ST446 Distributed Computing for Big Data

Lecture 6

Stream processing systems

Milan Vojnovic

https://github.com/lse-st446/lectures2021

Goals of this lecture

- Learn basic principles of stream processing systems
- Learn about Spark streaming and structured streaming concepts
- Learn about some streaming algorithms

Topics of this lecture

- Stream processing systems basic principles
- Structured streaming
- Streaming algorithms
- Streaming algorithms (cont’d)

Stream processing basic principles

What is stream processing?

- Stream processing: a computer programming paradigm that allows some applications to exploit some limited form of parallel processing

- Stream processing is related to
  - Dataflow programming: models a program as a directed graph of the data flowing between operations
  - Event stream processing: refers to event-driven information systems
  - Reactive programming: declarative programming paradigm concerned with data streams

Example applications

- Algorithmic trading in financial services
- Radio-frequency identification (RFID) event processing applications
- Fraud detection
- Process monitoring
- Location-based services in telecommunications
- Internet of Things (IoT) applications
- Large-scale online platforms, ex mobile phone apps
- …

Stream processing systems

- Stream: a sequence of data elements made available over time
  - Stream data elements processed on at a time rather than in large batches
- Filters: functions that operate on a stream, produce another stream
  - May operate on one item of a stream at a time
  - Or base an item of output on multiple items of input
- Pipelines of function transformations
  - Correspond to function compositions
- Why streaming computations?
  - Reduce latency, allow for incremental data processing
  - Scalability: computation by making passes through data using a small memory footprint at any point in time

Spark streaming

- Scalable and fault-tolerant stream processing in two different ways

- Streaming: built on core Spark API
  - RDD batch paradigm operating over mini-batches or batch intervals
  - Ex batch intervals of 500 ms or larger window duration
  - Fault tolerance semantics: exactly once for stateful computations

- Structured streaming: built on Spark SQL engine
  - DataFrame API used to express streaming aggregations, event-time windows, stream-to-batch joins and other computations

System components

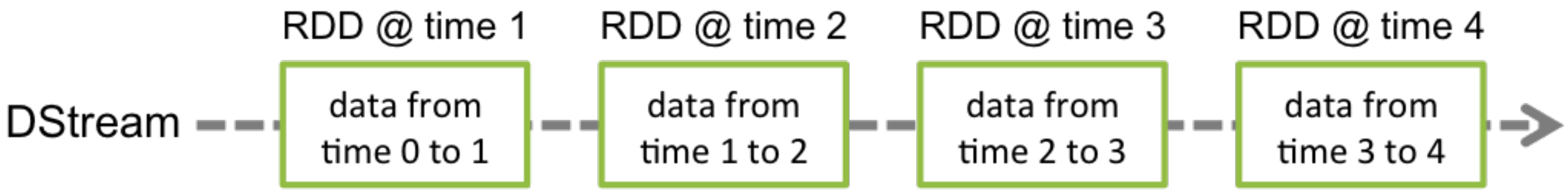
- Input streams can be ingested from different sources
- Data processed using operators like map, reduce, join and window, as well as by using machine learning and graph processing algorithms
- Output of data processing can be pushed out to different systems

Stream computation model

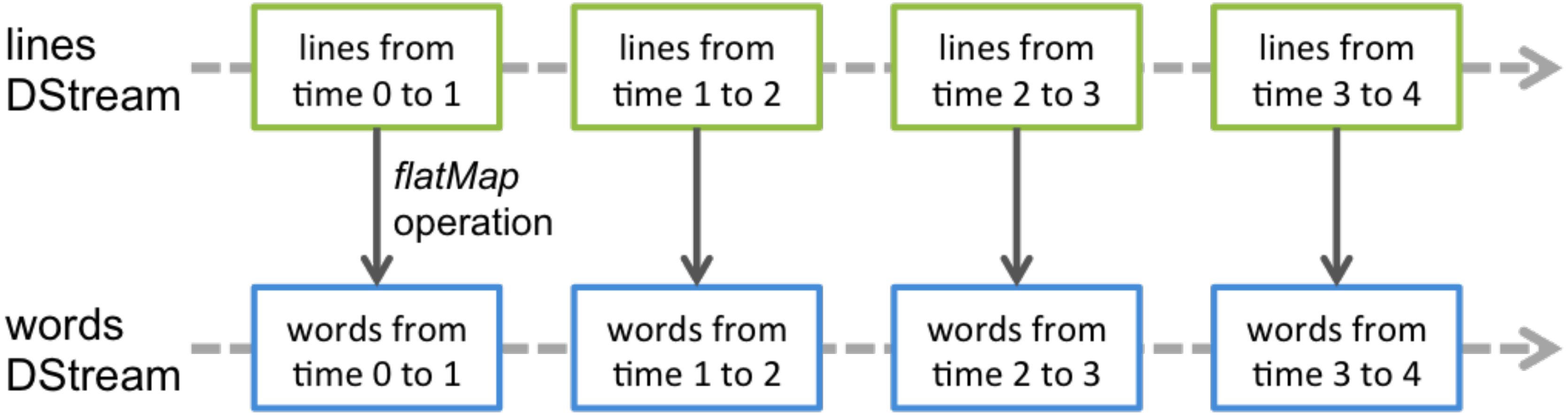
- Input: data stream
    - Divided into batches
- Output: batches processed by the Spark engine to generate the output stream
    - Output stream of batches

Discretized stream abstraction

- DStream: discretization of a continuous data stream
  - Internally represented as a sequence of RDDs
- An operation applied to a DStream is applied on the RDDs of this DStream



Discretized stream abstraction (cont'd)



Word count example

```python
from pyspark import SparkContext
from pyspark.streaming import StreamingContext

# Create a local StreamingContext with two threads and batch interval of 1 second
sc = SparkContext("local[2]", "NetworkWordCount")
ssc = StreamingContext(sc, 1)

# Create a DStream that will connect to hostname:port
lines = ssc.socketTextStream("localhost", 9999)
```

Word count example (cont'd)

```python
# Split each line into words
words = lines.flatMap(lambda line: line.split(" "))

# Count each word in each batch
pairs = words.map(lambda word: (word, 1))
wordCounts = pairs.reduceByKey(lambda x, y: x + y)

# Print the first ten elements of each RDD generated in this DStream to the console
wordCounts.pprint()

ssc.start()             # Start the computation
ssc.awaitTermination()  # Wait for the computation to terminate
```

Word count example (cont'd)

```
# TERMINAL 1: running netcat

$ nc -lk 9999

hello world
...

# TERMINAL 2: running network_wordcount.py

$ ./bin/spark-submit examples/src/main/python/streaming/network_wordcount.py localhost 9999
...
-------------------------------------------
Time: 2014-10-14 15:25:21
-------------------------------------------
(hello,1)
(world,1)
...
```

Stream sources

- Several built-in stream sources

- Basic: file systems, socket connections
  - Available through StreamingContext API

- Advanced: publish-subscribe systems, other stream processing systems
  - Ex Kafka, Flume, Kinesis
  - Requires linking against extra dependencies

File system sources

- Reading data from files in any file system compatible with HDFS API
- Created by `streamingContext.textFileStream(dataDirectory)`
- Spark streaming monitors and processes any files created in specified directory
  - Nested directories not supported, files must have the same format
  - Files must be created by atomically moving or renaming them
  - Once moved, the files must not be changed
  - If a file is continuously appended, new data is not read

