

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/

D4.2 – WP4 Scientific Report and Prototype Description – Y2

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, Sandra Ebro, Pavlos Kranas (LeanXcale) Luis Tomás Bolívar (RedHat) Marta Patiño, Ainhoa Azqueta (UPM) Christos Doukeridis, Maria Kanakari, Dimitris Pouloupoulos, Giannis Poulakis (UPRC)
Due Date	30.11.2019
Date	1.12.2019
Version	1.0

Dissemination Level

<input checked="" type="checkbox"/>	PU: Public (*on-line platform)
<input type="checkbox"/>	PP: Restricted to other program participants (including the Commission)
<input type="checkbox"/>	RE: Restricted to a group specified by the consortium (including the Commission)
<input type="checkbox"/>	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	10.10.2019	Yosef Moatti	Skeleton
0.2	10.10.2019	Yosef Moatti	Take into consideration Pavlos' review
0.3	11.11.2019	Yosef Moatti	Take into consideration Marta's review
0.4	12.11.2019	Yosef Moatti	Take into consideration Christos' review
0.5	25.11.2019	Yosef Moatti	Includes fixes for almost all Anestis and Chrysostomos' comments
0.6	26.11.2019	Yosef Moatti	Fix of authors list
0.7	28.11.2019	Yosef Moatti	Integrated version of all tasks. Some update still expected for T4.5
0.8	01.12.2019	Yosef Moatti	<ol style="list-style-type: none"> 1. Update of T4.4 goals 2. Update of T4.6 goals 3. Fixes in T4.5 4. Update of figures references following a new figure in T4.5 5. Improve figures and figure references in Data Skipping
1.0	01.12.2019	Yosef Moatti	Final version

Disclaimer

This document contains information that is proprietary to the BigDataStack Consortium. Neither this document nor the information contained herein shall be used, duplicated or communicated by any means to a third party, in whole or parts, except with the prior consent of the BigDataStack Consortium.

Table of Contents

1. Executive Summary	9
2. Introduction	10
2.1. Relation to other deliverables	10
2.2. Document structure	10
3. Solution Architecture	11
3.1. Vision	11
3.2. Platform Roles	13
3.3. Design	13
4. Implementation and Experimentation	15
4.1. Experimental Setting	15
4.2. Implementation Roadmap M19-36	19
5. Big Data Layout and Data Skipping	21
5.1. Introduction	21
5.2. Requirements Specification	21
5.3. Design	25
5.4. Indicators of Skipping Effectiveness	28
5.5. Extensible Data Skipping	29
5.5.1. Extensible Data Skipping APIs	30
5.5.2. Index Creation	30
5.5.3. Query Evaluation	31
5.5.4. Correctness	34
5.6. Prototype	36
5.7. Use Case Mapping	37
5.8. Use Cases and Experimental Evaluation	39
5.8.1. Data Skipping for Geospatial UDFs	40
5.8.1.1. Benefits of centralized Metadata Store	43
5.8.1.2. Skipping effectiveness of different layouts	45
5.9. Next Steps	48
6. Adaptable Distributed Storage	49
6.1. Requirements Specification	49
6.2. User Story	53
6.3. User Perspective	54
6.4. Detailed Design	55
6.5. Prototype	57
6.6. Experimentation Results	60
6.7. Next Steps	63
7. Seamless Data Analytics Framework	65
7.1. Requirements Specifications	65
7.2. User Story	70
7.3. User Perspective	70
7.4. Detailed Design	72

7.5.	Prototype	81
7.6.	Use Case Mapping	82
7.7.	Experimental Results	82
7.8.	Next Steps	83
8.	Data Quality Assessment & Improvement	85
8.1.	Requirements Specification	85
8.2.	Design	86
8.2.1.	Approach	86
8.2.2.	A simple example of capturing relational information	87
8.2.3.	Data Quality Assessment	89
8.3.	Prototype	91
8.4.	Use Case Mapping	92
8.5.	Experimental Plan	93
8.6.	Next Steps	95
9.	Predictive & Process Analytics	96
9.1.	Requirements Specification	97
9.2.	Design	99
9.3.	Prototype	100
9.4.	Use Case Mapping	101
9.5.	Experimental Evaluation	101
9.6.	Next Steps	103
10.	Real-time Complex Event Processing	104
10.1.	Requirements Specification	104
10.2.	Design	105
10.3.	Current Prototype	106
10.4.	Use Case Mapping	106
10.5.	Experimental Evaluation	107
10.6.	Next Steps	107
10.7.	Initial Performance Evaluation	107
10.7.1.	HiBench Benchmark	107
10.7.2.	Performance Evaluation of the CEP in Small Devices	108
10.7.3.	Performance Evaluation in a Data Center	111
10.7.4.	Subquery Migration	111
10.7.5.	Scalability in a Distributed System.	114

List of tables

Table 1 - BigDataStack Platform roles	13
Table 2 - Implementation Roadmap	20
Table 3 - Requirement REQ-BDL-01 for Big Data Layout	22
Table 4 - Requirement REQ-BDL-02 for Big Data Layout	22
Table 5 - Requirement REQ-BDL-03 for Big Data Layout	23
Table 6 - Requirement REQ-BDL-04 for Big Data Layout	23
Table 7 - Requirement REQ-BDL-05 for Big Data Layout	23
Table 8 - Requirement REQ-BDL-06 for Big Data Layout	24
Table 9 - Requirement REQ-BDL-07 for Big Data Layout	24
Table 10 - Requirement REQ-BDL-08 for Big Data Layout	24
Table 11 - Requirement REQ-BDL-09 for Big Data Layout	25
Table 12 - Requirement REQ-ADS-01 for Adaptable Distributed Storage	50
Table 13 - Requirement REQ-ADS-02 for Adaptable Distributed Storage	50
Table 14 - Requirement REQ-ADS-03 for Adaptable Distributed Storage	50
Table 15 - Requirement REQ-ADS-04 for Adaptable Distributed Storage	51
Table 16 - Requirement REQ-ADS-05 for Adaptable Distributed Storage	51
Table 17 - Requirement REQ-ADS-06 for Adaptable Distributed Storage	52
Table 18 - Requirement REQ-ADS-07 for Adaptable Distributed Storage	52
Table 19 - Requirement REQ-ADS-08 for Adaptable Distributed Storage	53
Table 20 - Requirement REQ-SDAF-01 for Seamless Data Analytics	65
Table 21 - Requirement REQ-SDAF-02 for Seamless Data Analytics	65
Table 22 - Requirement REQ-SDAF-03 for Seamless Data Analytics	66
Table 23 - Requirement REQ-SDAF-04 for Seamless Data Analytics	66
Table 24 - Requirement REQ-SDAF-05 for Seamless Data Analytics	67
Table 25 - Requirement REQ-SDAF-06 for Seamless Data Analytics	68
Table 26 - Requirement REQ-SDAF-07 for Seamless Data Analytics	68
Table 27 - Requirement REQ-SDAF-08 for Seamless Data Analytics	68
Table 24 - Requirement REQ-SDAF-09 for Seamless Data Analytics	69
Table 28 - Requirement REQ-SDAF-10 for Seamless Data Analytics	69
Table 29 - Requirement REQ-DQAI-01 for Data Quality Assessment & Improvement ..	85
Table 30 - Requirement REQ-DQAI-02 for Data Quality Assessment & Improvement ..	85
Table 31 - Requirement REQ-DQAI-03 for Data Quality Assessment & Improvement ..	86
Table 32 - Requirement REQ-DQAI-04 for Data Quality Assessment & Improvement ..	86
Table 33 - Requirement REQ-DQAI-05 for Data Quality Assessment & Improvement ..	86
Table 34 - Requirement REQ-DQAI-06 for Data Quality Assessment & Improvement ..	86
Table 35 - Datasets	93
Table 36 - Results for different values of k	95
Table 37 - Precision, Recall and F1 score for DQA and benchmark solutions	95
Table 38 - Requirement REQ-RD-01 for Predictive & Process Analytics	97
Table 39 - Requirement REQ-RD-02 for Predictive & Process Analytics	98
Table 40 - Requirement REQ-RD-03 for Predictive & Process Analytics	98
Table 41 - Requirement REQ-RD-04 for Predictive & Process Analytics	98
Table 42 - Requirement REQ-CEP-01 for CEP	104
Table 43 - Requirement REQ-CEP-02 for CEP	104
Table 44 - Requirement REQ-CEP-03 for CEP	104
Table 45 - Requirement REQ-CEP-04 for CEP	105

List of figures

Figure 1 - BigDataStack core platform capabilities (extracted from D2.3)	11
Figure 2 - Data as service components mapping to Big Data pipeline	14
Figure 3 - Data ingestion path.....	17
Figure 4 - Data Query Path.....	18
Figure 5 - Process and Predictive Analytics.....	19
Figure 6 - Spark SQL Query Execution Flow	25
Figure 7 - Data Ingestion flow	27
Figure 8 - Data Analytics flow	27
Figure 9 - Index Creation Flow.....	30
Figure 10 - Expression Tree for Example Query	31
Figure 11 - Query evaluation flow	32
Figure 12 - Result of a filter on ET	32
Figure 13 - Algorithm Merge Clause	34
Figure 14 - Algorithm Generate Clause	34
Figure 15 - Geospatial data skipping	41
Figure 16 - Data skipping on UDFs vs no skipping - data scanned.....	42
Figure 17 - Data skipping on UDFs vs no skipping - Run Time.....	43
Figure 18 - Data skipping vs rewrite - Data Scanned.....	44
Figure 19 - Data skipping vs rewrite - Run Time	45
Figure 20 - Indicators of Skipping Effectiveness for geospatial analysis 10km queries..	46
Figure 21 - Indicators of Skipping Effectiveness for geospatial analysis 100km queries	47
Figure 22 - Indicators of Skipping Effectiveness for geospatial analysis mixed queries.	47
Figure 23 - Adaptable Distributed Storage architectural design	56
Figure 24 - LeanXscale administration console	60
Figure 25 - Initial Deployment of the Adaptable Distributed Storage using 1 node	61
Figure 26- Monitoring information showing CPU is fully consumed	62
Figure 27 - Deployment after the manual scale-out action.....	62
Figure 28 - Monitoring information showing CPU load balanced	63
Figure 29 - Federation of the two data stores in the scope of the Seamless Data Analytical Framework	71
Figure 30 - LeanXscale data base to IBM Object Storage pipeline for historical data.....	72
Figure 31 - Interactions among SAF components.....	73
Figure 32 - Data Manager notifies Data Mover to start moving data slices	80
Figure 33 - Data Mover notifies the Manager that the movement succeeded	80
Figure 34 - Family tree.....	87
Figure 35 - Encoding relational information using NN	88
Figure 36 - Cramer's V table.....	89
Figure 37 - DQA network architecture.....	91
Figure 38 - Training phase.....	91
Figure 39 - Assessment phase	92
Figure 40 - Parallelization and distribution of analytics workflow	96
Figure 41 - The recommendation of next possible state during process modelling.....	97
Figure 42 - The interaction between Process Analytics and Global Decision Tracker and Process Modelling Framework	97
Figure 43 - Design of Process Analytics component.....	99
Figure 44 - Graph produced from the analysis of the Online Retail dataset	102
Figure 45 - The same graph colored for better visualization.	102

Figure 46 - The result of the community detection algorithm when applied on the afore-described graph (colors represent different communities).....	102
Figure 47 - CEP Components.....	106
Figure 48 - HiBench query and subqueries.....	108
Figure 49 - CEP Performance for one core and 200 MB memory.....	109
Figure 50 - Two IMs running one subquery in a Raspberry Pi	109
Figure 51 - Performance running a distributed query between Raspberry Pi and i7	110
Figure 52 - Time to process a load	111
Figure 53 - Subquery migration	112
Figure 54 - Migration duration.....	113
Figure 55 - Time for window reconfiguration.....	113
Figure 56 - Number of windows transferred during migration	114
Figure 57 - Number of transferred tuples during migration.....	114
Figure 58 - Scalability in a single node	115
Figure 59 - CEP scale-out	115

Acronyms

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

1. Executive Summary

The BigDataStack project was conceived as a data centric environment, integrating approaches for Data as a Service. The data services of the environment are covered in this deliverable in terms of design specification as well as in terms of integration and experimentation outcomes (i.e. prototypes evaluations) and are naturally at the core of BigDataStack. A full 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 second deliverable for WP4 is the natural evolution of the deliverable D4.1 (M11).

The main differences are:

1. An update of the global integration of the components between themselves, in line with the demonstrations that were held during the interim review.
2. An update of the WP4 component architecture to reflect the implementation level that was reached by this deliverable issuance (and which was mostly demonstrated during the interim review).
3. Initial experimentation results regarding the performance of the architecture components.
4. An update of the data services components architecture to reflect the planned work until the end of the project. This includes an update of the requirements.

2. Introduction

2.1. Relation to other deliverables

This deliverable is related to the following other project deliverables:

- D2.3 - Requirements & State of the Art Analysis III (M22). Collected requirements have been analysed to drive the design specifications of data services¹ (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).
- D2.5 - Conceptual model and Reference architecture II (M18) which is a high-level preview of the more detailed and advanced D4.2 which extends and details the relevant part of D2.5.
- D3.2 (Data-driven infrastructure management scientific report) and D5.2 (Dimensioning Modelling and Interaction services) respectively WP3 and WP5 Scientific Reports and Prototype Descriptions (M23). Alongside D4.2, D3.2 and D5.2 present the current technical status, of the BigDataStack project by M23.
- D6.1 Use case description and implementation (M18).
- D4.1 (M11) the first WP4 technical deliverable. D4.2 is an evolution of D4.1.

2.2. Document structure

The structure of this deliverable is similar to the structure of D4.1:

Section 3 presents the general architecture which 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 used during the interim review for the sake of the Real Time Maritime data management scenario. We enhanced this section to discuss the two other use cases (retail and insurance).

Each of the sections 5 to 10 detail the 6 tasks of the work package. Since there are no real dependencies between the components, the order of these sections is not very significant. One exception is the **seamless data analytics framework** (T4.4 presented at section 7), which is based on the **adaptable distributed storage** (T4.3 presented at section 6) and also takes advantage of the **big data layout and data skipping** (T4.2 presented at section 5), therefore it comes after both.

Two data analysis services are additionally offered in the context of the data services layer, the **data quality assessment** (T4.1 presented at section 8) and the **predictive & process analytics** (T4.5 presented at section **Error! Reference source not found.**).

Finally, section 10 presents T4.6: **real-time complex event processing** (CEP) task.

¹ Since D2.3 was delivered one month ahead of this document, it is possible that the requirements that figure in this document will be slightly different from D2.3

3. Solution Architecture

3.1. Vision

BigDataStack provides a complete data-driven infrastructure, (see Figure 1 **Error! Reference source not found.**). The envisioned BigDataStack platform capabilities could not be realized without a full stack that can facilitate the requirements of Big Data operations and applications.

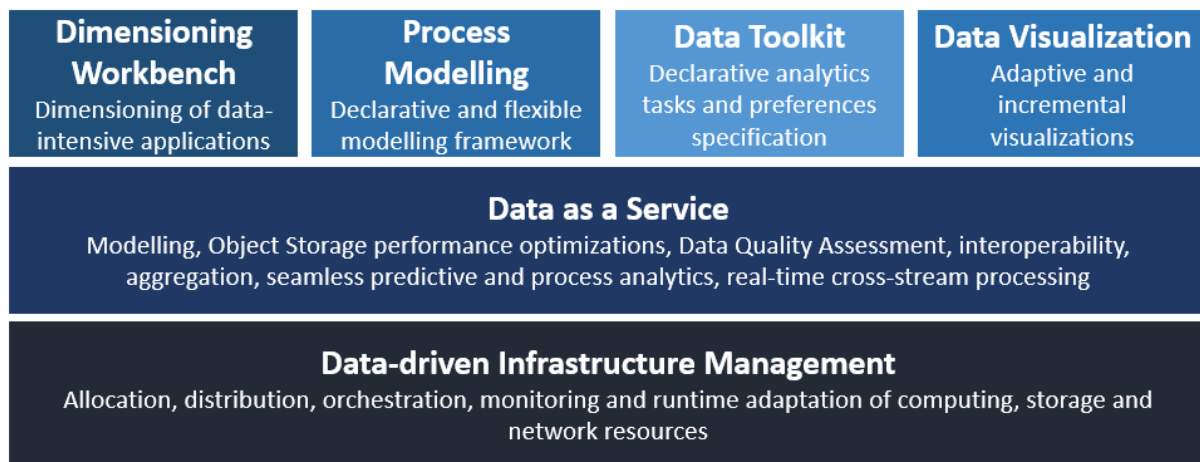


Figure 1 - BigDataStack core platform capabilities (extracted from D2.3)

The "Data as a Service" layer is the cornerstone on which the four upper platform capabilities of BigDataStack rely on. 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 layer along with the respective aforementioned capabilities:

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 this task 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 deliverable D6.1 in section 3.

Big Data Layout and Data Skipping yields novel capabilities for SQL analytics on object storage. It already provides a pluggable data skipping engine with the ability to handle complex queries as well as User Defined Functions (UDF). This technology has already been integrated within the IBM SQL Cloud Service at beta level. These enhanced data skipping

capabilities will be complemented by an online and offline data layout engine which will take into account data properties and query workload in order to enable efficient querying of data stored on object stores by maximizing the data skipping.

Distributed Storage is the first data storage approach of BigDataStack, it is based on the LeanXcale internal key value storage layer. This component for which most of the development was left for the second half of the project will enhance 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.
- 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 Storage²: BigDataStack presents advances both in data layout (or data partitioning) and data skipping. These advances have already been proved to greatly reduce the data ingestion problem. Many additional improvements are on their way and will further ease the deployment of the technology and amplify its benefits.

The Seamless Data Analytics Framework 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 sets. This was demonstrated during the interim review. During the second part of the project, we plan to implement additional improvements that are required to mature this technology, widen its scope and improve its performance.

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 will be utilized to extract knowledge from event logs which in turn will be transformed into insights and recommendations for the user in the Process Modelling phase of the project. In the second part of the project, we plan, under the umbrella of this task to develop a catalogue of analytics: a datastore that will store candidate algorithms that can be selected by the process mapping mechanism when passing from the process modelling to the data toolkit (in the flow).

2

https://www.researchgate.net/publication/317066213_Too_Big_to_Eat_Boosting_Analytics_Data_Ingestion_from_Object_Stores_with_Scoop

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 will be further developed and analysed. In particular optimization of communication to minimize overhead and also enabling of early generation of alarms will be central. The CEP will aggregate data as close as possible to where it is produced speeding up the analytical processing.

3.2. Platform 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: <ul style="list-style-type: none">a) Data Quality Assessment services such as Data Cleaningb) Complex Event Processing, which can be applied on the data streaming in, both before and after it passes through the unified Gateway.c) the possibility to store the data as a single logical data set on both transactional data bases and object store. These data sets can seamlessly be queried.

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 Object Store. Moreover, the seamless analytics framework takes advantage of the adaptable distributed storage in order to allow LeanXscale 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 10): 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.

Typical Big Data processing starts from data acquisition at the edge and then goes through a pipeline as described in **Error! Reference source not found.** 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).

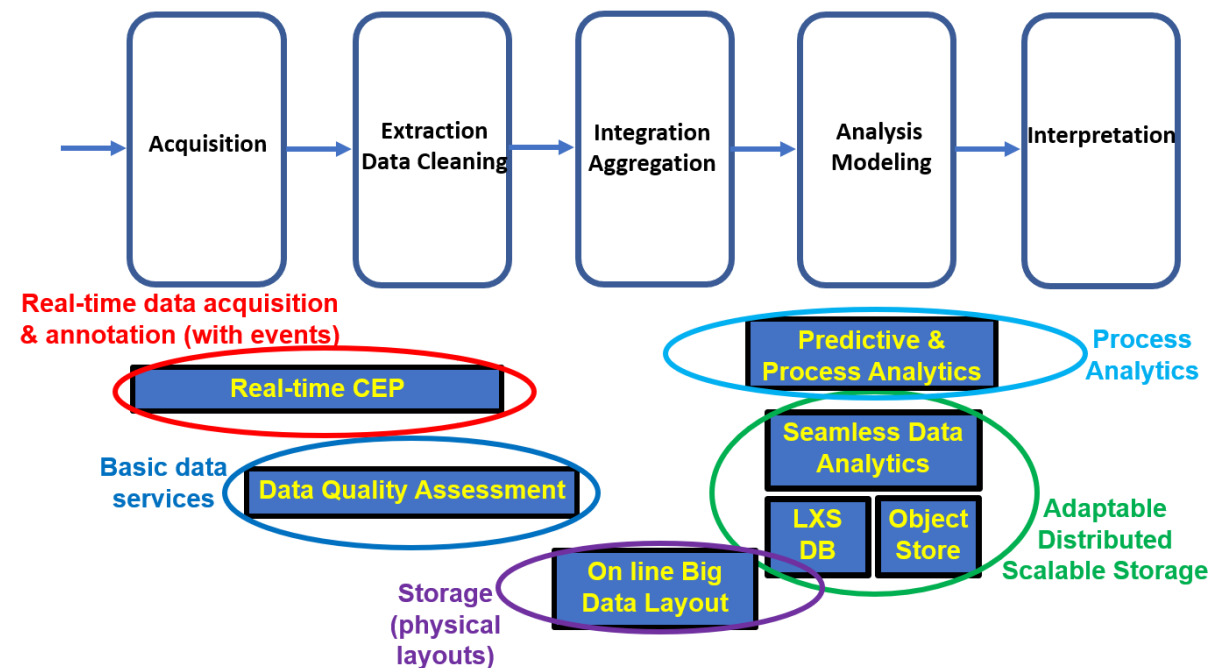


Figure 2 – Data as service components mapping to Big Data pipeline

The CEP (section 10) 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 to be supported in the incremental development of the solution.

Data as a Service is designed to fit a broad scale of Big Data analytics use cases with demanding requirements. During the first half of the project we developed and tested the majority of the components. They were integrated and demonstrated for the ship management use case (see deliverables D6.1 and D2.3 section 4.1). Indeed, the two main scenarios that are built around this use case: the full data ingestion path and then data querying through the seamless component exercised most of the capabilities of the Data as a Service components / mechanisms. Moreover, except for the Machine Learning software which was build for predicting vessel engine failures and which was specific to the use-case, all the components that were used, were used in a standard manner. We will detail this important point for each of the involved components.

As few general notes valid for this section and in fact this deliverable:

1. Our design and solution are agnostic of what exact object store we are using. It could be the IBM COS object store³, as well as OpenStack Swift⁴, or minio⁵, or Ceph⁶, 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.
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 performance⁷

³ <https://www.ibm.com/cloud/object-storage>

⁴ <https://wiki.openstack.org/wiki/Swift>

⁵ <https://www.minio.io/>

⁶ <https://ceph.com/>

⁷ https://cdn2.hubspot.net/hubfs/488249/Asset%20PDFs/Benchmark_BI-on-Hadoop_Performance_Q4_2016.pdf

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 that was handled during the first half of the project: first of all, we will discuss the use case data ingestion path. Figure 3 – Data ingestion path shows all the components that are involved in the ingestion path, which we will detail from the IoT data creation up to its upload in the object store. In the following description “[x]” will refer to the component with label “x”.

