




Operator Cost Estimation and Workflow Optimization Technology V1 Work Package 5 Task 5.1 & Task 5.2 Deliverable 5.1

Authors

Nikos Giatrakos, Alkis Simitsis, Theodoros Bitsakis, George
Stamatakis, Antonios Deligiannakis
Athena Research & Innovation Center

 Project supported by the European Commission Contract no. 825070	WP5 T5.1 & T5.2 Deliverable D5.1	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public



Distribution list:

Groups:	Others:
WP Leader: Athena Task Leader: Athena	Internal Reviewer Partner: SpringTechno (Spring) INFORE Management Board INFORE Project Officer

Document history:

Revision	Date	Section	Page	Modification
0.1	27/03/2020	Section 9	67-74	Creation.
0.2	30/03/2020	Sections 1-3	5-15	Creation.
0.3	02/04/2020	Section 3	12-15	Additions on tools for monitoring statistics and types of collected statistics.
0.4	07/04/2020	Section 4,5	16-29,30-39	Creation.
0.5	09/04/2020	Section 6,7	40-49	Creation
0.6	10/04/2020	All	All	Self-review, modifications and corrections throughout the text.
0.7	13/04/2020	Section 8	56-66	Submitted for internal review, creation of Section 8.
0.8	28/04/2020	All	All	Incorporation of internal review comments, self-review, modifications and corrections throughout the text.
0.9	28/04/2020	All	All	Self-review, modifications and corrections throughout the text.
1.0	29/04/2020	All	All	Final comments incorporated.

Approvals:

First Author: Nikos Giatrakos (Athena) Date: 25/04/2020

Internal Reviewer: Holger Arndt, Stefan Burkard (Spring) Date: 28/04/2020

Coordinator: Antonios Deligiannakis (Athena) Date: 30/04/2020



 Project supported by the European Commission Contract no. 825070	WP5 T5.1 & T5.2 Deliverable D5.1	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public




Table of contents:

1	Executive Summary	5
2	Introduction.....	6
2.1	Workflow Optimization in INFORE – The Big Picture	6
2.2	Connection with the INFORE Architecture	7
2.3	Physical Infrastructure	9
2.4	Supported Operators	9
3	INFORE Optimizer Design and Implementation	12
3.1	Optimizer Modules	12
3.2	Workflow File.....	13
3.3	Configuration Files	14
3.4	Endpoints	15
4	The INFORE Optimizer.....	16
4.1	Logical Plan Creation	16
4.2	Statistics Collection	17
4.3	Cost Estimator	21
4.3.1	Cost Estimator – Theoretic Foundations	21
4.3.2	Cost Estimator in Practice	23
4.3.3	Aggregative Cost Computations	24
4.4	Optimization Problem Formulation	24
4.5	Integration into an Exhaustive Search Algorithm.....	27
5	Advanced Optimization Algorithms	30
5.1	The A*-like Algorithm.....	30
5.2	Designed Algorithms.....	34
5.2.1	Dynamic Programming-alike Algorithm.....	34
5.2.2	Heuristic Algorithms.....	36
5.2.3	Greedy Algorithms.....	37
6	Extensions	40
6.1	Synopsis-based Optimization	40
6.1.1	...for enhanced horizontal scalability.....	40
6.1.2	...for vertical scalability.....	42
6.1.3	...for federated scalability.....	43
6.2	Optimizations tailored to Geo-distributed Complex Event Processing.....	43
6.2.1	Leveraging the push-pull rationale.....	44
6.2.2	Uncertainty-aware In situ Processing.....	47
7	Applications on INFORE Use Cases	50
7.1	Life Sciences Use Case.....	51
7.2	Financial Use Case	52
7.3	Maritime Use Case	54
8	Experimentation and Benchmarking.....	56
8.1	Benchmarking our Cost Estimator.....	56
8.2	Comparative Benchmarks on Optimization Algorithms.....	66
8.3	Benchmarks on Optimized Workflow Execution	69
9	Related Work	73
9.1	Statistics Collection and Monitoring Solutions	73
9.2	Query Optimization	73

 <p>Project supported by the European Commission Contract no. 825070</p>	<p>WP5 T5.1 & T5.2 Deliverable D5.1</p>	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public



9.3	Cost Estimation.....	75
9.4	Runtime Prediction	77
9.5	Cost Modeling in Multi-Engine Systems.....	77
9.6	Optimization Aspects over High Performance Computing Infrastructure	79
9.7	Geo-distributed CEP Optimization	80
10	Conclusions and Future Work.....	81
11	References	82

 Project supported by the European Commission Contract no. 825070	WP5 T5.1 & T5.2 Deliverable D5.1	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public




1 Executive Summary

Big Data processing workflows typically span a multitude of execution and storage platforms. Parts of the processing could be pushed to the input sensor level, as in the case of the wavegliders in the Maritime use case, while other more computationally intensive parts/operators (such as stock correlation functions in the Financial use case, or gene simulations in Life Sciences use case) could be executed either within one or more (potentially distributed) Big Data platforms or within other clusters (i.e., GPUs) of a supercomputer. Even within a single (i.e., BSC's MareNostrum 4) supercomputer one often finds different available clusters, with different hardware and processing capabilities, which could process a given workflow. Hence, the space of potential plans (a.k.a. physical execution plans) to process a Big Data workflow could be vast. Finding in a timely fashion the right plan that is both efficient and cost effective is not trivial.

This deliverable presents techniques for optimizing workflow execution in terms of a set of optimization objectives (e.g., throughput, resource utilization) of extreme-scale analytics across different, potentially geo-dispersed computer clusters each hosting one or more Big Data platforms.

WP5 interacts with WP4 since the Optimizer Component is a fundamental component of the overall INFOR architecture. WP5 receives a logical workflow as JSON formatted input from the Graphical Editor Component of the architecture via the Manager Component. It ingests statistics collected by the Manager Component to perform cost estimations and judge the performance of alternative execution plans i.e., the Optimizer Component transforms the logical workflow to a physical one to be deployed in the available computer clusters and Big Data platforms. Having performed this mapping, it returns it to the Manager Component to visualize it to the Graphical Editor Component of the INFOR architecture and deploy it to the available computer clusters. Moreover, WP5 interacts with the Synopses Data Engine Component and the Machine Learning and Data Mining Component of WP6 which provide the physical implementations of respective logical operators drawn in the Graphical Editor Component during code-free workflow specification. Finally, WP5 optimizes the logical workflows satisfying the application needs of the Biological (WP1), Financial (WP2) and Maritime (WP3) use cases.

 Project supported by the European Commission Contract no. 825070	WP5 T5.1 & T5.2 Deliverable D5.1	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

2 Introduction

The INFORE architecture aims at processing workflows involving large-scale streaming data. These are complex workflows, typically spanning multiple computer clusters and Big Data platforms. Identifying an efficient workflow execution could involve decisions made on more than one platform. For example, consider a stream involving two platforms, Kafka and Spark. It is possible to optimize the stream on each platform using best practices for Kafka and Spark, respectively. Furthermore, platforms employing an intrinsic optimizer (e.g., Spark Catalyst) can provide an efficient execution plan for the part of the workflow running on the said platform, as they have more knowledge of system internals. However, a platform cannot know that a workflow it is executing is part of a larger workflow that spans other platforms. In other words, no platform has a complete picture of the workflow.

The INFORE Optimizer helps with providing a holistic approach covering the entire workflow. Note, that the INFORE Optimizer is configurable so that it may optionally guide the platform specific optimizers (e.g. impose a certain parallelization degree for parts of a workflow executed over Spark), but always works complementarily by identifying optimization opportunities outside a platform and enabling further intra-platform optimizations by actions like *function shipping* (i.e., move a computation closer to the data) and *data shipping* (i.e., move the data closer to the computation). A basic optimization for the example Kafka-Spark workflow would be to push a filter from Spark down to Kafka to reduce the amount of data shipped over to Spark.

The optimization in INFORE is multi-dimensional. Given a Big Data workflow and a set of optimization goals, the optimizer must make a number of decisions, including the following:

- *Many flows.* Split the workflow in a number of subflows, such that each subflow contains operators to be executed on the same platform. If a workflow is executed entirely on a single platform, there is only one subflow the workflow itself.
- *Many platforms.* Choose a platform to process each subflow of the workflow. Note, that two non-sequential subflows can be executed on the same platform.
- *Many implementations.* Choose an implementation for each operator of the workflow. A workflow operator may have zero (i.e., the operator is not supported), one or more implementations within a single Big Data platform. For example, a join can be implemented in Spark¹ as a Sort Merge join or a Shuffle Hash join or a Broadcast join, and in Flink² as Broadcast Hash or Repartition Hash or Repartition Sort Merge, etc.
- *Many objectives.* Choose a plan that satisfies multiple optimization goals, such as minimize runtime, increase throughput, maximize resource utilization, and so on. As an extra complexity, often these objectives may be conflicting, e.g., a fault tolerant plan may not have the optimal execution performance.


The following subsections present a general overview of the Optimizer, its design, and its connectivity within the INFORE architecture.

2.1 Workflow Optimization in INFORE – The Big Picture

Workflow optimization in INFORE is based on a multitude of optimization objectives and configuration or system parameters such as resource availability, resource efficiency, workload parameters at runtime, efficiency of streaming technologies, availability of operator implementation on multiple platforms, availability of platforms, and so on. Typical goals considered by the optimizer include increased throughput, reduced latency, reduced communication under business and technical constraints. The multi-platform approach of the optimizer allows us to efficiently handle and select the best suited resource available. For example, a join operation on multiple streams is a common implementation on most frameworks and so the optimizer can decide which platform to use, for example based on network locality or on available compute resources.

¹ <https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#join-operations>

² https://ci.apache.org/projects/flink/flink-docs-stable/dev/batch/dataset_transformations.html#join

 <p>Project supported by the European Commission Contract no. 825070</p>	<p>WP5 T5.1 & T5.2 Deliverable D5.1</p>	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

The Optimizer receives a workflow from the Manager Component of the INFORE architecture. Then it optimizes the workflow (if applicable) and sends the optimized workflow back to the Manager which handles workflow execution. This operation can be done either offline or online, during the running execution of a workflow, which allows the INFORE architecture to adaptively react to changing condition of data streams and available resources. The runtime adaptation is future work planned for Deliverable D5.2.

The workflow metadata describe the designed streaming analysis process, the available resource information, the involved platforms from the connected streaming backends, and user design choices. This information is encoded in a JSON format. An informed design choice we made is to decouple the workflow encoding of the Manager Component from that of the Optimizer. This is based on two reasons. First, in doing so, both the Manager and the Optimizer Components can be replaced by other tools if needed. This allows for increased pluggability and easy enhancement of the INFORE architecture, and of potential reuse of the code in future applications.

The second reason was to enable a platform-agnostic workflow representation inside the Optimizer. Hence, a workflow designed for specific platforms (e.g., Spark) is *platform specific*. For example, consider a workflow getting data directly from Kafka and containing a filter operator and a join operator implemented in Spark. When the workflow is propagated into the Optimizer, it is converted to a *platform-agnostic* form that contains a logical filter operator and a logical join operator. This enables the Optimizer to look for optimization opportunities in other available engines (e.g., Kafka). A possible scenario for the example Kafka-Spark workflow could be as follows: (a) first, the workflow is transformed to a platform agnostic form and thus, the Spark filter and join are converted into a platform-agnostic filter and join, respectively; (b) then, the workflow is processed by the Optimizer that may identify an opportunity to push the filter back to Kafka; (c) next, the Optimizer converts the workflow to a platform specific workflow having two parts, one part with a filter to be applied to Kafka and one part with a join to be executed in Spark. We discuss this further in Section 4.1.

Once a workflow is sent to the Optimizer, the Optimizer enumerates the space of possible and promising execution plans for the workflow and estimates plan costs using a dynamic cost model that predicts workflow execution runtime. The navigation of the execution plan space can be done exhaustively or in a greedy fashion using heuristics for improved performance.


The optimization of complex Big Data processing workflows, consisting of several interconnected data processing operators, requires the collection of cost estimates for executing an operator, or a set of operators, at different available Big Data platforms, HPC systems, or even at the source/sensor level that generates data. This cost estimation is complex, since the number of patterns is large, the corresponding costs depend on the size and characteristics of the input data in each case, there may be system constraints (i.e., time or resources in an HPC system) imposed by an administrator on the time to collect the relevant statistics, etc. To deal with cost estimation, the Optimizer employs a dynamic cost model to enable accurate estimation of the processing cost and required resources (i.e., CPU utilization, memory requirements, etc.) for each workflow operator. The model is applicable at both Big Data platforms and HPC systems. Cost estimation is boosted by a statistics collection component that keeps a history of statistic observations over past workflow execution at various granularities such as at the workflow level, at the operator level, and at the platform level. We detail the cost model in Section 4.3.

Next, we present how the Optimizer interacts with the INFORE architecture.

2.2 Connection with the INFORE Architecture

The conceptual design of INFORE Architecture is illustrated in Figure 1 and it comprises the following components:

- Connection Component
- Graphical Editor Component
- Manager Component
- Optimizer Component
- Synopsis Data Engine Component (SDE)
- Complex Event Forecasting Component (CEF)
- Interactive Online Machine Learning and Data Mining Component (OMLDM)

	Project supported by the European Commission Contract no. 825070	WP5 T5.1 & T5.2 Deliverable D5.1	Doc.nr.:	WP5 D5.1
			Rev.:	1.0
			Date:	30/04/2020
			Class.:	Public

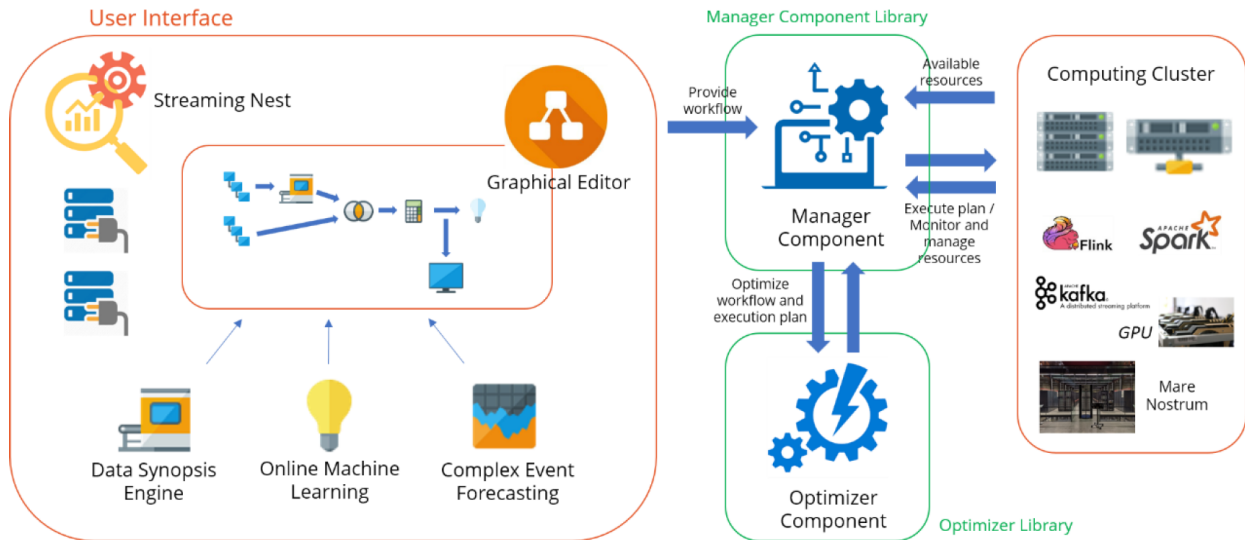


Figure 1: Overview of the conceptual design of the INFORE Architecture

All components are designed in a modular way, with clear interaction-interfaces between them. This enables an easy exchange between the modules and provides the necessary pluggability of the INFORE architecture. All architectural components communicate with each other via REST APIs.

The Manager is the central component that orchestrates the interaction among the other components and handles the execution and monitoring of the designed streaming analysis processes. The Manager serves as the connection point of the Optimizer with the overall INFORE Architecture. The Manager also orchestrates the collection of workflow execution statistics, which are used by the Optimizer's cost model.

The Optimizer receives a workflow from the Manager, optimizes it, and returns the optimized version back to the Manager for further action, such as inspection, dispatch, execution or scheduling.

The Graphical Editor and Connection Components are implemented in the RapidMiner Studio by the project partner RapidMiner. The RapidMiner Studio is extended to support a Streaming Optimization and a Streaming Nest operator, which are subprocess operators that allow an arbitrary number of operators to be placed inside them. The Streaming Optimization operator is used by the user of the Graphical Editor Component to define the initial logical workflow deprived of any platform specific details. The optimizer devises the corresponding physical workflow for the previously drawn logical one, with part of it to be executed in each of the available platforms. For each cluster, Big Data platform which undertakes the physical execution of a part of the workflow, a Streaming Nest operator is placed within the Streaming Optimization one. The Streaming Nest operator for each cluster, platform includes operators corresponding to the part of the workflow that has been assigned there. The operators in Streaming Optimization and Streaming Nest operators are inter-connected defining a streaming workflow. The connection of operators to underlying execution or messaging (e.g. Kafka) platforms are handled by RapidMiner Studio's Connection objects. INFORE adds Connection object classes for all supported streaming backends, thereby implementing the Connection Component. This is essential for the cross-platform optimization of data stream analysis in INFORE. An example workflow representation is illustrated in Figure 2. Use case specific workflows are discussed in Section 7, while throughout our discussion we sketch appropriate exemplary workflows as well.

<p>Project supported by the European Commission Contract no. 825070</p>	<p>WP5 T5.1 & T5.2 Deliverable D5.1</p>	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

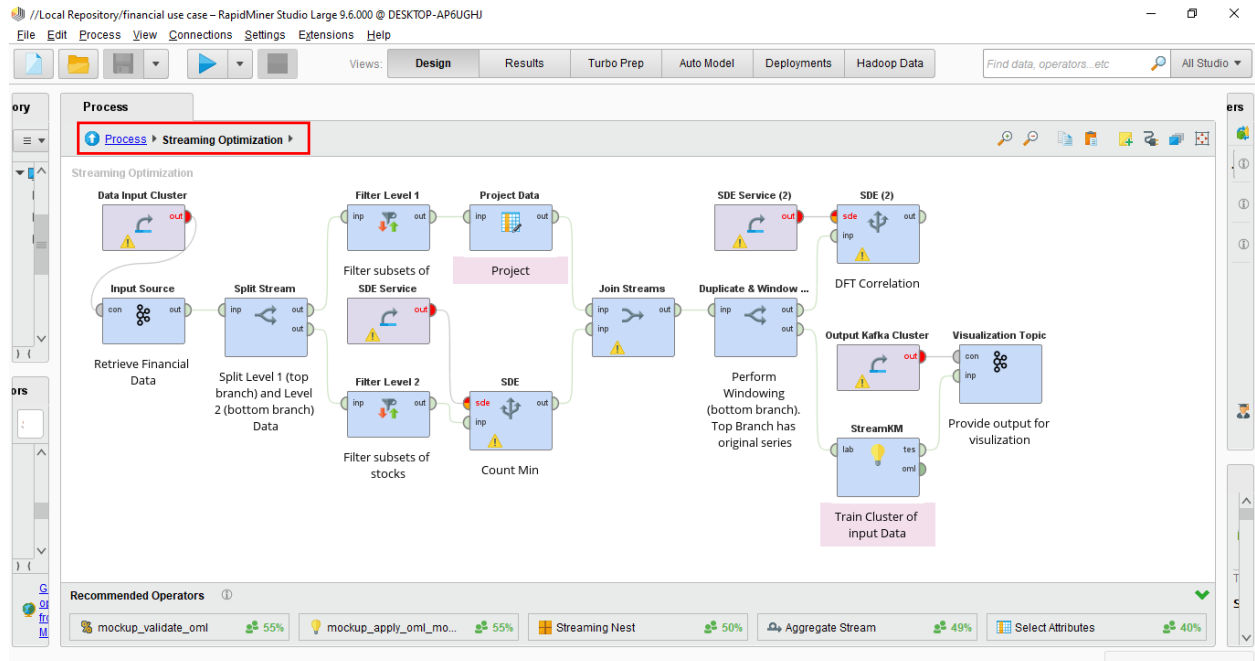


Figure 2: An example workflow design in the Graphical Editor component.

More details on the INFOR architecture can be found in Deliverable D4.1 and Deliverable D4.2.

2.3 Physical Infrastructure

The INFOR Optimizer is designed to work with the most popular Big Data Platforms and HPC systems. In the current implementation, it supports:

- Execution engines: Apache Spark³, Apache Flink⁴, and Akka⁵
- Messaging systems: Apache Kafka⁶

The architectural choices considered by the Optimizer include:

- Processing on on-prem clusters or HPC
- Processing components: CPU, GPU

The design is generic and can be straightforwardly extended to support additional systems and platforms.

2.4 Supported Operators

A typical data analysis workflow may contain a large variety of operators implementing algorithms spanning areas such as traditional data processing (e.g., relational database operators), data analysis and data mining, complex events forecasting, synopses, machine learning, deep neural networks, and so on. INFOR takes a generic, black-box approach to design, execute, and profile such rich set of options.

Therefore, INFOR supports stream operators provided in Apache Spark and Flink. In addition, INFOR supports additional, homegrown complex analytic streaming operators, including:


- *Classification*

³ <https://spark.apache.org/>


⁴ <https://flink.apache.org/>

⁵ <https://akka.io/>

⁶ <https://kafka.apache.org/>

 <p>Project supported by the European Commission Contract no. 825070</p>	<p>WP5 T5.1 & T5.2 Deliverable D5.1</p>	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

- *Passive Aggressive Classifier (PA)*. An online binary classification method for linearly separable data, capable of handling concept drifts [Cra+06]. The algorithm solves a constraint minimization problem and find the best possible separating hyperplane for the two classes. This method is based on minimizing the Hinge Loss function (the same used by SVM). This algorithm is used extensively for testing our component because of its simplicity.
- *Regression*
 - *Passive Aggressive Regressor (PAR)*. An online regression method that fits a linear function to the streaming data. This yet is another method used for testing our component [Cra+06].
 - *Online Ridge Regression (ORR)*. The online version of the well-studied ridge regression algorithm. The method incrementally solves the ordinary least squares regression problem with the addition of the l_2 norm penalty [HoKe88, Vovk01].
- *Preprocessing*
 - *Standardization*. A normalization method that subtracts the population mean from each individual data point and then divides the difference by the population standard deviation. The mean and variance vector can be computed incrementally, or they can be given by the user.
 - *Polynomial Features*. A pre-processing method that generates extra features for each data point. The generated data points consist of all polynomial combinations of the features with degree less than or equal to the specified degree.
- *CEF Operator*. The CEF operator is an implementation of a formal framework that attempts to address the issue of Complex Event Forecasting (CEF). It is based on symbolic automata and a variable-order Markov model and can capture long-term dependencies in a stream. Based on the dependencies it uncovers, it can provide forecasts in an online manner about when a complex event is expected to occur, before such an event is actually detected. More details can be found in Deliverable 6.2 and the interested reader may also refer to previous work on the topic [AlAP17, AlAP18].
- *Synopses operators*. A number of synopses is currently supported in INFOR via the Synopses Data Engine (SDE), including:
 - *CountMin*: A Count-Min Sketch is a two-dimensional array of $w \times d$ dimensionality used to estimate frequencies of elements of a stream using limited amount of memory.
 - *BloomFilter*: A Bloom filter is a space-efficient representation of a stream of elements from a certain universe, mainly used to deduce whether a certain element has been observed.
 - *FM Sketch*: The FM sketch is a bitmap used to estimate the number of distinct elements in a stream using a limited amount of memory.
 - *HyperLogLog*: The HyperLogLog algorithm constitutes the evolution of FM sketches. It is a simple, elegant algorithm that enables to extract distinct counts using limited memory and a simple error approximation formula.
 - *AMS Sketch*: The key idea in AMS sketch is to represent a streaming (frequency) vector v using a much smaller sketch vector $sk(v)$ that is updated with the streaming tuples and provide probabilistic guarantees for the quality of the data approximation.
 - *Discrete Fourier Transform (DFT)*: The Discrete Fourier Transform transforms a sequence of complex numbers into another sequence of complex numbers in increasing order of importance. In the context of the SDE, importance, involves preservation of cross-correlation similarity values among streams. Stream dimensionality reduction is achieved by keeping a subset of DFT coefficients.
 - *Random Hyperplane Projection (RHP)*: The RHP LSH scheme as utilized in the SDE operates over tumblers, i.e., disjoint windows of a data stream of $|W|$ size each. Given a tumbler w_i of observations for a stream i , RHP produces a bitmap of d dimensionality with $d \ll bit(w_i)$, where $bit(w_i)$, is the size in bits of the respective values in w_i .

 <p>Project supported by the European Commission Contract no. 825070</p>	<p>WP5 T5.1 & T5.2 Deliverable D5.1</p>	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public


- *Lossy Counting*: The Lossy Counting algorithm maintains a data structure, which is a set of entries of the form $\langle \text{element}, \text{frequency}, \epsilon \rangle$, where element is a data element, frequency is an integer representing the estimated frequency of the element and ϵ tunes the allowed maximum possible error in frequency estimation.
- *Sticky Sampling*: Sticky Sampling shares similarities with Lossy Counting, but differs in that (i) the size of buckets/windows is not steady and (ii) the count of an element is maintained with a certain sampling probability.
- *Chain Sampler*: The chain sampling algorithm provides a simple random sample without replacement of size k over a sliding window of n cardinality where k is expected to be much lower than n .
- *GKQuantiles*: quantile estimation using a small memory footprint.
- *CoreSetTree*: computes a weighted sample of a data stream, called the CoreSet of the data stream. A data structure termed CoreSetTree is used to speed up the time necessary for sampling non-uniformly during CoreSet maintenance.
- *STSampler*: The STSampler, tailored to the Maritime Use Case, samples vessel positions within its trajectory only when they are considered important, i.e., they cannot be deduced or approximated sufficiently well by interpolating other already sampled positions.

The development of the SDE library leverages subtype polymorphism and hence, it allows for adding easily new synopses definitions. More details can be found in [KoGD20].

- *Stream Transformations*
- These essentially constitute the stream processing operators that are supported in the scope of the functional programming paradigm followed by the supported Big Data platforms. Such as Join, Filter, Map in Spark's Structured Streaming, Flink's DataStream API and so on.

As on-going and future work is concerned, the INFOR team works on adding operators such as:

- *Clustering*
 - Sequential k-means clustering. An online method of the original k-means clustering algorithm [Bah+12].
 - BIRCH. A memory-efficient online clustering algorithm [ZhRL96].
- *Classification*
 - Online Support Vector Machines (OSVM). An online implementation of the linear Support Vector Machine classifier.
 - Vertical Hoeffding Tree (VHT). A distributed algorithm for learning decision trees in an online manner [KFMM16]. A novel way of distributing decision trees via vertical parallelism that performs well on non-linearly separable data.
- *Regression*
 - Autoregressive Moving Average (ARMA). An algorithm for time Series Analysis. An Autoregressive Moving Average model is used to describe weakly stationary stochastic time series in terms of two polynomials. The first of these polynomials is for autoregression, the second for the moving average.

	Project supported by the European Commission Contract no. 825070	WP5 T5.1 & T5.2 Deliverable D5.1	Doc.nr.:	WP5 D5.1
			Rev.:	1.0
			Date:	30/04/2020
			Class.:	Public

3 INFORE Optimizer Design and Implementation

The INFORE Optimizer is implemented as a Web service to enable lightweight communication with the other components of the INFORE Architecture and especially, with the Graphical Editor, the Connection, and the Manager components.

Figure 3 illustrates a high-level overview of the Optimizer Web Service, which is implemented using the Java Spark Web Framework⁷ for creating web applications. The service receives as an input (a) the workflow to be optimized, and (b) a set of configuration files encoded in JSON. The latter are independent of the input workflow and are used to feed the Optimizer with environmental settings, such as:

- *Dictionary*. It specifies the list of supported operators, available platform implementations, and a computed cost estimation per operator, implementation, and platform.
- *Network topology*. It describes the topology of available compute resources and sites (e.g., clusters and their locations).
- *Configuration parameters*. It contains tuning parameters for the optimization process, such as a choice of an optimization strategy, and heuristic values for various knobs.

The output of the optimization service is an optimized workflow.

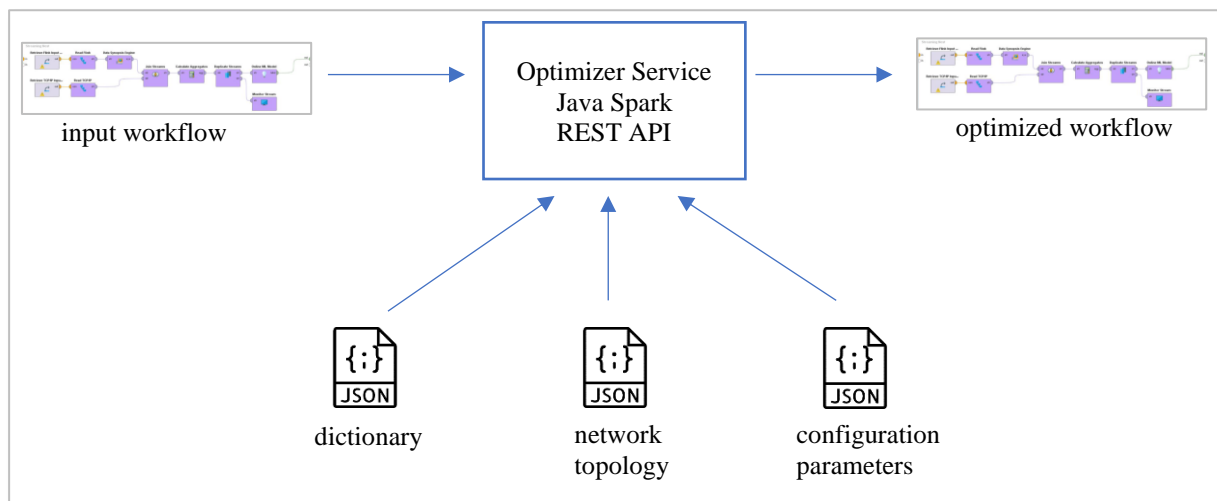



Figure 3: Optimizer Web Service

3.1 Optimizer Modules

The Optimizer implementation is based on Maven modules, comprising three main modules (see also Figure 4):

- *Optimizer*. It implements workflow optimization including the optimization algorithms, the cost model, and the plan creation.
- *Web*. It implements a Web service responsible for receiving submitted workflows and transmitting back optimized workflows.
- *Core*. It contains all Maven dependencies that are shared between modules and also the workflow parser.

⁷ <http://sparkjava.com/>

 <p>Project supported by the European Commission Contract no. 825070</p>	<p>WP5 T5.1 & T5.2 Deliverable D5.1</p>	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

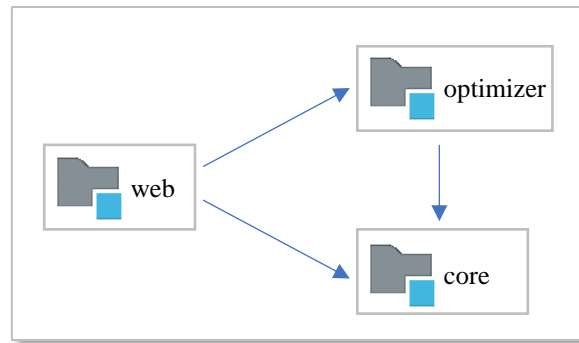


Figure 4: Optimizer modules

3.2 Workflow File


An INFOR workflow is encoded as a JSON file comprising three main elements: Operators, Operator Connections, and (optionally) Resources. These elements represent the operators of the workflow, the connections among them, and resources allocated to the workflow, respectively.

Figure 5 depicts an example snippet of a workflow JSON encoding. The example shows an operator named “Logical Decision Tree”, which receives input data from the port “output 1” and propagates its result to the port “training set”. The example encoding shows also implementation details like the class that implements this operator.

```

{
  "operatorConnections": [
    {
      "fromOperator": "Multiply",
      "fromPort": "output 1",
      "toOperator": "Logical Decision Tree",
      "toPort": "training set"
    },
    {
      "fromOperator": "Logical Decision Tree",
      "fromPort": "training set",
      "toOperator": "Logical Decision Tree",
      "toPort": "output 1"
    },
    {
      "fromOperator": "Logical Decision Tree",
      "fromPort": "output 1",
      "toOperator": "Logical Decision Tree",
      "toPort": "training set"
    },
    {
      "fromOperator": "Logical Decision Tree",
      "fromPort": "training set",
      "toOperator": "Logical Decision Tree",
      "toPort": "output 1"
    },
    {
      "fromOperator": "Logical Decision Tree",
      "fromPort": "output 1",
      "toOperator": "Logical Decision Tree",
      "toPort": "training set"
    },
    {
      "fromOperator": "Logical Decision Tree",
      "fromPort": "training set",
      "toOperator": "Logical Decision Tree",
      "toPort": "output 1"
    },
    {
      "fromOperator": "Logical Decision Tree",
      "fromPort": "output 1",
      "toOperator": "Logical Decision Tree",
      "toPort": "training set"
    }
  ],
  "operators": [
    {
      "name": "Logical Decision Tree",
      "classKey": "streaming:logical_decision_tree",
      "operatorClass": "com.rapidminer.extension.operator.logical.LogicalDecisionTree",
      "isLogicalOperator": true,
      "parameters": [
        {
          "name": "output 1",
          "value": "output 1"
        },
        {
          "name": "training set",
          "value": "training set"
        }
      ],
      "inputPortsAndSchemas": [
        {
          "port": "output 1",
          "schema": "output 1"
        },
        {
          "port": "training set",
          "schema": "training set"
        }
      ],
      "outputPortsAndSchemas": [
        {
          "port": "output 1",
          "schema": "output 1"
        },
        {
          "port": "training set",
          "schema": "training set"
        }
      ]
    }
  ],
  "resources": {
    "allocatedMemory": null,
    "maxCPU": null,
    "numberOfContainers": null,
    "inputSize": null,
    "networkBandwidth": null,
    "selectivity": null,
    "throughput": null,
    "latency": null
  }
}
  
```

Figure 5: Example snippet of a workflow JSON encoding

 <p>Project supported by the European Commission Contract no. 825070</p>	<p>WP5 T5.1 & T5.2 Deliverable D5.1</p>	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

3.3 Configuration Files

There are three types of Configuration files: dictionary, network topology, and configuration parameters.

Dictionary. A logical operator may have multiple physical implementations either on a single platform or across multiple platforms. In order to achieve platform inter-operability and preserve flow semantics across multiple platforms, we need a means for translating platform specific characteristics from one platform to another. To deal with this, INFORE uses a dictionary of mappings. In addition to keeping information useful for code interpretation and generation, the dictionary also contains attributes that can be used during workflow processing, like the operator cost models specific to an implementation and platform. The dictionary comprises the following elements: (a) supported operators; (b) available platform agnostic and platform specific implementations; and (c) a cost estimate per operator, implementation, and platform. Figure 6 shows an example snippet of a Dictionary entry. This snippet shows a fragment for a Filter operator, with implementations in SQL, Java, Apache Spark, Apache Flink, and Akka.

```
dictionary_name : dictionary1
▼ infore_operators [1]
  ▼ 0 {2}
    infore_name : Filter
    ▼ platform {5}
      ► sql {3}
      ► java {3}
      ▼ spark {3}
        operator_name : SparkFilter
        cond : filter
        cost : 10
      ► flink {3}
      ► akka {3}
```

Figure 6: Example snippet of a Dictionary entry

Network Topology. The Network file describes the topology of available compute resources and sites including metrics and statistics of profile characterization of these resources. An example snippet of a Network entry is shown in Figure 7. The figure shows an example site called “site1” and drills down to the details of an Apache Spark deployment on that site. It also shows an example metric representing the measured latency in communicating with that site. Such metrics are frequently updated based on periodic benchmarks (see also Section 4.3).

```
network : network1
▼ sites [3]
  ▼ 0 {3}
    site_name : site1
    ▼ available_platforms [2]
      ▼ 0 {6}
        platform_name : spark
        driver_memory_mb : 4096
        executors : 1
        executor_cores : 4
        executor_memory_mb : 4096
        topic_key : site1_spark1
      ► 1 {6}
    ▼ metrics {1}
      latency : 10.4
```

Figure 7: Example snippet of a Network entry

<p>Project supported by the European Commission Contract no. 825070</p>	<p>WP5 T5.1 & T5.2 Deliverable D5.1</p>	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

Configuration Parameters. The Configuration file lists a set of parameters required for tuning the optimization process. As described in Section 4.5 and Section 5, INFOR supports more than one optimization strategy for creating an optimized execution plan. Each strategy comes with a set of tuning knobs, which are tuned using the respective entries in the configuration file. Figure 8 shows an example snippet of a Configuration entry that includes an “exhaustive” optimization strategy, the cost estimator, and the concurrency level (a.k.a. multi-programming level or MPL).

```
{
  "version": "1.0",
  "algorithm": "exhaustive",
  "costEstimator": "Bayesian",
  "parallelism": 4
}
```


Figure 8: Example snippet of a Configuration entry

3.4 Endpoints

The Optimizer Web Service provides a set of endpoints via REST API to facilitate the communication with the other INFOR components. These endpoints include:

- *Monitor.* A number of services to monitor the status of internal optimization processes, such as:
 - /progress: Returns the progress of a running optimization strategy
 - /plan: Returns the current best plan (this may change as better plans may be found over time as the optimization process progresses)
- *Control.* A number of services to control an optimization process, such as
 - /start: Starts an optimization service
 - /pause: Pauses an optimization service
 - /kill: Stops an optimization service
- *Workflow.* A number of services to handle workflow files, such as:
 - /workflow: Submits a workflow and receives an optimized workflow
- *Dictionary.* A number of services to handle the dictionary, such as:
 - /dictionary: Loads a dictionary
- *Network.* A number of services to handle the network topology, such as:
 - /network: Loads a network topology
- *Configuration.* A number of services to handle the configuration parameters, such as:
 - /optcfg: Loads the configuration parameters

The last three endpoints also include services such as (a) find: retrieve entries in a dictionary, (b) add: add an entry, (c) delete: delete an entry, (d) update: update an entry, and so on.

	Project supported by the European Commission Contract no. 825070	WP5 T5.1 & T5.2 Deliverable D5.1	Doc.nr.:	WP5 D5.1
			Rev.:	1.0
			Date:	30/04/2020
			Class.:	Public

4 The INFORE Optimizer

This section describes the operation of core functions of the Optimizer.

4.1 Logical Plan Creation

As discussed, a workflow in INFORE may span multiple execution platforms. One may consider this workflow as a single, logical computation and it may be modeled as such. However, a logical flow has many possible implementations, each serving a different purpose. If the workflow designer directly creates an implementation (or physical workflow) that meets the desired objectives for the workflow, over time, objectives may change, data volumes may increase rendering an implementation sub-optimal, the underlying infrastructure may change, or the logical workflow may need modification. Creating and modifying physical workflows is labor-intensive, time-consuming, and error prone. From an optimization perspective, optimizing a physical workflow containing subflows designed and tuned for different platforms and thus, often having different semantics is hard, and in some cases inefficient. To deal with these challenges, INFORE adopts the notion of *platform independence* for workflows [JoSW14]. Similar to the logical data independence that insulates a data modeler from physical details of a relational database, there are benefits in designing and optimizing workflows at a logical level.

A logical, platform-independent workflow provides a unified, end-to-end view of the entire computation. This enables the optimizer to change the workflow design without altering its semantics and functionality. For example, simple, well-known, and effective techniques such as pushing a selective operator early in the computation can still be made even if the said operator is designed to run on a different platform. Simple cases as push down are already available in hybrid, batch-processing frameworks, but require additional plug-ins and specialized connectors; e.g., many relational databases employ specialized connectors to Map-Reduce systems (like Hadoop) to support pushing SQL filters down to external systems. In most cases though, this functionality has not been incorporated into the system optimizers and needs to be done manually. And still, these connectors support simple operators. Supporting much more complex operators as those described in Section 2.4 might be doable with specialized connectors, but building connectors for every possible operator is not trivial and does not scale. Considering the entire workflow at the logical level, opens optimization opportunities in a simpler way and at much lesser development cost.

