NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATION

BSc THESIS

Streaming Data Processing Frameworks

Aikaterini – M - Ntenti

Supervisor: Gunopoulos Dimitrios, Professor NKUA

ATHENS

MAY 2017

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Πλαίσια Επεξεργασίας Συνεχούς Ροής Δεδομένων

Αικατερίνη – Μ - Ντέντη

Επιβλέπων: Γουνόπουλος Δημήτρης, Καθηγητής ΕΚΠΑ

ΑΘΗΝΑ

ΜΑΙΟΣ 2017

BSc THESIS

Streaming Data Processing Frameworks

Aikaterini – M - Ntenti
S.N.: 1115201000130

SUPERVISOR: Gunopoulos Dimitrios, Professor NKUA

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Πλαίσια Επεξεργασίας Συνεχούς Ροής Δεδομένων

Αικατερίνη Μ. Ντέντη
Α.Μ.: 1115201000130

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Γουνόπουλος Δημήτρης,** Καθηγητής ΕΚΠΑ

ABSTRACT

The scope of this thesis is to review in detail some of the most significant Apache Streaming Processing Frameworks: Apache Spark, Apache Storm, Apache Flink, Apache Samza and Apache Kafka. Each one of these tools is designed in a specific way, but every one of them is used for the same purpose: streaming data processing, manipulation and handling. In this thesis, we go over their features and concepts and analyze in detail the way they work. Moreover, we provide a comparison report which includes the major differences and similarities between them. After obtaining a high level idea of how they differ as frameworks, we provide the Word Count example, which is similar to the Hello World example, but for streaming data programming. In the end, we conclude on how the Apache Streaming Processing Frameworks contribute to the world and answer the following questions:

- How and why enterprises use the tools?
- Which are the Use Cases?
- Is there a possible integration between them?

Key words: streaming data processing, concepts, WordCount example, integration, Apache streaming Processing frameworks

ΠΕΡΙΛΗΨΗ

Ο σκοπός αυτής της πτυχιακής εργασίας είναι να αναλύσει με λεπτομέρεια τα πιο σημαντικά Πλαίσια Επεξεργασίας Συνεχούς Ροής Δεδομένων: Apache Spark, Apache Storm, Apache Flink, Apache Samza και Apache Kafka. Κάθε ένα από αυτά τα εργαλεία, είναι σχεδιασμένο με έναν ξεχωριστό τρόπο, ωστόσο κάθε ένα από αυτά χρησιμοποιείται για τον ίδιο σκοπό: επεξεργασία συνεχής ροής δεδομένων, επεξεργασία και χειρισμός τους. Στην πτυχιακή αυτή, αναφέρουμε τα χαρακτηριστικά και τις έννοιές και αναλύουμε λεπτομερώς τον τρόπο με τον οποίο λειτουργούν. Επιπλέον, παρέχουμε μια έκθεση σύγκρισης που περιλαμβάνει τις μεγάλες διαφορές και ομοιότητες μεταξύ τους. Αφού αποκτήσετε μια μεγαλύτερη ιδέα για το πώς διαφέρουν ως πλαίσια, παρέχουμε το παράδειγμα Word Count, το οποίο είναι παρόμοιο με το παράδειγμα Hello World, αλλά αφορά τον προγραμματισμό συνεχούς ροής δεδομένων. Τέλος, καταλήγουμε στο συμπέρασμα για το πώς τα πλαίσια επεξεργασίας συνεχούς ροής δεδομένων της Apache συμβάλλουν στον κόσμο και απαντούν στις ακόλουθες ερωτήσεις:

- Πώς και γιατί οι επιχειρήσεις χρησιμοποιούν τα εργαλεία;
- Ποιες είναι οι περιπτώσεις χρήσης;
- Υπάρχει πιθανή ενοποίηση μεταξύ τους;

Λέξεις-κλειδιά: επεξεργασία δεδομένων ροής, έννοιες, παράδειγμα WordCount, ενοποίηση, πλαίσια επεξεργασίας συνεχούς ροής δεδομένων της Apache

CONTENTS

PREFACE 11

1. INTRODUCTION 12

2. APACHE STORM OVERVIEW 14

2.1 What is Apache Storm 14

2.2 Apache Storm Concepts 14

3. APACHE SPARK STREAMING OVERVIEW 16

3.1 What is Apache Spark Streaming 16

3.2 Spark Streaming Concepts 16

4. APACHE KAFKA OVERVIEW 20

4.1 What is Apache Kafka 20

4.2 Kafka basic concepts 20

5. APACHE FLINK OVERVIEW 24

5.1 What is Flink 24

5.2 Apache Flink Basic Concepts 24

6. APACHE SAMZA OVERVIEW 28

6.1 What is Samza 28

6.2 Apache Samza Basic Concepts 28

7. SIMILARITIES AND DIFFERENCES BETWEEN THE APACHE STREAMING ENGINES 30

7.1 Word Count example in Storm 31

7.2 Word Count example in Spark 31

7.3 Word Count example in Flink 32

7.4 Word Count example in Kafka 32

7.5 Word count example in Samza 33

8. STREAMING TOOLS USES CASES AND INTEGRATION IN THE BUSINESS FIELD 34

9. ΣΥΜΠΕΡΑΣΜΑΤΑ 35

REFERENCES 36

LIST OF IMAGES

Figure 1: DStream and series of RDDs ..... σελ. 17
Figure 2: List of Transformations ..... σελ. 18
Figure 3: Output Operations ..... σελ. 18
Figure 4: Kafka's APIs ..... σελ. 21
Figure 5: Anatomy of a topic ..... σελ. 21
Figure 6: Consumers in Kafka ..... σελ. 22
Figure 7: Example Dataflows ..... σελ. 25
Figure 8: Word Count in Storm ..... σελ. 31
Figure 9: Word Count in Spark ..... σελ. 32
Figure 10: Word Count in Flink ..... σελ. 32
Figure 11: Word Count in Kafka ..... σελ. 33
Figure 12: Word Count in Samza ..... σελ. 33

LIST OF TABLES

Table 1: Table with results of the comparison **.....** σελ. 30

PREFACE

This thesis has been written to fulfill the graduation requirements and get a bachelor in Informatics and Telecommunications in University of Athens. I was engaged in researching and writing this thesis from October 2016 to April 2017.I would like to thank my supervisor for their guidance and support during this process.

1. INTRODUCTION

The demand for stream processing is increasing a lot these days. The reason is that often processing big volumes of data is not enough. Data has to be processed fast and needs to be analyzed. The speed at which data is generated, consumed, processed, and analyzed is increasing at an unbelievably rapid pace.