During the 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 D6.1 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.

The IoT data events are the input to CEP1 on the ship, which has as 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 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 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].

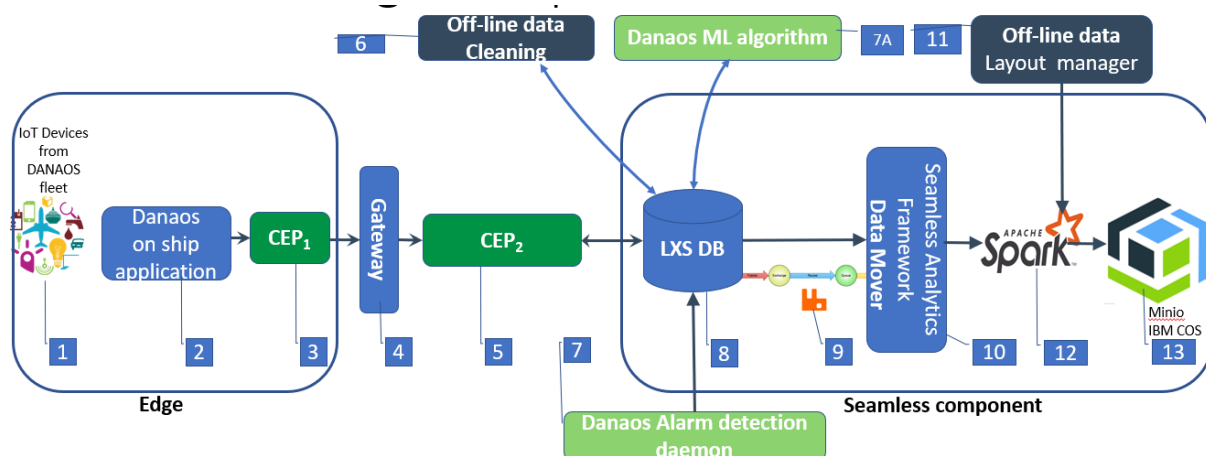


Figure 3 – Data ingestion path

The data query path (depicted in Figure 4 – Data Query Path) is based on the architecture detailed in the Seamless section 0 where the object store may either be remote or on premises. This query path was implemented and demonstrated in the interim review.

⁸ <https://istio.io/>

After data is ingested in the LeanXscale 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 LeanXscale database which stores the fresh IoT data as well as alarm predictions output by the machine learning algorithm [7A].

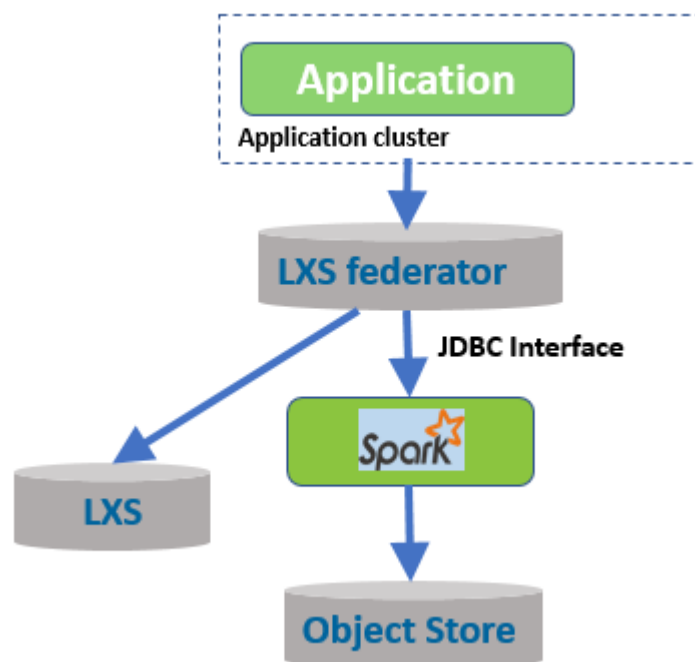


Figure 4 – Data Query Path

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

The entry point of the seamless component is the LeanXscale federator which federates the two underlying data stores. The component is based on the LeanXscale 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 LeanXscale 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.

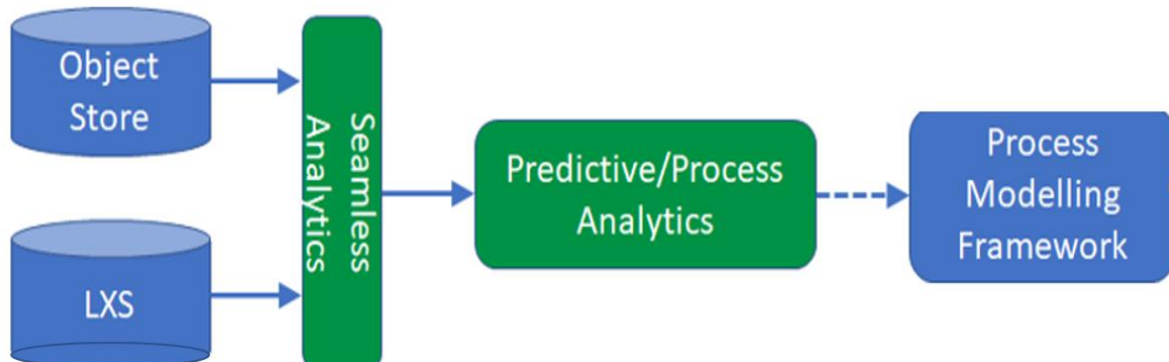


Figure 5 - Process and Predictive Analytics

4.2. Implementation Roadmap M19-36

At the current stage of the project, successful demonstrations of various WP4 prototypes have been demonstrated during the interim review (M19). The following tasks were demonstrated during this review: T4.1 (Data Quality), T4.2 (Data Skipping), T4.4 (Seamless framework), T4.6 (CEP). In addition, the Machine Learning software for predicting vessel engine failures was also demonstrated.

The design and initial implementation of task T4.3 (Adaptable Distributed Storage) and its prototype has taken place in the first part of the project. Its demonstration has been left to the second part of the project.

Task T4.5 (Predictive and process Analytics) has been left for the second part of the project.

Building on the initial integration of the tasks, the main goals for the Data as a Service components for the second part of the project will be:

1. to develop the more advanced capabilities of the different technologies
2. to get fully integrated with the other major building blocks of BigDataStack overall architecture
3. to be applied to the other use cases

The following table section gives a roadmap of the goals of the second part of the project.

Table 2 – Implementation Roadmap

Component	M23	M25	M30	M35
Data quality assessment: Support for categorical dataset	Complete			
Data quality assessment: Support for incremental learning	Working prototype		Full implementation	
Data quality assessment: Integration with all use cases		Working prototype	Full implementation	
Support Object Store as metadata store	Working prototype		Full implementation	
Support new indexes for data partitioning (e.g., Bloom Filter)	Working prototype		Full implementation	
Hive Metastore support	Working prototype		Full implementation	
SQL Grammar Extension			Working prototype	Fully supported
Migration to the Apache Calcite engine	Working prototype	Fully migrated		
Implementation of the JOIN operator		Working prototype	Full implementation	
Integration with the LXS query engine		Working prototype	Working prototype	
Experimentation using the tpc-ch benchmark			Implementation of the benchmark	Results are gathered
Support process mining algorithms based on ProM		Working prototype	Full implementation	
Empirical assessment of process mining			Initial evaluation	Final evaluation
CEP: Support for queries running in heterogeneous nodes	Working prototype		Full implementation	
CEP: Preliminary performance evaluation		Completed		
CAP: Support of distributed queries among heterogeneous nodes		Working prototype	Full implementation	
CEP: Performance evaluation			Initial evaluation	Final evaluation

5. Big Data Layout and Data Skipping

5.1. Introduction

This section has been fully updated as compared with the D4.1 version to reflect 12 months of progress. In addition, 3 completely new subsections have been added:

Indicators of Skipping Effectiveness

Extensible Data Skipping

Use Cases and Experimental Evaluation

5.2. Requirements Specification

We refer the reader to deliverable D2.3 section 9.10 (Big Data Layout) for a description of the problem tackled by this module and a comprehensive review of the best practices.

In order for this section to be self-contained, we reproduce here the basics of “data skipping” and “data layout”:

Data Skipping is a technique to minimize the data that has to be read from object store to Compute Store. This technique utilizes metadata to track information about objects and their dataset columns which can then be used for data skipping i.e. to show that an object is not relevant to a query and therefore does not need to be accessed from storage or sent on the network from Object Storage to Spark. To make the Data Skipping technology efficient, we index the metadata, so that during query execution, objects that are irrelevant to the query can be quickly filtered out from the list of objects to be retrieved for the query processing. This technique applies to all data formats and avoids touching irrelevant objects altogether (see IBM presentation⁹ at the Spark Summit).

To get good Data Skipping one typically needs to pay attention to Data Layout. Data layout refers to all details regarding the storage of the data including object size, format, Hive style partitioning, and data partitioning, i.e. the assignment of data records to objects. We focus now on data partitioning. For any given query, we would like the records which satisfy the query to be grouped together in a small set of objects, so that the remaining objects can be skipped. In general, we need to partition the data so that it gives as much as possible data skipping for an incoming stream of queries (i.e. a workload), not just a single query. Note that the various queries may have conflicting requirements. Moreover, the workload changes over time, as does the data. In 2017, IBM Research independently developed the notion of *k*-d tree partitioning which uses query history to choose the partitioning columns and dataset medians as cutting points for partitioning. In parallel, a paper was published by an MIT team which used a similar approach and similarly applied it to Apache Spark for data skipping¹⁰. The work in this paper went beyond previous work by providing an adaptive

⁹ <https://databricks.com/session/using-pluggable-apache-spark-sql-filters-to-help-gridpocket-users-keep-up-with-the-jones-and-save-the-planet>

¹⁰ A. Shanbhag, A. Jindal, S. Madden, J. Quiane, and A. J. Elmore, “A robust partitioning scheme for ad-hoc query workloads,” SoCC, 2017.

approach to repartition datasets on the fly according to a cost model. This is a cutting-edge research area which is also promising in terms of its applicability to analytics on real world big datasets.

This section undertakes further research in this area as well as apply it to a commercial setting. The following tables present the requirements that Big Data Layout and Data Skipping module should satisfy. Please refer to the footnotes for further clarifications.

Table 3 - Requirement REQ-BDL-01 for Big Data Layout

	Id ¹¹	Level of detail ¹²	Type ¹³	Actor ¹⁴	Priority ¹⁵
	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 4 - 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				

¹¹**Identifier:** To be used in D2.3 to allow for the correct traceability of requirements.

¹²**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).

¹³**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).

¹⁴**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.

¹⁵**Priority:** Requirements can have different priorities: MAN (mandatory requirement), DES (desirable requirement), OPT (optional requirement), ENH (possible future enhancement).

Table 5 - 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 heavily queried data already in object store.				
Additional Information	N/A				

Table 6 - 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 7 - 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 8 - 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	<p>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).</p> <p>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</p>				
Additional Information	N/A				

Table 9 - 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	<p>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).</p>				
Additional Information	N/A				

Table 10 - 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	<p>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.</p>				
Additional Information	N/A				

Table 11 - 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				

5.3. Design

Our Data Skipping module allows skipping over objects which can be proved not to be relevant to a query by using metadata which summarizes that data in one or more columns for the object. Currently we store this metadata in Elastic Search, per requirement [REQ-BDL-05] we plan to add the possibility to have the metadata stored within the Object Store itself. Examples of data skipping metadata include storing minimum and maximum values for numeric types, or bounding boxes for geospatial data. Our code intercepts the Apache Spark SQL query execution flow in the phase where an object listing is created and pruned and adds an additional phase which further filters the list of objects by skipping over objects that can be proved to be irrelevant to the query. Figure 6 shows the regular Spark SQL query flow on the left, and our modified flow on the right.

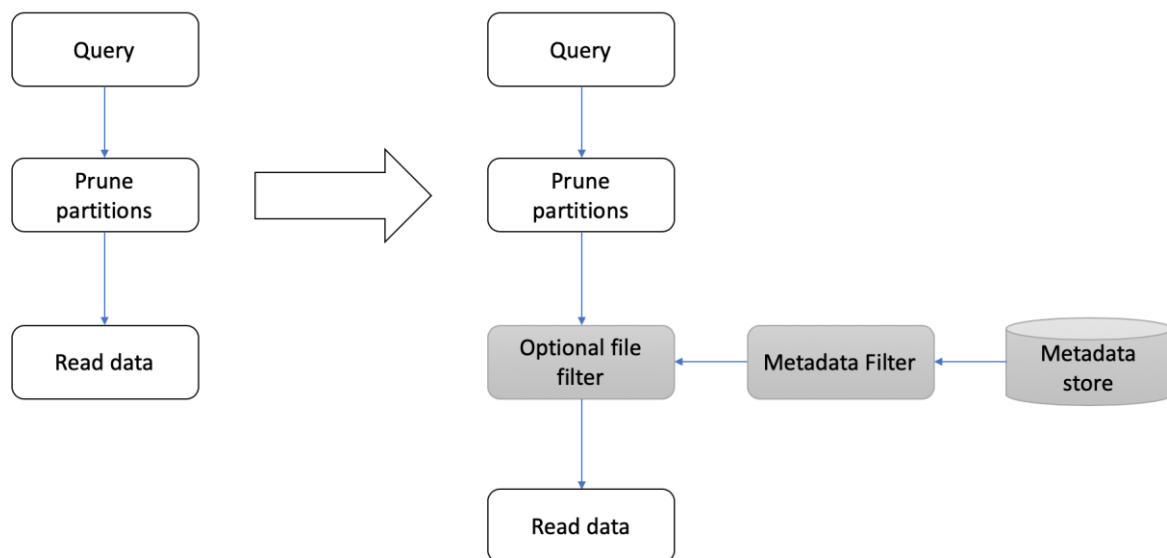


Figure 6 - Spark SQL Query Execution Flow

Our data skipping library code works with Apache Spark SQL but does not require any changes to core Spark. To do this we use the “Extra Optimizations” API of Catalyst, Spark’s optimizer. We add a special optimization rule which scans the logical plan tree and replaces the default *InMemoryFileIndex* with our enhanced *IndexedCatalog*, which extends the *InMemoryFileIndex* but further filters objects using Elastic Search as described above.

The Data Skipping module allows creating data skipping indexes on datasets residing in the Object Storage, and subsequently exploiting these indexes to skip over objects irrelevant to a given SQL query. We made novel and significant extensions to the Data Skipping component by:

1. Allowing users to define new data skipping indexes, without changing the core data skipping library;
2. Supporting data skipping for arbitrary query predicates including UDFs and AND/OR/NOT operators.

These two capacities are important and novel for different Big Data applications and operations, such as the ship management use case as was demonstrated during the interim review.

The purpose of the Data Layout Manager component is to organize/partition the rows of a dataset as objects in the Object Storage in a way that improves analytics performance, by reducing the number of bytes sent from the Object Storage to an analytics framework, such as the Apache Spark (the main KPI targeted by this component). This component is essentially a Spark application which works in two phases:

1. The first phase analyses the dataset and builds a *k*-d-tree
2. The second phase uses partitions the dataset into objects according to the *k*-d-tree

In our current implementation, the user specifies explicit commands to control the data layout, by specifying the columns to use for layout and their relative priorities. In the future, we plan to collect query history and data statistics for specific use cases, and based on this, to automatically recommend a data layout and associated choice of data skipping indexes for that use case to enhance the effectiveness of data skipping.

Data Layout can be performed on a dataset, which already exists in the Object Storage – we call this Offline Data Layout. Alternatively, Data Layout can be performed dynamically while data is ingested into Object Storage: we call this Online Data Layout.

There are two main flows in which Data Skipping and Data Layout participate. In the ingestion flow, Online/Offline Data Layout is used to organize the dataset rows into objects, either on the fly or after ingestion. Moreover, the Data Skipping module is used to create data skipping indexes to be used by subsequent analytics. The parameters such as the choice of the columns to be used for layout and skipping are currently provided by the user. In the future, there will be a query logging component which records the query history, as well as a data logging component which records dataset properties and their change over time as new data are ingested. This information can then be used to provide recommendations for the above parameters.

Figure 7 depicts the various possibilities for the ingestion flow. Note that data is directly ingested from the LeanXscale data base via one or many SQL queries and subsequently uploaded to the Object Storage.

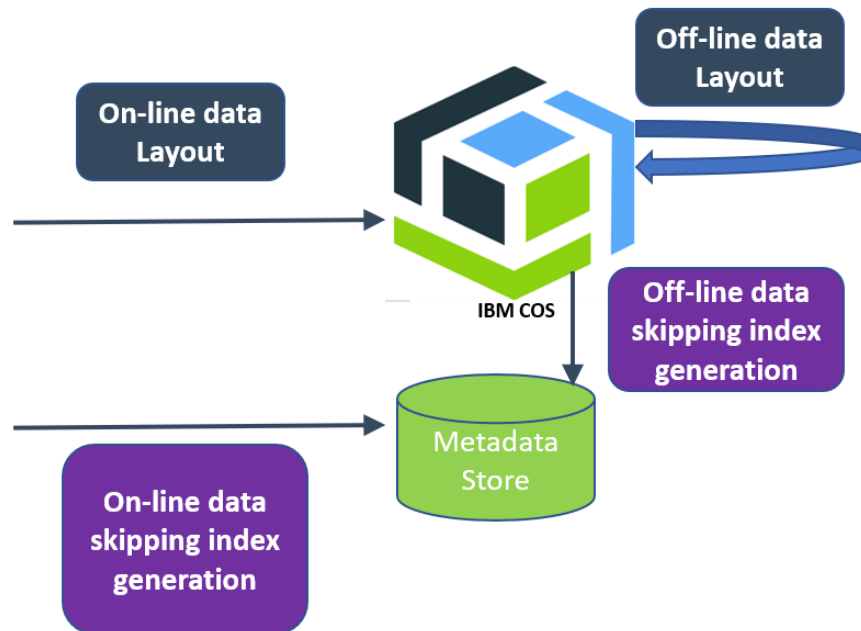


Figure 7 – Data Ingestion flow

In the analytics flow, (see Figure 8) given a query, the metadata store (in which the generated meta-data is stored) is consulted in order to skip over irrelevant objects whenever possible. Note that here a given SQL query is submitted by the LeanXscale data base and reaches Spark via a JDBC interface.

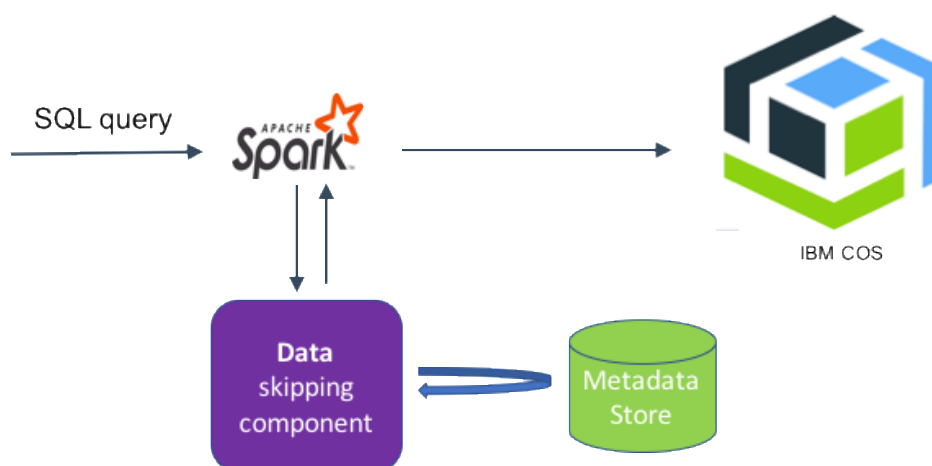


Figure 8 – Data Analytics flow

5.4. Indicators of Skipping Effectiveness

This subsection's aims at formalizing the notion of "skipping effectiveness". We assume that the query cost: c_q is proportional to the total number of bytes scanned.

Therefore, a good metadata representation should be compact as well as highly effective at skipping data.

Note that minimizing the metadata size would also be beneficial. However, small metadata is typically generated even for big datasets and therefore this optimization is of secondary importance and kept beyond the scope of our research.

Given a dataset 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 objects¹⁶, let O denote the set of all of the 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_m is a superset of O_r .

Note that $O_s = O - O_m$ is the set of objects that can safely be skipped. For an object or dataset v let $|v|$ denote the number of rows in v .

The **selectivity** σ of a query is the proportion of relevant rows $\sigma = |D_r| / |D|$

Data skipping can potentially reduce bytes scanned for selective¹⁷ queries. The definitions use relevant rows rather than rows in the result set to account for queries which perform further computations such as aggregation.

The **layout factor** λ of a query is the proportion of relevant rows in relevant objects $\lambda = |D_r| / \sum_{o \in O_r} |o|$

A high layout factor means that the right metadata index can potentially lead to good skipping for selective queries.

The **metadata factor** μ of a query is $\mu = \sum_{o \in O_r} |o| / \sum_{o \in O_m} |o|$

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.

The **scanning factor** ψ of a query is the proportion of rows actually scanned (using metadata) $\psi = \frac{\sum_{o \in O_m} |o|}{|D|}$

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

$$\psi = \sigma / \lambda \mu$$

¹⁶ Alternatively, other units can be considered such as blocks, row groups, etc.

¹⁷ selectivity ranges between 0 and 1. "Highly selective" queries have close to 0 selectivity

To achieve this for a selective query we need $\lambda\mu$ to be high, so we are dependent on both good layout and effective metadata. We focus here on metadata effectiveness for any given data layout, and refer the reader to previous work regarding data layout optimization.

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 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.

We apply the above definitions to real world datasets and workloads in section Use Cases and Experimental Evaluation to evaluate metadata effectiveness given a particular layout.

5.5. Extensible Data Skipping

In this section we present our extensible data skipping APIs as well as example use cases. Finally, we provide a proof of correctness of our approach. The following section will cover our implementation including how it is integrated into Apache Spark.

Our APIs allow the developer to (1) create data skipping indexes, including adding support for new types of summary metadata to support new index types, and (2) specify how to exploit data skipping indexes during query evaluation, including specifying how to map predicates used in a query to operations on the summary metadata, in order to evaluate whether data can be skipped.

Our general guidance to developers is firstly that the summary metadata needs to be significantly smaller than the data, otherwise this would defeat the purpose of reducing I/O. Secondly, in order to preserve correctness of query evaluation, skipping decisions should have no false negatives - whenever we skip data we need to be sure it contains no relevant rows. However false positives are acceptable, because we assume that subsequent to data skipping, the query evaluation engine will process and appropriately filter all data as specified in the query.

In principle, data skipping metadata can be stored for any row/column subset of a structured dataset. Our integration into Spark dictates that we operate at the object level and skip entire objects, although the same APIs could support different design choices.

For simplicity, we provide a running example explaining our APIs for min/max data skipping. Consider the query below, which applies to a weather data table and returns those rows representing heat waves (temperature > 101F).