A *logical workflow* is independent of an execution platform. A workflow can be represented as a directed acyclic graph whose vertices are the logical workflow operators and its edges represent the data flow among these operators. Hence, a logical workflow contains vertices that do not necessarily have information about specific implementation, or resource allocation, etc. A logical workflow merely represents the computation intended by the workflow designer.


A *physical workflow* is designed for a specific execution platform (e.g., Spark or Flink) and instantiates parts of a logical workflow or the logical workflow in its entirety. Therefore, the vertices of the graph representing a physical workflow contain the information needed to bound each operator to a specific implementation and platform.

A logical operator may have multiple physical implementations either on a single platform or across multiple platforms. In order to achieve platform inter-operability, preserve flow semantics across multiple platforms, and enable conversions from physical to logical workflows and vice versa, we maintain a dictionary of mappings between logical constructs and their physical incarnations in the supported platforms. Figure 6 depicts an implementation of an example mapping.

There are two ways to create a logical workflow: by design or by conversion.

In a typical use case scenario, the INFORE user designs a logical workflow in the Graphical Editor. Such a workflow represents only the intended computation, which is quite convenient for the casual data scientist who does not want to get distracted by implementation details and prefers to leave that task to INFORE.

The Optimizer also handles the case of physical workflows, which can either have been designed as such by the workflow designer (e.g., the designer creates a workflow to run on Spark) or they may have been created by a previous optimization process. In those cases, the Optimizer converts the physical workflow into a logical one as follows. For each workflow operator that contains information about a specific implementation, the Optimizer

 Project supported by the European Commission Contract no. 825070	WP5 T5.1 & T5.2 Deliverable D5.1	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

identifies the implementation type and uses the dictionary to map it to its corresponding logical type. An example logical type is shown in Figure 5: the type of the operator “2” can be seen by its “com.rapidminer.extension.operator.logical.LogicalDecisionTree” operator class.

The Optimizer operates on logical workflows. It decides an appropriate placement of operators and creates a plan (see Section 4.5 and Section 5). In the optimized plan, the logical operators have been instantiated with the information needed (e.g., implementation type, platform choice, respective cost) to get converted into their respective physical incarnations, by following the reverse process and mapping the logical operator types to the physical ones, again through the dictionary.

4.2 Statistics Collection

The most popular Big Data Platforms (such as Spark and Flink) offer several ways to monitor the execution of a workflow either online or after the fact, through a Web UI, metrics, and/or external instrumentation with cluster-wide monitoring tools, profiling tools, or stack traces sniffing tools. For statistics collection on workflow execution, we rely on platform specific configurable metrics systems that report a plethora of execution metrics to a variety of forms like HTTP, Java Management Extensions (JMX)⁸, and CSV files.

In INFOR, statistics collection relies on the REST API services provided by the Big Data Platforms to poll metrics and other useful information regarding running workflows, individual operators, cluster load, resource utilization, etc. In particular, we use Logstash⁹ to tap into these metrics via JMX connections. Using Logstash, we periodically probe the resource managers of the Big Data Platforms, collect metrics, transform them on the fly into a more readable format, and store them to a centralized storage. Our current choice for storage is Elasticsearch; other solutions such as a timeseries database can also be used.

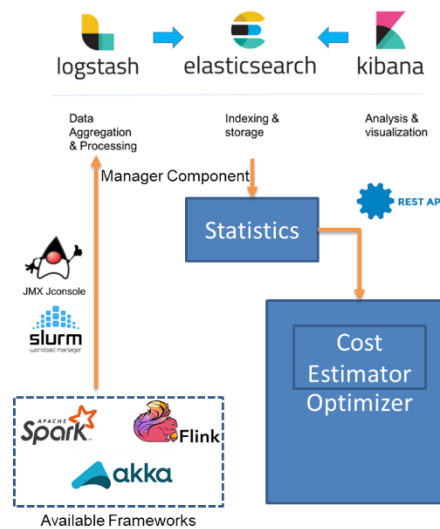


Figure 9: Statistics collection and utility from computer clusters to the INFOR Manager Component and finally to the Optimizer Component.

As an example, we describe next the process for metrics collection from Flink. We work with other platforms in a similar way.

A statistics collection driver starts with initializing an object to handle the data received from REST API calls and a parser used for JSON deserialization.

```

objectMapper = new ObjectMapper();
objectMapper.configure(SerializationFeature.INDENT_OUTPUT, true);
  
```

⁸ https://en.wikipedia.org/wiki/Java_Management_Extensions

⁹ <https://www.elastic.co/logstash>

<p>Project supported by the European Commission Contract no. 825070</p>	<p>WP5 T5.1 & T5.2 Deliverable D5.1</p>	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public



```
flinkParser = new FlinkParser(ClusterConstants.flink_master, objectMapper);
```

We also initiate a Timer to handle communication with Flink:

```
flink_timer = new Timer("FlinkMetricsTimer");
```

which can be called as:

```
flink_timer.scheduleAtFixedRate(new TimerTask() {  
    @Override  
    public void run() { ... }  
}, FlinkDelay, FlinkRefreshPeriod);
```

Then, a pseudocode to periodically collect metrics from Flink would be as follows.


```
// Connect to Flink JM and retrieve metrics  
jobmanager = flinkParser.fetchFlinkJobManager();  
// Fetch running jobs  
flinkJobs = flinkParser.fetchFlinkJobs(null, "RUNNING");  
// Get metrics per job  
for (FlinkJob job : flinkJobs) {  
    // process job metrics  
    ...  
}
```

The Flink runtime comprises two types of processes: the JobManagers (masters) and the TaskManagers (workers)¹⁰. The JobManagers coordinate the distributed execution (e.g., scheduling, checkpoint coordination, recovery from failures). The TaskManagers execute the tasks of a workflow and handle data stream buffers and exchanges. In INFORE, the metadata to orchestrate the metrics collection from Flink are stored in two files in JSON format, one for describing Flink Jobs and one for Flink Tasks. Example snippets for each file are as follows.

JobManager script:

```
{  
  "host": "host.docker.internal",  
  "port": 9250,  
  "queries": [  
    {  
      "object_name": "java.lang:type=Runtime",  
      "attributes": [ "Uptime", "StartTime"],  
      "object_alias": "Runtime"  
    },  
    {  
      "object_name": "java.lang:type=GarbageCollector,name=*",  
      "attributes": [ "CollectionCount", "CollectionTime"],  
      "object_alias": "${type}.${name}"  
    },  
    {  
      "object_name": "org.apache.flink.jobmanager.*:host=jobmanager",  
      "attributes": [ "Value"]  
    }  
  ]  
}
```

¹⁰ More details can be found here: <https://ci.apache.org/projects/flink/flink-docs-stable/concepts/runtime.html>


 Project supported by the European Commission Contract no. 825070	WP5 T5.1 & T5.2 Deliverable D5.1	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

The following example TaskManager script collects metrics such as number of buffers, numBytesOutPerSecond, numRecordsIn, numRecordsInPerSecond, numRecordsOut, numRecordsOutPerSecond. Additional metrics can be added trivially by extending the “queries” element.

TaskManager script:

```
{
  "host": "host.docker.internal",
  "port": 9251,
  "queries": [
    {
      "object_name": "java.lang:type=Runtime",
      "attributes": [ "Uptime", "StartTime"],
      "object_alias": "Runtime"
    },
    {
      "object_name": "java.lang:type=GarbageCollector,name=*",
      "attributes": [ "CollectionCount", "CollectionTime"],
      "object_alias": "${type}.${name}"
    },
    {
      "object_name": "org.apache.flink.taskmanager.Status*:host=*,tm_id=*",
      "attributes": ["Value"]
    },
    {
      "object_name": "org.apache.flink.taskmanager.job.task.buffers*:task_name=*,job_id=*,
        task_attempt_id=*,job_name=*,tm_id=*,task_id=*,
        task_attempt_num=*,host=*,subtask_index=*",
      "attributes": ["Value"]
    },
    {
      "object_name": "org.apache.flink.taskmanager.job.task.numBytesOutPerSecond:task_name=*,
        job_id=*,task_attempt_id=*,job_name=*,tm_id=*,task_id=*,
        task_attempt_num=*,host=*,subtask_index=*",
      "attributes": ["Count","Rate"]
    },
    {
      "object_name": "org.apache.flink.taskmanager.job.task.numRecordsIn:task_name=*,job_id=*,
        task_attempt_id=*,job_name=*,tm_id=*,task_id=*,task_attempt_num=*,
        host=*,subtask_index=*",
      "attributes": ["Count"]
    },
    {
      "object_name": "org.apache.flink.taskmanager.job.task.numRecordsInPerSecond:task_name=*,
        job_id=*,task_attempt_id=*,job_name=*,tm_id=*,task_id=*,
        task_attempt_num=*,host=*,subtask_index=*",
      "attributes": ["Count","Rate"]
    },
    {
      "object_name": "org.apache.flink.taskmanager.job.task.numRecordsOut:task_name=*,
        job_id=*,task_attempt_id=*,job_name=*,tm_id=*,task_id=*,
        task_attempt_num=*,host=*,subtask_index=*",
      "attributes": ["Count"]
    },
    {
      "object_name": "org.apache.flink.taskmanager.job.task.numRecordsOutPerSecond:task_name=*,
        job_id=*,task_attempt_id=*,job_name=*,tm_id=*,task_id=*,

```

 <p>Project supported by the European Commission Contract no. 825070</p>	<p>WP5 T5.1 & T5.2 Deliverable D5.1</p>	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

```
task_attempt_num=*,host=*,subtask_index=*,
"attributes": ["Count","Rate"]
},
]
}
```

In order to collect statistics from HPC infrastructures and the MareNostrum 4 in Barcelona's supercomputer center we also have to integrate Slurm in our statistics collector¹¹. Slurm stands for Simple Linux Utility for Resource Management and it is used at the HPC infrastructure available to INFOR as a job scheduler. Slurm interprets the amount of resources a job requires and controls how many resources are available for new jobs.

The following python snippet sketches how to go through all the nodes of a Slurm cluster and get their CPU/GPU/Memory Usage information per partition¹²:

```
nodes = pyslurmnode.get()
for node in nodes:
    node_data = nodes.get(node)

    metrics['partition']['cpu_total']['ALL'] += node_data['cpus']
    metrics['partition']['cpu_usage']['ALL'] += node_data['alloc_cpus']
    ...
    metrics['partition']['mem_total']['ALL'] += node_data['real_memory'] * 1048576
    metrics['partition']['mem_usage']['ALL'] += node_data['alloc_mem'] * 1048576
    ...
#Generic Resource Objects
if node_data['gres']:
    gres_total = pyslurm.node().parse_gres(node_data['gres'][0])
    gres_usage = pyslurm.node().parse_gres(node_data['gres_used'][0])
    for g in gres_total:
        is_gpu = re.match(r'^gpu:([0-9]+)\(?, g)
        if is_gpu:
            gpu_total = int(is_gpu.group(1))

    if gpu_total > 0:
        for g in gres_usage:
            is_gpu = re.match(r'^gpu:(?:[^\:]*:?) ([0-9]+)\(?, g)
            if is_gpu:
                gpu_usage = int(is_gpu.group(1))


    metrics['partition']['gpu_total']['ALL'] += gpu_total
    metrics['partition']['gpu_usage']['ALL'] += gpu_usage
    ...
payload = []
for grouping in ['partition', ...]: #we can also collect statistics per user, job etc
    for reading in ['cpu_total', 'cpu_usage', 'gpu_total', 'gpu_usage', 'mem_total',
'mem_usage']:
        if reading in metrics[grouping] and len(metrics[grouping][reading]) > 0:
            for key in metrics[grouping][reading].keys():
                payload.append({'measurement': '%s_%s' % (grouping, reading), 'time':
now, 'fields': {reading: float(metrics[grouping][reading][key])}, 'tags': {grouping:
key}})

client.writePoint(payload, database=config['elasticsearch'])
```

The statistics that can be tracked using the above collection mechanisms differ between platforms and collection tools, JMX, Slurm. However, tracking quantities related to number of records/bytes received/sent, degree of parallelism, available/used processing nodes along with their cores and available/used memory is supported. These primary statistics can be obtained at the operator, part of workflow and entire workflow level implicitly (e.g. running

¹¹ <https://www.bsc.es/user-support/faq.php#slurm>

¹² <https://slurm.schedmd.com/elasticsearch.html>

	Project supported by the European Commission Contract no. 825070	WP5 T5.1 & T5.2 Deliverable D5.1	Doc.nr.:	WP5 D5.1
			Rev.:	1.0
			Date:	30/04/2020
			Class.:	Public

1 operator per submitted job) or explicitly. From these statistics we can observe other, derived metrics such as throughput (number of records being processed per time unit), latency (composed of processing, read/write and communication latency) as well as communication cost, by appropriate configurations. For instance, should one want to measure the read/write latency of an operator when the input/output is directed to a Kafka cluster, she can do so by configuring the same operator receiving/sending its input/output once internally from/to the Big Data platform where it is deployed (e.g from/to one Flink operator to/from another) and once externally, e.g. to Kafka. Having measured this, one can similarly derive the communication latency between geo-dispersed computer clusters using Kafka as a messaging service and so on.

4.3 Cost Estimator

4.3.1 Cost Estimator – Theoretic Foundations

Before providing a formal formulation of our optimization problem let us consider the expected functionality of the INFORE optimizer. This will facilitate our discussion about the kind of cost estimator we need in order to evaluate alternative physical workflow execution plans. In its endeavor to convert the logical workflow to a physical one, the Optimizer Component shall attempt to examine the available options with respect to instantiating each logical operator to (a) a networked computer cluster (data center), (ii) a Big Data platform that is hosted there, (iii) prescribe the parallelization degree and account for resource capacity. These are the elements that should be provided so that the Manager Component that will receive the JSON of the prescribed physical plan, can dispatch jobs to be submitted to respective clusters and Big Data platforms.


In INFORE we aim at optimizing the execution of various types of machine learning, data mining, complex event forecasting and stream transformation operators. Moreover, INFORE aims at a pluggable and extensible architecture which should allow customization driven by application specific needs. Therefore, potential adopters should be allowed to plug-in new operators, for instance, by defining new synopses in the SDE Component (see Deliverable D6.1 for further details). Moreover, the INFORE architecture is not tied to specific Big Data platforms but can be extended to support any future platform that gains popularity among stakeholders in academia and in the industry. Therefore, a workflow engages a variety of operators which behave differently. Even within a single component, such as the OMLDM or the SDE Component, operators of heterogeneous nature exist.

For these reasons, we cannot rely on cost estimation methods that employ crisp assumptions to build analytic formulas for individual operator's behavior, since this cannot accommodate all architectural components and limits pluggability and extensibility. The cost estimator that INFORE's optimization module should adopt must allow us to treat each operator as a black box, i.e., the function $f(x)$ describing the performance of the same operator in different clusters, Big Data platforms and provisioned resources, is a black box function. We cannot assume an analytical expression for $f(x)$. What we want, is to find a set x that minimizes the cost $f(x)$ of executing the operator.

This is a set up where Bayesian optimization techniques [BrCF10] are most useful. They attempt to find the global optimum for predicting the value of $f(x)$ using a minimum number of iterations/samples. Bayesian optimization incorporates prior belief about $f(x)$ (i.e., a small set of the values) and updates the prior with samples drawn from $f(x)$ to get a posterior that better approximates $f(x)$. The model used for approximating the objective function is called *surrogate model*. Bayesian optimization also uses an *acquisition function* that directs sampling to areas where an improvement over what is currently considered as optimal is likely.

The most commonly used surrogate model is Gaussian processes (GPs). GPs can initially be used to exploit prior beliefs about $f(x)$. A Gaussian process is a random process where any point $x \in R^d$ is assigned a random variable $f(x)$ and where the joint distribution of a finite number m of these variables $Pr(f(x_1), \dots, f(x_m))$ is Gaussian as well, i.e., $Pr(f|X) = N(f|\mu, K)$, where $X=(x_1, \dots, x_m)$, $f=(f(x_1), \dots, f(x_m))$, the expected value (which can be set to zero) $\mu=(\mu(x_1), \dots, \mu(x_m))$ and $K_{i,j} \in K: K_{i,j}=\kappa(x_i, x_j), \forall (x_i, x_j) \in X$. κ is a positive definite kernel (or covariance) function. A Gaussian process is a distribution over functions. The smoothness of these functions is determined by the kernel. If points x_i, x_j are similar according to the kernel, the same would hold for the function values $f(x_i), f(x_j)$.

The GP prior $Pr(f|X)$ described above can yield a GP posterior $Pr(f|X, y)$ after having observed some more samples y . The posterior can then be used to make predictions f^* given a new input (query) as a set X^* . In particular:

 <p>Project supported by the European Commission Contract no. 825070</p>	<p>WP5 T5.1 & T5.2 Deliverable D5.1</p>	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

$$Pr(f^*/X^*, X, y) = \int Pr(f^*/X^*, f) \cdot Pr(f/X, y) df = N(f^* | \mu^*, \Sigma^*)$$

The above function is also a Gaussian with mean μ^* and covariance matrix Σ^* . The joint distribution of observed data y and predictions f^* is $\begin{pmatrix} y \\ f^* \end{pmatrix} = \begin{pmatrix} 0 \\ K_y \quad K^* \\ K^{*T} \quad K^{**} \end{pmatrix}$. $K_y = K + \sigma_y^2 I$, $K^* = \kappa(X, X^*)$, $K^{**} = \kappa(X^*, X^*)$. $K_y \in \mathbb{R}^{n \times n}$, $K^* \in \mathbb{R}^{n \times n^*}$, $K^{**} \in \mathbb{R}^{n^* \times n^*}$ with n training data and n^* new inputs. T denotes the conjugate transpose of the matrix. Moreover, $\mu^* = K^{*T} K_y^{-1} y$, $\Sigma^* = K^{**} - K^{*T} K_y^{-1} K^*$.

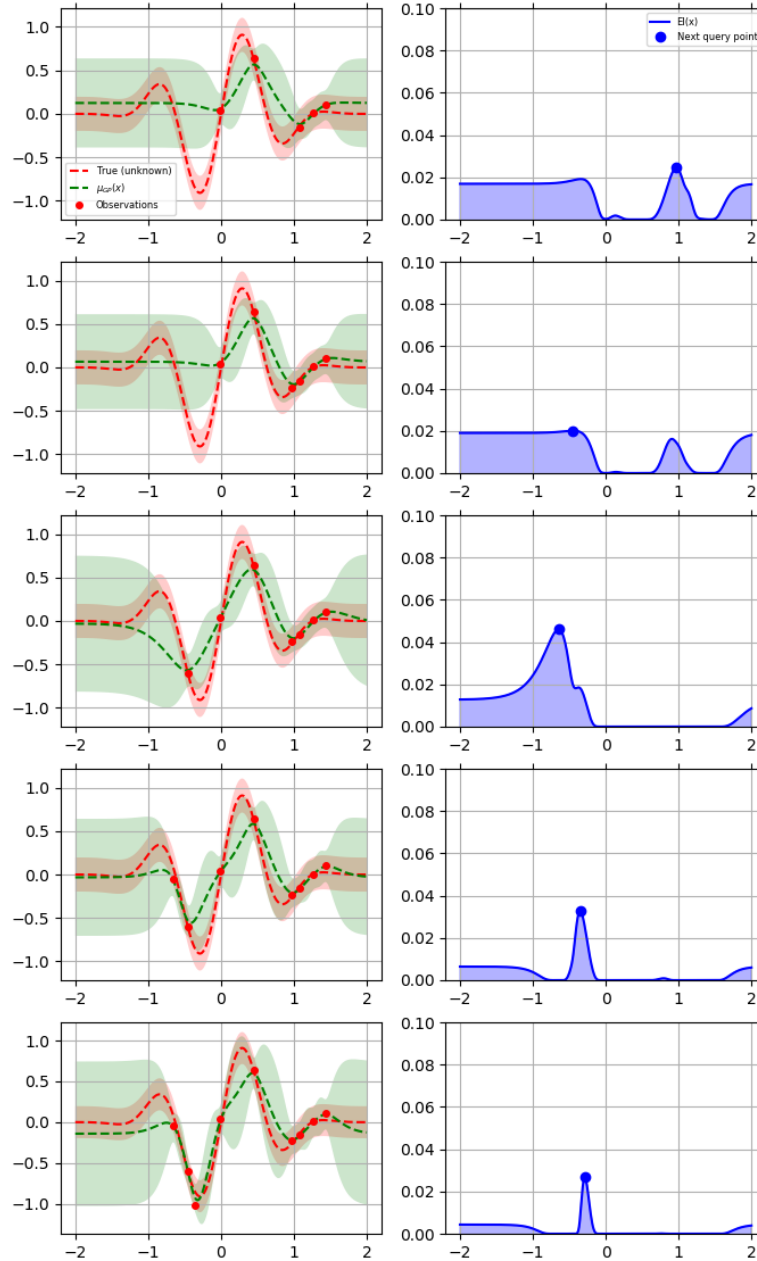



Figure 10: Operation of Bayesian optimization¹³ after a number of sampling iterations. Plots on the left show how well the true function (red line) is approximated by GP posterior (green line) in subsequent sampling iterations. The plots on the right show the shape of the acquisition function and the next point to sample. A combination of Matern and WhiteNoise kernels is used in the Gaussian Process. Expected Improvement (EI) is the utilized acquisition function.

¹³ https://scikit-optimize.github.io/stable/auto_examples/bayesian-optimization.html

	Project supported by the European Commission Contract no. 825070	WP5 T5.1 & T5.2 Deliverable D5.1	Doc.nr.:	WP5 D5.1
			Rev.:	1.0
			Date:	30/04/2020
			Class.:	Public

The GP posterior can be used to propose points in the search space where sampling is likely to yield an improvement. Proposing sampling points in the search space is done by the acquisition function. Popular acquisition functions are Maximum Probability of Improvement (MPI), Expected Improvement (EI) and Lower Confidence Bound (LCB). The acquisition function is expected to sample where GP predicts a high/low objective or where the prediction uncertainty is high due to noisy input. Both cases correspond to high/low acquisition function values and the goal is to maximize/minimize the acquisition function to determine the next sampling point. It is also possible to provide more sampled points in y without using the acquisition function's proposed samples. In this case, the posterior GP would improve, but the improvement will be much less targeted in better predictions. When new samples are used in y without using an acquisition function, a fitting process is taking place, while using the acquisition function corresponds to training the developed process. Figure 10 and Figure 11 depict our previous discussion in toy examples of open-source libraries¹³. In Section 8, we present respective plots tied to INFOR experimental scenarios.

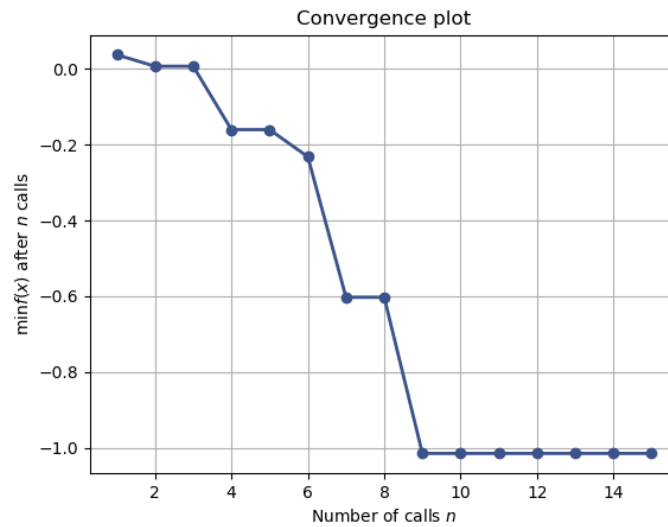



Figure 11: Convergence plot from ¹³ for the setup of Figure 10. Initially one x is fitted and then the acquisition function chooses subsequent samples, after 9 such samples, no new sample improves predictions, i.e., the training process converges as the objective function reaches its minimum.

4.3.2 Cost Estimator in Practice

Given the lack of an a priori known $f(x)$ for every possible operator and its alternative implementations, what we can do is to obtain some initial samples and construct a prior/posterior belief for the unknown function as described in the previous section. Practically this is achieved by performing some microbenchmarks and collecting information about respective statistics and performance metrics for each operator. In our case, $x \in R^d$ is a vector describing for each operator the statistics mentioned in Section 4.2 per computer cluster and Big Data platform with available operator implementations. In the description of our algorithms in Section 4.5 and Section 5, when we refer to performance computation with respect to our optimization objectives, we mean that we query the developed cost estimator with input parameters involving input stream rates (records and bytes sent/received, execution time etc) per networked cluster and involved Big Data platforms. The cost estimator responds with a prediction f^* with respect to the optimization objectives which are: throughput, latency, communication cost, CPU and memory usage (Section 4.4). Later on, in Section 6.1 we add another objective reflecting output accuracy when SDE operators are used.

Each experiment, in a microbenchmark we construct, involves the evaluation of the black box function, restricted to sampling each time at a point x and getting the result of the experiment including both the input parameters and performance objectives derived as described in Section 4.2. However, evaluating $f(x)$ at many points is an expensive procedure which involves performing micro-benchmarks over different clusters, platforms, resource configurations. Thus, we cannot a priori sample at every possible point. Therefore, what we do is, having the prior and posterior GP by some initial experiments, we then let the acquisition function ask the next sample that corresponds to a new, useful microbenchmark we should perform. This essentially enhances the y dataset for obtaining the GP posterior. In particular, for every new operator, we run a number of microbenchmarks, we provide a small percentage of them to

	Project supported by the European Commission Contract no. 825070	WP5 T5.1 & T5.2 Deliverable D5.1	Doc.nr.:	WP5 D5.1
			Rev.:	1.0
			Date:	30/04/2020
			Class.:	Public

initiate the GP posterior and let the acquisition function request the next samples that can be useful for training the model. If a sample is missing from the already performed microbenchmarks we run a new one.

Notice that by using microbenchmarks our approach is proactive, in the sense that we obtain cost estimators before existing or new operators get deployed to full-fledged workflows. The advantage of this is that we avoid cold-start issues, i.e., situations where we know very little or nothing about an operator's performance. The second advantage of the Bayesian optimization approach is that we can exploit a learn-by-example paradigm. That is, every time an operator is deployed in an actual workflow, we keep collecting statistics for it increasing the cardinality of the set X used to construct a more accurate GP posterior (fitting). A third advantage of our cost estimation approach is that it is not restricted to individual operators. The learn-by-example paradigm can be used for parts of a workflow or the workflow in its entirety. If parts or entire workflows are commonly used in submitted jobs, we are going to have accumulated a high number of statistics, because the INFORE architecture collects information for monitored jobs via the Manager Component which can be stored in Elasticsearch (Section 4.2). In the latter case, we can treat parts of the workflow, even spanning different Big Data platforms and clusters, as black boxes, use the statistics to better fit GPs and obtain predictions by querying the developed models. Having done this, a new job will be better optimized and deployed, and we are going to obtain new samples that can be useful for further enriching the GP posterior, simultaneously improving the overall accuracy of the predictions.

4.3.3 Aggregative Cost Computations

When we do have stored statistics about parts of a workflow or entire workflows, our algorithms can directly query the developed cost estimator to predict the performance of a candidate physical workflow execution plan for that part of the workflow. However, when a logical workflow includes a previously unseen combination of operators, since our microbenchmarks only ensure cost estimations per operator, we need an intuitive way to aggregate individual operator performances at each level of an examined part of a workflow. Therefore, below we define how this is performed.

Throughput: The aggregative throughput of a physical workflow is computed as the minimum throughput of the participating physical operators.

Latency: The aggregative latency of a physical workflow is computed as the sum of processing (per cluster, Big Data platform, parallelization degree), communication (per network link), read/write of its operators.

Memory Usage: The aggregative memory usage of a physical workflow is computed as the sum of memory resources used by its operators (per cluster, Big Data platform, parallelization degree).


CPU usage: The aggregative CPU usage of a physical workflow is computed as the sum of CPU resources used by its operators (per cluster, Big Data platform, parallelization degree).

Communication Cost: The aggregative communication cost of a physical workflow is computed as the sum of bytes sent per time unit among the participating operators (placed at different networked clusters and Big Data platforms) per network link.

In the sequel, we refer to operators which contribute input to another operator op as its upstream operators, while downstream operators are the ones that receive input from op. There is an important note we need to make here since it affects the ability of our proposed algorithms to provide optimal solutions or near optimal ones. When a logical workflow includes operators that share the same upstream operator, the computation of additive quantities, such as communication cost, is affected. If the downstream operators of the shared one are placed on the same cluster and Big Data platform the communication cost should be considered only once. This is a parameter that a designed algorithm should take into consideration in order to provide accurate estimations of the monitored metrics.

4.4 Optimization Problem Formulation

Given as input (i) a logical workflow (Section 3.2, Section 4.1), (ii) a set of configuration files (Section 3.3) describing the network topology with computer clusters, available Big Data platforms and resources, (iii) the dictionary enlisting how supported logical operators correspond to physical implementations and (iv) collected

 <p>Project supported by the European Commission Contract no. 825070</p>	<p>WP5 T5.1 & T5.2 Deliverable D5.1</p>	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

statistics (Section 4.2) and cost models (at least) per operator, we want to perform a mapping of the logical operators to physical ones such that a multi-objective, constrained optimization problem is optimally solved. The notion of optimality is thought of as minimizing a cost quantity composed of multiple objectives under various resource constraints.

A formal definition follows shortly, but before presenting it let us provide more details of our problem setup for clarity. The mapping provided by solving the optimization problem will include for each logical operator Op (i) the networked cluster on which it should be executed, (ii) the Big Data platform available at that cluster that will undertake its execution, (iii) the provisioned resources, mainly involving the required parallelization degree. The choice of a CPU or GPU unit for executing one or more operators (e.g. via virtualization) is orthogonal to our techniques.

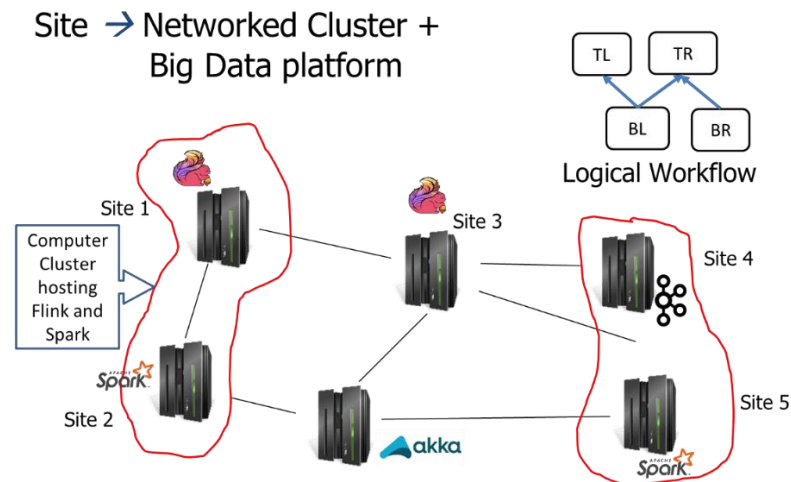


Figure 12: Abstract example of logical workflow to be executed over the available infrastructure

Furthermore, we want to be able to handle uniformly the cases of optimizing a single logical workflow or a number of such designed workflows that are submitted over the available networked infrastructure. To achieve that, given a set of logical workflows that reach the optimizer, we add a logical disjunction, OR, operator and connect each output operator (i.e., an operator with no downstream operators) of each logical workflow to that virtual OR operator. Moreover, note that operators of the workflow which involve data ingestion, such as Kafka topics in the financial use case or sensors on wavegliders in the Maritime use case, are included in the logical workflow and are considered as unary operators which map the logical operator to a single physical one with only downstream, but no upstream, operators.

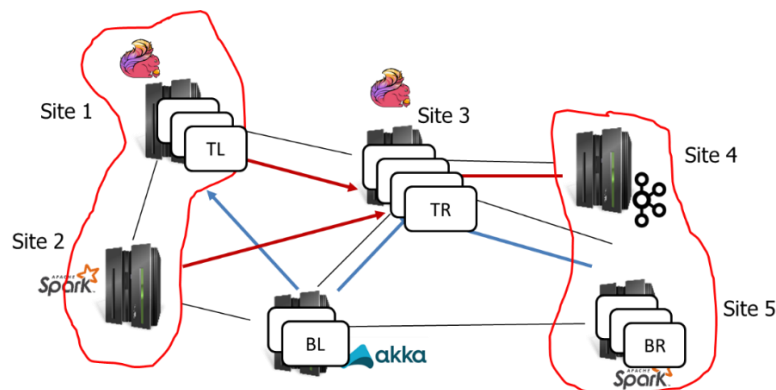


Figure 13: Physical workflow, i.e., having assigned the execution of each operator at an available site along with the parallelization degree.

To simplify our notation, we will consider a networked computer cluster (data center) hosting more than one Big Data platform as an equivalent number of separate clusters. For instance, a data center which hosts Flink, Spark and


<p>Project supported by the European Commission Contract no. 825070</p>	<p>WP5 T5.1 & T5.2 Deliverable D5.1</p>	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

Kafka will be considered as three computer clusters and, when needed, we will use the gathered statistics to derive that these clusters are physically near from a network (communication cost, network latency) perspective. We will term such a computer cluster, Big Data platform combination as “site” in what follows. Additionally, in our discussion hereafter, we will use the notation $CS_{i,j}^\mu$ to denote the set of candidate solutions for placing operator Op_i at site $S_j \in S$ and configuring it with a parallelization degree of μ . $CS_{i,j}^\mu$ is a set of solutions because each operator can be configured under various parallelization degrees, get instantiated in different Big Data platforms and available computer clusters. When needed, we will use the notation $cs_{i,j}^\mu \in CS_{i,j}^\mu$ to refer to a particular solution from the corresponding set. The virtual OR operator is assumed to be fixed at the query source and is given an index of $i=0$. It corresponds to a physical operator with no alternative implementations in Big Data platforms ($j=0$) and is also virtually run with $\mu=1$. However, since its cost depends on the choices made for its upstream operators, it also has a set $CS_{0,0}^1$ of solutions. This means that the cost computed when we reach the OR operator aggregates the values of the costs of all operators that directly or commutatively provide input to it. Recall that what we essentially want is to instantiate all operators so that we minimize the aggregative cost when we reach that OR operator. More formally, our optimization problem can be expressed as follows:

$$\begin{aligned}
 & \text{Minimize } cs_{0,0}^1 \cdot \text{cost} = cs_{0,0}^1 \cdot \sum_{k=1}^5 \lambda_k \cdot O_k \\
 & \text{s.t.} \\
 & \quad \text{system-wide constraints:} \\
 & \quad \quad cs_{0,0}^1 \cdot O_2 \leq LatConstr \\
 & \quad \quad cs_{0,0}^1 \cdot O_3 \leq CPUUsageConstr \\
 & \quad \quad cs_{0,0}^1 \cdot O_4 \leq MemConstr \\
 & \quad \quad cs_{0,0}^1 \cdot O_5 \leq CommCostConstr \\
 & \quad \quad \forall |\lambda_k| \leq 1, \sum_{k=1}^5 |\lambda_k| = 1 \\
 & \quad \text{site specific constraints:} \\
 & \quad \quad \forall S_j \in S, \sum_i cs_{i,j}^\mu \cdot O_4 \leq MemConstr_j \\
 & \quad \quad \sum_i cs_{i,j}^\mu \cdot O_3 \leq CPUUsageConstr_j \\
 & \quad \quad \forall (S_q \rightarrow S_j), \sum_i cs_{i,j}^\mu \cdot O_5 \leq BandConstr_{S_q \rightarrow S_j} \\
 & \quad \quad \text{where } cs_{0,0}^1 \in CS_{0,0}^1 \\
 & \quad \quad \text{and} \\
 & \quad \quad O_1: \text{Throughput (tuples being processed per time unit – second)} \\
 & \quad \quad O_2: \text{Overall Latency (processing + communication + read/write latency – seconds)} \\
 & \quad \quad O_3: \text{CPU usage (expected CPU units used/available CPU units per time unit –second)} \\
 & \quad \quad O_4: \text{Memory Usage (expected memory used/total available memory)} \\
 & \quad \quad O_5: \text{Communication cost (transmitted bytes per time unit – seconds)}
 \end{aligned}$$

Note that our problem definition is generic enough so that more objectives or constraints can be easily incorporated, while others may be omitted. Since we make the convention about optimally solving a minimization problem, some of the objectives participating in cost estimation may receive negative weight, i.e., λ_k , or own values, because we essentially want to maximize them. For instance, should we only care about maximizing throughput, we may set $\lambda_1 = -1$ (and all other weights to zero) because this converts the minimization problem to a maximization one.

The posed constraints are grouped to system-wide and site specific. System-wide objectives may affect, for instance, pricing of cloud services and are thus considered separately. Site-specific constraints refer to the hardware infrastructure that can be made available per site. Moreover, the site-specific constraint regarding memory usage, essentially says that for all operators (i.e., the sum runs over i) that will be placed at some S_j under a chosen parallelization degree, cannot exceed the memory capacity of S_j . Similarly, the site-specific bandwidth constraint says that the required bandwidth that will be consumed by an upstream operator placed at S_q directing input to

 <p>Project supported by the European Commission Contract no. 825070</p>	<p>WP5 T5.1 & T5.2 Deliverable D5.1</p>	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

others (the sum runs over i) placed in S_j should not exceed the bandwidth capacity of the path $S_q \rightarrow S_j$. Finally, there is a site-specific constraint that refers to CPU utilization. Note that since we want to serve all submitted workflows connected via the virtual OR operator, we do not directly relate the parallelization degree (μ) with CPU usage, since that would restrict the available options, such as exploiting pseudo-parallelism and hyperthreading or virtualization. Therefore, in the fourth constraint we refer to the expected CPU units used over the available CPU units per time unit at S_j . For instance, $CPUsageConstr_j$ maybe set to 60% so as to allow future, not currently drawn, workflows to have the possibility to be executed at S_j , or currently running workflows may have occupied only 40% of CPU usage available.

Our optimization objectives are conflicting. Thus, we cannot optimize all of them simultaneously, i.e., we cannot improve one objective without deteriorating at least one of the rest. For instance, increasing throughput or reducing communication cost may require placing all operators in a single site. However, this is also expected to increase CPU and memory usage. Therefore, we aim at providing Pareto optimal solutions. Pareto optimal solutions are solutions to the optimization problem so that no solution dominates another in all objectives. The set of solutions that satisfy this criterion forms a Pareto front as illustrated in Figure 14 for two dimensions (to ease the illustration). According to our problem definition, we have a multi-objective and thus multidimensional Pareto front. When we have to pick one solution out of the set of solutions that forms the Pareto front, we pick the one that minimizes the cost function as described above.

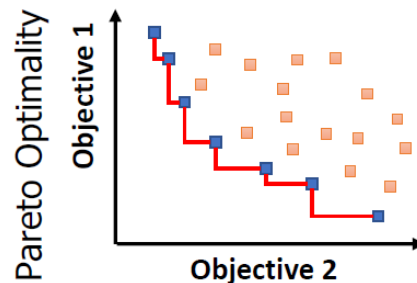



Figure 14: Pareto Front formation in two dimensions

The above setup and the algorithms we discuss (excluding our novel AStar-alike algorithm) generalize our research [FGD+20] published in the scope of INFORE.

4.5 Integration into an Exhaustive Search Algorithm

The Exhaustive Search Algorithm, as its name suggests, exhaustively enumerates the set of possible solutions (physical workflows) and computes the yielded performance with respect to the optimization objectives for each. To better explain the operation of the algorithm, let us proceed with an example. For exhibition purposes, we assume only 2 objectives and a fixed value μ for the parallelization degree at each site. Furthermore, we assume we are given a logical workflow as the one in Figure 15 to be executed over 3 sites. The Exhaustive Search Algorithm will first perform a topological sort of the logical workflow and then enumerate the set of possible physical workflows, i.e., solutions. In the example of Figure 15 we have 5 operators to be executed over 3 sites.

