Handling of Failures - Network Partition

- If the coordinator and all its participants remain in one partition, the failure has no effect on the commit protocol
- If the coordinator and its participants belong to several partitions:
  - Servers that are not in the partition containing the coordinator think the coordinator has failed, and execute the protocol to deal with failure of the coordinator.
    - No harm done, but servers may still have to wait for decision from coordinator
- The coordinator and the servers that are in the same partition as the coordinator think that the sites in the other partition have failed, and follow the usual commit protocol
    - Again, no harm done

Recovery and Concurrency Control

- **In-doubt transactions** have a <**ready** *T*>, but neither a <**commit** *T*>, nor an <**abort** *T*> log record.
- The recovering site must determine the commit-or-abort status of such transactions by contacting other servers; this can be slow and potentially block recovery
- Recovery algorithms can note lock information in the log
  - Instead of <**ready** *T*>, write out <**ready** *T*, *L*>
    - *L* = list of locks held by *T* when the log is written (shared locks can be omitted)
  - For every in-doubt transaction *T*, all the locks noted in the <**ready** *T*, *L*> log record are reacquired
  - After lock reacquisition, transaction processing can resume; the commit or rollback of in-doubt transactions is performed concurrently with the execution of new transactions
  - But any transaction that is "waiting for" these locks has to keep waiting until the stuck transaction is committed/aborted

end of material for mid term

Avoiding Blocking During Consensus

- Blocking problem of 2PC is a serious concern
  - **Any** participant that fails or is delayed will block all other nodes!
- Idea: involve multiple nodes in decision process, so failure of a few nodes does not cause blocking as long as majority don't fail
- More general form: **distributed consensus problem**
  - A set of *n* nodes need to agree on a decision
  - Inputs to make the decision are provided to all the nodes, and then each node votes on the decision
  - The decision should be made in such a way that all nodes will "learn" the same value even if some nodes fail during the execution of the protocol, or there are network partitions.
  - Further, the distributed consensus protocol should not block, as long as a **majority** of the nodes participating remain alive and can communicate with each other
- Several consensus protocols, Paxos and Raft are popular
- Consensus is also used to ensure consistency of replicas of a data item
  - For example, can be used to protect against failure of master nodes (e.g., the NameNode in HDFS)

Using Consensus to Avoid Blocking

- After getting response from 2PC participants, coordinator can initiate distributed consensus protocol by sending its decision to a set of participants who then use consensus protocol to commit the decision
  - If coordinator fails before informing all consensus participants
    - Choose a new coordinator, which follows 2PC protocol for failed coordinator
    - If a commit/abort decision was made as long as a majority of consensus participants are accessible, decision can be found without blocking
  - If consensus process fails (e.g., split vote), restart the consensus
    - Split vote can happen if a coordinator send decision to some participants and then fails, and new coordinator send a different decision
- The **three phase commit** protocol is an extension of 2PC which avoids blocking under certain assumptions
  - Ideas are similar to distributed consensus

Summary of 2PC

- Each server can be both a coordinator and a transaction manager
- Coordinators first prepare transaction, if all transaction managers respond they are ready, then coordinator can commit
- If coordinator fails immediately after all managers respond they are ready but before coordinator committed, we have to wait for it to recover to decide what to do
- This blocking problem can be solved using a consensus protocol, which replicates the state of each coordinator across multiple nodes, so if one fails we can still make progress without waiting for it to recover

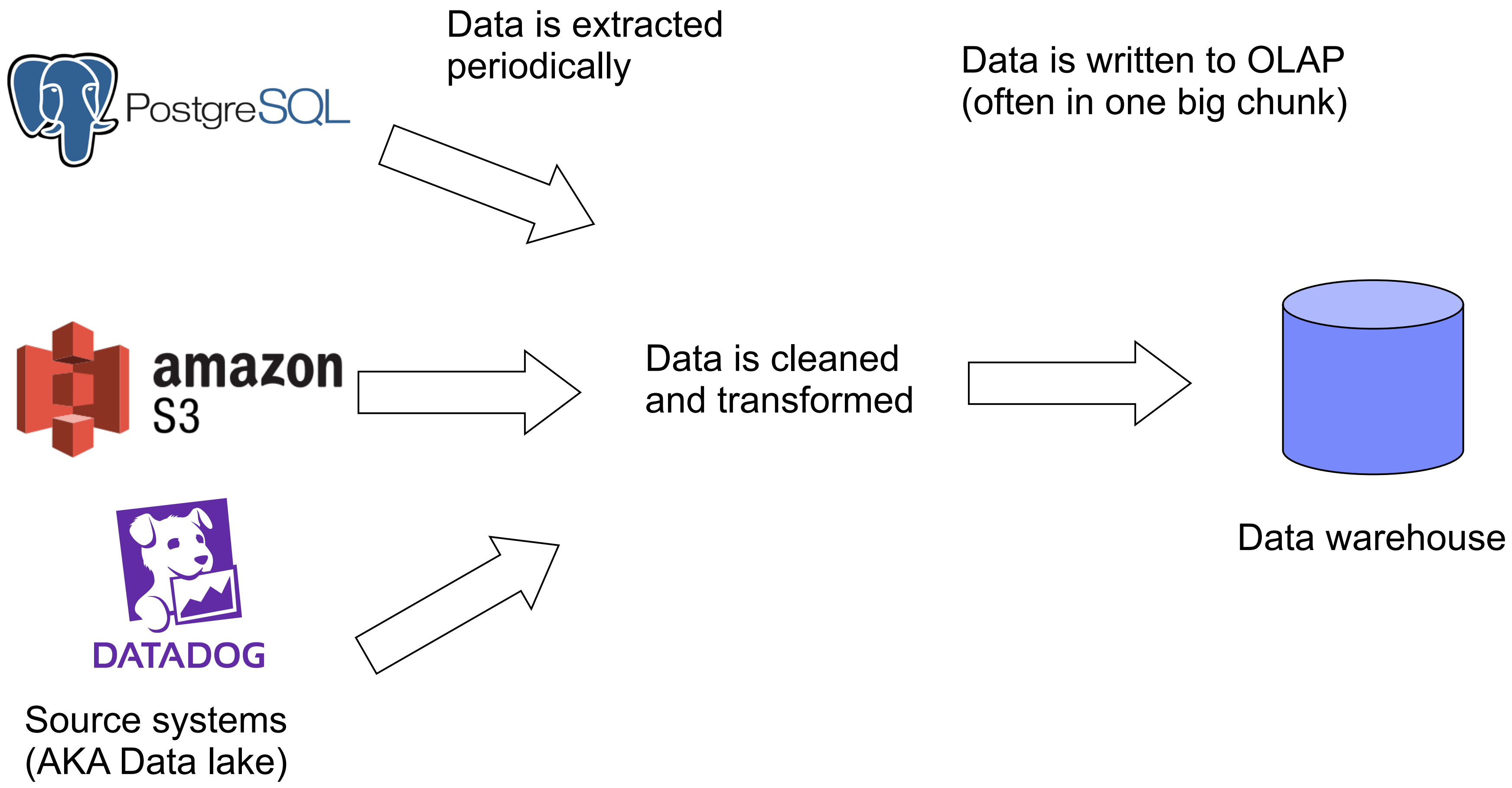
Today: OLAP

- So far in the class we've focused on OLTP (transactional), today we focus on OLAP (analytics)
- OLAP is "simpler": typically data is read-only, so isolation is much less of a problem
  - Although streaming breaks this
- Main challenges in OLAP: how to allow users to manipulate very large datasets, deal with scale and failures
- OLAP programming models
  - MapReduce: mappers and reducers
  - Spark: RDDs
  - Streaming: hybrid between OLAP and OLTP

OLAP

- Reminder: OLAP are **analytical** databases
  - Also called data warehouses
  - Typically read-only, update only in batch (no in-place updates or specific rows)
  - Read from many sources of data
  - Queries typically take a relatively long time
- Example: BigQuery!
- Most typical query language: today it’s SQL, but we’ll discuss other more limited query languages (MapReduce, Python-like PySpark) that are used for the ETL process (more on that in the next sldie)

