# Spark

## Hadoop

פלטפורמת קוד פתוח לעיבוד מידע בנפחים גדולים. היתרונות הגדולים שהופכים אותה למאוד פופולרית הם שמאפשרת מספר שיטות לעיבוד מקבילי, יכולה להתמודד עם כמות הולכת וגדלה של נתונים (scalability), גמישה וחינמית. מורכבת משני חלקים: HDFS ו-Map Reduce.

### HDFS - Hadoop Distributed File System

מערכת קבצים המסוגלת לשמור מידע רב באמינות ושרידות גבוהה. מחלקת את המידע לבלוקים ושומרת אותם ב-nodes. לכל בלוק יש כמה עותקים כך שאם node אחד נפל או שאינו זמין יש node אחר שממנו אפשר לשלוף את הבלוק.

### Map-Reduce

זוהי שיטת עיבוד שבה מריצים מספר אלגוריתמים במערכת מבוזרת המאפשרת לבצע חישוב מקבילי על כמות גדולה של מידע. באמצעות האלגוריתמים מחלקים את המידע למספר מחשבים (workers), כל מחשב מקבל חלק מהמידע ומבצע עליו חישוב. מאחדים את כל המסקנות ומחזירים תוצאה. כל תבנית זו מחולקת לשלושה שלבים, שכל שלב הוא אלגוריתם אחר:

1. Map - האלגוריתם הראשון תפקידו לסנן את המידע ולקחת רק את החלקים עליהם צריך להפעיל פעולת חישוב. בנוסף מבצע מיפוי על המידע הנותר ל-key ו-value, כך שנוכל לגשת אליו בצורה נוחה.
2. Sort/Grouping - מפרק את המידע לקבוצות הגיוניות, ומקצה לכל worker קבוצת מידע.
3. Reduce - כל worker מפעיל אלגוריתם זה על המידע שקיבל ומחזיר את התוצאה.

בחלק מהמקרים איחוד המידע שמוחזר מכל worker הוא עצמו התשובה, אמנם ישנם מקרים בהם צריך אלגוריתם נוסף שיעבד את כל המידע שהתקבל ויחזיר את התשובה. שלב זה נקרא finalize.

דוגמה לשימוש ב-Map Reduce. נתון מסד נתונים המחזיק מידע על סטודנטים כמו: ID, שם פרטי ומשפחה, גיל, כתובת וממוצע ציונים. נרצה למצוא את הציון הממוצע לכל שכבת גיל. בשלב ה-map נמפה בין הגיל לציון ונסנן את כל שאר המידע. בשלב ה-Grouping נאחד את הזוגות עם אותו גיל ונשלח ל-worker כלשהו. בשלב ה-Reduce כל worker ימצא את הממוצע ציונים בקבוצת הסטודנטים שקיבל ויחזיר את המידע. במידה וקבוצת הסטודנטים בגיל מסוים גדולה מכדי לשלוח את כולה ל-worker אחד, היא תפוצל כך שכל worker יחשב את הממוצע של הקבוצה שקיבל. לבסוף מכל הערכים שהוחזרו יעשה ממוצע ביחס לגודל כל קבוצה.

כיוון שמסדי נתונים מסוג NoSQL עובדים במערכת מבוזרת על כמויות גדולות של מידע, רבים מהם מאפשרים למתכנת לבצע Map-Reduce.

## Spark

ספארק הוא מסגרת תוכנה המשמש כדי לבצע חישובים על כמות גדולה של מידע במערכת המבוזרת למחשבים רבים. ספארק פותח לתת מענה למגבלות הקיימות בתבנית Map-Reduce אשר מאפשר רק עיבוד מבוסס batch ולא עיבוד בזמן אמת. כלומר, מחכה שיהיה לו כמות מסוימת של מידע לפני שמעבד אותו. ספארק לעומת זאת מאפשר גם עיבוד בזמן אמת ולא מחכה למידע נוסף. ספארק מספק עיבוד מהיר מאוד, מנהל זיכרון בצורה מיטבית, גמיש, ונוח לפתח בו. בנוסף, מאפשר להפעיל אלגוריתמים שחוזרים על עצמם מספר פעמים, וניתן לעשות בו עיבודים של ML.

בדרך כלל נשתמש בספארק במקרה שיש לנו המון מכונות או ליבות שהמידע מפוזר ביניהם. השימוש יהיה בדרך כלל באמצעות מערכת הקבצים Hadoop, שהיא מערכת קבצים מאוד מבוזרת.

ספארק חושף ממשק (API) למפתחים המתבסס על מבני נתונים הנקראים RDD ו-Data-frames. RDD משמש לאחסן נתונים מקובץ טקסט או רשימה, ו-Data-frames משמש כדי לאחסן טבלאות. ישנם מספר שפות שבהן ניתן להשתמש בממשק זה, כמו Scala, Java, Python. בדוגמאות כאן נשתמש ב-PySpark שהוא המפרש של Python עבור ספארק.

