Università degli studi di Genova

Corso di Studio in Ingegneria Informatica

Tesi di Laurea per il conseguimento del titolo di
Dottore Magistrale in Ingegneria Informatica

Big Data Streaming

Federico D'Ambrosio

Dicembre 2017

Relatore:	Prof.	Luca Oneto
Correlatore:	Dott.	Nome Cognome

Acknowledgements

The acknowledgments and the people to thank go here, don't forget to include your project advisor…

Contents

For/Dedicated to/To my…

Chapter 1

Introduction

1.1 Big Data: What are they?

Big Data, as the name suggests, are a collection of huge amount of data, that requires a sophisticated infrastructure to be managed and for its valuable information to be extracted.
It is customary to define Big Data as having three fundamental features known as the 3Vs:

Volume The quantity of data generated and stored must be huge, even in the orders of magnitude of the Brontobytes (10^{27} bytes). Every bit of data might hide some interesting piece of information. The infrastructure must always retrieve as much data as possible and store it for later use.

Velocity In the ever changing environment of our digital world, floating in a widespread network linking every device of our life (Internet of Things), data are flowing at an increasing pace. The infrastructure must be able to keep up with this rhythm, as data ages faster each day.

Variety With the rapid increase of published information, known as information explosion, data has begun to diversify more and more, and it is now unrealistic to expect just structured textual information such as classical Database tables; the new data flow includes unstructured data such as images, audio, PDFs and raw text files. The infrastructure must be able to extract information from heterogeneous sources.

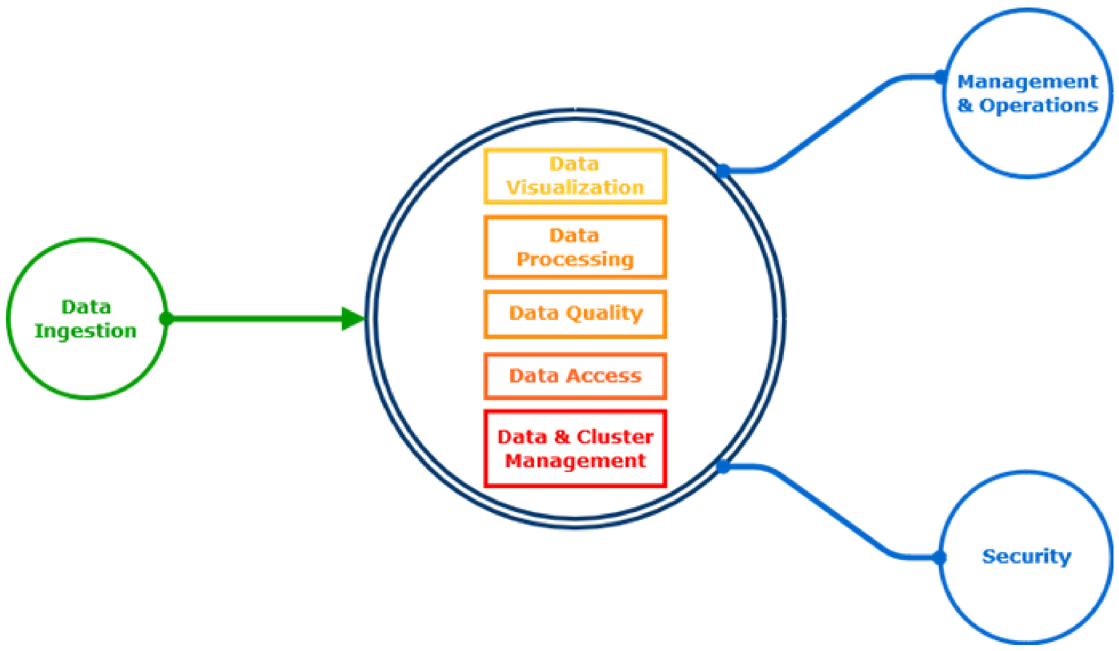

FIGURE 1.1: A model of a Big Data Infrastructure

1.2 A Big Data Infrastructure

As we have established, a Big Data Infrastructure must meet several requirements in order to function properly. To manage the complexity needed for this heavy feat, the whole system is organized in several layers; each layer may be implemented with one of several products without changing the rest of the structure.