ETL (Extract Transform and Load): Updating OLAP Systems

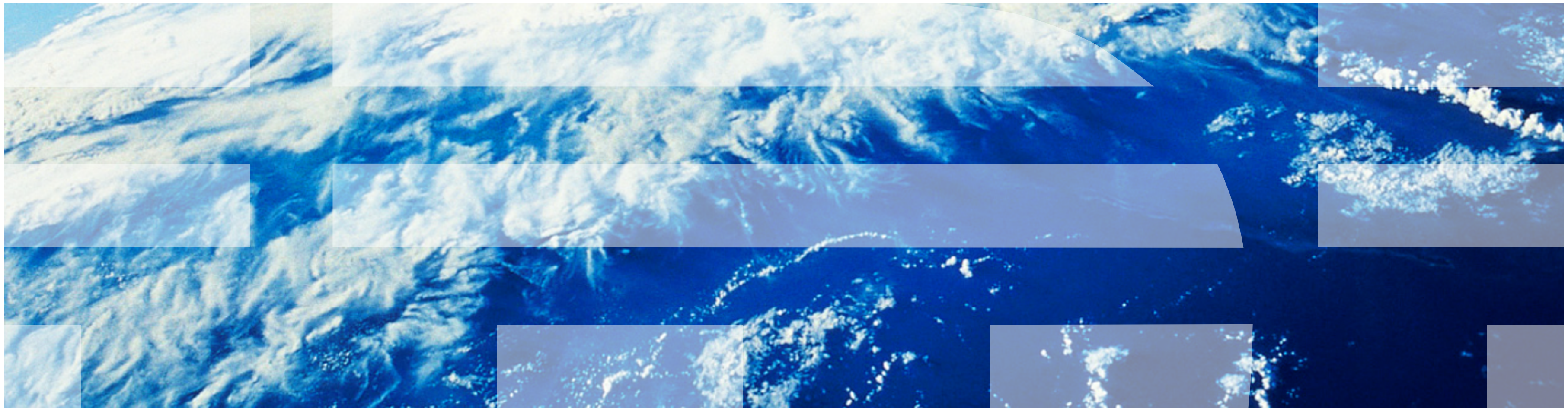

Data is extracted periodically
Data is written to OLAP (often in one big chunk)
PostgreSQL
amazon S3
DATADOG
Data is cleaned and transformed
Data warehouse
Source systems (AKA Data lake)

Let’s focus on cleaning / transformation stage

- How do take source data and transform it at scale?
  - Python?
- We will talk about two such programming frameworks
  - MapReduce
  - Spark RDDs

Borrowed from Jeff Ullman, Cristiana Amza and Indranil Gupta

MapReduce



MapReduce and Hadoop

- SQL and ACID are a very useful set of abstractions
- But: unnecessarily heavy for many tasks, hard to scale, especially for large-scale data transformation
- MapReduce is a more limited style of programming designed for:
  1. Easy parallel programming
  2. Invisible management of hardware and software failures
  3. Easy management of very-large-scale data
- It has several implementations, including Hadoop, Flink, and the original Google implementation just called “MapReduce.
- It is also used in Spark

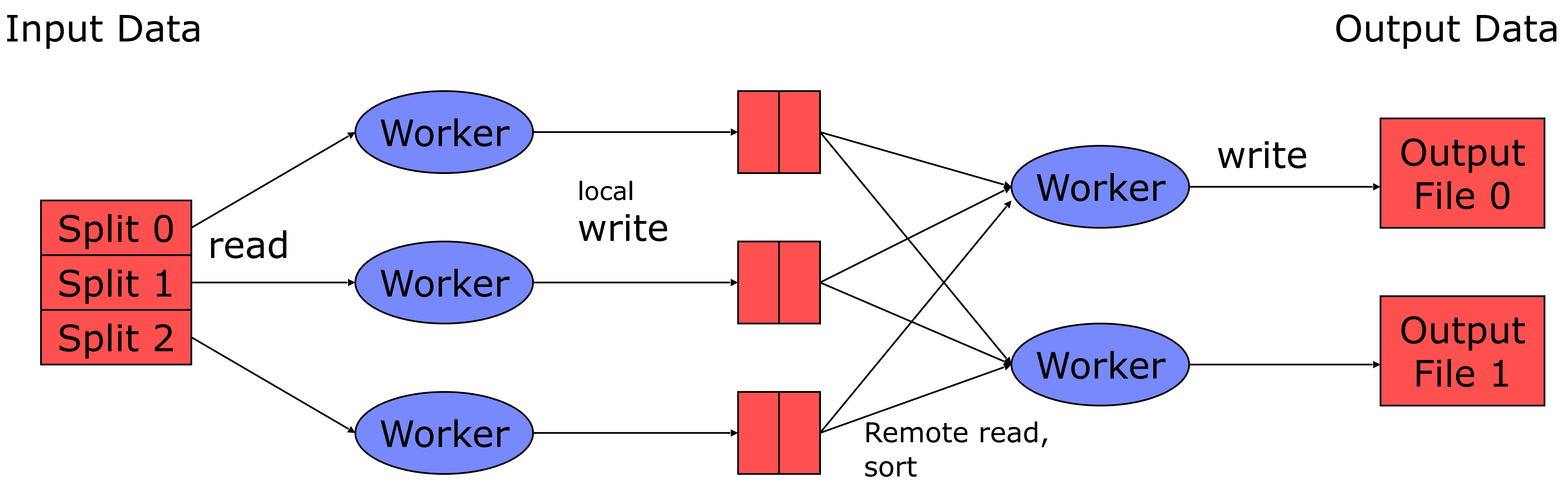
MapReduce in a Nutshell

- A MapReduce job starts with a collection of input elements of a single type.
  - Technically, all types are key-value pairs (key is the “name” of the type)
- Apply a user-written **Map function** to each input element, in parallel.
  - **Mapper** applies the Map function to a single element.
    - Many mappers grouped in a **Map task** (the unit of parallelism).
    - Usually a single Map task is run on a single node/server
- The output of the Map function is a set of 0, 1, or more *key-value pairs*.
- The system sorts all the key-value pairs by key, forming key-(list of values) pairs.

In a Nutshell – (2)

- Another user-written function, the **Reduce function**, is applied to each key-(list of values).
  - Application of the Reduce function to one key and its list of values is a **reducer**.
    - Often, many reducers are grouped into a **Reduce task**.
- Each reducer produces some output, and the output of the entire job is the union of what is produced by each reducer.

MapReduce workflow



Map
extract something you care about from each record

Reduce
aggregate, summarize, filter, or transform

Example: Word Count

- We have a large file of documents (the input elements).
- Documents are words separated by whitespace.
- Count the number of times each distinct word appears in the file.

Word Count Using MapReduce

```
map(key, value):
// key: document ID; value: text of document
  FOR (each word w IN value)
          emit(w, 1);

 reduce(key, value-list):
 // key: a word; value-list: a list of integers
           result = 0;
           FOR (each integer v on value-list)
                     result += v;
           emit(key, result);
```

Expect to be all 1's, but "combiners" allow local summing of integers with the same key before passing to reducers.

Mapper

- Reads in input pair <Key,Value>
- Outputs a pair <K', V'>
  - Let's count number of each word in user queries (or Tweets/Blogs)
  - The input to the mapper will be <queryID, QueryText>:

```
<Q1,"The teacher went to the store. The store was closed; the
store opens in the morning. The store opens at 9am." >
```

  - The output would be:

```
<The, 1> <teacher, 1> <went, 1> <to,
1> <the, 1> <store,1> <the, 1> <store,
1> <was, 1> <closed, 1> <the, 1>
<store,1> <opens, 1> <in, 1> <the, 1>
<morning, 1> <the 1> <store, 1>
<opens, 1> <at, 1> <9am, 1>
```

Reducer

- Accepts the Mapper output, and aggregates values on the key
  - For our example, the reducer input would be:

    <The, 1> <teacher, 1> <went, 1> <to, 1> <the, 1> <**store**, 1> <the, 1> <store, 1> <was, 1> <closed, 1> <the, 1> <**store**, 1> <opens,1> <in, 1> <the, 1> <morning, 1> <the 1> <**store**, 1> <opens, 1> <at, 1> <9am, 1>
  - The output would be:

    <The, 6> <teacher, 1> <went, 1> <to, 1> **<store, 3>** <was, 1> <closed, 1> <opens, 1> <morning, 1> <at, 1> <9am, 1>

Another example: Chaining MapReduce

Count of URL access frequency

- Input: Log of accessed URLs, e.g., from proxy server
- Output: For each URL, percent of total accesses for that URL