The industries demand data processing and analysis in real-time. Traditional big data-styled frameworks such as Apache Hadoop is not well-suited for these use cases. As a result, multiple open source projects have been started in the last few years to deal with the streaming data. All were designed to process a never-ending sequence of records originating from more than one source.

In contrast to the traditional database model where data is first stored and indexed and then subsequently processed by queries, stream processing takes the inbound data while it is in flight, as it streams through the server. Stream processing also connects to external data sources, enabling applications to incorporate selected data into the application flow, or to update an external database with processed information.

A stream processing solution has to solve different challenges:

- Processing massive amounts of streaming events (filter, aggregate, rule, automate, predict, act, monitor, alert)
- Real-time responsiveness to changing market conditions
- Performance and scalability as data volumes increase in size and complexity
- Rapid integration with existing infrastructure and data sources: Input (e.g. market data, user inputs, files, history data from a DWH) and output (e.g. trades, email alerts, dashboards, automated reactions)
- Fast time-to-market for application development and deployment due to quickly changing landscape and requirements
- Developer productivity throughout all stages of the application development lifecycle by offering good tool support and agile development
- Analytics: Live data discovery and monitoring, continuous query processing, automated alerts and reactions
- Community (component / connector exchange, education / discussion, training / certification)
- End-user ad-hoc continuous query access
- Alerting
- Push-based visualization

From Storm to Kafka, there are over a dozen open source projects in various stages of completion, such as:

- Apache Storm
- Apache Flink
- Apache Spark
- Apache Kafka
- Apache Samza

which we are going to analyze further.

2. APACHE STORM OVERVIEW

2.1 What is Apache Storm

Apache Storm is a distributed stream processing computation framework written in the Clojure programming language. Originally created by Nathan Marz and team at BackType, the project was open sourced after being acquired by Twitter. The initial release was on 17 September 2011.

A Storm application is designed as a topology in the shape of a directed acyclic graph with spouts and bolts acting as the graph vertices. It uses spouts and bolts to define information sources and manipulations to allow batch, distributed processing of streaming data. Edges on the graph are named streams and direct data from one node to another. Together, the topology acts as a data transformation pipeline.

The general topology structure is similar to a MapReduce job, with the main difference being that data is processed in real time as opposed to in individual batches. Additionally, Storm topologies run indefinitely until killed.

2.2 Apache Storm Concepts

The main concepts of Storm are listed below:

1. Topologies
2. Streams
3. Spouts
4. Bolts
5. Stream groupings
6. Reliability
7. Tasks
8. Workers

1**.** A Storm topology is a graph of spouts and bolts that are connected with stream groupings. It is a real-time application and similar to a MapReduce job. One major difference is that a MapReduce job eventually finishes, whereas a topology runs forever or until you kill it.

2. The stream is the core abstraction in Storm. A stream is an unbounded sequence of tuples that is processed and created in parallel. Streams are defined with a schema that names the fields in the stream's tuples. By default, tuples can contain integers, longs, shorts, bytes, strings, doubles, floats, booleans, and byte arrays. Every stream is given an id when declared.

3. A spout is a source of streams in a topology. Generally spouts will read tuples from an external source and emit them into the topology. Spouts can either be reliable or unreliable. A reliable spout is capable of replaying a tuple if it failed to be processed by Storm, whereas an unreliable spout forgets about the tuple as soon as it is emitted. Spouts can emit more than one stream. The main method on spouts is nextTuple().

4. Bolts are logical processing units. They can do anything from filtering, functions, aggregations, joins, talking to databases, and more. Spouts pass data to bolts and bolts process and produce a new output stream. They can, also, do simple stream transformations. Doing complex stream transformations often requires multiple steps and thus multiple bolts.

Bolts can emit more than one stream. The main method in bolts is the execute method which takes in as input a new tuple.

5. A stream grouping defines how the streams should be partitioned among the bolt's tasks. There are eight built-in stream groupings in Storm, and somebody can also implement a custom stream grouping:

Shuffle grouping: Tuples are randomly distributed across the bolt's tasks in a way such that each bolt is guaranteed to get an equal number of tuples.

Fields grouping: The stream is partitioned by the fields specified in the grouping. For example, if the stream is grouped by the "user-id" field, tuples with the same "user-id" will always go to the same task, but tuples with different "user-id"s may go to different tasks.

Partial Key grouping: The stream is partitioned by the fields specified in the grouping, like the Fields grouping, but are load balanced between two downstream bolts, which provides better utilization of resources when the incoming data is skewed. This paper provides a good explanation of how it works and the advantages it provides.

All grouping: The stream is replicated across all the bolt's tasks. Use this grouping with care.

Global grouping: The entire stream goes to a single one of the bolt's tasks. Specifically, it goes to the task with the lowest id.

None grouping: This grouping specifies that you don't care how the stream is grouped. Currently, none groupings are equivalent to shuffle groupings. Eventually though, Storm will push down bolts with none groupings to execute in the same thread as the bolt or spout they subscribe from (when possible).

Direct grouping: This is a special kind of grouping. A stream grouped this way means that the producer of the tuple decides which task of the consumer will receive this tuple.

Local or shuffle grouping: If the target bolt has one or more tasks in the same worker process, tuples will be shuffled to just those in-process tasks. Otherwise, this acts like a normal shuffle grouping.

6. Storm guarantees that every spout tuple will be fully processed by the topology. It does this by tracking the tree of tuples triggered by every spout tuple and determining when that tree of tuples has been successfully completed. Every topology has a "message timeout" associated with it. If Storm fails to detect that a spout tuple has been completed within that timeout, then it fails the tuple and replays it later.

7. Each spout or bolt executes as many tasks across the cluster. Each task corresponds to one thread of execution, and stream groupings define how to send tuples from one set of tasks to another set of tasks. It supports parallelism.

8. Topologies execute across one or more worker processes. Each worker process is a physical JVM and executes a subset of all the tasks for the topology. Storm tries to spread the tasks evenly across all the workers.

3. APACHE SPARK STREAMING OVERVIEW

3.1 What is Apache Spark Streaming

Apache Spark is a fast and general-purpose cluster computing system. Originally developed at the University of California, Berkeley's AMPLab, the Spark codebase was later donated to the Apache Software Foundation, which has maintained it since. Spark provides an interface for programming entire clusters with implicit data parallelism and fault-tolerance.

Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams. Data can be ingested from many sources like Kafka, Flume, Kinesis, or TCP sockets, and can be processed using complex algorithms expressed with high-level functions like map, reduce, join and window. Finally, processed data can be pushed out to file systems, databases, and live dashboards. In fact, you can apply Spark's machine learning and graph processing algorithms on data streams.

