MSc Thesis

Computing Apps Frequently Installed Together over Very Large Amounts of Mobile Devices

Martin Spielmann

Basel, BS, Switzerland
Matriculation Number: 2008-055-519

University of Zurich[UZH]

Departement of Informatics
Prof. Dr. Michael Böhlen

supervised by
Prof. Dr. Michael Böhlen
and Christian Ammendola

September 16th, 2015

Acknowledgments

I wish to express my sincere thanks to Prof. Dr. Michael Böhlen for giving me the opportunity to write this master thesis at the Database Technology Group of the University of Zurich and for his support in many meetings. Moreover, my sincere gratitude goes to my supervisor, Christian Ammendola, and also the team of *42 matters AG*, who supported me throughout the entire work with enthusiastic encouragement and careful critique, and for making the experiments using AWS possible by providing a liberal budget.

Last but not least, I would like to thank my family and friends. Especially for the careful counter-checking and support in this thesis, my thanks go to: Lena Asal, Gabriel Müller, Jonas Spielmann, and Simone Zuber.

Abstract

Knowing Apps frequently installed together is valuable for applications in the field of app recommendations and targeted advertising. In this work, a distributed algorithm based on Apache Spark is developed to find the apps most frequently installed together on a vast amount of mobile devices. The algorithm is optimized by understanding Spark's execution model and the characteristics of the input data. An available input dataset of 500'000 devices was examined by the distribution of its apps and app pairs. Furthermore, the algorithm was tested extensively on different cluster configurations and with differently sized input data to investigate both scalability and affordability.

Zusammenfassung

Zur passenden Empfehlung von Apps und zur zielgerichteten Einblendung von Werbung ist es wertvoll zu wissen, welche Apps oft zusammen installiert sind. In dieser Arbeit wurde ein verteilter Algorithmus auf Basis von Apache Spark implementiert, der die am meisten zusammen installierten Apps auf einer grossen Menge von mobilen Geräten findet. Der Algorithmus wurde durch das Verständnis des Ausführungsmodels von Spark und der Eigenschaften der Eingangsdaten bezüglich seiner Leistung optimiert. Dafür wurde ein verfügbares Datenset von über 500'000 Geräten im Bezug auf die Verteilung der Apps und App Paare untersucht. Desweiteren zeigen Experimente die Skalierbarkeit des Algorithmus auf unterschiedlichen Clustern und mit unterschiedlich grossen Eingangsdaten.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

42matters AG, a company based in Zurich, is active in the field of market intelligence for *Google Play* and *App Store*. Its services are based on the analysis of market offerings and the elaborate knowledge it has about app users and their demographics. Some services of *42matters* are based on the knowledge of which apps are frequently installed together. In this thesis, a distributed algorithm based on Apache Spark and running on a Apache Hadoop cluster is developed to find the apps most frequently installed together on a vast amount of mobile devices.

In the following subsections, the motivation, the formal problem definition and the technical preconditions of this thesis are introduced.

In chapter 2, the technical fundamentals are given for the later development of the algorithm. In that chapter the essential terms for the understanding of the following chapters are introduced.

In chapter 3, theoretical aspects as well as the implementation of the developed algorithm for computing the top-k app pairs most frequently installed together are presented. The analysis builds the foundation for further optimizations.

In chapter 4, performance experiments and several optimizations of the algorithm are discussed.

Chapter 5 outlines the contributions of this thesis and explains the main challenges and open ends for further work.

1.1 Motivation

The knowledge of apps which are frequently installed together is valuable in multiple situations. Basically, it can be an indicator that users interested in one of the apps are possibly interested in the other app as well. Two applications, both part of *42matters*'s products, are:

App Recommendation The knowledge that an app A is frequently installed with app B indicates app B to be a good candidate for recommendations for users which already have app A installed (and vice versa). Thus, it allows for personalized recommendations. The recommendation of apps to users of mobile devices in turn has several motivations, such as the

improvement of their engagement with the device itself or the enhancement of their user experience. Providers of digital distribution platforms for apps (app stores) and telephone companies branding their mobile devices have a use for recommendation data.

Targeted Advertising Targeted Advertising has the goal of placing advertisements as efficiently as possible. In this field, the knowledge of apps frequently installed together is valuable from two perspectives: To successfully promote an app to potential users and to create more appropriately targeted advertisements inside of apps.

42matters has collected a lot of information about the combination of apps which are installed on mobile devices with the Android OS. As data from 42matters shows, the *Google Play* app-store lists more than 1.5 million apps. In a worst case, there exist a total of 1.5 million squared app pairs, which would make finding the apps most frequently installed together a challenge regarding both time and memory. The number of apps as well as the number of available data of devices is constantly growing. This motivates the need of a scalable and affordable solution to find the apps most frequently installed together in a practical time frame. *42matters* predicts the availability of data from more than 100 million devices and has the need to recompute the apps most frequently installed together in one day to keep their model up to date.

1.2 Problem Definition - Top-K

In this thesis the apps most frequently installed together are defined as the top-k apps. k represents the number of sought apps for every other app. The challenge of finding the top-k apps in a set of devices is called top-k problem.

In this section, a formal definition of the top-k apps is provided. For these purposes, the following definitions are made:

$$M \coloneqq \text{set of all existing apps}$$

$$D \subseteq M \coloneqq \text{set of apps on a single device}$$

The input data is a list of devices, denoted as $\Delta = \{D_1, D_2 ... D_q\}$, the multiset of sets of installed apps on q devices. Let $A = \{a_1, a_2 ... a_n\} = \bigcup_{i=1}^{q} D_i$ be the set of all n apps installed on the devices in Δ. This means that every device in Δ is a subset of A: $D \subseteq A$, and every A is a subset of M: $A \subseteq M$.

A tuple of two apps (b, c) is called app-pair. The number of installations of an app-pair (b, c) on the same device over all devices in Δ is:

$$\text{installs}(b, c) = |\{D \in \Delta | b, c \in D\}| \in \mathbb{N}_0, \qquad b, c \in A$$

Additionally, we define score: $A^2 \to \mathbb{R}$ as a function which allocates a real number to every pair of apps. This score encodes the meaning of "most frequently installed together" and allows to have different definitions of the term. Two definitions used in this thesis will be presented in section 1.2.1 and 1.2.2.

Top-K The goal of the algorithm is to find $k \in \mathbb{N}$ apps for every app $a \in A$ in a specific input dataset Δ, with the highest score of all apps which are installed at least once together with a on one device, also called the *top-k* apps of a. Along the former definitions, this means finding for every app a set $R \subseteq A$ with $|R| = k$, for which for all elements $b \in A - R$ holds that

$$\text{score}(a, b) \leq \min\{\text{score}(a, c) | c \in R\}$$

The result of the algorithm is a set T with tuples of all n apps from the input data Δ together with the set of their *top-k* apps:

$$T = \{(a_1, R_1), (a_2, R_2), ..., (a_n, R_n)\}$$

1.2.1 Frequency Score

The most straightforward score is the verbal application of "most frequently installed together". In this case the score of two apps is the number of devices in a input dataset Δ with both apps installed, defined as:

$$\text{score}(a, b) \coloneqq \text{frequencyScore}(a, b) = \text{installs}(a, b) \qquad a, b \in A$$

Example For the input data $\Lambda = \{\{a_1, a_2, a_3\}, \{a_2, a_3, a_4\}, \{a_1, a_2\}\}$ [1] and $k = 1$, we have $D_1 = \{a_1, a_2, a_3\}$, $D_2 = \{a_2, a_3, a_4\}$, $D_3 = \{a_1, a_2\}$ and $A = \{a_1, a_2, a_3, a_4\}$. The following table shows $\text{frequencyScore}(a, b) = \text{installs}(a, b)$ for every combination of two apps occurring at least once:

	a_1	a_2	a_3	a_4
a_1		2	1	
a_2	2		2	1
a_3	1	2		1
a_4		1	1	

T is now the set of tuples of every app together with a set of $k = 1$ apps with the highest score. In the example, for every app in the columns the one app with the highest value can be picked. For this example multiple solutions exist, because for some apps multiple combinations with other apps are occurring equally often, e.g. a_2 occurs twice with a_1 and a_3. One possible solution is:

$$T = \{(a_1, \{a_2\}), (a_2, \{a_1\}), (a_3, \{a_2\}), (a_4, \{a_2\})\}$$

1.2.2 Dependency Score

Few apps have many installations and a vast amount of apps has only few. This leads to a strongly biased result of the top-k problem with the frequency score. If an app is installed very often (e.g. WhatsApp or Facebook) it has a high probability to be in the top-k of apps with few installations. The dependency score takes this factor into account and scores pairs of apps higher if they are more often installed together than it can be expected by their particular base

[1] In a real application, a_x would be the app identifier for an app. In Android apps, this is a package name such as "com.facebook.katana" or "com.whatsapp"

probability. To define the base probability, we first define the frequency of an app. It is the number of installations of a specific app a in the input dataset Δ:

$$\text{frequency}(a) = |\{D | a \in D, D \in \Delta\}|$$

The base probability of an app $a \in A$ in the input dataset Δ is the number of devices in Δ with the app divided by the number of all devices in Δ:

$$P(a) = \frac{\text{frequency}(a)}{|\Delta|}$$

The probability of a specific app-pair (a, b) in Δ is the number of devices with both apps installed divided by the number of all devices:

$$P(a \cap b) = \frac{installs(a, b)}{|\Delta|}$$

Two events x and y are independent if the probability of them occurring together is equal to the product of their individual probability: $P(x \cap y) = P(x) \cdot P(y)$. From this formula the following rule for an app-pair (a, b) can be inferred:

$$\frac{P(a \cap b)}{P(a) \cdot P(b)} \begin{cases} > 1 & \text{more often installed together than expected} \\ = 1 & \text{as often installed together as expected} \\ < 1 & \text{less often installed together than expected} \end{cases}$$

This formula is defined as the dependency score. The goal of the top-k algorithm with the dependency score is to find for every app a in Δ the k apps which are the most frequently installed together more often than expected in Δ by their base probability. With the above definitions, we get:

$$\begin{aligned} \text{score}(a, b) \coloneqq \text{dependencyScore}(a, b) &= \frac{P(a \cap b)}{P(a) \cdot P(b)} \\ &= \frac{installs(a, b)}{\text{frequency}(a) \cdot \text{frequency}(b)} \qquad a, b \in A \end{aligned}$$

Example For the same input data $\Lambda = \{\{a_1, a_2, a_3\}, \{a_2, a_3, a_4\}, \{a_1, a_2\}\}$ and $k = 1$, it is $D_1 = \{a_1, a_2, a_3\}$, $D_2 = \{a_2, a_3, a_4\}$, $D_3 = \{a_1, a_2\}$. Again, we can infer $A = a_1, a_2, a_3, a_4$. First we count the frequency of all apps in A:

a	a_1	a_2	a_3	a_4
frequency(a)	2	3	1	1

Based on these frequencies and the numbers of values of installs(a, b) already computed for the example in the last chapter, the dependency scores can be calculated:

	a_1	a_2	a_3	a_4
a_1		1/3	1/2	
a_2	1/3		2/3	1/3
a_3	1/2	2/3		1
a_4		1/3	1	

As in the former example we can now infer the result R:

$$T = \{(a_1, \{a_3\}), (a_2, \{a_3\}), (a_3, \{a_4\}), (a_4, \{a_3\})\}$$

Compared to the frequency score it is noticeable that a_2 is not the top (because $k = 1$) app of any other app. The reason for this is that a_2 is installed on every device (frequency$(a_2) = |\Delta|$), so it is expected to be installed often with every other app.

1.3 Preconditions and System Architecture

Alongside the given problem defined in the former chapter as the challenge, the following conditions and parts of the system are given for the specified reasons.

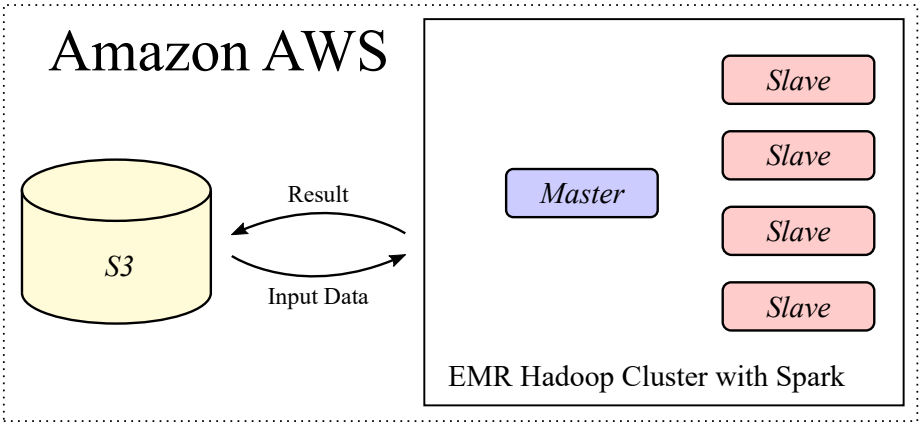

Figure 1.1: Illustration of the system architecture as given for the thesis.

Distributed Algorithm As the amount of available apps $|M|$ and the size of available input data Δ is constantly growing, so is the required effort to find the apps most frequently installed together. The necessity of finding a good solution of the top-k problem follows the limited scalability of a non-distributed solution.

Designed and implemented for Apache Spark[2] Apache Spark as a cluster computing framework promises fast large-scale data processing, based on a unique in-memory approach. Additionally, it has a very compact and clear interface. Both factors lead to the decision of using Spark to develop and implement the top-k problem. The required Spark fundamentals are described in section 2.2.

Data from and to *S3*[3] The Amazon *S3* (Simple Storage Service) by *Amazon Web Services (AWS)* is used to store the input and output dataset. *S3* provides cheap storage capacities for diverse data and is reachable through web service interfaces like REST and SOAP. *S3* is used by many systems of 42matters, so its use ensures the solution to be highly compatible with the systems of 42matters.

[2] http://spark.apache.org/
[3] https://aws.amazon.com/de/S3/

Use of Elastic MapReduce[4] Elastic MapReduce (EMR) is a service by *AWS* providing clusters of multiple virtual machines with a ready-to-use Spark environment. EMR allows the creation of clusters with a dynamic number of differently equipped machines in short time. Clusters can be shut down at any time and are priced at an hourly rate. EMR clusters are also prepared to interact with data on *S3*. For these reasons, EMR is a good choice for developing and testing different configurations as well as for productive purposes (only charged for hardware while calculating the top-k apps).

1.4 Related Work

The finding of frequent itemsets has been recognized by many researches as an important topic especially from a database perspective [1]. Hereby the Apriori algorithm attracts a lot of research attention [2] [3], also in the field of distributed systems [4]. The main application of the Apriori algorithm is association rule learning. The main difference to the present *top-k problem* is that in association rule learning only strong rules are desired, but the *top-k problem* is looking for k combinations of every app with other apps, no matter what the frequency of the combinations are.

[4] https://aws.amazon.com/de/elasticmapreduce/

Chapter 2

Fundamentals

The algorithm to solve the top-k problem as it will be introduced in chapter 3 is based on technologies as they are presupposed in section 1.3. The current chapter lays out the basics of the technologies as they are required to understand the subsequent work. Section 2.1 describes the concept of the underlying infrastructure as it is used by Apache Spark. Apache Spark itself, as the underlying framework of the algorithm, is described in section 2.2. Finally, section 2.3 describes the form of the available test data and how it is stored in *AWS S3*.