Receiver types

- Reliable: correctly send acknowledgement to a reliable source when the data has been received and stored in Spark with replication

- Unreliable: do not send acknowledgements to the source

Stateful transformations

- Stateful transformation: maintains an arbitrary state that can be updated as new information is received
  - Can be defined by `updateStateByKey`

- User needs to specify:
  - State (arbitrary data type)
  - State update function: mapping of current state to new state as new information is received in the input stream

- Returns a new state (DStream)
  - State for each key is updated by applying given state update function
  - Allows to maintain arbitrary state for each key

Example: stateful word count

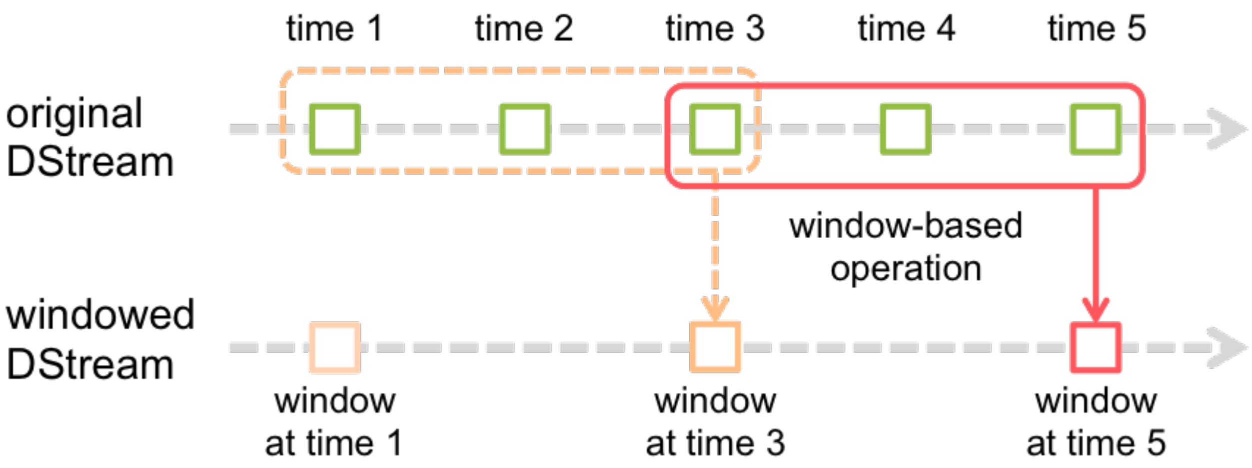
```python
import sys

from pyspark import SparkContext
from pyspark.streaming import StreamingContext

sc = SparkContext(appName="PythonStreamingStatefulWordCount")
ssc = StreamingContext(sc, 1)
ssc.checkpoint("checkpoint")

# RDD with initial state (key, value) pairs
initialStateRDD = sc.parallelize([(u'hello', 1), (u'world', 1)])
```

Example: stateful word count (cont'd)

```
def updateFunc(new_values, last_sum):
    return sum(new_values) + (last_sum or 0)

lines = ssc.socketTextStream(sys.argv[1], int(sys.argv[2]))
running_counts = lines.flatMap(lambda line: line.split(" "))\
                      .map(lambda word: (word, 1))\
                      .updateStateByKey(updateFunc, initialRDD=initialStateRDD)

running_counts.pprint()

ssc.start()
ssc.awaitTermination()
```

Window operations

- Window operation: apply a transformation on a window of input data
  - Window length: duration of the window
  - Sliding interval: interval at which the window operation is performed



Common window operations

Transformation	Meaning
window(*windowLength*, *slideInterval*)	return new DStream computed based on windowed batches of the source DStream
countByWindow(*windowLength*, *slideInterval*)	return a sliding window count of elements
reduceByWindow(*func*, *windowLength*, *slideInterval*)	return a new single-element stream by aggregating elements in the stream over a sliding window interval using *func*
reduceByKeyAndWindow	return a new DStream of (k,v) pairs from an input DStream of (k,v) pairs by aggregating values for each key using the given reduce function *func* over batches in a sliding window
reduceByKeyAndWindow(*func*, *invFunc*, *windowLength*, *slideInterval*, *[numTasks]*)	the reduce values of each window computed incrementally
countByValueAndWindow(*windowLength*, *slideInterval*, *[numTasks]*)	returns a new DStream of (k,v) pairs for an input DStream of (k,v) pairs where the value of each key is its frequency within a sliding window

Join operations

- Stream-stream join: `joinedStream = stream1.join(stream2)`
  - Per batch interval:

    RDD of stream1 is joined with RDD of stream2

  - Joins over windows:

    ```
    windowedStream1 = stream1.window(20)
    windowedStream2 = stream2.window(60)
    joinedStream = windowedStream1.join(windowedStream2)
    ```

- Stream-dataset join:

  ```
  dataset = … # some RDD
  windowedStream = stream.window(20)
  joinedStream = windowedStream.transform(lambda rdd:
  rdd.join(dataset))
  ```

Output operations on DStreams

- Output operations: allow DStream data to be pushed to an external system

Output operation	**Meaning**
pprint()	Print the first ten element of every batch of data in DStream on the driver node running the streaming application
saveAsTextFiles(*prefix*, [*suffix*])	save the content of this DStream as text files with file names *prefix-time_in_ms[.suffix]*
foreachRDD(*func*)	The most generic output operator that applies function *func* to each RDD generated from the stream (ex saving the RDD to files or writing it over the network to a database)

Other operations

- DataFrame and SQL
  - RDDs can be converted to DataFrames to use DataFrame API
  - DataFrames can be converted to tables to use Spark SQL

  To probe further: streaming programming guide dataframe and sql operations

- MLlib
  - Built-in streaming machine learning algorithms (linear regression, k-means, …)
  - Non-streaming machine learning algorithms: train offline, apply online

  To probe further: streaming programming guide mllib operations

Structured streaming

Structured streaming

- Scalable and fault-tolerant stream processing engine built on Spark SQL engine

- Streaming operations expressed in a similar way as for batch processing on static data

- Spark SQL engine takes care of running computations incrementally and continuously updating the output results as input data is received

- System ensures end-to-end exactly-once fault tolerance through
  - Checkpointing
  - Write ahead logs