It, also, provides a high-level abstraction called discretized stream or DStream, which represents a continuous stream of data. DStreams can be created either from input data streams from sources such as Kafka, Flume, and Kinesis, or by applying high-level operations on other DStreams. Internally, a DStream is represented as a sequence of RDDs.

The supported languages, by Spark, are: Scala, Java or Python.

3.2 Spark Streaming Concepts

The basic concepts of Spark Streaming are the following:

1. Discretized Streams (DStreams)
2. Input DStreams and Receivers
3. Transformations on DStreams
4. Output Operations on DStreams
5. DataFrame and SQL Operations
6. MLlib Operations
7. Caching / Persistence
8. Checkpointing
9. Accumulators, Broadcast Variables, and Checkpoints

1. Discretized Stream or DStream is the basic abstraction provided by Spark Streaming. It represents a continuous stream of data, either the input data stream received from source, or the processed data stream generated by transforming the input stream. Internally, a DStream is represented by a continuous series of RDDs, which is Spark's abstraction of an immutable, distributed dataset. Each RDD in a DStream contains data from a certain interval, as shown in the following figure.

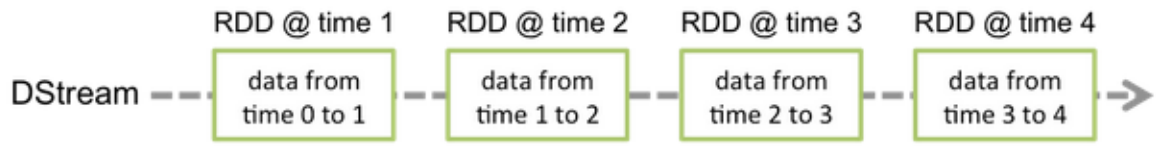

Figure 1: DStream and series of RDDs

Any operation applied on a DStream translates to operations on the underlying RDDs.

2. Input DStreams are DStreams representing the stream of input data received from streaming sources. Every input DStream is associated with a Receiver object which receives the data from a source and stores it in Spark's memory for processing. Spark Streaming provides two categories of built-in streaming sources:

- Basic sources: Sources directly available in the StreamingContext API. Examples: file systems, and socket connections.
- Advanced sources: Sources like Kafka, Flume, Kinesis, etc. are available through extra utility classes. These require linking against extra dependencies as discussed in the linking section.

Input DStreams can also be created out of custom data sources.

Regarding the Receiver Reliability, there can be two kinds of data sources based on their reliability. Sources like Kafka and Flume allow the transferred data to be acknowledged. If the system receiving data from these reliable sources acknowledges the received data correctly, it can be ensured that no data will be lost due to any kind of failure. This leads to two kinds of receivers:

- Reliable Receiver: A reliable receiver correctly sends acknowledgment to a reliable source when the data has been received and stored in Spark with replication.
- Unreliable Receiver: An unreliable receiver does not send acknowledgment to a source. This can be used for sources that do not support acknowledgment, or even for reliable sources when one does not want or need to go into the complexity of acknowledgment.

3. Transformations allow the data from the input DStream to be modified. Some of the common transformations are listed below:

Transformation	Meaning
map(*func*)	Return a new DStream by passing each element of the source DStream through a function *func*.
flatMap(*func*)	Similar to map, but each input item can be mapped to 0 or more output items.
filter(*func*)	Return a new DStream by selecting only the records of the source DStream on which *func* returns true.
repartition(*numPartitions*)	Changes the level of parallelism in this DStream by creating more or fewer partitions.
union(*otherStream*)	Return a new DStream that contains the union of the elements in the source DStream and *otherDStream*.
count()	Return a new DStream of single-element RDDs by counting the number of elements in each RDD of the source DStream.

Figure 2: List of Transformations

4. Output operations allow DStream's data to be pushed out to external systems like a database or a file systems. Since the output operations actually allow the transformed data to be consumed by external systems, they trigger the actual execution of all the DStream transformations .Currently, the following output operations are defined:

Output Operation	Meaning
print()	Prints the first ten elements of every batch of data in a DStream on the driver node running the streaming application. This is useful for development and debugging. Python API This is called **pprint()** in the Python API.
saveAsTextFiles(*prefix*, [*suffix*])	Save this DStream's contents as text files. The file name at each batch interval is generated based on *prefix* and *suffix*: *"prefix-TIME_IN_MS[.suffix]"*.
saveAsObjectFiles(*prefix*, [*suffix*])	Save this DStream's contents as `SequenceFiles` of serialized Java objects. The file name at each batch interval is generated based on *prefix* and *suffix*: *"prefix-TIME_IN_MS[.suffix]"*. Python API This is not available in the Python API.
saveAsHadoopFiles(*prefix*, [*suffix*])	Save this DStream's contents as Hadoop files. The file name at each batch interval is generated based on *prefix* and *suffix*: *"prefix-TIME_IN_MS[.suffix]"*. Python API This is not available in the Python API.
foreachRDD(*func*)	The most generic output operator that applies a function, *func*, to each RDD generated from the stream. This function should push the data in each RDD to an external system, such as saving the RDD to files, or writing it over the network to a database. Note that the function *func* is executed in the driver process running the streaming application, and will usually have RDD actions in it that will force the computation of the streaming RDDs.

Figure 3: Output Operations

5. One can easily use DataFrames and SQL operations on streaming data. Each RDD is converted to a DataFrame, registered as a temporary table and then queried using SQL. It also, allows the user to run SQL queries on tables defined on streaming data from a different thread.

6. Spark also offers the ability to use machine learning algorithms provided by MLlib.

7. Similar to RDDs, DStreams also allow developers to persist the stream's data in memory. That is, using the persist() method on a DStream will automatically persist every RDD of that DStream in memory. This is useful if the data in the DStream will be computed multiple times (e.g., multiple operations on the same data). Hence, DStreams generated by window-based operations are automatically persisted in memory, without the developer calling persist().

For input streams that receive data over the network (such as, Kafka, Flume, sockets, etc.), the default persistence level is set to replicate the data to two nodes for fault-

tolerance. Note that, unlike RDDs, the default persistence level of DStreams keeps the data serialized in memory.

8. A streaming application must operate 24/7 and hence must be resilient to failures unrelated to the application logic such as system failures, JVM crashes, etc. For this to be possible, Spark Streaming needs to checkpoint enough information to a fault-tolerant storage system such that it can recover from failures. There are two types of data that are checkpointed:

- Metadata checkpointing
- Data checkpointing

Metadata checkpointing is primarily needed for recovery from driver failures, whereas data or RDD checkpointing is necessary even for basic functioning if stateful transformations are used.

