BigStorage

STORAGE-BASED CONVERGENCE BETWEEN HPC AND CLOUD TO HANDLE BIG DATA

Deliverable number	D2.1
Deliverable title	Intermediate Report Big Data processing: state of the art WP2 DATA SCIENCE
Editor	Gabriel Antoniu (Inria)
Main Authors	Alvaro Brandon (UPM), Ovidiu Marcu (Inria), Pierre Matri (UPM)

Grant Agreement number	642963
Project ref. no	MSCA-ITN-2014-ETN-642963
Project acronym	BigStorage
Project full name	BigStorage: Storage-based convergence between HPC and Cloud to handle Big Data
Starting date (dur.)	1/1/2015 (48 months)
Ending date	31/12/2018
Project website	http://www.bigstorage-project.eu

Coordinator	María S. Pérez
Address	Campus de Montegancedo sn. 28660 Boadilla del Monte, Madrid, Spain
Reply to	mperez@fi.upm.es
Phone	+34-91-336-7380

Executive Summary

This document provides an overview of the progress of the work done until M24 of the Project BigStorage (from 01-01-2015 until 31-12-2016) in WP2 Data Science.

This report presents an overview of the major systems that have been studied by the Early Stage Researchers (ESRs). This covers the following topics:

- MapReduce, the de facto state-of-the-art programming model for Big Data processing and its reference implementations (Google MapReduce and Hadoop)
- Performance-optimized MapReduce processing based on in-memory storage (Spark, Flink)
- More generic models for BigData processing, supporting SQL-like queries, data streams, more generic workflows, general graph processing
- Systems leveraging machine-learning for Big Data processing.

This report presents an overview of each of these topics, including a brief analysis of the state-of-the-art in each case and a preliminary presentation of the specific problems to be addressed in the project.

Document Information

IST Project Number	MSCA-ITN-2014-ETN-642963
Acronym	BigStorage
Title	Storage-based convergence between HPC and Cloud to handle Big Data
Project URL	http://www.bigstorage-project.eu
Document URL	http://bigstorage-project.eu/index.php/deliverables
EU Project Officer	Mr. Szymon Sroda
Deliverable	D2.1 Intermediate Report on WP2
Workpackage	WP2 Data Science
Date of Delivery	Planned: 31.12.2016 Actual: 15.12.2016
Status	Version 1.0 final ■ draft □
Nature	prototype □ report ■ dissemination □
Dissemination level	public □ consortium ■
Distribution List	Consortium Partners
Document Location	http://bigstorage-project.eu/index.php/deliverables
Responsible Editor	Gabriel Antoniu (Inria), gabriel.antoniu@inria.fr, Tel: +33299847244
Authors (Partner)	Alvaro Brandon (UPM), Ovidiu Marcu (Inria), Pierre Matri (Inria)
Reviewers	All advisors
Abstract (for dissemination)	This report presents an overview of the major systems that have been studied by the Early State Researchers (ESRs). This covers the following topics: • MapReduce, the de facto state-of-the-art programming model for Big Data processing and its reference implementations (Google MapReduce and Hadoop) • Performance-optimized MapReduce processing based on in-memory storage (Spark, Flink) • More generic models for BigData processing, supporting SQL-like queries, data streams, more generic workflows, general graph processing • Systems leveraging machine-learning for Big Data processing. This report presents an overview of each of these topics, including a brief analysis of the state-of-the-art in each case and a preliminary presentation of the specific problems to be addressed in the project.
Keywords	Data processing, data science, Big Data processing, MapReduce, streaming, workflow processing, Spark, Flink

Version	Modification(s)	Date	Author(s)
0.1	Initial template and structure	22.09.2016	Gabriel Antoniu, Inria
0.2	Sections from all authors	14.10.2016	All authors
0.3	Internal version for review	7.12.2016	Gabriel Antoniu, Inria

0.4	Comments from internal reviewers	12.12.2016	Internal reviewers
0.5	Final version to commission	15.12.2016	Gabriel Antoniu, Inria

Project Consortium Information

Participants		Contact
Universidad Politécnica de Madrid (UPM), Spain		María S. Pérez Email: mperez@fi.upm.es
Barcelona Supercomputing Center (BSC), Spain		Toni Cortes Email: toni.cortes@bsc.es
Johannes Gutenberg University (JGU) Mainz, Germany		André Brinkmann Email: brinkman@uni-mainz.de
Inria, France		Gabriel Antoniu Email: gabriel.antoniu@inria.fr Adrian Lebre Email: adrien.lebre@inria.fr
Foundation for Research and Technology - Hellas (FORTH), Greece		Angelos Bilas Email: bilas@ics.forth.gr
Seagate, UK		Malcolm Muggeridge Email: malcolm.muggeridge@seagate.com
DKRZ, Germany		Thomas Ludwig Email: ludwig@dkrz.de
CA Technologies Development Spain (CA), Spain		Victor Muntes Email: Victor.Muntes@ca.com
CEA, France		Jacque Charles Lafoucriere Email: Charles.LAFOUCRIERE@CEA.FR
Fujitsu Technology Solutions GMBH, Germany		Sepp Stieger Email: sepp.stieger@ts.fujitsu.com

Table of Contents

1 Introduction

1.1 *WP2 Overview*

This subsection provides an overview of WP2 as was presented in the project proposal.

The Data Science has emerged as the fourth paradigm for scientific discovery, based on data-intensive computing [Tan09]. Motivated by the emergence of Big Data applications, Data Science involves several data-centric aspects: storage, manipulation, analysis with statistics and machine learning, decision-making, among others. During the recent years, the MapReduce [Dea04] programming model (e.g., Amazon Elastic MapReduce, Hadoop on Azure - HDInsight) has emerged as the de facto standard for Big Data processing. However, many Data Science applications do not fit this model and require a more general data orchestration, independent of any programming model. Modeling them as workflows is an option [3,4], but current cloud infrastructures lack specific support for efficient workflow data handling. State-of-the-art solutions rely on high-latency storage services (Azure Blobs, Amazon S3) or implement application-specific overlays that pipeline data from one task to another. However, in large-scale distributed environments consisting of multiple datacenters, this would suffer from latencies when accessing large remote datasets frequently.

Data Science involves unprecedented complexity in the Big Data management process, which is not addressed by existing cloud data handling services. This WP aims to investigate new models, mechanisms and policies that are needed to support dynamic coordination for data sharing, dissemination, analysis and exploitation across widely distributed sites with reasonable QoS levels. Another major aspect regards energy, where some research work has explored power savings in MapReduce clusters through virtual machine placement [5], data compression [6], and static data layout techniques [7]. However, such efforts were limited to a single MapReduce application: substantial effort is needed to cope with shared platforms, with multiple, diverse Data Science applications run concurrently. We plan to investigate such challenges in this WP.

WP2 is organized in 3 tasks, as follows.

Task 2.1. Big Data Processing Models
The objective of this task is to design next-generation data processing models for applications that require general data orchestration, independent of any programming model. Based on the use cases studied in WP1, we will focus on workflows composed of many tasks, linked via data- and control-flow dependencies as well as on stream data processing. We will examine the limitations and bottlenecks of current NoSQL/MapReduce based solutions; define a set of minimal requirements for efficient workflow and real-time data management; design techniques for fast transfers between workflow nodes (inter and intra datacenter) and allow efficient stream processing.

Task 2.2 Green Big Data Analysis

This task will focus on investigating new techniques for energy-efficient Big Data analysis, reducing the data set input by the application of machine learning techniques on shared clusters and by designing new energy-aware job schedulers for MapReduce-like data analysis. More specifically, we will define: a data model for energy consumption for MapReduce-like data analysis on clouds; mechanisms for scheduling data analysis tasks across clouds to meet response-time requirements while reducing the overall energy-consumption; new data layout schemes that guarantee data availability and reduce response time while reducing energy consumption.