2.1 Spark Infrastructure

Spark programs are running on a set of independent processes on a cluster of multiple processors and machines, called nodes. The processes are started and managed by a *cluster manager*[1].

Depending on the used *cluster manager* and its configuration, the underlying infrastructure, which is executing Spark programs, may differ substantially. The abstraction shown in figure 2.1 can be understood as a common foundation independent from the *cluster manager*.

A Spark application is coordinated by the *driver program* (called *driver*). The *driver* is running on a node on the cluster (also calles master)[2] and splits the program into individual *tasks*, which are executed independently. The n processes provided by the *cluster manager* are denoted as *executors* and mostly executed on their own node in the cluster, so called slaves. The number of *executors* is statically assigned to a Spark program[3]. An *executor* is providing resources for i in parallel running *tasks* - it has i *tasksslots* (in figure 2.1 denoted as "TS"). The *tasks* are running in their own thread in the *executor* process. i, the number of *tasksslots* per *executor* depends on the configuration of Spark and the *cluster manager*. A single *task* may be running on multiple CPUs. All *tasks* running on the same *executor* can access a common cache. This means

[1] Spark supports Mesos, YARN or a standalone cluster manager

[2] With YARN as the *cluster manager* it is possible to run the *driver* outside of the cluster

[3] Since Spark Version 1.2 it is possible to dynamically allocate resources as *executors* to Spark. This option is not active per default. Its activation is reasonable if multiple applications are running on one cluster.

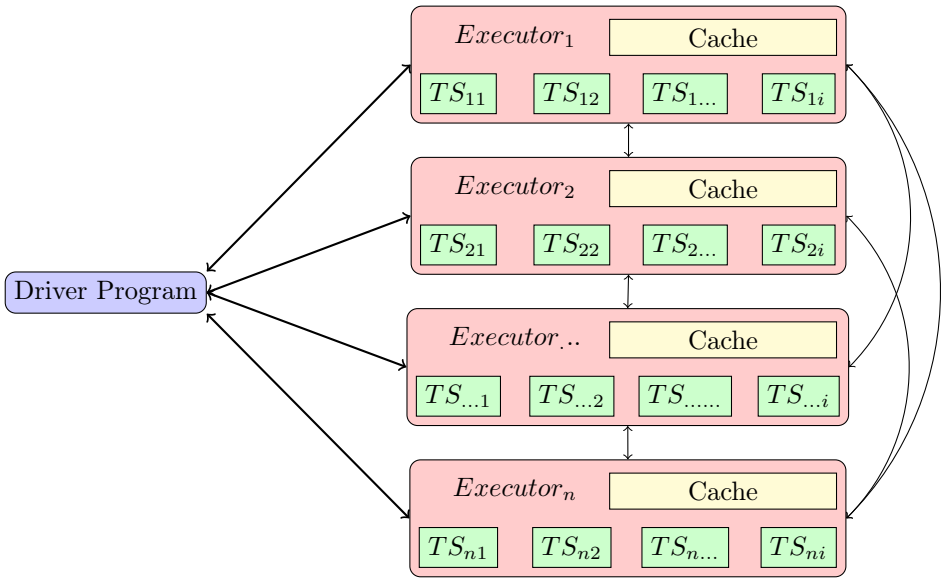

Figure 2.1: Underlying infrastructure of Spark, independent from the *cluster manager* [5]

that the *driver* is coordinating the execution of $n \cdot i$ parallel running *tasks* at maximum.

2.2 Apache Spark

Apache Spark (below denoted as Spark) is a cluster computing framework which is used to design and implement the algorithm in this thesis. Spark provides high-level APIs for several languages[4] which are used to implement algorithms. Spark executes these programs in an optimized manner on the underlying cluster. Even though this chapter explains the core concepts independent of the used API, in chapter 3 the Python API is used to implement the top-k algorithm.

To understand Spark as it is used in this thesis, two aspects have to be discussed: In section 2.2.1 Resilient Distributed Datasets (RDD) and their utilisation are explained. RDDs are an abstracted collection of distributed items. Section 2.2.2 explains the execution model of Spark, this means the way a Spark program is executed on a cluster.

2.2.1 Resilient Distributed Datasets (RDD)

Resilient Distributed Datasets (RDD) are the primary abstractions used by Spark. They are originally proposed in [6]. An RDD is a non-writable, distributed collection of data items. An RDD is created by loading data from a stable storage such as a hard drive or by a *transformation* of an other RDD. A Spark program is developed by defining one or more RDD by loading data from solid storage and then deriving consequential RDDs via *transformations*. On these RDDs so-called actions are applied which return data to the *driver* or

[4] As of version 1.4 APIs for Java, Scala, Python and R are available

save data to stable storage. The data items in a RDD may be arbitrary objects, depending on the used API [5]. Some operations of Spark (actions as well as *transformations*) are only applicable on key-value RDDs, a type of RDDs with some special properties All items in a key-value RDD consist of two elements, a key and a value. The following sections explain the operations used in this thesis. More operations are available but not used. [7][8]

Data Loading

An RDD can be created either by loading data from a distributed storage system[6] like *S3* (see 1.3) or *HDFS* [7] or by sending data from the *driver*. Both operations are applied on the *SparkContex* object which is provided by the Spark environment.

$parallelize(Seq[T])$ $Seq[T] \rightarrow RDD[T]$

This function creates an RDD with elements from the *driver*, which are provided as a sequence of arbitrary items $Seq[T]$. $parallelize$ is used mainly for testing purposes without much data.

$textFile(PATH)$ $TextFile \rightarrow RDD[U]$

This function creates a new RDD by loading a text file from a distributed storage like *S3* or *HDFS*[8]. Every line of the file becomes a item in the RDD. *textFile* function is used in this thesis to read the input dataset.

Transformations

All *transformations* transform one or more RDDs to a new RDD. The following *transformations* are essential for the proposed top-k algorithm.

$map(f: T \rightarrow U)$ $RDD[T] \rightarrow RDD[U]$

map is the most basic *transformation*. It transforms an RDD of type T to a RDD of type U. The function f defines how the elements are transformed from T to U.

$mapValues(f: T \rightarrow U)$ $RDD[(K,T)] \rightarrow RDD[(K,U)]$

mapValues is used for key-value RDDs instead of *map* if the keys should not be changed. The use of *mapValues* instead of *map* allows some further optimizations at runtime.

$flatMap(f: T \rightarrow Seq[U])$ $RDD[T] \rightarrow RDD[U]$

This *transformation* increases the number of items in an RDD. The function f defines how the elements are transformed from T to $Seq[U]$, a sequence of multiple elements of type U.

[5] For example: In Python, arbitrary Python objects can be used

[6] A distributed storage system is a stable storage where data is stored on multiple nodes and can be read or written in parallel from multiple nodes.

[7] Hadoop Distributed File System (*HDFS*) is a distributed storage system, also provided on Elastic MapReduce cluster.

[8] Local file systems are possible but not recommended because of the inability to read the data in parallel by different executors.

$reduceByKey(f\colon (V,V) \to V)$ $RDD[(K,V)] \to RDD[(K,V)]$

reduceByKey can only be applied on key-value RDDs. It decreases the number of items in an RDD. All elements with the same key K are reduced to one item. f defines how two values of items with the same key are combined to a value of the same type V.

$groupByKey()$ $RDD[(K,V)] \to RDD[(K,Seq[V])]$

Similar to *reduceByKey*, this function combines all items with the same key to one new value. In contrast, the values with the same keys are not combined with an individual function but linked to a sequence $Seq[V]$.

$combineByKey(f_1\colon V \to U, f_2\colon U \times V \to U, f_3\colon U^2 \to U)$ $RDD[(K,V)] \to RDD[(K,U)]$

This *transformation* is the most complex to use. It combines multiple values of a key-value RDD with the same key to a new value of a different type. f_1 defines the creation of a new value of type U from a single value of type V. f_2 defines how the other values of type V are integrated in the already existing new value and f_3 defines how multiple values of type U are being combined.

$distinct()$ $RDD[T] \to RDD[T]$

This *transformation* eliminates redundant values in an RDD. It may decrease the number of items in an RDD.

Action

Actions are operations on RDDs not deriving a new RDD. All actions either return data to the *driver* or export it to a stable storage.

$count()$ $RDD[T] \to Long$

count returns the number of elements in a RDD to the *driver*.

$collect()$ $RDD[T] \to Seq[T]$

collect returns the whole RDD with all elements to the driver. Similar to the *transformation parallelize*, this action is only used for small RDDs.

$saveAsTextFile(path\colon String)$

This action writes all elements of an RDD to a text file in a given directory in *S3* or *HDFS*[9]. Every object is converted to a string[10] and written on their own line.

Example

Listing 2.1 shows an example of a short Spark program implemented with the Python API of Spark. The Program reads a textfile and counts the frequency of each word in the file.

[9] Local filesystems are also possible

[10] By calling *toString* or equivalent functions in the particular API language.

```
textFileRDD = spark.textFile(INPUT_PATH)
wordCountsRDD = textFileRDD.flatMap(lambda line: line.split(" ")) \
                .map(lambda word: (word, 1)) \
                .reduceByKey(lambda x, y: x + y) \
wordCountsRDD.saveAsTextFile(OUTPUT_PATH)
```

Listing 2.1: Example of a Spark program

Firstly, an RDD is created by loading the file at *INPUT_PATH* with the function *textFile*. Then, via three consecutive *transformations* new RDDs are created. *flatMap* splits each line[11] into single words. After this *transformation* the RDD is a huge wordlist. *map* creates a key-value RDD by mapping every word to an element where the word is the key and the value is "1". *reduceByKey* reduces all elements with the same key (all instances of the same words) to a single element while summing up their values (frequency). Lastly, the wordlist is saved to a stable storage at *OUTPUT_PATH*.

2.2.2 Execution Model

The distributed processing in Spark is a result of splitting the items in an RDD into smaller subsets, called partitions. These partitions are processed simultaneously in as many *tasklots* as possible (see 2.1). The main program (e.g. as shown in listing 2.1) is running on the driver. Any operations on an RDD are deferred until an action is executed. At this point, returning values (e.g. *count*) to the *driver* or saving data to a stable storage (e.g. *saveAsTextFile*) is necessary. For this reason, Spark creates a lineage graph of the RDD as a directed acyclic graph (DAG). RDDs constitutes the nodes and *transformations* the edges. Spark classifies two different types of dependencies between two RDDs [8]:

Narrow Dependencies Every partition depends on only one partition of the previous RDD. The *map transformation* leads to a narrow dependency, for example. Only one item of the previous RDD is used to create an element in the new RDD, so only one partition is needed to create a new partition.

Wide Dependencies One or more partitions depend on more than one partition of the previous RDD. For example, the *reduceByKey transformation*; every key of the complete previous RDD (and so all partitions of it) is reduced to one item.

Based on the DAG, the execution of an action is split by the *driver* into so-called stages. Stages are then executed sequentially one after another. Thereby a stage covers as many *transformations* of RDD as possible which are related by *narrow dependencies*. Thus the boundaries of a stage are the *narrow dependencies*.

All *transformations* within a stage are *narrow dependencies* and can be processed directly and consecutively on a single partition in a *tasklot*. At the transition to a new stage at a *wide dependency* the rearrangement (see 2.2.3) of data is necessary. Thus, the processing of one single partition in a single stage

[11] Each line of the text file becomes one item in the RDD.

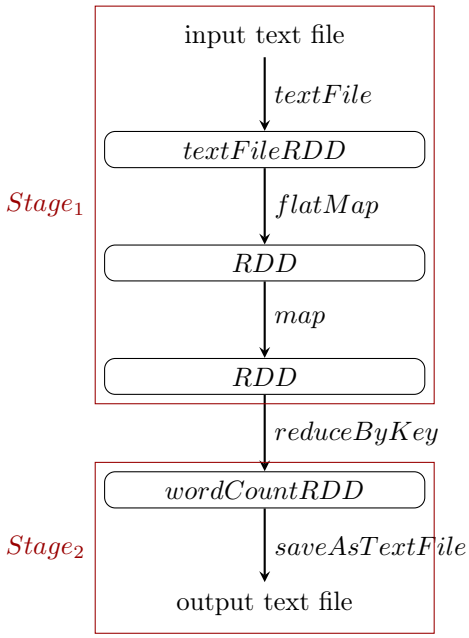

Figure 2.2: DAG of example Spark program from listing 2.1. In the program two RDDs have no name because the *transformations* are directly chained. In the figure they are simply marked as "RDD".

is the smallest unit of calculation, called task. While executing an action, the *driver* sends *tasks* to the *executors* which executes the *tasks* in a *tasksl*ot. From this follows that there exists one task per stage and partition. The amount of partitions emerging while rearranging the data at a *wide dependency* can be configured. When a new RDD is created by reading data from a stable storage one block is building one partition. The effectiveness of the parallelization depends amongst others on the configured number of partitions emerging at *wide dependencies*; it should be at least equal to the number of *taskslots* in the cluster.

2.2.3 Shuffle Operations

The previous chapter explains the different handling of *narrow* and *wide dependencies*. These *wide dependencies* induce a rearrangement of data over all partitions in an RDD, called a Shuffle. For example, the *reduceByKey transformation* generates a new RDD by bringing all elements with the same key together. For this purpose all elements on all partitions have to be considered, because it is possible that not every element with the same key is in the same partition. Operations used in this thesis which lead to a Shuffle are *combineByKey* and *reduceByKey*. A Shuffle operation involves many resources: Data has to be serialized, written to disk and sent over the network. Regarding a specific Shuffle operation, the *tasks* prior to the *transformation* with the *wide dependency* are called *map tasks* and are organizing the data. The *tasks* after the *transformation* are called *reduced tasks* and are aggregating the data. Because the work done in Shuffle operations often accounts for a large part of

the runtime of a Spark program, Shuffle operations are an important focal point of further Spark development. Essentially, the *map tasks* are sorting their resulting elements based on the target partition and writing them to an over all *tasks* on a *executor* consolidated file for every target partition. Subsequently, the reduce *tasks* are receiving their prepared data and aggregating them. [7]

2.2.4 Caching

As visible in figure 2.1, every *executor* manages its own cache. This is implemented by the ability to flag an RDD as *persistent*[12]. If after such a flagging an action on the RDD itself or a derived RDD is executed and the RDD has to be calculated, its concrete data is kept in memory by the executors. This enables the fast reuse of RDDs. [8]

2.2.5 Broadcast Variables

In operations like *map* the *transformation* of single elements is done with a function in the used API language (see the Python *lambda* functions in listing 2.1). These functions are serialized by Spark per task and then sent to the executors. All variables used in these functions are also serialized and available on the executors. With a *broadcast* variable, data can be sent to all *executors* and then used in functions. Hereby, it can be prevented that the same data is serialized and sent to *executors* again and again for every task.

2.3 Test Data

In this thesis, a dataset with data from 500'000 (called 500k test dataset) devices was used to implement and test the algorithm. The data was collected by *42matters* and contains data from devices all over the world. The data is provided by *42matters* on Amazon *S3* as a text file. Every line of the text file represents one device and is encoded in JSON[13]. An example for a single device is illustrated in listing 2.2. The field "id" is a unique identifier of the app, "apps" is the list of all apps installed on the device, and "pn" is the package name of an app, used as an app identifier. Additional metadata for every app installation is also part of the dataset but was not used in this thesis.