In Figure 16 we see, in blue dots, the performance estimations we derive with respect to the pair of optimization objectives upon placing Op_1 at each of the three available sites. Similarly, in Figure 17 for Op_2 . Figure 18 shows that having computed the performance per optimization objective for the upstream operators of Op_3 , the Exhaustive Search Algorithm will enumerate all possible physical plan executions for Op_3 in combination with Op_1 and Op_2 and compute their overall (up to this point of the topologically sorted graph) performance with respect to the optimization objectives. So, for instance, in Figure 18 at the frame termed “Site 1 plans” we have set Op_3 to be executed at S_1 and we examine the overall performance with respect to the optimization objectives for alternative placements of its upstream operators at the lower level of the topologically sorted, logical workflow. Commutatively, the alternative solutions that the downstream operator of Op_3 , which is Op_5 in Figure 15, will have to consider, after including itself in the search space, are the blue dots for each site in Figure 18.

 <p>Project supported by the European Commission Contract no. 825070</p>	<p>WP5 T5.1 & T5.2 Deliverable D5.1</p>	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

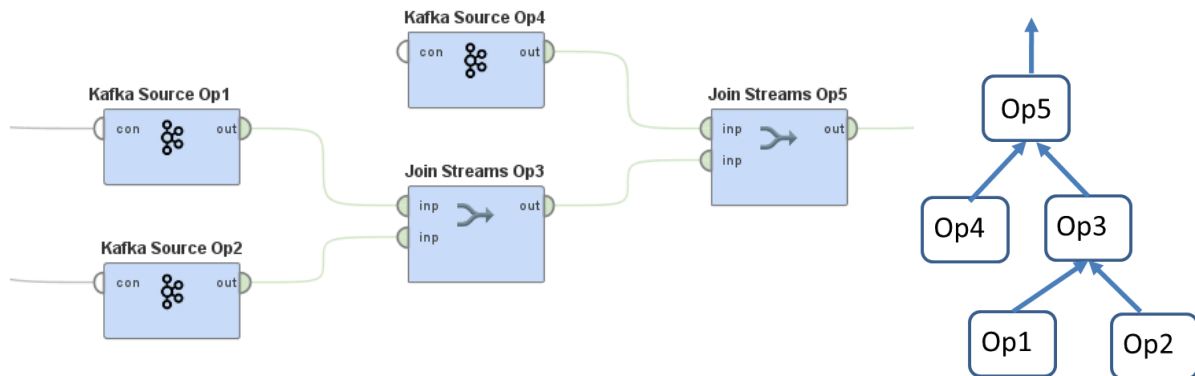


Figure 15: Exemplary logical workflow drawn in RM Studio (left) and abstractly as a graph (right), used in our running example.

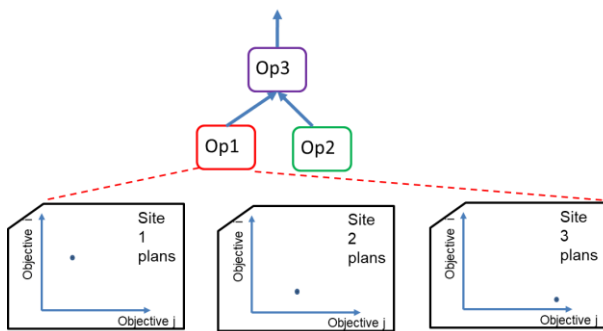


Figure 16: Performance for Op₁ with respect to the two objectives of our running example, upon placing it at various sites. Performance values correspond to the blue dots.

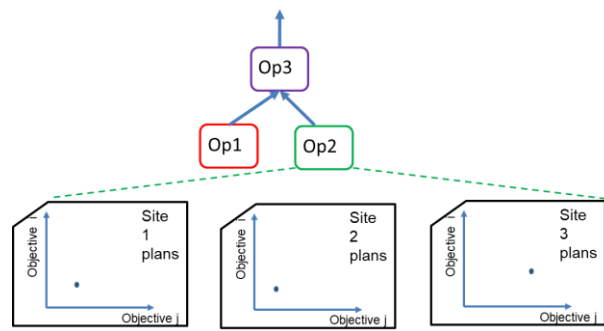


Figure 17: Performance for Op₂ with respect to the two objectives of our running example, upon placing it at various sites. Performance values correspond to the blue dots.

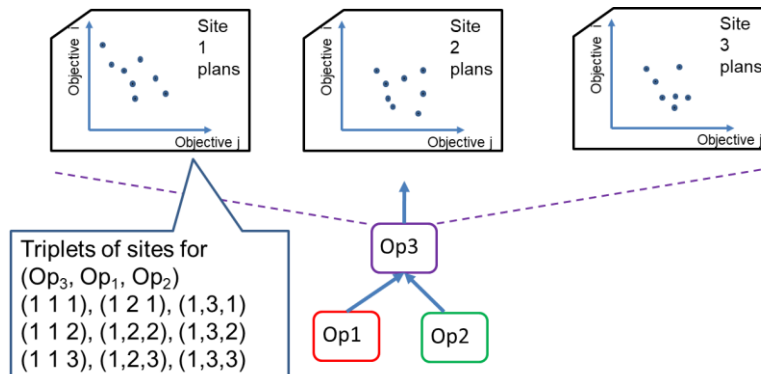



Figure 18: Performance for Op₃ with respect to the two objectives of our running example, upon combining alternatives for placing it at various sites along with alternative such placements of its upstream operators Op₁, Op₂. Performance values correspond to the blue dots. Each blue dot is examined in separate iterations of the algorithm.

Remarkably, each of these solutions (blue dots) for Op₃ will be examined at separate iterations of the algorithm, i.e., each iteration examines just one blue dot. That is, the Exhaustive Search Algorithm has an outer loop iterating over the set of all possible sites and parallelization degrees to produce a physical workflow's execution plans. It fixes a placement picked from that set for the whole workflow and then computes the yielded values of the optimization objectives. Finally, when the algorithm reaches the logical OR operator, it computes a single solution with the minimum cost as stated in Section 4.4. Note that this solution can only lie in the Pareto front computed for OR.

 <p>Project supported by the European Commission Contract no. 825070</p>	<p>WP5 T5.1 & T5.2 Deliverable D5.1</p>	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

Other solutions that do not lie in this Pareto front will be dominated by those in the front and their cost values will be worse.

The pseudo-code in Algorithm 1 formally presents the operation of our Exhaustive Search Algorithm. The algorithm starts by creating a topologically sorted list of all operators (connected via the top-level, virtual OR operator) of the logical workflow (Line 2) and then proceeds by iterating through all possible operator placement and parallelization degree combinations (Line 3). Based on the already fixed snapshot, where operator placement and degree has been decided, a new, overall plan is initiated (Line 4). Then, the algorithm iterates for each operator (Line 5) and the actual performance of each operator can be found by iterating over the topologically sorted list of operators, additionally taking into account all sharing dependencies (Lines 6-10). Topological sorting is used so as to ensure that each operator is examined after its upstream operators. Upon the iteration reaches the top level OR operator, the computed solution satisfies all input constraints and the cost of the currently computed plan is lower than the previously optimal one, stored in $CS_{0,0}^1$ (Line 9), the algorithm keeps the current plan as the new optimal (Line 10) and accordingly resets the rest of $CS_{i,j}^\mu$ with the current placement and parallelization degree for each operator (Line 12). Notice that for each operator we just keep in memory one candidate solution, which is the one that gives the minimum cost at the top-level OR. Therefore, $cs_{i,j}^\mu \equiv CS_{i,j}^\mu$ (Line 1).

Algorithm 1: Exhaustive Search Algorithm


```

1 Initialize all  $cs_{i,j}^\mu \equiv CS_{i,j}^\mu$  sets to empty
2 List topSortList = topologicalSort(LogicalWorkflow)
3 foreach  $placement \in setOfPossiblePlacementAndParallelDegreeCombinations$  do
4   Plan p = new Plan()
5   foreach  $op \in topSortList$  do
6      $p.add(opIndex, S_j = placement.siteOf(opIndex), \mu = placement.parallelDegreeOf(opIndex))$ 
7      $copse = findCoplacedOperatorsSharingUpstreamOps(opIndex,$ 
         $topSortList, placement)$ 
8      $p.computePerformancePerObjective(p.inputPerformancePerObjectiveFor(opIndex), cpose)$ 
9     if  $opIndex=0 \wedge p.satisfiesInputConstraints() \wedge p.isOptimalIn(CS_{0,0}^1)$  then
10      newOptimalPlan p
11 if newOptimalPlan then
12   reset all  $CS_{i,j}^\mu$  according to p

```

Algorithm 1: The Exhaustive Search Algorithm

In general, for $|Op|$ operators and $|S|$ sites we would have $O(|Op|^{S|})$ alternatives and for different parallelization degrees, not above Π , the complexity of the search space would increase to $O((|Op| \cdot \Pi)^{S|})$. We further have an $|Op|$ cost for computing the performance of each operator, which yields a total of $O(|Op| \cdot (|Op| \cdot \Pi)^{S|})$ running time for the Exhaustive Search Algorithm.

	Project supported by the European Commission Contract no. 825070	WP5 T5.1 & T5.2 Deliverable D5.1	Doc.nr.:	WP5 D5.1
			Rev.:	1.0
			Date:	30/04/2020
			Class.:	Public

5 Advanced Optimization Algorithms

5.1 The A*-like Algorithm


The Exhaustive Search Algorithm we presented in Section 4.5 is a baseline algorithm that provides optimal solutions to our optimization problem by enumerating all possible computer cluster, Big Data platform, i.e., site, and parallelization degree combinations. It does not make a distinction between these configurations at runtime. In other words, it cannot diagnose that one combination (candidate solution) is more promising to yield an overall Pareto optimal execution plan than another. This is because it fixes all possible instances of a candidate physical workflow and only then computes its performance. We now propose an advanced algorithm that can drastically reduce the number of examined physical execution plans in finding a proper solution to our optimization problem. This novel algorithm is a non-trivial variant of the A* algorithm [HNR68].

As is, the A* algorithm is not suitable for our needs. This is because it can only compute shortest (of minimum cost) paths for the input graph (logical workflow). This is not acceptable in our design since it implies that the computed execution plan would leave some operators out and thus it would not instruct physical operator implementations for all logical operators of the input workflow. Therefore, we must significantly redesign the original algorithm. We are going to explain the functionality of our new algorithm in terms of our workflow optimization setup.

For the new algorithm, besides the virtual OR operator, we add another virtual START operator which draws an edge towards source operators of the logical workflow, i.e., operators with no upstream operators. Our A*-like algorithm does not examine pre-fixed plans with respect to the placement of operators. It begins with the newly added START operator and dynamically explores the search space of available options based on which partial (up to the level of the workflow explored so far) solution – placement is more promising to yield an overall physical workflow of the lowest cost. What we keep from the original A* algorithm is the basic principle that makes it practically efficient. This is the fact that instead of computing, at each operator and level it examines in the logical workflow, only the Real Cost of reaching that operator from a source (without upstream) operator, it also takes into account two additional costs. The first one is a Heuristic Cost, which is the (underestimated as explained shortly) cost from the currently examined candidate physical operator to the destination one. The destination operator is the virtual OR we add, as in the Exhaustive Search's case. In our setting this heuristic cost is an estimation of the weighted combination (Section 4.4) of the optimization objectives from the examined operator, each time, to the virtual OR. The second cost we account for, is an Estimated Cost, which is the addition of the Real Cost and the Heuristic Cost. So, in all, we have three cost functions for real, heuristic and estimated costs that our algorithm should compute each time it examines a new physical operator assignment to a site along with its parallelization degree. All cost values can be computed using our cost estimator (Section 4.3).

Figure 19 schematically depicts the operation of our A*-like algorithm when it reaches an operator Op_3 , having started from the START operator. Roughly speaking (the exact algorithm will be presented shortly), Op_3 computes the Real Cost for each site configuration. This real cost is the cost of each physical plan for the part of the workflow from START up to Op_3 and is shown with the black-framed plots for site 1 and site 3 in the figure. Besides the real cost, Op_3 will consult the Estimated Cost, i.e., the additional cost to reach the OR operator from Op_3 is added to the real cost for all possible solutions in the black-framed plot. The red-framed plot in the figure shows the Estimated Cost for each such solution. This expresses how promising an overall (from START to OR) physical plan is predicted to be, when we make alternative choices for Op_3 and its upstream operators. Op_3 will then insert the candidate solutions of black-framed plot in a priority queue, based on increasing estimated cost. When the algorithm examines the downstream operators of Op_3 it will prioritize and examine those promising solutions first. When a solution reaches the OR operator, as we move upwards the logical workflow, it is added in the $CS_{0,0}^1$. And when a solution with higher estimated cost than the minimum cost of a solution in $CS_{0,0}^1$ is dequeued from the priority queue, the algorithm completes. Then, all solutions that remain in the priority queue are pruned from the search space of the algorithm. Thus, the alternative physical plans examined by the algorithm, though equivalent to the exhaustive search in the worst case, are practically by far fewer.

As holds for the original A* algorithm, to guarantee the optimality of the overall prescribed solution (physical workflow), the Heuristic Cost must be consistent and admissible. Admissible means that the Heuristic Cost should never overestimate the real cost. In a naive execution of the algorithm, we should be able to set all heuristic costs to zero and that would simulate the execution of the Exhaustive Search Algorithm. Consistency refers to monotonicity.

 <p>Project supported by the European Commission Contract no. 825070</p>	<p>WP5 T5.1 & T5.2 Deliverable D5.1</p>	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

Consider two operators u, v which are connected with an edge from u to v . A heuristic is consistent if $\text{Heuristic Cost}(u) \leq \text{Heuristic Cost}(v) + \text{Real Cost}(u, v)$.

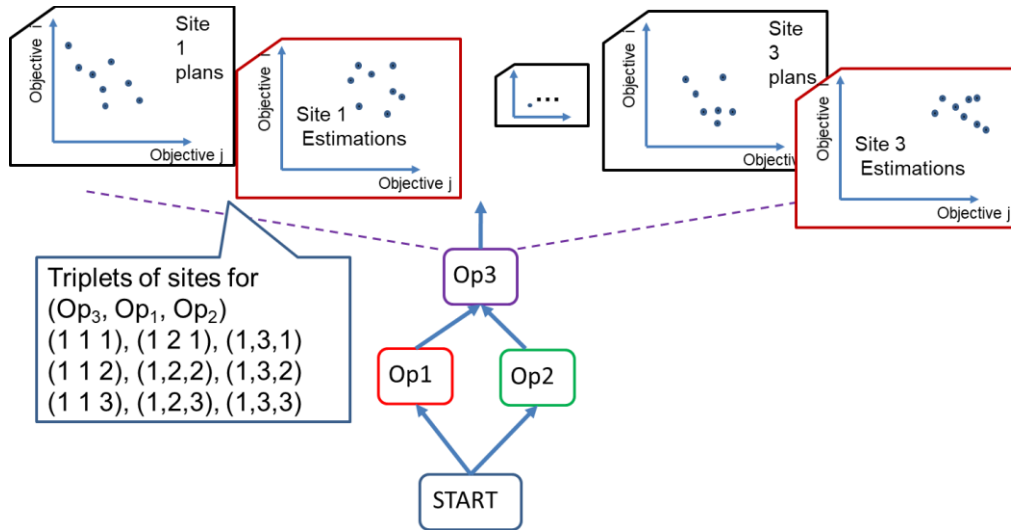



Figure 19: Rationale of our A*-like Algorithm. Among all solutions available in the black-framed sets of Op_3 , which shows the **RealCost** of partial plans (beginning from **START** and reaching Op_3) that have been considered so far, our algorithm will first examine those for which we estimate that will be overall more efficient i.e., the cost of the overall physical plan formed when the currently built plan reaches the OR operator will be lower. The decision is made based on **EstimatedCost** (red-framed set of solutions in the figure). The solutions that are most promising will have a lower **EstimatedCost** and will be placed first in a respective priority queue.


Algorithm 2 presents the pseudocode for our A*-like Algorithm. The main body of the algorithm lies in Lines 1-22, while Lines 23-35 and Line 36-40 sketch important methods utilized by the main algorithm. Let us first discuss what these methods do. The `computeHeuristicCosts` method receives as input a topologically sorted graph. That will be a topologically sorted version of the logical workflow in the main algorithm. It accesses the sorted logical workflow graph in reverse topological order and, thus, examines each operator op one by one starting from the OR operator (Lines 24-35). For each examined logical operator op , it initiates its Heuristic Cost to zero (Line 25) and then iterates through all its downstream operators (Lines 26-33). For each such downstream operator $opPrev$, it examines a particular site for which a physical operator (implementation) for $opPrev$ exists and sets a possible parallelization degree (Lines 28-33). It predicts the cost of $opPrev$ for the particular physical operator using our cost estimator and keeps the minimum cost for the physical instantiations of $opPrev$ examined so far (Line 31). Having examined all available site, parallelization degree pairs for $opPrev$, the algorithm proceeds with adding the **HeuristicCost** of $opPrev$ to the computed minimum (Line 32). The **HeuristicCost** for op is computed as the maximum $\text{minCost} + \text{HeuristicCost}$ of its downstream operators (Line 33). In case, we have reached the **START** operator we set its **HeuristicCost** to zero (Lines 34-35).

Let us now examine the functionality of the `selectOpToAdd` method in Lines 36-40. The method is used to select the operator that will be added to a partial plan that is being built by the main algorithm. It receives as input that partial plan (which is a graph of physical operators, instantiating logical ones for part of the logical workflow) and the logical workflow itself. It performs a graph subtraction operator which leaves in the produced graph, called **remaining** (Line 38), those logical operators without physical mapping in the plan. Among those, the algorithm selects the ones that have no upstream operator in the remaining graph. Practically, this says that we examine only operators in remaining that have all their upstream operators in the plan. Among the latter, refined set of candidate operators, the method returns the most seemingly efficient one, i.e., the one whose estimated cost upon added to the plan's real cost is minimum (Line 40). Note that the set subtraction operation is used to ease the presentation. Operators that do not have all their upstream operators in the plan can be computed directly from the topologically sorted logical workflow produced by the main algorithm. We are now ready to proceed with the description of the operation of the main, A*-like Algorithm.

 <p>Project supported by the European Commission Contract no. 825070</p>	<p>WP5 T5.1 & T5.2 Deliverable D5.1</p>	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

The A*-like Algorithm begins with initiating all $CS_{i,j}^{\mu}$ and the Estimated Cost-based priority queue (Line 1). It performs a topological sort of the logical workflow in Line 2 and computes the HeuristicCost of each operator of the logical workflow calling the respective method we already described. Notice that the HeuristicCost is the same irrespectively of the physical instantiation (physical operator) corresponding to the logical operator. In Line 4, the algorithm initializes a plan that includes START, it is placed at the query source (of zero index as is the case with OR) and admits a parallelization degree of 1. Then, the algorithm starts exploring the priority queue in Lines 5-22. Until the queue is not empty and the next plan in the queue does not have an Estimated Cost greater than the minimum Real Cost of a, if any, complete plan (reaching the OR operator, therefore the $CS_{0,0}^1$ – Line 5) the plan with the minimum Estimated Cost is dequeued (Line 6). In Line 7 the algorithm calls the selectOpToAdd method to select the next, preferable operator op that should be used to expand the plan with a physical implementation of an operator that lies at the next level of the topologically sorted logical workflow. For each possible configuration for op (loops in Lines 8-22) a new plan p' that includes such a physical operator is created (Lines 10-11). In other words, the algorithm creates as many new solutions as the number of combinations of available implementations of the dequeued operator and admissible parallelization degrees. In Lines 12-13 we compute the EndingOperators for p'. We term EndingOperators, the operators that are included in p', but some of their downstream operators are missing from p'. For each operator in the EndingOperatorSet of p' (Lines 14-16), we compute its Real Cost (to reach the EndingOperator from START) and its Estimated Cost (to get to the OR operator). The Estimated Cost for the partial plan p' we build, is the maximum of the EstimatedCost of its EndingOperators (Line 17). If op is the OR operator and the optimization constraints are satisfied, the whole plan is included in the set of candidate physical plans and all sets of the rest of the operators $CS_{i,j}^{\mu}$ are updated accordingly (Lines 19-20). Otherwise, p' receives a place in the priority queue (Line 22). Note that $CS_{0,0}^1$ includes only one solution at each time, the one that minimizes the cost of our optimization problem. Moreover, note that for simplicity we did not refer in our description to computing the optimization objectives separately, as we did in the Exhaustive Search Algorithm. These are given by our cost estimator and then we proceed with computed the weighed cost each time we need it.

As already commented for Line 5, the algorithm will stop when it peeks the plan at the head of the queue and sees that its EstimatedCost is greater than the cost of the current optimal solution in $cs_{0,0}^1 \in CS_{0,0}^1$. The plans that remain in the queue after that, are pruned altogether from the examined search space of possible physical execution plans.

 <p>Project supported by the European Commission Contract no. 825070</p>	<p>WP5 T5.1 & T5.2 Deliverable D5.1</p>	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public


Algorithm 2: A*-like Algorithm

```

1 Initialize all  $CS_{i,j}^\mu$ , set prQueue to empty
2 List topSortList = topologicalSort(LogicalWorkflow)
3 computeHeuristicCosts(topSortList)
4 prQueue.enqueue(new Plan(START,0,1)) // priority based on minimum Estimated Cost
5 while (!prQueue.isEmpty  $\wedge$  !(prQueue.peek().EstimatedCost > minCostIn( $CS_{0,0}^1$ )))
   do
6   Plan p=prQueue.dequeue()
7   op=selectOpToAdd(p,LogicalWorkflow)
8   foreach  $S_j \in S \wedge S_j.hasImplementationFor(op)$  do
9     foreach  $\mu \in possibleParallelDegreeFor(op)$  do
10      Plan  $p'=p$ 
11       $p'.add(op.opIndex,j,\mu)$ 
12      // EndingOperators in  $p'$  are all operators lacking some downstream operator in  $p'$ 
13       $p'.updateEndingOperatorSet(op, topSortList)$ 
14      foreach  $eop \in p'.EndingOperatorSet()$  do
15         $p'.EndingOperatorSet(eop).computeRealCost()$ 
16         $p'.EndingOperatorSet(eop).computeEstimatedCost()$ 
17       $p'.EstimatedCost=\max(p'.EndingOperatorSet().EstimatedCost)$ 
18      if  $op.opIndex = 0 \wedge p'.satisfiesInputConstraints()$  then
19         $CS_{0,0}^1.add(p')$ 
20        Update all  $CS_{op.opIndex,j}^\mu$ 
21      else
22        prQueue.enqueue( $p'$ )
23 Procedure computeHeuristicCosts (topSortList)
24   foreach  $op \in reverse(topSortList)$  do
25     op.HeuristicCost=0
26     foreach  $opPrev \in op.downstreamOperators$  do
27       minCost=0
28       foreach  $S_j \in S \wedge S_j.hasImplementationFor(op)$  do
29         foreach  $\mu \in possibleParallelDegreeFor(opPrev)$  do
30           ownCost=predictCost(opPrev,  $S_j, \mu$ ) //using the Bayesian Estimator
31           minCost=minCost<ownCost?minCost:ownCost
32       minCost=minCost+opPrev.HeuristicCost
33       op.HeuristicCost=op.HeuristicCost>minCost?op.HeuristicCost:minCost
34   if  $op=START$  then
35     op.HeuristicCost=0
36 Procedure selectOpToAdd (plan, workflow)
37 //Consider only operators with all upstream operators in plan
38 remaining=workflow \ plan
39 forall  $candOp \in remaining \wedge |candOp.upstreamOperators| = 0$  do
40   return candOp of minimum plan.RealCost+candOp.EstimatedCost

```

Algorithm 2: The A*-like Algorithm

	Project supported by the European Commission Contract no. 825070	WP5 T5.1 & T5.2 Deliverable D5.1	Doc.nr.:	WP5 D5.1
			Rev.:	1.0
			Date:	30/04/2020
			Class.:	Public

5.2 Designed Algorithms

In this section we outline the operation of algorithms we have designed and will be incorporated in the optimization process. So far, we have incorporated the Exhaustive Search Algorithm which does not scale well when the examined site, parallelization degree combinations increase and the A*-like Algorithm which examines fewer plans due to the utilized heuristic. Here, we present Dynamic Programming-alike, Heuristic and Greedy algorithms. The Dynamic Programming Algorithm improves the worst-case performance of the Exhaustive Search, but still may not be satisfying from a scalability perspective. Therefore, in the Heuristic Algorithm we attempt to reduce the number of examined sites for placing an operator and in the Greedy Algorithm we design, besides employing a heuristic, we also keep one Pareto optimal solution for each operator (the one with the minimum cost) before evaluating its downstream operators. As we move from Exhaustive Search to the Dynamic programming and to the Heuristic and Greedy algorithms, what we achieve is to design algorithms with progressively improved computational complexity. On the other hand, these algorithms do not necessarily provide optimal solutions to our optimization problem but attempt to approximate the optimal one in a best effort fashion.

5.2.1 Dynamic Programming-alike Algorithm

In a nutshell, the difference between the Exhaustive Search and the Dynamic Programming Algorithm we introduce in the current section lies in the number of possible solutions an operator examines from its upstream operators. More precisely, recall from Section 4.5 and Figure 16, Figure 17, Figure 18 that Op_3 will receive and consider all the blue dotted performances for solutions with respect to the two objectives in our running example. In addition, Op_3 will convey to the downstream operator Op_5 (see Figure 15) all the blue dotted performances for possible solutions, shown at the top of Figure 18, in separate iterations of the Exhaustive Search Algorithm. The difference of the Dynamic Programming-alike Algorithm we introduce, is that Op_5 will only examine solutions that lie at the Pareto front of Op_3 in Figure 18 and will visit Op_3 only once while traversing the topologically sorted logical workflow. This is illustrated in Figure 20 where only the solutions that correspond to blue-dotted performances for each site will be examined at Op_5 's level, while red-dotted ones will not.

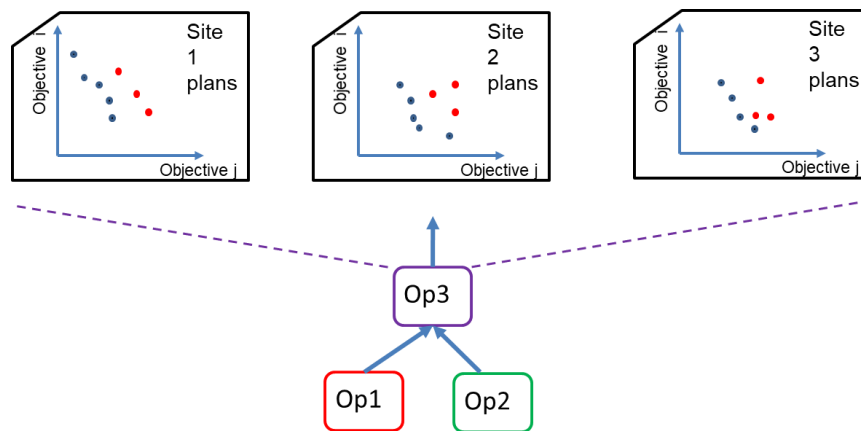



Figure 20: Contrary to the Exhaustive Search Algorithm in Figure 18, in the Dynamic Programming Algorithm, the downstream operator Op_5 (see Figure 15) of Op_3 will not examine alternatives for Op_3 that exhibit performances corresponding to the red dots in this figure. Only the blue ones, lying at the Pareto front of Op_3 , for each site will be examined at Op_5 's level.

Algorithm 3 presents our Dynamic Programming-alike Algorithm. The algorithm first sorts the operators in the logical workflow using a topological sort. The sortedList in Line 3 includes the result of the topological sorting. Then, it computes the Pareto optimal plans, calling the BuildPlans procedure, for each operator op (Line 5) using this sort order, which ensures that an operator is processed after all its input operators. The BuildPlans procedure (Lines 6-15) takes as input an operator op to process, along with its index $opIndex$ in the sortedList, and considers all potential (and valid with respect to available operator implementations) placements (Line 7) and all possible parallelization degree configurations (Line 8) for this operator. For each such combination of operator location S_j and parallelization degree μ , the procedure then iterates through all possible combinations of solutions computed at the input (upstream) operators of op (Lines 10-11) and computes their performance for each optimization objective

 <p>Project supported by the European Commission Contract no. 825070</p>	<p>WP5 T5.1 & T5.2 Deliverable D5.1</p>	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

(Line 12). Each computed candidate solution p is checked to see if it satisfies the input constraints (Section 4.4) and for Pareto optimality (Line 13) within the corresponding $CS_{i,j}^\mu$ set. If so, p is inserted into the set (Line 15), removing candidate solutions that were dominated by p (Line 14).

Let I denote the maximum number of upstream operators an operator in the logical workflow may have and let T denote the maximum number of Pareto optimal plans that is kept per placement and operator. Our Dynamic Programming-alike Algorithm makes $O(|S| \cdot I)$ iterations per operator op . Each iteration considers all combinations with candidate solutions at input operators of op , which are $O((|S| \cdot T)^I)$. This yields a total of $O(|Op| \cdot I \cdot |S|^I \cdot T^I)$ combinations for all operators, placements and parallelization degrees. The space complexity in this case will be $O(|Op| \cdot |S| \cdot T)$.

Contrary to the approach employed in Exhaustive Search Algorithm, the Dynamic Programming we introduce here does not examine all possible combinations of operator placements and parallelization degrees, but, for each operator, it considers only the physical instantiations of its upstream operators that exhibit Pareto optimal performance. Due to this behavior, this algorithm retains the property of optimality when either (i) there is no upstream operator sharing (see end of Section 4.3) or (ii) the considered objectives with non-zero weights in the optimization problem setup of Section 4.4 have an aggregative performance computation formula that is not duplicate sensitive. This holds for O_1 :Throughput which is computed based on a minimum value.

Algorithm 3: Dynamic Programming Algorithm


```

1 Initialize all  $CS_{i,j}^\mu$  sets to empty
2 Procedure CreatePlans (LogicalWorkflow)
3   List sortedList = topologicalSort(LogicalWorkflow)
4   foreach  $op \in sortedList$  do
5     BuildPlans ( $op, op.index$ )
6 Procedure BuildPlans ( $op, opIndex$ )
7   foreach  $S_j \in S \wedge S_j.hasImplementationFor(op)$  do
8     foreach  $\mu \in possibleParallelDegreeSetFor(op)$  do
9       Plan  $p = new Plan(opIndex, S_j, \mu)$ 
10      inputPoList = All combinations of solutions from  $op$ 's upstream operators
11      foreach  $subPlanSet \in inputPoList$  do
12         $p.computePerformancePerObjective(subPlanSet)$ 
13        if  $p.satisfiesInputConstraints() \wedge p.isParetoOptimalIn(CS_{opIndex,j}^\mu)$ 
14          then
15            Remove from  $CS_{opIndex,j}^\mu$  solutions dominated by  $p$ 
16             $CS_{opIndex,j}^\mu.add(p)$ 

```

Algorithm 3: The Dynamic Programming-alike Algorithm

Let us see the reason why the Dynamic Programming-alike Algorithm loses its optimality in any other case. Consider a shared operator as the one termed BL in Figure 1. It propagates its Pareto optimal solutions to both its downstream operators, but these operators will run their own BuildPlans procedure without knowing anything about sharing. Thus, each will compute the communication cost, latency of BL separately and, when the OR operator is reached, the respective cost will have been included in each objective's performance calculations, i.e., twice. For the communication cost, for instance, this is invalid. On the contrary if we compute a performance objective such as the minimum throughput of an operator in the workflow (which we seek to maximize), this performance calculation duplication does not affect the overall performance computed at the top-level OR.

 <p>Project supported by the European Commission Contract no. 825070</p>	<p>WP5 T5.1 & T5.2 Deliverable D5.1</p>	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

5.2.2 Heuristic Algorithms

The basic principle behind the algorithms we introduce in this section is that we use intuitive heuristics so as to (i) prune the number of sites (recall that this corresponds to <computer cluster, Big Data platform> pairs for which implementations for a given operator exist) the algorithm examines for placing the physical execution of operators and (ii) limit the number of configurations with respect to the parallelization degree of each operator.

In order to limit the number of examined sites our algorithms compute two sets. The first set is called Candidate Centers (CCs) and involves sites for which, if we determine to place – assign the physical execution of an operator there, optimize at least one of the individual objectives of our problem setup (Section 4.4). The CCs are intuitively considered as good starting points from which a heuristic that prunes examined sites should begin with. The idea is that the more we leave the neighborhood of candidate centers the worse some of the optimization objectives are expected to become and thus respective solutions are pushed away from the Pareto front. So, it is considered “safe” not to examine sites away from such neighborhoods. We also have a set of Candidate Locations (CLs) which will include sites that are not CCs but are included in the search space because they lie in the neighborhood of CCs and can also yield Pareto optimal solutions upon being examined. For operators with no upstream operators, which are source operators (e.g. Kafka Topics), the CCs and CLs coincide.

For each operator op that receives input only from sources, we set as its CCs the union of the CLs of the input streams of op . We then create a queue of candidate locations (candidateQueue) that initially contains these CCs. For each candidate location in candidateQueue, the algorithm will compute the performance of candidate solutions for op , starting with a default parallelization degree μ_0 that is possible for that operator (i.e., we leave the parallelization degree to be set by the intrinsic optimizer of the Big Data platform), for each solution. Each candidate solution that is Pareto optimal is examined further, expanding our search in two ways. First, the parallelization degree of the Pareto optimal candidate solution is expanded to look for solutions that potentially give new Pareto optimal solutions that satisfy the input constraints. Second, since it seems as an intuitive idea that op should be executed somewhere “in-between” its CCs, we consider a site to be in the vicinity of the CCs, if the optimization objective(s) that are optimized at the individual CCs do not get worse compared to a situation where we place the examined operator at either one of the CCs. We say, then, that such sites lie in the “vicinity” of the candidates. Given this, we expand the area of search with the potential insertion of such neighboring sites of the candidate location to the candidateQueue. The process ends when examining various, beyond the default, parallelization degrees cannot provide any more Pareto optimal solutions (that satisfy the input constraints) for op and when all candidate locations in the candidateQueue list have been processed. At that point, the locations corresponding to the remaining Pareto optimal solutions constitute the Candidate Locations for op . The algorithm for operators that have as inputs operators that are not stream sources is the same, since we just mentioned how we compute the CL list of an intermediate operator in the logical workflows. Formal descriptions for our Heuristic and Greedy algorithms are presented later on in Algorithm 4. For improved readability, Algorithm 4 does not include the details of determining CLs, but we just described their formation.

- Latency (S_4, S_9) = $7 + 7 = 14$
- Latency(S_7, S_9) = $7 < 14 \rightarrow S_7$ is in the vicinity of S_9
- No other site enters the vicinity of S_9, S_4
- S_7 may allow S_5 in its vicinity: Latency (S_9, S_5) = $12 < 14$

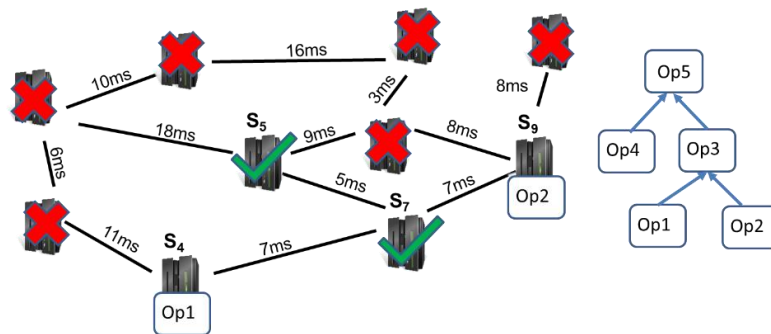


Figure 21: Example of candidate location set formation upon examining the (network) latency as an optimization objective. Network latency values are tagged on the depicted communication links.


 <p>Project supported by the European Commission Contract no. 825070</p>	<p>WP5 T5.1 & T5.2 Deliverable D5.1</p>	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

Figure 21 presents an example on how candidate locations are determined having set the location of Op_1 , Op_2 which in the example are considered as source operators. S_4 executes Op_1 while S_9 executes Op_2 , which are the respective CCs. On the right of the figure the overall logical workflow is depicted. For ease of exposition, assume that we examine one objective, that of (network) latency. The question is where to evaluate operator Op_3 that receives input from the sources. Popping Op_1 's CCs from the candidateQueue we will examine its two neighbors. Among them, only S_7 has a latency value that does not exceed the latency between S_4 and S_5 . Therefore, S_7 will be included in the candidateQueue. Then, we pop S_9 for which again, only S_7 does not surpass the latency constraint. Then S_7 is popped out of the queue which will in turn admit S_5 in the candidates. When S_5 is popped, no other site can be included in the candidate locations' set due to violating the latency constraint.

Let us return to our running example and initially see how our Heuristic algorithms would work in this simplified case. We have two heuristic variations that we propose. The first one, termed Heuristic, prunes examined sites according to the aforementioned rationale and only uses the default μ_0 parallelization degree. The second one, termed Heuristic+, prunes sites in the same way as Heuristic, but also examines alternative parallelization degrees that reach the Pareto front. The difference between the two approaches is illustrated in Figure 22 and Figure 23, correspondingly.

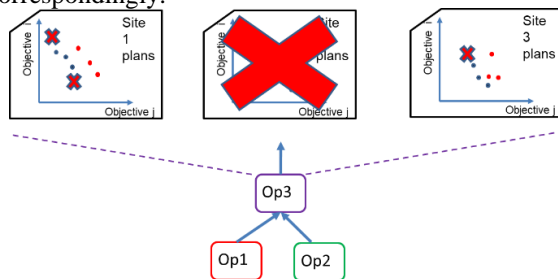


Figure 22: The Heuristic variant prunes examined sites and uses a default parallelization degree. It can lose pareto optimal solutions (marked with X) from the solution set of the sites it examines as well. Op_3 will provide to the downstream operator Op_5 (see Figure 15) only the blue – dotted solutions of its (even locally partial) Pareto front.

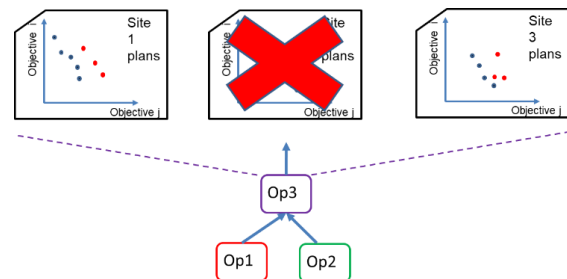



Figure 23: The Heuristic+ variant just prunes the examined sites. Op_3 will provide to the upstream operator Op_5 (see Figure 15) only the blue dotted solutions of its (locally complete) Pareto front.

5.2.3 Greedy Algorithms

Besides the Heuristic variant we also introduce two Greedy variants. The first variant, termed Greedy+, does what Heuristic+ does, but keeps only one Pareto front per operator. In other words Greedy+ will merge the two plots of Figure 23 and communicate the newly formed Pareto front to the downstream operator of Op_3 . The second variant, termed Greedy, will do what the Heuristic variant does, but keep only one solution from the Pareto front to convey to the downstream operator of Op_3 . That solution will be the one that minimizes (locally, up to the part of the workflow that reaches Op_3) what we refer to Section 4.4 as cost.

Recall that as we move from Heuristic+ to Heuristic, Greedy+ and Greedy the algorithms are expected to run much faster even for high numbers of networked computer clusters and hosted Big Data platforms. However, it is important to note that since we prune the search space heuristically and greedily, respectively, the fewer solutions are conveyed to downstream operators of the topologically sorted logical workflow, the more likely becomes to reach a situation where the input constraints are not satisfied by any computed solution in the partial Pareto fronts. In such a case, we rerun the optimizer using the next algorithm that is likely to provide more solutions. For instance, if Greedy reaches a situation where no execution plan can provide a solution that satisfies the given constraints, we then try with Greedy+. If the optimizer still cannot find a proper solution we proceed with Heuristic and Heuristic+, is necessary.