Structured streaming word count example

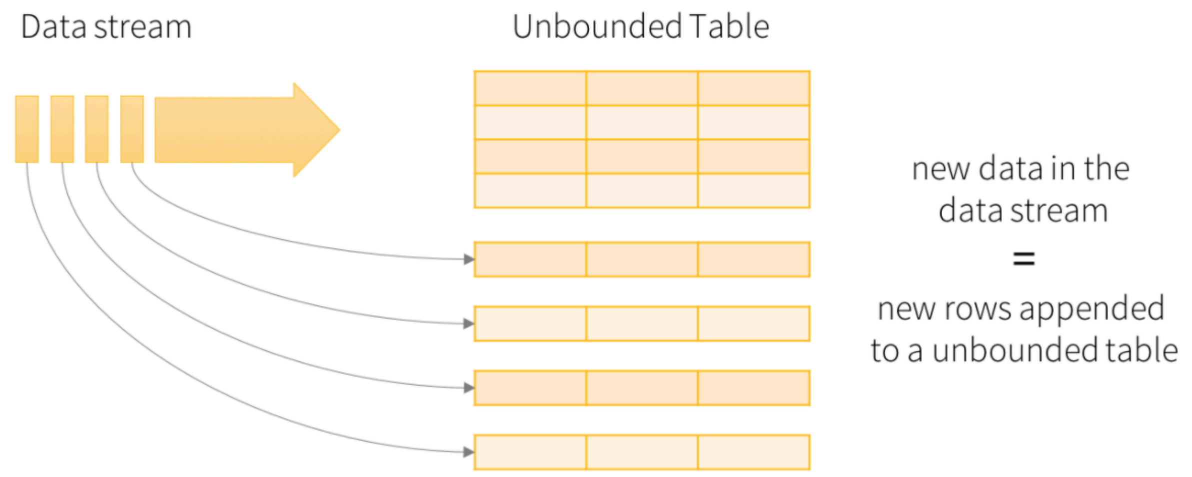
```
from pyspark.sql import SparkSession
from pyspark.sql.functions import explode
from pyspark.sql.functions import split

spark = SparkSession \
    .builder \
    .appName("StructuredNetworkWordCount") \
    .getOrCreate()

# create DataFrame representing the stream of input lines from connection to localhost:9999
lines = spark \
    .readStream \
    .format("socket") \
    .option("host", "localhost") \
    .option("port", 9999)
    .load()
```

Structured streaming word count example (cont'd)

```
# split the lines into words
words = lines.select(
    explode(
        split(lines.value, " ")
    ).alias("word")
)

# generate running word count
wordCounts = words.groupby("word").count()

# start running the query that prints the running counts to the console

query = wordCounts \
    .writeStream \
    .outputMode("complete") \
    .format("console")
    .start()

query.awaitTermination()
```

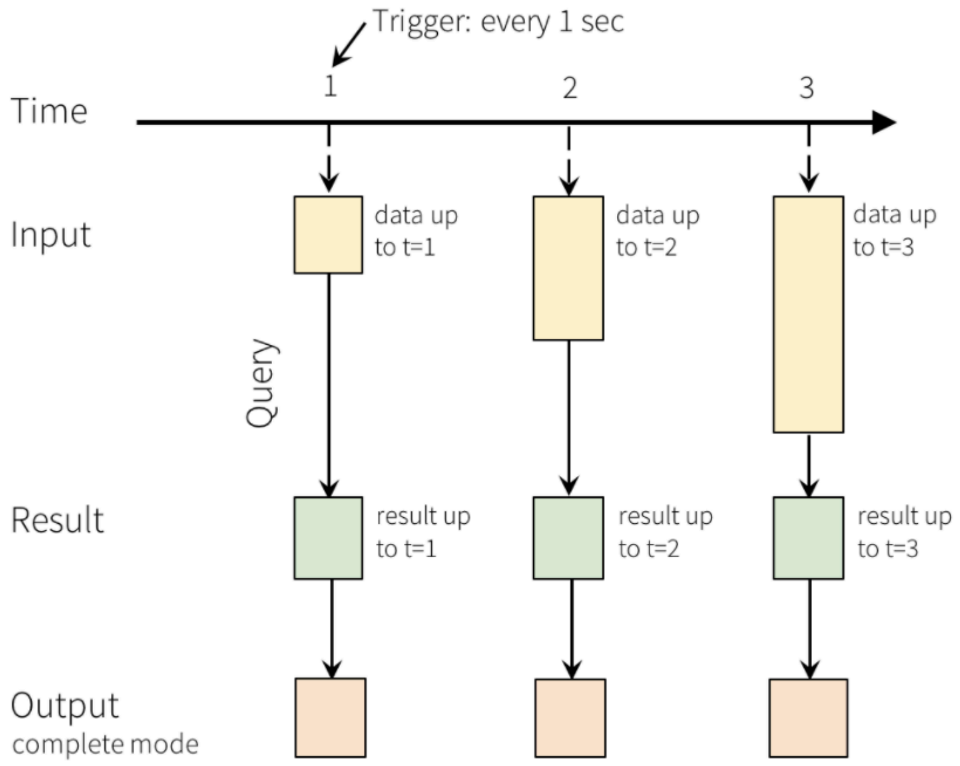
Programming model

- Data stream treated as a table that is being continuously appended
- Stream processing model like batch processing model
- Stream computations expressed as standard batch queries on a static table
- Spark runs it as an incremental query on an unbounded input table



Programming model (cont'd)

- Every trigger interval new rows appended (if any) to the input table
- This eventually updates the result table
- Whenever the result table gets updated, the changed result is written to an external sink



Output modes

- Complete: entire updated result table is written to external storage
- Append: only new rows appended in the result table since the last trigger are written to external storage
- Update: only the rows updated in the result table since the last trigger are written in external storage

API using Datasets and DataFrames

- DataFrames and DataSets can represent streaming unbounded data

- Streaming DataFrames can be created by using `DataStreamReader` interface returned by `SparkSession.readStream()`

- Built-in sources:
  - File sources: reads files written in a directory as a stream of data (supported file formats: text, csv, json, parquet)
  - Kafka source: poll data from Kafka (pub-sub system)
  - Socket source (for testing): reads UTF8 text data from a socket
  - Rate source (for testing): generates data at specified number of rows per second, each output row contains a timestamp and a value

To probe further: [structured streaming kafka integration]

Example: file source

```
spark = SparkSession ...

# Read all the csv files written atomically in a directory
userSchema = StructType().add("name", "string").add("age", "integer")
csvDF = spark \
    .readStream \
    .option("sep", ";") \
    .schema(userSchema) \
    .csv("/path/to/directory")  # Equivalent to format("csv").load("/path/to/directory")
```

Example: socket source

```
spark = SparkSession ...

# Read text from socket
socketDF = spark \
    .readStream \
    .format("socket") \
    .option("host", "localhost") \
    .option("port", 9999) \
    .load()

socketDF.isStreaming()    # Returns True for DataFrames that have streaming sources

socketDF.printSchema()
```

Basic operations: selection, projection, aggregation

```
df = ... # streaming DataFrame with IOT device data with schema {device: string, deviceType: string, signal: do
uble, time: DateType}

# select the devices which have signal more than 10
df.select("device").where("signal > 10")

# running count of the number of updates for each device type
df.groupBy("deviceType").count()
```

Spark SQL API

- Example: registering a streaming DataFrame as a temporary view and then applying SQL commands to it

```
df.createOrReplaceTempView("updates")
spark.sql("select count(*) from updates")  # returns another streaming DF
```

