Stream Processing Systems

Apache Spark and Spark Streaming

Objectives

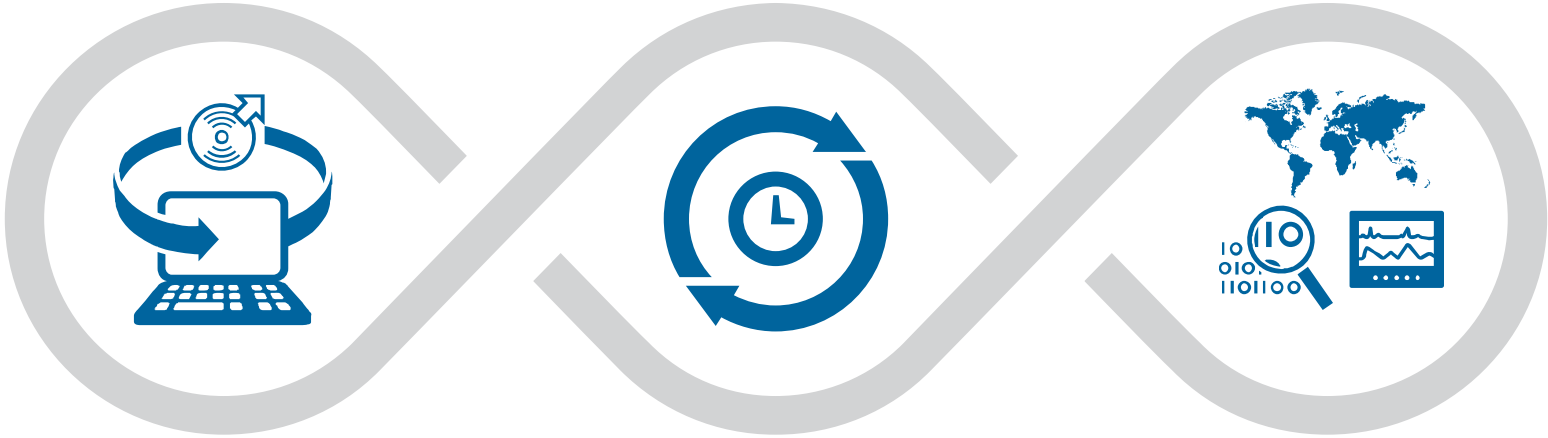
- Introduction to Streaming Applications
  - Core development concepts
- Apache Spark, SparkSQL and Spark Streaming
  - Provide enough overview to get started
- Programming Hands On
  - https://spark.apache.org/

Anatomy of a Stream Processing System

Integrated Development Environment

Scale-Out Runtime

Analytic Toolkits



Development and Management

Flexibility and Scalability

Functional and Optimized

Stream Processing Middleware Services

- Distribution
  - Plumbing related to moving processes around
- Resource management
  - Placement, scheduling, load shedding
- Job management
  - Starting/stopping/editing jobs, handling dynamic connections
- System monitoring
  - Health, state, performance of the system
- Security
- Logging

- Data Transport
  - Inter-Process, intra-Process, intra-host, inter-host
- Debugging
- Visualization

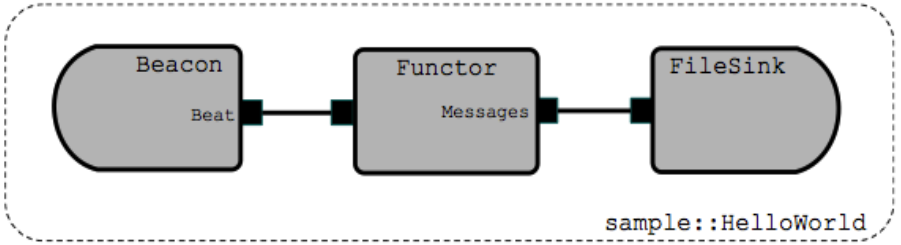
Stream Processing Applications

- Composition Language
- Defining Tuples and Streams
- Stream Sources
- Operators and Stream Transformers
- Windows and Stream Relational Processing
- Stream Sinks

Structure of a Streaming Application

SPL Hello World: A fully Declarative Model

```
1   namespace sample;
2
3   composite HelloWorld {
4     graph
5       stream<uint32 id> Beat = Beacon() {
6         param
7           period: 1.0; // seconds
8           iterations: 10u;
9         output
10          Beat: id = (uint32) (10.0*random());
11      }
12      stream<rstring greeting, uint32 id> Message = Functor(Beat) {
13        output
14          Message: greeting = "Hello World! (" + (rstring) id + ")";
15      }
16      () as Sink = FileSink(Message) {
17        param
18          file: "/dev/stdout";
19          format: txt;
20          flush: 1u;
21      }
22  }
```



Apache Spark and Spark Streaming

Apache Spark History

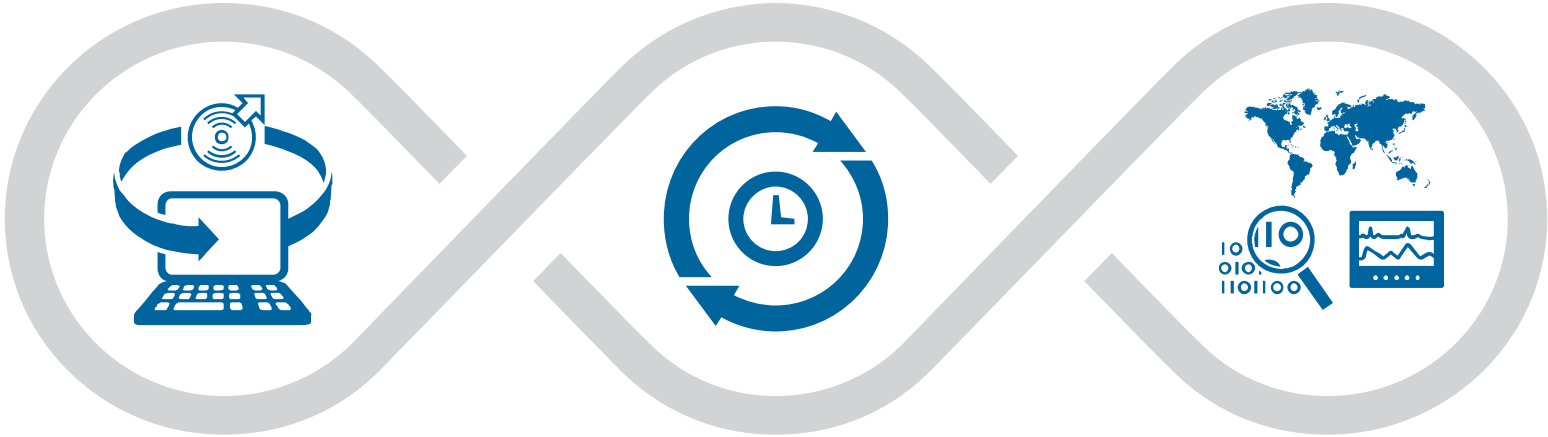
- Open-source in-memory cluster computing
- Developed at AMPLab
  - UC Berkeley (2009)
  - Open sourced in 2010
- First release: May 2014
- Overcomes several limitations of Hadoop Map-Reduce model

