ISIT312 Big Data Management

Spark Stream Processing

Dr Guoxin Su and Dr Janusz R. Getta

School of Computing and Information Technology -
University of Wollongong

Spark Structured Data and Stream Processing

Outline

Spark Stream Processing Modules

Stream processing is a key requirement in many big data applications

As soon as an application computes something of value, for example, a report or a machine learning model, an organization may want to compute this result continuously in a production environment

This capability is lacked in Hadoop MapReduce framework due to slowness of hard-disk IO

In-memory computation implemented in Spark make stream processing possible

Spark Streaming based on its low-level API Resilient Distributed Dataset is available since Spark 1.2

Spark Structured Streaming based on the Spark SQL engine is available since Spark 2.1

Luckily, our VM has installation of Spark 2.1.1

Spark Structured Data and Stream Processing

Outline

Stream Processing

Stream processing is the act of continuously incorporating the new data in the stream to compute a result

Sample sources of streams

- Bank transactions
- Clicks on a website
- Sensor readings from IoT devices
- Scientific observations and experiments
- Manufacturing processes, and the others

Stream processing vs. batch processing

- Batch processing runs to a fixed set of data, but stream processing handles an unbounded set of data
- Batch processing has low timeliness requirement, but stream processing requires to work at near realtime

Stream Processing

Use cases of stream processing

- Notifications and alerting
- Real-time reporting
- Incremental ETL
- Update data to serve in real time
- Real-time decision making
- Online machine learning

Stream Processing

To see the challenges of stream processing, we consider the following example

Suppose we received the following data from a sensor

Sample data

```
{value: 1,  time: "2017-04-07T00:00:00"}
{value: 2,  time: "2017-04-07T01:00:00"}
{value: 5,  time: "2017-04-07T02:00:00"}
{value: 10, time: "2017-04-07T01:30:00"}
{value: 7,  time: "2017-04-07T03:00:00"}
```

What actions should be performed when receiving single values, say, `5` ?

How to react to a pattern, say, `2` -> `10` -> `5`

What if data arrives out-of-order, for example, `10` before `5`

Other issues: What if a machine in the system fails, losing some state? What if the load is imbalanced? How can an application signal downstream consumers when analysis for some event is done, and so on

Stream Processing

Main challenges of stream processing are the following

- Processing out-of-order data based on application timestamps (also called event time)
- Maintaining large amounts of states
- Supporting high-data throughput
- Processing each event exactly once despite machine failures
- Handling load imbalance and strugglers
- Responding to events at low latency
- Joining with external data in other storage systems
- Determining how to update output sinks as new events arrive
- Writing data transactionally to output systems
- Updating application business logic at runtime

Spark Structured Data and Stream Processing

Outline

Quick Start

Structured Streaming Processing suppose to provide fast, scalable, fault-tolerant, end-to-end exactly-once stream processing without the user having to reason about streaming

A streaming version of the word-count example

Reading a stream

```
val lines = spark.readStream
                 .format("socket")             // socket source
                 .option("host", "localhost") // listen to the localhost
                 .option("port", 9999)         // and port 9999
                 .load()
```

Importing methods

```
import spark.implicits._
```

sql

```
val words = lines.as[String].flatMap(_.split(" "))
```

Grouping

```
val wordCounts = words.groupBy("value").count()
```

Writing stream

```
val query = wordCounts.writeStream
                      .outputMode("complete") // accumulate the counting result
                      .format("console")      // use the console as the sink
                      .start()
```

Quick Start

The input is simulated by Netcat (a small utility found in most Unix-like systems) as a data server

Starting Netcat

```
nc -lk 9999
```

In a different Terminal, we start Spark-shell and input the Scala code from the previous slides

If we input in the first Terminal session

Starting Netcat

```
nc -lk 9999
apache spark
apache hadoop
...
```

Quick Start

Then we should see the right hand-side output in Spark-shell

```
-------------------------------------------
Batch: 0
-------------------------------------------
+------+-----+
| value|count|
+------+-----+
|apache|    1|
| spark|    1|
+------+-----+

-------------------------------------------
Batch: 1
-------------------------------------------
+------+-----+
| value|count|
+------+-----+
|apache|    2|
| spark|    1|
|hadoop|    1|
+------+-----+
...
```

