

Cloud Programming: Lecture6 – Other Parallel Execution Frameworks

***National Tsing-Hua University
2015, Spring Semester***



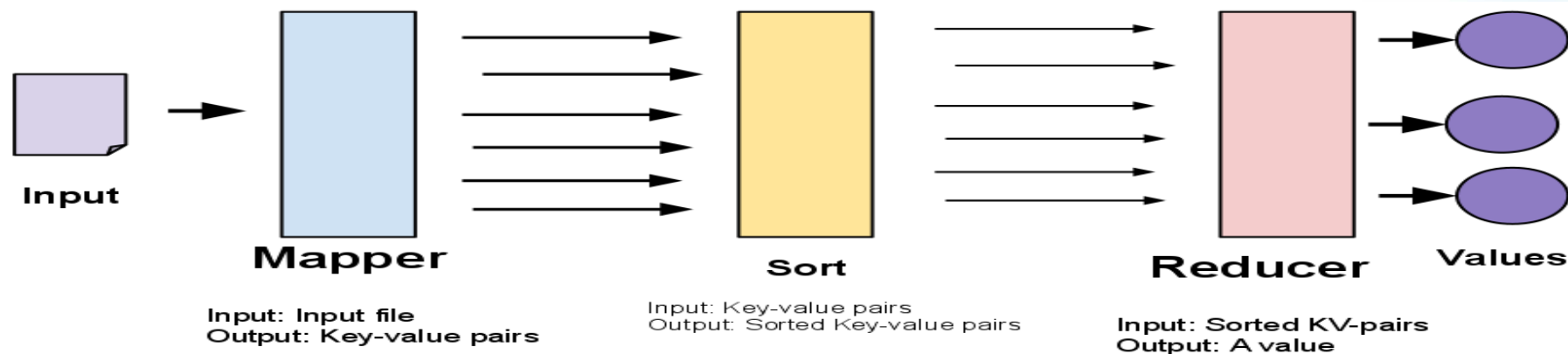
國立清華大學

Outline

- Overview of Hadoop/MapReduce Limitation
- Spark: In-memory computing
- Spark Streaming

Limitation of MapReduce

- Simple but limited programming model



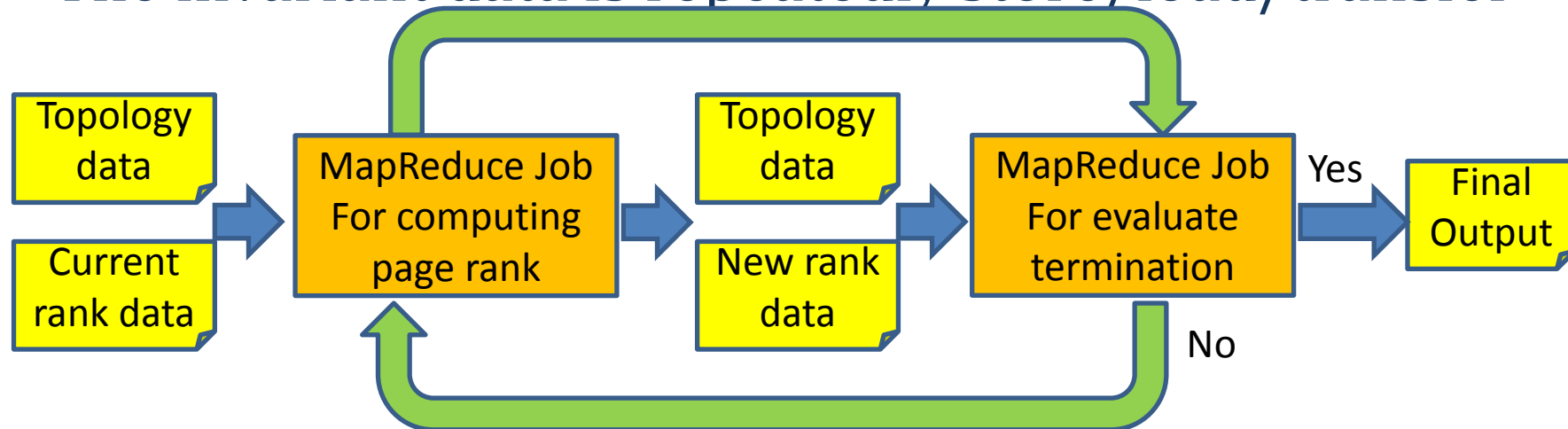
- Can only apply two computation functions in a job: Map&Reduce
 - ➔ More complex work must use **multiple** jobs
- The input and output of a job must store into a FS
 - ➔ FS(disk) is the only device to provide **data persistency**