Apache Spark Platform

Development Environment

Scale-Out Runtime

Analytic Toolkits

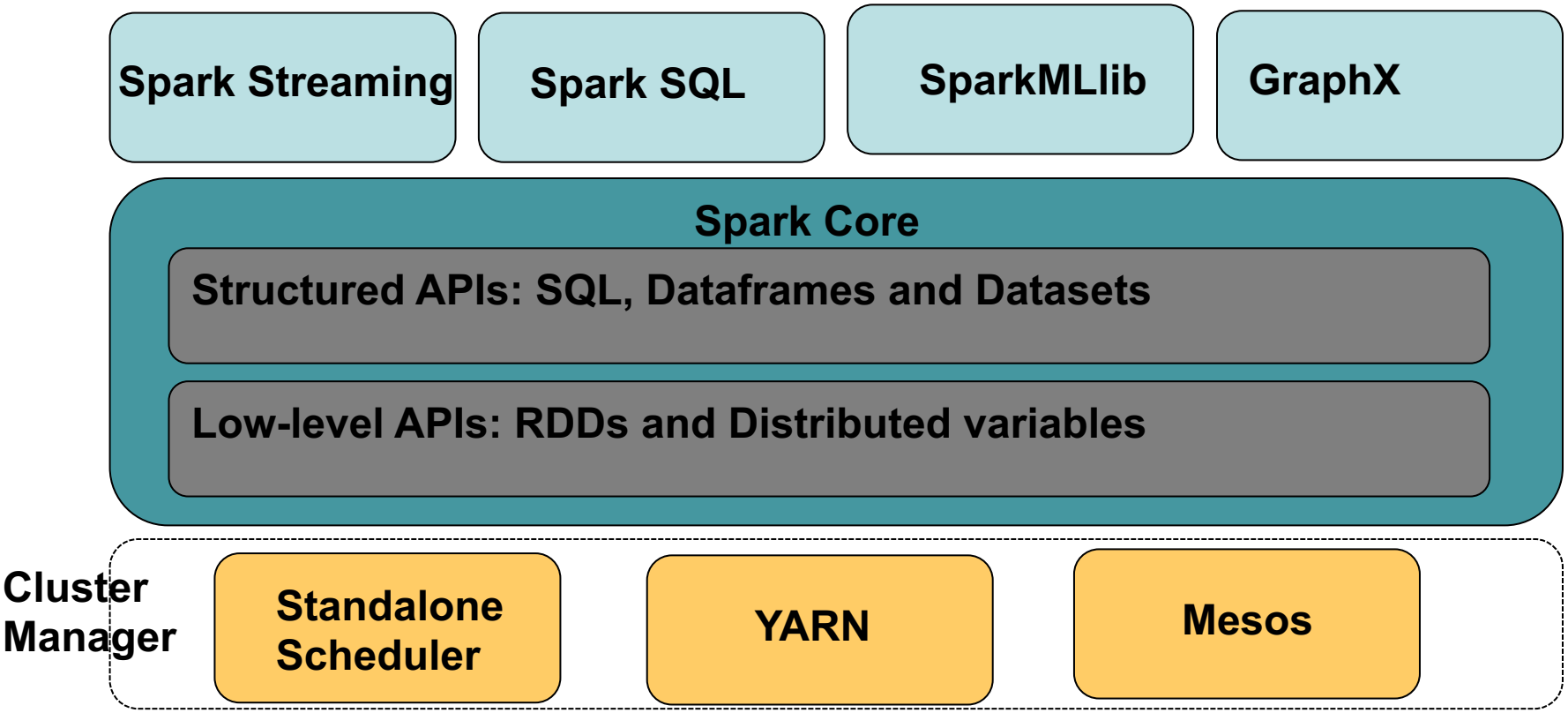


Python, Scala, Java

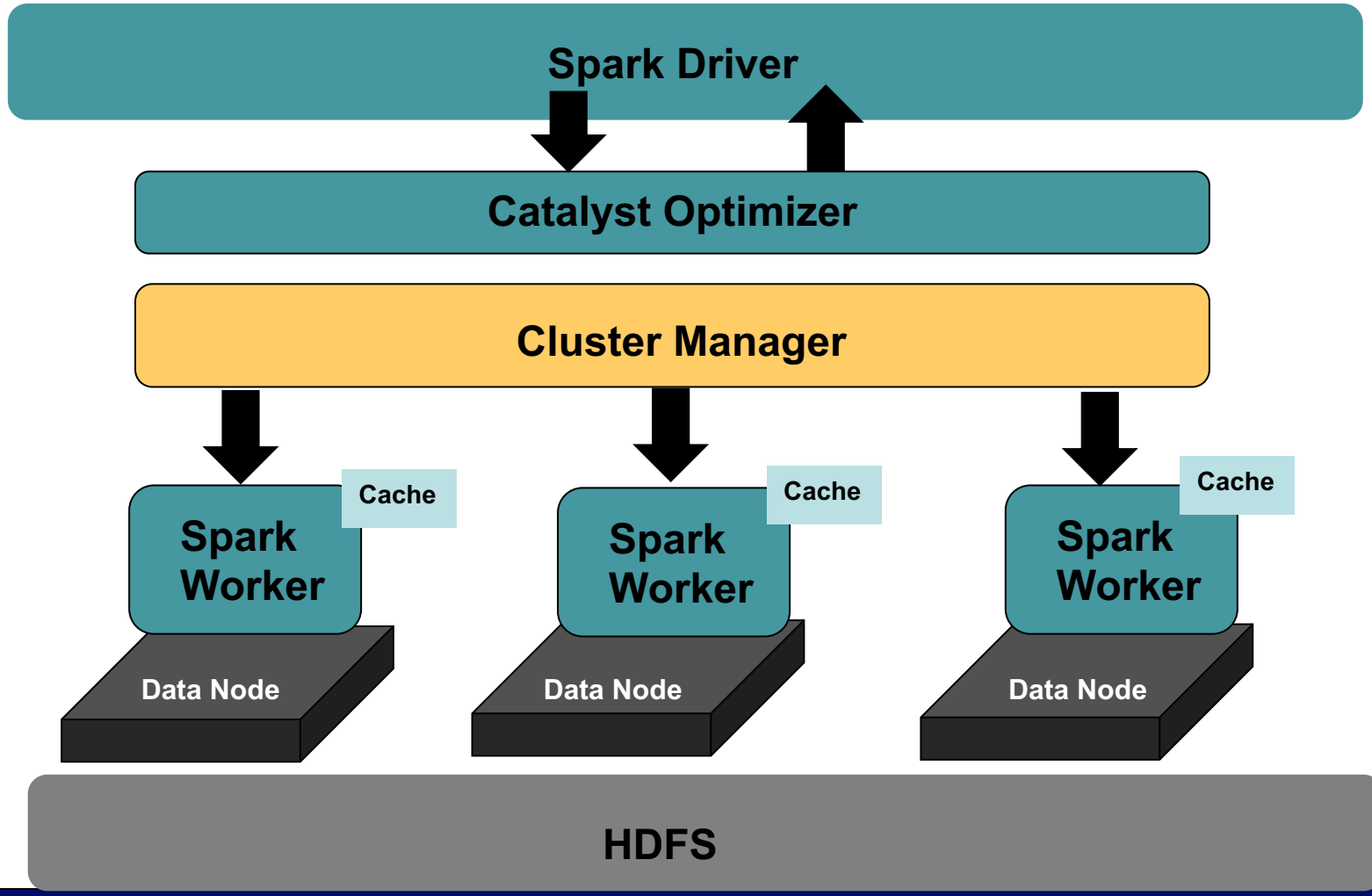
In-Memory Cluster Computing

SparkMLlib, Connectors

Spark Components



Spark Runtime Architecture



Recall: Middleware Support

- Access transparency
  - Local and remote components accessible via same operations
- Location transparency
  - Components locatable via name, independent of location
- Concurrency transparency
  - Components/objects can be run in parallel
- Migration transparency
  - Allows movement of components without affecting other components
- Replication transparency
  - Allows multiple instances of objects for improved reliability (mirrored web pages)
- Failure transparency
  - Components designed while accounting for failure of other services

Spark Basics

- Installation
  - Straightforward on Linux/Windows
  - Download binaries – unzip
    - http://spark.apache.org/downloads.html
  - `pip install pyspark`
- Includes
  - `Readme.md`
  - `bin` (binaries, including shell)
  - `core, streaming, python` (source code)
  - `examples`

Spark Shell (Python and Scala)

- Shell
  - `pyspark` (Python)
  - `spark-shell` (Scala)
- Interactive command line based interface
- Similar to those in R, Python, Matlab
- Log level can be controlled
  - `log4j.properties.template` in `conf` folder
  - Copy to `log4j.properties` and set log level
    - `log4j.rootCategory=WARN`
- Easy to get started

Resilient Distributed Datasets (RDDs)

- Resilient
  - Data can be recreated if lost
- Distributed
  - Stored in memory across the cluster
- Dataset
  - Immutable
  - Lazy Evaluated
  - Cacheable
  - Type Inferred
- Fundamental unit of processing in Spark
  - Dstreams for Streaming

Immutability

- Data cannot be changed
  - Allows improved parallelism and scalability
  - However requires more storage

```
const int a = 0; //immutable
int b = 0; //mutable
b++; //allowed in-place change
a++; //not allowed
c = a+1;//create new variable - leave old variable intact
```

  - Every transformation creates a new copy!

Lazy Evaluation

- Transformations are not computed till needed
  - Deferred evaluation
  - Separate execution from evaluation

```
C2 = C1.map(lambda val: val+1) //not evaluated
C3 = C2.map(lambda val: val+2) //not evaluated
print C3 //now evaluated as

C3 = C1.map(lambda val : ((val+1)+2))
```

  - Multiple transformations performed together when result needs to be reported
  - Laziness works only when immutable data

Type Inference

- Type of a variable/collection inferred from values by compiler
- Sometimes can be inferred from transformation applied
  - Some transformations have fixed return type

```
C1 = [1, 2, 3, 4, 5] //Type array (explicit)
C2 = C1.map(lambda val: val+1) //C2 type array
C3 = C2.count() //Type integer
```

Caching

- Immutable data can be cached over long periods
- Lazy transformations allow for data to be recreated as needed
- Transformations can also be saved