Spark Structured Data and Stream Processing

Outline

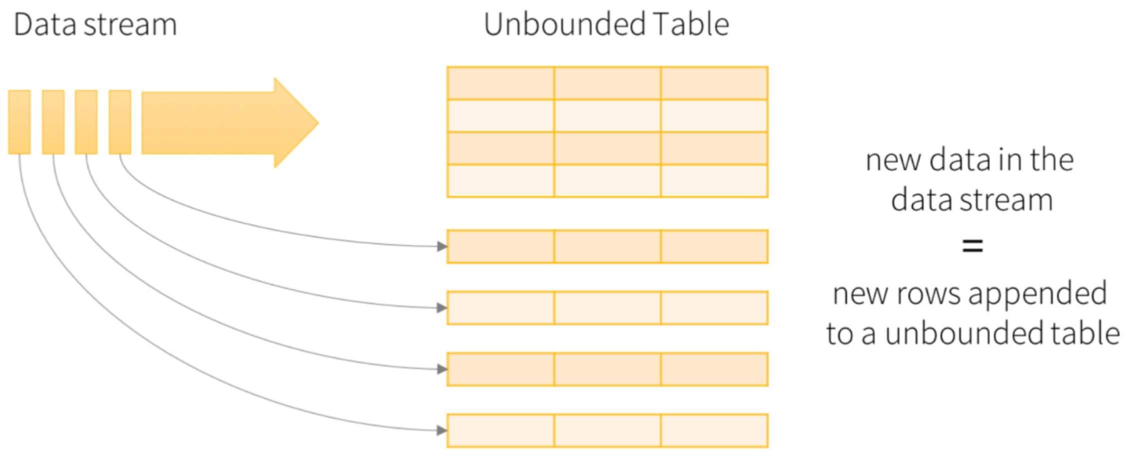
Programming Model

The key idea in Structured Streaming is to treat a live data stream as a table that is being continuously appended

This leads to a new stream processing model that is very similar to a batch processing model

Users can express the streaming computation as standard batch-like query as on a static table, and Spark runs it as an incremental query on the unbounded Input Table

A new data item arriving on the stream is like a new row being appended to Input Table



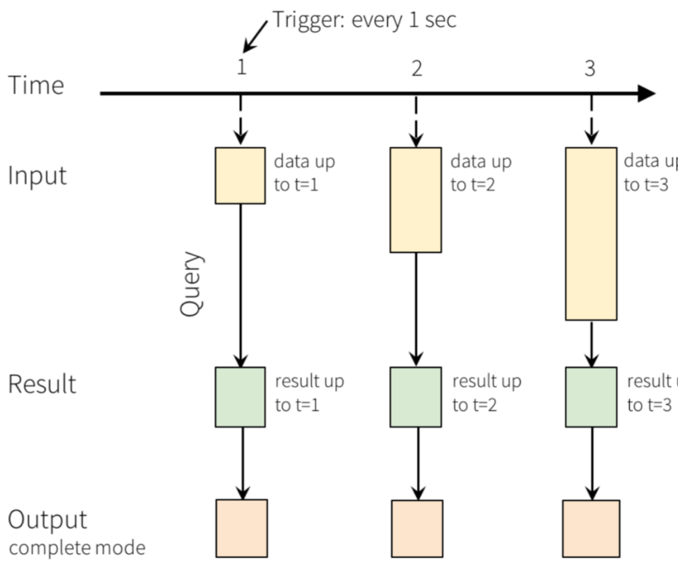
Programming Model

A query on the input will generate Result Table

Every trigger interval, let us say, every X seconds, the new rows get appended to Input Table

