Grant Agreement No: 687591

31/12/18

Big Data Analytics for Time Critical Mobility Forecasting

datAcron

D1.13 Integrated prototype evaluation report

Deliverable Form	
Project Reference No.	H2020-ICT-2015 687591
Deliverable No.	1.13
Relevant Work Package:	WP 1
Nature:	R
Dissemination Level:	PU
Document version:	1.0
Due Date:	31/12/2018
Date of latest revision:	30/12/2018
Completion Date:	30/12/2018
Lead partner:	UPRC
Authors:	Christos Doulkeridis (UPRC), Apostolos Glenis (UPRC), Panagiotis Nikitopoulos (UPRC), Giorgos Santipantakis (UPRC), Akrivi Vlachou (UPRC), George Vouros (UPRC)
Reviewers:	Michael Mock (FRHF), Alex Artikis (NCSR'D)
Document description:	This deliverable briefly describes the datAcron integrated prototype.
Document location:	WP1/Deliverables/D1.13/Final

HISTORY OF CHANGES

Version	Date	Changes	Author	Remarks
0.1	3/12/2018	First version of report	C. Doulkeridis	
0.2	5/12/2018	Added evaluation methodology and results (Sections 3,4)	A. Glenis	
0.3	7/12/2018	Added Section 2	G. Santipantakis	
0.4	17/12/2018	Updated Sections 3,4	C. Doulkeridis	
0.5	19/12/2018	First complete version for internal review (G.Vouros)	C. Doulkeridis	
0.6	20/12/2018	Updated Sections 4,5	C. Doulkeridis, G. Santipantakis, A. Glenis, P. Nikitopoulos	
0.7	21/12/2018	Second version for internal review (G.Vouros)	C. Doulkeridis	
0.8	21/12/2018	Added discussion sections	C. Doulkeridis, A. Vlachou	
1.0	31/12/2018	Final version	C. Doulkeridis	

EXECUTIVE SUMMARY

This report is deliverable D1.13 of datAcron work package 1 “System architecture and data management” with main objective to evaluate the datAcron integrated prototype system, in accordance with the requirements specified in deliverable D1.1, the architecture specified in deliverable D1.2, and the description of the integrated prototype reported in D1.12.

TABLE OF CONTENTS

TERMS & ABBREVIATIONS

ADS-B	Automatic Dependent Surveillance-Broadcast
AIS	Automatic Identification System
RDF	Resource Description Framework
LED	Low-level Event Detector
SG	Synopsis Generator
SI	Semantic Integrator
DM	Data Manager
T/FLP	Trajectory/Future Location Predictor
CER/F	Complex Event Recognition/Forecasting
TDA	Trajectory Data Analytics
IVA	Interactive Visual Analytics
Viz	Real-time Visualization

LIST OF FIGURES

LIST OF TABLES

1 Introduction

This document is the deliverable D1.13 "Integrated prototype evaluation report" of work package 1 "System Architecture and Data Management" of the datAcron project, submitted on month M36 of the project.

D1.13 is focused on the evaluation of the integrated prototype of datAcron, as an integrated system consisting of different modules developed in the context of the technical work packages WP1–WP4. As such, its objective is to report evaluation results of the integrated system as a whole, while for the evaluation of individual modules we refer to the corresponding deliverables.

1.1 Purpose and Scope

The datAcron integrated system prototype comprises a big data architecture that encompasses both a *stream processing* part, as well as a *batch processing* part.

In the *stream processing* part, online operations are performed on streams of surveillance data (data in motion); these operations include trajectory reconstruction, trajectory compression, trajectory enrichment, future location prediction, complex event recognition and forecasting. In terms of evaluation, the primary objective related to the stream processing part of the architecture is low-latency processing, with *operational latency* (i.e., in the order of one second) and *tactical latency* (i.e., in the order of a few seconds) – depending on the specific online operation – in the respective analysis of requirements (cf. deliverable D1.1). At the same time, the integrated system should achieve high throughput and scalability, so that it can exploit hardware provisioning to support higher input rates.

In the *batch processing* part, the functionality offered can be broadly classified in two categories: querying of integrated RDF data and data analytics. Spatio-temporal RDF query processing is achieved by the distributed datAcron processing engine, which a a distributed RDF storage and processing solution developed from scratch in the datAcron project. Its primary evaluation objective is the performance and scalability of query processing, for various data set sizes and queries of increased complexity. The data analytics solutions are distributed algorithms that scale gracefully as the volume and/or velocity of input data sets increases. In turn, their primary evaluation objective is twofold: first, processing performance, and, second, quality of results (e.g., accuracy of prediction).

To achieve the afore-described objectives, the individual modules are designed and implemented using state-of-the-art frameworks for Big Data processing, most notably Apache Flink and Apache Spark. In this way, scalability and fault-tolerance is achieved at the level of each individual module. The intercommunication of the different online modules is achieved using Apache Kafka, the de-facto standard for building real-time data pipelines nowadays, offering scalability, persistence, replication, and fault-tolerance.

This report aims to provide empirical evidence of the performance evaluation of the integrated system, thus demonstrating its suitability with respect to meeting the operational requirements that have been specified by the domain experts. Therefore, an experimental evaluation of the integrated system is performed using real data sets from the two domains, in the datAcron cluster that comprises ten computing nodes.

1.2 Approach for the Work package and Relation to other Deliverables

Work package 1 "System Architecture and Data Management" is responsible for: (a) the underlying system architecture and the integrated prototype of datAcron, and (b) the data management layer of the datAcron infrastructure. In consequence, the current deliverable D1.13 is particularly important for WP1, as it concerns the evaluation of the integrated prototype directly, and indirectly the overall datAcron architecture.

D1.13 builds on several previous deliverables of WP1:

- D1.1 "Requirements Analysis" that provides a specification for the requirements of both the integrated system as well as the individual constituent modules.
- D1.2 "Architecture Specification", which is the system architecture reported on M12, and later further refined during the project.
- D1.11 "Software design (final)", which presents the software stack adopted in order to realize the various modules, as well as the underlying communication platform.
- D1.12 "Integrated prototype (final)", which is the concrete description of the integrated prototype, as reported in month M33 of the project, and practically serving as the final description of the integrated datAcron system.
- Several deliverables of technical work packages, namely: D1.9, D1.10, D1.11 (WP1), D2.3 (WP2), D3.3, D3.4 (WP3), D4.5, D4.6, D4.7, D4.8 (WP4), which present the innovative methods and evaluate the optimized versions of the individual modules in more detail.
- Deliverables of work packages WP5 and WP6 describing the two use-case domains, maritime and air-traffic management (ATM), especially the ones: D5.3, D5.4, D5.5, D6.3, D6.4, D6.5.

1.3 Methodology and Structure of the Deliverable

This deliverable takes as input D1.12 describing the final version of the integrated prototype of datAcron, and essentially performs the experimental evaluation in order to validate its performance. The work methodology starts from the level of individual modules, in order to identify the best parameterization with respect to external parameters to the module itself, i.e., cluster and infrastructure-related parameters[1]. Then, the main focus of the evaluation is the integrated prototype as a whole, and the achieved performance for different setups and configurations, most notably with respect to the hardware resources provided to each module.

The remaining of this report is structured as follows:

- Section 2 presents succinctly an overview of the integrated prototype of datAcron, focusing in flows of information and the architecture that encompasses all envisioned flows and modules.

[1] Optimization of individual module operation with respect to internal parameters has been conducted in the context of each work package and has been documented in the respective deliverables, therefore it is out of the scope of this deliverable.

- Section 3 describes in more detail the methodology followed for the evaluation of the datAcron integrated prototype, focusing on the Kafka-based intercommunication that has been adopted in datAcron.
- Section 4 demonstrates the evaluation results obtained when testing the performance of the online part of the integrated prototype.
- Section 5 shows the performance and scalability of the batch processing part of the datAcron architecture.
- Section 6 briefly summarizes the findings of the evaluation of the integrated prototype and reports the main conclusions of this deliverable.

2 Design of datAcron Integrated Prototype

The purpose of this section is to provide a succinct presentation of the datAcron integrated prototype. The following sections discuss (a) the modules that implement the functionality (Section 2.1), and (b) the flows of information materialized by means of the modules (Section 2.2).

Module Nr.	Module acronym	Module Title	Task in Charge	Work package
1)	MDF	Maritime raw data feeder	T5.2	WP5
2)	ADF	Aviation raw data feeders	T6.2	WP6
3)	LED	In-situ processing 1 – Low-level event detector	T1.3.1	WP1
4)	SG	In-situ processing 2 – Synopses generator	T2.1	WP2
5)	SI	Semantic integrator	T1.3.2	WP1
6)	DM	Data manager	T1.3.3	WP1
7)	T/FLP	Trajectory / Future location predictor	T2.2	WP2
8)	TDA	Trajectory data analytics	T2.3	WP2
9)	CER/F	Complex event recognition / forecasting (CER/F)	T3.1-2-3	WP3
10)	IVA	Interactive visual analytics	T4.3	WP4
11)	Viz	Real-time visualization	T4.4	WP4

Table 1: The modules in the datAcron integrated prototype.

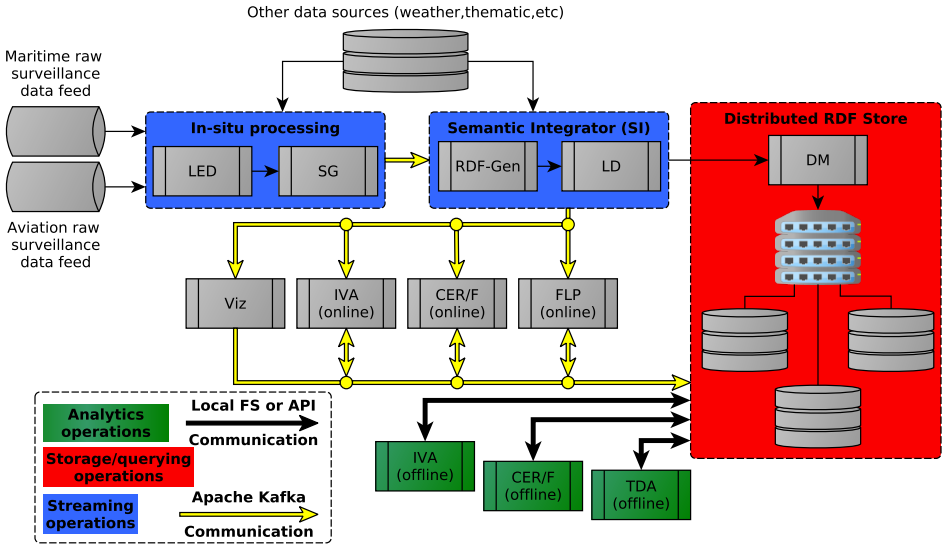


Figure 1: The datAcron Integrated Prototype.

Figure 1 illustrates the overall architecture of the integrated prototype. The datAcron modules are denoted by blocks in the diagram and links between them denote flows of information. The yellow links indicate Kafka-based communication between modules. In-situ processing components and the Semantic Integrator (SI) modules have direct access to third party data sources such as weather forecasts and thematic/contextual source. The in-situ processing module con-

sumes raw surveillance data (maritime or aviation) and the output is provided to SI. The output of SI is consumed by modules Viz, IVA online, CER/F and FLP, which use Kafka for communication with other modules. The output of SI towards DM indicates the data loading operation in the distributed RDF store maintained by DM. Finally, the flows from offline components (IVA offline, TDA, CER/F) towards DM correspond to the querying functionality supported, which allows spatio-temporal RDF queries and retrieval of corresponding data.

2.1 datAcron Modules

The datAcron architecture consists of the following modules listed in Table 1. A short description of each module follows below.