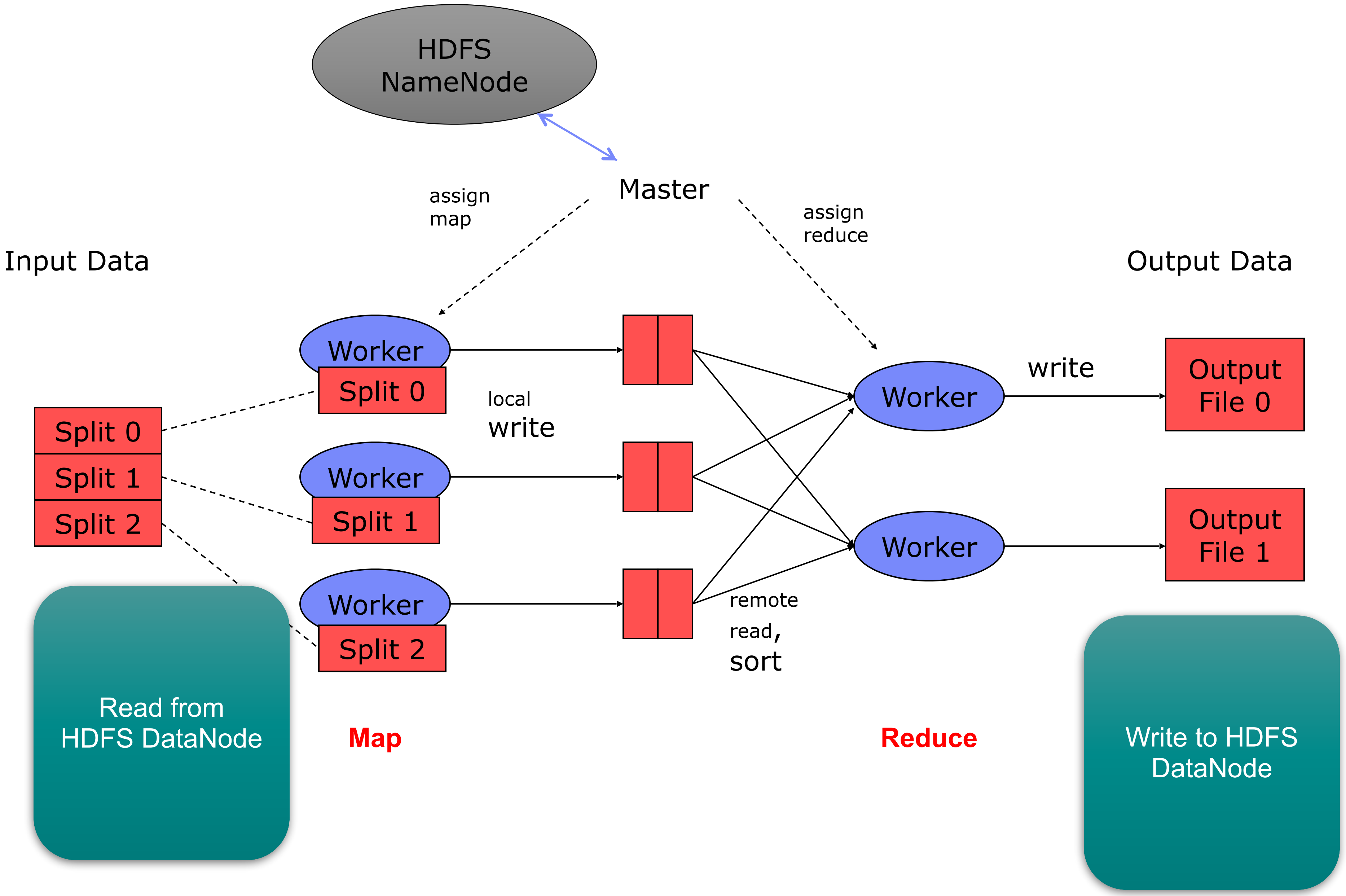
- First step:
  - Map – process web log and outputs <URL, 1>
  - Multiple Reducers - Emits <URL, URL_count>

    (So far, like Wordcount. But still need %)

- Chain another MapReduce job after above one
  - Map – processes <URL, URL_count> and outputs <1, (<URL, URL_count>)>
  - 1 Reducer – Does two passes. In first pass, sums up all URL_count's to calculate overall_count. In second pass calculates %'s

    Emits multiple <URL, URL_count/overall_count>

MapReduce


HDFS NameNode
Master
assign map
assign reduce
Input Data
Output Data
Split 0
Split 1
Split 2
Worker
Split 0
Worker
Split 1
Worker
Split 2
local write
Worker
Worker
write
Output File 0
Output File 1
remote read, sort
Read from HDFS DataNode
Map
Reduce
Write to HDFS DataNode

Locality Optimization

- **Master scheduling policy:**
  - Asks HDFS for locations of replicas of input file blocks
  - Map tasks scheduled so HDFS input block replica are on same machine or same rack
- Effect: Thousands of machines read input at local disk speed
  - Don't need to transfer input data all over the cluster over the network: eliminate network bottleneck!

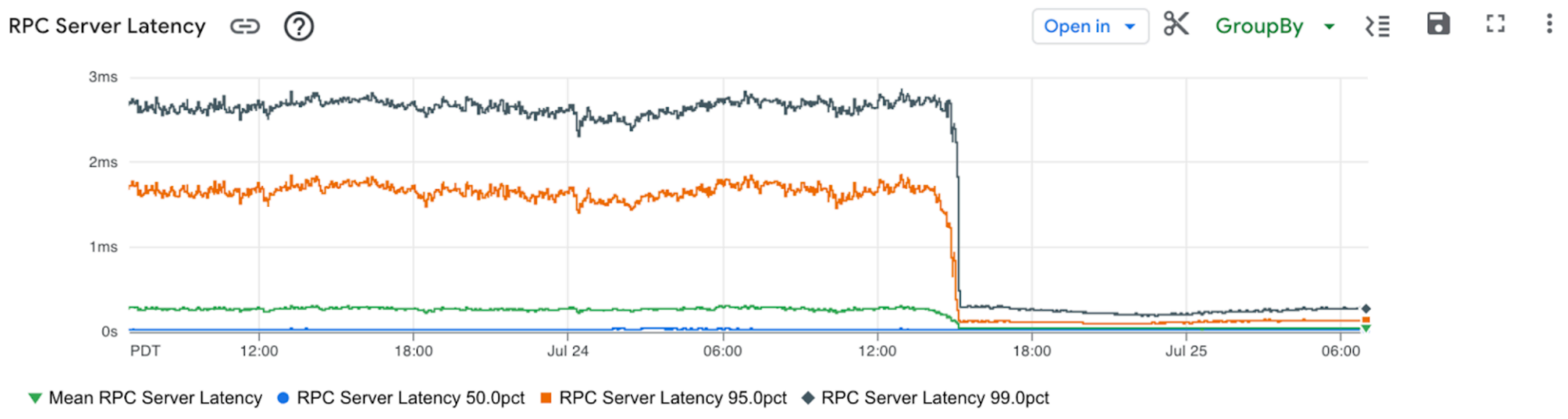
Failure in MapReduce

- Failures are the norm in data centers
- **Worker failure**
  - Master detects if workers failed by periodically pinging them (this is called “heartbeat”)
  - Re-execute in-progress map/reduce tasks
- **Master failure**
  - Single point of failure; Resume from Execution Log
- **Robust**
  - Lost 1600 of 1800 machines once, but finished fine

https://research.google.com/archive/mapreduce-osdi04-slides/index.html

Redundant Execution

- Slow workers or **stragglers** significantly lengthen completion time
  - Slowest worker can determine the total latency!
    - This is why many systems measure **99th percentile latency**

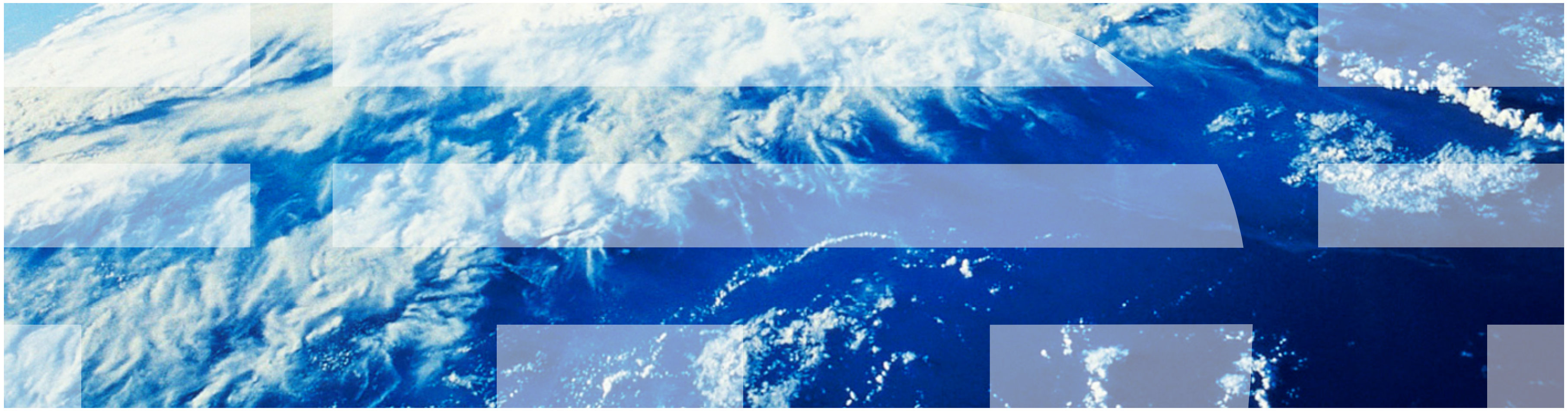


Redundant Execution

- Slow workers or **stragglers** significantly lengthen completion time
  - Slowest worker can determine the total latency!
    - This is why many systems measure **99th percentile latency**
  - Other jobs consuming resources on machine
  - Bad disks with errors transfer data very slowly
- **Simple solution**: spawn backup copies of tasks
  - Whichever one finishes first "wins"
  - I.e., treat slow executions as failed execute
    - very expensive! Almost never done.
- **Refined solution:**
  - Wait for task, if it is "suspected" of taking too long spawn another one
  - How long do you wait?
    - Waiting too long affects total delay
    - Waiting too little may spawn too many redundant tasks

Borrowed from Indranil Gupta, Faria Kalim, Patrick Wendell

Spark



Motivation

- MapReduce based tasks are slow
  - Data written to and read from storage
  - In the beginning and end of each Map and Reduce task
- Iterative algorithms not supported
  - Need to chain map reduce jobs → cumbersome, need to know how many jobs in advance (hard to do a loop)
  - Lots of unnecessary disk I/O when chaining
- No support for interactive data mining and analytics

