PROJECT 1

Big data

03.12.20

Mathias W. Nilsen & Maria E. Pedersen

Innhold

MILESTONE 1

Datasets described

Tv shows on Netflix, Prime Video, Hulu and Disney+

The dataset ("TV shows on Netflix, Prime Video, Hulu and Disney+", 2020) contains details about tv-shows and which platform they are available on. The data is scraped from Reelgood.com and the dataset's size is approximately 254kB.

Column name(s)	Description
Title	Title of show
Year	Release year
Age	Recommended age group
IMDb, Rotten Tomatoes	Show ratings, from various websites
Netflix, Prime Video, Disney++, Hulu	Availability for on corresponding platforms
Type	Describes show type, movie or tv-show. In this dataset all are type tv-show

Kickstarter Projects

The dataset ("Kickstarter Projects", 2020) contains details about supported projects and ideas scraped from Kickstarter.com. The dataset is totalling 321,616 different projects.

Column name(s)	Description
ID	Unique ID
Name	Name of the project
Subcategory, Main Category	Project subcategory and main category
Country	Project launch country
USD Pledged	Currency in USD
Deadline	Project deadline
Goal	Project money goal required for launch
Launched	Launch date
Pledged	Current amount backing project
Backers	Number of backers
State	Describes success or failure

Gun deaths in the US

The dataset ("Gun Deaths in the US: 2012-2014", 2020) includes information about gun deaths in the US in the years 2012-2014. It is approximately 6.01 MB in size.

Column name(s)	Description
#	ID
year, month	The year and month of fatality
Intent	Perpetrator intent (suicide, accidental, NA, homicide, undetermined)
Police	Was a police shooting or not
Sex	Victim sex
Age	Victim age
Race	Victim race
Hispanic	Code representing Hispanic origin

Place	Where the shooting occurred
Education	Victim educational status

Drug overdose deaths

The dataset ("Drug overdose deaths", 2020) contains data of drug related deaths during 2020-2018 in the state Connecticut in the United States. The data is scrapped from data.gov and is approximately 1.79MB in size.

Column name(s)	**Description**
#	Row value
ID	Unique ID
Date	Date of the death or date reported
DateType	Where date reported is value 1 and date of death is value 0
Age	Age of the deceased
Sex	Gender of the deceased
Race	Race of deceased
ResidenceCity, ResidenceCounty, ResidenceState	Residence city, county, and state of the deceased
DeathCity, DeathCounty, Location	Which city, county, and location the deceased overdosed
DescriptionofInjury, Injuryplace, InjuryPlace, InjuryCity, InjuryCounty, InjuryState	Description of injury an where the injury was taken place
COD, otherSignifican	Cause of death and other significant factors
<Type of drug>	17 columns of the different substances present in the body of the deceased
MannerofDeath	Manner of death
DeathCityGEO, ResidenceCityGEO, InjuryCityGEO	Geolocation of death

Crimes in context

The dataset ("Crime in Context, 1975-2015", 2020) contains data collected from The Marshall Project, analysed over 40 year, and is approximately 250kB in size. The data consists of four major crimes the FBI classifies as violent (homicides, rape, assault, and robbery) in 68 police jurisdictions. The dataset then calculates the rates related to these crimes.

Column name(s)	**Description**
report_year	The year the crime is reported
agency_code	Seven alphanumerical characters used to classify accounts to the federal agency
agency_jurisdiction	Agency jurisdiction
Population	Population the jurisdiction
violent_crimes	Total number of violent crimes committed in the jurisdiction
homicides, rapes, assaults, robberies	Number of crimes committed in each of the four categories of violent crimes
Months_reported	Months reported
crimes_percapita	Total number of violent crimes percapita

assault_percapita, rapes_percapita, homicides_percapita robberies_percapita	Number of crimes per capita committed in each of the four categories

MILESTONE 2

Theory

Differences between dataset and Dataframe

Dataframe is a collection of data in named columns and is conceptually like a relational database. Dataframes are created from data in various formats contained either in files, databases or an external file system like HDFS. Making it highly useable for various tasks. It is also immutable, in addition to being scalable from kilobytes to petabytes, handling large amounts of data.

The API is both easy and intuitive to utilize making it user friendly. As it bears a resemblance to systems such as Python, Java, Scala and R. Where it is primarily used in Python and R. Main reason for this is due to those languages being untyped. Furthermore, it does not catch analysis error until runtime.

Scala also support Dataframe, which is an alias for dataset[row]. As a result, a Dataframe with a collection of rows is called a Dataset. A dataset is a distributed collection of objects and a combination of RDD and Dataframe. Unlike Dataframe, primary languages for Datasets is Scala and Java and it catches analysis error under compile time.

Furthermore, a dataset is also a collection of JVM objects that also has type safety. This is because both languages are strong typed. It also uses Tungsten's code generation to optimize queries, making it more efficient at optimizing queries.

Analysis of Big Data

One example of analysis of big amounts of data can be found in the new AMS electricity meters. These meters records consumption of customers. They give real time streaming of power consumption. The electricity meters has a physical output called the HAN port(Home Area Network). Which server the purpose of delivering information to a customer about their power consumption immediately or during the last hour. It further also contains information about the voltage level.

Spark is a suitable tool for AMS meters since Spark has a higher performance than for example Hadoop. Spark also has higher fault tolerance since the data does not disappear under error and Spark fails gracefully. In addition, Spark reduces overhead related to reading and writing from/to a disk as Spark performs calculations in the memory itself. Which in turn means that calculations can be performed acyclically.

Alibaba, which is a Chinese technology company specializing in e-commerce, is dependent on Apache Spark for their big data analytics. They use Spark as a tool to increase their business visibility and customer satisfaction. Another company that uses big data analytics and AI to help direct marketing, sales and business decisions is Starbucks. Spark is a useful tool in personalized marketing by using machine learning. Spark along with MLib, which is Spark's scalable machine learning library, is convenient to predict demands early. MLib comes with distributed implementations of clustering and classification algorithms.

Spark is dynamic since it allows itself to be used in countless other systems, which again uses different technologies. Spark can be incorporated with Hadoop which further allows a slow replacement of older infrastructure, if this exists. Spark also postpones the execution of code until is an absolute requirement.

Spark can be incorporated with many database drivers which allows Spark to remove all handling off this to the database itself. This will increase the efficiency since the databases themselves are created to handle their own data formats efficient.

However, Spark is not recommended for all kinds of handling of big data. In-memory storage is one example. A consequence of using Spark for this is delays and higher production costs as Spark requires lots of RAM to run in-memory. An example of this is Amazon Redshift. Which is a cloud-based data ware-house product. Amazon Redshift is a relational database management system. Apache Spark would not be a suitable tool for this task.

With all this in mind Spark is useful for real time data analysis, large-scale SQL, batch processing and machine learning. Other use cases for Spark that have not been mentioned are for example telecommunication systems where communication between parties are important, fraud detection in banking systems or data from IoT devices. Spark however is not suitable for use under multiuser systems, cloud warehousing or just pure analysis storage. Thus, Spark is less suitable for permanent mass storage of information.

Partitioning

Datasets and Dataframes consists of data elements varying in size. Which, as whole, could be larger than what a single node could contain. Leading to slower execution time. Partitioning is a way to solve this as it allows for logical splitting of the dataset into smaller partitions/datasets. Thereby allowing work to be performed in parallel, over several executor nodes in the cluster. Which increases the processing speed of data, as data is processed concurrently using partitions.

Fewer partitions allow work to be performed in a larger batch, which might lead to increased execution performance. Further, more partitions could create problems since individual size for each partition is small increasing overall scheduling time. Which in turn reduces overall efficiency of the cluster.

A rule of thumb is choosing a partition size equal to that of the total number of CPU cores pr node. Where each partition contains enough data so that it does not finish too quickly. An example of this is to reduce number of partitions on query two, under “Datasets Combined”, from 200 to 1. As this result is somewhat small, and does not require 200 partitions, as this causes unnecessary small partitions.

```
print(overdoseByRace.join(gundDeatchByRace, "Date").coalesce(1).rdd.partitions.length)
```

Choosing a partition size on a single node is somewhat more complicated as there are no dedicated CPU cores. Which means that some or all cores may already be occupied beforehand making it not available to be used. Therefore, a better solution for number of partitions is to not overload the remaining free CPUs as well as not underloading them. In addition to not overload the RAM either as it is a single node. Another downside is that usage of parallelization is reduced as this is limited to the number of **available** CPU cores.

Programming

From Optimized Logical Plan a plan is generated which describes how it physically shall perform on the cluster. Before a physical plan is performed, many plans are generated by the catalyst optimizer based on different strategies. Every physical plan is estimated based on execution time and resource consumption, and only one is chosen.

There are a lot of actions and functions that appears more than one time in the queries of this project, so how to read a physical plan is described here along with what the actions and functions mean. However, observations and interesting findings will be discussed under individual query.

A physical plan is read from the bottom up. Every new action occurs by a new plus sign as well as the functions the action uses and the values the functions takes.

The actions are described:

Action	**Described**
Filescan CSV	Shows that data has been read from a data format. In this case this is CSV.
Filter	Represents the conditions of the filtering
Project	Represents what columns are chosen. Every time *select, with column* or *drop* transformations are deployed on a Dataframe, this operator is called.
TakeOrderedAndProject	Retrieves the columns based on the conditions given.
LocalTableScan	Scans the columns
HashAggregate	Represents data aggregations. Usually shows up as a pair of two operators, which are parted by an *Exchange*. The first aggregation makes partial aggregations, which collects each partition separately on each executor. The final merging of the partial partitions is executed un the second aggregation.
Exchange	Moves data between different threads, processes, and machines. So, this operator represents a shuffle. Which will lead to physical data movement on the cluster. The operator is quite expensive, since it moves data over the network. The information in the operator also explains how the data should be repartitioned (HashPartition, SinglePartition, RangePartition, RoundRobinPartition)
Sort	Sorts based on the condition
BroadcastNestedLoopJoin	Is called when a physical plan finds a logical Join-operator in the query which either has a *canBuilLeft* jointype such as cross, inner, left anti, left outer, left semi or existence. Or a *canBuilRight* jointype such as cross, inner or right outer.
BroadcastExchange	A unary physical operator for collecting and broadcasting rows with a child relationship to working nodes. This action is called when a *BroadcastHashJoinExec* or a *BroadcastNestedLoopJoinExec* operator has been executed.

Queries

1. *Which streaming service releases most adult (18+) series and which service releases most series for children?*

Showcases an interesting fact about which streaming service appeals mostly to adults, and which that appeals to children. This assumes that age limit is specified for every tv-show and none has NULL. Though this is not true, thereby making the query not entirely accurate.