- **Raw surveillance data feeder (MDF/ADF)**: The raw surveillance data feed comprises maritime data (MDF) from a range of terrestrial AIS receivers and 18 low earth orbit satellites, and aviation data (ADF) from European Surveillance data feeds. The maritime AIS data stream is collected, tested for veracity using a streaming analytics module, cleaned and annotated with various error flags, and then filtered to provide the data required for the datAcron project. The aviation data stream comprises Flightaware, ADSBExchange and ADSBHub sources.
- **In-situ processing**: In-situ processing in general refers to the ability to process data streams as close to the source where data originates. This part of the architecture contains the Low-level event detector (LED) and Synopses Generator (SG):
  - **LED**: In-situ processing is in particular challenging, when the stream processing of the data requires additional input from other sources, either other instances of the stream or from global system settings or user interaction. As a proof of achieving the architectural integration of in-situ processing, datAcron has implemented a distributed online learning framework on distributed streams as described in detail in D1.8, allowing for faster adaptation of models to changes in real-time and providing an enriched event stream for visualization in real-time. Based on this, event forecasting has been extended towards distributed online learning as described in D3.5, making it possible to learn forecasting models cross-stream from other moving objects.
  - **SG**: The Synopses generator consumes streaming positioning messages of raw surveillance data and eliminates any inherent noise such as delayed or duplicate messages. Moreover, it identifies critical points along each trajectory, such as stop, turn, or speed change, in order to provide an approximate, lightweight synopsis of movement per moving object.
- **Semantic integrator (SI)**: This module comprises two major parts: (a) it transforms data from all sources to RDF w.r.t. to the datAcron ontology and (b) it discovers links between entities in the RDF data. The first part is accomplished by RDFgen, which homogenizes different sources and allow trivial links between sources to be detected. The second part applies link discovery tasks (LD) and enriches the stream of positional information with links to additional data required by the use case scenarios. Both RDFgen and LD have been thoroughly discussed in Deliverable D1.9. The output of this module is also sent directly to Data Manager for storage and future use.

- **Data manager (DM)**: Its functionality is to store information into a distributed spatio-temporal RDF store and provide query answering facility.
- **Trajectory / Future location predictor (T/FLP)**: FLP calculates motion functions by harvesting the cleansed Kafka stream (from the Synopses Generator module). Specifically, it consults the most recent locations of a moving object to predict its short-term future location in real time, w.r.t. the tendency of the movement. Each predicted point is streamed out in real time to other modules. Regarding TP , it presents a similar functionality targeting at predicting the future trajectory of a moving object as far in time horizon as possible.
- **Trajectory data analytics (TDA)**: The goal of this module is twofold: on the one hand it provides advanced analytics that serve specialized requirements in the datAcron architecture (e.g. data-driven discovery of the networks/routes upon which the movement of the vessels/aircrafts take place), while on the other hand it provides global patterns that represent meta-models devised from the local patterns (e.g. clusters and sequential patterns of trajectories).
- **Complex event recognition/forecasting (CER/F)**: CER aims to real-time detection of complex events, whereas CEF aims to real-time forecasting and prediction of complex events. Both modules are working on synopses of moving objects generated by the two in-situ processing modules (LED & SG). It also consults information provided by the link discovery tasks performed in SI. The output of CER is a real-time stream with detected events. On the other hand, CEF enriches the input stream with a forecast about the probability of each monitored event pattern.
- **Interactive visual analytics (IVA)**: The IVA module builds on top of the real-time visualization module to provide limited analytical capacity on streaming data. The primary use is to allow analysts, and possibly advanced operators, to fine-tune and observe the impact of parameter adjustments to the T/FLP and CER/F modules compared to actual data in (near) real-time. It therefore complements the situation monitoring capabilities of the real-time visualization used by ordinary operators on the one hand (by providing parameter settings to the detection modules), and the full-fledged VA suite used for in-depth exploration and analysis in offline (strategic latency) settings.
- **Real-time visualization (Viz)**: This module provides a map-based visualization of the stream of enriched spatio-temporal events generated by the T/FLP and CER/F modules. It is able to display different event types (e.g., critical points) simultaneously with individual visual encoding for each type. In addition events associated with the same moving object identifier are automatically integrated into trajectory representations so operators can observe movement patterns. The overall design follows the "overview-first, zoom-and-filter, details-on-demand" approach, meaning that operators can define filters on the input stream to drill down on areas and event types of interest.

2.2 datAcron Flows

The modules in the integrated datAcron system communicate through three different information flows, classified as operational, tactical and strategic w.r.t. the reaction time [2].

In particular:

- *Information management* flows are about the reconstruction of trajectories and their enrichment with useful annotation, which is to be performed online (operational latency), and their storage for querying purposes, which is to be performed offline (tactical latency).
- *Online analytics* flows are about consuming the available streaming information, which is to be performed online (operational latency); and
- *Offline analytics* flows are about consuming the available stored information, which is to be performed offline (strategic latency).

Note that there exist three main consumers (namely, T/FLP, CER/F, and IVA), each operating both online and offline. In turn, this means that 3 online flows concern these modules, namely flows #3, #4 and #5, whereas the 3 offline flows are #6, #7 and #8. Table 2 presents the list of flows (along with the respective latency type).

Flow Nr.	**Flow Title**	**Latency**
Information management flows		
1)	Trajectory reconstruction and semantic enrichment	Operational
2)	RDF storage	Tactical
Online analytics flows		
3)	Trajectory/FL prediction online	Operational
4)	Complex event recognition / forecasting online	Operational
5)	Visual Analytics online	Tactical
Offline analytics flows		
6)	Trajectory data analytics offline	Strategic
7)	Complex event recognition / forecasting offline	Strategic
8)	Visual Analytics offline	Strategic

Table 2: The datAcron flows of information.

[2] Operational latency: milliseconds, Tactical latency: few seconds, Strategic latency: tens of seconds or minutes (defined in Deliverable D1.1 "Requirements Analysis")

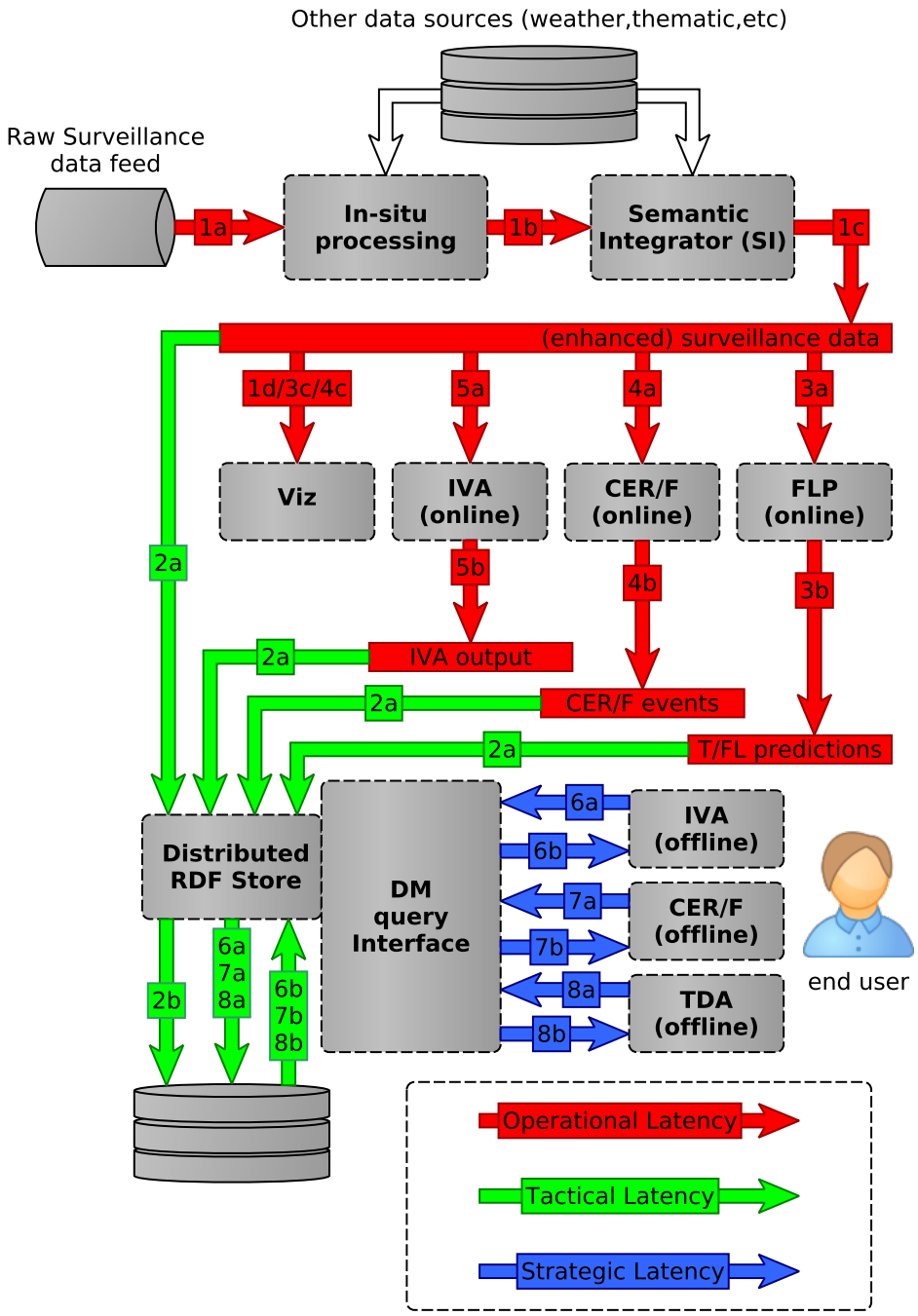


Figure 2: The datAcron Integrated Prototype (view equivalent to Figure 1) showing the flows more clearly.

The functionality of each flow is discussed in the following paragraphs. In accordance with the flows, Figure 2 illustrates the datAcron architecture, which is an equivalent view to the one presented in Figure 1. Also, this illustration is a refined version of the architecture specified in

Deliverable D1.2 "Architecture Specification".

Flow #1: Trajectory reconstruction & semantic enrichment

- Short description: Maritime / Aviation raw data stream is (1a) cleansed, enriched with derived information (e.g. speed) as well as low-level events (e.g. intersection with zones of interest), synopsized by tagging "critical points" (change of heading or altitude, etc.), and (1b) further enriched with info from other external data sources / streams (weather info, etc.); the final output (1c) is streamed out to be consumed by other modules, including its visualization (1d).
- Modules involved: Maritime / Aviation; LED; SG; SI; Viz.

Flow #2: RDF storage

- Short description: A subset of the enhanced surveillance data stream, i.e. the annotated, synopsized surveillance data, as well as selected output streamed out by other modules is (2a) processed and (2b) stored in the RDF store.
- Modules involved: DM.

Flow #3: Trajectory/FL prediction online

- Short description: T/FLP (3a) consumes the enhanced surveillance data stream (as well as other streams, provided internally to the datAcron integrated prototype, if needed) for the purposes of online trajectory / future location prediction and (3b) streams out its output to be consumed by other modules, including visualization (3c). Also, prediction results are stored in the form of Parquet files in HDFS for evaluation purpose.
- Modules involved: T/FLP; Viz.

Flow #4: Complex event recognition / forecasting online

- Short description: CER/F (4a) consumes the enhanced surveillance data stream (as well as other streams, provided internally to the datAcron integrated prototype, if needed) for the purposes of online event recognition / forecasting and (4b) streams out its output to be consumed by other modules, including its visualization (4c).
- Modules involved: CER/F; Viz.

Flow #5: Interactive visual analytics online

- Short description: IVA consumes the enhanced surveillance data streams (3c, 4c), streamed meta data on the T/FLP and CER/F modules (current parameter settings, 5a), and, if needed, base data for comparison (1d) for the purposes of online VA; and (5b) streams out its output (updated parameter settings, areas-of-interest) in KVP format to be consumed by other modules.
- Modules involved: IVA; T/FLP; CER/F.

Flow #6: Trajectory data analytics offline

- Short description: TDA (6a) queries the RDF store in order for complex patterns to be discovered and (6b) stores selected results back to the RDF store for future use.
- Modules involved: TDA; DM.