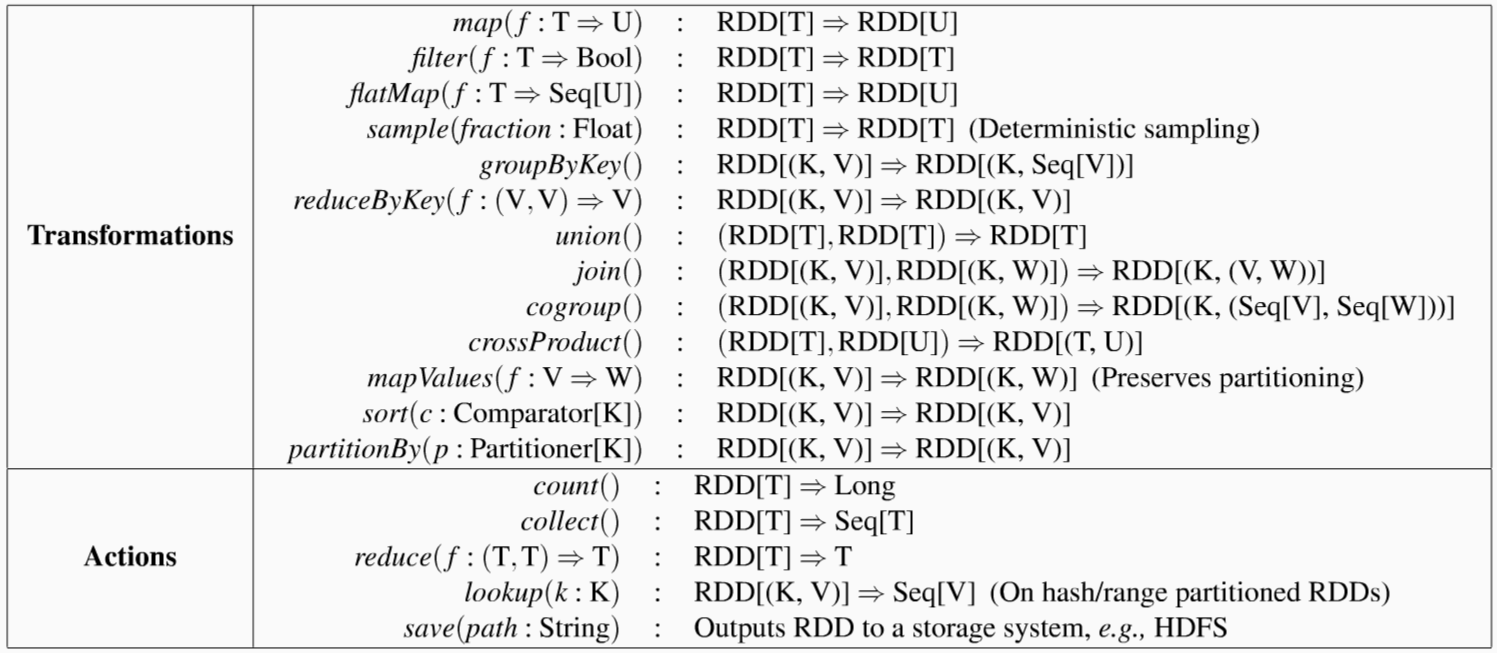
Spark’s Key Concept: Resilient Distributed Datasets (RDDs)

- Spark: in-memory analytics (avoid persistent storage as much as possible)
- A form of **distributed shared memory**
  - Eliminates the need to read/write to/from disk intermediate data between iterations
  - Read only / immutable, partitioned collections of records in memory
  - Deterministic
  - Formed by specific operations (map, filter, join, etc.)
  - Can be read from stable storage or other RDDs
- More expressive interface than MapReduce
  - Transformations (e.g. map, filter, groupBy)
  - Actions (e.g. count, collect, save)
- Recent versions of Spark introduced Datasets/Dataframes
  - Like an RDD, but you can run SQL queries over it
  - Organized into rows/columns, similar to DB relation
  - → at the end, everyone wants to use SQL! ☺
  - In practice, you’ll probably use dataframes and not RDDs, but it’s still a useful topic to understand, since dataframes in Spark are built on top of RDDs

Spark programming interface

- Lazy operations
  - Transformations not done until action
- Operations on RDDs
  - Transformations - build new RDDs
    - Can include both traditional map and/or reduce operations
  - Actions - compute and output results
    - E.g., to a file, to a Python collection
- Partitioning – layout across nodes
- Persistence – final output can be stored on disk

Example transformations and actions (no need to remember these)



Transformations	*map*(f : T ⇒ U)	:	RDD[T] ⇒ RDD[U]
	filter(f : T ⇒ Bool)	:	RDD[T] ⇒ RDD[T]
	flatMap(f : T ⇒ Seq[U])	:	RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(f : (V, V) ⇒ V)	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	union()	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	join()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	cogroup()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	crossProduct()	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	mapValues(f : V ⇒ W)	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	sort(c : Comparator[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	partitionBy(p : Partitioner[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	*count*()	:	RDD[T] ⇒ Long
	collect()	:	RDD[T] ⇒ Seq[T]
	reduce(f : (T, T) ⇒ T)	:	RDD[T] ⇒ T
	lookup(k : K)	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	save(*path* : String)	:	Outputs RDD to a storage system, *e.g.*, HDFS

Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

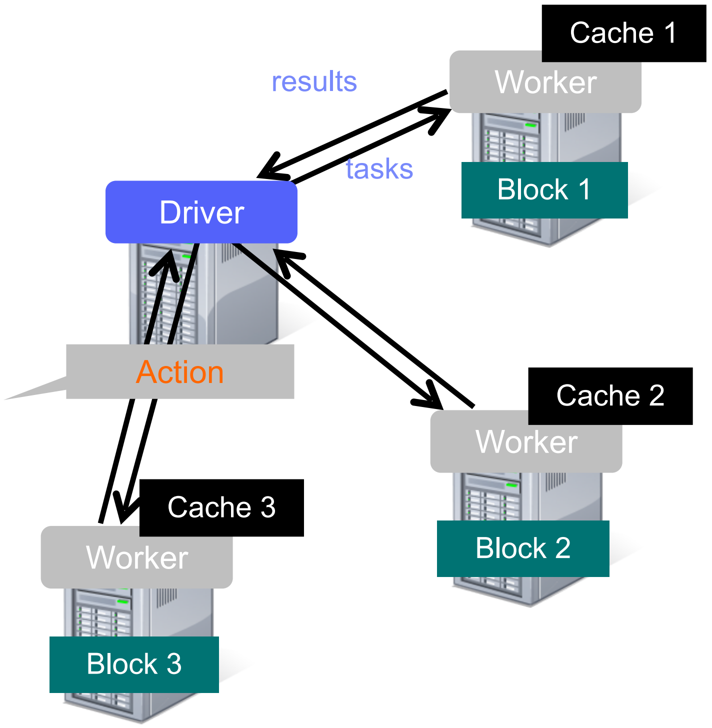
Transformed RDD

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()


messages.filter(lambda s: "mysql" in s).count()

messages.filter(lambda s: "php" in s).count()

. . .
```

Action

Full-text search of Wikipedia

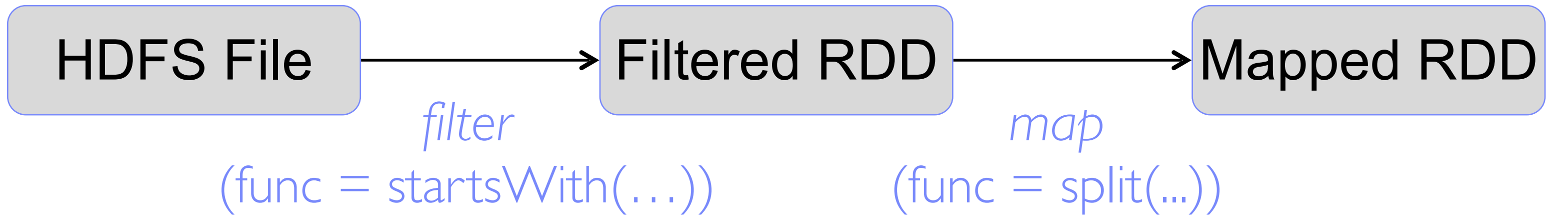
- 60GB on 20 EC2 machines
- 0.5 sec vs. 20s for on-disk



Fault Recovery

RDDs track ***lineage*** information that can be used to efficiently recompute lost data

```
msgs = textFile.filter(lambda s: s.startsWith("ERROR"))
               .map(lambda s: s.split("\t")[2])
```



Creating RDDs

```
# Turn a Python collection into an RDD
>sc.parallelize([1, 2, 3])

# Load text file from local FS, HDFS, or S3
>sc.textFile("file.txt")
>sc.textFile("directory/*.txt")
>sc.textFile("hdfs://namenode:9000/path/file")
```

Basic Transformations

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

# Pass each element through a function
>squares = nums.map(lambda x: x*x)   // {1, 4, 9}

# Keep elements passing a predicate
>even = squares.filter(lambda x: x % 2 == 0) // {4}

# Map each element to zero or more others
>nums.flatMap(lambda x: => range(x))
    ># => {0, 0, 1, 0, 1, 2}
```

