Grant Agreement No: 687591

30/12/16

Big Data Analytics for Time Critical Mobility Forecasting

datAcron

D1.2 Architecture Specification

Deliverable Form	
Project Reference No.	H2020-ICT-2015 687591
Deliverable No.	1.2
Relevant Work Package:	WP 1
Nature:	R
Dissemination Level:	PU
Document version:	2.1
Due Date:	31/12/2016
Date of latest revision:	30/12/2016
Completion Date:	30/12/2016
Lead partner:	UPRC
Authors:	Christos Doulkeridis (UPRC), Apostolos Glenis (UPRC), Giorgos Santipantakis (UPRC), Akrivi Vlachou (UPRC), George Vouros (UPRC), Michael Mock (FRHF), Nikos Pelekis (UPRC), Kostas Patroumpas (UPRC), Elias Alevizos (NCSR'D), Georg Fuchs (FRHF)
Reviewers:	Nikos Pelekis (UPRC), Elias Alevizos (NCSR'D), Alexander Artikis (NCSR'D), Georg Fuchs (FRHF), Christophe Claramunt (NARI), Cyril Ray (NARI), Anne-Laure Jousselme (CMRE), Elena Camossi (CMRE), Ernie Batty (IMIS), David Scarlatti (BRTE), Jose-Manuel Cordero (CRIDA)
Document description:	This deliverable specifies the integrated system architecture for datAcron.
Document location:	WP1/Deliverables/D1.2/Final

HISTORY OF CHANGES

Version	Date	Changes	Author	Remarks
0.1	26/9/2016	First version of table of contents	C. Doulkeridis	
0.2	15/10/2016	Added overview of technological solutions	A. Glenis	
0.3	15/11/2016	First version of architecture	A. Glenis	
0.4	28/11/2016	Refined version of architecture	C. Doulkeridis	
0.5	30/11/2016	Revised section on technological solutions	A. Glenis	
1.0	01/12/2016	Added description of components and software architecture	C. Doulkeridis	Version shared with all partners
1.1	07/12/2016	Added feedback from A. Artikis and E. Alevizos (WP3) and D. Scarlatti (WP6)	C. Doulkeridis	
1.2	09/12/2016	Added mapping of architecture to requirements	C. Doulkeridis	
1.3	15/12/2016	Added feedback from N. Pelekis (WP2)	C. Doulkeridis	
1.4	17/12/2016	Refined overview of technological solutions	A. Glenis	
1.5	19/12/2016	Added feedback from G. Fuchs (WP4)	C. Doulkeridis	
1.6	22/12/2016	Added feedback from M. Mock about in-situ processing	C. Doulkeridis	
2.0	23/12/2016	Near-final, homogeneous version	C. Doulkeridis	Version shared with all partners

EXECUTIVE SUMMARY

This report comprises the second deliverable (D1.2) of datAcron work package 1 "System architecture and data management" with main objective to specify the architecture of datAcron, its constituent components, and their interactions, in order to realize the requirements specified in deliverable D1.1.

The first part of this document provides a brief overview of technological solutions that are relevant to the innovative Big Data algorithms and methods that are being developed in datAcron. Its purpose is to narrow down the possible frameworks that can be adopted by datAcron, detect their advantages and disadvantages, and ultimately provide a guide for selecting the most appropriate technological solution for particular parts of the datAcron architecture.

The second part of this document describes the proposed datAcron system architecture, the main datAcron components, their interactions, as well as the internals of each component, thereby clarifying the roles and the specific operations performed in datAcron. In addition, a mapping is provided between the requirements identified in deliverable D1.1 and the datAcron architecture proposed in this deliverable, aiming at justifying the choice of the individual components and how the requirements are addressed by the architecture.

The final part of this document presents the current view of the datAcron software architecture, in terms of high level software modules and their interconnections. This part is going to be refined in the following months of the project's lifetime, and will eventually be documented in deliverables D1.6 and D1.11 about Software Design. However, the usefulness of this part in the present deliverable is to identify novel algorithmic solutions that will be developed in the context of datAcron, as well as contributions in the areas of Big Data processing and analytics.

TABLE OF CONTENTS

TERMS & ABBREVIATIONS

ADS-B	Automatic Dependent Surveillance-Broadcast
AIS	Automatic Identification System
METAR	Meteorological Terminal Air Report
NOAA	National Oceanic and Atmospheric Administration
RDF	Resource Description Framework
STAR	Standard Terminal Arrival Route
SID	Standard Instrument Departure

LIST OF FIGURES

LIST OF TABLES

1 Introduction

This document is the deliverable D1.2 "Architecture Specification" of Task T1.2 of work package 1 "System Architecture and Data Management" of the datAcron project. It defines the datAcron integrated system architecture, by capitalizing on the deliverable D1.1. "Requirements Analysis" but also on the deliverables D5.2 "Maritime data preparation and curation" and D6.2 "Aviation data preparation and curation". The specified architecture is subject to refinements during the course of the project, according to other deliverables of the individual work packages that are going to be prepared after month M12 in the project.

1.1 Purpose and Scope

The Architecture Specification aims at defining the overall architecture of the project, in terms of main components, their interactions, but also the software modules that are going to implement the architecture. The envisioned architecture is a Big Data architecture, as it addresses several aspects relevant to Big Data, such as: processing of multiple streaming data *(Velocity)*, handling vast volumes of archival data and incoming streaming data *(Volume)*, and integrating heterogeneous data sources in different formats, representations, and modalities *(Variety)*.

Deliverable D1.2 is submitted on month M12 of the project, in order to set the guidelines for the next development steps in the project. Even though it is foreseen that individual components may be developed in different frameworks, in order to achieve optimized operation, their interactions are specified in this document, in order to ease the necessary software integration of the prototype datAcron system.

1.2 Approach for the Work package and Relation to other Deliverables

D1.2 relies on the analysis of requirements specified in deliverable D1.1. In this sense, it validates the requirements and proposes concrete architectural components that constitute a comprehensive architecture for the datAcron system. In addition, D1.2 exploits the outcome of the data preparation reports for the maritime (D5.2) and aviation (D6.2) domains, which provide details on input data sources for the datAcron system in terms of historical data sources and streaming data sources.

It must be pointed out that D1.2 is prepared during the same time that deliverables D5.3 "Maritime experiments specification" and D6.3 "Aviation experiments specification" are prepared. Both these deliverables aim at providing more detailed descriptions of the experimentation process that is going to validate the research results of the project. As such, they connect to this deliverable, and intermediate versions of D5.3 and D6.3 have been considered during the preparation of the current deliverable.

1.3 Methodology and Structure of the Deliverable

Main inputs to the preparation of this document were both deliverable D1.1 as well as internal reports specifying how specific use-case scenarios are expected to be supported in datAcron. In terms of work methodology, the present deliverable builds upon the requirements identified in D1.1, and "translates" or "maps" these requirements to specific architectural components, whose interactions support the use-case scenarios of datAcron.

The objective of the adopted work methodology is twofold: (a) to justify the architectural decisions, and (b) to verify that all identified requirements are addressed by the architecture. Furthermore, the proposed architecture is in compliance with the research objectives of the project, addressing diverse processing and analytics requirements (both batch and real-time processing), and tackling with multiple aspects of Big Data. In this respect, it sets a solid ground for developing innovative algorithmic solutions for Big Data processing and analysis in the context of each individual part of the architecture.

To come up with an comprehensive architecture, the main rationale behind the production of this report was to identify specific *architectural components* that are going to perform well-defined operations at individual level, but also interact with each other in order to deliver optimized analysis results, such as prediction and forecasting over Big Data, which is the core research objective of datAcron. Going one step further, this report specifies the internal modules and organization of each component, in order to provide a coherent view of the processing and analysis tasks at all levels, from data acquisition and cleaning, to data representation and processing, and finally to data analysis and result interpretation.

The remaining of this report is structured as follows:

- Section 2 overviews the state-of-the-art in terms of technological solutions that are relevant for the datAcron architecture.
- Section 3 describes the architecture of datAcron in terms of main components, inputs and outputs, as well as their interactions.
- Section 4 provides the mapping of the proposed architecture and its constituent components to the requirements (stated in deliverable D1.1), thus validating the design of the architecture.
- Section 5 describes the software architecture that will be realized in order to achieve the research objectives of datAcron.

2 Overview of Technological Solutions

In this section we provide a crisp overview of available technological solutions related to Big Data management. The focus is on real-world, operational systems that are stable, reliable, have a significant user and developer base, and are being used in the implementation of novel Big Data systems and algorithms, rather on research prototypes that are optimized for specific operations only. In the context of datAcron, the technical challenges raised are due to different aspects of Big Data, and relate to the following areas:

- *Stream-based interconnection* for end-to-end real-time operations that rely on the aggregation of different modules providing distinct functionalities (Section 2.1).
- *Scalable, distributed and integrated storage* of voluminous archival data (data-at-rest) and new streaming data (data-in-motion) (Section 2.2).
- *Batch processing* functionality on top of the stored data to enable different data analytics on top of the stored data (Section 2.3).
- *Stream processing* for real-time operations, such as trajectory prediction and complex event detection and forecasting (Section 2.4).
- In order to answer queries that require *both streaming and historical information*, data from the streaming layer need to be integrated with historical information in real-time (Section 2.5).
- *Data serialization formats* which facilitate flexible communication between different modules (Section 2.6).

In the following, a brief review of the most prevalent technologies for the above areas is provided.

2.1 Message Bus

2.1.1 Apache Kafka

Kafka[1] is a distributed, partitioned, replicated commit log service [23]. It provides the functionality of a messaging system.

1. Kafka maintains feeds of messages in categories called *Topics*.
2. Each Topic is partitioned for scalability and *Partitions* are distributed in the cluster.
3. Processes that publish messages to a Kafka Topic are called *Producers*.
4. Processes that subscribe to Topics and process the feed of published messages are called *Consumers*.
5. Kafka runs in a *Cluster* comprised of one or more servers each of which is called a *Broker*.

[1] http://kafka.apache.org/

Messaging traditionally has two models: queuing and publish-subscribe. In a queue, a pool of consumers may read from a server and each message goes to one of them; in publish-subscribe the message is broadcast to all consumers. Kafka offers a single consumer abstraction that generalizes both of these, the *Consumer Group*.

At a high level, Producers send messages over the network to the Kafka cluster, which in turn serves them up to Consumers. A Topic is a category or feed name to which messages are published. For each Topic, the Kafka cluster maintains a partitioned log. The Kafka cluster retains all published messages, whether or not they have been consumed, for a configurable period of time.

Producers publish data to the Topics of their choice. The Producer is responsible for choosing which message to assign to which partition within the Topic. This can be done in a round-robin fashion simply to balance load or according to some custom partition function (say based on the key of the message).

In more detail, each partition is an ordered, immutable sequence of messages that is continually appended to a commit log. The messages in the partitions are each assigned a sequential id number called the *Offset* that uniquely identifies each message within the partition. Partitions act as the unit of parallelism and allow a Topic to be stored on more than one server. The Partitions of the log are distributed over the servers in the Kafka cluster, with each server handling data and requests for a share of the partitions. The partitioning function can be a custom partitioning scheme to optimally distribute the partitions according to the application requirements. Each partition is replicated across a configurable number of servers for fault-tolerance.

2.2 Data Storage

2.2.1 The Hadoop Distributed File System (HDFS)

The Hadoop Distributed File System (HDFS)[2] is a distributed file system designed to run on commodity hardware. It has many similarities with existing distributed file systems. However, the differences from other distributed file systems are significant. HDFS is highly fault-tolerant and is designed to be deployed on low-cost hardware. HDFS provides high throughput access to application data and is suitable for applications that have large data sets.

HDFS is designed and optimized for storing very large files and with a streaming access pattern. Since it is expected to run on commodity hardware, it is designed to take into account and handle failures on individual machines. HDFS is normally not the primary storage of the data. Rather, in a typical workflow, data is copied over to HDFS for the purpose of performing MapReduce, and the results then copied out from HDFS. Since HDFS is optimized for streaming access of large files, random access to parts of files is significantly more expensive than sequential access, and there is also no support for updating files, only append is possible. The typical scenario of applications using HDFS follows a write-once read-many access model.

HDFS adopts a master/slave architecture. An HDFS cluster consists of a single NameNode, a master server that manages the file system namespace and regulates access to files by clients. In addition, there are a number of DataNodes, usually one per node in the cluster, which manage storage attached to the nodes that they run on. HDFS exposes a file system namespace and allows user data to be stored in files. Internally, a file is split into one or more blocks and these blocks are stored in a set of DataNodes. The NameNode executes file system namespace operations

[2] https://wiki.apache.org/hadoop/HDFS

like opening, closing, and renaming files and directories. It also maintains the mapping of blocks to DataNodes. The DataNodes are responsible for serving read and write requests from the file system's clients. The DataNodes also perform block creation, deletion, and replication upon instruction from the NameNode. HDFS is the de-facto filesystem in the Hadoop environment [24]. It provides replication and fault-tolerance and is compatible with all the serialization frameworks.

Thus the first option to data storage for datAcron is to organize data as flat files appropriately partitioned across the cluster. One benefit of storing flat files on top of HDFS is that the distribution of the files in the cluster can be controlled and the file format can be chosen appropriately for each use-case. One drawback of this solution is that manual rebalancing of the nodes is required for non-static data.

