edge Node 1). This reconfiguration, represented in Figure 77, allows the system to process all the incoming load and consume the tuples stored in the queue.

During Period 3 the input load decreased till 20,000 tuples/second and the network bandwidth increased till 18 MB. SQ2 instance managers free memory (edge1_im2 and edge2_im1) are higher than the defined threshold (750 MB). The CPU usage of all instance managers is lower than the threshold of 90% so, no reconfiguration is triggered. Then, Period 4 lasts from second 1200 till 1700. The input load is increased up to 40,000 tuples/second for 60 seconds and later decreased to 30,000 tuples/second. The network rate is decreased to 10 MB and later to 9 MB. During this period, the instance managers free memory starts decreasing, and the system decides to reconfigure de query deployment. Since SQ21 is not able to process all incoming tuples, they are stored and the usage of memory increases. A new IM is deployed at Edge Node 2 and this SQ21 is scaled up

distributing the subquery load to the new instance SQ23. SQ22 is running at Edge Node 1, is also scale up and the deployment is the same as the initial one (before SQ21 reconfiguration). Edge Node 1 cannot host another IM therefore, the elastic manager migrates SQ22 to Edge Node 3. A new IM is deployed in that node and SQ22 is scaled up (SQ24). The deployment is shown in Figure 77. Finally, during Period 4 (second 1700 to 2000) the network bandwidth decreases till 18 MB. At this point the incoming load is increased till 40,000 tuples/second. The current deployment can process the incoming load and no reconfiguration is triggered.

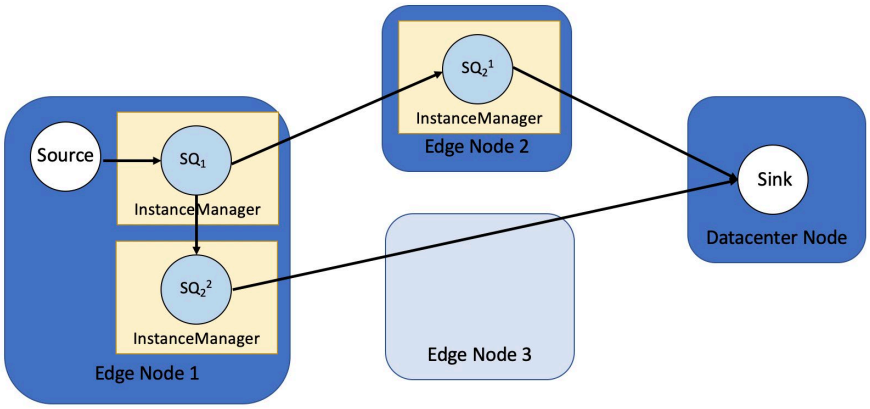


Figure 77: Deployment reconfiguration at Period 2

Figure 78: Instance Manager 1 queue size

11.9 Conclusions

At the third phase of the project, all envisioned functionalities and features of the Real-time Complex Event Processing component proposed have been implemented and delivered. During this phase, the different protocols to scale out/in the different subqueries have been implemented and integrated into a dynamic elastic system. This system allows to the queries running on the CEP component to adapt to the incoming load rate without affecting the performance. The dynamic elastic system decides to migrate, scale out or scale in the

subqueries taking into account CPU, memory and network metrics reported by each instanceManager every 5 seconds. For instance, if an instanceManager is reporting a CPU usage higher than 80% and there is only one subquery running in the instanceManager, the subquery is going scale out in other instanceManager distributing the load among both subqueries. Otherwise, if the subquery contains more than one instance deployed in the system and the CPU usage of the instanceManagers running the subqueries is lower than 30% the subquery is scaled in and one of the instanceManager will be released, saving system resources. The dynamic elastic system has been testing in both a local-area network and a wide-area network.

This component has been entirely developed at Universidad Politécnica de Madrid and is a proprietary software. The last CEP version contains all these new features and has been upstreamed into release 2.2.