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

Useful data skipping metadata for this case is the minimum and maximum temperature for a row subset.

5.5.1. Extensible Data Skipping APIs

Our Scala APIs can be grouped into those which support creating data skipping indexes, and those which support query evaluation using them.

In addition, users can define new metadata types which extend our **MetadataType** abstract class, such as the example below.

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

details of that API are beyond the scope of this deliverable.

5.5.2. Index Creation

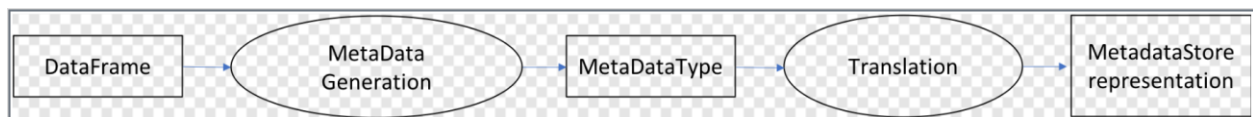


Figure 9 - Index Creation Flow

Our Spark APIs run against existing datasets and support creating indexes as a dedicated Spark job.

Index creation runs in 2 phases – see Figure 9. The first phase accepts a Spark DataFrame (which represents a row/column subset) and generates metadata having some **MetadataType**. The second phase translates this metadata to a metadata store representation, depending on the chosen store. These functions are applied to all data subsets (objects) using an overarching massively parallel Spark map/reduce job, where the map generates the metadata and the reduce collects it and stores it in the metadata store.

In order to implement the first phase, the developer extends the **Index** class.

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

Our **MinMaxIndex** extends **Index**, and its **collectMetadata** method returns a **MinMaxMetaData** object having the minimum and maximum values for the given column of the dataframe.

The second phase then translates this object to an appropriate metadata store representation.

5.5.3. Query Evaluation

Spark has an extensible query optimizer called Catalyst¹⁸, which contains a general 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 10 shows the expression tree for our example query.

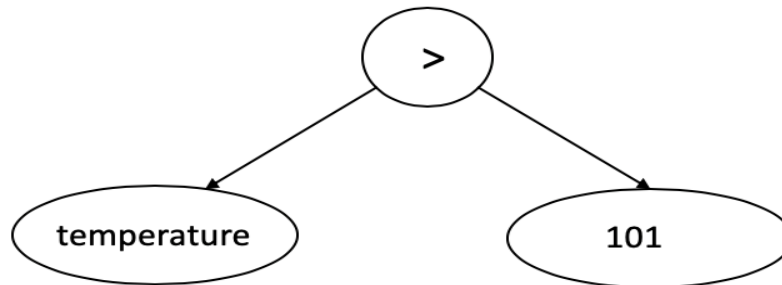


Figure 10 - Expression Tree for Example Query

We analyse these ETs and label the tree nodes with *Clauses*. A Clause is a boolean condition that can be applied to a data subset s , typically by inspecting its metadata.

Definition1. Denote the universe of possible data subsets (i.e., *files*) by U . A clause c is a Boolean function $U \rightarrow \{0,1\}$.

Definition2. For a Clause c and a (boolean) query expression e , we say that c **represents** e (denoted by $c \models e$), if for every data subset s , whenever there exists a row r in s that satisfies e , then s satisfies c .

This means that if s does **not** satisfy c , then s can be safely skipped when computing e . For example, let e be the query expression $temperature > 101$. Given a data subset s , let c be the clause which returns true iff $\max(\{temp(r) \mid r \text{ in } s\}) > 101$. Then c represents e . Therefore, if we discover objects *where* $\max(\{temp(r) \mid r \text{ in } s\}) \leq 101$, then these objects can safely be skipped.

Query evaluation is done in 2 phases as shown in Figure 11 - Query evaluation flow. In the first phase, a query's ET e is labelled using clauses and the clauses are combined to provide a single clause which represents e . Both the labelling process and the way labels are combined is extensible, and we will present the APIs used for that. In the second phase, this clause is translated to a form that can be applied to the metadata store to filter out the set of objects which can be skipped during query evaluation.

¹⁸ <https://databricks.com/glossary/catalyst-optimizer>

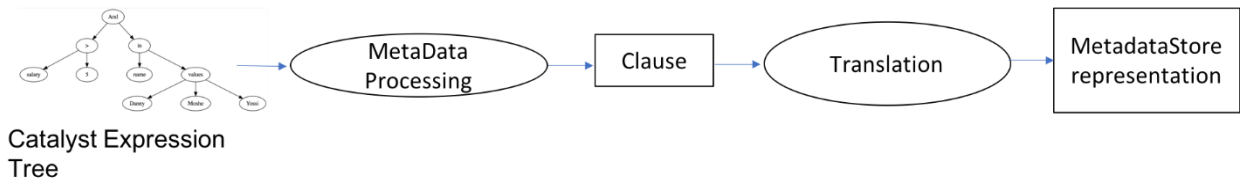


Figure 11 - Query evaluation flow

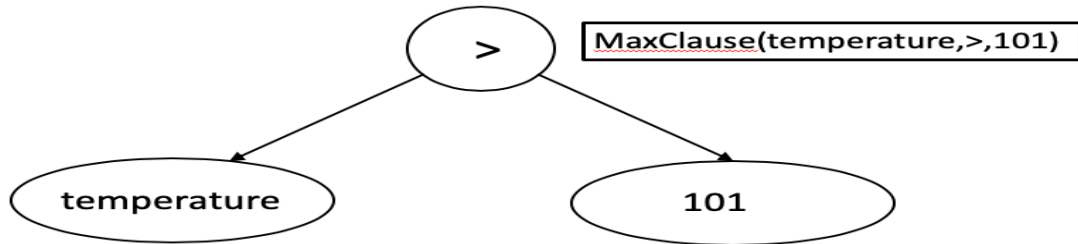


Figure 12 - Result of a filter on ET

We now describe the first phase in more detail, explain our extensible APIs, and prove the correctness of our approach.

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** which will 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 : $c \not\vdash v$ ¹⁹.

For example, our filter f might label our ET using **MaxClause**, as shown in Figure 12, where for a column name c and a value v , $\text{MaxClause}(c, >, v)$ is true for a data subset s iff $\max(\{c(r) \mid r \in s\}) > v$. Since $\text{MaxClause}(\text{temperature}, >, 101)$ represents the single node of our expression tree, f acted as a filter. Since $\max(\{c(r) \mid r \in s\})$ 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] = {
```

¹⁹ Note that for a particular node, a filter might not add any clauses (this is the special case of adding the empty set).


```

val res = node.expr match {
  case GreaterThan(attr: Attribute, v: Literal)
    if attr.name == col => MaxClause(col, GT, v)
  case LessThan(v: Literal, attr: Attribute)
    if attr.name == col => MaxClause(col, GT, v)
  case _ => null
}
Option(res)
}

```

Filters typically use pattern matching on the ET structure. For simplicity we left out the cases of \leq and \geq for MaxFilter above. Similarly, we can define a MinFilter which can label a tree with MinClauses.

Patterns can also match against UDFs used in expression trees in a query, and in this way the developer can specify data skipping for these too. An example of this will be given in the following.

Each MetaDataFilter needs to be registered in our system and 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 need a way to 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; - Algorithm Merge Clause and - Algorithm Generate Clause describe how this is done.

We point out that negation of an expression e can be handled if we can construct a clause representing $\neg e$.

Definition 4. 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 $c_e^* \not\models \neg e$

Note that this is not always possible. In the worst case, our algorithm will return None, meaning that no skipping can be done.

```

Input: an expression tree  $e$ 
Output: A Clause (possibly None)  $c$ 
if  $e = \text{AND}(a,b)$  then /* Case 1

    Let  $\phi := \bigwedge_{\gamma \in \text{CS}(e)} \gamma$ 
    Run the algorithm recursively on  $a$  and  $b$  and denote the result by  $\alpha, \beta$ 
    respectively
    return  $\alpha \wedge \beta \wedge \gamma$ 
else if  $e = \text{OR}(a,b)$  then /* Case 2

    Let  $\phi := \bigwedge_{\gamma \in \text{CS}(e)} \gamma$ 
    Run the algorithm recursively on  $a$  and  $b$  and denote the result by  $\alpha, \beta$ 
    respectively
    return  $(\alpha \vee \beta) \wedge \gamma$ 
else if  $e = \text{NOT}(a)$  then /* Case 3
    Run the algorithm recursively on  $a$  and denote the result by  $\alpha$ 
    if  $\alpha$  can be negated with respect to  $a$  then
        return  $\alpha_{a^*}$ 
    else
        return None
    end
else if  $e$  is a boolean operator then /* Case 4

    return  $\bigwedge_{\gamma \in \text{CS}(e)} \gamma$ 
else
    return None
  }
  
```

Figure 13 - Algorithm Merge Clause

```

Input: a boolean expression  $e$ , a sequence of filters  $O_1, \dots, O_n$ 
Output: A Clause (possibly None)  $c$ 
1 Apply  $O_1, \dots, O_n$  on  $e$ 
2 Run Merge - Clause( $e$ ) and return the result
  
```

Figure 14 - Algorithm Generate Clause

5.5.4. Correctness

As mentioned, our end goal is, whenever a query Q with predicates given by an expression e over a set of files $F=\{f_i\}_{i=1\dots n} \subset U$ needs to be computed, to obtain a clause c s.t. $c \not\models e$ (by running algorithm Merge Clause – see Figure 13), and compute the result over the set $\{f \in F \mid c(f)=1\}$ instead. By definition, because $c \not\models e$, no tuples that satisfy e exist in files which don't satisfy c . Thus, to show correctness, we prove the following theorem:

Theorem 1. Let e denote a boolean expression, Let O_1, \dots, O_k denote a sequence of *filters*. Denote by c the output of - Algorithm Merge Clause on e (with O_1, \dots, O_k). Then $C \not\models e$

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

Lemma 2 Let e denote a boolean expression, let c_1, c_2 s.t. $c_1 \not\models e \wedge c_2 \not\models e$. Then $(c_1 \wedge c_2) \not\models e$.

Proof: Assume the stated assumptions. we will show that $(c_1 \wedge c_2) \not\models e$.

By definition: let $f \in U$ s.t. $\exists r \in f: e(r)=1$. Then - since $c_1 \not\models e$, we get $c_1(f)=1$, identically we get $c_2(f)=1$, thus $c_1(f)=1 \wedge c_2(f)=1 \implies (c_1 \wedge c_2)(f)=1$.

Lemma 3 Let e_1, e_2 denote a pair of boolean expressions, let c_1, c_2 s.t. $c_1 \not\models e_1 \wedge c_2 \not\models e_2$ Then $(c_1 \wedge c_2) \not\models (e_1 \wedge e_2)$.

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

Lemma 4 Let e_1, e_2 denote a pair of boolean expressions, let c_1, c_2 s.t. $c_1 \not\models e_1 \wedge c_2 \not\models e_2$ Then $(c_1 \vee c_2) \not\models (e_1 \vee e_2)$.

Proof: Assume the stated assumptions and let $f \in U$ s.t. $\exists r \in f: (e_1 \vee e_2)(r)$, we will show that $(c_1 \vee c_2)(f)$: in particular, if $e_1(r)$ then $c_1(f)$, else we get $e_2(r)$, which implies $c_2(f)$, thus we get $c_1(f) \vee c_2(f) : (c_1 \vee c_2)(f)$.

Remark 5: The above-mentioned lemmas can easily be re-stated 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 6 Let e denote a boolean expression, and by T_e the expression tree rooted at e . Assume the following holds:

Assumption 7: $\forall v \in T_e \forall c \in CS(v): c \not\models v$.

Denote by C the output of - Algorithm Merge Clause on e , then $C \not\models e$.

Proof: By full induction on d - the depth²⁰. We assume WLOG that all $\{v, \wedge, \neg\}$ nodes are of degree ≤ 2 .

²⁰ in this case the depth is defined as the maximum length (in edges) of a path from the root (T_e) to a $\{v, \wedge, \neg\}$ node,

Case 1: (Base case, $d = 0$) In this case, e is a single boolean operator, so case 4 of - Algorithm Merge Clause is applied. By our assumption, $\forall c \in CS(e). c \not\vdash e$, by lemma 2, we

get $\bigwedge_{\gamma \in CS(e)} \gamma \not\vdash e$

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 $k \in \{0 \dots d-1\}$. Since $d > 0$, cases 1,2,3 of - Algorithm Merge Clause are the only options.

Case 2.a: if $e = \text{AND}(a, b)$: in this case, - Algorithm Merge Clause is called again on a, b , use α, β from - Algorithm Merge Clause's notation. a, b are both expressions of depth strictly smaller than d , so by the inductive hypothesis we have $\alpha \not\vdash a$ and $\beta \not\vdash b$; by lemma 3 we get $(\alpha \wedge \beta) \not\vdash (a \wedge b)$. By lemma 2 and from Assumption 7 we get $(\varphi$

$= \bigwedge_{\gamma \in CS(e)} \gamma) \not\vdash e$. Applying lemma 2 again we get $(\alpha \wedge \beta \wedge \varphi) \not\vdash e$, and indeed this is the output in this case.

Case 2.b: if $e = \text{OR}(a, b)$: in this case - Algorithm Merge Clause is called again on a, b , use α, β from notation. a, b are both expressions of depth strictly smaller than d , so by the inductive hypothesis we have $\alpha \not\vdash a$ and $\beta \not\vdash b$; by lemma 4 we get $(\alpha \vee \beta)$

$\not\vdash e$. By lemma 2 and from Assumption 7 we get $(\varphi = \bigwedge_{\gamma \in CS(e)} \gamma) \not\vdash e$. Applying lemma 2 again we get $((\alpha \vee \beta) \wedge \varphi) \not\vdash e$, and indeed this is the output in this case.

Case 2.c: if $e = \text{NOT}(a)$: in this case - Algorithm Merge Clause is called again on a , and the result is noted as α .

Case 2.d: if α can be negated with respect to a - Algorithm Merge Clause returns α_{a^*} and by definition $\alpha_{a^*} \not\vdash \neg a = e$

Case 2.e: if α can NOT be negated with respect to a : None is returned, which represents *any* clause.

We are now ready to prove Theorem 1:

Proof of Theorem 1. From - Algorithm Merge Clause's assumptions we know that O_1, \dots, O_n are *filters*, thus Assumption 7 holds. Thus, correctness follows from Lemma 6.

5.6. Prototype

IBM Data Skipping and Data Layout modules have been extended in several significant ways for the benefit of BigDataStack and was successfully demonstrated during the interim review.

Both Data Skipping and Data Layout code was extended to accurately measure bytes read from COS to Apache Spark. It is important to measure this accurately since it is the main KPI.

comprised of $\{V, \wedge, \neg\}$ nodes only, so for example the depth of $(a + b < 2) \wedge (c < 5)$ is 1

This is done using the Spark Measure library from CERN²¹. Bytes read from COS is measured when creating data skipping indexes, and also when running queries against COS. Moreover, when running the Data Layout Manager, we measure both the bytes read number and the bytes written to COS number. Work measuring bytes written to COS accurately is currently in progress since it requires a change to the underlying Stocator²² driver. Note that previously we were only able to measure the total number of objects skipped by a query, and to aggregate the total sum of bytes skipped in those objects. However, these statistics are not helpful when data is stored in formats such as Parquet or ORC, because when accessing these formats Spark typically accesses only parts of the objects, for example, it only reads those columns accessed by the query. Our new method is able to measure the number of bytes actually read. We authored a blog on existing best practices for Big Data Layout for Spark SQL analytics on data in IBM Cloud Object Storage.²³

The Data Skipping module was extended to support arbitrary query predicates, including UDFs (user defined functions) and AND/OR/NOT. This is particularly challenging, because UDFs can be arbitrary functions about which the Spark optimizer knows very little. This work will be applied to geospatial UDFs, which are important for the ship management use case. See the next section to understand why this is needed. In addition, the Data Skipping code now supports a truly pluggable architecture, where new data skipping index types can be added without changing the core data skipping library. For example, we added a new geospatial data skipping index type using this capability, which works well in conjunction with geospatial UDFs. External users can also exploit this capability. More technical details describing this work will appear in future.

The Data Layout Manager has undergone a complete refactoring. It has also been updated to handle string data types as well as numeric data types. String data types behave differently from numeric data types since in most cases they are queried using '=' or 'IN' predicates and not using inequality predicates. Therefore, our *k*-d-tree approach which splits the data according to the median value may not be ideal in this case. The *k*-d-tree approach was extended to support arbitrary cut nodes with unlimited number of children which enables each cut node to implement its own logic to decide how to split the data. We are experimenting with various alternatives of cut nodes, some of which take into account the number of distinct values in a column, and the number of appearances of each value. A key challenge is to do this efficiently. We also support sampling the input dataset when running the Data Layout Manager, which is important to achieve good performance.

5.7. Use Case Mapping

The ship management use case requires efficiently querying vessel (ship) trajectories and vessel engine data. The ship management use case also requires joining this data with weather data, to enable queries which combine both data sources. As described in the seamless section, this data will be stored across both the LeanXcale data base and Object

²¹ <https://github.com/LucaCanali/sparkMeasure>

²² <https://github.com/CODAIT/stocator>

²³ [How to Layout Big Data in IBM Cloud Object Storage for Spark SQL](https://www.ibm.com/blogs/bluemix/2018/06/big-data-layout/)
<https://www.ibm.com/blogs/bluemix/2018/06/big-data-layout/>

Storage, where the bulk of the data stored long term will be in Object Storage. Therefore, an efficient method of organizing the data in Object Storage and querying it is required.

Many of the queries required by the ship management use case are geospatial in nature. For example, the data scientists analysing the ship datasets might need to check whether there is a correlation between engine failures and a certain geospatial region. Moreover, joining the vessel data with weather data is geospatial in nature, because one needs to combine information about a ship with weather data from a similar location (and time).

In order to query geospatial data effectively, geospatial library functions are needed, which take into account the curvature of the Earth as well as handling various coordinate representations. Such libraries are often not built in to a Big Data engine, but can be added dynamically, as is the case for Spark SQL. Once such a library is added, its functions can be registered as UDFs (user defined functions), and queries can refer to them.

Optimizers such as Spark's Catalyst optimizer are typically unable to make intelligent decisions regarding UDFs since they know nothing about them. Specifically, for queries involving Big Data on Object Storage, essential optimizations are to choose an effective layout of the data and to enable skipping as much data as possible when it is not relevant to a query. This is critical, but challenging, when UDFs are involved. For example, consider the following query, where the ST_distance function computes the spatial distance between two points:

```
// Registration of Spatial Functions, "spark" is the sparkSession:
SqlGeometry.registerAll(spark)
// We assume df, is the original dataframe
// and that it does *not* contain a column named "location"
val locationDf = df.withColumn("location",toPointEG($"lat","long"))

val targetLat = someValue
val targetLng = someValue
val targetRadius = someValue

// We create a temp view named VESSEL_DATA
locationDf.createOrReplaceTempView("VESSEL_DATA")

val query = "select vessel_id
FROM VESSEL_DATA
where
  ST_Distance(location,ST_WKTTToSQL('POINT(targetLat , targetLng )'))
  <
  targetRadius"

// and run the query:
spark.sql(query)
```

Data layout and skipping are essential here to reduce the amount of data read from the Object Storage and shipped across the network to the area surrounding Barcelona. However, without any knowledge of the ST_distance UDF in the Spark Catalyst optimizer this is not possible.

In order to cater for the ship management use case, we will exploit the data layout and data skipping modules to handle cases where arbitrary query predicates including UDFs are involved. Moreover, we handle the case where data is multi-dimensional and data layout and skipping needs to take multiple dimensions (geospatial, time, and additional dimensions) into account.

Note that UDFs can be useful for many use cases and are not specific to geospatial scenarios. Moreover, the requirement to handle multi-dimensional data applies to most use cases.

We demonstrated during the interim review the Data Skipping technology applied to the ship management use case.

5.8. Use Cases and Experimental Evaluation

In the interim review the Data Skipping technology was demonstrated applied to the ship management use case. This technology was already proved to both reduce the size of the data read from Object Store as well as the overall time to run SQL queries of this use case.

However, the data skipping technology will be especially relevant for the intended future scale of data when the Danaos company will both manage many more vessels, and when each vessel will generate much more IoT data. Given the relatively small amount of data currently available with this use-case, we chose to test and evaluate the effectiveness of the Data Skipping technology with a geospatial analytics use case for which we have access to a proprietary production dataset comprising many hundreds of Gigabytes of data. It should be well understood that although we focus here on a specific data analysis but our work is applicable to analytics on any kind of structured data with geospatial attributes.

We demonstrate SQL queries on a proprietary data set containing 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 partitioned using a KD-Tree partitioner of height 13 on the latitude and longitude columns. One month of weather data was stored in 8192 Parquet objects using the default snappy compression with a total size of 191GB.

The data skipping that will be demonstrated involves UDFs applied to a geospatial use case. To our knowledge, no other SQL engine supports data skipping for queries using UDFs, since query optimizers typically know nothing about the UDF at hand.

All experiments were conducted on Spark 2.3.2 using an IBM Analytics Engine cluster of 4 nodes each with 128GB of RAM, 32 vCPU.

The datasets are stored in IBM COS. All experiments are run with cold caches.

5.8.1. Data Skipping for Geospatial UDFs

Geospatial data skipping is supported by integrating IBM's geospatial toolkit²⁴, a library of geospatial UDFs, into our framework. To achieve this a plugin library was created containing the implementation of our APIs. Supported predicates include containment, intersection, distance and many more.

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((-64.7 32.3,...))'),  
ST_POINT(lat, long))
```

In order to support skipping we can either use the GeoBox index on the pair of lat/long columns, or use independent MinMax indexes on both lat and long. For each case we map the relevant UDFs to the corresponding metadata. 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/long, we expect a high layout factor for geospatial queries, so the MinMax approach is also relevant.

²⁴ https://www.ibm.com/support/knowledgecenter/en/SS6NHC/com.ibm.swg.im.dashdb.analytics.doc/doc/geo_functions.html

A visualization for the query above can be seen in the - Geospatial data skipping (a) figure. In black we have the Bermuda Triangle polygon, in white a bounding box that bounds this polygon, and in grey one bounding box for each object. Each grey bounding box that does not intersect with the white bounding box represents an object we can skip. In our example, bounding boxes can be represented using either GeoBox or a pair of MinMax indexes. During query evaluation we use the constants in the query expression to construct a bounding box which represents the query and check whether it intersects any of the bounding boxes we saved as metadata. Each object whose metadata no intersection with the query has has bounding box can be safely skipped.

Examples where GeoBox is preferable include IoT data ingested and stored according to timestamp without geospatial layout. In that case a typical object could cover a large geospatial area. For example, the bike renting company Bay Wheels ²⁵ enables access to anonymized trip data which includes start and stop station locations. Station locations can be seen in the - Geospatial data skipping figure (b) marked in green. There are 3 main station areas: San Francisco, East Bay, and San Jose. In this case storing more than one bounding box per object makes sense because there are large gaps between the areas.