Task 2.3. Data-Driven Decision Making

Predictive insights and accurate predictive models from data are essential in nowadays business applications. This task aims to train researchers in the expertise of data-driven decision, connecting the low-level (scalable data infrastructures) to the real needs of applications We will: study of the current data-driven decision approaches, identifying their limitations in scalability and performance; define a set of minimal requirements for efficient and scalable data-driven decision approaches; design novel techniques for efficient and scalable data-driven decision making processes.

1.2 *ESR Participation and contributions*

The full list of ESRs in the current form of the project is:

ESR	Name	Institution	Main advisor
1	Ovidiu-Cristian Marcu	INRIA	Gabriel Antoniu/Alexandru Costan
2	*Alvaro Brandon*	*UPM*	*Maria S. Perez*
3	Pierre Matri	UPM	Maria S. Perez
4	Muhammad Umar Hameed	Mainz	Andre Brinkmann
5	*Rizkallah Touma*	*BSC*	*Anna Queralt/Toni Cortes*
6	Fotios Papaodyssefs	Seagate	Malcolm Muggeridge
7	Linh Thuy Nguyen	INRIA	Adrien Lebre
8	*Athanasios Kiatipis*	*Fujitsu*	*Sepp Stieger*
9	*Fotis Nikolaidis*	*CEA*	*Philippe Deniel*
10	*Dimitrios Ganosis*	*FORTH*	*Angelos Bilas/Manolis Marazakis*
11	*Nafiseh Moti*	*Mainz*	*Andre Brinkmann*
12	*Georgios Koloventzos*	*BSC*	*Ramon Nou/Toni Cortes*
13	Mohammed-Yacine Taleb	INRIA	Shadi Ibrahim
14	Yevhen Alforov	DKRZ	Thomas Ludwig / Michael Kuhn
15	*Michał Zasadziński*	*CA*	*Victor Muntes*

The ESRs participating in WP2 are ESR 1 (lead for Task 2.1), ESR2 (lead for Task 2.2), ESR3 (lead for Task 2.3). They are all also involved in other WPs.

Note: This report aims to present the state-of-the-art in Big Data processing by describing existing systems with their particular features. As WP5 is fully dedicated to energy-related aspects, to avoid redundancy, we decided not to detail the energy dimension in this report.

2 MapReduce: the de facto state-of-the-art programming model for Big Data analytics

2.1 Google MapReduce

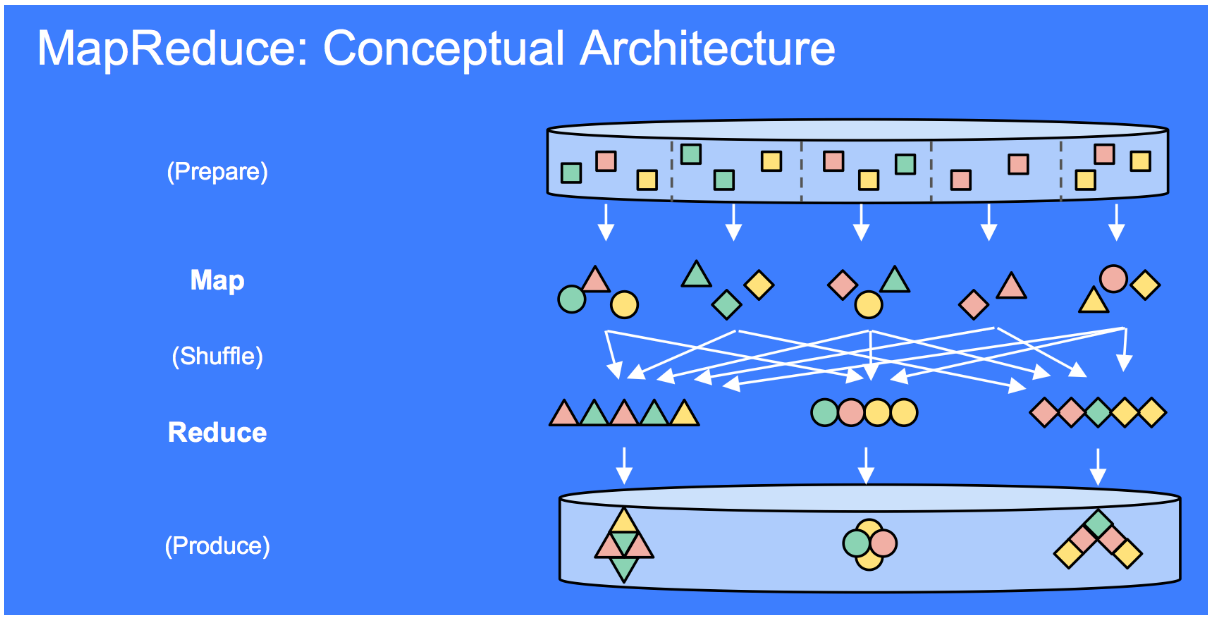

Figure 1. MapReduce Conceptual Architecture (extracted from [Dea04]).

A first milestone of the evolution of Data processing is the introduction of the MapReduce programming model by Google in 2004 [Dea04]. Among the motivations that triggered the proposal of the MapReduce model there were two important challenges at that time: 1) provide effective scalability (hard to get implemented efficiently in practice with traditional database-oriented techniques); 2) ensure that a fault-tolerance implementation does not generate a high complexity for system developers or the final users, and it is implemented correctly.

In its essence, a MapReduce job consists of a sequence of two processing stages (as illustrated in Figure 1): a Map stage, consisting in filtering the input data to produce intermediate data that is relevant for a particular data analysis request; a Reduce stage, consisting in aggregating the intermediate data to produce the final result. Each stage is (potentially) highly parallel: the initial input dataset is split into an arbitrary large number of subsets, each of which is typically processed by a separate Map task; the intermediate data produced by the Map tasks is shuffled and sorted, then processed by the Reduce tasks, with full parallelism again. Typically both the input and the output data of the job are stored in a file-system. The framework takes care of scheduling tasks, monitoring them and re-executing the failed tasks.

Google MapReduce is the first reference implementation of the MapReduce model. Over the years it has been updated with multiple features and optimizations [Aki16a], proving its efficiency for example by sorting 10 PB of data in about 6.5 hours. The system has been used at Google until 2015 [Aki16a].

2.2 Hadoop: the "standard" open-source implementation of MapReduce

Soon after the MapReduce model was published, the Apache Hadoop [Hada] Open Source ecosystem took birth and became the de facto standard for MapReduce processing, through its adoption by the main cloud computing providers. Hadoop currently consists of three main projects:

- Hadoop Distributed File System (HDFS) to provide high-throughput data access to the application;
- Hadoop YARN framework for resource and cluster management;
- Hadoop MapReduce which implements the MapReduce processing model.

The MapReduce model and its open-source implementation Hadoop MapReduce were quickly and widely adopted by both industry and academia, mostly because of their simplified yet powerful programming model. In order to understand how MapReduce works, its main components and architecture, we can follow the simple tutorial from [Hadb], briefly summarized below.

In a typical implementation of MapReduce such as Hadoop, the compute nodes and the storage nodes are the same, that is, the MapReduce framework and the Hadoop Distributed File System (see HDFS Architecture Guide) run on the same set of nodes. This configuration allows the framework to effectively schedule tasks on the nodes where data is already present, resulting in very high aggregate bandwidth across the cluster.

In a minimal utilization mode, applications specify the input/output locations and supply *map* and *reduce* functions via implementations of appropriate interfaces and/or abstract-classes. These, and other job parameters, comprise the *job configuration*.

The Hadoop *job client* then submits the job (jar/executable etc.) and configuration to a *ResourceManager* which then assumes the responsibility of distributing the software/configuration to the slaves, scheduling tasks and monitoring them, providing status and diagnostic information to the job-client.

Maps are individual tasks that transform input records into intermediate records. The transformed intermediate records do not need to be of the same type as the input records. A given input pair may map to zero or many output pairs. All intermediate values are passed to the Reducer(s) to determine the final output. Users can control the grouping by specifying a *Comparator*. The number of maps is usually driven by the total size of the inputs, that is, the total number of blocks of the input files.