Alternative 1:

```
val df = spark.read.format("csv")
  .option("header", "true")
  .option("inferSchema", "true")
  .load("C:\\Users\\marpe\\Documents\\bigdata_data\\tv-shows.csv")

/*
  Anonymous function with params:  Column, String
  Counts if the column(streaming service) is of value 1, meaning it has the tv-show available,
  and the "Age" column has the value all/18+
  The alias for the count column is the column name and "Age" value
 */
val condition = (column: Column, value: String) =>
    count(when(column === 1 && col("Age") === value, 1)).as(column.toString() + " " + value)

//Passes the streaming services and age values into the anonymous function
val countDf =df.agg( condition(col("Prime Video"), "18+"), condition(col("Netflix"), "18+"),
condition(col("Hulu"), "18+"), condition(col("Disney+"), "18+"), condition(col("Prime Video"),
"all"), condition(col("Netflix"), "all"), condition(col("Hulu"), "all"), condition(col("Disney+"),
"all"))

val cols = countDf.columns
//splits the column in two
val split = cols.splitAt(cols.length / 2)
//Separates the streaming services for 18+ and "all"
val predicate = (c:String) => struct(col(c).as("v"), lit(c).as("k"))
val structCols18 = split._1.map(predicate)
val structColsAll = split._2.map(predicate)

//Puts the streaming services with the most adult and children series in a new column
val maxDf = countDf .withColumn("maxCol 18+", greatest(structCols18: _*).getItem("k"))
  .withColumn("maxCol All", greatest(structColsAll: _*).getItem("k"))

maxDf.write.mode(SaveMode.Overwrite).format("csv").save("D:\\data\\testing")
```

```
+---------------+-----------+--------+-----------+---------------+-----------+--------+-----------+-----------+---------------+
|Prime Video 18+|Netflix 18+|Hulu 18+|Disney+ 18+|Prime Video all|Netflix all|Hulu all|Disney+ all| maxCol 18+|     maxCol All|
+---------------+-----------+--------+-----------+---------------+-----------+--------+-----------+-----------+---------------+
|            182|        359|     239|          0|            192|        171|     159|         81|Netflix 18+|Prime Video all|
+---------------+-----------+--------+-----------+---------------+-----------+--------+-----------+-----------+---------------+
```

```
== Parsed Logical Plan ==
'Project [Prime Video 18+#77L, Netflix 18+#79L, Hulu 18+#81L, Disney+ 18+#83L, Prime Video all#85L, Netflix all#87L, Hulu all#89L, Disney+ all#91L, maxCol 18+#127, greatest(struct(NamePlaceholder, 'Prime
 Video all AS v#119, k, Prime Video all AS k#120), struct(NamePlaceholder, 'Netflix all AS v#121, k, Netflix all AS k#122), struct(NamePlaceholder, 'Hulu all AS v#123, k, Hulu all AS k#124), struct(NameP
laceholder, 'Disney+ all AS v#125, k, Disney+ all AS k#126))[k] AS maxCol All#137]
+- Project [Prime Video 18+#77L, Netflix 18+#79L, Hulu 18+#81L, Disney+ 18+#83L, Prime Video all#85L, Netflix all#87L, Hulu all#89L, Disney+ all#91L, greatest(struct(v, Prime Video 18+#77L, k, Prime Vide
o 18+), struct(v, Netflix 18+#79L, k, Netflix 18+), struct(v, Hulu 18+#81L, k, Hulu 18+), struct(v, Disney+ 18+#83L, k, Disney+ 18+)).k AS maxCol 18+#127]
   +- Aggregate [count(CASE WHEN ((Prime Video#62 = 1) AND (Age#57 = 18+)) THEN 1 END) AS Prime Video 18+#77L, count(CASE WHEN ((Netflix#60 = 1) AND (Age#57 = 18+)) THEN 1 END) AS Netflix 18+#79L, count(
CASE WHEN ((Hulu#61 = 1) AND (Age#57 = 18+)) THEN 1 END) AS Hulu 18+#81L, count(CASE WHEN ((Disney+#63 = 1) AND (Age#57 = 18+)) THEN 1 END) AS Disney+ 18+#83L, count(CASE WHEN ((Prime Video#62 = 1) AND (
Age#57 = all)) THEN 1 END) AS Prime Video all#85L, count(CASE WHEN ((Netflix#60 = 1) AND (Age#57 = all)) THEN 1 END) AS Netflix all#87L, count(CASE WHEN ((Hulu#61 = 1) AND (Age#57 = all)) THEN 1 END) AS
Hulu all#89L, count(CASE WHEN ((Disney+#63 = 1) AND (Age#57 = all)) THEN 1 END) AS Disney+ all#91L]
      +- Relation[_c0#54,Title#55,Year#56,Age#57,IMDb#58,Rotten Tomatoes#59,Netflix#60,Hulu#61,Prime Video#62,Disney+#63,type#64] csv

== Analyzed Logical Plan ==
Prime Video 18+: bigint, Netflix 18+: bigint, Hulu 18+: bigint, Disney+ 18+: bigint, Prime Video all: bigint, Netflix all: bigint, Hulu all: bigint, Disney+ all: bigint, maxCol 18+: string, maxCol All: s
tring
Project [Prime Video 18+#77L, Netflix 18+#79L, Hulu 18+#81L, Disney+ 18+#83L, Prime Video all#85L, Netflix all#87L, Hulu all#89L, Disney+ all#91L, maxCol 18+#127, greatest(struct(v, Prime Video all#85L,
k, Prime Video all), struct(v, Netflix all#87L, k, Netflix all), struct(v, Hulu all#89L, k, Hulu all), struct(v, Disney+ all#91L, k, Disney+ all)).k AS maxCol All#137]
+- Project [Prime Video 18+#77L, Netflix 18+#79L, Hulu 18+#81L, Disney+ 18+#83L, Prime Video all#85L, Netflix all#87L, Hulu all#89L, Disney+ all#91L, greatest(struct(v, Prime Video 18+#77L, k, Prime Vide
o 18+), struct(v, Netflix 18+#79L, k, Netflix 18+), struct(v, Hulu 18+#81L, k, Hulu 18+), struct(v, Disney+ 18+#83L, k, Disney+ 18+)).k AS maxCol 18+#127]
   +- Aggregate [count(CASE WHEN ((Prime Video#62 = 1) AND (Age#57 = 18+)) THEN 1 END) AS Prime Video 18+#77L, count(CASE WHEN ((Netflix#60 = 1) AND (Age#57 = 18+)) THEN 1 END) AS Netflix 18+#79L, count(
CASE WHEN ((Hulu#61 = 1) AND (Age#57 = 18+)) THEN 1 END) AS Hulu 18+#81L, count(CASE WHEN ((Disney+#63 = 1) AND (Age#57 = 18+)) THEN 1 END) AS Disney+ 18+#83L, count(CASE WHEN ((Prime Video#62 = 1) AND (
Age#57 = all)) THEN 1 END) AS Prime Video all#85L, count(CASE WHEN ((Netflix#60 = 1) AND (Age#57 = all)) THEN 1 END) AS Netflix all#87L, count(CASE WHEN ((Hulu#61 = 1) AND (Age#57 = all)) THEN 1 END) AS
Hulu all#89L, count(CASE WHEN ((Disney+#63 = 1) AND (Age#57 = all)) THEN 1 END) AS Disney+ all#91L]
      +- Relation[_c0#54,Title#55,Year#56,Age#57,IMDb#58,Rotten Tomatoes#59,Netflix#60,Hulu#61,Prime Video#62,Disney+#63,type#64] csv

== Optimized Logical Plan ==
Aggregate [count(CASE WHEN ((Prime Video#62 = 1) AND (Age#57 = 18+)) THEN 1 END) AS Prime Video 18+#77L, count(CASE WHEN ((Netflix#60 = 1) AND (Age#57 = 18+)) THEN 1 END) AS Netflix 18+#79L, count(CASE W
HEN ((Hulu#61 = 1) AND (Age#57 = 18+)) THEN 1 END) AS Hulu 18+#81L, count(CASE WHEN ((Disney+#63 = 1) AND (Age#57 = 18+)) THEN 1 END) AS Disney+ 18+#83L, count(CASE WHEN ((Prime Video#62 = 1) AND (Age#57
 = all)) THEN 1 END) AS Prime Video all#85L, count(CASE WHEN ((Netflix#60 = 1) AND (Age#57 = all)) THEN 1 END) AS Netflix all#87L, count(CASE WHEN ((Hulu#61 = 1) AND (Age#57 = all)) THEN 1 END) AS Hulu a
ll#89L, count(CASE WHEN ((Disney+#63 = 1) AND (Age#57 = all)) THEN 1 END) AS Disney+ all#91L, greatest(struct(v, count(CASE WHEN ((Prime Video#62 = 1) AND (Age#57 = 18+)) THEN 1 END), k, Prime Video 18+)
, struct(v, count(CASE WHEN ((Netflix#60 = 1) AND (Age#57 = 18+)) THEN 1 END), k, Netflix 18+), struct(v, count(CASE WHEN ((Hulu#61 = 1) AND (Age#57 = 18+)) THEN 1 END), k, Hulu 18+), struct(v, count(CAS
E WHEN ((Disney+#63 = 1) AND (Age#57 = 18+)) THEN 1 END), k, Disney+ 18+)).k AS maxCol 18+#127, greatest(struct(v, count(CASE WHEN ((Prime Video#62 = 1) AND (Age#57 = all)) THEN 1 END), k, Prime Video al
l), struct(v, count(CASE WHEN ((Netflix#60 = 1) AND (Age#57 = all)) THEN 1 END), k, Netflix all), struct(v, count(CASE WHEN ((Hulu#61 = 1) AND (Age#57 = all)) THEN 1 END), k, Hulu all), struct(v, count(C
ASE WHEN ((Disney+#63 = 1) AND (Age#57 = all)) THEN 1 END), k, Disney+ all)).k AS maxCol All#137]
+- Project [Age#57, Netflix#60, Hulu#61, Prime Video#62, Disney+#63]
   +- Relation[_c0#54,Title#55,Year#56,Age#57,IMDb#58,Rotten Tomatoes#59,Netflix#60,Hulu#61,Prime Video#62,Disney+#63,type#64] csv

== Physical Plan ==
*(2) HashAggregate(keys=[], functions=[count(CASE WHEN ((Prime Video#62 = 1) AND (Age#57 = 18+)) THEN 1 END), count(CASE WHEN ((Netflix#60 = 1) AND (Age#57 = 18+)) THEN 1 END), count(CASE WHEN ((Hulu#61
= 1) AND (Age#57 = 18+)) THEN 1 END), count(CASE WHEN ((Disney+#63 = 1) AND (Age#57 = 18+)) THEN 1 END), count(CASE WHEN ((Prime Video#62 = 1) AND (Age#57 = all)) THEN 1 END), count(CASE WHEN ((Netflix#6
0 = 1) AND (Age#57 = all)) THEN 1 END), count(CASE WHEN ((Hulu#61 = 1) AND (Age#57 = all)) THEN 1 END), count(CASE WHEN ((Disney+#63 = 1) AND (Age#57 = all)) THEN 1 END)], output=[Prime Video 18+#77L, Ne
tflix 18+#79L, Hulu 18+#81L, Disney+ 18+#83L, Prime Video all#85L, Netflix all#87L, Hulu all#89L, Disney+ all#91L, maxCol 18+#127, maxCol All#137])
+- Exchange SinglePartition, true, [id=#53]
   +- *(1) HashAggregate(keys=[], functions=[partial_count(CASE WHEN ((Prime Video#62 = 1) AND (Age#57 = 18+)) THEN 1 END), partial_count(CASE WHEN ((Netflix#60 = 1) AND (Age#57 = 18+)) THEN 1 END), part
ial_count(CASE WHEN ((Hulu#61 = 1) AND (Age#57 = 18+)) THEN 1 END), partial_count(CASE WHEN ((Disney+#63 = 1) AND (Age#57 = 18+)) THEN 1 END), partial_count(CASE WHEN ((Prime Video#62 = 1) AND (Age#57 =
all)) THEN 1 END), partial_count(CASE WHEN ((Netflix#60 = 1) AND (Age#57 = all)) THEN 1 END), partial_count(CASE WHEN ((Hulu#61 = 1) AND (Age#57 = all)) THEN 1 END), partial_count(CASE WHEN ((Disney+#63
= 1) AND (Age#57 = all)) THEN 1 END)], output=[count#156L, count#157L, count#158L, count#159L, count#160L, count#161L, count#162L, count#163L])
      +- FileScan csv [Age#57,Netflix#60,Hulu#61,Prime Video#62,Disney+#63] Batched: false, DataFilters: [], Format: CSV, Location: InMemoryFileIndex[file:/home/student/Documents/prosjekt1/tv-shows.csv],
 PartitionFilters: [], PushedFilters: [], ReadSchema: struct<Age:string,Netflix:int,Hulu:int,Prime Video:int,Disney+:int>
```

Physical plan, alternative 1: File is read from a CSV using Filescan, before doing a HashAggregate on columns, using conditions given in the query. HashAggregate is then split using Exchange SinglePartition.

Alternative 2:

```
val df= spark.read.format("csv").option("header", "true").option("inferSchema", "true").load("tv-shows.csv")

//count occurrences of 18+ series for each streaming service
val prime = df.where(col("Prime Video") === 1 && col("Age") === "18+").count()
val netflix = df.where(col("Netflix") === 1 && col("Age") === "18+").count()
val hulu = df.where(col("Hulu") === 1 && col("Age") === "18+").count()
val disney = df.where(col("Disney+") === 1 && col("Age") === "18+").count()

//count occurrences of All = also for children for each streaming service
val prime2 = df.where(col("Prime Video") === 1 && col("Age") === "all").count()
val netflix2 = df.where(col("Netflix") === 1 && col("Age") === "all").count()
val hulu2 = df.where(col("Hulu") === 1 && col("Age") === "all").count()
val disney2 = df.where(col("Disney+") === 1 && col("Age") === "all").count()

//creates a new dataframe with the previous values
val df_max = List( ("prime (18+)", prime),  ("netflix (18+)", netflix), ("hulu (18+)", hulu), ("disney (18+)", disney), ("prime (all)", prime2), ("netflix (all)", netflix2), ("hulu (all)", hulu2), ("disney+ (all)", disney2)).toDF("show","count")

//Filter series for children and adults, then sort each respectively and picks first value
val max_all= df_max.filter($"show".like("%all%")).sort(col("count").desc).limit(1)
val max_18 = df_max.filter($"show".like("%18%")).sort(col("count").desc).limit(1)

/*
  Union the two dataframes to combine information,
  then write to disk
 */
var newDf = spark.createDataFrame(sc.emptyRDD[Row], max_all.schema)
val first_row = newDf.unionAll(max_all.select($"*"))
val finalDf = first_row.unionAll(max_18.select("*"))
finalDf.write.mode(SaveMode.Overwrite).format("csv").save("/home/student/Documents/prosjekt1/data")
```

```
+-------------+-----+
|         show|count|
+-------------+-----+
|  prime (all)|  192|
|netflix (18+)|  359|
+-------------+-----+
```

```
scala> df.where(col("Disney+") === 1 && col("Age") === "all").explain(true)
== Parsed Logical Plan ==
'Filter (('Disney+ = 1) AND ('Age = all))
+- Relation[_c0#16,Title#17,Year#18,Age#19,IMDb#20,Rotten Tomatoes#21,Netflix#22,Hulu#23,Prime Video#24,Disney+#25,type#26] csv

== Analyzed Logical Plan ==
_c0: int, Title: string, Year: int, Age: string, IMDb: double, Rotten Tomatoes: string, Netflix: int, Hulu: int, Prime Video: int, Disney+: int, type: int
Filter ((Disney+#25 = 1) AND (Age#19 = all))
+- Relation[_c0#16,Title#17,Year#18,Age#19,IMDb#20,Rotten Tomatoes#21,Netflix#22,Hulu#23,Prime Video#24,Disney+#25,type#26] csv

== Optimized Logical Plan ==
Filter ((((isnotnull(Disney+#25) AND isnotnull(Age#19)) AND (Disney+#25 = 1)) AND (Age#19 = all))
+- Relation[_c0#16,Title#17,Year#18,Age#19,IMDb#20,Rotten Tomatoes#21,Netflix#22,Hulu#23,Prime Video#24,Disney+#25,type#26] csv

== Physical Plan ==
*(1) Project [_c0#16, Title#17, Year#18, Age#19, IMDb#20, Rotten Tomatoes#21, Netflix#22, Hulu#23, Prime Video#24, Disney+#25, type#26]
+- *(1) Filter (((isnotnull(Disney+#25) AND isnotnull(Age#19)) AND (Disney+#25 = 1)) AND (Age#19 = all))
   +- FileScan csv [_c0#16,Title#17,Year#18,Age#19,IMDb#20,Rotten Tomatoes#21,Netflix#22,Hulu#23,Prime Video#24,Disney+#25,type#26] Batched: false, DataFilters: [isnotnull(Disney+#25), isnotnull(Age#19), (Disney+#25 = 1), (Age#19 = all)], Format: CSV, Location: InMemoryFileIndex[file:/home/student/Documents/prosjekt1/tv-shows.csv], PartitionFilters: [], PushedFilters: [IsNotNull(Disney+), IsNotNull(Age), EqualTo(Disney+,1), EqualTo(Age,all)], ReadSchema: struct<_c0:int,Title:string,Year:int,Age:string,IMDb:double,Rotten Tomatoes:string,Netflix:int,Hu...
```

```
scala> max_all.explain(true)
== Parsed Logical Plan ==
GlobalLimit 1
+- LocalLimit 1
   +- Sort [count#190L DESC NULLS LAST], true
      +- Filter show#189 LIKE %all%
         +- Project [_1#184 AS show#189, _2#185L AS count#190L]
            +- LocalRelation [_1#184, _2#185L]

== Analyzed Logical Plan ==
show: string, count: bigint
GlobalLimit 1
+- LocalLimit 1
   +- Sort [count#190L DESC NULLS LAST], true
      +- Filter show#189 LIKE %all%
         +- Project [_1#184 AS show#189, _2#185L AS count#190L]
            +- LocalRelation [_1#184, _2#185L]

== Optimized Logical Plan ==
GlobalLimit 1
+- LocalLimit 1
   +- Sort [count#190L DESC NULLS LAST], true
      +- LocalRelation [show#189, count#190L]

== Physical Plan ==
TakeOrderedAndProject(limit=1, orderBy=[count#190L DESC NULLS LAST], output=[show#189,count#190L])
+- LocalTableScan [show#189, count#190L]
```

Physical plan, alternative 2: There are two physical plans, each explaining a different part of the query. The latter projects the result given an order. Whilst the first is the one responsible for sorting and selecting the first element. At first it will read contents of the file, then filter on condition before finally calling project, projecting the result. As is executed on each where-statement in query. After this it will do a TableScan, retrieving all tables under given conditions. A quick glance at physical plan for alternative 2, reveals it to be more demanding and complex compared to alternative 1.