Hardware At the foundations of the stack we find the hardware (real or virtualized), which is, in general a group of several machines called a cluster.
The machines are tied together in a single network each to one another, so that communication between them is as easy as possible and the cluster can act like a single machine with high performance and reliability.

Data & Cluster Management Over the first layer of software provided by each machine's Operating System, the cluster needs an interface, so that from an higher point of

view, like that of a developer or of an application, the whole system may seem a single entity.
That interface is provided by the Data & Cluster Management layer, it must offer:

- A distributed File System, able to be accessed like a standard one while providing high fault tolerance thanks to the duplication of data on the different machines of the cluster.
- A distributed Operating System, able to manage access to resources on the whole cluster and to schedule application jobs.

Data Access Having reached a more unified view on the cluster thanks to the previous layer, we can forget about the details of the single machines.
As we would use Databases on a normal computer, so we, on the data access layer, add a distributed Database able to store our data efficiently and for easy access on the distributed File System. Distributed Databases come in different shapes and styles ranging from the familiar relational ones to non-relational document stores passing through key-value row based stores.

Data Quality As mentioned before, our goals are to store as much data as we feasibly can and then process them: the acquisition of big data sets will often lead to a certain amount of dirty data and, what's more, these huge swaths of data, even cleaned, will be hard to be accessed or made valuable because of their lack of structure.
Therefore data must go through the Data Quality layer in order to be ready for further processing, this layer must offer applications for:

- Data Cleaning, the process of detection, correction and removal of dirty data such as corrupt or wrong records or documents.
- Data Wrangling, the process of mapping raw data to a more structured format; wrangling unstructured data also helps their future Cleaning.

Data Processing Over all the previous strata, we can finally run some applications on the Data Processing layer; those applications, executed in jobs scheduled and managed

by the distributed Operating System, process the clean data stored on the distributed Database or on the distributed File System.
This layer provides for distributed processing frameworks where applications can be deployed and run seamlessly on the cluster as a whole, in-memory or on disk, in batches or in real time.
Processed data can then be stored again on the File System or on the Database, formatted or enriched in information for later visualisation or further elaboration.

Data Visualisation The topmost layer of the main part of the stack is the Data Visualisation layer which supplies a Visualizing software, whose job is to load data stored on the Database or File System, processed and enriched in the previous layer, and make its information intelligible to humans through the use of images, which can convey several dimensions on a 2D plot thanks to their geometric shape, position and size and their colours.

Data Ingestion Orbiting our main stack we find other transversal layers, first among them the Data Ingestion layer.
As described, Big Data infrastructures must be able to handle huge amounts of incoming data at high speed and in real time; this layer provides for applications that can manage incoming real time traffic of data from a source to the cluster, through filtering, mediating and queueing. It might also involve real time cleaning, wrangling and processing applications.

Management & Operations The complexity of the infrastructure with its numerous layers and strong connection between each of them requires an intuitive interface for humans to monitor, manage and maintain the whole infrastructure.
This layer provides a one stop shop for all cluster related operations, showing a dashboard for all the services running on it and supplying an easy way to perform maintenance and updating the system.

Security Dulcis in fundo, the valuable information we have extracted thus far, must not be accessible to our competitors or other malicious agents, therefore the cluster must

be secured against external attacks.
Also, we need to prevent users from performing operations they are not allowed to perform and to give access to information selectively to each user. This layer must then provide:

- A secure perimeter, so that only authenticated and authorized users might access services on the infrastructure.
- Tools for data governance and access control, so that each user is allowed to and only to the data he or she needs.
- Tools for auditing so that failed attempts at breaching the perimeter and at performing disallowed operations can be investigated.