A Reducer has 3 primary phases:

- *Shuffle*: in this phase the framework fetches the relevant partition of the output of all the mappers to place it on the local storage of the reducer tasks which will process the partition;
- *Sort*: the framework groups Reducer inputs by keys (since different mappers may have output the same key) in this stage;
- *Reduce*: the actual reduce processing takes place.

The shuffle and sort phases occur simultaneously; when map-outputs are fetched they are merged.

3 Going faster: in-memory MapReduce processing

The very simple API of MapReduce comes with the important caveat that users are forced to express applications in terms of map and reduce functions. However, more applications were developed with the requirement that should be independent of any programming model. For instance, iterative algorithms used in graph analytics and machine learning are not well served by the original MapReduce model. These limitations are addressed by a second generation of analytics platforms (e.g. Apache Spark [Spark], Apache Flink [Flink]) that emerged in an attempt to unify the landscape of Big Data processing.

Spark and Flink facilitate the development of multi-step data pipelines using directly acyclic graph (DAG) patterns. At a higher level, both engines implement a driver program that describes the high-level control flow of the application, which relies on two main parallel programming abstractions:

(1) *structures* to describe the data and
(2) parallel *operations* on these data.

While the data representations differ, both Flink and Spark implement similar dataflow operators (e.g. map, reduce, filter, distinct, collect, count, save etc.), or an API can be used to obtain the same result. For instance, Spark's *reduceByKey* operator (called on a dataset of key-value pairs to return a new dataset of key-value pairs where the value of each key is aggregated using the given reduce function) is equivalent to Flink's *groupBy* followed by the aggregate operator *sum* or *reduce*.

3.1 Spark

Apache Spark [Spark] introduced Resilient Distributed Datasets (RDDs) [Zah12a], a set of in-memory data structures able to cache intermediate data across a set of nodes, in order to efficiently support iterative algorithms. RDDs (read-only, resilient collections of objects partitioned across multiple nodes) hold provenance information (referred to as lineage). They can be rebuilt in case of failures by partial re-computation from ancestor RDDs.

Each RDD is by default lazy (i.e. computed only when needed) and ephemeral (i.e. once it actually gets materialized, it will be discarded from memory after its use). However, since RDDs might be repeatedly needed during computations, the user can explicitly mark them as persistent, which moves them in a dedicated cache for persistent objects.

The operations available on RDDs seem to emulate the expressivity of the MapReduce paradigm overall, however, there are two important differences:

1. Due to their lazy nature, maps will only be processed before a reduce, accumulating all computational steps in a single phase;
2. RDDs can be cached for later use, which greatly reduces the need to interact with the underlying distributed file system in more complex workflows that involve multiple reduce operations.

Tachyon

At the core of Spark stays Tachyon [Hao14], later renamed Alluxio, an open source in-memory distributed storage that enables reliable data sharing at memory speed across cluster frameworks like Spark or Flink. Tachyon eliminates the bottleneck of using replication for fault-tolerance, by leveraging lineage, the well-known technique used in RDDs. An experimental study [Hao14] shows that Tachyon achieves 110x higher write throughput than in-memory HDFS.

DataFrame

Recently, the new DataFrame API was developed for Spark [Data]. It is an extension of the RDD API, in which data is now organized into named columns. It is very similar to a relational database or to the widely used model of dataframes in R/Python. Its benefits come from knowing the structure of the data. This allows Spark to do optimizations under the hood. Spark's Catalys optimizer compiles operations into JVM bytecode with plans that involve intelligent actions like broadcasts or skipping reading irrelevant data for a particular operation.

Recently, the DataFrames API was merged with the Dataset API. This new unified API is supposed to be the single high level API for Spark [Datb] consisting of:

- An untyped API where a DataFrame is considered as a Dataset of a generic untyped object "Row" (in other words a Dataset[Row]);
- A strongly-typed API where a Dataset is a collection of strongly typed JVM objects (in other words Dataset[T]).

The benefit of a Dataset over a Dataframe is that it is statically typed and is safe at runtime since all the errors are checked at compile time. It also gives users a high level abstraction and a view of the structured data at hand with a rich API. Lastly, because Dataset is strongly typed, objects can be mapped to Tungsten's internal memory representation for efficient serialize/deserialize operations and compact bytecode, meaning superior performance.

3.2 Flink

Flink is built on top of DataSets (collections of elements of a specific type on which operations with an implicit type parameter are defined), Job *Graphs* and Parallelisation Contracts (*PACTs*) [War09]. *Job Graphs* represent parallel data flows with arbitrary tasks, that consume and produce data streams. *PACTs* are second-order functions that define properties on the input/output data of their associated user defined (first order) functions (UDFs); these properties are further used to parallelize the execution of UDFs and to apply optimization rules [Ale11].

In contrast to Flink, Spark users can control two very important aspects of the RDDs: the persistence (i.e. in memory or disk based) and the partition scheme across the nodes [Zah12a]. This fine-grained control over the storage approach of intermediate data proves to be very useful for applications with varying I/O requirements.

Iteration handling

Another important difference relates to iterations handling. Spark implements iterations as regular for-loops and executes them by loop unrolling. This means that for each iteration a new set of tasks/operators is scheduled and executed. Each iteration operates on the result of the previous iteration which is held in memory.

Flink executes iterations as cyclic data flows. This means that a data flow program (and all its operators) is scheduled just once and the data is fed back from the tail of an iteration to its head. Basically, data is flowing in cycles around the operators within an iteration. Since operators are just scheduled once, they can maintain a state over all iterations.

Flink's API offers two dedicated iteration operators to specify iterations:

1. *bulk iterations*, which are conceptually similar to loop unrolling, and
2. *delta iterations*, a special case of incremental iterations in which the solution set is modified by the step function instead of a full recomputation; they can significantly speed up certain algorithms since the work in each iteration decreases as the number of iterations goes on.

3.3 Experimental framework comparisons

One size fits all is a concept that rapidly proves to be wrong in Big Data processing frameworks.

Hadoop vs. Spark

In [Shi15] the authors analyze three major architectural components (shuffle, execution model and caching) in Hadoop and Spark and show that although Spark is more efficient than Hadoop, there is one case where for a Sort workload Hadoop MapReduce is twice faster than Spark, because of a more efficient execution model for shuffling data: MapReduce can overlap the shuffle stage with the map stage in order to effectively hide the network overhead.

Spark vs. Flink

In the framework of our BigStorage project we performed another comparative study [Mar16] which evaluates the performance of Spark versus Flink in order to identify and explain the impact of the different architectural choices and the parameter configurations on the perceived end-to-end performance.

Based on experimental evidence, the study points out that in Big Data processing there is not a single framework for all data types, sizes and job patterns and emphasizes a set of design choices that play an important role in the behavior of a Big Data framework: memory management, pipelined execution, optimizations and parameter configuration easiness. What raises our attention is that a streaming engine (i.e. Flink) delivers in many benchmarks better performance than a batch-based engine (i.e. Spark), showing that a more general Big Data architecture (treating batches as finite sets of streamed data) is feasible and may subsume both streaming and batching use cases.

4 Beyond MapReduce: more generic programming models for data analytics

4.1 Supporting SQL-like queries

Hive

Hive is a data-warehouse solution that is built on top of the Hadoop Ecosystem [Ash09]. It allows users to build SQL queries that extract data from several databases and file systems that integrate with Hadoop. These queries are later translated into MapReduce, Tez or Spark Jobs. It was initially developed by Facebook and is now used by several companies and offered in services like Amazon Web Services. In fact it was part for a long time of the core of Spark SQL.

The datamodel is divided into:

- Tables: they translate to HDFS directories. The data of each table is serialized and stored in files inside that directory. When a query is compiled or executed, tha data is deserialized thanks to the information stored in the system catalog about the serialisation format for that table.
- Partitions : denote the distribution of the data inside the directory. This is done through subdirectories that are named with the specific values of the partition, e.g a partition on columns date and country. Data with a particular value of 20091112 and US will be stored in subdirectories /date=20091112/country=US
- Buckets: data in partitions are further splitted into files based on the hash of a column in the table.