2. *Which streaming service has the most and least tv-series available?*

This will give an indication of which streaming service that gives the most and least value for your money.

```
var df = spark.read.format(format).option("delimiter", ",").option("header", value =
true).option("inferSchema", value = true).load(path).drop("type")

//calculates the sum of the streaming service columns
val summedDf = df.agg( sum("Hulu").as("Hulu_sum"), sum("Disney+").as("Disney_sum"),
sum("Netflix").as("Netflix_sum"), sum("Prime Video").as("Prime_Vid_sum") )

//creates a new struct column that composes of input column v and k
val structs = summedDf.columns.tail.map(c => struct(col(c).as("v"), lit(c).as("k")))

//adds a max and min column to summedDF and writes to disk
summedDf.withColumn("maxCol", greatest(structs: _*).getItem("k")).withColumn("minCol",
least(structs:
_*).getItem("k")).write.mode(SaveMode.Overwrite).format("csv").save("D:\\data\\testing")
```

```
+---------+-----------+------------+-------------+-------------+-----------+
|Hulu_sum|Disney_sum|Netflix_sum|Prime_Vid_sum|       maxCol|     minCol|
+---------+-----------+------------+-------------+-------------+-----------+
|    1754|        180|        1931|         2144|Prime_Vid_sum|Disney_sum|
+---------+-----------+------------+-------------+-------------+-----------+
```

```
== Parsed Logical Plan ==
'Project [Hulu_sum#138L, Disney_sum#140L, Netflix_sum#142L, Prime_Vid_sum#144L, maxCol#165, least(struct(NamePlaceholder, 'Disney_sum AS v#159,
 k, Disney_sum AS k#160), struct(NamePlaceholder, 'Netflix_sum AS v#161, k, Netflix_sum AS k#162), struct(NamePlaceholder, 'Prime_Vid_sum AS v#
163, k, Prime_Vid_sum AS k#164))[k] AS minCol#171]
+- Project [Hulu_sum#138L, Disney_sum#140L, Netflix_sum#142L, Prime_Vid_sum#144L, greatest(struct(v, Disney_sum#140L, k, Disney_sum), struct(v,
 Netflix_sum#142L, k, Netflix_sum), struct(v, Prime_Vid_sum#144L, k, Prime_Vid_sum)).k AS maxCol#165]
   +- Aggregate [sum(cast(Hulu#71 as bigint)) AS Hulu_sum#138L, sum(cast(Disney+#73 as bigint)) AS Disney_sum#140L, sum(cast(Netflix#70 as bigi
nt)) AS Netflix_sum#142L, sum(cast(Prime Video#72 as bigint)) AS Prime_Vid_sum#144L]
      +- Project [_c0#64, Title#65, Year#66, Age#67, IMDb#68, Rotten Tomatoes#69, Netflix#70, Hulu#71, Prime Video#72, Disney+#73]
         +- Relation[_c0#64,Title#65,Year#66,Age#67,IMDb#68,Rotten Tomatoes#69,Netflix#70,Hulu#71,Prime Video#72,Disney+#73,type#74] csv

== Analyzed Logical Plan ==
Hulu_sum: bigint, Disney_sum: bigint, Netflix_sum: bigint, Prime_Vid_sum: bigint, maxCol: string, minCol: string
Project [Hulu_sum#138L, Disney_sum#140L, Netflix_sum#142L, Prime_Vid_sum#144L, maxCol#165, least(struct(v, Disney_sum#140L, k, Disney_sum), str
uct(v, Netflix_sum#142L, k, Netflix_sum), struct(v, Prime_Vid_sum#144L, k, Prime_Vid_sum)).k AS minCol#171]
+- Project [Hulu_sum#138L, Disney_sum#140L, Netflix_sum#142L, Prime_Vid_sum#144L, greatest(struct(v, Disney_sum#140L, k, Disney_sum), struct(v,
 Netflix_sum#142L, k, Netflix_sum), struct(v, Prime_Vid_sum#144L, k, Prime_Vid_sum)).k AS maxCol#165]
   +- Aggregate [sum(cast(Hulu#71 as bigint)) AS Hulu_sum#138L, sum(cast(Disney+#73 as bigint)) AS Disney_sum#140L, sum(cast(Netflix#70 as bigi
nt)) AS Netflix_sum#142L, sum(cast(Prime Video#72 as bigint)) AS Prime_Vid_sum#144L]
      +- Project [_c0#64, Title#65, Year#66, Age#67, IMDb#68, Rotten Tomatoes#69, Netflix#70, Hulu#71, Prime Video#72, Disney+#73]
         +- Relation[_c0#64,Title#65,Year#66,Age#67,IMDb#68,Rotten Tomatoes#69,Netflix#70,Hulu#71,Prime Video#72,Disney+#73,type#74] csv

== Optimized Logical Plan ==
Aggregate [sum(cast(Hulu#71 as bigint)) AS Hulu_sum#138L, sum(cast(Disney+#73 as bigint)) AS Disney_sum#140L, sum(cast(Netflix#70 as bigint)) A
S Netflix_sum#142L, sum(cast(Prime Video#72 as bigint)) AS Prime_Vid_sum#144L, greatest(struct(v, sum(cast(Disney+#73 as bigint)), k, Disney_su
m), struct(v, sum(cast(Netflix#70 as bigint)), k, Netflix_sum), struct(v, sum(cast(Prime Video#72 as bigint)), k, Prime_Vid_sum)).k AS maxCol#1
65, least(struct(v, sum(cast(Disney+#73 as bigint)), k, Disney_sum), struct(v, sum(cast(Netflix#70 as bigint)), k, Netflix_sum), struct(v, sum(
cast(Prime Video#72 as bigint)), k, Prime_Vid_sum)).k AS minCol#171]
+- Project [Netflix#70, Hulu#71, Prime Video#72, Disney+#73]
   +- Relation[_c0#64,Title#65,Year#66,Age#67,IMDb#68,Rotten Tomatoes#69,Netflix#70,Hulu#71,Prime Video#72,Disney+#73,type#74] csv

== Physical Plan ==
*(2) HashAggregate(keys=[], functions=[sum(cast(Hulu#71 as bigint)), sum(cast(Disney+#73 as bigint)), sum(cast(Netflix#70 as bigint)), sum(cast
(Prime Video#72 as bigint))], output=[Hulu_sum#138L, Disney_sum#140L, Netflix_sum#142L, Prime_Vid_sum#144L, maxCol#165, minCol#171])
+- Exchange SinglePartition, true, [id=#124]
   +- *(1) HashAggregate(keys=[], functions=[partial_sum(cast(Hulu#71 as bigint)), partial_sum(cast(Disney+#73 as bigint)), partial_sum(cast(Ne
tflix#70 as bigint)), partial_sum(cast(Prime Video#72 as bigint))], output=[sum#182L, sum#183L, sum#184L, sum#185L])
      +- FileScan csv [Netflix#70,Hulu#71,Prime Video#72,Disney+#73] Batched: false, DataFilters: [], Format: CSV, Location: InMemoryFileIndex[
file:/home/student/Desktop/myProject/tv_shows.csv], PartitionFilters: [], PushedFilters: [], ReadSchema: struct<Netflix:int,Hulu:int,Prime Vide
o:int,Disney+:int>
```

First the physical plan reads the file with FileScan. Then a HashAggregate is split with an Exchange SinglePartition.

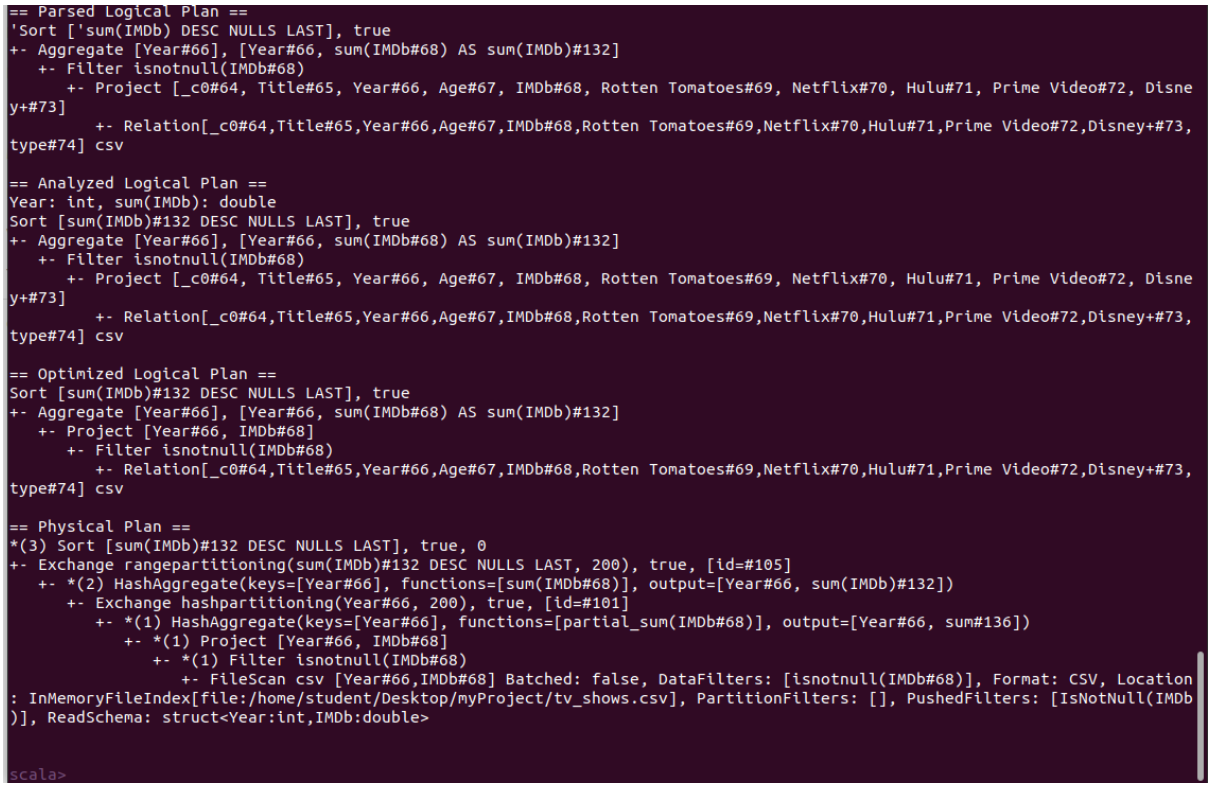
3. *What year released the shows with highest rating?*

This will give a certain indication of which tv-shows matches coming trends. It further gives an overall idea of which streaming service that contains highest rated shows.

Though using the total sum of every show per year, does not display correct result. As some years have more releases thereby leading to higher sums for the final rate.

```
var df = spark.read.format(format).option("delimiter", ",").option("header", value =
true).option("inferSchema", value = true).load(path).drop("type")

//groups IMDb by year, calculates the sum of ratings, sorts then picks the first value
val res = df.filter("IMDb is not
null").groupBy("Year").sum("IMDb").sort(column("sum(IMDb)").desc).limit(1)
.write.mode(SaveMode.Overwrite).format("csv").save("D:\\data\\testing")
```

```
|Year|sum(IMDb)|
+----+---------+
|2018|   3056.0|
+----+---------+
```



First the phyical plan reads the file with Filescan, then filters on the condition that the IMDb column is not NULL and thereafter retrieves the columns with Project. Then it performs a HashAggregate which is split with an Exchange Hashpartitioning. Finally, it does one more Exchange and sorts the columns.

4. What is the average number tv-shows released pr year?

By knowing the average of released tv-shows each year, one can get an indication of how many shows are released every year. Which will be reduced considerably as the number of years with few numbers of releases are larger than the years with many, in retrospect.

```
var df = spark.read.format(format).option("delimiter",
",").option("header", value = true).option("inferSchema", value =
true).load(path).drop("type")

/*groups by year, counts tv shows for each year, then calculates the
average*/
df.groupBy("Year").count().agg(avg("count").as("Avg_year")).write.mode
(SaveMode.Overwrite).format("csv").save("D:\\data\\testing")
```

```
+-----------------+
|         Avg_year|
+-----------------+
|69.27160493827161|
+-----------------+
```

```
scala> df.groupBy("Year").count().agg(avg("count").as("Avg_year")).explain(true)
== Parsed Logical Plan ==
'Aggregate [avg('count) AS Avg_year#111]
+- Aggregate [Year#66], [Year#66, count(1) AS count#107L]
   +- Project [_c0#64, Title#65, Year#66, Age#67, IMDb#68, Rotten Tomatoes#69, Netflix#70, Hulu#71, Prime Video#72, Disney+#
73]
      +- Relation[_c0#64,Title#65,Year#66,Age#67,IMDb#68,Rotten Tomatoes#69,Netflix#70,Hulu#71,Prime Video#72,Disney+#73,typ
e#74] csv

== Analyzed Logical Plan ==
Avg_year: double
Aggregate [avg(count#107L) AS Avg_year#111]
+- Aggregate [Year#66], [Year#66, count(1) AS count#107L]
   +- Project [_c0#64, Title#65, Year#66, Age#67, IMDb#68, Rotten Tomatoes#69, Netflix#70, Hulu#71, Prime Video#72, Disney+#
73]
      +- Relation[_c0#64,Title#65,Year#66,Age#67,IMDb#68,Rotten Tomatoes#69,Netflix#70,Hulu#71,Prime Video#72,Disney+#73,typ
e#74] csv

== Optimized Logical Plan ==
Aggregate [avg(count#107L) AS Avg_year#111]
+- Aggregate [Year#66], [count(1) AS count#107L]
   +- Project [Year#66]
      +- Relation[_c0#64,Title#65,Year#66,Age#67,IMDb#68,Rotten Tomatoes#69,Netflix#70,Hulu#71,Prime Video#72,Disney+#73,typ
e#74] csv

== Physical Plan ==
*(3) HashAggregate(keys=[], functions=[avg(count#107L)], output=[Avg_year#111])
+- Exchange SinglePartition, true, [id=#69]
   +- *(2) HashAggregate(keys=[], functions=[partial_avg(count#107L)], output=[sum#117, count#118L])
      +- *(2) HashAggregate(keys=[Year#66], functions=[count(1)], output=[count#107L])
         +- Exchange hashpartitioning(Year#66, 200), true, [id=#64]
            +- *(1) HashAggregate(keys=[Year#66], functions=[partial_count(1)], output=[Year#66, count#120L])
               +- FileScan csv [Year#66] Batched: false, DataFilters: [], Format: CSV, Location: InMemoryFileIndex[file:/hom
e/student/Desktop/myProject/tv_shows.csv], PartitionFilters: [], PushedFilters: [], ReadSchema: struct<Year:int>
```

First, a Filescan is done followed by a HashAggregate which is split by an Exchange HashPartitioning. Finally, one more HashAggregate is split with an Exchange SinglePartition.