9. Accumulators and Broadcast variables cannot be recovered from checkpoint in Spark Streaming. If you enable checkpointing and use Accumulators or Broadcast variables as well, you'll have to create lazily instantiated singleton instances for Accumulators and Broadcast variables so that they can be re-instantiated after the driver restarts on failure.

4. APACHE KAFKA OVERVIEW

4.1 What is Apache Kafka

Apache Kafka, although originally developed by LinkedIn, is an open-source stream processing platform developed by the Apache Software Foundation written in Scala and Java. This streaming platform allows you to publish and subscribe to streams of records, store streams of records in a fault-tolerant way and process streams of records as they occur. This makes it highly valuable for enterprise infrastructures to process streaming data on the grounds that Apache Kafka is fast, scalable, and durable.

Here are some of the main reason enteprises and organizations use Apache Kafka:

- Stream processing: A framework such as Spark Streaming reads data from a topic, processes it and writes processed data to a new topic where it becomes available for users and applications. Kafka's strong durability is also very useful in the context of stream processing.
- Messaging: Kafka works well as a replacement for a more traditional message broker. Message brokers are used for a variety of reasons (to decouple processing from data producers, to buffer unprocessed messages, etc). In comparison to most messaging systems Kafka has better throughput.
- Website activity tracking: The web application sends events such as page views and searches Kafka, where they become available for real-time processing, dashboards and offline analytics in Hadoop
- Operational metrics: Alerting and reporting on operational metrics.
- Log aggregation: Kafka can be used across an organization to collect logs from multiple services and make them available in standard format to multiple consumers, including Hadoop.
- Event Sourcing: Event sourcing is a style of application design where state changes are logged as a time-ordered sequence of records. Kafka's support for very large stored log data makes it an excellent backend for an application built in this style.

4.2 Kafka basic concepts

Kafka runs as a cluster on one or more servers. The Kafka cluster stores streams of records in categories called topics. Each record consists of a key, a value, and a timestamp.

Kafka has four core APIs:

- The Producer API allows an application to publish a stream records to one or more Kafka topics.
- The Consumer API allows an application to subscribe to one or more topics and process the stream of records produced to them.
- The Streams API allows an application to act as a *stream processor*, consuming an input stream from one or more topics and producing an output stream to one or more output topics, effectively transforming the input streams to output streams.

- The Connector API allows building and running reusable producers or consumers that connect Kafka topics to existing applications or data systems.

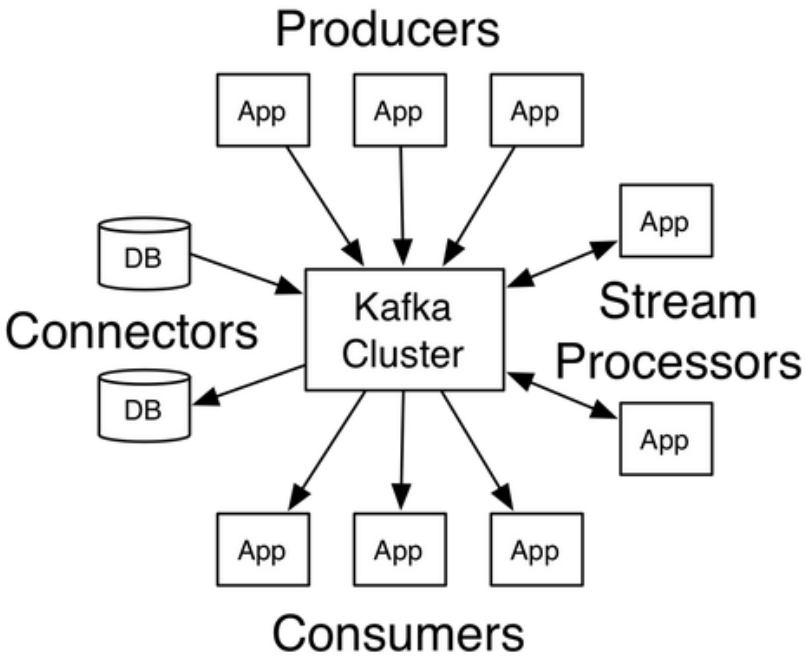

Figure 4: Kafka's APIs

In Kafka the communication between the clients and the servers is done with a simple, high-performance, language agnostic TCP protocol.

Topics and Logs

A topic is a category or feed name to which records are published. All Kafka messages are organized into topics that are always multi-subscriber, that is, a topic can have zero, one, or many consumers that subscribe to the data written to it.

For each topic, the Kafka cluster maintains a partitioned log that looks like this:

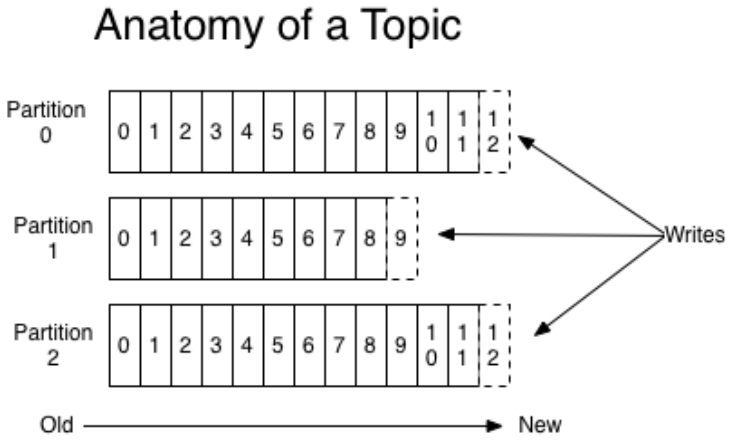

Figure 5: Anatomy of a topic

Each partition is an ordered, immutable sequence of records that is continually appended to a structured commit log. The records in the partitions are each assigned a sequential id number called the offset that uniquely identifies each record within the partition.

Distribution

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. Each partition is replicated across a configurable number of servers for fault tolerance.

Producers

Producers publish data to the topics of their choice. The producer is responsible for choosing which record to assign to which partition within the topic. This can be done in a round-robin fashion simply to balance load or it can be done according to some semantic partition function.

Consumers

Consumers read from any single partition, allowing you to scale throughput of message consumption in a similar fashion to message production. Consumer instances can be in separate processes or on separate machines. If all the consumer instances have the same consumer group, then the records will effectively be load balanced over the consumer instances. If they have different consumer groups, then each record will be broadcast to all the consumer processes.