2.2.2 Cassandra

Cassandra[3] is a column-oriented NoSQL database engine. It follows the data-model of Google's Bigtable [8] and has an architecture resembling Amazon's DynamoDB [21] for fault-tolerance. Cassandra follows the following storage model:

- Keyspaces: A keyspace holds Tables.
- Table: A Table consists of Rows.
- Row: A Row is uniquely identifiable by its RowKey that must be unique amongst different Rows. A row consists of Column Families.
- Column Family: A column family consists of several columns.
- Column: Each column has a name and a value.

In Cassandra, rows are partitioned by means of hashing, however customized partitioning is also supported. An in-order partitioner keeps row-keys in sorted order thus has the benefit that it supports efficient Range Queries, but can lead to hot nodes if the data distribution is skewed. Hash-partitioning on the other hand has the advantage that it ensures load-balancing through the random distribution of keys in the cluster, but leads to inefficient processing of range queries.

Column Families (i.e., user-defined groups of columns) are sorted either lexicographically based on their data-type or by a custom comparator. Because of the sorted row and columns Cassandra can provide slicing and filtering on both rows and columns. One common design pattern for Cassandra to avoid disk waste when we have a single value for each column name, is to store a value in the name of the column family. Also, because Cassandra does not support joins, it is common practice to compute and store materialized views. Cassandra offers tunable consistency and is a write-optimized store, since conflict resolution is resolved at read time. This may result in degraded read performance.

2.2.3 Apache Hbase

Apache Hbase[4] is an implementation of Google's Bigtable [8] for the Hadoop ecosystem. The data model is the same as Cassandra's (and Bigtable's) described above. One important distinction is that in Hbase there is no need for the valueless column design. Hbase provides several advanced features such as:

[3] http://cassandra.apache.org/
[4] https://hbase.apache.org/

- *Filters* allow the user to specify the subset of objects from the query result that will be returned to the client. The benefit is that they run on the server-side, thus reducing the amount of data that needs to be transferred. Examples include filtering rows based on key prefixes or some regular expression. Filters can also be combined in a *FilterList* (a set of filters), which is applicable on the result using either AND or OR semantics.
- *Coprocessors* provide even more flexibility than filters in terms of querying. They are essentially user code that can be deployed in the Hbase cluster. Mainly, there are 2 types of coprocessors:
  - *Observers*, where for each base operation, i.e., *put*, the user can deploy custom code in form of hooks that can either pre-process the input parameters or post-process the results.
  - *Endpoints*, that act like functions and can be invoked through Remote Procedure Calls (RPC).

Hbase supports Range Scans and Prefix Range Scans, which is the same as a Range Scan and a PrefixFilter.

2.2.4 Pivotal GemFire and Apache Geode

Pivotal GemFire or its Apache incubating equivalent Apache Geode[5] is an in-memory scalable datastore. Because of its in-memory nature it is a good match for storing data and state in the streaming component of the datAcron architecture. GemFire supports Range Partitioning for efficient range queries and table joins. It also supports co-locating rows from different tables.

2.3 Batch Processing

2.3.1 Apache Hadoop

Hadoop [24] is an open-source implementation of MapReduce. Hadoop consists of two main parts: the Hadoop distributed file system (HDFS) and MapReduce for distributed processing. Hadoop consists of a number of different daemons/servers: NameNode, DataNode, and Secondary NameNode for managing HDFS, and JobTracker and TaskTracker for performing MapReduce.

In Hadoop, the *JobTracker* is the access point for clients. The duty of the JobTracker is to ensure fair and efficient scheduling of incoming MapReduce jobs, and assign the tasks to the *TaskTrackers* which are responsible for execution. A TaskTracker can run a number of tasks depending on available resources (for example two map tasks and two reduce tasks), and will be allocated a new task by the JobTracker when ready. The relatively small size of each task compared to the large number of tasks in total helps to ensure load balancing among the machines. It should be noted that while the number of map tasks to be performed is based on the input size (number of splits), the number of reduce tasks for a particular job is user-specified.

In a large cluster, machine failures are expected to occur frequently, and in order to handle this, regular heartbeat messages are sent from TaskTrackers to the JobTracker periodically, and from the map and reduce tasks to the TaskTracker. In this way, failures can be detected and the JobTracker can reschedule the failed task to another TaskTracker. Hadoop follows a speculative execution model for handling failures. Instead of fixing a failed or slow-running task, it executes a

[5] http://geode.apache.org/

new equivalent task as backup. Failure of the JobTracker itself cannot be handled automatically, but the probability of failure of one particular machine is low so that this should not present a problem in general. Despite its numerous advantages, Hadoop also has significant limitations, as pointed out in [9].

2.3.2 Apache Spark

Spark[6] is an in-memory data processing system [31]. Spark solves most of the inefficiencies of Hadoop, and performs much faster in typical use-cases, as reported also in [20] (among others). Spark also has a rich ecosystem of auxiliary libraries such as GraphX [26] for graph processing and MLlib [17] for machine learning.

The main abstraction in Spark is the Resilient Distributed Datasets (RDDs) [30] that are cached across the memory hierarchy. RDDs are immutable and their operations are lazy; fault-tolerance is achieved by keeping track of the "lineage" of each RDD (the sequence of operations that produced it) so that it can be reconstructed in the case of data loss. RDDs can contain any type of Python, Java, or Scala objects.

Aside from the RDD-oriented functional style of programming, Spark provides two restricted forms of shared variables:

1. *broadcast variables*, reference read-only data that needs to be available on all nodes.
2. *accumulators*, that perform reductions in an imperative style.

Recently Spark introduced DataFrames[7] and its typed variant Datasets[8]. Dataframes allow a higher level abstraction to distributed collections of data by imposing structure to the data. Also, Dataframes use custom memory representation and allow advanced query optimization [10]. Datasets bring extra performance boost to Spark, by collapsing the whole query into a single function[9]. This makes Spark the primary candidate for implementing batch processing functionality nowadays.

2.4 Stream/Real-time Processing

2.4.1 Apache Storm and Heron

Storm[10] is a framework that targets real-time processing and analysis of data streams, with salient features such as scalability, parallelism and fault-tolerance [22]. There are two kinds of nodes on a Storm cluster:

1. the *master* node that runs a daemon called *Nimbus*. Nimbus is responsible for distributing code around the cluster, assigning tasks to machines, and monitoring for failures.
2. the *worker* nodes that run a daemon called the *Supervisor*. The supervisor listens for work assigned to its machine and starts and stops worker processes as necessary based on what Nimbus has assigned to it. Each worker process executes a subset of a *Topology*.

[6] http://spark.apache.org/
[7] https://databricks.com/blog/2015/02/17/introducing-dataframes-in-spark-for-large-scale-data-science.html
[8] https://databricks.com/blog/2016/01/04/introducing-apache-spark-datasets.html
[9] https://databricks.com/blog/2016/05/11/apache-spark-2-0-technical-preview-easier-faster-and-smarter.html
[10] http://storm.apache.org/

A Topology is a graph of computation. Each node in a Topology contains processing logic, and links between nodes indicate how data should be passed around between nodes. A running Topology consists of many worker processes spread across many machines. All coordination between Nimbus and the Supervisors is done through a *Zookeeper* cluster. Additionally, the Nimbus daemon and Supervisor daemons are fail-fast and stateless; all state is kept in Zookeeper. The core abstraction in Storm is the *stream*. A stream is an unbounded sequence of tuples. Storm provides the primitives for transforming a stream into a new stream in a distributed and reliable way. The basic primitives Storm provides for doing stream transformations:

1. *Spouts*: is a source of streams.
2. *Bolts*: consumes any number of input streams, does some processing, and possibly emits new streams.

Spouts and bolts have interfaces that implement application-specific logic.

Networks of spouts and bolts are packaged into a Topology which is the top-level abstraction that you submit to Storm clusters for execution. A Topology is a graph of stream transformations where each node is a spout or bolt. Edges in the graph indicate which bolts are subscribing to which streams. When a spout or bolt emits a tuple to a stream, it sends the tuple to every bolt that subscribed to that stream. Storm provides one-by-one tuple processing with either at least-once or at-most-once delivery guaranties. The *Trident* API, built on top of Storm, can provide a micro-batching abstraction, thus enabling support for efficient windowing, exactly once processing and higher throughput. Heron [13] addresses a lot of the shortcomings of Storm while keeping the same programming model.

2.4.2 Spark Streaming

Spark also provides a streaming processing framework, Spark Streaming [32], that provides a micro-batching abstraction for streaming data and offers a variety of streaming algorithms and approximations. Structured Streaming[11], introduced in Spark 2.0, generalizes the Datasets API for the streaming layer of the application. Using Structured Streaming, developers can build continuous applications without having to reason about transactional updates to the datastore, fault-tolerance, state and state keeping. It also supports event-time processing and joins with a Dataframe from the batch layer of Spark. A drawback is that not all input-output sources are supported at the moment and that the developer is restricted to the Datasets API. Structured streaming would be an appropriate choice for real-time processing tasks that belong to the *tactical* latency[12] in datAcron since it supports merging streaming with batch data out-of-the-box.

2.4.3 Apache Flink

Flink[13] is a streaming dataflow engine that provides data distribution, communication, and fault-tolerance for distributed computations over data streams [7]. It is based on results of the Stratosphere project [4] and also connects to Google's dataflow [3], and supports flexible windowing (time, count, sessions, custom triggers) across different time semantics (event time, processing time).

Flink includes several APIs for creating applications that use the Flink engine:

1. DataStream API for unbounded streams embedded in Java and Scala.

[11] https://databricks.com/blog/2016/07/28/structured-streaming-in-apache-spark.html

[12] Recall the three latency levels relevant for datAcron as defined in deliverable D1.1: *operational* (in milliseconds), *tactical* (in few seconds), and *strategic* (tens of seconds or minutes)

[13] https://flink.apache.org/

2. DataSet API for static data embedded in Java, Scala, and Python.

3. Table API with a SQL-like expression language embedded in Java and Scala.

It also provides libraries for streaming machine learning and graph processing. Flink has processing latency comparable to Storm and it is significantly superior to Spark Streaming especially for small window sizes[14]. Flink is the primary contender for tasks that require sophisticated algorithms and do not require the outmost lower latency.

2.4.4 Kafka Streams

Kafka streams[15] is a Java library implemented on top of Kafka for lightweight streaming processing. Kafka Streams provides two basic abstractions:

- KStream: In a stream, each key-value is an independent piece of information.
- KTable: A table is a changelog. If the table contains a key-value pair for the same key twice, the latter overwrites the mapping.

An important feature of Kafka Streams is stateful stream processors. Each task in Kafka Streams is a combination of a set of Topic's partitions and a topology, can have a local store, that is either in-memory or a persistent key-value data-store (by default this is RocksDB). Even though these stores are local (hence, giving fast access times), they are backed by a Kafka Topic, which contains the changelog for each such store to provide reliability.

It is possible to inner/outer/left join two KStreams, a KStream to a KTable or two KTables. A join between a KStream and a KTable works by storing the current (local) KTable state in the local store, and looking up a value for each incoming stream element. For a join between two KStreams it is mandatory to specify a time window, in which elements from both streams will be matched. For the joins to work, both streams must use same types of keys, as the joins always match on the key values.

It is possible to use different timestamps for windowing operations: event time (defined by whatever creates the event), ingestion time (when the event is stored into Kafka), and processing time (when the event is processed). When aggregating elements in time windows, it is possible to use any of these timestamps (by using a custom or one of the built-in TimestampExtractor)

Kafka Streams is the primary choice for simple tasks that require the lowest latency available.

2.4.5 Comparison

Due to the different options regarding stream processing, as well as the significance of real-time operations in datAcron, we provide a Table 1 that summarizes our findings for stream processing in a comparative way. In this comparison, we focus on four dimensions of interest: windowing functionality, state management, latency, and delivery guarantees.

An additional observation is that, in datAcron, the latency dimension is important, since many operations need to be performed in real-time and also with *operational* latency (in milliseconds). As such, it is expected that the choice of a particular streaming processing framework is going to be influenced by the minimum latency that can be achieved, thus narrowing down the pool of choices of framework for specific time-critical tasks in datAcron.

[14] https://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at
[15] https://www.confluent.io/blog/introducing-kafka-streams-stream-processing-made-simple/

	Storm	Storm + Trident	Flink	Spark Streaming	Kafka Streams
Windowing	time-based, count-based	time-based, count-based	time-based, count-based	time-based	time-based
State Management	record acknowledgment	record acknowledgment	distributed snapshots	checkpoints	local and distributed snapshots
Latency	very low	medium	low (configurable)	medium	very low
Delivery Guaranties	at-least once	exactly-once	exactly once	exactly once	at-least once and there is on going work for exactly once

Table 1: Comparison of the capabilities of streaming processing frameworks

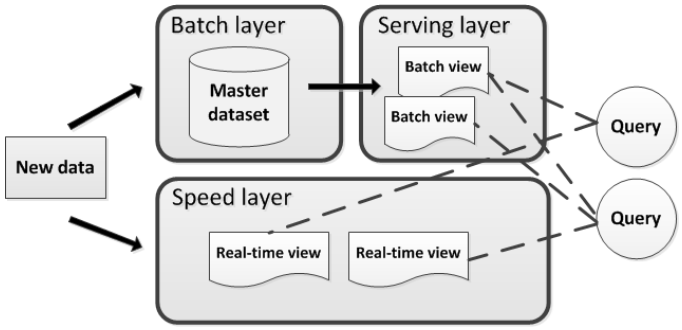

Figure 1: The Lambda Architecture

2.5 Combining Batch with Real-time Processing

2.5.1 Lambda Architecture

The Lambda Architecture, depicted in Figure 1, is a streaming and batch processing architecture paradigm that aims to simplify stream processing at scale. The Lambda Architecture makes a clear distinction between the real-time and the historical data in the system. Streaming data arrive at a high rate and the end-user needs to extract information quickly, even at the cost of lower accuracy. Historical data on the other side can be queried and aggregated at higher latencies.