## RDD

RDD (resilient distributed dataset), מחזיק קבוצה של פריטים המפוזרים בין מספר מחשבים, באופן שמספק עמידות בנפילות או שגיאות. RDD של ספארק פועל כסט מידע פעיל אשר באופן מכוון יוצר ביזור מידע בתצורה מוגבלת.

### יצירה

יצירת מבנה הנתונים של RDD יכולה להיות מקובץ טקסט או ממבנה נתונים אחר. במקרה של יצירת RDD מקובץ, כל שורה בקובץ היא פריט ב-RDD. השיטות בכל RDD מאוד מזכירות שיטות על זרימה ב-Java Stream. גם כאן נשתמש בשיטות המחזירות RDD וכך נוכל לשרשר שיטות בשורה אחת של קוד. דוגמאות ליצירת RDD:

text\_file = sc.textFile("myDir/story.txt")

hits = sc.parallelize(range(0, samples))

### שיטות

* flatMap – מטרתו לפרק כל פריט ב-RDD המהווה רשימה, אל פריטים נפרדים ב-RDD כמספר האיברים ברשימה. הוא מקבל ביטוי למדא המופעל על כל פריט ב-RDD הנוכחי, שיהפוך את הפריט לרשימה. השימוש הנפוץ בשיטה זו הוא כדי ליצור RDD שבו לא כל שורה בקובץ טקסט היא פריט אלא כל מילה היא פריט.

words = text\_file.flatMap(lambda line: line.split(" "))

* map – משמש כדי לבצע שינוי על כל הפריטים ב-RDD. הוא ממפה כל פריט אל פריט אחר. שיטה זו מקבלת ביטוי למדא המקבל פרמטר אחד, מבצע עליו פעולה ומחזיר את הערך שהתקבל. שיטת map תפעיל את הביטוי למדא על כל פריט בנפרד ותחזיר RDD חדש עם כל הפריטים המעודכנים.
* reduceByKey – במקרה שכל פריט ב-RDD מהווה זוג של key ו-value, שיטה זו תאחד את כל הזוגות שיש להם key זהה. היא מקבלת ביטוי למדא המקבל שני ערכים ומחזיר ערך אחד. מטרתו של ביטוי למדא זה היא לאחד בין ה-value של כל הפריטים עם key זהה. שני הערכים שהביטוי למדא מקבל הם תוצאות חלקיות של איחוד זה.
* sortByKey – ממין את כל הפריטים לפי ה-key. הברירת מחדל היא בסדר עולה, אם מעוניינים בסדר יורד יש להכניס לשיטה פרמטר false.
* cache – במקרה של RDD מאוד גדול נשתמש בשיטה זו כדי לטעון אותו בזיכרון, כדי שהפעולות עליו יבוצעו יותר מהר. מפעילים שיטה זו פעם אחת בפקודה הפרדת ואין צורך לחזור עליה שוב.
* filter – משמש כדי לסנן פריטים מ-RDD. מקבל ביטוי למדא המקבל פרמטר אחד ומחזיר ערך בוליאני. שיטת filter תפעיל את הביטוי למדא על כל פריט ותשמור רק את הפריטים עבורם הוחזר true מהביטוי למדא.
* count – מחזיר את מספר הפריטים ב-RDD.
* collect – ממיר RDD לרשימה. זוהי בדרך כלל הפעולה האחרונה שנרצה להפעיל על RDD.

### דוגמאות

1. מחשב קירוב של PI באמצעות הצבת נקודות רנדומליות בריבוע בגודל 1 הנמצא ברבע הראשון של מעגל היחידה. ההסתברות שנקודה תיפול בתוך המעגל היא . לכן כדי לחשב את PI נבדוק כמה נקודות מתוך הסך הכולל אכן נמצא במעגל היחידה ונכפיל ב-4.

import random

samples = 10\*\*8 # number of random dots

def inside(p): # check if dot inside the circle

x, y = random.random(), random.random()

return x\*x + y\*y < 1

hits = sc.parallelize(range(0, samples)) \

.filter(inside).count()

print("Pi is approx. %f" % (4.0 \* hits / samples))

1. תוכנית המדפיסה בסדר יורד את כמות ההופעות של כל מילה בקובץ טקסט.

text\_file = sc.textFile("myDir/story.txt")

word\_counts = (text\_file.flatMap(lambda line: line.split(" ")) \

.map(lambda word: (word, 1)) \

.reduceByKey(lambda a,b: a+b) \

.map(lambda xy: (xy[1],xy[0])) \

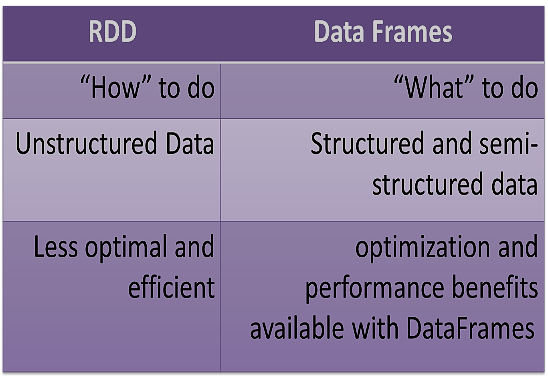
.sortByKey(False) \

.map(lambda xy: (xy[1],xy[0])) \

.collect())

for word,count in word\_counts[0:15]:

print("the word: \"%s\" appears %d time(s)" %(word,count))



