FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

An Overview of Distributed Big Data Frameworks

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Moritz Becker, Bsc
Registration Number 1026241

to the Faculty of Informatics
at the TU Wien
Advisor: Prof. Dr. Reinhard Pichler

Vienna, 15th August, 2017

Moritz Becker	Reinhard Pichler

Erklärung zur Verfassung der Arbeit

Moritz Becker, Bsc
Am Steindl 7
3500 Krems an der Donau

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 15. August 2017

Moritz Becker

Acknowledgements

Ich danke meiner Familie, meinen Freunden und all jenen, die mich auf meinem Weg unterstützen. Ohne euch wäre ich nicht so weit gekommen.

Besonderen Dank möchte ich an dieser Stelle an Herrn Prof. Dr. Reinhard Pichler für die hervorragende, freundliche sowie unkomplizierte Betreuung meiner Diplomarbeit richten.

Abstract

The ever increasing amount of data that the modern internet society produces poses challenges to corporations and information systems that need to store and process this data. In addition, novel trends like the internet of things even adumbrate a prospectively steeper increase of the data volume than in the past, thereby supporting the relevance of big data. In order to overcome the gap between storage capacity and data access speed while maintaining the economic feasibility of data processing, the industry has created frameworks that allow the horizontal scaling of data processing on large clusters of commodity hardware. The plethora of technologies that have since been developed makes the entrance to the field of big data processing increasingly hard. Therefore, this thesis identifies the major types of big data processing along with the programming models that have been designed to cover them. In addition, an introductory overview of the most important open source frameworks and technologies along with practical examples of how they can be used is given for each processing type. The thesis concludes by pointing out important extension projects to the presented base systems and by suggesting the conduction of a performance-centric comparison of Apache Spark and Apache Hadoop that can help to establish a more profound understanding of the nature of these systems and to identify potential novel research topics.

Contents

CHAPTER 1

Introduction

Along with the steadily rising number of internet users and the increasing bandwidth of connections also the amount of data that is produced, transferred and stored is advancing rapidly. Moreover, people are not the only entities that are producing data. Novel technological perspectives like the internet of things (IOT) are just starting to gain traction and promise an explosive increase in the number of connected devices which clearly goes hand in hand with a further increase in the volume of data.

The storage capacity of modern IT infrastructure has followed a steady upward trend since its ignition and the storage costs have declined. At the same time huge progress has been made in terms of the computing capability of IT systems. The advancements in this area even initiated a shift towards novel storage media based on flash memory that allow for data access speeds that are several orders of magnitudes higher than traditional hard disks can provide and that prevented storage access from becoming a real bottleneck for data processing. Nevertheless, data access remains a problem. Around 1990, the content on a typical hard drive with a capacity of 1,370 MB and read access of 4.4 MB/s could be read in about 5 minutes. Today, over 25 years later a typical 256 GB solid state drive (SSD) provides read access of 550 MB/s resulting in almost 8 minutes for the entire drive to be read [Whi12]. So even when considering the fastest storage available today, the ratio between storage capacity and access speed has clearly deteriorated. Moreover, the cost of SSD storage is still fairly high compared to HDD which is why SSDs tend to be used exclusively for performance applications and hardly for any offline data processing. For example, Google is currently not interested in widespread use of SSDs in its data centers due to the higher costs per gigabyte [BYG+16]. But a 600 GB enterprise-grade HDD only delivers read access of just around 220 MB/s. In this scenario, reading the full disk takes well over 45 minutes! Under this impression it is unsurprising that companies such as Google started to struggle with the data load well over a decade ago and began to build their own solution to this problem in the form of distributed big data processing frameworks which will simply be referred to as big data frameworks for the remainder

of this thesis. Since reading data from a single disk is so slow the simple idea is to use computer clusters to distribute the data across in oder to be able to read from multiple disks simultaneously during data processing.

Since then, a whole new research field has evolved around this topic and many systems were created, often simultaneously and by different actors, be it industry or academia. Some of them were built from scratch while others were created to improve on the shortcomings of their predecessors. Many of them are not in use any more while others are unabated popular and lead the field. Moreover, there are various types of applications dealing with big data that demand different programming models and abstractions to facilitate the creation of applications that conduct their tasks efficiently.

The high research and development activity in big data processing have resulted in a level of complexity and variety in the field that makes it increasingly hard for newcomers to enter and catch on. The goal of this thesis is to provide a broad and rather high level overview of big data processing. This involves the identification of the relevant types of big data processing along with the programming models that support these processing types. The thesis should further identify and describe key technologies in the field and provide the reader with knowledge of how they work and how they can be used in practice. To sum up, this work sets out to constitute a comprehensive introductory guide to big data processing that provides the required orientation and knowledge that allows the reader to further specialize as desired. It is not the purpose of this thesis to extensively compare or rate the presented technologies and systems against each other. Moreover, due to the availability of technical insight and information this thesis only considers open source software.

Three major types of big data applications are identified by this thesis along with the programming models that are targeting these application types. These models are discussed in detail and an overview of some of the most popular big data processing frameworks implementing the respective model is provided. The framework descriptions not only present a software architectural view of the system but they also contain concrete examples including source code that follows a common fictional scenario presented in Section 1.2 that attempts to mimic real world big data use cases. The structure of this thesis is organized around the identified application types:

1. Batch Processing
2. Graph Processing
3. Stream Processing

Every application type is discussed in a separate part that is independent from the other parts. Each part presents a programming model that suits the respective application type along with two relevant big data frameworks that support these models. The programming model descriptions contain a separate section that introduces a problem

statement drawn from the AirQuality Inc. example domain and illustrates how this task can be solved in theory using the respective programming model. After that, each framework's description lays out the system architecture and provides insights to the practical use of the framework by presenting the implementation of the theoretical solution contained in the preceding programming model chapter.

As part of the introduction a short technology overview is presented in the following Section 1.1. After that, Section 1.2 presents the running example scenario that is used throughout the remainder of this thesis to showcase the use of the frameworks presented.

1.1 Technology overview

Figure 1.1 provides an overview of some of the available technology concerning big data frameworks. Note that this illustration is concentrated on the systems that are presented in this paper and gives by no means a complete picture of the ecosystem.

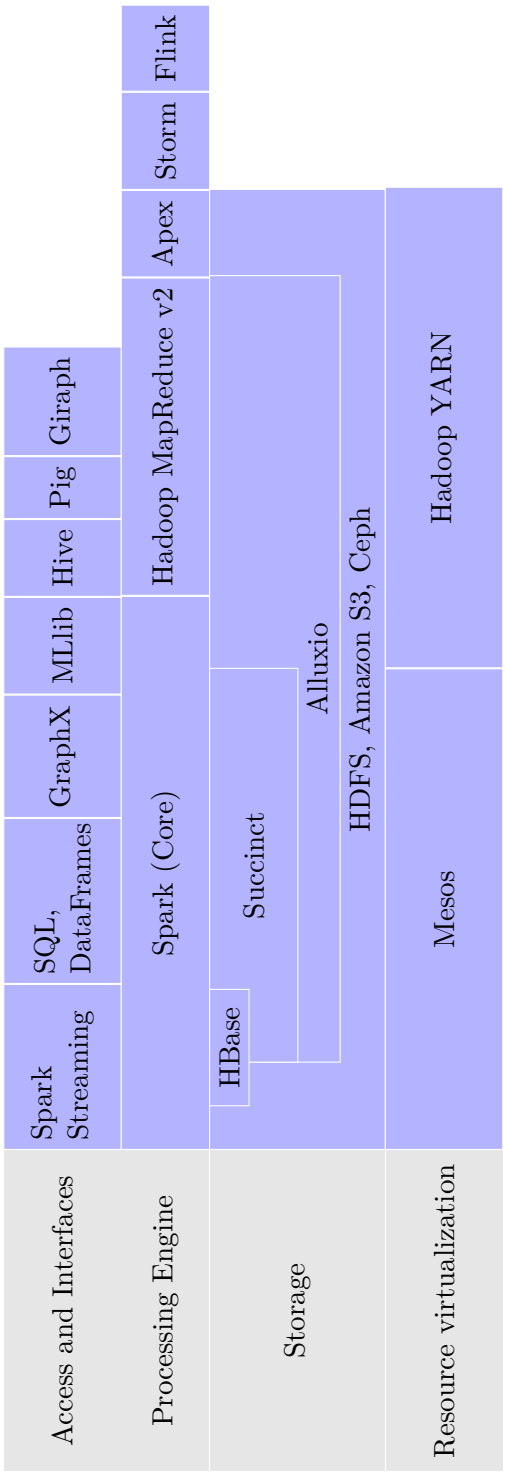


Figure 1.1: Big data technology overview

Mesos [1] and *Hadoop YARN* are cluster management systems that provide means to deploy third party applications to a cluster by allocating the required resources. Hadoop YARN is discussed in more detail in Section 3.3.

HDFS was originally developed as part of Hadoop and is a distributed file system that is used to reliably store data in a cluster to later access it for distributed processing. HDFS is further discussed in Section 3.1.

Amazon S3 [2] is a popular storage service provided by Amazon Web Services (AWS) and can be used as an alternative to HDFS for Hadoop installations that run on AWS.

Ceph [3] is another cluster storage system that allows for different types of storage abstractions like object, block and file system storage simultaneously.

Alluxio [4] a transparent distributed storage system that is deployed on top of any supported underlying storage system such as HDFS. The main idea of Alluxio is to speed up storage access by keeping data in-memory as much as possible. The concepts used by Alluxio are similar to the concept of resilient distributed datasets in Apache Spark as discussed in Section 4.1.

Succinct [5] is a storage system that allows the execution of a restricted set of queries on compressed data without decompressing the data first by using a compression scheme that facilitates random access to the compressed data. The benefit is an order of magnitude of reduction in required storage space for many tasks.

HBase [6] is a database that works on top of HDFS and is modeled after Google's BigTable database [CDG+08]. It is targeted to hosting huge tables in commodity clusters and to provide random, real time read/write access to the data.

Apex [7] is a big data framework that combines stream processing and batch processing and can be operated on top of a Hadoop YARN managed cluster.

Storm and *Flink* [8] are both distributed stream processing systems. Storm is discussed in more detail in Chapter 9.

Spark and *Hadoop MapReduce* are big data frameworks primarily focused on batch processing. They are further discussed in Chapter 4 and Section 3.3, respectively.

Spark Streaming is a Spark extension that allows to perform stream processing. Chapter 10 contains more detail about it.

[1] http://mesos.apache.org/
[2] https://aws.amazon.com/de/s3/
[3] http://ceph.com
[4] http://www.alluxio.org
[5] http://succinct.cs.berkeley.edu
[6] https://hbase.apache.org/
[7] https://apex.apache.org
[8] https://flink.apache.org/

Spark SQL [9] allows to apply SQL queries to DataFrames which are based on Spark resilient distributed datasets.

GraphX facilitates distributed graph processing on top of Spark. This extension is further discussed in Chapter 7.

MLlib [10] provides machine learning primitives for use on top of Spark.

Hive [11] provides a query language called Hive QL that is similar to SQL. The queries are translated to Hadoop MapReduce jobs to be executed on a Hadoop cluster.

In contrast to the declarative SQL-like interface that Hive provides, *Pig* [12] comes with a more procedural abstraction for writing Hadoop MapReduce called Pig Latin.

Giraph implements a programming model for distributed graph processing runs on top of Hadoop. Chapter 6 contains more details about it.

Core components of this ecosystem are described in the upcoming chapters of this thesis.

1.2 The AirQuality Inc. example domain

For each big data task type covered in this thesis, one real world problem from a common domain is presented and later solved using concrete means of the big data frameworks discussed. The proposed solutions are analyzed with respect to costs, complexity and framework specific characteristics. Most real world applications require big data frameworks targeted for different task types to complement each other in order to provide a valuable service. To illustrate such a scenario, a common, fictional problem domain is defined in the following.

AirQuality Inc. is an internet of things (IOT) business that sells access to fine grained air quality data and also offers a range of customer facing services based on this data. The company collects the data from specialized sensor devices that it sells to both public and private entities.

Such a sensor is solar-powered and can be mounted to almost any kind of structures like a house walls, for example. In regular intervals, the sensor analyzes the air quality at its location and transmits the measurements to a back-end system running in a data-center that is operated and maintained by AirQuality Inc. The transmitted data are considered to be tuples containing the following fields:

- Air quality indicator
- Region

[9] https://spark.apache.org/sql
[10] https://spark.apache.org/mllib
[11] https://hive.apache.org
[12] https://pig.apache.org

Region
Region A
Region B
Region C

Table 1.1: Regions that air quality data is received from

Indicator	Range
CO2	0 - 500
Fine dust	0 - 200
Radiation	0 - 50
Pesticides	0 - 100

Table 1.2: Measured air quality indicators and their value range

- Timestamp indicating the time of measurement
- Value

Tables 1.1 and 1.2 define the regions and air quality indicators that incoming data tuples may contain. Note that in reality, the air quality sensors would most likely transmit their GPS coordinates instead of region strings. However, the coordinates may be mapped to discrete regions in a pre-processing step that is not covered in this thesis.

The back-end system utilizes big data frameworks to process the received information. Preferably, the data is transmitted via one of the wireless wide area network (WWAN) stations operated by AirQuality Inc. While such a station provides a signal range of several miles/kilometers, many rural regions exist where this communication channel is not available. At the same time, the countryside represents an important customer segment because the local people are increasingly aware of the health risks posed by pesticides that are used by local farmers. Many of them are willing to install air quality sensors on their property to be notified when the wind blows pesticides onto their property. For this reason, AirQuality Inc. implements a secondary, peer-to-peer based data transmission functionality in their sensors. This technique establishes a sensor network by allowing nearby sensors to be used as message relays. As soon as a message reaches a sensor that is connected to a WWAN station, the relaying is terminated. Figure 1.2 illustrates the resulting topology.

For the problem statements derived from the AirQuality Inc. domain, refer to Sections 2.4, 5.2 and 8.6 for batch processing, graph processing and stream processing, respectively. The source code of the presented solutions is publicly available under `https://bitbucket.org/mobe1991/master-thesis-examples`.

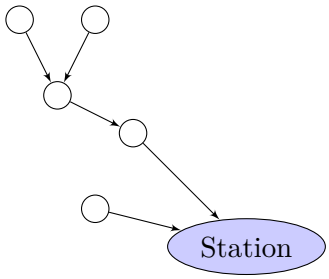


Figure 1.2: AirQuality Inc. sensor network topology

Part I

Batch Processing

CHAPTER 2

The MapReduce Programming Model

The MapReduce programming model was initially published by researchers from Google in 2004 and due to its conceptual simplicity and its applicability to a wide range of problems and applications it quickly emerged as the state of the art for doing distributed big data processing [DG04].

A MapReduce program consists of a *map* function and a *reduce* function. Such a program is executed on dataset, which is stored in chunks on a file system which, in practice, is a distributed file system. The map function turns a chunk into a sequence of key-value pairs. On the other hand, the reduce function combines all values with the same key to a result value.

An appropriate runtime environment provides an execution controller that allocates map tasks and reduce tasks which invoke the map and reduce operations, respectively. Apart from that, the controller steers the dataflow in the MapReduce program. It applies the map function to each chunk of the dataset and partitions the resulting key-value pairs by the reducer they are destined for. Once a map task is completed, the controller moves each partition to the responsible reducer task. Once all map tasks are finished and all data has been moved to reducer tasks, the partitions at each reducer task are combined and grouped by key. The reducer task then invokes the reducer function for each key. The procedure that the controller performs in between map and reduce is often referred to as the *shuffle* which is considered the heart of any MapReduce runtime environment.

Figure 2.1 schematically shows the functioning of a MapReduce program that counts word occurrences.

The MapReduce programming model is targeted towards cluster computing where node failures can happen any time. When a node fails, the map tasks and reduce tasks hosted

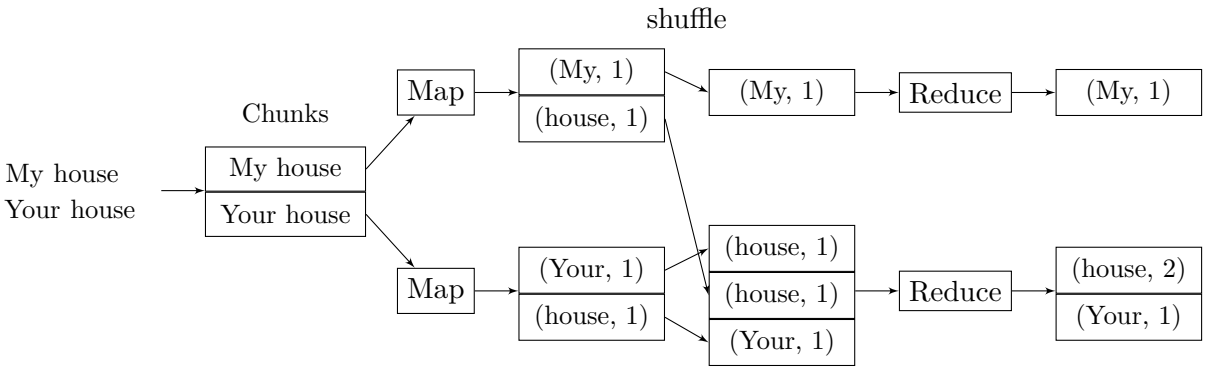


Figure 2.1: High-level view of a MapReduce program for word count

on the failed node are recreated on a healthy node by the execution controller. This is possible because the tasks implement a so-called blocking property, i.e. results are forwarded only once all work is finished. This prevents partial results from arriving at successive tasks which would be duplicated by the results delivered by recreated tasks [ABC+11].

2.1 Combiners

For reduce functions that are associative and commutative it is possible to pre-reduce the output of the map functions before the grouping takes place. In distributed environments where map and reduce functions run on different nodes, this pre-reduction can happen on the map node and is an important optimization since it minimizes the amount of data that needs to be transferred to the reduce node.

Example 1 To illustrate the purpose of combiners, assume a MapReduce program that aims to calculate the sum of each customer's purchases.

```
----- Chunk 1 -----
(customer1, EUR 43)
(customer2, EUR 21)
(customer1, EUR 33)
(customer2, EUR 102)
(customer2, EUR 10)
----- Chunk 2 -----
(customer1, EUR 13)
(customer2, EUR 4)
(customer1, EUR 73)
```

The result of the map function are the following key value pairs which would need to be transferred to the reducers if no combiner is utilized:

```
----- Chunk 1 -----
(customer1, [EUR 43, EUR 33])
(customer2, [EUR 21, EUR 102, EUR 10])
----- Chunk 2 -----
(customer1, [EUR 13, EUR 73])
(customer2, [EUR 4])
```

By applying a combiner function that sums up the output values, the data can be reduced to:

```
----- Chunk 1 -----
(customer1, [EUR 76])
(customer2, [EUR 133])
----- Chunk 2 -----
(customer1, [EUR 86])
(customer2, [EUR 4])
```

In contrast to example 1, if the program would aim to calculate the median of each customer's purchases, it would not be possible to utilize a combiner.

2.2 Extensions

Numerous extensions have been developed for the basic MapReduce programming model described so far. This section covers some of them briefly.

Not every problem is expressible in a single MapReduce program. Thus, extensions have been developed to allow the definition of MapReduce workflows where the output of one MapReduce task is used as the input of a successor task and so on. The sequence of operations can basically take the form of arbitrary directed acyclic graphs.

This workflow extension is crucial for the applicability of MapReduce for real world problems. Especially for more complex applications, the creation of MapReduce solutions by hand is time consuming and sometimes incomprehensible. Thus, systems have been developed on top of MapReduce that allow to create a declarative definition of a problem which is then translated into MapReduce workflows for execution. This approach is similar to relational database management systems that accept declarative SQL queries which are internally translated into a sequence of relational operations, the query plan. In fact, many systems such as Apache Hive [1] have been developed that translate SQL to MapReduce workflows.

Acyclic workflow extensions are of little help when dealing with algorithms that are recursive in nature such as:

[1] https://hive.apache.org/

- Fixpoint iterations as needed in the PageRank algorithm
- Calculating the transitive closure of a relation

Experimental systems like HaLoop have been created which support iterative MapReduce by iteratively invoking a job [BHBE10]. Hence, such systems can be used to emulate recursion via iteration. However, this approach is not very efficient since all MapReduce tasks must finish before the next iteration can commence which results in idle nodes. For real recursion, a task would need be able to produce output before all its input is consumed and the task ends. However, this contradicts the blocking property of tasks which is relied on for independent restart of failed tasks (see Section 2). Therefore additional methods for failure recovery would have to be introduced as discussed in [ABC+11].

2.3 Complexity & costs in MapReduce algorithms

For analyzing the quality of MapReduce algorithms it is important to define an appropriate cost model that is adapted to the operating conditions of the systems running those algorithms. Since the programming model is targeted towards computer clusters it is necessary to integrate the performance characteristics of such environments into the cost model.

A computer cluster is a set of computers or nodes that are connected via a network link. The bandwidth of these connections is slow compared to the processing speed of the individual nodes. Since map and reduce tasks can be independently executed on any cluster node, data consumed and produced by these tasks needs to be transferred over the network links. Therefore, the low bandwidth becomes a bottleneck for most applications.

As a consequence, a cost model for MapReduce algorithms must account for the cost of communication between nodes and can disregard the execution speed or computational complexity of the map and reduce functions which, in fact, tend to be very simple in practice and often run in linear time [LRU14].

Afrati et al [AU10] consider a MapReduce algorithm to be a DAG of map and reduce operations or processes. They define the *communication cost* of a process to be the size of its input. The cost of an algorithm is then the sum of the communication cost of all processes. Process outputs are not considered because every output is assumed to be the input to another process, hence outputs are already covered when considering inputs only. The size of the algorithm's overall output is not considered because it usually cannot be optimized anyway and, in practice, the output size is small compared to the input size because the output frequently is some form of aggregation.

However, this notion alone is insufficient as it tempts to assign all the work to a single process which would result in minimal communication cost but which would also greatly increase the wall-clock time, i.e. the actual execution time of the algorithm, because any

parallelism is eliminated. So some kind of balancing between communication cost and parallelism is required for achieving optimal results [LRU14].

For expressing the balancing of communication against parallelism, Afrati et al [SASU13] introduced a different model based on *reducer size* and *replication rate*:

- The reducer size q represents the maximum number of values per key, i.e. the maximum size of a reducer input. A small reducer size results in an increased number of reducer invocations. With many reduce tasks in place, this can increase the parallelism of the MapReduce program execution. By choosing a sufficiently low reducer size it is possible to ensure that reducers can hold the data in memory which greatly improves the performance.
- The replication rate r is the average number of key-value pairs to which each input is mapped by the map function. At the same time, this is the average communication cost between map and reduce tasks per input.

The tradeoff between communication cost and parallelism can then be described by expressing r as a function of q, i.e. $r = f(q)$. Examples 2 [SASU13] and 3 [LRU14] illustrate the tradeoff in concrete scenarios.

Example 2 Consider a cloud provider that charges users for communication and processor time. It is evident that the communication cost is proportional to r and the processor cost is a function of q. Hence, the computation cost for an algorithm on this cloud platform can be expressed as $ar + bq$ for some provider dependent constants a and b. For $r = f(q)$, this becomes $af(q) + bq$. Minimizing this expression results in some concrete value for q which can be used to look up the cost-optimal algorithm for some problem lying along the curve $r = f(q)$.

In case wall-clock time is more important than execution cost, the target function can be refined for some concrete problem. Assume an algorithm where the reducer has to perform pair-wise comparisons of its input values. The computational complexity of the reducer for a maximum of q input values trivially is in $O(q^2)$. Thus, the cost function can be extended to $ar + bq + cq^2$ for expressing the computational complexity of a reducer. This modification emphasizes a low reducer size q which results in higher parallelism and lower wall-clock time on the one hand but in potentially higher processor or communication cost on the other hand.

Example 3 To illustrate the tradeoff between reducer size q and replication rate r, the similarity join problem fits well. Given a domain D, a large set X of elements from D and a similarity function $s(a, b) : D \times D \to [0, 1]$ returning

a similarity measure for each element pair in the domain. The task is to find $\{(a, b) : a, b \in X, s(a, b) > t\}$ for some user defined threshold $t \in [0, 1]$.

For example, an instance of this problem is to find similar images in a large set - assume 1 million images and 1 MB per image, so 1 TB in total. One way to do this using MapReduce is to map the input data to one pair of images per key so each reducer would compute the similarity of two images by applying s and would then decide if the similarity is above the concrete threshold t.

While this algorithms looks fine on the first glance, it is, in fact, impractical because it requires an insane amount of communication between map tasks and reduce tasks. Each input image is mapped to 999.999 key-value pairs, hence the replication rate is 999.999. So the total number of bytes that needs to be transferred is 1.000.000 images times a replication rate of 999.999 times 2 MB for the size of each key-value pair containing 2 images. This is about 2×10^{12} MB or 2 exabytes. Transmitted over a single gigabit (128 MB/s) ethernet link would take $\frac{2 \times 10^{12}}{128 \times 86400} = \frac{10^{12}}{64 \times 86400} = 180.844$ days or 495 years.

A better approach is to form g groups and distribute the image set evenly accross the groups so that each contains $\frac{10^6}{g}$ elements. The map function produces $g - 1$ key-value pairs for each image I_i:

- The key is $\{u, v\}$ where u is the id of the group of I_i and v is any other group.
- The value is I_i.

It should be noted that the keys are considered to be unordered sets, i.e. a key $\{u, v\}$ is processed by the same reducer as $\{v, u\}$. Input to the reducer therefore consists of a key $\{u, v\}$ and an associated value list of size $2 \times \frac{10^6}{g}$ containing images I_i that belong to either group u or v. A reducer compares each pair of images (I_i, I_j) where the group of I_i is different from the group of I_j. To also cover the comparison of images residing in the same group, a reducer handling key $\{u, (u + 1) \mod g\}$ additionally compares image pairs where both elements belong to group u. Figure 2.2 illustrates the idea. The red lines indicate the comparisons of images in the same group that each reducer carries out in addition to the inter-group comparison.

The replication rate of this algorithm is $g - 1$ and the reducer size is $2 \times \frac{10^6}{g}$. The data that needs to be communicated between the map layer and the reduce layer is therefore $10^6 * (g - 1) * 1\text{MB}$. Choosing $g = 1000$, this results in roughly 10^9 MB or 1 Petabyte of data which is 1000 times less than in the previous approach.

MapReduce extensions as detailed in Section 2.2 allow algorithms to be composed of multiple MapReduce job invocations. Therefore it is often required to extend the notion

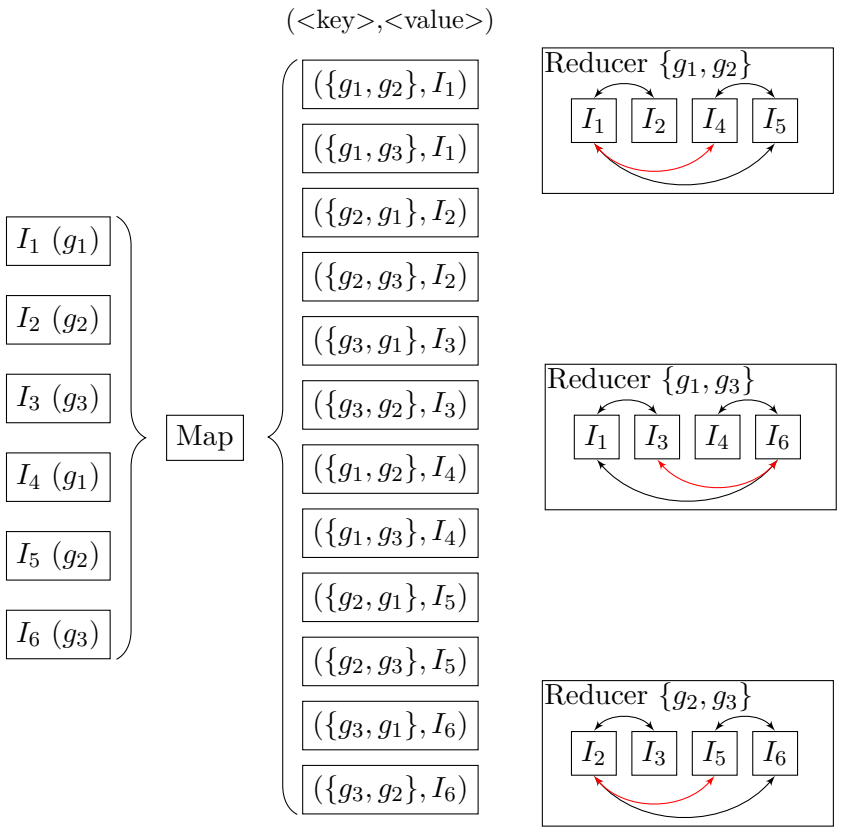


Figure 2.2: Illustration of MapReduce algorithm for similarity join

of MapReduce costs and complexity as presented so far to topologies of MapReduce jobs. Especially for iterative algorithms the number of job iterations needs to be incorporated into the costs and complexity analysis.

2.4 Running example: Air quality threshold monitoring

The following problem is used as illustrative real world example in the course of discussing Apache Hadoop and Apache Spark.

Suppose that national authorities ordered AirQuality Inc. to perform air quality measurement on their behalf to comply with international environmental protection treaties. Primarily, this involves the yearly creation of a report containing the number of days with threshold violations broken down by indicator and region.

The solutions to this problem are described in Sections 3.4 and 4.5 for Apache Hadoop and Apache Spark, respectively.

In general, this task can be accomplished using a pipeline of MapReduce jobs that operate on a set of air quality tuples as shown in Figure 2.3.

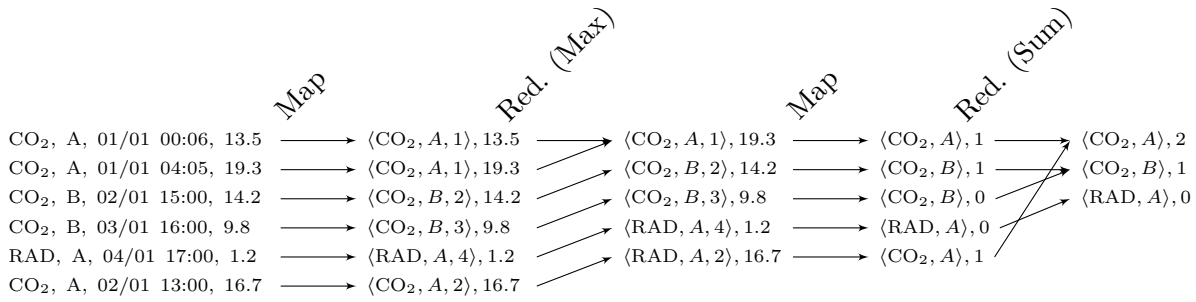


Figure 2.3: Pipeline of MapReduce jobs for generating air quality violation reports

The first job computes the maximum measurement values grouped by indicator, region and day producing output tuples of the form $\langle \text{indicator}, \text{region}, \text{dayGroup}, \text{maxValue} \rangle$. A subsequent job compares the maximum values per group against the thresholds and sums up the violations grouped by indicator and region producing a set of end results of the form $\langle \text{indicator}, \text{region}, \text{violations} \rangle$.

In terms of complexity, the only interesting part in the above algorithm is the maximum computation because this is where most of the data is reduced. Depending on the sensor sampling rate there might be thousands of measurements per indicator, region and day resulting in quite a big dataset that needs to be processed at this point in the pipeline. Since the output of this stage merely contains a single value per indicator, region and day, the remaining stages are negligible. Given an equal sampling size across the sensor topology and an evenly distributed amount of sensors per region and indicator, the reducer size for computing the maximum per indicator, region and day over 1 year is $|T|/(|R| * |I| * 365)$ for the set of tuples T, the set of regions R and the set of indicators I. Hence the job is very well parallelizable in practice and should be able to make good use of a large cluster of nodes because up to $|R| * |I| * 365$ may run in parallel.

CHAPTER 3

Apache Hadoop

Hadoop emerged out of the Apache Nutch project, a subproject of Apache Lucene, started by Doug Cutting and Mike Cafarella which aimed to build an open source web search engine. After starting Nutch in 2002, the project was soon inspired by Google's GFS and MapReduce papers. This resulted in the development of the Nutch Distributed File System (NDFS) and the porting of existing algorithms in Nutch to run on MapReduce and NDFS. In 2006, the MapReduce and NDFS parts were extracted from Nutch to form a separate subproject of Lucene called Hadoop and NDFS was renamed to Hadoop Distributed File System (HDFS). At that time, Cutting was hired by Yahoo! who assigned a dedicated team of developers to advance Hadoop and to incorporate it into Yahoo!'s web search infrastructure. By 2008, Yahoo!'s web search index was entirely generated by an Hadoop cluster [Whi12].

The remainder of this chapter describes the core components of Hadoop:

- HDFS. A distributed file system which is used to store large datasets in clusters of commodity machines.
- Classic MapReduce. Hadoop's original implementation of the MapReduce programming model (see 2).
- YARN. Hadoop's next generation MapReduce implementation.

The focus is put on HDFS since the majority of persistence layers of today's big data frameworks is based on it.

3.1 The Hadoop Distributed File System (HDFS)

In contrast to ordinary file systems, a distributed file system manages storage across multiple machines that are connected via a network.

3.1.1 Architecture

The architecture and design of HDFS is based on a set of assumptions and goals [hdfb]:

- HDFS is a distributed system targeted to operate on large server clusters. In such environments **hardware failure is the norm** rather than the exception. Therefore, fault detection and automatic recovery is a central goal.
- The target application for HDFS is batch processing and therefore, HDFS clients need **streaming data access** to datasets. This maximizes data access throughput rather than low latency access.
- HDFS applications typically work on **very large datasets**, i.e. many gigabytes or terabytes in size. HDFS is tuned to support such large files and to provide high, scalable bandwidth for accessing files.
- In a complex distributed system such as HDFS it is important to reduce failure modes. For this reason, HDFS offers a **simple coherency model** for files which restricts access to a write-once-read-many model. This restriction is also an enabler for high throughput access. Updates can, however, be appended to the end of a file.
- **Moving computation is cheaper than moving data**. The awareness of network speed as the most limiting factor for throughput is at the core of many of today's distributed systems. This becomes especially true in the case of HDFS when dealing with huge amounts of data. Thus, HDFS provides interfaces for applications to move *themselves* closer to the data they are operating on rather than instructing HDFS to move data.
- HDFS is designed to be *portable* to allow it to spread many platforms and to be used by a wide range of applications.