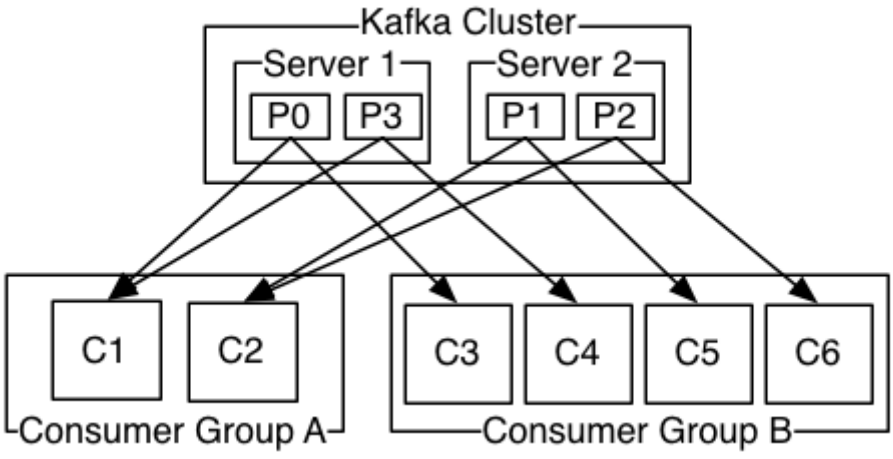

Figure 6: Consumers in Kafka

A two server Kafka cluster hosting four partitions (P0-P3) with two consumer groups. Consumer group A has two consumer instances and group B has four.

More commonly, however, we have found that topics have a small number of consumer groups, one for each "logical subscriber". Each group is composed of many consumer instances for scalability and fault tolerance.

The way consumption is implemented in Kafka is by dividing up the partitions in the log over the consumer instances so that each instance is the exclusive consumer of a "fair share" of partitions at any point in time. This process of maintaining membership in the group is handled by the Kafka protocol dynamically.

Guarantees

Kafka makes the following guarantees about data consistency and availability:

- Messages sent to a topic partition will be appended to the commit log in the order they are sent
- A single consumer instance will see messages in the order they appear in the log
- A message is ‘committed’ when all in sync replicas have applied it to their log
- Any committed message will not be lost, as long as at least one in sync replica is alive

5. APACHE FLINK OVERVIEW

5.1 What is Flink

Apache Flink is a community-driven open source framework for distributed big data analytics, like Hadoop and Spark. It originated from a research project called Stratosphere, of which the idea was conceived in 2008 by a professor named Volker Markl in Germany and was accepted from the Apache organization in 2014, as a top-level project.

This streaming dataflow engine, written in Java and Scala, provides data distribution, communication, and fault tolerance for distributed computations over data streams. It aims to bridge the gap between MapReduce-like systems and shared-nothing parallel database systems. More specifically, it provides:

- Stream execution, even if batch
- Programming in JVM but execution as Database
- Makes easier user handling
- Little required configuration and tuning
- Supports many file systems
- Many deployment options
- Integration with Hadoop
- Supports many use cases like Batch and Real - Time streaming on top of the same streaming engine
- Supports building complex data pipelines

5.2 Apache Flink Basic Concepts

Programs and Dataflows

The basic building blocks of Flink programs are streams and transformations. A stream is an intermediate result, and a transformation is an operation that takes one or more streams as input, and computes one or more result streams from them. When executed, Flink programs are mapped to streaming dataflows, consisting of streams and transformation operators. Each dataflow starts with one or more sources and ends in one or more sinks. The dataflows may resemble arbitrary directed acyclic graphs.

In most cases, there is a one-to-one correspondence between the transformations in the programs and the operators in the dataflow. Sometimes, however, one transformation may consist of multiple transformation operators.

Parallel Dataflows

Programs in Flink are inherently parallel and distributed. Streams are split into stream partitions and operators are split into operator subtasks. The operator subtasks execute independently from each other, in different threads and on different machines or containers.

The number of operator subtasks is the parallelism of that particular operator. Streams can transport data between two operators in a one-to-one pattern, or in a redistributing pattern:

- One-to-one streams
- Redistributing streams

Tasks & Operator Chains

Flink chains operator subtasks together into tasks. Each task is executed by one thread. Chaining operators together into tasks is a useful optimization: it reduces the overhead of thread-to-thread handover and buffering, and increases overall throughput while decreasing latency. The chaining behavior can be configured in the APIs.

The sample dataflow in the figure below is executed with five subtasks, and hence with five parallel threads.

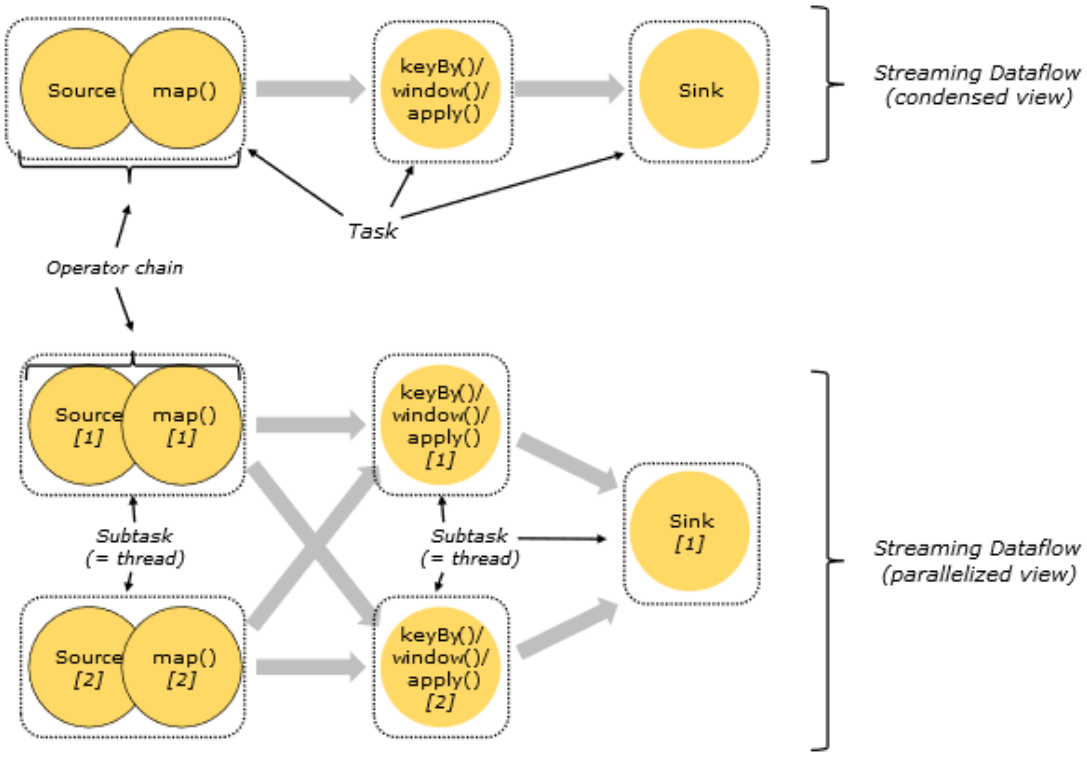

Figure 7: Example Dataflows

Distributed Execution

A Flink cluster consists of three types of processes: the client, the Job Manager, and at least one Task Manager. The master process or JobManager, coordinates the distributed execution. Jobmanagers schedule tasks, coordinate checkpoints, coordinate recovery on failures, etc. The worker processes or TaskManagers, execute the tasks of a dataflow, and buffer and exchange the data streams. There must always be at least one worker process.