The query language supports select, project, join, aggregate, union and subqueries. It also supports UDFs and aggregations (UDAF) implemented in Java. Users can also create tables with specific serialisation, partitioning and bucketing options and load and insert operations. Moreover, users can use map-reduce scripts on the rows of the tables.

The architecture of Hive, illustrated in Figure 2, is composed of :

- A Metastore : stores metadata for the tables and information to keep track of the datasets ;
- A Driver : manages the execution of the SQL queries from beginning to end and communicates with the Hadoop ecosystem ;
- Compiler: invoked by the driver and to transform the query into an execution plan ;
- Optimiser: transforms the execution plan to get the best DAG to be executed in the Hadoop platform ;
- Execution Engine: passes all the map-reduce jobs to the execution engine ;
- CLI, UI, and Thrift Server: interaction layer between users and the driver.

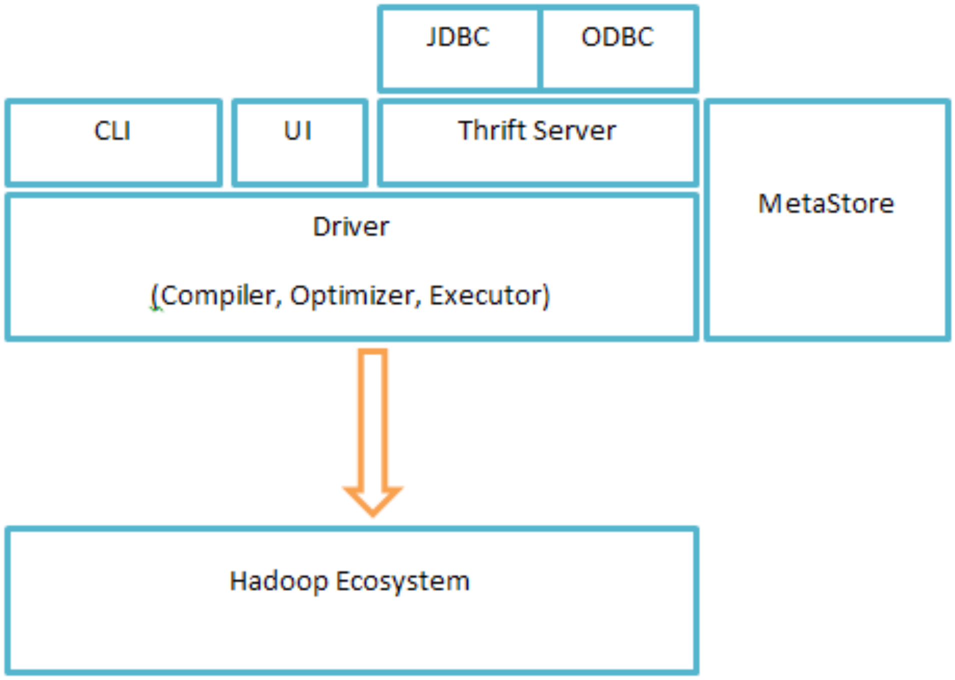

Figure 2. The Hive architecture [Ash09].

Spark support for SQL: Shark and SparkSQL

Shark uses the Spark engine to speed up SQL queries [Xin13]. It supports Hive queries with the advantages of an RDD: in memory data and fine grained fault-tolerance. It also includes some optimizations to the RDD abstraction:

- memory columnar storage and columnar compression
- optimized queries based on observed statistics

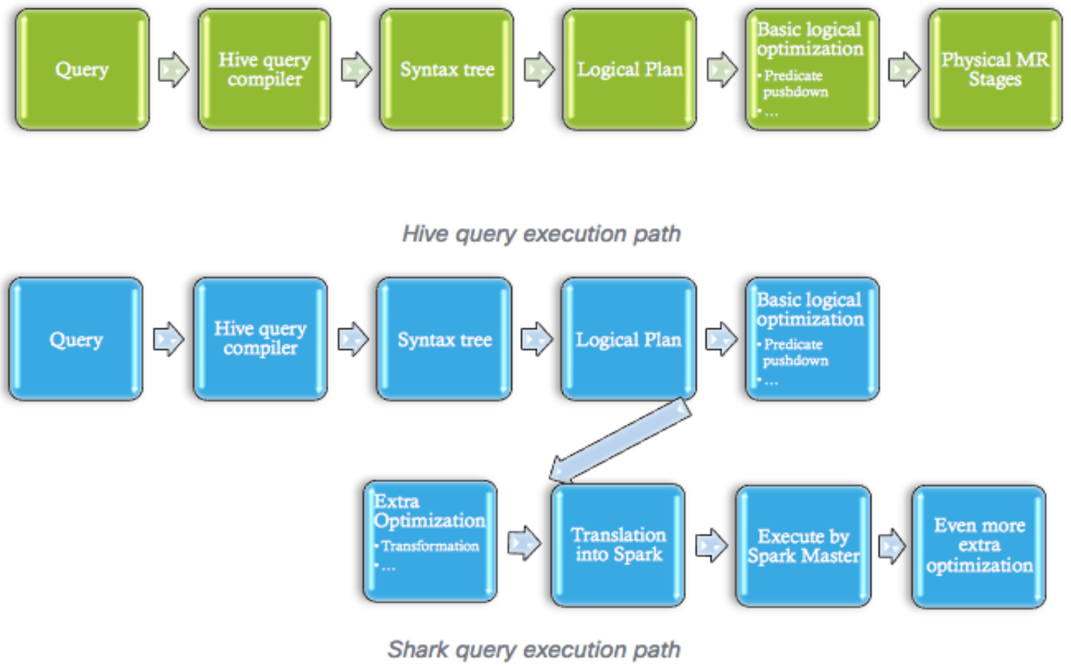

Figure 3. The Shark query execution plan [Lev13]

Shark used Hive's language, its metadata and its interfaces, being able to work over unmodified Hive data warehouses. However, that Hive basecode made it too difficult to maintain and optimize and it was eventually replaced by Spark SQL.

Spark SQL is the evolution of Shark based on all the insights learnt from Hive [Rey14]. Again, it can work with any existing Hive data sources and use its metastore. It provides an unification of the relation and procedural processing power of Spark, being able to mix SQL queries with normal procedural calls even with machine learning. Moreover, it provides support for semistructured data sources like JSON or parquet and infers characteristics of the data automatically. It also uses Catalyst [Arm15], an optimizer based on programming language features of Scala like pattern matching and quasiquotes. This engine analyses the query of the user to first create an optimal logical plan. Then this is translated into several physical plans that are evaluated thanks to a cost based model. The best physical plan is finally converted into Java code that is going to be run on the RDD. The objective is to free the user from the burden of optimizing queries. It can be extended with new rules for the cost model by the community.

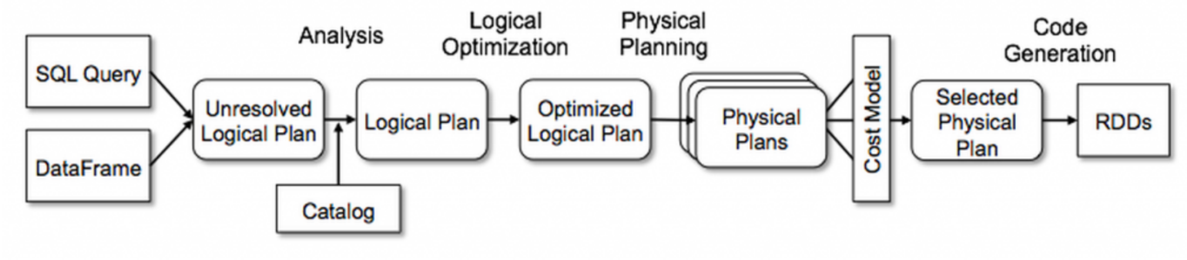

Figure 4. Optimiser flow in Catalyst [Arm15]

Flink support for SQL: Table API

Apache Flink uses the table API to allow SQL queries over batch or streamed data [Flia]. The queries can be made through an SQL syntax or an expression syntax using calls from the API. e.g.:

```
Table in = tableEnv.fromDataSet(ds, "a, b, c");
```

```
Table result = in.where("b = 'red'");
```

or

```
Table result = tableEnv.sql(
  "SELECT product, amount FROM Orders WHERE product LIKE '%Rubber%'");
```

The architecture of the system relies on optimizations made with Apache Calcite. The queries are validated against registered tables by the user and then converted to a Calcite logical plan. This plan is optimized by the engine depending on the data source: static or streaming. The optimization is executed as a normal Flink program with all the previous optimizations. The main purpose of this API is to provide a unique engine for both streaming and SQL like queries.