- Caching improves performance

Programming with Spark

- Hands on Hello World

```
//read file line by line to create RDD called lines
lines = sc.textFile("filename")
//return count of lines in lines RDD
lines.count()
//return the first element of lines - first line in file
lines.first()
```

- SC: Spark Context
  - Driver programs access Spark runtime through this context object
  - Represents a connection to the Spark Cluster
  - Created automatically in the shell

Programming with Spark

- Hands on Hello World

```
//read file line by line to create RDD called lines
lines = sc.textFile("filename")
//filter the RDD to extract lines with text test in them
filteredLines = lines.filter(lambda val: "test" in val)
//return count of lines in lines RDD
lines.count()
//return the first element of filteredLines
filteredLines.first()
```

- `filter`
  - Special type of transformation to create a new RDD
  - Recall immutability and lazy transformations

Programming with Spark

- `lambda` Syntax
  - Allows for definition of inline functions in Python on Spark

```
//filter the RDD to extract lines with text test in them
filteredLines = lines.filter(lambda val: "test" in val)
```

  - Defines a boolean text filter in this case – that is applied to all the data items within the RDD. Can also be defined explicitly

```
//filter the RDD to extract lines with text test in them
def findTest(x):
    return "test" in x

filteredLines = lines.filter(findTest)
```

  - Spark ships this function to all the executor nodes so that it can be run in parallel
- Equivalent => syntax for Scala

Standalone Programs with Spark

- Can invoke a program from outside the shell
  - Allows for ease of programming

```
//use spark-submit to submit custom python code
spark-submit mycode.py
```

```
from pyspark import SparkConf, SparkContext
sc = SparkContext("local", "myApp")
lines = sc.textFile("foo")
filteredLines = lines.filter(lambda x: "test" in x)
```

- In mycode.py "local" Master → single node deploy
  - Use a cluster URL that tells the driver how to connect to the cluster

Programming with Spark

- Two types of operations on RDDs
  - Transformations, Actions
- Transformations
  - Create new RDD by applying transformation to the RDD entries
- Actions
  - Compute a result based on an RDD and return to the driver program or write to output, e.g. HDFS
- Important distinction
  - Lazy evaluation model of Spark
  - Need for `persist` of RDDs: RDDs are recomputed every time an action is run on them. `persist` allows for caching of the result so that intermediate results can be reused

Programming with Spark

- RDD Creation

```
//parallelize method
lines = sc.parallelize(["My", "test", "example here"])
//read from file
lines2 = sc.textFile("filename")
```

- RDD Transformations

```
//filter
filteredLines = lines.filter(lambda x: "test" in x)
filteredLines2 = lines.filter(lambda x: "example" in x)

//union
final = filteredLines.union(filteredLines2)
```

- What happens if we do not persist? What about lazy evaluation? ORs and ANDs?