The Lambda Architecture periodically merges streaming with historical data and re-runs the computation using offline (batch processing) algorithms. Also, materialized views of the dataset are created, in order to provide efficient merging and query response. In more detail, the Lambda Architecture consists of three components:

1. The Batch Layer: The batch layer contains the immutable master dataset. Eventually all data are merged into the master dataset. The batch layer performs computation for the

entire dataset continuously.

2. The Serving Layer: The serving layer serves views of the data while the batch layer computes the next iteration and provides indexing and materialized views.

3. The Speed Layer: The speed layer contains the information that has not been yet incorporated into the batch or serving layer.

2.5.2 Kappa Architecture

One of the biggest drawbacks of the Lambda Architecture is that the code responsible for a certain query has to be implemented twice, one for the batch and one for the stream layer. This means that the code needs to be modified and maintained in two places. Unlike the Lambda Architecture, the Kappa Architecture, depicted in Figure 2, considers the batch layer as a finite stream so there is no need to implement the same algorithm twice since everything is a stream. This also alleviates the need for the batch layer to recompute data that the Speed Layer has already computed.

In the Kappa Architecture, the user starts a streaming job on the log, after she has specified the desired start offset. In more detail, the Kappa Architecture works as follows:

1. It requires Kafka or some other system that supports multiple subscribers and log retention for the desired period of time. For example, in case the application logic dictates that there is a need to reprocess up to 30 days of data, the retention of the log needs to be set to 30 days.

2. When reprocessing is required, a second instance of the stream processing job is initiated that starts processing from the beginning of the retained data, but this output data is directed to a new output table.

3. When the second job has caught up, the application is switched to read from the new table.

4. Then, the old version of the job is stopped and the old output table is deleted.

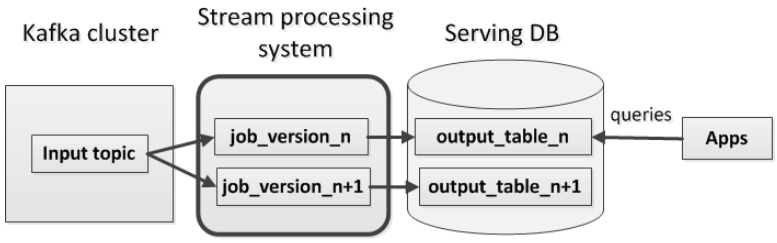

Figure 2: The Kappa Architecture

2.6 Data Serialization

2.6.1 Avro

Avro[16] is a row format for files. It is a top-level Apache project that has great integration with the Hadoop ecosystem and related projects (such as Parquet).

Avro provides:

1. Rich data structures.
2. A compact, binary data format.
3. A container file, to store persistent data.
4. Remote procedure call (RPC).
5. Simple integration with dynamic languages (languages that do not provide static types). Code generation is not required to read or write data files; in fact, code generation is an optional optimization. This leads to less verbose source code and more flexible schema evolution.

Avro relies on *Schemas*. When Avro data is read, the schema used when writing it is always present. This permits each datum to be written with no per-value overheads, making serialization both fast and small. This also facilitates use with dynamic, scripting languages, since data, together with its schema, is fully self-describing.

When Avro data is stored in a file, its schema is stored with it, so that files may be processed later by any program. If the program reading the data expects a different schema this can be easily resolved, since both schemas are present.

When Avro is used in RPC, the client and server exchange schemas in the connection handshake. This can be optimised so that, for most calls, no schemas are actually transmitted. Since both client and server both have the other's full schema, correspondence between same named fields, missing fields, extra fields, etc., can all be easily resolved. This is also possible when using Avro together with Kafka using Confluent's *Schema Registry*. Avro schemas are defined with JSON. This facilitates implementation in languages that already have JSON libraries.

Avro provides functionality similar to systems such as Thrift and Protocol Buffers. Avro differs from these systems in the following fundamental aspects.

1. Dynamic typing: Avro does not require that code be generated. Data is always accompanied by a schema that permits full processing of that data without code generation, static datatypes, etc. This facilitates construction of generic data-processing systems and languages. This also means that the schema can easily evolve over time.
2. Untagged data: Since the schema is present when data is read, considerably less type information need be encoded with data, resulting in smaller serialization size.
3. No manually-assigned field IDs: When a schema changes, both the old and new schema are always present when processing data, so differences may be resolved symbolically, using field names.

Avro being a row-store is appropriate when we need to extract all the fields of the record at once, while Parquet (described below) being a column store is very efficient for filtering records according to the value in a single column or retrieve a limited number of columns in our dataset.

[16] https://avro.apache.org/

2.6.2 Parquet

Parquet[17] aims to provide compressed, efficient columnar data representation for the Hadoop ecosystem. Parquet is built with complex nested data structures in mind, and uses the record shredding and assembly algorithm described in Dremel [15, 16]. Parquet supports very efficient compression and encoding schemes. Parquet allows compression schemes to be specified on a per-column level, and is future-proofed to allow adding more encodings as they are invented and implemented.

Parquet has the following building blocks:

1. *Block* (HDFS block): This means a block in HDFS and the meaning is unchanged for describing this file format. The file format is designed to work well on top of HDFS.
2. *File*: An HDFS file that must include the metadata for the file. It does not need to actually contain the data.
3. *Row group*: A logical horizontal partitioning of the data into rows. There is no physical structure that is guaranteed for a row group. A row group consists of a column chunk for each column in the dataset.
4. *Column chunk*: A chunk of the data for a particular column. These live in a particular row group and are guaranteed to be contiguous in the file.
5. *Page*: Column chunks are divided up into pages. A page is conceptually an indivisible unit (in terms of compression and encoding). There can be multiple page types which is interleaved in a column chunk.

Hierarchically, a file consists of one or more Row Groups. A Row Group contains exactly one column chunk per column. Column chunks contain one or more Pages

When reading or writing a Parquet file, operations are performed in parallel in the following granularity:

1. *MapReduce* - File/Row Group.
2. *IO* - Column chunk.
3. *Encoding/Compression* - Page.

Parquet has the following configurable parameters:

1. Row group size: Larger row groups allow for larger column chunks which makes it possible to do larger sequential IO. Larger groups also require more buffering in the write path (or a two pass write). Large row groups are recommended (512MB - 1GB).
2. Since an entire row group might need to be read, it should completely fit in one HDFS block. Therefore, HDFS block sizes should also be set to be larger. An optimised read setup would be: 1GB row groups, 1GB HDFS block size, 1 HDFS block per HDFS file.
3. Data page size: Data pages should be considered indivisible so smaller data pages allow for more fine grained reading (e.g. single row lookup). Larger page sizes incur less space overhead (less page headers) and potentially less parsing overhead (processing headers). It should be noted that for sequential scans it is not expected to read a page at a time; this is not the IO chunk.

[17] https://parquet.apache.org/

2.7 Recommendations for the datAcron Architecture

2.7.1 datAcron Data Management Architecture

The datAcron architecture cannot be straightforwardly mapped to the Lambda or the Kappa Architecture, although it resembles the Lambda architecture. Instead, due to the peculiarities of the real-time operations that must be performed, the following needs are identified:

1. There will be a Batch Layer as in the Lambda Architecture, containing contextual and historical information.
2. There will be multiple Streaming Layers with different time-constraints and optional integration with historical data.
3. For the initial phase, there will be no Serving Layer per-se. The individual data analytics component is going to have the responsibility of merging the data from the Batch Layer with a selection of data from the Streaming Layers.

The datAcron architecture will provide specific integration between the historical information and the streams in specific latency constraints. Apart from the integrated information provided by the data management system, the data analytics components will be responsible for merging the information. This means that the system will provide a pre-defined integration of specific data sources at the Batch Layer. Real-time and time-critical operations are going to be performed in the Streaming Layers.

2.7.2 Batch Processing

The choice for batch processing in datAcron is Apache Spark, mainly due to the high throughput and processing optimizations offered. After careful review of alternative choices for batch processing, which were available at the time of writing this report, it is quite clear that Spark offers superior performance to frameworks such as Hadoop, and it is more actively supported by the community, with new features and libraries being added continuously.

2.7.3 Stream Processing

With regards to stream processing in datAcron, we consider a tiered streaming solution:

1. For low latency operations (*operational latency*) that need to maintain state we have two options: either Kafka Streams, that provide both a high and a low-level API and efficient storage backed by Kafka, or Storm that supports bidirectional communication through Spouts.
2. For medium latency operation (on the order of a few milliseconds, still operational latency but with looser constraints) with complex event-time windowing logic, we opt to use Flink.
3. For near real-time operations (*tactical* and *strategic* latency) Spark Streaming can be used since from version 2.0 onwards it supports event-time windowing semantics in Structured Streaming together with out-of-the-box merging of the batch and streaming layer. On top of that, writing code for streaming Dataframes is not different than regular Dataframes, so there will be minimal code duplication in this incarnation of the Lambda Architecture.

2.7.4 Data Storage

For data storage, a tiered architecture can deal with the following constraints:

1. For read-only datasets that do not need range scan support (such as analysis results from the data analytics components) we opt for Parquet, since Parquet offers superior read performance, column projections and efficient compression, but at the time of this writing lacks support for indexed scans. Parquet is just a file format so there is no dependency apart from HDFS.

2. For historical datasets (such as the integrated data stored in datAcron) that are append-only, different alternatives exist depending on the access patterns. If there is no need for random access, then HDFS is a viable solution. If random access is necessary, then pure HDFS is not adequate, and alternatives such as Hbase or Cassandra are options. Hbase is part of the Hadoop ecosystem, thus there is no operational cost, whereas Cassandra requires a separate Cassandra Cluster.

3. For mostly-aggregates, update-able datasets, Hbase or Cassandra can be employed, in case these datasets are too large to fit entirely in memory and need persistence. For in-memory datasets, many other options are available.

4. For the streaming component we need in-memory structures to hold state and temporary aggregation, before they are ingested into persistent storage. Here our options are KTables, FiloDB or GemFire, depending on the nature of the input data, the queries and the ingestion process we want to support. KTables are part of Kafka, FiloDB builds on Cassandra for persistent storage (if this functionality is required), and GemFire requires a separate cluster.

2.7.5 Data Serialization

For the data serialization format, we plan to use either Avro or Parquet, depending on the use-case:

1. For generic data exchange with a predetermined schema, Avro is our choice. Avro has quick serialization and since we do not need filtering on the columns of the payload, its row format is a nice fit.

2. For exchanging information that will later be stored as immutable datasets in Parquet (such as the machine learning models and parameters from data analytics components and their respective analysis results) or data that we might need to filter and keep only a portion, of our choice will be Parquet.

3 The datAcron Integrated System Architecture

The project datAcron aims at recognizing and forecasting complex events and trajectories from a wealth of input data, both data-at-rest and data-in-motion, by applying appropriate techniques for Big Data analysis. The technical challenges associated with Big Data analysis are manifold, and perhaps better illustrated in [2, 11], where the *Big Data Analysis Pipeline* is presented. As depicted in Figure 3, five major phases (or steps) are identified in the processing pipeline:

1. Data Acquisition and Recording
2. Information Extraction and Cleaning
3. Data Integration, Aggregation, and Representation
4. Query Processing, Data Modeling, and Analysis
5. Interpretation

Figure 3: Major steps in analysis of Big Data.

The datAcron architecture also targets Big Data analysis and includes the afore-described major steps. The datAcron architecture is illustrated in Figure 4 and is composed of the following six main components:

- ***Synopses Generator***: Its main role is to provide the algorithms for trajectory compression, by eliminating many positions of moving objects that do not significantly affect the quality of the representation.
- ***In-situ Processing***: This component is responsible for executing processing tasks, such as detection of low-level events, on the premise of the actual streams.
- ***Data Manager***: The data management component stores integrated data-at-rest with data-in-motion, as well as analysis results from other components, and provides querying functionality on top of a unified view of data, due to the data integration. Persistent storage and querying of integrated data is provided by means of a *distributed RDF store*, which is a module maintained by the *Data Manager*.
- ***Trajectory Detection and Prediction***: This component performs trajectory prediction, both in real-time and offline, as well as advanced data analytics related to moving objects.
- ***Event Recognition and Forecasting***: This component is responsible for detection and forecasting of complex events related to the mobility of objects.

- ***Visual Analytics***: The exploratory data analytics component provides visualization facilities as well as the opportunity to explore different values for the parameters of the algorithms and provide better models for the event detection and trajectory prediction components.

Inputs to the datAcron architecture consist of data-at-rest (archival data) and data-in-motion (streaming data). Archival data are loaded in the *Data Manager*, and during this process they are transformed, integrated, and stored as will be described in detail in the sequel. On the other hand, a distinction is made for streaming data, namely whether they are positional data describing the spatio-temporal movement of objects (trajectories) or not. Trajectories are treated as "first-class" citizens in datAcron, thus trajectory data is summarized (at *Synopses Generator*) and associated with low-level events (during *In-situ Processing*). Also, they are integrated in the *Data Manager* component, before storing, with existing (static) data, such as ports, airports, information about the moving object (vessel/aircraft type, model, etc.). Other streaming data, such as weather forecasts, flight plans, regulations, etc., are directly fetched by the *Data Manager*, in order to be integrated with the other available data.