The master and worker processes can be started in an arbitrary fashion: Directly on the machines, via containers, or via resource frameworks like YARN. Workers connect to masters, announcing themselves as available, and get work assigned.

Each TaskManager is a JVM process, and may execute one or more subtasks in separate threads. To control how many tasks a worker accepts, a worker has so called task slots (at least one).

Each task slot represents a fixed subset of resources of the TaskManager. Slotting the resources means that a subtask will not compete with subtasks from other jobs for managed memory, but instead has a certain amount of reserved managed memory.

By default, Flink allows subtasks to share slots, if they are subtasks of different tasks, but from the same job. The result is that one slot may hold an entire pipeline of the job. Allowing this slot sharing has two main benefits:

- A Flink cluster needs exactly as many tasks slots, as the highest parallelism used in the job
- It is easier to get better resource utilization.

The slot sharing behavior can be controlled in the APIs, to prevent sharing where it is undesirable.

Time and Windows

Aggregating events works slightly differently on streams than in batch processing. Windows can be time driven or data driven. One typically distinguishes different types of windows, such as tumbling windows (no overlap), sliding windows (with overlap), and session windows (gap of activity).

Time

When referring to time in a streaming program, one can refer to different notions of time:

- Event Time is the time when an event was created. It is usually described by a timestamp in the events, for example attached by the producing sensor, or the producing service
- Ingestion time is the time when an event enters the Flink dataflow at the source operator
- Processing Time is the local time at each operator that performs a time-based operation

State and Fault Tolerance

While many operations in a dataflow simply look at one individual event at a time some operations remember information across individual events. These operations are called stateful. The state of stateful operations is maintained in what can be thought of as an embedded key/value store. The state is partitioned and distributed strictly together with the streams that are read by the stateful operators.

Fault Tolerance Checkpoints

Flink implements fault tolerance using a combination of stream replay and checkpoints. A checkpoint defines a consistent point in streams and state from which a streaming dataflow can resume, and maintain. The events and state updates since the last checkpoint are replayed from the input streams.

Batch on Streaming

Flink executes batch programs as a special case of streaming programs, where the streams are bounded. A DataSet is treated internally as a stream of data. The concepts above thus apply to batch programs in the same way as well as they apply to streaming programs, with minor exceptions:

- Programs in the DataSet API do not use checkpoints. Recovery happens by fully replaying the streams. That is possible, because inputs are bounded. This pushes the cost more towards the recovery, but makes the regular processing cheaper, because it avoids checkpoints
- Stateful operation in the DataSet API use simplified in-memory/out-of-core data structures, rather than key/value indexes
- The DataSet API introduces special synchronized iterations, which are only possible on bounded streams.

6. APACHE SAMZA OVERVIEW

6.1 What is Samza

Apache Samza is an open-source near-realtime, asynchronous computational framework for stream processing developed by the Apache Software Foundation in Scala and Java. It uses Apache Kafka for messaging, and Apache Hadoop YARN to provide fault tolerance, processor isolation, security, and resource management.

Samza is a stream processing framework with the following features:

- **Simple API:** Unlike most low-level messaging system APIs, Samza provides a very simple callback-based "process message" API comparable to MapReduce.
- **Managed state:** Samza manages snapshotting and restoration of a stream processor's state. When the processor is restarted, Samza restores its state to a consistent snapshot. Samza is built to handle large amounts of state (many gigabytes per partition).
- **Fault tolerance:** Whenever a machine in the cluster fails, Samza works with YARN to transparently migrate your tasks to another machine.
- **Durability:** Samza uses Kafka to guarantee that messages are processed in the order they were written to a partition, and that no messages are ever lost.
- **Scalability:** Samza is partitioned and distributed at every level. Kafka provides ordered, partitioned, fault-tolerant streams. YARN provides a distributed environment for Samza containers to run in.
- **Pluggable:** Though Samza works out of the box with Kafka and YARN, Samza provides a pluggable API that lets you run Samza with other messaging systems and execution environments.
- **Processor isolation:** Samza works with Apache YARN, which supports Hadoop's security model, and resource isolation through Linux CGroups.

6.2 Apache Samza Basic Concepts

Streams

Samza processes streams. A stream is composed of immutable messages of a similar type or category. For example, a stream could be all the clicks on a website or any other type of event data. Messages can be appended to a stream or read from a stream. A stream can have any number of consumers, and reading from a stream doesn't delete the message.

Jobs

A job in Samza performs a logical transformation on a set of input streams to append output messages to set of output streams. In order to scale the throughput of the stream processor, Samza splits streams and jobs up into smaller units of parallelism: partitions and tasks.

Partitions

Each stream is divided into one or more partitions. Each partition in the stream is a totally ordered sequence of messages. Each message in this sequence has an identifier called the offset, which is unique per partition. The offset can be a sequential integer, byte offset, or string depending on the system implementation.

When a message is appended to a stream, it is appended to only one of the stream's partitions. The assignment of the message to its partition is done with a key chosen by the writer.

Tasks

A job is scaled by breaking it into multiple tasks. The task is the unit of parallelism of the job, just as the partition is to the stream. Each task consumes data from one partition for each of the job's input streams. A task processes messages from each of its input partitions sequentially, in the order of message offset. There is no defined ordering across partitions. This allows each task to operate independently.

The number of tasks in a job is determined by the number of input partitions. However, one can change the computational resources assigned to the job satisfy the job's needs. The assignment of partitions to tasks never changes: if a task is on a machine that fails, the task is restarted elsewhere, still consuming the same stream partitions.

Dataflow Graphs

One can compose multiple jobs to create a dataflow graph, where the edges are streams containing data, and the nodes are jobs performing transformations. This composition is done purely through the streams the jobs take as input and output. The jobs are otherwise totally decoupled: they need not be implemented in the same code base, and adding, removing, or restarting a downstream job will not impact an upstream job.