- Check whether a DataFrame has streaming data or not:

```
df.isStreaming()
```

Window operations

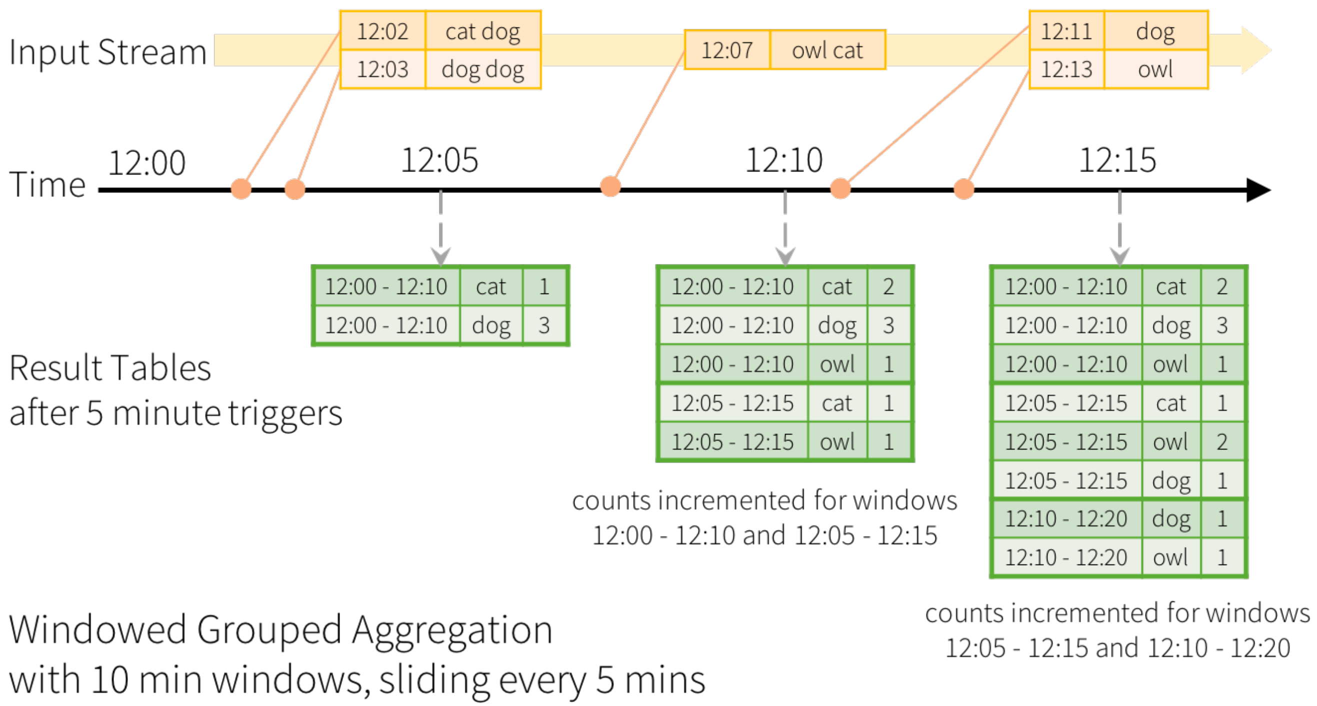
- Aggregations over a sliding event-time window with structured streaming are similar to group by aggregations

- Aggregate values are maintained for each unique value in the user-specified grouping column

- For window-based aggregations, aggregate values are maintained for each window in the event-time of a row falls into

- Example: count words within 10 mins windows, updating every 5 mins

Window operations (cont'd)



Window operations (cont'd)

Python code:

```
words = ...  # streaming DataFrame of schema { timestamp: Timestamp, word: String }

# Group the data by window and word and compute the count of each group
windowedCounts = words.groupBy(
    window(words.timestamp, "10 minutes", "5 minutes"),
    words.word
).count()
```

Handling late data and watermarking

- Input stream data elements may arrive with delay
- To update window aggregates with late data, the system needs to keep intermediate in-memory state it accumulates
- The amount of intermediate in-memory state must be bounded
- Watermarking: maintains an intermediate in-memory state to allow for late arrivals up to a prespecified threshold
    - For a window started at time t, the engine maintains the state and allows late date to update the state if max event time seen by the engine < t + late threshold

Watermarking example

```
words = ... # streaming DataFrame of schema {timestamp: Timestamp, word: String}

# Group the data by window and word and compute the count of each group
windowedCounts = words \
    .withWatermark("timestamp", "10 minutes") \
    .groupBy(
        window(words.timestamp, "10 minutes", "5 minutes"),
        words.word) \
    .count()
```

Join operations

- Streaming DataFrame can be joined with static DataFrame to create new streaming DataFrame

- Example:

```
staticDf = spark.read. ...
streamingDf = spark.readStream. ...
streamingDf.join(staticDf, "type")  # inner equi-join with a static DF
streamingDf.join(staticDf, "type", "right_join")  # right outer join with a static DF
```

Arbitrary stateful operations

- Many use cases require more advanced stateful operations than aggregations
  - Ex track sessions from data streams of events

- This can be done by using the operation `mapGroupsWithState` and the more powerful operation `flatMapGroupsWithState`

  - Both allow to apply a user-defined code on grouped Datasets to update a user-defined state

Managing streaming queries

- `StreamingQuery` object is crated when a query is started, which can be used to monitor and manage the query

- Examples:

```
# get the query object
query = df.writeStream.format("console").start()

# get the unique identifier of the running query that persists across restarts from checkpoint data
query.id()

# get the unique id of this run of the query, which will be generated at every start/restart
query.runId()

# get the name of the auto-generated or user-specified name
query.name()

# print detailed explanations of the query
query.explain()
```

Managing streaming queries (cont'd)

```
# stop the query
query.stop()

# block until query is terminated, with stop() or with error
query.awaitTermination()

# the exception if the query has been terminated with error
query.exception()

# an array of the most recent progress updates for this query
query.recentProgress()

# the most recent progress update of this streaming query
query.lastProgress()
```

Streaming algorithms

Examples of queries

- Count distinct: compute the number of distinct elements in a multiset of elements
- Heavy hitters: identify items that occur more frequently than a prespecified threshold or more frequently relative to other items (top k selection)
- Quantile computation: given $\phi \in [0,1]$, compute the ϕ-quantile value for a multiset of elements
- Sample: given an integer $k \geq 1$, draw a uniform random sample of size k (with or without replacement) from a multiset of items
- Frequency moments: given a multiset of elements with frequency vector $\boldsymbol{a} = (a_1, a_2, \dots, a_n)^\top$ and $p \geq 0$, compute the frequency moment $F_p(\boldsymbol{a}) = \sum_{i=1}^{n} |a_i|^p$

Streaming algorithms

- Streaming algorithms: algorithms for processing data stream in which the input is presented as a sequence of items that can be examined only in a few passes, ex. 1 pass

- Desiderata: answer queries about the data stream by using an algorithm that guarantees an error tolerance and has small space, update and query complexity

- Space complexity: a bound on the memory size used by the algorithm at any time
  - Desired to be sublinear in the input size (number of elements of the input stream)
- Update complexity: computation complexity of updating the state of the algorithm
  - Desired to be small, ex of constant order
- Query complexity: bound on the computation complexity for answering a query
  - Desired to be small, ex of constant order

Streaming algorithms (cont'd)

- Streaming algorithms typically use a data structure (defining the state of the algorithm) that is updated for each addition of an element of the input stream
  - Probabilistic data structure: a data structure designed for approximately answering a class of queries with some space, update, and query complexity
  - (ϵ, δ)-accuracy: answer to a query is within an (absolute or relative) ϵ-error tolerance with probability at least $1 - \delta$