Basically, the HDFS architecture consists of two types of actors or nodes:

- NameNode (NN) - manages file system metadata
- DataNode (DN) - manages file contents

An HDFS cluster is a set of machines managed by HDFS to form a single coherent file system. Traditionally, there was only one NN per HDFS cluster and for the sake of simplicity we will carry on this assumption for the general illustration of HDFS in the following sections. In a later release of HDFS, a feature called 'HDFS high-availability' was added which alleviated the restriction on the number of NNs (see Section 3.1.6).

Figure 3.1 schematically shows the architecture of HDFS and how clients interact with it.

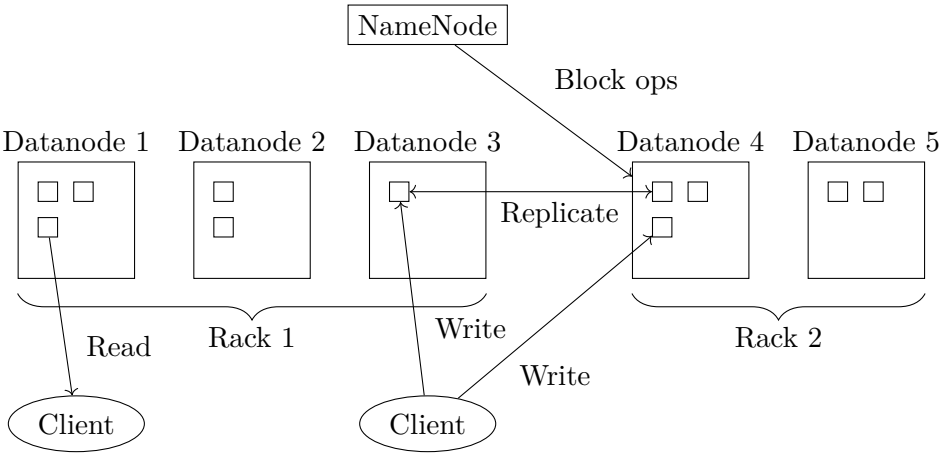


Figure 3.1: HDFS architecture

NameNode

A NameNode (NN) manages what is called the HDFS namespace which is nothing else than the directory tree in traditional file systems - a hierarchy of directories and files. These are represented via inodes which hold metadata such as permissions, modification and access timestamps. In addition, the NN also holds the block identifiers and the block locations for each file.

The content of files in HDFS is organized and stored in blocks of equal size. In contrast to traditional file systems, the block size in HDFS is much larger, typically 128 MB. The reason for this is to reduce the seek times when reading from the disk in order to utilize the disk transfer rate well and to maximize data throughput - one of the design goals of HDFS. The seek time is the time it takes for the hard disk's reader head to be placed before the start of the target block on the disk. Since large files are likely to touch multiple blocks it is possible to reduce the share of the seeks in the overall data access time by increasing the block size. Of course, this assumes that the content of an HDFS block is stored sequentially on disk. However, this argument does not apply for solid state drives (SSD) because the concept of a seek does not exist in this context. Although HDFS was designed in the pre-SSD age there are still good reasons for large block sizes even with SSDs:

- The NN keeps the block locations for each file in memory. Since a block size reduction implies an increase in the number of blocks it also implies an increase in memory consumption of the NN.

- File blocks read from HDFS by remote applications need to be transferred over a network link which involves a lot of overhead for setting up the connection etc. Thus, it makes sense to transfer larger blocks.

Conversely, because Hadoop MapReduce jobs operate on block level, pushing block sizes too hard impacts parallelism and thus the speed of job execution [Whi12].

The NN keeps the inode data, the block list and the block locations in memory to allow fast serving of clients requesting this information. The combination of inode data and block list is called *image*. The information contained in the image is the heart of the file system - if it is corrupted or lost, all data stored in HDFS is lost. Thus, there are persistent snapshots of the image called *checkpoint* or *fsimage* along with a persistent write-ahead-log called *edit log*. The edit log contains the history of file system changes since the last checkpoint. Every operation that modifies the file system is persisted in the edit log before being committed to the client. Checkpoints are created by replaying all edit log entries on the previous checkpoint. After creating a new checkpoint, the old one is deleted and the edit log is cleared. Checkpoint creation takes place:

- when the NN starts up
- on explicit request by an administrator
- via a CheckpointNode

On startup, the NN reads the current checkpoint and applies the history of changes from the edit log to it which results in the working image kept in memory. HDFS allows to store the checkpoint and the edit log in multiple storage locations such as NFS and other volumes. This is a recommended practice to protect from single volume and node failures.

By default, checkpoints are only created on NN startup. For long running HDFS clusters, this poses the risk of edit logs growing indefinitely and eating up all the disk space on the NN which endangers its stability. Moreover, large edit logs increase the startup time of NNs. For these reasons, HDFS provides the concept of a CheckpointNode (CN). The one and only purpose of such a node is to periodically create namespace checkpoints by downloading the current checkpoint and edit log from the NN, merging them locally and uploading the new checkpoint back to the NN. While HDFS does not prevent a single node from taking on both the roles of a NN and a CN, a separate node is usually chosen as a CN in practice because both roles impose the same memory requirements.

The HDFS user guide [hdff] also mentions the Secondary NameNode (SNN) along with the CN and a lot of confusion seems to be going on as to the differences between SNN and CN. It was suggested that the upload of new checkpoints back to the NN does not take place in the SNN scenario and that this is the essential difference [blo]. In fact, both SNN and CN perform exactly the same operations but the SNN has been deprecated in favour for CN when efforts were started towards providing a hot standby to the NN in order to pave the road for automatic NN failover. The original plan was to convert the SNN to a Standby NameNode but things have evolved differently leading to the introduction of CN and BackupNode and to the deprecation of SNN [hdfa].

The BackupNode (BN) maintains an in-memory, up-to-date copy of the namespace. The NN streams all edit log changes to the BN which applies the changes to its namespace

copy and also appends them to its own edit log. Thus, the NN's and BN's namespaces are always synchronized. This allows the BN to create new checkpoints simply from its own locale state without the need for fetching the latest checkpoint and the edit log from the NN. Having a BN in place also allows to operate the NN without any persistent storage.

DataNode

While an HDFS cluster can contain only a single NN (except for HDFS HA), it may comprise thousands of DataNodes (DNs) which hold the actual HDFS blocks that make up the content of files managed by HDFS. Each file block is independently replicated in HDFS and the number of copies or replicas per block is determined by a file's replication factor. The default replication factor is three, i.e. three copies of each block are stored in the cluster.

A block on a DN is represented by two files:

- Data file - contains the actual data
- Metadata file - contains metadata about a block including a checksum and the generation stamp

When an HDFS file system is formatted, a namespace ID is created for the file system which is persistently stored on all nodes in the cluster. A newly initialized DN receives the namespace ID when it joins a cluster. On startup, a DN contacts the NN and a handshake is performed where the DN compares its namespace ID and its software version with the NN. In case of any mismatch, the DN shuts itself down so that the integrity of the file system is not endangered.

After a successful handshake, the DN registers itself with the NN and receives a unique storage ID from the NN when the DN registers for the first time. The registration process also involve sending a block report to the NN which identifies all the HDFS blocks that are managed by the DN. Further block reports are sent periodically during the operation of the DN in order to keep the NN's view of block locations up to date.

DNs periodically send heartbeats to the NN that serve a multitude of purposes. First, a heartbeat signals to the NN that the sending DN and all the block replicas managed by it are still alive. A heartbeat message also carries data about the storage capacity and the I/O load of a DN. This information is used by the NN for decisions concerning space allocation and load balancing. Finally, a NN never communicates with DNs directly but it sends its commands as replies to heartbeat messages. Since such commands are important to protect the file system integrity it is essential to allow for frequent heartbeats.

The cluster administrator can order the removal or decommissioning of DNs. Once a DN is marked for decommissioning, it is not selected as replication target by the NN any more and the NN schedules the replication of blocks to other DNs. During this process,

the DN being decommissioned continues to serve read requests. Once the NN detects that all pending replications for the DN are completed, it marks the DN as decommissioned at which point it can safely be removed from the cluster.

3.1.2 File I/O

User applications access HDFS via an HDFS client library. This allows to make the access to the file system completely transparent, hence the user application requires no knowledge about the distributed nature of the system. Like ordinary file systems, HDFS allows to read, write and delete files and to create and delete directories.

For reading a file, the client contacts the NN to request the list of block identifiers for the file along with the DNs that host replicas of these blocks. Next, the client contacts the DNs directly to fetch the required blocks. The client does not only read the block content but also the block metadata containing a checksum and it performs a verification of the fetched block by comparing the received checksum with the checksum calculated from the received block content. When the client detects a corrupted replica, it notifies the NN and fetches a different replica instead.

While a client can read multiple blocks of a file in parallel, file writes take place one block at a time. For each block, the client requests DNs from the NN where the block replicas should be hosted. Then the client establishes a pipeline among the returned DNs in an order that minimizes the total network distance among the nodes and writes the block content along with a block checksum to the pipeline.

For example, let's assume a pipeline of 3 DNs DN_a, DN_b, DN_c. The client sends the block content to DN_a which persists the data and sends it to DN_b. DN_b, in turn, persists the data and sends it to DN_c which also persists the data.

HDFS implements a single-writer, multiple-reader access model for files, i.e. only on client at a time can write to a file while multiple clients can read from it concurrently, even while it is written.

3.1.3 Block placement, replication & integrity

The placement of block replicas in a distributed file system is important since it influences data reliability and availability as well as data access performance.

Computer clusters are commonly composed by racks of nodes where all nodes in a rack share a single switch and the rack switches are connected via a set of core switches. Thus, communication between nodes of different racks needs to go through multiple switches. More importantly, network links among nodes within the same rack sometimes provide higher bandwidth. For these reasons, intra-rack communication is generally considered faster than inter-rack communication.

HDFS relies on a heuristic for estimating network distances based on this assumption when it comes to block placement decisions. The placement policy is also configurable

and administrators can add rack-awareness by supplying a script which returns the rack number for a given node.

The default placement policy guarantees:

- No DN hosts more than one replica of a block
- No rack contains more than two replicas of a block (provided that there are enough racks in the cluster)

The NN constantly checks block replications when block reports from DNs arrive and if it detects any over- or under replicated blocks it issues commands to restore the desired replication factor.

In case of over replication the NN selects a replica to remove and in doing so, it attempts to balance storage utilization across the cluster without reducing a block's availability.

In case of under replication, the respective block is inserted into the replication priority queue where a block's priority is determined by its availability. I.e. a block which has only 1 replica left has a higher priority than a block with 2 replicas. The NN chooses a target DN for replication with the goal to minimize the cost of replication while keeping availability high.

The NN also detects when all block replicas end up on the same rack without actually falling below the replication factor. In this case it first treats the block as under replicated which causes the creation of a new replica on a different rack. This results in the block becoming over replicated which in turn triggers the removal of one of the replicas in the same rack.

Cluster balancing

The core components of HDFS do not guarantee the storage utilization to be uniformly distributed accross the cluster. The default block placement strategy does not take storage utilization of individual DNs into account and empty DNs might be added to the cluster at any time introducing a great skew to the block distribution.

For this reason, HDFS provides a standalone balancing tool which runs in the background and continuously checks if the cluster requires balancing based on a user defined threshold. The balancer moves replicas from nodes with higher storage utilization to nodes with lower utilization and in doing so, it maintains a replica's availability by guaranteeing that neither the number of racks hosting a replica nor the number of replicas for the same block is reduced.

Block monitoring

HDFS does not leave the detection of corrupted blocks solely to the checksum verifications performed by clients when reading files. Each DN runs a background process that

periodically scans the hosted blocks and compares a freshly computed checksum with the checksum stored in the block metadata. When a corrupted block is detected, the NN is notified which marks the respective replica as corrupted and schedules the replication of an intact replica. Once completed, the NN schedules deletion for the corrupted replica. This way, the invalid data is preserved for inspection through the user in case all other replicas are corrupt as well.

3.1.4 Snapshots

HDFS provides a snapshot mechanism which records that state of namespace and storage and allows to return to this state at a later point in time. This is especially helpful when applying software upgrades of HDFS components. To protect the file system from data loss in case of software bugs in the new version, a snapshot can be created before applying the update.

The snapshot creation can be triggered via an option on NN start up. In this case, the NN creates a new checkpoint and stores the it along with an empty edit log in a *new* location without overriding the old checkpoint. During DN handshake, the NN instructs the DNs to snapshot their locally stored blocks. If implemented trivially by copying each block, this would result in doubling the occupied storage of the whole cluster which is, of course, impractical. Instead, a DN creates a new storage directory where it only creates hard links to the blocks residing in the old storage directory. Thus, on block removal, only the hard link is deleted but the block remains in the old directory. The DN only create a copy of a block in the new directory when content is to be appended. Likewise, a snapshot restore can be triggered via an option when restarting the NN.

3.1.5 HDFS federation [hdfc]

The original architecture of HDFS allows only one NN per cluster. However, this prevents the namespace from being scaled horizontally. This can be an issue for large file systems where the memory requirements of the NN can become enormous because all information is kept in memory.

HDFS federation was introduced in a later release of HDFS for the purpose of making NN horizontally scalable. It allows for multiple NNs in the cluster where each NN is independently managing a distinct namespace. The NNs share all the DNs in the cluster for storage and each DN registers with each NN when starting up.

DNs contain blocks grouped in block pools which are managed independently and where each block pool belongs to a single namespace. This allows NNs to generate block IDs without the need for coordination with other NNs.

The namespace ID that is used as a cluster identifier in HDFS without federation becomes insufficient in HDFS federation. Thus, a separate cluster ID is introduced to supplement the namespace ID.

The summarized benefits offered by HDFS federation are:

- Scalability
- Performance
  The file system's I/O throughput can be improved by scaling across multiple NNs.
- Isolation
  A single NN failure does not take down the entire file system.

3.1.6 HDFS high-availability [hdfe] [Whi12]

While HDFS federation improves cluster scalability, it does nothing to improve the availability of individual namespaces. In the event of NN failure or maintenance operations such as software or hardware upgrade, a namespace or the whole cluster would become unavailable.

HDFS high-availability tackles this shortcoming by allowing to run two redundant NNs per namespace where one NN serves as hot standby for the primary/active node which serves all client requests in the cluster. Each NN runs a lightweight background process that monitors the NN process and initiates failover when necessary.

To allow for a fast failover it is important that both NNs hold the locations of HDFS blocks. This is realized by registering both NNs at the DNs resulting in block reports being sent to both NNs.

Apart from the block locations, also the remaining namespace data needs to be synchronized between the active and the standby NN. HDFS provides two distinct solutions for this which are explained in the following.

Shared Storage [hdfe] [Whi12]

The shared storage based snychronization requires the active NN and the standby NN to have access to a shared storage such as an NFS mount.

The active NN logs any namespace changes to an edit log stored in a directory on the shared storage while the standby NN monitors this edit log and applies any changes to its own namespace state. In the event of a failover, the standby node applies all outstanding changes from the edit log and enters the active state.

Special care needs to be taken for avoiding a split-brain scenario with this solution. There are basically two kinds of failover scenarios: graceful and ungraceful.

The former case is triggered manually by an administrator and allows to perform an ordered transition for the NNs to switch roles leaving no room for a split-brain situation to develop.

On the other hand, in an ungraceful scenario, failover is triggered by a sudden, unexpected unavailability of the active NN. However, it is not certain if the active NN process did indeed halt or if a temporary network slow down or partition has occurred, for example.

Therefore, when the standby NN enters the active state there are potentially two active NNs in the cluster. This situation leads to the divergence of namespace states managed by both NNs independently and results in file system inconsistencies and data loss. As a consequence, it is crucial to put fencing mechanisms in place to reliably cut of access to the shared storage for the unavailable NN.

Quorum Journal Manager [hdfd]

The active NN streams the namespace changes to a group of separate JournalNodes (JNs) instead of writing them to a shared storage. The standby NN monitors the JNs and applies the modifications to its own namespace state.

To prevent a split-brain scenario, the JNs only allow one NN as a writer. In the event of a failover, the standby NN takes over the role of writing to the JNs. So even if the previously active NN remains running, it is not able to write any changes to the JNs.

3.2 Hadoop Classic MapReduce [Whi12]

Hadoop Classic MapReduce is the initial implementation of MapReduce in Hadoop. It was later supplemented by YARN (see Section 3.3) but is still supported.

The unit of work in Hadoop's classic MapReduce implementation is called a *job*. It consists of the input data, the MapReduce program and a job configuration. Hadoop divides a job into two types of *tasks*: *map tasks* and *reduce tasks*.

For steering the job execution, a single *job tracker* and one *task tracker* per task is deployed. The job tracker is responsible for scheduling tasks for execution on task trackers which execute the task and send progress reports back to the job tracker. When a task fails, the job tracker will reschedule the task.

Task trackers have a fixed number of slots for map tasks and reduce tasks depending on the node resources. This limits the simultaneous work loads that can be executed by a task tracker.

Figure 3.2 illustrates the general data flow in Hadoop MapReduce jobs. Red lines indicate data being moved over the network while transitions on the same node are outlined by dashed arcs. A job's input data is divided into fixed-size *splits* by Hadoop where each split consists of *records*. For each split, one map task is scheduled which receives the split as an input and runs the user defined map function on each record in the split. Small split sizes lead to a better load-balancing on computation nodes. On the other hand, the smaller the split size, the higher the overhead fraction becomes for scheduling and processing the split. Apart from that, the data locality optimization performed by Hadoop should be considered for choosing the block size.

As detailed in Section 3.1, the input data for a MapReduce job is stored on HDFS which distributes the corresponding data block replicas across the cluster. Since bandwidth is considered the scarcest resource in such an environment, tasks are scheduled as close

to the data as possible. Considering that one input split might be composed of several blocks, this means that the scheduler attempts to move the computation to a node which hosts the largest fraction of required blocks. In case of a real world cluster, the probability of one node hosting all split blocks is very low, hence some movement of data almost certainly has to take place for a split size spanning multiple blocks. This is the reason why the optimal split size equals the HDFS block size for most applications since this choice maximizes the likelihood of processing a split without data movement.

The output of map tasks is locally sorted by key to later allow efficient merging of results from different mappers on the reducer side. All sorted mapper outputs are stored on a node's local file system since they only represent a temporary result and are deleted after being processed by the reduce task. Storing the data in HDFS would involve too much overhead and would not pay off. When the result of a map task is lost due to a node failure, for example, the map task is rescheduled by the job task on a different node to reproduce the lost data.

The reduce tasks cannot exploit data locality because the map tasks are usually run in parallel on different nodes in the cluster storing the map results at these nodes. Once the mapper results have been moved to the responsible reduce tasks, the map outputs are merged to form one input chunk grouped by key for each reducer. This step also involves sorting the chunks by key because Hadoop guarantees the reducer invocations to be ordered by key. Finally, a reducer's result is stored in HDFS for reliability.

Hadoop MapReduce jobs may also be defined without any reducer. In this case, the map output is directly persisted to HDFS.

To minimize the amount of data that needs to be transferred to the reduce tasks, Hadoop MapReduce supports combiner functions which can be thought of as reduction pre-processors that are executed on the output of map tasks before the data is transferred to the reducer nodes (see Section 2.1).

3.3 Yet Another Resource Negotiator (YARN) / MapReduce 2.0

Hadoop's classic MapReduce implementation has some major shortcomings that impact the scalability and agility of the system in large clusters. These issues were brought up by Yahoo! engineer Arun C. Murthy in 2007 [hada]. Development on a redesigned implementation began in 2010 and was finished in 2011 [Whi12] [hada]. Figure 3.3 depicts the architecture of YARN.

Although MapReduce 2.0 introduces new APIs for writing MapReduce jobs, most existing compiled MapReduce jobs should be binary compatible and should thus run fine on YARN. However, there may be source incompatibilities depending on the APIs used. Also, the MapReduce command line interface remains compatible with Classic MapReduce [yar].

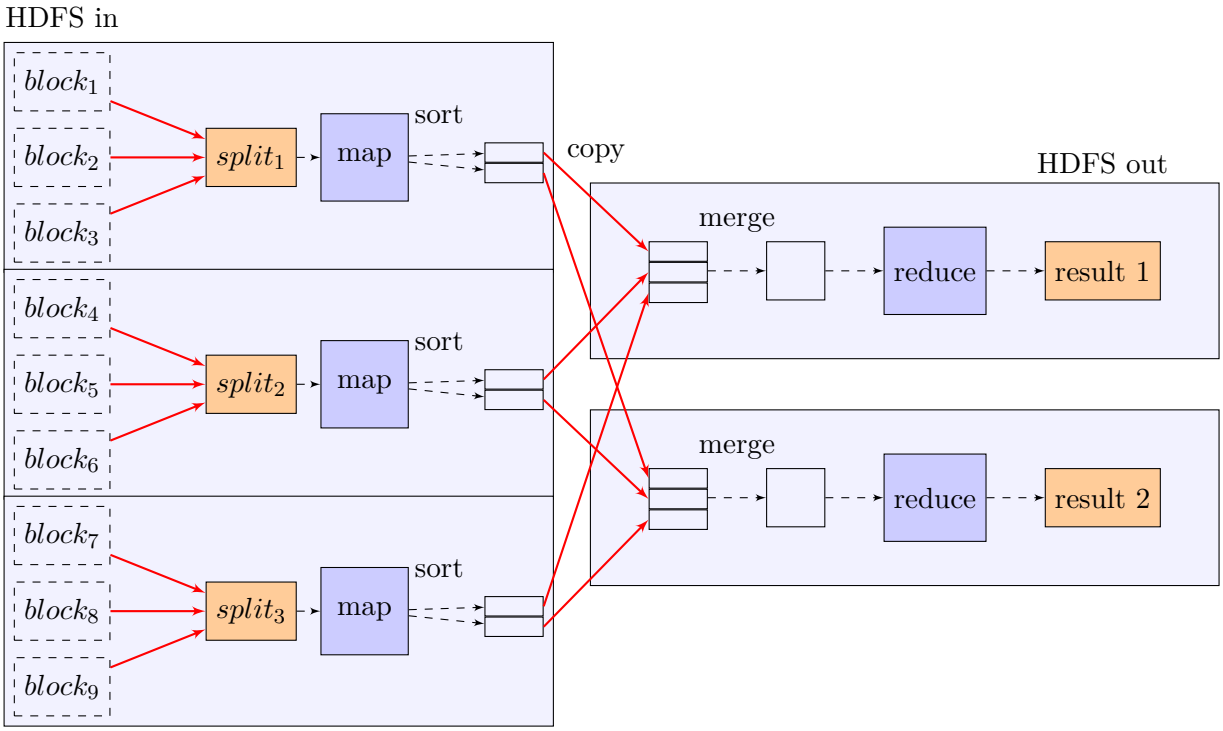


Figure 3.2: Hadoop MapReduce data flow

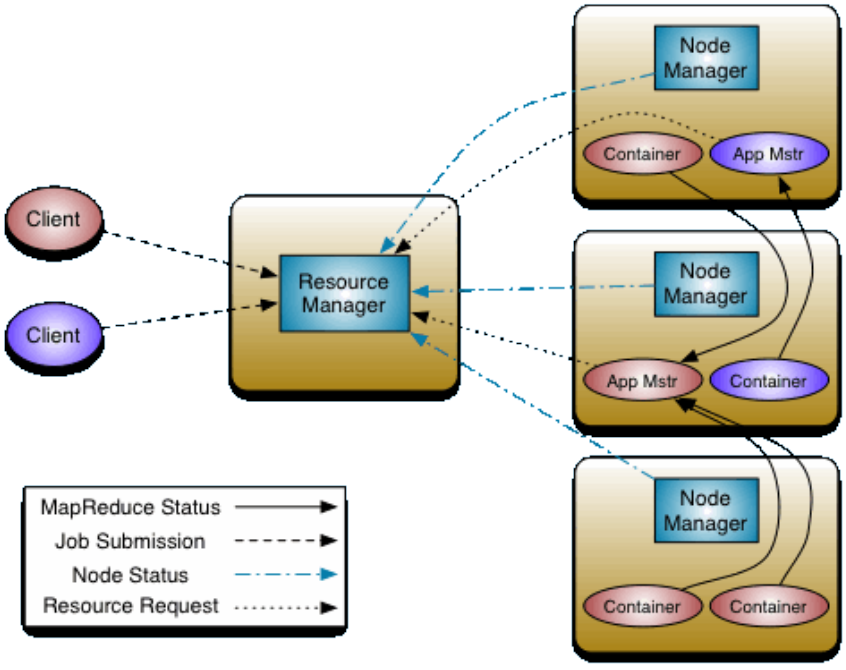


Figure 3.3: YARN architecture [hadb]

The groundbreaking change in MapReduce 2.0 is the reorganization of the job tracker responsibilities into two separate entities. Resource management is taken over by a global *ResourceManager* (RM) in conjunction with *NodeManager* (NM) daemons running on the cluster nodes and job scheduling/monitoring is handled by a per-application *ApplicationMaster* (AM). The AM requests resources for job executions from the RM and works together with the NMs to execute and monitor the tasks [hadd].

In this new setting, the RM is very generic and, in fact, independent from the applications running on the cluster, thereby facilitating programming models beyond MapReduce [mur].

With classic MapReduce, software updates had to be applied for the whole Hadoop cluster at once which posed problems for operability. MapReduce 2.0 uses new wire protocols for network communication which allows different versions of clients and servers to work together and thereby eliminating the need for big bang software updates [mur].

As mentioned in Section 3.2, task trackers have a fixed number of map and reduce slots. In practice, this turns out as a major drawback for cluster utilization because either map slots or reduce slots are scarce in a cluster at different times, wasting node resources that are locked up in empty slots for the under-utilized task type [mur].

MapReduce 2.0 takes a more abstract approach to resource allocation by treating each node as a set of isolated containers each occupying a fraction of the node resources without statically allocating quotas to specific task types [mur].

YARN also brings great improvements in terms of availability. In classic MapReduce, the job tracker is a single point of failure - if it fails, any job that was running needs to be re-submitted so all the results that were produced are lost. MapReduce 2.0 allows for manual or automatic RM failover by using an Apache Zookeeper cluster for storing its state. Thus, in case of failure, a secondary RM can quickly take over by reading the state from Zookeeper. Still, all previously running applications need to be re-submitted but this is not as tragic as in MapReduce 1.0 since AMs are able to create checkpoints of their work that are stored on HDFS. This way, not all work is lost and has to be redone as it would be the case with classic MapReduce [mur] [hadc].

3.4 Air quality threshold monitoring with Hadoop

This section describes an implementation of the solution to the air quality monitoring problem statement from Section 2.4 using Apache Hadoop.

In essence, a job in Apache Hadoop is represented by a Java archive (JAR) file containing any external classes that are required for executing the job as well as a class with a main method that acts as a client for the Hadoop cluster. It defines the job structure, initializes the required datasets in HDFS, submits the job to the Hadoop cluster, waits for its completion and retrieves the results from HDFS. The Hadoop distribution provides a command line utility that can be used to invoke an Hadoop JAR. Listing 3.1 shows the main method for the air quality report generation.

```
1  public static void main(String[] args) throws Exception {
2      Path hdfsMaxPath = new Path("data/max");
3      Path hdfsViolationsSumPath = new Path("data/violationSum");
4      java.nio.file.Path localMaxPath = Paths.get("data", "max");
5      java.nio.file.Path localViolationsSumPath = Paths.get("data", "
           ↪ violationsSum");
6
7      String hadoopClusterIp = System.getProperty("hadoop.cluster.address", "
           ↪ localhost");
8      Configuration conf = new Configuration();
9      conf.set("fs.defaultFS", "hdfs://" + hadoopClusterIp + ":9000");
10     conf.set("mapreduce.framework.name", "yarn");
11     conf.set("yarn.resourcemanager.address", hadoopClusterIp + ":8032");
12     conf.set("mapreduce.app-submission.cross-platform", "true");
13     conf.set("mapreduce.output.textoutputformat.separator", ";");
14
15     FileSystem fs = FileSystem.get(conf);
16     fs.copyFromLocalFile(new Path("data/tuples"), new Path("tuples"));
17
18     if (fs.exists(hdfsMaxPath)) {
19         fs.delete(hdfsMaxPath, true);
20     }
21
22     Job job1 = Job.getInstance(conf, "Max");
23     job1.setJarByClass(AirQualityReportJob.class);
24
25     job1.setMapperClass(MaxMapper.class);
26     job1.setCombinerClass(MaxReducer.class);
27     job1.setReducerClass(MaxReducer.class);
28     job1.setOutputKeyClass(AirQualityDayGroup.class);
29     job1.setOutputValueClass(DoubleWritable.class);
30     job1.setInputFormatClass(AirQualityTupleInputFormat.class);
31     job1.setOutputFormatClass(MaxOutputFormat.class);
32     TextInputFormat.addInputPath(job1, new Path("tuples"));
33     TextOutputFormat.setOutputPath(job1, hdfsMaxPath);
34
35     job1.waitForCompletion(true);
36
37     copyAndMerge(fs, hdfsMaxPath, localMaxPath);
38
39     if (fs.exists(hdfsViolationsSumPath )) {
40         fs.delete(hdfsViolationsSumPath, true);
41     }
42
43     Job job2 = Job.getInstance(conf, "Violations␣Sum");
44     job2.setJarByClass(AirQualityReportJob.class);
45     job2.setMapperClass(ThresholdMapper.class);
46     job2.setCombinerClass(SumReducer.class);
47     job2.setReducerClass(SumReducer.class);
48     job2.setOutputKeyClass(AirQualityGroup.class);
49     job2.setOutputValueClass(IntWritable.class);
50     job2.setInputFormatClass(MaxInputFormat.class);
51     TextInputFormat.addInputPath(job2, hdfsMaxPath);
```

```
52      TextOutputFormat.setOutputPath(job2, hdfsViolationsSumPath);
53
54      job2.waitForCompletion(true);
55
56      copyAndMerge(fs, hdfsViolationsSumPath, localViolationsSumPath);
57  }
```

Listing 3.1: Hadoop air quality report job definition

Lines 8 - 13 define the configuration to use for running the Hadoop job. In line 9, the endpoint for the HDFS file system is set. The job makes use of the newer YARN architecture rather than the classic MapReduce implementation as specified in line 10. To this end, an endpoint for talking to the YARN resource manager is configured in line 11. The configuration property specified in line 12 enables cross platform support for job submission and makes it possible to submit a JAR from a Windows host to a Hadoop cluster running on Linux. Line 13 merely specifies that a semicolon should be used in outputs generated by Hadoop to separate key-value pairs.

Hadoop expects the data to operate on to be present in HDFS. Thus, it is required to manually upload the respective files in the main method before submitting the job which happens in lines 15 - 16. First, a handle for HDFS is created using the previously defined configuration. The `copyFromLocalFile` method is then used to copy the local dataset located in `data/tuples` to a file `tuples` on HDFS.

The output of a single Hadoop job is stored as a file in HDFS and it is necessary to specify the HDFS path that should be used to store output. E.g. for Job 1 this is done in line 33. However, when the same job is executed repeatedly, old outputs from previous job runs might still be present in HDFS. In such a case, Hadoop does not simply overwrite old results but terminates the job execution with an error. Thus, in lines 18 - 20 we delete any existing outputs prior to starting the job.

The job definition itself consists of specifying the classes for mapper, combiner and reducer that implement the business logic of the MapReduce job. For Job 1, this is done in lines 25 - 27. It is also necessary to specify the types of output keys and values as shown in lines 28 - 29. For reading and writing input and output files it is possible to specify custom formats as done in lines 30 - 31. Finally, the input and output HDFS file paths are set in lines 32 - 33. A Hadoop job is submitted and started by calling the `waitForCompletion` method with a boolean parameter that indicates whether job progress should be logged to the console or not. This method blocks until the job is completed or failed.

The output that is produced by a Hadoop job on HDFS consists of a directory containing a separate file per reducer containing the reducer's output. To obtain a view of the overall job results on the client it is required to manually copy the reducer files to the client and to merge them into a single file which is what the call to `copyAndMerge` on line 37 does.

In contrast to Spark, Hadoop requires a lot of configuration work to yield a working job and also the manual interactions with HDFS increase the complexity of creating a job as demonstrated in the preceding paragraphs. However, the implementation of the mappers and reducers remains very straightforward as shown in the following. A mapper class in Hadoop needs to be a subtype of `org.apache.hadoop.mapreduce.Mapper` and likewise, a reducer class needs to be a subtype of `org.apache.hadoop.mapreduce.Reducer`. A `map` or a `reduce` method needs to implement the respective business logic. Both methods receive a context object that can be used to emit new key-value pairs.

Listing 3.2 shows the implementation of the mapper for computing the maximum measurements per indicator, region and day.

```
1  public static class MaxMapper extends Mapper<NullWritable, AirQualityTuple,
       ↪ AirQualityDayGroup, DoubleWritable> {
2      @Override
3      protected void map(NullWritable key, AirQualityTuple value, Context
           ↪ context) throws IOException, InterruptedException {
4          context.write(
5                  new AirQualityDayGroup(
6                          value.getIndicator(),
7                          value.getRegion(),
8                          value.getTimestamp().query(MaxMapper::
                               ↪ queryDayOfYearGroup)
9                  ), new DoubleWritable(value.getValue())
10         );
11     }
12
13     private static int queryDayOfYearGroup(TemporalAccessor
           ↪ temporalAccessor) {
14         return temporalAccessor.get(ChronoField.MONTH_OF_YEAR) * 31
15                 + temporalAccessor.get(ChronoField.DAY_OF_MONTH);
16     }
17 }
```

Listing 3.2: Hadoop mapper for maximum computation

The `map` method barely constructs and emits one new key-value pair per input tuple where the key constructed as an instance of `com.bitlawine.bigdatathesis.examples.hadoop.AirQualityDayGroup`.

Listing 3.3 depicts the corresponding maximum reducer.

```
1  public static class MaxReducer extends Reducer<AirQualityDayGroup,
       ↪ DoubleWritable, AirQualityDayGroup, DoubleWritable> {
2      @Override
3      protected void reduce(AirQualityDayGroup key, Iterable<DoubleWritable>
           ↪ values, Context context) throws IOException,
           ↪ InterruptedException {
4          OptionalDouble max = StreamSupport.stream(values.spliterator(),
               ↪ false)
5                  .mapToDouble(DoubleWritable::get)
6                  .max();
```

```
7
8           if (max.isPresent()) {
9               context.write(key, new DoubleWritable(max.getAsDouble()));
10          }
11      }
12  }
```

Listing 3.3: Hadoop reducer for maximum computation

It is passed the reducer key and and all corresponding values that it iterates through to compute the maximum which is then emitted on line 9.