5. *What is the top 10 highest rated shows on the whole dataset and which streaming services offers these shows?*

This can be relevant to know if one wants to watch a high-quality show and where someone can watch it. This will also give an indication on what streaming service has the highest rated shows.

```
val df = spark.read.format("csv")
  .option("header", value = true)
  .option("inferSchema", "true")
  .load("C:\\Users\\marpe\\Documents\\bigdata\\data\\tv-
shows.csv")

/* Selects the streaming services, sorts by the IMDDb column,
and picks the 10 first tv-shows
Then checks if a streaming service has the value 1, meaning
it has the tv-show available.
If it does then the streaming show is added to the topp10
column*/
val result = df
  .select("Title", "IMDb", "Netflix", "Hulu", "Prime Video",
"Disney+")
  .sort(desc("IMDb"))
  .limit(10)
  .withColumn("Topp10",
    concat_ws(", ",
        when(col("Hulu") === 1, lit("Hulu")),
      when(col("Prime Video") === 1, lit("Prime Video")),
      when(col("Disney+") === 1, lit("Disney+")),
      when(col("Netflix") === 1, lit("Netflix"))
    )
  //Unnecessary columns are dropped
  ).drop("Netflix", "Hulu", "Prime Video", "Disney+")

result.write.mode(SaveMode.Overwrite).format("csv").save("D:\
\data\\testing")
```

```
+--------------------+----+----------------+
|               Title|IMDb|          Topp10|
+--------------------+----+----------------+
|             Destiny| 9.6|            Hulu|
|        Breaking Bad| 9.5|         Netflix|
|       Hungry Henry| 9.5|            Hulu|
|        Malgudi Days| 9.5|      Prime Video|
|The Joy of Painting| 9.4|Hulu, Prime Video|
|    Band of Brothers| 9.4|      Prime Video|
|              Ramayan| 9.3|         Netflix|
|          Our Planet| 9.3|         Netflix|
|            The Wire| 9.3|      Prime Video|
|      Green Paradise| 9.3|      Prime Video|
+--------------------+----+----------------+
```

```
== Parsed Logical Plan ==
Project [Title#17, IMDb#20, Topp10#168]
+- Project [Title#17, IMDb#20, Netflix#22, Hulu#23, Prime Video#24, Disney+#25, concat_ws(, , CASE WHEN (Hulu#23 = 1) THEN Hulu END, CASE WHEN (Prime Video#24 = 1) THEN Prime Video END,
CASE WHEN (Disney+#25 = 1) THEN Disney+ END, CASE WHEN (Netflix#22 = 1) THEN Netflix END) AS Topp10#168]
   +- GlobalLimit 10
      +- LocalLimit 10
         +- Sort [IMDb#20 DESC NULLS LAST], true
            +- Project [Title#17, IMDb#20, Netflix#22, Hulu#23, Prime Video#24, Disney+#25]
               +- Project [_c0#16, Title#17, Year#18, Age#19, IMDb#20, Rotten Tomatoes#21, Netflix#22, Hulu#23, Prime Video#24, Disney+#25]
                  +- Relation[_c0#16,Title#17,Year#18,Age#19,IMDb#20,Rotten Tomatoes#21,Netflix#22,Hulu#23,Prime Video#24,Disney+#25,type#26] csv

== Analyzed Logical Plan ==
Title: string, IMDb: double, Topp10: string
Project [Title#17, IMDb#20, Topp10#168]
+- Project [Title#17, IMDb#20, Netflix#22, Hulu#23, Prime Video#24, Disney+#25, concat_ws(, , CASE WHEN (Hulu#23 = 1) THEN Hulu END, CASE WHEN (Prime Video#24 = 1) THEN Prime Video END,
CASE WHEN (Disney+#25 = 1) THEN Disney+ END, CASE WHEN (Netflix#22 = 1) THEN Netflix END) AS Topp10#168]
   +- GlobalLimit 10
      +- LocalLimit 10
         +- Sort [IMDb#20 DESC NULLS LAST], true
            +- Project [Title#17, IMDb#20, Netflix#22, Hulu#23, Prime Video#24, Disney+#25]
               +- Project [_c0#16, Title#17, Year#18, Age#19, IMDb#20, Rotten Tomatoes#21, Netflix#22, Hulu#23, Prime Video#24, Disney+#25]
                  +- Relation[_c0#16,Title#17,Year#18,Age#19,IMDb#20,Rotten Tomatoes#21,Netflix#22,Hulu#23,Prime Video#24,Disney+#25,type#26] csv

== Optimized Logical Plan ==
Project [Title#17, IMDb#20, concat_ws(, , CASE WHEN (Hulu#23 = 1) THEN Hulu END, CASE WHEN (Prime Video#24 = 1) THEN Prime Video END, CASE WHEN (Disney+#25 = 1) THEN Disney+ END, CASE WH
EN (Netflix#22 = 1) THEN Netflix END) AS Topp10#168]
+- GlobalLimit 10
   +- LocalLimit 10
      +- Sort [IMDb#20 DESC NULLS LAST], true
         +- Project [Title#17, IMDb#20, Netflix#22, Hulu#23, Prime Video#24, Disney+#25]
            +- Relation[_c0#16,Title#17,Year#18,Age#19,IMDb#20,Rotten Tomatoes#21,Netflix#22,Hulu#23,Prime Video#24,Disney+#25,type#26] csv

== Physical Plan ==
*(1) Project [Title#17, IMDb#20, concat_ws(, , CASE WHEN (Hulu#23 = 1) THEN Hulu END, CASE WHEN (Prime Video#24 = 1) THEN Prime Video END, CASE WHEN (Disney+#25 = 1) THEN Disney+ END, CA
SE WHEN (Netflix#22 = 1) THEN Netflix END) AS Topp10#168]
+- TakeOrderedAndProject(limit=10, orderBy=[IMDb#20 DESC NULLS LAST], output=[Title#17,IMDb#20,Netflix#22,Hulu#23,Prime Video#24,Disney+#25])
   +- FileScan csv [Title#17,IMDb#20,Netflix#22,Hulu#23,Prime Video#24,Disney+#25] Batched: false, DataFilters: [], Format: CSV, Location: InMemoryFileIndex[file:/home/student/Desktop/my
Project/tv_shows.csv], PartitionFilters: [], PushedFilters: [], ReadSchema: struct<Title:string,IMDb:double,Netflix:int,Hulu:int,Prime Video:int,Disney+:int>
```

First, a Filescan is done, thereafter the plan performs a TakeOrderedAndProject where it sorts the columns and selects the first ten rows. Finally, the action Project is executed, where all the when-conditions are called.

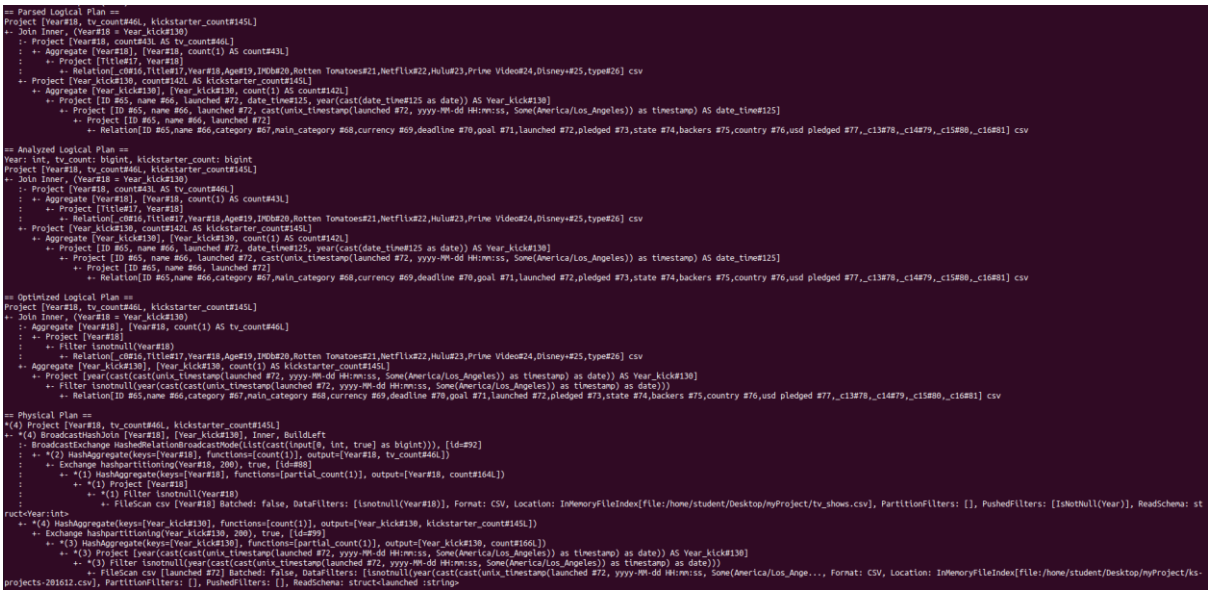
Datasets combined

1. *How many kickstarters was launched and tv-series was produced the same year?*

The dataset tv-shows.csv and kistarter.csv is combined to execute this query.

```
/*
Reads the tv-show file and drops unnecessary columns.
Then groups by year, counts and re-names the count column
 */
val tvDf = spark.read.format("csv")
  .option("header", value = true)
  .option("inferSchema", "true")
.load("C:\\Users\\marpe\\Documents\\bigdata\\data\\tv-shows.csv")
  .drop("_c0", "Age", "IMDb", "type", "Netflix",
    "Rotten Tomatoes", "Hulu", "Prime Video", "Disney+")
  .groupBy("Year")
  .count().withColumnRenamed("count", "tv_count")

//Reads the kickstarter file and drops unnecessary columns
val kickDf = spark.read.format("csv")
  .option("header", value = true)
  .option("inferSchema", "true")
  .load("C:\\Users\\marpe\\Documents\\kickstarter.csv")
  .drop("currency ", "category ", "main_category ", "backers ",
    "goal ", "deadline ", "pledged ", "state ", "country ", "usd pledged ",
    "_c14", "_c15", "_c16", "_c13")

/*
Add column "date_time" to kickDf and cast column "Launched" to timestamp.
Extrapolate year from "date_time" column
Group by year and count occurrences
 */
val dateKick = kickDf.withColumn("date_time",
    unix_timestamp(kickDf("launched "), "yyyy-MM-dd HH:mm:ss").cast("timestamp"))
   .withColumn("Year_kick", year(col("date_time")))
   .groupBy("Year_kick")
   .count().withColumnRenamed("count", "kickstarter_count")

//Join the dataframes on year and drop redundant column.
val kickTv = tvDf.join(dateKick, tvDf.col("Year")
  .equalTo(dateKick("Year_kick"))).drop("Year_kick")
kickTv.write.mode(SaveMode.Overwrite).format("csv").save("D:\\data\\testing")
```

Year	tv_count	kickstarter_count
2015	454	77056
2013	306	44551
2014	373	67436
1970	6	7
2012	304	40898
2009	148	1315
2016	573	54422
2010	203	10387
2011	251	26016



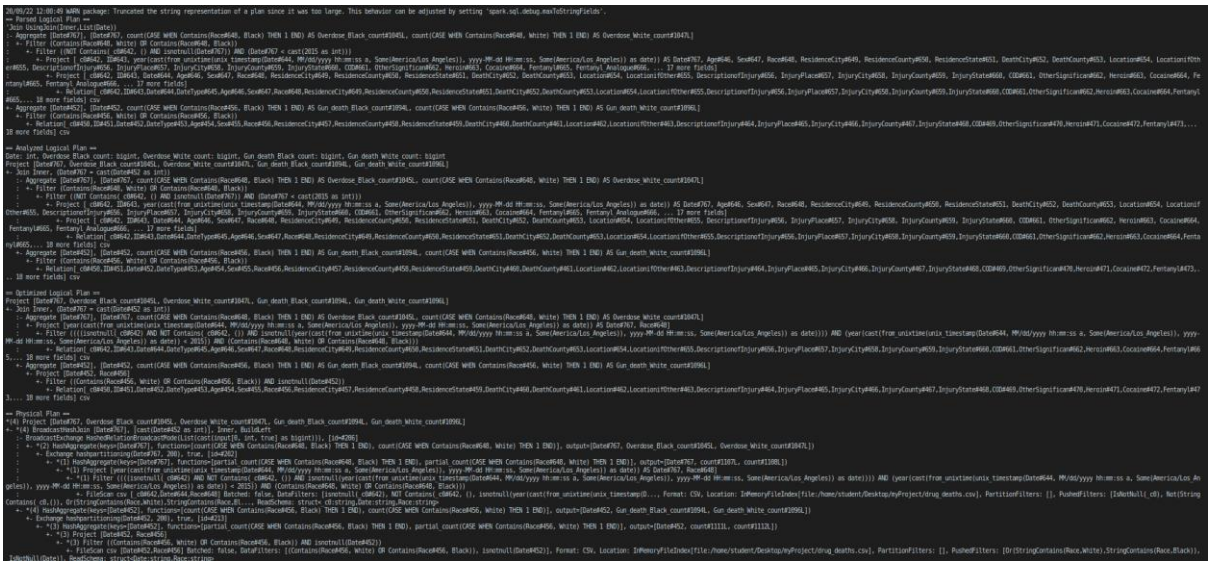
Firstly, the kickstarter file is read, then filtered on the conditions in the query and extracted. Then a HashAggregate is split with an Exchange HashPartitioning. When this is done the tv-shows file is read and the same actions as on the the former file is performed. When this is done a BroadcastExchange combined with a BroadcastHashJoin is performed. It is observed in the plan that the join executed is a left join. Finally, a Project action extracts the columns.

2. *How many black and white people committed suicide with firearms and how many overdosed from drugs each year?*

The CSV file "gun-deaths.csv" refers to all deaths with firearms throughout the US, but the file "drug_overdose.csv" only refers to overdoses in Connecticut. Thus, the query will not give any meaningful results. However, this can be combined with the total population of the US as well as Connecticut, for those years, to be able to say something more about the rate between the various deaths on a national basis.

```
def loadDf(file: String): DataFrame = (spark.read format "csv").option("header",
"true").option("inferSchema", "true").load(file)

val guns_file = "D:\\data\\guns.csv"
val drug_file = "D:\\data\\drug_deaths.csv"

val white = "White"
val black = "Black"
val race_col = col("Race")

//Reads the guns file
var gunDf = loadDf(guns_file).withColumnRenamed("year", "Date")
//Reads the drug file and adds column "Date" which is cast to a timestamp
var drugDf = loadDf(drug_file).drop("DateType").withColumn("Date",
year(from_unixtime(unix_timestamp(col("Date"), "MM/dd/yyyy hh:mm:ss a"))))

//Filters out empty data and dates after 2014, then sorts
drugDf = drugDf.filter((!col("_c0").contains("("))
              && ("Date").isNotNull
              && col("Date") < lit("2015"))
              .sort(col("Date").desc)

val predicate = race_col.contains(white) or race_col.contains(black)

/*
Function that counts on the condition that a column contains a value. Then names the count
column after the given alias
*/
val countByRace = (column: Column, value: String, alias: String) =>
  count(when(column.contains(value), 1)).as(alias)

/*
New dataframes that filters on whether the race is black or white and groups by date.
Then uses the predefined function to calculate the number of deaths for each race per year
 */
val overdoseByRace = drugDf.filter(predicate).groupBy(col("Date")).agg(countByRace(race_col,
black, "Overdose_Black_count"), (race_col, white, "Overdose_White_count"))
val gunDeathByRace = gunDf.filter(predicate).groupBy(col("Date")).agg(countByRace(race_col,
black, "Gun_death_Black_count"), countByRace(race_col, white, "Gun_death_White_count"))

//joins the two dataframes and writes to disk
overdoseByRace.join(gunDeathByRace,
"Date").write.mode(SaveMode.Overwrite).format("csv").save("D:\\data\\testing")
```

Date	Overdose_Black_count	Overdose_White_count	Gun_death_Black_count	Gun_death_White_count
2013	44	440	7712	22222
2014	30	522	7638	22222
2012	38	312	7946	21793



First, the file "drug_deaths.csv" is read with a FileScan and the columns are filtered, thereafter it is extracted with a Project action. Then a HashAggreate is executed and split with an Exchange HashPartitioning. After the aggregation is finished on the first file, the next file "drug_deaths.csv" is read and filtered the same way. Then a BroadcastExchange and a BroadCastHashJoin is called because Join is used in the query. The join is a left join. The rows are then extracted with a Project.