Chapter 2

Data Management & Data Access

2.1 HDFS

2.2 YARN

2.3 SQL: Hive

Apache Hive is a relational database for Big Data, developed as a part of the Hadoop environment to provide fast access to huge data sets through its own query language, **HiveQL**, and through several possible execution engines such as Apache Tez, MapReduce and Apache Spark. Hive, as of recently, added support for ACID transactions, making it viable as a data storage for more complex endeavours, including streaming applications.

2.3.1 Components

Hive architecture can be divided in five main components:

- **Shell/UI**: Beeline is the frontend tool for interactive querying on the database. Hive supports connections via its own JDBC driver, allowing easy integration with clients and user applications.
- **Driver**: The component which receives the queries. This component implements the notion of session handles and provides execute and fetch APIs modeled on JDBC/ODBC interfaces.

- **Metastore**: It stores metadata about HDFS file locations and the table schemas, but also column and column type information and the serializers and deserializers necessary to read and write data.
- **Compiler**: It manages query parsing, planning and optimization, does semantic analysis on the different query blocks and query expressions generating, eventually, an execution plan with the help of the table and partition metadata looked up from the metastore. For what concerns the parsing, HiveQL is a SQL extension, compliant with the SQL:2011 standard.
- **Execution Engine**: Hive uses Apache Tez as a default execution engine which allows low latency querying through its LLAP (Low Latency Analytical Processing) daemons: persistent processes running on YARN enabling the system to avoid the overhead caused by the container deployment for query executions. Other execution engines usable by Hive are, as already mentioned, MapReduce and Spark. The first one is needed, as of Hive 2.1, in order to use Hive Streaming API, while the other uses its own LLAP daemons, similarly to Tez, and still lags behind, performance wise.

2.3.2 Hive Low Latency Processing on Tez

Starting from Hive2, OLAP, OnLine Analytical Processing support has been introduced as default mode for query execution on top of Apache Tez.
This functionality has been implemented on top of YARN where, while deploying the endpoint for the clients queries (HiveServer2), two applications are deployed:

- a Tez query coordinator application, which deals with a single query execution planning, a series of Map and Reduce operations, but also the concurrency of many queries, if needed;
- a Slider [1] application that, together with the persistent daemons containers, deals with the actual processing of the single operation that needs to be executed.

With respect to the usual query execution, this architecture allows:

[1]Apache Slider is an application which allows dynamic deployment and monitoring of YARN applications

- a critical decrease in the latency caused by the creation of the YARN containers needed for the query execution, which is usually the biggest time consuming task.
- parallel and concurrent query execution, shared between all the daemons instances, taking also advantage of the In-Memory Cache of the single daemon, especially useful if different queries need to access the same data.

The introduction of LLAP/OLAP daemons provided Hive2 with an average 2600% performance gain when compared to Hive1 using Tez, on a dataset of 1 TB.

2.3.3 HiveQL

HiveQL, which stands for Hive Query Language, is a SQL:2011 compliant language used for query expressions in Hive. Together with all the features coming from the SQL standard, it introduces concepts like external tables and table bucketing in its DDL[2], together with the possibility to use User Defined Functions for custom aggregations and operators.

Data Definition Language

HiveQL Data Definition Language allows creation and alteration of databases, tables and indices.
When it comes to DBs, the classic statements `CREATE`, `ALTER` & `DROP` are available for creation, alteration and deletion of databases, together with the possibility to specify ownership, filesystem location and other custom properties.
Concerning tables, Hive DDL provides the ability to define external tables, located in the filesystem somewhere other than on the default warehouse, which can be read and queried like a normal table. In addition, it is possible to define constraints on column keys, such as foreign and primary keys, partitioning, bucketing, skewing and sorting for optimization purposes. It allows to define the kind of deserialization Hive needs to use in order to read the data stored, according to their file format with the `ROW FORMAT` clause.