Iterative Data Processing

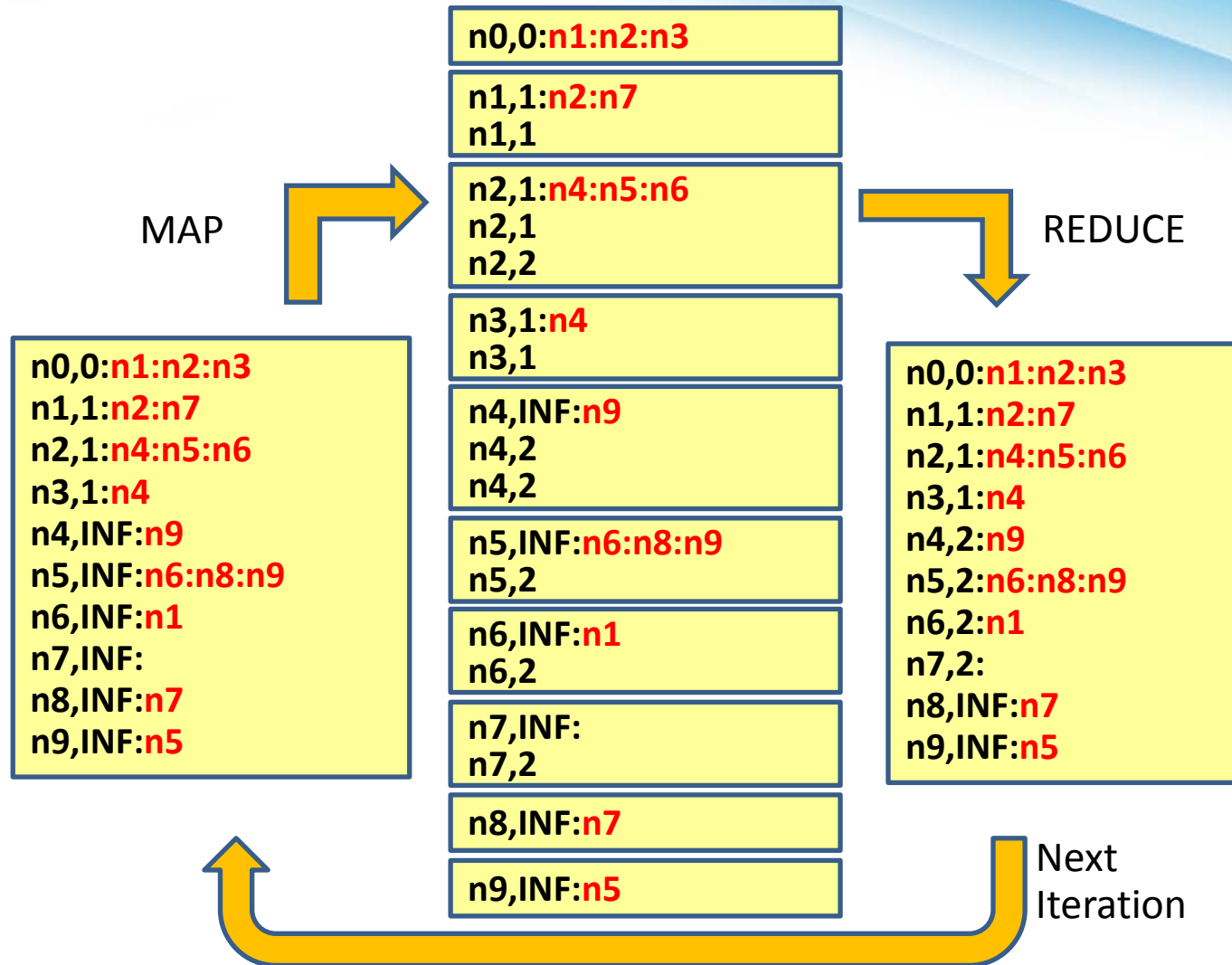
- **Definition:**
 - A mathematical procedure that generates a sequence of improving **approximate solutions** for a class of problems
 - The procedure iterates until converge or reach some termination criteria
- **Property of computation**
 - Termination criteria must be evaluated after each iteration
 - The output from an iteration will be the input of the next iteration
 - Invariant data must re-load and re-processing in the loop
- **Applications:**
 - Machine learning algorithm: K-mean
 - Graph algorithm: Page-rank
 - Approximation algorithm