MILESTONE 3

RDD – programming

Load file from the local file system into an RDD

Challenges when running on a cluster

Problems that can appear while running on a cluster is that closures may get undefined behaviour as one often uses these to update outlying variables which is local to each executor. Though a solution for this is to utilize accumulators. Another issue is that every file resource must exist on the same path on all nodes on the cluster. But only if said file resource is defined local.

There is also the issue of running out of memory with the functions like collect and foreach. Though it is possible to mitigate this by either persisting data across more partitions which is then stored on disk. This would also further allow to reduce overall computationally intensive tasks.

Printing RDD

```
val filePath = "D:\\data\\crime_in_context_19752015.csv"
var file = sc.textFile(filePath)
file.collect().foreach(println)
```

```
import org.apache.log4j.{Level, Logger}
import org.apache.spark.sql.{Column, SaveMode, SparkSession}



spark: org.apache.spark.sql.SparkSession = org.apache.spark.sql.SparkSession@65679292



import spark.implicits._

filePath: String = D:\data\crime_in_context_19752015.csv
sc: org.apache.spark.SparkContext = org.apache.spark.SparkContext@8d8ba96
file: org.apache.spark.rdd.RDD[String] = D:\data\crime_in_context_19752015.csv MapPartitionsRDD[1] at textFile at <console>:19

report_year,agency_code,agency_jurisdiction,population,violent_crimes,homicides,rapes,assaults,robberies,months_reported,
 crimes_percapita,homicides_percapita,rapes_percapita,assaults_percapita,robberies_percapita
1975,NM00101,"Albuquerque, NM",286238,2383,30,181,1353,819,12,832.52,10.48,63.23,472.68,286.13
1975,TX22001,"Arlington, TX",112478,278,5,28,132,113,12,247.16,4.45,24.89,117.36,100.46
1975,GAAPD00,"Atlanta, GA",490584,8033,185,443,3518,3887,12,1637.44,37.71,90.3,717.1,792.32
1975,CO00101,"Aurora, CO",116656,611,7,44,389,171,12,523.76,6,37.72,333.46,146.58
1975,TX22701,"Austin, TX",300400,1215,33,190,463,529,12,404.46,10.99,63.25,154.13,176.1
1975,MD00301,"Baltimore County, MD",642154,1259,25,137,347,750,12,196.06,3.89,21.33,54.04,116.79
1975,MDBPD00,"Baltimore, MD",864100,16086,259,463,6309,9055,12,1861.59,29.97,53.58,730.12,1047.91
1975,MA01301,"Boston, MA",616120,11386,119,453,3036,7778,12,1848.02,19.31,73.52,492.76,1262.42
1975,NY01401,"Buffalo, NY",422276,3350,63,192,755,2340,12,793.32,14.92,45.47,178.79,554.14
1975,NC06001,"Charlotte, NC",262103,1937,68,71,976,822,12,739.02,25.94,27.09,372.37,313.62
1975,ILCPD00,"Chicago, IL",3150000,37160,818,1657,12514,22171,12,1179.68,25.97,52.6,397.27,703.84
1975,OHCIP00,"Cincinnati, OH",433367,3578,64,261,1508,1745,12,825.63,14.77,60.23,347.97,402.66
1975,OHCLP00,"Cleveland, OH",659931,10403,288,491,2524,7100,12,1576.38,43.64,74.4,382.46,1075.87
1975,OHCOP00,"Columbus, OH",572797,3980,62,416,1100,2402,12,694.84,10.82,72.63,192.04,419.35
1975,TXDPD00,"Dallas, TX",864665,7655,237,547,3485,3386,12,885.31,27.41,63.26,403.05,391.6
1975,CODPD00,"Denver, CO",508140,4960,74,480,1838,2568,12,976.11,14.56,94.46,361.71,505.37
1975,MI82349,"Detroit, MI",1432444,30387,633,1424,7013,21317,12,2121.34,44.19,99.41,489.58,1488.16
1975,TX07102,"El Paso, TX",369000,1618,21,122,635,840,12,438.48,5.69,33.06,172.09,227.64
1975,VA02901,"Fairfax County, VA",487905,805,21,104,247,433,12,164.99,4.3,21.32,50.62,88.75
1975,TX22012,"Fort Worth, TX",381275,1939,69,169,494,1207,12,508.56,18.1,44.32,129.57,316.57
1975,CA01005,"Fresno, CA",176500,1141,25,84,465,567,12,646.46,14.16,47.59,263.46,321.25
1975,HI00200,"Honolulu, HI",705262,1596,58,169,319,1050,12,226.3,8.22,23.96,45.23,148.88
1975,TXHPD00,"Houston, TX",1372342,8924,347,588,1567,6422,12,650.28,25.29,42.85,114.18,467.96
1975,INIPD00,"Indianapolis, IN",503411,4655,95,351,1117,3092,12,924.69,18.87,69.72,221.89,614.21
1975,FL01602,"Jacksonville, FL",542792,4579,91,316,2445,1727,12,843.6,16.77,58.22,450.45,318.17
1975,MOKPD00,"Kansas City, MO",489094,6072,114,302,2575,3081,12,1241.48,23.31,61.75,526.48,629.94
1975,NV00201,"Las Vegas, NV",249186,2594,37,169,1043,1345,12,1040.99,14.85,67.82,418.56,539.76
1975,CA01941,"Long Beach, CA",340758,3189,54,169,1007,1959,12,935.85,15.85,49.6,295.52,574.89
1975,CA01900,"Los Angeles County, CA",1022888,9792,138,471,6128,3055,12,957.29,13.49,46.05,599.09,298.66
1975,CA01942,"Los Angeles, CA",2729878,30405,554,1768,13493,14590,12,1113.79,20.29,64.76,494.27,534.46
1975,KY05680,"Louisville, KY",,,,,,,,,,,,
1975,TNMPD00,"Memphis, TN",646483,5429,120,521,1805,2983,12,839.77,18.56,80.59,279.2,461.42
1975,AZ00717,"Mesa, AZ",113176,267,2,22,145,98,12,235.92,1.77,19.44,128.12,86.59
1975,FL01300,"Miami-Dade County, FL",619795,6702,94,174,4088,2426,11,1081.33,15.17,28.07,646.67,391.42
```

Converting to RDD[Array[String]]

```
val headers = file.first()
val head = headers.split(",")
file = file.filter(line => line != headers && !line.contains("United States"))
val splitFile = file.map(line => {
  line.split(",(?=(?:[^\"]*\"[^\"]*\")*[^\"]*$)", head.length)
})
splitFile.collect().foreach ( x => println(x.mkString(", ")))
```

```
2015, OR02602, "Portland, OR", 615672, , , , , , , , , , ,
2015, MD01721, "Prince George's County, MD", 685191, 3256, 66, 248, 1583, 1359, , 475.2, 9.63, 36.19, 231.03, 198.3
2015, NC09201, "Raleigh, NC", 438363, 1676, 17, 108, 873, 678, , 382.33, 3.88, 24.64, 199.15, 154.67
2015, CA03404, "Sacramento, CA", 489717, 3611, 43, 105, 2289, 1174, , 737.36, 8.78, 21.44, 467.41, 239.73
2015, UT01803, "Salt Lake City, UT", 191992, 1642, 8, 291, 874, 469, , 855.24, 4.17, 151.57, 455.23, 244.28
2015, TXSPD00, "San Antonio, TX", 1463586, 8594, 94, 1049, 5465, 1986, , 587.19, 6.42, 71.67, 373.4, 135.69
2015, CA03711, "San Diego, CA", 1400467, 5582, 37, 566, 3601, 1378, , 398.58, 2.64, 40.42, 257.13, 98.4
2015, CA03801, "San Francisco, CA", 863782, 6710, 53, 344, 2703, 3610, , 776.82, 6.14, 39.82, 312.93, 417.93
2015, CA04313, "San Jose, CA", 1031458, 3400, 30, 375, 1855, 1140, , 329.63, 2.91, 36.36, 179.84, 110.52
2015, WASPD00, "Seattle, WA", 683700, 4093, 23, 144, 2394, 1532, , 598.65, 3.36, 21.06, 350.15, 224.07
2015, MOSPD00, "St. Louis, MO", 317095, 5762, 188, 263, 3521, 1790, , 1817.12, 59.29, 82.94, 1110.39, 564.5
2015, NY05101, "Suffolk County, NY", 1341453, 1675, 24, 79, 895, 677, , 124.86, 1.79, 5.89, 66.72, 50.47
2015, FL02902, "Tampa, FL", 364383, 2298, 34, 77, 1516, 671, , 630.66, 9.33, 21.13, 416.05, 184.15
2015, AZ01003, "Tucson, AZ", 529675, 3472, 31, 422, 1960, 1059, , 655.5, 5.85, 79.67, 370.04, 199.93
2015, OK07205, "Tulsa, OK", 401520, 3628, 55, 365, 2354, 854, , 903.57, 13.7, 90.9, 586.27, 212.69
2015, VA12800, "Virginia Beach, VA", 452797, 626, 19, 103, 234, 270, , 138.25, 4.2, 22.75, 51.68, 59.63
2015, DCMPD00, "Washington, DC", 672228, 8084, 162, 494, 4024, 3404, , 1202.57, 24.1, 73.49, 598.61, 506.38
2015, KS08703, "Wichita, KS", 389824, 3839, 27, 349, 2730, 733, , 984.8, 6.93, 89.53, 700.32, 188.03
headers: String = report_year,agency_code,agency_jurisdiction,population,violent_crimes,homicides,rapes,assaults,ro
ita,rapes_percapita,assaults_percapita,robberies_percapita
head: Array[String] = Array(report_year, agency_code, agency_jurisdiction, population, violent_crimes, homicides, r
ita, homicides_percapita, rapes_percapita, assaults_percapita, robberies_percapita)
file: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[6] at filter at <pastie>:30
splitFile: org.apache.spark.rdd.RDD[Array[String]] = MapPartitionsRDD[7] at map at <pastie>:31
```

RDD Queries

1. Which city had the highest crime rate per capita?

This is will give an indication about what city have most crime and this information can be used by authorities to find countermeasures to fight it. And possibly try and find the reasons for why.

Assumptions for this query is that all types of crimes in America are covered under “crimes per capita”, but that is not true. Therefore, the result may not be fully representative of the actual crime rate.

```
//Converts String to Double
val convertToDouble = (str: String) => {
  var i = 0.0
  if (!(str == null) && !str.isBlank) {
    i = str.toDouble
  }i}

//Creates a key-value RDD by mapping
val keyValueRDD= splitFile.map(arr => (arr(2), convertToDouble(arr(10))))
//combines values with the same key
val reducedRDD = keyValueRDD.reduceByKey ((a, b) => (a + b))

//Execute custom sorting from high to low and select max key
val maxKey = reducedRDD.max()(new Ordering[(String, Double)]() {
  override def compare(x: (String, Double), y: (String, Double)): Int =
    Ordering[Double].compare(x._2, y._2)
})
print(maxKey)
sc.parallelize(Seq(maxKey2)).saveAsTextFile("/data/max")
```

```
("Atlanta, GA",99462.95000000001)
```

RDD Lineage:

```
(2) ShuffledRDD[5] at reduceByKey at <console>:20 []
 +-(2) MapPartitionsRDD[4] at map at <console>:21 []
    |  MapPartitionsRDD[3] at map at <console>:21 []
    |  MapPartitionsRDD[2] at filter at <console>:21 []
    |  D:\data\crime_in_context_19752015.csv MapPartitionsRDD[1] at textFile at
<console>:21 []
    |  D:\data\crime_in_context_19752015.csv HadoopRDD[0] at textFile at <console>:21 []
```