Flow #7: Complex event recognition / forecasting offline

- Short description: CER/F (7a) queries the RDF store in order to fetch data for complex events to be detected/forecasted and (7b) stores selected results back to the RDF store for future use.
- Modules involved: CER/F; DM.

Flow #8: Interactive visual analytics offline

- Short description: IVA (8a) queries the RDF store to get large batches of raw data for complex offline analysis and (8b) stores selected results (derived attributes, spatio-temporal patterns, clustering results, parameter settings) back to the RDF store for future use.
- Modules involved: IVA; DM.

2.3 Online Part: Stream Processing

The majority of data analytics operations in datAcron are performed in an online fashion using stream processing. These operations include data transformation and integration, as described in FAIMUSS [6], trajectory compression and semantic trajectory construction, as presented in SPARTAN [7], and complex event recognition supported by efficient spatio-temporal link discovery, as outlined in [9].

2.4 Offline Part: Batch Processing

The main component responsible for the batch processing in the datAcron architecture is the distributed spatio-temporal RDF engine [2]. Figure 3 depicts the architecture of the datAcron distributed RDF engine. It has been designed and built from scratch during the course of the project. By design, the datAcron distributed RDF engine targets the following objectives:

- Scalable storage and processing for vast volumes of RDF data, in the order of Billions of RDF triples (targeting the *Volume* dimension of Big Data)
- Support for spatio-temporal RDF data, i.e., RDF data which are mostly associated with spatio-temporal information
- Efficient processing strategies that employ optimization techniques, in order to prune significant subsets of data and reduce execution time

- A prototype implementation that includes the functionality provided by a typical data warehouse system, from loading data to querying and optimization.

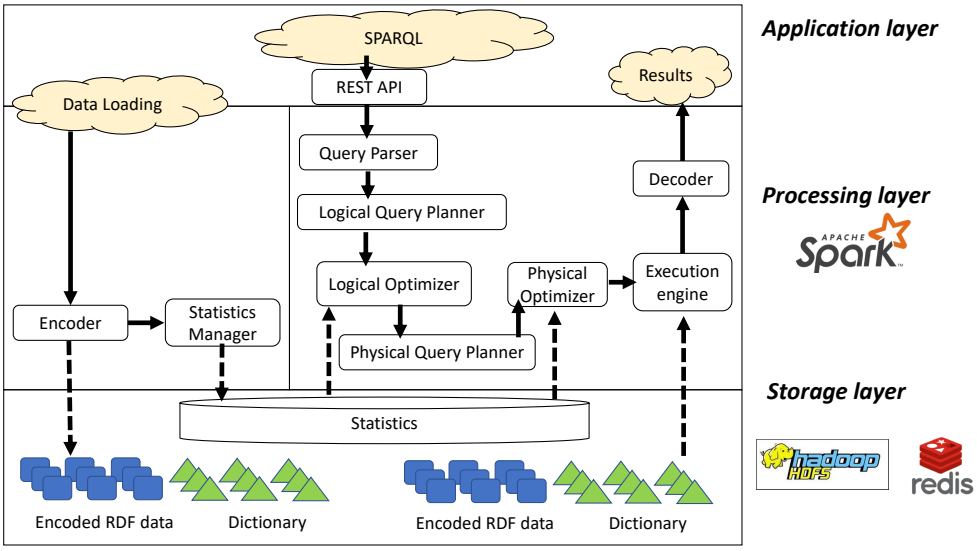


Figure 3: Overview of the datAcron distributed RDF engine.

The datAcron distributed RDF engine encompasses two main layers: (a) storage layer, and (b) processing layer. In addition, an application layer has been introduced to demonstrate the offered functionality and allow its use by external applications.

An application that uses the datAcron distributed RDF engine can use two main operations: data loading and querying.

The *data loading* operation considers as input an RDF data set that can be arbitrarily large. In the context of datAcron, this data set consists of spatio-temporal RDF data and it is the output of the online data transformation and link discovery components described in Deliverable D1.9. During data loading, the RDF data is encoded to integer values based on the Encoder [12]. The Encoder produces as output encoded RDF data and a dictionary that allows decoding of integer values back to strings (URIs or literals). The encoded RDF data is stored in HDFS, whereas the dictionary is stored in-memory using REDIS, a scalable in-memory distributed key-value store. In addition, during the data loading operation, the Statistics Manager computes various statistics about the loaded data, which are also stored to be used during query execution, in order to derive an optimized query execution plan.

The *querying* operation takes as input a SPARQL query, which is directed to the processing layer of the engine for query evaluation, and provides the matched RDF triples as result. The processing layer is implemented in Apache Spark, a state-of-the-art solution for batch, in-memory processing of Big Data. First, the SPARQL query undergoes query parsing, which aims to ensure correct formulation and to represent it as a logical query plan that can be manipulated by the other components of the processing engine. The logical query plan is given as input to the *logical optimizer* whose role is to perform a set of standard optimizations (such as deciding the order of join operations), and generate a set of potential physical execution plans. Then, the *physical optimizer* examines these physical plans and performs cost-based optimization, also exploiting the

available statistics, to select the best physical execution plan. This is provided to the execution engine that performs query processing and provides the resulting RDF triples. Prior to result delivery to the application, a *decoder* performs translation of the encoded triples back to their string representation.

3 Evaluation Methodology

This section describes the methodology adopted for the evaluation of the integrated prototype as well as the underlying technical details related to the communication of the datAcron components. To this end, first, a brief technical description of components is provided. Then, the intercommunication mechanism is presented, which is based on Kafka, and serves as the communication platform for the implemented online flows in datAcron. Finally, the mechanism for measuring the performance of Kafka-based communication is presented, which is used in our experimental evaluation.

The evaluation methodology has been designed so as to cover two main aspects: stream/online processing and batch/offline processing, in order to demonstrate the basic functionalities provided by the integrated prototype end-to-end.

Regarding stream/online processing, we choose to evaluate flow #1 (trajectory reconstruction and semantic enrichment), which is the fundamental processing flow of online processing in datAcron, as it "feeds" the prediction/forecasting components with an appropriately enriched stream of data.

Regarding batch/offline processing, we evaluate the interoperability of trajectory clustering (as a representative data analysis technique) with the distributed RDF store that provides the necessary batch processing of integrated data. In this way, the evaluation covers both aspects of Big Data processing in the context of datAcron.

3.1 Technical Description of Components

In the following, a brief technical description of each component is provided, focusing mainly on implementation aspects related to the parallelization of processing. The implemented components have been developed using Big Data technologies, and owe their efficiency and scalability to the underlying parallelization techniques used for partitioning the work to different computing nodes in a cluster.

3.1.1 Trajectory Synopses component

The Trajectory Synopses component [3, 4] is written in Flink. Since maintaining state and communication turned out to be troublesome (see the corresponding deliverable D2.3) the trajectory synopses application uses a utility called KafkaSplitter to partition the incoming stream into multiple Kafka Topics. Then, multiple Flink instances of the Trajectory Synopses application are launched each reading from a single Topic.

Because of the fixed partitioning policy that Flink uses, by default all instances of the application wrote their result in the first partition of the Kafka Topic. To overcome this limitation, a custom partitioner that writes to different partitions for each instance has been implemented.

3.1.2 RDFgen

The RDFgen component [8] is at its heart a simple Map function that takes as input the result of the Trajectory Synopses component and returns the result as an RDF fragment. Since it is

a simple Map function with no need for aggregations or complex load balancing we tried two approaches:

1. A YARN container implementation. This implementation takes the multi-threaded Java implementation and launches multiple instances inside YARN containers.

2. A Flink implementation. This implementation uses the single thread logic of the RDF-gen as a basis for a Map function in Flink. Flink then takes care of the launching and parallelisation of the application.

3.1.3 Link Discovery Framework

The link discovery tasks in datAcron project are mainly proximity and topological relations between spatio-temporal representations of entities reported in the datAcron data sets. We have extended the baseline grid partitioning technique with the MaskLink approach [5]. MaskLink exploits the empty space of the cells, to prune comparisons between entities of the *source* and *target* data sets. Salient features of MaskLink include that it has been designed both for point-to-point and point-to-region topological and proximity relations, and can operate with streams as inputs. Supporting proximity relations in the context of link discovery of spatio-temporal data should be emphasized, since existing works have only targeted topological relations.

However, the benefits of MaskLink are highly dependent to the data sets used. Specifically, the first observation is that if we obtain a spatial distribution of target data set, s.t. each cell has exactly one geometry, there is no use for mask, since the baseline method also needs exactly one comparison to decide whether a link holds. Another important observation, is that the method can be used in data sets that are known to allow empty spaces between spatial representations of entities in the target data set. Such data sets in datAcron project are Fishing regions, coastlines of island clusters (e.g. as in Aegean Sea), Natura2000 regions. Apparently, it cannot be applied when the geometries of the target data set are adjacent with is no emtpy space, since the method is regressed to the baseline method (i.e., grid partitioning). Such data sets in the datAcron project, are the Exclusive economic zones (EEZ), the Administrative regions, and the airblocks or sectors of airspace configurations.

Parallel Link Discovery Porting the MaskLink method to a parallel processing environment (e.g. Flink), requires the definition of: a) how data are partitioned to workers, and b) what data are replicated across the workers.

In addition to that, the link discovery component has to satisfy the requirement that all the information about a single entity (including the discovered links) should be provided as a single message in the output, i.e. a self-contained RDF graph fragment. This requirement is set by datAcron components that directly consume data computed by the link discovery tasks, and it can be omitted when components directly request data from the triple store (i.e. since in this case the way triples are provided from the link discovery task, does not affect the results of the triple store).

The first approach was to distribute the link discovery task per cell to workers, i.e. it partitions the data sets by space and replicates the grid (or part of the grid) to the workers. In this approach each worker is responsible for a (non-empty) set of cells of the grid, and evaluates each spatio-temporal representation reported in its cells, independently from the rest of the workers.

This approach however is not adaptive to the distribution of data, i.e. there can be a data set s.t. most of the work load will always be assigned to only a few (or only one) workers. One such example in the datAcron project is the link discovery task for detecting "within" 3D relations between surveillance positions and 3D airblocks. Figure 5 illustrates the distribution of source

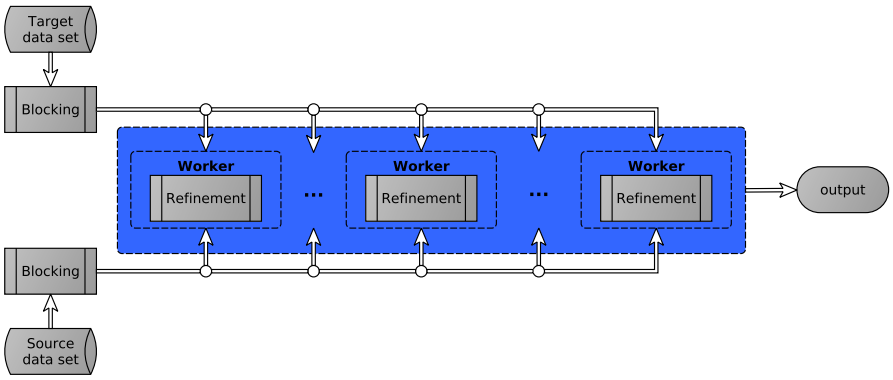


Figure 4: The abstract illustration of the grid based LD approach

data set entries in the cells of a 3.5x3.5 grid. We observe that five cells contain more than $3 * 10^5$ entries, i.e. these workers will have most of the workflow for this data set. Configuring a more fine grained grid will not solve the problem, i.e. some regions (e.g., close to airports) will always have more dense data compared to the rest of the covered space.

A different approach that is more prominent to handle skewed data, and achieve a balanced load distribution, is the construction of a spatial index from the target data set, and assign to workers the refinement tasks, as illustrated in Figure 6. In this case, we partition the data set per entity (i.e. by ID), but we have to replicate the spatial index. The method is initialized with the population of the spatial index by entities of the target data set. Next, for each spatio-temporal representation reported in the source data set, the spatial index is queried and the set of candidate entities of target data set for comparison are selected. This set also specifies the set of refinement tasks to be carried out by workers. Each worker will compute all the comparisons of a single source data set entity, independently from the rest of the workers. Thus, this method is expected to achieve better load distribution to workers, compared to grid-based approaches. On the other hand, the spatial index is replicated to the workers, thus the method cannot be used for huge or streaming target data sets.