12 Bibliography

[1] Amazon Athena pricing. https://aws.amazon.com/athena/pricing/.
[2] Apache Hudi. https://hudi.apache.org/.
[3] Apache Iceberg. https://iceberg.apache.org/.
[4] Apache ORC. https://orc.apache.org.
[5] Apache Parquet. https://parquet.apache.org/.
[6] Databricks Delta Guide. https://docs.databricks.com/delta/optimizations/file-mgmt.html#data-skipping.
[7] Delta Lake (open source version). https://delta.io/.
[8] Elastic Search. https://www.elastic.co.
[9] Geospatial Toolkit functions. https://www.ibm.com/support/knowledgecenter/SSCJDQ/com.ibm.swg.im.dashdb.analytics.doc/doc/geo functions.html.
[10] IBM Analytics Engine. https://www.ibm.com/cloud/analytics-engine.
[11] IBM Cloud Pak for Data. https://www.ibm.com/products/cloud-pak-for-data.
[12] IBM Cloud SQL Query. https://www.ibm.com/cloud/sql-query.
[13] IBM Cloud SQL Query Pricing. https://cloud.ibm.com/catalog/services/sql-query.
[14] IBM Geospatial Toolkit. https://www.ibm.com/support/knowledgecenter/SSCJDQ/com.ibm.swg.im.dashdb.analytics.doc/doc/geo intro.html.
[15] Microsoft Hyperspace. https://github.com/microsoft/hyperspace.
[16] Parquet Modular Encryption. https://github.com/apache/parquet-format/blob/master/Encryption.md.
[17] Test Driving Parquet Encryption. https://medium.com/@tomersolomon/test-driving-parquet-encryption-3d5319f5bc22.
[18] Yauaa: Yet Another UserAgent Analyzer. https://yauaa.basjes.nl.
[19] Stocator - Storage Connector for Apache Spark. https://github.com/CODAIT/stocator, 2019.
[20] A. M. Aly, A. R. Mahmood, M. S. Hassan, W. G. Aref, M. Ouzzani, H. Elmeleegy, and T. Qadah. Aqwa: adaptive query workload aware partitioning of big spatial data. Proceedings of the VLDB Endowment, 8(13):2062–2073, 2015.
[21] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark sql: Relational data processing in spark. In Proceedings of the 2015 ACM SIGMOD international conference on management of data, pages 1383–1394. ACM, 2015.
[22] C. Ballinger. TPC-D: Benchmarking for Decision Support http://people.cs.uchicago.edu/ chliu/doc/benchmark/chapter3.pdf
[23] N. Basjes. Yauaa: Making sense of the user agent string. https://techlab.bol.com/making-sense-user-agent-string.
[24] R. Binna, E. Zangerle, M. Pichl, G. Specht, and V. Leis. Hot: A height optimized trie index for main-memory database systems. In Proceedings of the 2018 International Conference on Management of Data, SIGMOD '18, pages 521–534, New York, NY, USA, 2018. ACM.
[25] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. Communications of the ACM, 1970.
[26] R. Bordawekar, B. Bandyopadhyay, and O. Shmueli. Cognitive database: A step towards endowing relational databases with artificial intelligence capabilities. CoRR, abs/1712.07199, 2017.
[27] B. Braams. Predicate Pushdown in Parquet and Apache Spark. PhD thesis, Universiteit van Amsterdam, 2018.
[28] E. Ch´avez and G. Navarro. An effective clustering algorithm to index high dimensional metric spaces. In Proceedings Seventh International Symposium on String Processing and Information Retrieval. SPIRE 2000, pages 75–86. IEEE, 2000.
[29] E. Chlamt´ac, M. Dinitz, C. Konrad, G. Kortsarz, and G. Rabanca. The densest k-subhypergraph problem. CoRR, abs/1605.04284, 2016.
[30] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In Proceedings of the 23rd International Conference on Very Large Data Bases, VLDB '97, pages 426–435, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[31] M. Y. Eltabakh, F. O¨ zcan, Y. Sismanis, P. J. Haas, H. Pirahesh, and J. Vondrak. Eagle-eyed elephant: Split-oriented indexing in hadoop. In Proceedings of the 16th International Conference on Extending Database Technology, EDBT '13, pages 89–100, New York, NY, USA, 2013. ACM.
[32] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized search trees for database systems. In Proceedings of the 21th International Conference on Very Large Data Bases, VLDB '95, pages 562–573, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.
[33] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht. Occupy the cloud: Distributed computing for the 99%. In Proceedings of the 2017 Symposium on Cloud Computing, pages 445–451, 2017.
[34] S. Kambhampati. Customize Spark for your deployment. https://developer.ibm.com/technologies/analytics/blogs/customize-spark-for-your-deployment/, 2019.
[35] S. Kandula, L. Orr, and S. Chaudhuri. Pushing data-induced predicates through joins in big-data clusters. Proceedings of the VLDB Endowment, 13(3):252–265, 2019.
[36] M. Kornacker. High-performance extensible indexing. In Proceedings of the 25th International Conference on Very Large Data Bases, VLDB '99, pages 699–708, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.
[37] G. Moerkotte. Small materialized aggregates: A light weight index structure for data warehousing. In Proceedings of the 24rd International Conference on Very Large Data Bases, VLDB '98, pages 476–487, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.
[38] S. Nishimura and H. Yokota. Quilts: Multidimensional data partitioning framework based on query-aware and skew-tolerant space-filling curves. In Proceedings of the 2017 ACM International Conference on Management of Data, pages 1525–1537. ACM, 2017.
[39] V. Pandey, A. Kipf, T. Neumann, and A. Kemper. How good are modern spatial analytics systems? Proceedings of the VLDB Endowment, 11(11):1661–1673, 2018.
[40] E. R. Fielding, Ed. J. Reschke. Hypertext transfer protocol (http/1.1): Semantics and content. RFC 7231, RFC Editor, June 2014.
[41] V. Raman et al. DB2 with BLU acceleration: So much more than just a column store. Proceedings of the VLDB Endowment, 6(11):1080–1091, 2013.
[42] A. Shanbhag, A. Jindal, S. Madden, J. Quiane, and A. J. Elmore. A robust partitioning scheme for ad-hoc query workloads. In Proceedings of the 2017 Symposium on Cloud Computing. ACM, 2017.
[43] D. Slezak, J. Wrblewski, V. Eastwood, and P. Synak. Brighthouse: An analytic data warehouse for ad-hoc queries. Proceedings of the VLDB Endowment, 1:1337–1345, 08 2008.
[44] L. Sun, M. J. Franklin, S. Krishnan, and R. S. Xin. Fine-grained partitioning for aggressive data skipping. In Proceedings of the 2014 SIGMOD. ACM, 2014.
[45] A. S. Szalay, J. Gray, G. Fekete, P. Z. Kunszt, P. Kukol, and A. Thakar. Indexing the sphere with the hierarchical triangular mesh, 2007.
[46] G. Vernik, M. Factor, E. K. Kolodner, P. Michiardi, E. Ofer, and F. Pace. Stocator: providing high performance and fault tolerance for apache spark over object storage. In 2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), pages 462–471. IEEE, 2018.
[47] H. Zhang, H. Lim, V. Leis, D. G. Andersen, M. Kaminsky, K. Keeton, and A. Pavlo. Surf: Practical range query filtering with fast succinct tries. In Proceedings of the 2018 International Conference on Management of Data, SIGMOD '18, pages 323–336, New York, NY, USA, 2018. ACM.
[48] M. Ziauddin, A. Witkowski, Y. J. Kim, D. Potapov, J. Lahorani, and M. Krishna. Dimensions based data clustering and zone maps.
Proceedings of the VLDB Endowment, 10(12):1622–1633, 2017.
[49] P. Ta-Shma, G. Khazma, G. Lushi, and O. Feder. Extensible Data Skipping. 2020 IEEE International Conference on Big Data. To be held in December 2020.
[50] IBM Cloud SQL Query Catalog Management documentation https://cloud.ibm.com/docs/sql-query?topic=sql-query-hivemetastore&_ga=2.164493306.47548308.1603632815-546098588.1573663209
[51] Dynamic Partition Pruning in Spark 3.0 https://dzone.com/articles/dynamic-partition-pruning-in-spark-30
[52] Delta Lake https://docs.databricks.com/delta/index.html