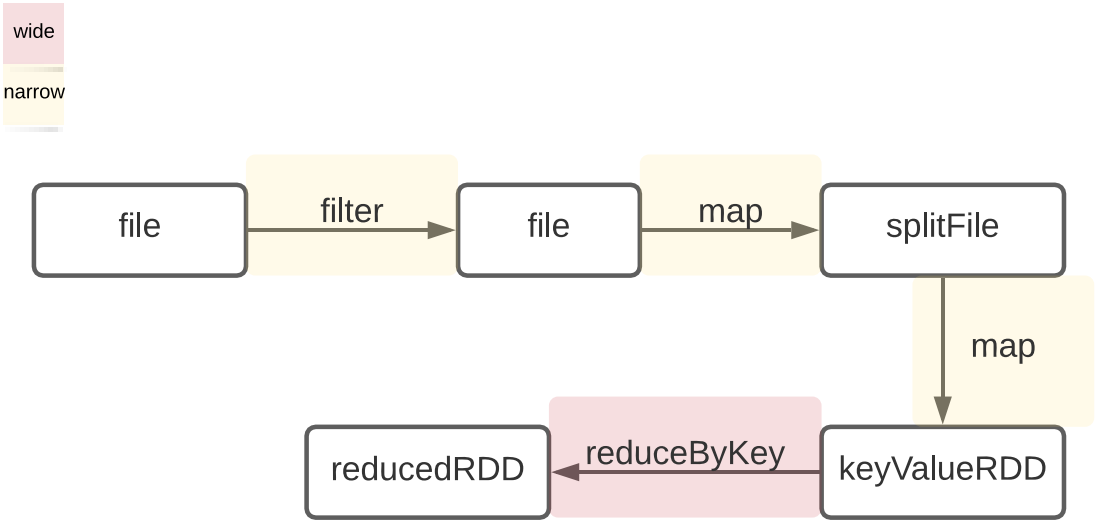


Figure 1 RDD Lineage - Query 1

2. What is the most common crime in Portland per year?

This is useful as one gets to see the most registered crimes pr year, which in turn allows one to focus on reducing those crimes. Something that would possibly help delegate resources more accurately. This query also assumes that all types of crimes are covered in the dataset, which is false. Therefore, the result may not be fully accurate.

```
//Get important indexes
val indexOfCity = head.indexWhere(str => str.equals("agency_jurisdiction"))
val crimeFirst = head.indexWhere(str => str.equals("homicides_percapita"))
val crimeLast = head.indexWhere(str => str.equals("robberies_percapita"))
val yearIndex = head.indexWhere(str => str.equals("report_year"))
/*Maps indexes to corresponding headers -> Used later to find corresponding
header name for query*/
var map = mutable.HashMap[Int, String](-1 -> "")
for (i <- head.indices) {
  map += (i -> head(i))
}
val filtered = splitFile.filter(arr => arr(indexOfCity).contains("Portland"))

//Converts array of Strings to array of Double
val convertToDouble = (arr: Array[String]) => {
  val convertedArr = ListBuffer[Double]()
  for (i <- crimeFirst to crimeLast) {
    val str = arr(i)
    if(str == null || str.isBlank){
      convertedArr += 0
    }else{
      convertedArr += str.toDouble
    }
  }
  convertedArr.toArray
}
//Converts String to Int
val convertToInt = (str: String) => {
  var i = 0
  if (!(str == null) && !str.isBlank) {
    i = str.toInt
  }
  i
}
//Determine max value and its corresponding category
  var maxValIndex = (arr: Array[Double]) => {
  var max = 0.0
  var maxIndex = -1
  for (i <- arr.indices) {
    val d = arr(i)
    if(max < d){
      max = d
      maxIndex = i
    }
  }
  maxIndex = if(maxIndex == -1) maxIndex else maxIndex + crimeFirst
  (max, map(maxIndex))
}
//Finds most common crime in Portland by category and max value
val summedRDD = filtered.map(
  arr => (
    convertToInt(arr(yearIndex)),
    arr(indexOfCity).replaceAll("[\",]", ""),
    maxValIndex(convertToDouble(arr))
  )
)
summedRDD.collect().foreach(arr => {
  println(arr._1, arr._2, arr._3._1, arr._3._2)
})
//Write result to disk
val finalRDD = summedRDD.map(row => {
  val str = row._1 + "," + row._2 +
    "," + row._3._2 + "," + row._3._1
  str
})
```

```
(1975,"Portland, OR",508.68,assaults_percapita)
(1976,"Portland, OR",534.47,assaults_percapita)
(1977,"Portland, OR",491.14,assaults_percapita)
(1978,"Portland, OR",507.04,assaults_percapita)
(1979,"Portland, OR",565.03,assaults_percapita)
(1980,"Portland, OR",630.59,assaults_percapita)
(1981,"Portland, OR",831.35,robberies_percapita)
(1982,"Portland, OR",877.51,assaults_percapita)
(1983,"Portland, OR",942.14,assaults_percapita)
(1984,"Portland, OR",1079.4,assaults_percapita)
(1985,"Portland, OR",1255.27,assaults_percapita)
(1986,"Portland, OR",1203.18,assaults_percapita)
(1987,"Portland, OR",1182.82,assaults_percapita)
(1988,"Portland, OR",1196.15,assaults_percapita)
(1989,"Portland, OR",1158.32,assaults_percapita)
(1990,"Portland, OR",1106.29,assaults_percapita)
(1991,"Portland, OR",1085.46,assaults_percapita)
(1992,"Portland, OR",1127.84,assaults_percapita)
(1993,"Portland, OR",1231.73,assaults_percapita)
(1994,"Portland, OR",1298.72,assaults_percapita)
(1995,"Portland, OR",1322.65,assaults_percapita)
(1996,"Portland, OR",1138.05,assaults_percapita)
(1997,"Portland, OR",1108.31,assaults_percapita)
(1998,"Portland, OR",957.63,assaults_percapita)
(1999,"Portland, OR",884.31,assaults_percapita)
(2000,"Portland, OR",730.27,assaults_percapita)
(2001,"Portland, OR",551.69,assaults_percapita)
(2002,"Portland, OR",522.21,assaults_percapita)
(2003,"Portland, OR",501.04,assaults_percapita)
(2004,"Portland, OR",442.23,assaults_percapita)
(2005,"Portland, OR",439.68,assaults_percapita)
(2006,"Portland, OR",417.21,assaults_percapita)
(2007,"Portland, OR",392.1,assaults_percapita)
(2008,"Portland, OR",368.34,assaults_percapita)
(2009,"Portland, OR",320.37,assaults_percapita)
(2010,"Portland, OR",307.31,assaults_percapita)
(2011,"Portland, OR",312.21,assaults_percapita)
(2012,"Portland, OR",316.37,assaults_percapita)
(2013,"Portland, OR",291.56,assaults_percapita)
(2014,"Portland, OR",288.47,assaults_percapita)
(2015,"Portland, OR",0.0,)
```

RDD Lineage:

```
(1) CoalescedRDD[7] at coalesce at <console>:21 []
 |  MapPartitionsRDD[6] at map at <console>:21 []
 |  MapPartitionsRDD[5] at map at <console>:26 []
 |  MapPartitionsRDD[4] at filter at <console>:21 []
 |  MapPartitionsRDD[3] at map at <console>:21 []
 |  MapPartitionsRDD[2] at filter at <console>:21 []
 |  D:\data\crime_in_context_19752015.csv MapPartitionsRDD[1] at textFile at <console>:21
[]
 |  D:\data\crime_in_context_19752015.csv HadoopRDD[0] at textFile at <console>:21 []
```

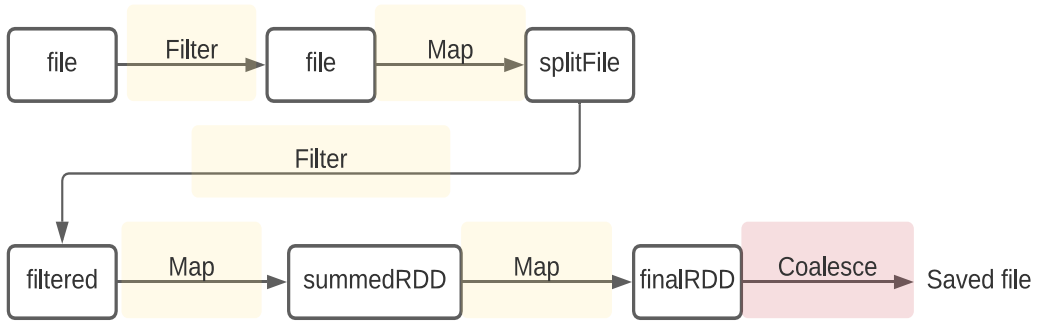


Figure 2 RDD Lineage - Query 2

HDFS

Manipulating filesystem

Firstly, a directory is created in HDFS and the dataset is uploaded to that directory. The directory's content is then showed. At last the file is transferred to the local filesystem again.

```
student@ubuntu:~$ hadoop fs -mkdir /data
student@ubuntu:~$ hadoop fs -copyFromLocal '/home/student/Documents/melipel3/crime_in_context_19752015.csv' /crime_in_context.csv
student@ubuntu:~$ hadoop fs -ls /
Found 2 items
-rw-r--r--   1 student supergroup     263935 2020-10-14 04:14 /crime_in_context.csv
drwxr-xr-x   - student supergroup          0 2020-10-14 04:14 /data
student@ubuntu:~$ hadoop fs -copyToLocal /crime_in_context.csv '/home/student/Documents/melipel3/crime_in_context.copy.csv'
student@ubuntu:~$
```

Modified program from RDD using dataset from HDFS

```
Altered filepath from: val filePath = "D:\\data\\crime_in_context_19752015.csv"
to: val filepath = "hdfs://127.0.0.1:19000/crime_in_context_19752015.csv"

val pathOut = "hdfs://127.0.0.1:19000/crime_in_protland.csv"
Altered from: finalRDD.coalesce(1).saveAsTextFile("D:\\data\\t.csv")
to: finalRDD.coalesce(1).saveAsTextFile(pathOut)
```

Theory HDFS

When the datafile is 20MB, which is marginally smaller than the block size(default 128MB), it can lead to not enough splitting. This again leads to underutilizing parallelization of the cluster. Because of the limited number of blocks available, the blocks will be wasted, and you will run out of them before utilizing all the storage capacity.

When the datafile is 20GB, the set will be split into about 160 files, which in turn would be evenly distributed across all cluster nodes. Combine this with a high replicating factor which would again lead to easier access to files across the cluster. This in turn would make it easier to make the computation local to each dataset. Thereby in turn leading to higher overall computational efficiency.

More replicas lead to higher usage of space and more writes across the cluster. As the cluster will try to spread each replica around, leading to higher fault tolerance and more robustness. If the replica factor is 3, the default policy is to store one on a node in the same rack, another on another rack and the last on another node on the same remote rack.

Reading a file requires the client to ask the namenode for the individual block locations. A pipeline consisting of all locations in order is given. Any unavailable blocks will cause the client to read the nearest copy of that block. So, if a file size is 20MB and the block size is 128MB then that would require reading only a single block whilst if the file was 20GB then that would require one to read about 160 blocks spread across different nodes.

Writing a file to HDFS requires the client to ask the namenode for a list of nodes to create replica blocks. After this the client will go through said list in order, writing block data and finally inform namenode that the content has been written. If a datanode should fail, then the block will be given a new ID and a new pipeline generated. Namenode will then be responsible for replicating the missing replica. Which means that if a file is 20MB then that will be written in one go. Though for a file that is 20GB, with around 160 blocks, would require to write and possibly replicate and larger number of blocks something that would increase the likelihood of possible failing datanodes.

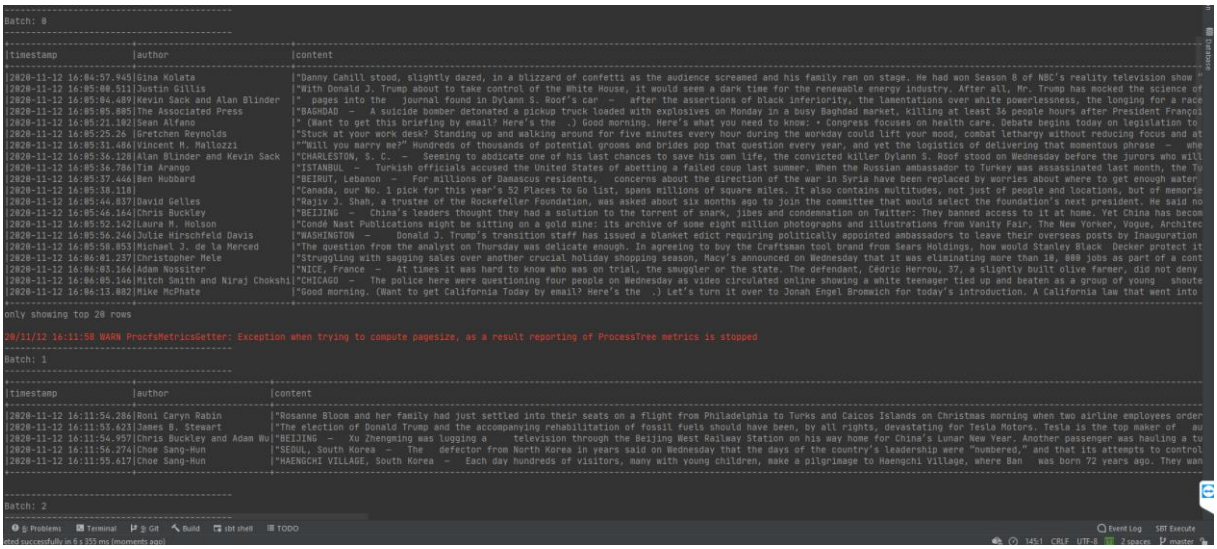
Streaming queries

The following defined function getSchema is used in every query under “Streaming”.

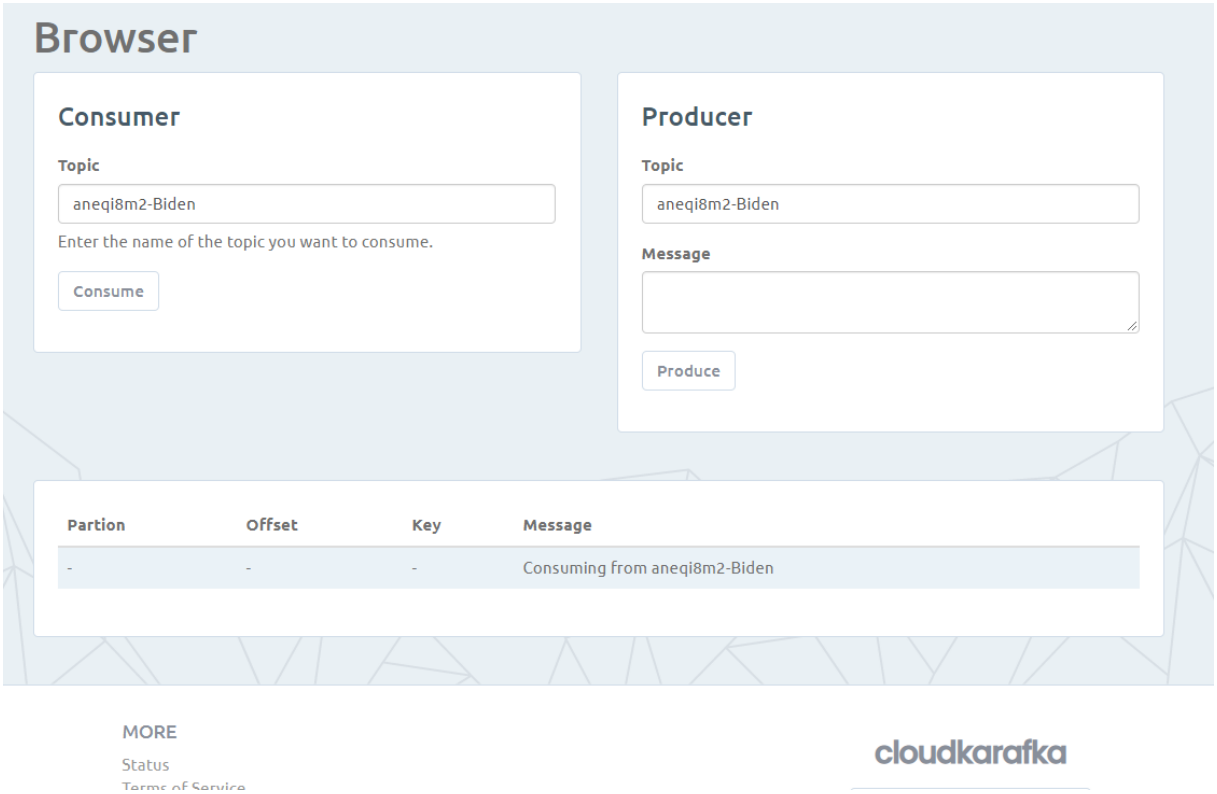
```
def getSchema:StructType={
  StructType(
    StructField("author", StringType, nullable = true) ::
    StructField("content", StringType, nullable = true) ::
    StructField("date", StringType, nullable = true) ::
    StructField("id", IntegerType, nullable = true) ::
    StructField("month", DoubleType, nullable = true) ::
    StructField("publication", StringType, nullable = true) ::
    StructField("retrieved", StringType, nullable = true) ::
    StructField("title", StringType, nullable = true) ::
    StructField("url", StringType, nullable = true) ::
    StructField("year", DoubleType, nullable = true) :: Nil
  )
}
```