 <p>Project supported by the European Commission Contract no. 825070</p>	<p>WP5 T5.1 & T5.2 Deliverable D5.1</p>	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

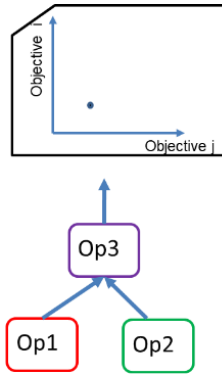


Figure 24: Greedy conveys to downstream operators only the solution with the locally minimum cost among the solutions examined by Heuristic.

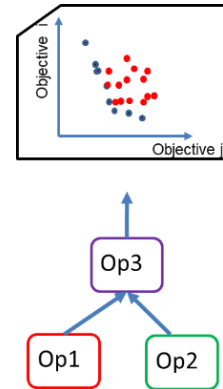



Figure 25: Greedy+ conveys to downstream operators the Pareto front formed from all possible placements at the available sites, i.e., it merges the Pareto fronts of all sites examined by Heuristic+ and provides a new Pareto front of blue-dotted solutions.

Algorithm 4 presents the pseudocode of the discussed Greedy and Heuristic variations. The algorithm is easily configured to one of the proposed variants using a pair of Boolean variables, `isHeuristic`, `isPlus`, to determine the set of solutions that each operator will convey to its downstream operators and the number of examined parallelization degrees. It begins with a topological sort of the logical workflow (Line 3). Then, for each operator in the topologically sorted graph, we build alternative plans (Line 5), after having computed its Candidate Centers (CCs) (in Line 5 as well). In the `BuildPlans` procedure (Lines 6-32) for each CC (initially) and CL (in subsequent iterations) in the `candidateQueue` (Lines 7-24) we pick a site for placement of `op` and we also keep the examined sites in a list (Line 9) so that the Greedy variant can later merge the Pareto optimal solutions computed for each site. Lines 10-11 determine the number of times the loop in Lines 12-20 will be executed by either setting the allowed parallelization degree to a default one or not. If the allowed parallelization degree is set to μ_0 , the loop in Lines 12-20 will be executed only once. Inside that loop, we create a new plan (Line 13) and consider the set of Pareto optimal solutions that are input to this plan by `op`'s upstream operators. The latter set is determined in Lines 25-32 which will be explained shortly. Having acquired the set of input Pareto optimal solutions from `op`'s upstream operators, Lines 15-20 compute the performance of the newly formed plan per objective. If the plan provides a Pareto optimal solution that satisfies input constraints, it is added to the $CS_{i,j}^\mu$ set of `op`. Then, $CS_{i,j}^\mu$ is updated so that candidate solutions that are not Pareto optimal anymore are deleted. We use a Boolean variable `pWasAdded` to distinguish the fact that a new plan for the examined site was added (Lines 17-20). We need `pWasAdded` since if the examined site helped at least once in the loop to produce Pareto optimal solutions, we should expand our search around it for Candidate Locations (Lines 21-24). Note that the same site may be examined multiple times for candidate locations (Lines 21-24) may be executed for different parallelization degrees) since changing the parallelization degree may render the plan/site optimal (or not) for different optimization objectives. If the `isHeuristic` variant is not false (i.e., we run a Greedy variant), we need to merge the $CS_{i,j}^\mu$ of the examined sites to one Pareto front (Lines 25-32) and if we also run Greedy (instead of Greedy+), we need to keep only the solution with the minimum cost (see Section 4.4) for `op` (Lines 31-32). The algorithm terminates when the `CreatePlans` procedure finishes examining candidate solutions for the OR operator (of zero index).

	Project supported by the European Commission Contract no. 825070	WP5 T5.1 & T5.2 Deliverable D5.1	Doc.nr.:	WP5 D5.1
			Rev.:	1.0
			Date:	30/04/2020
			Class.:	Public


Algorithm 4: Heuristic, Greedy and Plus Algorithmic Variants

```

1 Initialize all  $CS_{i,j}^\mu$  sets to empty
2 Procedure CreatePlans (LogicalWorkflow, isHeuristic, isPlus)
3   List sortedList = topologicalSort(LogicalWorkflow)
4   foreach  $op \in sortedList$  do
5     BuildPlans ( $op$ ,  $op.index$ ,  $op.findCandidateCenters()$ , isHeuristic, isPlus)
6 Procedure BuildPlans ( $op$ ,  $opIndex$ , candidateQueue, isHeuristic, isPlus)
7   while ( $(S_j = candidateQueue.dequeue()) \neq null$ ) do
8     Boolean pWasAdded=false
9     examinedSitesForOp.add( $S_j$ )
10    if !isPlus then
11      possibleParallelDegreeSetFor( $op$ )= $\mu_0$ For( $op$ )
12      foreach  $\mu \in possibleParallelDegreeSetFor(op)$  do
13        Plan p = new Plan( $opIndex$ ,  $j$ ,  $\mu$ )
14        inputPoList = All unique combinations of solutions from  $op$ 's upstream
        operators
15        foreach  $subPlanSet \in inputPoList$  do
16          p.computePerformancePerObjective(subPlanSet)
17          if  $p.satisfiesInputConstraints() \wedge p.isParetoOptimalIn(CS_{opIndex,j}^\mu)$  then
18            Remove from  $CS_{opIndex,j}^\mu$  solutions dominated by p
19             $CS_{opIndex,j}^\mu.add(p)$ 
20            pWasAdded=true;
21      if pWasAdded then
22        foreach ( $candidate \in S_j.getNeighbors()$ ) do
23          if  $isInTheVicinity(candidate, op.candidateCenters)$  then
24            candidateQueue.enqueue(candidate)
25    if !isHeuristic then
26      Initialize outputPoList to empty
27      foreach  $S_j \in examinedSitesForOp$  do
28        outputPoList= outputPoList  $\cup CS_{opIndex,j}^\mu$ 
29        outputPoList.keepParetoFront()
30        set all  $CS_{opIndex,j}^\mu$  according to outputPoList
31      if !isPlus then
32        set all  $CS_{opIndex,j}^\mu$  to outputPoList.minCostSolution()

```

Algorithm 4: Heuristic and Greedy Algorithmic Variations. Depending on the value of *isHeuristic* and *isPlus*, input Boolean variables, the search space of solutions that is examined is determined.

 <p>Project supported by the European Commission Contract no. 825070</p>	<p>WP5 T5.1 & T5.2 Deliverable D5.1</p>	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

6 Extensions

6.1 Synopsis-based Optimization

The Synopses Data Engine (SDE) Component of the INFOR architecture can be exploited in a number of ways for optimization purposes. In this section we discuss the detailed of synopsis-based optimizations that are admissible by our algorithms.

6.1.1 ...for enhanced horizontal scalability


First, the SDE includes a number of operators as detailed in Section 2.4. For instance, a CountMin sketch provides approximate counts and frequency values for monitored quantities or the Discrete Fourier Transform operator receives a stream and reduces its dimensionality preserving the values of stream similarity metrics, such as correlation coefficient, in the reduced output streams. Different synopses possess different characteristics, but the bottom-line is that synopses attempt to reduce the amount of utilized memory and/or speed up the processing. The latter is because parallel versions of data summarization techniques hosted in the SDE, besides scaling out the computation to a number of processing units, as operators in Big Data platforms typically do, reduce the volume of processed high-speed data streams. Hence, the complexity of the problem at hand is harnessed and execution-demanding tasks are severely sped up. For instance, sketch summaries can aid in tracking the pairwise correlation of streams in space/time that is sublinear in the size of the original streams. The trade-off for reduced resource consumption is that the accuracy of the output stream is controllably compromised, with predefined accuracy bounds.

Hence, one way to exploit the SDE Component for optimization purposes is to treat it as an additional site hosting implementation of operators. These implementations are equivalent in the form of the output they provide but constitute approximate versions of other, exact operators. For instance, a CountMin sketch applied to count the number of trades per stock in a stream of thousands of stocks of the Financial Use Case in INFOR, can use only a fraction of the memory an equivalent exact count operator would use. This is an optimization related to objective O_4 in our problem setup (Section 4.4) and can be directly supported by our algorithms. In fact it is as simple as modifying the optimizer's dictionary (see Section 3.3) to add an additional implementation for a count operator provided the SDE Component (approximate count), besides the supported Big Data platforms (exact count).

Nevertheless, there exist two issues that arise in practice. The operators that are matched in the dictionary, this time, are not precisely equivalent since the one stemming from the SDE provides output streams which are inaccurate to some extent, as described by the theoretical background of each synopsis [KoGD20]. This introduces an additional objective, that of O_6 : Accuracy, in our optimization problem that should be appropriately weighted to compute the overall cost of a physical workflow. The whole cost of a physical workflow/ execution plan is affected, because some objectives such as O_1, O_4 are improved upon using synopses while O_6 is deteriorated. While it is easy to plug-in this new objective in the optimization setup of Section 4.4 and also mute it by assigning a weight of zero if we do not allow approximate output for the design workflow, it is much more complex to impose constraints on the new objective.

Because SDE operators are part of a workflow, having one of them providing, even quality-aware, approximate output may affect the output of the entire workflow in various ways depending on the structure of the logical workflow. For instance, between approximate operators there might exist stream transformations operators that alter, e.g. filter, the approximated stream. Consider the scenario in Figure 26 which performs a join operation which is a commonly used operator in analytic workflows. In the left scenario of the figure, we apply a filter operator after the join. Assume that the filter operator is applied on one of the fields of the upper Kafka sourced stream. Then, the workflow will be equivalent in terms of the output to the one on the right of the figure, where the filter is applied on the stream where the field belongs to. These two logical workflows are equivalent in terms of their output streams. This will be true for any physical execution plan which maps the logical operators to their physical implementations in various computer clusters, Big Data platforms and parallelization degrees. The profound reason for this is that they engage relational algebra operators and we are aware of their equivalent rewritings.

Now consider the physical operators the optimizer may prescribe upon including synopses operators in its search space. In Figure 27, the uniform sample is applied on a different (unfiltered) stream of different cardinality

 <p>Project supported by the European Commission Contract no. 825070</p>	<p>WP5 T5.1 & T5.2 Deliverable D5.1</p>	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

compared to the physical plan on the right of the figure. Although there exist works in the literature that describe the notion of equivalence of workflows engaging synopsis in terms of relational algebra operators [NDJ13], none of them generalizes the discussion to non-relational algebra operators important for INFORE (such as machine learning operators) and for any given synopsis, beyond samples. Thus, such issues involve a broad range of open research topics, for different operator classes and synopsis.

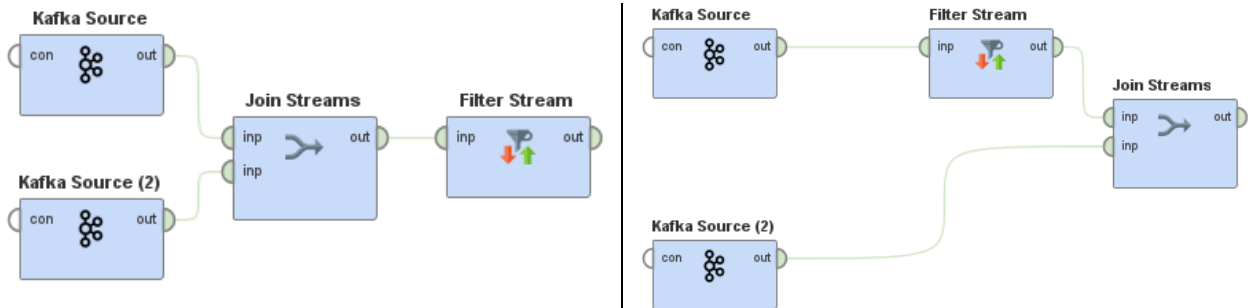


Figure 26: Equivalent Logical workflows drawn on the Graphical Editor Component of INFORE’s architecture. They both apply a filter logical operator based on a field belonging to the stream of the upper Kafka source after the join operation (left) or before (right), respectively.

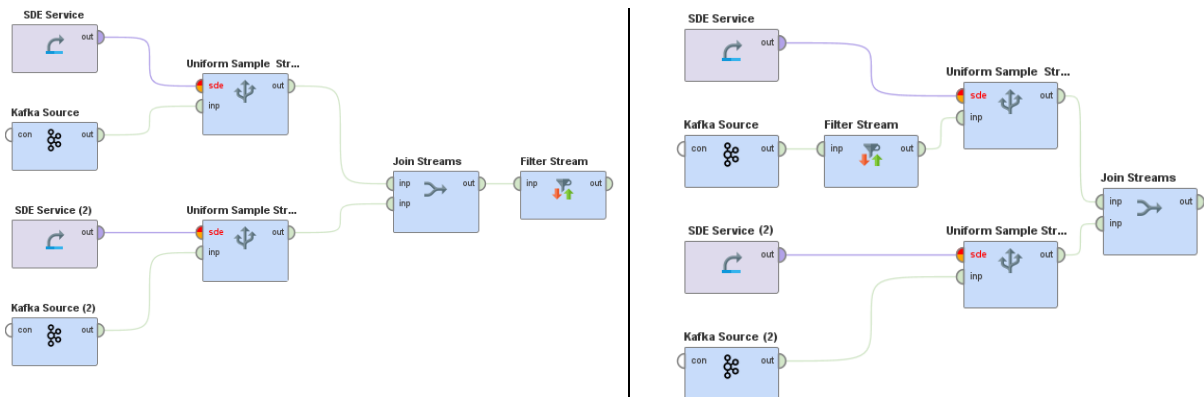



Figure 27: Alternative physical plans for the respective logical workflows. The resulted physical workflows that engage approximations are not directly comparable as far as their structure and accuracy is concerned.

On the other hand, since we can indeed accommodate accuracy in our optimization objectives and synopsis in the prescribed physical plans, employing the SDE would be important to boost interactivity in the execution of workflows at extreme scale. The plausible convention that we make is that we leave examining the notion of equivalence as future work and treat a topologically sorted logical workflow as an entity that cannot be further rewritten or reordered.

But then, we run into another, important issue we need to deal with. It is not easy to figure out how to compute aggregative accuracy values for an entire workflow or parts of it, during the operation of our algorithms, similarly to the way we did in Section 4.3.3. The difficulty comes from the fact that there is no single way to compute aggregated accuracies of streams that have been approximated by different data summarization techniques. Despite the fact that certain synopsis have been proven to be mergeable and thus an overall, aggregated accuracy bound can be provided [ACH+13], the position of the synopsis operator in a given workflow may also affect individual (per synopsis operator) and overall accuracy bounds.

If we try to interpret the above remark, we observe that the unknown aggregated accuracy that we seek is essentially another black box function. The difference between SDE operators and the rest of the supported operators is that since we cannot aggregate the cost per SDE operator, micro-benchmarks are not enough, and a cold start cannot be prevented. However, since we have built models for the exact equivalent operators, we can tolerate a cold start only for their approximate versions. After that, upon a workflow is submitted the optimizer can built up small-scale replicas that will be used for optimization purposes to which it will substitute combinations of exact operators with

 <p>Project supported by the European Commission Contract no. 825070</p>	<p>WP5 T5.1 & T5.2 Deliverable D5.1</p>	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

approximate ones and use the learn-by-example ability of the cost estimator of Section 4.3.2 to build models for the aggregated accuracy in various parts of the workflow and the entire workflow.

To sum up, our optimization algorithms can perform synopsis-based optimization without internal changes. The only thing we should do is to assign a non-zero weight to the newly introduced optimization objective of accuracy O_6 and train Bayesian optimization models to predict the aggregated accuracy during the operation of the algorithms. In that, we can both judge the performance of each examined execution plan for this objective and check the abundance of the plan with any given accuracy constraint.

6.1.2 ...for vertical scalability

There is another form of optimization that can be provided using the SDE Component and is not directly spawned by the optimization algorithms we have discussed so far. The talk regards vertical scalability, i.e., the ability to scale the computation with the number of processed streams. For instance, to detect systemic risks in the Financial Use Case of INFOR, i.e., stock level events that could trigger instability or collapse of an entire industry or economy, requires discovering and interactively digging into correlations among tens of thousands of stock streams. The problem involves identifying the highly correlated pairs of stock data streams under various statistical measures, such as Pearson's correlation over M distinct, high speed data streams, where M is a very large number. To track the full $\Theta(M^2)$ correlation matrix results in a quadratic explosion in space and computational complexity which is simply infeasible for very large M . The problem is further exacerbated when considering higher-order statistics (e.g., conditional dependencies/correlations). The same issue arises in the Maritime Surveillance Use Case for trajectory similarity scores over hundreds of thousands of vessels. Clearly, techniques that can provide vertical scaling are sorely needed for such scenarios.

INFOR's SDE Component specifically provides techniques such as the Discrete Fourier Transform (DFT) and the Locality Sensitive Hashing (LSH) operator which can be inserted by an informed user in a workflow for vertical scalability related optimization purposes, instead of substituting an exact operator with its equivalent version. Indicatively, the coefficients of DFT-based synopses or the number of set bits (a.k.a. Hamming Weight) in LSH-based bitmaps can be used for correlation-aware hashing of streams to respective processing units. Based on the synopses, using DFT coefficients or Hamming Weights as the hash key respectively, highly uncorrelated streams are assigned to be processed for pairwise comparisons at different processing units. Thus, such comparisons are pruned for streams that do not end up together. All these details can be configured graphically in the workflow in a simple way.

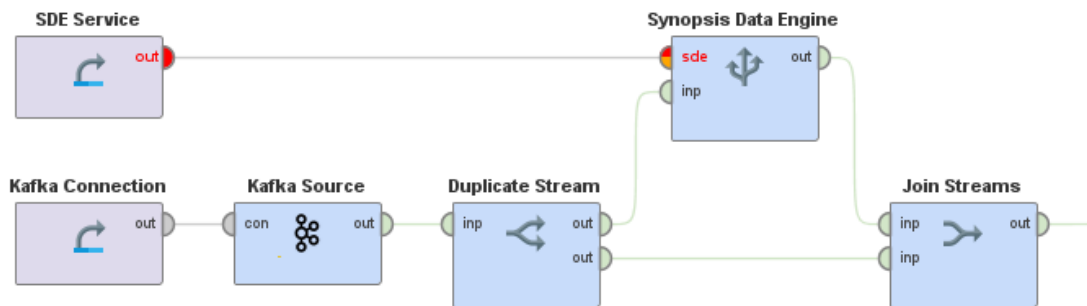



Figure 28: Exemplary workflow utilizing the SDE for vertical scalability

Figure 28 shows such an example of using the SDE operator of DFT for achieving horizontal scalability in the entire workflow. In the figure all input streams arrive at a Kafka topic and we finally (after the Join operator in the figure) want to perform pairwise comparisons for correlation estimation. We duplicate the incoming streams (note that this may be acceptable if we can then prune millions or billions of pairwise comparisons later on in the workflow) directing their original tuples once to the join operator and once to the SDE. The SDE applies the DFT operator configured to return a bucket ID. We then join the bucket ID, stream ID pair provided by the SDE with the respective stream IDs and fields of the original streams. The output of the join can be directed to a reduce/aggregate operator using the bucketID as the key. When the physical operators are submitted as a job all streams with the same key will be directed to the same reducer which will perform pairwise comparisons locally. All streams directed to different buckets, even within the same reducer, will not be compared for similarity. Therefore, the number of performed comparisons is drastically reduced and vertical scalability is achieved by the design of the workflow.

 <p>Project supported by the European Commission Contract no. 825070</p>	<p>WP5 T5.1 & T5.2 Deliverable D5.1</p>	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

Note that in this case the result of pairwise comparisons is not approximate, but exact. The SDE operator is used only to determine the keys. All such optimizations for vertical scalability are orthogonal to the optimization algorithms we presented in Section 4.5 and Section 5.

6.1.3 ...for federated scalability

Recall our discussion in Section 6.1.1 where we pointed out the advantages of defining the approximate versions of operators provided by the SDE, as additional, implemented physical operators for the ones defined in the logical workflow. Substituting exact operators with approximate ones also enhances federated scalability, i.e., the ability to scale the computation in settings where data arrive at multiple geographically dispersed computer clusters each hosting Big Data platforms. Communicating the result of synopses instead of exact operators (i) reduces the communication load in the network of clusters we have available and (ii) reduces the network and read/write to Kafka times, which are ingredients of the overall latency of a physical workflow.

Therefore, substituting exact logical operators with the approximate versions of their physical representatives, besides what we discussed earlier in this section, also affects the optimization objectives of communication, that is O_5 , and overall latency, that is O_2 , in Section 4.4. The reduced communication and latency will be depicted in the cost estimation of learned-by-example Bayesian models and all our algorithms will favor solutions engaging synopses, provided they abide by the posed accuracy constraints on O_6 , introduced in Section 6.1.1.


6.2 Optimizations tailored to Geo-distributed Complex Event Processing

Our algorithms are directly extensible to completely decentralized settings composed, apart from computer clusters, of sensor boards placed in environments of interest and let to function in an unsupervised manner for a protracted period of time. Such sensor boards possess certain processing and memory capabilities as well as limited power supply, mainly stemming from installed batteries or solar panels. Since in sensor settings communication is by far the biggest culprit in energy drain, it is important to reduce unnecessary communication as much as possible, simultaneously abiding by constraints on the latency introduced in the analytics procedures which collected data participate in.

The above described setting arises in INFORE especially in the Maritime use case where sensor boards correspond to on site observation devices, called wavegliders. Wavegliders are used in WP3 so that Maritime Surveillance is enhanced with non-collaborative data collection from acoustic sensors, thermal cameras etc in order to detect illegal or other abnormal activities at sea. Such activities cannot be pinpointed solely using collaboratively collected AIS data, since the availability of the latter, e.g., in case of piracy or smuggling, is highly unlikely just because, pirates and smugglers will turn off their AIS antennas in their pursuits.

We consider two approaches to reduce communication in such settings. Both approaches could exploit the rationale of the Complex Event Forecasting (CEF) Component of the INFORE architecture in order to achieve our optimization goals. In Complex Event Processing (CEP) [GAA+20], the detection of Complex Events (CEs) engages operators such as logical conjunctions (logical AND operation), or time ordered conjunctions (SEQUENCES) of simple events. It may also involve thresholded versions of aggregate, linear functions such as counts and sums of variables. In the Maritime Surveillance use case, as simple events may be thought of the reception of an AIS message or a data tuple describing a detected vessel, while CEs involve piracy, smuggling, illegal fishing and other activities of interest composed of simple events. The difference among CEP and CEF is that CEP detects CEs, while CEF probabilistically predicts such CEs well in advance to allow for proactive decision making.

The first approach to reduce communication, also accounting for latency constraints, uses a lazy communication approach, termed push-pull rationale. The push-pull rationale prioritizes the transmission of frequent simple events or CEs conditional upon the occurrence of rarer ones. Thus, if rare events do not occur, a sensor will not transmit the rest of the more frequent events it collected. After a streaming window expires, these events will expire as well. In that, communication will be avoided altogether. The second approach involves the installation of local filters on sensor boards. These filters are constructed in such a way, so that if they hold, the conditions of the posed CEP query cannot be satisfied and thus communication can be safely suppressed. Unless in situ filters are violated in at least one of the sensors participating in the analytics procedure, it is not necessary for communication to take place.

 <p>Project supported by the European Commission Contract no. 825070</p>	<p>WP5 T5.1 & T5.2 Deliverable D5.1</p>	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

We here sketch how these techniques are encompassed by our algorithms and get tailored to specific use cases. Further details are included in our recent publications in the scope of INFOR [FGD+20][GAD+19]. We will henceforth use the term endpoint site to distinguish computer clusters from other data collection devices.

6.2.1 Leveraging the push-pull rationale

Consider for exhibition purposes that a piracy event lies in the higher level of a hierarchy of events. The lower level events are as follows: e_1 – low speed of detected vessel, e_2 – U turn of detected vessel, e_3 – high speed of detected vessel. The lowest level of the event hierarchy consists of simple events of collected data regarding target vessels, their speed and estimated distance from a sensor, or their AIS location itself. The described simplistic scenario essentially says that a detected vessel attempts a maneuver where it initially slows down, it then changes direction and then starts speeding up as if it tries to run away from, potentially, a pirate vessel. In terms of CEP, the above can be expressed in SQL-like syntax as:

```
SEQ(lowSpeed  $e_3$ , UTurn  $e_2$ , highSpeed  $e_1$ )
Partition By VesselId
Within W seconds
```

The additional statement `Partition By` says that input data are partitioned for each distinct vessel and the `Within W` statement restricts the allowed interval within which e_1 , e_2 , e_3 should occur.

To see the big picture, consider that we are given a logical workflow which engages some CEP/CEF operator such as a simple SEQ(quence) one in the example. The versions of our algorithms presented in Section 4.5 and Section 5 will determine the physical operator for the corresponding logical one and assign its execution to a computer cluster of the available network, hosting a Big Data platform, e.g. Apache Flink and prescribing a specific parallelization degree [FGD+20].

In order to further optimize the execution of the physical workflow and reduce communication in a case where query operators (such as AND, SEQ) require all of their input events to occur to output a higher-level event, we can additionally use the push-pull rationale [FGD+20]. According to the push-pull mechanism, the transmission of frequent events is rendered conditional upon the occurrence of rare ones. So, for instance, if e_2 is a frequent event, but e_1 and e_3 are not, e_2 will not be transmitted until e_1 and e_3 occur. In other words, e_2 will be set in pull mode and it will be cached at the endpoint sites, where it is detected, until a request for transmitting such an event to the respective endpoint site is received. At the same time, e_1 and e_3 will be set in push mode, meaning that they will be transmitted immediately towards the query source as soon as they occur.

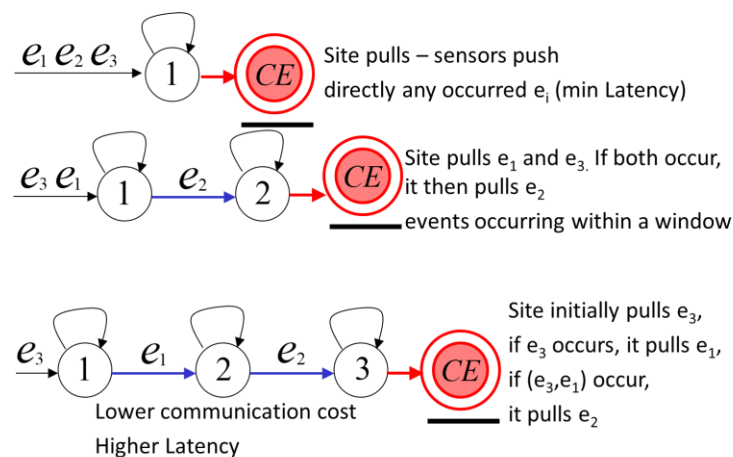


Figure 29: Example of different push-pull applications for a SEQ operator receiving 3 events (e_1 , e_2 , e_3) expressed using NFAs. This is a subset of alternative push-pull strategies that can be considered for the specific operator and its input. Self-transiting edges are used for the occurrence of e_i 's at the corresponding state. Blue edges represent events that may be pulled. Pulling events reduces bandwidth, increasing latency.

<p>Project supported by the European Commission Contract no. 825070</p>	<p>WP5 T5.1 & T5.2 Deliverable D5.1</p>	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

In our exemplary scenario and in generic setups, this is not the only way to go. There are a number of alternative push-pull strategies that may be employed. In Figure 29 and henceforth in this subsection, we utilize Non-deterministic Finite Automata (NFAs) [FGD+17] to enumerate push-pull strategies. A first option (left) may set all engaged events in push mode, i.e., they are transmitted as soon as they occur. This is what our optimization algorithms will do in case they are not enhanced in the ways we describe in this section. In the middle of the figure, we see the push-pull strategy we described above. Finally, a third option shown in the figure will be to set e_3 to push mode and e_1, e_2 in two pull mode steps. That is, if e_3 occurs, only e_1 is transmitted. And then, if e_1 has occurred and has been transmitted, e_2 is pulled. Still, these are not the only admissible push-pull strategies. In fact, what is not shown in Figure 29 and would be preferable from the Maritime Surveillance application perspective would be to set e_2 in push mode and e_1, e_3 in a single pull mode step. This admits that a U turn event is the rarest event that can occur, and, upon its occurrence, we should instantly examine events related to vessels' speed. In fact, the number of alternative push-pull strategies that should be examined for AND and SEQ operators is equivalent to the Bell number [FGD+20].

The trade-off that occurs here is that the more we delay communication of frequent events, i.e., the more pull steps we introduce, the more we increase the potential of event expiration when the respective window expires. Thus, we increase the potential for overall communication reduction. On the other hand, if both rare and frequent events involved in the monitored, higher-level CE do occur, the fact that we did not flash all events upon their occurrence increases the overall latency of the physical workflow. Therefore, introducing the push-pull rationale in our optimization plans affects objectives related to communication cost, that is O_5 , and latency, which is O_2 .

Our optimization algorithms can directly incorporate the examination of different push-pull strategies. The only modifications required is iterating and keeping solution sets, apart for all admissible parallelization degrees of an operator, for all possible push-pull strategies that the CEP operator admits. That is instead of $CS_{i,j}^\mu$ we have $CS_{i,j}^{\mu,v}$ where v ($=1$ for non-CEP operators in a logical workflow) enumerates all admissible push-pull strategies computing the performance per objective. For our Greedy and Heuristic (i.e., non-plus (+)) algorithmic variants the default push-pull alternative v_0 is the one that sets all events in push mode.

However, the physical plan that the optimizer will return to the Manager Component needs modifications with respect to window computations that may have been described in the logical workflow. Next, we outline the required physical operator window rewritings needed, so as to describe and implement the chosen push-pull strategy at the (non-endpoint) sites that undertake the AND or SEQ, CEP operator physical execution. For ease of presentation, we utilize examples of operator executions using three input events, but the discussion easily generalizes to broader operator instances engaging more input events.

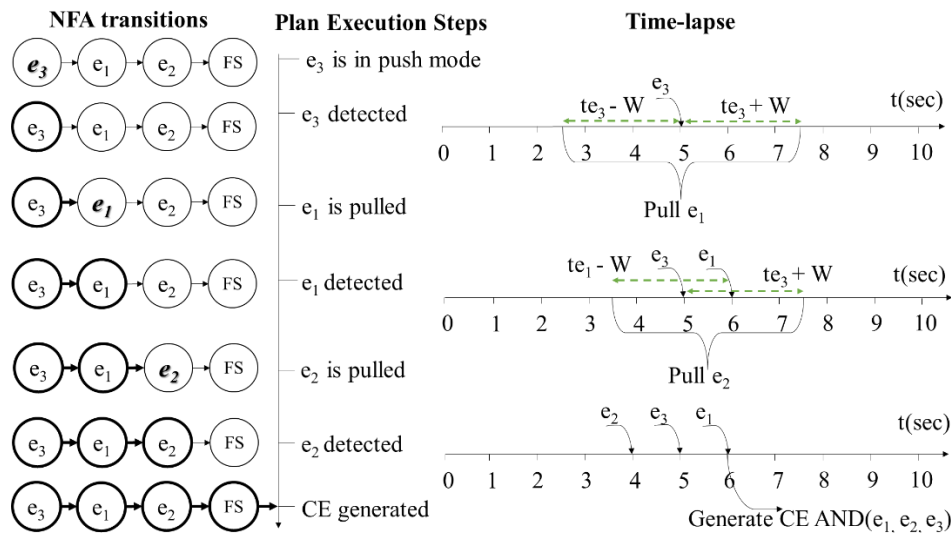



Figure 30: AND(e_1, e_2, e_3) with window $W = 2.5$ secs execution example. FS denotes the Final State which upon reached a CE is detected.

 <p>Project supported by the European Commission Contract no. 825070</p>	<p>WP5 T5.1 & T5.2 Deliverable D5.1</p>	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

Operator-Specific Rewriting - AND operator. For an NFA state, let t_{\min}/t_{\max} denote the minimum/maximum realOccurrence timestamp of all detected events in the previous NFA states of the AND operator. Then, the pull request when the NFA state is activated includes all detected events with OccurrenceTime within the window $t_{\max}-W \leq t_{\text{pull}} \leq t_{\min}+W$. Figure 30 presents an example of the execution of the complex event $\text{AND}(e_1, e_2, e_3)$ within a window $W=2.5$ secs and an execution plan composed of 3 push-pull steps ($e_3 \rightarrow e_1 \rightarrow e_2$). All depicted timestamps are realOccurrence timestamps. The site that has been assigned the aforementioned operator waits for events of type e_3 . Upon detection of an event of type e_3 , a pull request is issued upon the transition to the second state of the NFA that includes events of type e_1 . The pull request searches for events of type e_1 from sensor/ data sources in our network with realOccurrence time: $t_{e3}-W \leq t_{e1} \leq t_{e3}+W$. Upon detection of an event of type e_1 a pull request is issued for events of type e_2 with realOccurrence time: $t_{e1}-W \leq t_{e2} \leq t_{e3}+W$. Upon the arrival of an e_2 event within the requested time range, a complex event is generated. All these window calculations should be included in the physical plan that will be delivered to the cluster, Big Data platform that will undertake the physical execution of the CEP operator and the rationale described above is easily extensible to broader AND operator instances engaging more input events.

Operator-Specific Rewriting - SEQ operator. The sequence operator is similar to the AND operator but additionally requires that the time ordering of the events will also uphold $t_{1\text{st event}} \leq \dots \leq t_{i\text{-th event}}$. As such, the SEQ operator is transformed into a series of AND operators (1 per state of the NFA) and the transition from one state to the next marks the pull request of the events included in the next state. The pull request must simultaneously conform with the time ordering of the events and with the window constraints.

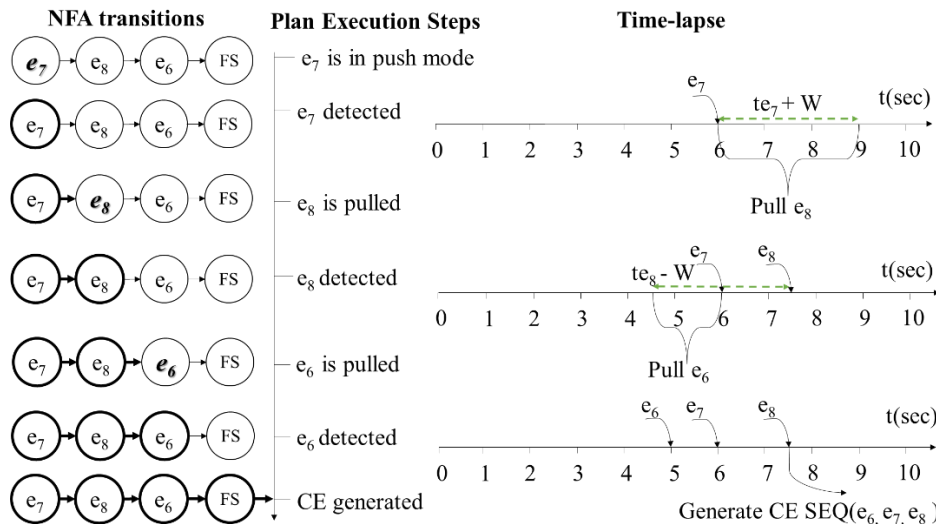



Figure 31: SEQ(e_6, e_7, e_8) with window $W=3$ secs execution example. FS denotes the Final State which upon reached a CE is detected.

Figure 31 presents an example for the execution of the complex event $\text{SEQ}(e_6, e_7, e_8)$ within $W=3$ secs and a 3-state push-pull strategy ($e_7 \rightarrow e_8 \rightarrow e_6$). In this example, we use different input event indices to make the distinction with the previously discussed AND operator easier. Upon detection of an event of type e_7 , a pull request is issued for the transition to the second state of the NFA that includes events of type e_8 . The pull request involves events of type e_8 with realOccurrence time: $t_{e7}+W \leq t_{e8}$ that may occur in the future. Upon detection of an event of type e_8 , a pull request is issued for events of type e_6 with realOccurrence time: $t_{e8}-W \leq t_{e6} \leq t_{e7}$.

Having argued about how our algorithms can be extended to examine push-pull strategies for physical execution plans that incorporate CEP operators, the final step in order to incorporate the push-pull rationale and the above-mentioned rewritings is to include the respective configurations in the JSON file that will be returned to the Manager Component of the INFOR architecture.

The more intriguing part is that these CEP-oriented optimizations require the flow of the data to change. So far, in all the operators supported by INFOR, the information data flow is always forward. That is, a site (computer cluster, Big Data platform) undertakes to execute parts of the workflow and as soon as a site completes its part, it

 <p>Project supported by the European Commission Contract no. 825070</p>	<p>WP5 T5.1 & T5.2 Deliverable D5.1</p>	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

delivers the output via Kafka to the next site. In the settings we described in this subsection this forward-only data flow model does not hold. The site that undertakes the execution of a CEP operator need to go through rounds of two-way communication with other endpoint sites, such as sensors, in order to evaluate the operator using a non-default (all input events in push mode) strategy. This change in the flow of data cannot be taken for granted since it requires access to the endpoint sites themselves, so that they are programmed on-the-fly to implement the push-pull rationale chosen by the optimizer, instead of simply transmitting detected events.

6.2.2 Uncertainty-aware In situ Processing

In this section we present additional optimizations that involve the evaluation of uncertainty-aware CEP queries engaging not only SEQ and AND but also AGGREGation, such as count, sum, operators [GAD+19]. For exhibition purposes, we use a running example instantiating a particular motivating scenario. Assume we are given a CEP query of the form:

```
AND (COUNT(CARGO_VESSEL) ≥ Tcv Q1), (COUNT(HIGH_SPEED_CRAFT) > THSC) Q2 Q
PARTITION BY vesselID
HAVING Q.certainty > C
WITHIN W minutes, Area
```


The above query aims at detecting the existence of a number (greater than T_{cv}) of cargo ships in a spatiotemporal window together with the existence of smaller, high-speed vessels. Again, the idea is to detect a potential piracy event that may take place in an area of interest. Further assume that AIS data are not available and, thus, the vessel type has been deduced with a correctness probability for each of the two mentioned vessel types. What is expressed in the query by `HAVING Q.certainty > C` is that we do not simply want to receive a data update whenever the AND pattern is satisfied, but instead, a CE is produced when the vessel types have been aggregated in respective counts are identified with a certain level of confidence, higher than C . Given such a query, the goal of the optimization process we introduce in this section is to decompose the given query into local filters – constraints that can be installed in each endpoint site so that unless the local filter is violated in at least one endpoint site, no communication takes place. The trade-off with respect to communication and latency for our optimization algorithms is analogous to our discussion in Section 6.2.1. The difference is that here (leaving the push-pull rationale which is orthogonal to the in situ filters we introduce, aside) we do not have to examine many alternatives, but just choose whether the physical plan will use the prescribed in situ filters or not.

The first step the optimizer does is to decompose the given query Q to individual Q_1 , Q_2 ones. Assuming event independence, the intersection of the engaged events (having the corresponding counts of vessel types exceed the defined threshold T_{cv}, T_{HSC} , respectively) provides a probability (certainty) that comes as the product of individual query certainties. Therefore, the first decomposition step gives:

```
PATTERN COUNT(CARGO_VESSEL) ≥ Tcv Q1      PATTERN COUNT(HIGH_SPEED_CRAFT) > THSC Q2
PARTITION BY vesselID                          PARTITION BY vesselID
HAVING Q1.certainty > C                        HAVING Q2.certainty > C
WITHIN W minutes, Area                        WITHIN W minutes, Area
```

This enables the optimizer to examine in situ filters for each query separately. Taking Q_1 as an example (the case of Q_2 is similar) let us see when communication of the count of detected cargo vessels is necessary. If we leave the certainty criterion aside for the moment and we assume that we have N endpoint sites (e.g. wavegliders) in the area, communication is meaningless if $\text{COUNT}(\text{CARGO_VESSEL}) < T_{cv}/N$ at every endpoint site. This is because then $\sum_{i=1}^N \frac{T_{cv}}{N} < T_{cv}$ and thus Q_1 will never output a CE. However, even this filter allows for unnecessary communication. This is because the (un)certainty criterion in Q_1 has been neglected. Thus, even if the probability of every ship to be of cargo type is very low, communication will still take place, which is unnecessary based on the definition of the given query.