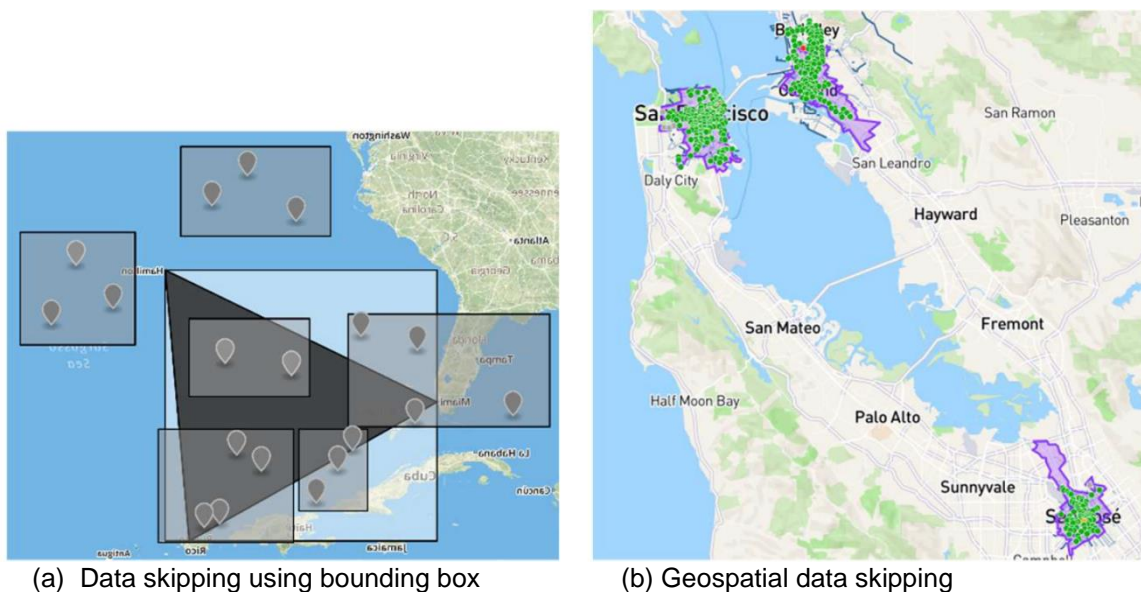


Figure 15 - Geospatial data skipping

Figure 16 and Figure 17 compare running ST_Contains queries with and without data skipping. The results for ST_Distance are similar. The queries were run on an extrapolation of the dataset to a 5-year period. Queries use time windows ranging between 1 month to 5 years.

The specific query we ran selects data with location in the Research Triangle area of North Carolina. We used MinMax indexes resulting in 11MB of metadata for close to 12TB of data.

²⁵ https://en.wikipedia.org/wiki/Bay_Wheels

We achieved a cost and performance gap which is several orders of magnitude - the gap increases in proportion to the size of the time window. For a 12-month window we achieved a x100 speedup, and we extrapolate that for a 5-year window we achieve a x10000 speedup²⁶. The cost gaps reflected by amount of data scanned are similar. Running these queries directly on object storage without data skipping is clearly not feasible.



Figure 16 - Data skipping on UDFs vs no skipping - data scanned

²⁶ in the latter case only the data skipping results completed in a reasonable timeframe since each month of data takes roughly 30 mins to analyze without skipping



Figure 17 - Data skipping on UDFs vs no skipping - Run Time

5.8.1.1. Benefits of centralized Metadata Store

We now demonstrate the benefits of our centralized metadata approach, compared to relying on metadata stored together with the data in formats Parquet and ORC²⁷. We focus on our geospatial use case, although it applies in general.

In the geospatial domain, an alternative approach to that of section 5.8.1 is to apply geospatial data layout and to rewrite queries to exploit min/max metadata if available in the underlying storage format. This approach either requires users to rewrite the queries manually, or else query rewrite needs to be implemented for each query template. For example, the previous query could be rewritten to add a condition filtering using a bounding box surrounding the polygon used in the query:

²⁷ Recall that our approach supports all Spark supported formats.

```
SELECT * FROM weather WHERE
ST_CONTAINS(ST_WKTToSQL('POLYGON((-64.7 32.3,...))'),
ST_POINT(lat, long)) AND
lat BETWEEN xxxx AND yyyy AND
long BETWEEN vvvv AND wwwww
```

In general, there is no framework available to assist the application developer achieve correctness and effectiveness using query rewrite, unlike our APIs for extensible data skipping.

Our approach uses centralized metadata which avoids reading the footers of irrelevant Parquet/ORC objects, achieving a significant performance boost over object storage where overheads for each GET request are high, and reducing bytes scanned.

Figure 16 and Figure 17 compares the cost and performance of extensible data skipping to a query rewrite approach.

Since the data is partitioned using a KD-Tree both approaches skip the same objects (we collect one bounding box per object). However, our centralized metadata approach performs x1.7 better at run time at x2.7 lower cost for 5-year time windows.

The bytes scanned are reduced both by avoiding reading irrelevant footers and by metadata compression.

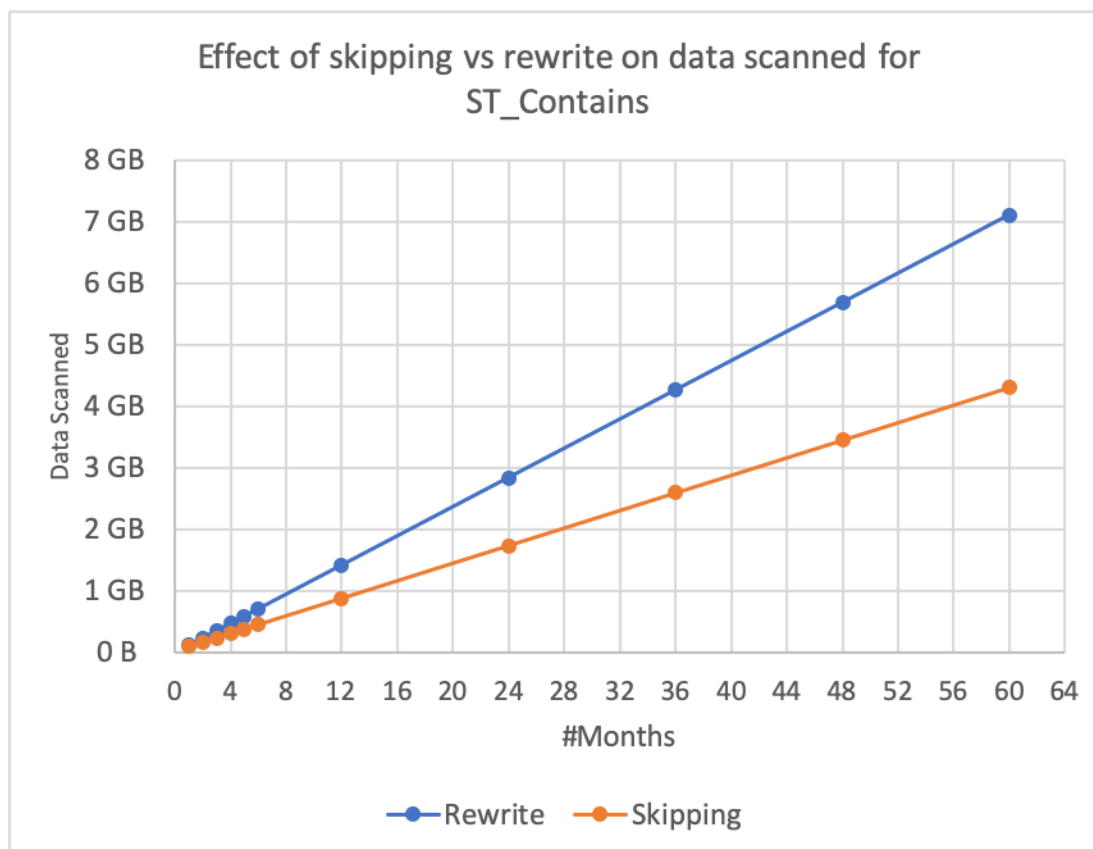


Figure 18 - Data skipping vs rewrite - Data Scanned

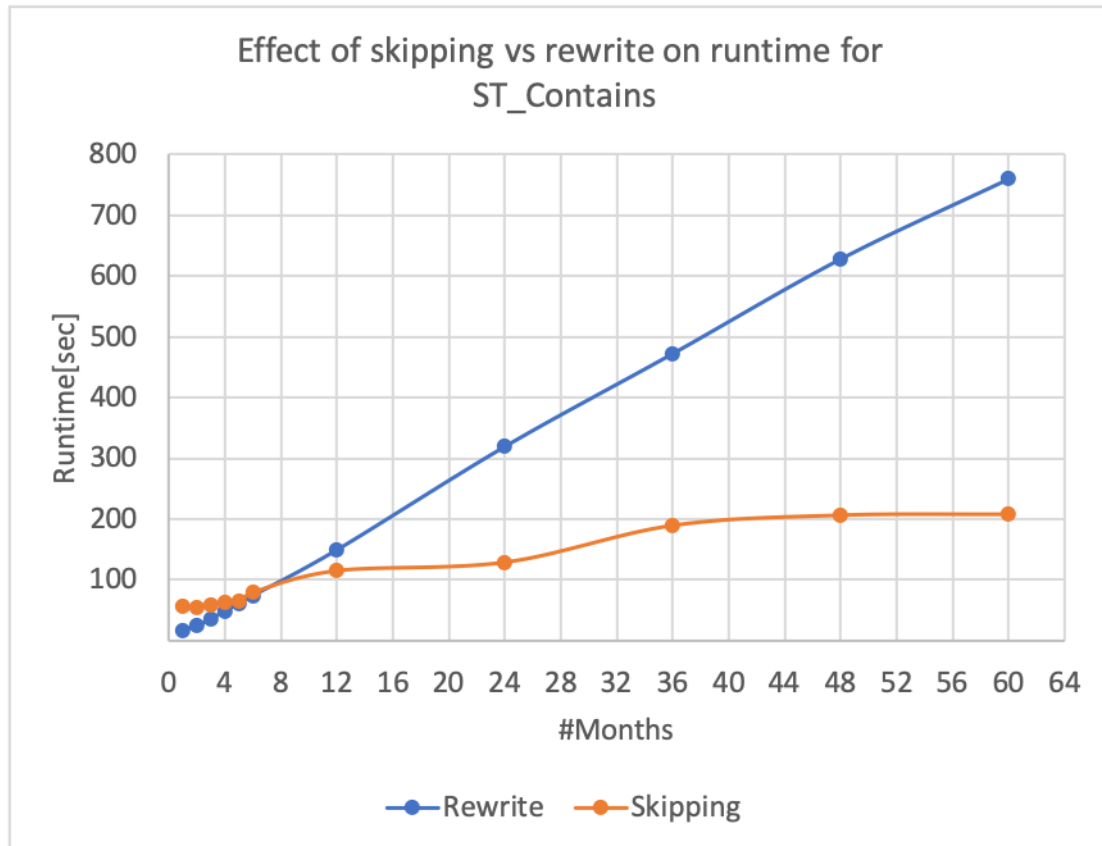


Figure 19 - Data skipping vs rewrite - Run Time

5.8.1.2. *Skipping effectiveness of different layouts*

In this section we demonstrate the indicators presented in sub-section 5.4 and show how those can be used to determine the preferred layout.

Specifically, we compare the following alternatives for the monthly dataset presented above:

Monthly KD-Tree - partitioning of the data using a KD-Tree of height 13 - 7 cut points on longitude and 6 cut points on latitude resulting in 8192 files.

Daily KD-Tree - hive style partitioning on day and underneath partitioning using a KD-Tree of height 7 - 4 cut points on longitude and 3 cut points on latitude resulting in 128 files per day.

Hourly KD-Tree - hive style partitioning on day and hour and underneath partitioning using a KD-Tree of height 3 - 2 cut points on longitude and 1 cut point on latitude resulting in 8 files per hour.

Monthly Geohash - A range partitioning of 13bit geohash to 8192 partitions using spark range partitioner.

All layouts aimed at getting objects sizes which are not too small while having geospatial layout in order to provide maximum skipping.

We evaluated the layouts on the following workloads:

- 10km radius query for a period of 1 month - running 10 queries - for each query choosing a random origin.
- 100km radius query for 1 hour period - running 10 - for each query choosing a random origin.
- Mixed workload - running 50 queries - for each query choosing a random origin, radius and time span (between 1 hour to 1 month).

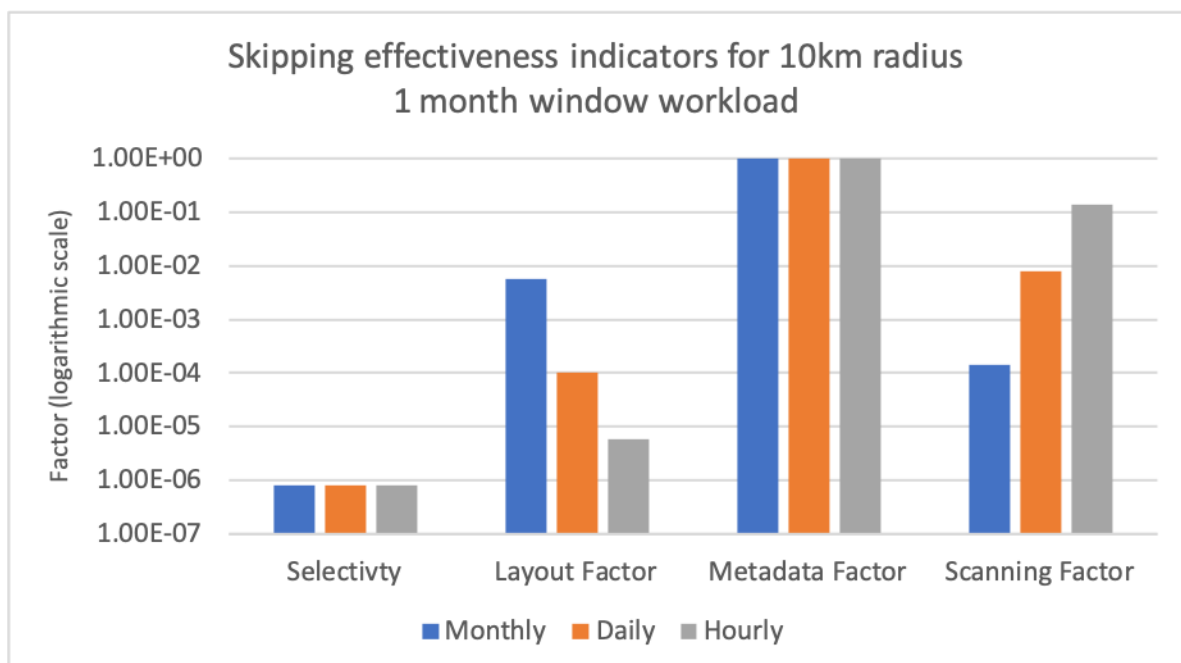


Figure 20 - Indicators of Skipping Effectiveness for geospatial analysis 10km queries

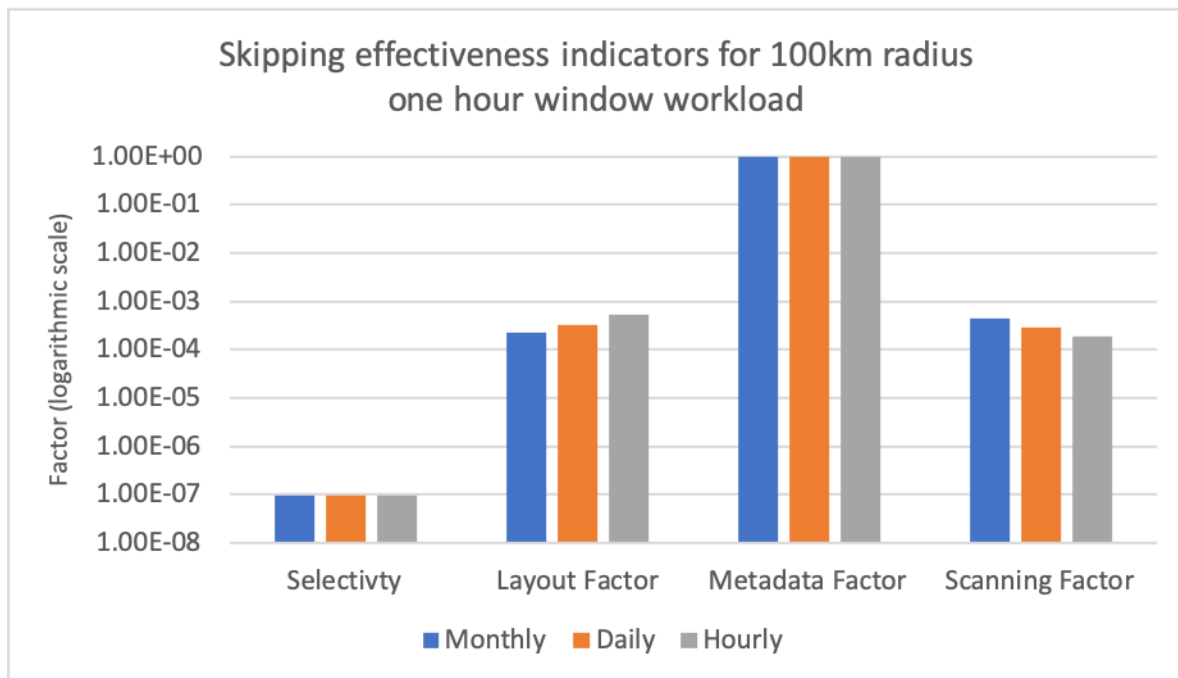


Figure 21 - Indicators of Skipping Effectiveness for geospatial analysis 100km queries

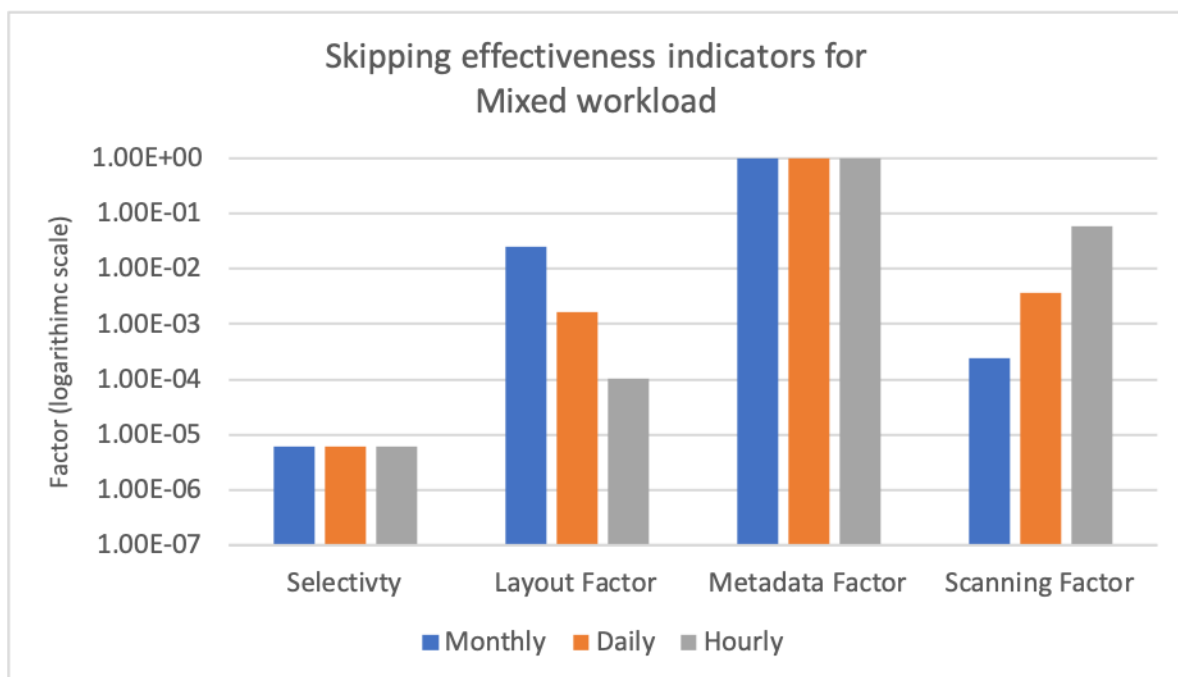


Figure 22 - Indicators of Skipping Effectiveness for geospatial analysis mixed queries

The 3 preceding figures show the skipping effectiveness indicators of running the above workloads. Note that the graphs are shown in a logarithmic scale so for selectivity factor and scanning factor lower values means better, for metadata factor and layout factor higher values means better.

First, we can see that as expected the selectivity is the same across all layouts and all workloads, In addition the metadata factor for all layouts is almost optimal where the main reason for the metadata being not optimal is when the origin of the query is close to the file edge such that the bounding box we build for the query spans more than one object.

The monthly KD-Tree layout has the best scanning factor for the 10km workload, while the hourly KD-Tree has the best scanning factor for the 100km workload, the reason for that is the difference in the layout factor, in the 10km workload the query looks for a large time span and a small area which favours the monthly layout while the 100km we look for a small time span and a wider area which favour the hourly layout. However, as we can see the difference between the hourly and monthly layout for the 100km is not that big as the difference for the 10km workload, the reason for that is that since we aim for file sizes which are not too small each file in the monthly layout contains large enough geospatial space.

This behaviour can also be seen in the results for running the mixed workload where the monthly layout has a better scanning factor.

5.9. Next Steps

The Data Skipping technology has already been transferred to the IBM Cloud SQL service where it can be used at beta level.

Obviously one major work item will be to analyse the feedback that we received and will receive from industry customers to improve the Data Skipping technology.

We also plan to improve our Layout technology and want to further apply our data skipping and layout technology to other use cases.

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. For the second phase of the project, the delivered functionalities will be further extended in order to provide truly elasticity.

In this section we provide an updated version of the overall requirements, as they were modified or extended during the first phase of the project and the initial implementation of the prototype. An updated design of the solution will be described, along with a documentation of the functionalities that are planned to be implemented until M23. Moreover, the results from the internal validation process is provided, and a descriptive work plan and for the next steps that are required is included, as the remaining functionalities were initially planned to be implemented and delivered at the end of the second phase of the project.

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 at 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 updated list of those requirements, as identified during the second 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 at this phase of the project.

Table 12 – 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 13 - 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	N/A				

Table 14 - 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 15 - 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	<p>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:</p> <ol style="list-style-type: none"> 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. <p>This requirement also includes the need from the adaptable storage to inform the infrastructure that it can release resources that are not needed.</p>				

Table 16 - 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 17 - 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, that 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 outlier 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 18 - 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 19 - 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 463 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 on the 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, will allow to dynamically adapt and reconfigure the data regions to deal with increased workloads. Finally, based on that, scaling in/out the data nodes of LeanXcale will transparently trigger the reconfiguration process, making this component truly elastic.