For example, the statement

[2]DDL: Data Definition Language, subset of a grammar which specifies the syntactic rules for the creation and alteration of objects such as databases, tables and indices.

```
CREATE TABLE IF NOT EXISTS log_table(id string, count int, time timestamp)
PARTITIONED BY (date string)
CLUSTERED BY id SORTED BY (count DESC) INTO 10 BUCKETS
SKEWED BY id ON 3, 4, 10
STORED AS ORC;
```

creates a table named ”log_table”, using the default database, stored in ORC format partitioned according to a certain user-input string ”date”, in 10 ORC files sorted in descending order, where values are skewed on the id values 3, 4 and 10.

Data Manipulation Language

HiveQL Data Manipulation Language allows to modify data in Hive through multiple ways:

- `LOAD` allows file loading into Hive tables from HDFS or local filesystem.
- `INSERT` allows to insert query results into Hive tables or filesystem directories. In case it's needed, it is possible to overwrite or insert them into a dynamically created partition.
- `UPDATE` and `DELETE` allow to update or delete values in tables supporting Hive Transactions.
- `MERGE` allows to merge files belonging to the same table, if it supports Hive Transactions.
- `IMPORT` and `EXPORT` allow importing and exporting of both table values and metadata, to use them with other DBMS.

2.3.4 Warehouse

HDFS is the default physical storage of database and tables in Hive, but support for S3 and any other HDFS compatible filesystems is available. Since all of the files are stored on a distributed filesystem, redundancy and fault tolerance are granted when it comes

to data integrity. Hive default storage format is ORC, a compressed format able to decrease file size up to 78% with respect to normal Text Files.
Hive has built-in direct serialization and deserialization of CSV, JSON, AVRO and Parquet files as row formats, and allows to easily add custom SerDe[3] components.

Hive Data Model

Data in Hive is organized into:

- **Tables** – These are analogous to Tables in Relational Databases. Tables can be filtered, projected, joined and unioned. Additionally all the data of a table is stored in a directory in its default warehouse, usually HDFS. Hive also supports the notion of external tables wherein a table can be created on pre-existing files or directories in HDFS by providing the appropriate location to the table creation DDL. The rows in a table are organized into typed columns similar to Relational Databases.

- **Partitions** – Each Table can have one or more partition keys which determine how the data is stored, for example a table `T` with a date partition column `ds` had files with data for a particular date stored in the `<table location>/ds=<date>` directory in HDFS. Partitions allow the system to prune data to be inspected based on query predicates, for example a query that is interested in rows from `T` that satisfy the predicate `T.ds = '2008-09-01'` would only have to look at files in `<table location>/ds=2008-09-01/` directory in HDFS.

- **Buckets** – Data in each partition may, in turn, be divided into Buckets based on the hash of a column in the table. Each bucket is stored as a file in the partition directory. Bucketing allows the system to efficiently evaluate queries that depend on a sample of data (these are queries that use the SAMPLE clause on the table).

Apart from primitive column types (integers, floating point numbers, generic strings, dates and booleans), Hive also supports arrays and maps. Additionally, users can compose their own types programmatically from any of the primitives, collections or other user-defined types. The typing system is closely tied to the SerDe and object inspector

[3]SerDe: Serialization/Deserialization

interfaces. Users can create their own types by implementing their own object inspectors, and using these object inspectors they can create their own SerDes to serialize and deserialize their data into HDFS files).
These two interfaces provide the necessary hooks to extend the capabilities of Hive when it comes to understanding other data formats and richer types. Built-in object inspectors like `ListObjectInspector`, `StructObjectInspector` and `MapObjectInspector` provide the necessary primitives to compose richer types in an extensible manner. For maps (associative arrays) and arrays useful built-in functions like size and index operators are provided. The dotted notation is used to navigate nested types.