The following streams of data are available in the datAcron system, either as input streams or as generated streams by datAcron components:

- *raw data stream* of surveillance data, as it arrives in the datAcron system
- *compressed stream* of surveillance data, which consists of "critical points" only, after having discarded positions that do not carry useful information about the movement of the object
- *enriched stream* of surveillance data, which identifies low-level events associated to positions of moving objects
- *integrated stream* of surveillance data, which adds contextual information to the positions after performing integration with all different data sources (data-at-rest) available to the *Data Manager*

The afore-described streams are available to *all components* that perform data analytics in real-time, namely *Trajectory Detection and Prediction*, *Event Recognition and Forecasting* and *Visual Analytics*, in order to provide the input necessary for the respective data analysis tasks. In essence, this results in a *loosely-coupled architecture*, where higher level components that perform data analytics can consume the output of other components that perform data extraction or integration, in order to optimize their operation in real-time. Also, data analytics components may also interact with each other; for instance, the *Visual Analytics* component visualizes the events detected or predicted by the *Event Recognition and Forecasting* component in order to perform visual analytics, and the *Event Recognition and Forecasting* component takes as input the trajectories detected or predicted by *Trajectory Detection and Prediction* to identify complex events related to trajectories.

When considering outputs of the datAcron architecture to the end-user, these consist of detected and forecast events, data analytics results (detected and predicted trajectories and events, exploratory visual analytics, etc.) initiated by a user that performs a specific task, and results to queries over the integrated data provided by the *Data Manager*.

The key issues for the datAcron architecture are as follows:

- The data synopses computed near to the sources aim to largely reduce at a high compression rate the streaming data that the data management and analytics layers have to manage. However, access to the raw streaming data is still an option for the analytics components, in case a component requires this explicitly.

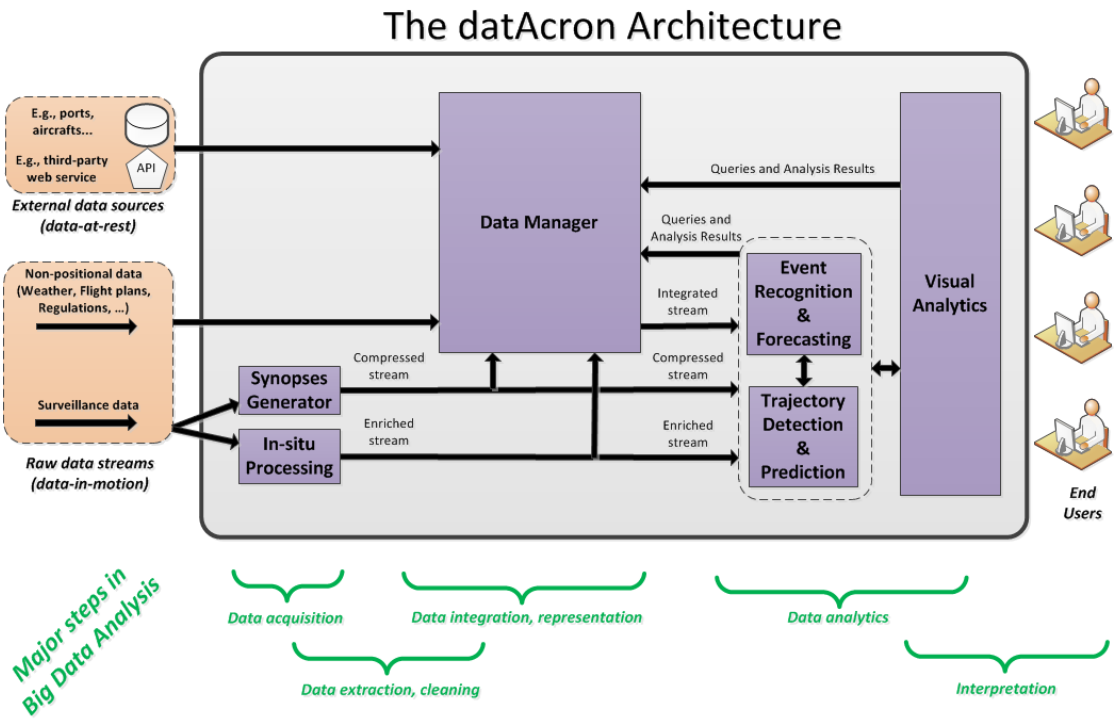

Figure 4: The datAcron Architecture.

- The data synopses computed from multiple streams can already be integrated at the lower processing components (near to the sources). Data synopses and archival data are transformed into a common form according to the dataAcron RDFS schema, are integrated (where necessary) and are pipelined to the rest of the analytics components directly, in real-time. This alleviates the need for analytics components to access datAcron stores frequently.
- "Raw" streaming data are not stored as they enter the system: Persistent storage concerns data synopses, semantically annotated and integrated to archival data, trajectories and events detected. The datAcron stores will provide advanced query answering services for other system components and human or software clients to access these data, according to their requirements on integrated data views.

The above architecture has certain benefits:

- All data from streaming and archival data sources, as well as trajectories and events computed by analytics components can be semantically integrated by discovering links between respective instances, providing semantically-rich coherent views of data. Doing so, datAcron seamlessly annotates trajectories and events with semantic information, and it links these among themselves as well as with the rest of archival and cross-streaming data.
- All analytics components can take full benefit of the computations of others, also taking advantage of interlinking between their results. Thus, the trajectory detection and forecasting methods can benefit from events detected or forecasted and vise-versa. Similarly for the visual analytics methods.

- Users can interact and explore data via integrated data views, being supported for decision-making.

In the following, we delve in more details regarding the **functionality** of each individual component, but also their **interactions** in order to realize the objectives of datAcron.

3.1 Synopses Generator

The *Synopses Generator* is responsible for producing trajectory synopses by examining the raw input stream of positional data and eliminating those positions that do not carry useful information regarding the movement of the object. As depicted in Figure 5, *Synopses Generator* contains two discrete modules:

- *Trajectory Constructor*
- *Trajectory Compressor*

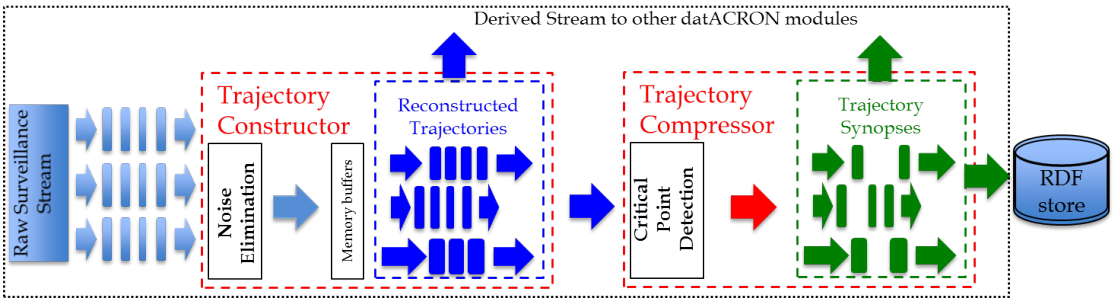

Figure 5: The internal operation of the *Synopses Generator* component.

Trajectory Constructor: Reconstructing trajectories from surveillance data will provide representation in time of the original (raw) positions that are being received as input. In effect, distinct sequences of timestamped positions per moving object will be obtained, after excluding any inherent noise detected in the streaming positions due to e.g., delayed arrival of messages, duplicate messages, etc. The resulting trajectories will be available in a streaming fashion, including every "clean" position from the input raw data. It is not envisaged that this information will be permanently stored in RDF repository.

Trajectory Compressor: Maintaining trajectory synopses from surveillance data will offer representation in time of characteristic positions from each moving object. Based on the reconstructed trajectories emitted by the previous module, this compressor will track major changes along each object's movement. Given that vessels and aircrafts normally follow planned routes (except for adverse weather conditions, congestion situations, accidents, etc.), this process will instantly identify "critical points" along each trajectory. These are point locations of each object that can be characterized from (strictly) mobility features, such as stop, turn, or speed change, without taking into account any contextual information. Therefore, an approximate, simplified trajectory (what we call trajectory synopsis) may be maintained consisting of critical points only, effectively discarding redundant locations along a "normal" course. Thanks to online computation of such trajectory synopses, object traces should remain lightweight for efficient processing

without sacrificing accuracy, and can be easily compared to each other irrespectively of the actual reporting frequency that may differ among objects.

Once detected, each critical point in such trajectory synopses will be handled by the *Data Manager* component. First, critical points are going to be semantically linked and contextually enriched. This information is going to be streamed out from the *Data Manager* as an *integrated stream*, so as to be immediately available to other components for real-time operations. In addition, this information will eventually result in semantic trajectories, stored in the RDF store, and will be available for querying, offline processing, mining, etc.

3.2 In-situ Processing

In-situ Processing refers in general to the possibility to process streaming data as "downwards" in-stream as possible. Processing streaming data close to data source provides a number of inherent advantages:

- Communication can be reduced in size and number of messages. Too many or too large messages often induce a communication overhead that implies penalties in terms of performance and transmissions costs. Reducing size and number of messages at an early stage helps to control overall costs and resource consumption of the complete system.
- Latency is reduced and reactivity increased. Processing data close to the source reduces the number of steps and/or systems the data has to pass in order to be analyzed. Consequently, latency in detection of analytical results is reduced and critical events, such as for example trajectory deviations, can be detected and signaled earlier to the applications.
- Complexity of the applications is reduced. In-situ processing can already provide aggregated, filtered and condensed views on the current state of the external views such that applications and visualization is relieved from the burden of processing raw data. For example, detection of derived events such as crossing of section boundaries will be implemented in-situ such that the application can make use of these derived complex events without any need for further analytics inside of the application.

Figure 6 depicts the principle architectural design and placement of the *In-situ Processing* components in the datAcron architecture. Referring this figure, the data integrator provides functionality for providing an integrated, semantically enriched view on the data which is being stored in the RDF triple store. As it could be done in the Lambda Architecture, *In-situ Processing* is placed before the entry of the data into the data integrator. It consumes data from incoming data streams and generates via in-stream analytics higher level, enriched data streams, which are consumable by:

- further, higher level analytical steps in the in-situ processing chain,
- the data integrator, or
- the datAcron application and visualization modules

In order to support this kind of multi-stage, multi-target stream processing, the architecture makes use of a message bus system as provided by Kafka. In this design, different streams can be distinguished by name (e.g. "Topic" in Kafka) and the same data, potentially aggregated

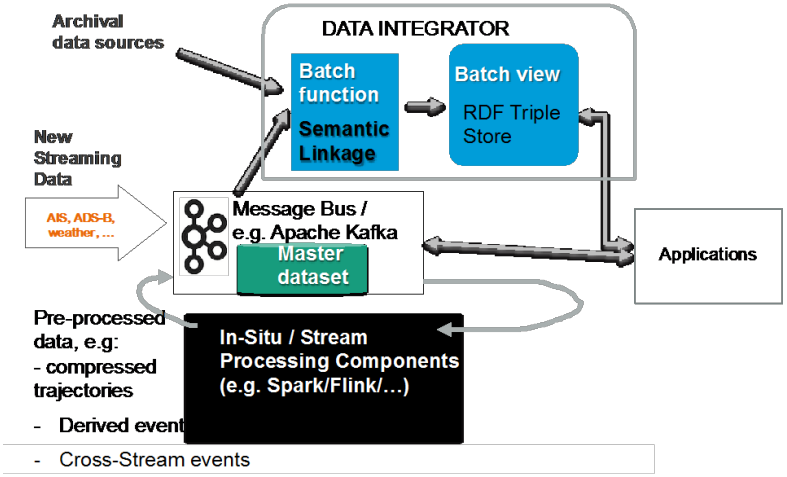

Figure 6: The *In-situ Processing* components in the datAcron Architecture.

or enriched via in-stream analytic, can be communicated in different streams, reflecting the order and hierarchy of the analytical steps performed by the *In-situ Processing* components. In addition, as Kafka offers the possibility to persist the data contained in the streams, it can be used as basic store for the unprocessed, raw data in style on the Kappa architecture (Figure 2). As the *In-situ Processing* components consume, analyze, and produce Kafka streams, they can be implemented in any stream processing system that connects to Kafka. In sum, the architectural integration of the *In-situ Processing* components is exhibited by the following characteristics:

- Kafka acts as central message bus and repository for raw data (in style of the Kappa architecture)
- Every component working in-stream/in-situ reads some Kafka stream, processes the data, and sends back results as some other Kafka stream
- For stream processing, several options are possible and can be intermixed, depending on use case requirements and algorithmic choices of each component: Storm, Spark Streaming, Flink, Kafka Streams
- The *Data Manager* also extracts the data from Kafka and converts it to semantically linked triples that are stored in the RDF store
- Batch applications from all data analytics components read/write data from/to the RDF store
- Real-Time monitoring apps in the data analytics components read data from Kafka
- Application-dependent input and parameters to the *In-situ Processing* components is sent via Kafka streams to the *In-situ Processing* components

The last mentioned point addresses an important aspect and challenge of in-situ processing that occurs when then *In-situ Processing* components depend on input or parameters provided by the applications. For instance, in the detection of context related events, the context may change or be varied by the application. In terms of use cases, this occurs for example in monitoring

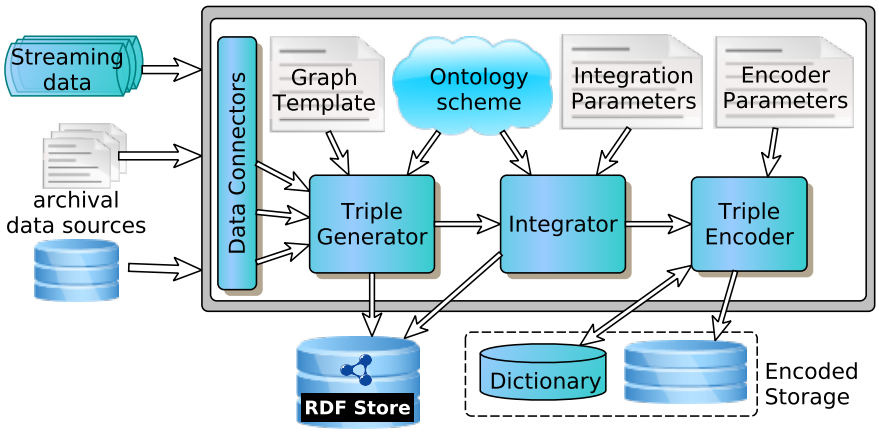

Figure 7: The *Data Integrator*: The data flow during the data transformation and integration process is depicted with main modules: (a) data connectors, (b) triple generator, (c) integrator, and (d) triple encoder.

flights via ADS-B data for detecting complex events such as entering or leaving sectors. When the definition of sectors changes, this must be forwarded to the*In-situ Processing* component as well. Further examples include the application-dependent tuning of parameters for the *In-situ Processing* components, which can occur on human feedback on the basis of visual analytics.