2.3.1 Meta Data

For further optimizations of the algorithm, additional metadata for many apps is available. *42matters* collected this data by crawling the official website of *Google Play*. The metadata is also accessible on *S3* in a single text file with one JSON encoded dataset per line. As an example, listing 2.3 shows all available metadata for an app with the package name "com.poynt.android".

[12] In Python the function *cache()* is used to flag an RDD as *persistent*
[13] JavaScript Object Notation, RFC 7159

```
{
    "id": "549df753-d1a0-4a99-8110-846e6c93fe39"
    "apps": [
        {"pn": "canvasm.myo2"},
        {"pn": "com.android.chrome"},
        {"pn": "com.android.vending"},
        {"pn": "com.andymstone.metronome"},
        {"pn": "com.beyondinfinity.tetrocrate"},
        % more apps....
    ]
}
```

Listing 2.2: Example of one device in 500k test data. The line has been formatted with line breaks and blanks for better representation.

```
{
    "rating": 4.033398151397705,
    "lang": "en",
    "package_name": "com.poynt.android",
    "creator": "Poynt Inc",
    "downloads_min": 5000000,
    "category": "Travel & Local",
    "cat_int": 23,
    "downloads_max": 10000000,
    "version": "2.4.6",
    "version_update_timestamp": 1418169600000,
    "cat_type": 0,
    "ratingsCount": 26257
}
```

Listing 2.3: Example of metadata for app with package name "com.poynt.android".

Chapter 3

Algorithm

In this chapter, first the basic algorithm to solve the top-k pairs problem in Spark is developed conceptually and concretely. Afterwards the algorithm is analysed by its execution in spark and its complexity.

3.1 Design

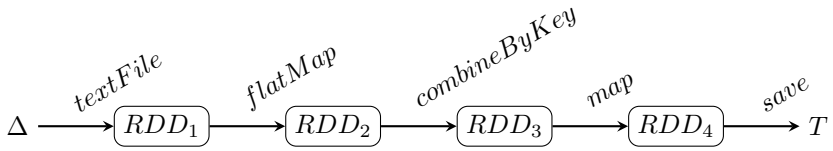

Figure 3.1: Basic algorithm RDD operations

A basic graph of RDD operations for the algorithm is shown in figure 3.1. As defined in 1.2, Δ represents the input data and T the result. The 5 operations are defined as the following:

1. $textFile$ The input data is read from a distributed dataset like *HDFS*, Cassandra or Amazon *S3*. The resulting RDD RDD_1 consists of q elements $D_1, D_2 ... D_a$ as sets of installed apps on a specific device.

2. $flatMap$ In every set of installed apps D all possible pairs $a, b \in D, a \neq b$ of apps are created and returned as a tuple (a, b). The result is a multiset containing all pairs as often as they are occurring all over the devices. a, as the first element, is handled by spark as the key and b as the value of an element. So, RDD_2 is a key-value-RDD.

3. $combineByKey$ All values of elements in RDD_2 with the same key (app a of the former operation) are combined to one value. The resulting RDD RDD_3 contains an element for every app a, with a as the key and a set of tuples of the form (b, n) as value in which n is the number of installations of app b with a.

4. map In this operation the top-k pairs for every app are selected either based on the frequency or the dependency score. If the frequency is selected, for

every value in the elements of RDD_3 only the k tuples with the largest n in (b, n) are kept. Otherwise the dependency score is calculated first for every tuple with the formula from 1.2.2[1] an then used to pick k tuples with the largest score.

5. *save* The data is written to a persistent storage system as the input data was read in the first operation.

Figure 3.2 illustrates the algorithm with $\Delta = \{\{a_1, a_2, a_3\}, \{a_2, a_3, a_4\}, \{a_1, a_2\}\}$ and $k = 1$. The example evidently shows that the top-k pair problem may have multiple solutions. For example, both, $(a_4, \{a_2\})$ and $(a_4, \{a_3\})$ are possible elements of the result set, because a_4 is installed exactly once with a_2 and once with a_3.

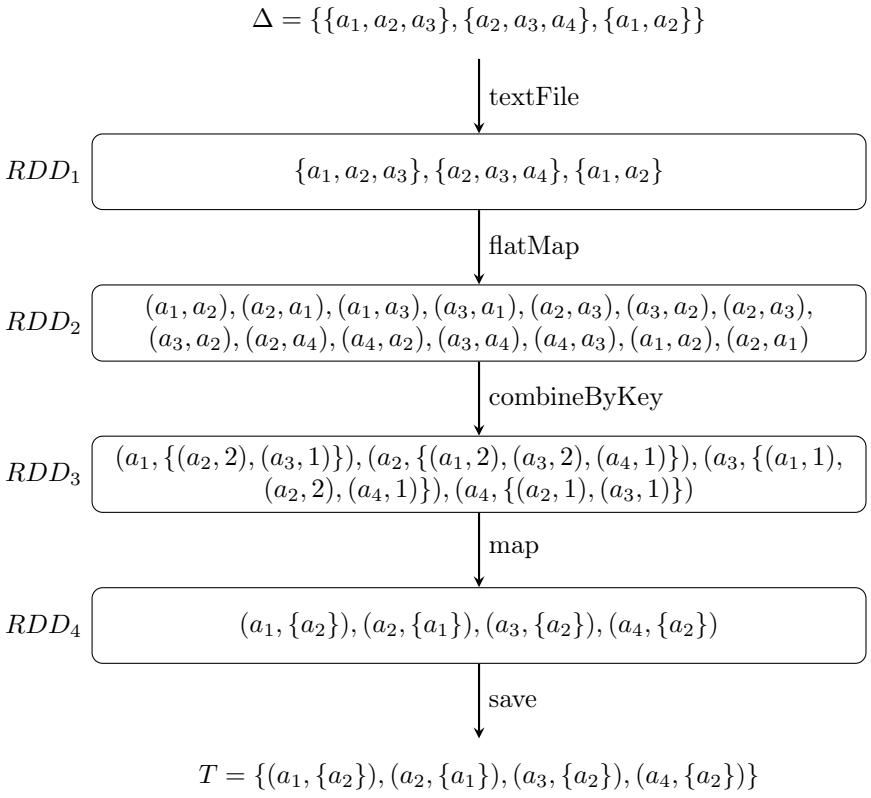

Figure 3.2: Example of the basic algorithm with frequency score and $k = 1$

3.2 Implementation

The actual algorithm implementation is created for the Python interface of Spark. The complete scripts can be found in Appendix B. Below, the main

[1]The base probabilities frequency(a) of all $a \in A$ are precomputed in a separate step previous to the actual algorithm

parts regarding the operations described in the former chapter are listed and discussed in order of their execution.

$\Delta \rightarrow RDD_1$ Data loading and preparation The input data is prepared as a newline-delimited JSON file [2] in Amazon *S3* as it is described in 2.3. The package names, used as identifier for apps, are URL like strings (e.g. '*com.facebook.katana*'). To shrink the amount of data which is processed in further steps, part of the data preparation is to append an integer-mapping for package names. Listing 3.1 shows how data loading, JSON parsing, and the integer-mapping of package names are implemented.

```
RDD1 = sc.textFile(INPUT_PATH)
RDD1 = RDD1.map(lambda x: [x['pn'] for x in json.loads(x)['apps']])
    .cache()

app_list = RDD1.flatMap(lambda x: x).distinct().collect()
pn_int_list = sc.broadcast(dict((n, i) for i, n in enumerate(
    app_list)))
int_pn_list = sc.broadcast(dict(zip(pn_int_list.value.values(),
    pn_int_list.value.keys())))

RDD1 = RDD1.map(lambda x: [pn_int_list.value[y] for y in x])
```

Listing 3.1: Data loading and preparation

On line 2, Spark's functionality to cache elements of an RDD (see chapter 2.2.4 is used. Without invoking the '*cache()*' function, the former map *transformation* and data loading would be executed twice: once for the creation of the app list on line 4 and once the application of the package name integer-mapping on line 8. Line 6 creates a package name-integer dictionary as a Spark *broadcast* variable, which is afterwards used in the *transformation* on line 8 to apply the mapping.

$RDD_1 \rightarrow RDD_2$ Pair creation The pair creation is implemented in a function that gets a list of app identifiers (installed apps on one device) and returns a list of tuples for all pairs of identifiers ((a, b) and (b, a)) in this list. This function is later on used with the Spark *flatMap-transformation* on line 9 to create RDD_2 which holds an element for every pair of two apps installed together amongst all devices. RDDs with Python tuples as elements are treated as key-value RDDs by Spark, whereby the first element of the tuple is the key and the second element is the value. This renders key-value operations possible on RDD_2.

```
def pairsInList(elements):
    pairs = []
    for i in xrange(0, len(elements)):
        for j in xrange(i + 1, len(elements)):
            pairs.append((elements[i], elements[j]))
            pairs.append((elements[j], elements[i]))
    return pairs

RDD2 = RDD1.flatMap(pairsInList)
```

Listing 3.2: Pair creation

[2] http://jsonlines.org/

$RDD_2 \rightarrow RDD_3$ **Pair combination** The combination of all apps (respectively app identifiers) which are installed together with a specific app is done with the Spark *transformation combineByKey*. *CombineByKey* transforms a key-value RDD into another key-value RDD with another type of values while combining all values with the same key: $RDD[(K,V)] \rightarrow RDD[(K,W)]$. In this case, K and V are integers (app identifiers) and W is a Python defaultdict[3] containing app identifiers (integers) as keys and frequencies as values (integers). For example the elements $(a_1,a_2),(a_1,a_5),(a_1,a_2),(a_1,a_6)$ should be combined as $(a_1,\{a_2:2,a_5:1,a_6:1\})$.

```
def createCombiner(firstPair):
    pairDict = defaultdict(int)
    pairDict[firstPair] = 1
    return pairDict

def mergeValue(pairDict, newpair):
    pairDict[newpair] += 1
    return pairDict

def mergeCombiners(pairDictA, pairDictB):
    res = pairDictB.copy()
    for app, counter in pairDictA.iteritems():
        res[app] += counter
    return res

RDD3 = RDD2.combineByKey(createCombiner, mergeValue, mergeCombiners
    )
```

Listing 3.3: Pair combination

As listing 3.3 shows, the *combineByKey* function requires three arguments:

createCombiner creates the values of type W out of the first value V for a unique key K. In this case this means instancing a python defaultdict and inserting the value "1" for the first app identifier as the key.

mergeValue adds another value of type V to a value of type W, if it already exists for a specific K. Here it just increments the value (frequency) for the app identifier by one. The default for not existing keys in a Python defualtdict of type '*int*' is "0", therefore the frequency of a new app identifier is set to "1".

mergeCombiners combines two values of type W. In this case it sums up the frequencies of two defaultdicts for corresponding keys (app identifiers).

$RDD_3 \rightarrow RDD_4$ **Pair selection** To select the top-k pairs the '*heapq*' module of the Python Standard Library is used[4].

```
def sortAndSelect(pairDict):
    topK = heapq.nlargest(K, pairDict.iteritems(), key=lambda x: x
    [1])
    return topK

RDD4 = RDD3.mapValues(sortAndSelect)
```

Listing 3.4: Pair selection

[3] Available in the collections module as of version Python version 2.5
[4] https://docs.python.org/2/library/heapq.html

In the dependency score version, the score has to be calculated for every app combination, which is directly done in the lambda statement. Therefore, overall app frequencies are calculated in an earlier step and saved together with the package name-integer mapping. The whole relevant python script can be reviewed in the appendix.

$RDD_4 \rightarrow T$ **Data saving** Before the result is saved to a storage system, it is encoded to the desired JSON string.

```
1 def listToJson(device):
      # see Appendix A
3
  RDD4.map(listToJson).saveAsTextFile(OUTPUT_PATH)
```

Listing 3.5: Data saving

3.2.1 Base Probability Calculation

For the dependency score and as basis for some optimization approaches introduced in chapter 4 the base probabilities are needed. In these cases, the implementation of its calculation takes place at the creation of RDD_1 showed in listing 3.1. This part of code is replaced by the code in listing 3.6.

```
  RDD1 = sc.textFile(INPUT_PATH)
2
  RDD1 = RDD1.map(lambda x: [x['pn'] for x in json.loads(x)['apps']])
      .cache()
4
  num_of_devices = RDD1.count()
6
  app_frequency_list = RDD1 \
8     .flatMap(lambda x: map(lambda x: (x,1), x)) \
      .reduceByKey(lambda x, y: x + y) \
10    .mapValues(lambda x: float(x) / num_of_devices) \
      .collectAsMap()
12 pn_int_list = {}
  int_pn_list = {}
14 for index, app in enumerate(app_frequency_list.iteritems()):
      pn_int_list[app[0]] = (index, app[1])
16    int_pn_list[index] = (app[0], app[1])

18 pn_int_list = sc.broadcast(pn_int_list)
  int_pn_list = sc.broadcast(int_pn_list)
20
  RDD1 = RDD1.map(lambda x: [pn_int_list.value[y][0] for y in x])
```

Listing 3.6: Base probability calculation implemented while creating RDD_1

At the point where in listing 3.1 the app list is created, in listing 3.6 on line 7 a chain of *transformations* is calculating the base probability of every app. Firstly, with the *flatMap* and *reduceByKey transformation* the frequency of every app is counted and then it is divided by the number of devices in a *mapValues transformation* to get the base probability. Finally, the key-value RDD is collected on the *driver* with the *collectAsMap* action. As a result, for every package name as a key the Python dictionary "app_frequency_list" contains the corresponding base probability as a value. The loop at line 14 then creates

the dictionaries for the package name-integer mapping with the base probabilities as a addition. Theses dictionaries are then distributed to all *executors* with a Spark *broadcast* variable. As a consequence, the base probabilities are available on every *executor*.

3.3 Analysis

To develop possible approaches for the algorithm optimization, it is essential to understand the way it is executed by Spark. Through the abstraction of RDDs, some fundamental ways in which data is processed, stored and transferred are hidden. Chapter 3.3.1 dissect the way Spark executes the algorithm using the terms described in chapter 2.2. Afterwards, chapter 3.3.2 analyses the characteristic numbers with an influence on the runtime of the algorithm.

3.3.1 Execution

The algorithm, the way it is implemented in chapter 3.2 as a python script, is executed on the *driver* node on a cluster. Thereby a lineage graph of RDDs is constructed as long as no *actions* are executed on a RDD. The first *action* inducing cluster work is *collect()* (see listing 3.1, line 4). At this point, the input data is loaded and processed by the executors, and a list of distinct package names is finally send to the driver. This process is not explicitly discussed here, because it its relevance for the program's overall runtime costs is minimal.

The actual work is done when *saveAsTexFile* (see listing 3.5, line 4) is executed at the end of the program. At this point, the result should be written to an external storage point and it therefore has to be calculated first. In other words, data processing cannot be deferred any longer.

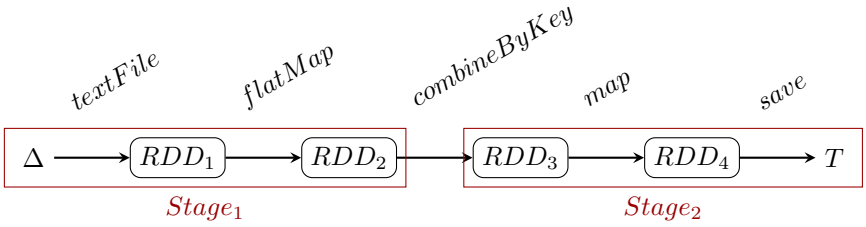

Figure 3.3: Algorithm splitting in stages

Spark splits the lineage graph into two individual *stages* which are serially processed by the *executors* (see figure 3.3). *combineByKey* as a *transformation* with wide dependencies indicates the boundary of the two stages[5]. Then, it splits the workload of one stage into smaller chunks, so-called *tasks*. Every task processes one partition of elements. The number of partitions at the first stage is defined by the number of blocks the input file has on the storage system (*S3*). At the second stage it depends on the configuration of the cluster[6]

[5]Elements with the same key must come together on one executor, therefore the data has to be shuffled.