## Data-frames

מבנה נתונים המשמש כדי לאחסן טבלאות. Data-frame בנוי על גבי RDD, אמנם בעוד RDD משמש כדי לאחסן נתונים לא מסודרים ומקוטלגים (כמו קבצי טקסט), Data-frame משמש כדי לאחסן נתונים עם מבנה מאוד מסודר ומקוטלג לרשומות ועמודות. למעשה, פעולות חישוב על Data-frame יותר מהירות מאשר ב-RDD מאחר ומשתמש בהמון אופטימיזציות כדי לייעל את החישובים שמבצע. הוא מאפשר בנוסף להריץ פקודות בשפת SQL ממש על הנתונים שמאחסן, וגם מוגדרות עליו שיטות המזכירות את הפקודות ב-SQL.

### יצירה

ניתן ליצור Data-frame משלושה סוגי נתונים:

1. ממסד נתונים רלציוני או מסד נתונים שהוא כמעט רלציוני, כמו Cassandra.
2. מ-RDD שכל פריט בו הוא אובייקט Row. באובייקט זה הפריט מחולק לשדות פנימיים עם שם.
3. מקבצי Json, XML, CSV וכו'.

דוגמה ליצירת Data-frame מרשימה שהופכת ל-RDD, שהופך לRDD שהפריטים בו מסוג Row.

from pyspark.sql import Row

from pyspark.sql import SQLContext  
sqlContext = SQLContext(sc)

student\_list = [(111, 'Chaya', 'Glass', 21),

(222, 'Tal','Negev',28),

(333, 'Gadi', 'Golan', 24),

(444, 'Moti', 'Cohen', 23)]

student\_rdd = sc.parallelize(student\_list)

students\_rows = student\_rdd.map(lambda x: Row(id=int(x[0]),age=int(x[3]), firstName=x[1], lastName=x[2]))

df\_students = sqlContext.createDataFrame(students\_rows)

דוגמה ליצירת Data-frame מקובץ Json. ההתאמה לקבצים מסוג אחר תהיה להחליף את השיטה הבאה אחרי read. הפרמטר multiline גורם לכך שיתעלמו מירידת שורה במקרה שהקובץ Json אינו שורה אחת ארוכה, אלא מסודר בצורה הנוחה לקריאה עם הזחות וירידות שורה.

df = sqlContext.read.json(“myDir\students.json”, multiLine=True)

### שיטות

show – מדפיס למסך בצורה קריאה את הטבלה שמאחסנת ה-Data-frame.

rdd – הופך data-frame ל-RDD. כל Data-frame בנוי על גבי RDD, לכן ההמרה הזו מאוד טבעית.

withColumn – מאפשר להוסיף עמודה לטבלה שבתוך ה-Data-frame. מקבלת שני פרמטרים: הפרמטר הראשון הוא שם העמודה הדשה, והפרמטר השני הוא מה יכנס בכל שורה בעמודה החדשה. בדרך כלל הפרמטר השני יהיה חישוב כלשהו בין שני עמודות קיימות, כך שבכל שורה יתקבל ערך אחר בהתאם לערכים הקיימים בו.

filter – מסננת רשומות מתוך ה-Data-frame. מקבלת ביטוי לוגי כלשהו על העמודות ב-Data-frame. השיטה תסנן את כל הרשומות עבורן הוחזר מהביטוי הלוגי false.

### פקודות SQL

כדי להריץ פקודות SQL יש ליצור טבלה זמנית שהיא עותק של ה-Data-frame באמצעות שיטת registerTempTable. השליפה תהיה על טבלה זמנית זו. לאחר מכן נפעיל את השיטה הסטטית sqlContext.sql המקבלת שאילתת SQL ומחזירה Data-frame המכיל את תוצאת השליפה. לדוגמה, שליפה מתוך Data-frame, המכיל פרטים על סטודנטים, את השם וה-ID של כל הסטודנטים שהשם משפחה שלהם נגמר באות n:

df\_students.registerTempTable("student\_table")

ending\_with\_n = sqlContext.sql("SELECT firstName, lastName, id

FROM student\_table

WHERE lastName LIKE '%n'")

ישנם עוד כל מיני שיטות על Data-frame שמבצעות פעולות כמו פקודות SQL, כמו: groupBy, sum, agg וכו'. סדרך כלל השימוש הפקודות SQL טבעיות ולא בשיטות אלו יהיה יותר מהיר.

# PySpark

## הקדמה

Apache Spark כתוב בשפת Scala. כדי לתמוך ב-Python עם Spark שוחררה PySpark. באמצעות PySpark ניתן לעבוד עם RDD גם ב-Python. PySpark מציעה PySpark Shell שמקשרת את ה-API של Python לליבת ה-spark.