Listing 3.4 shows the implementation of the mapper for the second MapReduce job that indentifies and sums up the threshold violations per indicator and region. It receives its input from the maximum reducer and creates new key objects stripping the day number from the key. Moreover, the measured maximum value is compared against the threshold and either 1 or 0 is assigned as value to the new key depending on whether the threshold is violated or not.

```
1   public static class ThresholdMapper extends Mapper<AirQualityDayGroup,
        ↪ DoubleWritable, AirQualityGroup, IntWritable> {
2       @Override
3       protected void map(AirQualityDayGroup key, DoubleWritable value,
            ↪ Context context) throws IOException, InterruptedException {
4           context.write(
5                   new AirQualityGroup(key.getIndicator(), key.getRegion()),
6                   new IntWritable(value.get() > key.getIndicator().
                        ↪ getThreshold() ? 1 : 0)
7           );
8       }
9   }
```

Listing 3.4: Hadoop mapper for threshold violations

Finally, Listing 3.5 shows the reducer that sums up the threshold violations that were emitted in the mapper from Listing 3.4 and emits the existing key along with the computed sum.

```
1   public static class SumReducer extends Reducer<AirQualityGroup, IntWritable
        ↪ , AirQualityGroup, IntWritable> {
2       @Override
3       protected void reduce(AirQualityGroup key, Iterable<IntWritable> values
            ↪ , Context context) throws IOException, InterruptedException {
4           int sum = StreamSupport.stream(values.spliterator(), false)
5                   .mapToInt(IntWritable::get)
6                   .sum();
7           context.write(key, new IntWritable(sum));
8       }
9   }
```

Listing 3.5: Hadoop reducer for summing up threshold violations

CHAPTER 4

Apache Spark

Spark was started by Matei Zaharia at the University of California Berkeley in 2009 and presented in a research paper in 2010[ZCF+10]. In 2013, the project was donated to the Apache Software Foundation and it became a top-level project in 2014.

The motivation for creating Spark was to better support MapReduce algorithms that reuse a working set of data across multiple parallel operations. For example, this includes any algorithm using iterations and is often the case in machine learning. Another problem in traditional MapReduce implementations like Hadoop is the high latency of operations which is impractical for a range of applications like ad-hoc interactive analytics or serving web client requests. The reason for this is the invocation of a new MapReduce job for each request which potentially needs to load data from across a cluster every time [ZCF+10].

The basic primitive introduced by Spark to alleviate these shortcomings is called a resilient reliable dataset (RDD). It represents a read-only object collection that can be distributed in partitioned form across a cluster and can be cached in memory to be reused in multiple parallel operations. Availability is achieved via lineage, i.e. enough information is maintained about how a particular RDD has been derived from other RDDs which allows the selective recreation of lost partitions [ZCF+10].

Spark is cluster agnostic, i.e. it does not rely on being able to manage cluster resources itself but relies on a third party cluster manager to integrate with [spaa] [spab]. This architecture prevents reinventing the wheel and, more importantly, it allows Spark to operate alongside other cluster applications by sharing the cluster manager [ZCD+12]. Spark supports the following managers [spaa] [spab]:

- Spark Standlone
  This is the default cluster manager that comes with Spark. It is good for getting started fast or for operating Spark on clusters where no other applications ought to run [spab].

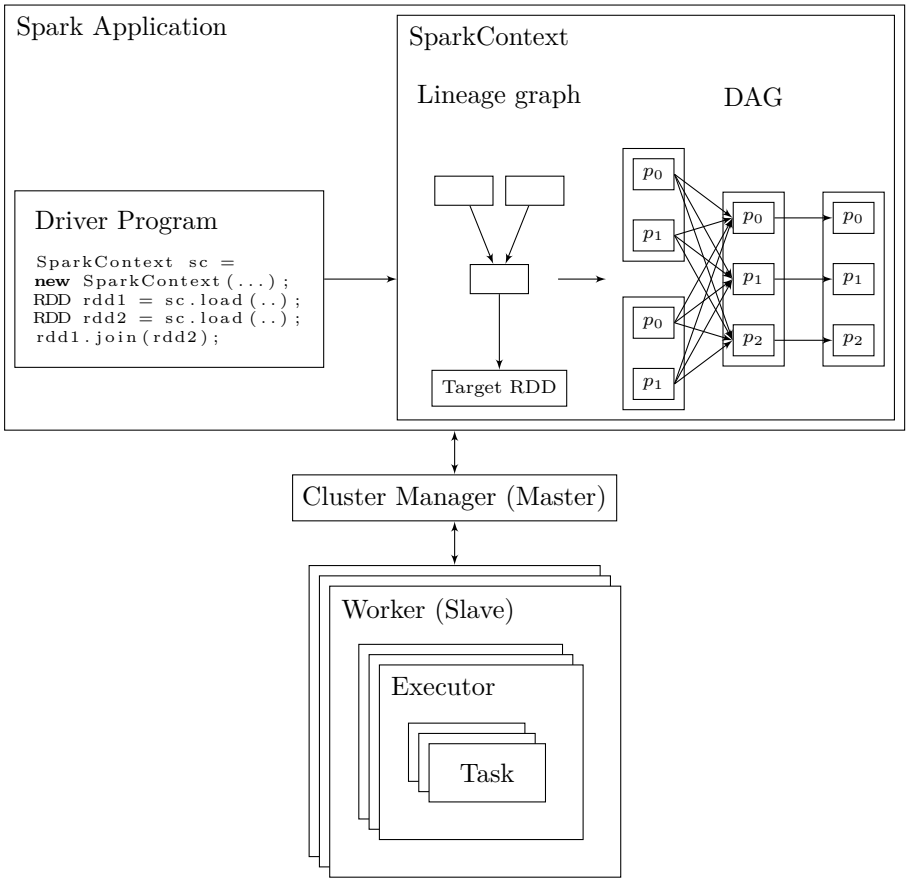


Figure 4.1: Apache Spark execution workflow

- Apache Mesos [1]
- Hadoop YARN (see Section 3.3)

Spark programs are created in the form of *drivers* that create a SparkContext and define the computation using RDDs [ZCF+10]. The computation plan gets passed to the job scheduler that transforms it into a task set which, in turn, is passed to a task scheduler that interacts with the cluster manager to allocate resources and to spawn the required task executors [ZCD+12] [spal]. Figure 4.1 illustrates the execution workflow. While Hadoop spawns a new JVM for each task that is executed on a node, a task in Spark is just a pooled thread in an already running JVM, hence Spark jobs are much more lightweight than Hadoop MapReduce jobs [spah].

The remainder of this chapter covers RDDs in detail before Spark's job scheduler is described subsequently.

[1] `mesos.apache.org`

4.1 Resilient Distributed Dataset (RDD)

RDDs are read-only, partitioned collections of records. They can only be created from

- another RDD
- data in stable storage

by applying a set of predefined operations, called *transformations*, like *map*, *filter* or *join* [ZCD+12].

The partitions of an RDD can be distributed across a cluster, thus allowing different nodes to operate on distinct fractions of the data. Since RDDs are designed to be held in memory and to be evaluated lazily, mechanisms are required to cope with lost partitions due to failures of nodes holding a partition in memory [ZCD+12].

This is done by ensuring that every partition of an RDD is independently recomputable if necessary. Part of the information stored in any RDD describes how this RDD has been constructed. This property is also called *lineage* and includes a set of dependencies - other RDDs - and a function for computing the current RDD from its dependencies. Spark distinguishes between two types of dependencies [ZCD+12]:

- *Narrow* dependency: Each partition of the parent RDD is used by at most one partition of the child RDD
- *Wide* dependency aka shuffle dependency: Partitions of the parent RDD are used in multiple child partitions

Dependency types can be used to classify transformations into narrow or wide transformations depending on what kind of dependencies a transformation introduces. For example, *map* transformations lead to narrow dependencies whereas shuffle style operations like *reduce* result in wide dependencies (see Figure 4.2). The distinction of dependency types has an important impact on planning and scheduling the computation of an RDD because narrow transformations are considered eligible for pipelining by Spark whereas wide transformations are treated as pipeline breakers [ZCD+12] [spac]. The details of the scheduling process are discussed in Section 4.2.

The dependencies between RDDs result in a directed acyclic graph, the *lineage* graph, that allows every RDD and each of its partitions to be transitively recomputed based on data in stable storage [ZCD+12]. See Figure 4.3 for an illustration of the lineage graph for a slightly extended instance of the word count problem. The basic word count problem assumes a given text corpus and the task is to count the number of appearances of each distinct word in the corpus. To make the example more interesting, the complexity of the problem is slightly raised by reading the corpus from two input files rather than one and by applying some sort of filtering on sentence level prior to counting.

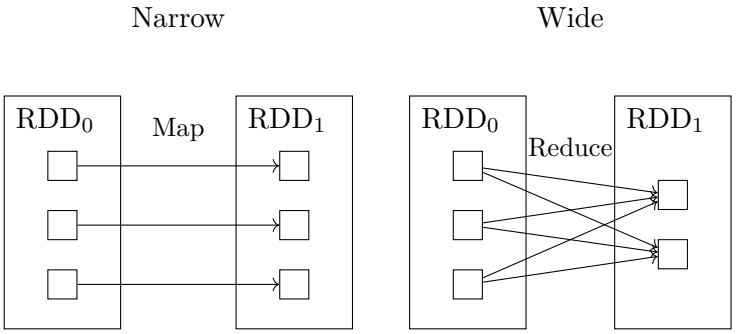


Figure 4.2: RDD dependency types

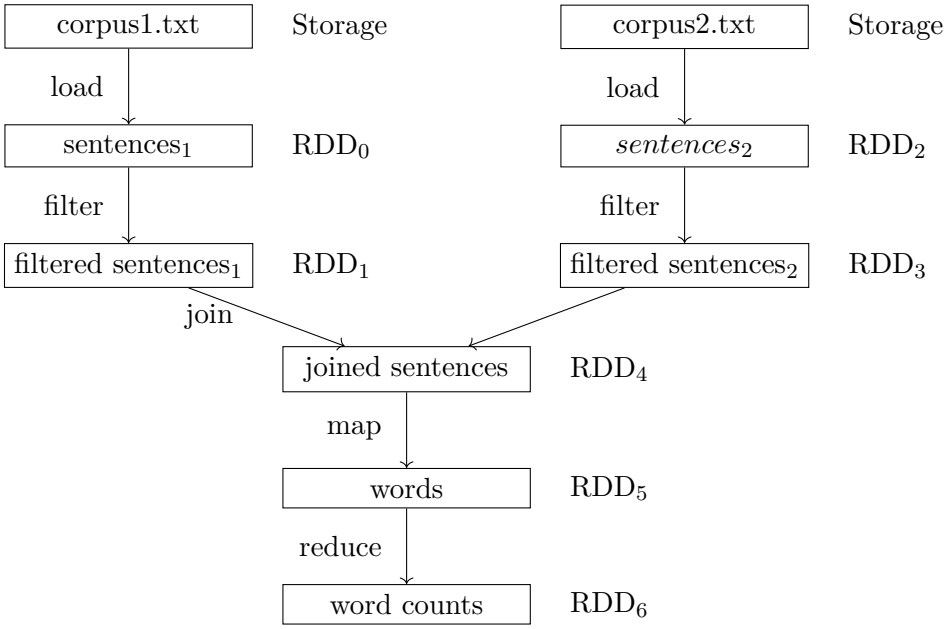


Figure 4.3: Lineage with RDDs

The left branch of the lineage graph represents the input from the first file named *corpus1.txt* in this example and the right branch stands for the input from the second file named *corpus2.txt*. RDD_0 and RDD_2 are the root RDDs created from the sentence-tokenized contents of the respective files. Whenever either of these RDDs is lost they can be recreated by reading in the input files again. Clearly, this assumes that the corpus data is stored on a reliable storage such as HDFS. RDD_1 and RDD_3 are created by applying a filtering operation on each of the two sentence RDDs. The filtered RDDs are then joined creating another RDD before a map and a reduce operation is performed to produce the final word count result which, again, is represented as RDD.

Since the lineage graph keeps track of an RDD's parent and the operation that has been applied to the parent to create the RDD, it is possible to recompute an RDD whenever needed given that the parent RDD is still available. If this is not the case, the recovery procedure needs to follow the lineage graph backwards until an available RDD is discovered and recompute the whole chain of RDDs. For example, when RDD_5 and RDD_4 are lost the recovery procedure follows the lineage back until it encounters RDD_1 and RDD_3 that are still available and recomputes RDD_4 by reapplying the join operation. Once RDD_4 has been recovered, RDD_5 can be reconstructed from it.

This lineage approach can unfold its full potential when used in a pipeline of successive transformations. In such a scenario, systems like Hadoop, for example, need to create checkpoints or write results back to HDFS after each step. This not only makes a successive read from disk necessary to retrieve the input for the next transformation but it also incurs overhead for creating replications. In contrast, due to the independent re-computability of RDD partitions there is no need to immediately write to disk. This is clearly the most important benefit the RDDs provide since it helps to greatly speed up the process [ZCD+12] [spac].

The partitioning strategy of an RDD is configurable by the user. For example, this can be used to apply data locality optimizations in case of a join transformation of two RDDs by ensuring that two partitions getting joined reside on the same node [ZCD+12].

The user can also influence the persistence of RDDs which is useful in case the same RDD is reused across multiple passes to circumvent the need of recomputing it each time. For this purpose, Spark provides several persistence strategies like in-memory or on-disk persistence or mixtures of both [ZCD+12].

Finally, RDDs support operations called *actions* that trigger the computation and materialize the results [spac].

Spark provides specialized RDD implementations for data sources such as HDFS, for example. As shown in Figure 4.4, this type of RDD establishes a 1:1 mapping between HDFS blocks and RDD partitions, thus providing optimal data locality properties when working with datasets stored in HDFS [ZCD+12] [spac].

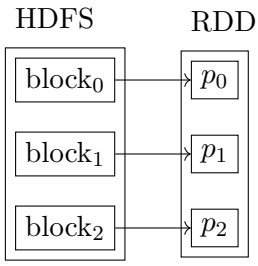


Figure 4.4: RDD implementation optimized for HDFS

4.2 Job scheduler

A *job* in Spark is submitted by the SparkContext to the job scheduler when an action is called on an RDD in the driver program. It consists of a single target RDD, a placeholder for the end result, and the action that should be executed on the parent of the target in order to actually obtain the result. The job scheduler is part of the driver process. It transforms the lineage graph of the target RDD, also called the logical execution plan, into a physical execution plan, also called DAG for directed acyclic graph, consisting of multiple stages, the physical unit of execution in Spark, each operating on partitions of a single RDD. A stage consists of a set of parallel tasks, one task for each partition of the stage's RDD and each task containing a sequence of narrow transformations. It is valid to view a stage as a set of parallel pipelines where each task executes one pipeline. Pipelining has a positive impact on performance since there is no need for storing intermediate results between successive pipelined transformations. The stage boundaries are marked by wide transformations, i.e. shuffle operations. The physical execution plan is therefore the result of introducing stage boundaries at wide transformations in the logical execution plan. Figure 4.5 illustrates a DAG for the filtered word count lineage graph from Figure 4.3 [ZCD+12] [spac] [spaj].

When a job is submitted to the scheduler, it builds the physical execution plan for the job and creates the stages if they do not exist. Whenever possible, the scheduler reuses stages that have been created by other jobs [spag]. It also tracks which RDDs have been cached on user request and avoids recomputing them. The scheduler then launches tasks to compute missing partitions for each stage until the target RDD has been fully computed. This approach of starting at the target RDD guarantees that only such partitions get evaluated that are required for the result [ZCD+12]. Figure 4.6 illustrates this effect using the example of a spark program that joins two RDDs but only returns the first element of the joined RDD. The red lines mark the partitions that are evaluated by Spark to produce this result.

A stage might depend on the output of some preceding stage to be available in order to compute its output partitions. When a task fails it is recreated on a different node as long as the dependencies of the task's stage are still available. In case the output of a required stage has been lost, tasks are launched to re-compute the missing partitions

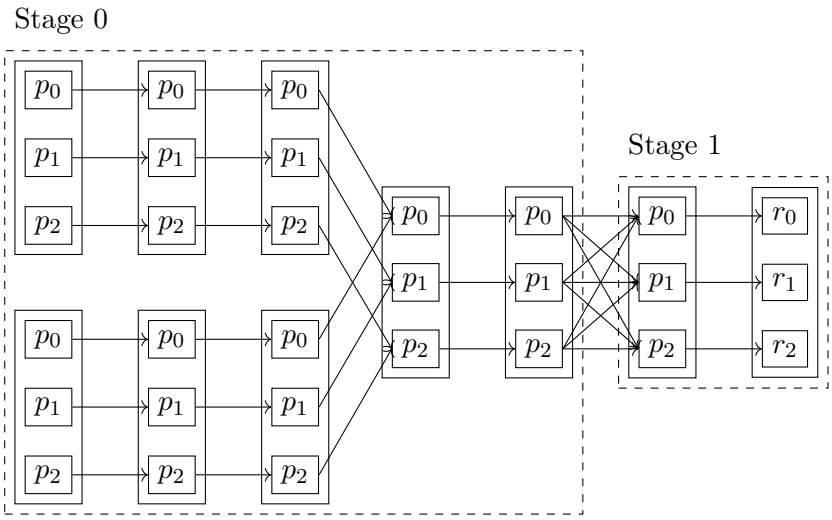


Figure 4.5: DAG for lineage graph from Figure 4.3

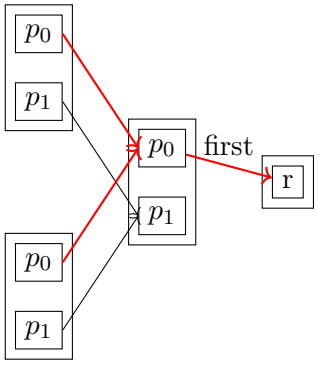


Figure 4.6: Lazy computation of partitions in Spark

[ZCD+12].

4.3 Memory management

Spark provides different storage or persistence modes for RDDs:

- In-memory deserialized
- In-memory serialized
- On-disk

The in-memory deserialized storage is the fastest option because the JVM is able to natively access any requested parts of an object without prior deserialization. Serialized

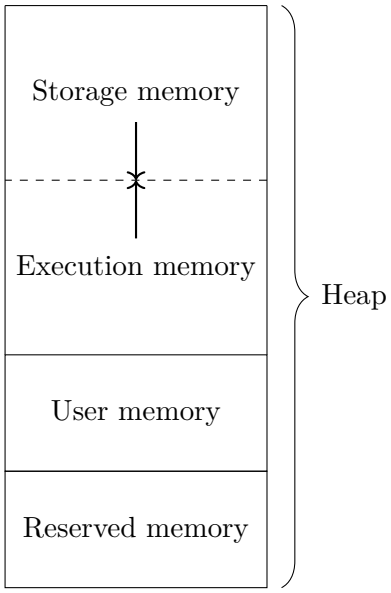


Figure 4.7: Memory model used by Spark's Unified Memory Manager

storage reduces the memory footprint but increases the data access time. Spark is also able to transparently spill data to disk when it runs out of memory [ZCD+12].

For this purpose, Spark employs its own memory management on top of the JVM's heap in order to control the amount of heap space consumed and to decide when to spill data to disk. Originally Spark managed a single static memory area for storage of RDDs but it did not do explicit bookkeeping for temporary memory consumption induced by operations such as joins or aggregations. In practice, this was often the cause for memory exhaustions during memory intensive operations. This shortcoming was resolved by introducing the Unified Memory Manager with Spark 1.6.0 which provides a separate, managed memory area for exactly the purpose of storing temporary data during transformations [ADD+15] [spak]. The memory model, as depicted in Figure 4.7, allows the two managed memory areas to dynamically borrow memory from each other.

Both the execution memory and the storage memory are organized in blocks and grow dynamically towards each other.

The *execution memory* holds objects like shuffle buffers, for example, that are required for the execution of spark tasks. It automatically consumes any free space from the storage memory and can also trigger eviction of blocks in the storage memory in case it needs to grow but no free memory is available. Spilling to disk when low on memory is the execution memory's own responsibility and forceful eviction by other tasks is not allowed because the stored data represents intermediate state that is required by ongoing computations.

The *storage memory* can borrow free memory from the execution memory and is used to

store RDD data. Eviction is performed according to a least recently used (LRU) policy [ZCD+12].

The *user memory* can be used by the computations supplied by the driver program to store data. It is not managed by Spark and the user application is responsible for not violating the memory bounds.

A fixed size *reserved memory* space is budgeted for the purpose of running the Spark worker process along with the task executors which also requires some memory to be available.

4.4 Shuffle

The shuffle operation is performed at stage boundaries and Spark supports two flavors of it:

- Hash shuffle
- Sort shuffle

For the sake of simplicity, the following explanations of the shuffle algorithms rely on the MapReduce naming convention for the two sides of the shuffle, namely the mapper and the reducer. In reality, Spark performs shuffle for a more diverse set of transformations than just MapReduce.

The hash shuffle is the naive way of performing the shuffle. Each mapper task creates one file for each reducer where it writes the corresponding records to. The algorithm works well for small reducer sizes but in practice, this method is problematic because of the large number of files created. Spark provides an optimization for this approach by pooling output files and reusing them across mapper tasks but this only eases symptoms [spai].

Due to the shortcomings of the hash shuffle, Spark 1.2.0 introduced the sort shuffle which is similar to the algorithm used by Hadoop MapReduce. Records are written *sorted* by reducer id to a single output file per mapper task. By maintaining an index of file offsets for each reducer id, the appropriate chunk can be read quickly when a reducer queries it. Because hashing is generally faster than sorting, there is a threshold on the number of reducers below which the records are hashed to separate files and then merged to a single file [spai].

4.5 Air quality threshold monitoring with Spark

This section describes an implementation of the solution to the air quality monitoring problem statement from Section 2.4 using Apache Spark.

Listing 4.1 shows the really compact implementation of the report generation job using Spark. Line 3 starts by reading in the raw air quality data in textual form which is then parsed to a tuple format by applying the `parse` method shown in Listing 4.2. This method also transforms the timestamps to distinct numbers per day that become part of the tuple key. The subsequent reduction operation scheduled in line 5 computes the maximum tuple values per indicator, region and day. Another map operation is then used to perform the threshold comparison with the maximum values per day. Each value is either mapped to 1 or 0 depending on whether it violates the threshold or not. Also the tuple key is changed in this step by dropping the day number. Hence, only the indicator and the region remain as tuple key components. Finally, `reduceByKey` sums up the violations per indicator and region. The end results are collected via the `collectAsMap` method and printed to the console.

```
1  JavaSparkContext sc = new JavaSparkContext(conf);
2
3  Map<Tuple2<Indicators, Regions>, Integer> counts = sc.textFile("data/tuples
       ↪ ")
4          .mapToPair(SparkRunner::parse)
5          .reduceByKey((t1, t2) -> t1.getValue() > t2.getValue() ? t1 : t2)
6          .mapToPair(t -> Tuple2.apply(
7                  Tuple2.apply(t._1()._1(), t._1()._2()),
8                  t._2().getValue() > t._2().getIndicator().getThreshold() ?
                       ↪ 1 : 0
9          ))
10         .reduceByKey((v1, v2) -> v1 + v2)
11         .collectAsMap();
12
13 System.out.println(counts);
```

Listing 4.1: Air quality violations report with Spark

```
1  private static Tuple2<Tuple3<Indicators, Regions, Integer>, AirQualityTuple
       ↪ > parse(String s) {
2      String[] parts = s.split(";");
3      Indicators indicator = Indicators.valueOf(parts[0]);
4      Regions region = Regions.valueOf(parts[1]);
5      Double value = Double.valueOf(parts[2]);
6      ZonedDateTime ts = ZonedDateTime.parse(parts[3]);
7
8      return Tuple2.apply(
9              Tuple3.apply(
10                     indicator,
11                     region,
12                     ts.query(SparkRunner::queryDayOfYearGroup)
13             ), new AirQualityTuple(indicator, region, value, ts)
14     );
15 }
```

Listing 4.2: Air quality tuple parsing

Part II

Graph Processing

CHAPTER 5

The Pregel programming model

While the MapReduce programming model is a good fit for a variety of applications, large scale graph processing is one domain that is inherently a bad fit for MapReduce style processing since graph algorithms are mostly iterative in nature. As described in Section 2.2, MapReduce iterations are possible in principle and it has been demonstrated [Coh09] that graph algorithms can indeed be implemented using iterations over MapReduce jobs. However, this approach is highly inefficient when used with systems such as Hadoop because the graph data needs to be stored and reloaded at each iteration, potentially in different cluster locations. Projects like Surfer [BCR06] and GBASE [KTS+11] attempted to optimize MapReduce for graph processing but with limited success. Apart from the downside with respect to performance, implementing graph algorithms on top of the MapReduce model is not intuitive because it usually comes down to manipulating adjacency matrices.

In 2010, Google presented Pregel - their solution to large-scale graph processing [MAB+10]. Pregel stands synonymous for an abstract programming model as well as Google's C++ implementation of a runtime environment for Pregel programs. This thesis does not cover the latter and focuses on describing the theoretical programming model and open source implementations of it.

Pregel is a vertex-centric programming model and allows the user to "think like a vertex" by creating programs or functions that define the behavior of a single vertex. The execution of a Pregel program is organized in a sequence of *supersteps*, each concluding with a synchronization barrier. This is inspired by the bulk synchronous parallel (BSP) model of Valiant [Val90]. During each superstep S the runtime executes the user defined program on all active vertices of a graph. A vertex can read messages received from other vertices during superstep $S-1$, send messages to other vertices that will be received in superstep $S+1$, modify its state and alter the topology of the graph by deleting and creating edges and vertices. Moreover, a vertex can vote to halt whereby it transitions into the *inactive* state causing the runtime to stop invoking the vertex in upcoming

supersteps. However, when messages are received by a vertex it is invoked and put back into the active state by the runtime. A Pregel execution terminates when all vertices are simultaneously inactive and no more messages are pending. The output of a Pregel program is the set of values that is returned by each vertex which might form a transformed graph or some aggregation, for example [MAB+10].

The big conceptual advantage of Pregel over MapReduce is that graph data does not need to be moved across the cluster. Each vertex along with its state resides at the same node for the entire execution of a Pregel program and the invocation of a vertex function is co-located. Hence, network transfer is only required for message passing.

A vertex in Pregel holds a unique identifier, the set of outgoing edges and modifiable user defined properties [MAB+10].

Example 4

For this example, assume that each vertex is assigned an integer number and that the vertex identifiers are ordered alphabetically. Moreover, assume that each vertex is passed the minimum vertex id in the graph. This information can be precomputed with a separate Pregel algorithm, for example. Given a strongly connected graph, an algorithm is presented for computing the sum of the vertex numbers among all connected vertices. Listing 5.1 shows an implementation with pseudo code.

```
sum (Vertex v, Integer minimumVertexId, List msgs) {
        // Compute message sum and update stored value
        for (Integer msg : msgs) {
                v.number += msg
        }
        // Update counter for total messages received
        v.totalMessagesReceived += msgs.size

        if (  v.number > 0 AND
              v.totalMessagesReceived > 0 AND
              v.id != minimumVertexId) {

                // Send stored value to adjacent vertex
                // with minimum id
                sendMessage(v.minNeighbor, v.number)
                v.number = 0
                voteToHalt()
        } else if (msgs.isEmpty) {
                voteToHalt()
        }
}
```

Listing 5.1: Sum algorithm

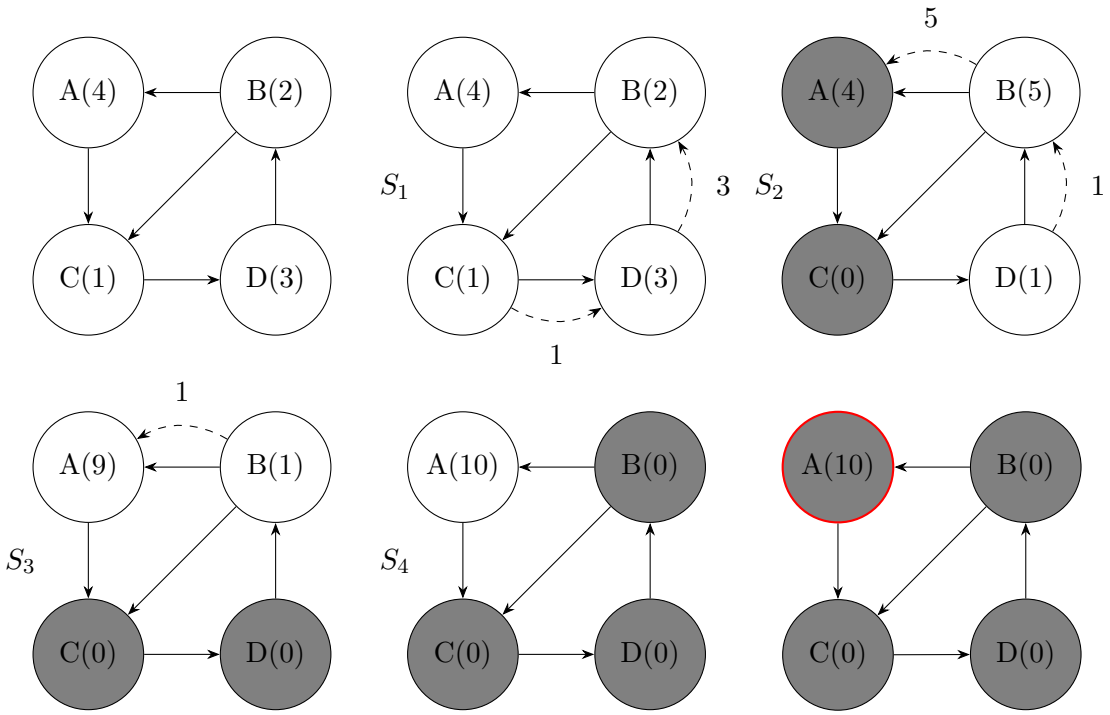


Figure 5.1: Pregel sum algorithm for strongly connected graphs

A vertex V receives messages from directly connected vertices containing partially aggregated numbers s_i. During each superstep, a vertex V sends one message whenever all of the following conditions are met:

1. The stored number $V.num$ is greater than 0
2. V has received at least one message from all its direct predecessors V' with $V'.id > V.id$
3. $V.id$ is not the minimum id in the graph. Each vertex can easily check this condition independently because it has the minimum vertex id stored by assumption.

Condition 1 implies that vertices without any predecessors V' with $V'.id > V.id$ send their values immediately during the first superstep. The message's target vertex is the direct successor of V with minimum id. After sending a message, V sets its stored number to 0 and votes to halt. V also votes to halt when it has not received any messages during the last superstep. When the algorithm terminates, the sum resides at the vertex with minimum id.

Figure 5.1 illustrates the algorithm. It shows the initial graph, the intermediate vertex states at the end of each superstep, the messages sent during each superstep and the final state of the graph after the algorithm has terminated.

Gray nodes denote inactive vertices and dashed edges indicate messages. The course of action is briefly explained in the following.

During S_1 vertex C sends message $M_1(1)$ to its only successor, vertex D, because it does not have any predecessors with higher id. It then sets $C.num = 0$ and votes to halt. For the same reasons vertex D sends message $M_2(3)$ to vertex B, sets $D.num = 0$ and votes to halt. Vertex A and B stay inactive because condition 1 is not satisfied for them.

In superstep S_2 vertex D is reactivated because it $M_1(1)$ received from vertex C. It updates $D.num = 0 + 1$, sends another message $M_3(1)$ to vertex B, sets $D.num = 0$ and votes to halt. At the same time, vertex B receives $M_2(3)$ and updates $B.num = 2 + 3$. In addition, condition 1 is now satisfied for vertex B so it sends $M_4(5)$ to vertex A (because $A < C$), sets $B.num = 0$ and votes to halt. Vertex A also votes to halt because it has not received any messages during S_1.

In superstep S_3 vertex A receives $M_4(5)$ and updates $A.num = 4+5$. However, vertex A does not send any messages because condition 3 is not satisfied. Vertex B is reactivated because it receives $M_3(1)$ and it updates $B.num = 0+1$. After that it immediately sends out $M_5(1)$ to vertex A and votes to halt.

In superstep S_4 vertex A receives $M_5(1)$ and updates its value to $A.num = 9 + 1$.

In superstep S_5 vertex A votes to halt because it has not received any new messages in S_4. At this point, all vertices are simultaneously inactive and the algorithm terminates.

Similar to MapReduce, Pregel supports *combiners* to reduce the number of messages that need to be sent over the network. The user can define an associate and commutative combiner function to be applied by the runtime to multiple messages destined for the same target node in order to form a single, aggregated message instead.

For purposes like global coordination, monitoring etc., Pregel supports *aggregators* that allow each vertex to provide a value for each superstep. The supplied values are aggregated via some reduction operator and made accessible to the vertices in the successive superstep.

Fault tolerance in Pregel can be achieved by creating checkpoints at superstep boundaries. After a defined number of supersteps, each vertex saves a copy of its state to distributed storage. In addition each vertex also logs its outgoing messages. When a cluster node fails during superstep S, graph partitions that were located on the failed node are recreated on a healthy node using the latest snapshot S'. The supersteps in between S' and S need to be redone for the lost partition which also requires replaying the messages to lost vertices that were logged by healthy vertices.

In Pregel, a graph is partitioned across cluster nodes via an edge-cut, so each vertex is uniquely assigned to a cluster node. The interaction of a vertex with other vertices on

the same node is cheaper than with vertices on remote nodes because no intermediate network access is required. However, Pregel does not expose partitioning information to the user via its API which leaves no room for locality optimizations targeting the node interactions in Pregel algorithms. For this reason, researchers have suggested a divergence from a vertex centric to a more coarse-grained graph centric [TBC+13], sub-graph centric [SKW+14] or block centric [YCLN14] model. Such an approach makes partitioning information accessible to a user program and is able to yield a vast reduction in network communication and execution speedup for algorithms that utilize this information.

Likewise, the choice of the algorithm for partitioning a graph has a big performance impact. While Pregel allows the communication between vertices regardless of the existence of a connecting edge, most practical algorithms evaluate graphs or edges in the context of their neighborhood and send messages along graph edges only. Thus, the number of edges between graph partitions directly correlates with the communication overhead. Minimizing the number of non-local edges, i.e. edges between vertices that lie on different nodes, is therefore important for the performance of Pregel algorithms or distributed graph algorithms in general [XGFS13].

5.1 Complexity & costs in Pregel algorithms

Since Pregel closely follows the BSP model, the BSP cost model is adoptable for the purpose of estimating the costs of Pregel algorithms.

A Pregel execution consists of a sequence of supersteps, hence the total execution cost is simply the sum of the costs of each superstep. The costs for an individual superstep are composed of

- the computational cost of the longest running vertex function
- the communication cost of message passing and other global data exchange
- the synchronization cost at the end of the superstep

The execution cost of a single superstep is therefore expressed by

$$v + mg + l$$

where v is the vertex function cost, m is the number of messages sent, g is the communication cost per message and l is the synchronization cost per superstep.