[6]The number of partitions emerging at *wide dependencies* can be configured in Spark, per default it is the number of available *tasksslots*. It is also possible to individually set the minimum number of partitions while applying a *transformation*.

RDD	# Elements	Avg. Size per Element	Size RDD
RDD_1	$\|\Delta\|$	$\|M\|$	$\|M\|\|\Delta\|$
RDD_2	$\|M\|^2\|\Delta\|$	2	$2\|M\|^2\|\Delta\|$
RDD_3	$\|M\|$	$1+2\|M\|$	$\|M\|+2\|M\|^2$
RDD_4	$\|M\|$	$1+k$	$\|M\|+k\|M\|$

Table 3.1: Upper limit of RDD sizes

Figure 3.4 on page 26 shows an example with two *tasks* per stage. In a Shuffle, elements are distributed over the partitions with a simple hash code [7]. While executing the program, the number of partitions in $Stage_1$ and $Stage_2$ most likely vary: the number of partitions in $Stage_1$ is given by the number of blocks of the input data on *S3*.

3.3.2 Complexity

Both time and space complexity of a spark program are closely related to the execution model of spark and the subjacent cluster. As a simplified approach, the amount and size of RDD elements at the time of execution are taken in this section to get a deeper insight in the overall time complexity of the discussed algorithm.

Due to the strong dependency between the RDD numbers (number of elements and size) and the input data Δ, this chapter takes two perspectives. The worst case argues to an upper limit in an analytical manner and the average case models the numbers on the basis of experiments with testdata.

Worst Case

In the worst case, all existing apps are installed on all devices in the input data. This means that if M is the set of all existing apps we have that $D = A = M$ for all $D \in \Delta$. In this case, all app-pairs (a, b) for $a, b \in A, a \neq b$ are appearing on every device $D \in \Delta$. Table 3.1 shows the upper bounds of all RDDs.

We can assume that input data is available for many more than two devices, so that $|\Delta| >> 2$. Thus the largest RDD is RDD_2 in the worst case.

If we assume that not every app is installed on every device but just an equal sized subset of all apps, so $|D| = q, q < |A|$ and $D \subset A$ for all $D \in \Delta$ and still every possible app-pair is appearing, the size of RDD_2 shrinks in the worst case to $2q^2\,|\Delta|$.

Average Case

The worst case scenarios of the former chapter are not a good foundation to predict the algorithm's costs in real cases, because the numbers strongly depend on the input data's characteristics. For instance, only a small part of all possible app-pairs is actually occurring, so that the number of pairs in reality is much smaller than $|A|^2$. To enable better assumptions about the calculation cost of large input datasets Δ, the testdata was analysed to model a connection between

[7] Partitioning with a hash means the determination of the partition of an element by the formula: partition = $hash(\text{data}) \bmod \text{numOfPartitions}$. $hash()$ is a simple hash function, and *mod* stands for the modulo operation.

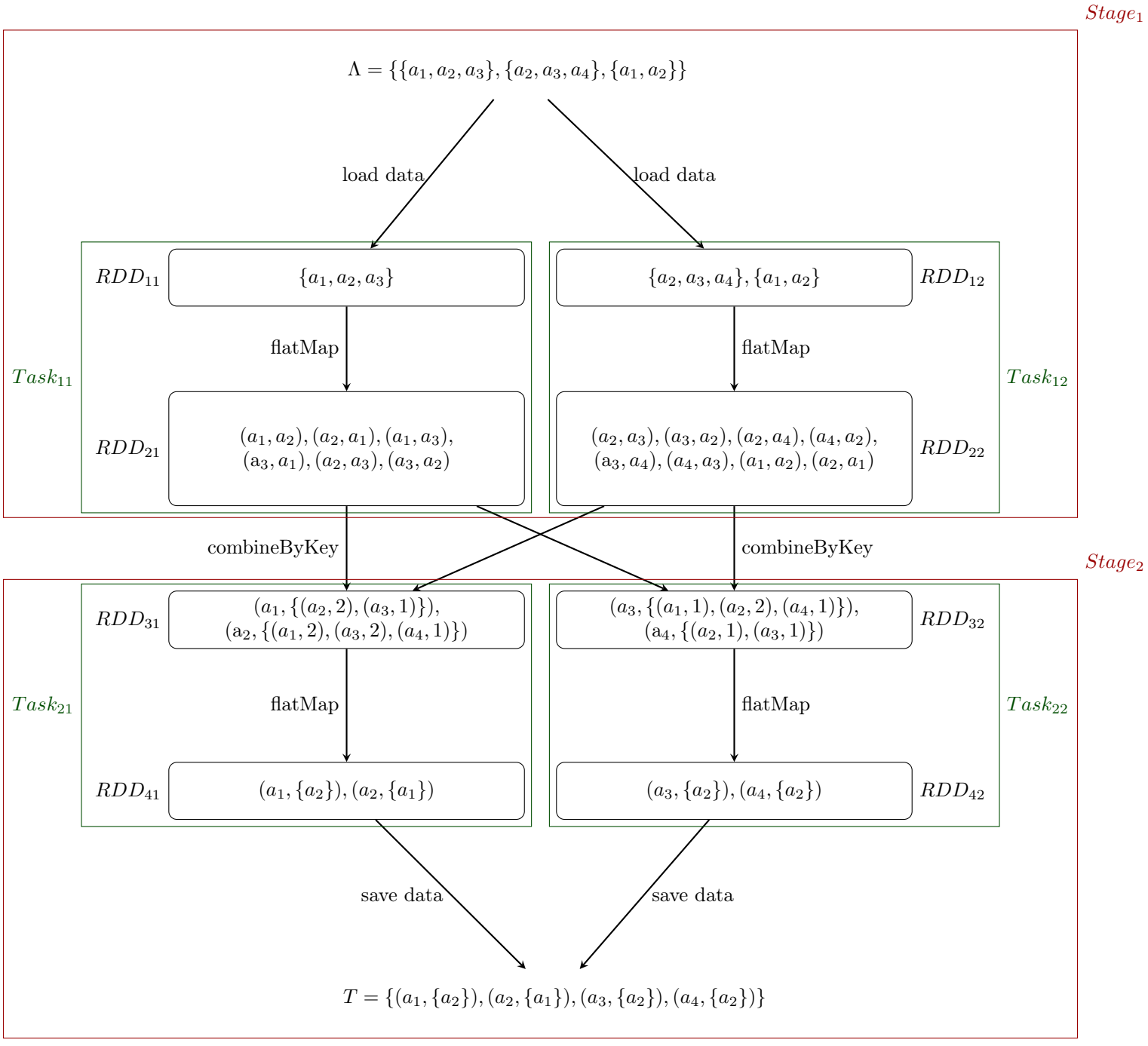

Figure 3.4: Basic algorithm example with two partitions/*tasks*

the input data and the sizes of RDDs. For this purpose, it is assumed that the characteristics of a new input dataset are equal to the ones of the used dataset (see 2.3).

To analyze the relations between the size of the input data and the expected size of the RDDs, the test dataset was used to create smaller datasets of several sizes. These datasets were then analysed by a spark program for the most important characteristics wich are used in the chapters below. Table 3.2 on page 28 shows the result of the analysis.

RDD_1 **Number of Apps per Device** This RDD contains one element per device in Δ. Every element contains a number of app identifiers of installed apps. The average amount of installed apps over all $500'000$ devices is 59.95 ($SD = 37.61$). Based on this, a good estimator for the size of RDD_1 is $\Delta \cdot 60$.

RDD_2 **Number of App-Pairs** RDD_2 contains one element for every pair of two apps which are installed together on one device. Many pairs are occurring multiple times, because the specific combination of two apps is occurring on several devices. The number of all such app-pairs would be the sum of squares of the number of apps on a device over all devices. This number highly depends on the characteristics of the distribution of the number of apps on a device, which we do not know. For this reason, the numbers of the 500k dataset are used to estimate the average number of pairs on one device: the number of pairs on the whole dataset can be calculated by multiplying the number of distinct pairs by the average of their quantity. The result divided by the number of devices gives the desired result:

$$\frac{\#\text{distinct pairs} \cdot \text{avg. of pair-occurences}}{\#\text{devices}} = \frac{224297359 \cdot 5.38}{500000} \approx 2413.5$$

As an implication, the number of elements in RDD_2 can be estimated as $2413.5\,|\Delta|$ and so the size of the whole RDD as $4826.88\,|\Delta|$.

RDD_3 **Number of distinct Apps and distinct App-Pairs** The size of RDD_3 depends on the amount of distinct apps and app-pairs. The amount of distinct apps means $|A|$, the number of different apps installed on all devices. Distinct app-pairs DP means the set of all pairs of apps which are installed together at least once all over the devices, so $DP = \{(a,b) | \,\text{installs}(a,b) \geq 1\}$ for all $a \neq b, a, b \in A$. RDD_3 contains one element per distinct App. For every distinct app-pair two elements in RDD_3 (element with the key of the two elements in the pair) are growing by two (app identifier and frequency). This leads to the following formula to calculate the size of RDD3:

$$\text{size of RDD3} = |A| + 4 \cdot |DP|$$

The number of distinct apps $|A|$ installed on devices in a input dataset Δ is upper limited by the number of existing apps $|M|$, it is $|A| \leq |M|$. If all existing devices would be in the input dataset and every existing app has at least one installation then $|A| = |M|$. Since in real use only a subset of all devices is present, it can be assumed that $|A| < |M|$. We can presume that the number of distinct apps is increasing with every additional device in a dataset,

		# Apps per Device			# Distinct App Occurrences			# App-Pair Occurrrences	
Name	# Devices	Avg.	SD	# Distinct Apps	Avg.	SD	# Distinct Pairs	Avg.	SD
1k	1040	60.97	36.49	14401	4.4	32.10	1511170	1.68	8.84
2k	1993	59.5	38.27	21686	5.47	49.49	2686434	1.79	12.36
3k	2957	59.16	36.98	28907	6.05	63.29	3636763	1.91	15.71
5k	5039	60.55	36.14	40800	7.48	92.65	5691629	2.12	22.14
10k	9978	60.41	38.86	65789	9.16	142.52	11117943	2.23	30.54
20k	19876	59.5	36.46	95872	12.34	235.54	17823879	2.62	48.11
50k	50096	59.99	37.68	164705	18.25	454.67	39659076	3.06	81.91
100k	100098	59.92	37.06	236422	25.37	758.70	66405422	3.61	126.62
200k	199811	59.97	37.88	334617	35.81	1270.91	116119719	4.18	190.47
300k	300304	59.95	37.59	404915	44.46	1738.12	155489092	4.66	247.77
400k	400276	59.95	37.64	460359	52.12	2172.71	191883428	5.04	297.28
500k	500000	59.95	37.61	507372	59.08	2584.53	224297359	5.38	343.35

Table 3.2: Data characteristics of the testdata subsets. All numbers are rounded to two digits after the decimal point.

but that this increase diminishes because a larger subset of all possible apps $|M|$ is already in Δ. The growth is limited by $|M|$. Because the number of distinct apps can easily be calculated, we can take $|A|$ as an input parameter for further analysis.

In contrast, the number of distinct app-pairs $|DP|$ in a specific input dataset cannot be easily calculated. This makes a further analysis of the relation between the known number of distinct apps $|A|$ and the unknown number of distinct app-pairs $|DP|$ beneficial. In the worst case, all apps are installed at least once with every other app, so that $\text{installs}(a, b) \geq 1$ for $a \neq b$ and all $a, b \in A$. For the test data this is not even remotely the case. In the 500k dataset there are 507 372 distinct apps and $p = 227\,297\,359$ app-pairs. So, only $p/|A|^2 = 8.71 \times 10^{-2}\%$ of the possible pairs are actually occurring. Figure 3.5 shows the relation with the data from table 3.2 on a plot with logarithmic scales on both axes. The green line represents a least squares fit of the function $y = x^b$ on the testdata: $|DP| = |A|^{1.462}$. The function was chosen as a generalization of the worst case $|DP| = |A|^2$. As an implication we can now give an estimator for the size of RDD_3: $|A| + 4 \cdot |A|^{1.462}$.

RDD_4 Dependency of K From RDD_3 to RDD_4 the K best scored entries in every list in RDD_3 are selected. As a consequence, the size of RDD_4 is $|A| \cdot K$.

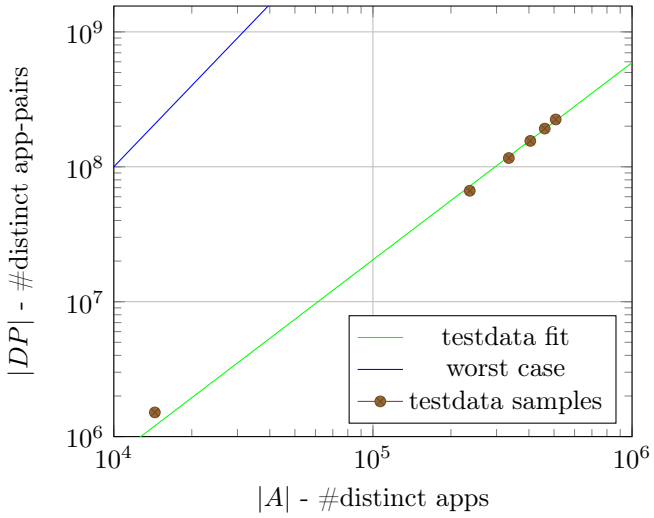

Figure 3.5: Relation $|A|$ and $|DP|$ on a log-log plot. Shows the worst case $|DP| = |A|^2$ and the fit of testdata samples $|DP| = |A|^{1.462}$.

Chapter 4

Optimization and Experiments

This chapter examines the algorithm developed in the last chapter regarding its performance. Section 4.1 explains influential optimizations already implemented and presents experiments to test their effectiveness. In section 4.2, conducted performance experiments are analysed to gain insights in the practical scalability of the algorithm. The last two sections present two different approaches for further optimizations.

4.1 Integral Optimizations

The algorithm developed in chapter 3 implements some optimizations and prevents some common pitfalls of programs implemented in Spark. Two important issues are discussed in this chapter.

4.1.1 Package name mapping

The implementation of the algorithm as shown in chapter 3.2 maps the package names to integers in the beginning and maps the integers back to the package name before it writes the result. This has two benefits: The amount of processed data is minimized whilst the predictability is maximized. The additional costs of collecting all distinct package names (see listing 3.1, line 4) are by far compensated by the savings of the further steps of the program. The amount of time gained was tested with subsets of the 500k testdata up to 100k on a cluster with a EC2 m1.xlarge (4 cores, 15 GB memory, 8 EC2 compute units) instance as master and a c3.xlarge instance as slave (4 cores, 7.5 GB, 14 EC2 compute units). Figure 4.1 shows performance while calculating the result for both implementations.