This method has been implemented with STR-Tree as a spatial index. The STR-tree is a query-only R-tree created using the Sort-Tile-Recursive (STR) algorithm [1] to use for two-dimensional spatial data. We have selected the STR packed R-tree, since it maximizes space utilization, i.e. as many leaves as possible are filled to capacity, while overlap between nodes is far less than in a basic R-tree. The STR-Tree is broadcast to all the workers.

3.2 Kafka as Message Bus

Deliverable D1.2 which describes the datAcron system architecture has documented and justified the choice of Apache Kafka as intercommunication platform for the online datAcron components.

Nowadays, Kafka is the de-facto message bus for Big Data architectures, due to its salient properties including high performance, built-in partitioning, replication and fault-tolerance. In addition, Kafka is compatible with a variety of Big Data processing frameworks, such as Apache Spark or Apache Flink, which are the preferred development frameworks in datAcron for the various components.

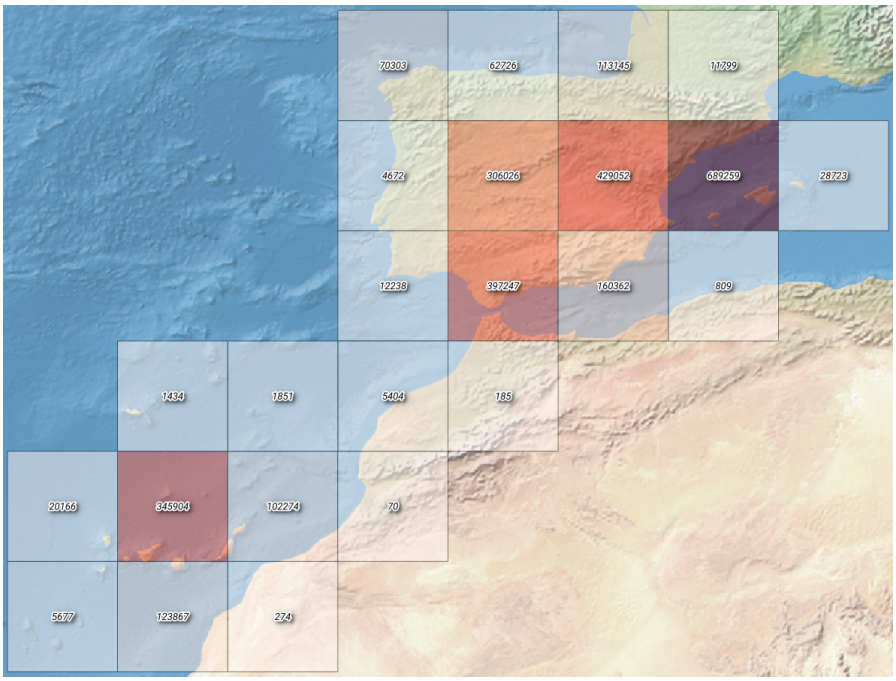


Figure 5: The distribution of spatio-temporal representations provided by source data set in a grid with cell size 3.5x3.5

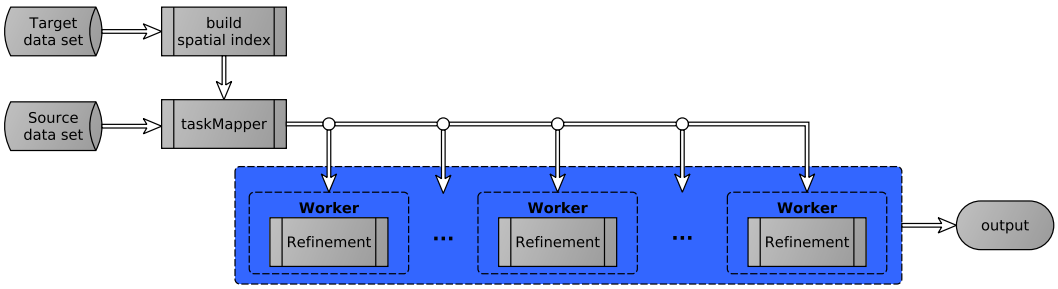


Figure 6: The abstract illustration of the spatial index implementation.

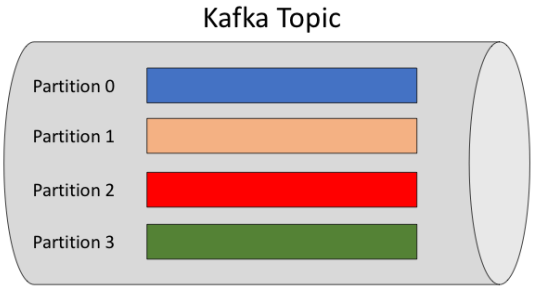


Figure 7: Kafka partitioning.

Kafka is a distributed, partitioned, replicated commit log service. The unit of data within Kafka is called a message. A message typically corresponds to a record or a row of a relational table.

Kafka Topics. Each message in Kafka is a key-value pair. Kafka maintains feeds of messages in categories called *Topics*. Each Topic is partitioned for scalability and Partitions are distributed in the cluster. Figure 7 illustrates a Kafka Topic that contains four partitions, depicted using different colors. Essentially, a data set that is published as Kafka Topic can be split in different partitions. The default partitioning strategy is hash partitioning based on the value of the key. The user can also define a custom partitioning strategy to accommodate complex business logic. Topic partitioning enables applications that will read data from the Topic to consume simultaneously different pieces of data, i.e., read from different partitions.

For efficiency, messages are written into Kafka in batches. A batch is just a collection of messages, all of which are being produced to the same topic and partition. A partition is a single log. Messages are written to it in an append-only fashion, and are read in order from beginning to end. A topic generally has multiple partitions and there is no guarantee of time-ordering of messages across the entire topic, just within a single partition.

Kafka Brokers. A Kafka architecture in a cluster consists of *Kafka brokers*, which are essentially the server nodes of the cluster. In other words, Kafka runs in a cluster comprised of one or more servers each of which is called a broker. Partitions allow parallelization by splitting the data in a particular topic across multiple brokers.

Thus, each broker maintains a number of partitions and each of these partitions can be either a leader or a replica for a topic, as depicted in Figure 8. All writes and reads to a topic are directed to the leader partition, which is responsible for updating the replicas in order to keep them consistent. Also, this replication mechanism is exploited in the case of failure of a leader; in such a case, a replica is assigned the role of new leader.

Kafka Producers and Consumers. Processes that publish messages to a Kafka Topic are called *Producers*. Processes that subscribe to Topics and process the feed of published messages are called *Consumers*. Consumers work as part of a *consumer group*. This is one or more consumers that work together to consume a topic. The group assures that each partition is only consumed by one member.

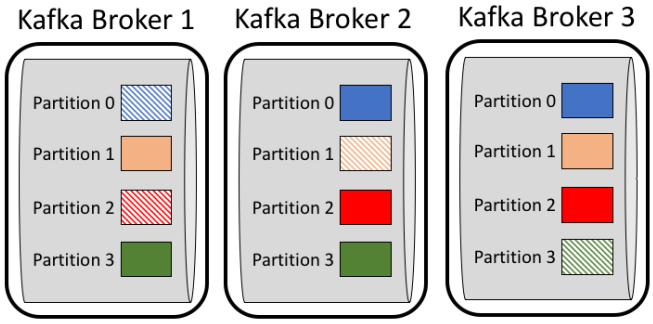


Figure 8: Kafka brokers and partitions of a topic (partitions with texture correspond to leader partitions, whereas partitions with solid coloring correspond to replicas).

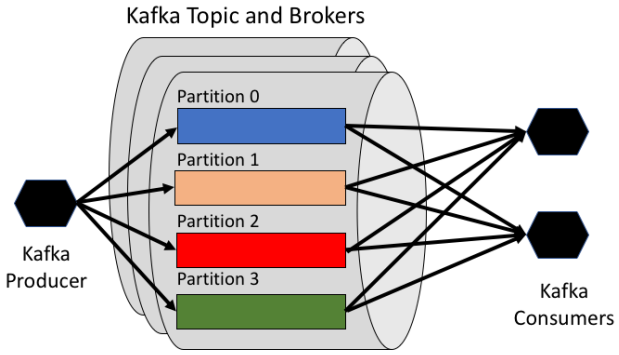


Figure 9: A Kafka producer publishes a topic and two Kafka consumers read from the topic.

A Kafka broker receives messages from producers, assigns offsets to them, and commits the messages to storage on disk. It also services consumers, responding to fetch requests for partitions and delivering the messages that have been committed to disk.

Kafka Partitioning. Partitioning of Topics is the standard way of achieving parallelism in Kafka. Since each consumer in a parallel processing framework can poll at least one partition of a Topic and no two consumers in the same consumer group can poll the same partition, the partitions of a Topic implicitly determine the available parallelism for the application. This is why for the standalone measurement of the components we ensure that the number of partitions in the input topic is at least as high as the number of consumers in the application.

3.3 Kafka Interceptors

Kafka Interceptors are a mechanism provided by Kafka to mutate records before they are received by the `poll` function (Consumer Interceptors) or before sending them over the wire (Producer

Interceptors).

In our case we use Producer Interceptors to measure latency on the packets before they are sent over the wire.

To measure the latency of each flow using Kafka Interceptors we use the following procedure.

1. For each record entering the datAcron pipeline we have a field called `IngestionTimestamp`
2. The consumer interceptor records the current time in milliseconds when the record enters the pipeline into the `IngestionTimestamp` field. The latest versions of Kafka (since Kafka 0.10) have also this field into each record and if the end user does not want to use the Consumer Interceptor all he has to do is to record the current time in milliseconds and enter it into the `IngestionTimestamp` field.
3. Just before the processed record is about to get into the wire, we measure the latency for the record using a `Producer Interceptor` in which we have overridden the `OnSend` method. The `OnSend` method is the method called by Kafka just before the data is sent over the wire.

In our test case each Kafka Producer records the totalLatency for all records, the minimum and maximum latency as well as the average latency.

3.3.1 Guidelines

According to the guidelines provided to WP2 and WP3, to measure latency using Kafka Interceptors the user (in our case "component developer") has to perform the following steps:

1. Create his version of consumer and producer interceptors using our code listings as a starting point. Alternatively the user can Import Interceptors for the data types of the datAcron project, using Gradle[3] as the build system.
2. Import the consumer and producer interceptors in the consumer and producer properties respectively. The consumer and producer properties is a hash table containing settings for the Kafka Producers and Consumers respectively.

In our use case we make the assumption that each record entering the pipeline exits the pipeline enhanced with new information. The input and output datatypes do not need to be the same, but if they are not we cannot use the `Consumer Interceptor`.

If the assumption above does not hold then the user must inject the `IngestionTimestamp` field at the start of the processing for each output record and use the producer interceptor to measure the latency.

Our implementation of the Interceptors measures the minimum, maximum and average latency. One way to provide more fine-grained information is to have the Kafka Topics serialized into a file and then acquire the latency for each message by using the `ingressTimestamp` and `outgressTimestamp` fields.

3.4 Prometheus TSDB for Monitoring Kafka and Spark Applications

In the datAcron cluster, we have setup a Prometheus time-series database (TSDB) to monitor the number of messages inside Kafka Topics as well as the performance of Spark Applications.

[3] https://gradle.org/

Prometheus is an open-source systems monitoring and alerting toolkit originally built at SoundCloud[4].

- It is a standalone Go binary
- It provides an HTTP Interface to get the metrics
- It provides a query language, PromQL to get values and aggregates

Each broker in the cluster is tied to a JMX exporter, that exposes JMX beans as json over HTTP. The master node of Spark application also exposes JMX beans in a similar manner. The master node runs a prometheus TSDB, that scrapes the JSON objects from multiple nodes at regular time intervals.