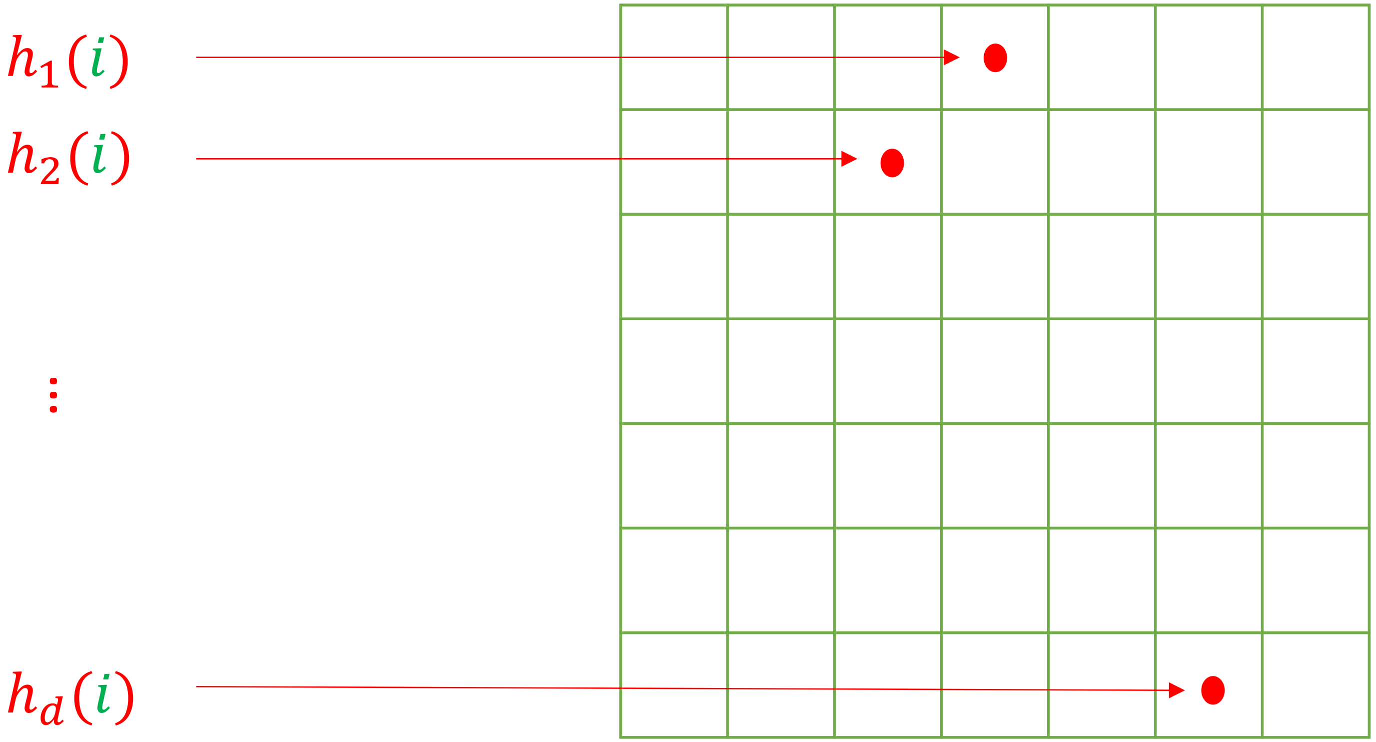
Input stream:	Update:
(i_t, c_{i_t})	$a_{i_t} \leftarrow a_{i_t} + c_{i_t}$

- Input stream conditions:
  - Cash register: c_i's are strictly positive (additions only)
  - Turnstile: c_i's can be negative (additions and deletions)
- Having a condition on the input stream may allow to design more efficient streaming algorithms (ex. cash register vs more general turnstile)

Count-min

- Count-min: a streaming algorithm for approximately answering queries about values of elements of a vector $\boldsymbol{a} \in \mathbf{R}^n$ (updated by a stream of value updates)

Input: c_i



depth d

width w

Hash functions $h_1, h_2, \ldots, h_d$: $\{1, \ldots, n\} \rightarrow \{1, \ldots, w\}$
uniform random pairwise-independent functions

Count-min (cont'd)

- Update procedure:

  Initialization: $\mathrm{count}[j, i] = 0$ for $1 \leq j \leq d$ and $1 \leq i \leq w$

  For each input (i_t, c_t) update the data structure as follows

  For $j = 1$ to d

  $$\mathrm{count}[j, h_j(i_t)] \leftarrow \mathrm{count}[j, h_j(i_t)] + c_t$$

- Query answer procedure: answer to query about value of a_i:

  $$\hat{a}_i \leftarrow \min\{\mathrm{count}[j, h_j(i)]: j = 1, \dots, d\}$$

Count-min accuracy guarantee

- Assume $d = \lceil \log\left(\frac{1}{\delta}\right) \rceil$ and $w = \left\lceil \frac{e}{\epsilon} \right\rceil$

- Thm: For every $\epsilon \in (0,1]$ and $\delta \in (0,1)$, count-min algorithm satisfies, for all $i \in \{1,2, \ldots, n\}$,

$$\mathbf{P}[\widehat{a_i} \geq a_i] = 1$$

  and

$$\mathbf{P}[\hat{a}_i \leq a_i + \epsilon \|\boldsymbol{a}\|_1] \geq 1 - \delta$$

- Space complexity in words: wd for array of counts plus $O(d)$ for storing hash functions

  Hence, total space complexity = $O\left(\frac{1}{\epsilon} \log\left(\frac{1}{\delta}\right)\right)$ words

Count-min accuracy guarantee (cont'd)

- Simple proof by using Markov's inequality: $\mathbf{P}[X > x] \leq \frac{\mathbf{E}[X]}{x}$

- Let us decompose $\mathrm{count}\left[j, h_{j(i)}\right] = a_i + X_{i,j}$ where

$$X_{i,j} = \sum_{i' \neq i} a_{i'} \mathbf{1}_{\{h_j(i) = h_j(i')\}}$$

- Clearly, $\mathrm{count}\left[j, h_{j(i)}\right] \geq a_i$ for all $j \in \{1, \dots, d\}$, hence $\mathbf{P}[\hat{a}_i \geq a_i] = 1$

- Now, note

$$\mathbf{E}\left[X_{i,j}\right] = \sum_{i' \neq i} a_{i'} \mathbf{P}[h_j(i) = h_j(i')] \leq \sum_{i'=1}^{n} a_{i'} \frac{1}{w} \leq \frac{1}{e} \epsilon \|\boldsymbol{a}\|_1$$

Count-min accuracy guarantee (cont'd)

- Finishing steps:

$$\mathbf{P}[\hat{a}_i > a_i + \epsilon\|\boldsymbol{a}\|_1] = \mathbf{P}\left[\cap_{j=1}^{d} \{\mathrm{count}[j, h_j(i)] > a_i + \epsilon\|\boldsymbol{a}\|_1\}\right]$$

$$= \mathbf{P}\left[\cap_{j=1}^{d} \{X_{i,j} > \epsilon\|\boldsymbol{a}\|_1\}\right]$$

$$\leq \mathbf{P}\left[\cap_{j=1}^{d} \{X_{i,j} > e\mathbf{E}[X_{i,j}]\}\right]$$

$$\leq e^{-d}$$ (Markov inequality)

$$\leq \delta$$

Count distinct: HyperLogLog algorithm

Input: stream of items from multiset M of items from domain D

Initialization: $m = 2^b$ for positive integer b, $\alpha_m \leftarrow \left(m \int_0^\infty \left[\log\left(\frac{2+u}{1+u}\right)\right]^m du\right)^{-1}$