4.1.2 Wide dependency avoidance

Multiple combinations of spark *transformations* are possible to find a solution for the top-k problem. Some may be more obvious, especially when the use of the more complex *combineByKey transformation* is omitted. Figure 4.3 shows

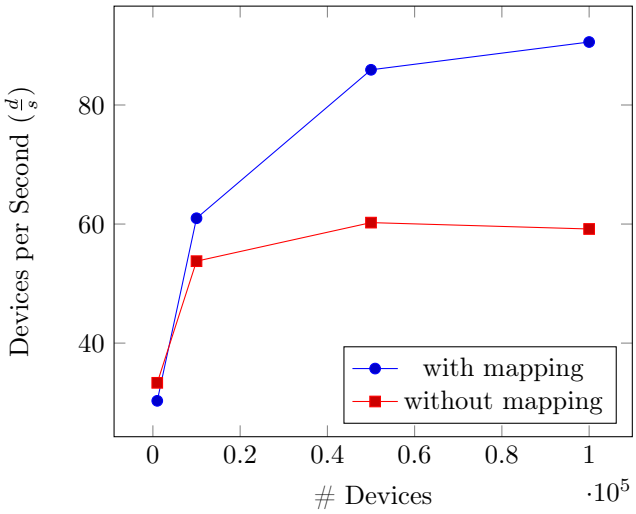

Figure 4.1: Performance relation between the algorithm with and without package name mapping

such an alternative option with the same setting as the example in figure 3.4. In a first step, the individual pairs are summed up in a *reduceByKey transformation* and are then combined by the respective app in the pair with the *groupByKey transformation*. During the execution, this implementation leads to three consecutive stages instead of two. The reason for this is that both *transformations*, *reduceByKey* and *groupByKey*, are creating wide dependencies between RDDs so that a shuffle of data between partitions is necessary. The difference was tested with small subsets of the 500k testdata on a cluster with a EC2 m1.xlarge (4 cores, 15 GB memory, 8 EC2 compute units) instance as master and a c3.xlarge instance as slave (4 cores, 7.5 GB, 14 EC2 compute units). Figure 4.2 shows the performance of both implementations. The lower complexity of the two stage approach is reflected by an about two times better performance.

4.2 Performance Experiments

To test the performance of the algorithm several experiments were conducted. Their goal is to investigate the scalability of the algorithm. For this reason, the implementation was run with differently sized input datasets on the same cluster. Additional experiments were conducted to investigate the algorithms' performance for different values of k and for different cluster configurations.

4.2.1 Input Sizes

The tests were conducted on different sized subsets of the test dataset on a cluster with a EC2 c3.xlarge (4 cores, 7.5 GB, 14 EC2 compute units) instance as master and three c3.2xlarge (8 cores, 15 GB, 28 EC2 compute units) instance as slave. For the test, k was set to 10, a common value for app recommendations. Figure 4.4 and table 4.1 show the results of this test.

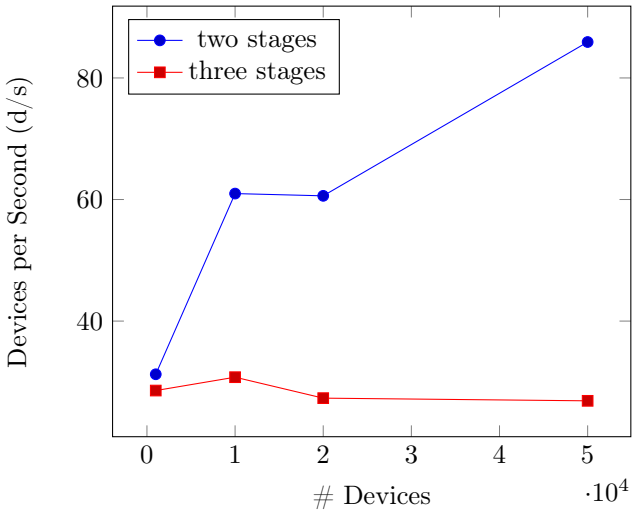

Figure 4.2: Performance relation between the algorithm in two stages and three stages with the sames hardware setting

	Frequency Score		Dependency Score	
# Devices	Runtime [s]	Performance $[\frac{d}{s}]$	Runtime [s]	Performance $[\frac{d}{s}]$
1040	14	74.286	16	65.000
100098	336	297.911	440	227.495
199811	678	294.706	946	211.217
300304	1086	276.523	1380	217.612
500000	1995	250.627	2280	219.298

Table 4.1: Algorithm performance with differently sized input datasets. The performance is indicated as devices per second ($\frac{d}{s}$).

Due to the additional effort of calculating the base probabilities, the dependency score version is constantly slower than the frequency score version. The much poorer performance for the 1k dataset can be explained with overhead costs for the start up, which become irrelevant with larger datasets. The datasets of more practical relevance with 100'000 and more devices showed a more or less stable performance with the dependency score and a slightly decreasing performance with the frequency score. The convergence of dependency and frequency score performance could be the consequence of a higher overhead at the dependency score originating from the additional actions for calculating the base probabilities.

4.2.2 Different values of k

The parameter k of the top-k problem accomodates different values depending on the application of the result. For example, *42 matters* uses the algorithm with $k > 100$ to preselect interesting app combinations and then processes the result with other methods. For this reason the relation between k and the performance of the algorithm is an interesting aspect, which is also examined with an experiment. The experiment was conducted with the the 100k dataset

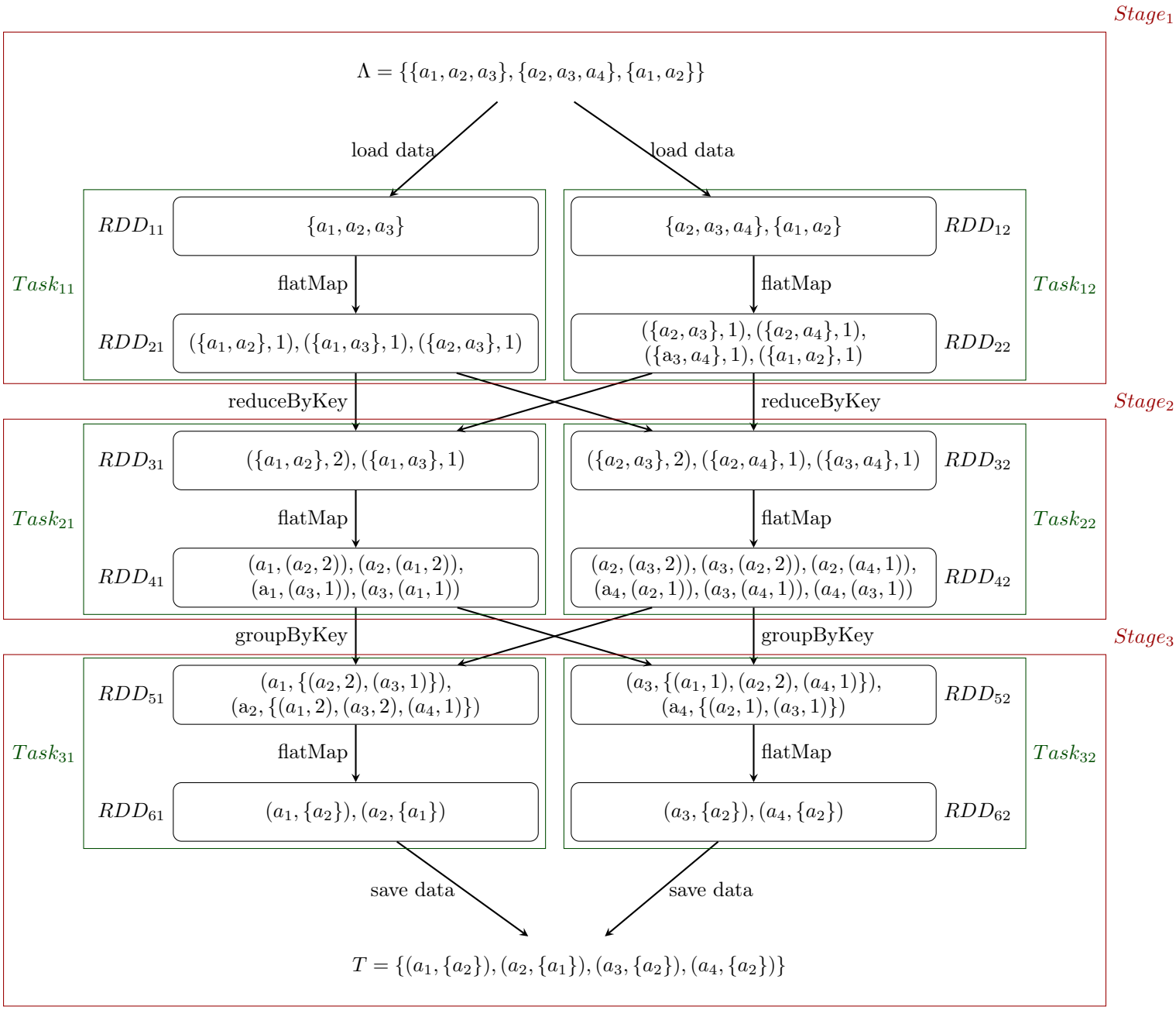

Figure 4.3: Algorithm with three stages instead of two. $k = 1$

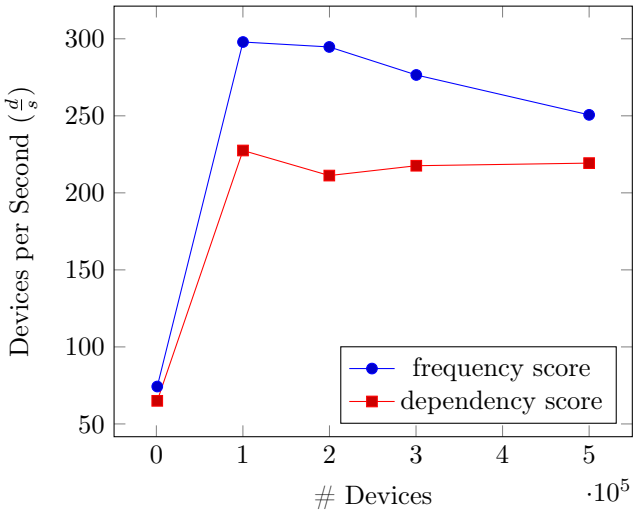

Figure 4.4: Algorithm performance with different sized input datasets.

on a cluster with a EC2 c3.xlarge (4 cores, 7.5 GB, 14 EC2 compute units) instance as master and three c3.2xlarge (8 cores, 15 GB, 28 EC2 compute units) instance as slave. The program was executed with $k = 1, 10, 100, 1000$ and for the frequency and dependency score. The results are visible on figure 4.5 with a log scale on the abscissa.

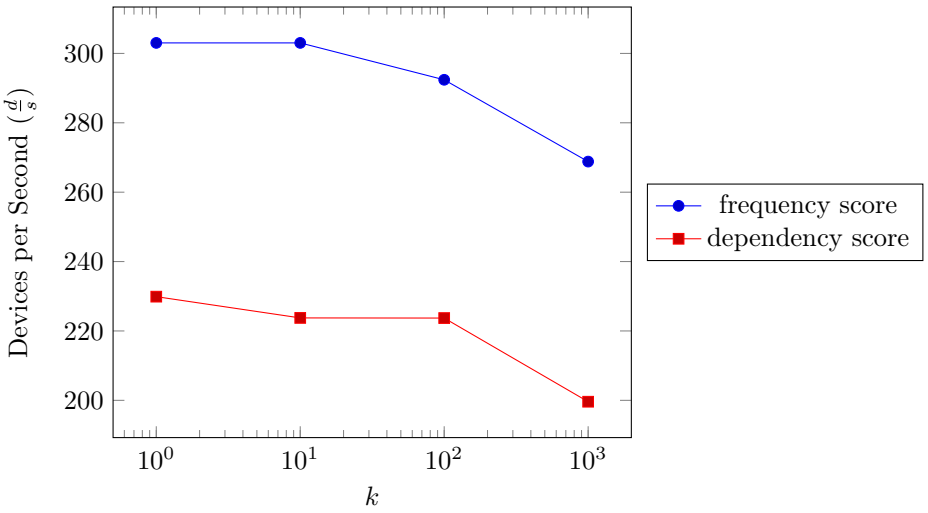

Figure 4.5: Algorithm performance with different k and the 100k dataset on the same cluster shown on a lin-log plot.

The experiment showed that the performance is affected only little by different values of k inside a practical relevant range lesser than thousand. For example, with the dependency score the performance loss in devices per second is 13.2% when $k = 1000$ instead of $k = 1$ for the tested scenario.

4.2.3 Cluster Size

The number of *executors* determines the number of *tasks* which can be computed in parallel. More *executors* lead to more but smaller partitions which therefore can be processed in less time. Still, the relation between the number of *executors* and the performance of the algorithm is not inversely proportional for one main reason: If an RDDs data is distributed on more executors, then more networking is necessary at Shuffle operations. Additionally, more *tasks* lead to a larger overhead. On the other side, too few *executors* and partitions lead to more memory pressure and the *executors* have to spill data to the local disk[1]. To examine the effect of different number of executors, the algorithm was executed on different EMR clusters. All clusters had a EC2 c3.xlarge (4 cores, 7.5 GB, 14 EC2 compute units) instance as master and one to three c3.2xlarge (8 cores, 15 GB, 28 EC2 compute units) instance as slave. Thus, the algorithm was tested on clusters with 8, 16, and 32 executors. Figure 4.6 shows the results of the experiment.

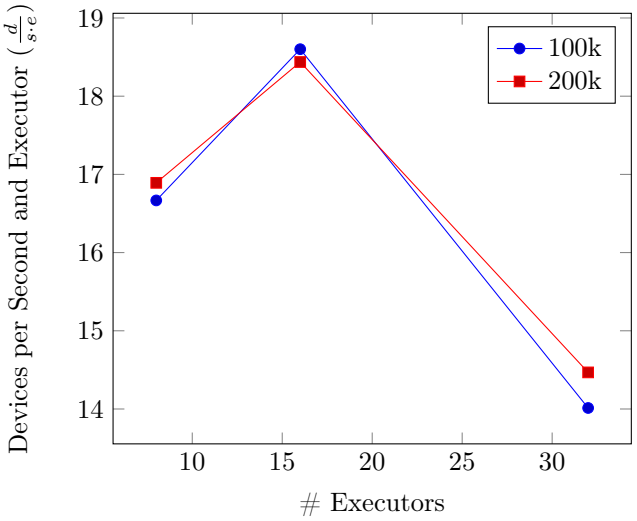

Figure 4.6: Experiments with different sized clusters, showing the relation between the number of *executors* and the performance in devices per second and *executor* ($\frac{d}{s \cdot e}$).

As the plot shows, the best performance per *executor* could be reached on the cluster with 16 executors. 8 *executors* as well as 32 *executors* lead to a poorer performance. In the case of 8 *executors* data was spilled to disk, because the partitions were too big to fit into the memory. In the case of 32 *executors* the additional networking lead to a poorer performance.

4.3 Pair Filtering

The example on figure 3.2 shows that from RDD_2 to RDD_3 a lot of pairs are created which are not part of the final result. In the example, a_1 is combined

[1]Spark implements an external sorting algorithm which if necessary spills data to the local disk while aggregating data in reduce *tasks*.

with a_2 and a_3, but only a_2 is part of RDD_4 and the result set, because $k = 1$, so only the best scored app is demanded. In the 500k testdata we have $p = 224'297'359$ distinct pairs and $|A| = 507'372$ distinct apps, if $k = 10$, the result will include 10 other apps per distinct app, so only $\frac{|A|*k}{p} \approx= 2.26\%$ distinct pairs are part of the result set. In this chapter, the idea of filtering out specific pairs at the point of their creation, based on the base probability, is discussed. To calculate all base probabilities (see chapter 1.2.2) in a dataset, no app-pairs have to be build, so the costs are relatively low. The base probabilities $P(a)$ of all apps in $a \in A$ are calculated first, and then used to decide whether a pair may be skipped at the moment of its creation.