Coping with approximate queries: Blink DB

The main characteristic of BlinkDB is to allow fast queries over big amounts of data by using data samples [Aga13]. Different stratified samples of the data can be created through frequently used queries. These samples are later selected by a strategy that depends on a confidence interval and a time limit for a query, both defined by the user. The goal is to achieve fast query times by losing a small part of the information of the data.

4.2 Processing Big Data Streams

Streaming systems are dealing with unbounded input, unordered data streams at high data rates, and are required to process large amounts of data with low latency. Real-time data processing got recently a lot of attention in both academia and industry, with numerous open source frameworks now available in order to solve various use cases: Apache Storm [Tos14, R12], Apache Spark Streaming [Zah12], Apache Flink [Flink2], Twitter Heron [Heron], Apache Samza [Samza].

Stream processing in real-time: Storm, Flink, Twitter Heron, Spark Streaming

Storm

Big Data stream processing started more consistently with the first ideas that were gathered inside a project that will later become Apache Storm. To represent a stream as a distributed abstraction (to produce and process streams in parallel), two concepts were introduced [Nat05]: a *spout* produces new streams and a *bolt* takes streams as input and produces other streams as output; spouts and bolts are handled in parallel just like mappers and reducers in MapReduce; finally, a *topology* is a network of spouts and bolts.

In [Tos14] the authors describe the use of Storm at Twitter, its architecture and the way topologies are executed in Storm. A topology is a directed graph that represents computation through vertices and the data flow between components through edges. Vertices can be spouts and bolts. A typical spout can pull data from a queue such as Kafka. Bolts are responsible for processing incoming tuples and passing them to the next set of bolts.

The architecture can be summarized into the next actions: a client submits a topology to a master node which is responsible for the distribution and execution of a topology. A master coordinates a set of workers nodes, which run one or more worker processes. Each worker process is mapped to a single topology and starts a JVM in which one or more executors run one or more tasks.

Storm provides two types of processing semantics guarantees: *at least once* guarantees meaning each tuple of a topology will be processed at least once; *at most once* semantics dictates that a tuple is either processed once or is dropped in the case of a failure.

Heron

Pressed by an increased scale of data processing and the diversity of new use cases, along with many limitations of Storm [Kul15], Twitter developed Heron, a new stream processing framework which is compatible with Storm's API, with its main goals being performance predictability, improved developer productivity and ease of management.

One of the mechanisms that Heron implemented and which was missing in Storm is the backpressure mechanism (handling spikes and congestion): if the receiver component is unable to handle incoming data/tuples, in Storm the sender will simply drop tuples. Heron's tuple processing semantics are similar to that of Storm, hence neither Heron implements exactly-once guarantees, although the authors claim that the design allows it and they are considering its implementation [Kul15]. Finally, the authors mention that at Twitter, Storm was replaced by Heron, showing large improvements in both resource usage, throughput and latency.

Spark Streaming

Almost at the same time with Storm being released as an Apache open source project, Spark Streaming came into play with its discretized streams efficient and fault-tolerant model for stream processing [Zah12]. Spark Streaming takeaways are the high-level functional programming API, strong consistency (exactly-once) and efficient fault recovery (avoiding traditional replication or upstream backup where messages sent are buffered and replayed on a copy of the failed downstream node). It introduced the concept of *D-Streams* [Aki15]:

"*The key idea behind D-Streams is to treat a streaming computation as a series of deterministic batch computations on small time intervals" and it is based on RDDs. Spark Streaming capabilities are limited to in-order streaming processing and provide windowing semantics that are limited to tuple- or processing time-based windows"*

Samza

Apache Samza [Samza] is a distributed stream processing framework. It uses Apache Kafka for messaging and Apache Hadoop YARN to provide fault tolerance, processor isolation, security, and resource management. It features several properties:

- *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, replayable, 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.

Apex

Apex is a Hadoop YARN native platform that unifies stream and batch processing. It processes Big Data in-motion in a way that is highly scalable, highly performant, fault tolerant, stateful, secure, distributed, and easily operable [Apex].

[DEC] provides a survey of Kafka, Samza, Heron, Flink Streaming Systems.

Stream processing in Flink

Apache Flink provides one interface for batch processing (i.e the *DataSet* API) and another one for distributed stream processing (the *DataStream* API). The latter is dedicated to unbounded streams and enables Java and Scala programmers to write programs that implement various transformations on data streams, which can be either stateless (simply looking at one individual element at a time) or stateful (operations that remember information across individual events, e.g window operators).

Flink's basic building blocks are *streams* (intermediate results) and *transformations* (operations that takes one or more streams as input - sources, and compute one or more result streams from them as output - sinks). When executed, Flink programs are mapped to streaming dataflows (may resemble arbitrary directed acyclic graphs - DAG), consisting of streams and transformation operators. Each dataflow starts with one or more sources and ends in one or more sinks.

Parallelism in Flink. 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. The parallelism of a stream is always that of its producing operator. Different operators of the program may have a different parallelism.

One of the fundamental challenges of distributed stateful stream processing is providing processing guarantees under failures. To overcome existing approaches, which rely on periodic global state snapshots that impact the overall computation, a lightweight algorithm was proposed and implemented on top of Flink [Car15].

Flink's architecture for stream processing. The Flink runtime consists of two types of processes:

- The *master* process (also called JobManagers) coordinate the distributed execution. It schedules tasks, coordinate checkpoints, coordinate recovery on failures, etc. (high-availability setups includes multiple master processes, one of which is always the leader, the other remain in standby).
- The *worker* processes (also called TaskManagers, at least one) execute the subtasks of a dataflow, and buffer and exchange the data streams. Each worker is a JVM process and is splitted into task slots, each task slot representing a fixed subset of the resources.

The client is not part of the runtime and program execution, but is used to prepare and send a dataflow to the master. After that, the client can disconnect, or stay connected to receive progress reports.

Flink executes batch programs as a special case of streaming programs (a DataSet is treated internally as a stream of data), where the streams are bounded (finite number of elements) [Flink2].

Stream data transfer and management

Kafka

Apache Kafka is a distributed stream platform that provides durability and publish/subscribe functionality for data streams (making streaming data available to multiple consumers), being the de facto open source solution (for data durability and availability) used in end-to-end pipelines with streaming engines like Spark or Flink, which at their turn provides data movement and computation.

A Kafka cluster is a set of one or more servers that store streams of records in categories called *topics*. Each topic can be split into multiple partitions that are managed by a Kafka broker, a service collocated to the node storing the partition, that further enables consumers and producers to access the topic's data.

Apache Kafka gets used by:

- Producer services that are required to put data into a topic (Producer API).
- Consumer services that can subscribe and process records from a topic (Consumer API).
- Applications (e.g. streaming engines) that can consume input streams from one or more topics and can produce output streams to one or more output topics (Streams/Connector API)

More technical details about how Kafka handles data or how Spark and Flink integrate with Kafka can be further explored in [Kafka1, Kafka3, Kafka2].

Streaming SQL

Open source systems for distributed stream processing are evolving fast and the (STREAM) SQL interface for processing SQL queries on the fly as stream processing has recently started to be considered and implemented in both Spark [SparqS] and Flink [FlinkS].