Consequently, the cost of a Pregel execution is

$$\sum_{s=0}^{S} v_s + g \sum_{s=0}^{S} m_s + Sl$$

where S is the number of supersteps [HCB96].

Yan et al [YCX+14] define characteristics of efficient or balanced, practical Pregel algorithms (BPPA), as they call it. A BPPA needs to fulfill the following requirements:

- Linear space usage in $O(d(v))$ per vertex v where $d(v)$ is the degree of v
- Linear computation cost in $O(d(v))$
- Linear communication cost in $O(d(v))$
- Number of supersteps in $O(\log n)$

The balancing property is lost when the above per-vertex requirements are relaxed such that merely *overall* linear space usage, computation and communication usage remains. This class of algorithms is referred to as PPA.

5.2 Running example: Improving the availability of AirQuality Inc.'s sensor network

The following problem is used as illustrative real world example in the course of discussing Apache Giraph and Apache Spark GraphX.

As mentioned in Section 1.2, AirQuality Inc. provides a peer-to-peer based secondary communication channel to connect sensors in rural areas without dedicated WWAN stations. However, the resulting topologies tend to be weakly connected and thus fragile. The failure of single sensor devices can cause many other sensors to be disconnected because the relay chain is interrupted. This problem is illustrated in Figure 5.2. When sensor 3 fails, sensors 4 and 5 lose connection to the back-end system.

Therefore it is important for the business of AirQuality Inc. to continuously expand the coverage of the WWAN network in order to reduce the single point of failures. For this reason, the infrastructure planners at AirQuality Inc. continuously identify eligible placement points for new WWAN stations. In order to choose the best locations for new stations from the candidate spots a special algorithm is required that analyzes the current sensor network topology and performs placement optimization. The number of WWAN stations that can be selected is restricted by a yearly budget. Facilities that query the current topology from the field sensors and store the resulting toplogy graph in the data-center are already in place. Since the topology is too big to be processed by a single machine, the use of a big data graph processing framework is required to distribute the computations across multiple nodes. The only additional information that each vertex carries is its node type which is one of

- SENSOR - an existing sensor node

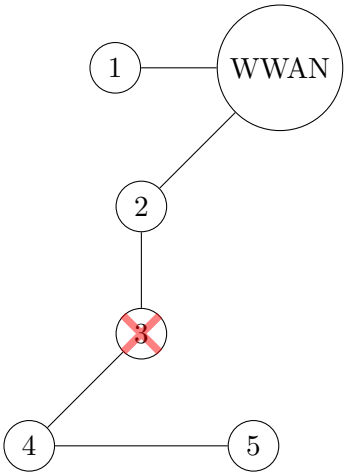


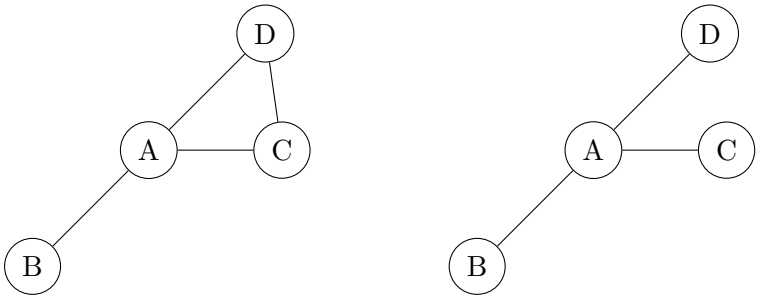
Figure 5.2: Fragility of sensor connectivity in long peer-to-peer chains

- STATION - an existing station node
- ELIGIBLE_NEW_STATION - a potential new station node
- NEW_STATION - a new station node that has been selected by the optimization algorithm

The proposed algorithm performs brute-force optimization against some metric designed to capture a topology's quality with respect to sensor connectivity. The algorithm's input consists of the current network topology along with the eligible placement points and the number of stations that the budget allows to be created. First, the algorithm generates the set of permissible permutations of new station placements. Each permutation is then used to update the current topology by connecting existing field sensors to every new station when they lie within the station's reception range. The resulting topology is rated against the quality metric. Finally, the algorithm outputs the topology with the highest rating. Note that in practice, the use of heuristic optimization techniques like simulated annealing would be desirable for this task in order to reduce the number of solutions that need to be evaluated.

For quantifying the quality of a topology a variation of the graph *compactness* metric [BRS92] is used. The compactness $C_p \in [0; 1]$ of a graph is a global metric that was originally used to rate the complexity of hypertext in terms of connectivity. A compactness of 0 indicates a fully disconnected graph whereas a compactness of 1 indicates a fully connected graph. Compactness metric is defined as follows:

$$C_p = \frac{\text{Max} - \sum_i \sum_j C_{ij}}{\text{Max} - \text{Min}}$$



(a) Graph G_1 (b) Graph G_2

Figure 5.3: Example graphs for illustrating the compactness metric.

In the above formula, C_{ij} denotes the shortest path between node i and node j in the graph. Max and Min represent an upper and a lower bound to the sum of the length of the shortest paths between any two nodes, respectively. Intuitively, the denominator represents the maximum gap between a maximally weakly connected and a fully connected graph. On the other hand, the counter represents the gap between the same maximally weakly connected graph and the graph under consideration. Hence, the more connected a graph is the larger the counter becomes which results in a compactness closer to 1. Example 5 illustrates the compactness metric.

Example 5 Consider the two example graphs shown in Figure 5.3. It is easy to see that G_1 is stronger connected than G_2.

For calculating the compactness of graphs G_1 and G_2 it is first necessary to identify upper and lower bounds Max and Min for the sum of shortest path lengths. In general, a lower bound can be given assuming a fully connected graph where the shortest path length C_{ij} is 1 for all vertices i and j with $i \neq j$. Since there are up to $n * (n-1)$ connected vertex pairs in a graph, a possible lower bound is given by:

$$Min = n \times (n-1)$$

A safe upper bound can be formulated by introducing a constant k holding the length of the "longest" shortest path between any two vertices. The final upper bound is then obtained by multiplying with the number of vertex pairs analogous to the lower bound.

$$k = max_{\forall i,j\, i \neq j}(C_{ij})$$

$$Max = n \times (n-1) \times k$$

	A	B	C	D
A	0	1	1	1
B	1	0	1	2
C	1	1	0	1
D	1	2	1	0

Table 5.1: Shortest paths G_1

	A	B	C	D
A	0	1	1	1
B	1	0	1	2
C	1	1	0	2
D	1	2	2	0

Table 5.2: Shortest paths G_2

Apart from the upper and lower bounds it is required to calculate the shortest paths between all node pairs in the graph as shown in Tables 5.1 and 5.2.

The concrete upper and lower bounds can now be calculated.

$$Min = 4 \times (4 - 1) = 12$$
$$Max = 4 \times (4 - 1) \times 2 = 24$$

Given the above results it is now possible to calculate the compactness C_{G_1} and C_{G_2} of the given graphs.

$$C_{G_1} = \frac{24 - 14}{24 - 12} \approx 0,83$$
$$C_{G_2} = \frac{24 - 16}{24 - 12} \approx 0,66$$

As expected, C_{G_1} is higher than C_{G_2} because G_1 is more connected than G_2.

Note that the upper bound chosen in this example is a theoretical one because it is impossible to construct a graph where the sum of the shortest path lengths equals this bound. As a result, the compactness value cannot fall below a certain threshold which may or may not pose a problem depending on the application.

While the original compactness metric considers the pair-wise connectivity of all nodes in the graph, the interest for the WWAN station placement optimization solely lies on the connectivity between each sensor node and its closest WWAN station node. Thus, a variation of the compactness metric is defined as follows:

$$C_{\text{var}} = \frac{\text{Max}_{\text{var}} - \sum_{i \in N_s} \min_{j \in N_w} C_{ij}}{\text{Max}_{\text{var}} - \text{Min}_{\text{var}}}$$

$$\text{Max}_{\text{var}} = |N_s| \times \max_{i \in N_s} \min_{j \in N_w} C_{ij}$$

$$\text{Min}_{\text{var}} = |N_s|$$

In the above formula, N_s and N_w represent the set of sensor nodes and the set of WWAN station nodes, respectively. Since the new metric only considers the single minimum shortest path from each sensor node to some station node, the definitions of the upper and lower bounds are revised accordingly as shown.

It is easy to see that the core of the whole optimization algorithm is the shortest path computation required for the compactness metric. This task requires knowledge about the structure of the whole graph which is assumed to be too large to be processed by a single machine. Thus, the shortest path computation is performed using a big data graph processing framework. A distributed shortest path algorithm that computes the shortest path from every graph node to a set of target nodes can be expressed very concisely using Pregel as explained in the following.

Each vertex v maintains a list of key-value pairs (u, l) that contain the length l of the current shortest path from v to u. Before the first round of the algorithm, the key-value pair lists are initialized by inserting $(v, 0)$ for all $v \in N_T$ where N_T is the set of target nodes. During the superstep iterations, key-value pairs are propagated along the reverse edge directions starting at the target nodes. When a vertex receives a key-value pair list it merges the list with its own state. Before passing on a list of key-value pairs, the shortest path lengths in the list are incremented by 1 and the sending node performs a preliminary merge of the the list that is to be sent with the state of the receiving node to check if the transmission will have any effect on the receiving node's state. In case no changes are detected, the message is not sent. As a result, no more messages are sent when all vertex states have converged to the true shortest path lengths and the algorithm terminates. Figure 5.4 illustrates an execution of this algorithm.

The single target node E is marked as green. Before the first superstep, the vertex states are initialized. Because E is a target node, its key-value pair list is initialized with E $\rightarrow$ 0. In superstep S_1 E increments its state and sends it to B and D which update their state with $E \rightarrow 1$ based on the information received from E. Next, B and D send their incremented state to A and C in superstep S_2. Note that node E does not send any message at this point because it recognizes that the states in B and D would not change anyway. At this point, the state at all nodes has converged to the real shortest path lengths. Thus, no more messages are sent and the algorithm terminates.

Section 5.1 discusses the notion of a BPPA and we now evaluate the shortest path algorithm presented above against the characteristics of a BPPA:

- Linear space usage in $O(d(v))$ per vertex v where $d(v)$ is the degree of v
  The upper bound on the size of the key-value pair list that makes up the vertex state in the shortest path algorithm is given by the number of target vertices which is a constant for each execution of the algorithm. Hence, this property can be considered satisfied.

- Linear computation cost in $O(d(v))$
  The computation performed by each vertex consists of the merging of the messages

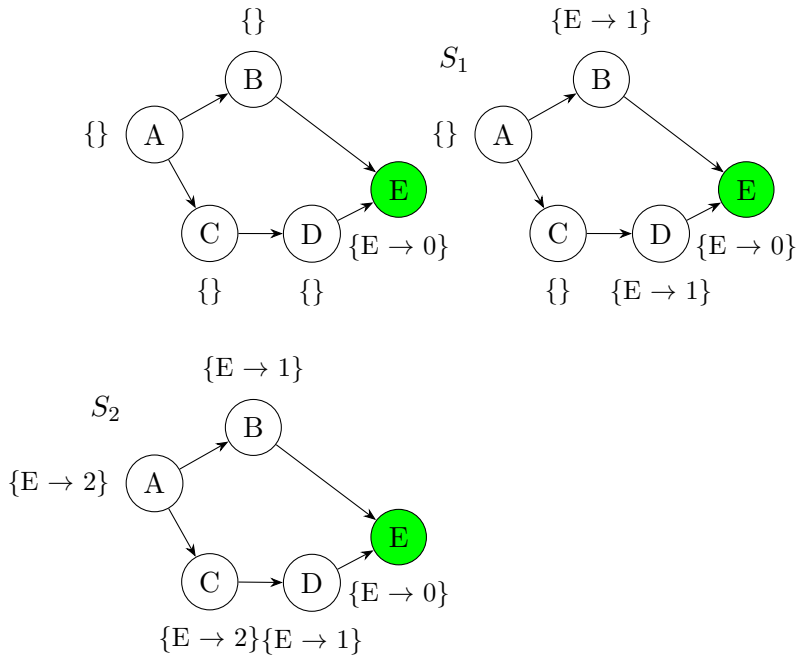


Figure 5.4: Pregel shortest path algorithm

received and the sending of new messages. The computational complexity of the latter task clearly is in $O(d_{\text{in}}(v))$ as argued in the next point. The number of messages to merge is in $O(d_{\text{out}}(v))$ and the merge operation itself only depends on the size of the key-value pair list which is a constant as argued in point 1. Hence, this property is satisfied.

- Linear communication cost in $O(d(v))$
  During each superstep each vertex sends at most 1 message along each incoming edge. Moreover, the message size is constant and depending on $|N_T|$, hence the communication cost per vertex is in $O(d_{\text{in}}(v))$ which satisfies this property.

- Number of supersteps in $O(\log n)$
  Consider the extreme case when the graph takes the shape of a linked list with the target node at the very end of the list. In this scenario, the algorithm requires $|N| - 1$ supersteps to terminate. Hence, this property is not satisfied.

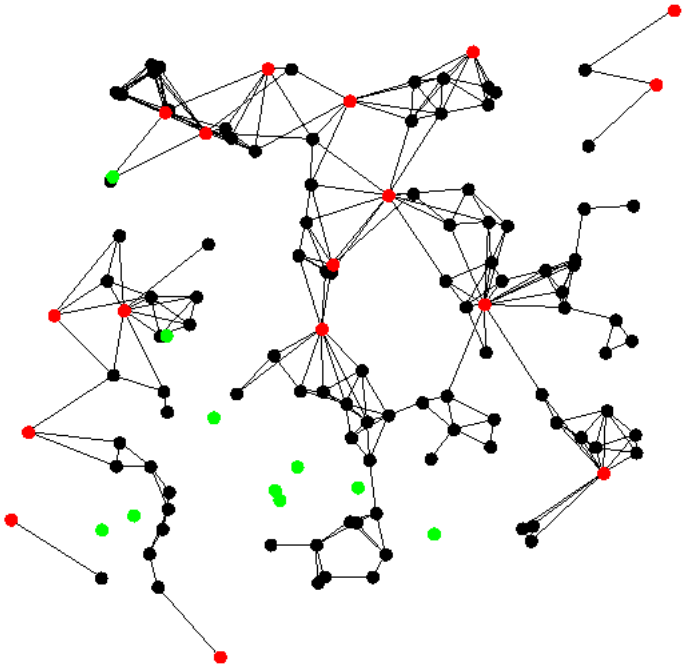
Except for the number of supersteps, the presented shortest path algorithm fulfills all characteristics of a BPPA.

Figure 5.5 shows an example input topology and the optimized topology that is output by the optimization algorithm:

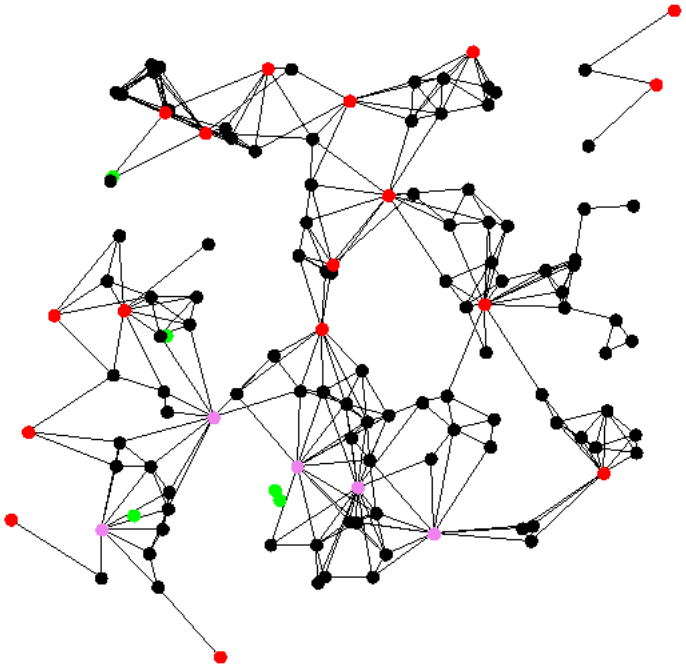
- Sensor nodes are black

- Existing WWAN stations are red
- Eligible positions for new WWAN stations are green
- Selected new WWAN stations are violet

The implementations of the optimization algorithm described above are detailed in Sections 6.3 and 7.5 for Apache Giraph and Apache Spark GraphX, respectively.



(a) Input topology



(b) Optimized topology

Figure 5.5: Example input topology and the corresponding optimized output topology as produced by the optimization algorithm.

CHAPTER 6

Apache Giraph

Initially developed at Yahoo!, Apache Giraph [1] was designed to mimic Google's Pregel but has received numerous extensions to this model over time. It is closely related to Apache Hadoop since it can utilize YARN or Hadoop classic MapReduce to run Giraph applications on a cluster and it supports many I/O integrations with systems related to Hadoop such as HDFS or Hive [MSL15].

The execution of a Giraph application requires 3 different types of actors [MSL15]:

- Coordinators
- Masters
- Workers

Coordinators are simply an ensemble of Apache ZooKeeper nodes that merely serve as a central point for storing various types of coordination data but they do not run any logic specific to Giraph [MSL15].

Giraph employs a configurable amount of master nodes to ensure high availability. At most one master can be active at a time and the active master is determined during a bidding process at startup using the coordinator service. The role of a master node is to coordinate execution of a Giraph application. It starts by creating the input splits by establishing a mapping from worker nodes to input files that determines which files containing portions of the graph are to be loaded by each worker. The mapping is stored in the coordination service to guarantee its availability even if the active master dies. After the input split has been created, the master node enters the superstep loop and coordinates the execution of supersteps on the workers. This involves the following tasks [MSL15]:

[1] `giraph.apache.org`

- Worker nodes periodically report their health status to the coordinator service. The master node listens to these reports and initiates corrective actions in case of worker failures.
- In each superstep, the master node may decide to re-balance the graph data partitioning based on considerations like worker load and resource availability, for example. When a partition is assigned to a worker that does not hold the corresponding graph data, the data is copied from the source worker.
- The master node fetches aggregated values from the aggregator owners and makes them available to the master computation. Before the next superstep, the master redistributes the aggregated values back to the worker owners (see 6.1).
- The enforcement of the synchronization barrier at the end of a superstep is also carried out by the master service.
- When the checkpoint condition is met, the master waits for all workers to checkpoint their state and then finalizes the checkpointing process.
- It is another responsibility of the master to invoke the so called master computation which is a custom function supplied by the user to be centrally executed on the master.

Worker nodes are used to run the user supplied vertex functions for all vertices in the assigned graph partitions. The lifecycle of a worker consists of two phases. During the first phase, each worker loads the input files that were assigned to it by the master when creating the input splits. Once every worker has loaded its data the master signals to enter the second phase which is the superstep loop. During a superstep a worker first handles any API requests that were received during the last superstep. This may involve graph topology changes, for example. After that, each worker handles any partition assignment changes ordered by the master and fetch the latest aggregator state from the master before entering the execution of the vertex functions [MSL15].

Giraph only comes with primitive hash based and id range based graph partitioning algorithms that construct edge-cuts. Depending on the application it might thus be necessary to implement appropriate strategies manually.

Example 6 Giraph has been used at Facebook to answer queries on Facebook's social graph. Frequently, such queries involve an aggregation touching all friends of a user, i.e. a user's neighborhood. If vertices were assigned to partitions randomly using hash based or id range based algorithms as provided by Giraph, such a query would hit almost all partitions in a cluster inducing high communication overhead and increased latency [gir].

At Facebook, this issue was solved by implementing a custom partitioning strategy that aims to create uniform or balanced graph partitions [gir] [AR04].

Given an undirected graph $G = (V, E)$ and a natural number k an algorithm for balanced graph partitioning splits the vertex set V into k subsets of equal size while minimizing the weight of edges in between the subsets. The balancing constraint is important to antagonize the extreme case of all vertices being assigned to the same subset which would result in zero non-local edges but would eliminate any parallelism [gir]. Also, creating balanced partitions maximizes the likelihood that each Giraph worker is able to keep its graph partitions in memory.

Since the problem of balanced graph partitioning is NP-complete, it is infeasible to optimally solve this problem for real world graphs [GJ02]. Apart from the computational complexity, large-scale graphs that require distributed processing certainly do not fit in memory on a single machine which would further complicate the search for an optimal solution [gir].

Facebook's solution for this difficult situation was to create the graph partitioning using a separate Giraph application that applies probabilistic hill climbing to iteratively reduce the number of non-local edges while maintaining the balancing of the solution. The idea is illustrated in Figure 6.1. The algorithm initially starts with a randomly balanced partitioning established by one of Giraph's built-in primitive algorithms [gir].

In superstep $2i$, all vertices communicate their current partition assignment to their neighbors so each vertex obtains a view of the partition assignment of its neighborhood. When a vertex discovers that many of its neighbors belong to a different partition than the vertex itself, it can choose to relocate to one of the neighboring partitions. To prevent the algorithm from getting stuck, the choice of the target partition is constructed probabilistically with a bias towards partitions with a higher number of edges to the vertex. Vertices that strive for a relocation send a signal to the master which decides needs to coordinate and restrict the graph-wide movements of vertices between partitions in order to maintain the balance. Depending on the signaled movement intentions, the master computes a movement probability for each pair of partitions and distributes these values back to the vertices [gir].

In superstep $2i + 1$, each vertex that signaled for relocation in superstep $2i$ tosses a coin biased towards the probability received by the master and implements the desired movement only on favorable outcome [gir].

The Facebook engineers were also able to find a better starting configuration than a randomly balanced assignment based on special features of the social graph. That is, users tend to be friends with other geographically close users. Hence, initializing the partitions based on geographical proximity yielded a starting point with much less non-local edges compared to a random assignment [gir].

Example 6 illustrates that choosing an appropriate method for graph partitioning is

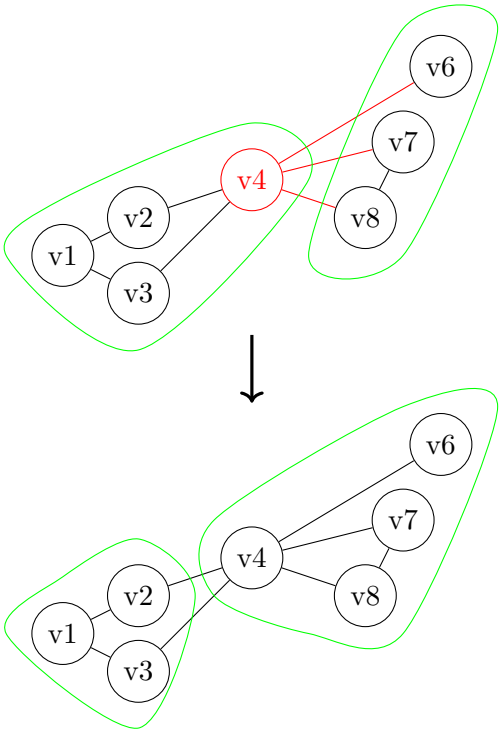


Figure 6.1: A graph partitioning heuristic that iteratively moves vertices between partitions to reduce non-local edges

vital for the performance of distributed graph algorithms. Moreover, application specific features can sometimes be exploited to find a better partitioning in less time.

The remainder of this chapter describes the non-trivial implementation of aggregators in Giraph and discusses fault tolerance.

6.1 Sharded aggregators

Aggregators in Giraph are realized as sharded aggregators, i.e. they are not exclusively managed by the master to prevent overloading it. Instead, the master randomly assigns each aggregator to a worker making it the aggregator's owner. Each worker performs the aggregation of values supplied by vertices within its own partitions and sends the result to the aggregator owner which performs the final aggregation of all the partial results received from the workers before sending the ultimate result to the master [CEK+15]. Figure 6.2 illustrates this process.

The reverse direction works analogously. After invoking the master computation, the master sends the aggregation values back to the assigned worker owners which, in turn, distribute them to the remaining workers [CEK+15].

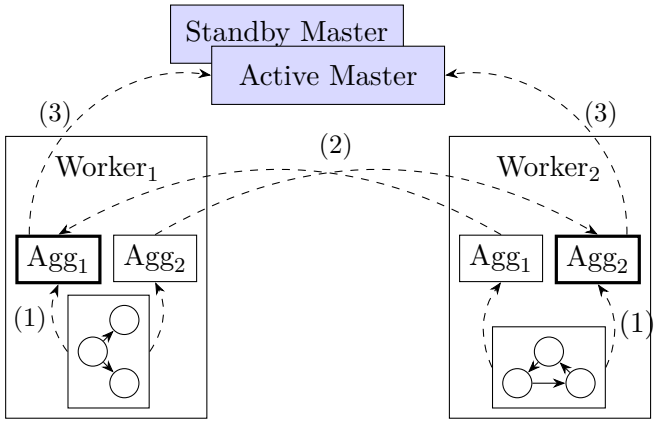


Figure 6.2: Sharded aggregators in Apache Giraph

6.2 Fault tolerance

This section discusses the different failure modes that Giraph is exposed to an how it recovers from them.

When a master fails, the coordination service will trigger one of the standby master nodes to immediately take over. A master node is stateless because all the relevant state is either stored in the coordinators or in HDFS. Hence, there is no internal state that needs to be recovered by a standby node [MSL15].

In contrast to a master node, a worker node holds the current state of its graph partitions as well as the state from owned aggregators. Failures during the initial loading phase of a worker are not recovered. In this case the master simply aborts the entire application. To be able to recover from worker failures during the superstep loop, Giraph periodically creates checkpoints at superstep boundaries via a cooperative effort of master and worker nodes and stores them in HDFS. When a worker fails during the superstep loop, the coordination service notifies the master which marks the current superstep as failed and restarts the superstep loop from the last checkpoint for the entire application. This means that all healthy worker nodes load the state from the last checkpoint and continue the execution from there [MSL15].

The coordinators represent an ensemble of ZooKeeper instances. Such a configuration provides high availability but it is out of the scope for this thesis to elaborate on the design of ZooKeeper in more detail.

Giraph itself does not have any mechanisms in place to recover from disk failures. Instead it trusts on the reliability of the underlying storage layer like HDFS, for example [MSL15].

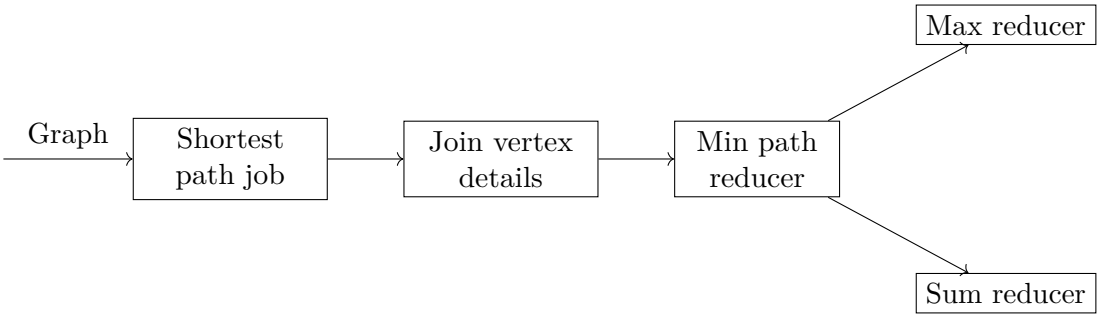


Figure 6.3: Structure of computation in Giraph based topology optimizer

6.3 AirQuality Inc. WWAN station placement with Apache Giraph

This section describes an implementation of the solution to the air quality WWAN station placement problem statement from Section 5.2 using Apache Giraph. It relies on the notations introduced in Section 5.2.

The implementation consists of 5 different components as illustrated in Figure 6.3.

First, a Giraph job computes the shortest paths C_{ij} to the given set of station nodes with $i \in V_s \cup V_w$ and $j \in V_w$. The output produced by the shortest path step only consists of vertex IDs and path lengths, i.e. all vertex detail information is stripped.

Since information like the node type (see Section 5.2) is needed to carry out the subsequent processing steps, a join is performed that populates the shortest path dataset with the required vertex information. This join is implemented with two MapReduce jobs, one joining the source vertex details and the other one joining the target vertex details for each path.

The output produced by the shortest path step may contain paths between two station nodes. Such routes are not of interest and need to be filtered out. Moreover, this step may produce multiple paths from a single sensor vertex to multiple station vertices that need to be reduced to a single path to the closest station vertex. These tasks are carried out by a subsequent MapReduce job.

To compute the compactness of the input graph, the maximum shortest path among all sensor/station pairs as well as the sum of the shortest path lengths is required for computing the upper and lower bounds for the compactness metric as detailed in Section 5.2. Thus, two subsequent MapReduce jobs consume the output of the filtering and minimization step and carry out the respective global aggregations.

The following paragraphs discuss the client side application that invokes the MapReduce/Giraph jobs.

The essential part of the client implementation is located in the `GiraphCompactnessOptimizer` ↪ class which contains the `maximizeCompactness` method shown in Listing 6.1.

```
1  public Graph maximizeCompactness(Graph graph, Iterator<Graph>
       ↪ solutionIterator) {
2      try {
3          this.fs = FileSystem.get(hadoopConf);
4
5          Set<Vertex> baseVertices = new HashSet<>();
6          for (Vertex v : graph.getVertices()) {
7              if (EnumSet.of(Vertex.NodeType.SENSOR, Vertex.NodeType.STATION)
                   ↪ .contains(v.getNodeType())) {
8                  baseVertices.add(v);
9              }
10         }
11         Set<Edge> baseEdges = graph.getEdges();
12
13         Graph bestExtension = null;
14         double maxCompactness = 0.0;
15         while (solutionIterator.hasNext()) {
16             Graph extension = solutionIterator.next();
17
18             double compactness = compactness(extendBaseGraph(baseVertices,
                   ↪ baseEdges, extension));
19             if (compactness > maxCompactness) {
20                 maxCompactness = compactness;
21                 bestExtension = extension;
22             }
23         }
24
25         this.fs.close();
26         return bestExtension;
27     } catch (IOException e) {
28         throw new RuntimeException(e);
29     }
30 }
```

Listing 6.1: Implementation of the MapReduce/Giraph client application

The method takes the full input graph and a solution iterator that produces extension graphs representing subsets of eligible station vertices. On lines 5 - 11 the full input graph is reduced to a base graph by removing all nodes of types ELIGIBLE_NEW_STATION and NEW_STATION and any attached edges. The loop on lines 15 - 23 represents the main optimization loop. In each round it uses the solution iterator to produce a new extension graph which is the applied to the base graph using the `extendBaseGraph` ↪ method to form a new solution graph which is then passed to the `compactness` method computing a solution rating on line 18. After the method returns, the obtained rating is compared with the rating of previous solutions. The current solution is only accepted when the rating is better than the rating of any solutions before. When no more extensions are available the method returns the best extension on line 26.

Listing 6.2 shows the implementation of the `compactness` method.

```
1  private double compactness(Graph graph) throws IOException {
```

```
2       Path shortestPaths = new Path("data/graph-paths");
3       if (fs.exists(shortestPaths)) {
4           fs.delete(shortestPaths, true);
5       }
6
7       Path hdfsGraphPath = new Path("data/graph");
8       Path vertexDetail = new Path("data/vertices");
9       try (HdfsGraphWriter graphWriter = new HdfsGraphWriter(new
            ↪ OutputStreamWriter(fs.create(hdfsGraphPath)))) {
10          graphWriter.write(graph);
11      }
12      try (BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(fs.
            ↪ create(vertexDetail)))) {
13          for (Vertex v : graph.getVertices()) {
14              bw.write(String.format("%d;%s", v.getId(), v.getNodeType().name
                    ↪ ()));
15              bw.newLine();
16          }
17      }
18
19      String landmarksOption = graph.getVertices().stream()
20              .filter(v -> EnumSet.of(Vertex.NodeType.STATION, Vertex.
                    ↪ NodeType.ELIGIBLE_NEW_STATION).contains(v.getNodeType())
                    ↪ )
21              .map(v -> Long.toString(v.getId()))
22              .collect(Collectors.joining("-"));
23
24      GiraphConfiguration giraphConfiguration = new GiraphConfiguration(
            ↪ hadoopConf);
25      giraphConfiguration.setVertexInputFormatClass(
26              VertexInputFormat.class);
27      giraphConfiguration.setVertexOutputFormatClass(
28              ShortestPathVertexOutputFormat.class);
29      giraphConfiguration.setStrings(ShortestPathAlgorithm.TARGET_IDS.getKey
            ↪ (),
30              landmarksOption);
31      giraphConfiguration.setWorkerConfiguration(1, 2, 100.0f);
32      giraphConfiguration.setYarnLibJars(jarLocation);
33      giraphConfiguration.setComputationClass(ShortestPathAlgorithm.class);
34      GiraphFileInputFormat.addVertexInputPath(giraphConfiguration,
            ↪ hdfsGraphPath);
35      giraphConfiguration.set(FileOutputFormat.OUTDIR, shortestPaths.toString
            ↪ ());
36
37      GiraphYarnClient giraphYarnClient = new GiraphYarnClient(
            ↪ giraphConfiguration, getClass().getName());
38      try {
39          if (!giraphYarnClient.run(true)) {
40              throw new RuntimeException("Giraph␣job␣failed.");
41          }
42      } catch (YarnException e) {
43          throw new RuntimeException(e);
44      }
```

```
45
46      Path joinOutput = new Path("data/joinOut");
47      Path shortestPathToStation = new Path("data/shortestPathToStation");
48      joinVertexDetails(shortestPaths, vertexDetail, joinOutput);
49      reduceShortestPathToStation(joinOutput, shortestPathToStation);
50
51      int k = getMaxShortestPathLength(shortestPathToStation);
52      int shortestPathSum = getShortestPathLengthSum(shortestPathToStation);
53
54      long numSensors = graph.getVertices().stream().filter(v ->
            ↪ getNodeType() == Vertex.NodeType.SENSOR).count();
55      long max = k * numSensors;
56      long min = numSensors;
57      return (double) (max - shortestPathSum) / (max - min);
58  }
```

Listing 6.2: Implementation of the MapReduce/Giraph client application

The first part of the method writes data to HDFS that is needed for the computations such as the input graph on lines 9 - 11 and the vertex details on lines 12 - 17 that are joined to the shortest paths later on.

At this point it should be noted that Giraph only supports directed graphs and a vertex is only aware of its outgoing edges by default. However, the input graph structure only contains at most one edge between to vertices that is considered undirected. In order to not lose the undirected semantics when using the graph within a Giraph algorithm it is necessary to write out and inverse duplicate of every edge in the input graph. This duplication is performed in the course of writing the graph to HDFS using the custom `HdfsGraphWriter`.