1) Title/content contains Trump or Biden

```
      val streamIn = spark.readStream
      .format("kafka")
      .option("kafka.security.protocol", "SASL_SSL")
      .option("kafka.sasl.mechanism", "SCRAM-SHA-256")
      .option("kafka.sasl.jaas.config", """org.apache.kafka.common.security.scram.ScramLoginModule
required username="aneqi8m2" password="tiYqB_68T6l8OZU30p22LqTrXAsfEmCJ";""")
      .option("kafka.bootstrap.servers", "rocket-01.srvs.cloudkafka.com:9094,rocket-
02.srvs.cloudkafka.com:9094,rocket-03.srvs.cloudkafka.com:9094")
      .option("subscribe", "aneqi8m2-news")
      .option("startingOffsets", "earliest")
      .load()

    //Watermark the stream and cast timestamp to type timestamp
    val waterMarked = streamIn.withWatermark("timestamp", "1 second")
    val formattedDF = waterMarked
      .select($"timestamp", from_json($"value".cast("string"), getSchema).alias("data"))
      .select("timestamp", "data.*")

    //Filter for Trump and Biden
    val filteredDf = formattedDF.filter(
      lower($"title").contains("trump") ||
      lower($"title").contains("biden") ||
      lower($"content").contains("trump") ||
      lower($"content").contains("biden")
    )
    //Seperate into two distinct filtered streams
    val trumpFilteredDf = filteredDf.filter(
      lower($"title").contains("trump") ||
        lower($"content").contains("trump")
    )
    val bidenFilteredDf = filteredDf.filter(
      lower($"title").contains("biden") ||
        lower($"content").contains("biden")
    )

    bidenFilteredDf.select($"id".cast(StringType).as("key"), to_json(struct( $"title", $"id",
$"content")).as("value"))
      .writeStream.format("kafka")
      .option("kafka.security.protocol", "SASL_SSL")
      .option("kafka.sasl.mechanism", "SCRAM-SHA-256")
      .option("kafka.sasl.jaas.config", """org.apache.kafka.common.security.scram.ScramLoginModule
required username="aneqi8m2" password="tiYqB_68T6l8OZU30p22LqTrXAsfEmCJ";""")
      .option("kafka.bootstrap.servers", "rocket-01.srvs.cloudkafka.com:9094,rocket-
02.srvs.cloudkafka.com:9094,rocket-03.srvs.cloudkafka.com:9094")
      .option("topic", "aneqi8m2-Biden").option("checkpointLocation",
"D:\\projects_git\\Semester5\\big_data\\test2")
      .start()

    trumpFilteredDf.select($"id".cast(StringType).as("key"), to_json(struct( $"title", $"id",
$"content")).as("value"))
      .writeStream.format("kafka")
      .option("kafka.security.protocol", "SASL_SSL")
      .option("kafka.sasl.mechanism", "SCRAM-SHA-256")
      .option("kafka.sasl.jaas.config", """org.apache.kafka.common.security.scram.ScramLoginModule
required username="aneqi8m2" password="tiYqB_68T6l8OZU30p22LqTrXAsfEmCJ";""")
      .option("kafka.bootstrap.servers", "rocket-01.srvs.cloudkafka.com:9094,rocket-
02.srvs.cloudkafka.com:9094,rocket-03.srvs.cloudkafka.com:9094")
      .option("topic", "aneqi8m2-Trump").option("checkpointLocation",
"D:\\projects_git\\Semester5\\big_data\\test")
      .start()
      .awaitTermination()
```



Partion	Offset	Key	Message
-	-	-	Consuming from aneqi8m2-Trump
4	998	-	{"author":"Nick Corasaniti","content":"\"GREEN BAY, Wis. — Donald J. Trump belatedly endorsed the of Speaker Paul D. Ryan and Senators John McCain and Kelly Ayotte on Friday, moving to heal a deepening rift within the Republican Party touched off by Mr. Trump's feud with the parents of a slain American soldier. "I support and endorse our speaker of the House, Paul Ryan," Mr. Trump said at a rally here after announcing his backing of the senators. "He's a good man. We may disagree on a couple of things, but mostly we agree. " Mr. .."}
2	1237	-	{"author":"Patricia Cohen","content":"\"Donald J. Trump may rail against Wall Street and business elites at his campaign rallies, but that has not stopped him from turning to many of them for economic advice. On Friday, Mr. Trump announced his economic team, just days before he is expected to give a speech in Detroit on Monday about what he would do to improve American growth. The team — all men — includes several billionaire bankers and investment managers, and even a professional poker player. Many have been in business w.."}
3	1679	-	{"author":"Nicholas Confessore","content":"\"Carl Icahn was late, but he still had something to say. Wedging himself around a table crowded with Republican donors at a Hamptons beach house, he jumped in as Senator Mitch McConnell described how important it was to hold on to the Senate this fall. It was important to help Republicans hold the Senate, Mr. Icahn told the room. But they were kidding themselves if they thought they could leave Donald J. Trump twisting in the wind. "We have to get behind Trump," Mr. Icahn said, according to two .."}
-	2476	-	{"author":"David E. Sanger and Maggie Haberman","content":"\"Fifty of the nation's most senior Republican national security officials, many of them former top aides or cabinet members for President George W. Bush, have signed a letter declaring that Donald J. Trump "lacks the character, values and experience" to be president and "would put at risk our country's national security and . " Mr. Trump, the officials warn, "would be the most reckless president in American history. " The letter says Mr. Trump would weaken the United States' moral authority and.."}
1	1323	-	{"author":"Liam Stack","content":"\"Does Newt Gingrich, the former Republican House speaker and hopeful, think Donald J. Trump has the mental fitness to be president of the United States? His answer in an interview on Monday was not very convincing. "Yeah, and my answer would be, sure," Mr. Gingrich said, after a sigh and a pause, in an interview for the first episode of "The " a new politics podcast from The New York Times that was published on Tuesday. The former speaker declined to give a more emphatic endorsement of Mr. T.."}
-	2477	-	{"author":"Eli Rosenberg and Megan Jula","content":"\"A lone man's climb up the side of Trump Tower became a New York City spectacle on Wednesday afternoon after thousands watched his ascent in real time on television,

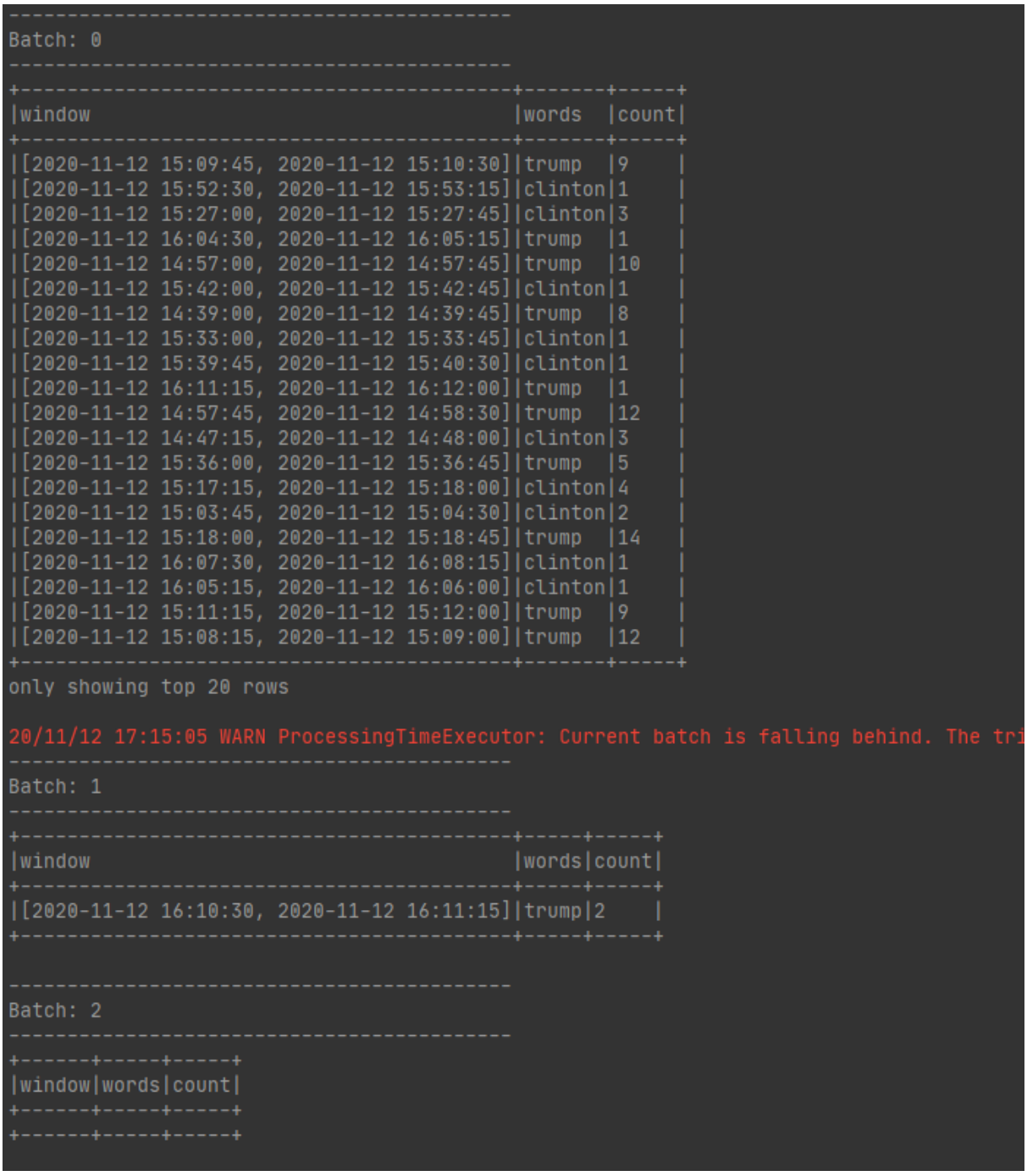


2) Counting number of times each name occurs in a window of 45 minutes with a watermark of 120 minutes

Importance of watermark

Utilization of watermark helps spark to know when to clean up its state and therefore, limiting the amount of intermediary data. It also defines a threshold for keeping late data. Which will be dropped if above said threshold. In this case as retrieved is created with a random time within same hour, allows data that arrives late to also be joined. Though in this case as both the retrieved stamp is within an hour long, and the processing steps are relatively small. Which means that the watermark could be smaller, eg: 80 minutes. Something that would lessen the amount of intermediary data. And thereby the strain on memory.

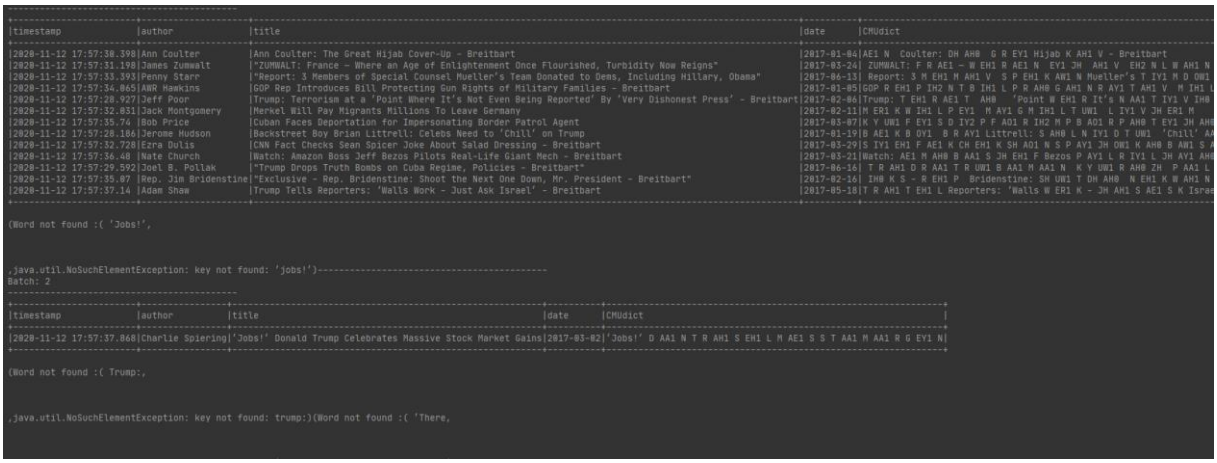
```
def extractAll = udf((str: String, exp: String) => {
    val pattern = Pattern.compile(exp)
    val matcher = pattern.matcher(str)
    var res = Seq[String]()
    while (matcher.find) {
      res = res :+ matcher.group(0)
}
    res.mkString(",")
})
val streamIn = spark.readStream
      .format("kafka")
      .option("kafka.security.protocol", "SASL_SSL")
      .option("kafka.sasl.mechanism", "SCRAM-SHA-256")
      .option("kafka.sasl.jaas.config",
"""org.apache.kafka.common.security.scram.ScramLoginModule required username="aneqi8m2"
password="tiYqB_68T6l8OZU30p22LqTrXAsfEmCJ";""")
      .option("kafka.bootstrap.servers", "rocket-01.srvs.cloudkafka.com:9094,rocket-
02.srvs.cloudkafka.com:9094,rocket-03.srvs.cloudkafka.com:9094")
      .option("subscribe", "aneqi8m2-news")
      .option("startingOffsets", "earliest")
      .load()

    val formattedDF = streamIn
      .select($"timestamp", from_json($"value".cast("string"), getSchema).alias("data"))
      .select("timestamp", "data.*")

//Watermark the stream and cast timestamp to type timestamp. With threshold of 2 hours.
    val waterMarked = formattedDF
      .withColumn("retrieved", unix_timestamp(formattedDF("retrieved"), "yyyy-MM-dd
HH:mm:ss.SSSSSS").cast("timestamp"))
      .withWatermark("retrieved", "120 minutes")

val filteredDf = waterMarked.filter(
      lower($"title").contains("trump") ||
        lower($"title").contains("clinton") ||
        lower($"title").contains("biden") ||
        lower($"content").contains("trump") ||
        lower($"content").contains("clinton") ||
        lower($"content").contains("biden")
    )
    val myWindow = window($"retrieved", "45 minutes", "45 minutes")
    val windowed = countedDf.groupBy(myWindow, $"words").count()

//extracts titles containing Trump, Biden or Clinton using capture groups
val countedDf = filteredDf
      .withColumn("words", extractAll(lower($"title"), lit("(clinton|biden|trump)")))
      .filter($"words".notEqual(""))
      .withColumn("words", explode(split($"words", ",")))
    //sets window and counts instances in that window
    val myWindow = window($"retrieved", "45 minutes", "45 minutes")
    val windowed = countedDf.groupBy(myWindow, $"words").count()

    windowed.writeStream
      .format("console")
      .option("truncate", value = false)
      .trigger(Trigger.ProcessingTime("10 seconds"))
      .outputMode(OutputMode.Update())
      .start()
      .awaitTermination()
```



```
-------------------------------------------
Batch: 0
-------------------------------------------
+------------------------------------------+-------+-----+
|window                                    |words  |count|
+------------------------------------------+-------+-----+
|[2020-11-12 15:09:45, 2020-11-12 15:10:30]|trump  |9    |
|[2020-11-12 15:52:30, 2020-11-12 15:53:15]|clinton|1    |
|[2020-11-12 15:27:00, 2020-11-12 15:27:45]|clinton|3    |
|[2020-11-12 16:04:30, 2020-11-12 16:05:15]|trump  |1    |
|[2020-11-12 14:57:00, 2020-11-12 14:57:45]|trump  |10   |
|[2020-11-12 15:42:00, 2020-11-12 15:42:45]|clinton|1    |
|[2020-11-12 14:39:00, 2020-11-12 14:39:45]|trump  |8    |
|[2020-11-12 15:33:00, 2020-11-12 15:33:45]|clinton|1    |
|[2020-11-12 15:39:45, 2020-11-12 15:40:30]|clinton|1    |
|[2020-11-12 16:11:15, 2020-11-12 16:12:00]|trump  |1    |
|[2020-11-12 14:57:45, 2020-11-12 14:58:30]|trump  |12   |
|[2020-11-12 14:47:15, 2020-11-12 14:48:00]|clinton|3    |
|[2020-11-12 15:36:00, 2020-11-12 15:36:45]|trump  |5    |
|[2020-11-12 15:17:15, 2020-11-12 15:18:00]|clinton|4    |
|[2020-11-12 15:03:45, 2020-11-12 15:04:30]|clinton|2    |
|[2020-11-12 15:18:00, 2020-11-12 15:18:45]|trump  |14   |
|[2020-11-12 16:07:30, 2020-11-12 16:08:15]|clinton|1    |
|[2020-11-12 16:05:15, 2020-11-12 16:06:00]|clinton|1    |
|[2020-11-12 15:11:15, 2020-11-12 15:12:00]|trump  |9    |
|[2020-11-12 15:08:15, 2020-11-12 15:09:00]|trump  |12   |
+------------------------------------------+-------+-----+
only showing top 20 rows

20/11/12 17:15:05 WARN ProcessingTimeExecutor: Current batch is falling behind. The tri
-------------------------------------------
Batch: 1
-------------------------------------------
+------------------------------------------+-----+-----+
|window                                    |words|count|
+------------------------------------------+-----+-----+
|[2020-11-12 16:10:30, 2020-11-12 16:11:15]|trump|2    |
+------------------------------------------+-----+-----+


-------------------------------------------
Batch: 2
-------------------------------------------
+------+-----+-----+
|window|words|count|
+------+-----+-----+
+------+-----+-----+
```