In section 4.3.1 a proof of concept is made to tests the cost savings with pair filtering. In section 4.3.2, a metric is introduced to evaluate the approach. And finally, section 4.3.3 and 4.3.4 discuss and test pair filtering with the frequency and dependency score.

4.3.1 Proof of Concept

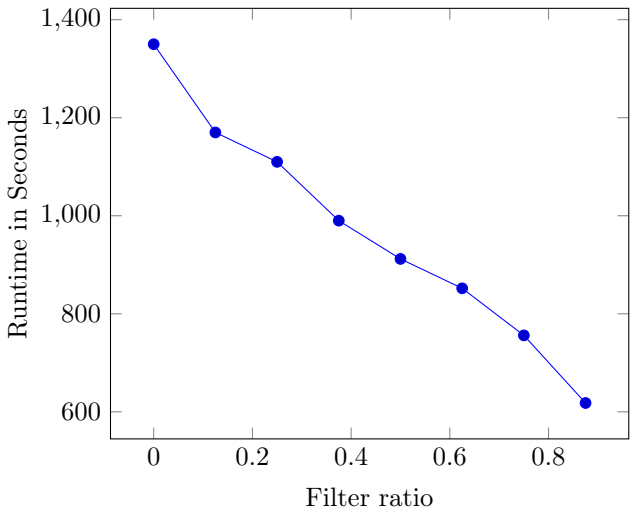

Figure 4.7: Pair filtering test. Runtime for the 100k testdata subset with different ratio of pairs filtered out.

The possible benefit of pair filtering is shown by an experiment. For the calculation of the 100k testdata subset, a filtering based on the hash of the pair was implemented. The program was executed while filtering different percentages of the pairs on a cluster with a EC2 m1.xlarge (4 cores, 15 GB memory, 8 EC2 compute units) instance as master and a c3.xlarge instance as slave (4 cores, 7.5 GB, 14 EC2 compute units). Figure 4.7 shows the results of this test. The test shows that the runtime may drop around 30% if half of the pairs could be filtered out. The final benefit also depends on the additional costs of deciding whether a pair is filtered out or not.

4.3.2 Metric

To test the correctness of different results, a new metric called *top-k equality* is introduced. The metric is defined as a function topKEquality$\colon L, \text{score} \rightarrow [0,1] \in \mathbb{Q}$, where L is the result of a specific top-k problem. The metric must address the fact that the top-k problem may have multiple correct solutions. For this reason along T a Q is defined as the union of all possible solutions of the top-k algorithm:

$$Q = \{(a_1, O_1), (a_2, O_2), ..., (a_n, O_n)\}$$

Every O_i referred to a_i is defined as the set of all apps $c \in A$ with a score(a_i, c) greater or equal to the minimum score in one equivalent R_i, this means that $|O_i| \geq k$ and for every element $b \in O_i$ holds that

$$\text{score}(a_i, b) \geq \min\{\text{score}(a_i, c) | c \in R_i\}$$

For a new result set $L = \{(a_1, U_1), (a_2, U_2), ..., (a_n, U_n)\}$ we define w as the ratio of apps included in U wich are also in O:

$$w = \frac{|U \cap O|}{|U|}$$

The metric is no defined as the arithmetic mean of all w:

$$\text{topKEquality}(L, \text{score}) \coloneqq \frac{1}{n} \sum_{i=1}^{n} \frac{|U_i \cap O_i|}{|U_i|}$$

If topKEquality$(L, score) = 1$, then L is a correct result for the top-k problem. Generally, topKEquality$(L, score) = v$ means that v of the top-k apps in L are correct.

For the example in section 1.2.1, it holds that

$$O = (a_1, \{a_2\}), (a_2, \{a_1, a_3\}), (a_3, \{a_2\}), (a_4, \{a_2, a_3\})\}$$

So if we take as an example the result set

$$L = \{(a_1, \{a_1\}), (a_2, \{a_3\}), (a_3, \{a_2\}), (a_4, \{a_1\})\}$$

then it is

$$\text{topKEquality}(L, \text{frequencyScore}) = \frac{1}{4}(\frac{0}{1} + \frac{1}{1} + \frac{1}{1} + \frac{0}{1}) = \frac{1}{2}$$

This means that half of the top-k apps in L are correctly classified with respect to the well defined top-k problem with the frequency score.

4.3.3 Frequency Score

To get an idea of how the base probabilities of apps in an app-pair are related to the presence of this app-pair in the top-k, figure 4.8 shows all app-pairs of the 1k data set with $k = 10$. Every app pair (c, d) is represented as two points. The first point at $(P(c), P(d))$ is turquoise if d is in the top-k of c, and the second

point $(P(d), P(c))$ is turquoise if c is in the top-k of d. It is noticeable that most of the top-k pairs are in the top-left corner, so that $P(a) \leq P(b)$[2]. All pairs in the top-k with $P(a) > P(b)$ are highlighted with a black circle, of 143988 top-k pairs only 296[3]. This means, if pairs with $P(a) > P(b)$ are skipped (about 42%), the result T would only slightly differ while the costs for calculation strongly decrease.

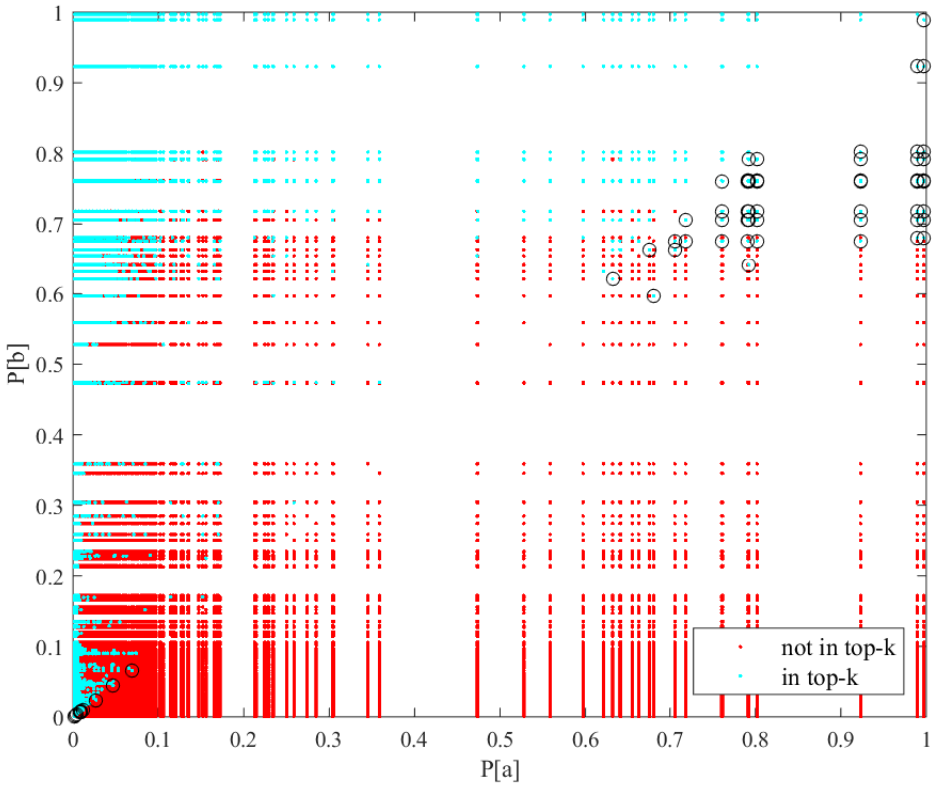

Figure 4.8: App-pairs in the 1k dataset. Coloring for $K = 10$ while applying the frequency score. The axes correspond to the base probability of the unique apps in the pair. Pairs in top-k and with $P(a) < P(b)$ are marked with a black circle.

The approach of filtering pairs was added in the implementation of the algorithm from 3.2. To test the results, a spark program was implemented which calculates the top-k equality of a result. The implementation was then tested on a cluster with an EC2 c3.xlarge (4 cores, 7.5 GB, 14 EC2 compute units) instance as master and three c3.2xlarge (8 cores, 15 GB, 28 EC2 compute units) as slaves. Table 4.2 shows the results of this test. The filtering of pairs caused a significant increase of the runtime whereas the correctness of results is preserved. Of 83044 apps in the 15k dataset only 55 do have an incorrect top-k list

[2]The stripes of points are the result of the the fact that just few apps have a high base probability. Therefore, most of the points are in the bottom left corner.

[3]These numbers and figure 4.8 are created by generating data about app-pairs with a Spark program and analysing them with Matlab.

	5k	50k	100k
Performance without filtering in $[\frac{d}{s}]$	86.201	219.298	268.817
Performance with filtering $[\frac{d}{s}]$	98.039	320.513	387.597
Top-k equality	0.999895	0.999898	0.999907
Ratio of incorrect top-k lists	$6.623e-4$	$6.337e-4$	$6.218e-4$

Table 4.2: Results of a frequency based pair-filtering tests with frequency score on two testdata subsets

(40800 apps and 26 wrong lists for 5k). This means that frequency based pair filtering may be a good approach to decrease costs of finding the top-k apps in a large dataset, depending on the required quality of the result.

4.3.4 Dependency Score

The same principle as for figure 4.2 with frequency score was used to create figure 4.9 with dependency score. The pattern of top-k points is dramatically different. The same filter as for the frequency score would miss about 29% of the top-k pairs and lead to a insufficient top-k equality. For this reason the filter proposed in the previous chapter can not be applied when using the dependency score.

4.4 App filtering

As already mentioned before, the most apps have just a few installations and just a few apps have many installations. To illustrate this relation, the 5k test data is used. In the 5k dataset the amount of apps on $|\Delta| = 5039$ devices is $|A| = 40800$. The app with the highest frequency[4] is installed on 5027 devices, so its base probability is $P(a) = \frac{5027}{5039} = 0.99762$. On the other hand, 26234, that is $\frac{26234}{40800} \approx 64.3\%$ of all apps are installed only once and therefore have a base probability of $(a) = \frac{1}{40800} \approx 0.00019845$. The histogram on figure 4.10 shows the base probability distribution of all apps in $|A|$ from the 5k testdata set. The histogram impressively shows that few apps are installed often.

If the frequency of an app in Δ is very low its top-k have a very poor significance. If for example the frequency score is used and an app is installed only once, then the top-k app for this app will be a random subset of all apps installed on the one device together with the app. For this reason the skipping of apps with a low base probability leads to a result with a greater significance whilst it also decreases the costs of calculating the result. Skipping of apps correspondingly decreases the number of apps for which the top-k apps is calculated with a specific dataset. In some applications it is preferable to have a bad result instead of no result. For this reason the app filter is implemented together with a new parameter f. f is the minimum frequency an app a must have in Δ to be considered, $f \leq \text{frequency}(a)$.

The app filter is implemented in while RDD_1 is created.

App filtering was then tested on a cluster with a EC2 c3.xlarge (4 cores, 7.5 GB, 14 EC2 compute units) instance as master and three c3.2xlarge (8 cores, 15

[4]The package name of this app is 'com.google.android.gms'. The app is called part of the *Google Play Service SDK* and pre-installed on every Device with apps from Google.

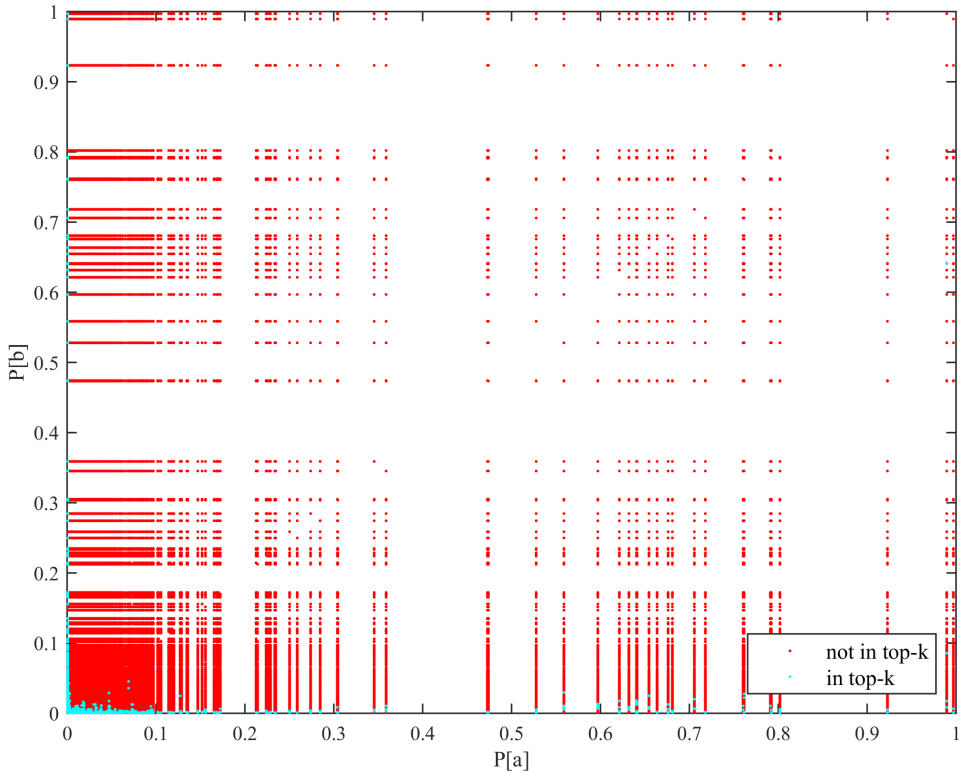

Figure 4.9: App-pairs in the 1k dataset. Coloring for $K = 10$ while applying the dependency score. The axes correspond to the base base probability of the unique apps in the pair.

GB, 28 EC2 compute units) instance as slave. Figure 4.11 shows the results of this test. $f = 2$ lead to a decreased runtime of about 13%, $f = 5$ to savings of around 21% and $f = 20$ to savings of around 34%. Corresponding to the test, the savings tend to slightly decrease with larger datasets, especially for high f.

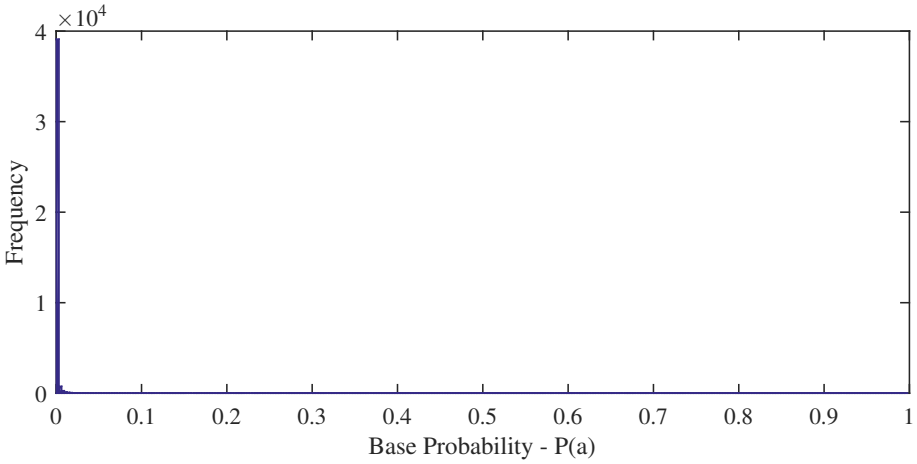

Figure 4.10: Histogram of base probabilities of all distinct apps in the 5k testdata subset.