[4] https://prometheus.io/docs/introduction/overview/

4 Evaluation Results: Online Part

In this section, we present the evaluation of the integrated prototype performed in the datAcron private cluster. As the cluster has limited resources (10 computing nodes), this puts a barrier to testing scalability, since setups with higher provision of resources cannot be tested.

In the following, we first describe the cluster setup in Section 4.1, along with details about the experimental evaluation. Then, in Section 4.2, we perform the standalone component evaluation, which is helpful in order to identify tuning parameters for each component individually. In Section 4.3, we evaluate the performance achieved by a selected sequence of components that run simultaneously as a pipeline. Finally, we discuss the overall results in Section 4.4.

4.1 Experimental Setup

In a nutshell, the datAcron cluster is comprised of 10 computing nodes in total located in a rack. Three nodes are configured as Kafka brokers, but they can also run other tasks. The hardware specifications for each computing node is as follows:

- 2*XEON e5-2603v4 6-core 1.7GHz with 15MB cache
- 128 GB of RAM
- 256 GB SSD

Methodology Our main focus is on the stream processing part of the datAcron architecture, which consists of many components that interact and intercommunicate to achieve the required functionality.

More specifically, we focus on the components that perform trajectory compression, data transformation to RDF, and link discovery, as this chain of components is the most critical for serving higher-level components, and at the same time perform the most resource-demanding operations.

Therefore, the evaluated system takes as input raw surveillance data, generates synopses, performs trajectory reconstruction, transforms trajectories in RDF based on the datAcron ontology [10, 11], and links trajectory positions with information about crossed 3D sectors.

Data sets To evaluate the integrated prototype we use data from the air-traffic management (ATM) use-case and domain. The reason is that this is the more challenging setup, since in the maritime domain data is 3D and not 4D.

In particular, we use surveillance data from the data source IFS, covering the area of Spain for the period of April 2016.

In addition, we use a contextual data source that consists of 3D sectors, as we are interested to link an aircraft's surveillance data with those sectors crossed by the aircraft.

Kafka configuration The Kafka Producers have been configured with the following settings aimed to provide a middle ground between throughput and latency requirements:

1. batch.size 16384

2. buffer.memory 33554432
3. compression.type = none
4. linger.ms = 1

Metrics In the evaluation, we use the following metrics:

1. *Throughput*: expressed as number of messages per second.
2. *Speedup*: defined as the throughput achieved by the parallel version of a component (or sequence of components) divided with the throughput of the exact same component with parallelism equal to 1, i.e., centralized case.
3. *Latency*: the average, minimum and maximum processing latency of each component.

4.2 Standalone Component Evaluation

For the evaluation of individual components as "standalone", we use the following ones:

- Trajectory synopses generation (SG)
- Data transformation to RDF, also mentioned as "RDFizer" in the following, which is part of Semantic Integrator (SI)
- Link discovery component, which is part of Semantic Integrator (SI)

In this way, we evaluate both the functionality of in-situ processing as well as the semantic integrator, which comprise the backbone of the datAcron streaming architecture. This provides an adequate view of the system bottlenecks. At the end of this chain of components, any other (online or offline) component can be added.

We measure the performance and scalability of each component, when varying the level of parallelism from 1 (centralized execution) to 60. The level of parallelism corresponds to the number of different tasks initiated in the cluster.

4.2.1 Trajectory Synopses Generation

Figure 10 shows the throughput results obtained for all components. In terms of throughput, the Synopses Generator (SG) scales very well on Flink achieving 100K messages/sec using 9 workers and 360K messages/sec for 20 workers. The run with the 40 workers shows worse performance, because there were not enough resources in the YARN cluster to be allocated for the execution. Therefore, for this component, it does not make sense to test higher level of parallelism in the given infrastructure.

Figure 11 depicts the speedup for all components. The Synopses Generator (SG) achieves a speedup of 8x for 9 workers, and 31x for 20 workers. Recall that the speedup has been defined using messages/sec as unit of measurement, and practically shows the scalability of the component due to parallelism, compared to the centralized case.

Figure 12 depicts the latency for all components. The average latency the Synopses Generator (SG) is around 60 msec for the setup of 20 workers. This is an improvement over the centralized case, where the average latency is around 120 msec. This result shows that the trajectory synopses generation process manages to meet the requirement of operational latency.

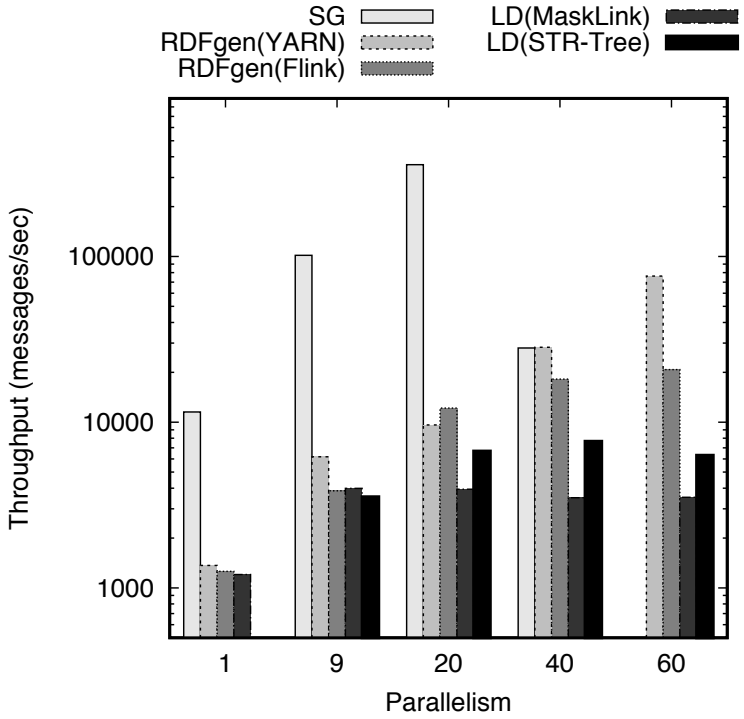


Figure 10: Standalone Component Evaluation: Throughput in messages/sec.

4.2.2 RDFgen

We have tested two different ways to parallelize the data transformation to RDF. In the first case, we simply run the RDFgen application on YARN, thus parallelizing the application by execution in multiple nodes and cores. In the second case, we test a Flink-based implementation of the RDFgen, where parallelization is performed by Flink.

RDFgen on YARN The RDFgen on YARN scales really well up to 60 virtual cores achieving a performance of 76K messages per second (Figure 10) and a speedup of 55x (Figure 11).

In Figure 12, the latency remains under 2 msec for all executions of the RDFgen. This proves that the RDFgen can scale really well for large number of cores.

RDFgen on Flink Executing RDFgen on Flink seems to scale worse than its counterpart in YARN. More specifically for 60 virtual cores the RDFgen achieves throughput of 20K messages/sec (Figure 10) and a speedup of 16x (Figure 11).

The above results suggest that while running the RDFgen on Flink seems to add unnecessary overhead, it also seems to help scalability. For all the experiments on Flink the latency was less than 1 msec (Figure 12).

4.2.3 Link Discovery

For the link discovery task, as explained in Section 3.1.3, we employed two variants, one based on MaskLink and one using the STR-Tree.

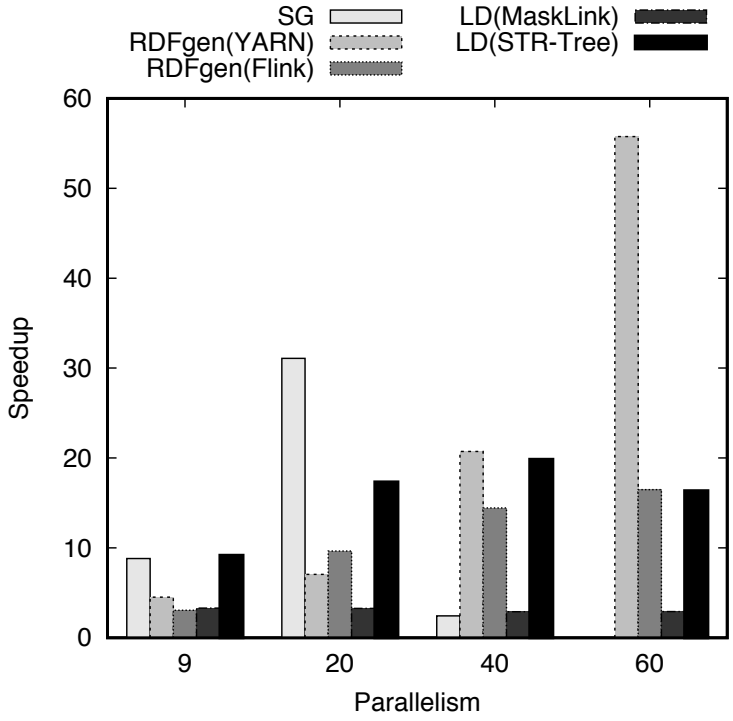


Figure 11: Standalone Component Evaluation: Speedup.

MaskLink For the evaluation of the LD component we use a grid constructed by cells of 3.5×3.5 degree. As discussed in section 3.1.3, MaskLink method is beneficial in data sets where spatial representations of entities are not adjacent, i.e. there is an amount of "empty" space within cells. In this set of experiments, we evaluate MaskLink on detecting within (in three dimensions) relations between surveillance data (source data set) and 3D airblocks (target data set). Since this target data set leaves no empty space in the cells and the source data set is highly skewed, this configuration can be considered as the worst case scenario. Figures 10, 11 and 12 illustrate the throughput, speedup and average latency respectively, in configurations varying from 1 to 60 workers. We observe that the configuration of 9 workers achieves the maximum speedup, which can be explained by the skewness of the source data set, i.e. although we provide more workers, a large number of records in the source data set will be processed by the same workers. Although, we can use a more dense grid (i.e. cells of size less than 3.5×3.5) to exploit more workers, the skewness in the source data set will always force a number of workers to become a bottleneck for the overall scalability.

STR-tree LD The STR-tree index employed in the LD component is useful in configurations where target data set is expected to contain adjacent spatial representations and source data set is skewed. The STR-tree index is built from the target data set instead of cells of a uniform grid, prior to processing the source data set for link discovery. Figures 10, 11 and 12 illustrate the throughput, speedup and average latency for the LD component when exploiting the STR-tree index. We observe that the maximum speedup is now detected in the configuration of 40 workers. We also observe that this configuration achieves higher throughput and lower latency compared to the configuration without STR-tree, as a result of better distribution of load to the workers.

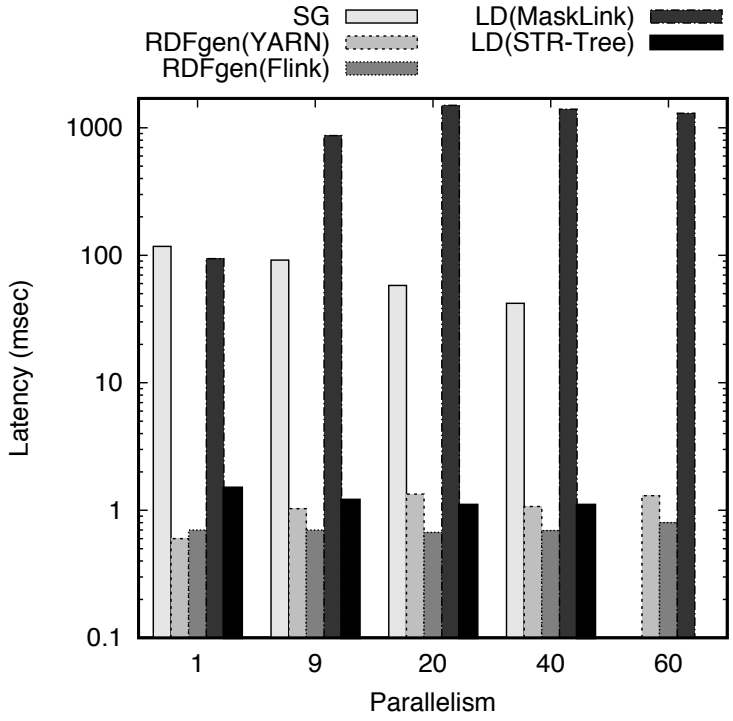


Figure 12: Standalone Component Evaluation: Latency in msec.