3.3 Data Manager

The *Data Manager* component plays a central role in the overall architecture, as it stores the integrated data and provides parallel data processing functionality on top of the integrated data. Its functionality is: (a) to transform and integrate any incoming data, whether streaming data, archival data, or analysis results, into a coherent and unified view, in sync with the datAcron model, and (b) to effectively store the integrated data and develop query processing primitives that allow parallel data processing at scale. Internally, *Data Manager* contains two main modules: the *Data Integrator* and the *Distributed Spatio-temporal RDF Store*.

3.3.1 Data Integrator

The *Data Integrator* comprises four main modules, a) the *Data Connector*, b) the *Triple Generator*, c) the *Integrator*, and d) the *Triple Encoder*. Figure 7 depicts the interactions between these modules.

Data Connector The *Data Connector* module implements the functions that accept data from an individual data source. Moreover, it performs data conversion on values of specific fields as provided by the source. As the data sources can vary significantly in terms of representation, size, rate of data access, and noise, a *Data Connector* is provided for each type of source. Data consumption from a wide range of sources/formats is going to be supported, such as: a) CSV format, b) direct access to databases, c) JSON messages, d) XML files, e) METAR/SPECI

weather reports from offline/continuous feeds from online services, f) binary (GRIB2) files for weather reports, g) SPARQL endpoints to online open data, h) ESRI shape-files to convert information about spatial object to entities and relations of the ontology, and i) proprietary formats of streaming. The list of supported data formats can be easily extended, to support the inclusion of any other data format required in the future.

Given a specific type of source, an editable configuration file determines the connector's functionality. Thus, the same *Data Connector* can be reused across similar data sources, given the appropriate configuration. Furthermore, some basic data cleaning operations are implemented and are applicable on the individual record level. However, such operations should have low complexity, as the data consumption process should not incur significant additional latency. The data connector can also be charged with the responsibility of data conversion on values of specific fields in the source.

As an abstraction of the data source in hand, the *Data Connector* considers all data sources as streams, i.e. it consumes data record-by-record (or tuple-by-tuple), as a stream of records to be processed with minimal latency. This data access model makes no distinction on the nature of data (e.g. archival data or streaming data). In addition, it minimizes the memory footprint of *Data Connectors*, and enables scalability and parallelization as a multi-threaded process, both for a single source and across sources.

Triple Generator The role of *Triple Generator* is to consume the data provided by the *Data Connector* and generate the corresponding set of triples. This procedure depends on configuration files providing a Graph Template $\mathbf{G}_T$, and a vector of variable names $\mathbf{V}$. The vector $\mathbf{V}$ contains the variables that appear in the Graph Template, and binds the values of the input record to the variables in $\mathbf{G}_T$. The Graph Template consists of a set of *triple patterns*, i.e. any of the three elements in the triple (subject, predicate, object) can be replaced by a variable. The Graph Template differs from the standard RDF Graph Pattern, in that variables can be used as arguments in functions, to compute dynamic values at runtime. The functions employed in Graph Templates, add no significant overhead to the data processing workflow.

Integrator The purpose of the *Integrator* is to consult one or more streams of triples as generated by the *Triple Generator*, in order to discover links between: a) resources[18] in the triples, or b) resources in the triples and resources at archival sources. In the first case, this mechanism can be applied for deduplication tasks, in which case the links to be discovered are owl:SameAs. In both cases, links can be identified between a resource and other entities (e.g. spatial areas of interest, spatio-temporal points or regions, etc.)

The component employs a set of filters to select only those triples that are related to the link discovery process. The filtered triples are directed to a blocking mechanism (or a spatial index for spatial relations) that detects a set of candidate related resources. The final step in the process is the refinement of candidates, which is performed using multi-threading, to discover the related resources and return the corresponding triples

The integration depends on a mechanism that can decide whether two (or more) resources should be linked by a specific relation. A naive approach where every resource in the first source should be evaluated against every resource in the second source is inefficient for data streams. Thus, the filtering, the blocking mechanism and parallel refinement in this component drastically improve performance. Blocking mechanisms and algorithms often vary, depending on the relation to be discovered and the sources.

The configuration of this component is crucial for the overall performance of the workflow. A link discovery component should generate high-quality links w.r.t. metrics such as precision and

[18] a resource is a distinguished entity that may appear in the object or subject position in any triple.

recall. This means that the sets of links generated should be correct (precision) and complete.

Triple Encoder This component is responsible for compressing the RDF triples produced by the *Triple Generator*, in order to optimize the space consumption of the RDF encoded data. For this purpose, a dictionary provides a mapping between URIs and unique identifiers (integer values), thus allowing the transformation of RDF triples into triples of integers. Apart from compression, this technique also enables more efficient indexing techniques, since integer values can be indexed more effectively than strings. The encoded triples and the dictionary can – in theory – be stored in another system that supports query processing facilities, such as a relational DBMS. When an RDF store that uses internal encoding methods is used as target, the *Triple Encoder* can be disabled, and RDF triples are produced in their original form. However, both these alternatives (relational DBMS and centralized RDF store) are not viable solutions in datAcron, due to scalability limitations.

In addition, the *Triple Encoder* is designed with a parametric assignment policy for unique identifiers. The simplest form of such a policy is to use integers and generate the next identifier by increasing the previously assigned identifier by one. However, it is foreseen that other more advanced policies can be plugged in. In our current implementation the *Triple Encoder* is a centralized process which handles the output of multiple *Triple Generators*.

3.3.2 The Distributed Spatio-temporal RDF Store

The integrated data in datAcron need to be stored and efficiently queried in order to support various data analysis tasks. The choice of data storage depends on many different parameters, including *data-dependent* characteristics: the structure of the data, its volume, the rate of generation, but also on *query-dependent*: namely the access patterns that need to be supported.

Data in datAcron are Big Spatio-temporal RDF data. The representation model is RDF, in order to address the *variety* challenge present in the miscellaneous types of raw input data. The *volume* of the data is vast and increasing every second, as new data arrive from the various input streams. Also, there is a need for addressing the potentially high stream rate, which raises a *velocity* challenge. Finally, a high percentage of the data is of spatio-temporal nature, describing trajectories of moving objects, spatial or spatio-temporal areas of interest, events with spatio-temporal dimensions, and other variables (e.g., meteorological, environmental, etc.) that have a spatio-temporal validity.

On the other hand, different data analysis tasks need to be supported over this integrated data, including complex event detection, prediction of movement, pattern discovery, etc. In addition, querying functionality is also necessary to support both one-time queries with filters over the data, as well as data exploration assisted by visual analytics.

All above requirements motivate the need and guide the design and development of a *distributed spatio-temporal RDF store* that additionally supports *parallel data processing* in order to provide a scalable solution.

3.3.3 Interconnections

The *Data Manager* receives the following inputs in terms of data sources:

- external data (a.k.a. data-at-rest), in the form of archival data, that feeds the RDF store, after data integration has been performed
- streaming data (a.k.a data-in-motion), apart from positional streaming data, which include weather forecasts, flight plans and updates thereof, regulations, which are integrated and stored persistently

A significant streaming data source is positional data of moving objects. This source is not accessed directly by the *Data Manager*, but instead it is processed by the *In-situ Processing* and the *Synopses Generator* components, which produce processed streams that are going to be integrated and stored persistently. In more detail, *Synopses Generator* produces trajectory synopses of positional data, both from maritime and aviation trajectory data, and this is the main positional data that is integrated with the afore-described data sources. Moreover, *In-situ Processing* detects low-level events associated with positions of moving objects, as close to the data sources as possible. Such low-level events include single stream events, cross-stream events, as well as events between a stream and contextual information. For instance, when a moving object enters a spatial area of interest (e.g., a protected sea area or an air sector of interest), this event is detected and given as input to the *Data Manager* for storage.

A final input source for *Data Manager* is analysis results produced by the different components that perform data analytics. Such results may include discovered patterns (e.g., clusters), detected complex events, forecast events or trajectories, routes, as well as visual analytics results of exploratory analysis. Again, these analysis results are integrated with existing data, prior to storage. For instance, a discovered movement pattern followed by multiple moving objects is associated with the trajectories of the moving objects that support the pattern.

In terms of outputs, *Data Manager* also produces a stream of integrated data (named *integrated stream*) where any links discovered between entities or objects are provided to the other components of the architecture in real-time.

Last, but not least, *Data Manager* offers querying functionality over the wealth of integrated data. In essence, queries with various filters are supported, including spatio-temporal constraints, but also other parameters relevant to the input data.

3.4 Trajectory Detection and Prediction

The *Trajectory Detection and Prediction* component is internally composed of two main modules: the *Trajectory Predictor* and the *Data Analytics* modules, which are responsible for computing mining models and motion functions over trajectory synopses and RDF data in order to support semantic location and trajectory prediction, predictive queries and advanced analytics. It achieves this objective by exploiting the past movements of moving objects and as well other related information (i.e. meteorological, contextual, etc.) to predict a moving object's anticipated movement in short and long term. It exploits data mining, machine learning and data access methods to assist the production of effective and scalable predictive analytics. As depicted in Figure 8, the *Trajectory Predictor* and the *Data Analytics* components are tightly interconnected (i.e. predictions usually rely on analytical models) consist of three modules:

- Location Predictor
- Local Model Extractor
- Data Analytics

3.4.1 Location Predictor

This component consists of two modules: a) the Continuous RMF Discovery module and b) the Predictive Query Processing module.

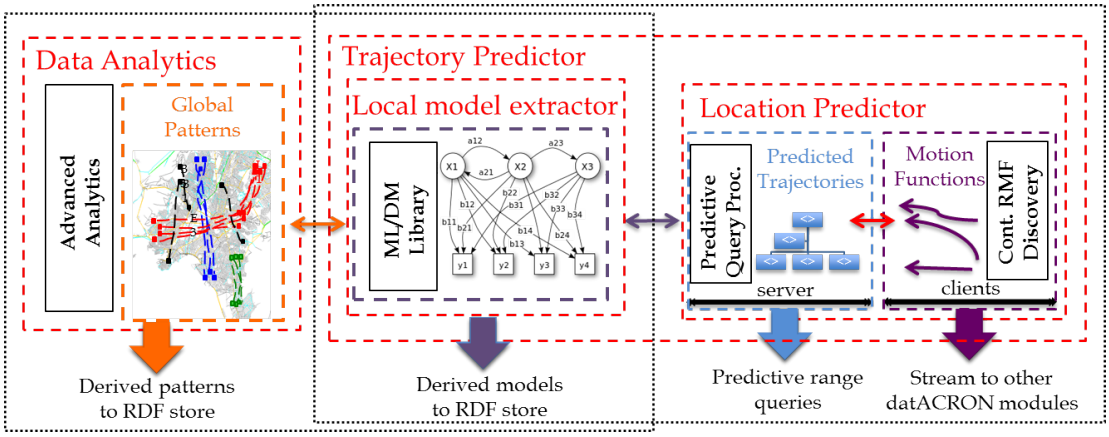

Figure 8: The *Trajectory Detection and Prediction* internal architecture.

The *Continuous RMF Discovery* module calculates *motion functions* by harvesting the most recent locations from a moving object to predict its short-term future location in real time by taking into consideration the tendency of the movement (i.e. linear, quadratic, curving, etc.). This is realized by a recursive approach, titled as Recursive Motion Function (RMF), which takes as input the motion state of a moving object along with the previous spatiotemporal instances (i.e. locations and timestamps) and the motion matrix of the past movements. It then calculates future object's locations with respect to its individual moving behavior. As the motion functions progressively discover future locations for different moving objects, the results will be available in streaming fashion to other datAcron modules.

The *Predictive Query Processing* module is based on an indexing scheme and query processing mechanism that is able to organize and efficiently query the future states of moving objects. In detail, this module will device novel encodings of not only the data as they are, but more importantly their anticipated future locations and semantics. These encodings will be organized in appropriate access methods so as to support predictive range queries (e.g. how many/which aircrafts will be on Athens FIR the following ten minutes).