These graphs are often acyclic, however, it is possible to create cyclic graphs if you need to.

Containers

Partitions and tasks are both logical units of parallelism but they don't correspond to any particular assignment of computational resources. Containers are the unit of physical parallelism, and a container is essentially a Unix process. Each container runs one or more tasks. The number of tasks is determined automatically from the number of partitions in the input and is fixed, but the number of containers is specified by the user at run time and can be changed at any time.

7. SIMILARITIES AND DIFFERENCES BETWEEN THE APACHE STREAMING ENGINES

After analyzing each one of the Apache Streaming Tools above, it is now time to compare their functionalities and characteristics and provide a walk-through all the similarities and differences between them.

When choosing between different systems there are a couple of points that should be taken into consideration:

Table 1: Table with results of the comparison

	Storm	**Spark Streaming**	**Flink**	**Kafka**	**Samza**
Category	ESP/CEP	ESP	ESP/CEP	ESP	ESP
Streaming API	Spout, Tuple	HDFS, DStream	DataStream	KafkaStream	Message
Cluster Management	Mesos, Yarn	Mesos, Yarn, Standalone	Yarn, Tez, Standalone	Any	Yarn
Process Model	Event	Micro-batch Batch	Event Micro-Batch Batch	Event	Event
Latency	Very low	Medium	Low	Low	Low
Throughput	Low	High	High	Medium	High
Fault-Tolerance	Yes	Yes	Yes	Yes	Yes
Language Support	Any JVM	Scala/Java/Python	Scala/Java/Python	Java	Scala/Java
Delivery Guarantee	At Least Once	Exactly Once	Exactly Once	At Least Once	At Least Once

Data Flow	Topology	Application	Streaming Dataflow	Process Topology	Job
Windowing	Time-based Count-based	Time-based	Time-based Count-based	Time-based	Time-based

In fact, there are even more points of interest, that make each tool special regarding the needs of the enterprise or the individuals who want to process and transform streaming data.

The most simple and characteristic example that we can provide in order to distinct the different APIs is the WordCount example, which is the similar to as the HelloWorld example, but it is used in Streaming Tools programming.

7.1 Word Count example in Storm

In the following example, we will define a Topology that reads sentences of a spout and streams out of WordCountBolt the total number of times it has seen that word before.

So, at first, we have to define a topology. Then a spout and a bolt must be defined to split the words. Numbers 5, 8 and 12 are the parallelism hints and they define how many independent threads around the cluster will be used for execution of every component.

```
TopologyBuilder builder = new TopologyBuilder();

builder.setSpout("sentenceSpout", new RandomSentenceSpout(), 5);
builder.setBolt("split", new Split(), 8).shuffleGrouping("sentenceSpout");
builder.setBolt("count", new WordCount(), 12).fieldsGrouping("split", new Fields("word"));
```

Figure 8: Word Count in Storm

SplitSentence emits a tuple for each word in each sentence it receives, and WordCount keeps a map in memory from word to count. Each time WordCount receives a word, it updates its state and emits the new word count. In this example, we used shuffle grouping which sends the tuple to a random task. There's a few different kinds of stream groupings. In addition, bolts can be defined in any language.

7.2 Word Count example in Spark

Word Count by Apache Spark can be implemented in Scala. A simple implementation of the Word Count example follows below:

```
val conf = new SparkConf().setAppName("wordCount")
val sc = new SparkContext(conf)
val input =  sc.textFile(inputFile)
val words = input.flatMap(line => line.split(" "))
val counts = words.map(word => (word, 1)).reduceByKey{case (x, y) => x + y}
counts.saveAsTextFile(outputFile)
```

Figure 9: Word Count in Spark

Firstly, we have to create a Scala Spark context. Then, load the input file (inputFile) in order to split up the words in it. Then, we add how many time each word appears on the file and save the results in an output file.

7.3 Word Count example in Flink

The Word Count example implementation in Flink, is supported in Scala as well.

```
val env = ExecutionEnvironment.getExecutionEnvironment
val text = env.fromElements("Who's there?",
      "I think I hear them. Stand, ho! Who's there?")

val counts = text.flatMap { _.toLowerCase.split("\\W+") filter { _.nonEmpty } }
                 .map { (_, 1) }
                 .groupBy(0)
                 .sum(1)

counts.print()
env.execute("Scala WordCount Example")
```

Figure 10: Word Count in Flink

The fromElements method of the ExecutionEnvironment and Stream ExecutionEnvironment determines the DataSet/DataStream type by extracting the type of the first input element.

Using flatMap(), map(), groupBy() and sum() methods, it counts how sums how many times a word appears in the sentence and groups by the words.

7.4 Word Count example in Kafka

In the Word Count example implemented in Kafka, we first create a KStreamBuilder instance and with builder.stream() allow it to generate elements individually and adding them to the Builder. Then, it splits each text line, by whitespace, into words. After that, it invokes ‘flatMapValues’ instead of the more generic ‘flatMap’ and use ‘map’ to ensure the key of each record contains the respective word. Finally it counts the occurrences of each word. We must provide a name for the resulting KTable, which will be used to name e.g. its associated state store and change log topic.

```
KStreamBuilder builder = new KStreamBuilder();

KStream<String, String> textLines = builder.stream(stringSerde, stringSerde,"TextLinesTopic");
Pattern pattern = Pattern.compile("\\W+", Pattern.UNICODE_CHARACTER_CLASS);

KStream<String, Long> wordCounts = textLines
                                    .flatMapValues(value-> Arrays.asList(pattern.split(value.toLowerCase())))
                                    .map((key, word) -> new KeyValue<>(word, word))
                                    .countByKey("Counts")
                                    .toStream();

wordCounts.to(stringSerde, longSerde, "WordsWithCountsTopic");

KafkaStreams streams = new KafkaStreams(builder, streamsConfiguration);
streams.start();
```

Figure 11: Word Count in Kafka

7.5 Word count example in Samza

Below it's a sample of how the Word Count example is implemented with Scala in Samza. In this case, the whole topology is the WordCountTask, which does all the work. In Samza the components are defined by implementing particular interfaces, in this case it's a StreamTask. Its parameter list contains all what's need for connecting with the rest of the system.

```
class WordCountTask extends StreamTask {

  override def process(envelope: IncomingMessageEnvelope, collector: MessageCollector,
    coordinator: TaskCoordinator) {

    val text = envelope.getMessage.asInstanceOf[String]

    val counts = text.split(" ").foldLeft(Map.empty[String, Int]) {
      (count, word) => count + (word -> (count.getOrElse(word, 0) + 1))
    }

    collector.send(new OutgoingMessageEnvelope(new SystemStream("kafka", "wordcount"), counts))

 }
```

Figure 12: Word Count in Samza

8. STREAMING TOOLS USES CASES AND INTEGRATION IN THE BUSINESS FIELD

Big data technologies are widely adopted in small and large companies, since the need of real-time processing and analysis is critic. In finance industry, banks are using the Hadoop alternative - Spark to access and analyze call recordings, complaint logs, emails, forum discussions, etc. to gain insights which can help them make right business decisions for credit risk assessment, targeted advertising and customer segmentation. Financial institutions are, also, leveraging big data to detect fraudulent transactions in real-time. Apache Spark plays a significant role in e-commerce, as well.