Programming with Spark

- Element wise transformations
  - `filter` and `map`
- `map`
  - Apply stateless transform to each element within RDD

```
lines = sc.parallelize([1, 4, 3])
//read from file
lines2 = lines.map(lambda x: x*x)
res = lines2.collect()
for num in res:
    print num
```

- Element wise squaring of all the input values
- What does `collect` do?
  - Brings all the results back to the driver program as an array
  - Important to know – when RDD distributed across multiple nodes

Other Element Wise Transformations

- Element-wise one to many transformation
  - `flatMap`
  - Can create more (or less) than one element in output RDD per element in input RDD

```
//parallelize method
lines = sc.parallelize(["My", "test", "example here"])
//parse the text into words using space separator
words = lines.flatMap(lambda x: x.split(" "))
```

  - Third element in `lines` gets mapped to more than one element in the `words` RDD
  - What is the difference from?

```
//parallelize method
tokenLines = lines.map(lambda x: x.split(" "))
```

Set Operations on RDDs

- `union`
  - Set-wise union of two RDDs – retains duplicates
- `distinct`
  - Remove duplicate values in RDD
  - Expensive operation as needs to aggregate across all partitions of the RDD
- `subtract`
  - Set-wise difference between RDDs
  - Also removes duplicates → expensive operation
- `intersection`
  - Set-wise common elements between RDDs
  - Also removes duplicates

Set Operations on RDDs

```
lines = sc.parallelize(["coffee", "coffee", "panda", "monkey",
"tea"])
lines2 = sc.parallelize(["coffee", "monkey", "kitty"])

res1 = lines.union(lines2)
//{"coffee", "coffee", "coffee", "panda", "monkey", "monkey", "tea",
"kitty"}

res2 = lines.distinct()
//{"coffee", "panda", "monkey", "tea"}

res3 = lines.intersection(lines2)
//{"coffee", "monkey"}

res4 = lines.subtract(lines2)
//{"panda", "tea"}
```

Set Operations on RDDs

- `cartesian` operation
  - Create cartesian product of sets

```
lines = sc.parallelize(["coffee", "coffee", "panda"])
lines2 = sc.parallelize(["coffee", "monkey"])

res1 = lines.cartesian(lines2)
//{("coffee", "coffee"), ("coffee", "monkey"), ("coffee", "coffee"),
("coffee", "monkey"), ("panda", "coffee"), ("panda", "monkey")}
```

  - Can be very expensive for large sets

Summary of Basic RDD Operations

```
lines = sc.parallelize([1, 2, 3, 3])
```

Name	Purpose	Example	Result
`map()`	Element-wise function, same number of output elements	`lines.map(lambda x: x*x)`	{1, 4, 9, 9}
`flatMap()`	Element-wise function, different number of output elements	`lines.flatMap(lambda x: [0,x])`	{0,1,0,2,0,3,0,3}
`filter()`	Select RDD elements that satisfy a condition	`lines.filter(lambda x: x>1)`	{2, 3, 3}
`distinct()`	Remove duplicates - expensive	`lines.distinct()`	{1, 2, 3}
`sample(withReplacement, fraction, [seed])`	Randomly sample from RDD	`lines.sample(false, 0.5)`	{1,3} Non-deterministic

Summary of Two-RDD Operations

```
a = sc.parallelize([1, 2, 3])        b = sc.parallelize([3, 4, 5])
```

Name	Purpose	Example	Result
`union()`	Union across all elements – duplicates retained	`a.union(b)`	{1, 2, 3, 3, 4, 5}
`subtract()`	Set-difference – duplicates removed	`a.Subtract(b)`	{1, 2}
`intersection()`	Common elements – duplicates removed	`a.intersection(b)`	{3}
`cartesian()`	Cartesian product - expensive	`a.cartesian(b)`	{(1,3), (1,4), (1,5), (2, 3), (2,4), (2,5), (3,3), 3,4), (3,5)}

Actions on RDDs: Aggregations

- Produce result of running an operation on RDD
  - not new RDD
  - Often recursive application of some function
- `reduce` action
  - Takes function that operates on two elements on RDD and returns element of same type
  - Applies function recursively across all elements in RDD

```
nums = sc.parallelize([1, 2, 3, 4])
res = nums.reduce(lambda x, y: x+y)
//returns sum of entire RDD = 10
```

  - Note that `res` is not a RDD – it is an integer value

Actions on RDDs: Aggregations

- Similar to `reduce` is the `fold` action
  - Except we can add an initial value for the result

```
nums = sc.parallelize([1, 2, 3, 4])
res = nums.reduce(lambda x, y: x+y)
//returns sum of entire RDD = 10
res2 = nums.fold((5),(lambda x,y:x+y))
//returns the sum staring with an initial value 5 = 15
```

- More complex action is `aggregate`
  - Frees the constraint on the function that return value has to be same as the element value
  - Takes initial value, function to combine elements into accumulator, and another function to combine accumulators

Actions on RDDs: Aggregations

- Example to compute sum and count at same time
  - Initialize with 0,0

```
nums = sc.parallelize([1, 2, 3, 4])
res = nums.aggregate(
      (0,0),
      (lambda acc, val: (acc[0]+val, acc[1]+1)),
      (lambda acc1, acc2: (acc1[0]+ acc2[0], acc1[1] + acc2[1])))
//returns sum and count = (10,4)
//can compute average
print res[0]/res[1]
```

  - First function sums RDD values and increments counter by 1
  - Next function combines the accumulators – potentially across multiple partitions of the RDD
- Can we do something equivalent with map and reduce functions only?

Actions on RDDs: Collection

- The `collect` action
  - Collects all values across the RDD across all partitions and workers and brings them back as a list to driver program (can be expensive for a large RDD)
  - Expects the resulting list to fit in memory
- The `take(n)` action
  - Takes n values from the RDD and returns to the driver program as a list
  - Order of return values may vary based on optimization across partitions
- The `top` action
  - Return top value based on a specified order
- The `takeSample(withReplacement, num, seed)` action
  - Return num samples using random sampling – with or without replacement

Actions on RDDs

```
lines = sc.parallelize([1, 2, 3, 3])
```

Name	Purpose	Example	Result
`collect()`	Collect all the RDD elements into a list	`lines.collect()`	{1, 2, 3, 3}
`count()`	Count elements in RDD	`lines.count()`	4
`countByValue()`	Count based on value of RDD – expensive action	`lines.countByValue`	{(1,1),(2,1),(3, 2)}
`take(num)`	Collect num elements into a list	`lines.take(2)`	{1, 2}
`top(num)`	Collect num elements based on order	`lines.top(2)`	{3,3}