6.3. User Perspective

At this phase, it is still unclear which of the three use cases will be used to demonstrate the adaptable distributed storage, however its purpose is general and it is not targeting a specific use case rather than it is suitable for all scenarios that require from the storage to be able to adapt. For the purpose of the text and in order to highlight its usability, we will consider the ship management use case as the demonstrator that underlines the advantages and innovations of the tools developed. As described in the corresponding section, the ship fleet 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. Given future steep increase in number of supported vessels and for each vessel the number of sensors, the continuous data ingestion will impose significant requirements in order for this component to be able to store all data. Moreover, the size of the required storage will constantly increase, as new data are continuously ingested. Due to this, the *Adaptable Distributed Storage* component should have the ability to automatically scale out. This will be performed by requesting additional data nodes from the infrastructure, when they are needed at runtime. The provision 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. This might lead to under spending of the available resources, which are not needed at that time, so the *Adaptable Distributed Storage* can request from the infrastructure to de-allocate the reserved resources.

6.4. Detailed Design

Figure 23 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.

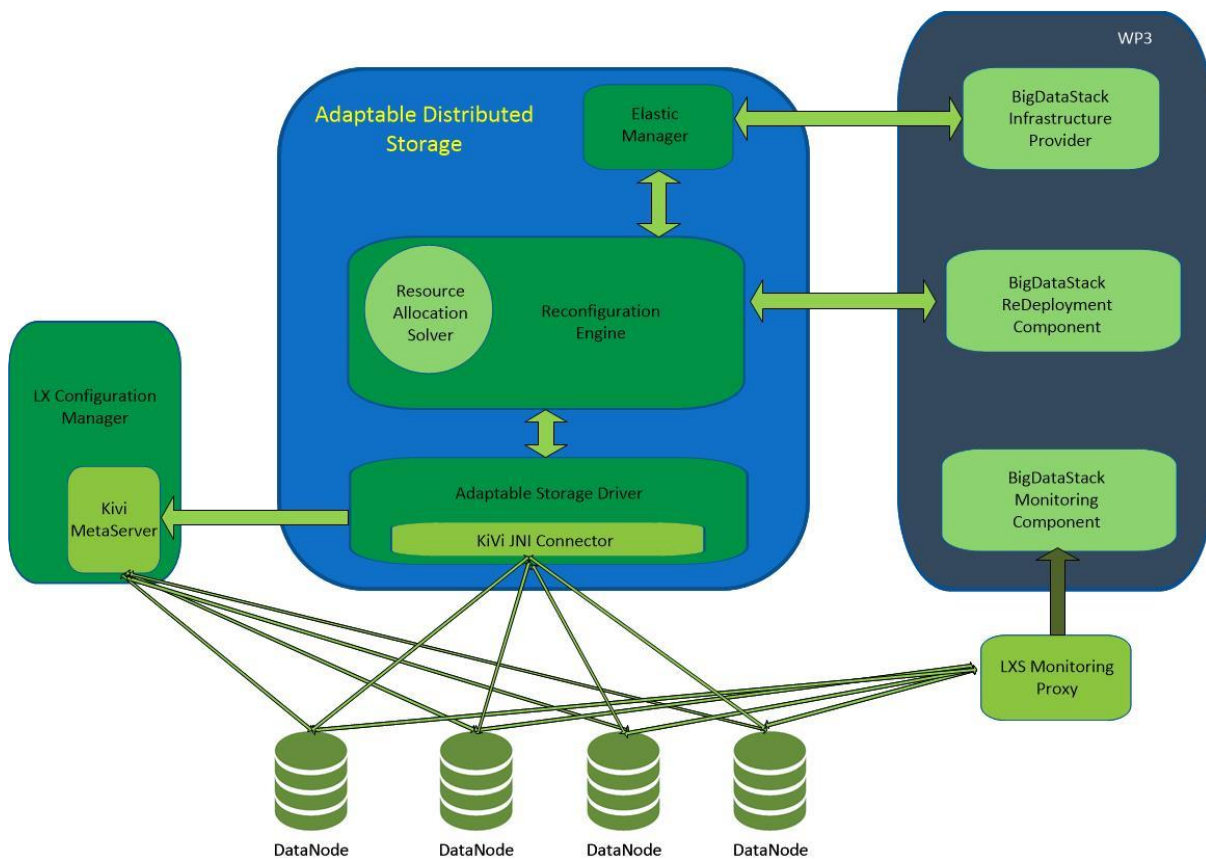


Figure 23 - Adaptable Distributed Storage architectural design

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 JNI²⁸ 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 LeanXscale 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 outlier 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.

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* has 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 has been delivered that will allow the development of the *Reconfiguration Engine* during the second phase that will actually implement the business logic of the component, along with the integration with the WP3 components in order to allocate/de-allocate resources when needed.

Regarding the *Adaptable Storage Driver*, the following methods are indicative on what has been implemented and which enables the management of the data regions of the storage:

- **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 splitted 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:

```
appuser@a4888e23be10:/lx/LX-BIN/bin$ ./lxConsole
LeanXcale's console
The following values are accepted for the administration operations:
- [0] quit
- [1] help
- [2] components
- [3] running
- [4] deads
- [5] buckets
- [6] halt
- [7] loggers
- [8] trace
- [9] zkTree
- [10] zksts
- [11] updZkSts
- [12] markAsRecovered
- [13] csDiscardLtm
- [14] setLogger
- [15] listServers
- [16] listRegions
- [17] moveRegion
- [18] splitTable
- [19] splitTableUniform
- [20] startGraph
- [21] kiviSts
- [22] cgmState
- [23] persistKiviSts
- [24] getEpoch
- [25] replicateRegion
- [26] removeReplica
- [27] joinRegion
- [28] increaseTable
- [29] decreaseTable
- [30] listServer
- [31] cloneKVDS
- [32] setKvconPath
- [33] kvcon
- [34] printRecoveryStats
- [35] syncDS
- [36] getProperty
- [37] getOsts
- [38] getAlerts
- [39] kvmsEndpoint
lx>
```

Figure 24 - LeanXcale administration console

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 benchmark²⁹, 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, each one with specific characteristic: a first one is a long running read-only transaction that

²⁹

<http://www.tpc.org/tpcc/>

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 25, 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.

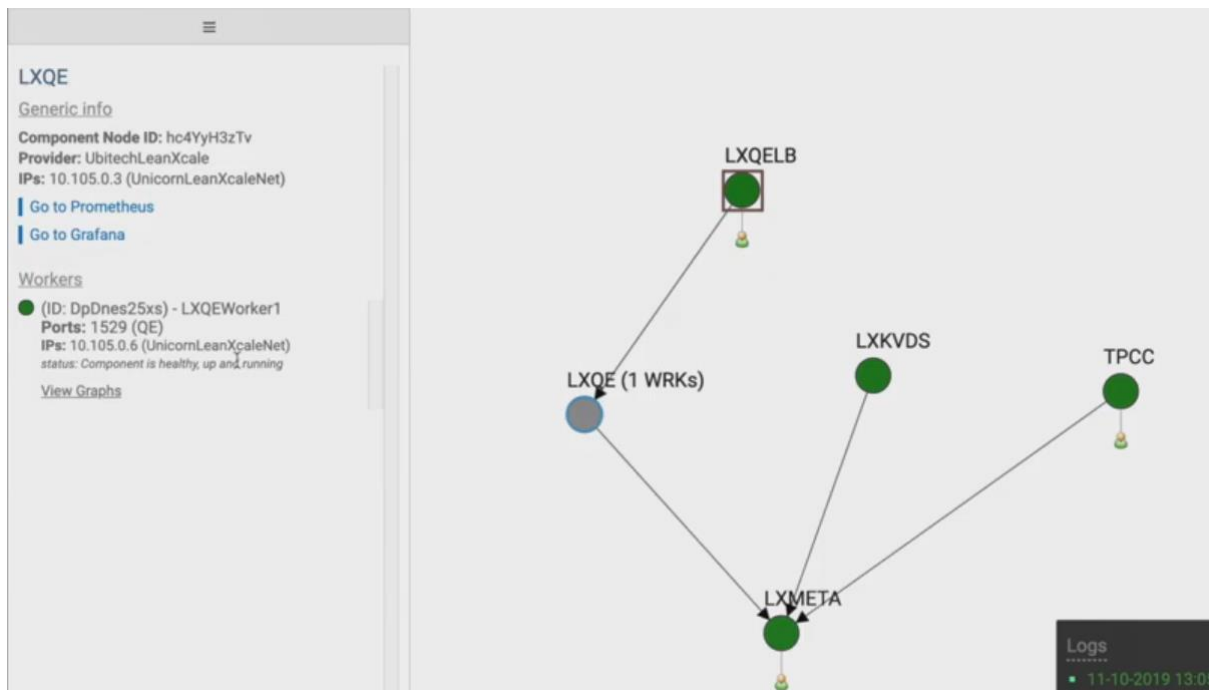


Figure 25 - Initial Deployment of the Adaptable Distributed Storage using 1 node

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.

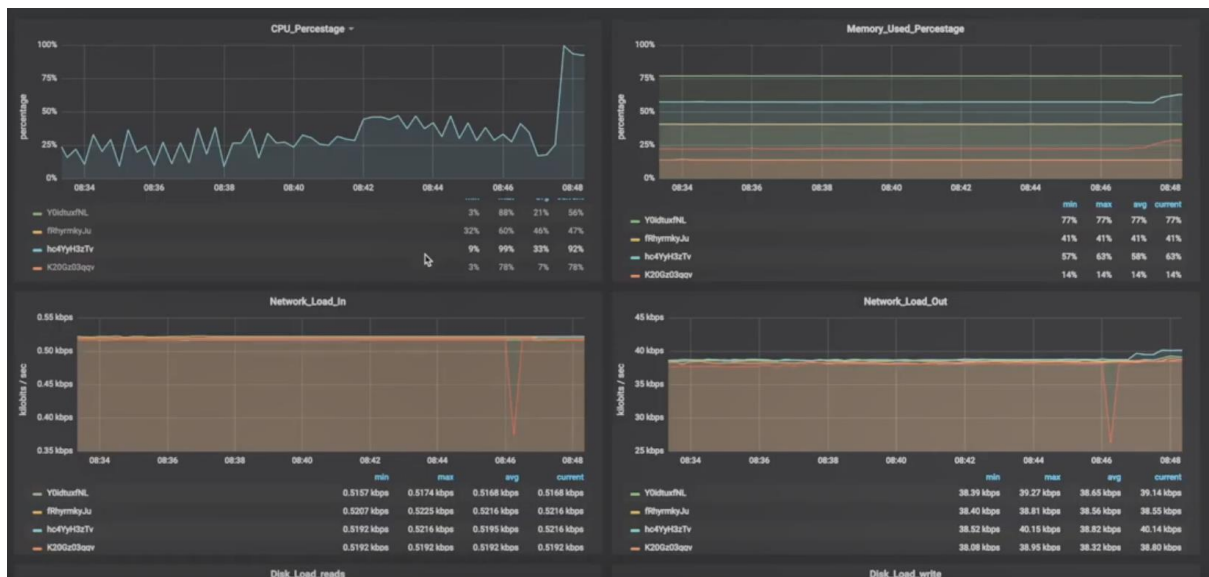


Figure 26- Monitoring information showing CPU is fully consumed

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.

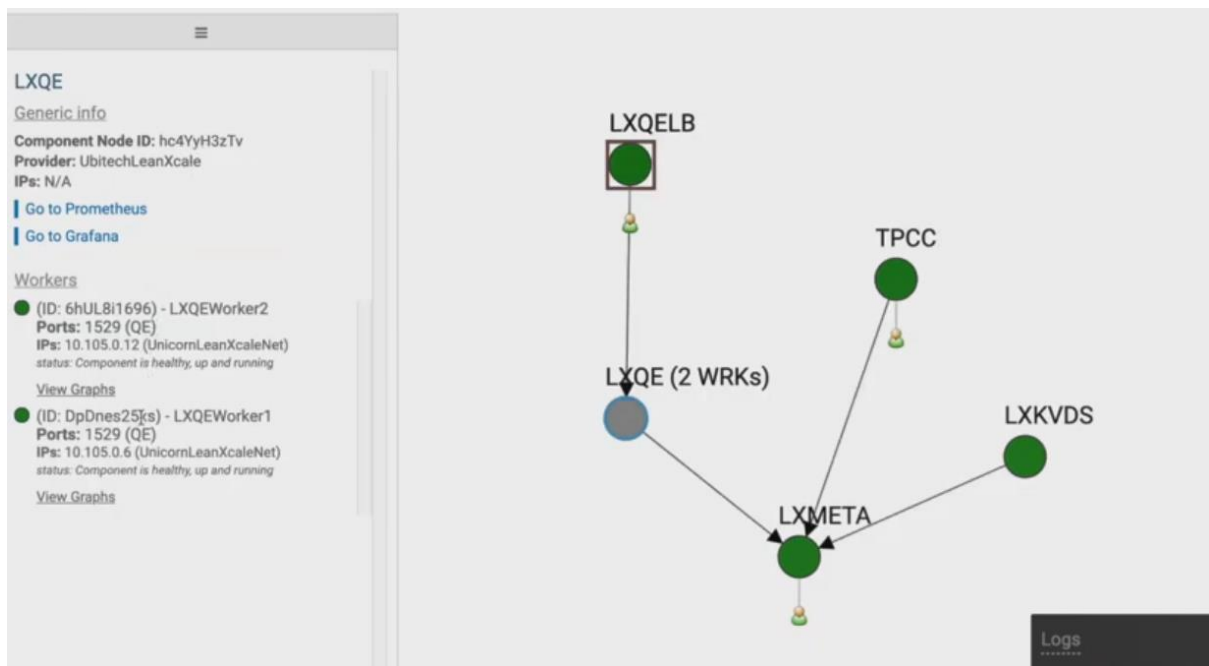


Figure 27 - Deployment after the manual scale-out action

The result of these actions is that load has finally been distributed and balanced among the two nodes, as it is shown in Figure 27.

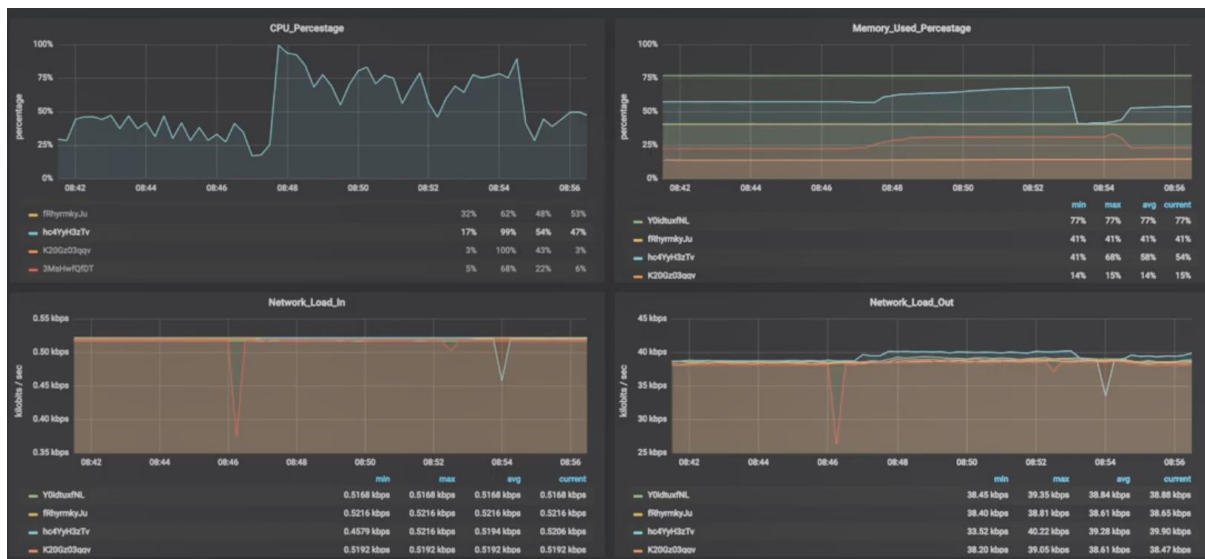


Figure 28 - Monitoring information showing CPU load balanced

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 6.2, 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. Next Steps

As it has been already mentioned in the previous subsection, the target objective for the first phase of the project regarding the Adaptable Distributed Storage was the implementation of the basic functionalities that will serve as the fundamental pillars and will enable the run-time adaptation of the storage according to its workload, both in terms of operations that need to be served per second and in terms of data volume. At M23, the initial version of the BigDataStack platform has been already delivered, thus enabling this component to be integrated with the platform in order to request for the allocation/de-allocation of resources from the infrastructure. As a result, the natural next steps for the implementation of the full version of this prototype can be summarized as two orthogonal aspects: the implementation of the automated scalability and the integration with the BigDataStack infrastructure. In more detail:

In the previous subsections we described and validated via experimentation the ability of the component to adapt by enabling scalability actions: It can distribute a data table into different regions, split/join/move regions to different data nodes, clone and remove the data nodes etc. Additionally, all these actions can be invoked at run-time, while the storage is under intensive operational workload, without incurring neither downtime nor

downgrading of its performance, while in parallel, it continues ensuring transactional semantics and data consistency. However, all these actions need to be invoked manually and require the database administrator to watch the monitoring mechanism in order for her to identify *hot spots* and to decide to intervene by invoking a scalability action from the administration console. The important next step will be to automate this process. For this, it will be required the implementation of the *Reconfiguration Engine* that will identify possible *hot spots* that require corrective actions, by exploiting the monitoring subcomponent, and then, after executing its internal heuristic algorithm to solve the non-linear problem, it will decide to re-configure the data nodes and redistribute the data load, by using the methods that have been already implemented and exposed by the underlying layer.

The second important next step that is planned to be delivered by the end of the BigDataStack project is the integration with the WP3 tools. Even if the *Reconfiguration Engine* can decide to re-deploy the data regions in order to re-distribute and balance the load, there might be cases where the overall resources that have been provided to the storage are not sufficient. In these cases, the intervention of the database administrator is still required in order to set up an additional storage server in order to scale out. However, at this phase of the project, this process has been automated via the tools provided by the WP3 component and can be used by the Adaptable Distributed Storage as well. As a result, we need to integrate this component with the WP3 tools in order for the storage to automatically request the allocation of additional resources from the infrastructure when it identifies that its own overall ones are not adequate to serve the incoming load.

As a conclusion, the preliminary validation of the prototype via the results gathered by the extensive experimentation are very promising, however not enough. At the end of the project when these two important next steps will have been implemented, then the Adaptable Distributed Storage can offer real elasticity capabilities, while ensuring transactional semantics and data consistency.

7. Seamless Data Analytics Framework

7.1. Requirements Specifications

The following tables show the requirements for the *Seamless Data Analytics Framework*, as redefined after the interim review (M19) and up to M23. They include improvements and additional functionalities that are scheduled to be implemented during the second phase of the project, along with updates and modifications of the requirements that were identified during the implementation of the initial prototype. Therefore, the requirements for this component are as follows:

Table 20 – 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 21 - 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 <i>freshness window</i> (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 <i>freshness window</i> 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 22 - 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 <i>federator</i> 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 <i>federator</i> and can be safely discarded by the <i>vacuum</i> process of the LeanXcale database.				

Table 23 - 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 24 - 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	<p>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.</p> <p>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.</p>				
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 25 - 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 26 - 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 LeanXscale.				

Table 27 - 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 LeanXscale 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 database 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 28 - 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 29 - 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.
--	--

7.2. 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 Store³⁰. The distribution of data across two datastores is problematic since: a) Data 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.

7.3. User Perspective

Figure 29 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.

³⁰ 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 COS may be replaced by any object store that presents an interface compatible with the S3 API such as CEPH or minio.

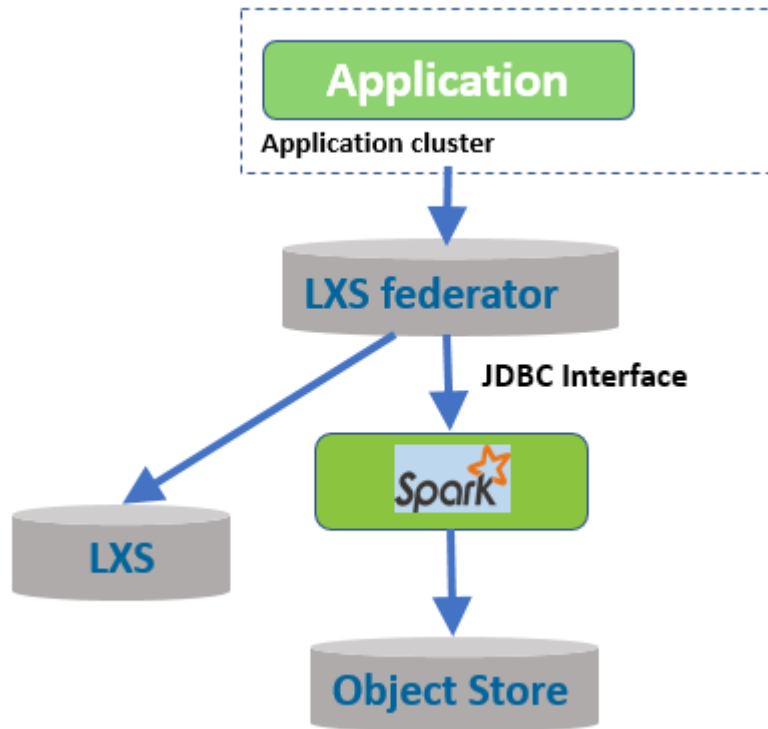


Figure 29 - Federation of the two data stores in the scope of the Seamless Data Analytical Framework

The federation between the two data stores is based on the LeanXscale internal Query Engine which has polyglot capabilities and can join data coming from different sources. Having LeanXscale Query Engine (QE in the following) as the *federator* of the framework, 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 LeanXscale 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 within can be stored within any of both of these stores. In order to retrieve data from the LeanXscale 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 LeanXscale 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 LeanXscale Query Engine can be distributed across various nodes and uses a random instance to coordinate the distributed execution. In the case of the IBM object store, the latter is integrated with Spark, which can be used on top to provide additional query capabilities. Moreover, Spark provides a JDBC interface to enable data connectivity. Thus, by exploiting the polyglot capabilities of the LeanXscale data base, its Query Engine can be used

to push down operations both to the LeanXscale 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 LeanXscale.

Figure 30 shows the pipeline designed to move historical data from the LeanXscale relational data store to the IBM object store. The LeanXscale data base periodically is informed to get ready to export a dump of the latest updates, which now fall outside the freshness window (depicted in the diagram as cold data slices). Data are pulled by the *Data Mover* component, injects this data into the object store. As a result, eventually the data exported by the LeanXscale database is now available in the latter. After the successful ingestion of the historical data, the LeanXscale 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.