Lines 19 - 22 extract all nodes of type ELIGIBLE_NEW_STATION from the input graph and create a dash-separated string of their vertex IDs that is used as an input to the Giraph job. Lines 19 - 35 set up the required configuration for running the Giraph job. This involves specifying a vertex input and output format on lines 25 and 27 so that Giraph knows how to extract the graph data from the input file specified on line 34 and how to write the final state of the graph to the output path specified on line 35 once the computation has finished. On line 29 the configuration is populated with the landmark vertex IDs that were previously converted to a string form. Line 31 sets some configuration on the number of workers and before line 32 designates the location of the JAR file on the client that contains the required artifacts for running the job. Giraph needs to be aware of the entry point to the algorithm which is designated on line 33 by setting the `ShortestPathAlgorithm` class reference. Giraph supplies the `GiraphYarnClient` to submit Giraph jobs to the cluster. This client is instantiated on line 37 and the job is started shortly thereafter on line 39.

Once the shortest path job has finished, the client schedules the MapReduce join jobs that populate the generated shortest paths with the node types of the start and end vertices. This is done inside the `joinVertexDetails` method called on line

48. After that, the path filtering and reduction is carried out by the invocation of `reduceShortestPathToStation` on line 49. Finally, the maximum shortest path length and the shortest path length sum are computed on lines 51 and 52. The results are immediately used to calculate the compactness metric for the input graph which is returned on line 57.

The rest of this section examines the implementation of each MapReduce/Giraph job illustrated in Figure 6.3 in sequence.

6.3.1 Giraph shortest path algorithm

The implementation of the shortest path algorithm as shown in Listing 6.3 exactly follows the approach described in Section 5.2.

```
1  @Algorithm(name = "shortest path")
2  public class ShortestPathAlgorithm extends BasicComputation<LongWritable,
       ↪ ShortestPathMap, NullWritable, ShortestPathMap> {
3      private static final Logger LOG = LoggerFactory.getLogger(
           ↪ ShortestPathAlgorithm.class);
4      public static final StrConfOption TARGET_IDS =
5              new StrConfOption("shortestPath.targetIds", "",
6                      "The target vertex ids to compute the shortest paths to
                           ↪ ");
7
8      private Set<Long> targetVertexIds;
9
10     @Override
11     public void initialize(GraphState graphState,
           ↪ WorkerClientRequestProcessor<LongWritable, ShortestPathMap,
           ↪ NullWritable> workerClientRequestProcessor,
           ↪ CentralizedServiceWorker<LongWritable, ShortestPathMap,
           ↪ NullWritable> serviceWorker, WorkerGlobalCommUsage
           ↪ workerGlobalCommUsage) {
12         super.initialize(graphState, workerClientRequestProcessor,
               ↪ serviceWorker, workerGlobalCommUsage);
13
14         targetVertexIds = Arrays.stream(TARGET_IDS.get(getConf()).split("-"
               ↪ ))
15                 .filter(s -> !s.isEmpty())
16                 .map(Long::valueOf)
17                 .collect(Collectors.toSet());
18     }
19
20     @Override
21     public void compute(
22             Vertex<LongWritable, ShortestPathMap, NullWritable> vertex,
23             Iterable<ShortestPathMap> messages) throws IOException {
24         if (getSuperstep() == 0) {
25             initState(vertex);
26             LOG.debug("Initial state vertex " + vertex.getId() + ": " +
                   ↪ vertex.getValue());
27         }
```

```
28
29          // shortest path computation
30          ShortestPathMap updatedState = vertex.getValue();
31          for (ShortestPathMap msg : messages) {
32              updatedState = mergeMaps(updatedState, msg);
33          }
34
35          if ((getSuperstep() == 0 && !updatedState.isEmpty()) ||
36                  !updatedState.equals(vertex.getValue())) {
37              vertex.setValue(updatedState);
38              ShortestPathMap incrementedState = increment(updatedState);
39              sendMessageToAllEdges(vertex, incrementedState);
40          }
41
42          vertex.voteToHalt();
43      }
44
45      private ShortestPathMap mergeMaps(ShortestPathMap map1, ShortestPathMap
            ↪ map2) {
46          Set<Long> combinedKeySet = new HashSet<>(map1.keySet());
47          combinedKeySet.addAll(map2.keySet());
48          return combinedKeySet.stream()
49                  .collect(Collectors.toMap(
50                          Function.identity(),
51                          vertexId -> Math.min(map1.getOrDefault(vertexId,
                                ↪ Integer.MAX_VALUE), map2.getOrDefault(
                                ↪ vertexId, Integer.MAX_VALUE)),
52                          (a, b) -> a,
53                          ShortestPathMap::new
54                  ));
55      }
56
57      private ShortestPathMap increment(ShortestPathMap shortestPathMap) {
58          return shortestPathMap.entrySet()
59                  .stream()
60                  .collect(Collectors.toMap(
61                          Map.Entry::getKey,
62                          entry -> entry.getValue() + 1,
63                          (a, b) -> a,
64                          ShortestPathMap::new
65                  ));
66      }
67
68      private void initState(Vertex<LongWritable, ShortestPathMap,
            ↪ NullWritable> vertex) {
69          ShortestPathMap newState = new ShortestPathMap();
70          if (isTarget(vertex)) {
71              newState.put(vertex.getId().get(), 0);
72          }
73          vertex.setValue(newState);
74      }
75
76      private boolean isTarget(Vertex<LongWritable, ?, ?> vertex) {
```

```
77            return targetVertexIds.contains(vertex.getId().get());
78        }
79    }
```

Listing 6.3: Implementation of the shortest path algorithm with Giraph

A Giraph job or algorithm is created by implementing the `org.apache.giraph.graph.Computation` interface. In the case of the shortest path algorithm the implementation class derives from the abstract `org.apache.giraph.graph.BasicComputation` class that provides some basic functionality like message sending assuming that incoming and outgoing messages are of the same type. The type parameters on line 2 specify the vertex id type, the vertex state type, the edge state type and the message type, in that order.

Since the shortest path algorithm takes a set of target vertex IDs as input, a respective configuration option is defined on line 4 that facilitates the transmission of this parameter from the client application as part of the Giraph job configuration.

On line 11 the `initialize` method is overridden to read and parse the supplied target vertex IDs aka landmarks and to make them available to the main algorithm.

The `compute` method starting on line 21 represents the per-vertex logic that is invoked by the Giraph framework and contains the essential algorithmic logic. At the beginning of the first superstep, each vertex invokes the `initState` method which initializes the vertex state depending on whether it encounters a target vertex or not. On line 31 any received messages are read and merged into the local state using the `mergeMaps` function on line 32. The method combines the entries of both passed shortest path maps and uses the minimal path lengths when clashing entries are encountered. After processing the received messages from the last superstep, each vertex checks if it needs to propagate its state by comparing the newly updated state against its old state on line 35. Superstep 0 represents a special case since every landmark vertex needs to send out its state irrespective of any state changes. Before the actual sending of the state on line 39 the path lengths need to be incremented by calling the `increment` method on line 38 to reflect the additional edge on the path when the state is received by a neighboring node. As a last action in each superstep, every vertex votes to halt on line 42. Recall that vertices that receive messages in the upcoming superstep will be reactivated automatically by the framework even if they have previously voted to halt.

6.3.2 Joining vertex details

The MapReduce join job is capable of combining the records of multiple input files based on a common key. The input for the job consists of the input file paths to join apart from configuration entries containing a common column separator string and the number input files to join. Moreover, for each input file, two configuration entries need to be supplied:

- keyIndex - Column index of the join key

- joinOrder - Ordering index that indicates for a file at which position its record columns should be placed in the resulting joined records

The join operation consists of a mapper and a reducer. It is the job of the `JoinMapper` to separate the join key from the record and to emit it as MapReduce key along with the remaining record columns as value. Listing 6.4 shows the `map` method that performs these steps. Also note that on line 4 the input file dependent join order is prepended to every emitted tuple value. This information is later used by the reducer to construct the combined tuples with the correct column order.

```
1 protected void map(LongWritable key, Text value, Context context) throws
    ↪ IOException, InterruptedException {
2     List<String> values = new ArrayList<>(Arrays.asList(value.toString().
        ↪ split(separator)));
3     String joinKey = values.remove(keyIndex);
4     values.add(0, Integer.toString(joinOrder));
5     String valuesWithOutKey = values.stream().collect(Collectors.joining(
        ↪ separator));
6     keyWrapper.set(joinKey);
7     data.set(valuesWithOutKey);
8     context.write(keyWrapper, data);
9 }
```

Listing 6.4: map method of the JoinMapper

Listing 6.5 shows the implementation of the corresponding reducer.

```
1  public class JoinReducer extends Reducer<Text, Text, NullWritable, Text> {
2
3      private Text joinedText = new Text();
4      private StringBuilder builder = new StringBuilder();
5
6      @Override
7      protected void reduce(Text key, Iterable<Text> values, Context context)
           ↪ throws IOException, InterruptedException {
8          String separator = context.getConfiguration().get("separator");
9          int joinSites = context.getConfiguration().getInt("joinSites", -1);
10         Map<Integer, List<String>> joinTuplesPerSite = StreamSupport.stream
               ↪ (values.spliterator(), false)
11                 .map(value -> value.toString().split(separator))
12                 .collect(Collectors.toMap(
13                         valueParts -> Integer.parseInt(valueParts[0]),
14                         valueParts -> new ArrayList<>(Collections.
                               ↪ singletonList(Arrays.asList(valueParts).
                               ↪ subList(1, valueParts.length).stream().
                               ↪ collect(Collectors.joining(separator)))),
15                         (a, b) -> { a.addAll(b); return a; }
16                 ));
17
18         join(joinTuple -> {
19             builder.append(key.toString()).append(separator);
20             for (String value : joinTuple) {
```

```
21                  builder.append(value).append(separator);
22              }
23              builder.setLength(builder.length() - 1);
24              joinedText.set(builder.toString());
25              try {
26                  context.write(NullWritable.get(), joinedText);
27              } catch (IOException e) {
28                  throw new UncheckedIOException(e);
29              } catch (InterruptedException e) {
30                  Thread.currentThread().interrupt();
31                  throw new RuntimeException(e);
32              }
33              builder.setLength(0);
34          }, joinTuplesPerSite, joinSites, 0, new ArrayList<>(joinSites));
35      }
36
37      private void join(Consumer<List<String>> joinTupleConsumer, Map<Integer
            ↪ , List<String>> joinTuplesPerSite, int joinSites, int
            ↪ currentJoinSite, List<String> currentJoinTuple) {
38          if (currentJoinSite < joinSites) {
39              if (joinTuplesPerSite.containsKey(currentJoinSite)) {
40                  for (String tuple : joinTuplesPerSite.get(currentJoinSite))
                        ↪ {
41                      currentJoinTuple.add(tuple);
42                      join(joinTupleConsumer, joinTuplesPerSite, joinSites,
                            ↪ currentJoinSite + 1, currentJoinTuple);
43                      currentJoinTuple.remove(currentJoinTuple.size() - 1);
44                  }
45              }
46          } else {
47              joinTupleConsumer.accept(currentJoinTuple);
48          }
49      }
50  }
```

Listing 6.5: Implementation of the JoinReducer

On line 4 the reducer uses the join order that has been prepended to each input value by the mapper to group the input values accordingly. After that, the joined tuples are created and emitted by recursively combining the values of each join order using the `join` method on line 18.

6.3.3 Path filtering and minimization

The filtering and minimization step removes any shortest path entries between station nodes and reduces multiple path entries for a single sensor node to the path entry with minimum length.

Listing 6.6 shows the implementation of the mapper.

```
1   public class FilteringMinPathMapper extends Mapper<LongWritable, Text,
        ↪ LongWritable, Text> {
```

```
2
3       private LongWritable keyWrapper = new LongWritable();
4
5       @Override
6       protected void map(LongWritable key, Text value, Context context)
            ↪ throws IOException, InterruptedException {
7           List<String> columns = new ArrayList<>(Arrays.asList(
8                   value.toString().split(";")
9           ));
10          Vertex.NodeType nodeType1 = Vertex.NodeType.valueOf(columns.get(1))
                ↪ ;
11          Vertex.NodeType nodeType2 = Vertex.NodeType.valueOf(columns.get(4))
                ↪ ;
12
13          if (nodeType2 == Vertex.NodeType.SENSOR) {
14              keyWrapper.set(Long.parseLong(columns.get(2)));
15              context.write(keyWrapper, value);
16          }
17      }
```

Listing 6.6: Mapper implementation for path filtering and minimization

On lines 13 - 16 the mapper uses the joined node type information to only emit paths with the starting vertex being a sensor node. The ID of the sensor vertex is used as reducer key resulting in any path entries originating from the same sensor being processed by the same reducer which can then perform the required minimization as shown in Listing 6.7.

```
1   protected void reduce(LongWritable key, Iterable<Text> values, Context
        ↪ context) throws IOException, InterruptedException {
2       String minValue = StreamSupport.stream(values.spliterator(), false)
3               .map(Object::toString)
4               .min(Comparator.comparingInt(
5                       v -> Integer.parseInt(v.split(";")[3])
6               )).get();
7
8       valueWrapper.set(minValue);
9       context.write(NullWritable.get(), valueWrapper);
10  }
```

Listing 6.7: Reducer implementation for path filtering and minimization

6.3.4 Global reductions

The reduction jobs for computing the maximum shortest path and the shortest path length sum share a common mapper that is shown in Listing 6.8. The index of the record column that is to be reduced is provided via a configuration option. Since the goal is to reduce all path entries to a single value the mapper just emits the reduction column value without any key causing all entries to be processed by the same reducer.

```
1   valueWrapper.set(value.toString().split(";")[reduceColumn]);
2   context.write(NullWritable.get(), valueWrapper);
```

```
3 }
4 }
```

Listing 6.8: Common global reduction mapper

The maximum and sum reducers are very straight-forward as shown in Listings 6.9 and 6.10, respectively.

```
1  protected void reduce(NullWritable key, Iterable<Text> values, Context
       ↪ context) throws IOException, InterruptedException {
2      OptionalInt optionalMax = StreamSupport.stream(values.spliterator(),
           ↪ false)
3              .mapToInt(text -> Integer.parseInt(text.toString()))
4              .max();
5
6      if (optionalMax.isPresent()) {
7          valueWrapper.set(Integer.toString(optionalMax.getAsInt()));
8          context.write(NullWritable.get(), valueWrapper);
9      }
10 }
```

Listing 6.9: Reducer for maximum shortest path reduction

```
1  protected void reduce(NullWritable key, Iterable<Text> values, Context
       ↪ context) throws IOException, InterruptedException {
2      int sum = StreamSupport.stream(values.spliterator(), false)
3              .mapToInt(text -> Integer.parseInt(text.toString()))
4              .sum();
5
6      valueWrapper.set(Integer.toString(sum));
7      context.write(NullWritable.get(), valueWrapper);
8  }
```

Listing 6.10: Reducer for shortest path length sum reduction

CHAPTER 7

Apache Spark GraphX

Graph-parallel abstractions such as Pregel are based on iterative vertex-centric transformations. A user-defined vertex program iteratively changes vertex properties based on messages received from adjacent vertices, i.e. the neighborhood. Parallelism is produced by concurrently executing the vertex program on different vertices. While this works well for iterative graph algorithms like PageRank that rely on static neighborhood structures, it is problematic in case non-adjacent vertices need to interact or when the graph structure needs to be changed, for example. In contrast, a data-parallel abstraction adopts a record-centric view and derives parallelism by processing multiple records in parallel [XCD+14].

GraphX does not directly adopt graph-parallel paradigms like Pregel but builds on Spark's existing general-purpose distributed dataflow model (see Chapter 4). In order to deliver performance comparable to systems like Apache Giraph that directly implement a graph-parallel execution model the general dataflow model is extended by optimizations that specifically target graph processing. In addition, GraphX also manages to provide a Pregel API that utilizes the extended distributed dataflow framework. This allows GraphX to cover a wider range of computations than purely graph-parallel systems because it intuitively supports neighborhood-centric algorithms via Pregel as well as graph-wide, neighborhood-independent algorithms via MapReduce-style data-parallel transformations. Moreover, it facilitates the flexible reuse and combination of datasets by allowing the same physical data to be exposed as graph or collection and by supporting joins of graph-structured data with unstructured or tabular data. This enables powerful analytics pipelines that reduce data movement and duplication to a minimum which is not possible in graph-parallel systems [XCD+14] [GXD+14].

The remainder of this chapter discusses

- the graph partitioning strategy of GraphX

- the representation of graphs in GraphX
- the implementation of Pregel programming model on top of data-parallel operators
- essential optimizations like indexes that GraphX employs for efficient graph-processing

7.1 Graph partitioning

Pregel and most other graph-parallel programming models assume edge-cut based graph partitioning which ensures that each vertex is uniquely assigned to a partition. As detailed in Section 5, a balanced edge-cut is the optimal configuration for distributed graph processing because it minimizes the communication overhead. However, this problem is NP-complete and thus infeasible for large real-world graphs [GJ02]. While, in practice, heuristics can be used to find good partition assignments in reasonable time, they do not provide any guarantees or bounds with respect to the quality of the solution (see example 6). Also, real world graph-parallel systems such as Apache Giraph only come with primitive algorithms that effectively create random edge-cuts. While optimally balancing the vertices across nodes, this approach has been shown to result in nearly worst-case communication overhead because most edges are cut [GLG+12].

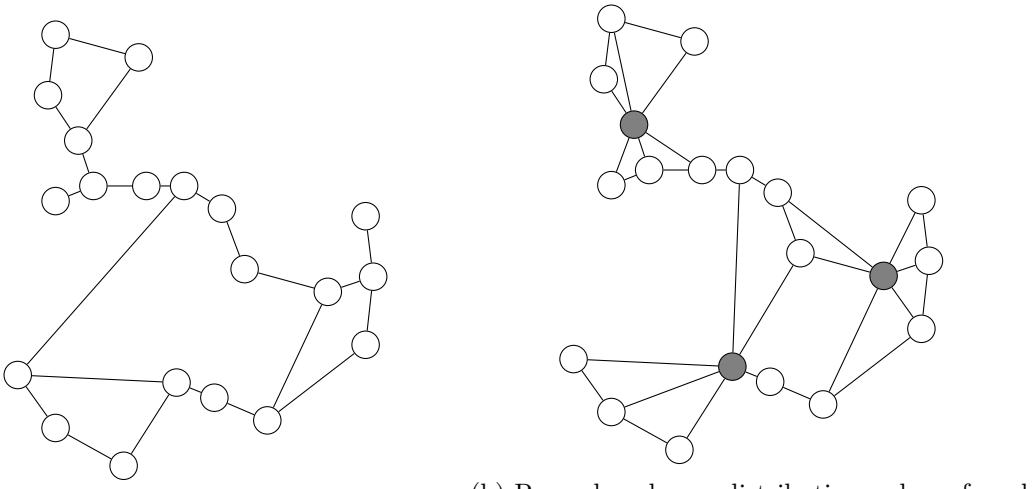
Contrary to edge-cuts, vertex-cuts evenly assign edges to partitions and allow vertices to span multiple partitions (see Figure 7.2). In this model, the communication overhead correlates with the number of split vertices. An optimal configuration for distributed graph processing can therefore be achieved by balancing edges across cluster nodes while minimizing the vertex splits [XGFS13]. Like with edges-cuts the computation of vertex-cuts is infeasible for large graphs [XGFS13]. However, it has been shown that a simple random vertex-cut can be shown to reduce the communication overhead by orders of magnitudes for power-law graph compared to random edge-cuts [GLG+12]. But also for ordinary graphs, vertex-cuts are a better option for partitioning because it can be shown that for any edge-cut can be converted to a strictly better vertex-cut [GLG+12].

Another argument in favor of vertex-cut is the fact that most real world graphs have a power-law degree distribution [GLG+12], i.e. vertex degrees are power-law distributed which results in graphs composed of hubs as illustrated in Figure 7.1. While this is known to be a problem for edge-cuts [XCD+14], percolation theory suggests that power-law graphs have good vertex-cuts [AJB00].

For the above reasons, GraphX uses vertex-cut based graph partitioning. By default, the edge partitioning is inherited from the input collection, e.g. HDFS blocks. GraphX supplies four different algorithms for repartitioning which are described in the following.

The *RandomVertexCut* algorithm partitions edges by computing a combined hash value for the source and destination vertex.

The *CanonicalRandomVertexCut* algorithm works like RandomVertexCut but treats edges as undirected, i.e. edges (i, j) are assigned to the same partition as edges (j, i).



(a) Random degree distribution

(b) Power-law degree distribution - also referred to as *scale-free*

Figure 7.1: Illustration of power-law distributed graphs

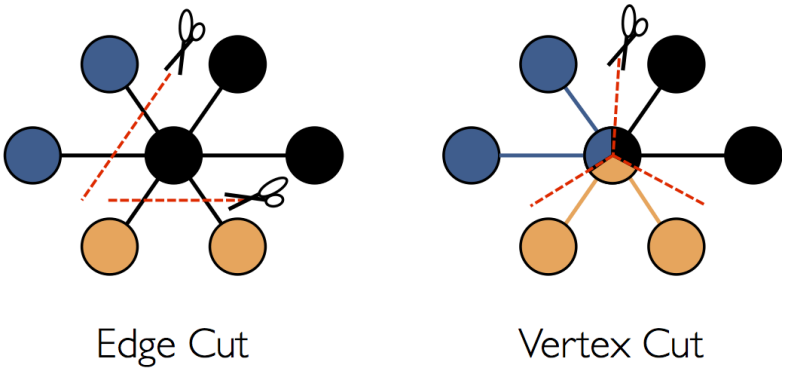


Figure 7.2: Edge cut versus vertex cut [graa]

The *EdgePartition1D* algorithm partitions edges based on their source vertex id.

The *EdgePartition2D* algorithm adopts a the graph adjacency matrix to perform the partitioning. Given M partitions for assigning edges, the adjacency matrix is divided into $\sqrt{M} \times \sqrt{M}$ regions, each corresponding to one partition. See Figure 7.3 for an illustration of the idea. This division allows to express an upper bound of $2\sqrt{M}$ on the replication factor of any vertex in the graph [XGFS13].

7.2 Graph representation

The data model adopted by GraphX is the *property graph* $G(V, E, P)$ consisting of a vertex set V, and edge set E and a property set $P = (P_V, P_E)$ of vertex properties P_V and edge properties P_E. An important characteristic of this model is the independence

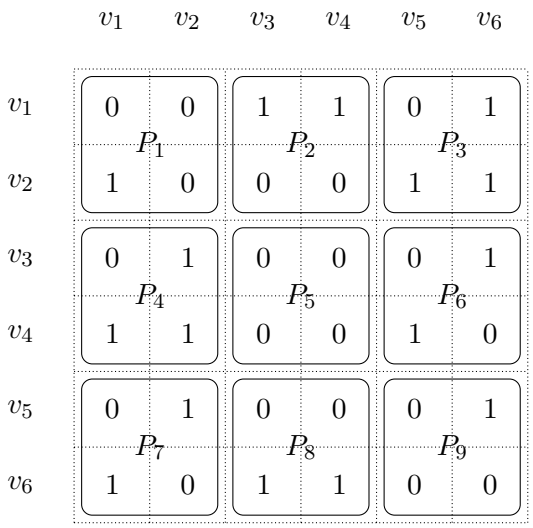


Figure 7.3: Illustration of the 2D graph partitioning algorithm in GraphX

of graph structure and properties. This allows GraphX to join an existing graph with additional data resulting in extended properties but retained graph structure [XCD+14].

Internally, a property graph is represented by two RDDs (see Section 4.1), one vertex RDD and one edge RDD containing records $(i, P_V(i))$ with $i \in V$ and tuples $((i,j), P_E((i,j)))$ with $(i,j) \in E$, respectively [XCD+14]. This representation is also referred to as resilient distributed graph (RDG). An RDG inherits the fault tolerance properties from its constituent RDDs, including immutability. Hence, any transformation performed on an RDG produces a new graph [XGFS13].

For efficient lookup and access to edges, GraphX provides a clustered index on source vertex ID and an unclustered index on target vertex id. The vertex RDD is hash partitioned by vertex ID and the vertices are stored alongside a vertex ID based, clustered hash index in each partition. Every vertex partition also contains a bitmask and has a corresponding routing table. The former is used to exclude specific vertices from graph operations while the latter is required for optimizing distributed joins and for index sharing (see Section 7.4). The routing table is a map from vertex ID to the set of edge partitions containing adjacent edges [XCD+14]. Figure 7.4 illustrates the RDG layout.

7.3 From graph-parallel to data-parallel

GraphX provides graph-parallel APIs like Pregel on top of Spark's data-parallel computation model. This is possible by using gather-apply-scatter (GAS) decomposition [GLG+12] to decompose graph-parallel vertex programs supplied by the user into three data-parallel stages:

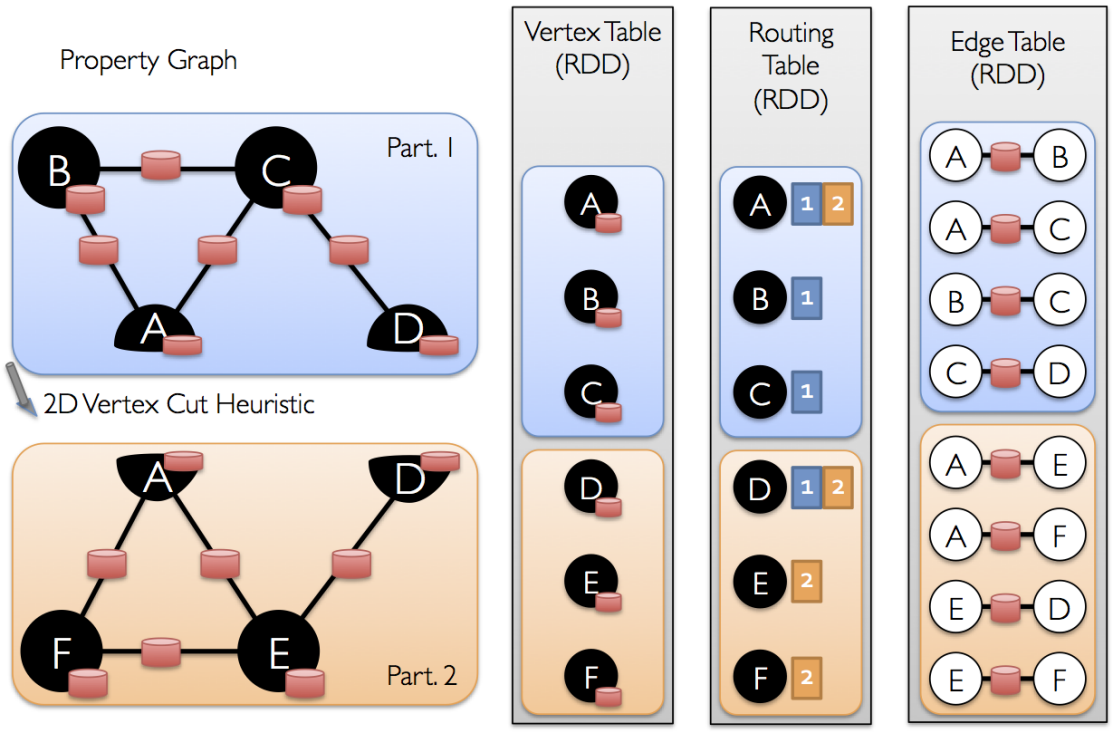


Figure 7.4: Illustration of the RDG layout [grab]

- Gather - aggregate incoming messages (edge-parallel operation)
- Apply - update vertex state (vertex-parallel operation)
- Scatter - compute outgoing messages (edge-parallel operation)

Instead of executing a vertex program independently for each vertex, the above decomposition allows each stage to be performed in a data-parallel fashion for the whole graph [GXD+14]. In an abstract sense, the gather and scatter phases replace the active message passing performed by each vertex in the graph-parallel model by a pull-based approach. The runtime fetches the messages destined for any vertex in the graph along a vertex' incoming edges and applies the gather function to perform a reduction. Finally, the scatter phase computes the new messages to be gathered in the successive round or superstep. One implication of this decomposition is that the range of possible communication patterns is more restricted because messaging is only allowed between adjacent vertices. In contrast, the original Pregel model also allows communication between non-adjacent vertices [XCD+14].

Example 7 For illustration purposes, the GAS decomposition is applied to the Pregel sum algorithm from Example 4. Listing 7.1 shows the resulting algorithm.

```
class Message {
        Integer number = 0
        Integer messageCounter = 0
}

gather (Message msg1, Message msg2) {
        msg1.number += msg2.number
        msg1.messageCounter += msg2.messageCounter
        return msg1
}

apply (Vertex v, Message aggregatedMsg) {
        v.number += aggregatedMsg.number;
        v.totalMessagesReceived += aggregatedMsg.
            ↪ messageCounter
}

scatter (Vertex v, Vertex adjacentVertex) {
        Message msg;
        if (  v.number > 0 AND
                  v.totalMessagesReceived > 0 AND
                  v.id != minimumVertexId AND
                  adjacentVertex == v.minNeighbor) {

                // Send stored value to adjacent vertex
                // with minimum id
                msg.number = v.number
                msg.messageCounter = 1
                v.number = 0
                voteToHalt()
        } else if (msgs.isEmpty) {
                voteToHalt()
        }
        return msg
}
```

Listing 7.1: GAS sum algorithm

The `gather` function receives two messages and aggregates them without making any assumptions with respect to the ordering of the message parameters. It is necessary to introduce a compound message type `Message` because the number sum as well as the number of received messages need to be aggregated.

The `apply` function receives the current vertex and the corresponding aggregated message and trivially updates the vertex state.

> Most of the logic in the sum algorithm is concerned with message passing and is thus moved to the `scatter` function which generates an outgoing message for every edge `(v, adjacentVertex)`.

From a system perspective, the basis for executing a GAS algorithm in GraphX is the *mrTriplets* operator. It is based on the *triplets* operator that joins edges with adjacent vertices and produces a collection of key-value items of the form $((i, j), (P_E(i, j), P_V(i), P_V(j)))$. The *mrTriplets* extends this functionality by additionally performing a MapReduce operation on the produced triplets. This can be used to construct messages in the map phase that are then grouped and reduced by destination vertex id in the reduce phase.

The Pregel programming model can be implemented on top of this operator as listing 7.2 shows. The code wraps the user defined `sendMsg` function in a `send` function because the GraphX `mrTriplets` API expects a scatter function of the form `Triplet[V, E] ↪ => Iterator[(VertexId, M)]`.

In line 13, the *mrTriplets* operator is called on the input graph. It produces triplets for the graph and then applies the `send` function on the triplets which in turn calls the user defined `sendMsg` function. This produces a collection of messages, one for each triplet, which is then subject to reduction via the user defined `gather` function. The outcome of this sequence of operations is a collection of aggregated messages, one for each connected vertex in the graph.

At the beginning of line 15, the messages are actually s̈entẗo the vertices by joining the graph with the messages. A left join is used in order to retain unconnected vertices that do not receive any messages. Immediately after, the vertex program is applied on the join result using the `mapV` operator.

Also observe that this implementation actually performs a scatter-gather-apply sequence of operation. Otherwise there could not possibly be any messages in the first iteration for the vertex program to operate on since the first messages would only be sent *after* the vertex program invocation. Hence, this reordering makes sense in practice.

```
1  def pregel(g: Graph[V,E],
2          vprog: (V, M) => V,
3          sendMsg: Triplet[V, E] => M,
4          gather: (M, M) => M):
5  Graph[V, E] = {
6          def send(t: Triplet[V, E]) = {
7                  Iterator(t.dstId, sendMsg(t))
8          }
9          var live = g.vertices.count
10         // Loop until convergence
11         while (live > 0) {
12                 // Compute the messages
13                 val msgs = g.mrTriplets(send, gather, Out)
14                 // Receive the messages and run vertex program
15                 g = g.leftJoin(msgs).mapV(vprog)
```

```
16                  // Count the active vertices
17                  live = g.vertices.filter(v=>!v.halt).count
18          }
19          return g
20  }
```

Listing 7.2: Implementation of Pregel using *mrTriplets* [XCD+14]

7.4 GraphX optimizations

This section describes some of the special optimizations that are employed in GraphX.

As detailed in Section 7.2, the graph representation used in GraphX contains indexes for edge and vertex RDDs to allow fast lookups. The immutability of graphs in GraphX allows such indexes to be shared and reused across structurally equal graphs. This not only reduces memory consumption but it also accelerates local graph operations like joins and aggregations with custom RDDs. Graphs resulting from only restrictive graph structure modifications like edge or vertex removal can also share the index with the original graph by using the bitmaps stored in the vertex partitions to selectively disable vertices. As a result, such vertices are no longer considered by operations on the new graph although they are still present in the shared index [GXD+14].

For the graph-parallel operations, GraphX employs special optimizations in order to meet the performance of purely graph-parallel systems. Because the *mrTriplets* operator is the heart of all graph-parallel functionality in GraphX, speedups in this area have the largest overall impact. Building the triplets involves a three-way join between an edge and its source and destination vertices which requires data movement in one of two possible ways: move edge data to the vertex partitions or move vertex data to the edge partitions. Since real world graphs have much more edges than vertices and because there is a chance that an edge partition contains multiple edges adjacent to a single vertex, GraphX uses the edge partitions as join sites. The shipment of vertex data to edge partitions is also referred to as vertex replication. To further optimize the process, GraphX utilizes the routing table to move individual vertices exclusively to edge partitions containing adjacent edges.

In theory, for iterative graph-parallel algorithms, the *mrTriplets* operator needs to be invoked in each superstep. However, GraphX supports two important join optimization techniques called *incremental view maintenance* and *automatic join elimination* that both exploit the nature graph-parallel applications to reduce the amount of data that needs to be communicated when constructing the triplets [XCD+14].

As graph-parallel computations converge, more and more vertices become inactive and change their state. *Incremental view maintenance* prevents the shipment of unchanged vertices for constructing the triplets by tracking changes in a separate bitmap [GXD+14] [XCD+14].

A graph-parallel computation might not access vertex attributes of both the source and the target vertex inside the *mrTriplets*' map function. *Automatic join elimination* analyzes the Java Virtual Machine (JVM) bytecode comprising the map function and rewrites the three-way join to prevent joining vertices which are not required in the map function. Thereby, the original three-way join can sometimes be reduced to a two-way join or eliminated altogether [GXD+14] [XCD+14].

The *mrTriplets* operator invokes the map function only on triplets containing active vertices, i.e. vertices that have not yet finished their processing. This requires a sequential scan over all vertices which is wasteful in late iterations of graph-parallel computations when a large majority of vertices is usually inactive. For this reason, depending on the remaining fraction of active vertices, GraphX enables index scanning for iterating over the active vertices to reduce the overhead [GXD+14] [XCD+14].