2.3.5 Hive vs SQL Server

2.4 NoSQL: HBase & Cassandra

Chapter 3

Data Ingestion

3.1 Apache Kafka

3.2 Apache NiFi

Chapter 4

Data Processing

Data Processing is the layer immediately following the Data Quality: once data have been formatted and cleansed, adapted then to our needs, we can start analysing and processing them through the use of adequate frameworks, in order to extract from, transform and enrich them for visualization purposes.

4.1 Tecniche per il Data Processing

4.1.1 Batch Processing

4.1.2 Graph Processing

4.1.3 Stream Processing

Streaming data processing differentiates itself from the classic data processing for its use of unbounded datasets, that is a continuous and endless data flow which needs adequate abstractions in order to be able to apply operators or transformations like Map, Reduce and Filter operations.

Generally, there are 2 execution models usable when to approach an unbounded dataset:

- **Streaming**: continuous elaboration as long as data are being produced.

- **(Micro-)Batch**: finite time execution, which releases resources when the batch processing ends.

A Batch based execution is feasible when requirements on the state management, in-order consuming and windowing are not present or very relaxed and, thus, a Streaming approach is generally favoured because of the conceptual paradigm which defines it: Dataflow Programming.

Dataflow Programming

Dataflow Programming is a programming paradigm which models an application as a Direct Acyclic Graph (DAG) and thus it differentiates itself very heavily from the Imperative Programming (or Control Flow), which models a program like finite sequence of operations.
This paradigm emphasizes the continuous flow of data between operators, defined as Black Boxes with explicit input and output ports used to connect with other operators. An operation runs as soon as all of its inputs become valid. Thus, dataflow languages are inherently parallel and can work well in large, decentralized systems[1].

4.2 Apache Spark

4.3 Apache Flink

Apache Flink is an open source framework developed for the continuous and distributed processing of data flows. Based on the Dataflow Programming model, it provides a set of abstractions specifically designed for real time stream processing.

4.3.1 Abstraction levels

[1]**Johnston:2004:ADP:1013208.1013209**.

- **Stateful Streaming**: It's the lowest abstraction layer, which allows developers to freely manage data flows and their processing, use fault-tolerant states and callback registration on events.
- **Core API**: it's the basic API, which is divided between **DataStream API**, specifically conceived for bounded and unbounded dataset processing, and **DataSet API**, implemented as a special case of the first API set, used only for bounded datasets. These 2 APIs offer transformations and operations like unions, aggregations, windowing and state management, needed for the data flow upon which they are applied.
- **Table API**: it's a declarative DSL [2] which follows the extended relational model and offers the possibility to model a data stream like a table upon which it's possible to execute operations like projections, aggregations and groupings. Despite being less expressive with respect to the Core API, they allow for a greater compactness when it comes to describing the operations to be executed on the data.
- **SQL**: it's the highest level of abstraction offered by Flink and interacts directly with the Table APIs in order to provide a representation of the application being developed through SQL queries. It can be executed on the tables defined by the Table API, directly on the data.

4.3.2 Programs Dataflows

The base elements of a Flink application are the following:

- **Streams**, data structures containing data.
- **Transformation Operators**, which can change the stream, for example, via aggregations, groupings, mappings and reductions.
- **Sources**, they are primary data entry points and can be files or records from queues like Kafka. They are the starting nodes of the application DAG.

[2] Domain Specific Language: a language with a limited expressiveness which focuses on a given domain of use.