Range object (sequence of numbers 0, 1, …, x-1)

Basic Actions

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

# Retrieve RDD contents as a local collection
>nums.collect() # => [1, 2, 3]

# Return first K elements
>nums.take(2)   # => [1, 2]

# Count number of elements
>nums.count()   # => 3

# Merge elements with an associative function
>nums.reduce(lambda x, y: x + y)  # => 6

# Write elements to a text file
>nums.saveAsTextFile("hdfs://file.txt")
```

Working with Key-Value Pairs

Spark’s “distributed reduce” transformations operate on RDDs of key-value pairs

Python:

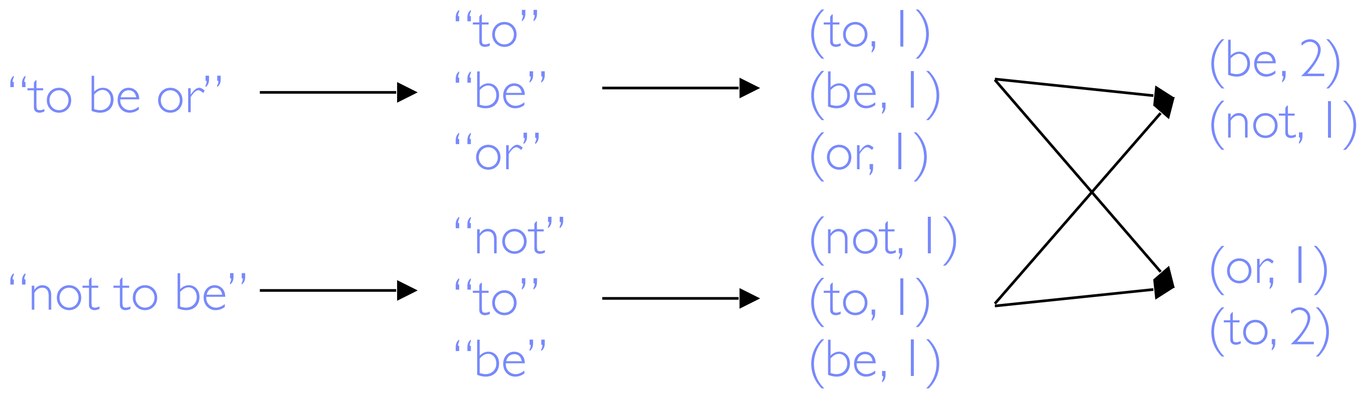
```
pair = (a, b)
pair[0] # => a
pair[1] # => b
```

Some Key-Value Operations

```
>pets = sc.parallelize(
   [("cat", 1), ("dog", 1), ("cat", 2)])

>pets.reduceByKey(lambda x, y: x + y)
                  # => {(cat, 3), (dog, 1)}

>pets.groupByKey() # => {(cat, [1, 2]), (dog, [1])}

>pets.sortByKey()  # => {(cat, 1), (cat, 2), (dog, 1)}
```

Example: Word Count

```
>lines = sc.textFile("hamlet.txt")

>counts = lines.flatMap(lambda line: line.split(" "))
               .map(lambda word => (word, 1))
               .reduceByKey(lambda x, y: x + y)
```

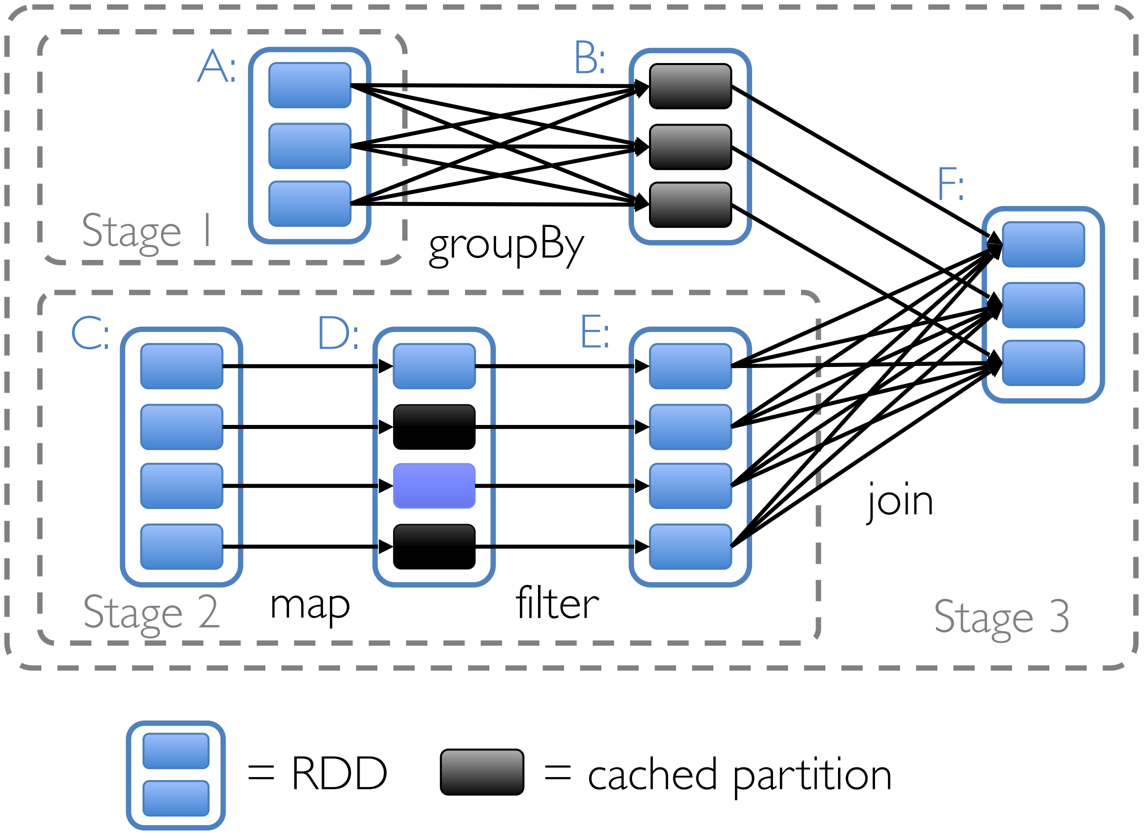


More RDD Operators

map	reduce	sample
filter	count	take
groupBy	fold	first
sort	reduceByKey	partitionBy
union	groupByKey	save ...
join	cogroup	
leftOuterJoin	zip	

Under The Hood: DAG Scheduler

- General task graphs
- Automatically pipelines functions
- Data locality aware
- Partitioning aware to avoid shuffles



SQL on Spark

- Spark SQL allows you to use SQL on Spark
- Instead of using RDDs, it uses DataFrames
  - Like an RDD, but in a table format
  - Each column has a name
- A few useful operations:
- df.collect()
  - [Row(price=100, company='Ford'), Row(price=5, company='VW')]

Return records as a list

- df.columns
  - ['price', 'company']

Returns column format

- df.count()
  - 2

Returns number of rows

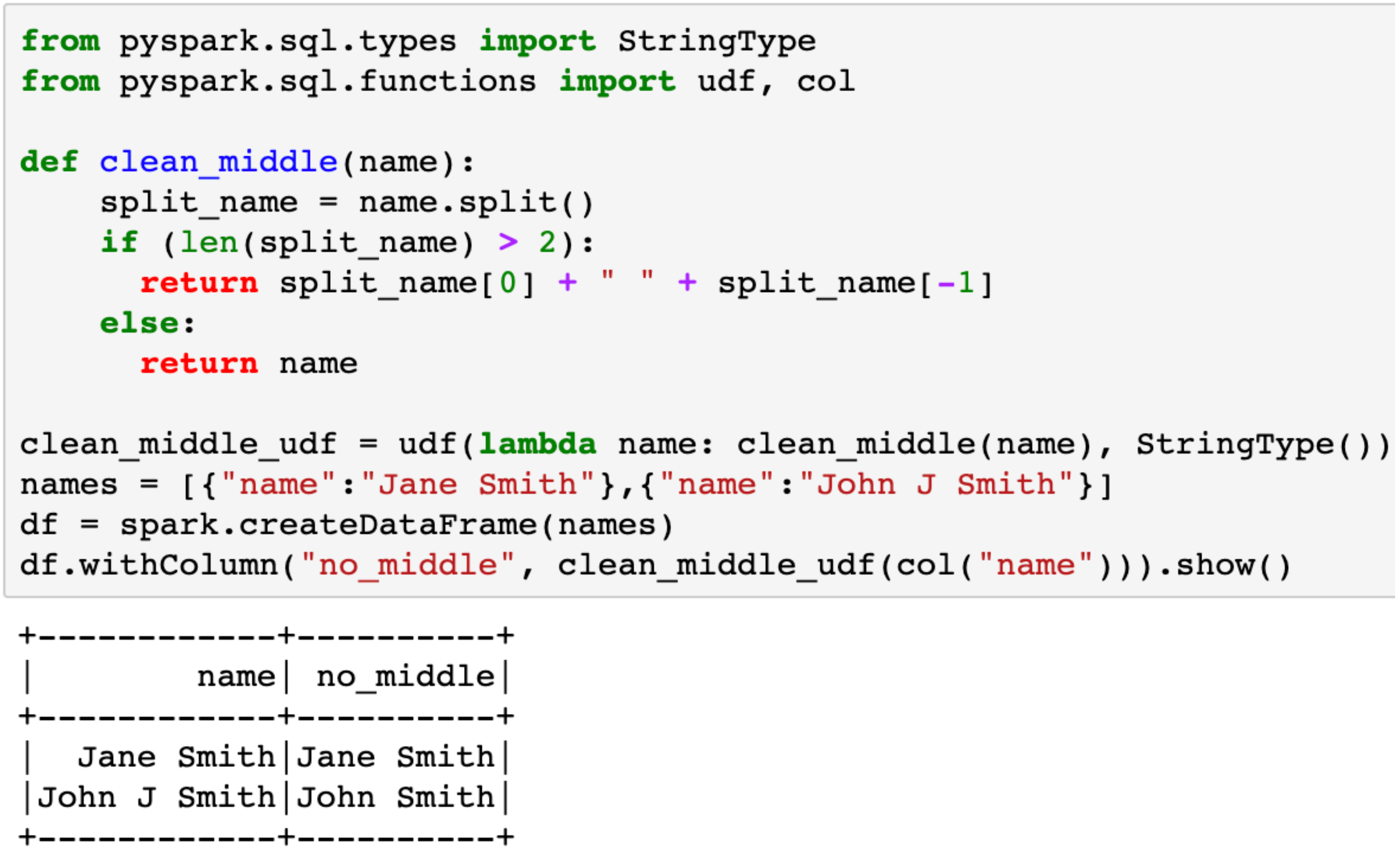
- df.filter(df.price > 50).collect()
  - [Row(price=100, company='Ford')]

Filters

User-Defined Functions (UDF) in Spark

- Sometimes you need to write a custom operation that is run on each row in a dataframe
  - Cleaning/mapping/filtering strings
  - Custom math operation
- For example:
  - For a column that contains names, remove all middle names and just keep first and last name
  - Extract the name of the service from an error log
  - Change the names "Nick, Rick" to "Nicholas, Richard"
- UDF allows you to define a new custom function that will be applied to each row
  - Word of caution: be sure to consider NULL case
    - Most commonly within the UDF itself
  - Need to register UDF function (different name than the Python function)

UDF Example



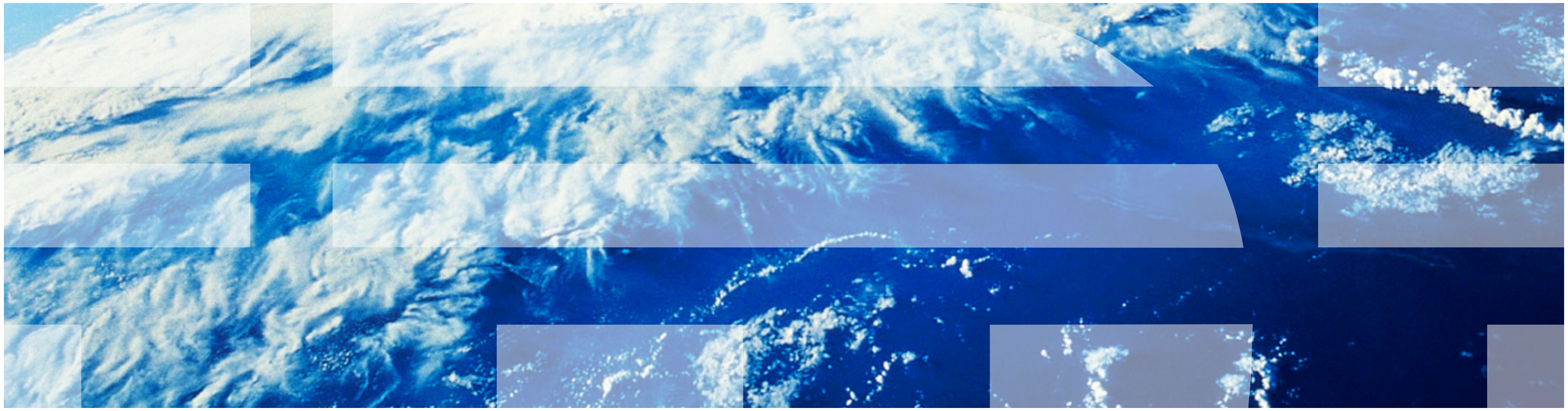
```python
from pyspark.sql.types import StringType
from pyspark.sql.functions import udf, col