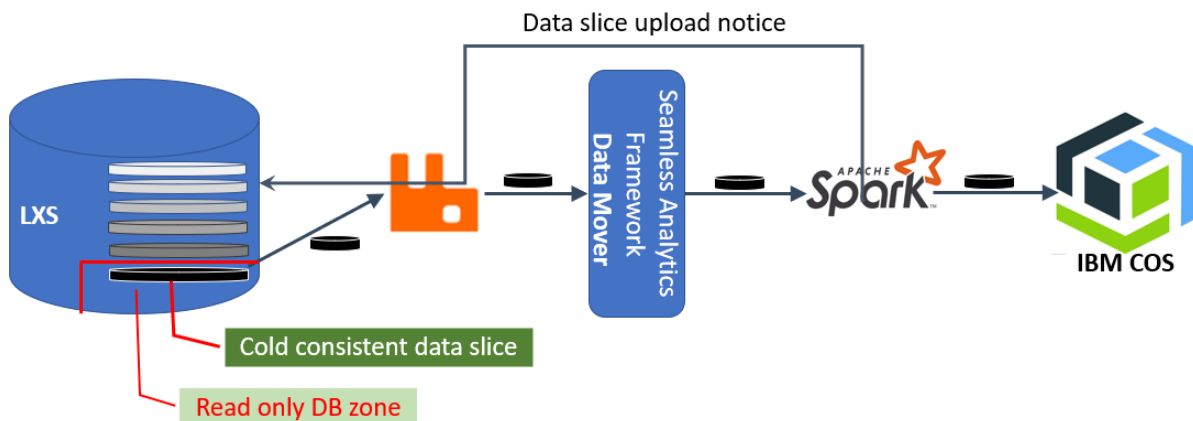


Figure 30 - LeanXscale data base to IBM Object Storage pipeline for historical data

7.4. Detailed Design

The main architectural components of the Seamless Analytical Framework appear in Figure 31 which also shows the sequence of interactions, when a data slice is moved:

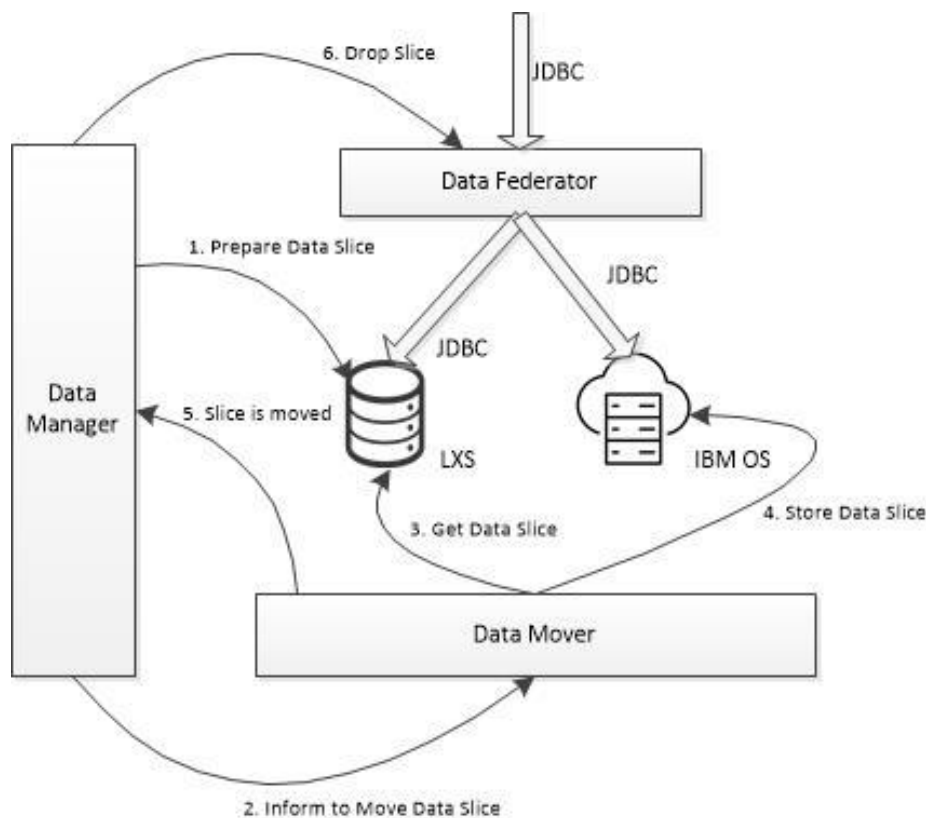


Figure 31 - Interactions among SAF components

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 32 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 be noted 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:136209240000  
0: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 catalog 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 be 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. 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 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, taken into account the specific supported dialects of each one (i.e. the syntax of the LIMIT operation in LeanXscale 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 LeanXscale 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. Currently, the supported operations are 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 summarizes 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.

As it can be noticed, the JOIN operation has not been implemented yet and it is under design at this phase of the project with the plan to be delivered at the final version of the prototype.

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 catalog. This catalog can either be federated or unified. We use Apache Hive metastore mainly for having a consistent (federated) catalog.

Federated catalog - since the OS has its own catalog (when using Apache Hive metastore) and LeanXcale has its own catalog, one possibility is to maintain the “unified” catalog using both by processing the TABLE CREATE DDLs through the seamless API and updating both catalogs each one with the relevant information.

Maintaining federated yet consistent catalogs 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_statement": "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 32 - 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 33 - Data Mover notifies the Manager that the movement succeeded

7.5. 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 of 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 of rows in each federated datastore, and that this changed when a data movement takes place, 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 MaxWindSpeed,  
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.

7.6. 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 it 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 IBM object store. Regarding the latter, the ship management use case, as described in the previous section, will mainly be used to validate its functionality. This use case produces IoT data coming from various sensors deployed on its fleet, which are ingested in the store as a per-minute basis. This information can be considered historical after a certain point in time, and will be transferred to the object store, through this pipeline.

Regarding the *federator*, as the main access point to the data stored in the platform, it will be used by all 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. When it comes to update operations, the *federator* will push down these operations directly to the LeanXcale relational data store that ensures transactional semantics. When it comes to read operations, it will push down the queries both the LeanXcale data store and IBM object Store.

7.7. 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.

7.8. 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 the majority of all 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 executes 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.

Last but not least, the initial design of the Seamless Analytical Framework was performed based on the integral assumption that historical data can never be updated, and therefore, once they are moved to the object store, no further modifications are accepted. This assumption comes from real life scenarios, where a data warehouse gets a snapshot of the data stored in an operational datastore via ETLs, and the data are never subject to further modifications. However, to truly reveal the potential of putting an operational datastore in front of a data warehouse using our solution, thus extending the analytical database with truly operational behaviour and vice versa, this assumption should not stand. As a result, a *nice to have* feature will be the ability to additionally allow the update of records stored in the object store, by moving a specific data slice back to the operational datastore that can ensure transactional semantics when a data modification operation takes place.

8. Data Quality Assessment & Improvement

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 specific 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 given dataset.

8.1. 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 30 - 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 is able to ingest random typos, format errors and value swaps and is an optional component in the DQAI pipeline.				

Table 31 - 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 32 - 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 33 - 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 34 - 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 35 - Requirement REQ-DQAI-06 for Data Quality Assessment & Improvement

8.2. Design

8.2.1. Approach

Our approach uses Deep Neural Networks (DNNs) to learn an abstract feature representation of the elements in a data set, as well as how they relate to each other. By exploiting these rich, high dimensional embeddings, we train a neural network to compute the probability of an element being "dirty" or, in our terminology, invalid. To intuitively understand the methodology, section 8.2.2 presents a toy example of capturing simple

relational information with a neural network and then, we extend this idea to fit our needs and introduce our approach.

8.2.2. A simple example of capturing relational information

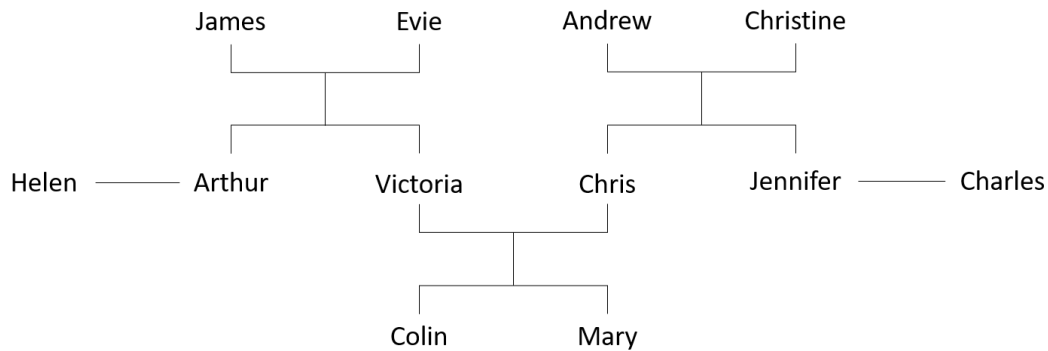


Figure 34 - Family tree

To instinctively understand how neural networks can capture relational information, we will start with a simple example from the 1980s. Geoffrey Hinton experimented with a simple real-life analogy to demonstrate how we can use the back-propagation algorithm to turn relational information into feature vectors³¹. Those feature vectors can capture semantic and latent information that describe the elements in a data set and their underlying associations.

Figure 34 depicts a simple family tree of three generations. For instance, in this tree James and Evie marry, and have two children; Arthur and Victoria. Arthur then marries Helen and have no children while Victoria marries Chris and have two children of their own; Colin and Mary. The goal is to train a neural network that can understand the information in this family tree. We also have a similar tree, with the same structure, where the only difference is that the names are in Greek.

To encode the information presented in the family trees, we first create a set R of propositions that express the relationships that are seen. Thus, we let $R = \{son, daughter, nephew, niece, father, mother, uncle, aunt, brother, sister, husband, wife\}$. Then, to construct a training set from this tree we generate a set triples T , such that $T = \{James - wife - Evie, James - son - Arthur, Arthur - mother - Evie, \dots\}$. These triplets can be read as *James has wife Evie*, *James has son Arthur* etc. As we can see, at least in this simple example, the third triple follows from the first two³². Hence, the learning task of mastering the information given in these trees can be viewed as figuring out the regularities that exist in the generated triples.

The easiest way to express the regularities in these triples would be the use of symbolic rules:

³¹ Geoffrey E Hinton et al.1986. Learning distributed representations of concepts. In Proceedings of the eighth annual conference of the cognitive science society, Vol. 1. Amherst, MA, 12.

³² Assuming that, at least in this simple world, no man should marry two women and so on.

$$x \text{ haswife } y \ \& \ x \text{ hasson } z \Rightarrow z \text{ hasmother } y$$

But given a large enough family tree, the search for such rules would be prohibitive as we would have to search a combinatorially large space of discrete possibilities. Thus, we transform the problem using a neural network that can capture the same information by searching through a continuous space of weights.

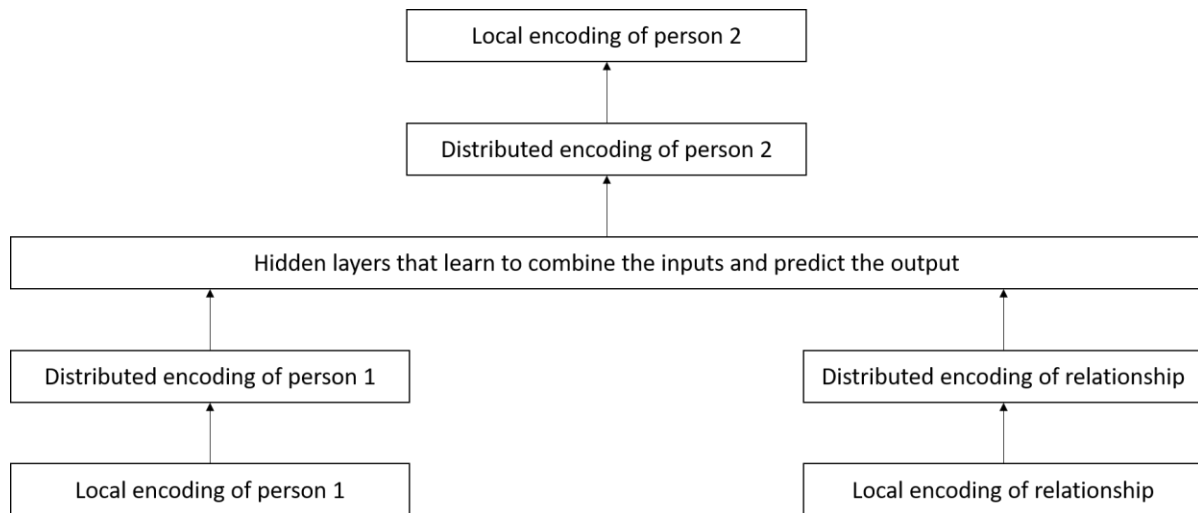


Figure 35 - Encoding relational information using NN

Figure 35 depicts a neural network architecture, that tries to predict the third entity of a triple given the first two entities. In the example we have, there are 24 possible people (12 names in English and 12 names in Greek). To produce the local encoding for each of those people in a neutral way, that is we don't accidentally give the network any similarities between them, we use the one-hot encoding technique; we create a 24 dimensional vector, in which every element is equal to 0 except the element that refers to the person we want to encode. We set that vector element equal to 1. As follows, the bottom layer of the neural network in the figure, has two inputs; the *Local encoding of person 1* that has 24 neurons and the *Local encoding of relationship* that has 12 neurons (the cardinality of set R). The output layer of the network is also a 24-dimensional vector that represents the *Local encoding of person 2*.

The interesting work happens in the second from bottom layers, namely the *distributed encoding* layers. These are, as we call them, bottleneck layers, or in simple terms they consist of much less neurons than the layers before them. For example, because there are 24 persons a *distributed encoding* layer with half the neurons cannot dedicate one neuron for each person. It is forced represent each person in a lower dimensional space. The idea is that when the network is trained, the bottleneck layer's activations for each person will encode semantic, higher-level features of the attributes we care about. For instance, one neuron could represent the nationality of a person while another could represent the generation or the branch in the family tree. But the intention is for the network to discover all the information implicitly via back-propagation and not instruct it which features to use. Similarly, bottleneck layers are all over this architecture, encoding the relationship, and person 2 in the layer before the output. Consequently, the middle hidden layers learn to

combine the features that encode person 1 and the relationship in such a way that the model can predict person 2 correctly.

This research was done in the 1980s and it was a way to demonstrate that back-propagation can be used to automatically learn interesting features. Today, we have data sets of millions of relational facts of the form $(A \ R \ B)$ (i.e., identity A relates to identity B in some way). In the next section we propose a way to extend and use this idea to detect errors in such data set.

8.2.3. Data Quality Assessment

In this section we present how we adapted the idea of extracting relational information into embedding matrices, to train an error detection model. The first challenge we address is the disproportionate ratio of observations for each class, as only a few corrupted examples (i.e., errors) exist for each data set. The second challenge is to identify pairs of attributes that relate to each other, in a domain-agnostic way. This leads to the implementation of a generic methodology that can produce an augmented training set of triples (i.e., attribute pairs and their relationship). This set is fed to a network architecture similar to that shown in Figure 35, which we have slightly modified to better fit our problem.

Address	Phone Number
1720 University St	205 3258 100
1300 South Montgomery Av	256 3864 556
1108 Ross Clark Circle	334 7938 701
1720 University St	256 5938 310
1300 South Montgomery Av	256 7688 400
...	...

Figure 36 - Cramer's V table

Data processing. To attend to the first challenge of imbalanced classes, we created a data augmentation mechanism, that introduces common mistakes according to a probability distribution. Those mistakes can be spelling mistakes, format errors or random value permutations. As we assume that most of the data set is error-free, an example of a format error would be writing *NY* as *New York*, when most of the time we come across this item it appears as *NY*. Moreover, we find random permutations to be especially demanding. For instance, if we randomly switch the state that relates tightly to an address (i.e. street number and zip code), with another state drawn from the same data set, we create an invalid triple that is hard to spot.

The next step is to identify which pairs of attributes relate to each other and create a training set of triples. Not all attributes have information that our model needs to learn. There are also attributes that do not follow any underlying distribution and can harm our model's predictions, for example a primary key column. To extract the attributes that provide useful insights for our challenge and do that in a domain-agnostic fashion (i.e. no human involvement), we turn to attribute correlation.

In most of our experiments we dealt with categorical attributes. Furthermore, when we encounter a numerical attribute, we bucketize it and treat it as categorical. Thus, we turn into ways of calculating the correlation between two categorical attributes. Instead of transforming every categorical value into numbers or one-hot vectors, we looked at specifically defined metrics. The first we consider is the *Cramer's V* metric, that measures association between two categorical variables and is defined below, where n is the grand total of observations, c is the number of columns, r is the number of rows and χ^2 is the chi square statistic.

$$V = \sqrt{\frac{\chi^2/n}{\min(c-1, r-1)}}$$

Although *Cramer's V* gives an easy way of defining correlation between two categorical variables, it is symmetrical, meaning it is insensitive to swapping x and y . In some cases, this leads to losing important information in a data set. Consider the table presented in Figure 36. We could easily infer the *Address* attribute value just by knowing the value of the *Phone Number*; the opposite is not true. This is because a building can have many phone numbers, but a phone number is associated with only one address. Hence, we found that the *Uncertainty coefficient* or *Theil's U* worked better as a measure of nominal association. *Theil's U* is based on the conditional entropy between x and y and unlike *Cramer's V* is not symmetric; $U(x, y) \neq U(y, x)$. Given two random variables X and Y we can calculate the entropy of a single distribution as: $H(X) = -\sum_n P_X(x) \log P_X(x)$. The conditional entropy is given as: $H(X|Y) = -\sum_{x,y} P_{X,Y}(x,y) \log P_{X|Y}(x|y)$. The *Uncertainty coefficient* then is defined below:

$$U(X|Y) = \frac{H(X) - H(X|Y)}{H(X)}$$

The last step is to create the training data set of triples that will be fed to the neural network. Having the top- k correlated features, we construct triples drawn from the augmented data set, as pairs of attributes x and y , that are highly correlated, and their relationship. We treat the number k as a hyper-parameter of our model. This data set takes the form of $\{(x \text{ relates-to } y)\}$ and the corresponding label.

Network Architecture. The network architecture is depicted in Figure 37. It is similar to the network architecture presented in the toy example in section 8.2.2, but slightly altered to better address our needs.

The network takes three inputs, the two attributes of a triple and their relationship. The two attributes are embedded into the same high-dimensional space, while their relationship has a separate distributed representation. The outputs of these embedding layers are concatenated in later hidden layers and combined to predict the output.

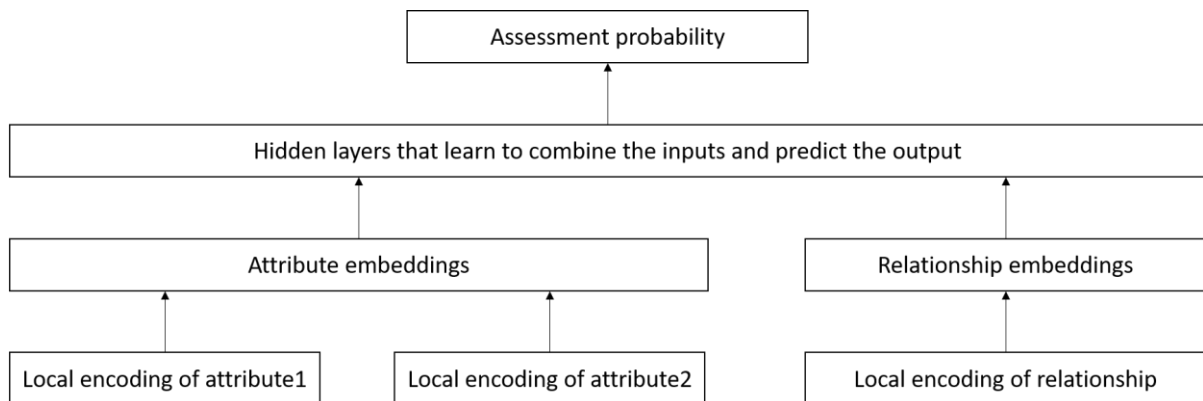


Figure 37 - DQA network architecture

The output prediction of the network tries to define a probability value for the triple's validity, designed to express the likelihood that those two attributes can co-exist. Thus, instead of predicting the second entity of the triple given the first and a relationship proposition, it takes the full triple as input and scores it according to its correctness.

8.3. Prototype

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

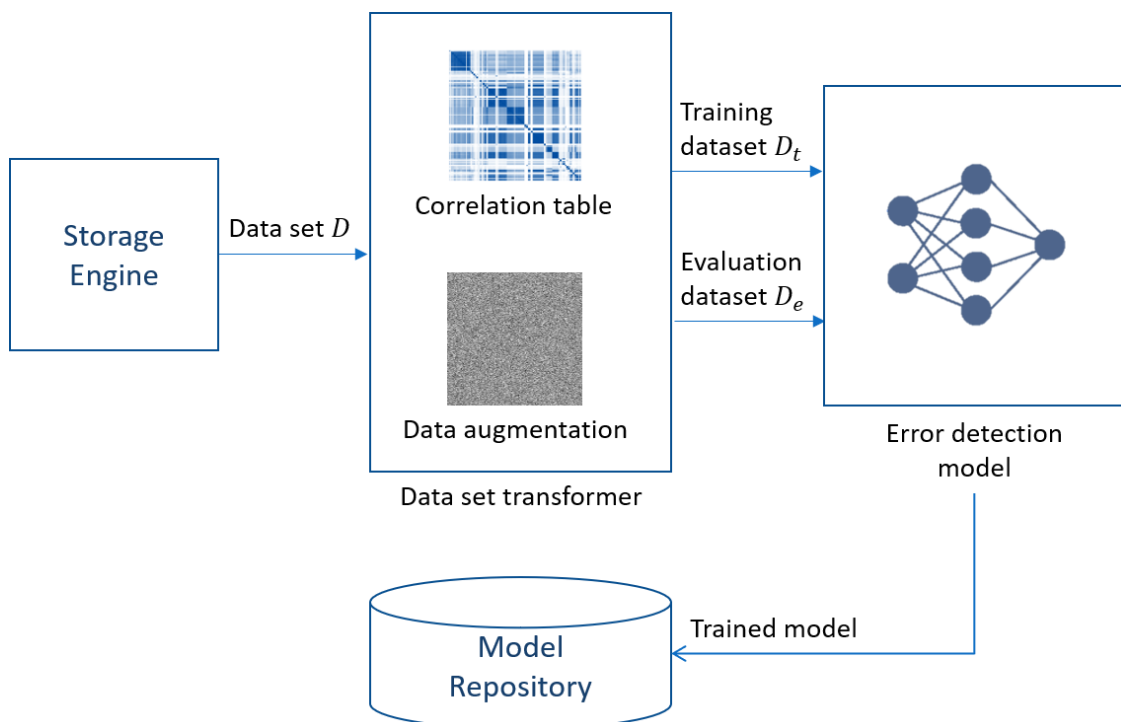


Figure 38 - Training phase

Figure 38 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 39, 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.