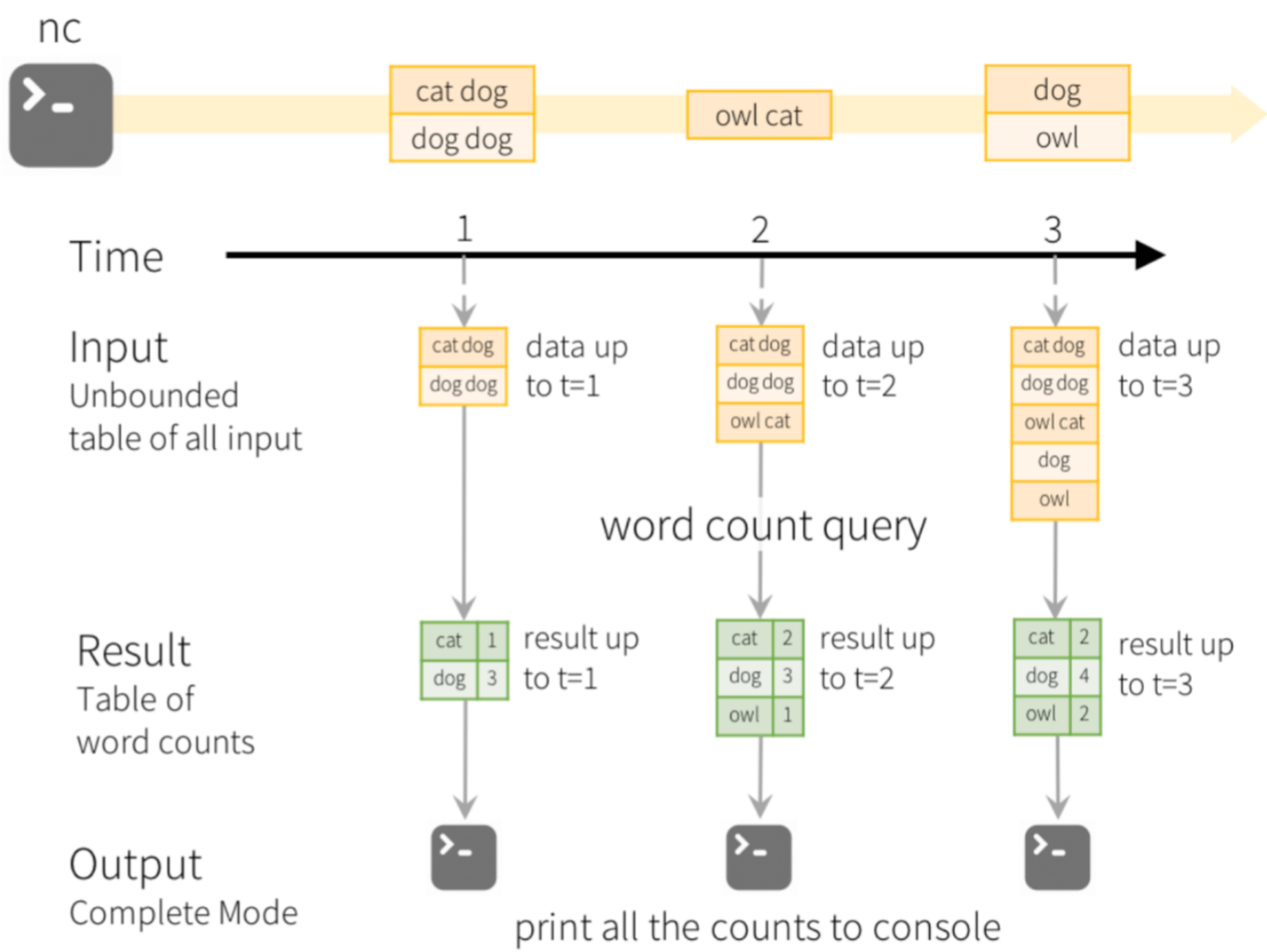
It will eventually updates Result Table

Whenever Result Table gets updated, we would want to write the changed result rows to an external sink



Programming Model

A complete process



Programming Model

Transformations and actions in Structured Streaming

- The same concepts of transformations and actions are used as in DataFrame/Dataset
- Additionally, Structured Streaming supports a subset of transformations applied to the Output Mode (presented later)
- Normally there is one action available in Structured Streaming: Start Streaming
- Starting Streaming runs continuously and produce results continuously
- Start Streaming is manifested in a different way in Spark shell and in a self-contained application (presented later)

Programming Model

Input sources

- Apache Kafka: input source is one or more topics in Kafka (Kafka is a distributed messaging system, providing high-performance, scalable message publish/subscribe services, used in data-intense production environment)
- File source: input source is a folder with files, for example HDFS
- Socket source: input source is a socket connection from a stimulated data server, for example NetCat
- Rate source: input source that generates data at the specified number of rows per second, putting a timestamp when the rows are dispatched (mainly used for testing)