Actions on RDDs

```
lines = sc.parallelize([1, 2, 3, 3])
```

Name	Purpose	Example	Result
`takeOrdered(num, (ordering))`	Collect num elements based on order function	`lines.takeOrdered(2,(myorderingfunc))`	{1, 2}
`takeSample(withReplacement, num, [seed])`	Get elements in RDD	`lines.takeSample(false, 2)`	Non-deterministic
`reduce(func)`	Recursive aggregation of RDD based on function	`lines.reduce(lambda x,y: x+y)`	9
`fold(init,func)`	Recursive aggregation of RDD based on function with init val	`lines.fold(2,(lambda x,y:x+y))`	11
`Aggregate(init,seqOp, combOp)`	Recursive aggregation that allows combination function with return type different than RDD element type	`See earlier example`	
`foreach(func)`	Apply function for each element of RDD	`lines.foreach(func)`	Depends on func

RDD Persistence

- Due to lazy evaluation
  - RDDs are processed only when action applied
  - If multiple actions applied to same RDD – it may be computed multiple times
  - Persistence allows for caching of intermediate results

```
nums = sc.textFile("foo")
res = nums.top()
res2 = nums.count()
//File read two times
```

```
nums = sc.textFile("foo")
nums.persist()
res = nums.top()
res2 = nums.count()
//File read only once
```

  - Each node stores its computed partition for reuse
  - If a node fails, the partition is recomputed
  - Old partitions evicted using a Least Recently Used (LRU) policy

Pair RDDs

- RDDs that have key,value elements
  - Common to a majority of data analysis tasks
  - Expose new operations and partitioning
  - Can be read from certain data formats
  - Created using the map function

```
lines = sc.textFile("foo")
pairlines = lines.map(lambda x: (x.split(" ")[0], x))
```

  - Taking the first word in a line as the key to the line
  - This returns tuple elements within the resulting RDD
  - Have additional transformation functions that can be applied

Pair RDD Transformations

`Rdd: {(1,2), (3,4), (3, 6)}  lines: {("a","This is"),("b", "a test")}`

Name	Purpose	Example	Result
`groupByKey()`	Group elements in RDD by key to make element lists in new RDD	`Rdd.groupByKey()`	{(1,[2]),(3,[4,6])}
`mapValues(func)`	Apply function to each value without changing the key	`Rdd.mapValues(lambda x: x*x)`	{(1,4),(3,16),(3,36)}
`flatMapValues(func)`	Apply function to each RDD element without changing the key – each function can return more than one value	`lines.flatMapValues((lambda x:x.split(" "))`	{("a","This"), ("a","is"),("b","a"),("b","test)}
`keys()`	Return an RDD with just the keys	`Rdd.keys()`	{1,3,3}

Pair RDD Transformations

`Rdd: {(1,2), (3,4), (3, 6)}  other: {(3,9)}`

Name	Purpose	Example	Result
`values()`	Return an RDD with just the values	`Rdd.values()`	{2,4,6}
`sortByKey()`	Create new RDD sorted by key	`Rdd.sortByKey()`	{(1,2),(3,4), (3,6)}
`subtractByKey()`	Set operation - remove elements with key common to second RDD . Removes duplicates	`Rdd.subtractByKey(other)`	{(1,2)}
`cogroup()`	Group elements across two RDDs with the same key. Also does a groupByKey within each RDD	`Rdd.cogroup(other)`	{(1,[2],[]), (3,[4,6],[9])}

Join: A Database Perspective

- An SQL join on tables in a relational database :

```
SELECT NAME, EQUIP
     FROM EMP-DEPT JOIN DEPT-EQUIP ON
     EMP-DEPT.DEPT = DEPT_EQUIP.DEPT
```

- *Inner* Join: Result contains only tuples that were matched

EMP-DEPT

Name	Dept.
Alice	Accounting
Bob	Programming
Carol	Sales
Dan	HR
Ellen	Programming
Frank	Accounting
George	Research

$\bowtie$

DEPT-EQP

Dept.	Equipment
Accounting	Calculator
Programming	Computer
Sales	Computer
HR	Tablet
Security	Vehicle
Sales	Phone

$=$

Name	Equipment
Alice	Calculator
Bob	Computer
Ellen	Computer
Dan	Tablet
Carol	Computer
Carol	Phone
Frank	Calculator

Join: A Database Perspective

- An SQL join on tables in a relational database :

```
SELECT NAME, EQUIP
     FROM EMP-DEPT LEFT OUTER JOIN DEPT-EQUIP ON
     EMP-DEPT.DEPT = DEPT_EQUIP.DEPT
```

- *Left Outer* Join: Result contains tuples that were matched, and unmatched tuples from the “left” table

EMP-DEPT

Name	Dept.
Alice	Accounting
Bob	Programming
Carol	Sales
Dan	HR
Ellen	Programming
Frank	Accounting
George	Research

$\bowtie$

DEPT-EQP

Dept.	Equipment
Accounting	Calculator
Programming	Computer
Sales	Computer
HR	Tablet
Security	Vehicle
Sales	Phone

=

Name	Equipment
Alice	Calculator
Bob	Computer
Ellen	Computer
Dan	Tablet
Carol	Computer
Carol	Phone
Frank	Calculator
George	NULL

Join: A Database Perspective

- An SQL join on tables in a relational database :

```
SELECT NAME, EQUIP
     FROM EMP-DEPT RIGHT OUTER JOIN DEPT-EQUIP ON
     EMP-DEPT.DEPT = DEPT_EQUIP.DEPT
```

- *Right Outer* Join: Result contains tuples that were matched, and unmatched tuples from the “right” table

EMP-DEPT

Name	Dept.
Alice	Accounting
Bob	Programming
Carol	Sales
Dan	HR
Ellen	Programming
Frank	Accounting
George	Research

⋈

DEPT-EQP

Dept.	Equipment
Accounting	Calculator
Programming	Computer
Sales	Computer
HR	Tablet
Security	Vehicle
Sales	Phone

=

Name	Equipment
Alice	Calculator
Bob	Computer
Ellen	Computer
Dan	Tablet
Carol	Computer
Carol	Phone
Frank	Calculator
NULL	Vehicle

Aside: Additional Joins in Spark

- Semi-joins: Left and Right

  Only return rows from the left (or right) table that match, and no duplicates

EMP-DEPT

Name	Dept.
Alice	Accounting
Bob	Programming
Carol	Sales
Dan	HR
Ellen	Programming
Frank	Accounting
George	Research

⋈

DEPT-EQP

Dept.	Equipment
Accounting	Calculator
Programming	Computer
Sales	Computer
HR	Tablet
Security	Vehicle
Sales	Phone

=

Name	Equipment
Alice	Accounting
Bob	Programming
Ellen	Programming
Dan	HR
Carol	Sales
Frank	Accounting

- Anti-joins: Apply a not to a Semi-join
  - Left and Right

Joins on Pair RDDs

- Joins operate like database counterparts
  - Inner, LeftOuter and RightOuter
  - Equality conditions are defined based on keys

```
Rdd: {(1,2), (3,4), (3, 6)}  other: {(3,9)}

Rdd.join(other): {(3,(4,9)),(3,(6,9))}
Rdd.leftOuterJoin(other): {((1,(2,None)), (3,(4, 9)),
                    (3,(6,Some(9)))}
Rdd.rightOuterJoin(other): {(3,(4,9)),(3,(6,9))}
```

  - Is Join a symmetric operation?
  - How about RightOuterJoin and LeftOuterJoin?