The approach used so far was to extend their current APIs and to offer some basic operations while promising to cover in the future many unsupported operations, e.g. multiple streaming aggregations, sorting operations or joins between two or many (un)bounded datasets. It is not clear how the open source community will drive and extend the streaming SQL semantics, api and execution frameworks, but it is one of the topics that recently got an increased attention [Jai08].

4.3 Machine Learning tools for Big Data processing

ML tools in the Spark stack: MLlib

The machine learning support in Spark consists of three components [MLB]:

- MLlib: Apache's Spark distributed machine learning library. It inherits from the previous ones but it's now considered a library on its own out of the MLBase bounds. It is widely supported by the community and the most commonly used.
- MLoptimiser: optimises the whole machine learning pipeline.
- MLI: an API that facilitates feature extraction and algorithm development through an easy to use API

MLlib [MLLib] provides an API that uses the Apache Spark engine to execute common machine learning algorithms in a distributed setting [Spa]. It also uses Spark in memory data capabilities to speed up the computations. It is widely supported by the community and easy to use. It is also tightly integrated with other Spark components like Spark SQL or GraphX allowing to seamlessly change between them. Some of the main functionalities are:

- Several classification and regression algorithms
- Feature transformations: Standardization, Normalization, Hashing…
- Model evaluation and hyperparameters
- MLPipelines: spark.ml package allows to create sequences of machine learning steps like data preprocessing, feature extraction, model fitting and validation.
- Machine Learning persistence: Saving ML models for later use
- Distributed Linear Algebra: PCA, SVD.
- Statistics: Hypothesis testing

In addition, internal optimizations are applied to adapt these algorithms to a distributed setting in Spark. Some examples are optimizing Java Garbage collection times, using feature discretization to reduce communication costs and using C++ specialized libraries in the worker nodes [Men16].

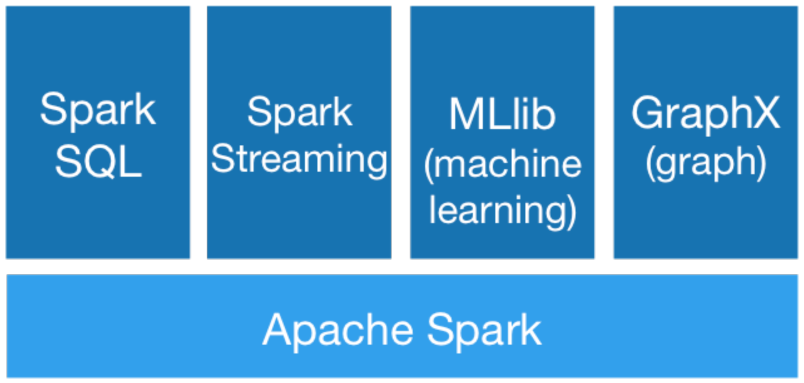

Figure 5. MLib sits on top of the Apache Spark Core and it can work on data read from streaming, graph or SQL sources.

ML tools in the Flink stack: FlinkML

Similar to MLlib in Spark, FlinkML [Flib] provides a series of machine learning algorithms to execute on the Flink engine. The main objective was to provide not only algorithms but also means to extract information from datasets and build maintainable machine learning pipelines. At the current version 1.2 it only allows to apply these algorithms over batch data sources. Also there are fewer algorithms available compared to Spark.

ML tools: Spark versus Flink

A short comparison of the ML-oriented features in Spark and Flink is summarized in the table below.

Characteristic	FlinkML	MLlib
Number of Algorithms	3	20+
Data Preprocessing	3	25+
Pipelines	Yes	Yes
Model Selection	Yes	Yes
Parameter Tuning	No	Yes
Basic Statistics	No	Yes
ML over Streaming	No	Yes

Conclusion: Spark not only has more algorithms and features but it also unifies the streaming, SQL and machine learning capabilities of the framework to use them together. The reason is that Spark always uses the RDD abstraction to analyze the data. At the moment MLlib provides more functionality than its Flink counterpart.

4.4 Graph processing

GraphX on Spark

GraphX adapts graph processing algorithms and abstractions to a distributed computation environment like Spark [GraphX]. It is a thin layer that sits on top of Spark and has several implementations of the most important graph algorithms like PageRank, Triangle Count to name a few. It allows to easily build graphs with tabular or unstructured data and view it as both a normal collection or a graph. The API provides graph operators like *subgraph, joinVertices* or a*ggregateMessages* that facilitate operations to the user. It also partitions the data of the graph in a pair of vertex and edge collections that are RDD's [Gon14]. The collections provide a special indexing and partitioning adapted to the graph. To achieve maximum efficiency GraphX implements several optimizations to exploit graph specific properties and operations.

Again the main benefit is that it provides a unified environment where we can use SQL to build graphs, or apply machine learning techniques over the graph structure. GraphX uses the RDD abstraction, same as MLlib, Spark Streaming and SparkSQL. Recently GraphFrames has been introduced, allowing to easily perform queries about the graph vertices and edges and even search for structural patterns in the data like, triplets of nodes A,B,C where A is connected with B and B connected with C, but A is not connected with C. For that reason GraphX can be easily adapted in many scenarios and by users with a lot of domain knowledge.

Gelly on Flink

Gelly is an API for Flink [Gelly] to create graphs from a set of vertices and edges read from files or collections. This Graph abstraction has several properties and functions that allows the user to get information of the graph and apply applications like map, translate or filter. Same as in GraphX it provides implementations for popular algorithms like PageRank or TriangleCount.

4.5 Workflows

The Dataflow Model

Massive scale data processing is now shaped by three main factors:

- Global-scale processed datasets are unbounded and unordered.

- Consumers impose more sophisticated requirements such as event-time ordering.
- Optimizations trade-offs between correctness, latency and cost.

With these aspects in mind, Google proposed a fundamental shift of approach [Aki15], that is to “stop trying to groom unbounded datasets into finite pools of information that eventually become complete” and to work with principled abstractions that allow the final user to choose the “tradeoffs along the axes of interest: correctness, latency, and cost”. In [Aki15] the authors proposed one such approach, giving the dataflow model composed by:

- A windowing model to support unaligned event-time windows.
- A triggering model to bind output semantics to runtime characteristics of the pipeline.
- An incremental processing model to integrate updates and retractions into the windowing and triggering models.

Tyler Akidau, one of the authors of the Dataflow model, makes in [Bey16a] and [Bey16b] a high-level survey of modern data processing concepts: streaming, unbounded data (processing), event time vs processing time, windows, sessions, watermarks, triggers, and accumulation.

Apache Beam

Apache Beam is the open source extension of the Google DataFlow Model, still a work in progress with yet no stable releases, that aims to propose a unified programming model in order to build pipelines that can be further executed by multiple runners, such as Apex, Flink, Spark or Google Cloud Dataflow commercial engine. The Beam capability matrix [Beam2] responds to four questions:

- *What is being computed?* (transformations)
- *Where in event time?* (event-time windowing)
- *When in processing time?* (watermarks and triggers)
- *How do refinements relate?* (accumulating and discarding).

With Apache Beam it is clear that the authors try to move the one size fits all solution paradigm from the execution layer to the semantics (API) layer, being built on top of many models in order to sustain efficient (un)bounded data processing. This work is highly influenced by Google, which is again reshaping the world of modern Big Data processing, allowing the final user not only to choose the tradeoffs of its interest between correctness, latency, and cost, but also to leverage various engine runners.

5 **Big Data processing: what requirements for storage?**

Big Data systems share the goals of traditional data warehousing systems (extract value from the analysis of data), while aiming to cope with their limitations. Both systems significantly diverge in the analytics processes and the organization of the source data. In practice, traditional data warehouses used to organize the data in repositories, collecting them from other source databases such as enterprise management systems, analytics engines, etc. Warehousing systems are poor at organizing and querying data streams like stream logs, sensor data, location data from mobile devices, etc. Big Data technologies aim to overcome these weaknesses of data warehousing systems by leveraging and harnessing new sources of data and new processing paradigms to facilitate scalable data analytics despite challenges due to unprecedented volume, velocity and variety of data.

