# Java Streams

## הקדמה

יש שני שיטות להתעסק עם מידע:

1. במסד הנתונים - כל הפעולות על המידע נבצע במסד הנתונים המאחסן את המידע, באמצעות שאילתות המובנות למסד הנתונים.
2. בקוד עצמו - החישובים על המידע יעשו בתוך התוכנה. במידה והנתונים מאוחסנים במסד הנתונים נשתמש בו רק כדי לשלוף את המידע.

עד כה כאשר פעלנו לפי השיטה השנייה, אחסנו את המידע במבנה נתונים כלשהו והפעולות על המידע היו באמצעות לולאות. הבעיה בלולאות שהן לא מתעסקות עם המידע עצמו אלא עם האינדקסים של המבנה נתונים. בעיה נוספת היא שיותר מסובך וארוך ליישם באמצעותן תכנות מקבילי. השימוש ב-Java Stream פותר בעיות אלו בכך שמתעסק עם המידע עצמו ובכך מוריד את התקורה של הלולאה, וגם מאפשר חישוב מקבילי.

## הגדרה

זרמים הם אובייקטים ב-Java המייצגים רצף של בתים המכילים אובייקטים. ניתן ליצור אובייקט מסוג stream אמנם בדרך כלל השימוש בהם יהיה להפעיל שיטה stream על מערכים ואוספים שהופכות אותם לאובייקט זרימה. על אובייקטים מסוג זרימה מוגדרות כל מיני שיטות המאפשרות לנו לבצע חישובים שונים על המידע עצמו בצורה מהירה וקצרה, כמו שנראה בהמשך. באמצעות Java Stream ניתן לבצע על המידע גם חישוב מקבילי וזאת על ידי המרת המערך או האוסף לאובייקט הזרימה parallelStream.

לפני שנפרט על השיטות המוגדרות על אובייקטים מסוג זרימה, נפרט מהם ביטויי למדא, המהווים את הפרמטרים העיקריים לשיטות שנלמד.

## Lambda Expressions

אלו הן פונקציות שאין להם שם ויוצרים אותם תוך כדי הקוד, ולא במקום מיוחד כמו שאנו מכירים מפונקציות רגילות.

כדי להגדיר ביטוי למדא, יש לציין בסוגריים את הפרמטרים שהפונקציה מקבלת, לאחר מכן חץ, ולבסוף את הפעולות שהפונקציה תבצע בתוך סוגריים מסולסלות, כמו בפונקציה רגילה. כאשר הפונקציה מקבלת פרמטר אחד אין צורך להקיף את הפרמטר בסוגריים עגולות, וכן כאשר הפונקציה מכילה שורה אחת אין צורך להקיף אותה בסוגריים מסולסלות.

(argument-list) -> {body}

השימוש העיקרי בביטויי למדא הוא להכניס אותם כפרמטרים לפונקציות או שיטות. ביטוי למדא שאינו מקבל אף פרמטר נקרא callable.

יש שני דרכים להכניס ביטוי למדא לאובייקט:

* ניתן ליצור ממשק עם שיטה אחת, ולהגדיר שאובייקט זה שווה לביטוי למדא. פעולה זו תאפשר לנו להפעיל את השיטה על האובייקט גם בלי שהגדרנו מחלקה שמממשת את הממשק.
* באופן דומה, ניתן ליצור אובייקט מסוג Function<T,R> ולהגדיר שיהיה שווה לביטוי למדא. T מציין את סוג הפרמטר שמקבל הביטוי למדא, ו-R מציין את הסוג של הערך שמחזיר הביטוי למדא. כעת באמצעות שיטת apply על האובייקט נוכל להפעיל את הביטוי למדא.

## שיטות על אובייקט זרימה

ניתן לחלק את השיטות לשני סוגים: שיטות המחזירות אובייקט stream ושיטות שאינן מחזירות stream. שיטות מהסוג הראשון נקראות Intermediate Operations (פעולות ביניים), מפני שמאפשרות לשרשר עליהם שיטות נוספות. שיטות מהסוג השני נקראות Terminal Operations (פעולות סופיות) מפני שנמצאות תמיד בסוף השרשור. בכל המקומות שציינתי שמקבל ביטוי למדא ניתן לקבל גם מצביע לפונקציה.

### Intermediate Operations

* map – משמש כדי לבצע שינוי על כל האובייקטים בזרימה. הוא ממפה כל אובייקט בזרימה נוכחית אל אובייקט אחר. שיטה זו מקבלת ביטוי למדא המקבל פרמטר אחד, מבצע עליו פעולה ומחזיר את הערך שהתקבל. שיטת map תפעיל את הביטוי למדא על כל אובייקט בנפרד ותחזיר זרימה חדשה על כל האובייקטים המעודכנים.
* sorted – ממיין את כל האובייקטים בזרימה לפי ערך הברירת מחדל שמוגדר בהם. ניתן להכניס שיטה זו comparator שהמיון יעשה לפיו.
* filter – משמש כדי לסנן אובייקטים מהזרימה. מקבל ביטוי למדא המקבל פרמטר אחד ומחזיר ערך בוליאני. שיטת filter תפעיל את הביטוי למדא על כל אובייקט ותשמור רק את האובייקטים עבורם הוחזר true.

### Terminal Operations

* collect – ממיר את הזרימה לאוסף. בתוך שיטה זו נקרא לאחת השיטות של המחלקה Collectors כדי לציין לאיזה מבנה נתונים אנו רוצים להמיר את המידע.