For $i = 1$ to m, $C[i] = -\infty$ # count registers

For $a \in M$:

$\boldsymbol{x} \leftarrow h(a)$ // hash function $h : D \rightarrow \{0,1\}^\infty$

$j \leftarrow 1 + \mathrm{bin2dec}(x_1 x_2 \cdots x_b)$ // partitioning points over m disjoint sets

$\boldsymbol{w} \leftarrow x_{b+1} x_{b+2} \cdots$

$C[j] \leftarrow \max\{C[j], \rho(\boldsymbol{w})\}$ // $\rho(\boldsymbol{w})$:= position of the leftmost bit of value 1

$Z \leftarrow \left(\sum_{j=1}^m 2^{-C[j]}\right)^{-1}$

Return $N \leftarrow \alpha_m m^2 Z$

Intuition

- Each subset of points indexed by j will have about $\frac{n}{m}$ elements
- Each $C[j]$ should be of value about $\log_2\left(\frac{n}{m}\right)$
- The harmonic mean $Z \leftarrow \left(\sum_{j=1}^{m} 2^{-C[j]}\right)^{-1}$ should be of value near to $\frac{n}{m^2}$
- Thus $m^2 Z$ should be of value approximately n

HyperLogLog approximation guarantee

- Thm: Assume that the hash function maps items to independent uniformly distributed random numbers in [0,1].

  For a multiset M with n distinct items and $m \geq 3$, HyperLogLog algorithm satisfies

  - Nearly asymptotically unbiased:

    $$\frac{1}{n}\mathbf{E}[N] = 1 + c_n + o(1) \text{ where } |c_n| \leq 0.000005 \text{ for all } m \geq 16$$

  - Standard error:

    $$\frac{1}{n}\sqrt{\mathbf{Var}[N]} = \beta_m \frac{1}{\sqrt{m}} + c_n' + o(1) \text{ where } |c_n'| \leq 0.00005 \text{ for all } m \geq 16$$

    where β_m is a sequence bounded by constants

    $$\lim_{m \to \infty} \beta_m = \sqrt{3\log(2) - 1} \approx 1.039$$

HyperLogLog approximation guarantee (cont'd)

- To guarantee relative error ϵ with a constant probability, it suffices that

$$\frac{1}{n}\sqrt{\mathbf{Var}[N]} = \beta_m \frac{1}{\sqrt{m}} \le \epsilon$$

- Hence, it is sufficient that

$$m = O\left(\frac{1}{\epsilon^2}\right)$$

- Space complexity:

$$m \underbrace{\log(\log(n))}_{\substack{\text{Number of bits for} \\ \text{a single register count}}} + O(\underbrace{\log(n)}_{\substack{\text{Number of bits for} \\ Z,\ \alpha_m}}) = O\left(\frac{1}{\epsilon^2}\log(\log(n)) + \log(n)\right) \text{ bits}$$

HyperLogLog: applications

- Counting unique values in BigQuery with HyperLogLog++

  To probe further:

  [Google Cloud Platform Blog]

- Counting the number of views of a post at Reddit

  To probe further:

  [View Counting at Reddit]

- Facebook Presto

  To probe further:

  [HyperLogLog in Presto: A significantly faster way to handle cardinality estimation]

Streaming algorithms (cont’d)

Reservoir sampling

- Reservoir sampling: a family of randomized algorithms for selecting a random sample of `k` items (with or without replacement) from a population of unknown size `n` in a *single pass* over the input set of items
- Simple algorithm
  Input stream: `s[1], s[2], …, s[n]`

```
For i = 1 to k:
    r[i] <- s[i]

For i = k + 1 to n
    j <- random integer [1,i]
    If j <= k:
         r[j] <- s[i]
```

- Claim: Every given item is in the final sample with probability `k/n` (show this)
- Space complexity: `k` elements at any time
- Computation complexity: $\Theta(n)$ computations (inefficient – we can do better!)

Optimal reservoir sampling

- Key idea: compute how many input items are discarded before the next item enter the reservoir (this follows a geometric distribution and can be computed in constant time)

```
Input stream: s[1], s[2], …, s[n]
For i = 1 to k:
      r[i] <- s[i]

w <- random(0,1)**(1/k)
While i <= n
      i <- i + floor(log(random(0,1))/log(1-w)) + 1          (*)
      if i <= n
            r[rndInt[1,k]] <- s[i]
            w <- w * random(0,1)**(1/k)                        (**)
```

- Computation complexity: $O\left(k\left(\log\left(\frac{n}{k}\right)+1\right)+c_n\right)$ where c_n is the computation time to scan a sequence of length n

Optimal reservoir sampling (cont'd)

- The reservoir sampling emulates the following algorithm:
  - Assign values to elements of the input stream according to independent samples from uniform distribution on `[0,1]`
  - Taking `k` elements with the smallest values is a random sample without replacement

- Variable `w` represents the maximum value of an item in the reservoir

- (❉) step: the algorithm samples the number of input elements to skip over before a new element is added to the reservoir
  - The number of elements to skip follows the geometric distribution with parameter `w`
  - Sampling from the geometric distribution is realized by the inverse CDF method

Optimal reservoir sampling (cont'd)

- (❋ ❋) step: for a full reservoir with maximum value of an item in the reservoir equal to `w`, conditional on adding an item to the reservoir (by replacing a randomly picked item in the reservoir), the new maximum value has distribution of a random variable defined as the maximum value of `k` independent samples from uniform distribution on `[0,w]`

- The expected number of times step (❋ ❋) is performed

$$= 1 + \sum_{i=k+1}^{n} \mathbf{P}[i\text{th element added to the reservoir}]$$

$$= 1 + \sum_{i=k+1}^{n} \frac{k}{i}$$

$$= k \log\left(\frac{n}{k}\right) + O(1)$$

Streaming linear regression

- Input stream: $(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \dots, (\boldsymbol{x}_n, y_n)$
- Loss function (ordinary least squares or linear least squares):

$$f(\boldsymbol{w}) = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{2}\left(y_i - \boldsymbol{x}_i^{\top}\boldsymbol{w}\right)^2$$

- May also use regularization
  - Ex. Ridge regression (L2 regularization) or Lasso (L1 regularization)

- Fitting of parameter vectors on each batch of input stream data

- Parameter vector update: $\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta\widehat{\nabla f}(\boldsymbol{w})$
  - Where $\widehat{\nabla f}(\boldsymbol{w})$ is the gradient vector computed for an input batch of data

Streaming linear regression (cont'd)

```python
import sys

from pyspark.mllib.linalg import Vectors
from pyspark.mllib.regression import LabeledPoint
from pyspark.mllib.regression import StreamingLinearRegressionWithSGD

def parse(lp):
    label = float(lp[lp.find('(') + 1: lp.find(',')])
    vec = Vectors.dense(lp[lp.find('[') + 1: lp.find(']')].split(','))
    return LabeledPoint(label, vec)

trainingData = ssc.textFileStream(sys.argv[1]).map(parse).cache()
testData = ssc.textFileStream(sys.argv[2]).map(parse)

numFeatures = 3
model = StreamingLinearRegressionWithSGD()
model.setInitialWeights([0.0, 0.0, 0.0])

model.trainOn(trainingData)
print(model.predictOnValues(testData.map(lambda lp: (lp.label, lp.features))))

ssc.start()
ssc.awaitTermination()
```