The storage layer supports the processing of the data by providing core functionality such as data organization, access and retrieval of the data. This layer indexes the data and organizes them over the distributed store devices. The data is partitioned into blocks and organized onto the multiple repositories. The data to organize can be any one of several forms; hence, several tools and techniques are employed for effective organization and retrieval of the data. The examples include key/value pair; column oriented data, document database, semi structured XML data, raw formats etc.

– **File systems**: The file system is responsible for the organization, storage, naming, sharing, and protection of files. Big Data file management is similar to distributed file system, however the read/write performance, simultaneous data access, on demand file system creation, efficient techniques for file synchronizations would be major challenges for design and implementation. The goals in designing the Big Data file systems should include certain degree of transparency as mentioned below.

(a) Distributed access and location transparency: Clients are unaware that files are distributed and can access them in the same way local files are accessed. Consistent name space encompasses local as well as remote files without any location information.

(b) Failure handling: The data processing layer should operate even with a few component failures in the system. This can be achieved with some level of replication and redundancy.

(c) Heterogeneity: File service should be provided across different hardware and operating system platforms.

(d) Support fine-grained distribution of data: To optimize performance, individual objects should be located near the processes that use them.

Examples of file systems implementing the above features and consequently suitable for Big Data analytics include Apache HDFS [Hdfs] or Ceph [Ceph].

Distributed file systems can cope with large data volumes of various types. However, compared with a file system, a database allows a far better and more efficient access to the data, primarily due to the query language included. Currently the most widespread database is doubtlessly the relational database. Relational databases are a perfectly suited to transaction processing of structured data of a limited size. They are optimized for a record-wise parallel access by many users, as well as for inserting, updating and deleting records, while queries cause a higher timely effort. Big Data exceeds the size limits, includes a lot of data which is not structured, and it is about analyzing data which includes frequent queries. All these are aspects that show that relational databases are not the best choice for Big Data.

Consequently, Big Data analytics usually runs atop data storage models that are best suited for these specific requirements:

– **Key-Value Stores**: Key-value pair (KVP) tables are used to provide persistence management for many NoSQL technologies. The table has only two columns: one is the key, the other is the value. The value could be a single value or a data block containing many values, the format of which is determined by program code. KVP tables may use indexing, has tables or sparse arrays to provide rapid retrieval and insertion capability, depending on the need for fast look up, fast insertion or efficient storage. KVP tables are best applied to simple data structures and on the Hadoop Map Reduce environment. Examples of key-value data stores are Amazon's Dynamo and Oracle's Berkeley DB.

– **Document Oriented Database**: A document oriented database is a database designed for storing, retrieving and managing document oriented or semi-structured data. The central concept of a document oriented database is the notion of a document where the contents within the document are encapsulated or encoded in some standard format such as JavaScript Object Notation (JSON), Binary JavaScript Object Notation (BJSON) or XML. Examples of these databases are Apache's CouchDB [couch] and 10gen's MongoDB [mongo].

– **Column family/big table database**: Instead storing key-values individually, they are grouped to create the composite data, each column containing the corresponding row number as key and the data as value. This type of storage is useful for streaming data such as web logs, time series data coming from several devices, sensors etc. The examples are HBase [Hbase] and Google Big Table [Bigt].

– **Graph database**: A graph database uses graph structures similar to nodes, edges and properties for data storing and semantic query on the data. In a graph database, every entity contains direct pointers to its adjacent element and index look-ups are not required. A graph database is useful when large-scale multi-level relationship traversals are common and desirable for processing complex many-to-many connections such as social networks. A graph may be captured by a table store, which supports recursive joins such as Big Table [Bigt] and Cassandra [Cass]. Examples of graph databases include Infinite Graph [igraph] from Objectivity and the Neo4j [neo4j] open source graph database.

A detailed description of state-of-the-art storage systems in cloud and HPC systems that can serve as a basis for converged infrastructures is developed in deliverable D3.1.

6 Limitations and goals

The previous sections presented a detailed survey of state-of-the art systems for Big Data processing. In this section we summarize a set of gaps in the state-of-the-art we identified and state a number of corresponding challenges that will be addressed in the second phase of the BigStorage project.

6.1 Identified limitations of the state of the art

- There is a lack of **tools that sit between the parallel computing platforms for Big Data and the user**. The goal is to be able to explain the impact of changes in the cluster infrastructure (how many machines, what hardware, etc.) and framework configuration (a vast list of parameters and settings) to the user and provide good recommendations based on the use they give to their data.
- **Unifying the management of a multisite infrastructure** is also an open challenge. Allocating jobs depending on the data location, load of the machines, job characteristics and data privacy it is of great importance for many organizations nowadays.
- **Fast decisions** on our cluster infrastructure is also an interesting line of research. Deactivating machines to save energy depending on the current load or detecting node failures and communication problems in the cluster on real time can reduce costs for the organization
- **Big Data (streaming)** frameworks are not architected for efficiently processing data that is not limited to a single geographical location. One work that tries to fill this gap is JetStream [Tud14], a high performance batch-based streaming middleware for efficient transfers of events between cloud datacenters.
- **SQL streaming** for Big Data processing is an important interface that needs multiple enhancements. The difficulty of providing new features in a short time span (requires very fast development on top of existing frameworks) indicates some limitations: no general consent on semantics, need of more general abstractions that can easily integrate new use-cases, etc.
- RDDs are an efficient fault-tolerant immutable data structure for batch and stream processing. There is a need to efficiently define and implement a new abstraction **Hybrid Resilient Distributed Datasets** (HRDD) that can manage data (stream) sources to which the user is provided with partially (or fully) random access, while considering that data can be **both mutable and immutable.**
- The data storage system used as a data source for Big Data analytics greatly impacts the performance – and sometimes the available features – of the upper-level data processing system. Transitioning from relational databases to Big Data-oriented systems lead to the removal of some features such as **transaction processing**. As a new generation of Big Data storage systems now offer such functionality, there may be some possible improvements,

optimizations or new capabilities that could be added to existing big data processing frameworks. Such possibility should be further studied.

6.2 Goals

For the next steps of the project, we set up the following goals for WP2:

- Based on the identified limitations and bottlenecks of current frameworks for general workflow/streaming Big Data applications, formalize the corresponding requirements in relation with the use-cases studied in WP1.
- Propose processing models and techniques for optimized data transfers between workflow nodes and for efficient stream processing (WP2 T2.1).
- Research the efficient replication of data when we have a multisite structure, its constraints (e.g. data privacy) and its automation (WP2 Task 2.1).
- Apply machine learning techniques to model optimal decisions when preparing big data analytic processes (WP2 Task 2.3 and Task 2.1).
- Find and extract features of parallel computing applications and workflows and model their impact in the infrastructure and machines (WP2 Task 2.3).
- Model the level of utilization and load in one cluster and estimate the efficiency of an incoming job in that cluster WP2 (Task 2.1 and Task 2.3).
- Study multicriteria DSS that are able to take into account multiple aspects, including energy consumption, to help companies make decisions on how to build big data analytics processes (WP2 Task 2.2).
- Ultimately, design next generation data processing models, independent of any programming model, with a focus on general data orchestration for workflows and stream data processing (WP2 T2.1).

7 References

[Aga13] Agarwal, S., Mozafari, B., Panda, A., Milner, H., Madden, S., & Stoica, I. (2013). BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. *Eurosys'13*, 29–42. http://doi.org/10.1145/2465351.2465355

[Aki15] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernandez-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, Sam Whittle. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing, Proc. VLDB Endow., August 2015

[Aki16a] Tyler Akidau, The Evolution of Massive-Scale Data Processing https://docs.google.com/presentation/d/13YZy2trPugC8Zr9M8_TfSApSCZBGUDZdzi-WUz95JJw/present?slide=id.g63ca2a7cd_0_527

[Ale11] A. Alexandrov et al., "MapReduce and PACT - comparing data parallel programming models" in Proceedings of the 14th Conference on Database Systems for BTW 2011. Kaiserslautern, Germany, pp. 25–44.