  - How about non-equi joins?

Aggregations on Pair RDDs

- `reduce`, `fold` and `aggregate` actions extended to work by key
  - `reduceByKey`, `foldByKey`, `aggregateByKey`
  - These operate as transformations rather than as actions, i.e. return RDDs, since each RDD may have multiple keys

```
Rdd: {(1,2), (3,4), (3, 6)}  other: {(3,9)}

Res = Rdd.reduceByKey(lambda x,y:x+y)
//Res = {(1,2),(3,10)}

Res2 = Rdd.foldByKey(3,lambda x,y:x+y))
//Res2 = {(1,5),(3,13)}

Res3 = Rdd.mapValues(lambda x: (x,1)).reduceByKey(lambda x,y:
(x[0]+y[0],x[1]+y[1])
```

  - What does the above do?

WordCount Example on Spark

- WordCount
  - Count occurrences of each word within a file

```
lines = sc.textFile("foo")
words = lines.flatMap(lambda x: x.split(" "))
wordMap = words.map(lambda x: (x,1))
wordCount = wordMap.reduceByKey(lambda x,y: x+y)

wordCount2 = words.countByKey()

wordCount3 = words.count()
```

  - What is the difference in these different results?

Partitioning and Parallelism

- How does Spark parallelize the processing?
  - Every RDD has a fixed number of partitions
  - When grouping or aggregating RDDs, we can specify degree of partitions
    - Especially true and useful for pair RDDs

```
Data = sc.parallelize([("a",1),("b",2),("c",3)])
Res = Data.reduceByKey(lambda x,y: x+y)
//default parallelism
Res = Data.reduceByKey(lambda x,y: x+y, 10)
//Custom parallelism use 10 partitions
```

  - Sometimes data may need to be repartitioned to improve efficiency of the operation – for groupings

Partitioning and Parallelism

- Optimize partitioning to minimize communication across nodes
  - Especially useful when a dataset is used multiple times
- Spark does not allow explicit control on which worker node a key goes to
  - Can specify which sets of keys go together
  - E.g. hash partition by key mod 100 → all keys with the same remainder after hashing to 100 will be on same node
  - Range partition based on key values – so that keys in same range appear on same node

```
Data = sc.parallelize([("a",1),("b",2),("c",3)])
Res2 = Data.partitionBy(new HashPartitioner(100)).persist()
```

  - Scala example
  - Custom partitioners can be written in Scala and Java

Other Actions on Pair RDDs

`Rdd: {(1,2), (3,4), (3, 6)}  other: {(3,9)}`

Name	Purpose	Example	Result
`collectAsMap()`	Collect all results as a map and return to driver program	`Rdd.collectAsMap()`	Map object
`countByKey()`	Count number of elements for each key	`Rdd.countByKey()`	{(1,1),(3,2)}
`lookup()`	Return values that have a matching key as an array	`Rdd.lookup(3)`	[4,6]

Next

- Spark Structured APIs and Dataframes

Structured Spark: Dataframes

- Part of the structured API to handle and process data
  - Equivalent to a table with named and typed columns (i.e. with a schema)
  - Just that the table is distributed over multiple workers
- Analogous to Pandas Dataframes

```
linesdf =
spark.read.option("inferSchema","true").option("header","true"
).csv("data/structured_data.csv")

linesdf.first()
```

- Same notions of lazy execution and actions (like for RDD) carry over

Transformations/Actions on Dataframes

- What do you think this does?

```
from pyspark.sql.functions import desc

linesdf.groupBy("Line").count().withColumnRenamed("count","
n_products").sort(desc("n_products")).show(5)
```

Transformations on Dataframes

```
df1 = src, dest, flights
      NY   , CA , 100
      CA   , CA , 25
      CA   , NY , 100
```

Name	Purpose	Example	Result
`select()`	Select columns in dataframe	`df1.select("src")`	`{NY, CA, CA}`

`selectExpr()`	Select and apply transforms and aggregations (can create new cols or dfs)	`df1.selectExpr("*", "(src = dest) as inState")`	`NY, CA, 100, False`
			`CA, CA, 25, True`
			`CA, NY, 100, False`

`withColumn()`	Explicitly create new columns – can use literals (as constants)	`df1.withColumn("const",lit(50))`	**`src, dest, flights, const`**
			`NY, CA, 100, 50`
			`CA, CA, 25, 50`
			`CA, NY, 100, 50`

Transformations on Dataframes

df1 = src, dest, flights

NY , CA , 100
CA , CA , 25
CA , NY , 100

Name	Purpose	Example	Result
drop()	Drop columns	df1.drop("src")	**dest, flights** CA, 25 NY, 100 CA, 100
where()	Filter rows	df1.where("flights<50")	CA, CA, 25
distinct()	Unique rows	df1.select ("src").distinct()	NY CA
sort()/orderBy()	Sort based on some columns	df1.orderBy("src","flights")	**src, dest, flights** CA, CA, 25 CA, NY, 100 NY, CA, 100
union()	Append dataframe rows with the same schema	df1.union(df1)	

Transformations on Dataframes

```
df1 = src, dest, flights
      NY   , CA , 100
      CA   , CA , 25
      CA   , NY , 100
```

```
def manyFlights(x):
    return x<50
```

Name	Purpose	Example	Result
`udf()`	Register a user defined function	`manyFlightsUdf = udf(manyFlights)`	

	Can use this UDF with `select` or `selectExpr`	`df1.select (manyFlightsUdf(col("flights")))`	**`manyFlights(flights)`**
			`False`
			`True`
			`False`

Two Dataframe Transforms

```
df1 = src, dest, flights
      NY   , CA , 100
      CA   , CA , 25
      CA   , NY , 100
```

```
df2 = state, pop
      NY   , 100
      CA   , 200
```

Name	Purpose	Example	Result

`join()`	Applies join across dataframes	`je = (df1.col("src") == df2.col("state"))`
		
		`df1.join(df2, je)`
			src, dest, flights, state, pop
			CA, CA, 25, CA, 200
			CA, NY, 100, CA, 200
			NY, CA, 100, NY, 100