7.5 AirQuality Inc. WWAN station placement with Apache Spark GraphX

This section describes an implementation of the solution to the air quality WWAN station placement problem statement from Section 5.2 using Apache Spark GraphX. It relies on the notations introduced in Section 5.2.

Because GraphX does not provide a Java API at the time of writing the Scala API is used instead to implement this example. In contrast to the implementation of the air quality WWAN station placement problem using Apache Giraph described in Section 6.3 the GraphX implementation is very compact though it follows the exact same approach. Listing 7.3 contains the full source code of the implementation.

```
1  class GraphXCompactnessOptimizer extends CompactnessOptimizer {
2
3    def maximizeCompactness(graph: Graph,
4                            extensionIter: util.Iterator[Graph]): Graph = {
5      val conf = new SparkConf()
6        .setAppName("WWAN Station Positioning")
7        .setMaster("local[2]")
8      val sc = new SparkContext(conf)
9
10     val vertexRdd = sc.parallelize(
11       JavaConversions.collectionAsScalaIterable(graph.getVertices)
12         .filter(v => util.EnumSet.of(
13           NodeType.SENSOR,
14           NodeType.STATION
15         ).contains(v.getNodeType)
16         )
17         .map(v => (v.getId, v.getNodeType)).toSeq
18     )
19     val edges = JavaConversions.asScalaSet(graph.getEdges)
20     val edgeRdd = sc.parallelize(
21       ( edges.map(e => org.apache.spark.graphx.Edge(
```

```
22            e.getVertex1.getId,
23            e.getVertex2.getId)
24          )
25            ++
26            // add reverse direction edges
27            edges.map(e => org.apache.spark.graphx.Edge(
28              e.getVertex2.getId,
29              e.getVertex1.getId)
30            )
31            ).toSeq
32        )
33
34        var maxCompactness = 0.0
35        var bestExtension: Graph = null
36        for (extension: Graph <- JavaConversions.asScalaIterator(extensionIter)
              ↪ ) {
37          val extensionVertexRdd = sc.parallelize(
38            JavaConversions.collectionAsScalaIterable(extension.getVertices)
39              .map(v => (v.getId, v.getNodeType))
40              .toSeq
41          )
42          // no need to add reverse direction edges because the extension edges
43          // always point towards the station vertex and we are not interested
44          // in paths starting at station vertices anyway
45          val extensionEdgeRdd = sc.parallelize(
46            JavaConversions.asScalaSet(extension.getEdges)
47              .map(e => org.apache.spark.graphx.Edge(
48                e.getVertex1.getId,
49                e.getVertex2.getId
50              ))
51              .toSeq
52          )
53
54          val extendedGraph = org.apache.spark.graphx.Graph.apply[NodeType,
                ↪ Nothing](
55            vertexRdd ++ extensionVertexRdd,
56            edgeRdd ++ extensionEdgeRdd
57          )
58          val compactness = this.compactness(extendedGraph)
59          if (compactness > maxCompactness) {
60            maxCompactness = compactness
61            bestExtension = extension
62          }
63        }
64        sc.stop()
65        bestExtension
66      }
67
68      def compactness(graph: org.apache.spark.graphx.Graph[NodeType, Nothing]):
            ↪ Double = {
69        val landmarks = graph.vertices
70          .filter(v => util.EnumSet.of(
71              NodeType.STATION,
```

```
72              NodeType.ELIGIBLE_NEW_STATION
73            ).contains(v._2)
74          ).map(v => v._1)
75          .collect()
76
77        val shortestPaths = ShortestPaths.run[NodeType, Nothing](
78          graph,
79          landmarks
80        )
81        val shortestPathLengths = shortestPaths.vertices
82          .innerJoin(graph.vertices)((id, spmap, nodeType) => (spmap, nodeType)
              ↪ )
83          .flatMap(key => key._2._1.map(value => (key._1, key._2._2, value._1,
              ↪ value._2)))
84          .keyBy(t => t._3)
85          .join(graph.vertices)
86          // filter out station-to-station paths
87          .filter(t =>
88            t._2._1._2 == NodeType.SENSOR ||
89            t._2._2 == NodeType.SENSOR
90          )
91          .map(t => (t._2._1._1, t._2._1._4))
92          // only consider the length of the path to the closest station
93          .reduceByKey((dist1, dist2) => Math.min(dist1, dist2))
94          .map(t => t._2)
95
96        val k = shortestPathLengths.max()
97        val shortestPathSum = shortestPathLengths.sum()
98
99        val numSensors = graph.vertices
100         .filter(v => v._2 == NodeType.SENSOR)
101         .count()
102       val max = k * numSensors
103       val min = numSensors
104       val compactness = (max - shortestPathSum) / (max - min)
105
106       println("Shortest Path Landmarks: " + landmarks.length)
107       println("Shortest Path Lengths: " + shortestPathLengths.count())
108       println("Compactness: (%d - %.2f) / (%d - %d)"
109         .format(
110           max,
111           shortestPathSum,
112           max,
113           min
114         )
115       )
116       println("Compactness: " + compactness)
117       compactness
118     }
119   }
```

Listing 7.3: Implementation of the compactness optimization algorithm using GraphX

The `SparkContext` is created at the beginning of the `maximizeCompactness` method on line 8 so the same context can be reused across all optimization passes. On the lines 10 - 34 the base graph is created. This involves the filtering of vertices on line 10 so that only nodes of type STATION and SENSOR remain in the graph. The filtered vertex collection is passed to the `parallelize` method provided by the `SparkContext` that turns returns an RDD representation of the collection. Like Apache Giraph, GraphX also respects the edge directions so it is necessary to duplicate each existing edge in the graph and reverse its direction which is done on lines 20 - 32. Again, the processed edge collection is converted into an RDD.

The main optimization loop starts on line 36 and pulls graph extensions from the solution iterator until no more extensions exist. Each edges and vertices of each extension graph are extracted and converted into RDDs on the lines 37 - 52. Finally, on line 54, the base RDDs and the extension RDDs are combined converted into a GraphX graph representation which is then passed to the `compactness` method on line 58. The returned compactness metric is compared against the currently best compactness metric observed. The method finally returns the best extension on line 65.

The `compactness` method starts by extracting all vertices with station and eligible station node types on lines 69 - 75 to later use them as target vertices aka landmarks in the shortest path computation. Fortunately, GraphX comes with a built-in shortest path algorithm that works exactly as described in Section 5.2. The shortest path computation is invoked on line 77 and returns an RDD containing the shortest paths in multi-map layout like $(s, \langle(t_1, l_1), (t_2, l_2), ...\rangle)$ where s is the source vertex ID, t_i is a landmark vertex ID and l_i is the length of the shortest path from s to t_i, i.e. C_{st_i}. On lines 81 - 94 this result is further processed. First, the vertex properties are joined to the source vertices on line 82 resulting in the layout $(s, \langle\langle(t_1, l_1), (t_2, l_2), ...\rangle, n_s\rangle)$ where n_s is the node type of s. The nested layout is flattened to $(s, n_s, t_1, l_1), (s, n_s, t_2, l_2), ...$ in line 83. In lines 84 - 85 the vertex details for the landmark vertex are joined resulting in tuples $(s, n_s, t_1, l_1, n_{t_1}), (s, n_s, t_2, l_2, n_{t_2}), ...$ Paths in between stations should not be regarded for the compactness computation so they are filtered out in lines 87 - 90. On line 91 the tuple layout is prepared for the path minimization by only retaining the source vertex ID and the path length. Hence, the tuple layout changes to $(s, l_1), (s, l_2)$. A reduction by source vertex ID is performed by choosing the minimum path length on line 93. Finally, only the path length is retained on line 94. The remainder of the `compactness` method simply calculates and returns the metric based on the formula presented in Section 5.2.

Part III

Stream Processing

CHAPTER 8

Stream Processing Principles

While the main application of batch processing and graph processing lies in analytical computation and query evaluation over an existing, limited dataset, stream processing, on the other hand, is concerned with acting on unbounded streams of data entering a system. It is an increasingly important requirement for modern applications to act on incoming data in a timely manner, for example to trigger alerts in real time based on certain stream patterns or to filter out irrelevant aspects before persisting the data. Similar to batch processing and graph processing, it is infeasible to perform large scale stream processing on a single node due to technical limits. For this reason, distributed stream processing frameworks are capable of leveraging multiple cluster nodes for their purpose.

In contrast to the fields of batch processing (MapReduce) and graph processing (Pregel), no dominant, general purpose, high-level programming model for stream processing has emerged so far [ZDL+12]. Thus, existing frameworks offer different and often incompatible paradigms like DStreams in Spark Streaming or topologies of spouts and bolts in Storm to express stream processing tasks. However, there are common underlying principles, characteristics and limitations that are important to be aware of in order to understand the design of practical frameworks. This chapter attempts to provide an overview of the essential aspects in stream processing systems or data stream management systems (DSMS), as they are commonly referred to in the research literature.

8.1 Data streams

This section formalizes the notion of a data stream and points out important differences to traditional datasets.

A data stream S is a possibly infinite bag (multiset) of *elements* (s, τ), where s is a tuple of the schema of S and $\tau \in T$ is a timestamp associated with the element. (s, t) denotes

that element s arrives on stream S at time τ [AW04].

A relation R is a mapping from T to a finite but unbounded bag of tuples belonging to the schema of R. $R(\tau)$ denotes the set of tuples in R at instant τ. This definition is different from the traditional relational one which does not have a notion of time [AW04].

Data streams differ from traditional stored data in important points [BBD+02]:

- Data elements arrive online
- The order in which the data elements arrive is unpredictable and uncontrollable
- Streams are unbounded
- Data points that have been processed are usually either dropped or persisted and are not easily accessible any more

However, stream processing does not entirely preclude stored data because stream elements frequently need to be enriched by joining them with e.g. traditional relational data to process them in a meaningful way [BBD+02].

8.2 Querying data streams

In general, there are two types of queries that can be applied to data streams [BBD+02]:

- *One-time queries* are evaluated once on a bounded snapshot of stream elements that were received in the past. This class of queries is comparable to traditional relational queries expressible in SQL, for example.
- *Continuous queries* on streams run forever and produce new or updated results over time as new elements arrive on the stream. Output produced by such queries can be stored or can form a new data stream.

Since continuous queries are more interesting in the context of stream processing, the remainder of this section focuses on this type of query.

Another discrimination of queries can be made with respect to the definition time of the query [BBD+02]:

- *Predefined queries* are defined prior to the arrival of stream elements and can be of both one-time or continuous type.

- *Ad-hoc queries* are created dynamically at any point during the lifecycle of stream processing and can again be of either one-time or continuous type. This query category poses big challenges to the design of stream processing systems because the computation of the precise result could possibly require stream elements that have already been received and discarded by the system. However, for many applications it is sufficient to apply ad-hoc queries to future data only. Alternatively, systems might periodically store compacted summaries of the received data that allow the subsequent answering of a wide range of ad-hoc queries.

Since data streams are generally unbounded in size the computation of exact query results might require unbounded memory space as well. This could either be due to an unbounded result size or because of an unbounded amount of stream elements that need to be buffered in order to compute the result. The fall back to off-memory storage is impractical because of the high throughput requirement of usual stream processing systems. There are two common approaches to deal with this issue. One way is to restrict the expressiveness of queries such that a bounded answer size or a bounded buffer size can be guaranteed. Alternatively, the requirement for the query result precision can be relaxed and the query could return approximate results [BBD+02] [BW01]. For more details about the memory requirements of continuous queries refer to Section 8.4.

An intuitive technique for approximate query answering are *sliding windows* that restrict the scope of a query to a specific range based on time or sequence numbers. For example, by choosing a window size of 1 day, a continuous query would only consider stream elements received in the last 24 hours and any older data would be discarded. In many real world applications scenarios, such a restriction makes sense because recent data tends to be more informative than old data. However, the use of sliding windows presumes that stream elements arrive in chronological order which is not necessarily the case in practice due to environmental conditions like network latency [BBD+02]. Moreover, this method relies on window sizes small enough so that the containing stream elements fit in memory. The window size may be *time-based*, specifying to only include elements within a certain time range, or *row-based*, specifying to include a fixed number of elements that have arrived most recently [ABB+04].

Blocking query operators like aggregations or sorting pose special challenges to stream processing systems as they need to consume the entire input before producing any output. As such, they do not fit well with the stream computation model. Depending on the placement of blocking operators in the query tree, there are different approaches to mitigate this conceptual incompatibility [BBD+02].

When a blocking operators is placed at the query root, it is the last building block that is invoked to produce the final query result. As long as the operator produces a small amount of output, updates to the query result can be streamed out as new input arrives. This is usually the case for aggregations, for example. However, when the output of the blocking operator is larger in size, continuously streaming a new versions of the output

Operator	Blocking	Unbounded state
select	No	No
project	No	No
dupelim	No	Yes
join	No	Yes
merge	No	No
intersect	No	Yes
difference	Yes	Yes
group-by	Yes	Yes
sort	Yes	Yes

Table 8.1: Non-exhaustive assembly of query operators and their blocking or unbounded stateful nature [TMSF02]

would be too costly. In cases like sorting, for example, applying incremental updates to an output data structure is a more efficient way of realizing the operator [BBD+02].

These approaches are inadequate for blocking operators that occur as intermediate nodes in the query tree because their output can change when new input arrives and thus, successive operators cannot rely on stable input [BBD+02]. One possible solution is to replace blocking operators by approximative non-blocking versions. For example, stream local sorting operators have been suggested to approximate a complete ordering of stream elements [RRH99]. Another approach is to enrich data streams with *punctuations* that provide certain guarantees to consuming operators with regard to the remaining stream content [TMSF02]. For example, when grouping by a stream element field `year`, a punctuation `year > 2012` indicates that the remaining stream does not contain any more elements with a `year` smaller or equal to 2012. A group by operator can use this information to emit results for the years 2012 and before.

Contrary to blocking operators, *unbounded stateful operators* do not need to consume the complete input to produce precise results but instead, they require unbounded memory for state tracking in order to perform their function. For example, the purpose of the *dupelim* operator is to remove duplicates from a stream. In order to do so, it needs to store all stream elements that it ever witnessed which requires an unbounded amount of memory for an unbounded stream. Again, a punctuated stream can help to mitigate the theoretical unbounded memory requirements. For example, when applying the *dupelim* operator to the previously illustrated data stream containing `year` fields, the same punctuation `year > 2012` can trigger the purge of operator state for the years 2012 and before [TMSF02].

Timestamps in stream elements can be either explicit or implicit. Explicit timestamps are stream element data attributes provided by the stream data source. This is usually the case for real-world events that take place at a specific time. On the other hand, implicit timestamps are provided by the stream processing system when the incoming tuples do not have any timestamp attribute attached or when the point in time associated

with a tuple is is not important for further processing. The disadvantage of explicit timestamps is that the ordering cannot be guaranteed as tuples may not arrive in the order of their timestamps. This poses a problem for timestamp based sliding window computations. However, as long as the stream is almost sorted because the order of the arriving tuples roughly corresponds to the chronological order, limited buffering and reordering of incoming tuples is sufficient to correct the order [BBD+02].

Nevertheless, a stream processing system needs to know when buffered elements can finally be propagated further. For this purpose, the use of *heartbeats* has been proposed [SW04]. This concept is similar to punctuations as the stream is enriched with heartbeat elements τ. The receipt of such a heartbeat guarantees that all subsequent tuples have a timestamp greater than τ. Heartbeats can either be sent by the stream source or by the stream processing system via an algorithm that takes environmental parameters like clock skew and network latency into account to calculate a bound for element buffering. These parameters can be supplied manually but they may also be estimated based on previous stream data received [SW04].

Timestamp related problems also arise when merging or joining multiple streams to form a new stream because it is unclear what timestamp to attach to the joined stream elements. A simple approach is to associate each element with an implicit timestamp corresponding to the time of its creation. The advantage is that the resulting stream is implicitly ordered by timestamp. However, this order is now implementation dependent as it is influenced by the algorithm of how the underlying streams are joined. Consequently, this prevents deterministic reasoning about the query semantics [BBD+02].

Another approach is to leave the appointment of the timestamp to use for joined stream elements to the user as a part of the query definition. Clearly, the drawback in this case is that the resulting stream is unordered and needs to be buffered and sorted if an exact order is required for further processing. As already pointed out, such a reordering is only possible if the join inputs are bounded by a sliding window [BBD+02].

8.3 Load management

Stream processing systems are exposed to varying load when handling continuous queries. Due to environmental factors like changing network latencies or capacity changes at stream sources, the rate of incoming stream elements may vary over time. A skewed ratio between the rate of incoming stream elements and the time it takes for a continuous query to process the elements leads to overloading of the node running the query. This section covers techniques that can be used to deal with such situations.

While sliding windows provide a means to limit the memory requirements for continuous queries, similar approximative query answering techniques can be applied to reduce load. For illustration, suppose an incrementally maintainable data structure that is operated on by the query and that supports two operations: `update(element)` and `compute`. The `update` operation adds a new stream element to the data structure and the `compute`

operation computes the query result based on the current state of the data structure. There are no issues as long as both operations are fast relative to the rate of incoming stream elements but recall the two problematic scenarios:

1. The rate of incoming stream elements is too high. This is equivalent with the `update` operation that is called for each arriving stream element being too slow.
2. The `update` operation is fast but the `compute` operation is slow.

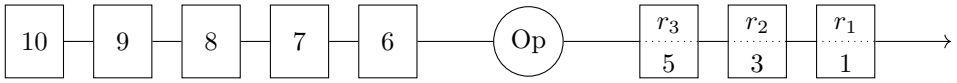
In scenario 1, *sampling* can be applied to skip a fraction of the incoming elements in order to keep the pace. As a consequence, query results are an approximation only [BBD+02].

On the other hand, to mask slow `compute` operations, *batch processing* can be applied. This way, the `compute` operation is not invoked for every stream element added to the data structure which results in batches of new stream elements processed at once in regular intervals. As long as the `update` operation is invoked for every stream element, this technique results in accurate query output. However, because the output comes with some latency it is also considered approximative [BBD+02].

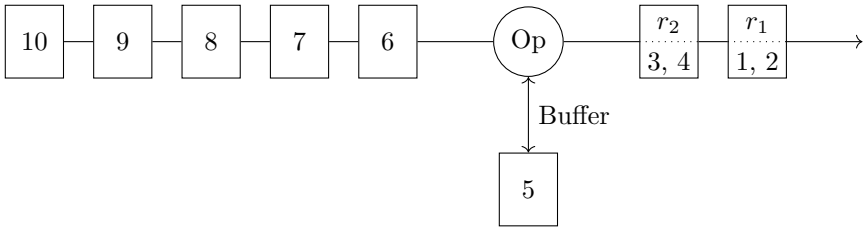
Synopsis can often server as an alternative technique to sampling and batch processing. The idea is to create compacted summaries from incoming stream elements in order to reduce the computation per element [BBD+02] [BW01].

Obviously, the above techniques can arbitrarily be combined in theory. Figure 8.1 provides an illustrative comparison of sampling, batch processing and synopsis. Note that the stream elements 2 and 4 have been dropped by sampling in Figure 8.1a. In contrast, batch processing produces less outgoing elements since it performs result computation for a set of buffered elements. In Figure 8.1b, just 2 results have been produced by the operator and element 5 has been buffered. The next output is produced once element 6 has been received. The idea of synopsis is visualized in Figure 8.1c with a separate `Syn` stage that buffers and summarizes elements before it streams the summaries to the query operator. Result r_1 is based on summary s_1 which is a summary of inputs 1 and 2. Summary s_2 is currently being streamed to the operator and input element 5 has been buffered to be summarized with element 6 upon its arrival.

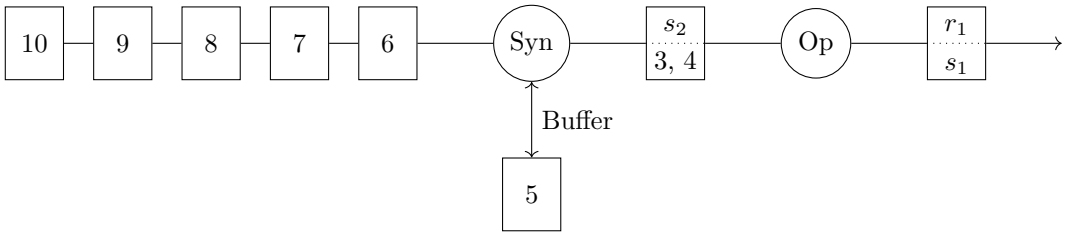
Load shedding is a technique similar to sampling since it drops tuples to reduce load. As such, this technique also results in approximative query results. While sampling is usually part of the query processing, load shedding is applied by the stream processing system at an earlier point and transparent to queries consuming a stream. This allows the dynamic adjustment of shedding strategies in order to adapt to changing loads. When multiple nodes take part in query processing in distributed stream processing systems, transparent load shedding gains importance because coordination is required among all participating nodes. The reason for this is that different nodes may differ in terms of capacity and thus, load shedding decisions at one node can lead to over- or underutilization at other nodes. Hence, load shedding in distributed environments without global coordination



(a) Sampling with probability 0.5



(b) Batch processing with batch size 2



(c) Synopsis, producing summaries for every 2 elements

Figure 8.1: Illustration of load management techniques. The boxes represent stream elements. The lower half of divided boxes indicates the original stream elements which are used to compose the current element.

leads to suboptimal shedding an arbitrary changes in query output quality [CÇC+02] [MWA+02] [TZ06] [TÇZ07].

8.4 Memory requirements

As mentioned earlier, continuous queries on unbounded data streams potentially need to consume unbounded memory to produce correct results and one common approach to mitigate this problem is to limit the expressiveness of queries. This section elaborates on query characteristics that lead to unbounded state requirements. It further investigates stream characteristics that can be exploited to limit the memory consumption.

A notation for continuous queries closely related to relational algebra is first defined for later use in illustrative examples.

- π - the duplicate-eliminating projection
- $\dot{\pi}$ - the duplicate-preserving projection
- σ - the selection operator

	Query	Bounded-state computable	
		$\Pi = \pi$	$\Pi = \dot{\pi}$
Q_1	$\Pi_A(\sigma_{A>10}(S))$	No	Yes
Q_2	$\Pi_A(\sigma_{A=D}(S \times T))$	No	No
Q_3	$\Pi_A(\sigma_{A=D \wedge A>10 \wedge D<20}(S \times T))$	Yes	Yes
Q_4	$\Pi_A(\sigma_{B<D \wedge A=10}(S \times T))$	Yes	No
Q_5	$\Pi_A(\sigma_{B<D \wedge C<E \wedge A=10}(S \times T))$	No	No
Q_6	$\Pi_A(\sigma_{B<D \wedge C<E \wedge B<E \wedge C<D \wedge A=10}(S))$	Yes	No

Table 8.2: Examples of continuous queries along with indications about their bounded memory computability depending on duplicate eliminating and duplicate preserving projection [ABB+04]

- $\times$ - the cross product operator

Π is used as a placeholder for either π or $\dot{\pi}$. Consider the example queries listed in Table 8.2 that use two data streams $S(A, B, C)$ and $T(D, E)$ [ABB+04].

Q_1 is a simple filter query on a single stream. For duplicate-preserving output, the query can check each element independently and output it if $A > 10$, hence no state needs to be maintained. In the duplicate-eliminating case, the query needs to keep track of all tuples with $A > 10$ that it has output so far. Thus, there is no finite bound to the consumed memory.

Q_2 joins stream S and T on the equality of attributes A and D. For the duplicate preserving case it is necessary to store every received attribute A and D to correctly evaluate the join. Note that storing the whole tuple is not required because the final projection only selects attribute A. With duplicate-elimination elements A and D matched once can be removed from memory but nevertheless, there is no specific bound on how long elements are stored in memory.

Q_3 is similar to Q_2 as it performs an equi-join of streams S and T on attributes A and D but in addition, it includes more restrictive selection constraints. Thus, it is possible to bound the state that needs to be held for both streams irrespective of duplicate-elimination. For both attributes A and D only values in the range $[11; 19]$ need to be stored.

Q_4 joins stream elements on the condition $B < D$ but only if $A = 10$. To evaluate the duplicate-eliminating projection it is sufficient for the query to store $B_{\min} = \min_{\{s \in S(\tau_{\text{now}}) | s.A=10\}}(s.B)$ and $D_{\max} = \max_{t \in T(\tau_{\text{now}})}(t.D)$ where $S(\tau)$ and $T(\tau)$ denote the set of stream elements received on streams S and T until time τ and τ_{now} refers to the current time. On arrival of a new tuple t_{new} on stream T it is easy to check if any tuple received on stream S in the past is joining with t_{new} by evaluating $t_{\text{new}}.D > B_{\min}$. Likewise, for any new tuple s_{new} on stream S it is possible to check for past matching tuples on stream T by evaluating $s_{\text{new}}.B < D_{\max}$. In contrast, for answering the

duplicate-preserving projections correctly, it is necessary to know the exact number of past matching tuples which cannot be done in finite memory without further constraints.

Q_5 extends Q_4 by adding an additional predicate $C < E$. On first sight, it might seem apparent that further restricting the query should not worsen the memory requirements. However, when taking a closer look it becomes clear that applying the same method for the new predicate as in Q_4 does not lead to correct results because it is invalid to independently compare the aggregates for $S.B/T.D$ and $S.C/T.E$ since it is not guaranteed that matching aggregates stem form the same tuple.

Q_6 further extends Q_5 by two more predicate $B < E$ and $C < D$. In combination with the old predicates, the requirements for a join are now:

- $C < D$ and $B < D$
- $C < E$ and $B < E$
- $A = 10$

Since both C and B must now be lower than both D and E it is valid to use a strategy analogous to that of Q_4. The query needs to store $BC_{\min} = \min_{\{s \in S(\tau_{\text{now}}) | s.A=10\}}(\max(s.B, s.C))$ and $DE_{\max} = \max_{t \in T(\tau_{\text{now}})}(\min(t.D, t.E))$. With these aggregations, the handling of arriving element works the same way as in Q_4 and also the inability of providing memory bounds for the duplicate-preserving projection remains.

The above examples show that determining the bounded memory computability of continuous queries is nontrivial. Yet, an algorithm has been proposed that is capable of determining bounded state computability for conjunctive queries with optional aggregation and which also outputs an appropriate evaluation strategy [ABB+04].

Apart from query characteristics also stream-specific properties can be exploited to limit memory requirements of otherwise unbounded state queries. For example, techniques like punctuations and timestamps as mentioned in Section 8.2 are based on this idea as they provide certain guarantees to queries with respect to the arrival of stream elements. Such assertions can be used to construct bounded state evaluation strategies.

The concept of *k-constraints* is inspired by the same motivation [BSW04]. In this model, k serves as adherence parameter that specifies the degree to which a stream satisfies the strict interpretation some constraint. Babu et al identify 4 types of constraints that are useful for memory reduction: *stream-based referential integrity on many-one joins*, *ordering* and *clustering*. Each type is discussed briefly in the following assuming row-based parameter k. However, the same principle can be applied with time-based models.

A *many-one join* is a join of multiple elements in a stream S_1 with unique join elements in another stream S_2. I.e. one element of S_1 joins with at most one element in S_2 but multiple elements in S_1 may join with the same element in S_2. The referential integrity

constraint with adherence parameter k for a stream element $s_1 \in S_1$ and its unique joining element $s_2 \in S_2$ is defined such that s_2 arrives within k elements on S_2 after s_1. For $k = 0$ this constraint is called *strict referential integrity* as s_2 is assumed to always arrive before s_1. The application of this constraint allows the query processor to buffer elements from S_1 for at most k tuple arrivals on S_2 [BSW04].

As mentioned in Section 8.2, the out-of-order arrival of elements on a stream with defined order can be problematic. The *ordered-arrival* constraint addresses this issue. It guarantees that for any element s in stream S and for all elements x that arrive at least $k + 1$ elements after s, $x \geq s$ holds. Hence, this constraint limits the buffer size for reordering elements [BSW04].

Sometimes there is no ordering relation on stream elements but it is often the case that elements arrive clustered on specific attributes. For example, assume some sensor units deployed in different timezones and all units continuously send data for some hours per day to a single data center. While there is no defined ordering for data elements within a timezone, they are going to roughly arrive clustered by timezone. The *clustered-arrival* constraint is therefore similar to the *ordered-arrival* constraint and guarantees for two elements in stream S belonging to the same cluster at most k elements belonging to different clusters arrive in between them. This constraint can be used to limit the buffer size for continuous queries that group by cluster, for example [BSW04].

The concrete value for k is implementation dependent and determining it manually would be unmanageable. Moreover, the value may change over time along with environmental conditions like network latency. However, it is possible to determine the appropriate values algorithmically on the fly by continuously analyzing incoming stream data [BSW04].

8.5 Fault-tolerance

This section discusses fault tolerance strategies in distributed stream processing systems. Such systems typically distribute continuous query handling by placing query tree operators on different nodes in order to facilitate the processing of high volume streams. Hence, the failure of individual nodes can cause downstream query operators to not receive input any more and consequently, the query as a whole seizes to produce output. In order to provide high-availability of continuous queries, a distributed stream processing system must therefore employ precautions to gracefully react to node failures.

Possible solutions involve the use of backup servers to take over the processing when primaries fail. However, in order to guarantee the same outcome of a query regardless of failures, backup and primary serves need to be synchronized so that no stream elements are duplicated. However, this involves high runtime overhead and is neither always practical nor required in every application. Thus, it makes sense to support various recovery strategies with nuanced guarantees to trade runtime overhead against recovery consistency [HBR+05] [BBMS08].

To give an example, Hwang et al [HBR+05] suggest three recovery levels:

Time range	Step size
Last hour	5 minutes
Last day	2 hours
Last 7 days	12 hours

Table 8.3: Required time range and step size aggregates

- *Precise recovery* provides fully consistent recovery.
- *Rollback recover* prevents information loss but the output in the failure case may differ from output without failure because duplicated stream data is possible.
- *Gap recovery* is the weakest recovery as some data may be lost during recovery.

When failing nodes cannot be replaced by replicas there are still scenarios where query processing may continue with incomplete information. For example, when a query operator like a join consumes multiple streams and some of them fail, the query might still be able to produce meaningful output. Balazinska et al [BBMS08] describe a proposed system that marks outputs created with incomplete inputs as tentative. Such output may be corrected later on when missing upstream nodes have been fixed.

8.6 Running example: Real time air quality statistics

The following problem is used as illustrative real world example in the course of discussing Apache Storm and Apache Spark Streaming.

Users can view rich air quality statistics for specific regions and time ranges via the website of AirQuality Inc. It is a requirement that a large number of users must be able to view the statistics concurrently and that real time data must be incorporated as well. Therefore, the incoming streaming data needs to be pre-aggregated for all possible combinations of time ranges, regions and air quality indicators to allow fast data access and short response times.

Among statistical figures like minimum, maximum and average the website should also support the display of trend diagrams. Hence, sub-aggregates need to be stored in regular intervals as defined in Table 8.3 that serve as steps for rendering the diagrams.

The solution implementations are designed to deliver timely output for all time ranges while minimizing the memory consumption for tuple buffering. For minimum and maximum over the past hour, the tuple stream is aggregated over a 1 hour sliding window. While this approach works fine for relatively short time ranges it would most certainly not work for a time range of 1 week or even 1 day, depending on the rate of incoming tuples. The reason for this is that all stream elements need to be buffered for the duration of the window. With a rate of 100 tuples per second this would mean that over 60 million messages need to be held in memory for a sliding window over 1 week. For this reason,

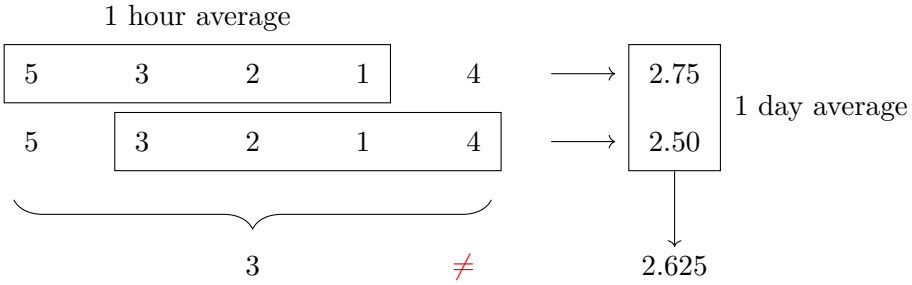


Figure 8.2: Invalid use of sliding window results for average computation

the min/max aggregations over 1 day and 1 week do not directly consume the raw source tuple stream but instead, the elements are received from a derived 1 hour aggregate stream and a 1 day aggregate stream, respectively. It is important to note that the units that produce the aggregate streams are designed to only emit state changes, i.e. for received tuples that do not change the current min/max, no new tuples are emitted. Assuming an hourly min/max change for the 1 day range, the number of tuples that need to be buffered for the 1 week sliding window decreases to just 84. This approach does not impact the timeliness of results produced by the 1 day or 1 week aggregation unit since any change produced by the 1 hour aggregator is immediately propagated to the downstream aggregators.

While the above strategy works well for min/max aggregation, it cannot be adopted for computing sliding averages because this would cause repeated inclusion of the same tuples into the average calculation when using sliding windows leading to incorrect results. In Figure 8.2, the second member of the 1 day average window is partly based on the same tuples as the first member which results in an incorrect average aggregation for the 1 day window.

For generating a stream of sliding averages in a timely and precise manner there is no way around storing the entire set of received tuples for the observed time range. While it is possible to produce timely but approximative results with reduced memory consumption using statistical tools such as stratified sampling, it does not seem to be desirable for the application presented in this section. Typically, the longer the time range the larger the set of values included in the range. Additionally, the more values an average computation includes the less impact single values have on the overall outcome even in cases of extreme outliers. Hence, for average aggregations over the range of days or weeks, it does not make much of a difference if results are produced once a minute or once an hour because they will likely be very similar. For this reason, the suggested solution chooses to abandon timely average results in favor of memory consumption and to produce precise results once an hour.