Programming Model

Output sinks

- Apache Kafka: stores the output to one or more topics in Kafka
- File sink: stores the output to a folder, for example on HDFS, as text, csv, json, orc, parquet, and the others
- Console sink (for debugging): prints the output to the console/stdout
- Memory sink (for debugging): stores the output in-memory tables that can be queried later on
- "Foreach" sink: runs arbitrary computation on records in the output

Programming model

An Output Mode defines what records (rows) in the results are written out to an output sink

Output modes

- Complete mode: all results (up to the present time) are written to an output sink
- Append mode: only the new rows are appended to the results since the last trigger fired
- Update mode: only the rows that were updated in the results since the last trigger are written

Programming model

Whereas an output mode defines how data is output, triggers defines when data is output

For example, when structured streaming application should check for new data on input and update the results every 5 seconds

If unset, a default trigger is used

A default trigger reads the inputs as long as the previous batch of data is updated in the results

Practical issues with triggers

- Latency verus Throughput versus Computational burden

Programming model

Event time is the time when input data is produced/provided

- Usually it is embedded in the data as a timestamp
- It is important because it provides a more robust way of comparing events against one another
- By contrast, processing time refers to the time at which the stream-processing system actually receives that data

Structured streaming enable windows partition on data based on event time

It can also set watermarks to handle late data

For example, the newly received data will be kept up to a certain point of time (watermark) when we do not expect more late data

Programming model

Transformations on streams

Selections and filtering (applicable to all output modes)

- `select()`, `where()`, `filter()`

Aggregations (not applicable to append mode)

- `groupBy()`
- `agg()`, which usually follows `groupBy()` and contains operations like `count()`, `sum()`, `avg()` within it

Joins (applicable to all output modes)

- Currently supports the Join between atructured stream and a static DataFrame/Dataset

Transformation with function passing for Dataset streams

- `map()`, `flatMap()`, `filter()` and the others

Spark Structured Data and Stream Processing

Outline

Streaming Query Example

Imports

```
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.functions.{window, col, desc, sum}
import org.apache.spark.sql.streaming.Trigger
```

Query stream of retail data

```
object RetailDataStreamQuery {
```

Main

```
  def main(args: Array[String]): Unit = {
```

Spark session

```
    val spark = SparkSession.builder.appName(" ")
                                .config("spark.master", "local[*]")
                                .getOrCreate()
    spark.sparkContext.setLogLevel("ERROR")
    val retail_data = ".../retail-data/by-day/*.csv"

        // to be inserted...
```

Stop

```
    spark.stop()
}
```

Streaming Query Example

Create dataframe

```
val staticDataFrame = spark.read.format("csv")
                               .option("header", "true")
                               .option("inferSchema", "true")
                               .load(retail_data)
```

Create view

```
staticDataFrame.createOrReplaceTempView("retail_data")
```

Show schema

```
staticDataFrame.printSchema()
// root
// |-- InvoiceNo: string (nullable = true)
// |-- StockCode: string (nullable = true)
// |-- Description: string (nullable = true)
// |-- Quantity: integer (nullable = true)
// |-- InvoiceDate: timestamp (nullable = true)
// |-- UnitPrice: double (nullable = true)
// |-- CustomerID: double (nullable = true)
// |-- Country: string (nullable = true)
```

Streaming Query Example

Load static schema

```
val staticSchema = staticDataFrame.schema
```

Read stream

```
spark.conf.set("spark.sql.shuffle.partitions", 3)
val streamingDataFrame = spark.readStream
                              .schema(staticSchema)
                              .option("maxFilesPerTrigger", 10)
                              .format("csv")
                              .option("header", "true")
                              .load(retail_data)
```

Streaming

```
streamingDataFrame.isStreaming // true if streaming
```

Streaming Query Example

Process query

```
val purchaseByCustomerPerDay = streamingDataFrame
                               .selectExpr("CustomerId",
                                           "(UnitPrice * Quantity) as total_cost",
                                           "InvoiceDate")
                               .groupBy(col("CustomerId"),
                                        window(col("InvoiceDate"), "1 day"))
                               .agg(sum("total_cost").alias("TotalCostPerDay"))
                               .orderBy(desc("TotalCostPerDay"))
                               .where(col("CustomerId").isNotNull)
```