- **Sinks**, They are the application output, being it a file, any process or another queue. They are the ending nodes of the application DAG.

```
val lines = env.addSource(new Consumer[String](...))

val events = lines.map(line -> parse(line))

val stats = events
    .keyBy("id")                          // Stream is partitioned by the key "id"
    .timeWindow(Time.seconds(10))         // All the events in the given window are
    .apply(MyWindowAggregationFunction()) // All the records are aggregated accordin

stats.addSink(new RollingSink(path))
```

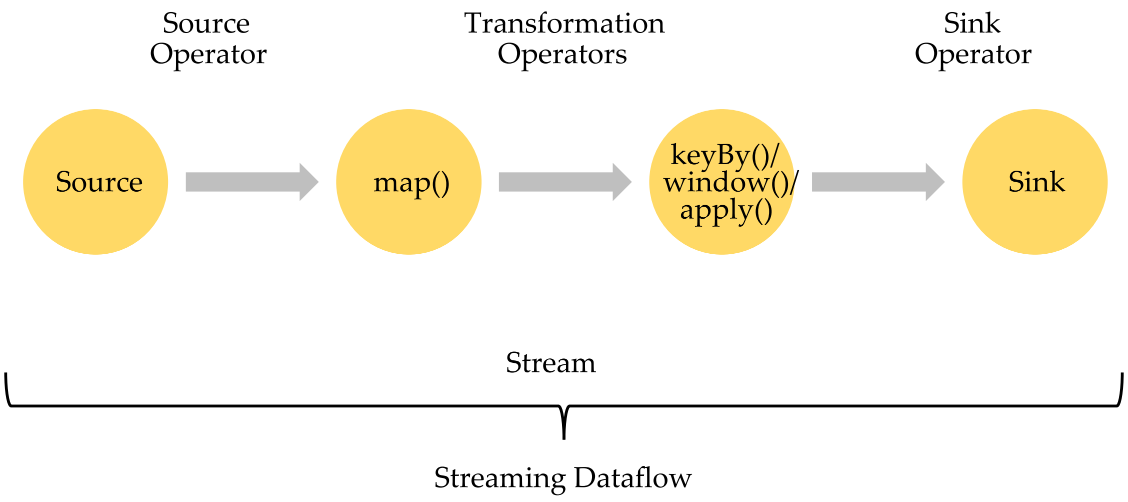

FIGURE 4.1: DAG outlining the previous code snippet

Parallelism

Applications using Flink are inherently parallels and distributed. During execution, a stream may be divided in more than one partition; each operator can have more than a single sub-task operating on data, each one independent from the other and executed on different threads or, if possible, on different machines or containers.
The parallelism of a task is the parameter indicating the number of subtasks an operator can have and, consequentially, the number of partitions of the stream getting outputted from that operator.

Streams can transport data between a pair of operators following two different patterns: One-to-one pattern and redistributing pattern:

- **One-to-one streams** preserve the stream partitioning and the order of the elements passed to the following operator. For example, the subtask[0] of a map() operator will get to see the same elements produced by the subtask[0] of the previous Source operator.
- **Redistributing streams** change their partitioning, sending the data to different subtasks, based on the transformation being applied. For example, a keyBy() operator repartitions the stream using the hash of the chosen key. In this case, the elements' order will be preserved only between neighbouring operations pairs, and won't be ever guaranteed being the same in the following operators.

4.3.3 Flink Extensions

FlinkCEP: Complex Events Processing

FlinkCEP is a Flink extension adding the possibility to analyse events pattern in a DataStream, thanks to the Pattern API. This API allows the definition of patterns to be found in a stream and their selection in order to create a new DataStream of Alerts.

```
val input: DataStream[Event] = ...
```