Assume, for exhibition purposes, that each detected cargo ship has a probability p of being correctly classified and $1-p$ probability of being reported as cargo while it is not. This confidence value may be set by a domain expert or get derived from past data. Therefore, each detected cargo vessel is a Bernoulli random variable with success


	Project supported by the European Commission Contract no. 825070	WP5 T5.1 & T5.2 Deliverable D5.1	Doc.nr.:	WP5 D5.1
			Rev.:	1.0
			Date:	30/04/2020
			Class.:	Public

probability p regarding its correct classification. The total number of cargo ships detected in the area (Bernoulli trials) is the sum of the number of such ships detected by all nearby endpoint sites $\{A_1, \dots, A_N\}$: $\sum_{i=1}^N b_i$ with b_i being the local count of cargo vessels for endpoint site A_i . Let X denote the random variable representing the count of cargo vessels throughout the network of endpoint sites. Then X follows a Binomial distribution, i.e., $X \sim B(b, p)$, where b is the total number of detected cargo vessels the last Y minutes and p the probability of correct classification. Our contribution comes exactly because of the ability of our in situ filters to account for both the distribution and uncertainty dimensions of the posed query Q_1 . In particular, our in situ filters will recognize the fact that if globally $X \sim B(b, p)$, then for each endpoint site A_i , $X_i \sim B(b_i, p)$. Our techniques exploit probability theory to recognize that, for common p , the binomial distribution is self-decomposable, i.e., if $X_i \sim B(b_i, p) \rightarrow X = \sum_{i=1}^N X_i \sim B(b, p)$. Therefore, the in situ filter constructed by our approach will be: each A_i suppresses communication if the probability of the local count X_i of vessels of cargo type is at most T_{CV}/N with probability above $\sqrt[N]{1-C}$: $\Pr[X_i \leq \frac{T_{CV}}{N}] \geq \sqrt[N]{1-C}$ or, equivalently, communication is suppressed when for every endpoint site $CDF[X_i, \frac{T_{CV}}{N}] \geq \sqrt[N]{1-C}$. The latter filter accounts for both the criteria (cargo vessel count, confidence) included in the posed query. Our work [GAD+19] shows that such in situ filters are applicable in a wide variety of distributions that are self-decomposable, some of which are summarized in Table 1.

Table 1: Some supported probability distributions, uncertainty criteria and respective in situ filter examples. PDF: Probability, CDF: Cumulative Distribution Function. For log-versioned distributions a product instead of some of X_i is used. $1 - CDF(X, T) > C \Leftrightarrow P[X > T] > C$ in the fifth column exemplifies the query uncertainty criterion, which corresponds to the global filter: $P[X > T] \leq C$.

Distribution	PDF	Remarks	Decomposition Example	In-situ Filter for $1 - CDF(X, T) > C$
Normal	$\frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$	$\forall x \in \mathbb{R}$	$X_i \sim Normal(\mu_i, \sigma_i^2)$ $X = \sum_{i=1}^N X_i \sim Normal(\sum_{i=1}^N \mu_i, \sum_{i=1}^N \sigma_i^2)$	$\sqrt[N]{1-C} \leq CDF(X_i, \frac{T}{N})$
Log-Normal	$\frac{1}{x\sigma\sqrt{2\pi}} e^{-\frac{(\ln x - \mu)^2}{2\sigma^2}}$	$\forall x > 0$ $\mu \in \mathbb{R} (\neq \text{mean})$ $\sigma > 0 (\neq \text{st.dev.})$	$X_i \sim LogNormal(\mu_i, \sigma_i^2)$ $X = \prod_{i=1}^N X_i \sim LogNormal(\sum_{i=1}^N \mu_i, \sum_{i=1}^N \sigma_i^2)$	$\sqrt[N]{1-C} \leq CDF(X_i, \sqrt[N]{T})$
Chi-Square	$\frac{1}{2^{\nu/2}\Gamma(\frac{\nu}{2})} x^{\frac{\nu}{2}-1} e^{-\frac{x}{2}}$	$\forall x > 0$ $\nu \in \mathbb{N}^+$ degrees of freedom	$X_i \sim x^2(v_i)$ $X = \sum_{i=1}^N X_i \sim x^2(\sum_{i=1}^N v_i)$	$\sqrt[N]{1-C} \leq CDF(X_i, \frac{T}{N})$
Cauchy	$\frac{1}{\pi s [1 + (\frac{x-\nu}{s})^2]}$	$\forall x \in \mathbb{R}$ $\nu \in \mathbb{R} (\text{location})$ $s > 0 (\text{scale})$	$X_i \sim Cauchy(v_i, s_i)$ $X = \sum_{i=1}^N X_i \sim Cauchy(\sum_{i=1}^N v_i, \sum_{i=1}^N s_i)$	$\sqrt[N]{1-C} \leq CDF(X_i, \frac{T}{N})$
Poisson	$\frac{\lambda^x e^{-\lambda}}{x!}$	$\forall x \in \mathbb{N}$ $\lambda > 0$	$X_i \sim Poisson(\lambda_i)$ $X = \sum_{i=1}^N X_i \sim Poisson(\sum_{i=1}^N \lambda_i)$	$\sqrt[N]{1-C} \leq CDF(X_i, \frac{T}{N})$
Gamma	$\frac{1}{\Gamma(\alpha)\theta^\alpha} x^{\alpha-1} e^{-\frac{x}{\theta}}$	$\forall x > 0$ $\alpha > 0 (\text{shape})$ $\theta > 0 (\text{scale})$	$X_i \sim Gamma(\alpha_i, \theta)$ $X = \sum_{i=1}^N X_i \sim Gamma(\sum_{i=1}^N \alpha_i, \theta)$	$\sqrt[N]{1-C} \leq CDF(X_i, \frac{T}{N})$
Logistic	$\frac{e^{-\frac{x-\nu}{s}}}{s(1+e^{-\frac{x-\nu}{s}})^2}$	$\forall x \in \mathbb{R}$ $\nu \in \mathbb{R} (\text{location})$ $s > 0 (\text{scale})$	$X_i \sim Logistic(v_i, s_i) (\text{approx.})$ $X = \sum_{i=1}^N X_i \sim Logistic(\sum_{i=1}^N v_i, \sqrt{\sum_{i=1}^N s_i^2})$	$\sqrt[N]{1-C} \leq CDF(X_i, \frac{T}{N})$
Log-Logistic	$\frac{(\beta/\alpha)(x/\alpha)^{\beta-1}}{(1+(x/\alpha)^\beta)^2}$	$\forall x > 0$ $\alpha > 0 (\text{scale})$ $\beta > 0 (\text{shape})$ $\nu = \log(\alpha)$ $s = 1/\beta$	$X_i \sim LogLogistic(v_i, s_i) (\text{approx.})$ $X = \prod_{i=1}^N X_i \sim LogLogistic(\sum_{i=1}^N v_i, \sqrt{\sum_{i=1}^N s_i^2})$	$\sqrt[N]{1-C} \leq CDF(X_i, \sqrt[N]{T})$
Exponential	$\lambda e^{-\lambda x}$	$\forall x > 0$ $\lambda > 0 (\text{rate})$	$X_i \sim Gamma(\frac{\alpha_i}{N}, \frac{1}{\lambda})$ $X = \sum_{i=1}^N X_i \sim Exp(\lambda)$	$\sqrt[N]{1-C} \leq CDF(X_i, \frac{T}{N})$
Binomial	$\binom{n}{x} p^x (1-p)^{n-x}$	$x = 0, 1, \dots, n$ $p \in [0, 1]$ $n \in \mathbb{N}$	$X_i \sim Binomial(n_i, p)$ $X = \sum_{i=1}^N X_i \sim Binomial(\sum_{i=1}^N n_i, p)$	$\sqrt[N]{1-C} \leq CDF(X_i, \frac{T}{N})$

Having decided the in situ filter, again a communication protocol between the cluster at which the CEP operators are installed and the various endpoint sites should exist to determine the rounds of communication that need to take place. There is a variety of complex communication protocols that one can use to exploit such in situ filters. We discuss such elaborate communication protocols in our research in the scope of the project [GAD+19]. We here

	Project supported by the European Commission Contract no. 825070	WP5 T5.1 & T5.2 Deliverable D5.1	Doc.nr.:	WP5 D5.1
			Rev.:	1.0
			Date:	30/04/2020
			Class.:	Public




describe a simple communication protocol that requires only one round of communication (from endpoint sites to sites) so that it can more easily be applied simultaneously with the push-pull strategy discussed in Section 6.2.1:

Initialization Phase: During the initialization phase, the computer cluster running a Big Data platform that will execute the involved CEP operators (i.e., the site) constructs the local filters and transmits them to endpoint sites together with the chosen push-pull rationale. The transmission of in situ filters to endpoint sites can also be done directly by the Manager Component, provided that information as of to which computer cluster each site should report is included in the exchanged information.

Monitoring Phase: Each endpoint site keeps up receiving updates of its local data and contacts the handling computer cluster only in case it finds its local in situ filter violated and a pull request has arrived to set the event in push mode.

Synchronization Phase: When a local filter is violated at an endpoint site and the involved event is in push mode, the endpoint site pushes its events to the computer cluster than runs the respective CEP operator. Then, that site pulls the relevant events from the rest of the endpoint sites. In the scope of a push-pull rationale this means that an event may have occurred. In our running example, this is the event implied by query Q_1 . If this is true, the site handling the CEP operator pulls the events that follow in the next step of the push pull strategy. Otherwise, the endpoint sites return to the Monitoring Phase.

 Project supported by the European Commission Contract no. 825070	WP5 T5.1 & T5.2 Deliverable D5.1	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

7 Applications on INFORE Use Cases

In this section we describe application scenarios related to INFORE use cases and showcase the functionality of the Optimizer Component in practice. The logical workflows are defined in the Graphical Editor Component of the INFORE architecture. At the workflow design time, there is a higher-level part of the logical workflow that is common in all use cases that utilize the streaming extensions of RapidMiner Studio developed in the scope of the project.

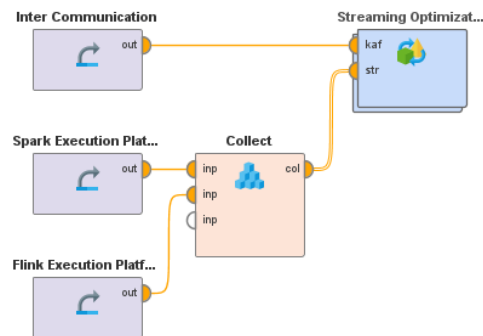


Figure 32: Higher level of the logical workflow definition. Inter Communication connection object defines the Kafka cluster with which parts of workflow later assigned to different clusters and Big Data platforms will communicate with. Spark/Flink Execution connection objects provide connections to the available clusters hosting Big Data platforms, grouped together via the Collect operator and passed to the Streaming Optimization operator.

Figure 32 shows this higher-level part of the logical workflow definition. It includes an Inter Communication connection object which defines the Kafka cluster with which parts of workflow later assigned to different clusters and Big Data platforms will communicate. Spark/Flink Execution connection objects provide connections to the available cluster hosting respective Big Data platforms, grouped together via the Collect operator and passed to the Streaming Optimization operator. The Streaming Optimization operator that has been introduced in the RapidMiner Studio is a subprocess operator, i.e., the user can double click on it to go through a lower level workflow definition level, to start drawing the desired workflow using families of operators stemming from stream transformations provided by the Big Data platforms which have been abstracted in the Streaming extension of the RapidMiner Studio, the Synopses Data Engine, the Online Machine Learning and Data Mining or the Complex Event Forecasting Components.

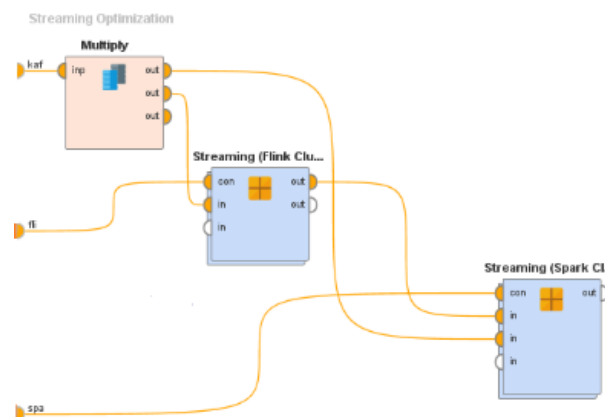



Figure 33: The intermediate level created after the submission of the logical workflow of Figure 32 creates a Multiply operator to duplicate Kafka connection objects for cluster, Big Data platform communication and creates one Streaming Nest operator for Flink and one for Spark in this example

In the use cases we discuss in this section, when the user has finished drawing the desired workflow within the Streaming Optimization operator, she presses a submit button. The response of the Optimizer Component is visualized to the user and within the Streaming Optimization operator, the logical operators are now placed in one

 <p>Project supported by the European Commission Contract no. 825070</p>	<p>WP5 T5.1 & T5.2 Deliverable D5.1</p>	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

(if all operators have been assigned to the same cluster/Big Data platform) or more Streaming Nest operators. In other words, a Streaming Nest operator appears for each cluster, Big Data platform that has been used in the prescribed physical workflow. The initial logical workflow has now been divided to parts that will be submitted in different Big Data platforms. The Streaming Nest operator is another subprocess operator that groups together the parts of the initial workflow that will be executed in Spark or Flink as shown in the example of Figure 33. In the figure, the streaming execution connections are provided to the corresponding Streaming Nest operators. The Multiply operator in the figure, just replicates the Kafka connection object to be used by each Streaming Nest operator. The two Streaming Nest operators in the figure are connected to denote that one part of the workflow provides input to another. The parts of the initial logical workflows that lie in the two Streaming Nest operators are not logical anymore, but have been interpreted to physical because the cluster and Big Data platform they are to be executed in is defined in the connections of the respective Streaming Nest operator as shown in Figure 33 for Spark and Flink.

Given these, in what we discuss here, when we refer to the logical workflow we mean the workflow the user designs within a Streaming Optimization operator and when we refer to physical workflows in Spark and Flink we mean those assigned to a respective Streaming Nest operator. The higher level of the logical and physical workflow in each use case is equivalent.

7.1 Life Sciences Use Case

In this use case the aim is to use INFORE to provide a virtual laboratory for simulating tumor behavior under various drug combinations. We produce simulated tumor data of several GB/min from various instances of the PhysiBoss¹⁴ framework. INFORE ingests from PhysiBoss data related to the state of each cell agent, the concentration of various densities such as oxygen and time series data on the number of necrotic, apoptotic or proliferating cells. In this model, effective drugs' activity forces tumor cells into necrosis and fewer to apoptosis.

Figure 34 depicts an exemplary logical workflow, where our target is to distinguish which simulations and thus respective drug combinations are effective in terms of increasing the number of necrotic and apoptotic cells, simultaneously reducing the proliferating ones. For that purpose, data from the various PhysiBoss instances are loaded via two Kafka topics for unlabeled and labeled simulations. Using the SDE.DFT operator of the SDE Component, respective time series are approximated via their Discrete Fourier Transform (DFT) coefficients [KoGD20] for interactivity purposes. Then, they are fed to an online classification operator from the OMLDM Component to train models for distinguishing “useful” drug combinations from “unuseful” ones. The extracted model is validated via the Forecast Validation operator and is then fed to the Tag Simulation operator which applies the model and classifies the running simulations. The Split operator separates unuseful simulations from useful ones. Those with unuseful outcomes correspond to PhysiBoss instances that should be killed (Figure 34, top split branch). The top-k, in terms of killed tumor cells, of the rest are chosen for visualization and further study (Figure 34, bottom split branch).

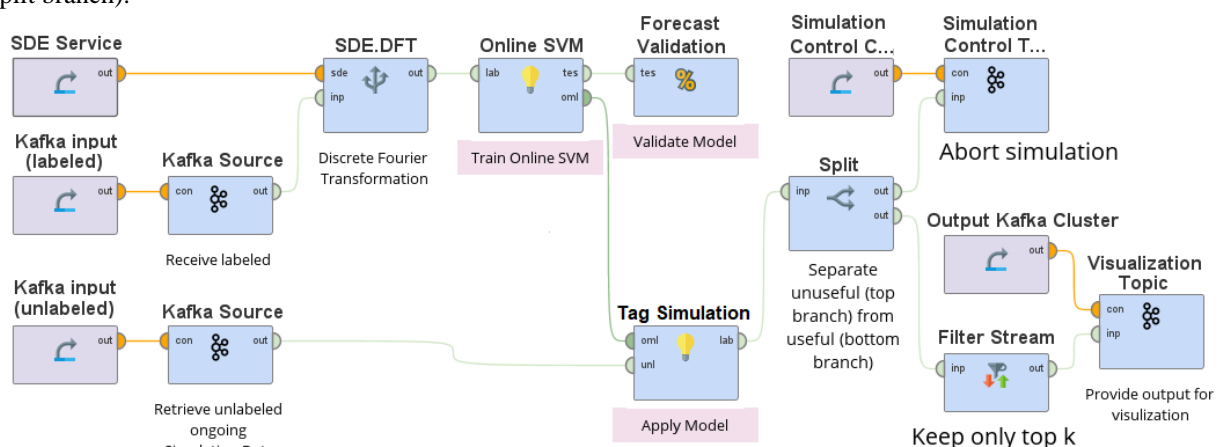



Figure 34: Exemplary Logical Workflow for the Life Science Use Case submitted to the Optimizer.

¹⁴ <https://github.com/gletort/PhysiBoSS>

 <p>Project supported by the European Commission Contract no. 825070</p>	<h3>WP5 T5.1 & T5.2</h3> <h3>Deliverable D5.1</h3>	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

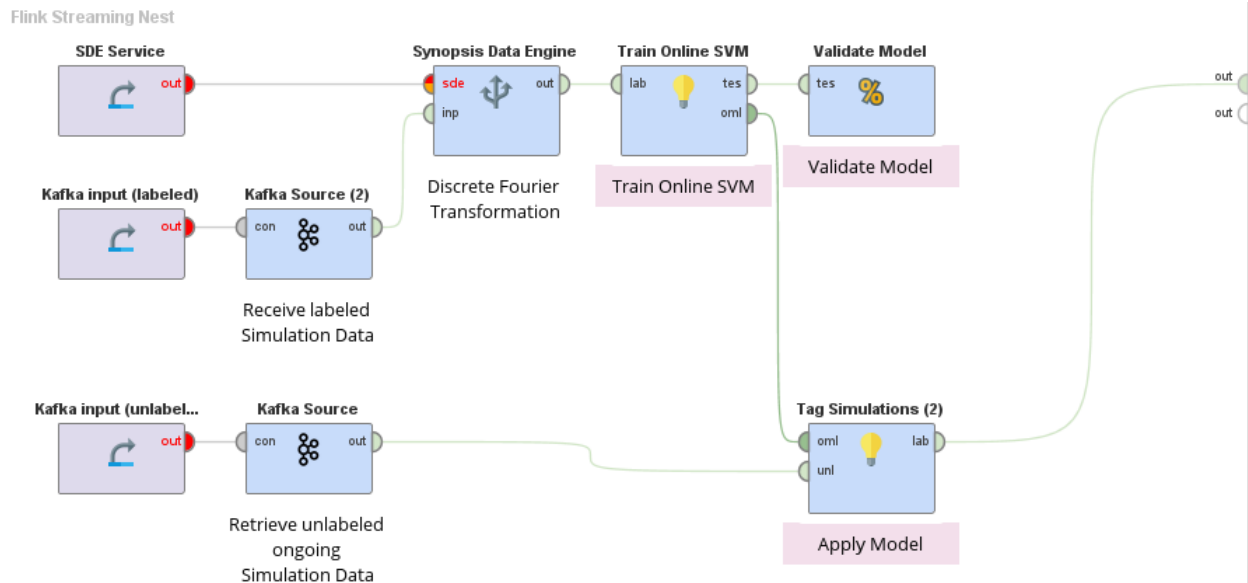


Figure 35: Part of the physical workflow the Optimizer prescribes to be executed in Flink for the logical workflow of Figure 34.

Figure 35 and Figure 36 show the lower level of the physical workflow when the Optimizer Component decides to assign the execution of parts to workflow in Flink and Spark Structured Streaming, respectively. The structure of the intermediate level including the corresponding Streaming Nest operators lying in the higher-level Streaming Optimization operator remain as described at the beginning of this section.

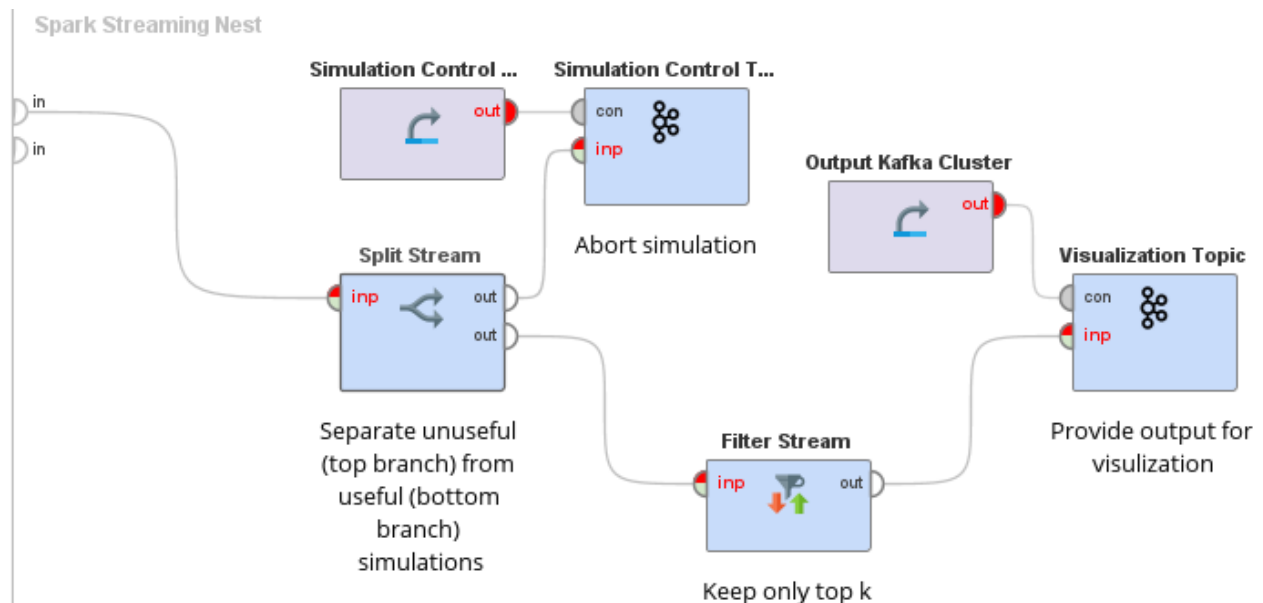



Figure 36: Part of the physical workflow the Optimizer prescribes to be executed in Spark Structured Streaming for the logical workflow of Figure 34.

7.2 Financial Use Case

We use Level 1¹⁵ and Level 2¹⁶ stock data provided by Spring Techno INFOR partner stemming from 9 markets. More precisely, Level 1 data involve stock trades of the form < Date, Time, Price, Volume > for each data tick of an

¹⁵ <https://www.investopedia.com/terms/l/level1.asp>

¹⁶ <https://www.investopedia.com/terms/l/level2.asp>

	Project supported by the European Commission Contract no. 825070	WP5 T5.1 & T5.2 Deliverable D5.1	Doc.nr.:	WP5 D5.1
			Rev.:	1.0
			Date:	30/04/2020
			Class.:	Public

asset (stock). Level 2 data show the activity that takes place before a trade is made. Such an activity includes information about offers of shares and corresponding prices as well as respective bids and prices per stock. Thus, Level 2 data are shaped like series of $\langle \text{Ask price, Ask volume, Bid price, Bid volume} \rangle$ until a trade is made. These pairs are timestamped by the time the stock trade happens. The higher the number of such pairs for a stock, the higher the popularity of the stock.

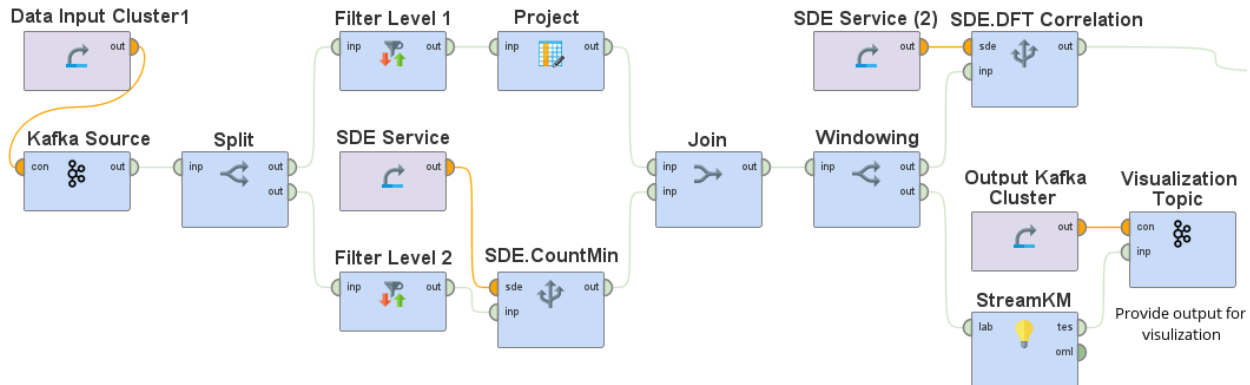


Figure 37: Exemplary Logical Workflow for the Financial Use Case submitted to the Optimizer.

The logical workflow of Figure 37 illustrates an exemplary scenario that utilizes Level 1 and Level 2 stock data to discover cross-correlations and clusters of stocks. In the figure, both Level 1 and Level 2 data arrive at a Kafka source. The Split operator separates Level 1 from Level 2 data. It directs Level 2 data to the bottom branch of the workflow. There, the Level 2 bids are Filter-ed (i.e., for monitoring only a subset of stocks or keep only bids above a price/volume threshold). The bids per stock are counted using a CountMin sketch provided by the SDE Component. When a trade for a stock is realized, a new Level 1 tuple is directed by Split to the upper part of the workflow. A Project operator keeps only the timestamp of the trade for each stock. The Join operator joins the stock trade, Level 1 tuple with the count of bids the stock received until the trade. The result is inserted in a time Window of recent such counts, forming a time series. Pairwise similarities of stocks' time series are computed using the approximations of the Discrete Fourier Transform operator of the SDE. The StreamKM operator of the OMLDM Component computes clusters of stocks based on the original time series.

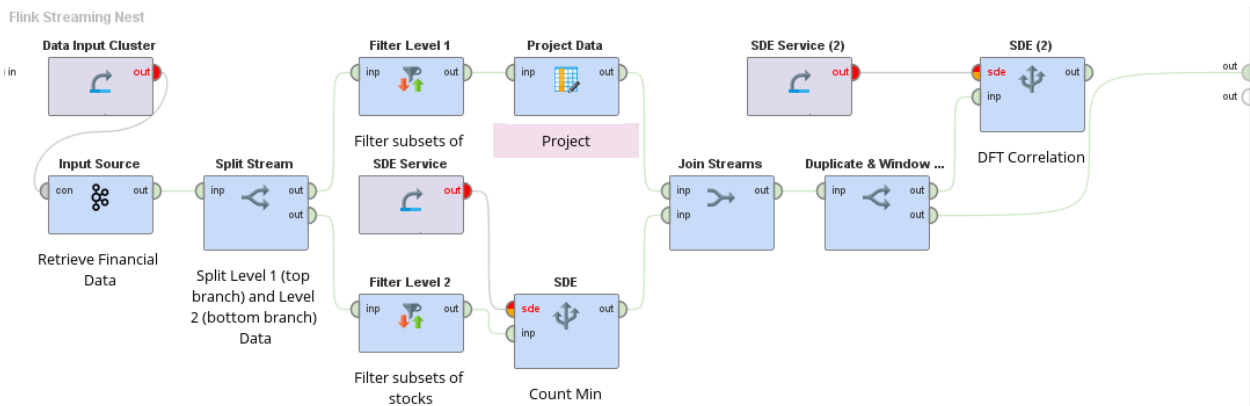



Figure 38: Part of the physical workflow the Optimizer prescribes to be executed in Flink for the logical workflow of Figure 37.

Figure 38 and Figure 39 show the lower level of the physical workflow when the Optimizer Component decides to assign the execution of parts of workflow to Flink and Spark Structured Streaming, respectively. In this particular occasion, only the clustering operator is assigned to a cluster, Big Data platform other than Flink. The reason is that the visualization topic for the result of the clustering algorithm is in a Kafka cluster co-placed with Spark. Again, the structure of the intermediate level including the corresponding Streaming Nest operators lying in the higher-level Streaming Optimization operator remain as described at the beginning of this section.

 <p>Project supported by the European Commission Contract no. 825070</p>	<p>WP5 T5.1 & T5.2 Deliverable D5.1</p>	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

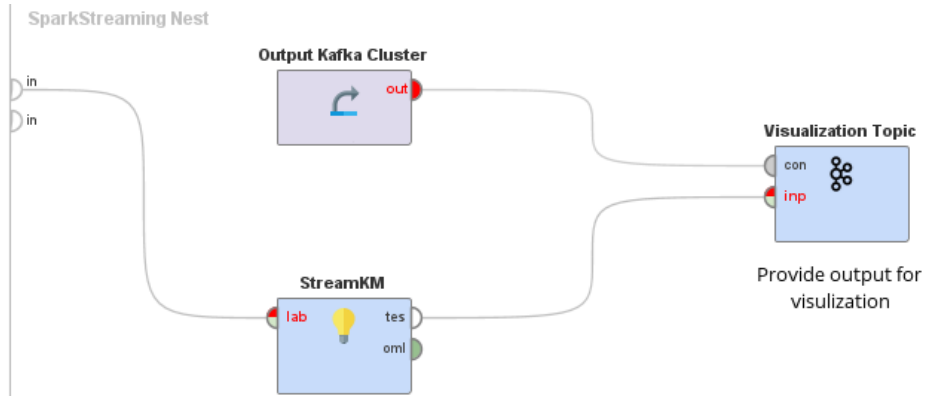


Figure 39: Part of the physical workflow the Optimizer prescribes to be executed in Spark for the logical workflow of Figure 37.

7.3 Maritime Use Case

Let us now continue with a logical workflow for the Maritime use case. Assume that we focus on one type of event that indicates a zig zag movement that we want to detect for each vessel. Such a pattern may correspond to an illegal fishing activity in practice. Furthermore, consider that in order to identify such a simple event we check a threshold on the variance of the direction of each vessel within a given window of AIS message updates.

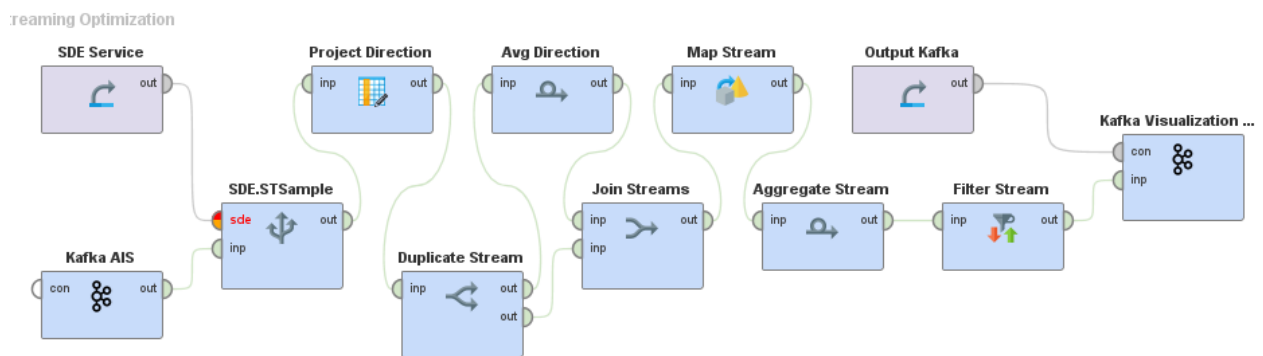



Figure 40: Exemplary logical workflow for zig zag movement detection in the Maritime Use Case.

As shown in the logical workflow of Figure 40, AIS messages are ingested via a Kafka topic and each vessel's trajectory undergoes a sampling process, denoted by SDE.STSample, using the operator provided by the SDE Component (see Section 2.4). We then perform a projection that keeps only the vessel-id and the direction from the sampled AIS messages. The resulted trajectory stream is duplicated (Duplicate Stream operator in the figure). We use an aggregate operator to compute the average direction of the vessel's movement within a window and we then join the resulted stream with the original one, so that we have a stream with vessel-id, direction, avg_direction. The Map Stream operator computes squared differences of direction-avg_direction divided by the window size, for each streaming tuple and these are then summed up to compute the variance in direction using the Aggregate Stream operator. The Filter Stream operator keeps the vessels which exhibited a variance in direction of movement surpassing a given threshold.

Figure 41 and Figure 42 show the lower level of the physical workflow when the Optimizer Component decides to assign the execution of parts to workflow in Flink and presumably Akka, respectively. In this particular occasion, only the SDE operator is assigned to Flink where the current implementation of the SDE Service runs. The reason is that the optimizer has been configured so that it detects that the Kafka cluster used for (i) interconnection of the components of the INFOR architecture, (ii) input Kafka Source topic and (iii) the Visualization topic are co-placed with the Akka cluster.

 <p>Project supported by the European Commission Contract no. 825070</p>	<p>WP5 T5.1 & T5.2 Deliverable D5.1</p>	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

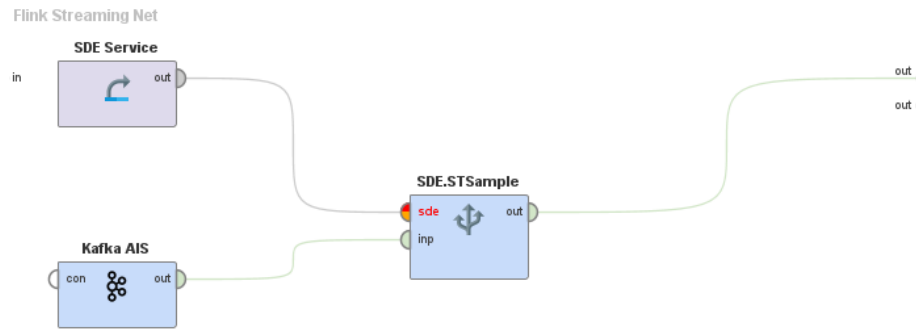


Figure 41: Part of the physical workflow the Optimizer prescribes to be executed in Flink for the logical workflow of Figure 40.

Again, the structure of the intermediate level including the corresponding Streaming Nest operators lying in the higher-level Streaming Optimization operator remain as described at the beginning of this section, replacing Spark with Akka. Note that this is the output of the optimizer with respect to the definitions of operators that exist in its dictionary. These definitions may include a superset of the currently supported Big Data platforms.

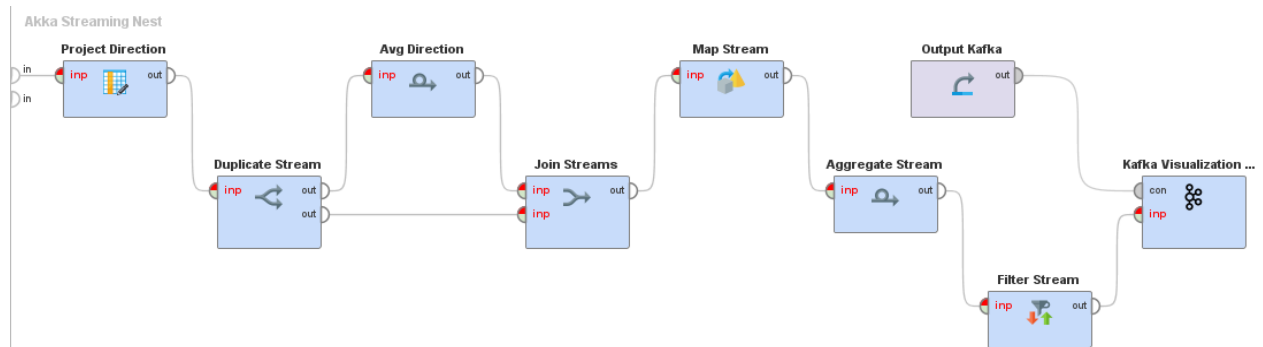


Figure 42: Part of the physical workflow the Optimizer prescribes to be executed in Akka for the logical workflow of Figure 40.

<p>Project supported by the European Commission Contract no. 825070</p>	<p>WP5 T5.1 & T5.2 Deliverable D5.1</p>	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

8 Experimentation and Benchmarking

8.1 Benchmarking our Cost Estimator

In this section we benchmark our cost estimator presented in Section 4.3. In these experiments, the evaluation of the cost estimation process is performed as follows:

- Step 1: for each supported operator we perform a set of microbenchmarks. In each microbenchmark, we run a job including only the benchmarked operator and we provide the following input parameters which are varied among microbenchmarks:
 - Big Data platform: where the job is submitted, such as ‘Apache Flink’, ‘Apache Spark Structured Streaming’.
 - Source type: here there are two options ‘Custom Source’ or ‘Kafka Source’. ‘Custom Source’ accounts for the case when an upstream operator of the benchmarked one is executed in the same Big Data platform. ‘Kafka Source’ accounts for the case when an upstream operator is executed in a different Big Data platform or cluster, since these two operators will communicate via Kafka according to the design of the INFOR architecture.
 - Sink type: similar to the above, ‘Custom Sink’ for conveying the output to downstream operators in the same Big Data platform, or ‘Kafka Sink’ otherwise.
 - Rate of arrival of input streams: we test the performance with respect to the optimization objectives (Section 4.4) of the operator under various incoming stream rates.
 - Degree of parallelism: in each microbenchmark we also alter the chosen parallelization degree of the said operator.

The output of the benchmark is the value of each optimization objective mentioned in Section 4.4.

- Step 2: Having performed microbenchmarks for each operator where we alter the input parameters as described in Step 1, we feed the majority (~80%) of the results to the cost estimator to fit and/or train the Bayesian optimization model for this operator. Note here that the model may converge earlier without using all the results that were intended for the training step. This is denoted in the experiments as ‘percentage of explored space’ which is the percentage of benchmarking results, which the training (sampling) process asked for. This percentage shows how fast the model converges (see also Figure 11). The lower the percentage, the less microbenchmarks we need to develop what the Bayesian optimization considers a “good” model.
- Step 3: We test the predictive accuracy of the cost estimator using the rest of the microbenchmarks. In particular, having trained the cost estimator, we interrogate the developed model at each stage of the fitting/training process. Each time we query the model using the input parameters (those of Step 1, but for ~20% of the microbenchmarks that were not used in Step 2) receiving a response about the predicted performance of the operator. We measure the accuracy of the model in terms of cumulative L_1 distance¹⁷ and R^2 score¹⁸ (coefficient of determination) between the actual performance during the microbenchmarks and the predicted one.

For ease of presentation, in this section, we focus on one of the most commonly used operators in many real-life workflows, that of a 2-way inner join, over Flink and to limit the number of presented graphs, we bin the input stream rates to 1K, 10K, 100K, 1M records/sec and we present results for a parallelization degree of 12. For the same reason, we also concentrate on the accuracy of our cost estimator in predicting the throughput (objective O_1 in Section 4.4) among our optimization objectives. Table 2 presents a couple of instances of the result of Step 1 of the evaluation process. The first 6 columns are part of the input and having executed the microbenchmark we obtain the last column related to throughput. This type of records will be fed in our cost estimator during the fitting and training process of Step 2.