Source: https://spark.apache.org/docs/latest/mllib-linear-methods.html#streaming-linear-regression

References

- Spark streaming programming guide
- Structured streaming programming guide
- Python streaming examples
- Spark-Kafka integration guide
- Spark-Kinesis integration guide
- T. Das, A Deep dive into structured streaming, Spark Summit 2016
- Structured Kafka streaming wordcount example
- Zaharia M., Das T., Li H., Hunter T., Shenker S. and Stoica I., Discretized Streams: A Fault-Tolerant Model for Scalable Stream Processing, Technical Report UCB/EECS-2012-259, University of Berkeley, 2012

References (cont'd)

- Karau H., Konwinski A., Wendell P. and Zaharia M., Learning Spark, O'Reilly, 2015
  - Chapter 10: Spark Streaming

- Drabas T. and Lee D., Learning PySpark, Packt, 2016
  - Chapter 10: Structured Streaming

- B. Chambers and M. Zaharia, Spark: The Definitive Guide

References (cont'd)

- P. Flajolet and G. N. Martin, Probabilistic Counting Algorithms for Data Base Applications, Journal of Computer and System Sciences, 31, 182-209, 1985, online copy here
- Huele et al, HyperLogLog in Practice: Algorithmic Engineering of a State of the Art Cardinality Estimation Algorithm, EDBT/ICDT 2013, online copy here
- G. Cormode and S. Muthukrishnan, An Improved Data Stream Summary: The Count-Min Sketch and its Applications, Journal of Algorithms, Vol 55, 29-38, 2005, online copy here
- J. S. Vitter, Random Sampling with a Reservoir, ACM Trans. on Mathematical Software, Vol 11, No 1, 1985, online copy here
- K.-H. Li, Reservoir-Sampling Algorithms of Time Complexity O(n(1+log(N/n))), ACM Trans. on Mathematical Software, Vol 20, No 4, 1994
- C. M. Bishop, Pattern Recognition and Machine Learning, Section 3.3 Bayesian Linear Regression, MIT Press, 2006, online copy here
- Bahmani et al, Scalable K-Means++, VLDB 2012, online copy here

Seminar class 6: Spark streaming

- Basic Spark streaming examples
- DataFrames and SQL example
- Stateful operations example
- HDFS word count
- Kafka
- Twitter example

```
-------------------------------------------
Time: 2019-02-28 21:35:10
-------------------------------------------
('trump', 24)
('rt', 20)
('https', 18)
('cohen', 12)
('tax', 5)
('nt', 5)
('texas', 3)
('maxine', 3)
('wants', 3)
('investigate', 3)
...

-------------------------------------------
Time: 2019-02-28 21:35:11
-------------------------------------------
('rt', 27)
('https', 19)
('trump', 17)
('president', 6)
('nt', 5)
('house', 3)
('cohen', 3)
('troops', 3)
('amp', 3)
('la', 3)
...

-------------------------------------------
Time: 2019-02-28 21:35:12
-------------------------------------------
('trump', 31)
('rt', 22)
('https', 18)
('donald', 6)
('ivanka', 4)
('kim', 4)
('people', 4)
('house', 3)
('jong', 3)
('un', 3)
...
```

https://github.com/lse-st446/lectures2021/blob/master/Week07/class/README.md

Extras

Fault tolerance semantics

- Sources, sinks, and execution engine are designed to reliably track progress of the processing so that any kind of failure by restarting and/or reprocessing can be handled

- Every streaming source has offsets to track the read position in the stream
  - Like Kafka's offsets or Kinesis sequence numbers

- The engine uses checkpointing (saving application state) and write-ahead-logs to record the offset range of the data being processed in each trigger

- Streaming sinks are designed to be idempotent (can be applied multiple times producing the same result) for handling reprocessing

- All the mechanisms put together guarantee end-to-end exactly-once semantics under any failure

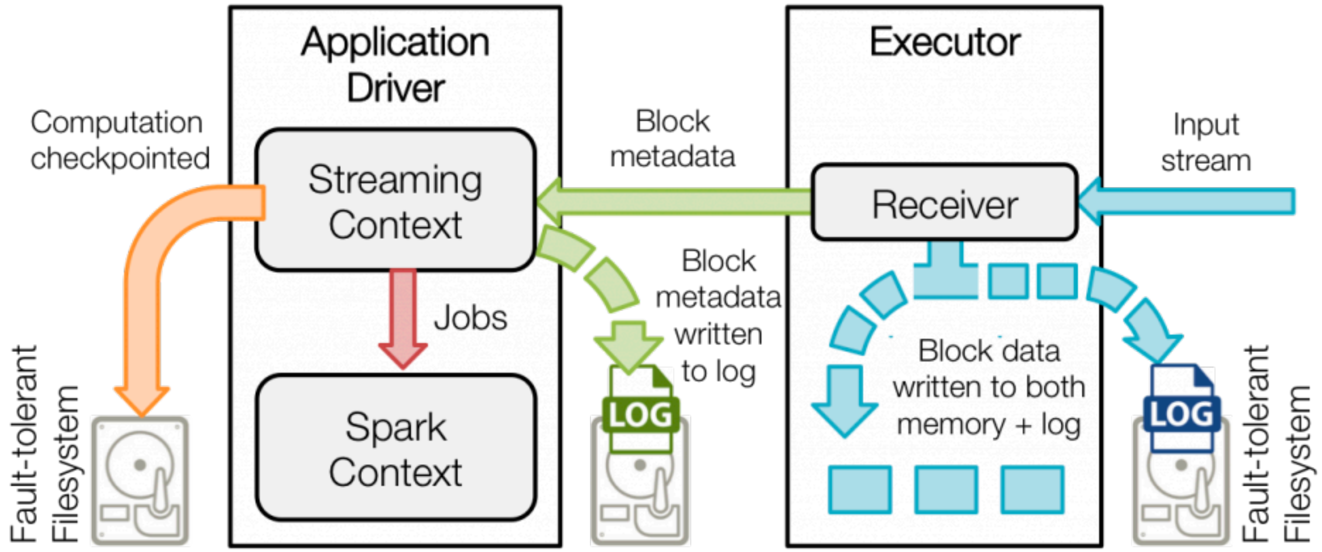
Fault tolerance: zero data loss

- For some data sources, input data could get lost while the system is recovering from failures
  - Either driver or worker machine failures
- Spark and its RDD abstraction designed to seamlessly handle failures of any worker node
- Streaming applications require also recovery from driver process failures
- Streaming computation structured around processing minibatches allows to save (checkpoint) the application state periodically in a reliable storage and recover the state on driver restarts
- For some data sources (ex Kafka and Flume): some of the received data that was buffered in memory not yet processed could get lost
  - Buffered data cannot be recovered even if the driver is restarted
  - Write ahead logs used to resolve this