4.3 Evaluation of Chain of Components

Apart from the single component evaluation we run an end-to-end typical scenario using the datAcron architecture. Each component was given the same amount of resources per run; 9 and 20 in our case. Each component was given the same amount of resources to keep the number of Kafka partitions the same, so that all the components can achieve maximum parallelism. We used the Flink variant of RDFgen, and STR-Tree for the link discovery method.

Figure 13 depicts the obtained results. The end-to-end throughput is determined by the slowest component. For parallelism equal to 9 the integrated prototype achieves 3.5K messages per second for the slowest component. All the components of the architecture exhibit more or less the same performance. Note that the performance of the trajectory synopses component seems lower because this time we measure its throughput by the emitted critical points and not by the number of locations processed. This is because in our architecture we transform to RDF and store only critical points. In a real-life scenario, we would be interested in how many critical points are emitted, while for benchmarking purposes in standalone mode we would be interested in how many input records are processed per second to keep the measurements comparable to the other components.

For parallelism equal to 20 both the trajectory synopses component and the RDFgen seem to scale well, while the Link Discovery framework exhibits the same performance as in the 9 node case. This could be due to resource contention in YARN.

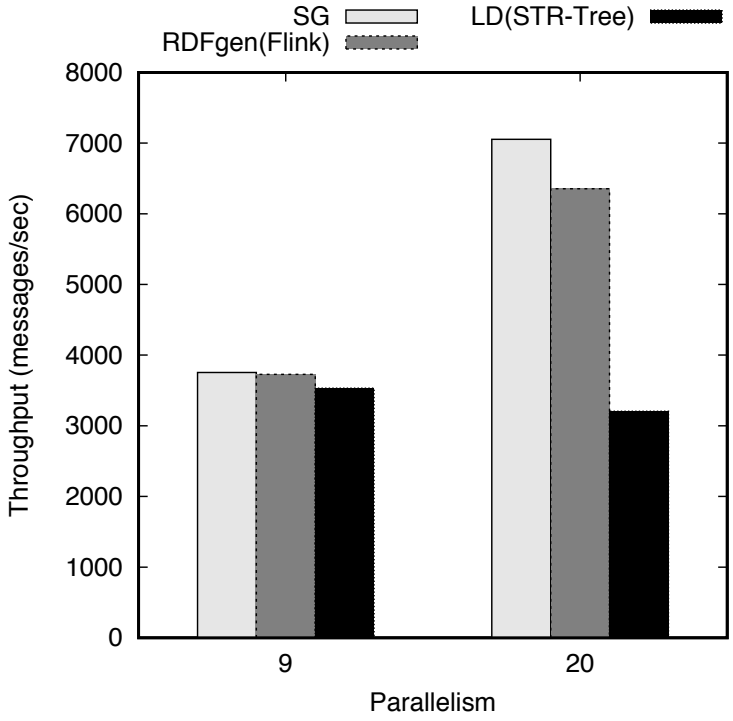


Figure 13: Integrated Prototype Evaluation: Throughput in messages/sec.

4.4 Discussion of Results

In this section, we discuss the obtained results from the online part of the architecture in the context of the datAcron objectives:

- The individual online components achieve operational latency (under 1 sec), with appropriate parameterization, thus matching the requirements set at deliverable D1.1.
- The Synopses Generator and RDFgen achieve always latency lower than 100 msec, and in most setups much lower.
- The Link Discovery component, which performs the most compute-intensive operation, exhibits operational latency when the appropriate technique is used (see the benefits of STR-Tree over MaskLink for the specific link discovery task evaluated in this report).
- The Kafka-based intercommunication has enabled efficient access to the output of components, without imposing significant overhead in the overall latency.
- In terms of throughput, all online components demonstrate scalability when provided with more resources. The Synopses Generator and RDFgen can exploit the provisioned resources better, and demonstrate high scalability, as shown by the speedup measurements.
- The Link Discovery component also demonstrates high throughput when provided with more resources, however its speedup is lower than the other two components, which is due

to the complex join operation that needs to be performed to identify links between data sets.

- In the tested setups, the integrated prototype achieves throughput of more than 3,000 messages/sec, when each component is provisioned with 20 workers (parallelism).
- The bottleneck in the integrated prototype is the Link Discovery component, which is natural as it performs the most complex operation, combining different data sets and computing complex geometrical relations. However, acceptable throughput values (more than 3,000 messages/sec) are achieved with reasonable level of parallelism.

5 Evaluation Results: Offline Part

In this section, we present the results from the offline part of the architecture. This is briefly termed as *DiStRDF* and has been presented in [2]. The main component is the datAcron distributed RDF engine, outlined in Section 2.4, therefore we turn our attention to its performance and scalability. Data analytics components are served with data from the distributed RDF engine, in order to perform their task. Our algorithms are implemented using Scala 2.11 and Apache Spark 2.1.

Parameter	**Values**
Spatio-temporal range sizes	10%, 20%, 40% (of the data set spatio-temporal size)
Queries	Maritime: (Q1, Q2, Q3), Aviation: (Q4, Q5, Q6)
Physical plans	**Sort-Merge Join**, Broadcast Hash Join
Number of Spark executors	2, 4, **9**

Table 3: Experimental setup parameters (default values in bold).

5.1 Experimental Setup

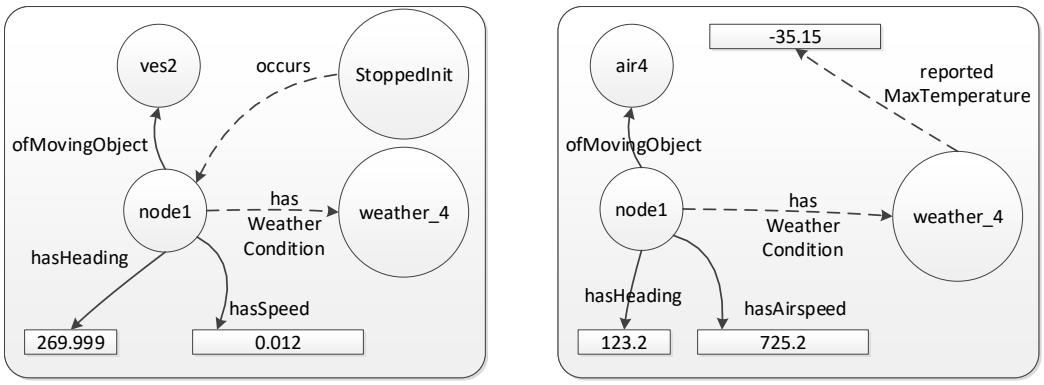
Data sets We used surveillance and static information from the maritime and aviation domains. The maritime surveillance data concern the entire month of January 2016, covering the Mediterranean Sea and part of the Atlantic Ocean. The aviation surveillance data refer to a week of April 2016, covering the entire space of Europe. We use the datAcron ontology described in [10] to represent all data in RDF format.

The total size of the data set is roughly 400 million triples: 300 million triples from the maritime and 100 milliton triples from the aviation domain. These triples were encoded to integer values using the method described in [12] to form the one-triples table, which is approximately 9GB in text format or 4GB in Parquet format using snappy compression. We also built property tables for the *Semantic Node* and *Vessel* entities of datAcron Ontology to enable efficient access to their corresponding properties during query execution. In this section, we experiment with data stored in HDFS using Parquet file format, to enable efficient access for our Spark applications and benefit by columnar storage, compression and predicate push-down.

A dictionary containing the mapping between encoded and decoded values was also created and stored in a Redis cluster instance, running on our cluster, with no replication enabled. The total number of records (key-value pairs) stored in the Redis dictionary is approximately 106 million.

Configuration We configured Spark on YARN, using Hadoop 2.7.2. One node was set to be the driver node, while the others contain the Spark Executors. All experiments conducted, use executors with 5 CPU cores and 8GB memory. We experiment with different number of executors, in order to test scalability. Providing a higher number of executors corresponds

to higher resource provision and should increase the parallelism of the application. HDFS is configured with replication factor of 3. We also used the Jedis [5] library, to communicate with the Redis cluster.



(a) RDF graph for maritime domain.

(b) RDF graph for aviation domain.

Figure 14: RDF graphs for queries on maritime and aviation domains.

Type of queries We conducted experiments using 6 real-world SPARQL queries from both aviation and maritime domains. Queries 1 and 4 are comprised of three triple patterns each and require no join operation to be evaluated, since their triple patterns are already joined in the property tables formatted data source. Queries 2 and 5, are composed of four triple patterns each, requiring a single join operation to be actually evaluated. The rest queries 3 and 6, contain five triple patterns, and require 2 join operations to be evaluated during query execution. All queries are presented in SPARQL format in Appendix B.

Figure 14 shows the parts of the RDF graphs which correspond to the evaluated queries. The left graph (Figure 14a), depicts the RDF graph for the first three queries, which refer to the maritime domain, while the right part (Figure 14b), demonstrates the RDF graph for queries 4 to 6, on the aviation domain. For convenience, the relationships between RDF resources which are pre-computed in the property tables, are depicted as solid lines. Hence, the dashed lines represent the join operations, which should be computed during execution time. For example, the property $hasWeatherCondition$ is stored in the leftover triples data source, while the $hasHeading$ property is stored in the property table; the first property requires a join to compute its relationship with $node1$, while for the second property, the join is pre-computed in the data source. Interestingly although both queries 3 and 6 require two join operations each, they actually perform different types of queries: Query 3 performs two star join operations, while query 6 performs chain join operations.

Furthermore, we added a spatio-temporal range constraint to all of our experiments. We use various ranges of different sizes, based on the volume of total space and time that they cover, namely 10%, 20% and 40%. Put differently, each query has an associated spatio-temporal filter that is applied on the data set, and (for instance) 10% means that the size of this filter in space and time is equal to one tenth of the data set's spatial and temporal extent.

[5] https://github.com/xetorthio/jedis

Metrics Our main evaluation metric is the execution time needed for each SPARQL query to be processed on the Spark cluster. We focus on measuring the actual execution time needed for our queries to be evaluated, thus omitting (a) any overhead caused by Spark initialization processes and (b) the time needed to store the result set in HDFS. In technical terms, we measure only the time needed to calculate the result set in main memory, by executing a call to the count method of the Spark DataFrame containing this result. Moreover, we ran each experiment 10 times, as a warm-up procedure, and report only the time needed for the 11th execution. This warm-up procedure, ensures that the cost of the Java JIT compiler and the overhead for establishing connections to the Redis cluster from all YARN containers is also omitted.

Methodology All of our experiments are executed on both maritime and aviation domains. Furthermore, for each experiment, we also vary the size of the spatio-temporal range constraint. We evaluate the scalability of DiStRDF, by varying the number of Spark executors. Lastly we experiment with Sort-merge Join and Broadcast Hash Join physical operators. Our experimental setup is summarized in Table 3, which marks default values in bold.