¹⁷ https://scikit-learn.org/stable/modules/generated/sklearn.metrics.pairwise.manhattan_distances.html

¹⁸ https://scikit-learn.org/stable/modules/generated/sklearn.metrics.r2_score.html


	Project supported by the European Commission Contract no. 825070	WP5 T5.1 & T5.2 Deliverable D5.1	Doc.nr.:	WP5 D5.1
			Rev.:	1.0
			Date:	30/04/2020
			Class.:	Public

Table 2: Example Input/ Output of a performed microbenchmark

Source 1	Source 2	Source 1 rate	Source 2 rate	Output Sink	Parallelism	Throughput
CustomSource	CustomSource	1000000	1000000	CustomSink	12	204541,0556
CustomSource	CustomSource	1000	1000	KafkaSink	1	1966,666667

To execute Step 2, we use Gaussian Processes (GPs) as the surrogate model and we experiment with 2 commonly used acquisition functions, Expected Improvement (EI) and Lower Confidence Bound (LCB). In each experiment, excluding the one where we just perform fitting and no acquisition function is needed, we also alter a $\xi=[0.0001,0.001,0.01(\text{default}),100,1000]$ parameter for EI and $\kappa=[1K, 2.5K, 5K, 10K, 100K]$ for LCB which trade-off exploration vs exploitation. In a nutshell, exploitation drives the training on sampling microbenchmarks in the vicinity of the current best results by penalizing for higher variance values, while exploration pushes the search towards unexplored regions. The higher the ξ or κ parameters, the farthest from the current best exploration is allowed to reach, i.e., the higher these values are, the more we are favoring exploration over exploitation.

Because of the fact that, in this set up, we want to maximize throughput (which has a negative weight in the optimization set up of Section 4.4), the convergence plots that follow show $\max f(x)$ in their vertical axes, while the horizontal axes account for the percentage of explored space as commented above.

In a first experiment illustrated in Figure 43, we use no acquisition function and we simply perform a fitting process in order to improve the posterior probability of the surrogate model, but in a less targeted way compared to sampling based on exploitation or exploration criteria. This is useful in order to check the effect of the learn-by-example training paradigm (see Section 4.3.2) where we use the statistics that are made available to us due to the execution of submitted workflows of the application domain, to improve the accuracy of the cost estimator after deploying it.

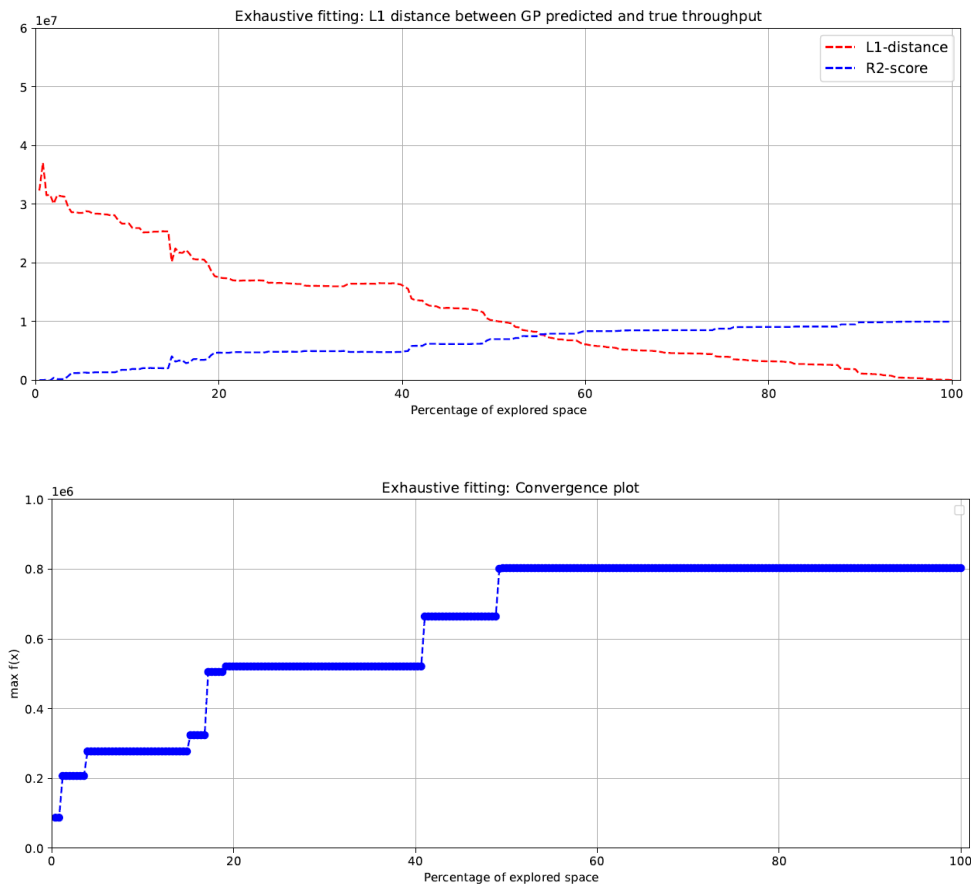



Figure 43: Accuracy (up) and convergence speed (down) of cost estimator performing a fitting process

 <p>Project supported by the European Commission Contract no. 825070</p>	<p>WP5 T5.1 & T5.2 Deliverable D5.1</p>	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

As shown at the bottom of Figure 43, the maximum value of the objective function is reached after half (50%) of the microbenchmarks are used for fitting. The rest of the microbenchmarks that are fed to the model do not alter this maximum value. What they do alter, as illustrated at the top of the figure, is the accuracy of the model since the L_1 distance progressively zeros and R^2 reaches a value of 1 as the corresponding lines approach the 100% of the fed microbenchmarks. Recall that this is the result of executing Step 3 of the cost estimator evaluation process, measuring the distance between the predicted and the actual throughput using the corresponding metric. As a general observation, using 50-60% of the performed experiments makes the fitting process converge to the maximum of the objective function and keep acceptable absolute and relative error values.

Note that the L_1 distance measures a cumulative absolute error, while R^2 score shows whether the trends (rather than the absolute values) in predicted and actual throughput vectors (of the 20% of the microbenchmarks not used for fitting in Step 2) are similar. This is also important because the bottom line is that we want the optimizer to choose the best site for each examined operator. Thus, even if predictions are not accurate in terms of absolute values (L_1 distance), if they can still capture the trends of throughput (R^2 score), they can tell which option is the best to include in a physical workflow. Hence, our overall goal of choosing the physical workflow that maximizes throughput is practically achieved.


We then proceed with experimenting on EI and LCB acquisition functions. In this series of experiments, we first provide 10 microbenchmarks to the Bayesian optimization process and we then let it ask for samples driven by the utilized acquisition function. The 10 initially fitted microbenchmarks are chosen uniformly at random from the set of all microbenchmarks with a random seed (rand_state in the graphs) of 10. Later on, in this section, we will also alter the number of microbenchmarks used for initial fitting and the way they are chosen.

In Figure 44 and Figure 45 we plot the prediction accuracy (left columns) and convergence speed (right columns) of Bayesian optimization using EI and LCB while we vary the ξ and κ parameters respectively. Increasing the ξ and κ values improves the accuracy of the model, but also increases the percentage of explored space. This says that the training process asks for more samples before it concludes. Comparing the two figures we extract a number of useful observations:

- With respect to L_1 and R^2 , EI and LCB behave similarly in the sense that for the same percentage of explored microbenchmarks they provide comparable accuracy.
- EI uses less microbenchmarks before it concludes the training process, i.e., the percentage of explored space is lower compared to that of LCB, but also provides lower accuracy by the end of the training process.
- LCB samples more microbenchmarks before it concludes, i.e., the percentage of explored space is higher and, due to that, it can also provide better prediction accuracy.
- If we focus on the convergence speed plot, we can observe that EI approaches faster $\max f(x)$, while LCB needs more samples to do that. This is shown by the fact that the first step, after the 10 initially fed microbenchmarks, of the blue line in the respective graphs is always flatter for LCB which means that the first samples it asks for do not help in approaching the maximum of the objective function.

Based on the above observations, from an application viewpoint, EI and LCB are useful in different situations. If one has a priori performed an exhaustive list of microbenchmarks and then applies the training phase, LCB seems the most preferable option since it will exploit most of the microbenchmarks to improve accuracy as much as possible. On the other hand, if one performs only an initial set of microbenchmarks and then lets the training process indicate which microbenchmark should be conducted afterwards, EI will conclude the training phase faster, with acceptable accuracy and thus fewer, on-demand microbenchmarks are needed. In other words, LCB is more accurate, but also more expensive in terms of the required effort that should be devoted for benchmarking each operator.

Another observation involving LCB that can be extracted by studying its convergence plots (right column of Figure 45) is that it samples a lot of microbenchmarks which do not actually improve the currently best ($\max f(x)$) value. This is evident by the fact that the blue line remains steady although the training process does not conclude for a considerable percentage of the explored space (in the horizontal axis).

 <p>Project supported by the European Commission Contract no. 825070</p>	<p>WP5 T5.1 & T5.2 Deliverable D5.1</p>	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

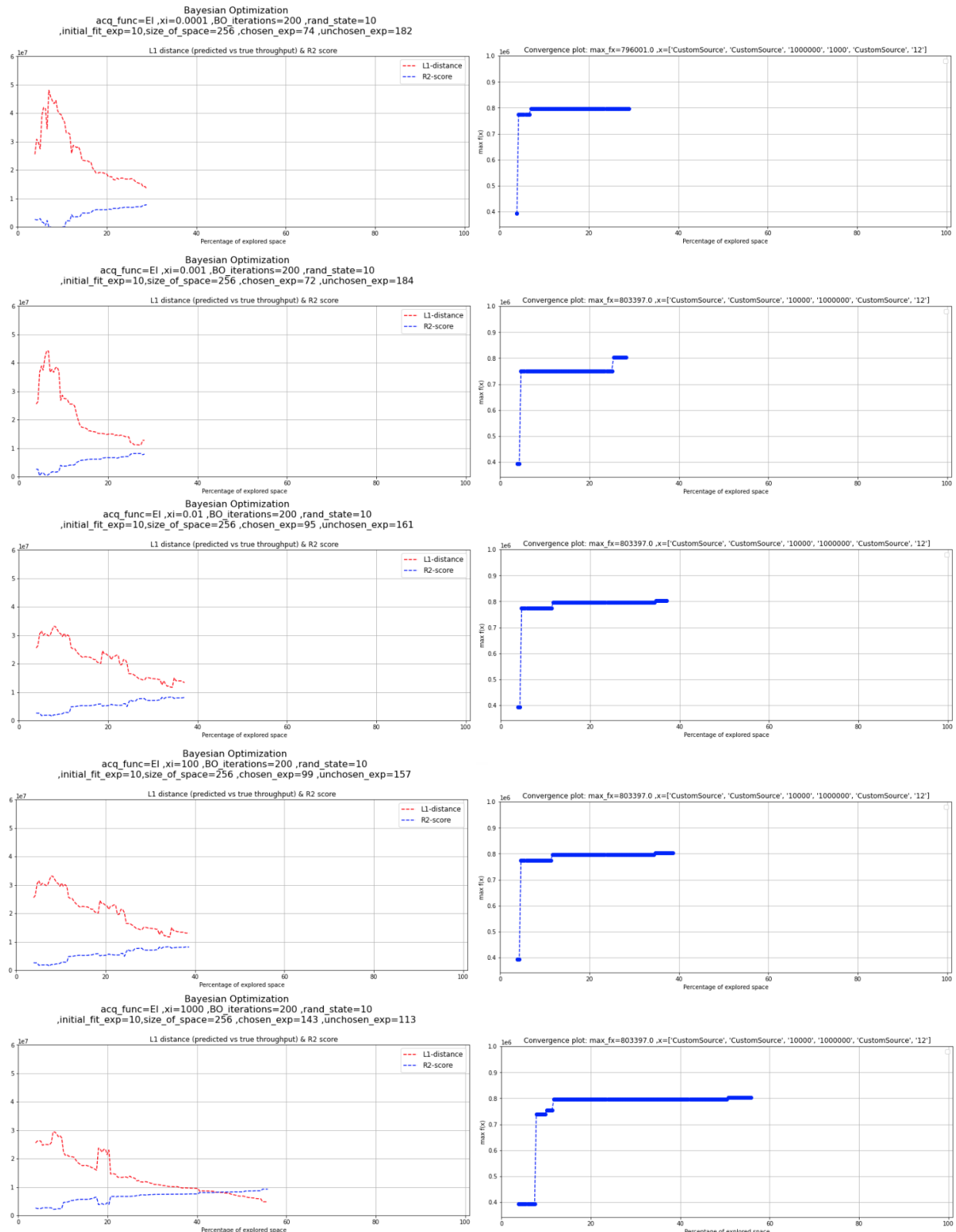



Figure 44: Accuracy (left) and convergence speed (right) of cost estimator using EI with 10 randomly chosen micro benchmarks for initial fitting, varying the xi (exploitation vs exploration) parameter.

 <p>Project supported by the European Commission Contract no. 825070</p>	<h3>WP5 T5.1 & T5.2</h3> <h3>Deliverable D5.1</h3>	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

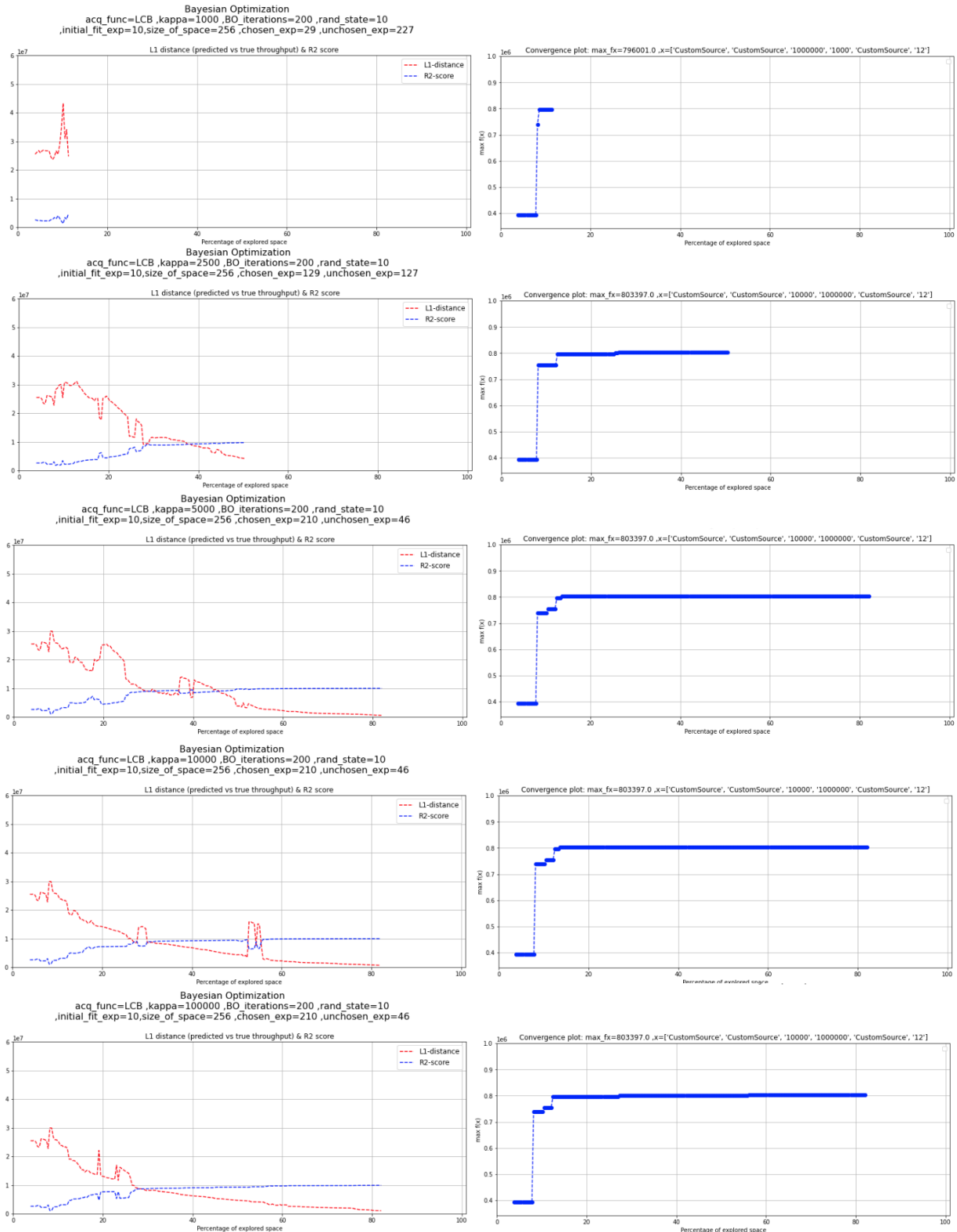



Figure 45: Accuracy (left) and convergence speed (right) of cost estimator using LCB with 10 randomly chosen microbenchmarks for initial fitting, varying the kappa (exploitation vs exploration) parameter.

In the next set of experiments, we investigate the effect of the number of microbenchmarks that are used for initial fitting before beginning the sampling, training process. We fix $\xi_i=100$ and $\kappa=100K$ for EI and LCB

 <p>Project supported by the European Commission Contract no. 825070</p>	<h3>WP5 T5.1 & T5.2</h3> <h2>Deliverable D5.1</h2>	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public


respectively. As shown in Figure 46 and Figure 47, the trend is that the more initial microbenchmarks we use, the higher the accuracy, in terms of L_1 distance and R^2 score, for lower percentages of the explored space. This is particularly true for LCB in Figure 47, while EI occasionally (for instance, switching from 8 to 10 initial microbenchmarks) provides worse L_1 distance values (but R^2 is improved).

The other characteristic that is affected, by providing more initial microbenchmarks, is how quickly the model converges to the final $\max f(x)$ value. In the right columns of Figure 46 and Figure 47 we observe that the steps of the blue line are fewer and the distance between these steps is shorter as we increase the number of initially fed microbenchmarks. As a whole, the percentage of the explored space when the training phase concludes is not affected in case of LCB in Figure 47, while for EI (Figure 46) the percentage of the explored space increases by up to 10%.

Among all the presented graphs we have seen so far, from Figure 43 to Figure 47, the best one in terms of convergence speed and accuracy of prediction is that of the last line of plots in Figure 47. There, LCB is used as the acquisition function with $\kappa=100K$ and 12 randomly chosen microbenchmarks are utilized for initial fitting. The L_1 distance steeply decreases, the R^2 approaches 1 even for low percentages of the explored microbenchmark space (left), while the training process also quickly approaches the final, maximum value of the objective function (throughput in our setup).

If we try to compare the last line of plots in Figure 47 with Figure 43 (fitting all experiments without using an acquisition function), we see that the L_1 distance and R^2 score lines of LCB meet at the half of the percentage of explored space used by the fitting process. The same happens for the converge plots where $\max f(x)$ is approached at almost $\sim 20\%$, for LCB, compared to $\sim 50\%$ for exhaustive fitting.

To further investigate the effect of choosing initial microbenchmarks for fitting, we perform another iteration of experiments, where we alter the number of fitted microbenchmarks as before but, instead of selecting them uniformly at random, we use Sobol sequences [Sob67]. In Figure 49, the generic observation for LCB is that choosing the initial microbenchmarks using Sobol sequences worsens the accuracy and the speed of convergence of the model. On the contrary, Figure 49 illustrates that choosing initial benchmarks employing Sobol sequences may be beneficial for EI compared to sampling initial points uniformly at random. This is due to the fact that the left column of Figure 48 shows that the training process approaches faster the final $\max f(x)$, exploring less (5%-10%) of the space of microbenchmarks. Moreover, Sobol sequences simultaneously provide similar L_1 distance and R^2 score values with the uniform sampling case.

	Project supported by the European Commission Contract no. 825070	WP5 T5.1 & T5.2 Deliverable D5.1	Doc.nr.:	WP5 D5.1
			Rev.:	1.0
			Date:	30/04/2020
			Class.:	Public

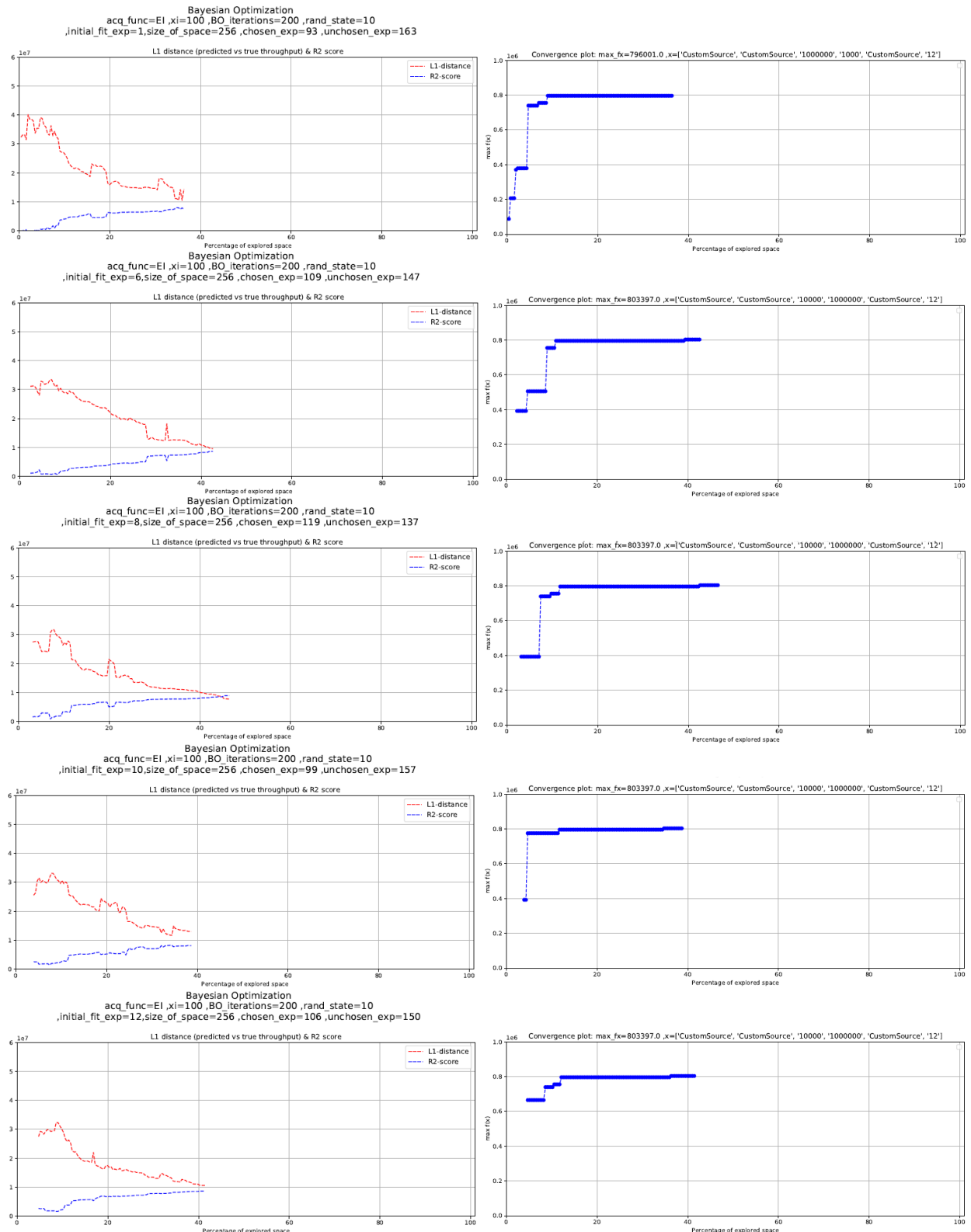



Figure 46: Accuracy (left) and convergence speed (right) of cost estimator using EI, for xi=100, varying the number of randomly chosen microbenchmarks for initial fitting initial_fit_exp=1,6,8,10,12 in successive rows of plots.

 Horizon 2020 European Union funding for Research & Innovation	Project supported by the European Commission Contract no. 825070	WP5 T5.1 & T5.2 Deliverable D5.1	Doc.nr.:	WP5 D5.1
			Rev.:	1.0
			Date:	30/04/2020
			Class.:	Public

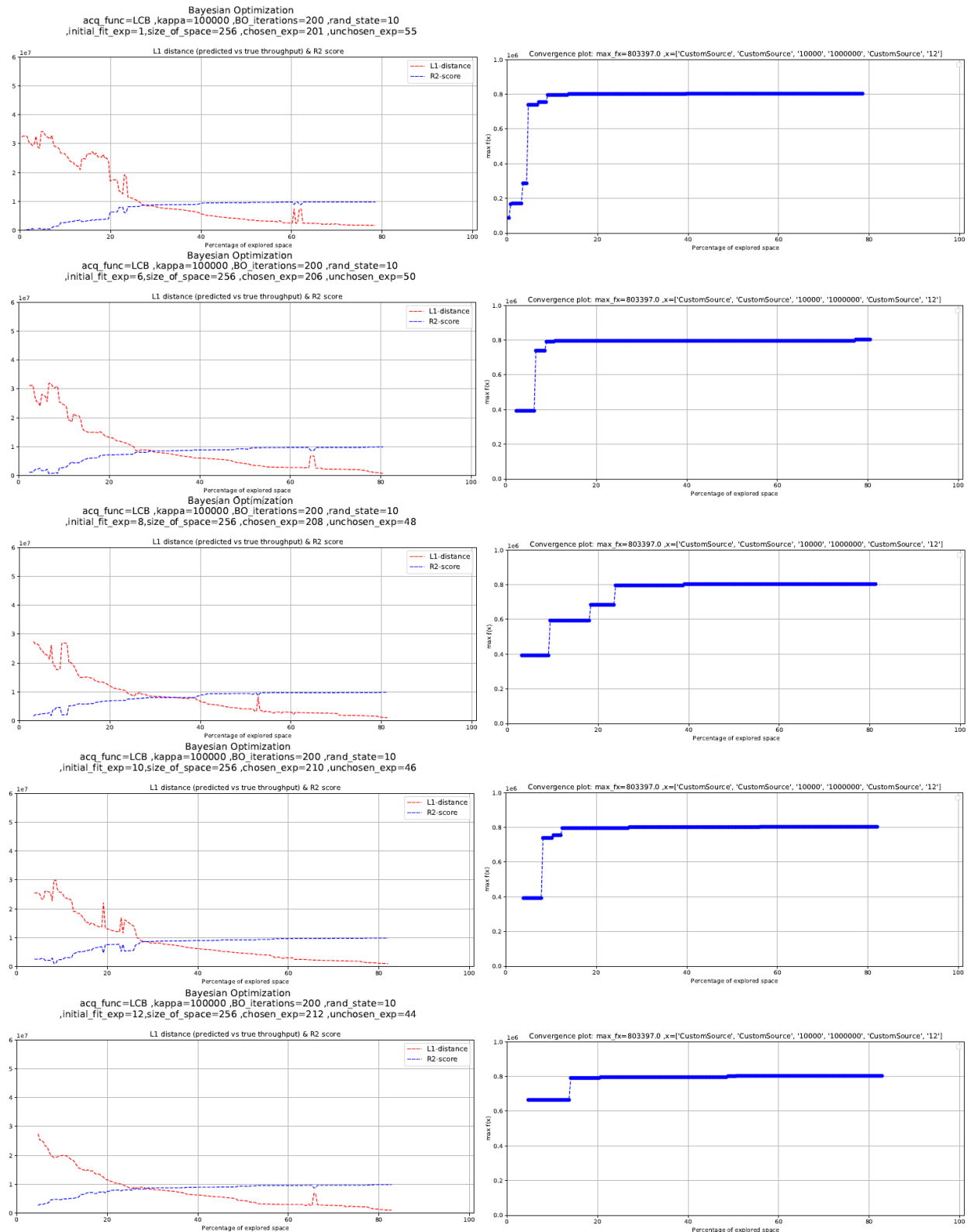



Figure 47: Accuracy (left) and convergence speed (right) of cost estimator using LCB, for kappa=100K, varying the number of randomly chosen microbenchmarks for initial fitting initial_fit_exp=1,6,8,10,12 in successive rows of plots.

 <p>Project supported by the European Commission Contract no. 825070</p>	<h3>WP5 T5.1 & T5.2</h3> <h2>Deliverable D5.1</h2>	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

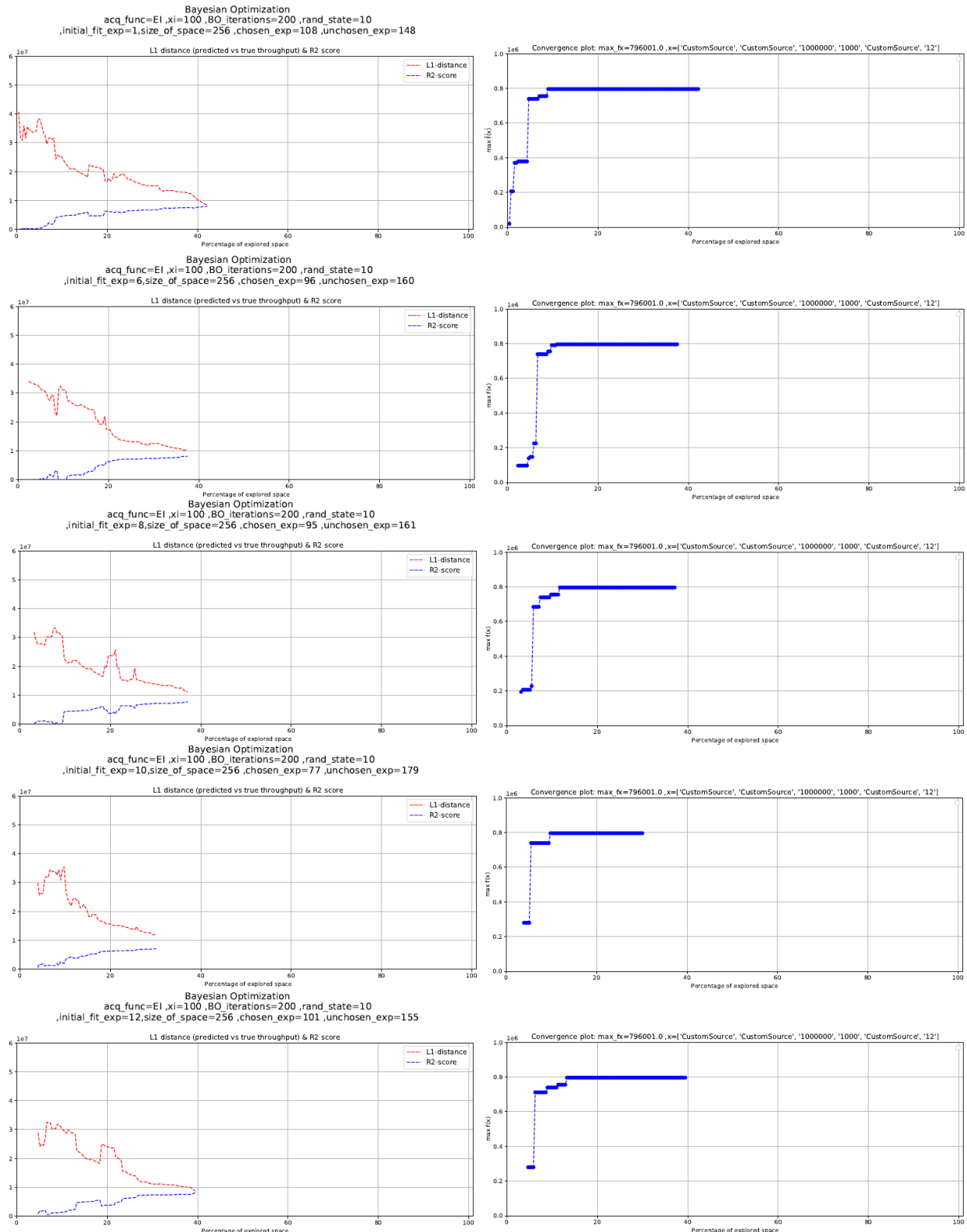



Figure 48: Accuracy (left) and convergence speed (right) of cost estimator using EI, for $\xi=100$, varying the number of Sobol sequence-chosen microbenchmarks for initial fitting $\text{initial_fit_exp}=1,6,8,10,12$ in successive rows of plots.

 <p>Project supported by the European Commission Contract no. 825070</p>	<p>WP5 T5.1 & T5.2 Deliverable D5.1</p>	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

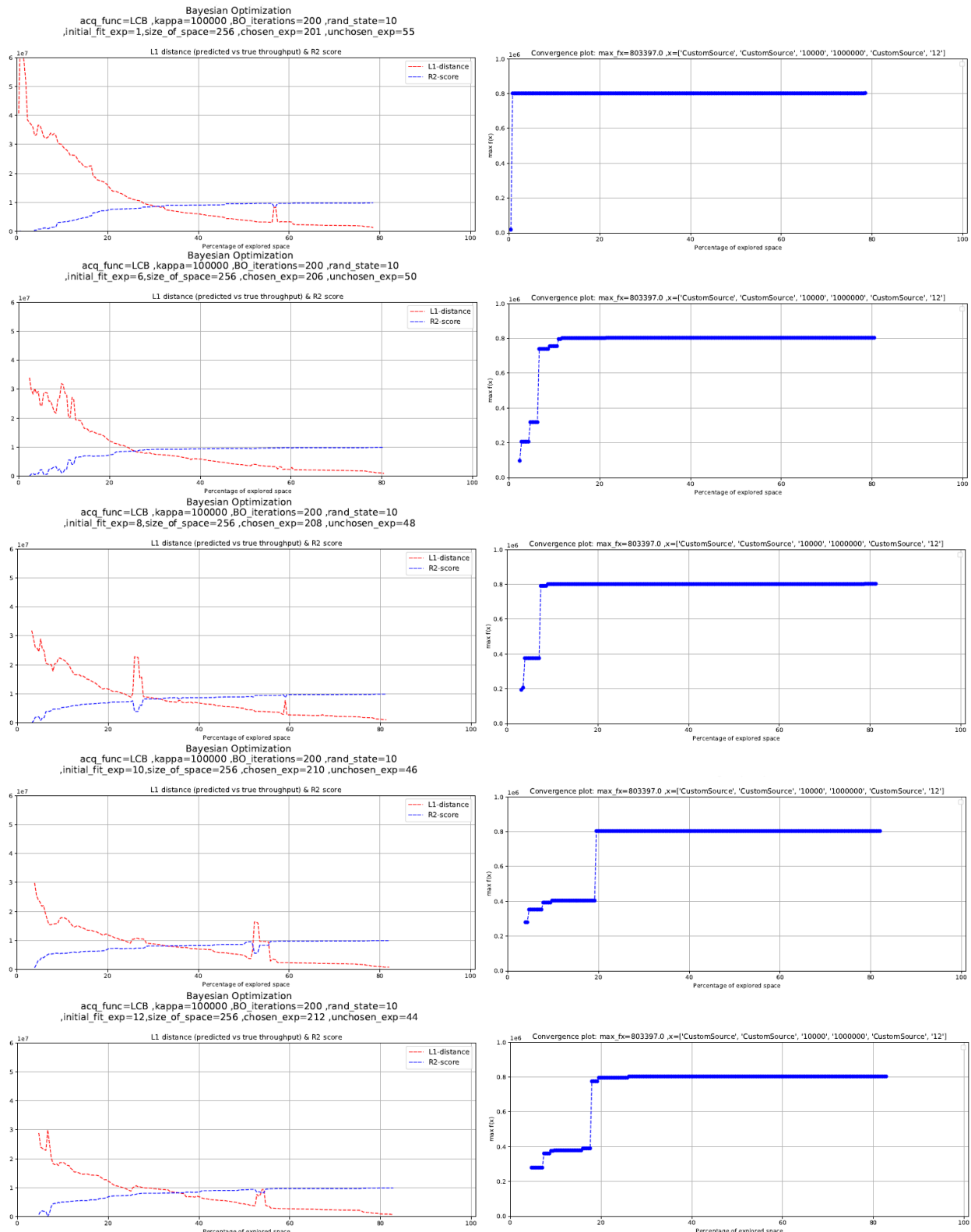



Figure 49: Accuracy (left) and convergence speed (right) of cost estimator using LCB, for $\xi=100$, varying the number of Sobol sequence-based chosen microbenchmarks for initial fitting $\text{initial_fit_exp}=1,6,8,10,12$ in successive rows of plots.

 <p>Project supported by the European Commission Contract no. 825070</p>	<h3>WP5 T5.1 & T5.2</h3> <h2>Deliverable D5.1</h2>	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

8.2 Comparative Benchmarks on Optimization Algorithms

We now compare the performance of the optimization algorithms we have already implemented (Exhaustive Search, A*-like) in terms of the number of physical workflows they examine during their execution (recall that the A*-like Algorithm dynamically prunes candidate physical workflows, while Exhaustive Search does not) as well as their execution time. Note that this is the number of candidate physical plans each algorithm considers during its execution, before devising the optimal one to be deployed. Similarly, the execution time of the algorithm accounts for the time it takes to the algorithm to find the physical workflow that should be deployed.

We use the workflows described in Section 7, for each use case. Since the logical workflows of the Life Science and the Financial use case have similar structure, the performance of the algorithms was similar, so we plot these two use cases together. In all experiments of this section, we leave the intrinsic optimizer of the Big Data platform to set the parallelization degree μ for each physical operator participating in a physical workflow.

In the first of the experiments we run the Exhaustive Search and the A*-like Algorithm over a network composed of a single cluster hosting three Big Data platforms, assuming each of the involved operators is implemented in all three platforms. Figure 50 shows the ability of the A*-like algorithm to prune the number of examined physical workflows, in a per use case fashion (horizontal axis). More precisely, we observe that in the Life Science and Financial workflows the A*-like Algorithm reduces the number of examined physical workflows (vertical axis – Log Scale) by almost 6 orders of magnitude, while, for the Maritime workflow, the A*-like Algorithm reduces the number of examined physical workflows by almost 5 orders of magnitude.

Upon switching to Figure 51, where we plot the execution time of the respective algorithms on the vertical axis (Log Scale, again), we observe that, in the Life Science and Financial workflows, the A*-like Algorithm is faster by 3.5 orders of magnitude, while, for the Maritime workflow, the A*-like Algorithm reduces the time it takes to devise the preferable physical workflow by more than two orders of magnitude.