	Support different types of joins		

Actions on Dataframes

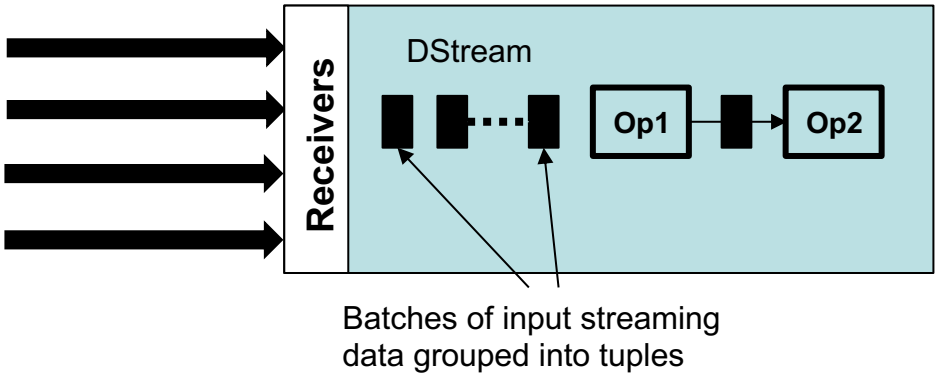
Name	Purpose	Example	Result
`count()`	Count the number of occurrences	`df.select(count("src"))`	3
`countDistinct()`	Count number of distinct occurrences	`df.select(countDistinct("src"))`	2
`avg(), sum(), min(), max()`	Apply aggregates to column	`df.select(max("flights")`	100
`approx_count_distinct()`	Get approximate distinct counts		
`show()`	Show a few rows of the dataframe	`df1.show()`	
`collect()`	Collect and bring to local memory – show metadata and content	`df1.collect()`	
`take(n)`	Take n rows from dataframe – show metadata and content, but flatten	`df1.take(5)`	

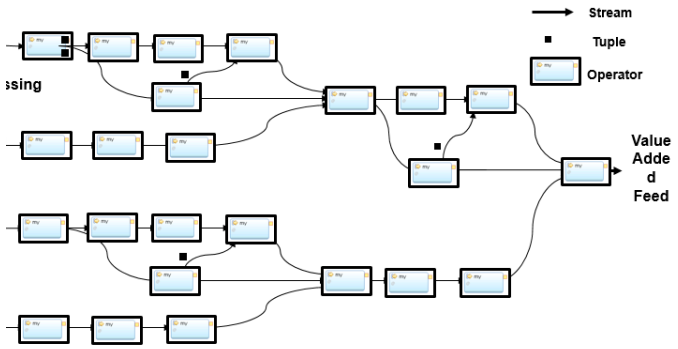
Next

- Spark Streaming

Spark Streaming

- Introduce concept `Dstream` (discretized streams)
  - Sequence of RDDs arriving over time
  - Dstreams can be created from different data sources
    - Network interfaces, Flame, Kafka, Flume, File System etc.
  - Support transformations and actions (output operations)





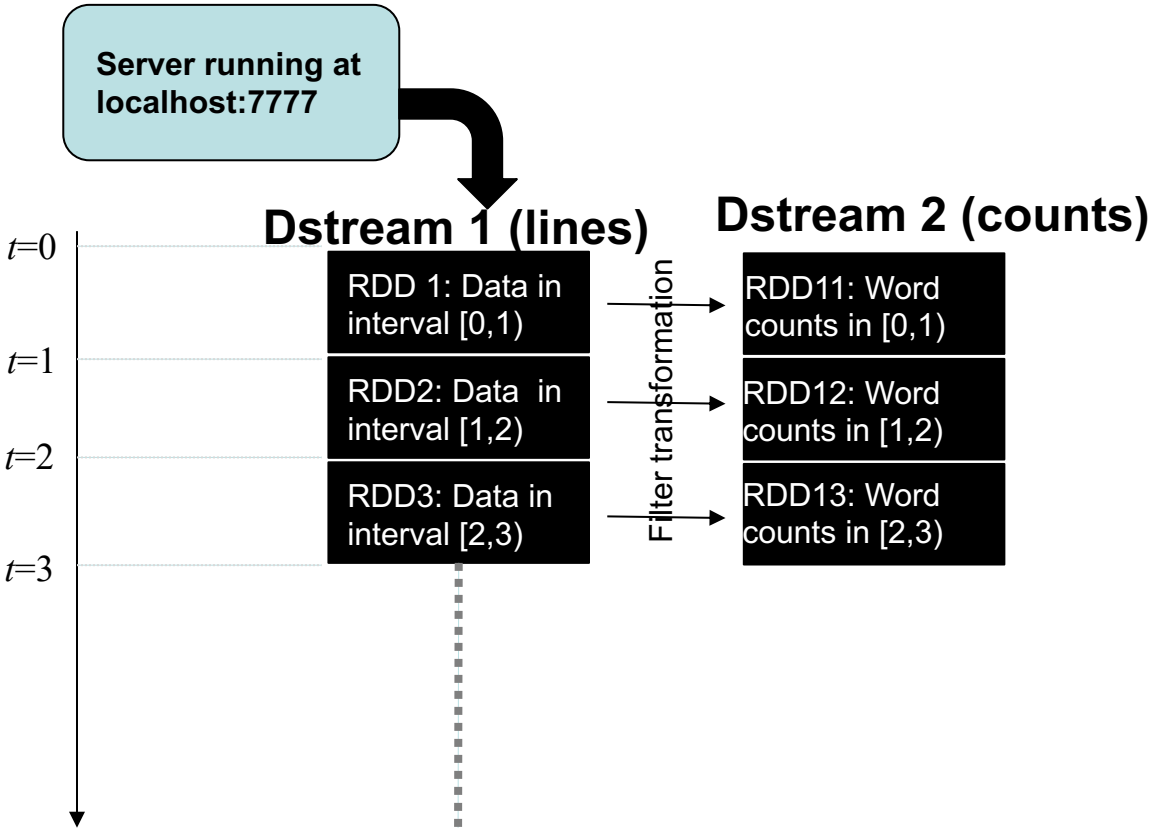
Streaming Example

- Example of constructing Dstream from network socket
  - Setup a Dstream

```
sc = SparkContext(appName="StreamingNetworkWordCount")
ssc = StreamingContext(sc, 1)
lines = ssc.socketTextStream(IP, Port)
counts = lines.flatMap(lambda line: line.split(" "))\
              .map(lambda word: (word, 1))\
              .reduceByKey(lambda a, b: a+b)
counts.pprint()
ssc.start()
ssc.awaitTermination
```

  - Take all data received in one second
  - Break it into words and then do a wordcount
  - Print all results to screen

Streaming Example



The operation of this word count is "stateless" – each window handled independently