def clean_middle(name):
    split_name = name.split()
    if (len(split_name) > 2):
      return split_name[0] + " " + split_name[-1]
    else:
      return name

clean_middle_udf = udf(lambda name: clean_middle(name), StringType())
names = [{"name":"Jane Smith"},{"name":"John J Smith"}]
df = spark.createDataFrame(names)
df.withColumn("no_middle", clean_middle_udf(col("name"))).show()
```

```
+------------+----------+
|        name| no_middle|
+------------+----------+
|  Jane Smith|Jane Smith|
|John J Smith|John Smith|
+------------+----------+
```

UDF Example (raw text)

```
from pyspark.sql.types import StringType
from pyspark.sql.functions import udf, col


def clean_middle(name):
    split_name = name.split()
    if (len(split_name) > 2):
      return split_name[0] + " " + split_name[-1]
    else:
      return name


clean_middle_udf = udf(lambda name: clean_middle(name), StringType())

names = [{"name":"Jane Smith"},{"name":"John J Smith"}]
df = spark.createDataFrame(names)
df.withColumn("no_middle", clean_middle_udf(col("name"))).show()
```

Detour: Regex



Regular expressions (Regex)

- A scripting language for matching and/or manipulating strings of text
  - Manipulating text with built-in Python string functions can be cumbersome
  - Python and PySpark support regex
- Examples of operations
  - Find all substrings with “helloworld” and replace with: “Hello World”
  - Find all substrings with “fox”, but exclude “foxtrot”, and replace with the word ”wolf”
  - Remove all special characters (e.g., !%&), punctuation (e.g., .,;:)
- Useful in many programming contexts
  - Spark
  - Python
  - SQL
  - Etc.
- Hint: can be useful in homework assignment (coupled with a UDF perhaps?)
- Use free online regex tool for debugging: https://regex101.com/

Regex matching examples

Example text: fox foxtrot ox box

fox: fox foxtrot ox box

ox: fox foxtrot ox box

fox\w: fox foxtrot ox box (\w matches any word character)

fox\w*: fox foxtrot ox box (* matches zero or more of the preceding character)

fox\w+: fox foxtrot ox box (+ matches one or more of the preceding character)

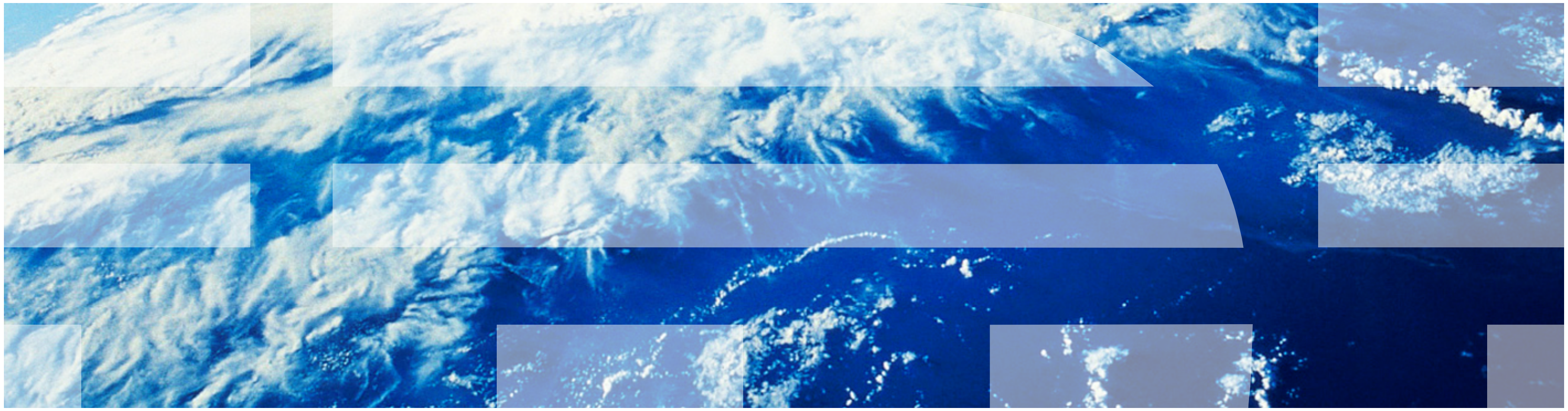
[^f]ox: fox foxtrot ox box (without the letter f in the beginning)

[^f\W]ox: fox foxtrot ox box (without the letter f or a non-word character)

[bf]ox: fox foxtrot ox box (match on b or f followed by ox)

Borrowed from Tathagata Das and Indranil Gupta

Stream Processing



Motivation: ”Hybrid” Between OLAP and OLTP

- Large amounts of data => Need for real-time views of data
  - Social network trends, e.g., Twitter real-time search
  - Website statistics, e.g., Google Analytics
  - Intrusion detection systems, e.g., in most datacenters
- Process large amounts of data
  - With latencies of few seconds
  - With high throughput
- Not exactly OLTP, because updates don’t happen in place (e.g., on rows in table)
- Not exactly OLAP, because it is real time, updates are granular
- Streaming is a relatively new class of data system