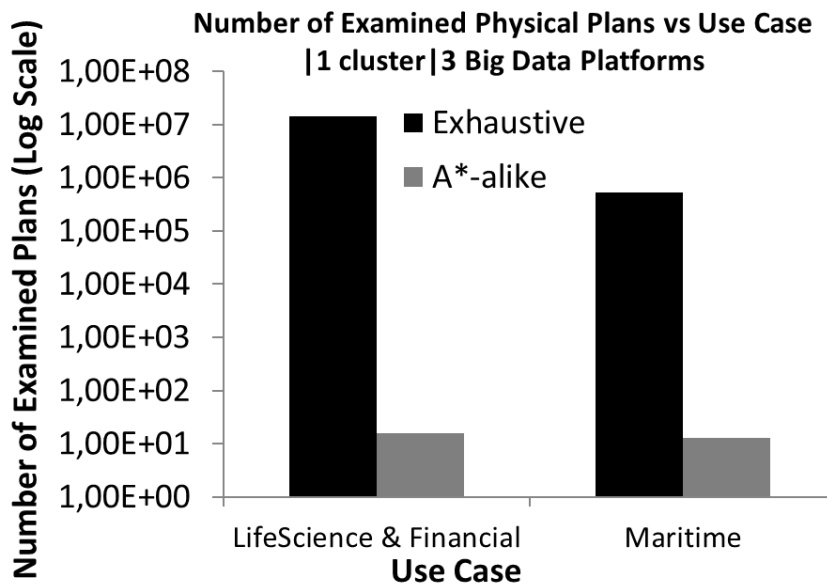


Figure 50: Number of Examined Physical Workflows per use case: Exhaustive vs A*-like Optimization Algorithms

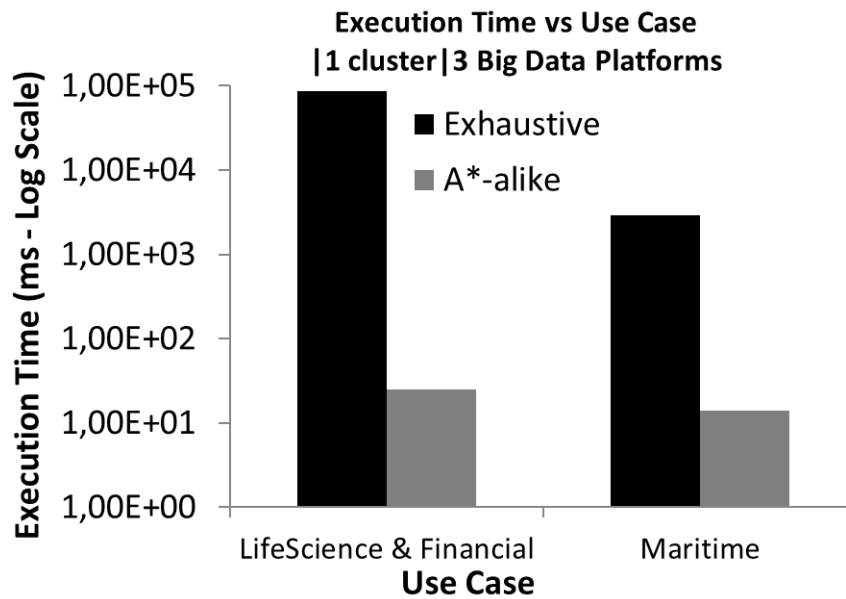


Figure 51: Execution time for determining the prescribed physical workflow per use case: Exhaustive vs A*-like Optimization Algorithms

In order to study the behavior of the algorithms under various numbers of operators' implementations in different Big Data platforms, in Figure 52, we plot in the vertical axis the ratio between the number of physical workflows examined by the Exhaustive Search Algorithm over the respective number of the A*-like alternative. In this experiment we assume that there are implementations for all operators in all available Big Data platforms. As shown in the figure, the more we increase the number of Big Data platforms (horizontal axis), i.e., from 2 to 3, the higher the gains of the A*-like algorithm compared to the Exhaustive Search candidate. In particular, for 2 Big Data platforms, A*-like examines almost 3 orders of magnitude fewer physical plans while this ratio approximately doubles (in orders of magnitude) upon switching to 3 Big Data platforms. In Figure 52, for 1 Big Data platform the Exhaustive Search Algorithm creates just 1 physical workflow, while the A*-like, due to its design, instantiates various partial physical workflows. Therefore, it inserts more partial plans in the priority queue, that converge to the same physical workflow.

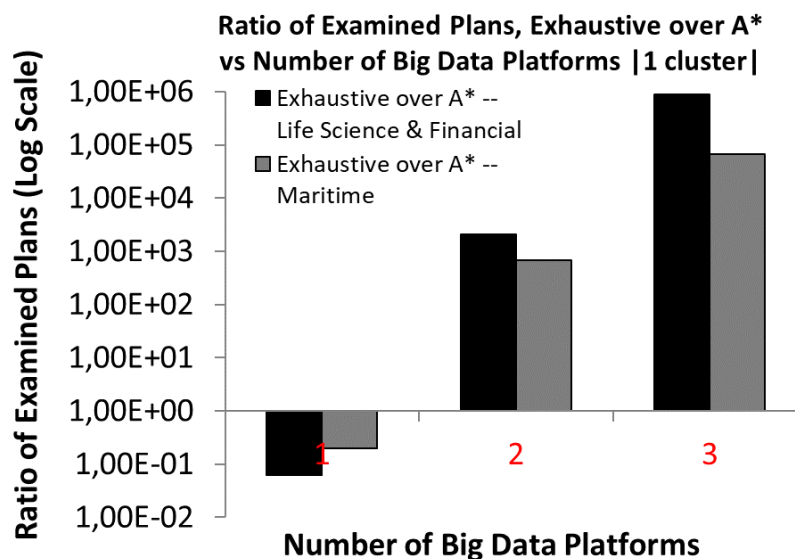

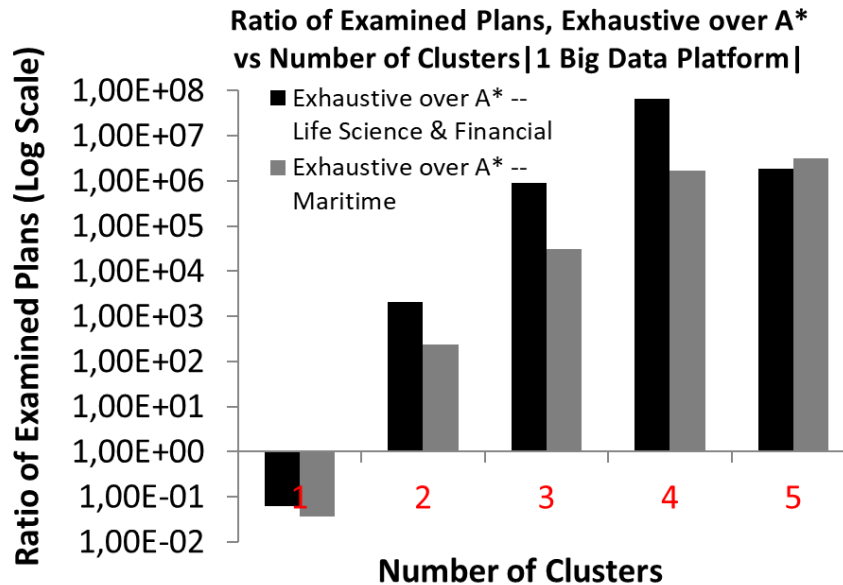


Figure 52: Ratio of Examined Plans varying the number of available Big Data platforms: Exhaustive vs A*-like Optimization Algorithms

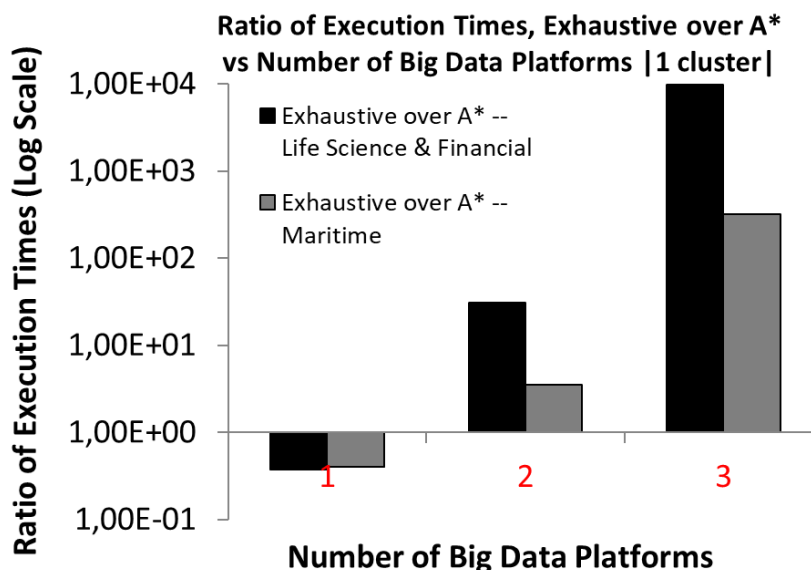
 <p>Project supported by the European Commission Contract no. 825070</p>	<p>WP5 T5.1 & T5.2 Deliverable D5.1</p>	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

Let us now see what happens if we keep the number of available implementations for each operator steady to, for instance, 1 Big Data platform, but vary the number of clusters in the network from 1 to 5. As Figure 53 illustrates, the A*-like Algorithm can examine from 2 and up to 8 orders of magnitude fewer physical workflows across various network sizes and use case setups.




**Figure 53: Ratio of Examined Plans varying the number of available clusters:
Exhaustive vs A*-like Optimization Algorithms**

In Figure 51 we saw that the ratio of Exhaustive Search over A*-like upon considering execution time was lower compared to the respective ratio in terms of examined physical workflows. To further study this issue, in Figure 54 we plot the ratio of respective execution times varying the number of available Big Data platforms. If we compare the current figure with Figure 52, we again see that the ratios are lower in terms of execution time. For instance, for 2 Big Data platforms in Figure 52 the ratio of examined plans is 3 and 2 orders of magnitude for the (Life Sciences, Financial) and the Maritime use case, respectively. In Figure 54, the corresponding ratios in terms of execution time are reduced to 1.5 and 0.5 orders of magnitude for each use case.



**Figure 54: Ratio of Algorithm Execution Time varying the number of available Big Data platforms:
Exhaustive vs A*-like Optimization Algorithms**

 <p>Project supported by the European Commission Contract no. 825070</p>	<p>WP5 T5.1 & T5.2 Deliverable D5.1</p>	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

The difference between these ratios is due to the fact that A*-like consumes more memory for keeping plans in the priority queue, most of which are never dequeued and thus they are pruned altogether at the end of the algorithm. These are the plans that are never further examined. Moreover, the A*-like Algorithm keeps more complex data structures and performs more memory accesses during its operation, e.g., for determining Ending Nodes and for selecting the next operator to add (see Section 5.1). On the other hand, the memory requirements of the Exhaustive Search Algorithm are negligible because in each iteration it fixes a plan and simply computes its cost so that it updates the currently best physical workflow, if needed. When scalability to high number of clusters and Big Data platforms is considered, the Exhaustive Search Algorithm fails to improve, in contrast to the A*-like alternative. Therefore, our immediate goal is to improve the implementation of the A*-like Algorithm so that more efficient memory management is achieved, since based on the assessed ability of the algorithm to prune examined physical workflows, a more fine-tuned implementation can make it scale even better than what we showed in the previous plots.

8.3 Benchmarks on Optimized Workflow Execution


To show the improvement in performance yielded by having our optimization algorithms (Exhaustive Search and A*-like) prescribing the physical workflow of the logical one fed to the optimizer, we utilize a Kafka cluster with 3 Dell PowerEdge R320 Intel Xeon E5-2430 v2 2.50GHz machines with 32GB RAM each and one Dell PowerEdge R310 Quad Core Xeon X3440 2.53GHz machine with 16GB RAM. We use a Flink cluster with 10 Dell PowerEdge R300 Quad Core Xeon X3323 2.5GHz machines with 8GB RAM each. We employ a real dataset composed of 5000 stocks of Level 1 and Level 2 data from the Financial use case of INFOR provided by the Spring Techno partner in the project.

We feed the optimizer with the workflow discussed in Section 7.2. We ingest every related tuple in a Flink cluster via Kafka and we essentially let the optimizer decide the parallelization degree of the physical plan execution for each operator and in some experiments, we allow synopsis-based optimization. Note that we form a worst-case scenario for the effect of our optimizer where it can only improve performance by choosing the parallelization degree (not the cluster or the Big Data platform). In fact, we have split the workflow into two parts for more detailed experimentation purposes. The upper part of Figure 37 which computes pairwise correlations among stocks is included in one experiment along with the rest of the workflow, while the lower part of the same figure involving the StreamKM operator is tested in a separate experiment along with the rest of the workflow.

For pairwise correlation of stocks in the first of our experiments, indicatively, we note that when 5K stocks are monitored, the pairwise similarity comparisons that need to be performed by naive approaches are 12.5 M, which shows the importance of optimizing the execution of the workflow and, as we will shortly see, of the synopsis-based optimization approach we propose in Section 6.1.

In Figure 55 we measure the performance of the physical workflow instructed by the Optimizer Component against two alternative physical plans. One that does not use parallelization or synopses and another one that uses synopses, but still does not distribute the processing. More precisely, the compared approaches are:

- *Naive*: This is the baseline approach which involves a physical workflow where sequential processing of incoming tuples without parallelism or any synopsis is performed. Pairwise comparisons are executed without using the DFT operator we described, but on the original data. So that no enhanced horizontal or vertical scalability is achieved and no optimization whatsoever is performed.
- *DFT(NoParallelism)*: The performance of DFT(NoParallelism) physical execution plan utilizes DFT synopses to (via hashing) bucketize financial time series so as to prune pairwise comparisons for stocks that are not hashed in the same bucket. DFT also achieves dimensionality reduction of the original financial time series, but no parallelism is used for executing the workflow.
- *Parallelism(NoDFT)*: This physical workflow performs parallel processing prescribed by the optimizer, but there is not any synopsis defined in the optimizers dictionary. In other words, the optimizer does not use the ideas of synopsis-based optimization in Section 6.1 and pairwise comparisons of stocks are executed without using the DFT operator of the SDE.

	Project supported by the European Commission Contract no. 825070	WP5 T5.1 & T5.2 Deliverable D5.1	Doc.nr.:	WP5 D5.1
			Rev.:	1.0
			Date:	30/04/2020
			Class.:	Public




- *SDEaaS(DFT+Parallelism)*: This is the performance of synopsis-based optimization which combines the virtues of parallelism and data summarization (for bucketizing financial time series so that pairwise correlation comparisons are pruned and for dimensionality reduction) and is termed as *SDEaaS*, because the SDE Component of INFOR architecture is provided as a service (see Deliverable D6.1). Here, we have defined DFT as an equivalent operator for pairwise comparisons in the optimizers dictionary and the optimizer chooses this approximate operator instead of the exact one to perform pairwise comparisons.

It is important to emphasize, that *SDEaaS(DFT+Parallelism)* shows the performance of plans prescribed by either of the implemented optimization algorithms of Section 4.5 and Section 5.1 upon allowing the SDE to be considered as an additional execution platform for an operator. This is to be declared in the optimizer dictionary. In cases when *SDEaaS(DFT+Parallelism)* performs better than *Parallelism(NoDFT)* in the experiments that we present in this section, this does not mean that using synopsis is better than the optimizer, but the fact that if we equip the INFOR Optimizer Component with the ability to perform synopsis-based optimization in one of the ways discussed in Section 6.1, using our devised optimization algorithms, the physical workflows it will prescribe ensure interactivity to the major extent.

Each line in the plot of Figure 55 measures the ratio of throughputs of each examined physical execution plan over the Naive approach varying the amount of monitored stock streams. Let us first examine each line individually. It is clear that when we monitor few tens of stocks (50 in the figure), the use of DFT in the *DFT(NoParallelism)* marginally improves (1.5 times higher throughput) the throughput of the Naive approach. On the other hand, the use of the INFOR optimizer *Parallelism(NoDFT)* improves the Naive by 2.5 times. The *SDEaaS(DFT+Parallelism)*, taking advantage of both the synopsis and parallelism improves the Naive by almost 4 times. Note that when 50 streams are monitored, the number of performed pair-wise similarity checks in the workflow of Figure 37 for the Naive approach is 2.5K/2.

This is important because, according to Figure 55, when we switch to monitoring 500 streams, i.e., 250K/2 similarity checks are performed by Naive, the fact that the *Parallelism(NoDFT)* physical execution lacks the ability of the DFT to bucketize time series and prune unnecessary similarity checks, makes its throughput approaching the Naive approach. This is due to DFT Correlation operator in Figure 38 starting to become a computational bottleneck for *Parallelism(NoDFT)* in the workflow. On the contrary, the *DFT(NoParallelism)* line remains steady when switching from 50 to 500 streams. The *DFT(NoParallelism)* physical workflow starts to perform better than *Parallelism(NoDFT)* on 500 monitored streams showing that the importance of using synopsis-based optimization becomes higher than the importance of having the optimizer simply inscribing the parallelization degree, as more streams are monitored. The line corresponding to the *SDEaaS(DFT+Parallelism)* physical workflow exhibits steady behavior upon switching from 50 to 500, improving the Naive execution plan by 4 times, the *DFT(NoParallelism)* execution plan by 3 and the *Parallelism(NoDFT)* one by 3.5 times.

 Project supported by the European Commission Contract no. 825070	WP5 T5.1 & T5.2 Deliverable D5.1	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

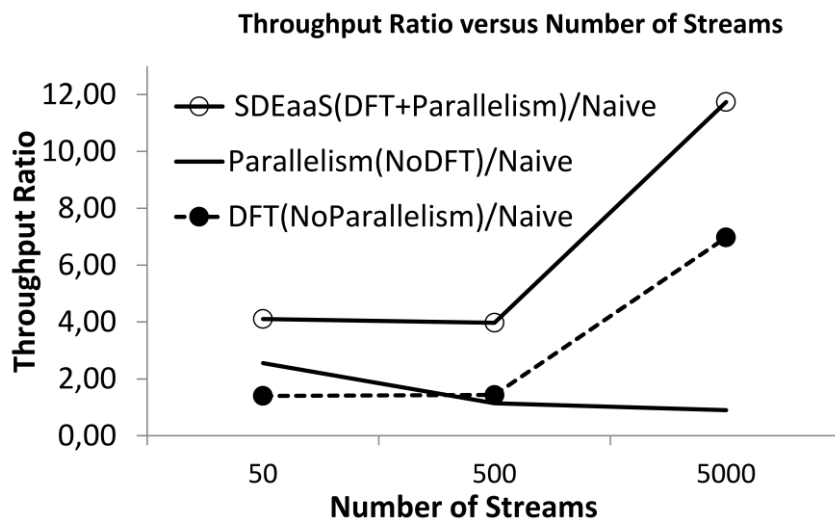


Figure 55: Performance of different execution plans prescribed by INFOR's optimizer Parallelism(NoDFT), SDEaaS(DFT+Parallelism) versus two (Naïve, DFT(NoParallelism)) baselines for the physical workflow of the upper part of Figure 37 which computes pairwise similarities of stocks.

The most important findings come upon switching to monitoring 5000 stocks (25M/2 similarity checks using Naive or Parallelism(NoDFT)). Figure 55 says that because of the lack of synopses that ensure vertical scalability (see Section 6.1.2), the Parallelism(NoDFT) physical execution plan becomes equivalent to the Naive one. The DFT(NoParallelism) execution plan improves the throughput of the Naive and of Parallelism(NoDFT) by 7 times. The SDEaaS(DFT+Parallelism) physical workflow exhibits 11.5 times better performance compared to Naive, Parallelism(NoDFT) and almost doubles the performance of DFT(NoParallelism). This validates the potential of SDEaaS(DFT+Parallelism) to support interactive analytics upon judging similarities of millions of pairs of stocks. In addition, studying the difference between DFT(NoParallelism) and SDEaaS(DFT+Parallelism) we can quantify which part of the improvement over Naive, Parallelism(NoDFT) is caused due to comparison pruning based on time series bucketization, i.e., vertical scalability and which part is yielded by parallelism. That is, the use of DFT for bucketization and dimensionality reduction increases throughput by 7 times (equivalent to the performance of DFT(NoParallelism)), while the additional improvement entailed by SDEaaS(DFT+Parallelism) is roughly equivalent to the parallelization degree prescribed by the optimizer. This indicates the success of synopsis-based optimization in integrating the virtues of data synopsis and parallel processing.

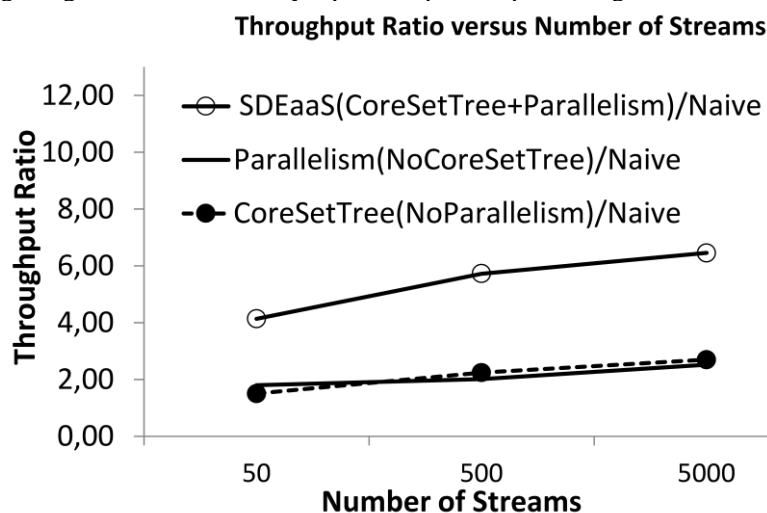




Figure 56: Performance of different execution plans prescribed by INFOR's optimizer Parallelism(NoCoreSetTree), SDEaaS(CoreSetTree +Parallelism) versus two (Naïve, CoreSetTree (NoParallelism)) baselines for the physical workflow of the lower part of Figure 37 which computes clusters of stocks.

 <p>Project supported by the European Commission Contract no. 825070</p>	<p>WP5 T5.1 & T5.2 Deliverable D5.1</p>	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public



We then perform a similar experiment for the lower part of the workflow in Figure 37 which engages the StreamKM [AMR+12] clustering operator instead of the DFT correlation one. In particular, in this experiment the Naive execution plan corresponds to StreamKM++ clustering without parallelism and coreset sizes equivalent to the original data points (time series). The Parallelism(NoCoresetTree) execution plan involves performing StreamKM++ with coreset sizes equivalent to the original data points, but allowing the optimization algorithm to prescribe the parallelization degree. The CoresetTree(NoParallelism) exploits the CoresetTree synopsis but uses no parallelism, while SDEaaS(CoresetTree +Parallelism) combines the two.

The conclusions that can be drawn from Figure 56 are very similar with what we discussed in Figure 55. However, the respective ratios of throughput over the Naive physical workflow execution plan are lower (2-3 times higher throughput than the second best candidate in Figure 56). This is by design of the data mining algorithm and the reason is that the clustering procedure includes a reduce step with the only admissible parallelization degree of 1. This is in contrast with the DFT Correlation operator of the previous experiment which can be executed by different processing units independently for each bucket.

 Project supported by the European Commission Contract no. 825070	WP5 T5.1 & T5.2 Deliverable D5.1	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

9 Related Work

This section describes related work on (a) statistics collection, (b) query optimization, including learning query optimizers, (b) cost estimation, including join ordering, cardinality and selectivity estimation, (c) query runtime prediction using machine learning, and (d) query and operator cost modelling in multi-engine systems, (e) optimizations in high performance computing.

9.1 Statistics Collection and Monitoring Solutions

Monitoring and collecting statistics is an essential task for analysing performance and behaviour of distributed systems. The goal of the first monitoring systems [MCC04], [Nagios], [Zabbix] is to monitor infrastructure or, in other words, to monitor resource metrics. The limitation of non-conclusive infrastructure metrics motivated the design and development of monitoring systems aiming to enable effective monitoring on the service-level, often known as Application Performance Monitoring (APM) systems. The general architecture of these systems is to have a set of APIs that interface with resources and services and collect raw information; a collection mechanism that may perform some preliminary filtering and aggregation; a storage system that holds collected data for analysis; a query mechanism to compute metrics; and a visualisation layer to interface with system managers and operators.

For instance, the popular ELK stack that we adopt is composed by Logstash, ElasticSearch, and Kibana. An alternative is the Graphite/Grafana combination [Grafana], an open-source monitoring system that only stores numeric time-series data and display plots of stored data in real-time. Prometheus [Prometheus] is a very popular open-source full monitoring system initially developed at SoundCloud. It collects measurements related to both infrastructure and services and stores them in their own time series database. A flexible query language is also provided for querying metrics and the status of the monitored system, thus enabling the relation of infrastructure and service-level metrics. Nonetheless, this relation is limited since it does not consider dependencies between processes and thus services, in a higher level, which consequently represents a limit on the scope of retrievable knowledge.


A deeper insight into monitored systems can be achieved by instrumentation that reveals system and operation structure. For instance, the CoherentPaaS X-Ray subsystem targets federated polyglot query systems and is able to trace execution across multiple query engines with a combination of ad-hoc and general-purpose instrumentation [GP15]. Systems such as Falcon [NMP18] are able to trace interactions in distributed systems and derive causality relations, key to verification of distributed properties, although at small scale.

DynaTrace [DynaTrace] is a cloud-based Monitoring-as-a-Service solution. Besides raw resource information, it aims at automatically detecting system structure such as containers, processes, services, and applications. However, it lacks a query language for detailed analysis.

All these approaches are orthogonal to our optimization approach. Since we use Bayesian optimization for our cost estimation, raw metric collection suffices while interrelations of processes can be modelled similarly by benchmarking cost estimations for parts of or entire workflows. Additionally, we preferred to rely on statistic collection tools that are either available for every platform that works over JVMs or are currently used in the infrastructure available to INFORE, as is the case with Slurm in Barcelona's supercomputer center.

9.2 Query Optimization

There has been extensive work in query optimization since the early '70s and presenting the breadth and depth of this work is out of the scope of this document. Still, the specific characteristics of the INFORE Architecture that is designed for performing efficient analytics at scale over data stream workflows running on a multiplicity of distributed, heterogeneous platforms make it clear that traditional, relational style optimizers (e.g., Starbust [HFLP89] and Volcano/Cascades [GrMc93, Grae95]) do not suffice to overcome the challenges in such a complex environment. In a streaming context, several techniques have been proposed including batching, placement, load balancing, load shedding, state sharing, operator separation, operator reordering, fusion, redundancy elimination, algorithm selections, and so on [HSGS19]. Although such techniques are still applicable in INFORE, applying them to a multi-platform setting is not straightforward.

 <p>Project supported by the European Commission Contract no. 825070</p>	<p>WP5 T5.1 & T5.2 Deliverable D5.1</p>	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

Due to the large variety and heterogeneity of the operators considered in INFORE (see Section 2.4), the INFORE Optimizer follows a black-box approach, where the operator semantics and cost are not, generally, available. Hence, the Optimizer needs to *learn* the operators' behavior over time. This is similar to another line of work in query optimization, namely the *learning query optimizers*.


[MaLR03] presents Leo, DB2's L^Earning Optimizer, which learns from its mistakes by adjusting its cardinality estimations over time. Although Leo was a promising start, it still relied on a human-engineered cost model, a hand-picked search strategy, and developer-tuned heuristics. In addition, Leo had the ability to learn better cardinality estimations, but not new optimization strategies (e.g., how to account for uncertainty in cardinality estimates, operator selection, etc.). [TzSJ08] proposes an unsupervised learning approach to adaptive query optimization [DeIR07], which models query execution with eddies [AvHe00] as a reinforcement learning problem with quantitative rewards. This was a promising idea for a specific class of conjunctive selection and join queries, but it did not consider practical aspects such as correlated selectivities and complex (e.g., content-based) routing techniques with adaptation characteristics.

Many argue that the advances in machine learning can be used to build better, smarter, and easier to use (as in maintain, tune, optimize) data management systems. In fact, the argument goes further and supports that learned components can fully replace core components of a data management system, such as indices, sorting algorithms, and query execution [BLC+15, KBC+18]. As an example, [KAB+19] presents SageDB, a vision about a new type of a data processing specialized to an application through code synthesis and machine learning. SageDB models the data distribution, workload, and hardware to learn the structure of the data, optimal access methods, and query plans. [TWM+19] presents SkinnerDb that uses reinforcement learning to improve adaptive query processing. To that end, this system uses a specialized query execution engine, which has some limitations (e.g., operator pipeline, parallelism), but still the ideas it describes although not directly applicable to production systems, they can still be useful in improving learning query optimizers.

[MNM+19] presents Neo (NEural Optimizer), an attempt to build an end-to-end learned optimizer. Neo learns to make decisions about join ordering, physical operator selection, and index selection. But it cannot learn these tasks from scratch and makes a few assumptions such as (a) it requires a-priori knowledge about all possible query rewrite rules to guarantee semantic correctness; (b) it works with project-select-equi-join-aggregate-queries; (c) it does not generalize from one database to another. However, it can generalize to unseen queries containing any number of known tables. The Neo paper reports that the learning process can be extremely time consuming (days, weeks). To reduce it, Neo uses a technique called learning from demonstration [HVP+18], which is it employs a traditional query optimizer (i.e., PostgreSQL optimizer) as a source of expert demonstration to bootstrap its initial policy. Neo replaces most traditional optimizer components with machine learning models. The query representation is performed through features instead of an object-based operator tree à la Volcano [GrMc93]. The cost model is represented as a deep neural network (DNN) instead of cost formulae. Neo uses a DNN-guided learned best-first search strategy instead of plan space enumeration or dynamic programming. For cardinality estimation, it uses histograms and/or a learned vector embedding scheme, combined with a learned model.

In order to train a neural network to predict the latency of partial or complete query plans, it employs two encodings: (a) a query encoding (e.g., as an R-Vector inspired by word2vec [MCCD13]) that involves e.g., tables and predicates, but it is independent of the query plan, and (b) a plan encoding (as a tree of vectors) that represents a partial execution plan. The query encoding provides the predicate semantics and it plays a strategic role to the efficacy of the method; however, producing it is a slow operation (each row in every table is treated as a training sequence, augmented by rows in other tables that are functionally dependent) and currently, it does not lend itself nicely to database changes.

[WJA+18] presents a machine learning based approach, in the context of shared cloud workloads, to learn cardinality models from previous job executions and use them to predict the cardinalities in future jobs. The core idea is to extract overlapping subgraph templates that appear over multiple query graphs and learn cardinality models over varying parameters and inputs to those subgraph templates. The paper describes the application of three models: linear regression, Poisson regression, and multi-layer perceptron (MLP) neural network. Although the latter in theory could provide a more sophisticated solution, in practice the paper argues that training and using an MLP for cardinality estimation is more challenging as (a) it requires careful designing of the neural network architecture as well as a significant hyper-parameter tuning effort, (b) without “enough” training data for a given subgraph

 <p>Project supported by the European Commission Contract no. 825070</p>	<p>WP5 T5.1 & T5.2 Deliverable D5.1</p>	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

template, there is the risk of overfitting, and (c) it is more challenging to explain and justify the output of MLP to human analysts even though the output might as well be quite an accurate prediction of the cardinality.

9.3 Cost Estimation


Query optimization largely depends on cardinality and selectivity estimation, and in particular, on having reasonably good estimates for intermediate result sizes (e.g., as in join-crossing correlations). Many approaches have been proposed employing a large variety of techniques e.g., histograms [BrCG01, DeGR01, GKTD05, MuDe88, PoGI99, ShDG17], entropy [MHK+07, ReSu12], samples [HNSS96, EsNa06, HeKM15] and index-based sampling [LRG+17], histograms and samples [MuMK18], probabilistic models [GeTK01, TzDJ11], randomized hashing and data sketching [CaBS19], and so on. These approaches work with assumptions (e.g., attribute value independence – AVI, uniformity, data independence) and when these assumptions cannot be met the techniques usually fall back to an educated guess. Hence, in practice the estimates are routinely wrong by orders of magnitude causing slow queries and unpredictable performance.

Motivated by such limitations, another line of work departs from traditional techniques and exploits machine learning to improve cardinality estimation. A very first attempt to use neural networks for cardinality estimation for udf predicates dates to 1998 [LaZh98]. Several works have focused on optimizing specific operators using reinforcement learning. These works aim to learn more efficient search strategies for the best join ordering [e.g., KYG+18, MaPa18], to improve cardinality estimation [e.g., KKR+19, NMM+20, WHT+19] or selectivity estimation [e.g., DWZ+19, LXY+15, PaZM18, YLK+19,], or to learn the entire plan generation process through reinforcement learning [OBGK18]. Although in general these works do not describe how the techniques introduced lead to improved query plans [LRG+18], overall it is a promising start.

[KYG+18] and [MaPa18] combine reinforcement learning with a human-engineered cost model to automatically learn search strategies to navigate the space of possible join orderings. These methods rely on the optimizer’s heuristics for cardinality estimation, physical operator selection, and plan cost estimation. Furthermore, they make assumptions that, in general, are hard to meet in practice. [KYG+18] assumes perfect cardinality estimation for predicates over the base table.

[KKR+19] formulates cardinality estimation as a supervised learning problem, where query features is the input and the estimated cardinality is the output. Query features are expressed as sets (e.g., $(A \bowtie B) \bowtie C$ and $A \bowtie (B \bowtie C)$ are both represented as $\{A, B, C\}$) and fed to a multi-set convolutional network (MSCN). This saves capacity by avoiding numerous permutations and results into smaller models. The join enumeration and cost model are left to the query optimizer. This approach builds on sampling-based estimation by including cardinalities or bitmaps derived from samples into the training signal. As this work studies join-crossing correlations, it is different from works that create samples per table samples and sketches, which focus on single-table queries and are vulnerable to the 0-tuple problem (empty base table samples) [EsNa06, WuNS16]. Each table is represented by a unique one-hot vector identifying the table id and optionally, the number of qualifying table samples or a bitmap indicating their positions. Joins are represented with a unique one-hot encoding. For predicates of the form (col, op, val) , columns (col) and operators (op) are represented one-hot vectors and values (val) are represented as a normalized value in $[0,1]$ w.r.t. the minimum and maximum values of the respective column after logarithmization obtained from the training set. Note that this needs to be recomputed (i.e., it requires re-training) when data changes. The MLP (multi-layer perceptrons) modules used in the model are in general are two-layer fully connected neural networks with ReLU activation functions for hidden layers. The model is trained to minimize the q-error (i.e., the factor between an estimate and the true cardinality or vice versa) [MoNS09] and the Adam optimizer [KiBa15] is used for training.

To handle the cold start problem, the authors obtained an initial training corpus by generating random queries (unique queries containing up to two joins) based on schema information and recording their true cardinalities, while skipping queries with empty results. They also enrich the training data with information about materialized base table samples; for each table in a query, the corresponding predicates on a materialized sample are evaluated and the query is annotated with the number of qualifying samples. The training is performed on an immutable database snapshot. Although the method does not support updates, in the presence of those the main challenge as identified by the authors would be to address catastrophic forgetting when data distribution shifts over time [KPR+16]; the neural network would overfit to the most recent data and forget what it has learned in the past.

 <p>Project supported by the European Commission Contract no. 825070</p>	<p>WP5 T5.1 & T5.2 Deliverable D5.1</p>	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

[WHT+19] proposes a cardinality estimation technique based on building smaller neural networks (models) representing local models, each of which focuses on a small part of the database schema. This differs from most approaches that consider a global approach, by creating a neural network (global model) over the entire database schema. The local models are built at the granularity of n -ary joins and their corresponding filter predicates occurring at a given workload. The local approach has two advantages: (a) the local models are simpler and smaller in structure, and (b) the query sampling is less sparse. The approach is presented in the context of equi-joins with selections on non-key predicates, and employs a regression neural network whose input is a vectorized query (with one-hot encoding) and its output is the estimated cardinality of the query.


[LXY+15] uses neural networks to learn and approximate selectivity functions that take a bounded range on each column as input, effectively estimating selectivities for all relational operators. It considers queries with range predicates, each having a lower value and an upper value specified only by non-strict range operators (i.e., \leq and \geq). Presumably (but not shown in the paper) the technique could be expanded to other relational operators as OR, IN, NOT, etc. Then, a selectivity function is expressed in terms of the lower and upper values of all predicates involved in the query, and a three-layer neural network with a sigmoid activation function is used to produce the output of this function. For training, it samples each column proportionately to the number of distinct values within it and applies predicates on up to k columns at a time; k is tunable to ensure that the amount of column-tuples is not too large for efficient random generation.

[YLK+19] present Naru, a technique to formulate join ordering and general-purpose selectivity estimation as a reinforcement learning problem. It uses a Monte Carlo integration technique called progressive sampling on top of deep autoregressive models to estimate range queries (including numeric and categorical attributes) at high dimensionality. By leveraging the availability of conditional probability distributions provided by the model, progressive sampling steers the sampler into regions of high probability density, and then corrects for the induced bias by using importance weighting. The estimator works without supervision (like classical synopses), but it approximates the joint data distribution without any independence assumptions. Autoregressiveness is achieved via information masking and training is done via maximum likelihood estimation. The loss function between the data distribution and the loss estimate is fed into a gradient descent optimizer (Adam). Encoding values is a lossless transformation (i.e., a bijection). Values in table columns are dictionary-encoded into integers; for numerics or strings, the domain is sorted so that the dictionary order follows the column order. One-hot encoding is used for small-domain columns and embedding encoding for large-domain columns. Although Naru does not consider incremental model update, for periodic updates (e.g., daily partitions) it is possible to consider that each partition trains its own Naru model.

[OBGK18] presents an experimental study of deep learning techniques for cardinality estimation, evaluating the trade-offs between the size of the model (measured by the number of trainable parameters), the time it takes to train the model, and the accuracy of the predictions. The study compares neural networks, recurrent neural networks, tree ensembles, and PostgreSQL's optimizer.

In [DWZ+19], the authors explore application of neural networks and tree-based ensembles to improve selectivity estimation of multi-dimensional range predicates. The paper describes that straightforward application of these techniques could be worse than simple baselines, but they can be improved by simple design choices, such as regression label transformation (i.e., log-transform with base 2 of selectivity values) and feature engineering motivated by the selectivity estimation context e.g., using heuristic estimators like AVI, EBO, and MinSel (i.e., the minimum selectivity across individual predicates) as features. A query (q) in a labeled set (S) is represented as a vector containing the query predicate value ranges and the actual selectivity value ($\text{act}(q)$), which serves as a regression label. For example, $S = \{(q1:\text{act}(q1), q2:\text{act}(q2), \dots)\}$ and an instance might be $S = \{((10 \leq A1 \leq 20) \wedge (0 \leq A2 \leq 20):500), ((10 \leq A1 \leq 20) \wedge (40 \leq A2 \leq 80):300), \dots\}$, in which case the input features would be (10,20,0,100) and (12,20,40,80) with regression labels 500 and 300, respectively.

The regression model takes as input a point location in the query space defined over the domain of range features, and the task for the regression method is to learn a function over the query space to approximate multi-dimensional range selectivity estimation. To evaluate the accuracy of the model the q -error is used, and in particular the geometric mean of q -error values, which is more resilient to outlier errors compared to the arithmetic mean. The regression techniques evaluated include: (a) neural networks with ReLU units as the activation functions for hidden neurons and a linear function of input features as the regression function; and (b) tree-based ensembles, where leaf

	Project supported by the European Commission Contract no. 825070	WP5 T5.1 & T5.2 Deliverable D5.1	Doc.nr.:	WP5 D5.1
			Rev.:	1.0
			Date:	30/04/2020
			Class.:	Public

nodes correspond to a query space region defined by the conjunction of the ranges on input features from the internal nodes along the path from root; hence, each learned tree partitions the query space using hyper-rectangular regions corresponding to its leaf nodes. Updates resulting into a significant change in data distribution pose a significant limitation as the models trained on actual selectivities (which is the most time-consuming step of the process) become inaccurate quickly.

[PaZM18] describes a selectivity learning framework that uses a mixture model to capture the unknown distribution of the data. A mixture model is a probabilistic model to approximate an arbitrary probability density. Although a promising approach, it has significant practical limitations as for example it does not support join selectivity estimates.

[MaBC07] describes a semi-automatic alternative for explicit machine learning where the feature space is partitioned using decision trees and for each split a different regression model was learned.

In [KAB+19], the authors describe their experience to make a traditional cost model [SAC+79] differentiable; i.e., start with the original model, but then improve the model after every query to customize it for a particular instance of data. That would solve the initialization model and answer the question of how the model would handle ad hoc queries. Although this effort shows promising results, it also shows that it cannot achieve practical gains without having significant improvements in cardinality estimation.

CherryPick [ALC+17] aims at building accurate performance models. The goal is to find the optimal (or near-optimal) deployment configuration, in terms of number and types of VM instances, that allows achieving a given performance target. CherryPick is flexible and incurs low overhead, as it relaxes the need to find the best cloud configuration for the application. Instead, it applies Bayesian optimization to only a few samples of deployment configurations in order to obtain a performance prediction that suffices to quickly prune out inefficient setups.

9.4 Runtime Prediction


Previous work has shown that post-processing an optimizer's cost estimate is not generally effective for predicting query runtime. [ACR+12] shows that using linear regression to map PostgreSQL's estimate to actual execution time is not effective. Similar disappointing results are shown in [GKD+09], where the authors tried to map HP Neoview's estimates to actual runtime using linear regression.