3.4.2 Local Model Extractor

This module is responsible for applying data mining (e.g. clustering and sequential patterns), machine learning and statistical methods (e.g. Hidden Markov Models) on historical (semantic) trajectories retrieved by the RDF store in order to extract patterns whose primary goal is to facilitate long-term location and/or trajectory prediction. More specifically, the idea is that the *Location Predictor* module consults the extracted patterns and decides whether it is preferable to use the knowledge about the collective tendency of a group of moving objects encapsulated in the pattern or the individual's short history results in more accurate predictions. The derived models are envisaged to be stored/retrieved in/by RDF repository.

3.4.3 Data Analytics

The goal of this module is twofold: on the one hand it provides advanced analytics that are going to serve specialized requirements in the datAcron architecture (e.g. data-driven discovery of the networks 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 patterns extracted

by the *Local Model Extractor*. As a concrete example of this case, consider that the patterns extracted from the *Local Model Extractor* are sequential patterns, then a kind of global pattern nay be a hierarchical organization (e.g. a tree) of these sequential patterns), which facilitates efficient predictive analysis. Again, the derived global models are to be stored/retrieved in/by RDF repository.

3.5 Event Recognition and Forecasting

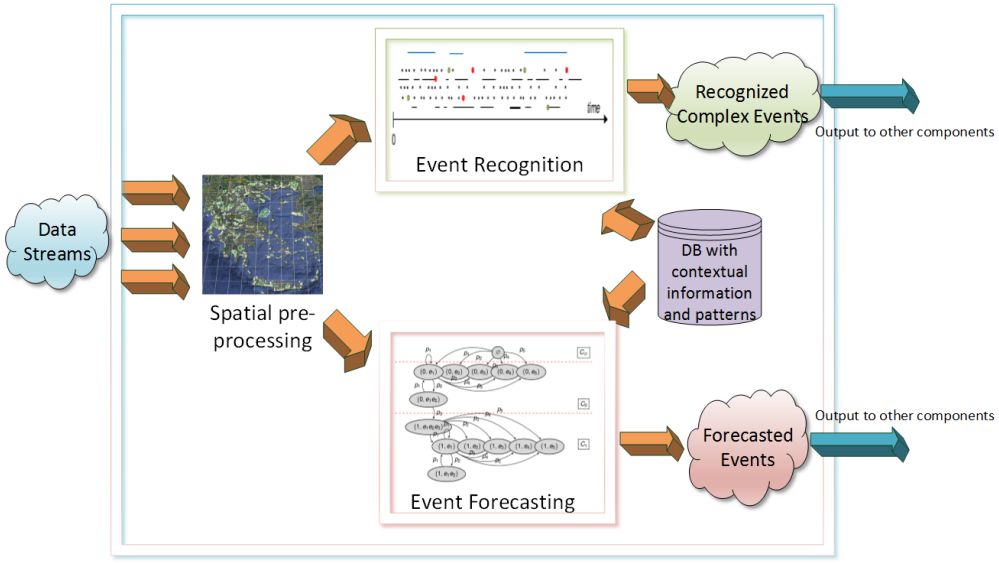

Figure 9: The architecture of the *Event Recognition and Forecasting* component.

The *Event Recognition and Forecasting* component consists of two modules.

- The first one concerns event recognition and forecasting. Its main sub-modules are the following:
  - A formal **Complex Event Recognition** engine which will consume streams (compressed stream of critical points, enriched stream, and integrated stream) produced by datAcron components and produce a stream of complex events. This engine currently works in a "pull" mode, i.e., at each query step it collects the new critical points that have arrived within a pre-defined temporal window. The Complex Event Recognition module is based on the "Event Calculus for Run-Time reasoning" [6]. The Event Calculus is a logic programming action language. RTEC has a formal, declarative semantics—Complex Event patterns in RTEC are (locally) stratified logic programs. In contrast, almost all complex event processing languages and several data stream processing languages lack a rigorous, formal semantics. Reliance on informal semantics constitutes a serious limitation for maritime monitoring, where validation and traceability of the effects of events are crucial. Moreover, the semantics of event query

languages and production rule languages often have an algebraic and less declarative flavor.

- An **Event Forecasting** engine which will consume critical points and attempt to provide predictions for the occurrences of events. This module works with sequence patterns provided by the user as events of interest. The engine allows for patterns defined by languages that are a subset of the regular languages. By embedding the pattern into a Markov chain, it becomes possible to derive various statistics about it. Of particular interest for the purposes of forecasting is the waiting time distributions, i.e., the time required until the pattern is completed. Based on this distribution, predictions can be provided about the expected completion time of the provided pattern.
- An internal database holding the necessary contextual information, appropriately indexed for the purposes of event recognition and forecasting. This information is asynchronously fetched by the *Data Manager* and is efficiently indexed, in order to support the interactive processing time required by the real-time operations related to event detection and forecasting.
- A spatial pre-processing module, responsible for a special class of spatial tasks, required by the recognition and forecasting engines. This refers to purpose-specific spatial tasks for event detection and forecasting, which may not have already been performed by any of the other components of the architecture.

- The second module will be that of Machine Learning, which is responsible for building models that will be used for the internal operation of *Event Recognition and Forecasting*. This module will work in off-line mode and thus it will not affect the rest of the architecture.

Initially, the *Event Recognition and Forecasting* component will not have to interact with the *Data Manager* during run-time in order to retrieve contextual information not present in the stream of critical points and annotated trajectories (e.g., protected areas). Such information will be retrieved beforehand and will be stored in a database internal to *Event Recognition and Forecasting*.

Both the Event Recognition and the Event Forecasting modules will be developed in the Scala programming language. In the future, distributed versions may be developed, based on the Apache Flink stream processing platform.

3.6 Visual Analytics

The *Visual Analytics* (VA) component provides facilities for the visual exploration of data and visual-interactive support for building and refining models and their parameter settings. It therefore targets analysis experts working on the strategic level using data at rest, but may in suitable cases also provide visualizations with limited interactivity on the tactical or even operational level. Therefore as indicated in Figure 4, the VA component interacts with both the analytical components for event and trajectory processing, as well as the *Data Manager* directly for random access to historical data.

The purpose of the Visual Analysis approach is to combine algorithmic analysis with the human analyst's insight and tacit knowledge in the face of incomplete or informal problem specifications and noisy, incomplete, or conflicting data. Visual Analysis therefore is an iterative process where intermediate results are visually evaluated to ascertain and inform subsequent

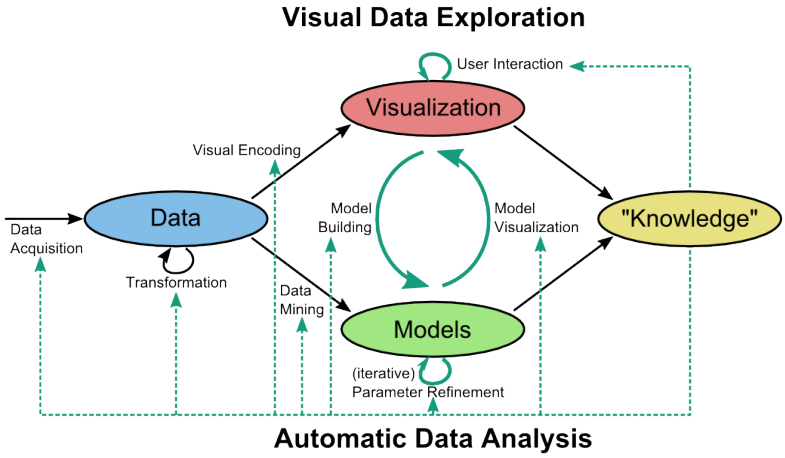

Figure 10: The Visual Analytics Loop supported by datAcron's VA component, adapted from [12].

analysis steps based on prior knowledge and gathered insights. The underlying conceptual model is the Visual Analytics Loop (adapted from [12], cf. Fig. 10). Specifically, it is worth noting that due to the exploratory focus, VA does not prescribe a rigid pipeline of algorithmic processing steps, nor does it prescribe a fixed composition of specific visualizations, as opposed to typical KPI dashboards.

To cope with these requirements in an efficient and scalable way, the *Visual Analytics* component within the integrated datAcron architecture is itself of a modular, extensible design, as shown in Figure 11. It comprises four principal component groups - data storage, analysis methods, data filtering and selection tools, and of course, visualization techniques. Different components are typically composed in an ad-hoc fashion, through visual-interactive controls, to facilitate the workflow required by the human analyst's task at hand. In particular, this allows creating linked multiple views to simultaneously visualize complementary aspects of complex data or analytical models. Figure 11 indicates by color marks matching colors from Figure 10 what components are typically involved in which phases of the VA loop.

The module architecture is a conceptual one. By integration of the VA module into the overall datAcron architecture, functionality of the principal VA components is partially provided by other datAcron components, notably, the *Data Manager* (Section 3.3), as well as the *Trajectory Detection and Prediction* (Section 3.4) and *Event Recognition and Forecasting* (Section 3.5) components.

3.6.1 Data Storage

The data storage component serves three functions. First, it provides an interface to the Data Manager (Section 3.3). This allows read access to historical data, specifically, from the distributed spatio-temporal RDF store, but may in certain cases also encapsulate direct access to other intermediate storage, such as Cassandra or Hive data stores, if required for a given use case scenario (cf. documents D5.3, D6.3). In cases where analysis results in data artifacts that are worth persisting, for example interactively defined area boundaries, "typical" vessel trajectories, or analytical models for later use such as spatio-temporal flow graphs of aircraft, these are pushed down to the Data Manager for long-term storage and retrieval.

Second, this component provides management of intermediate analysis results. This is nec-

essary as interactive analysis frequently requires data representations that are different from archival storage for efficiency reasons, e.g. by denormalizing data kept in a relational schema. In addition, the explorative and iterative nature of analysis often results in intermediate data attributes that are almost immediately discarded for a refined result (e.g., cluster associations of entities after interactive parameter changes to the algorithm). Such data is never persisted and so is not handled by the distributed data store.

Third, ad-hoc analyses might often have the need to integrate external data not yet ingested by the *Data Manager*. Typical examples include data and models generated by scripts or other tools in standard formats (CSV files, Shape files, XML files, or local databases) by an analyst. Enabling this loose, file-based integration into the VA platform has proven essential in maintaining flexibility and extensibility in terms of the analyst's tool capabilities.

3.6.2 Data Selection and Grouping

Similar to the data storage component, functionality of this module is divided between the central *Data Manager* and the VA components internal selector module. Complex queries to large historical or synopsis data are handed off to the query engine of the distributed RDF store (see Section 3.3.2).

One key feature of Visual Analytics, however, is the ability to directly manipulate data and algorithm parameters through visual interaction. Therefore, interactive selection of data elements across multiple views allows the analyst to define complex, multi-faceted filters on data before analytical processing. An example is the simultaneous specification of a geospatial region in a map display, a specific time range from a time graph, and a subset of entities according to some cluster visualization (e.g., see [5]). Such compound queries are mostly executed on the in-memory representation held by the VA module's data storage component. Further investigation of use case scenarios are expected to refine the understanding which such filter conditions and associated queries should be offloaded to the central *Data Manager*, and which are more efficient as internal VA data storage operations.

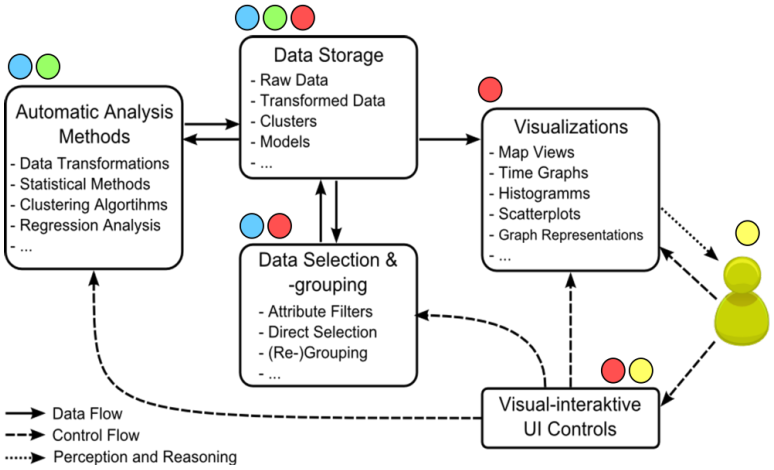

Figure 11: The Visual Analytics module architecture with its principal components to support the VA loop.

3.6.3 Analysis Methods

From the perspective of the *Visual Analytics* module, analysis methods fall into two main categories:

- in-stream processing algorithms that operate directly on data streams under predefined parameter settings: synopses generation (Section 3.1), trajectory detection and prediction (Section 3.4), event detection and forecasting (Section 3.5), and
- those applied in exploratory analysis on the strategic level: selection and parameter tuning of algorithms prior to their application at operational (real-time) or tactical latency levels.

In case of the former, integration is indirect: the VA component may be used to observe the impact of parameter changes on-line through a suitable user interface. This interface may be provided by the VA component in selected cases, but supporting end users in operative level settings is explicitly not the focus of Visual Analytics research or module development.

In the case of the latter, the VA component will facilitate the integration of a wide range of algorithms by loose coupling on the level of tabular and graph-structured data handled by the data storage component. The datAcron project can built on years of practical experience building a similar, standalone framework for Visual Analytics.