For this purpose, the solutions compute the 1 hour average over a tumbling window. This is a specialization of a sliding window with a sliding interval equal to the window size and ensures that adjacent windows do not contain overlapping elements. However, this also

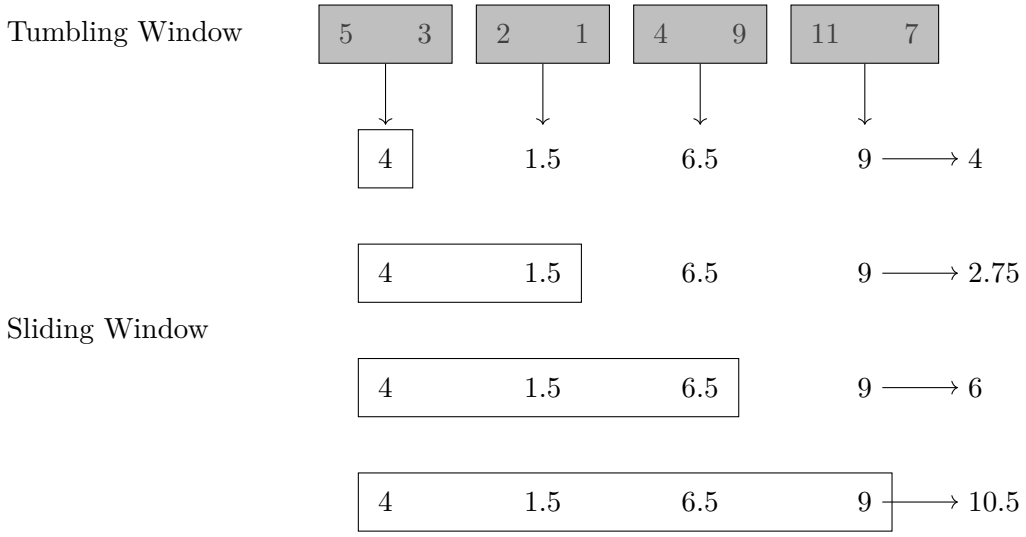


Figure 8.3: Sliding window over tumbling window results for reduced memory average computation

means that a tumbling window is not continuously evaluated but only once new elements exceed its bounds. For example, a 1 hour tumbling window starting at time 01:00 is only evaluated once an element with timestamp greater or equal to 02:00 is encountered. The 1 day and 1 week average aggregations are then performed using a sliding window with the respective duration over the 1 hour tumbling window result stream. Since the tumbling window only emits one new element per hour, the amount of stream data that needs to be buffered in the successive sliding window is negligible. This approach is also illustrated in Figure 8.3. When the first tumbling window result is ready, the sliding window begins and emits the first result equal to the tumbling window result. Once the next tumbling window is evaluated, the sliding window produces a combined output.

The solutions to this problem are described in more detail in Sections 9.3 and 10.3 for Apache Storm and Apache Spark Streaming, respectively.

CHAPTER 9

Apache Storm

Storm is a distributed stream data processing system that was initially created by Nathan Marz at BackType. After Twitter acquired BackType in 2011, Storm was further developed and put to use within Twitter. Finally, Storm became an Apache top-level project in 2014 [TTS+14] [stoa].

The top-level abstraction in Storm are *topologies*. A topology is a directed graph representing the flow of stream tuples. Edges stand for data flow and vertices represent one out of two types of components: *spouts* and *bolts*. Spouts serve as stream sources and typically pull data from third party queuing software like Apache Kafka [1]. On the other hand, bolts perform the actual processing of stream tuples and may produce new stream elements as output. One can think of a topology as an execution plan for a continuous query where the bolts represent query operators [TTS+14].

Figure 9.1 contains a view on the high level system architecture of Storm. Topologies are executed on a cluster of *worker nodes* running *worker processes*. Each worker process represents a JVM instance that runs one or more executors, each executing a set of *tasks* that represent a single component, i.e. a spout or a bolt. There is a single master node, called *Nimbus*, which is comparable to the JobTracker in Hadoop as it distributes and coordinates the execution of topologies on the available cluster nodes. It creates at most one worker process per node and topology and each worker process only executes tasks for the same topology. Apart from the executors, a worker process runs a *worker receive thread* and a *worker send thread* that handle the receiving of all ingoing tuples destined for any of the tasks executed in the worker process and the sending of all outgoing tuples produced by those tasks.

Each worker node runs a *supervisor* program that acts as an agent for Nimbus. The supervisor uses a timer to periodically schedule events that trigger three types of synchronization actions. Regular *heartbeats* are sent to the Nimbus to signal the liveness of the

[1] `https://kafka.apache.org/`

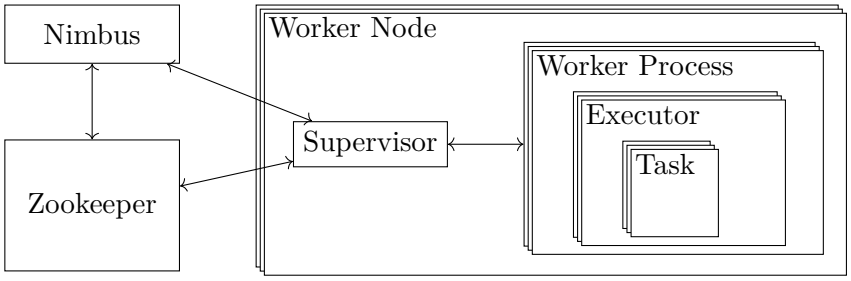


Figure 9.1: Storm system architecture

worker node. Upon a *synchronize supervisor* event, any assignment changes for existing topologies or the creation of new topologies is handled. The *synchronize process* event triggers the health status check of worker processes running on the same node [TTS+14].

Spouts or bolts are instantiated as sets of tasks running in executors across a cluster. An executor corresponds to a single JVM thread and only runs tasks of the same component. The number of tasks created for individual components can be defined by the user and never changes throughout the lifetime of a component. A component's parallelism is adjustable at runtime by altering the number of executors per component. The message flow inside worker processes is illustrated in Figure 9.2. Each executor has dedicated in and out queues assigned. The receive thread of a worker process inserts arriving tuples into the appropriate in queue of the executor running component targeted by the tuple. Tasks consume tuples from this queue and insert any outgoing tuple into the out queue of their containing executor. Apart from the user logic thread that executes component logic, each executor runs a separate send thread that consumes tuples from the out queue and moves them to a global transfer queue that summons the outgoing tuples produced by all executors of a worker process. The worker send thread is responsible for consuming the contents of the global transfer queue and propagates the tuples to their destination worker process [stof].

As detailed above, stream data is shuffled from producer spouts/bolts to consumer bolts. Data shuffling follows a specific partitioning strategy that is customizable by that user. However, Storm pre-defines the following strategies [TTS+14]:

- Shuffle - partitions stream tuples randomly
- Fields - partitions stream tuples based on a set of tuple fields
- All - replicates all stream tuples to all consumer tasks
- Global - sends all stream tuples to a single consumer task
- Local - sends tuples to the consumer task running in the same executor

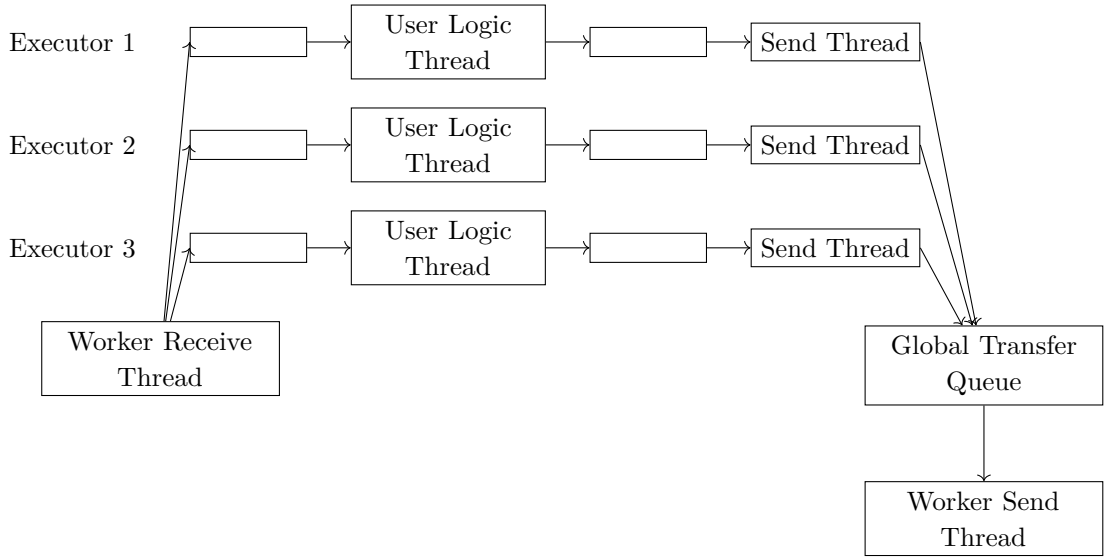


Figure 9.2: Stream tuple flow inside worker processes

Users submit storm topologies to the Nimbus for deployment on the Storm cluster. A topology is represented as Apache Thrift [2] object and can thus be created with any programming language [TTS+14].

9.1 Tuple processing guarantees

Storm basically supports two types of tuple processing semantics: at-least-once and at-most-once. At-least-once semantics ensures that each stream tuple is processed at least once which implies that a tuple might be processed multiple times. In contrast, at-most-once semantics makes no guarantees that a tuple is processed at all but it certainly is not getting processed multiple times. User applications that use Storm output as well as Storm component implementations need to be designed with the concrete processing semantics in mind [TTS+14].

Storm relies on the delivery semantics provided by transactional spout stream sources to provide the mentioned processing guarantees. To enforce at-least-once semantics, Storm must ensure that each incoming tuple is fully processed, i.e. the initial tuple itself, also called spout tuple, as well as any intermediary tuples created from the spout tuple have to be processed. In principle, this involves the tracking of any derived tuple and its lineage to the spout tuple as it traverses the topology. Whenever a tuple leaves the topology a backflow mechanism would need to acknowledge the processing of lineage tuples. This leads to the spout tuple being acknowledged last at which point an acknowledge must be sent to the transactional stream source to prevent the tuple from being resent. Figure 9.3 illustrates this idea. Since tuple B and C have been acknowledged, their common

[2] `https://thrift.apache.org/`

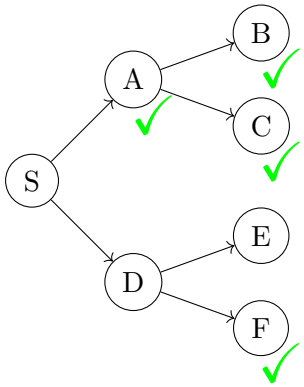


Figure 9.3: Backflow mechanism for acknowledging tuples, S is the spout tuple

ancestor A is acknowledged as well. On the other hand, tuple D cannot be acknowledged because an acknowledgment for tuple E is outstanding. Considering that every spout tuple might trigger the creation of thousands of derived tuples, keeping track of each tuple's lineage would become very memory intensive. This is the point where one of the major breakthroughs introduced by Storm comes into play.

A storm topology is augmented with a set of separate *Acker* tasks that track each spout tuple and notify the originating spout task when a tuple completion is registered. Storm assigns a random 64-bit ID to each new tuple, whether it is originating from the spout or created in a bolt. When a spout task emits a tuple it determines a responsible Acker task based on mod hashing of the tuple ID. It then sends a message to the Acker task containing the spout task ID and the tuple ID. The Acker task maintains a map data structure that maps a spout tuple ID to the originating spout task ID and a corresponding *ack value* of small constant size that is used to keep track of the state of the tuple tree irrespective of its size. The ack value is the result of an XOR operation over all tuple IDs that have been created or acknowledged so far and it is initialized with 0. Each tuple ID is included twice in the XOR aggregation, once at the time of creation and once again when it is acknowledged. This results in the ack value eventually becoming 0 once all tuples have been acknowledged. Due to the random selection of tuple IDs from the ID space, there is a tiny chance of the ack value becoming 0 before all tuples have been acknowledged. Example 8 illustrates the algorithm. In practice, this error rate is negligible since it statistically only happens after 50 million years of operation with 10.000 tuple acknowledgments per second. Moreover, such an error causes harm when the processing of the respective spout tuple fails because in this case, no resubmission of the tuple would take place [TTS+14] [stod].

Example 8 This example illustrates the application of XOR on the ack value in Storm Acker tasks to track the acknowledgment status of tuple trees. Assume a tuple tree as shown in Figure 9.4 containing the assigned tuple IDs

Event	XOR operation	New ack value
Initialization		0000
NEW(S)	$0000 \oplus 1110$	1110
NEW(A)	$1110 \oplus 1011$	0101
NEW(B)	$0101 \oplus 1001$	1100
NEW(C)	$1100 \oplus 0110$	1010
NEW(E)	$1010 \oplus 0100$	1110
NEW(D)	$1110 \oplus 1010$	0100
ACK(C)	$0100 \oplus 0110$	0010
ACK(D)	$0010 \oplus 1010$	1000
ACK(A)	$1000 \oplus 1011$	0011
NEW(F)	$0011 \oplus 0010$	0001
ACK(E)	$0001 \oplus 0100$	0101
ACK(F)	$0101 \oplus 0010$	0111
ACK(B)	$0111 \oplus 1001$	1110
ACK(S)	$1110 \oplus 1110$	0000

Table 9.1: Sequence of tuple creations and acknowledgments and the change of the ack value triggered by them

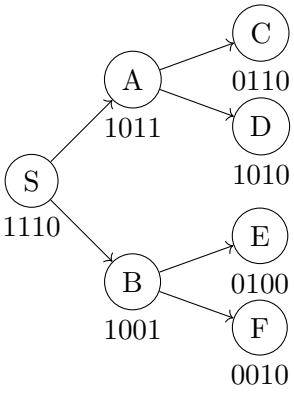


Figure 9.4: Exemplary tuple tree with tuple IDs

and, for the sake of compactness, further assume that the tuple ID space is shrunk to 4 bits instead of 64 bits.

Table 9.1 contains an arbitrary sequence of tuple creation and acknowledgment events in the example tuple tree and it describes the triggered updates to the ack value. Note that the ack value is indeed reduced to zero after the final acknowledgment of spout tuple S.

When disabling the acknowledgment mechanism described above, Storm can only guarantee at-most-once semantics by immediately sending an acknowledgment message to the

stream source when a tuple is received in the spout. This way it is guaranteed that the same tuple is not retransmitted by the stream source. When an error happens during tuple processing in the Storm topology, the tuple is lost [TTS+14].

A Storm extension with a more high-level API called Trident exists that also supports exactly-once semantics. With Trident, stream tuples are not processed one-by-one but instead, small batches of stream elements called *transactions* are logically processed at once. Trident provides a batch processing interface similar to Apache Spark to express batch computations that are applied to each transaction. To enable aggregates across transactions, Trident allows the persistence of state and it guarantees exactly-once semantics for updates to this state. It does so by issuing a unique transaction ID for each transaction and by storing the transaction ID in the state along with the state updates. This allows Trident to prevent duplicate updates by checking for existing transaction IDs in the state. However, both a transactional store for the state and a transactional spout is required to facilitate exactly-once semantics [stoe].

Note that the exactly-once semantics provided by Trident has a different meaning than the tuple processing guarantees provided by Storm as the exactly-once guarantee only relates to state updates.

9.2 Fault tolerance

There are a number of failure modes that can impact the functionality of a Storm cluster. However, Storm is designed to resist and isolate such failures in order to minimize the impact on ongoing operations.

When a worker process terminates unexpectedly, the supervisor process running on the same node restarts the failed worker process. Moreover, a worker process regularly sends heartbeat messages to the Nimbus to signal its liveness. Thus, when a worker process fails to be restarted, the Nimbus notices a timeout and reassigns the worker process to a different node. This mechanism also applies when a whole worker node fails. In this case, all worker processes that were running on the failed node are reassigned [stob].

In order to cater for failures in the supervisors or in the Nimbus, a continuous process monitoring needs to be setup for these components that is responsible for immediately restarting a process if it terminates. All cluster state is stored in a Zookeeper cluster or on disk so the supervisor and the Nimbus are practically stateless services. Hence, no state needs to be recovered on failure. Crucially, a failure of the Nimbus does *not* impact running topologies. This is different from Classic Hadoop, for example, where a failing JobTracker causes all currently running Hadoop jobs to be terminated. However, in absence of a running Nimbus process, no new topologies can be submitted and worker node failures are not handled any more [stob].

9.3 The aggregation of streaming air quality data with Apache Storm

This section describes a solution to the problem statement from Section 8.6 using Apache Storm.

For producing stream data, the solution relies on the spout implementation for producing randomized tuples of the form $\langle \text{indicator}, \text{region}, \text{value}, \text{timestamp} \rangle$. In reality, the spout would pull these tuples from an external data source but this does not impact the general function and applicability of the presented solution. A spout is defined by implementing the `org.apache.storm.spout.ISpout` interface or one of its supplied subtypes. The essential part of the spout implementation is the `nextTuple` method as shown in listing 9.1 that is called by Storm and produces the next tuple in the stream.

```
public void nextTuple() {
    String region = regions.get(ThreadLocalRandom.current().nextInt(regions
        ↪ .size()));
    int indicatorIdx = ThreadLocalRandom.current().nextInt(
        ↪ airQualityIndicatorNames.size());
    String indicatorName = airQualityIndicatorNames.get(indicatorIdx);
    Integer indicatorMaxVal = airQualityIndicators.get(indicatorName);
    Double nextVal = ThreadLocalRandom.current().nextDouble(0,
        ↪ indicatorMaxVal + 1);
    Values tuple = new Values(
            indicatorName,
            region,
            nextVal,
            Instant.now().toEpochMilli()
    );
    collector.emit(tuple);
    try {
        Thread.sleep(250);
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
        throw new RuntimeException(e);
    }
}
```

Listing 9.1: nextTuple method of the RandomAirQualitySpout producing random air quality tuples

The `nextTuple` method depends on the following external parameters:

- `regions` - a set of region name strings
- `airQualityIndicatorNames` - a set of air quality indicator name strings, e.g. CO2, pesticides
- `airQualityIndicators` - maps air quality indicator names to a maximum value for the respective indicator

The method starts by randomly choosing a region and an indicator name. It then retrieves the value range for the indicator by using the selected indicator name to access the `airQualityIndicators` map. After that, a random value in the range $[0, \text{maxVal}]$ is generated and the tuple instance is constructed. Finally, the tuple is emitted using `collector.emit` and the spout thread is sent to sleep for a while in order to reasonably limit the tuple production rate.

Figure 9.5 illustrates the Storm topology of the solution. The print bolt on the right side simply logs the incoming tuples to the console. In a real application scenario, the print bolt may be replaced by a bolt that stores tuples to a database or supplies it to a web service. Similarly, the diagram step bolt just logs the steps to the console that would otherwise be written to an external data store.

The topology design follows the approach of reducing memory consumption for tuple buffering described in Section 8.6. Consequently, the bolts for min/max aggregation over 1 day and 1 week do not directly consume the tuple stream from the spout but instead, the elements are received from the 1 hour aggregation bolt and the 1 day aggregation bolt, respectively. Averages for longer durations such as a day or a week are generated using bolts that perform sliding average aggregation over a stream produced by a of 1 hour tumbling window average bolt. This results in less timely but precise long-term averages.

There are different types of bolt and each type provides a separate interface to implement for creating a custom bolt. For example, a windowed bolt like the maximum aggregation bolt in this example needs to implement the `org.apache.storm.topology.IWindowedBolt` interface. Essentially, this interface prescribes an `execute` method that gets a `org.apache.storm.windowing.TupleWindow` as parameter, representing the tuples of the window that should be evaluated by the method. Apart from the `execute` method there are the `prepare` and `declareOutputFields` methods that need to be implemented. The prepare method is invoked between the initial creation of the bolt and the first invocation of the `execute` method and is designated to carry out initialization tasks. The `declareOutputFields` method is responsible for declaring the schema of the tuples that are produced by the bolt.

Listings 9.2 and 9.3 show the implementation of the maximum bolt - the minimum bolt is analogous. Some of the details that `MinBolt`, `MaxBolt` and `AvgBolt` have in common, are moved to the common supertype `AbstractWindowAggregateBolt` in Listing 9.2 to prevent code duplication. The only thing the `MaxBolt` effectively does is to iterate over the tuple window and to emit the tuple with the maximum value. In general, Storm allows the windowing details like duration and sliding interval to be configured independent of the bolt implementation, hence the same maximum bolt can be used for all time ranges in this example.

```
1 public abstract class AbstractWindowAggregateBolt extends BaseWindowedBolt
    ↪ implements WindowAggregateBolt {
2     private final Fields fields;
3     private OutputCollector collector;
```

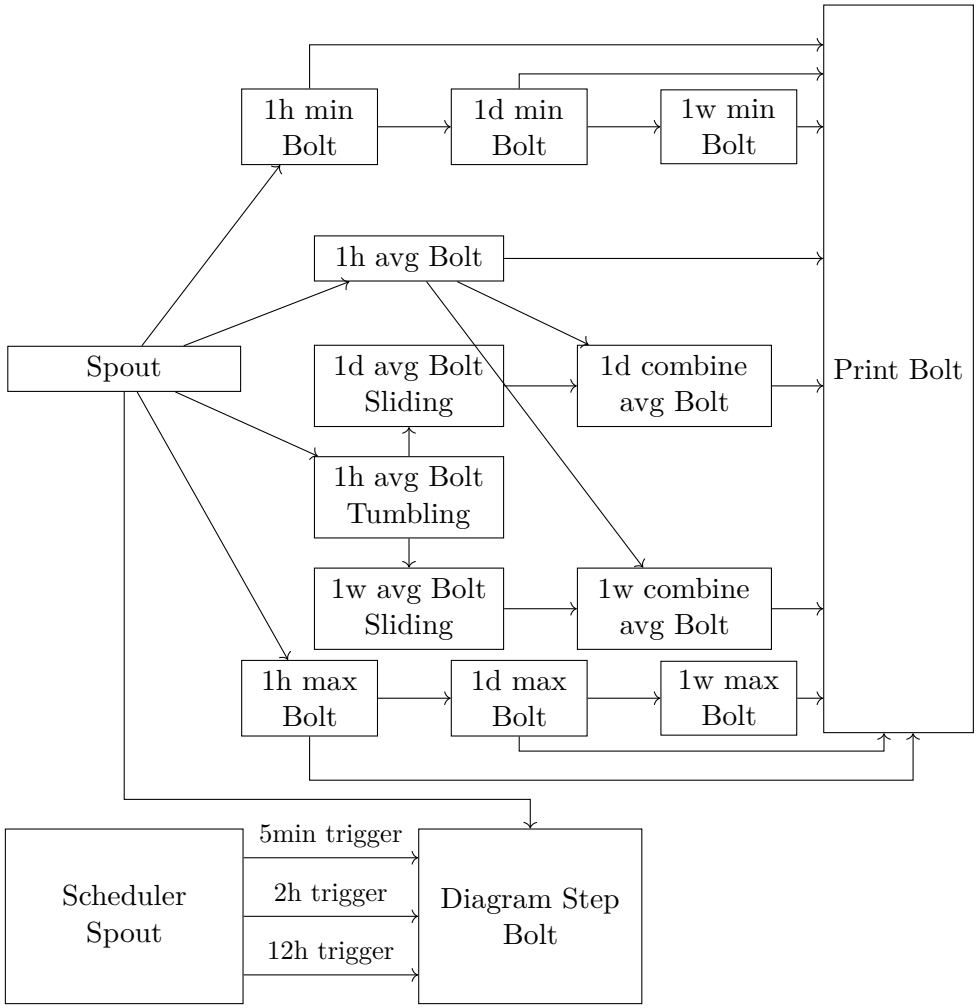


Figure 9.5: Storm topology for generating continuous air quality statistics

```
4
5       public AbstractWindowAggregateBolt(Fields fields) {
6           this.fields = fields;
7       }
8
9       @Override
10      public void prepare(Map stormConf, TopologyContext context,
            ↪ OutputCollector collector) {
11          this.collector = collector;
12      }
13
14      @Override
15      public void execute(TupleWindow inputWindow) {
16          List<Object> aggregate = aggregate(inputWindow.get());
17          if (aggregate != null) {
```

```
18                 collector.emit(aggregate);
19             }
20         }
21
22         @Override
23         public void declareOutputFields(OutputFieldsDeclarer declarer) {
24             declarer.declare(fields);
25         }
26     }
```

Listing 9.2: AbstractWindowAggregateBolt

```
1  public class MaxBolt extends AbstractWindowAggregateBolt {
2      private final String numberField;
3
4      public MaxBolt(Fields fields, String numberField) {
5          super(fields);
6          this.numberField = numberField;
7      }
8
9      @Override
10     public List<Object> aggregate(List<Tuple> tuples) {
11         Tuple max = tuples.stream().max(Comparator.comparingDouble(t -> t.
               ↪ getDoubleByField(numberField))).orElse(null);
12         return max == null ? null : max.getValues();
13     }
14 }
```

Listing 9.3: MaxBolt

The `AbstractWindowAggregateBolt` class just provides default implementations of common methods that would otherwise need to be duplicated in every concrete aggregation bolt. On line 16 it passes the contents of the tuple window to the abstract `aggregate` method that is provided by the `WindowAggregateBolt` interface which the `AbstractWindowAggregateBolt` implements. Concrete bolts extending the `AbstractWindowAggregateBolt` define the aggregation semantics by implementing this method. The `MaxBolt` implementation shown in Listing 9.3 does exactly this. On line 10 it provides an implementation of the `aggregate` method that just returns the tuple with the maximum field value.

Section 8.6 mentions that change detection is crucial for preventing unnecessary buffering of repeated tuples in downstream aggregator units. In the Storm implementation of this example, change detection is performed transparently in a custom `OutputCollector` implementation.

Both the sliding and the tumbling average window use the same average bolt implementation shown in Listing 9.4 that is very similar to the maximum bolt from Listing 9.3. However, the implementation is not as compact as for the `MinBolt` and `MaxBolt` because it is insufficient to just emit one of the existing tuples in the input window. Instead, it is required to create a new tuple with the calculated window average. In this

implementation, the first tuple in the window is chosen as a template for constructing the new tuple. The value of the new tuple is set to the computed window average and all remaining fields are copied from the template.

```
public List<Object> aggregate(List<Tuple> tuples) {
    if (tuples.isEmpty()) {
        return null;
    }
    Tuple firstTuple = tuples.get(0);
    Double avg = tuples.stream().mapToDouble(t -> t.getDoubleByField(
        ↪ numberField)).average().orElse(0);
    List<Object> tuple = new ArrayList<>(fields.size());
    for (String field : fields) {
        if (field.equals(numberField)) {
            tuple.add(avg);
        } else {
            tuple.add(firstTuple.getValueByField(field));
        }
    }
    return tuple;
}
```

Listing 9.4: AvgBolt

For generating diagram steps, no window-based aggregation is required. The only requirement is to store snapshots of the current spout stream at regular intervals, e.g. every 5 minutes. Since the execution of some action on a stream in regular intervals is a often recurring requirement for Storm applications, Storm provides a feature called *tick tuples* for this purpose [stoc]. This allows a bolt to receive special tuples in configurable intervals from the Storm runtime which it can use to trigger custom actions. However, the problem with this approach is that the exact point in time when a snapshot is taken depends on the start time of the scheduler that produces the tick tuples. This prevents the generation of consistent diagram steps with fixed interval starting points, e.g. for every full hour.

For this reason, the presented solution takes a different approach that employs a separate spout to produce custom scheduling tuples. The tuple generation in the spout is triggered via a Quartz [3] scheduler job as shown in Listing 9.5. The scheduler job uses a shared queue to communicate with the spout. The spout itself is periodically invoked by the Storm runtime and polls the queue for new trigger elements as shown in Listing 9.6. When a new element is encountered, the spout emits a respective tuple to its output stream. Otherwise, the spout thread is sent to sleep for a while to not waste processor resources. Note that the Storm interface forbids the spout to access the queue in a blocking way.

The output of the Quartz spout is consumed by a bolt that is responsible for storing the diagram steps. Additionally, this bolt also consumes the air quality spout stream. Listing

[3] http://www.quartz-scheduler.org/

9.7 shows the implementation of the diagram step bolt. It stores the latest air quality tuples for each tuple group in lines 13 - 14 and upon the receipt of a scheduling tuple, it imitates storing the latest tuple of every group by writing a message to the console in lines 3 - 11. In a real application, this part can be replaced by storing the values to a database, for example.

```
public static class SchedulerJob implements Job {

    @Override
    public void execute(JobExecutionContext jobExecutionContext) throws
        ↪ JobExecutionException {
        Queue<Instant> queue = (Queue<Instant>) jobExecutionContext.
            ↪ getJobDetail().getJobDataMap().get("queue");
        queue.offer(jobExecutionContext.getFireTime().toInstant());
    }

}
```

Listing 9.5: SchedulerJob

```
public void nextTuple() {
    Instant trigger = triggers.poll();
    if (trigger == null) {
        try {
            Thread.sleep(500);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            throw new RuntimeException(e);
        }
    } else {
        collector.emit(new Values(triggerName, trigger.getEpochSecond() *
            ↪ 1000));
    }
}
```

Listing 9.6: QuartzSpout

```
1  public void execute(Tuple input, BasicOutputCollector collector) {
2      if (isTriggerTuple(input)) {
3          String triggerName = input.getStringByField("trigger");
4          Instant triggerTime = Instant.ofEpochMilli(input.getLongByField("
               ↪ timestamp"));
5          StringBuilder sb = new StringBuilder("\n");
6          for (Tuple storedTuple : currentValues.values()) {
7              sb.append('\t');
8              sb.append(storedTuple);
9              sb.append('\n');
10         }
11         LOG.info("Store␣tuples␣for␣trigger␣" + triggerName + "/" +
               ↪ triggerTime + ":␣" + sb);
12     } else {
13         List<Object> groupValues = input.select(new Fields("field", "region
               ↪ "));
```

```
14            currentValues.put(groupValues, input);
15        }
16    }
```

Listing 9.7: DiagramStepBolt

Finally, this section discusses in excerpts how to construct the storm topology illustrated in Figure 9.5 to configure and wire the various components discussed so far. Listing 9.8 shows the registration of the air quality stream spout and the subsequent attachment of the bolt that computes a 1 hour minimum sliding window. The desired window is configured via the call to `withWindow` in line 7. It is necessary to designate the tuple field to use as a timestamp for time based window processing which is done by the call to `withTimestamp` in line 8. Finally, the `fieldsGrouping` in line 9 method performs the wiring with the air quality spout identified by `spoutId`.

fieldsGrouping is one of the shuffling mechanisms supported by Storm that defines how tuples output by one component shall be distributed to the consuming components. Recall that a Storm component, i.e. a spout or a bolt, may consist of multiple tasks that are potentially executed in parallel. Hence, there are different ways of how to distribute a single incoming element stream to this set of tasks. The *fieldsGrouping* strategy allows to specify a set of grouping fields. This guarantees that all tuples with the same grouping field values end up at the same task which is crucial for the correctness of the custom window grouping carried out inside the aggregation bolts.

```
1  // hour min
2  topologyBuilder.setBolt(hourMinBoltId, new GroupedWindowAggregatorBolt(
3              groupFields,
4              (Serializable & Supplier<WindowAggregateBolt>) () -> new
                   ↪ ChangeDetectingWindowAggregateBoltWrapper(new MinBolt(
                   ↪ RandomAirQualitySpout.AIR_QUALITY_TUPLE_FIELDS, "value")
                   ↪ , RandomAirQualitySpout.AIR_QUALITY_TUPLE_FIELDS.
                   ↪ fieldIndex("value"))
5          ).withWindow(BaseWindowedBolt.Duration.hours(1), BaseWindowedBolt.
               ↪ Duration.seconds(10))
6          .withTimestampField("timestamp"))
7          .fieldsGrouping(spoutId, new Fields("field", "region"));
8  // hour max
9  topologyBuilder.setBolt(hourMaxBoltId, new GroupedWindowAggregatorBolt(
```

Listing 9.8: Air quality spout wiring

Listing 9.9 shows the definition of the scheduling spouts and the wiring with the diagram step generation bolt. In total, three scheduling spouts are defined for the three different intervals that require diagram step generation.

```
1
2          topologyBuilder.setBolt(printerBoltId, new PrinterBolt())
3                  .shuffleGrouping(hourMinBoltId)
4                  .shuffleGrouping(hourMaxBoltId)
5                  .shuffleGrouping(hourAvgBoltId)
```

```
6                .shuffleGrouping(dayMinBoltId)
7                .shuffleGrouping(dayMaxBoltId)
8                .shuffleGrouping(dayAvgBoltId)
9                .shuffleGrouping(weekMinBoltId)
10               .shuffleGrouping(weekMaxBoltId)
```

Listing 9.9: Wiring of scheduling spouts with the diagram step generation bolt

Lines 7 - 10 connect the diagram step bolt with the spout stream and the different quartz spout trigger streams. Note the usage of the *allGrouping* shuffling strategy for distributing the trigger streams. This ensures that every trigger element is sent to every task of the consuming bolt.

CHAPTER 10

Apache Spark Streaming

Spark Streaming is an add-on to the Spark batch processing engine that reuses Spark's core abstraction of RDDs for fault-tolerant distributed stream processing. It introduces the novel concept of *discretized streams* (DStreams) that are internally represented as a set of RDDs and therefore inherit the fault tolerance properties that come with RDDs. However, this design restrains Spark's ability to only perform batched stream processing in contrast to frameworks like Apache Storm that support both a batched model via its Trident API and record-at-a-time processing of streams.

This chapter first discusses DStreams in more detail before covering the extensions to Spark's system architecture to enable stream processing. Finally, this chapter covers the fault-tolerance and recovery strategies used in Spark Streaming.

10.1 DStreams

The key idea behind the concept of DStreams is the unification of the business logic of stream processing and batch processing jobs. By treating a stream as a sequence of micro batches, the same computational model applies for stream processing as for batch processing. This allows the combination of DStreams with static RDDs holding historical data, for example. Moreover DStream applications can be executed on stored data without any change which alleviates the traditional burden of duplicating common business logic when working on both real time stream data and historical data in a batch processing fashion.

Another important conceptual benefit of DStreams is the fast fault-recovery that they facilitate as described in Section 10.2.

Figure 10.1 provides a high level overview of the Spark Streaming system. Incoming streams are divided into batches and saved as RDDs. The user defines stream processing programs on top of the DStream abstraction. The resulting computation graph is

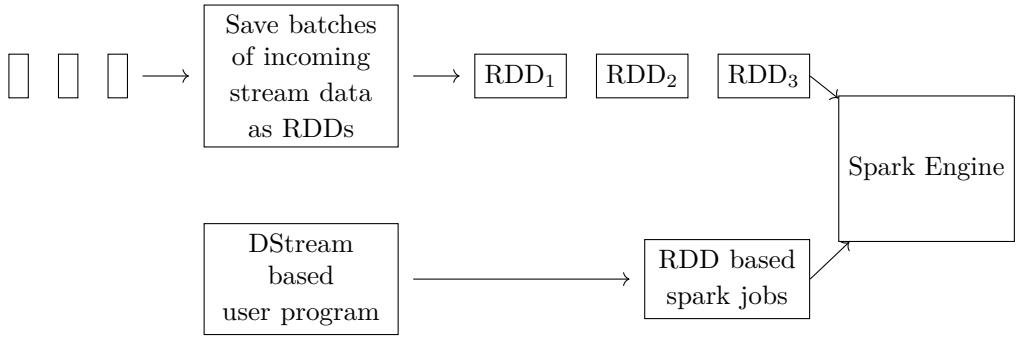


Figure 10.1: High level view of the Spark Streaming system

transformed to RDD level transformation and executed by the Spark batch processing engine on the batch RDDs representing the stream. Each transformation can output a new RDD and the stream of output RDDs can again be treated as a DStream [ZDL+13]. Thus, DStreams represent an execution strategy for RDD based batch operations rather than a completely new technology.