13 Appendices

13.1 Appendix A: Data Skipping Formal description and proofs

We point out that negation of an expression e can be handled if we can construct a Clause representing ¬e.

Definition 14. Let c be a Clause that represents an expression e, we say that a Clause c_e is a negation of c with respect to e if $c_e \wr \neg e$
In the worst case, our algorithm will return None, meaning that no skipping can be done.

Algorithm 1: Merge-Clause

input : an expression tree e with root v
output: A Clause C (possibly None)

if e = AND(a,b) **then**
  /* Case 1 */
  Let $\phi := \Lambda_{\gamma \in CS(v)} \gamma$
  Run the algorithm recursively on a and b and denote the result by α, β respectively
  Return $\alpha \Lambda \beta \Lambda \phi$
else if e = OR(a,b) **then**
  /* Case 2 */
  Let $\phi := \Lambda_{\gamma \in CS(v)} \gamma$
  Run the algorithm recursively on a and b and denote the result by α, β respectively
  Return $(\alpha \vee \beta) \Lambda \phi$
else if e = NOT(a) **then**
  /* Case 3 */
  Run the algorithm recursively on a, denote the result by α
  if α can be negated with respect to a **then**
    Return α_a
  else
    Return None
  end
else
  /* Case 4 */
  Return $\Lambda_{\gamma \in CS(v)} \gamma$

Algorithm 2: Generate-Clause

input : a boolean expression e, a sequence of filters $f_1, \ldots, f_n$
output: A Clause (possibly None) c

1 Apply $f_1,\ldots..,f_n$ to e
2 Run Merge-Clause(e) and return the result

13.1.1 Correctness

Given a query Q with ET e, we apply algorithm 2 to achieve a Clause C using the filters defined using our extensible APIs and registered in our system. We show that $C \wr e$. Therefore we can safely skip all objects whose metadata does not satisfy C.