Stateless Streaming Transformations

Name	Purpose	Example
`map()`	Apply function defined inline to each tuple (RDD) in the stream	`ds.map(lambda x: x+1)`
`flatMap()`	Apply inline function to each tuple in stream and create tuple with more than one element per input element	`ds.flatMap(lambda x: x.split(" "))`
`filter()`	Filter elements in input tuple to create new tuple	`ds.filter(lambda x: x.contains("error"))`
`reduceByKey()`	Perform reduce by key within tuple to create new tuple	`ds.reduceByKey( lambda (x,y): x+y)`
`groupByKey()`	Group values by key within each tuple	`ds.groupByKey()`
`transform()`	Apply any RDD to RDD function on each tuple	`ds.transform(lambda rdd: myFunc(rdd))`

Stateless Two-Stream Transformations

- Stateless Join on two Streams
  - Perform a standard RDD join on two tuples, one from each stream
  - Fast stream needs to wait for slow stream

```
//Create a streaming context with a 1 second batch
sc = SparkContext(appName="StreamingStatelessJoin")
ssc = StreamingContext(sc, 1)
ipDstream = accessLogs.map(lambda x: (x.getIpAddress(),1))
ipCountDstream = ipDstream.reduceByKey(lambda (x,y): x+y)

ipByteDstream = accessLogs.map(lambda x: (x.getIpAddress(),x.getContentSize())
ipByteCountDstream = ipByteDstream.reduceByKey(lambda (x,y): (x+y))

ipBCDstream = ipByteCountDstream.join(ipCountDstream)
ssc.start()
ssc.awaitTermination()
```

This can cause memory issues if one stream is much slower than the other

Also – limiting in terms of match conditions, given data arrival order etc,

Stateful Transformation on Streams

- Create and maintain state as tuples processed
- Such state, along with internal algorithm, affects the results
- e.g. `DeDuplicate`
  - *tuple is considered a duplicate if it shares the same key with a previously seen tuple within a pre-defined period of time*
- Runtime support is challenging
  - Require synchronization in a multi-threaded context
  - No trivial way to parallelize
  - Require some persistence mechanism for fault-tolerance

Windowing

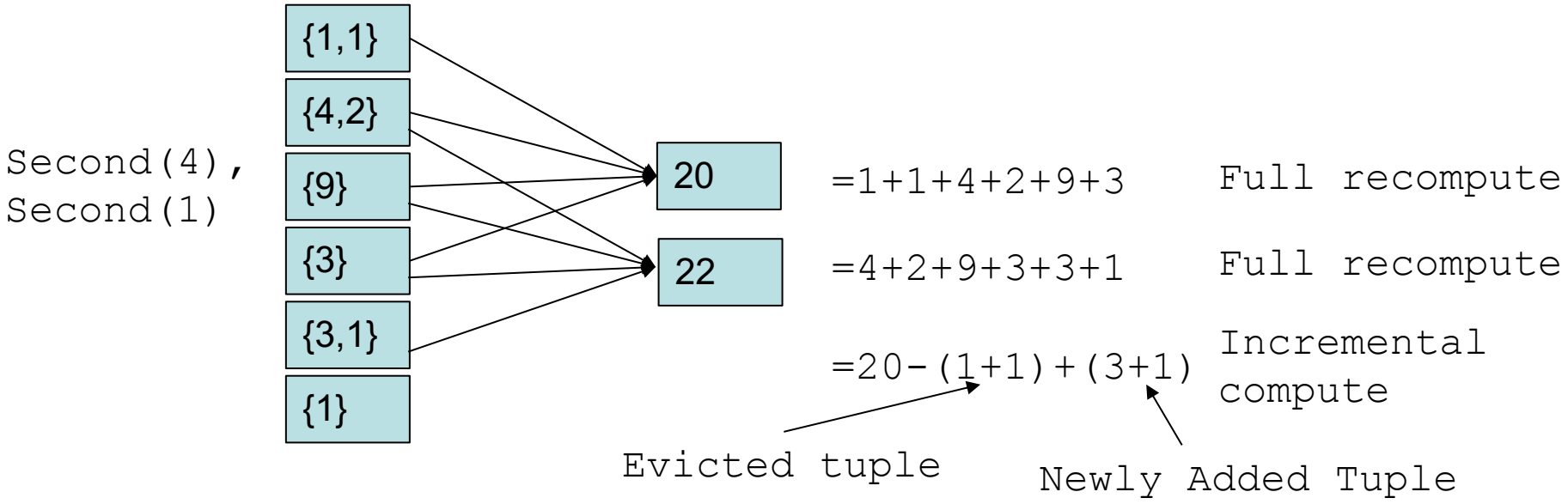
- Window creates a collection of tuples/RDDs within window
- Has size, slide – corresponds to eviction, trigger policies

```
//Create a streaming window
winLogs = accessLogs.window(30, 10)
winCounts = winLogs.countByKey()
```

- Slide cannot be smaller than batch size of Dstream
- Performs a Union across RDDs in the window
- Can then apply a reduce across this to compute additional results
  - More efficient aggregations available

Windowed Aggregations

- Incremental operations for aggregations
  - `reduceByWindow`, `reduceByKeyAndWindow`
  - Instead of re-computing fully, remove effect of evicted tuples and add in effect of newly added tuples
  - Require easily reversible functions, e.g. sum



Windowed Aggregations

```
//Create a streaming reduce with a 1 second batch, 4 second window
ipDstream = accessLogs.map(lambda x: (x.getIpAddress(),1))
ipCountDstream = ipDstream.reduceByKeyAndWindow(
                  lambda (x,y): x+y,
                  4,
                  1)
ipCountEDstream = ipDstream.reduceByKeyAndWindow(
                  lambda (x,y): x+y,
                  lambda (x,y): x-y,
                  4,
                  1)
```

Reversible function specification followed by window specification allows incremental computation, as opposed to full recompute

Other functions include `reduceByWindow()`, `countByWindow()` and `countByKeyAndWindow()`

Spark Streaming: Inputs and Outputs

- `print()` – grabs first 10 elements from each tuple and prints them to screen
- `saveAsTextFiles("output","txt")` – writes multiple files into a directory, one per tuple in txt format

- Stream of files
  - `val logData = ssc.textFileStream("logDirectory")`
  - Each file read as a tuple in the Dstream
- Other sources
  - Twitter
  - Kafka, Kinesis, Flume, and others

Limitations of Spark Streaming Model

- All windows defined in wall-clock time
  - Sizes and slides limited to multiples of batch size
- No data-time windows
  - Skewed by randomness in data arrival
  - Cannot handle out of order data
- Limited join and aggregation conditions

Next

- Structured Spark Streaming

References

- H. Karau, A. Konwinski, P. Wendell and M. Zaharia, “Learning Spark”, O’Reilly Press
- http://spark.apache.org/
- Chapter 4 in Stream Processing Book