3)Translating title to CMUdict

```
def mapFonetToWords = udf((str: String) => {
    var fonets = Seq[String]()
    str.split(",").foreach((word:String) => {
      try {
        val v = mapped(word.toLowerCase)
        fonets = fonets :+ v
      } catch {
        case e: Exception =>
          print("Word not found :( " + word, "\n\n\n\n", e)
          fonets = fonets :+ word
      }
    })
    fonets.mkString(" ")
  })
val schema = StructType(
      StructField("_c0", StringType, nullable=false) ::
      StructField("_c1", StringType, nullable=false) ::
      StructField("_c2", StringType, nullable=false) ::
      StructField("_c3", StringType, nullable=false) :: Nil
    )
//Utilizes a custom hdfs that was setup locally on a home server
val path = "hdfs://10.0.0.95:9000/cmudict.dict"

//Using a map to set options is sort off redundant as it only has one option though useful to note
that possibility
val df = spark.read.schema(schema).options(Map("delimiter"->" ")).csv(path)

//Fill missing data with empty strings. Concat to fonet column and drop the rest.
val df1 = df.na.fill("")
val dictDf = df1.withColumn("fonet", concat_ws(" ", col("_c1"), col("_c2"),
col("_c3"))).drop("_c1", "_c2", "_c3")

//Map word to fonet -> Makes it faster to access the correct fonet
mapped = dictDf.map(row => (row.getAs[String](0), row.getAs[String](1))).collect.toMap

val streamIn = spark.readStream
  .format("kafka")
  .option("kafka.security.protocol", "SASL_SSL")
  .option("kafka.sasl.mechanism", "SCRAM-SHA-256")
  .option("kafka.sasl.jaas.config", """org.apache.kafka.common.security.scram.ScramLoginModule
required username="aneqi8m2" password="tiYqB_68T6l8OZU30p22LqTrXAsfEmCJ";""")
  .option("kafka.bootstrap.servers", "rocket-01.srvs.cloudkafka.com:9094,rocket-
02.srvs.cloudkafka.com:9094,rocket-03.srvs.cloudkafka.com:9094")
  .option("subscribe", "aneqi8m2-news")
  .option("startingOffsets", "earliest")
  .load()

val waterMarked = streamIn.withWatermark("timestamp", "1 second")
val formattedDF = waterMarked.select($"timestamp", from_json($"value".cast("string"),
getSchema).alias("data")).select("timestamp", "data.*")
//Map fonets to title and remove certain characters
val filteredDF = formattedDF
  .withColumn("CMUdict", mapFonetToWords(array_join(split($"title", "[,\\.\"\'\\?\\@\\s]"), ",")))
  .select($"timestamp", $"author", $"title", $"date", $"CMUdict")
val query = filteredDF.writeStream
  .format("console")
  .option("truncate", value = false).start()
query.awaitTermination()
```



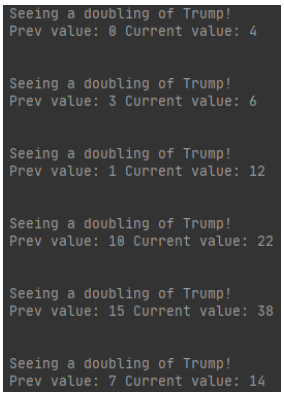
4) Number of news articles published that contains the word "Trump" that has multiplied according to the last interval, in a window of one hour

```
    val streamIn = spark.readStream
      .format("kafka")
      .option("kafka.security.protocol", "SASL_SSL")
      .option("kafka.sasl.mechanism", "SCRAM-SHA-256")
      .option("kafka.sasl.jaas.config", """org.apache.kafka.common.security.scram.ScramLoginModule
required username="aneqi8m2" password="tiYqB_68T6l8OZU30p22LqTrXAsfEmCJ";""")
      .option("kafka.bootstrap.servers", "rocket-01.srvs.cloudkafka.com:9094,rocket-
02.srvs.cloudkafka.com:9094,rocket-03.srvs.cloudkafka.com:9094")
      .option("subscribe", "aneqi8m2-news")
      .option("startingOffsets", "earliest")
      .load()
    //Watermark the stream and cast timestamp to type timestamp
    val waterMarked = streamIn
      .withColumn("timestamp", unix_timestamp(streamIn("timestamp"), "yyyy-MM-dd
HH:mm:ss").cast("timestamp"))
      .withWatermark("timestamp", "1 second")

    val formattedDF = waterMarked
      .select($"timestamp", from_json($"value".cast("string"), getSchema).alias("data"))
      .select("timestamp", "data.*")

    val trumpFilteredDf = formattedDF
      .filter(lower($"author").contains("trump") || lower($"content").contains("trump"))

    //Group by and count number of occurrences inside the determined time interval
    val myWindow = window($"timestamp", "60 seconds", "60 seconds")
    val trumpWindowed = trumpFilteredDf.groupBy(myWindow).count()

    var lastIntervalCount = Long.MaxValue
    var lastTime:Row = null
    val staticWindow = Window.orderBy($"window")
```

Streaming 1/2, Query 4

```
/*Utilizes a custom function for handling each batch and therein each row
    Utilized a memory sink as it does not really matter if data is lost in memory in this query.*/
val query = trumpWindowed.writeStream
      .format("memory")
      .option("truncate", value = false)
      .option("checkpointLocation", "D:\\projects_git\\Semester5\\big_data\\test")
      .foreachBatch { (df: DataFrame, _: Long) => {
        //Use lag to remember previous values by 1 offset
        val newDf = df.withColumn("prev_count", lag($"count", 1, 0).over(staticWindow))
        newDf.foreach((row:Row) => {
          val count = row.getLong(1)
          val windowStruct = row.getStruct(0)
          var prev = row.getLong(2)
          //Handle special case where query is first ran or entering a new batch.
          if (lastTime != null && prev == 0 && (windowStruct.equals(lastTime) ||
windowStruct.getTimestamp(1).equals(lastTime.getTimestamp(0)))){
            prev = lastIntervalCount
          }
/*Assumed it was greater than or equal, as this also accounts for greater rates of trump articles per
interval. Which would make sense from a business perspective*/
          if (count >= prev * 2) {
            print("Seeing a doubling of Trump!" + "\n")
            print( "Prev value: " + prev + " Current value: " + count + "\n\n\n")
          }
        })
        //Reverse sort, such that max value, last entry becomes first entry.
        val maxTime = newDf.sort($"window.end".desc).limit(1)
        val firstVal = maxTime.first()
        lastIntervalCount = firstVal.getLong(1)
        lastTime = firstVal.getStruct(0)

      }}.trigger(Trigger.ProcessingTime("61 seconds"))
      .outputMode(OutputMode.Update())
      .start()
    query.awaitTermination()

  private def getSchema:StructType={
    StructType(
      StructField("author", StringType, nullable = true) ::
      StructField("content", StringType, nullable = true) ::
      StructField("date", StringType, nullable = true) ::
      StructField("id", IntegerType, nullable = true) ::
      StructField("month", DoubleType, nullable = true) ::
      StructField("publication", StringType, nullable = true) ::
      StructField("retrieved", StringType, nullable = true) ::
      StructField("title", StringType, nullable = true) ::
      StructField("url", StringType, nullable = true) ::
      StructField("year", DoubleType, nullable = true) :: Nil
    )
  }
```

Streaming 2/2, Query 4



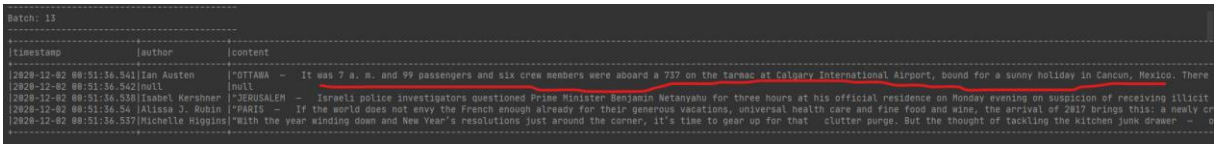
Bonus: Rewrote python script to scala script, utilizes spark.

```
def parseTextToJson = udf(f = (str: String) => {
  val regex = "^\\d+,(?<id>(\\\"(?:[^\\\"\\\\]|\\\\.)*\\\")|([^,]*))," +
    "(?<title>(\\\"(?:[^\\\"\\\\]|\\\\.)*\\\")|([^,]*))," +
    "(?<publication>(\\\"(?:[^\\\"\\\\]|\\\\.)*\\\")|([^,]+))," +
    "(?<author>(\\\"(?:[^\\\"\\\\]|\\\\.)*\\\")|([^,]*))," +
    "(?<date>(\\\"(?:[^\\\"\\\\]|\\\\.)*\\\")|([^,]*))," +
    "(?<year>(\\\"(?:[^\\\"\\\\]|\\\\.)*\\\")|([^,]*))," +
    "(?<month>(\\\"(?:[^\\\"\\\\]|\\\\.)*\\\")|([^,]*))," +
    "(?<url>(\\\"(?:[^\\\"\\\\]|\\\\.)*\\\")|([^,]*))," +
    "(?<content>(\".*\"))$"
  //Had to alter the last regex(content) to avoid recurring group captures. Which caused
fatal errors in spark

  val pattern = Pattern.compile(regex)
  val matcher = pattern.matcher(str)
  var res = ""

  //Find all groups
  if (matcher.find()) {
    try {
      val random = new scala.util.Random
      //Generate a random time difference as is done in the python script
      val timeRetrieved =
LocalDateTime.now().minusSeconds(random.nextInt(3600)).toString

      val id = matcher.group("id")
      val title = matcher.group("title")
      val publication = matcher.group("publication")
      val author = matcher.group("author")
      val date = matcher.group("date")
      val year = matcher.group("year")
      val month = matcher.group("month")
      val content = matcher.group("content")

      //Identify data with headers
      val data = Map(
        "id" -> id,
        "title" -> title,
        "publication" -> publication,
        "author" -> author,
        "date" -> date,
        "year" -> year,
        "month" -> month,
        "content" -> content,
        "retrieved" -> timeRetrieved
      )
      println("Writing data to cloud karafka.")

      //Convert content to json format
      res = Json(DefaultFormats).write(data)
      println(res)

    } catch {
      case _: Exception => print("Unparsable: ", str)
    }
  }
  res
})
```

```
Logger.getLogger("org").setLevel(Level.WARN)
    Logger.getLogger("akka").setLevel(Level.WARN)
    val path = "D:\\data\\all_the_news2" //Contains only smaller files from article1.csv,
had to limit the processing rate due to limitations of karafka and local bandwidth.

    val spark = SparkSession.builder()
      .master("local[*]")
      .appName("Produce messages")
      .getOrCreate()

    val streamIn = spark.readStream
      .option("maxFilesPerTrigger", 1)
      .text(path)

    val preparedStream = streamIn
      //First parse raw text from badly formatted csv file into json using a udf
      .withColumn("value", parseTextToJson(streamIn("value")).cast(StringType))

      //Then extract id by converting it from json and grabbing the id, which is then
cast to String type as is required by cloudkarafka, either strings or binary
      .withColumn("key", from_json(col("value"), schema).getItem("id").cast(StringType))

    //And finally setup config for streaming to kafka
    preparedStream
      .writeStream
      .format("kafka")
      .option("kafka.security.protocol", "SASL_SSL")
      .option("kafka.sasl.mechanism", "SCRAM-SHA-256")
      .option("kafka.sasl.jaas.config",
"""org.apache.kafka.common.security.scram.ScramLoginModule required username="aneqi8m2"
password="tiYqB_68T6l8OZU30p22LqTrXAsfEmCJ";""")
      .option("kafka.bootstrap.servers", "rocket-01.srvs.cloudkafka.com:9094,rocket-
02.srvs.cloudkafka.com:9094,rocket-03.srvs.cloudkafka.com:9094")
      .option("topic", "aneqi8m2-test")
      .option("checkpointLocation", "D:\\projects_git\\Semester5\\big_data\\test")
      .option("maxOffsetsPerTrigger", 20)
      .trigger(Trigger.ProcessingTime("3 seconds"))
      .start().awaitTermination()

val schema = new StructType()
    .add("id", StringType, nullable = true)
    .add("title", StringType, nullable = true)
    .add("publication", StringType, nullable = true)
    .add("author", StringType, nullable = true)
    .add("date", StringType, nullable = true)
    .add("year", DoubleType, nullable = true)
    .add("month", DoubleType, nullable = true)
    .add("url", StringType, nullable = true)
    .add("content", StringType, nullable = true)
    .add("retrieved", StringType, nullable = true)
```

Picture from streaming to Cloud karafka:

Picture from streaming from Cloud karafka:



Datasets used

Crime in Context, 1975-2015. Kaggle.com. (2020). Retrieved 12 November 2020, from https://www.kaggle.com/marshallproject/crime-rates.

Drug overdose deaths. Kaggle.com. (2020). Retrieved 2 November 2020, from https://www.kaggle.com/ruchi798/drug-overdose-deaths.

Gun Deaths in the US: 2012-2014. Kaggle.com. (2020). Retrieved 2 November 2020, from https://www.kaggle.com/hakabuk/gun-deaths-in-the-us.

Kickstarter Projects. Kaggle.com. (2020). Retrieved 2 November 2020, from https://www.kaggle.com/kemical/kickstarter-projects?select=ks-projects-201801.csv&fbclid=IwAR340k2Cpm2EcDg_3kN83s3hB58h3JhTsaOXdEHBFtA0hyKtvxKUpSZYNPA.

TV shows on Netflix, Prime Video, Hulu and Disney+. Kaggle.com. (2020). Retrieved 2 November 2020, from https://www.kaggle.com/ruchi798/tv-shows-on-netflix-prime-video-hulu-and-disney.

Attachments

- All code and sbt build configs for the project. Found under folder: "code"
- Datasets found under folder: "datasets"