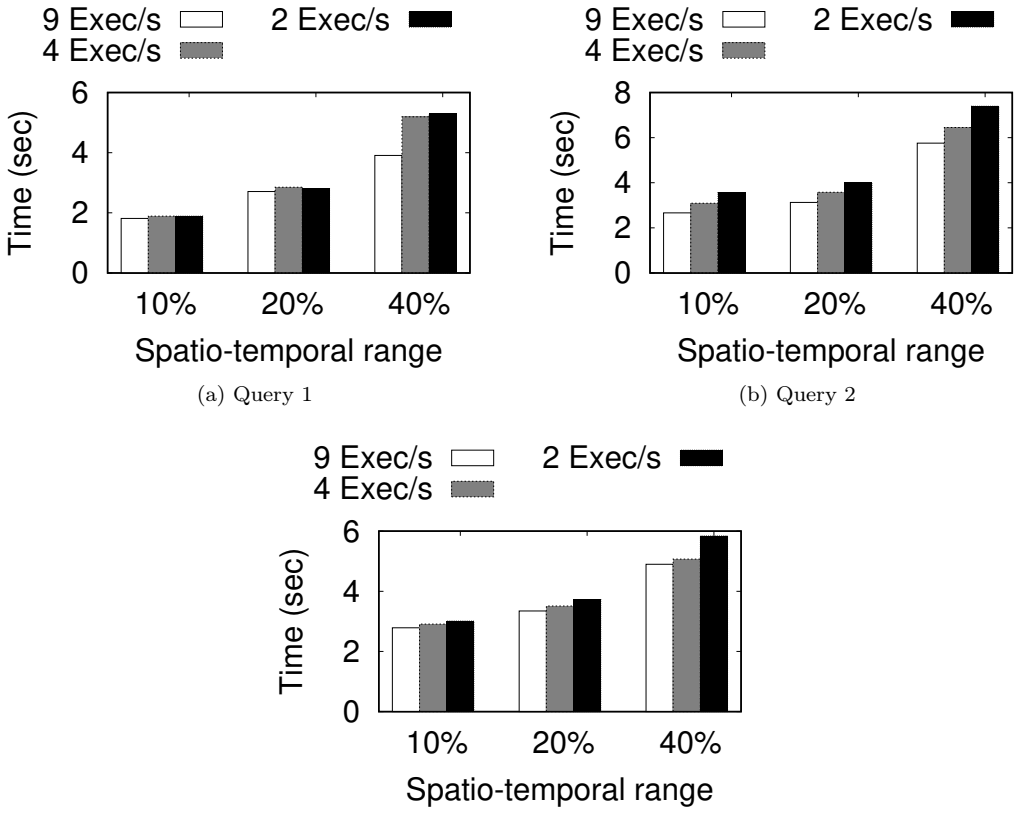
5.2 Results

Comparing the Performance by Varying the Number of Executors: Figures 15 and 16 demonstrate the execution times needed to evaluate queries from maritime and aviation domains respectively, by varying the number of employed Spark Executors. Essentially, this set of experiments is an indication of the scalability of our approach, since they demonstrate the case of having fewer nodes to evaluate the result set.

By using a spatio-temporal range query, Spark performs data processing only on the subset of executors which contain data satisfying that constraint. Since our data is partitioned by their spatio-temporal position, a range query might be satisfied by using only a small set of executors. As such, the case of the 10% range size query, clearly benefits by this partitioning scheme, since in Figures 15a,15b,15c its performance is not affected by the number of executors. However, in aviation queries (Figures 16a,16b,16c), the 10% range size query benefits slightly less by the partitioning scheme, since more nodes are involved to the query evaluation, especially in Queries 5 and 6. The results in the rest of this set of experiments, perform as expected: more executors lead to higher efficiency, especially for the 40% range size queries, which process the larger volume of data. Hence, our DiStRDF system is able to efficiently process higher volume of data, by providing more executors to the cluster.

Comparing the Performance of Physical Join Operators: Figures 17 and 18 demonstrate the impact on query evaluation performance for maritime and aviation queries respectively, by selecting different physical join operators. Queries 1 and 4, do not perform a join operation, thus are excluded from this set of experiments.

Generally, the Sort-merge join operator performs better than the Broadcast join operator. These results can be justified by the fact that a broadcast hash join performs better for small sets of input data, whereas the queries of our experiments provide large sets of data to the join operators. Sort-merge operator also benefits by the the executor's shared memory between executor cores. By exploiting the executor's shared memory, Spark is able to exchange data between executor cores without transferring them over the network. Since we used 5 CPU cores per executor, our experiments had plenty of data being exchanged locally on the executor's



(a) Query 1

(b) Query 2

(c) Query 3

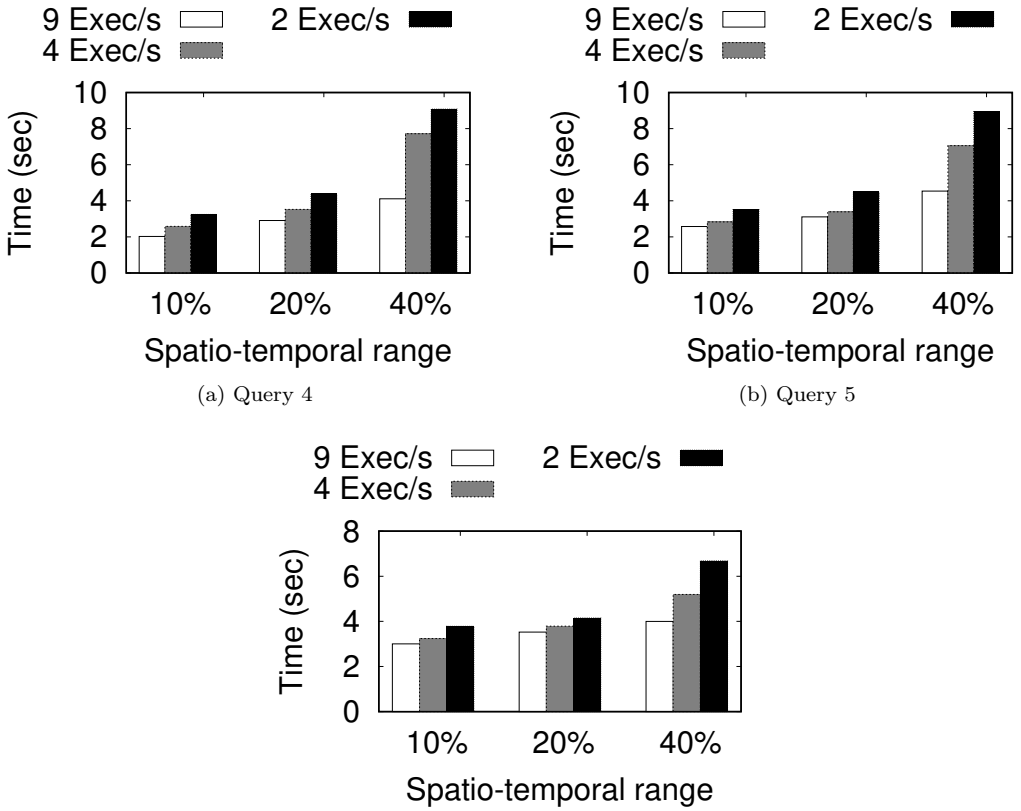
Figure 15: Performance of *DiStRDF* when varying the number of Spark Executors on maritime data.

shared memory. It is also worth noting that Sort-merge Join algorithm, as implemented by Spark SQL API, performs a re-partitioning of the entire data set, to a user configurable number of partitions. This number was set to be 20 during this set experiments.

As expected, the experiments having 40% range size, perform worse than the others, since the approximate filtering is able to prune less data early. This leads to more data being forwarded to the parent join operators, thus increasing their computational cost, especially for the case of Broadcast Hash Join operators. Also, the impact of selecting a physical join operator is less significant in Figures 17a and 18a, when compared to Figures 17b and 18b where only a single join is computed during query execution.

5.3 Discussion of Results

In this section, we summarize and discuss the obtained results from the offline part of the architecture in the context of the datAcron objectives:



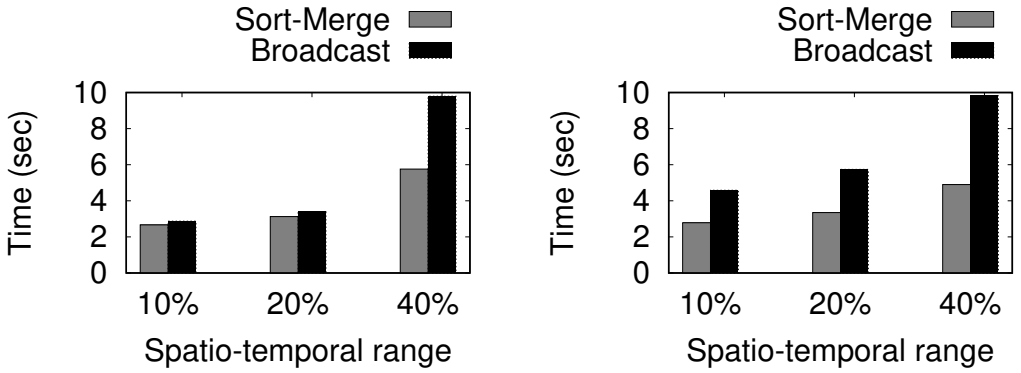
(a) Query 4

(b) Query 5

(c) Query 6

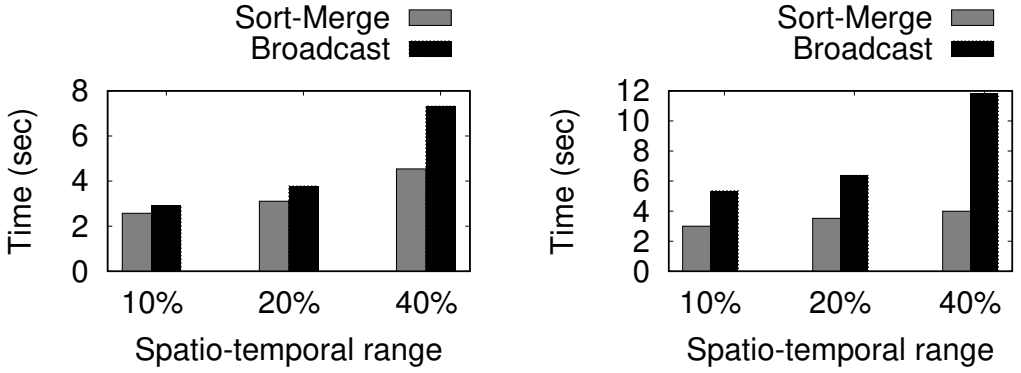
Figure 16: Performance of *DiStRDF* when varying the number of Spark Executors on aviation data.

- We tested the performance and scalability of the distributed datAcron RDF engine using real-life queries over data in the order of 0.5 billion triples and returning integrated data.
- All queries were processed in few seconds, thus achieving the objective of tactical latency, as specified in deliverable D1.1.
- Scalability has been studied under three aspects: (a) when the number of executors is increased, which reflects the parallelism and the efficiency of the engine, (b) when using queries of increased complexity, and (c) when queries with larger output size are used, which reflects the scalability with query result size.
  - The first important finding is that we achieve higher performance, in terms of lower execution time, by the provision of more executors in Spark. This shows that the engine is indeed scalable when provided with more resources.
  - The second observation is that the complexity of the query affects the execution time significantly, since very complex queries that require many join operations over distributed data clearly require more processing time.



(a) Query 2 (b) Query 3

Figure 17: Performance of *DiStRDF* when varying the physical join operator on maritime data.



(a) Query 5 (b) Query 6

Figure 18: Performance of *DiStRDF* when varying the physical join operator on aviation data.

  - Another significant factor affecting performance is the size of data that needs to be accessed for query processing, which is also reflected in the query output size. The respective experiment shows that the distributed datAcron RDF engine scales well with increased query output size.

- The distributed datAcron RDF engine achieves the objective of efficient distributed management and querying of integrated spatio-temporal data.

6 Concluding Remarks

This report has the objective to serve as the evaluation of the datAcron integrated prototype, focusing on the two parts of the architecture: online (stream processing) and offline (batch processing).

For the online part, we have presented the evaluation methodology, which relies on the use of Apache Kafka as message bus for the intercommunication of the datAcron components. We have focused on the fundamental processing flow of online processing in datAcron, namely trajectory compression, transformation to RDF, and link discovery, as it "feeds" all other prediction/forecasting components with an appropriately enriched stream of data. The experiments have shown that all components achieve operational latency, under appropriate parameterization. Also, when the entire chain of components is evaluated, and given the available resources in the datAcron cluster, we observe a throughput value of more than 3,000 messages/sec.