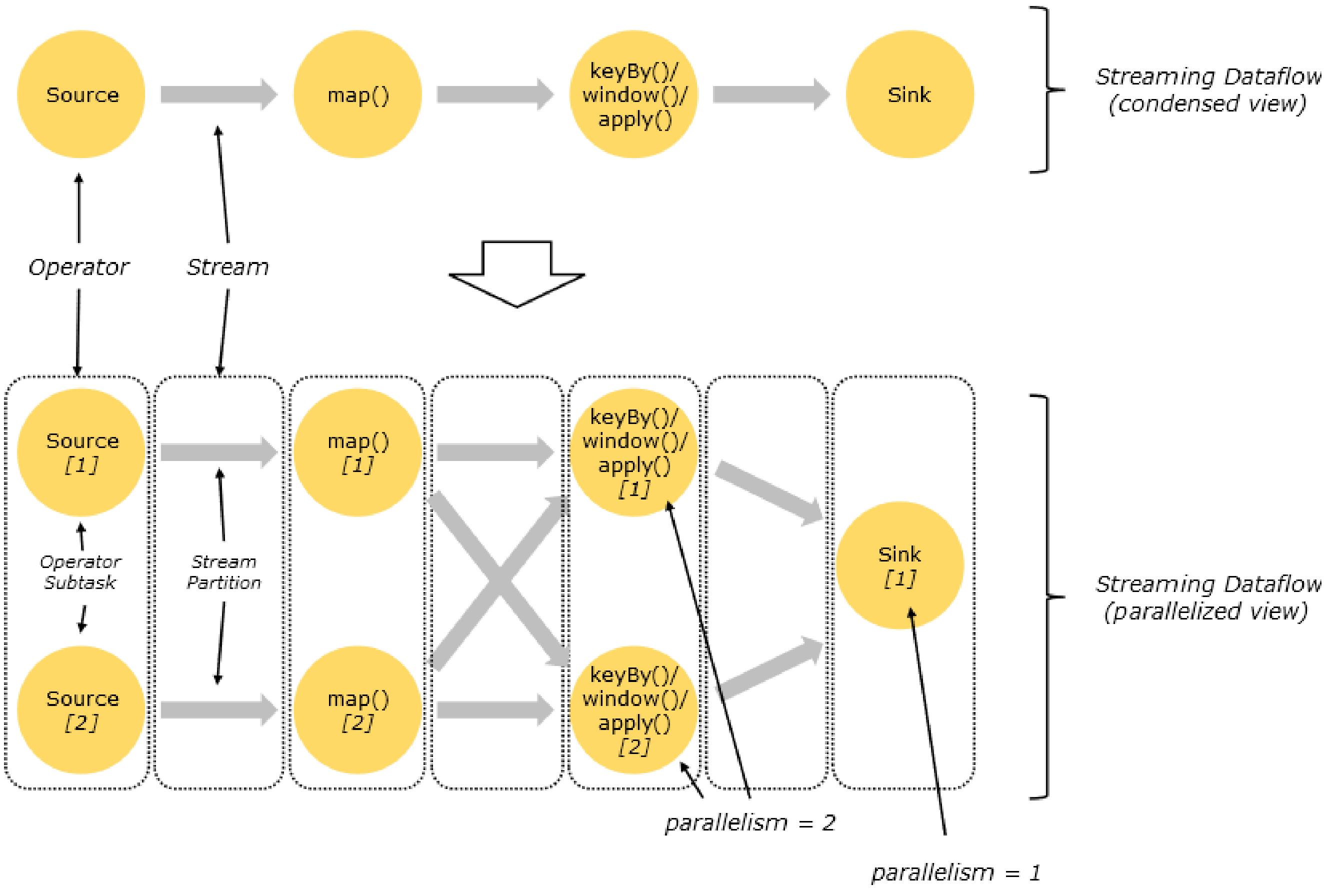
Source
map()
keyBy()/ window()/ apply()
Sink
Streaming Dataflow (condensed view)
Operator
Stream
Source [1]
map() [1]
keyBy()/ window()/ apply() [1]
Operator Subtask
Stream Partition
Sink [1]
Streaming Dataflow (parallelized view)
Source [2]
map() [2]
keyBy()/ window()/ apply() [2]
parallelism = 2
parallelism = 1

FIGURE 4.2: DAG with Parallelism=2

```
val pattern = Pattern.begin("start").where(_.getId == 42)
.next("middle").subtype(classOf[SubEvent]).where(_.getVolume >= 10.0)
.followedBy("end").where(_.getName == "end")

val patternStream = CEP.pattern(input, pattern)

val result: DataStream[Alert] = patternStream.select(createAlert(_))
```

SNIPPET 4.1: An example of Pattern API usage

In FlinkCEP, a **Pattern**, similarly to a pattern of a Regular Expression, can be **single** or **iterative**. For example, in a pattern like `"a b+ c? d"`, `a`, `c?` e `d` are single patterns, while `b+` is iterative.