3.6.4 Visualizations

Outwardly the core component of a visual analysis system, this component will provide the set of interactive visualization techniques needed for – primarily exploratory – analysis. The final set of visualizations is not prescribed at this point of time, however. Rather, it is derived from use case requirements (see documents D5.3, D6.3), and is likely to involve research in designing novel visualization techniques. Available visualizations are expected to include established base techniques such layered map displays, time graph displays, but also specialized and complex representations such as dynamic flow graph visualizations that are one current focus of VA research in datAcron. Similar to integrated analysis techniques, visualization technique will be integrated on the level of internal data representation, i.e., based on how types of entities and their attribute structure are mapped to specific visual encodings by a given visualization technique.

4 Mapping the datAcron Architecture to Requirements

This section provides the necessary mapping between the architectural requirements identified in Deliverable D1.1 and the proposed datAcron system architecture outlined in Section 3. Essentially, its objective is to answer the question "how are the requirements reflected on the architecture", or, put differently, "how does the architecture address the requirements".

To this end, in the following, we recall the architectural requirements (reported in D1.1), and then proceed to explain the role of each individual component with respect to the requirements, as well as how the combination of distinct components addresses the requirements whenever necessary.

4.1 Architectural Requirements

The proposed architecture is in accordance with the analysis performed in Deliverable D1.1 ("Requirements Analysis"). In brief, in D1.1, the following issues were identified:

- The **data sources** are multiple streaming data sources, as well as archival data sources.
- The **in-situ processing components** aim to compress and integrate where appropriate data-in-motion from streaming sources in communication efficient ways computing single and multi-streaming data synopses at high rates of data compression – without affecting the quality of analytics – capitalizing on low-level primitive operators (e.g. selections and projections, joins to cater for cross-stream processing as close to the data as possible, etc) that are applied directly on the data streams.
- The **data transformation components** aim to convert data in (a) single and multi-streaming data synopses, (b) archival data and (c) results from the datAcron higher levels analytics components to a common form.
- The **data integration component** interlinks semantically annotated data using link discovery techniques for automatically computing correspondences between data from disparate sources. This produces integrated data views, including data and their correspondences. Integrated data are provided to the analytics components in real-time, while they are also stored in parallel stores.
- The **spatiotemporal query-answering component** provides parallel query processing techniques for spatio-temporal query languages. Interlinked data (processed and compressed data-in-motion and linked data-at-rest) are stored in parallel RDF stores, using sophisticated RDF partitioning algorithms in domain specific, spatial and temporal dimensions.
- The **data analytics components** include trajectory and complex event recognition and forecasting, as well as visual analytics. These consume the data provided by the data integration component: Synopses computed by the bottom layer, being integrated (where necessary) with archival data.

The data analytics components also use internal stores for frequent and fast data write/read, well tuned to their requirements and to the rest of the architecture, so as to provide real-time results.

Based on the above, Deliverable D1.1 defined a set of *architectural requirements* (cf. D1.1/Section 4), which are summarized below:

- R1.1: Real-time integration/interlinking of spatial and/or spatio-temporal entities
- R1.2: Interplay of in-situ and stream processing components
- R1.3: Integration/interlinking over stored data
- R1.4: Spatio-temporal RDF querying of integrated data
- R1.5: Retrieval of spatio-temporally constrained subsets of integrated data
- R2.1: Computation of trajectory similarity and clustering
- R2.2: Pattern discovery
- R2.3: Prediction of trajectories and locations
- R2.4: Computation of surveillance data synopses, reconstruction of trajectories by data synopses
- R3.1: Event detection and forecasting in the maritime domain
- R3.2: Event detection and forecasting in the aviation domain

In the following, it is shown how the datAcron architecture (proposed in the current deliverable) covers these requirements.

4.2 Mapping datAcron Components to Requirements

Table 2 briefly summarizes how each datAcron component is mapped to each of the eleven requirements identified in Deliverable D1.1. As will be documented in the sequel, this mapping justifies how the proposed architecture reflects the architectural requirements of the project.

4.2.1 Synopses Generator

The *Synopses Generator* component receives raw surveillance data in a streaming fashion and produces a compressed stream of "critical points", after having discarded the many input surveillance data that do not contribute useful information about the object's movement.

This component is going to address directly the requirement R2.4 (Computation of surveillance data synopses, reconstruction of trajectories by data synopses), which is of major importance for the project, as most trajectory prediction and event forecasting models operate on these synopses, rather than at the level of raw data.

4.2.2 In-situ Processing

In-situ Processing refers to stream processing near to the streaming sources, aiming at decentralized processing per streaming source and minimal overhead to the component that integrates the different streams. As such, *In-situ Processing* plays a significant role in the following requirements:

- R1.1 (Real-time integration/interlinking of spatial and/or spatio-temporal entities): *In-situ Processing* facilitates this real-time interlinking task, by identifying low-level events that trigger data interlinking as early as possible in the architecture. Representative examples of the operation of *In-situ Processing* include the discovery of spatio-temporal relations between surveillance data and various spatial or spatio-temporal regions of interest. Taking an example from the aviation domain, a low-level event of interest is when a flight enters (or leaves) a sector. In the maritime domain, a corresponding low-level event is when a vessel enters (or leaves) a protected area. By detecting these low level events as soon as possible, the task of integration/interlinking is expedited and will be performed in real-time.
- R1.2 (Interplay of in-situ and stream processing components): *In-situ Processing* is able to perform some stream processing operations near to the sources, for example low-level event detection. More complex events are computed by other streaming components (e.g., *Event Recognition and Forecasting* component), but processing data near to the sources has advantages such as reduced communication, lower latency, and reduced complexity of the application, as explained in Section 3.2.

Component	**R1.1**	**R1.2**	**R1.3**	**R1.4**	**R1.5**	**R2.1**	**R2.2**	**R2.3**	**R2.4**	**R3.1**	**R3.2**
Synopses Generator									X		
In-situ Processing	X	X									
Data Manager	X		X	X	X						
Data Integrator	X		X								
Distributed RDF store				X	X						
Trajectory Detection and Prediction						X	X	X			
Local Model Extractor							X	X			
Location Predictor							X	X			
Data Analytics						X	X	X			
Event Recognition and Forecasting										X	X
Visual Analytics		X				X	X	X	X	X	X

Table 2: Mapping between requirements and architecture.

4.2.3 Data Manager

The *Data Manager* component performs two main tasks: (a) integration of data coming from different sources (both streaming and archival) into a common representation model, and (b) distributed storage and parallel query processing over integrated spatio-temporal RDF data. In this way, *Data Manager* contributes to the following requirements:

- R1.1 (Real-time integration/interlinking of spatial and/or spatio-temporal entities): All streaming data that is going to be stored in datAcron is transformed in RDF and integrated with other data in real-time. This real-time integration task is performed by the *Data Integrator* module. For the real-time integration requirements, the main streaming data source is surveillance data, which is integrated with weather and contextual data. The output of the *Data Integrator* module is stored in the distributed RDF store, but also streamed out in order to facilitate timely access by the other components of the architecture.

- R1.3 (Integration/interlinking over stored data): Another functionality offered by the *Data Integrator* is to link trajectories and events, after they have been detected, with stored data. This refers to the various analysis results (predicted trajectories, trajectory clusters, detected or forecast events, etc.), which need to be stored and integrated with existing data, in order to be available for subsequent analysis tasks or human interpretation.

- R1.4 (Spatio-temporal RDF querying of integrated data): One of the main responsibilities of *Data Manager* is to enable access to the integrated RDF data by implementing query processing algorithms and access methods. Several challenges are associated with providing this querying functionality, in particular because of the distributed storage but also in order to allow parallel data processing and scalability. To this end, the *Data Manager* will provide query primitives and access methods for processing spatio-temporal RDF data at scale, thus covering requirement R1.4.

- R1.5 (Retrieval of spatio-temporally constrained subsets of integrated data): This requirement can be viewed as part of R1.4, since it also entails querying the integrated data. However, the focus here is on the extraction of integrated datasets based on spatio-temporal constraints. Essentially, the *Data Manager* component will provide access to integrated data for a given spatio-temporal constraint, thus serving the data analytics components with interlinked data, which is expected to facilitate improved predictions, complex event detection, as well as visual analytics.

4.2.4 Trajectory Detection and Prediction

The *Trajectory Detection and Prediction* component is responsible for trajectory prediction and analytics with respect to trajectories. Thus, it addresses the following architectural requirements:

- R2.1 (Computation of trajectory similarity and clustering): This is a basic building block for higher level data analytics, including trajectory prediction. A fundamental operation is computing the similarity of a pair of trajectories, which requires the specification of a similarity function. Different choices for similarity functions do exist; others are defined based only on the spatio-temporal positions of each trajectory, whereas others are *semantic* similarity functions, which take into account additional information, as for example the enriched information which is produced in datAcron. Capitalizing on appropriate trajectory similarity functions, the component will also perform trajectory clustering in order to discover popular "routes", and to group trajectories based on various criteria.

- R2.2 (Pattern discovery): Another important part of data analytics is pattern discovery with respect to trajectories. Essentially, the objective here is to identify sets of trajectories that show similar behavior. Identifying such movement patterns from historical data is particularly significant, as it may assist in the characterization of trajectories in real-time, as long as a trajectory can be classified to belong to a discovered pattern. This component is going to discover movement patterns as specified by the requirement R2.2.

- R2.3 (Prediction of trajectories and locations): Last, but not least, this component will perform prediction of trajectories and future locations. In the case of trajectory prediction, the aim is to predict the future movement of the object either in the short-term or in the long-term. In the case of location prediction, the objective is to identify the location (not necessarily the complete movement) of a moving object at some time in the future. Both predictions are going to be supported by the *Trajectory Detection and Prediction* component.

4.2.5 Event Recognition and Forecasting

The *Event Recognition and Forecasting* component targets the need to detect and forecast complex events related to the movement of moving objects.

- R3.1 (Event detection and forecasting in the maritime domain): The first task is to identify events of interest with respect to areas, vessels and trajectories in the maritime domain. With respect to vessels, the goal is to identify a vessel's characteristics, such as its type or whether it has been black-listed. With respect to areas, the goal is to identify areas with high fishing pressure. In order to achieve this, we first need to characterize the trajectories of vessels moving in such areas, e.g. determining whether a vessel is actively fishing.

- R3.2 (Event detection and forecasting in the aviation domain): The second task is event detection and forecasting in the aviation domain. With respect to flow management, the goal is to predict regulations that need to be imposed, detect affected flights and detect capacity/demand imbalances. With respect to flight planning, the goal is to detect a number of significant events for an aircraft's trajectory, such as: Terminal Boundary Crossing Point, Hold Entry, Hold Exit, Fly-by, Fly-over, STAR Entry, SID Entry, Aircraft not following planned route.

4.2.6 Visual Analytics

The *Visual Analytics* component aims at exploratory and interactive analysis of data, in order to enable the task of human interpretation, which is necessary in the case of Big Data. As explained also in D1.1, visual analytics does not represent a single, specific analysis technique but rather a methodological approach to gain insight into large, complex, noisy and often conflicting data, to develop and test hypotheses, and to build and understand complex analytical models. The key aspect is the collaborative work between the computer and the human analyst, whereby the human expert imparts background knowledge about the current analysis task's context and reasoning on the overall analytical process.

As such, the *Visual Analytics* component expands upon automated analyses developed and applied in the context of *Trajectory Detection and Prediction* and *Event Recognition and Forecasting*. Therefore, visual analytics approaches will operate on the same data types and structures identified for the afore-described components. This includes the raw data and enriched data as applicable for model building, as well as analytical models and their current parameterizations for model understanding, verification, and refinement.

5 The datAcron Software Architecture

This section describes the software architecture that realizes the datAcron prototype system, in terms of software modules and their interactions. The contents of this section are subject to refinements during the next months, and eventually the concrete software design is going to be documented in deliverables D1.6 "Software Design (interim)" and D1.11 "Software Design (final)", which are due on months 18 and 30 respectively.

As a starting point, there is a clear separation between *batch processing* and *real-time processing* in datAcron. Both data processing alternatives are supported by the datAcron architecture and accommodate different use-case scenarios.

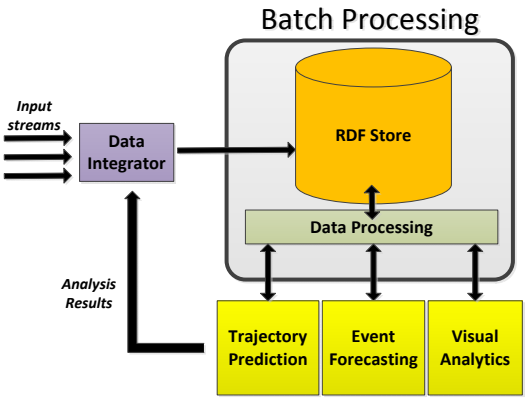

Figure 12: Batch processing.

5.1 Batch Processing

Batch processing (Figure 12) refers to the *Data Manager* component, which provides access to integrated data, both historical and fresh data. However, there is an unavoidable gap between real-time data and fresh data, due to sheer volume of data handled by the *Data Manager* and the induced latency for integrating the streaming data with the historical data. As such, it is expected that *Data Manager* provides access to data up to a certain time point in the near past, whereas the most recent data is handled in a streaming fashion. Eventually, this streaming data will update the data stored in *Data Manager*.

On top of the *Data Manager* component, other datAcron components operate and perform long-running analysis tasks. For example: the *Trajectory Detection and Prediction* component performs processing-intensive analytics tasks offline, such as clustering, classification, or others; the *Event Recognition and Forecasting* component analyzes historical data to identify patterns of events; the *Visual Analytics* component retrieves data in order to facilitate interactive analysis, data exploration and visualization tasks. All afore-described analysis tasks translate to batch processing over subsets of the historical data available in datAcron, and do not necessarily require the most recent data that arrived in the system.