For the offline part, we demonstrated the efficiency and scalability of the distributed datAcron RDF engine: (a) when the number of executors is increased, which reflects the parallelism and the efficiency of the engine, (b) when using different, complex, real-life queries from the maritime and ATM domain, and (c) when queries with larger output size are used, which reflects the scalability with query result size. For all queries the execution time was in the order of a few seconds, thus matching the requirements set in deliverable D1.1.

As a final note, this report provides empirical evidence regarding the performance of the integrated prototype, not only by means of the experimental results presented in this document, but also by the published papers in well-established international conferences and workshops (such as [2, 4, 6, 8, 9, 10, 12]) and journals of high impact (such as [3, 7]).

References

[1] Scott T. Leutenegger, J. M. Edgington, and Mario A. López. STR: A simple and efficient algorithm for r-tree packing. In *Proceedings of the Thirteenth International Conference on Data Engineering, April 7-11, 1997, Birmingham, UK*, pages 497–506, 1997.

[2] Panagiotis Nikitopoulos, Akrivi Vlachou, Christos Doulkeridis, and George A. Vouros. Distrdf: Distributed spatio-temporal RDF queries on spark. In *Proceedings of the Workshops of the EDBT/ICDT 2018 Joint Conference (EDBT/ICDT 2018), Vienna, Austria, March 26, 2018.*, pages 125–132, 2018.

[3] Kostas Patroumpas, Elias Alevizos, Alexander Artikis, Marios Vodas, Nikos Pelekis, and Yannis Theodoridis. Online event recognition from moving vessel trajectories. *GeoInformatica*, 21(2):389–427, 2017.

[4] Kostas Patroumpas, Nikos Pelekis, and Yannis Theodoridis. On-the-fly mobility event detection over aircraft trajectories. In *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, SIGSPATIAL 2018, Seattle, WA, USA, November 06-09, 2018*, pages 259–268, 2018.

[5] Georgios M. Santipantakis, Christos Doulkeridis, George A. Vouros, and Akrivi Vlachou. Masklink: Efficient link discovery for spatial relations via masking areas. *CoRR*, abs/1803.01135, 2018.

[6] Georgios M. Santipantakis, Apostolos Glenis, Nikolaos Kalaitzian, Akrivi Vlachou, Christos Doulkeridis, and George A. Vouros. FAIMUSS: flexible data transformation to RDF from multiple streaming sources. In *Proceedings of the 21th International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018.*, pages 662–665, 2018.

[7] Georgios M. Santipantakis, Apostolos Glenis, Kostas Patroumpas, Akrivi Vlachou, Christos Doulkeridis, George Vouros, Nikos Pelekis, and Yannis Theodoridis. Spartan: Semantic integration of big spatio-temporal data from streaming and archival sources. *Future Generation Computer Systems*, to appear, In Press.

[8] Georgios M. Santipantakis, Konstantinos I. Kotis, George A. Vouros, and Christos Doulkeridis. Rdf-gen: Generating RDF from streaming and archival data. In *Proceedings of the 8th International Conference on Web Intelligence, Mining and Semantics, WIMS 2018, Novi Sad, Serbia, June 25-27, 2018*, pages 28:1–28:10, 2018.

[9] Georgios M. Santipantakis, Akrivi Vlachou, Christos Doulkeridis, Alexander Artikis, Ioannis Kontopoulos, and George A. Vouros. A stream reasoning system for maritime monitoring. In *25th International Symposium on Temporal Representation and Reasoning, TIME 2018, Warsaw, Poland, October 15-17, 2018*, pages 20:1–20:17, 2018.

[10] Georgios M. Santipantakis, George A. Vouros, Christos Doulkeridis, Akrivi Vlachou, Gennady L. Andrienko, Natalia V. Andrienko, Georg Fuchs, Jose Manuel Cordero Garcia, and Miguel Garcia Martinez. Specification of semantic trajectories supporting data transformations for analytics: The datacron ontology. In *Proceedings of the 13th International Conference on Semantic Systems, SEMANTICS 2017, Amsterdam, The Netherlands, September 11-14, 2017*, pages 17–24, 2017.

[11] Georgios M. Santipantakis, George A. Vouros, Apostolos Glenis, Christos Doulkeridis, and Akrivi Vlachou. The datacron ontology for semantic trajectories. In *The Semantic Web: ESWC 2017 Satellite Events - ESWC 2017 Satellite Events, Portorož, Slovenia, May 28 - June 1, 2017, Revised Selected Papers*, pages 26–30, 2017.

[12] Akrivi Vlachou, Christos Doulkeridis, Apostolos Glenis, Georgios M. Santipantakis, and George A. Vouros. Efficient spatio-temporal rdf query processing in large dynamic knowledge bases. In *Proceedings of the 34th Annual ACM Symposium on Applied Computing, SAC 2019 (to appear)*, 2019.

A Example of Kafka Interceptors Usage

In this section, we provide a simple example for using Kafka Interceptors in the datAcron components. As a showcase example, we present the Synopses Generator (SG) has been modified to work with Kafka Interceptors.

A.1 Trajectory synopses usecase

In order to incorporate Kafka Interceptors into the trajectory synopses tool in a reusable way we had to do the following:

1. Create a Gradle project containing the Kafka interceptors.
2. Port the trajectory synopses build tool from Maven to Gradle
3. Include the Kafka interceptors project into the Trajectory synopses tool

A.1.1 Create a Gradle project containing the Kafka interceptors

We have created a sample Gradle project containing the Kafka Interceptors to measure latency for the various data types in datacron named `DatacronKafkaUtils`

A.1.2 Include the Kafka interceptors into the Trajectory synopses tool

The settings.gradle file

```
rootProject.name = 'trajectory_synopses_uprc'

include ":rdf_lib"
project(":rdf_lib").projectDir = file("../rdf_lib")

include ":DatacronKafkaUtils"
project(":DatacronKafkaUtils").projectDir = file("../DatacronKafkaUtils")
```

The build.gradle file

```
// Apply the scala plugin to add support for Scala
buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'com.github.jengelman.gradle.plugins:shadow:2.0.2'
    }
}

apply plugin: 'scala'
```

```
apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'application'
apply plugin: 'com.github.johnrengelman.shadow'

allprojects {
    ext {
        javaVersion = '1.8'
        kafkaVersion = '0.10.1.0-cp2'
        kafkaVersion2 = '0.8.2.0'
        kafkaScalaVersion = '2.11'
        scalaVersion = kafkaScalaVersion + '.7'
        confluentVersion = '3.1.1'
        scalatestVersion = '2.2.6'
        avroVersion = '1.8.2'
        jettyVersion = '9.2.12.v20150709'
        jerseyVersion = '2.19'
        jedisClientVersion = "2.9.0"
        flinkVersion = "0.10.2"
        jacksonVersion = "2.4.4"
        flinkKafkaVersion = "0.10.1"

    }
    // Apply the java plugin to add support for Java


    // In this section you declare where to find the dependencies of your project
    repositories {
        // Use 'jcenter' for resolving your dependencies.
        // You can declare any Maven/Ivy/file repository here.

        maven {
            url  "http://packages.confluent.io/maven/"
        }

        jcenter()
        mavenCentral()
        mavenLocal()

    }
    configurations.all {
        resolutionStrategy {
            // fail eagerly on version conflict (includes transitive dependencies)
            // e.g. multiple different versions of the same dependency (group and name are equal)
            //failOnVersionConflict()
            force('org.apache.kafka:kafka_2.11:0.8.2.0')

        }
    }
```

```
}



dependencies {
    // Use Scala 2.11 in our library project

    compile 'org.scala-lang:scala-library:2.11.7'
    //compile 'org.apache.flink:flink-shaded-hadoop1'
    //compile 'org.apache.flink:flink-shaded-hadoop2'
    //compile 'org.apache.flink:flink-shaded-curator-recipes'
    //compile 'org.apache.flink:flink-core:$flinkVersion'
    //compile 'org.apache.flink:flink-java:$flinkVersion'
    compile group: 'org.apache.flink', name: 'flink-scala_2.11', version: flinkVersion
    compile group: 'org.apache.flink', name: 'flink-runtime_2.11', version: flinkVersion
    compile group: 'org.apache.flink', name: 'flink-optimizer_2.11', version: flinkVersion
    compile group: 'org.apache.flink',name:'flink-clients_2.11', version: flinkVersion
    compile group:'org.apache.avro', name: 'avro' , version: avroVersion
    compile group:'org.apache.flink', name: 'flink-streaming-java_2.11', version: flinkVersion
    compile group:'org.apache.flink', name: 'flink-streaming-scala_2.11', version: flinkVersion
    compile group: 'org.apache.flink', name:'flink-connector-kafka_2.11', version: flinkKafkaVers
    compile group: 'org.apache.kafka', name: 'kafka_' + kafkaScalaVersion , version: kafkaVersion
    compile group: 'org.apache.kafka', name: 'kafka-clients' , version: kafkaVersion2
    compile project(":rdf_lib")
    compile project(":DatacronKafkaUtils")

    // Use Scalatest for testing our library

    // Need scala-xml at test runtime

}
```

A.1.3 Instruct Flink to use the Kafka interceptors

In order for Flink ,or any other framework for that matter, to use the Kafka Interceptors we have to insert the following into the Properties of the consumer and the producer respectively:

```
kafkaConsumerProperties
.setProperty(
ConsumerConfig.INTERCEPTOR_CLASSES_CONFIG,
  classOf[JSONConsumerInterceptor].getName)
val kafkaConsumer_Messages = new FlinkKafkaConsumer[String](
      config.TOPIC_MESSAGES,
      new SimpleStringSchema(),
      // Read input messages as CSV strings; currently, NOT pertaining to
      // the AVRO schema of critical points
      kafkaConsumerProperties,
      OffsetStore.FLINK_ZOOKEEPER,
      FetcherType.LEGACY_LOW_LEVEL
    )
```

```
props.put(
  org.apache.kafka.clients.producer.
  ProducerConfig.INTERCEPTOR_CLASSES_CONFIG,
   classOf[CriticalPointProducerInterceptor].getName)
val configNotifications = new kafka.producer.ProducerConfig(props)
val kafkaProducer_notifications = new Producer[String, String](configNotifications)
```

B SPARQL Queries Used In Experiments

SPARQL Query 1 (Maritime Domain)

```
Prefix : <http://www.datacron-project.eu/datAcron#>
SELECT * WHERE {
    ?n :ofMovingObject ?ves ;
    :hasHeading ?heading ;
    :hasSpeed ?speed .
}
```

SPARQL Query 2 (Maritime Domain)

```
Prefix : <http://www.datacron-project.eu/datAcron#>
SELECT * WHERE {
    ?n :ofMovingObject ?ves ;
    :hasHeading ?heading ;
    :hasSpeed ?speed ;
    :hasWeatherCondition ?w .
}
```

SPARQL Query 3 (Maritime Domain)

```
Prefix : <http://www.datacron-project.eu/datAcron#>
SELECT * WHERE {
    ?n :ofMovingObject ?ves ;
    :hasHeading ?heading ;
    :hasSpeed ?speed ;
    :hasWeatherCondition ?w .
    :StoppedInit :occurs ?n .
}
```

SPARQL Query 4 (Aviation Domain)

```
Prefix : <http://www.datacron-project.eu/datAcron#>
SELECT * WHERE {
    ?n :ofMovingObject ?aircraft ;
    :hasHeading ?heading ;
    :hasAirspeed ?speed .
}
```

SPARQL Query 5 (Aviation Domain)

```
Prefix : <http://www.datacron-project.eu/datAcron#>
SELECT * WHERE {
    ?n :ofMovingObject ?aircraft ;
    :hasHeading ?heading ;
    :hasAirspeed ?speed ;
    :hasWeatherCondition ?w .
}
```

SPARQL Query 6 (Aviation Domain)

```
Prefix : <http://www.datacron-project.eu/datAcron#>
SELECT * WHERE {
    ?n :ofMovingObject ?aircraft ;
    :hasHeading ?heading ;
    :hasAirspeed ?speed ;
    :hasWeatherCondition ?w .
    ?w :reportedMaxTemperature ?temp .
}
```