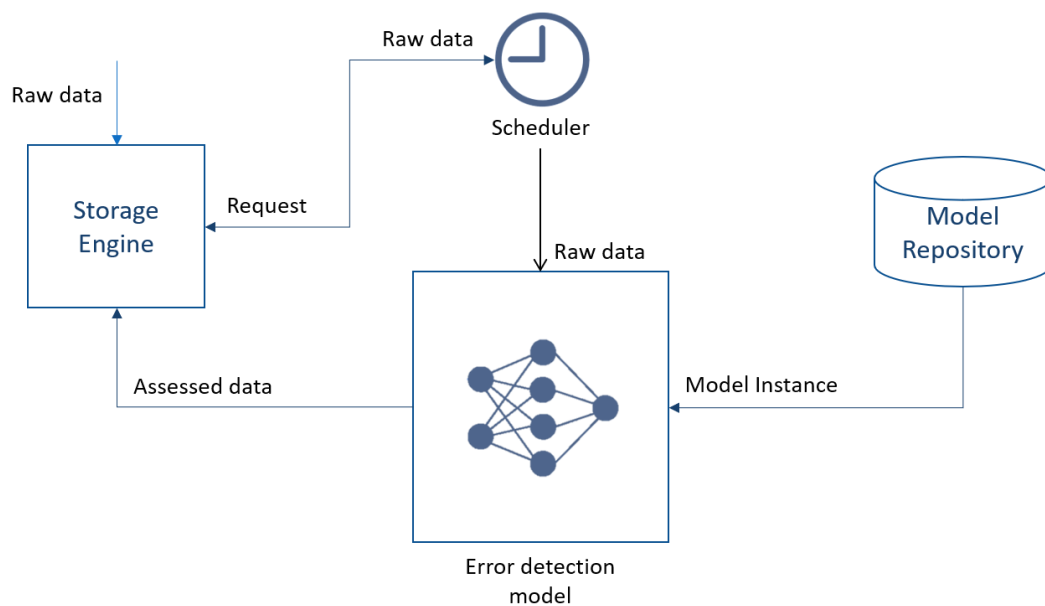


Figure 39 – Assessment phase

8.4. Use Case Mapping

In the scope of the Data as a Service capability of BigDataStack, the ship management and connected consumer use cases will be used as the two main demonstrators to highlight the functionalities of the Data Assessment and Improvement module. The ship fleet produces IoT data from numerous sensors deployed on each of its vessels, while in the connected consumer use case, a recommender system for a retail customer is being built.

In the case of the ship management use case, the data is pre-processed by different installations of CEP subsystems, which contain a simple, rule-based cleaning procedure (e.g., removing null values), and then, the Data Assessment and Improvement module can act as a second layer, automated and domain-agnostic data quality estimation. This

operation could discover anomalies in data, allowing users to act and pinpoint deterioration in the functionality of different IoT sensors. Also, as we demonstrated in the interim review. The results of the data quality assessment component are crucial for the predictive maintenance algorithm, increasing its accuracy.

In the case of the connected consumer use case scenario, the data analyst could discover anomalies or errors in the datasets used to train the recommendation system model(s), which could affect performance and accuracy.

8.5. Experimental Plan

We compare our approach against several, well established error detection methods, on three heavily used benchmark data sets. 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), and what our augmentation methodology offers in our attempt to alleviate the problem of class imbalance.

Experimental setup

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.

Data sets

Table 36 - Datasets

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

Two data sets from diverse areas of domains were used to assess our methodology. Table 36 presents a summary of each data set and their attributes. As shown the data sets vary significantly in size and number of attributes.

- The hospital data set is considered to be 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.

Baselines

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:

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

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 37, 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 37 - 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 38 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 section 8.4.

Table 38 - Precision, Recall and F1 score for DQA and benchmark solutions

Dataset	M	DQA	AUG	LR	CV	OD	FBI
Hospital	<i>P</i>	0.953	0.903	0.0	0.030	0.640	0.008
	<i>R</i>	1.000	0.989	0.0	0.372	0.667	0.001
	<i>F₁</i>	0.976	0.944	0.0	0.055	0.653	0.003
Ship management	<i>P</i>	0.870	n/a	n/a	n/a	n/a	n/a
	<i>R</i>	1.000	n/a	n/a	n/a	n/a	n/a
	<i>F₁</i>	0.931	n/a	n/a	n/a	n/a	n/a

8.6. Next Steps

During the next stages of the project, we will see to the integration and applicability of the data quality assessment and component with the other two use cases scenarios of BigDataStack.

9. Predictive & Process Analytics

This section describes the BigDataStack components that implement the functionality of Predictive and Process Analytics, namely:

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 40).

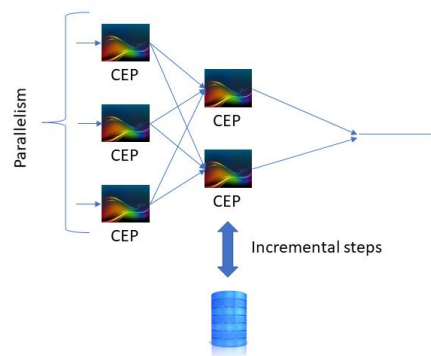


Figure 40 - Parallelization and distribution of analytics workflow

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 41 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.

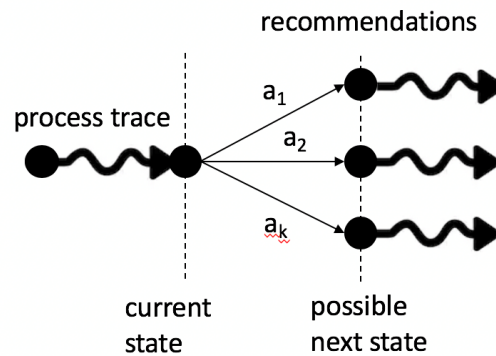


Figure 41 - The recommendation of next possible state during process modelling

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.

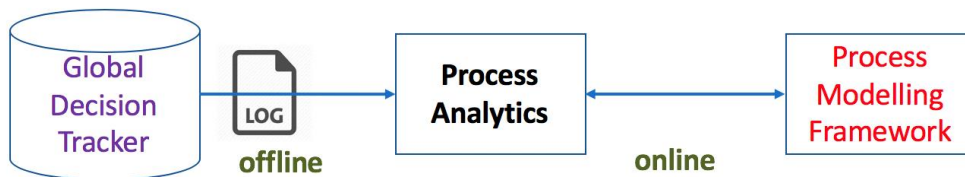


Figure 42 - The interaction between Process Analytics and Global Decision Tracker and Process Modelling Framework

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 42.

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 39 - 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 40 - 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 41 - 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 ³³ (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 is included in this standard.				

Table 42- 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				

³³ <http://www.xes-standard.org/>

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 43 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.

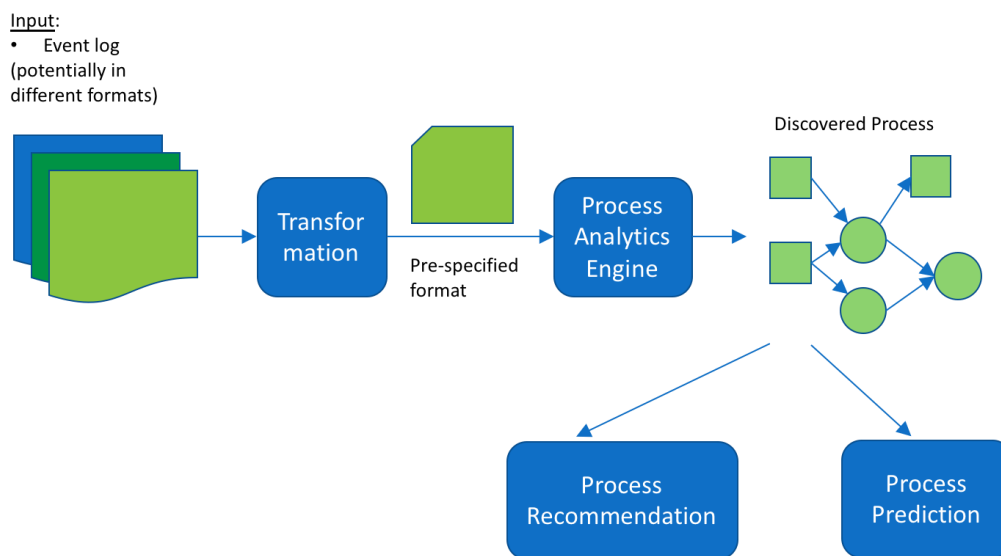


Figure 43 - Design of Process Analytics component

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 M23), 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 $n \times n$ 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*³⁴ which is an efficient way to extract communities from large graphs. The method follows a greedy optimization approach with observed run in time $O(n \log n)$. 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:

³⁴ Vincent D. Blondel, Jean-Loup Guillaume, Renaud Lambiotte, Etienne Lefebvre: Fast unfolding of community hierarchies in large networks. CoRR abs/0803.0476 (2008).

- 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.

9.4. Use Case Mapping

In principle, 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.

In consideration with the specific use-cases of BigDataStack (real-time ship management, connected consumer, and smart insurance), Process Analytics is a good match for the connected consumer use-case, since it would allow the analysis of event logs related to product purchases, in order to discover subsets of products that are frequently bought together by many customers, so as to derive product recommendations for customers. At the time of this writing, we turn our attention to this use-case, and present in our experimental evaluation how Process Analytics can be applied on retail data.

9.5. Experimental Evaluation

In this section, we report results obtained from the working prototype of Process Analytics. Due to the lack of real event log data obtained from BigDataStack that can be exploited by our prototype, we use 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.

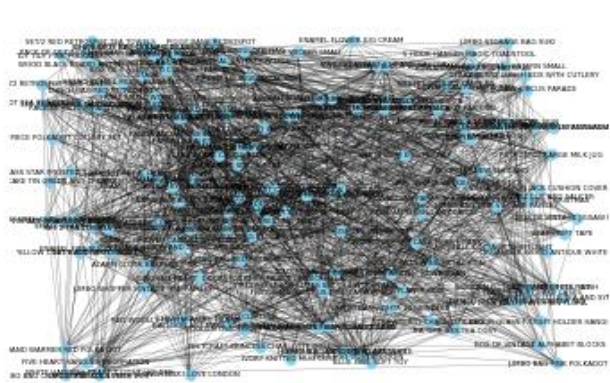


Figure 44 - Graph produced from the analysis of the Online Retail dataset.

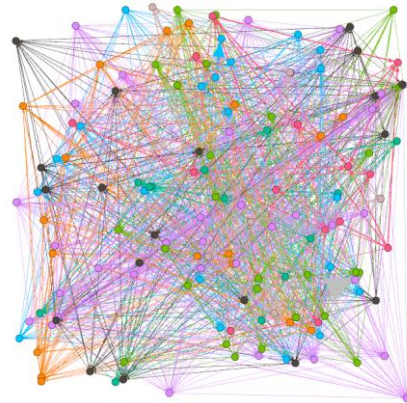


Figure 45 - The same graph colored for better visualization.

Figure 44 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 45 depicts the same graph with colored nodes, simply for improved visualization.

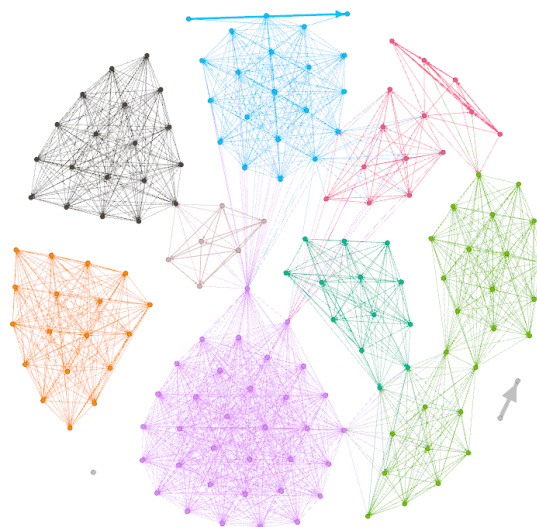


Figure 46 - The result of the community detection algorithm when applied on the afore-described graph (colors represent different communities).

Figure 46 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.

9.6. Next Steps

In summary, the Process Analytics component has been designed to operate on event logs and the current implementation of the working prototype allows the discovery of process models. This is performed in two ways: (a) by building a weighted graph, coupled with a community detection algorithm for narrowing down the search space, and (b) by means of Process Mining algorithms (*Alpha Miner*, *Heuristic Miner*, and *Inductive Miner*). Eventually, the Process Analytics component is able to perform the following task: given a process trace, recommend the next step (or the next top-k steps) to the Process Modeller.

The roadmap for the third year of the project is as follows:

- First, we aim to assess the discovered processes from the event logs, also in the presence of noisy data. This is an ongoing activity which is based on the use of PLG2 that allows the generation of noisy data in a controlled way.
- Second, we aim to evaluate the recommendations produced by the Process Analytics component, in order to quantify the gain obtained.
- In parallel with these activities, system integration is going to be performed, in order for the component to interact with the Process Modelling framework, and also to obtain event log data from the infrastructure of BigDataStack.
- Finally, depending on the availability of sufficiently large event logs produced from BigDataStack applications, we will demonstrate how Process Analytics can be applied in an end-to-end scenario.

10. Real-time Complex Event Processing

10.1. Requirements Specification

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) or 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.

Table 43 - 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 analyse 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 44 - 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 45 - Requirement REQ-CEP-03 for CEP

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

	03				
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 46 - Requirement REQ-CEP-04 for CEP

	Id	Level detail	of 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.				

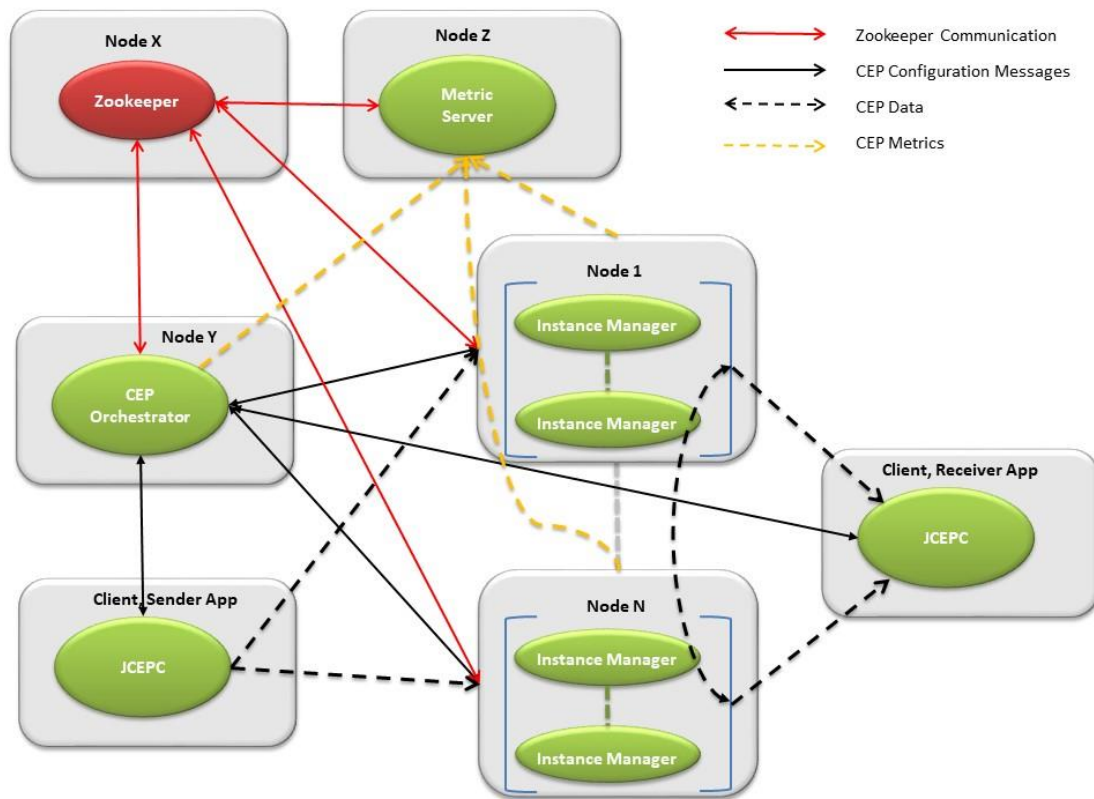
10.2. Design

Figure 47 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.



n

Figure 47 - CEP Components

10.3. Current Prototype

At M23 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. Raspberry Pis are used to gather, process and aggregate events in remote locations so that, actions can be triggered as soon as possible. The results of this processing are sent to a data center for aggregating the data coming from several remote locations. The initial performance evaluation using Raspberry Pis shows that even with that cheap and small hardware the CEP is able to process thousands of events per second in a single small node.

10.4. Use Case Mapping

In the scope of WP4, the ship management use case will be 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 if the average fuel consumption during the last half an hour was higher than a given value). The data will also be 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.

10.5. Experimental Evaluation

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

- Performance evaluation with stateless queries. Thousands of events are generated and processed per second.
- Performance evaluation with stateful queries over windows of time with thousands of events per second.
- Running the CEP on hardware with different resources.
- Live migration of queries without stopping the query processing.

The benchmarking of the CEP will use ship management data provided and the HiBench³⁵ benchmark as well as micro-benchmarks with synthetic data.

10.6. Next Steps

The next planned activities for CEP consist at completing the implementation of a full functional CEP running on local cluster and executing a more through performance evaluation. In parallel, the implementation of the CEP capabilities for running federated queries on WAN environment with heterogeneous processing nodes will start. The deployment of queries will take into account the hardware features in order to improve performance of queries and avoid whenever possible the communication overhead in these environments.

10.7. Initial Performance Evaluation

In this initial evaluation two aspects of the CEP have been evaluated. On the one hand, the performance of the CEP has been evaluated both in a single Raspberry Pi and in a distributed system using the HiBench query. On the other hand, the cost of migrating queries online has been measured in the data center.

The purpose of the first evaluation was to stress the CEP both in small devices and in the data center and see whether it can run in small environments where thousands of events per second are received and its processing capabilities in a data center.

The second evaluation goal is needed for live migration of queries that is, moving queries from one node to another one without stopping the CEP. This migration is needed in the advent of burst loads that may saturate a single node and when new nodes are provisioned to the system. This initial evaluation has been conducted in AMD computers.

10.7.1. HiBench Benchmark.

³⁵ <https://github.com/Intel-bigdata/HiBench/blob/master/docs/run-streamingbench.md>

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 48). 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, it at least one tuple for each IP address is received. We will use the maximum load that the system can handle as performance metric. 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.

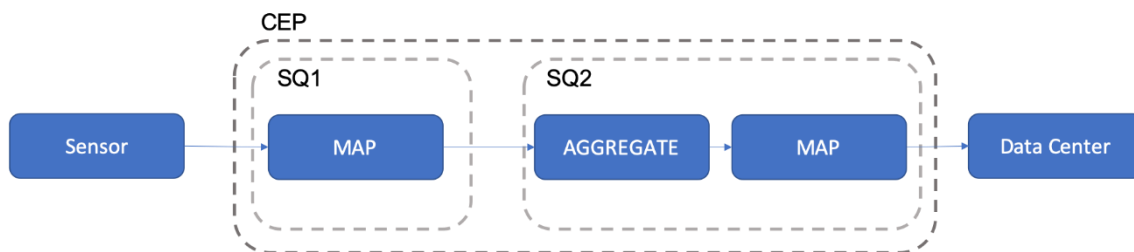


Figure 48 - HiBench query and subqueries

10.7.2. Performance Evaluation of the CEP in Small Devices

The performance of the CEP has been evaluated using the FixedWindow query in one Raspberry Pi 3 Model B, equipped with a Cortex-A53 processor with four cores running at 1.4GHz with a L1 cache and a shared L2 cache, 1 GB RAM and 100Mbit Base Ethernet. The OS running at Raspberry Pi is Raspbian OS 8 (Jessie).

We have tested several configurations in order to achieve the maximum throughput. First, we ran the query in a single core with a fourth of the available memory (200 MB), then we tested the query using two instance managers and then running SQ1 in a Raspberry Pi and SQ2 in an Intel i7. The Intel i7 is an Intel(R) Core(TM) i7-6700 CPU at 3.40GHz, 32GB RAM, Ethernet interface 100Mb/s and a direct attached SSD disk of 256GB. The OS running is Ubuntu 16.04.6 LTS. This case represents a setting where part of the query (SQ1) is runs at the edge (in a Raspberry Pi) and the other part of the query (SQ2) runs in a data center. For now, this experiment is run in a local area network (LAN).

Figure 49 shows the load the CEP handles running the HiBench query in one IM with 200 MB RAM in a single core of the Raspberry Pi. The maximum load the CEP can handle is up to 11,000 tuples per second. The first map becomes a bottleneck with 12000 tuples and is not

able to process at that rate. The right-hand side of the graph shows the CPU consumption, which reaches 80% with the maximum load.

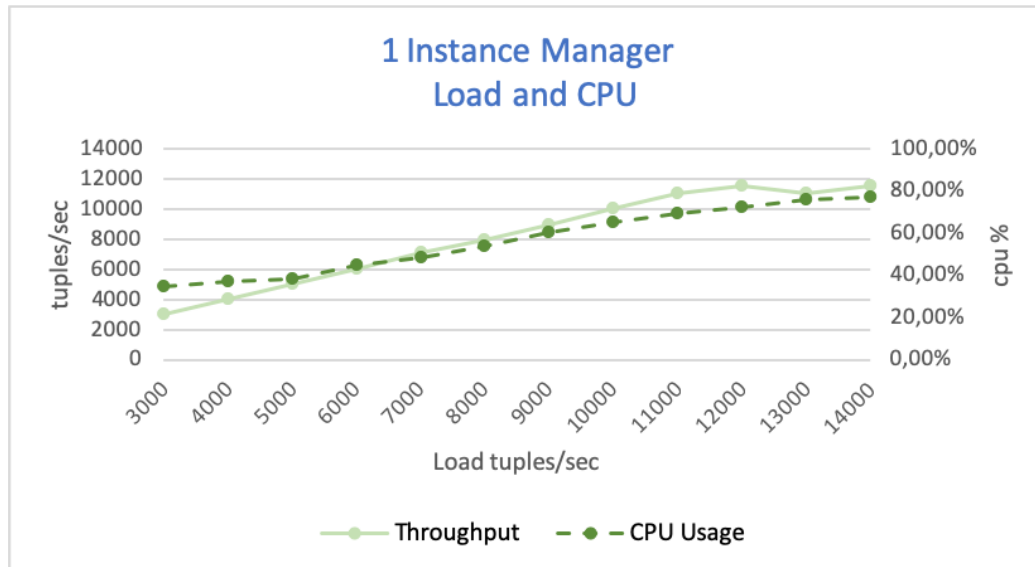


Figure 49 - CEP Performance for one core and 200 MB memory

When we deploy two instance managers in the Raspberry Pi, the query is divided and the instance manager are running SQ1 and SQ2, respectively. The CEP is able to process double the load as compared with the previous configuration where a single instance manager running the full query (Figure 50). The reason for doubling the load the CEP can handle has to do with the use of two instance managers (IM). These components are single threaded and as soon as one operator is saturated, it blocks the rest of the operators. When two IMs are used, there is asynchronous communication among them and they can process at their own pace.