In a Pattern there can be, analogously to Regular Expressions, Quantifiers and Conditions.

Quantifiers Iterative patterns can be specified with the following self explanatory methods:

- `pattern.oneOrMore();`
- `pattern.times(#ofTimes);`
- `pattern.optional()`

Conditions Each pattern can have additional conditions specified, which can be related to a property of an incoming event, or the contiguity to other events.

Methods `pattern.where()` and `pattern.or()` can be used to specify `IterativeConditions`, `SimpleConditions` or a combination of multiple conditions. In this case, combinations can be specified concatenating all the previous methods, including the quantifiers, optionally adding more constraints on the contiguity of the same pattern by calling `consecutive()`, for the strict contiguity, and `allowCombinations()` for the relaxed

non-deterministic contiguity.
Some examples:

```
middle.oneOrMore().where(
    (value, ctx) => {
        lazy val sum = ctx.getEventsForPattern("middle")
                          .asScala.map(_.getPrice).sum
        value.getName.startsWith("foo") && sum + value.getPrice < 5.0
    }
)
```

Snippet 4.2: Iterative Condition

```
//Example of conjunction through where concatenation
start.where(event => event getName startsWith "foo" ) )
     .where(event => event getTotal equals 50  )

start.subtype(classOf[SubEvent]).where(subEvent => ... /* some condition */)
```

Snippet 4.3: Conditions combinations

Pattern Combination Similarly to conditions combination, pattern combination is possible through the consecutive calls to the following methods allowing to specify the contiguity constraints which a given pattern should satisfy:

- `next()`, for strict contiguity.
- `followedBy()`, for relaxed contiguity.
- `followedByAny()`, for non deterministic relaxed contiguity.
- `notNext()`, NOT pattern with strict contiguity.

- `notFollowedBy()`, NOT pattern with relaxed contiguity.

Each pattern combination can be followed by a time constraint specified by concatenating `within(Time)`.

Pattern Detection Once a pattern sequence has been specified, it needs to applied to the target `Datastream`, creating this way a `PatternStream`. On this new stream `select` or `flatSelect` operations need to be applied in order to enable further processing on the events satisfying the pattern.

Other extensions: Flink Gelly and FlinkML

In addition to the aforementioned library for complex event handling, Flink makes specific sets of API available, which provide useful tools when it comes to enable advanced processing on data.

Flink Gelly is an API based on the Flink `Dataset`s API specifically developed for Graph Processing. Graphs are stored as a finite set of vertices and edges, a special case of a `Dataset`. This library allows the application of operators like Map, Filter, Intersect and Union, Mutations like edge and vertex removal, but also advanced techniques belonging to Graph Theory like *Iterative Graph Processing* (Vertex-centric, Scatter-Gather and Gather-Sum-Apply), *Single Source Shortest Paths*, *Clustering* and Similarity analysis.

FlinkML is a set of tools which allows the application of Machine Learning techniques on Flink data streams, including both algorithmic implementations for supervised (SVMs and Multiple Linear Regressors) and unsupervised (K-nearest neighbours) learning, Data Preprocessing and Recommendation (ALS)

Chapter 5

Data Visualization

Chapter 6

Air Traffic monitoring: a use case

6.1 Introduction

6.2 Infrastructure overview

6.3 Deployment & Operations

6.4 Development

6.4.1 Ingestion

6.4.2 Processing

6.4.3 Serving & Security

6.4.4 Visualization