Would MapReduce or normal Spark work?

- **Batch Processing** => need to wait for entire computation on large dataset to complete
- Not intended for long-running and real-time stream-processing

Examples of stream processing jobs

A) Uber
    Calculating surge prices

B) LinkedIn
    Aggregating updates into one email

C) Netflix
    Understanding user behavior to improve personalization

D) TripAdvisor
    Calculating earnings per day & fraud detection

Discretized Stream Processing

- Run a streaming computation as a **series of very small, deterministic batch jobs**
- Batch processing models, like MapReduce, recover from faults and stragglers efficiently
  - Divide job into deterministic tasks
  - Rerun failed/slow tasks in parallel on other nodes
- Same recovery techniques at lower time scales
  - Transformations are not lost (or performed twice) if a worker dies

Spark Streaming

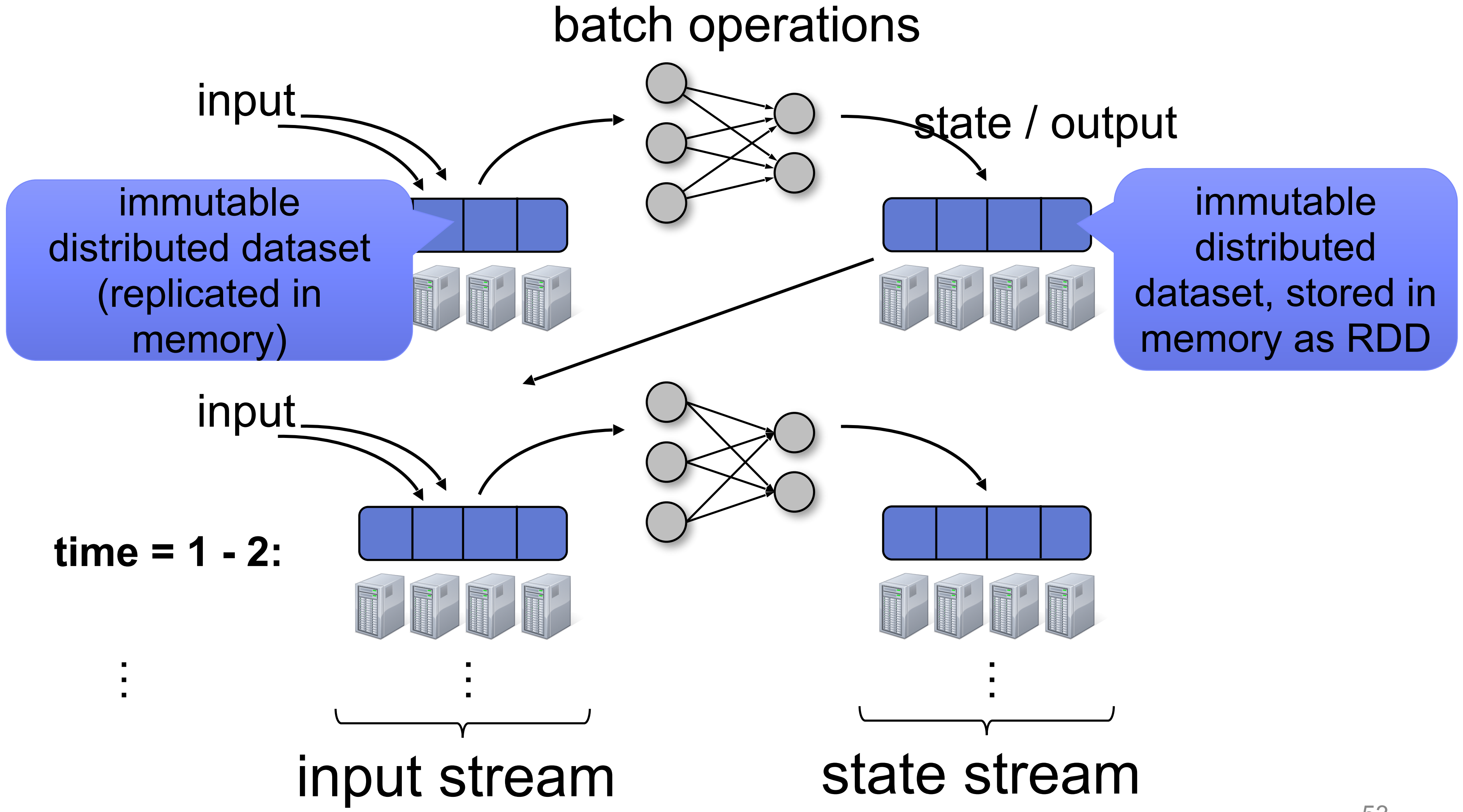
- State between batches kept in memory in **fault-tolerant dataset**
  - Specifically as an RDD/Dataframe/Dataset

- Batch sizes can be reduced to as low as 1/2 second to achieve ~ 1 second latency

- Combines streaming and batch workloads

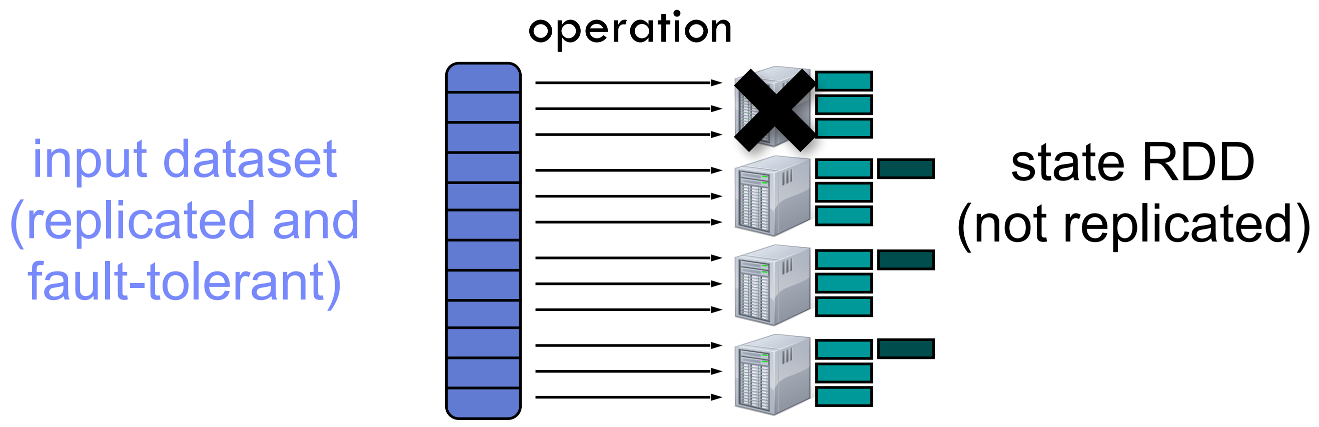
- Many other alternatives:
  - Apache Storm
  - Apache Flink
  - Amazon Kinesis
  - Google Dataflow
  - …

Discretized Stream Processing


batch operations
input
state / output
immutable distributed dataset (replicated in memory)
immutable distributed dataset, stored in memory as RDD
input
time = 1 - 2:
⋮
⋮
⋮
input stream
state stream

Fault Recovery

- State stored as RDD
  - Deterministically re-computable parallel collection
  - Remembers lineage of operations used to create them
- Fault / straggler recovery is done **in parallel** on other nodes



Fast recovery from faults without full data replication

Programming Model

- A Discretized Stream or **DStream** is a series of RDDs representing a stream of data
  - API *very similar* to RDDs

- DStreams can be created…
  - Either from live streaming data
  - Or by transforming other DStreams

DStream Data Sources

- Many data sources can be inupts
  - HDFS
  - Kafka
  - Flume
  - Twitter
  - …

Transformations

Build new streams from existing streams

- Filters/aggregate operations
  - map, flatMap, filter, count, reduce,
  - groupByKey, reduceByKey, sortByKey, join
  - etc.
- New window and stateful operations
  - window, countByWindow, reduceByWindow
  - countByValueAndWindow, reduceByKeyAndWindow
  - updateStateByKey
  - etc.

Output Operations

Send data to outside world

- saveAsHadoopFiles
- print – prints on the driver’s screen
- foreach - arbitrary operation on every RDD

Example

Process a stream of Tweets to find the 20 most popular hashtags in the last 10 mins, updated every 1 second

1. Get the stream of Tweets and isolate the hashtags
2. Count the hashtags over 10 minute window, updated every 1 second

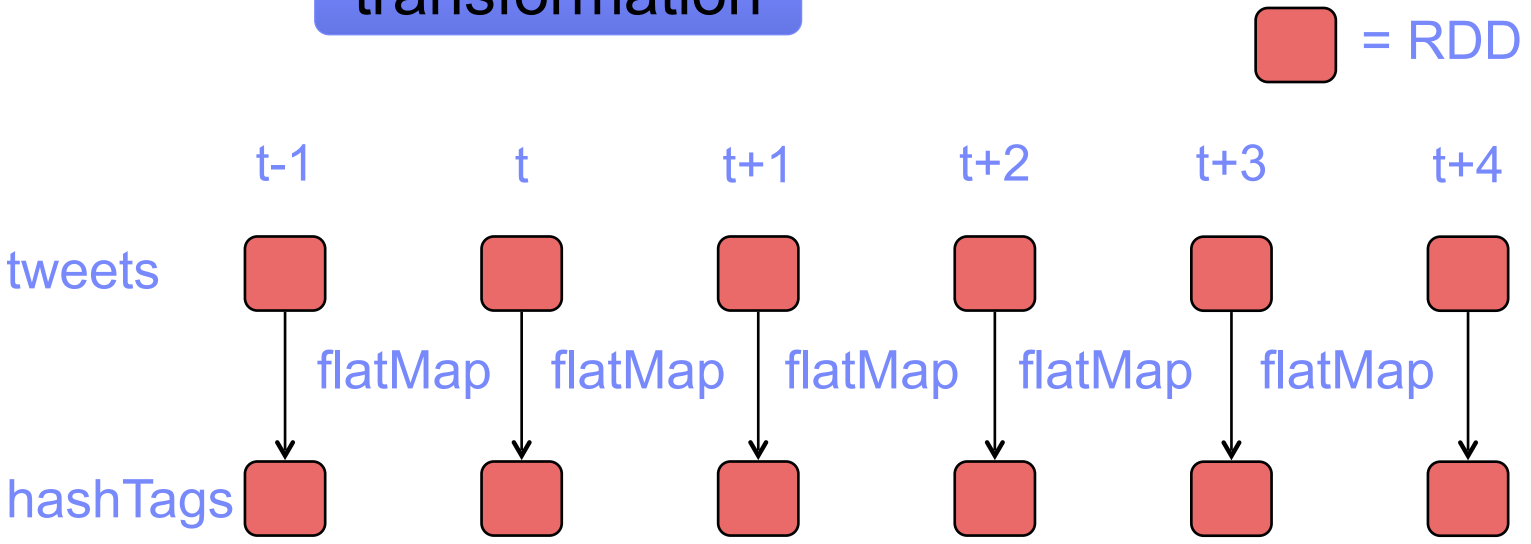
1. Get the stream of Hashtags