Remark 15. *A good perspective of how extensibility is achieved is by viewing each extensible part's role:* ***metadata types*** *stand for* ***what is the collected metadata, filters*** *stand for* ***how to utilize the available metadata on a given query****, and* ***metadata stores*** *stand for* ***how the metadata is stored****.*

Theorem 16. *Let e denote a boolean expression, and $f_1,\ldots..,f_k$ denote a sequence of filters. Denote by C the output of algorithm 2 on e with $f_1,\ldots..,f_k$. Then $C \wr e$.*

13.1.2 Proof of theorem 16

To prove the theorem, we will use the following lemmas:

Lemma 17. *Let e denote a boolean expression, let c_1, c_2 s.t. $c_1 \wr e \wedge c_2 \wr e$. Then $(c_1 \wedge c_2) \wr e$.*

Proof. Assume the stated assumptions. we will show that $(c_1 \wedge c_2) \wr e$.
By definition: let $o \in U$ s.t. $\exists r \in o$: e(r) = 1. then - since $c_1 \wr e$ we get $c_1(o) = 1$, identically we get $c_2(o)=1$, thus $c_1(o)=1 \wedge c_2(o)=1 \Rightarrow (c_1 \wedge c_2)(o)=1$.

Lemma 18. *Let e_1, e_2 denote a pair of boolean expressions, let c_1, c_2 s.t. $c_1 \wr e_1 \wedge c_2 \wr e_2$. Then $(c_1 \wedge c_2) \wr (e_1 \wedge e_2)$.*

Proof. Assume the stated assumptions and let $o \in U$ s.t. $\exists r \in o$: $(e_1 \wedge e_2)(r)$, we will show that $(c_1 \wedge c_2)(o)$: in particular, $e_1(r)$, which implies $c_1(o)$. identically we get $c_2(o)$, thus $c_1(o) \wedge c_2(o) \Rightarrow (c_1 \wedge c_2)(o)$.

Lemma 19. Let e_1, e_2 denote a pair of boolean expressions, *let c_1, c_2 s.t. $c_1 \wr e_1 \wedge c_2 \wr e_2$*. Then $(c_1 \vee c_2) \wr (e_1 \vee e_2)$.
Proof. Assume the stated assumptions and let $o \in U$ s.t. $\exists r \in o$:$(e_1 \wedge e_2)(r)$, we will show that $(c_1 \wedge c_2)(o)$: in particular, if $e_1(r)$ then $c_1(o)$, else we get $e_2(r)$, which implies $c_2(o)$, thus we get $c_1(o) \vee c_2(o) \Rightarrow (c_1 \vee c_2)(o)$

Remark 20. *The above-mentioned lemmas can easily be restated and re-proved for an arbitrary number of expressions, by a simple induction. we omit these parts and from now we will use the lemmas as if stated for an arbitrary number of expressions.*

Lemma 21. *Let e denote a boolean expression, denote by T_e the expression tree rooted at e. Assume the following holds:*

Assumption 22. $\forall v \in T_e \; \forall c \in CS(v) : c \wr v$.
Denote by C the output of Algorithm 1 on e, then C $\wr$ *e.*

Proof. By full induction on d - the depth[37]. We will assume WLOG that all {∨,∧,¬} nodes are of degree ≤ 2.

Case 1: (**Base case**, *d = 0*) In this case, *e* is a single Boolean operator, so case 4 of Algorithm 1 is applied. By our assumption, ∀c ∈ CS(e):c$\wr$e, by lemma 17 , we get $\Lambda_{\gamma \in CS(v)} \gamma \wr e$, and indeed this is the output in this case.

Case 2: (**Induction Step**) Let $d \in \mathbb{N}^+$ and assume the claim holds for all $k \in \{0 \ldots d-1\}$. since *d > 0*, cases 1, 2, 3 of Algorithm 1 are the only options.