List<String> list = streamObject.**collect**(Collectors.*toList*()));

* forEach – משמש כדי לבצע פעולה על כל האובייקטים בזרימה. מקבל ביטוי למדא עם פרמטר אחד. שיטת forEach תפעיל ביטוי למדא זה על כל אובייקט בזרימה.
* match – ישנם שלוש שיטות על זרמים מסוג match, המקבלות ביטוי למדא עם פרמטר אחד המחזיר ערך בוליאני. anyMatch מחזירה true רק אם היה לפחות אובייקט אחד שהביטוי למדא החזיר עברו true. allMatch מחזיר true אם עבור כל האובייקטים בזרימה הוחזר true מהביטוי למדא. noneMatch מחזיר true רק אם עבור כל האובייקטים בזרימה הוחזר false מהביטוי למדא.
* find – שיטות findAny, findFirst. צריך להשלים.
* count – מחזיר את מספר האובייקטים בזרימה.
* sum – מפעיל פעולת חיבור על כל האובייקטים בזרימה.

### דוגמאות

1. מיון של list והדפסת כל ערך ב-list.

myList.**stream**().**sorted**().**forEach**(System.*out*::println);

1. נתון מיפוי של כיתות לתלמידים הלומדים בהם. אנו רוצים להחזיר רשימה של כל הכיתות בהם יש תלמיד הצריך הנגשה.

List<String> roomsReqAccess = classrooms.entrySet()

.**stream**()

.**filter**(a->a.getValue().**stream**().**anyMatch**(x->x.reqAccesability))

.**collect**(Collectors.toList(c->c.getKey()));

## reduce

זוהי שיטה נוספת על זרימה, מסוג Terminal Operations, המפעילה איזושהי פונקציה על כל האובייקטים, אלא שפונקציה זו יכולה להתחשב גם בתוצאות של החישובים על האובייקטים הקודמים. דוגמה לשימוש מובנה ב-reduce הם שיטות count ו-sum שמבצעות פעולות סוכמות על האובייקטים. התחביר של שיטת reduce הוא:

reduce([identity], accumulator, [combiner])

שיטה זו יכולה לקבל עד שלושה פרמטרים, כאשר הפרמטר האמצעי הוא חובה, והשניים האחרים אופציונליים.

1. Identify – זהו האלמנט שממנו אנו מתחילים את החישוב. לדוגמא, בפעולת סכימת מספרים יהיה 0, בהכפלת מספרים יהיה 1, ובפעולת על מחרוזות בדרך כלל יהיה "" - מחרוזת ריקה.
2. Accumulator – זהו ביטוי למדא המקבל שני פרמטרים, x ו-y. הפרמטר הראשון x, זהו הפלט של כל תוצאות החישוב שנעשו עד שלב זה. Y מייצג את האובייקט הנוכחי. בתוך גוף הביטוי למדא נבצע חישוב על x ו-y. הערך שנחזיר מהביטוי למדא ייכנס לפרמטר x באובייקט הבא שנגיע אליו.
3. Combiner – במקרה של חישוב מקבילי נרצה להגדיר כיצד יש לאחד תוצאות של כל worker. לשם כך מוגדר ה-combiner, שהוא ביטוי למדא המקבל שני פרמטרים x ו-y, כאשר כל אחד מהם מייצג תוצאה חלקית של מספר אובייקטים. הביטוי למדא יחזיר תוצאה המאחדת שני תוצאות אלו.

לדוגמה, נתונה רשימה של מחרוזת, ואנו מעוניינים להחזיר מחרוזת המורכבת מהתו השביעי של כל מחרוזת ברשימה. נוכל לעשות זאת באמצעות reduce בדרך הבאה:

myList.stream.reduce(**""**, (x,t) -> x + t.charAt(6), (x,y) -> x + y);

## חישוב מקבילי

כדי לבצע חישוב מקבילי כל מה שצריך לעשות הוא להמיר את המערך או האוסף המכילים את המידע, לא לאובייקט stream אלא לאובייקט parallelStream. על אובייקט זה ניתן להפעיל את כל השיטות שיש באובייקט stream, אלא ש-parallelStream מבצע כל חישוב בצורה מקבילית ולכן הרבה יותר מהיר. נשתמש בחישוב מקבילי במקרים הבאים:

* סדר החישוב על האובייקטים בזרימה אינו חשוב.
* יש המון אובייקטים בזרימה ואנו רוצים שהחישובים יתבצעו מהר.
* החישוב על כל אובייקט בנפרד לוקח הרבה זמן.

# Spark

## הגדרה

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 מכריח מעבר ליניארי של מידע. 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 כל 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])) \

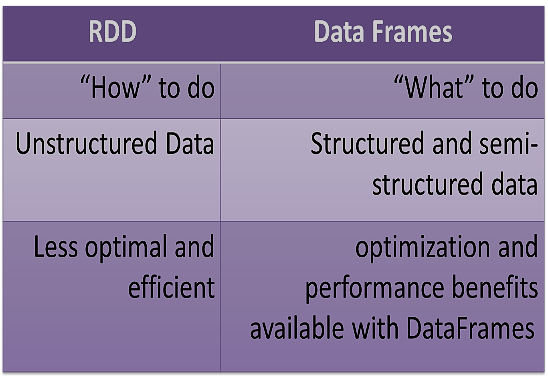
.sortByKey(False) \

.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.