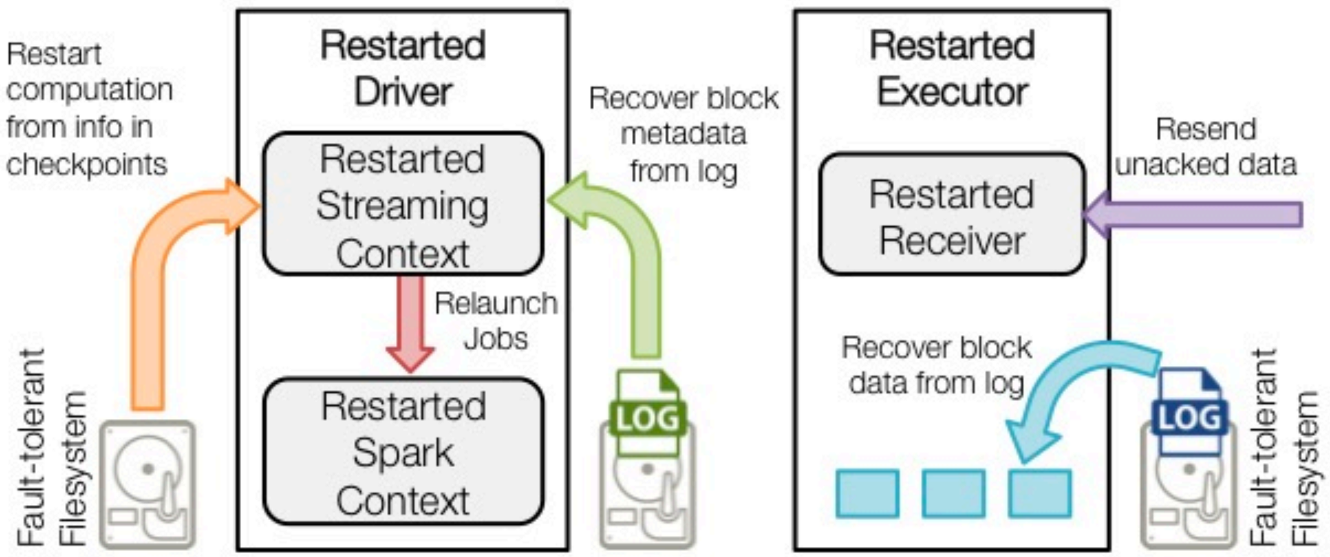
Write ahead logs

- Write ahead logs: the intention of the operation is first written down in a durable log, and then the operation is applied to the data
  - Also referred to as journaling
  - If the system fails in the middle of applying an operation, it can recover by reading the log and reapplying the operation it has intended to do
  - Used in databases and file systems to ensure durability of data operations
- Main concepts:
  - All the received data is saved to log files in a fault-tolerant file system
  - Receiver correctly acknowledges receiving data only after the data has been saved in write ahead logs (buffered but unsaved data can be resent by the source after the driver is restarted)
  - The two steps ensure that there is no zero data loss
  - All data is either recovered from the logs or resent by the source

Checkpointing and write ahead logs



Recovery from failures



- To probe further: Das, [Improved fault-tolerance and zero data loss in Apache Spark streaming](), Databricks Engineering Blog, 2015

Bayesian linear regression

- Likelihood function: $p(\boldsymbol{y} \mid \boldsymbol{X}, \boldsymbol{w}, \beta) = \prod_{i=1}^{n} N(y_i \mid \boldsymbol{x}_i^\top \boldsymbol{w}, 1/\beta)$

  where $\boldsymbol{y} = (y_1, y_2, \dots, y_n)^\top$ and $\boldsymbol{X} = (\boldsymbol{x}_1, \boldsymbol{x}_2, \dots, \boldsymbol{x}_n)^\top$

- Prior distribution: $p(\boldsymbol{w}) = N(\boldsymbol{w} \mid \boldsymbol{\mu}_0, \boldsymbol{\Sigma}_0)$

- Posterior distribution: $p(\boldsymbol{w} \mid \boldsymbol{y}) = N(\boldsymbol{w} \mid \boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1)$

$$\boldsymbol{\mu}_1 = \boldsymbol{\Sigma}_1(\boldsymbol{\Sigma}_0^{-1}\boldsymbol{\mu}_0 + \beta \boldsymbol{X}^\top \boldsymbol{y})$$

$$\boldsymbol{\Sigma}_1^{-1} = \boldsymbol{\Sigma}_0^{-1} + \beta \boldsymbol{X}^\top \boldsymbol{X}$$

To probe further: Chapter 3, Bishop 2006

Streaming k-means

- k-means objective: given a set of points $\boldsymbol{x}_1, \boldsymbol{x}_2, \dots, \boldsymbol{x}_n$ in $\mathbf{R}^d$ and integer $1 \leq k \leq n$, find a partition of points $S_1, S_2, \dots, S_k$ that minimizes

$$f(S_1, S_2, \dots, S_k) = \sum_{i=1}^{k} \sum_{x \in S_i} \|\boldsymbol{x} - \boldsymbol{\mu}_i\|^2$$

  where $\boldsymbol{\mu}_i$ is the centroid of points in S_i

$$\boldsymbol{\mu}_i = \frac{1}{|S_i|} \sum_{x \in S_i} \boldsymbol{x}$$

- Lloyd's algorithm:
  - Initialization: initialize centroids
  - Assignment step: assign each point to a nearest current centroid
  - Update step: update centroids

Streaming k-means (cont'd)

- Streaming k-means: for each batch of data, assign all points to their nearest cluster, compute new cluster centers, then update each cluster center as:

$$c_{t+1} = \frac{\alpha n_t c_t + m_t x_t}{\alpha n_t + m_t}$$
$$n_{t+1} = n_t + m_t$$

c_t is the previous center of the cluster
n_t is the number of points assigned to the cluster thus far
x_t is the new cluster center from the current batch
m_t is the number of points added to the cluster in the current batch
α is the decay factor ($\alpha = 0$ only the most recent data is used, $\alpha = 1$ all data is used)

Streaming k-means: code example

```python
from pyspark.mllib.linalg import Vectors
from pyspark.mllib.regression import LabeledPoint
from pyspark.mllib.clustering import StreamingKMeans

# we make an input stream of vectors for training,
# as well as a stream of vectors for testing
def parse(lp):
    label = float(lp[lp.find('(') + 1: lp.find(')')])
    vec = Vectors.dense(lp[lp.find('[') + 1: lp.find(']')].split(','))

    return LabeledPoint(label, vec)

trainingData = sc.textFile("data/mllib/kmeans_data.txt")\
    .map(lambda line: Vectors.dense([float(x) for x in line.strip().split(' ')]))

testingData = sc.textFile("data/mllib/streaming_kmeans_data_test.txt").map(parse)

trainingQueue = [trainingData]
testingQueue = [testingData]

trainingStream = ssc.queueStream(trainingQueue)
testingStream = ssc.queueStream(testingQueue)
```

Source: https://spark.apache.org/docs/2.2.0/mllib-clustering.html

Streaming k-means: code example (cont'd)

```
# We create a model with random clusters and specify the number of clusters to find
model = StreamingKMeans(k=2, decayFactor=1.0).setRandomCenters(3, 1.0, 0)

# Now register the streams for training and testing and start the job,
# printing the predicted cluster assignments on new data points as they arrive.
model.trainOn(trainingStream)

result = model.predictOnValues(testingStream.map(lambda lp: (lp.label, lp.features)))
result.pprint()

ssc.start()
ssc.stop(stopSparkContext=True, stopGraceFully=True)
```

Source: https://spark.apache.org/docs/2.2.0/mllib-clustering.html