While stream processing systems such as Apache Storm allow the the use of stream supplied, explicit timestamps for tuple ordering, Spark Streaming always uses the arrival time of a tuple for placement in the produced batch datasets. Likewise, Spark currently does not provide any means to perform sliding window computations based on explicit timestamps [ZDL+13]. This can pose challenges to applications that need intend to use Spark Streaming but need to base their computations on explicit timestamps and rely on exact window boundaries. For example, consider an application that simply counts the number of incoming stream tuples over a 1 hour tumbling window and assume the current window to range from 01:00 to 02:00. Since the window is based on tuple arrival time it may potentially includes a late tuple with timestamp 00:59:55 but arrival time 01:00:05 from the previous window. Likewise, if the clock of the stream source happens to be out of synchronization with the system running Spark it may happen that tuples with timestamp 00:02:05 arrive at system time 00:01:59:55 and would therefore also be illegally included in the 01:00 to 02:00 window. For a large fraction of real world applications, such a minor imprecision is certainly negligible but it still needs to be factored in. On the one hand, the authors of Spark argue that techniques for order-independent processing can be applied on the application level to ensure the correctness of results if required [KFD+10]. On the other hand, the lack of support for explicit timestamp based processing also effectively reduces the complexity and increases the efficiency of Spark as it does not need to cope with late tuple arrivals.

The DStream API supports standard stateless batch operations such as *map*, *reduce*, *group by* and *join* but it is enriched with stateful operations specific to stream processing such as windows and incremental aggregation. Incremental aggregation can be used to perform efficient reduction over sliding windows for invertible aggregate functions. For example, when summing up values over a sliding window, instead of recomputing the

sum for all window tuples again every time the window slides, it is a better approach to add new elements entering the window and deduct old elements falling out of the window [ZDL+13]. Other stateful operations such as *mapWithState* or *updateStateByKey* allow the user to specify stateful stream operations with custom state that is transparently managed and distributed by Spark in a fault-tolerant way using RDDs [spae]. Finally, DStreams support output operations to write DStream RDDs to external systems like databases or file systems [ZDL+13].

Since Spark manages any state related to the stream processing operations transparently, the task scheduler has greater freedom when assigning tasks to executors. For example, when scheduling operations on a partitioned stream there is not strict requirement for the same partition to always be processed on the same executor. Tasks for the same partitions in different batches may be assigned to different executors on potentially different cluster nodes which can be used for dynamic load balancing of unevenly partitioned streams. To further illustrate this, consider the air quality statistics problem described in Section 8.6. Assume an air quality tuple stream that is partitioned by region. When the number of tuples for region A is much larger than for region B, the processing of the region A partition is going to take longer than for region B unless more executors are utilized for scheduling the processing tasks for region A [spad]. This is only possible when the state used by the stream operations can be made available on multiple nodes in a way that is transparent to the user. In contrast, the effect of partitioning a stream in Apache Storm is that each partition is processed by the same task on the same node hence intra-partition load balancing is not possible.

Another advantage with regard to skewed workloads resulting from the discretized processing of streams is the clear consistency semantics that Spark provides. When processing a partitioned streams on a record-at-a-time basis it is possible that particular partitions fall behind because they encounter more load than others. With each partition producing output independently, inconsistencies may arise when viewing the stream of results that have been produced up to a single point in time. For instance, when counting the number of events received from region A and region B, the count for region B may be inconsistent with the count of region A because the latter one lags behind. In order to avoid this problem in record-at-a-time stream processing systems it is necessary to introduce task synchronization mechanisms. With Spark's discretized stream model there is no need for synchronization because every output RDD reflects all the tuples contained in the input RDD regardless of the partitioning. Hence, the computational model itself prevents inconsistencies among partitions and the semantics are clear [ZDL+13].

Spark Streaming builds on the Spark batch processing engine but it introduces certain extensions to allow stream processing.

As mentioned before Spark Streaming transforms input streams into DStreams by batching incoming tuples over a constant time interval. Because stream input can be received at multiple worker nodes it is necessary to communicate the contents of the interval or timestep to the master node so that it can schedule the required tasks. For this purpose, Spark Streaming assumes the system clocks on worker nodes to be synchronized, e.g. via

the network time protocol (NTP). As described in Section 4.3, the Spark worker nodes store data in the form of blocks either in-memory or spilled to disk. At the end of an interval, each worker node sends the identifiers of the blocks received in the last interval to the master node [ZDL+13].

10.2 Fault-tolerance

For providing fault-tolerance, Spark relies on the source data for the RDD lineage graph to be available at any time in case RDD partitions need to be recomputed. While this is a reasonable assumption for static data stored in fault-tolerant storages like HDFS it does not generally apply in the case of input streams that often originate directly from unreliable clients. Thus, Spark Streaming replicates received data across multiple nodes before acknowledging the receive to the client. Alternatively, Spark Streaming can periodically load new data from reliable external storage systems such as HDFS in which case no replication is necessary [ZDL+13].

An input receiver in Spark Streaming can either be reliable or unreliable. While a reliable receiver acknowledges the reception and replication of stream data in Spark Streaming to the input source, an unreliable receiver performs no acknowledgment. Reliable receivers allow the guarantee of certain delivery and processing guarantees for stream data depending on their implementation and on the reliability of the input source [spaf].

When a worker node fails, Spark Streaming allows the recovery of state RDD partitions and all tasks that were running on the failed node by recomputing them in parallel on healthy nodes. Checkpoints of state RDDs are periodically stored to cut off the lineage graph and to prevent long dependency chains which would lead to long recovery times. During re-computation of lost RDD partitions, Spark Streaming exploits parallelism across partitions and also across timesteps for operations that do not depend on the results of previous timesteps. The high degree of parallelization is important to utilize a large number of cluster nodes for recovery. This is especially critical in the context of stream processing systems as new data continues to arrive at worker nodes even during the recovery phase hence the processing throughput of the system should be large enough to allow it to perform the recovery along with processing the current data [ZDL+13].

Spark Streaming also detects and mitigate stragglers, i.e. tasks that run slower than comparable tasks potentially due to node faults, by scheduling speculative backup copies of tasks that can take over the computation [ZDL+13].

Another important requirement of Spark Streaming to be useful as a stream processing system is the recoverability of the master node. To allow this, Spark Streaming stores the state of computation reliably in HDFS at the start of each timestep. This information includes the DStream graph, the driver programs, the time of the last checkpoint and the identifiers of stream data RDDs that were received since the last checkpoint. On recovery, the new master reads this data, reconnects to the worker nodes and registers

the partitions that are currently held by each worker. After that, the master schedules the processing of the timesteps that were missed during recovery [ZDL+13].

10.3 The aggregation of streaming air quality data with Apache Spark Streaming

This section describes a solution to the problem statement from 8.6 using Apache Spark Streaming.

For demonstration purposes, stream data is produced by a custom input receiver implementation that produces random air quality tuples in a similar fashion as the implementation of the Apache Storm Spout described in Section 9.3. For implementing an input receiver the generic supertype `org.apache.spark.streaming.receiver.Receiver` needs to be extended. This also involves the specification of a type parameter that designates the tuple types to allow the creation of stream processing driver programs in a type-safe way. This is an important difference to Storm which does not support typed tuples. The receiver implementation needs to define the `onStart` and `onStop` methods that are responsible for starting and stopping the production of stream tuples. However, the actual receipt and emission of stream tuples needs to take place in a separate thread that is managed by the receiver implementation. The `onStart` and `onStop` methods are dedicated for starting and stopping this thread. Listing 10.1 shows the implementation of these methods for the random air quality input receiver.

```
@Override
public void onStart() {
    thread = new Thread(() -> {
        while (!stop) {
            String region = regions.get(ThreadLocalRandom.current().nextInt
                ↪ (regions.size()));
            int indicatorIdx = ThreadLocalRandom.current().nextInt(
                ↪ airQualityIndicators.size());
            String indicatorName = airQualityIndicatorNames.get(
                ↪ indicatorIdx);
            Integer indicatorMaxVal = airQualityIndicators.get(
                ↪ indicatorName);
            Double nextVal = ThreadLocalRandom.current().nextDouble(0,
                ↪ indicatorMaxVal + 1);
            AirQualityTuple tuple = new AirQualityTuple(
                    indicatorName,
                    region,
                    nextVal,
                    Instant.now()
            );
            store(tuple);
            try {
                Thread.sleep(250);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                throw new RuntimeException(e);
```

```
                }
            }
        });
        thread.start();
    }

    @Override
    public void onStop() {
        stop = true;
        try {
            thread.join();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            throw new RuntimeException(e);
        }
    }
```

Listing 10.1: Input receiver for creating a random air quality stream

The `onStart` method simply creates and starts a new receiver thread. The thread consists of a while loop producing a tuple of type `AirQualityTuple` with random air quality indicator, region, value and the current timestamp. Note that Spark Streaming can actually not make any use of this timestamp as detailed in Section 10.1. Once the tuple is constructed, the thread submits it to Spark Streaming via the `store` method that is provided by the `org.apache.spark.streaming.receiver.Receiver` supertype. After producing a tuple, the thread is sent to sleep for a short period of time to limit the tuple production rate. The `onStop` method just sets a flag that causes the while loop in the receiver thread to exit and waits for the thread to end.

Like in ordinary Spark, the driver program that defines the Spark Streaming computation can be a simple Java `main` method. The first thing to do is to set up the spark streaming context that provides the entry points to the D-Stream API. Listing 10.3 shows the corresponding source code fragment.

```
SparkConf conf = new SparkConf().setMaster("local[2]").setAppName("
    ↪ AirQuality Statistics");
JavaStreamingContext ssc = new JavaStreamingContext(conf, Durations.seconds
    ↪ (10));
JavaSparkContext sc = ssc.sparkContext();

ssc.checkpoint(Paths.get(System.getProperty("user.dir"), ".spark").
    ↪ toAbsolutePath().toString());
```

Listing 10.2: Creation of the stream context in Spark Streaming

For creating the context, a configuration object of type `org.apache.spark.SparkConf` is required like in ordinary Spark. The example implementation specifies a Spark configuration with a local master with 2 threads. After the context configuration is created, the program creates a `org.apache.spark.streaming.api.java.JavaStreamingContext` which is the Java pendant to the Scala `org.apache.spark.streaming.StreamingContext`.

Besides the configuration object, it is possible to specify the batch duration that should be used for stream discretization as a second parameter. The choice of this parameter is application dependent and determines the minimum latency between the arrival of a stream tuple and its processing by Spark Streaming. In order to perform stateful stream operations like `mapWithState` it is necessary to also configure the creation of checkpoints for the created stream context. This comes down to specifying a directory for storing the checkpoints as shown in line 31. In a real world scenario the directory should reside on a reliable file system like HDFS.

Once the stream context is available it is possible to create a DStream from the `RandomAirQualityReceiver` as shown in Listing 10.3. This can be done using the `receiverStream` method provided by the stream context that accepts an instance of the `RandomAirQualityReceiver`. When instantiating the receiver it is also required to pass a storage level that indicates how Spark Streaming should store the incoming stream data. The example uses the *MEMORY_AND_DISK_SER* storage level which stores RDDs as serialized objects in memory and spills to disk when no more memory is available. Storing RDDs in serialized form is more space efficient but it also is computationally more expensive. After creating the stream, the checkpoint interval is specified in line 2.

```
1 JavaReceiverInputDStream<AirQualityTuple> stream = ssc.receiverStream(new
    ↪ RandomAirQualityReceiver(StorageLevel.MEMORY_AND_DISK_SER_2(),
    ↪ airQualityIndicators, regions));
2 stream.checkpoint(Durations.minutes(1));
```

Listing 10.3: Creation of a DStream from the input receiver

Once a DStream object is acquired it can be used to define further stream operations using the DStream API. Listing 10.4 shows the definition of the 1 hour minimum stream.

```
1 JavaDStream<AirQualityTuple> hourMinStream = detectChanges(sc,
2         stream
3         .mapToPair(t -> Tuple2.apply(new AirQualityTupleGroup(t.
              ↪ getIndicator(), t.getRegion()), t))
4         .reduceByKeyAndWindow((t1, t2) -> t1.getValue() < t2.getValue() ?
              ↪ t1 : t2, Durations.minutes(60), Durations.seconds(10))
5 );
6 print("1hMin", hourMinStream);
```

Listing 10.4: Definition of the 1 hour minimum stream

The listing is discussed in evaluation order starting at line 2. First, the `mapToPair` operation is performed that extracts a group key out of the stream of `AirQualityTuple` objects and produces a stream of key-value pairs where the key is of type `AirQualityTupleGroup` and the original `AirQualityTuple` objects remains as value. The `AirQualityTupleGroup` ↪ type includes the fields by which the stream should be grouped in the following reduction operations, namely the indicator name and the region. `mapToPair` returns a `org.apache.spark.streaming.api.java.JavaPairDStream` which offers operations that use the key for grouping. In this example, the `reduceByKeyAndWindow` operation

is applied to the pair stream along with a reducer function that specifies how two tuples with the same key should be reduced to form a single output tuple. In this concrete case, the reducer function simply returns the tuple with the smaller value. Moreover, the method allows to specify a window size and a sliding interval which is 60 minutes and 10 seconds, respectively in this example. I.e. the stream tuples of the last 60 minutes take part in the reduction and updated results are produced every 10 seconds. Up to this point no change detection is performed, i.e. the stream would be enriched with a new tuple every 10 seconds regardless of whether the actual minimum has changed or not. For this reason, the custom `detectChanges` method is applied to the resulting stream. It applies further stream operations that handle the change detection as shown in Listing 10.5.

```
1  if (!state.exists() || !state.get().equals(tuple.get())) {
2      state.update(tuple.get());
3      return Optional.of(tuple.get());
4  } else {
5      return Optional.<V> empty();
6  }
7  }).initialState(sc.parallelizePairs(Collections.emptyList()))
8  ).filter(Optional::isPresent)
9  .map(Optional::get);
10 }
11
12 private static void print(String streamName, JavaDStream<?> stream) {
13 stream.foreachRDD(rdd -> {
```

Listing 10.5: Change detection in Spark Streaming

The method applies the `mapWithState` operation to the passed stream which allows to provide a stateful function for mapping tuples. On line 4 this mapping function checks for existing states and state changes and either updates the state and returns the new tuple or returns an *empty* value. The returned values form the new stream that is output by the `mapWithState` method. The subsequent `filter` and `map` operations are used to filter out the *empty* values from the stream and to unwrap the non-empty values.

Listing 10.4 continues by simply printing the stream contents to console. The stream definitions for aggregation over the other time ranges is analogous.

For the computation of diagram steps, a different approach has to be taken compared to the Storm implementation. Because Storm performs record-at-a-time processing the latency between the creation of a trigger event via a Quartz scheduler task and the actual processing of the event is negligibly low. However, in the case of Spark Streaming, such events would be batched over a duration of 10 seconds as configured in the example. Hence a trigger event for the 5 minute diagram step would only be processed up to 10 seconds later. Moreover, it is not enough to have a tumbling window with the desired interval length because the window is started at a random point in time depending on the start of the scheduling time of the stream processing job. E.g. with the window starting at 01:08, the produced diagram step tuples for a 5 minute interval would be

01:13, 01:18 etc instead of 01:10, 01:15 etc. More importantly, Spark Streaming cannot use tuple timestamps for windowing which might further impact the precision of emitted tuples. Thus, a different approach is required in the case of Spark Streaming for timing the emission of diagram step tuples. Listing 10.6 shows the implementation for the 5 minute diagram steps. The implementation for the other intervals is analogous.

```
1 JavaDStream<AirQualityTuple> fiveMinStream = stream
2         .mapToPair(t -> Tuple2.apply(new AirQualityTupleGroup(t.
              ↪ getIndicator(), t.getRegion()), t))
3         .mapWithState(StateSpec.function((AirQualityTupleGroup group,
              ↪ Optional<AirQualityTuple> optionalTuple, State<
              ↪ AirQualityTuple> state) ->
4             updateAirQualityStepState(optionalTuple.get(), state, 1, 10)
5         ).initialState(sc.parallelizePairs(Collections.emptyList())))
6         .filter(Optional::isPresent)
7         .map(Optional::get);
8 print("5min", fiveMinStream);
```

Listing 10.6: 5 minute diagram step generation with Spark Streaming

The stream definition starts by directly consuming the input source stream and by carrying out the same pair mapping as in the 1 hour minimum aggregation described earlier. Then a `mapWithState` operation is applied to the stream with the purpose of ensuring that only such tuples are emitted that correspond to the desired interval borders. As shown in Listing 10.7, the used mapping function uses the modulo operation on the epoch seconds of the tuple timestamp to filter out such tuples that do not fit the desired interval. After the `mapWithState` operation it only remains to filter out empty tuples and unwrap the remaining ones before the stream is printed.

```
1  private static Optional<AirQualityTuple> updateAirQualityStepState(
       ↪ AirQualityTuple tuple, State<AirQualityTuple> state, double
       ↪ secondToUnitFactor, int stepSizeInUnits) {
2      if (!state.exists()) {
3          if (((int) (tuple.getTimestamp().getEpochSecond() *
               ↪ secondToUnitFactor)) % stepSizeInUnits == 0) {
4              state.update(tuple);
5              return Optional.of(tuple);
6          } else {
7              return Optional.empty();
8          }
9      } else {
10         if (state.get().getTimestamp().isBefore(tuple.getTimestamp())) {
11             int stateUnits = (int) (state.get().getTimestamp().
                   ↪ getEpochSecond() * secondToUnitFactor);
12             int tupleUnits = (int) (tuple.getTimestamp().getEpochSecond() *
                   ↪  secondToUnitFactor);
13             if (stateUnits != tupleUnits && tupleUnits % stepSizeInUnits ==
                   ↪  0) {
14                 state.update(tuple);
15                 return Optional.of(tuple);
16             }
```

```
17            }
18            return Optional.empty();
19        }
20    }
```

Listing 10.7: Stateful tuple filtering for diagram step creation

All in all the example implementation with Spark Streaming is much more concise compared to Storm and the different processing models sometimes require different implementations for yielding the same outcome.

CHAPTER 11

Conclusion

Distributed big data processing frameworks have evolved around the skewed ratio between storage capacity and storage access speed. While the capacity of storage mediums like HDDs has experienced a steady growth over the years, companies in the early 2000s increasingly struggled with writing and reading fast enough the ever growing amount of data that today's society produces. Since that time, a plethora of frameworks has been created as a solution to this problem. Their main principle is to scale storage access horizontally by distributing it across large clusters of cost effective commodity hardware.

Early systems like Hadoop were initially targeted to perform offline batch processing tasks. But soon, other types of applications emerged that were covered well by the existing batch processing frameworks. For example, graph algorithms operating on distributed graphs are hard to express using programming models for batch processing. Also, graphs exhibit different locality constraints than other datasets and thus require special treatment with respect to partitioning in order to allow for efficient execution of algorithms. The processing of continuous, high-volume data streams is another important task that inherently differs from traditional batch processing and which is fueled by the trend towards the internet of things.

Programming models that abstract away the fallacies induced by distributed nature of big data frameworks and that facilitate the development of comprehensible algorithms to be executed by such systems play a critical role in their success. With the early introduction of MapReduce and Pregel, Google has taken over a pioneering position in the field of big data processing. Subsequent open source frameworks like Hadoop, Giraph and Spark have adopted these programming models and have opened big data processing to a wider audience.

This thesis has covered the state-of-the-art programming models in the various fields of big data processing and has given a comprehensible introduction to the inner workings of some of today's most popular big data frameworks across the before mentioned

application types. It has further provided the reader with concrete examples of how to use each of the presented technologies in practice by introducing selected problem statements from a realistic application domain and by presenting the corresponding solution implementations.

Many extension products and systems have been developed on top of the basic technologies presented in this thesis that further simplify the development of solutions for certain problems. Since these technologies are out of the scope of this paper it remains to future work to provide insights on them. In conclusion, some of these extensions are pointed out:

SQL is a powerful tool for querying data and many developers are familiar with using it. Extensions to Hadoop and Spark such as *Hive* and *Spark SQL*, respectively, that bring SQL support to these frameworks can therefore be considered an important cornerstone for their wide adoption. While Hive is rather targeted for data warehousing tasks and for issuing ad-hoc queries in a declarative way, *Pig* is another platform targeted towards data preparation that also provides an abstract but more procedural way to express computation tasks in Hadoop.

Other projects aim to improve the performance of applications running on big data frameworks by transparently augmenting lower system layers. For example, *Alluxio* is a tiered storage that sits on top of other storage systems such as HDFS and aims to keep data as close to memory as possible by utilizing a tiered storage architecture.

Based on usage examples presented in this thesis it appears that the Spark suite of tools is simpler to use and allows for a more concise expression of big data processing program logic than it is possible with Hadoop. Also, Spark eases the creation of type-safe processing pipelines which results in more robust and more maintainable programs. While performance benchmarks suggest that Spark performs considerably better than Hadoop for a large set of workloads, a detailed performance analysis of both systems under realistic conditions certainly is an interesting topic for a follow-up work as it helps to establish in-depth understanding of the peculiarities of both frameworks. After all, such understanding is the key to identify and develop approaches to further advance and optimize these systems.

List of Figures

List of Tables

Bibliography

[ABB+04] Arvind Arasu, Brian Babcock, Shivnath Babu, Jon McAlister, and Jennifer Widom. Characterizing memory requirements for queries over continuous data streams. *ACM Transactions on Database Systems (TODS)*, 29(1):162–194, 2004.

[ABC+11] Foto N Afrati, Vinayak Borkar, Michael Carey, Neoklis Polyzotis, and Jeffrey D Ullman. Map-reduce extensions and recursive queries. In *Proceedings of the 14th international conference on extending database technology*, pages 1–8. ACM, 2011.

[ADD+15] Michael Armbrust, Tathagata Das, Aaron Davidson, Ali Ghodsi, Andrew Or, Josh Rosen, Ion Stoica, Patrick Wendell, Reynold Xin, and Matei Zaharia. Scaling spark in the real world: Performance and usability. *PVLDB*, 8(12):1840–1843, 2015.

[AJB00] Réka Albert, Hawoong Jeong, and Albert-László Barabási. Error and attack tolerance of complex networks. *nature*, 406(6794):378–382, 2000.

[AR04] Konstantin Andreev and Harald Räcke. Balanced graph partitioning. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 120–124. ACM, 2004.

[AU10] Foto N Afrati and Jeffrey D Ullman. Optimizing joins in a map-reduce environment. In *Proceedings of the 13th International Conference on Extending Database Technology*, pages 99–110. ACM, 2010.

[AW04] Arvind Arasu and Jennifer Widom. A denotational semantics for continuous queries over streams and relations. *ACM Sigmod Record*, 33(3):6–11, 2004.

[BBD+02] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16. ACM, 2002.

[BBMS08] Magdalena Balazinska, Hari Balakrishnan, Samuel R Madden, and Michael Stonebraker. Fault-tolerance in the borealis distributed stream processing system. *ACM Transactions on Database Systems (TODS)*, 33(1):3, 2008.

[BCR06] Avrim Blum, TH Hubert Chan, and Mugizi Robert Rwebangira. A random-surfer web-graph model. In *2006 Proceedings of the Third Workshop on Analytic Algorithmics and Combinatorics (ANALCO)*, pages 238–246. SIAM, 2006.

[BHBE10] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D Ernst. Haloop: Efficient iterative data processing on large clusters. *Proceedings of the VLDB Endowment*, 3(1-2):285–296, 2010.

[blo] Hadoop - namenode, checkpoint node and backup node. `http://morrisjobke.de/2013/12/11/Hadoop-NameNode-and-siblings/`. Accessed: 2017-03-12.

[BRS92] Rodrigo A Botafogo, Ehud Rivlin, and Ben Shneiderman. Structural analysis of hypertexts: identifying hierarchies and useful metrics. *ACM Transactions on Information Systems (TOIS)*, 10(2):142–180, 1992.

[BSW04] Shivnath Babu, Utkarsh Srivastava, and Jennifer Widom. Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. *ACM Transactions on Database Systems (TODS)*, 29(3):545–580, 2004.

[BW01] Shivnath Babu and Jennifer Widom. Continuous queries over data streams. *ACM Sigmod Record*, 30(3):109–120, 2001.

[BYG+16] Eric Brewer, Lawrence Ying, Lawrence Greenfield, Robert Cypher, and Theodore T'so. Disks for data centers. *White paper for FAST*, 1(1):p4, 2016.

[CÇC+02] Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Monitoring streams: a new class of data management applications. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 215–226. VLDB Endowment, 2002.

[CDG+08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[CEK+15] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proceedings of the VLDB Endowment*, 8(12):1804–1815, 2015.

[Coh09] Jonathan Cohen. Graph twiddling in a mapreduce world. *Computing in Science & Engineering*, 11(4):29–41, 2009.

[DG04] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[gir] Large-scale graph partitioning with apache giraph. https://code.facebook.com/posts/274771932683700/large-scale-graph-partitioning-with-apache-giraph/. Accessed: 2017-03-31.

[GJ02] Michael R Garey and David S Johnson. *Computers and intractability*, volume 29. wh freeman New York, 2002.

[GLG+12] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, volume 12, page 2, 2012.

[graa] Spark graphx edge cut versus vertex cut diagram. http://spark.apache.org/docs/latest/img/edge_cut_vs_vertex_cut.png. Accessed: 2017-04-07.

[grab] Spark graphx rdg layout diagram. https://spark.apache.org/docs/latest/img/vertex_routing_edge_tables.png. Accessed: 2017-04-07.

[GXD+14] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*, volume 14, pages 599–613, 2014.

[hada] Issues with classic mapreduce. https://issues.apache.org/jira/browse/MAPREDUCE-278. Accessed: 2017-03-15.

[hadb] Mapreduce 2.0 - resourcemanager high availability. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html. Accessed: 2017-03-16.

[hadc] Mapreduce 2.0 - resourcemanager high availability. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/ResourceManagerHA.html. Accessed: 2017-03-16.

[hadd] Mapreduce 2.0 proposal. https://issues.apache.org/jira/browse/MAPREDUCE-279. Accessed: 2017-03-15.

[HBR+05] J-H Hwang, Magdalena Balazinska, Alex Rasin, Ugur Cetintemel, Michael Stonebraker, and Stan Zdonik. High-availability algorithms for distributed stream processing. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 779–790. IEEE, 2005.

[HCB96] Jonathan MD Hill, Paul I Crumpton, and David A Burgess. Theory, practice, and a tool for bsp performance prediction. In *European Conference on Parallel Processing*, pages 697–705. Springer, 1996.

[hdfa] Discussion on secondary namenode, checkpointnode and backupnode in hdfs. `https://issues.apache.org/jira/browse/HADOOP-4539`. Accessed: 2017-03-12.

[hdfb] Hdfs architecture. `http://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html`. Accessed: 2017-03-10.

[hdfc] Hdfs federation. `http://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/Federation.html`. Accessed: 2017-03-13.

[hdfd] Hdfs high-availability with quorum journal manager. `http://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HDFSHighAvailabilityWithQJM.html`. Accessed: 2017-03-13.

[hdfe] Hdfs high-availability with shared storage. `http://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HDFSHighAvailabilityWithNFS.html`. Accessed: 2017-03-13.

[hdff] Hdfs user guide. `http://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html`. Accessed: 2017-03-12.

[KFD+10] Sailesh Krishnamurthy, Michael J Franklin, Jeffrey Davis, Daniel Farina, Pasha Golovko, Alan Li, and Neil Thombre. Continuous analytics over discontinuous streams. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 1081–1092. ACM, 2010.

[KTS+11] U Kang, Hanghang Tong, Jimeng Sun, Ching-Yung Lin, and Christos Faloutsos. Gbase: a scalable and general graph management system. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1091–1099. ACM, 2011.

[LRU14] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining of massive datasets*. Cambridge University Press, 2014.

[MAB+10] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[MSL15] Claudio Martella, Roman Shaposhnik, and Dionysios Logothetis. Giraph architecture. In *Practical Graph Analytics with Apache Giraph*, pages 137–162. Springer, 2015.

[mur] The next generation of apache hadoop mapreduce. http://yahoohadoop.tumblr.com/post/98210076241/the-next-generation-of-apache-hadoop-mapreduce. Accessed: 2017-03-15.

[MWA+02] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query processing, approximation, and resource management in a data stream management system, 2002.

[RRH99] Vijayshankar Raman, Bhaskaran Raman, and Joseph M Hellerstein. Online dynamic reordering for interactive data processing. In *VLDB*, volume 99, pages 709–720. Citeseer, 1999.

[SASU13] Anish Das Sarma, Foto N Afrati, Semih Salihoglu, and Jeffrey D Ullman. Upper and lower bounds on the cost of a map-reduce computation. In *Proceedings of the VLDB Endowment*, volume 6, pages 277–288. VLDB Endowment, 2013.

[SKW+14] Yogesh Simmhan, Alok Kumbhare, Charith Wickramaarachchi, Soonil Nagarkar, Santosh Ravi, Cauligi Raghavendra, and Viktor Prasanna. Goffish: A sub-graph centric framework for large-scale graph analytics. In *European Conference on Parallel Processing*, pages 451–462. Springer, 2014.

[spaa] Apache spark cluster managers: Yarn, mesos, or standalone? http://spark.apache.org/docs/2.1.0/cluster-overview.html. Accessed: 2017-03-24.

[spab] Apache spark cluster managers: Yarn, mesos, or standalone? http://www.agildata.com/apache-spark-cluster-managers-yarn-mesos-or-standalone/. Accessed: 2017-03-24.

[spac] Apache spark: core concepts, architecture and internals. http://datastrophic.io/core-concepts-architecture-and-internals-of-apache-spark/. Accessed: 2017-03-24.

[spad] Diving into apache spark streaming's execution model. https://databricks.com/blog/2015/07/30/diving-into-apache-spark-streamings-execution-model.html. Accessed: 2017-05-14.

[spae] Faster stateful stream processing in apache spark streaming. https://databricks.com/blog/2016/02/01/faster-stateful-stream-processing-in-apache-spark-streaming.html. Accessed: 2017-05-14.

[spaf] Input dstreams and receivers. https://spark.apache.org/docs/2.1.1/streaming-programming-guide.html#input-dstreams-and-receivers. Accessed: 2017-05-16.

[spag] Mastering apache spark. https://jaceklaskowski.gitbooks.io/mastering-apache-spark/. Accessed: 2017-03-24.

[spah] Spark architecture. https://0x0fff.com/spark-architecture/. Accessed: 2017-03-24.

[spai] Spark architecture: Shuffle. https://0x0fff.com/spark-architecture-shuffle/. Accessed: 2017-03-24.

[spaj] Spark internals. https://github.com/JerryLead/SparkInternals. Accessed: 2017-03-24.

[spak] Spark memory management. https://0x0fff.com/spark-memory-management/. Accessed: 2017-03-24.

[spal] Spark taskscheduler. https://jaceklaskowski.gitbooks.io/mastering-apache-spark/content/spark-TaskScheduler.html. Accessed: 2017-03-24.

[stoa] History of apache storm and lessons learned. http://nathanmarz.com/blog/history-of-apache-storm-and-lessons-learned.html. Accessed: 2017-04-28.

[stob] Storm fault tolerance. http://storm.apache.org/releases/current/Fault-tolerance.html. Accessed: 2017-05-03.

[stoc] Tick tuples within storm. http://kitmenke.com/blog/2014/08/04/tick-tuples-within-storm/. Accessed: 2017-05-07.

[stod] Trident tutorial. http://storm.apache.org/releases/current/Guaranteeing-message-processing.html. Accessed: 2017-05-02.

[stoe] Trident tutorial. http://storm.apache.org/releases/current/Trident-tutorial.html. Accessed: 2017-05-01.

[stof] Understanding the parallelism of a storm topology. https://storm.apache.org/releases/current/Understanding-the-parallelism-of-a-Storm-topology.html. Accessed: 2017-05-01.

[SW04] Utkarsh Srivastava and Jennifer Widom. Flexible time management in data stream systems. In *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 263–274. ACM, 2004.

[TBC+13] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. From think like a vertex to think like a graph. *Proceedings of the VLDB Endowment*, 7(3):193–204, 2013.

[TÇZ07] Nesime Tatbul, Uğur Çetintemel, and Stan Zdonik. Staying fit: Efficient load shedding techniques for distributed stream processing. In *Proceedings of the 33rd international conference on Very large data bases*, pages 159–170. VLDB Endowment, 2007.

[TMSF02] Pete Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. Enhancing relational operators for querying over punctuated data streams. *Unpublished manuscript*, 2002.

[TTS+14] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 147–156. ACM, 2014.

[TZ06] Nesime Tatbul and Stan Zdonik. Dealing with overload in distributed stream processing systems. In *Data Engineering Workshops, 2006. Proceedings. 22nd International Conference on*, pages 24–24. IEEE, 2006.

[Val90] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[Whi12] Tom White. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[XCD+14] Reynold S Xin, Daniel Crankshaw, Ankur Dave, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. Graphx: Unifying data-parallel and graph-parallel analytics. *arXiv preprint arXiv:1402.2394*, 2014.

[XGFS13] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM, 2013.

[yar] Migrating from mapreduce 1 (mrv1) to mapreduce 2 (mrv2, yarn). `https://www.cloudera.com/documentation/enterprise/5-3-x/topics/cdh_ig_mapreduce_to_yarn_migrate.html#concept_xfg_cmy_xl`. Accessed: 2017-04-11.

[YCLN14] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *Proceedings of the VLDB Endowment*, 7(14):1981–1992, 2014.

[YCX+14] Da Yan, James Cheng, Kai Xing, Yi Lu, Wilfred Ng, and Yingyi Bu. Pregel algorithms for graph connectivity problems with performance guarantees. *Proceedings of the VLDB Endowment*, 7(14):1821–1832, 2014.

[ZCD+12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[ZCF+10] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[ZDL+12] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. *HotCloud*, 12:10–10, 2012.

[ZDL+13] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 423–438. ACM, 2013.