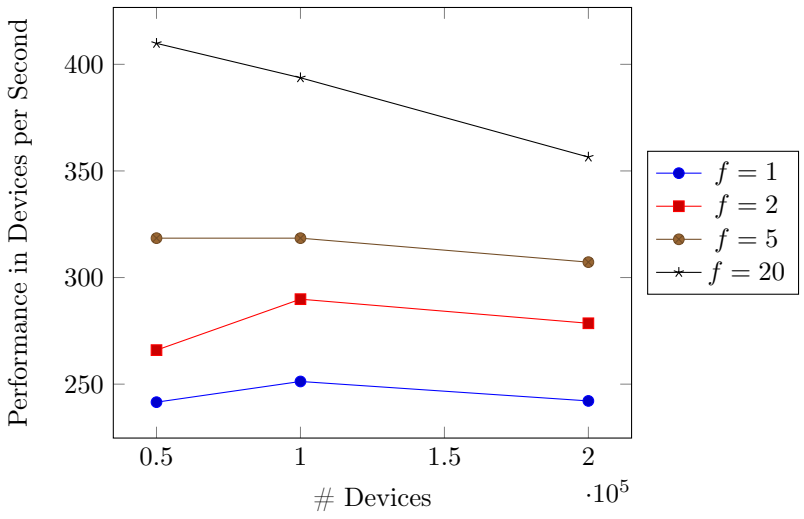

Figure 4.11: Performance relation for different f with app filtering, dependency score, and $k = 10$.

Chapter 5

Summary and Future Work

In this thesis, the challenge of finding the apps most frequently installed together in a large number of devices was formally defined as the top-k problem. With the frequency and the dependency score as a module of the top-k problem, two different versions of the problem were examined.

Based on Apache Spark, an algorithm was developed and implemented which solves the top-k problem for both scores and arbitrary input datasets.

To understand the algorithms' execution, Spark's execution model was investigated and illustrated with an abstraction independent of the underlying cluster manager. On this basis the execution of the algorithm has been understood and analysed. The analysis of the existing input data has shown that the algorithm has to manage about $n^{1.462}$ app pairs in an input dataset, where n is the number of apps. This means a much lower complexity than the expected n squared app pairs in the worst case.

To test the performance of the algorithm, experiments were conducted. On an EMR cluster provided by *AWS* billed with \$1.313 per hour[1] the processing of a dataset with 500'000 devices has been done in 33:15 minutes using the frequency score. This means that the costs can be estimated as about \$0.1455 per 100'000 devices[2].

Based on the knowledge of the input data, further optimizations were implemented and tested. To check the quality of optimizations which are modifying the result of the algorithm, a new metric called *top-k equality* was introduced. The metric was used to test pair filtering, a well-working optimization approach with the frequency score. Pair filtering lead to a performance gain of more than 40% in experiments. Additionally, the complete filtering of pairs with low frequencies lead to another essential performance gain. As a combined example: the same data as in the former paragraph with pair filtering and app filtering with $f = 5$ is processed in 17:50 minutes. This means a performance gain of more than 45% and a cost decrease to \$0.078 per 100'000 devices.

[1] Prices for *AWS* services are published online. EMR costs are visible at `https://aws.amazon.com/elasticmapreduce/pricing/`.

[2] When perusing these calculations, it has to be taken into account that EMR clusters are billed on an hourly base; this means that a full hour has to be paid even if the cluster is terminated after 30 minutes. It has also to be taken into account, that the costs do not linearly scale up with the devices.

5.1 Challenges and Future Work

The system architecture as preconditioned creates a very broad field of possible optimizations. The actual runtime of a Spark program on an EMR cluster depends on many factors like software versions and on configurations of different modules like Spark itself and the cluster manager. Besides, the interactions between Spark and the cluster manager and between different languages like Scala (Spark) and Python (Scala API) make understanding the different cost factors to a very challenging task.

Before the main challenges are summarized in the following subsections, some pitfalls which were uncovered in the work for this thesis are explained:

AWS Budget For this thesis a liberal budget provided by *42matters* was used to run tests and experiments on *AWS* infrastructure. Nevertheless, it is important to mention that experiments and testing on a cluster causes costs which have to be planned carefully. As the work for this thesis has shown, Spark runs stable if the slave nodes have at least 15 GB memory. This means that suitable tests with Spark programs cannot be made with cheap *AWS* instances, because they do not have enough memory.

Development Cycle of Used Technologies The work in this thesis highly depends on established software and services with a fast paced development cycle. The advantage of this situation is that a big community of different actors is improving the used technologies. For example, a new Spark version could bring a faster type of object serialization in Python which accelerates every Spark program using the Python API. In contrast, the fast development cycles brings with them the risk of highly integrated optimizations stop working soon after they have been developed, when new versions of the underlying technologies are released.

Developing on a Cluster The development of an algorithm with Spark running on a cluster is made further complicated by the fact that logs are often distributed over the cluster. This makes tracking down the source of a problem challenging. Additionally, errors are often not propagated up the stack, so multiple levels in the system have to be addressed to find an error. Because of this, it is a good idea to carefully study the system architecture and to write scripts to collect the most important logs.

5.1.1 Testing

The experiments in this thesis were done with a dataset of 500'000 devices at maximum. The tendencies of the results can be used to create estimators for larger datasets, but must not be overrated. Despite the investigation of the execution model, a lot of work is still hidden by Spark internals[3]. This leads to difficulties in predicting the behaviour of Spark programs and restricts the use of the results. Future work could make use of the extensive knowledge about the data presented in this thesis and create large synthetic test datasets to conduct

[3]One example for this is the Shuffle operation. Although it is often a bottleneck in Spark programs, it was not examined in every detail for this thesis, because it is enhanced in most subsequent versions of Spark and therefore frequently changes in its details.

elaborate performance tests. The results could then be used to predict *AWS* costs for *42matters* when their available data grows.

5.1.2 Performance and Optimization

In the work for this thesis, many promising fields for further optimization were found, but could not be pursued in detail. The following list introduces three such fields:

Domain Knwoledge The algorithm proposed in this thesis is a very common solution for the top-k problem. Its optimizations depend on the structure and characteristics of the input data. If the concrete use of the result is known, further optimizations of the algorithm may be possible. An example for a basic domain specific optimization is app filtering, explained in section 4.4.

Spark and Cluster Configuration Spark and its underlying infrastructure allow many different configurations. For instance, Spark allows the setting of parameters to ajust its handling of memory, network connections, cache and more[4]. These settings have a strong influence on the runtime of any program running on Spark. For example: If the amount of partitions emerging at a *wide dependency* does not fit the specific cluster, the consequence is a very suboptimal parallelization. Some experiments for this thesis showed that a higher number of partitions than *executors* often leads to less runtime.

Meta Data Based Filtering As introduced in section 2.3.1, additional meta data is available for many apps. An interesting approach would be to use this data to skip apps or app pairs which have no influence over the result in a similar way to the pair filtering approach in section 4.3. For example, one hypothesis could be that apps with a one star rating in the app store are never occurring in the top-k of other apps.

Scala or Java Spark API With the proposed algorithm a lot of time is used to serialize and deserialize objects with Pythons cPickle library, as a basic profiling showed. One would expect that the use of Spark's Scala or Java API could accelerate the algorithm, because their serialization has a better performance. This assumption could be checked in future work.

5.1.3 Further Topics

Besides testing and optimizations two other possible topics of future work emerged:

Cost Optimization on AWS The infrastructure provided by *AWS* is charged at hourly rates. For instance, this means that a EMR cluster costs the same amount of money no matter whether it is running 61 minutes or 120 minutes. A further cost optimization idea is to use the extensive knowledge about the algorithm to chose a cluster configuration that makes use of the hourly rated infrastructure insofar as the cluster can be determined shortly before the next running hour has to be paid.

[4]Spark's many configuration parameters can be found at `https://spark.apache.org/docs/latest/configuration.html`

Other Distributed Frameworks The use of Spark for finding the apps most frequently installed is not questioned in this thesis. It is well possible that a solution for the top-k problem implemented for other distributed programming models and frameworks (e.g. Apache Flink) may better fit the requirements of *42matters* and other users. One reason for the choice of Spark was its in-memory approach; but the advantages of this approach could not have been used, because for the top-k problem no iterative calculation with the same dataset have to be done.

Bibliography

[1] M.-S. Chen, J. Han, and P. S. Yu, "Data mining: An overview from a database perspective," *Knowledge and data Engineering, IEEE Transactions on*, vol. 8, no. 6, pp. 866–883, 1996.

[2] C. Yadav, S. Wang, and M. Kumar, "An approach to improve apriori algorithm based on association rule mining," in *Computing, Communications and Networking Technologies (ICCCNT), 2013 Fourth International Conference on*, IEEE, 2013, pp. 1–9.

[3] R. Chang and Z. Liu, "An improved apriori algorithm," in *Electronics and Optoelectronics (ICEOE), 2011 International Conference on*, IEEE, vol. 1, 2011, pp. V1–476.

[4] R. Sumithra and S. Paul, "Using distributed apriori association rule and classical apriori mining algorithms for grid based knowledge discovery," in *Computing Communication and Networking Technologies (ICCCNT), 2010 International Conference on*, IEEE, 2010, pp. 1–5.

[5] (Aug. 2015). Cluster mode overview, [Online]. Available: `https://spark.apache.org/docs/latest/cluster-overview.html`.

[6] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, vol. 10, 2010.

[7] (Jul. 2015). Spark programming guide, [Online]. Available: `https://spark.apache.org/docs/latest/programming-guide.html`.

[8] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. 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*, USENIX Association, 2012.

[9] H. Karau, A. Konwinski, P. Wendell, and M. Zaharia, *Learning Spark: Lightning-Fast Big Data Analysis*. " O'Reilly Media, Inc.", 2015.

[10] (Jul. 2015). Spark 1.3.1 python api docs, [Online]. Available: `http://spark.apache.org/docs/1.3.1/api/python/`.

[11] S. Ryza. (2015). How-to: Tune your apache spark jobs (part 1), [Online]. Available: `http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-1/`.

Appendices

Appendix A

Technical Documentations

The following two sections show the main steps used to develop the programs for this thesis.

A.1 Starting a Spark Program

Spark programs are developed locally and then send to a EMR cluster. With a EMR Step they are started on the EMR Cluster. The following steps can be pursued to run a Spark program on a EMR cluster. The steps assume that an SSH keyfile to access the EMR cluster is available at *$SSH_KEY*.

1. Start the EMR cluster with the desired configuration via AWS console in an internet browser[1]. The experiments in this thesis have shown that the machines should have at least 15 GB of main memory to stably run Spark programs. Choose Spark as a pre-installed framework on the cluster.

2. Wait until the AWS console shows the status *read*. Then lookup the *Master Public DNS*, *Jobflow ID* and the *Region* of the cluster on the cluster detail page on the AWS console.

3. Use *Secure copy* to upload the Python program to the master:
```
#  scp -i $SSH_KEY $LOCAL_FILE_PATH \
hadoop@$MASTER_PUBLIC_DNS:$SERVER_FILE_ PATH
```

4. Add an EMR Step which starts the spark-submit script at the master which in turn starts the Spark program. Having a well configured *AWS Command Line Interface* installed is a precondition[2]. Use *Secure copy* to upload the Python program to the master:
```
#  aws emr add-steps -cluster-id $JOBFLOW_ID -steps \
Name=ASparkStep, \
Jar=s3://$REGION.elasticmapreduce/libs/script-runner/script-runner.jar, \
Args=[/home/hadoop/spark/bin/spark-submit,-master,yarn-client, \
$SERVER_FILE_PATH],ActionOnFailure=CONTINUE
```

[1] https://aws.amazon.com/elasticmapreduce/
[2] https://aws.amazon.com/cli/

5. Optionally add more EMR Steps by repeating the last step. Then wait until all steps are *Completed* or *Failed*.

6. Terminate the cluster at the AWS console. Not terminating a cluster leads to unnecessary costs.

A.2 Debugging a Spark Program

While a cluster is running, different useful information about the cluster and the Spark program are available.

Web Interface While a Spark program is running, a web interface providing information about action, stages and tasks is available. It can be reached under *$MASTER_PUBLIC_DNS:4040*. The same information of already finished Spark programs is available by the *history server*, its adress is *$MASTER_PUBLIC_DNS:18080*.

Log files Useful log files are available at different places. The most important once are collected on the HDFS of the cluster and can be viewed at *$MASTER_PUBLIC_DNS:9026*.

To get access to the different web interfaces SSH tunneling with dynamic port forwarding to the master node has to be established. Additionaly, the internet browser has be configured using this tunnel when accessing the *$MASTER_PUBLIC_DNS*. AWS provides extensive documentation for these purposes[3].

[3] http://docs.aws.amazon.com/ElasticMapReduce/latest/DeveloperGuide/emr-web-interfaces.html

Appendix B

Source Code

The main Python Programs developed in the work for this thesis are printed in this appendix. A technical documentation how the programs can be started on an EMR cluster can be found at appendix A.