```
tweets = ssc.twitterStream(username, password)
```

DStream

```
hashtags = tweets.flatMap (lambda tweet: tweet.getTags())
```

transformation

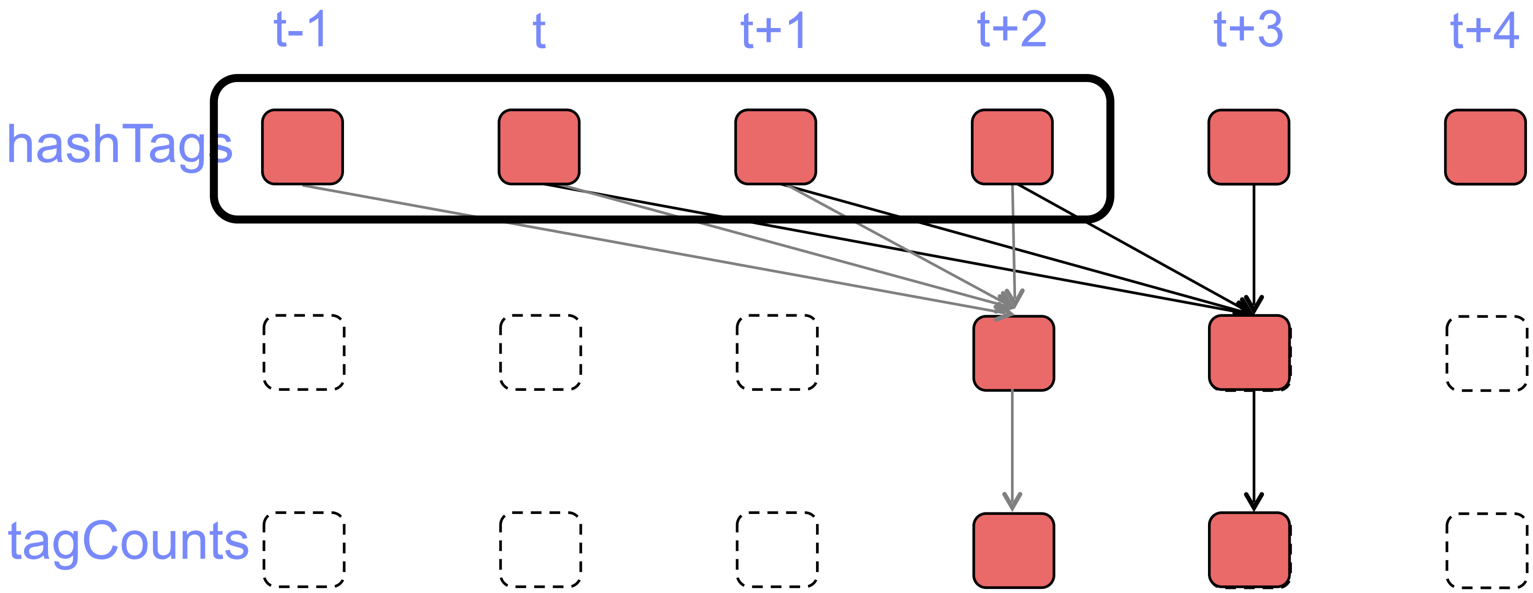


2. Count the hashtags over 10 min

```
tweets = ssc.twitterStream(userna
hashtags = tweets.flatMap (lambda tweet: tweet.getT
tagCounts = hashTags.window(Minutes(10), Seconds(1)).countByValue()
```

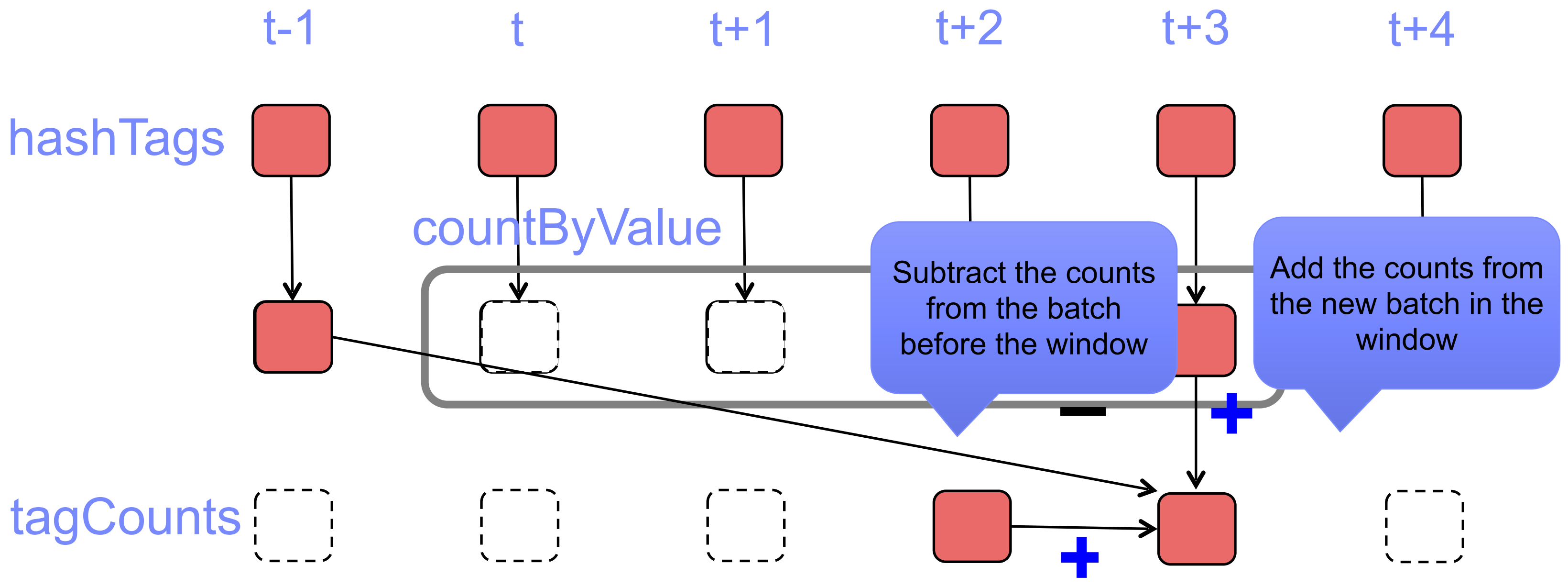
Window length

Sliding interval



2. Count the hashtags over 10 min

```
tweets = ssc.twitterStream(username, password)
hashtags = tweets.flatMap (lambda tweet: tweet.getTags())
tagCounts = hashtags.countByValueAndWindow(Minutes(10), Seconds(1))
```


t-1
t
t+1
t+2
t+3
t+4
hashTags
countByValue
tagCounts
Subtract the counts from the batch before the window
Add the counts from the new batch in the window
–
+
+