[Apex] http://apex.apache.org/

[Arm15] Armbrust, M., Ghodsi, A., Zaharia, M., Xin, R. S., Lian, C., Huai, Y., … Franklin, M. J. (2015). Spark SQL. *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data - SIGMOD '15*, 1383–1394. http://doi.org/10.1145/2723372.2742797

[Ash09] Thusoo, A., Sarma, J. Sen, Jain, N., Shao, Z., Chakka, P., Anthony, S., … Murthy, R. (2009). Hive - A Warehousing Solution Over a Map-Reduce Framework. Sort, 2, 1626–1629. http://doi.org/10.1109/ICDE.2010.5447738

[Bdb] Michael A. Olson, Keith Bostic, and Margo Seltzer. 1999. Berkeley DB. In Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC '99). USENIX Association, Berkeley, CA, USA, 43-43.

[Beam] http://beam.apache.org/

[Beam2] http://beam.apache.org/documentation/runners/capability-matrix/

[Bey16a] https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-101

[Bey16b] https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-102

[Bigt] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. ACM Trans. Comput. Syst. 26, 2, Article 4 (June 2008), 26 pages. DOI: http://dx.doi.org/10.1145/1365815.1365816

[Car15] Paris Carbone, Gyula Fora, Stephan Ewen, Seif Haridi, Kostas Tzoumas. Lightweight Asynchronous Snapshots for Distributed Dataflows, 2015, https://arxiv.org/pdf/1506.08603.pdf

[Cass] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev. 44, 2 (April 2010), 35-40. DOI: http://dx.doi.org/10.1145/1773912.1773922

[Ceph] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. 2006. Ceph: a scalable, high-performance distributed file system. In Proceedings of the 7th symposium on Operating systems design and implementation (OSDI '06). USENIX Association, Berkeley, CA, USA, 307-320.

[Couch] Jan Lehnardt J. Chris Anderson and Noah Slater. CouchDB: The Definitive Guide. O'Reilly, first edition edition, 2010

[Data] https://databricks.com/blog/2015/02/17/introducing-dataframes-in-spark-for-large-scale-data-science.html

[Datb] https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html

[Dea04] Jeffrey Dean and Sanjay Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, OSDI 2004, San Francisco, http://dl.acm.org/citation.cfm?id=1251254.1251264.

[DEC] http://sites.computer.org/debull/A15dec/A15DEC-CD.pdf

[Dynamo] Swaminathan Sivasubramanian. 2012. Amazon dynamoDB: a seamlessly scalable non-relational database service. In Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12). ACM, New York, NY, USA, 729-730. DOI: http://dx.doi.org/10.1145/2213836.2213945

[Flia] https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/table_api.html

[Flib] https://cwiki.apache.org/confluence/display/FLINK/FlinkML%3A+Vision+and+Roadmap

[Flink] http://flink.apache.org/

[Flink2] https://ci.apache.org/projects/flink/flink-docs-release-1.2/concepts/index.html

[FlinkS] https://flink.apache.org/news/2016/05/24/stream-sql.html

[Gelly] https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/libs/gelly/index.html

[Gon14] Gonzalez, J. E., Xin, R. S., Dave, A., Crankshaw, D., Franklin, M. J., Gonzalez, J. E., … Stoica, I. (2014). GraphX: Graph Processing in a Distributed Dataflow Framework. 11th USENIX Symposium on Operating Systems Design and Implementation, 599–613. Retrieved from https://www.usenix.org/conference/osdi14/technical-sessions/presentation/gonzalez

[GraphX] http://spark.apache.org/graphx/

[Hada] http://hadoop.apache.org/

[Hadb] https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html

[Hao14] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, Ion Stoica, Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks, SoCC 2014, Seattle USA, pp. 6:1--6:15, http://doi.acm.org/10.1145/2670979.2670985

[Hbase] Hbase, https://hbase.apache.org/

[Hdfs] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST) (MSST '10). IEEE Computer Society, Washington, DC, USA, 1-10. DOI: http://dx.doi.org/10.1109/MSST.2010.5496972

[Heron] https://blog.twitter.com/2015/flying-faster-with-twitter-heron

[Igraph] Infinite Graph, http://www.objectivity.com/products/infinitegraph/

[Jai08] Namit Jain et al. Towards a Streaming SQL Standard, Proc. VLDB Endow., August 2008, pages 1379--1390, http://dx.doi.org/10.14778/1454159.1454179.

[Kafka] https://kafka.apache.org/

[Kafka1] http://kafka.apache.org/intro

[Kafka2] http://data-artisans.com/kafka-flink-a-practical-how-to/

[Kafka3] https://spark.apache.org/docs/1.6.1/streaming-kafka-integration.html

[Kul15] Sanjeev Kulkarni et al. Twitter Heron: Stream Processing at Scale, SIGMOD 2015, Melbourne, Victoria, Australia, pages 239--250, http://doi.acm.org/10.1145/2723372.2742788

[Lev13] http://blogs.cisco.com/security/big-data-in-security-part-ii-the-amplab-stack

[Mar16] Ovidiu-Cristian Marcu, Alexandru Costan, Gabriel Antoniu, Maria S. Perez-Hernandez, Spark versus Flink: Understanding Performance in Big Data Analytics Frameworks, CLUSTER, Sept. 2016, Taiwan, Taipei, https://hal.inria.fr/hal-01347638v2.

[Men16] Meng, X., Bradley, J., Street, S., Francisco, S., Sparks, E., Berkeley, U. C., … Hall, S. (2016). MLlib : Machine Learning in Apache Spark, *17*, 1–7.

[MLB] http://web.cs.ucla.edu/~ameet/mlbase_website/mlbase_website/index.html
[MLlib] http://spark.apache.org/mllib/
[Mongo] https://www.mongo.com
[Nat05] http://nathanmarz.com/blog/history-of-apache-storm-and-lessons-learned.html
[Neo4j] https://neo4j.com/
[Rey14] https://databricks.com/blog/2014/07/01/shark-spark-sql-hive-on-spark-and-the-future-of-sql-on-spark.html
[Samza] http://samza.apache.org/
[Shi15] J. Shi et al., "Clash of the titans: Mapreduce vs. spark for large scale data analytics," Proc. VLDB Endow., vol. 8, pp. 2110–2121, Sep. 2015, http://dx.doi.org/10.14778/2831360.2831365
[Spark] http://spark.apache.org/
[SparqS] http://spark.apache.org/docs/latest/structured-streaming-programming-guide.html
[Storm] http://storm.apache.org/
[Tan09] Stewart Tansley, Kristin Michele Tolle. The Fourth Paradigm: Data-intensive Scientific Discovery, Microsoft Research, 2009
[Tos14] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel*, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, Dmitriy Ryaboy. Storm @Twitter, SIGMOD 2014, Snowbird, Utah, USA, pages 147--156, http://doi.acm.org/10.1145/2588555.2595641
[Tud14] Radu Tudoran, Alexandru Costan, Olivier Nano, Ivo Santos, Hakan Soncu, Gabriel Antoniu. JetStream: Enabling high throughput live event streaming on multi-site clouds. Future Generation Computer Systems, Elsevier, 54: 274-291, 2016. DOI : 10.1016/j.future.2015.01.016
[War09] D. Warneke and O. Kao, "Nephele: Efficient parallel data processing in the cloud," in Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers 2009. NY, USA Portland, Oregon, 8:1--8:10, http://doi.acm.org/10.1145/1646468.1646476
[Xin13] Xin, R. S., Rosen, J., Zaharia, M., Franklin, M. J., Shenker, S., Stoica, I., … Xin, R. S. (2013). Shark: SQL and rich analytics at scale. *Proceedings of the 2013 International Conference on Management of Data - SIGMOD '13*, 13. http://doi.org/10.1145/2463676.2465288
[Zah12a] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica, Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing, NSDI 2012, San Jose, http://dl.acm.org/citation.cfm?id=2228298.2228301
[Zah12] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, Ion Stoica. Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters. HotCloud June 2012, Boston, MA, http://dl.acm.org/citation.cfm?id=2342763.2342773