Case 2.a: if e = AND(a,b) : in this case, Algorithm 1 is called again on *a,b*, use α, β from Algorithm 1's notation. *a,b* are both expressions of depth strictly smaller than *d*, so by the inductive hypothesis we have $\alpha \wr a$ and $\beta \wr b$; by lemma 18 we get $(\alpha \wedge \beta) \wr (a \wedge b)$. by lemma 17 and from Assumption 22 we get ($\phi = \Lambda_{\gamma \in CS(v)} \gamma$) $\wr e$). Applying lemma 17 again we get $(\alpha \wedge \beta \wedge \phi) \wr e$, and indeed this is the output in this case.

Case 2.b: if e = OR(a,b) : in this case, Algorithm 1 is called again on *a,b*, use α, β from Algorithm 1's notation. *a,b* are both expressions of depth strictly smaller than *d*, so by the inductive hypothesis we have $\alpha \wr a$ and $\beta \wr b$; by lemma 19 we get $(\alpha \vee \beta) \wr$ e by lemma 17 and from Assumption 22 we get ($\phi = \Lambda_{\gamma \in CS(v)} \gamma$) $\wr e$). Applying lemma 17 again we get $((\alpha \vee \beta) \wedge \phi) \wr e$, and indeed this is the output in this case.

Case 2.c: **if** e = NOT(a): in this case, Algorithm 1 is called again on *a*, and the result is denoted as α.
Case 2.d: **if** α **can be negated with respect** to *a*: Algorithm 1 returns α_a , and by definition $\alpha_a \wr \neg a = e$
o :a = e
Case 2.e: if α **CAN NOT be negated with respect to** *a*: *None* is returned, which represents *any* expression.

We are now ready to prove Theorem 16

Proof of Theorem 16. From Algorithm 1's assumptions we know that $f_1, \ldots, f_n$ are filters, thus Assumption 22 holds. Thus, correctness follows from Lemma 21.

[37] in this case the depth is defined as the maximum length (in edges) of a path from the root (Te) to a {∨,∧,¬} node, comprised of {∨,∧,¬} nodes only, so for example the depth of (a + b < 2) ∧ (c < 5) is 1

13.2 Appendix B: Data Skipping Indexing Statistics

Table 61 - Index Statistics

Index Type	Col Size (GB)	Num. Objects	MD Size (MB)	Indexing Time (min)
ValueList	6.73	4000	163.2	9.4
BloomFilter	6.73	4000	67.8	10.0
Hybrid	6.73	4000	40.4	9.7
Prefix(15)	6.73	4000	17.0	10.0
Suffix(15)	6.73	4000	35.5	10.0
Value List	0.39	4000	34.2	8.5
BloomFilter	0.39	4000	38.9	9.4
Hybrid	0.39	4000	34.3	10.0
Formatted[1]	0.72	4000	0.27	15
MinMax	0.56	4000	0.125	0.97
MinMax	12.16	8192	0.38	1.2

13.3 Appendix C: Example Data Skipping Index

The following is a simplified version of the user agent index from sub-section 5.5.6. We registered a UDF in Spark which uses the Yauaa library [18] to extract the user agent name from a user agent string.

```
import nl.basjes.parse.useragent._
object UserAgentUDF {
  // Define the analyzer at the object level
 val analyzer = UserAgentAnalyzer.newBuilder().build()
 val getAgentName = (userAgent: String) => {
   val res = analyzer.parse(userAgent).get(UserAgent.AGENT_NAME)
   res.getValue
}}
```

The UDF registration is done in the main code using

```
spark.udf.register("getAgentName", UserAgentUDF.getAgentName)
```

The user agent index collects a list of distinct agent names, therefore, we reuse the Value List MetaDataType (representing a set of strings) and its Clause as well as the translation for both.

A. Index creation

```
case class UserAgentNameListIndex(column : String) extends Index(Map.empty, column) {
  def collectMetaData(df: DataFrame):
    MetadataType = {
   ValueListMetaData(df.select(getAgentName(col("user_agent"))).distinct().collect().map(_.getString(0)))
}}
```

B. *Query Evaluation*

The following filter identifies the query pattern appearing in sub-section 5.5.6.

```
case class UserAgentNameFilter(col:String) extends BaseMetadataFilter {
  def labelNode(node:LabelledExpressionTree): Option[Clause] = {
node.expr match {
  case EqualTo(udfAgentName: ScalaUDF,v: Literal) if
  isUserAgentUDF(udfAgentName, col) => Some(ValueListClause(col, Array(v.value.toString)))
   case _ => None
}}
```