Well-known e-commerce platforms, Ali Baba, the world's largest retailer, and eBay, prefer Spark to analyze hundreds of petabytes of data or to provide targeted offers and enhance customer experience, which are proven examples of the Spark efficiency and performance. Of course, Spark is used in Social Media applications, such as Pinterest or other common applications like TripAdvisor, which provides advice to millions of travellers by comparing hundreds of websites to find the best hotel prices for its customers.

Apache Storm, on the other hand, is used in various fields, from healthcare to social media. More specifically, it is used by Twitter to power a wide variety of Twitter systems, ranging in applications from discovery, real-time analytics, personalization, search, revenue optimization etc. Yahoo chooses Storm for stream/micro-batch processing of user events, content feeds, and application logs and Spotify powers a wide range of real-time features at Spotify, including music recommendation, monitoring, analytics, and ads targeting. Together with Kafka and Cassandra, based messaging, Storm is used to build low-latency fault-tolerant distributed systems with ease.

Integration with Kafka is a common thing in the big data area, due to its performance characteristics and its scalability. Apache Kafka is, also, used at LinkedIn for activity stream data and operational metrics. This powers various products like LinkedIn Newsfeed, LinkedIn Today in addition to our offline analytics systems like Hadoop. In addition, as the demand for real-time (sub-minute) analytics grew, Netflix moved to using Kafka as its primary backbone for ingestion via Java APIs or REST APIs. Netflix's system now supports ingestion of ~500 billion events per day and at peak up to ~8 million events per second. It has paired Kafka with streaming stacks like Apache Spark and Apache Samza to route data and load it into back-end data stores like ElasticSearch and Cassandra, as well as directly into real-time analytics engines.

Apart from LinkedIn, where Apache Samza it is currently used to process tracking data, service log data, and for data ingestion pipelines for real-time services, and Netflix, other companies use Samza is used in integration with Spark in TripAdvisor, in order to process billions of events daily for analytics, machine learning, and site improvement. Uber, in addition, uses Samza to provide stream processing as a service. Currently, this platform supports two categories of use cases: metrics aggregation and near real time state machine for doing database updates. Nevertheless, Samza could be related to fraud detection and root cause analysis in the near future.

Finally, Alibaba’s search infrastructure team uses, also, Apache Flink to update product detail and inventory information in real-time, improving relevance for users. Ericsson, on the other hand, used Apache Flink to build a real-time anomaly detector with machine learning over large infrastructures. Apache Flink, thrives also in ETL for business intelligence infrastructure, as Zalando uses it to transform data for easier loading into its data warehouse, converting complex payloads into relatively simple ones and ensuring that analytics end users have faster access to data.

9. ΣΥΜΠΕΡΑΣΜΑΤΑ

In the era of exponential technology, as the speed of business accelerates, organizations are producing increasingly vast volumes of high velocity data in very different formats. Hence, the more streaming frameworks such as Apache Spark, Apache Storm, Samza, Flink etc. will be leveraged in the business field, in order to process and handle real-time data streams and information, allowing the ability to analyze risks before they occur.

From Finance to Healthcare, from Social Media to the Internet of Things, the need to deal with the disproportionate size of data sets, is transparent. Traditional big data-styled frameworks like Hadoop are not appropriate for these use cases.

In conclusion, enterprises should utilize state of the art technical equipment for streaming data analysis in order to survive the technological competition, though supporting much faster decision making. The benefits of streaming integration and intelligence are plenty and the right choice of streaming tools, can drastically change a company’s market position.

REFERENCES

[1] Apache Storm http://storm.apache.org [Προσπελάστηκε 02/12/16]
[2] "Apache Storm Graduates to a Top-Level Project". [Προσπελάστηκε 02/12/16]
[3] Spark Overview http://spark.apache.org/docs/2.0.2/index.html [Προσπελάστηκε 04/12/16]

[4] Spark Streaming http://en.wikipedia.org/wiki/Apache_Spark#Spark_Streaming [Προσπελάστηκε 04/12/16]
[5] http://spark.apache.org/docs/latest/streaming-programming-guide.html#deploying-applications [Προσπελάστηκε 4/12/16]
[6] Apache Kafka A distributed streaming platform http://kafka.apache.org/ [Προσπελάστηκε 01/04/17]

[7] Apache Kafka for Beginners http://blog.cloudera.com/blog/2014/09/apache-kafka-for-beginners/ [Προσπελάστηκε 06/01/17]

[8] Apache Flink https://flink.apache.org/ [Προσπελάστηκε 06/01/17]

[9] https://www.slideshare.net/sbaltagi/apacheflinkwhathowwhywhowherebyslimbaltagi-57825047 [Προσπελάστηκε 14/01/16]
[10] Apache Flink: What, How, Why, Who, Where? https://www.youtube.com/watch?v=G77m6Ou_kFA [Προσπελάστηκε 30/01/16]
[11] Samza http://samza.apache.org/ [Προσπελάστηκε 06/01/17]
[12] All the Apache Streaming Projects: An Exploratory Guide https://thenewstack.io/apache-streaming-projects-exploratory-guide/ [Προσπελάστηκε 06/01/17]

[13] V2 of Apache streaming technologies https://twitter.com/ianhellstrom/status/710917506412716033 [Προσπελάστηκε 06/02/17]
[14] http://spark.apache.org/powered-by.html [Προσπελάστηκε 07/04/17]
[15] http://storm.apache.org/releases/current/Powered-By.html [Προσπελάστηκε 07/04/17]
[16] Apache Storm Use Cases - Edureka Blog https://www.edureka.co/blog/apache-storm-use-cases/ [Προσπελάστηκε 07/04/17]
[17] Apach Samza Powered By https://cwiki.apache.org/confluence/display/SAMZA/Powered+By [Προσπελάστηκε 07/04/17]

[18] Flink Use Cases https://flink.apache.org/usecases.html [Προσπελάστηκε 07/04/17]

[19] Powered by https://kafka.apache.org/powered-by [Προσπελάστηκε 07/04/17]