[WCZ+13] argues that the optimizer estimates can be useful if the optimizer's internal cost model is tuned just before making an estimate. The paper describes an approach based on sampling. Sampling-based approaches can be very expensive when searching for a good execution plan. But in this paper, the authors start from a ready-to-be-executed plan and use sampling to correct the potentially erroneous cardinality estimates. Still, getting random samples requires significant random reads at the sampling phase, which can be very expensive in practice. Hence, the sampling is performed offline and the results are stored as materialized views in the database. Their experiments show that the number of samples required is quite small, and thus can be cached in memory at runtime. Two of the arguments they present to favor their approach vs. machine learning based approaches is that (a) the profiling stage finishes in significantly less time than a typical training phase and also that (b) once the cost units are being computed through calibration (which is orthogonal to data distribution) they do not have to be recomputed as long as the hardware does not change, whereas in machine learning approaches usually when the data distribution changes significantly the training data collection and prediction model creation need to be repeated.

There are several attempts to predicting query execution time using various machine learning techniques, which treat a data management system as a black box and aim at building efficient query runtime prediction models, e.g., [ACR+12], [GKD+09], [MAH+18], [ToBA10], [XCZT11].

9.5 Cost Modeling in Multi-Engine Systems

There is related work in the context of heterogeneous databases, but their focus is limited to a single problem. For example, [DuKS92] focuses on cost models for particular operators like selections and 2-way joins. Other works focus on a subset of cost units dedicated to a particular subsystem. For example, [SMA+10] focuses on CPU in the context of DB resource virtualization and [ZTPH11] focuses on I/O in the context of storage type selection.

 Project supported by the European Commission Contract no. 825070	WP5 T5.1 & T5.2 Deliverable D5.1	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

In the last decade or so, several approaches to multi-engine execution have been proposed. These depart from previous work on federated databases that provide a single interface to disparate DBMSs and on parallel databases that achieve high performance through replication, partitioning, and scale-out architecture. The main reason is the need for processing many types of data (not only relational) and support many types of analytics, which often is difficult to perform with a single programming or data model.

An example work is the BigDAWG architecture, a polystore database, which supports multiple database and storage engines (e.g., PostgreSQL, Accumulo, and SciDB) [GCD+16]. In the core of BigDAWG, the cross-engine query execution has been described in the context of shuffle joins (equijoin operations), where data is migrated to multiple different engines and computed in parallel [GuGS16]. The key idea to achieve parallelizable join execution of large tables across multiple engines is for the matching tuples of the two inputs to always be hosted on the same engine; this allows for parallel tuple comparison where each engine joins local data. When the planner determines a query execution plan, it passes it to the executor, which runs it in parallel in a blocking fashion as it waits for all dependency nodes to complete.

The optimization process assigns join-units (small non-overlapping ranges of tuples) to participating engines. A join-unit consists of a fraction of the full query predicate, and tuples are assigned to a join-unit based on the value of their join attribute. Tuples belonging to a single join-unit may be distributed over multiple engines participating in the join but are brought to the same engine for the join computation.

To dealing with skew, the executor collects information about the data distribution on each engine using a histogram that encodes the distribution of the join attribute for each table. The histogram is either created or extracted from the internal statistics utilized by each engine. The histogram creation is done either with (a) full table scan, which scans every element involved in the join, for every participating table (accurate histogram, slow performance), or (b) table sampling, which samples at random a fixed number or proportion of tuples from each table. If sampling statistics are provided by an engine's built-in query planner, table sampling is skipped.


Strategies for join-unit assignments are determined based on a number of factors including (a) the cost of migrating tuples between engines, the network can be a scarce resource for joins in a shared-nothing architecture [MeDe97]; (b) the number of tuple migrations, as the join computation at an engine is blocked until all tuples assigned to that engine have been migrated; and (c) the runtime of the tuple-comparison phase, which is dominated by the engine that takes the longer to complete this phase. Potential strategies include: (a) full table assignment (the entire table is a join-unit) with full broadcast (works best for smaller and less skewed tables); (b) join-attribute assignment, with a skew agnostic hash-assignment and skew-aware minimum bandwidth heuristic and tabu search (both adopted from SciDB).

The cost estimation primarily includes the cost of migrating join units to a given engine and the cost of comparing the migrated tuples once the migrations have completed ($\text{cost} = \text{maxMigrationCost} + \text{tupleComparisonCost}$). The cost of the overall plan is equivalent to the maximum cost of all engines, as every engine should complete its local executions before the results can be propagated to a union operator and returned to the user.

A previous work on multi-engine execution presented HFMS, a multi-engine system focused on information flows consisting of programs written in different programming languages (e.g., SQL, Pig, Hive) and involving data stored in multiple storage engines [SWDH13]. The HFMS plan writer inputs a flow definition, checks its validity, translates it into an internal, machine-interpretable form, and passes it to the optimizer. The optimizer generates a functionally equivalent flow graph optimized for user-specified objectives (e.g., partitioning the flow into subgraphs for different execution engines). It considers alternative execution plans and estimates their cost. The optimal plan (i.e., the least-cost relative to the objectives) is sent to the executor. The executor schedules flows for execution, generates executable code, dispatches flow fragments to execution engines, and monitors flow execution.

HFMS has a hybrid rule-based and cost-based optimization model. The core elements for the cost model are: (a) operator cost, and (b) statistical summary of operator's output. Both depend on statistical summaries of the data.

Each flow operator has associated cost models, a different model per implementation and engine. The operator cost involves measures like CPU, memory (e.g., buffer sizes), I/O, and communication costs. Unless the operator cost is known (not typical), it is computed through a series of micro-benchmarks [SiWi12]. This provides a baseline for

	Project supported by the European Commission Contract no. 825070	WP5 T5.1 & T5.2 Deliverable D5.1	Doc.nr.:	WP5 D5.1
			Rev.:	1.0
			Date:	30/04/2020
			Class.:	Public

operator cost. More elaborate computations (e.g., including concurrent execution of multiple flows) employ prediction models using machine learning techniques over execution logs [GuMD08]. The operator cost models can be refined by running flows with a sample of the input data (e.g., with Reservoir sampling) and feed the optimizer with the statistics obtained.

The complexity of dealing with multiple streaming platforms has been studied in the literature. [Beg+19] describes a standardization initiative aiming for a unified interface for SPEs and other data management systems, which integrates streaming into the SQL standard, including time-varying relations as a foundation for classical tables and streaming data, event time semantics, and keyword extensions to control the materialization of time-varying query results. Still, the SQL extensions it proposes are not rich enough to capture the semantics and the functionality of the complex Bid Data analytics operators involved in INFOR workflows.

9.6 Optimization Aspects over High Performance Computing Infrastructure

Besides sheer scale, HPC technologies can increase the cost of extreme-scale analytics which means that they should be judiciously employed and configured. There is a growing number of research proposals that exploit HPC hardware in data science platforms. HERD [KKG14] is a key-value system focused on reducing network round trips while using efficient RDMA primitives. Mega-KV [ZWY+15] shows how to use GPUs to accelerate the operations of in-memory key-value stores.


Facebook researchers have shown that they can reduce the training time for a convolutional neural network (RESNET-50 on ImageNet) from two weeks to one hour, using 256 GPUs spread over 32 servers [GDG+17]. In software, they introduced a technique to train convolutional neural networks with very large mini-batch sizes to make the learning rate proportional to the mini-batch size. This means anyone can now scale out distributed training to 100s of GPUs using TensorFlow.

Some efforts have focused on enabling the use of GPU on typical Big Data computing frameworks, such as MapReduce. PMGMR [JCQ+15] is a C++ implementation of the MapReduce programming model that runs on GPUs, providing speed-ups of up to two orders of magnitude when compared with CPU execution. Heter-oSpark [LLZ+15] exposes GPUs to Spark jobs through Java RMI, enabling users to offload computation to GPUs. Spark-GPU [YSH+16] is another solution that enables the execution of applications in both CPU and GPU, while extending/improving Spark to avoid overheads with memory copies and scheduling, providing speed-ups of an order of magnitude on machine learning applications.

All the above approaches do not account for cross-platform and stream processing optimizations. A number of works have taken some preliminary steps towards optimizing data stream processing on HPC infrastructure [ZZW+19].

[VSS11] presents an algorithm for processing data streams with real-time stream scheduling constraints on GPUs. This algorithm assigns data streams to CPUs and GPUs based on their incoming rates. It tries to provide an assignment that can satisfy different requirements from various data streams. [ZHH15] develops a holistic approach to building stream management systems using GPUs. They design a latency-driven GPU-based framework, which mainly focuses on real-time stream processing. Due to the limited memory capacity of GPUs, the work concludes that the window size of the stream operator plays an important role in system performance. To tackle this issue, [PBS15] studies the influence of window size and proposes a partitioning method for splitting large windows into different batches, considering both time and space efficiency. SABER [KWF+16] is a window-based hybrid stream processing framework aiming to utilize CPUs and GPUs concurrently.

Multi-GPU systems provide tremendous computation capacity, but also pose challenges like how to partition or schedule workloads among GPUs. [VSS12] extend their method [VSS11] to a single node with multiple GPUs. GStream [ZM11] is a data streaming framework for GPU clusters. GStream supports stream processing applications in the form of a C++ library; it uses MPI to implement the data communication between different nodes and uses CUDA to conduct stream operations on GPUs. [NL16] develops a GPU performance model for stream workload partitioning in multi-GPU platforms. [CXT+15] proposes G-Storm, which enables Apache Storm to utilize GPUs.

 Project supported by the European Commission Contract no. 825070	WP5 T5.1 & T5.2 Deliverable D5.1	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public


9.7 Geo-distributed CEP Optimization

The role of works attempting to optimize geo-distributed CEP is to assign the evaluation of a physical CEP operator, in-network, i.e., to a cluster that is closer to the event producing sources, so as to reduce communication simultaneously harnessing the network latency. Few of them also employ the push-pull rationale we discuss to further limit the amount of communicated data or account for uncertainty in event occurrence.

There is a number of approaches on geo-distributed CEP including SIENA [CRW01], Gryphon [ASS+99], Hermes [PSB03], PADRES [LG05] and the more recent works of Cordies [KKR10], DHCEP [SKR11] and FAIDECS [WEJ14]. SIENA and Gryphon do not consider in-network aggregation of events, but have focused on the efficient routing of simple events by reducing the communication costs between clients and brokers in pub/sub systems, thus avoiding the flooding of events to all subscribers. Hermes (a.k.a DistCED) uses a Distributed Hash Table (DHT) to determine in-network, CEP operator placement for execution, while FAIDECS employs Hermes's DHT for the same reason. However, as discussed in [PLS+06] DHT tables minimize the hop count as opposed to network latency or communication cost. Thus, DHT routing paths lead to inefficient CEP operator physical execution plans. Although PADRES and Cordies opt for optimizations involving network traffic and routing delay, they neither take into account any network-, latency- or system-specific information, nor provide any specific algorithmic suite for operator placement. DHCEP, uses network usage in its optimization process. Network usage is defined as the sum of products of dataRate \times latency on communication links. However, using such a blended metric, instead of seeking for Pareto optimal solutions as we do, does not allow for latency-constrained optimization and communication cost minimization, separately. All constraints examined in DHCEP involve processing, security or domain restrictions, but not bandwidth consumption or latency.

The FERARI framework [FMG+16] enables CEP over multiple clusters or clouds. Geo-distributed CEP is optimized by a query optimizer that receives as input the CEP query composed of a number of CEP operators and determines the physical execution of CEP operators assigned to networked clusters, patched with a proper push-pull strategy. To achieve that, FERARI enhances the algorithms of [ACT08] with in-network placement. It seeks for overall Pareto optimal solutions in terms of both in-network processing and push-pull strategy not only per operator, but also among operators shared by multiple queries. However, it does not account for other important optimization objectives such as resource consumption minimization or throughput maximization and works only with one Big Data platform, i.e. Apache Storm.

The work in [WCZ13] is reported as the only one that considers uncertain CEP over distributed settings. However, contrary to the techniques we present, it does not impose in-situ filters in order to avoid communication. Instead, the proposed technique lets every site compute probabilities of full or partial pattern matches locally per site and then accumulates these results to a central site to compute the final CEs. These final CEs are then forwarded to the CEP query source. The techniques we develop significantly differ because, by employing in-situ filters, we totally suppress communication among sites in case these local filters indicate that a CE cannot have occurred even upon synthesizing data from other sites.

	Project supported by the European Commission Contract no. 825070	WP5 T5.1 & T5.2 Deliverable D5.1	Doc.nr.:	WP5 D5.1
			Rev.:	1.0
			Date:	30/04/2020
			Class.:	Public

10 Conclusions and Future Work


In this deliverable we describe the internal architectural design of the Optimizer Component of INFOR and the algorithmic arsenal it has been equipped with, to boost timely analytic results at scale. The goal of the optimizer is to receive the initial workflow drawn by the user, i.e., a logical workflow, and optimize its execution over multiple, networked clusters, Big Data platforms and admissible parallelization strategies, prescribing a preferred physical plan.

Our optimization problem examines conflicting constraints related to throughput maximization, latency, communication cost, memory, CPU usage minimization under global (for the entire workflow) or local (e.g., memory, CPU per platform) constraints. Since we cannot improve one objective without deteriorating another, we resort to finding Pareto optimal solutions to our optimization problem. We propose optimal (Exhaustive Search, A*-like, Dynamic Programming-like) algorithms and fast Heuristic and Greedy ones. These algorithms trade-off reduced execution time for proximity of the prescribed physical execution plan to the optimal one. Our algorithms utilize Bayesian Optimization to deal with the complexity and heterogeneity of INFOR data processing operators in predicting the performance of a physical plan.

We extend our algorithms to exploit the SDE Component of INFOR and we also devise algorithms for fully decentralized networked settings incorporating, apart from clusters, sensor devices of certain hardware capabilities.


Our experimental evaluation shows that our cost estimator can accurately predict the performance of important workflow operators utilizing a limited number of microbenchmarks for each. Moreover, the A*-like algorithm we propose can drastically reduce the number of alternative physical execution flows that are examined, compared to the exhaustive search approach, in its effort to prescribe a proper physical execution plan (workflow). We further experimentally observe the ability of our synopsis-based optimization approach in yielding physical workflows whose performance enable interactive, advanced analytics to the major extent.

Our future work concentrates on optimization algorithms that instead of trying to predict the performance of a physical plan, quickly compute and start with one such plan and then attempt to detect bottlenecks and stragglers so as to improve it at runtime. This new approach together with enhanced versions of the algorithms presented in this deliverable will have to rely on robust and fault tolerant adaptation mechanisms across multiple clusters and Big Data platforms to ensure or preserve, respectively, the desirable performance of physical workflows in the long run. This introduces additional objectives to our optimization goals, such as migration cost minimization upon switching between execution plans, which calls for new, robust cost estimations. Besides, our cost estimator should be extended to not only provide the primitive so that the developed algorithms pick a good execution plan for a physical workflow, but also allow them to predict the ability of the devised plan to maintain high performance as time passes and stream characteristics are altered. All these issues, together with fine-tuned optimization over heterogeneous infrastructures composed of both GPUs and CPUs, open new horizons both for research and applied solutions to get materialized in INFOR's prototype in Month 32 of the project.


 Project supported by the European Commission Contract no. 825070	WP5 T5.1 & T5.2 Deliverable D5.1	Doc.nr.:	WP5 D5.1
		Rev.:	1.0
		Date:	30/04/2020
		Class.:	Public

11 References


- [ACR+12] Mert Akdere, Ugur Çetintemel, Matteo Riondato, Eli Upfal, Stanley B. Zdonik: Learning-based Query Performance Modeling and Prediction. ICDE 2012: 390-401
- [ACT08] Mert Akdere, Ugur Cetintemel, and Nasime Tatbul. Plan-based complex event detection across distributed sources. PVLDB, 1(1):66–77, 2008.
- [AIAP17] Elias Alevizos, Alexander Artikis, George Paliouras: Event Forecasting with Pattern Markov Chains. DEBS 2017: 146-157
- [AIAP18] Elias Alevizos, Alexander Artikis, Georgios Paliouras: Wayeb: a Tool for Complex Event Forecasting. LPAR 2018: 26-35
- [ACH+13] Pankaj K. Agarwal, Graham Cormode, Zengfeng Huang, Jeff M. Phillips, Zhewei Wei, Ke Yi: Mergeable summaries. ACM Trans. Database Syst. 38(4): 26:1-26:28 (2013)
- [ALC+17] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, Ming Zhang: CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. NSDI 2017: 469-482
- [AMR+12] M. R. Ackermann, M. Mörtens, C. Raupach, K. Swierkot, C. Lammersen, and C. Sohler. 2012. StreamKM++: A clustering algorithm for data streams. ACM Journal of Experimental Algorithmics 17, 1 (2012).
- [ASS+99] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In PODC, pages 53–61, 1999.
- [AvHe00] Ron Avnur, Joseph M. Hellerstein: Eddies: Continuously Adaptive Query Processing. SIGMOD Conference 2000: 261-272
- [Bah+12] Bahman Bahmani, Benjamin Moseley, Andrea Vattani, Ravi Kumar, Sergei Vassilvitskii: Scalable K-Means++. PVLDB 5(7): 622-633 (2012)
- [Beg+19] Edmon Begoli, Tyler Akidau, Fabian Hueske, Julian Hyde, Kathryn Knight, Kenneth Knowles: One SQL to Rule Them All - an Efficient and Syntactically Idiomatic Approach to Management of Streams and Tables. SIGMOD Conference 2019: 1757-1772
- [BLC+15] Debabrota Basu, Qian Lin, Weidong Chen, Hoang Tam Vo, Zihong Yuan, Pierre Senellart, Stéphane Bressan: Cost-Model Oblivious Database Tuning with Reinforcement Learning. DEXA (1) 2015: 253-268
- [BrCF101] Eric Brochu, Vlad M. Cora, Nando de Freitas: A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modelling and Hierarchical Reinforcement Learning. CoRR abs/1012.2599 (2010)
- [BrCG01] Nicolas Bruno, Surajit Chaudhuri, Luis Gravano: STHoles: A Multidimensional Workload-Aware Histogram. SIGMOD Conference 2001: 211-222
- [CaBS19] Walter Cai, Magdalena Balazinska, Dan Suciu: Pessimistic Cardinality Estimation: Tighter Upper Bounds for Intermediate Join Cardinalities. SIGMOD Conference 2019: 18-35
- [Cra+06] Koby Crammer, Ofer Dekel, Joseph Keshet, Shai Shalev-Shwartz, Yoram Singer: Online Passive-Aggressive Algorithms. J. Mach. Learn. Res. 7: 551-585 (2006)
- [CRW01] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. ACM Trans. Comput. Syst., 19(3):332–383, 2001.
- [CXT+15] Zhenhua Chen, Jielong Xu, Jian Tang, Kevin A. Kwiat, Charles A. Kamhoua: G-Storm: GPU-enabled high-throughput online data processing in Storm. BigData 2015: 307-312
- [DeGR01] Amol Deshpande, Minos N. Garofalakis, Rajeev Rastogi: Independence is Good: Dependency-Based Histogram Synopses for High-Dimensional Data. SIGMOD Conference 2001: 199-210
- [DeIR07] Amol Deshpande, Zachary G. Ives, Vijayshankar Raman: Adaptive Query Processing. Foundations and Trends in Databases 1(1): 1-140 (2007)
- [DuKS92] Weimin Du, Ravi Krishnamurthy, Ming-Chien Shan: Query Optimization in a Heterogeneous DBMS. VLDB 1992: 277-291
- [DWZ+19] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek R. Narasayya, Surajit Chaudhuri: Selectivity Estimation for Range Predicates using Lightweight Models. PVLDB 12(9): 1044-1057 (2019)
- [DynaTrace] Dynatrace. Deep dive application monitoring and performance management. <https://www.dynatrace.com/platform/deep-dive-application-monitoring/>.
- [EsNa06] Cristian Estan, Jeffrey F. Naughton: End-biased Samples for Join Cardinality Estimation. ICDE 2006: 20

	Project supported by the European Commission Contract no. 825070	WP5 T5.1 & T5.2 Deliverable D5.1	Doc.nr.:	WP5 D5.1
			Rev.:	1.0
			Date:	30/04/2020
			Class.:	Public


- [FGD+17] Ioannis Flouris, Nikos Giatrakos, Antonios Deligiannakis, Minos N. Garofalakis, Michael Kamp, Michael Mock: Issues in complex event processing: Status and prospects in the Big Data era. *J. Syst. Softw.* 127: 217-236 (2017)
- [FGD+20] Ioannis Flouris, Nikos Giatrakos, Antonios Deligiannakis, Minos N. Garofalakis: Network-wide complex event processing over geographically distributed data sources. *Inf. Syst.* 88 (2020)
- [FMG+16] I. Flouris, V. Manikaki, N. Giatrakos, A. Deligiannakis, M. N. Garofalakis, M. Mock, S. Bothe, I. Skarbovsky, F. Fournier, M. Stajcer, T. Krizan, J. Yom-Tov, and T. Curin. FERARI: A prototype for complex event processing over streaming multicloud platforms. In *SIGMOD*, pages 2093–2096, 2016.
- [GAA+20] Nikos Giatrakos, Elias Alevizos, Alexander Artikis, Antonios Deligiannakis, Minos N. Garofalakis: Complex event recognition in the Big Data era: a survey. *VLDB J.* 29(1): 313-352 (2020)
- [GAD+19] Nikos Giatrakos, Alexander Artikis, Antonios Deligiannakis, Minos N. Garofalakis: Uncertainty-Aware Event Analytics over Distributed Settings. *DEBS 2019*: 175-186
- [GCD+16] Vijay Gadepally, Peinan Chen, Jennie Duggan, Aaron J. Elmore, Brandon Haynes, Jeremy Kepner, Samuel Madden, Tim Mattson, Michael Stonebraker: The BigDAWG polystore system and architecture. *HPEC 2016*: 1-6
- [GDG+17] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, Kaiming He: Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *CoRR abs/1706.02677* (2017)
- [GeTK01] Lise Getoor, Benjamin Taskar, Daphne Koller: Selectivity Estimation using Probabilistic Models. *SIGMOD Conference 2001*: 461-472
- [GKD+09] Archana Ganapathi, Harumi A. Kuno, Umeshwar Dayal, Janet L. Wiener, Armando Fox, Michael I. Jordan, David A. Patterson: Predicting Multiple Metrics for Queries: Better Decisions Enabled by Machine Learning. *ICDE 2009*: 592-603
- [GKTD05] Dimitrios Gunopulos, George Kollios, Vassilis J. Tsotras, Carlotta Domeniconi: Selectivity estimators for multidimensional range queries over real attributes. *VLDB J.* 14(2): 137-154 (2005)
- [GP15] Pedro Guimarães and José Pereira. X-ray: Monitoring and analysis of distributed database queries. In *IFIP International Conference on Distributed Applications and Interoperable Systems*, pages 80–93. Springer, 2015.
- [Grae95] Goetz Graefe: The Cascades Framework for Query Optimization. *IEEE Data Eng. Bull.* 18(3): 19-29 (1995)
- [Grafana] Grafana. The open observability platform. <https://grafana.com/>
- [GrMc93] Goetz Graefe, William J. McKenna: The Volcano Optimizer Generator: Extensibility and Efficient Search. *ICDE 1993*: 209-218
- [GuGS16] Ankush M. Gupta, Vijay Gadepally, Michael Stonebraker: Cross-engine query execution in federated database systems. *HPEC 2016*: 1-6
- [GuMD08] Chetan Gupta, Abhay Mehta, Umeshwar Dayal: PQR: Predicting Query Execution Times for Autonomous Workload Management. *ICAC 2008*: 13-22
- [HeKM15] Max Heimerl, Martin Kiefer, Volker Markl: Self-Tuning, GPU-Accelerated Kernel Density Models for Multidimensional Selectivity Estimation. *SIGMOD Conference 2015*: 1477-1492
- [HFLP89] Laura M. Haas, Johann Christoph Freytag, Guy M. Lohman, Hamid Pirahesh: Extensible Query Processing in Starburst. *SIGMOD Conference 1989*: 377-388
- [HNR68] Hart, P. E.; Nilsson, N. J.; Raphael, B. (1968). "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". *IEEE Transactions on Systems Science and Cybernetics*. 4 (2): 100–107. doi:10.1109/TSSC.1968.300136
- [HNSS96] Peter J. Haas, Jeffrey F. Naughton, S. Seshadri, Arun N. Swami: Selectivity and Cost Estimation for Joins Based on Random Sampling. *J. Comput. Syst. Sci.* 52(3): 550-569 (1996)
- [HoKe88] A.E. Horel, R.W. Kennard: Ridge regression. *Encyclopaedia of statistical sciences* (1988)
- [HSGS19] Martin Hirzel, Robert Soulé, Bugra Gedik, Scott Schneider: Stream Query Optimization. *Encyclopedia of Big Data Technologies 2019*
- [HVP+18] Todd Hester, Matej Vecerík, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Dan Horgan, John Quan, Andrew Sendonaris, Ian Osband, Gabriel Dulac-Arnold, John Agapiou, Joel Z. Leibo, Audrunas Gruslys: Deep Q-learning From Demonstrations. *AAAI 2018*: 3223-3230
- [JCQ+15] Hai Jiang, Yi Chen, Zhi Qiao, Tien-Hsiung Weng, Kuan-Ching Li: Scaling up MapReduce-based Big Data Processing on Multi-GPU systems. *Cluster Computing* 18(1): 369-383 (2015)

	Project supported by the European Commission Contract no. 825070	WP5 T5.1 & T5.2 Deliverable D5.1	Doc.nr.:	WP5 D5.1
			Rev.:	1.0
			Date:	30/04/2020
			Class.:	Public

- [JoSW14] Petar Jovanovic, Alkis Simitsis, Kevin Wilkinson: Engine independence for logical analytic flows. ICDE 2014: 1060-1071
- [KAB+19] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H. Chi, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, Vikram Nathan: SageDB: A Learned Database System. CIDR 2019
- [KBC+18] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, Neoklis Polyzotis: The Case for Learned Index Structures. SIGMOD Conference 2018: 489-504
- [KFMM16] Nicolas Kourtellis, Gianmarco De Francisci Morales, Albert Bifet, Arinto Murdopo: VHT: Vertical hoeffding tree. BigData 2016: 915-922
- [KiBa15] Diederik P. Kingma, Jimmy Ba: Adam: A Method for Stochastic Optimization. ICLR 2015
- [KKG14] Anuj Kalia, Michael Kaminsky, David G. Andersen: Using RDMA efficiently for key-value services. SIGCOMM 2014: 295-306
- [KKR10] G. G. Koch, B. Koldehofe, and K. Rothermel. Cordies: Expressive event correlation in distributed systems. In DEBS, 2010.
- [KKR+19] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, Alfons Kemper: Learned Cardinalities: Estimating Correlated Joins with Deep Learning. CIDR 2019
- [KoGD20] Antonis Kontaxakis, Nikos Giatrakos, Antonios Deligiannakis: A Synopses Data Engine for Interactive Extreme-Scale Analytics. arXiv:2003.09541v1 (2020)
- [KPR+16] James Kirkpatrick, Razvan Pascanu, Neil C. Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, Raia Hadsell: Overcoming catastrophic forgetting in neural networks. CoRR abs/1612.00796 (2016)
- [KWF+16] Alexandros Koliousis, Matthias Weidlich, Raul Castro Fernandez, Alexander L. Wolf, Paolo Costa, Peter R. Pietzuch: SABER: Window-Based Hybrid Stream Processing for Heterogeneous Architectures. SIGMOD Conference 2016: 555-569
- [KYG+18] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph M. Hellerstein, Ion Stoica: Learning to Optimize Join Queries With Deep Reinforcement Learning. CoRR abs/1808.03196 (2018)
- [LaZh98] M. Seetha Lakshmi, Shaoyu Zhou: Selectivity Estimation in Extensible Databases - A Neural Network Approach. VLDB 1998: 623-627
- [LG05] G. Li and H. Jacobsen. Composite subscriptions in content-based publish/subscribe systems. In Middleware, 2005.
- [LLZ+15] Peilong Li, Yan Luo, Ning Zhang, Yu Cao: HeteroSpark: A heterogeneous CPU/GPU Spark platform for machine learning algorithms. NAS 2015: 347-348
- [LRG+17] Viktor Leis, Bernhard Radke, Andrey Gubichev, Alfons Kemper, Thomas Neumann: Cardinality Estimation Done Right: Index-Based Join Sampling. CIDR 2017
- [LRG+18] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, Thomas Neumann: Query optimization through the looking glass, and what we found running the Join Order Benchmark. VLDB J. 27(5): 643-668 (2018)
- [LXY+15] Henry Liu, Mingbin Xu, Ziting Yu, Vincent Corvinelli, Calisto Zuzarte: Cardinality estimation using neural networks. CASCON 2015: 53-59
- [MaBC07] Tanu Malik, Randal C. Burns, Nitesh V. Chawla: A Black-Box Approach to Query Cardinality Estimation. CIDR 2007: 56-67
- [MCC04] Matthew L Massie, Brent N Chun, and David E Culler. The ganglia distributed monitoring system: design, implementation, and experience. Parallel Computing, 30(7):817-840, 2004.
- [MAH+18] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, Geoffrey J. Gordon: Query-based Workload Forecasting for Self-Driving Database Management Systems. SIGMOD Conference 2018: 631-645
- [MaLR03] Volker Markl, Guy M. Lohman, Vijayshankar Raman: LEO: An autonomic query optimizer for DB2. IBM Systems Journal 42(1): 98-106 (2003)
- [MaPa18] Ryan Marcus, Olga Papaemmanouil: Deep Reinforcement Learning for Join Order Enumeration. aiDM@SIGMOD 2018: 3:1-3:4
- [MCCD13] Tomas Mikolov, Kai Chen, Greg Corrado, Jeffrey Dean: Efficient Estimation of Word Representations in Vector Space. ICLR (Workshop Poster) 2013
- [MeDe97] Manish Mehta, David J. DeWitt: Data Placement in Shared-Nothing Parallel Database Systems. VLDB J. 6(1): 53-72 (1997)
- [MHK+07] Volker Markl, Peter J. Haas, Marcel Kutsch, Nimrod Megiddo, Utkarsh Srivastava, Tam Minh Tran: Consistent selectivity estimation via maximum entropy. VLDB J. 16(1): 55-76 (2007)

	Project supported by the European Commission Contract no. 825070	WP5 T5.1 & T5.2 Deliverable D5.1	Doc.nr.:	WP5 D5.1
			Rev.:	1.0
			Date:	30/04/2020
			Class.:	Public

- [MNM+19] Ryan C. Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, Nesime Tatbul: Neo: A Learned Query Optimizer. PVLDB 12(11): 1705-1718 (2019)
- [MoNS09] Guido Moerkotte, Thomas Neumann, Gabriele Steidl: Preventing Bad Plans by Bounding the Impact of Cardinality Estimation Errors. PVLDB 2(1): 982-993 (2009)
- [MuDe88] M. Muralikrishna, David J. DeWitt: Equi-Depth Histograms For Estimating Selectivity Factors For Multi-Dimensional Queries. SIGMOD Conference 1988: 28-36
- [MuMK18] Magnus Müller, Guido Moerkotte, Oliver Kolb: Improved Selectivity Estimation by Combining Knowledge from Sampling and Synopses. PVLDB 11(9): 1016-1028 (2018)
- [Naggios] Nagios. The industry standard in it infrastructure monitoring. <https://www.nagios.org/>
- [NDJ13] Supriya Nirkhiwale, Alin Dobra, Christopher M. Jermaine: A Sampling Algebra for Aggregate Estimation. PVLDB 6(14): 1798-1809 (2013)
- [NL16] Dong Nguyen, Jongeun Lee: Communication-aware mapping of stream graphs for multi-GPU platforms. CGO 2016: 94-104
- [NMM+20] Parimarjan Negi, Ryan Marcus, Hongzi Mao, Nesime Tatbul, Tim Kraska, Mohammad Alizadeh. Cost-Guided Cardinality Estimation: Focus Where it Matters. SMDDB 2020.
- [NMP18] F. Neves, N. Machado, and J. Pereira. "Falcon: A Practical Log-based Analysis Tool for Distributed Systems". In the 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2018.
- [OBGK18] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, S. Sathya Keerthi: Learning State Representations for Query Optimization with Deep Reinforcement Learning. DEEM@SIGMOD 2018: 4:1-4:4
- [PaZM18] Yongjoo Park, Shucheng Zhong, Barzan Mozafari: QuickSel: Quick Selectivity Learning with Mixture Models. CoRR abs/1812.10568 (2018)
- [PBS15] Marcus Pinnecke, David Brönske, Gunter Saake: Toward GPU Accelerated Data Stream Processing. GvD 2015: 78-83
- [PoGI99] Viswanath Poosala, Venkatesh Ganti, Yannis E. Ioannidis: Approximate Query Answering using Histograms. IEEE Data Eng. Bull. 22(4): 5-14 (1999)
- [Prometheus] Prometheus. Prometheus - Monitoring system and time series database. <https://prometheus.io/>
- [PSB03] P. R. Pietzuch, B. Shand, and J. Bacon. A framework for event composition in distributed systems. In Middleware, 2003.
- [PLS+06] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream processing systems. In ICDE, 2006.
- [ReSu12] Christopher Ré, Dan Suciu: Understanding cardinality estimation using entropy maximization. ACM Trans. Database Syst. 37(1): 6:1-6:31 (2012)
- [SAC+79] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, Thomas G. Price: Access Path Selection in a Relational Database Management System. SIGMOD Conference 1979: 23-34
- [ShDG17] Michael Shekelyan, Anton Dignös, Johann Gamper: DigitHist: a Histogram-Based Data Summary with Tight Error Bounds. PVLDB 10(11): 1514-1525 (2017)
- [SiWi12] Alkis Simitsis, Kevin Wilkinson: Revisiting ETL Benchmarking: The Case for Hybrid Flows. TPCTC 2012: 75-91
- [SKR11] B. Schilling, B. Koldehofe, and K. Rothermel. Efficient and distributed rule placement in heavy constraint-driven event systems. In HPCC, pages 355–364, 2011.
- [SMA+10] Ahmed A. Soror, Umar Farooq Minhas, Ashraf Aboulmaga, Kenneth Salem, Peter Kokosieli, Sunil Kamath: Automatic virtual machine configuration for database workloads. ACM Trans. Database Syst. 35(1): 7:1-7:47 (2010)
- [Sob67] I. M. Sobol: Distribution of points in a cube and approximate evaluation of integrals. U.S.S.R Comput. Maths. Math. Phys. 7: 86–112, 1967
- [SWDH13] Alkis Simitsis, Kevin Wilkinson, Umeshwar Dayal, Meichun Hsu: HFMS: Managing the lifecycle and complexity of hybrid analytic data flows. ICDE 2013: 1174-1185
- [ToBA10] Sean Tozer, Tim Brecht, Ashraf Aboulmaga: Q-Cop: Avoiding bad query mixes to minimize client timeouts under heavy loads. ICDE 2010: 397-408
- [TWM+19] Immanuel Trummer, Junxiong Wang, Deepak Maram, Samuel Moseley, Saehan Jo, Joseph Antonakakis: SkinnerDB: Regret-Bounded Query Evaluation via Reinforcement Learning. SIGMOD Conference 2019: 1153-1170

	Project supported by the European Commission Contract no. 825070	WP5 T5.1 & T5.2 Deliverable D5.1	Doc.nr.:	WP5 D5.1
			Rev.:	1.0
			Date:	30/04/2020
			Class.:	Public

- [TzDJ11] Kostas Tzoumas, Amol Deshpande, Christian S. Jensen: Lightweight Graphical Models for Selectivity Estimation Without Independence Assumptions. PVLDB 4(11): 852-863 (2011)
- [TzSJ08] Kostas Tzoumas, Timos Sellis, Christian S. Jensen: A Reinforcement Learning Approach for Adaptive Query Processing. Technical Report, June 2008
- [Vovk01] V. Vovk: Competitive on-line statistics. International Statistical Review, 69(2), pp.213-248 (2001)
- [VSS11] Uri Verner, Assaf Schuster, Mark Silberstein: Processing data streams with hard real-time constraints on heterogeneous systems. ICS 2011: 120-129
- [VSS12] Uri Verner, Assaf Schuster, Mark Silberstein, Avi Mendelson: Scheduling processing of real-time data streams on heterogeneous multi-GPU systems. SYSTOR 2012: 7
- [WCZ13] Y. H. Wang, K. Cao, and X. M. Zhang. 2013. Complex event processing over distributed probabilistic event streams. Computers and Mathematics with Applications 66, 10 (2013), 1808–1821.
- [WCZ+13] Wentao Wu, Yun Chi, Shenghuo Zhu, Jun'ichi Tatemura, Hakan Hacigümüs, Jeffrey F. Naughton: Predicting query execution time: Are optimizer cost models really unusable? ICDE 2013: 1081-1092
- [WEJ14] G. A. Wilkin, P. Eugster, and K. R. Jayaram. Decentralized fault-tolerant event correlation. ACM Trans. Internet Technol., 14(1):5:1–5:27, aug 2014.
- [WHT+19] Lucas Woltmann, Claudio Hartmann, Maik Thiele, Dirk Habich, Wolfgang Lehner: Cardinality estimation with local deep learning models. aiDM@SIGMOD 2019: 5:1-5:8
- [WJA+18] Chenggang Wu, Alekh Jindal, Saeed Amizadeh, Hiren Patel, Wangchao Le, Shi Qiao, Sriram Rao: Towards a Learning Optimizer for Shared Clouds. PVLDB 12(3): 210-222 (2018)
- [WuNS16] Wentao Wu, Jeffrey F. Naughton, Harneet Singh: Sampling-Based Query Re-Optimization. SIGMOD Conference 2016: 1721-1736
- [XCZT11] PengCheng Xiong, Yun Chi, Shenghuo Zhu, Jun'ichi Tatemura, Calton Pu, Hakan Hacigümüs: ActiveSLA: a profit-oriented admission control framework for database-as-a-service providers. SoCC 2011: 15
- [YLK+19] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Peter Chen, Pieter Abbeel, Joseph M. Hellerstein, Sanjay Krishnan, Ion Stoica: Deep Unsupervised Cardinality Estimation. PVLDB 13(3): 279-292 (2019)
- [YSH+16] Yuan Yuan, Meisam Fathi Salmi, Yin Huai, Kaibo Wang, Rubao Lee, Xiaodong Zhang: Spark-GPU: An accelerated in-memory data processing engine on clusters. BigData 2016: 273-283
- [Zabbix] Zabbix. The Enterprise-class Monitoring Solution for Everyone. <https://www.zabbix.com/>
- [ZHH15] Kai Zhang, Jiayu Hu, Bei Hua: A holistic approach to build real-time stream processing system with GPU. J. Parallel Distributed Comput. 83: 44-57 (2015)
- [ZhRL96] Tian Zhang, Raghu Ramakrishnan, Miron Livny: BIRCH: An Efficient Data Clustering Method for Very Large Databases. SIGMOD Conference 1996: 103-114
- [ZM11] Yongpeng Zhang, Frank Mueller: GStream: A General-Purpose Data Streaming Framework on GPU Clusters. ICPP 2011: 245-254
- [ZTPH11] Ning Zhang, Jun'ichi Tatemura, Jignesh M. Patel, Hakan Hacigümüs: Towards Cost-Effective Storage Provisioning for DBMSs. PVLDB 5(4): 274-285 (2011)
- [ZWY+15] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Lee, Xiaodong Zhang: Mega-KV: A Case for GPUs to Maximize the Throughput of In-Memory Key-Value Stores. PVLDB 8(11): 1226-1237 (2015)
- [ZZW+19] Shuhao Zhang, Feng Zhang, Yingjun Wu, Bingsheng He, Paul Johns: Hardware-Conscious Stream Processing: A Survey. SIGMOD Rec. 48(4): 18-29 (2019)