B.1 Frequency Score

The following Python program for Spark was used to find the top-k apps with the frequency score:

```
from pyspark import SparkContext, SparkConf
from collections import defaultdict
import json
import cStringIO
import heapq

# Input Parameters
INPUT_PATH = "s3://..."
OUTPUT_PATH = "s3://..."
K = 10

# Initiate SparkContext
conf = SparkConf().setAppName("AppsFrequentlyInstalledTogether")
sc = SparkContext(conf=conf)

# Lambda -> RDD1, Data loading and preparation
RDD1 = sc.textFile(INPUT_PATH)

RDD1 = RDD1.map(lambda x: [x['pn'] for x in json.loads(x)['apps']])
    .cache()

app_list = RDD1.flatMap(lambda x: x).distinct().collect()
pn_int_list = sc.broadcast(dict((n, i) for i, n in enumerate(
    app_list)))
int_pn_list = sc.broadcast(dict(zip(pn_int_list.value.values(),
    pn_int_list.value.keys())))

RDD1 = RDD1.map(lambda x: [pn_int_list.value[y] for y in x])

# RDD1 -> RDD2, Pair creation
def pairsInList(elements):
    pairs = []
    for i in xrange(0, len(elements)):
```

```
        for j in xrange(i + 1, len(elements)):
            pairs.append((elements[i], elements[j]))
            pairs.append((elements[j], elements[i]))
    return pairs

RDD2 = RDD1.flatMap(pairsInList)

# RDD2 -> RDD3, Pair combination
def createCombiner(firstPair):
    pairDict = defaultdict(int)
    pairDict[firstPair] = 1
    return pairDict

def mergeValue(pairDict, newpair):
    pairDict[newpair] += 1
    return pairDict

def mergeCombiners(pairDictA, pairDictB):
    res = pairDictB.copy()
    for app, counter in pairDictA.iteritems():
        res[app] += counter
    return res

RDD3 = RDD2.combineByKey(createCombiner, mergeValue, mergeCombiners)

# RDD3 -> RDD4, Pair selection
def sortAndSelect(pairDict):
    topK = heapq.nlargest(K, pairDict.iteritems(), key=lambda x: x[1])
    return topK

RDD4 = RDD3.mapValues(sortAndSelect)

# RDD4 -> T, Data saving
def listToJson(device):
    jsonString = cStringIO.StringIO()
    jsonString.write('{"package_name":"')
    jsonString.write(int_pn_list.value[device[0]])
    jsonString.write('","pairs_list":[')
    first = True
    for pair in device[1]:
        if first:
            first = False
            jsonString.write('{"pn":"')
        else:
            jsonString.write(',{"pn":"')
        jsonString.write(int_pn_list.value[int(pair[0])])
        jsonString.write('"}')
    jsonString.write("]}")
    return jsonString.getvalue()

RDD4.map(listToJson).saveAsTextFile(OUTPUT_PATH)

sc.stop()
```

python–frequency–score.py

B.2 Dependency Score

The following Python program for Spark was used to find the top-k apps with the dependency score:

```
from pyspark import SparkContext, SparkConf
from collections import defaultdict
import json
import cStringIO
import heapq

# Input Parameters
INPUT_PATH = "s3://..."
OUTPUT_PATH = "s3://..."
K = 10

# Initiate SparkContext
conf = SparkConf().setAppName("AppsFrequentlyInstalledTogether")
sc = SparkContext(conf=conf)

# Lambda -> RDD1, Data loading and preparation
RDD1 = sc.textFile(INPUT_PATH)

RDD1 = RDD1.map(lambda x: [x['pn'] for x in json.loads(x)['apps']])
    .cache()

# Generate local dictionary with package names as keys
# and app frequencies as values.
num_of_devices = RDD1.count()

app_frequency_list = RDD1 \
    .flatMap(lambda x: map(lambda x: (x,1), x)) \
    .reduceByKey(lambda x, y: x + y) \
    .mapValues(lambda x: float(x) / num_of_devices) \
    .collectAsMap()
pn_int_list = {}
int_pn_list = {}
for i, n in enumerate(app_frequency_list.iteritems()):
    pn_int_list[n[0]] = (i, n[1])
    int_pn_list[i] = (n[0], n[1])

# Broadcast the dictionaries
pn_int_list = sc.broadcast(pn_int_list)
int_pn_list = sc.broadcast(int_pn_list)

RDD1 = RDD1.map(lambda x: [pn_int_list.value[y][0] for y in x])

# RDD1 -> RDD2, Pair creation
def pairsInList(elements):
    pairs = []
    for i in xrange(0, len(elements)):
        for j in xrange(i + 1, len(elements)):
            pairs.append((elements[i], elements[j]))
            pairs.append((elements[j], elements[i]))
    return pairs

RDD2 = RDD1.flatMap(pairsInList)

# RDD2 -> RDD3, Pair combination
def createCombiner(firstPair):
    pairDict = defaultdict(int)
    pairDict[firstPair] = 1
```

```
57      return pairDict

59  def mergeValue(pairDict, newpair):
        pairDict[newpair] += 1
61      return pairDict

63  def mergeCombiners(pairDictA, pairDictB):
        res = pairDictB.copy()
65      for app, counter in pairDictA.iteritems():
            res[app] += counter
67      return res

69  RDD3 = RDD2.combineByKey(createCombiner, mergeValue, mergeCombiners
        )

71  # RDD3 -> RDD4, Pair selection
    def sortAndSelect(e):
73      global num_of_devices

75      appA = e[0]
        pairDict = e[1]
77      appAFrequency = int_pn_list.value[appA][1]

79      # select the K elements with the largest score
        topK = heapq.nlargest(K, pairDict.iteritems(), \
81          key=lambda appB: (float(appB[1]) / num_of_devices) / float(
        appAFrequency*int_pn_list.value[appB[0]][1]))
        return (appA, topK)
83
    RDD4 = RDD3.map(sortAndSelect)
85
    # RDD4 -> T, Data saving
87  def listToJson(dev):
        jsonString = cStringIO.StringIO()
89      jsonString.write('{"package_name":"')
        jsonString.write(int_pn_list.value[dev[0]][0])
91      jsonString.write('","pairs_list":[')
        first = True
93      for p in dev[1]:
            if first:
95              first = False
                jsonString.write('{"pn":')
97          else:
                jsonString.write(',{"pn":')
99          jsonString.write(int_pn_list.value[int(p[0])][0])
            jsonString.write('"}')
101     jsonString.write("]}")
        return jsonString.getvalue()
103
    RDD4.map(listToJson).saveAsTextFile(OUTPUT_PATH)
105
    sc.stop()
```

python–dependency–score.py

B.3 Pair and App Filter

The following Python program implements the app and pair filter approaches with the frequency score:

```
from pyspark import SparkContext, SparkConf
import json
import cStringIO
import heapq
from collections import defaultdict

# Input Parameters
INPUT_PATH = "s3://..."
OUTPUT_PATH = "s3://..."
K = 10
F = 5

# Initiate SparkContext
conf = SparkConf().setAppName("AppsFrequentlyInstalledTogether_FS")
sc = SparkContext(conf=conf)

# Lambda -> RDD1, Data loading and preparation
RDD1 = sc.textFile(INPUT_PATH)

RDD1 = RDD1.map(lambda x: [x['pn'] for x in json.loads(x)['apps']])
    .cache()

# Generate local dictionary with package names as keys
# and app frequencies as values.
num_of_devices = RDD1.count()
min_base_probability = float(F) / num_of_devices

app_frequency_list = RDD1 \
    .flatMap(lambda x: map(lambda x: (x, 1), x)) \
    .reduceByKey(lambda x, y: x + y) \
    .mapValues(lambda x: float(x) / num_of_devices) \
    .collectAsMap()
pn_int_list = {}
int_pn_list = {}
for i, n in enumerate(app_frequency_list.iteritems()):
    pn_int_list[n[0]] = (i, n[1])
    int_pn_list[i] = (n[0], n[1])

# Broadcast the dictionaries
pn_int_list = sc.broadcast(pn_int_list)
int_pn_list = sc.broadcast(int_pn_list)


# Filter based on F and the P(a)
def mapAndFilter(dev):
    apps = []
    for appPN in dev:
        v = pn_int_list.value[appPN]
        if v[1] >= min_base_probability:
            apps.append(v[0])
    return apps

RDD1 = RDD1.map(mapAndFilter)

# RDD1 -> RDD2, Pair creation
def pairsInList(elements):
    pairs = []
    for i in xrange(0, len(elements)):
        for j in xrange(i + 1, len(elements)):
            je = elements[j]
            jeP = int_pn_list.value[je][1]
            ie = elements[i]
```

```
62              ieP = int_pn_list.value[ie][1]
                if(jeP >= ieP):
64                  pairs.append((ie, je))
                if(jeP <= ieP):
66                  pairs.append((je, ie))

68      return pairs

70  RDD2 = RDD1.flatMap(pairsInList)

72  # RDD2 -> RDD3, Pair combination
    def createCombiner(firstPair):
74      pairDict = defaultdict(int)
        pairDict[firstPair] = 1
76      return pairDict

78  def mergeValue(pairDict, newpair):
        pairDict[newpair] += 1
80      return pairDict

82  def mergeCombiners(pairDictA, pairDictB):
        res = pairDictB.copy()
84      for app, counter in pairDictA.iteritems():
            res[app] += counter
86      return res

88  RDD3 = RDD2.combineByKey(createCombiner, mergeValue, mergeCombiners
        )

90  # RDD3 -> RDD4, Pair selection
    def sortAndSelect(pairDict):
92      topK = heapq.nlargest(K, pairDict.iteritems(), key=lambda x: x
        [1])
        return topK
94
    RDD4 = RDD3.mapValues(sortAndSelect)
96
    # RDD4 -> T, Data saving
98  def listToJson(dev):
        jsonString = cStringIO.StringIO()
100     jsonString.write('{"package_name":"')
        jsonString.write(int_pn_list.value[dev[0]][0])
102     jsonString.write('","pairs_list":[')
        first = True
104     for p in dev[1]:
            if first:
106             first = False
                jsonString.write('{"pn":')
108         else:
                jsonString.write(',{"pn":')
110         jsonString.write(int_pn_list.value[int(p[0])][0])
            jsonString.write('"}')
112     jsonString.write("]}")
        return jsonString.getvalue()
114

116 RDD4.map(listToJson).saveAsTextFile(OUTPUT_PATH)

118 sc.stop()
```

python–frequency–pairfilter–appfilter.py

B.4 Top-K Equality

To calculate the *top-k equality* two programs were used. The first program created the extended solution Q (see 4.3.2) for a specific input dataset and the second one calculated the *top-k equality* by comparing a arbitrary result with Q:

```
from pyspark import SparkContext, SparkConf
import json
import cStringIO
from collections import defaultdict
from itertools import imap, izip, tee
from operator import itemgetter

# Input Parameters
INPUT_PATH = "s3://..."
OUTPUT_PATH = "s3://..."
K = 10

# Initiate SparkContext
conf = SparkConf().setAppName("
    AppsFrequentlyInstalledTogether_createQ")
sc = SparkContext(conf=conf)

# Lambda -> RDD1, Data loading and preparation
RDD1 = sc.textFile(INPUT_PATH)

RDD1 = RDD1.map(lambda x: [x['pn'] for x in json.loads(x)['apps']])
    .cache()

app_list = RDD1.flatMap(lambda x: x).distinct().collect()
pn_int_list = sc.broadcast(dict((n, i) for i, n in enumerate(
    app_list)))
int_pn_list = sc.broadcast(dict(zip(pn_int_list.value.values(),
    pn_int_list.value.keys())))

RDD1 = RDD1.map(lambda x: [pn_int_list.value[y] for y in x])

# RDD1 -> RDD2, Pair creation
def pairsInList(elements):
    pairs = []
    for i in xrange(0, len(elements)):
        for j in xrange(i + 1, len(elements)):
            pairs.append((elements[i], elements[j]))
            pairs.append((elements[j], elements[i]))
    return pairs

RDD2 = RDD1.flatMap(pairsInList)

# RDD2 -> RDD3, Pair combination
def createCombiner(firstPair):
    pairDict = defaultdict(int)
    pairDict[firstPair] = 1
    return pairDict

def mergeValue(pairDict, newpair):
    pairDict[newpair] += 1
    return pairDict

def mergeCombiners(pairDictA, pairDictB):
    res = pairDictB.copy()
    for app, counter in pairDictA.iteritems():
```

```
52          res[app] += counter
        return res
54
RDD3 = RDD2.combineByKey(createCombiner, mergeValue, mergeCombiners
    )
56
"""
58 This function returns a list with the n largest elements from the
    dataset defined by 'iterable.''
   'key' specifies a function of one argument that is used to extract
    a comparison
60 key from each element in the iterable

62 The solution also includes all elements equal to the n-th element.

64 Adapted by
   https://github.com/apache/spark/blob/branch-1.3/python/pyspark/
    heapq3.py
66 """
   def nlargest_extended(n, iterable, key):
68      def count(start=0, step=1):
            n = start
70          while True:
                yield n
72              n += step

74      def _siftdown_max(heap, startpos, pos):
            newitem = heap[pos]
76          while pos > startpos:
                parentpos = (pos - 1) >> 1
78              parent = heap[parentpos]
                if parent < newitem:
80                  heap[pos] = parent
                    pos = parentpos
82                  continue
                break
84          heap[pos] = newitem

86      def _siftup_max(heap, pos):
            endpos = len(heap)
88          startpos = pos
            newitem = heap[pos]
90          childpos = 2*pos + 1    # leftmost child position
            while childpos < endpos:
92              rightpos = childpos + 1
                if rightpos < endpos and not heap[rightpos] < heap[
    childpos]:
94                  childpos = rightpos
                heap[pos] = heap[childpos]
96              pos = childpos
                childpos = 2*pos + 1
98          heap[pos] = newitem
            _siftdown_max(heap, startpos, pos)
100
        def heappop_max(heap):
102         lastelt = heap.pop()
            if heap:
104             returnitem = heap[0]
                heap[0] = lastelt
106             _siftup_max(heap, 0)
            else:
108             returnitem = lastelt
```

```
        return returnitem

    def heappush_max(heap, item):
        heap.append(item)
        _siftdown_max(heap, 0, len(heap)-1)

    in1, in2 = tee(iterable)
    it = izip(imap(key, in1), count(0, -1), in2)
    h = []
    for i in it:
        heappush_max(h, i)
    res = []
    last = heappop_max(h)
    res.append(last)
    while h:
        next = heappop_max(h)
        if len(res) < n or next[0] == last[0]:
            last = next
            res.append(last)
        else:
            break
    res = map(itemgetter(2), res)

    return res

def sortAndSelect(pairDict):
    topK = nlargest_extended(K, pairDict.iteritems(), key=lambda x:
     x[1])
    return topK

RDD4 = RDD3.mapValues(sortAndSelect)

# RDD4 -> T, Data saving
def listToJson(dev):
    jsonString = cStringIO.StringIO()
    jsonString.write('{"package_name":"')
    jsonString.write(int_pn_list.value[dev[0]][0])
    jsonString.write('","pairs_list":[')
    first = True
    for p in dev[1]:
        if first:
            first = False
            jsonString.write('{"pn":')
        else:
            jsonString.write(',{"pn":')
        jsonString.write(int_pn_list.value[int(p[0])][0])
        jsonString.write('"}')
    jsonString.write("]}")
    return jsonString.getvalue()

RDD4.map(listToJson).saveAsTextFile(OUTPUT_PATH)

sc.stop()
```

python–createQ.py

```
from pyspark import SparkContext, SparkConf, MarshalSerializer
import json


DATASET_PATH = "s3://..."
Q_PATH = "s3://..."
```

```
# Local filename to write the results
RESULT_OUTPUT_PATH = "/..."

conf = SparkConf().setAppName("
    AppsFrequentlyInstalledTogether_topKEquality")
sc = SparkContext(serializer=MarshalSerializer(), conf=conf)

f = open(RESULT_OUTPUT_PATH, "wb")

equalityCounter = sc.accumulator(0)
numberOfIncorrectTopKLists = sc.accumulator(0)

def func(JSONLine):
    app_data = json.loads(JSONLine)
    return (app_data["package_name"], [x['pn'] for x in app_data['
    pairs_list']])

output_A = sc.textFile(d[0]).map(func).cache()
output_B = sc.textFile(d[1]).map(func).cache()

output_AB = output_A.fullOuterJoin(output_B)

def calculateEquality(AB_joined):
    global equalityCounter, numberOfIncorrectTopKLists

    if AB_joined[1][0] == None:
        A = set()
    else:
        A = set(AB_joined[1][0])

    if AB_joined[1][1] == None:
        B = set()
    else:
        B = set(AB_joined[1][1])

    lenA = len(A)

    if lenA == 0:
        equality = 1
    else:
        n = len(A.intersection(B))
        equality = float(n) / lenA

    if equality < 1.0:
        numberOfIncorrectTopKLists += 1

    equalityCounter += equality

output_AB.foreach(calculateEquality)
ab_size = output_AB.count()

f.write("Average Subset Ratio \n")
f.write("A: {0}, length: {1} \n".format(d[0], output_A.count()))
f.write("B: {0}, length: {1} \n".format(d[1], output_B.count()))
f.write("Full Outer Join Size: {0}\n".format(ab_size))
f.write("Similarity: {0}\n".format(
jaccardCounter.value / ab_size))
f.write("Number of incorrect Top-K Lists: {0}\n\n".format(
    numberOfIncorrectTopKLists.value))
f.flush()
```

```
66 sc.stop()
```

python–topKEquality.py