Trigger

```
  val query = purchaseByCustomerPerDay
                               .writeStream.format("console")
                               .queryName("customer_purchases")
                               .outputMode("complete")
                               .trigger(Trigger.ProcessingTime("4 seconds"))
                               .start()
```

```
// the following lines activates the query stream.
```

Final

```
query.awaitTermination(10000) // ms
```

Spark Structured Data and Stream Processing

Outline

Time Window Function

Time window is a standard function that generates stream time window ranges (on a timestamp column)

The function signature of time window

Time window
```
window( timeColumn: Column,
        windowDuration: String,
        slideDuration: String,
        startTime: String): Column
```

Parameters `slideDuration` and `startTime` are optional

If `slideDuration` is unset, its default value equals to the `windowDuration` value (tumbling window)

If `startTime` is unset, its default value is `0`

Spark Structured Data and Stream Processing

Outline

Stock Data Analysis Example

The stock trend of Apple Inc. (from Yahoo! Finance)

Currency in USD

Download data

Date	Open	High	Low	Close*	Adj. close**	Volume
14-Sep-2018	225.75	226.84	222.52	223.84	223.84	3,19,02,700
13-Sep-2018	223.52	228.35	222.57	226.41	226.41	4,17,06,400
12-Sep-2018	224.94	225.00	219.84	221.07	221.07	4,92,78,700
11-Sep-2018	218.01	224.30	216.56	223.85	223.85	3,57,49,000
10-Sep-2018	220.95	221.85	216.47	218.33	218.33	3,95,16,500

Out of those six columns in the data, we are interested in `Date`, which signifies the date of trade and `Close` which signifies end of the day value

Stock Data Analysis Example

Read input data

```
val stocks = spark.read
                  .option("header", "true")
                  .option("inferSchema", "true")
                  .csv(".../AAPL.csv")
```

Show data

```
 stocks.show(2)
```

Input data

```
+-------------------+--------+--------+--------+--------+---------+---------+
|               Date|    Open|    High|     Low|   Close|Adj Close|   Volume|
+-------------------+--------+--------+--------+--------+---------+---------+
|1980-12-12 00:00:00|0.513393|0.515625|0.513393|0.513393| 0.415317|117258400|
|1980-12-15 00:00:00|0.488839|0.488839|0.486607|0.486607| 0.393649| 43971200|
|        ...        |  ...   |  ...   |  ...   |  ...   |   ...   |   ...   |
+-------------------+--------+--------+--------+--------+---------+---------+
only showing top 2 rows
```

Stock Data Analysis Example

Imports

```
import org.apache.spark.sql.functions._
```

Filters

```
val stocks2017 = stocks.filter(year(col("Date"))===2017)
```

Aggregations

```
val winStock2017 = stocks2017.groupBy( window(col("Date"), "1 week") )
                             .agg(max(col("Close")), min(col("Close")))
                             .orderBy("window.start")
```

Stock Data Analysis Example

Show

```
winStock2017.show()
```

Results

```
+------------------------------------------+----------+----------+
|window                                    |max(Close)|min(Close)|
+------------------------------------------+----------+----------+
|[2016-12-29 11:00:00, 2017-01-05 11:00:00]|116.610001|116.019997|
|[2017-01-05 11:00:00, 2017-01-12 11:00:00]|119.750000|117.910004|
|[2017-01-12 11:00:00, 2017-01-19 11:00:00]|120.000000|119.040001|
|[2017-01-19 11:00:00, 2017-01-26 11:00:00]|121.940002|119.970001|
|                    ...                   |   ...    |   ...    |
+------------------------------------------+----------+----------+
only showing top 4 rows
```

References

A Gentle Introduction to Spark, Databricks, (Available in `READINGS` folder)

RDD Programming Guide

Spark SQL, DataFrames and Datasets Guide

Karau H., Fast data processing with Spark Packt Publishing, 2013 (Available from UOW Library)

Srinivasa, K.G., Guide to High Performance Distributed Computing: Case Studies with Hadoop, Scalding and SparkSpringer, 2015 (Available from UOW Library)

Chambers B., Zaharia M.,Spark: The Definitive Guide, O'Reilly 2017

Perrin J-G., Spark in Action, 2nd ed., Manning Publications Co. 2020