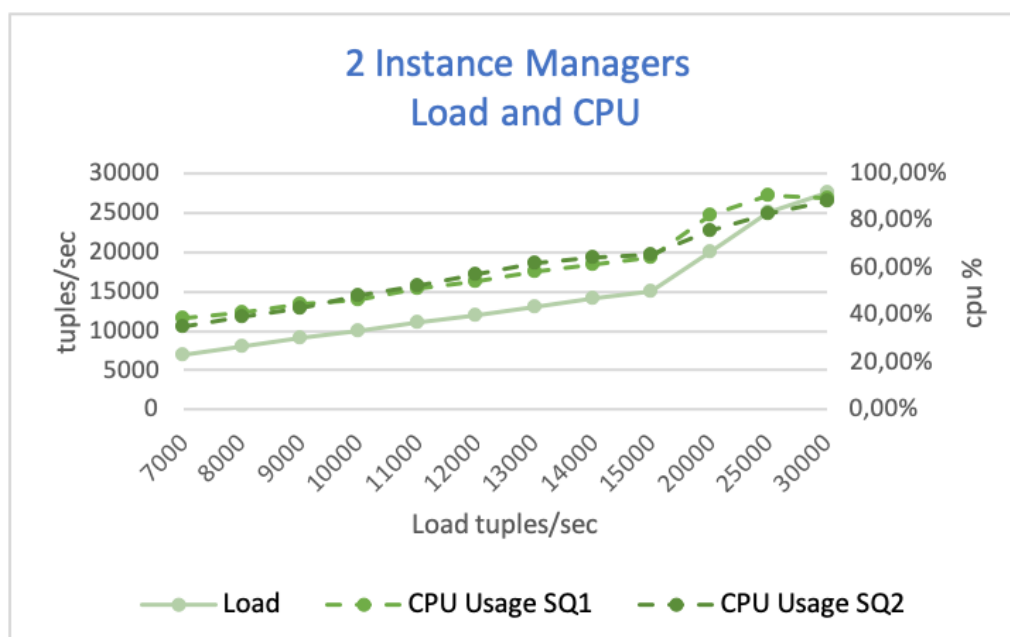


Figure 50 - Two IMs running one subquery in a Raspberry Pi

The next configuration runs SQ1 at the Raspberry Pi and SQ2 at an Intel i7. Figure 51 shows the performance when deploying one and two SQ1 instances at the Raspberry Pi.

With one SQ1 instance the system is able to process around 28K tuples/sec and with two SQ1 instances the double of load is processed by the IMs running in a single Raspberry Pi, being able to support around 60K tuples/sec.

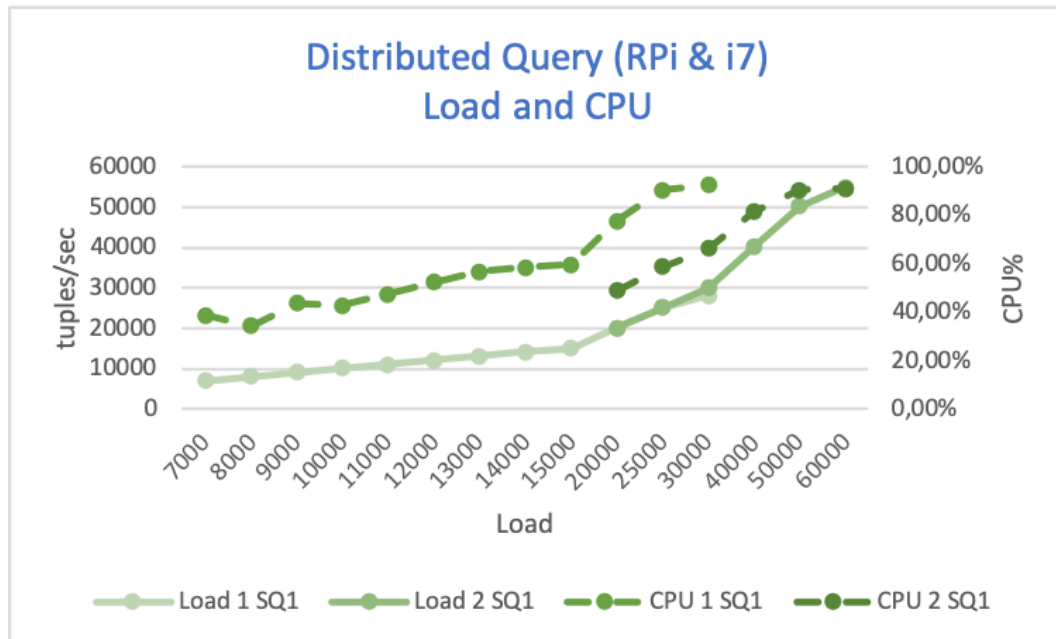


Figure 51 - Performance running a distributed query between Raspberry Pi and i7

In the next experiment the time taken to process a given load sent at different rates in a Raspberry Pi is measured. Figure 52 shows the time needed to process the submitted load with different configurations in which SQ1 and SQ2 run in the Raspberry Pi (1 SQ and 1 RPi), only SQ1 runs on the Raspberry Pi and finally with 2 SQ1 (1 SQ1 RPi) in the Raspberry Pi (2SQ1 RPi). And SQ2 in the i7. The first scenario is the one where SQ1 and SQ2 are running at Raspberry Pi, injecting 7,2 million and 9 million tuples the system is able to process the load in 360 seconds. When 9,9 million tuples are sent, it takes 370 seconds to process all the load. When the SQ2 is deployed at the Intel i7 machine and the SQ1 remains in the Raspberry Pi the system is able to process the 9,9 million tuples in 360 seconds. This time increases by 17 seconds when 10,8 million tuples are injected. The third scenario deploys 2 SQ1 at the Raspberry Pi and SQ2 at the Intel i7 node, with this configuration the system is able to double the supported load, being able to process up to 18M tuples at a rate of 50K tuples/sec. When the rate is increased in 55K tuples/sec the system is no able to process the load in the expected time.

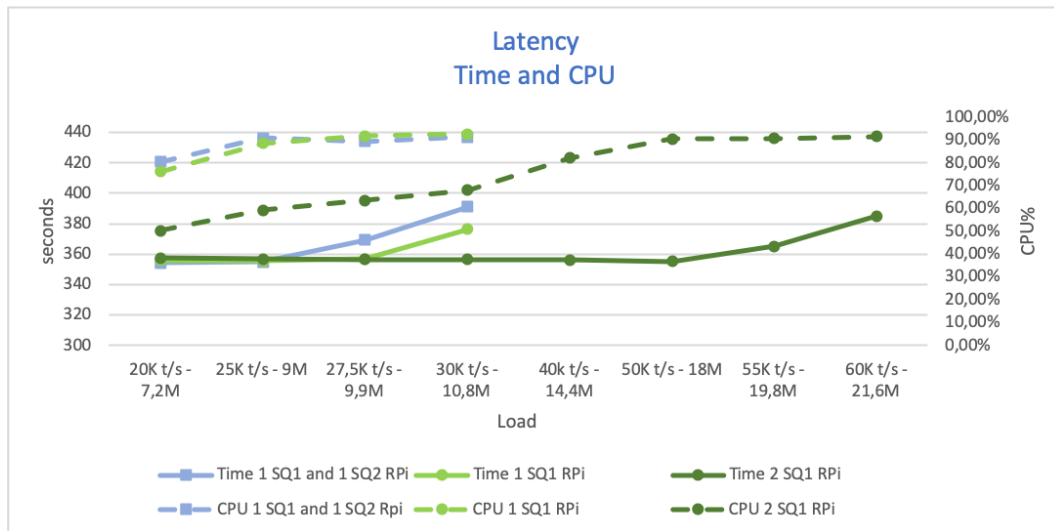


Figure 52 - Time to process a load

10.7.3. Performance Evaluation in a Data Center

As defined at the beginning of Section 10.7 the purpose of the first performance evaluation was to stress the CEP both in small devices and in the data center and see whether it can run in small environments where thousands of events per second are received and its processing capabilities in a data center. On the other hand, the cost of migrating queries online has been measured in the data center. In this section we evaluate the performance of the CEP in a data center made of homogeneous servers.

10.7.4. Subquery Migration

The goal of this section is to evaluate the cost of migrating a query without stopping the CEP. This is needed for online provisioning and decommissioning nodes.

When the system is saturated another node or CEP InstanceManager (IM) can be deployed and one of the subqueries can be migrated from the original IM to the new one in order to distribute the load. To evaluate this, we have used a local cluster of 11 AMD Opteron nodes with 64 vcores 6376 @ 2.3GHz, 128GB of RAM, 1Gbit Ethernet and 3 direct attached SSD of 480GB.

We use the same query of HiBench shown in Figure 48. SQ2 is migrated from one IM to another one. Figure 53 shows how the migration process is implemented. Initially there are two IMs running in the same node, one of the IMs (IM2) is running both SQs. The orchestrator receives the migration requests and prepares the new configuration to migrate SQ2 from IM2 to IM1 (step 1). This configuration is sent to IM1 that is going to deploy SQ2 (steps 2 and 3). Once the SQ2 is running at IM1 the streams SQ1 to SQ2 and SQ2 to Data Sink from the SQ2 running at IM1 are stopped, meanwhile tuples that goes out SQ1 are being batched (step 4). The configuration of the aggregate operator at IM2 is send to IM1, all the tuples that were stored by the aggregate operator at the time the migration stars are sent to the new SQ2 aggregate operator. Once this has finished, the SQ1 stored tuples start

to be sent to the new SQ2 and the normal flow of the tuples continues (steps 5 and 6). To finish the migration process, SQ2 from IM2 is undeployed.

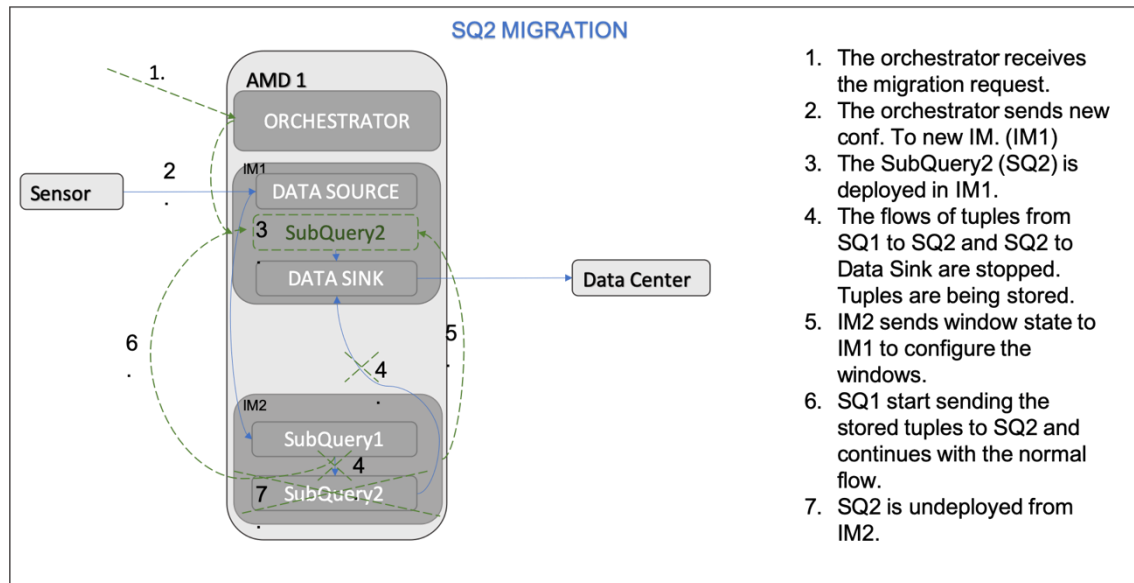


Figure 53 - Subquery migration

To evaluate the migration process different window sizes (5, 10 ,15 and 30 second) and different loads (10000 to 40000 tuples/second). Several metrics have been analyzed such as the time taken for all the migration process, the time taken to reconfigure the status of the window, the number of windows and the number of tuples transferred (Figure 54, Figure 55, Figure 56, Figure 57). The duration of the migration process performance is directly related with the number of stored tuples by the aggregate operator. The migration with 30 second windows size has a duration of 2839 ms when the load 10000 tuples/sec and 6550ms with a load of 30000 tuples/sec. With a load of 40000 tuples/sec the time needed for the migration is lower than in the previous case (5879 ms). Figure 57 shows that the number of tuples that have to be sent during the migration process from the original aggregate operator to the new aggregate operator is around 78M tuples while with a load of 40,000 tuples/sec and 30 second window size, the number of transferred tuples is around 68M.

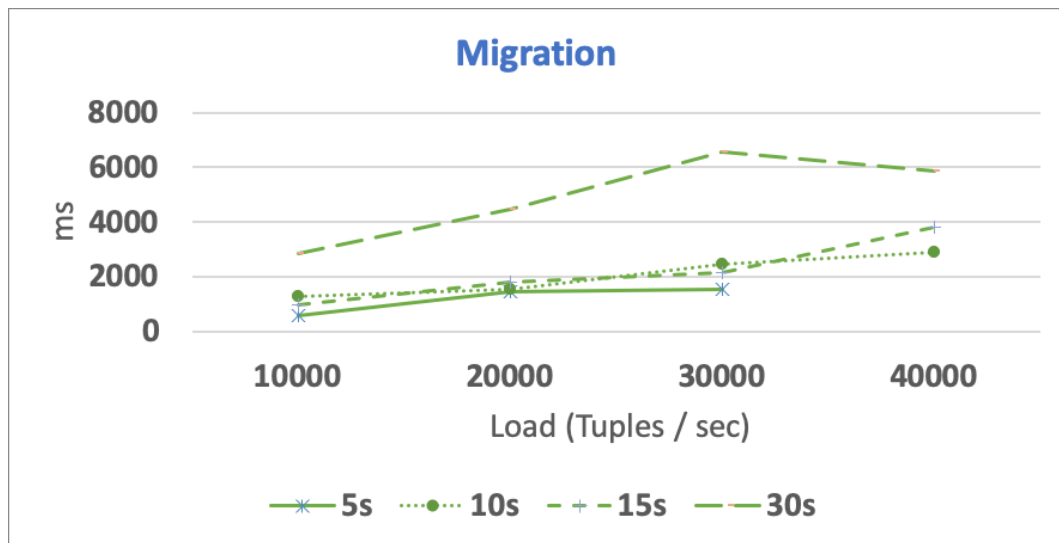


Figure 54 - Migration duration

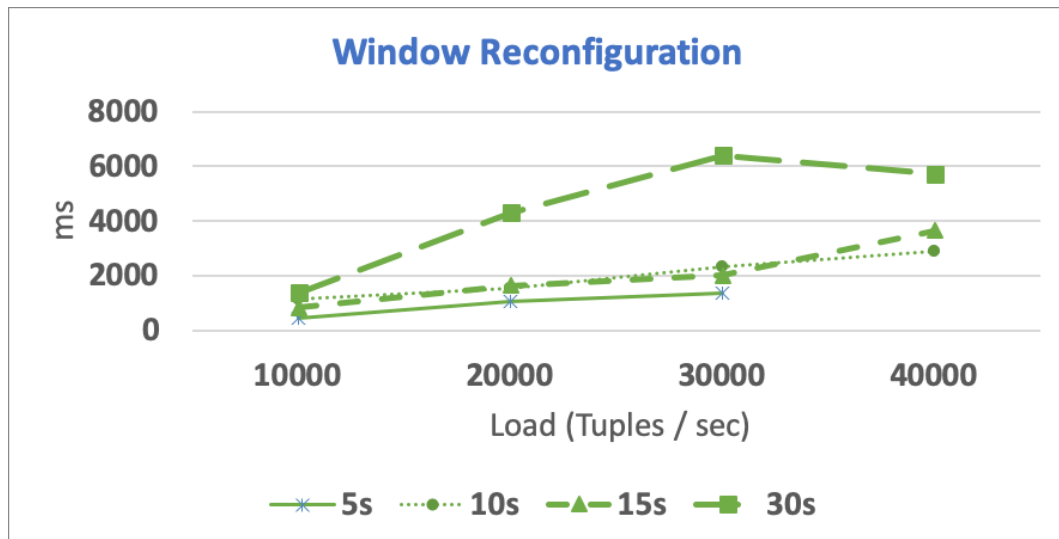


Figure 55 - Time for window reconfiguration



Figure 56 - Number of windows transferred during migration

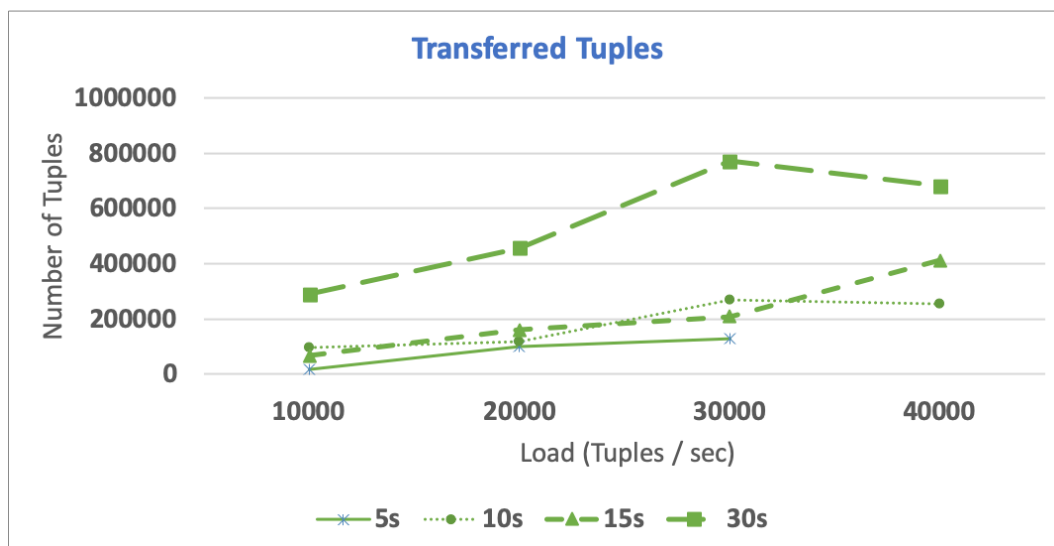


Figure 57 - Number of transferred tuples during migration

10.7.5. Scalability in a Distributed System.

The goal of this evaluation is to measure the scalability of the CEP both in a single node and in a homogeneous distributed system. This evaluation has been performed using 5 Intel XEON nodes, each one equipped with 2 CPU socket with Intel XEON E5-2620 v3, each one with 6 cores (12 vcores), for a total of 24 virtual cores. 128 GB RAM divided in 8 slots; each slot has 16GB RAM card and 3 SSD disks of 480GB each one. The nodes are connected by a 10Gbit Infiniband network. One node hosts the client and also receives the results. The test of the nodes host the CEP.

Figure 58 Figure 58 - Scalability in a single node shows the scalability of the CEP in a single node. The smallest configuration uses three IMs, two of them run SQ1 and the third IM runs SQ2 in a single node. In this case, the system process 200,000 tuples per second. This

configuration is replicated in the same node (6 IMs). Then, the handled load is also duplicated (400,000 events per second are processed). When the initial configuration is triplicated (9 IMs), the CEP processes 600,000 events per second. The CPU usage of SQ2 decreases with the increment of the number of instances since the number of windows is constant. In the first case with 3 IMs where there is one SQ2 instance that handles 65,025 windows. With 6 IM there are 2 SQ2 instances, each one will handle 32,512 windows.

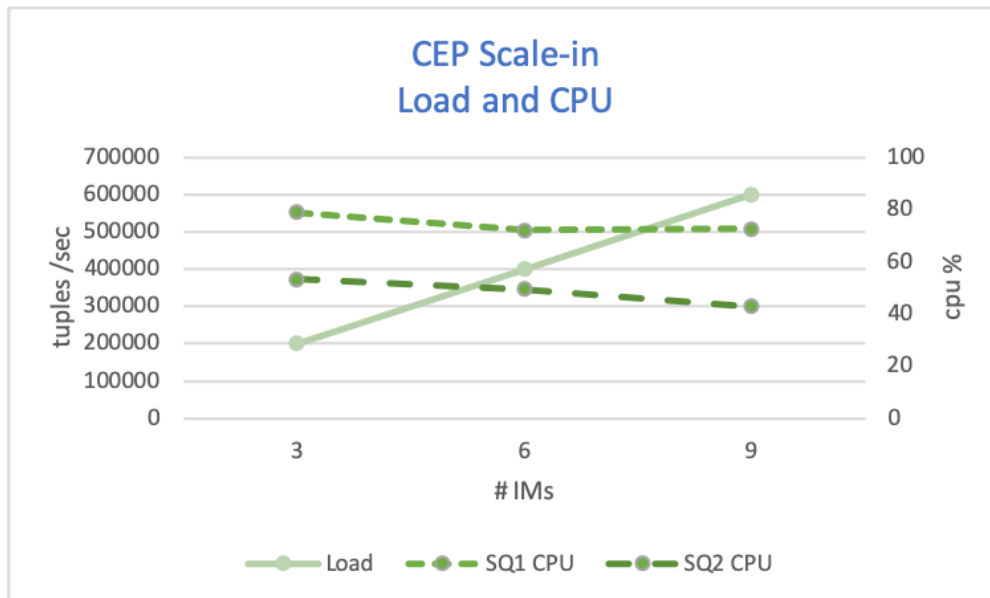


Figure 58 - Scalability in a single node

Once a single node has been scaled-in till no more IMs can be deployed, this configuration is deployed 2, 3 and 4 nodes. Figure 59 shows the CEP scalability. The results show that the CEP is able to scale linearly for the tested configurations.

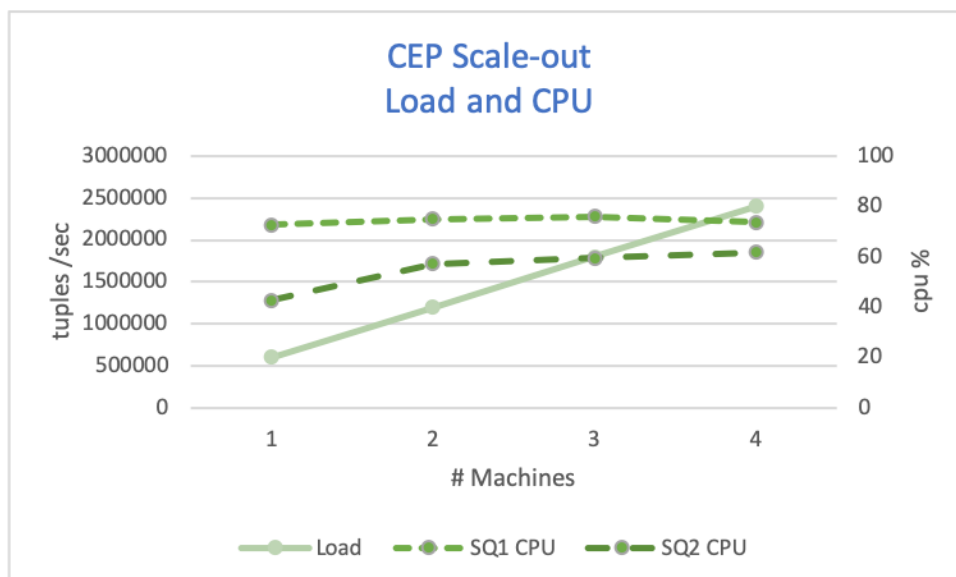


Figure 59 - CEP scale-out