At the time of this writing, the state-of-the-art solution for batch processing is Apache Spark [31, 33]. This framework will be adopted in datAcron in order to develop the batch processing functionality. In the following, it is described how Spark is going to be extended, in order to accommodate the needs and peculiarities associated with processing and analyzing trajectory data represented in RDF.

5.1.1 Processing Spatio-temporal RDF Data in Apache Spark

In datAcron, all incoming data is integrated in RDF, in compliance with the datAcron ontology. However, due to application domains, the data have a strong spatio-temporal flavour, meaning that the majority of the represented RDF resources correspond to spatial or spatio-temporal objects (positions of moving objects, regions of interest, point of interest, weather variables, etc.). Therefore, the need is raised to efficiently manage this spatio-temporal RDF data and provide query processing mechanisms.

Existing prototype systems that extend Spark mainly focus either on spatial data [27, 28, 29] only or on RDF processing [19]. In datAcron, the spatio-temporal nature of the data combined with the RDF representation poses new challenges for efficient data management and parallel processing, which go well beyond the state-of-the-art in the field. Put differently, using existing implementations of spatial operations and RDF operations is going to lead to inferior performance, as this approach cannot harness the merits of combining spatial and RDF processing at the same time. As an indicative example, better processing performance can be achieved when pruning the underlying data using both spatio-temporal and RDF filtering at the same time, while the alternative is to perform each filtering individually and then merge the results, which will clearly lead to wasteful processing. One of the core research contributions in datAcron is to extend Spark by providing a library that manages spatio-temporal RDF data in an holistic way. This extension is depicted in Figure 12 as a layer named *Data Processing* operating on top of the RDF store.

5.1.2 Distributed Storage of Spatio-temporal RDF Data

In addition, the RDF storage functionality is going to be developed by designing and implementing a distributed spatio-temporal RDF store, where different alternatives with respect to physical organization of data are going to be examined. To this end, we are going to revisit existing approaches for RDF storage in order to identify their weaknesses and limitations, thereby proposing the storage method that best fits the needs of datAcron.

Alternatives for RDF storage, inspired by relational databases, include "one triples table" [18] where all triples are stored in a single table, property tables [25] where subjects with common predicates are grouped in the same table, vertical partitioning [1] where a two-column table is created for each property, and extended vertical partitioning [19] where additional information on which subjects are joined to objects is materialized, in order to speed up processing by trading off space for execution time.

Such methods for RDF storage are going to be studied meticulously under the prism of spatio-temporal data and also in the context of NoSQL storage systems. This will lead to the design of distributed RDF store for spatio-temporal data that additionally provides primitive query operators for parallel data processing. This is another extension of Spark that is going to be provided in datAcron.

5.2 Real-time Processing

Real-time processing (Figure 13) refers to all operations in datAcron that need to be performed in real-time at *operational* latency. This dictates access to the following streams that flow in the system, including:

- the raw stream of positions of moving objects (input to the datAcron architecture)
- the stream of trajectory synopses (generated by *Synopses Generator*)
- the stream of events (generated by *In-situ Processing*)
- the stream of integrated position data with data from other sources (generated by *Data Manager*)

Moreover, the analysis results of every component are also provided in the form of output streams from each component respectively, so that they can be consumed by any other component of the system.

In order to develop the necessary real-time processing in datAcron, in compliance with the architecture of Figure 4, two main functionalities are required: (a) stream processing, and (b) stream-based communication.

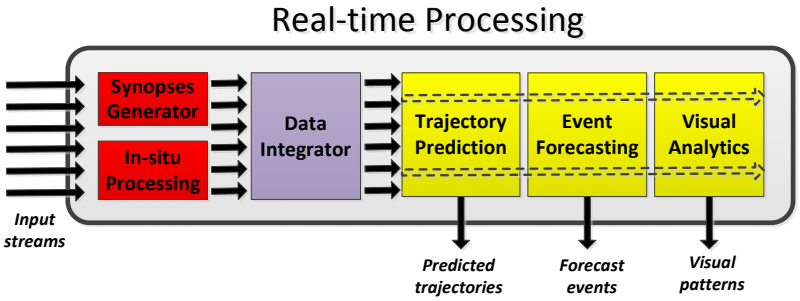

Figure 13: Real-time processing.

5.2.1 Stream Processing

Multiple operations in datAcron are performed on streaming data: the generation of trajectory synopses, data integration, event detection (low-level events and complex events), trajectory prediction, event forecasting, and real-time (interactive) visual analytics. The majority of these operations must be performed in near real-time, respecting the requirements for *operational latency*.

Based on the overview of Section 2, and taking into account the requirements for stream processing of the datAcron components, Apache Flink is going to be used primarily for implementing stream processing functionality. However, this does not exclude the usage of other stream processing frameworks (e.g., Kafka streams) for specific components of datAcron, if such a need arises.

5.2.2 Stream-based Communication

To support different stream-based functionality, components in datAcron need an interconnection infrastructure, since typically the operation of one component may rely on the output of another component. In order to support flexible interconnection and communication between components without imposing a rigid architecture, we opt for a *loosely-coupled architecture* for datAcron, which has the additional advantage that different components can be developed using different technologies.

In particular, all outputs of components are streamed out, thus allowing any other component to connect to any output stream(s) and have access to its content. In technical terms, datAcron adopts the use of Apache Kafka as messaging system to implement the interconnection infrastructure necessary to support our loosely-coupled architecture.

5.3 Combining Batch with Real-time Processing

In addition, applications in datAcron may require a combination of data from real-time and historical. In this case, data from real-time processing are going to fill the missing data (the latest data) from the batch processing and analysis. In this sense, the datAcron software architecture resembles the Lambda architecture [14]. One challenge is how to combine the results of batch processing (batch views) with the real-time processing results (real-time views), in particular because these results are not necessarily simple aggregates, but may require some additional processing in order to be merged before being ready for presentation to the end user.

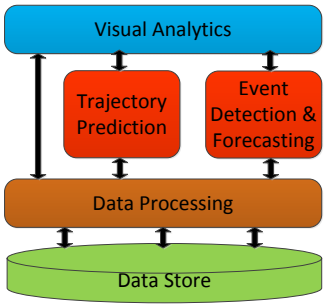

Figure 14: The datAcron software stack.

5.4 Software Modules

Figure 14 shows the datAcron software stack for realizing the advanced data analytics tasks necessary for forecasting trajectories and events. The software stack contains the following main modules, from bottom to the top:

- The Data Store

- Data processing
- Trajectory detection and prediction
- Event detection and forecasting
- Visual analytics

In more detail, right above the level of the data store, there exists a *data processing* layer whose role is to provide primitive operations for data access, thereby facilitating data fetching and retrieval that is going to support the advanced analytics. As such, all analytics can use the facilities provided by the data processing layer. Then, two separate modules realize the *trajectory prediction* and *event detection and forecasting*. These modules exploit the capabilities offered by the data processing layer, in order to efficiently load the necessary data for the subsequent data analysis tasks.

Moreover, the *visual analytics* layer plays multiple roles, both by interacting with the data processing layer as well as the analytics layers. In the former case, when exploratory visual analytics is performed, access to the underlying data must be directly provided, possibly over multiple requests and data retrieval operations, in order for the user to effectively discover what she is looking for. In the latter case, visual analytics interplays with the other analytics modules (trajectory prediction and event forecasting) in order to facilitate the exploration, parameterization, and validation by the end users of patterns, models, events, and trajectories discovered.

References

[1] Daniel J. Abadi, Adam Marcus, Samuel Madden, and Katherine J. Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB)*, pages 411–422, 2007.

[2] Divyakant Agrawal, Philip Bernstein, Elisa Bertino, Susan Davidson, Umeshwar Dayal, Michael Franklin, Johannes Gehrke, Laura Haas, Alon Halevy, Jiawei Han, H.V. Jagadish, Alexandros Labrinidis, Sam Madden, Yannis Papakonstantinou, Jignesh M. Patel, Raghu Ramakrishnan, Kenneth Ross, Cyrus Shahabi, Dan Suciu, Shiv Vaithyanathan, and Jennifer Widom. Challenges and opportunities with big data – a community white paper developed by leading researchers across the united states. Technical report, March 2012.

[3] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *PVLDB*, 8(12):1792–1803, 2015.

[4] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. The stratosphere platform for big data analytics. *VLDB J.*, 23(6):939–964, 2014.

[5] Gennady L. Andrienko, Natalia V. Andrienko, Christophe Claramunt, Georg Fuchs, and Cyril Ray. Visual analysis of vessel traffic safety by extracting events and orchestrating interactive filters. In *Proceedings of Maritime Knowledge Discovery And Anomaly Detection Workshop (MKDAD)*, 2016.

[6] Alexander Artikis, Marek J. Sergot, and Georgios Paliouras. An event calculus for event recognition. *IEEE Trans. Knowl. Data Eng.*, 27(4):895–908, 2015.

[7] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.

[8] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[9] Christos Doulkeridis and Kjetil Nørvåg. A survey of large-scale analytical query processing in mapreduce. *VLDB J.*, 23(3):355–380, 2014.

[10] Wissem Inoubli, Sabeur Aridhi, Haithem Mezni, and Alexander Jung. Big data frameworks: A comparative study. *arXiv preprint arXiv:1610.09962*, 2016.

[11] H. V. Jagadish, Johannes Gehrke, Alexandros Labrinidis, Yannis Papakonstantinou, Jignesh M. Patel, Raghu Ramakrishnan, and Cyrus Shahabi. Big data and its technical challenges. *Commun. ACM*, 57(7):86–94, 2014.

[12] Daniel A. Keim, Gennady L. Andrienko, Jean-Daniel Fekete, Carsten Görg, Jörn Kohlhammer, and Guy Melançon. Visual analytics: Definition, process, and challenges. In *Proceedings of Information Visualization - Human-Centered Issues and Perspectives*, pages 154–175. 2008.

[13] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter Heron: Stream processing at scale. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 239–250, 2015.

[14] Nathan Marz and James Warren. *Big Data: Principles and best practices of scalable realtime data systems.* Manning Publications Co., 2015.

[15] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: Interactive analysis of web-scale datasets. *PVLDB*, 3(1):330–339, 2010.

[16] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: interactive analysis of web-scale datasets. *Commun. ACM*, 54(6):114–123, 2011.

[17] Xiangrui Meng, Joseph K. Bradley, Burak Yavuz, Evan R. Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, D. B. Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. Mllib: Machine learning in Apache Spark. *CoRR*, abs/1505.06807, 2015.

[18] Thomas Neumann and Gerhard Weikum. The RDF-3X engine for scalable management of RDF data. *VLDB J.*, 19(1):91–113, 2010.

[19] Alexander Schätzle, Martin Przyjaciel-Zablocki, Simon Skilevic, and Georg Lausen. S2RDF: RDF querying with SPARQL on Spark. *PVLDB*, 9(10):804–815, 2016.

[20] Juwei Shi, Yunjie Qiu, Umar Farooq Minhas, Limei Jiao, Chen Wang, Berthold Reinwald, and Fatma Özcan. Clash of the Titans: MapReduce vs. Spark for large scale data analytics. *PVLDB*, 8(13):2110–2121, 2015.

[21] Swaminathan Sivasubramanian. Amazon dynamoDB: a seamlessly scalable non-relational database service. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 729–730, 2012.

[22] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthikeyan Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy V. Ryaboy. Storm@twitter. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 147–156, 2014.

[23] Guozhang Wang, Joel Koshy, Sriram Subramanian, Kartik Paramasivam, Mammad Zadeh, Neha Narkhede, Jun Rao, Jay Kreps, and Joe Stein. Building a replicated logging system with Apache Kafka. *PVLDB*, 8(12):1654–1655, 2015.

[24] Tom White. *Hadoop: The definitive guide.* O'Reilly Media, Inc., 2012.

[25] Kevin Wilkinson, Craig Sayers, Harumi A. Kuno, and Dave Reynolds. Efficient RDF storage and retrieval in Jena2. In *Proceedings of the First International Workshop on Semantic Web and Databases (SWDB)*, pages 131–150, 2003.

[26] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. GraphX: a resilient distributed graph system on Spark. In *Proceedings of the First International Workshop on Graph Data Management Experiences and Systems (GRADES)*, page 2, 2013.

[27] Simin You, Jianting Zhang, and Le Gruenwald. Spatial join query processing in cloud: Analyzing design choices and performance comparisons. In *Proceedings of the 44th International Conference on Parallel Processing Workshops (ICPPW)*, pages 90–97, 2015.

[28] Jia Yu, Jinxuan Wu, and Mohamed Sarwat. GeoSpark: a cluster computing framework for processing large-scale spatial data. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 70:1–70:4, 2015.

[29] Jia Yu, Jinxuan Wu, and Mohamed Sarwat. A demonstration of GeoSpark: A cluster computing framework for processing big spatial data. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 1410–1413, 2016.

[30] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the USENIX conference on Networked Systems Design and Implementation (NSDI)*, pages 2–2, 2012.

[31] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Workshop on Hot Topics in Cloud Computing*, 2010.

[32] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. In *Proceedings of the 4th USENIX Workshop on Hot Topics in Cloud Computing*, 2012.

[33] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache Spark: a unified engine for big data processing. *Commun. ACM*, 59(11):56–65, 2016.