Iterative Method in MapReduce

- Each iteration is submitted as an independent job
 - hard to ask the scheduler to manage and guarantee the performance of the whole application
- Termination criteria is evaluated after each iteration
- Data is written and read out disk after each iteration
- The invariant data is repeatedly store/load/transfer

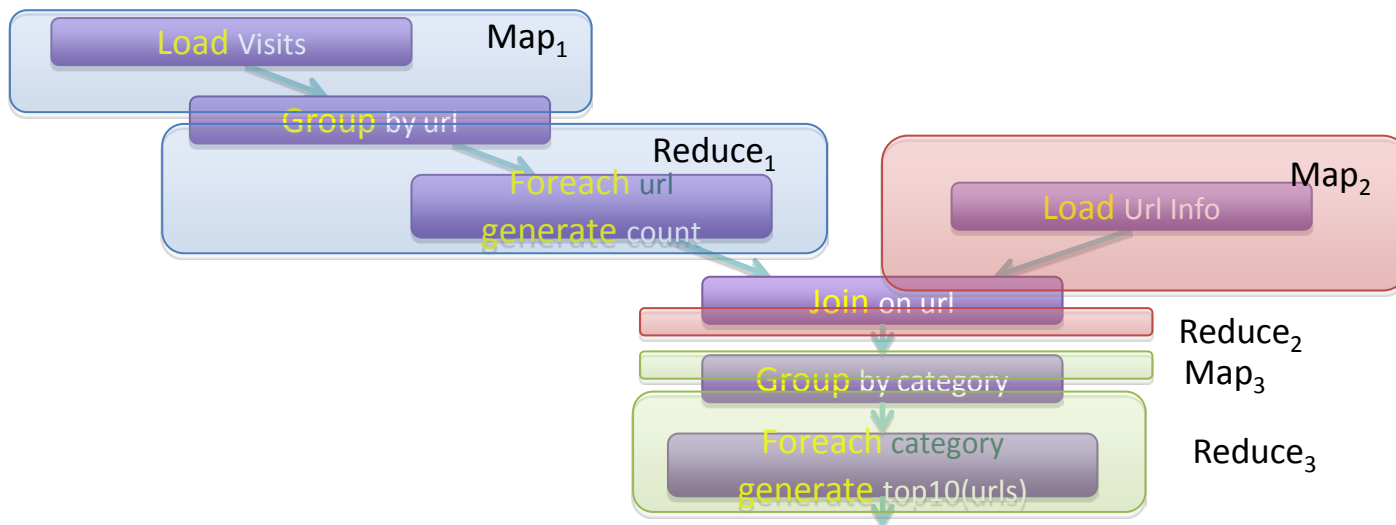


Iterative Method in MapReduce



Interactive Processing

- Definition: computation involving the exchange of information between a user and the computer
- Property:
 - Require short response time → disk is too slow
 - Repeatedly process on the same set of data → redundant I/O
 - Complex data-flow → can be specified by one job
- Example:
 - Ad-Hoc query processing, ex: Pig, Hive

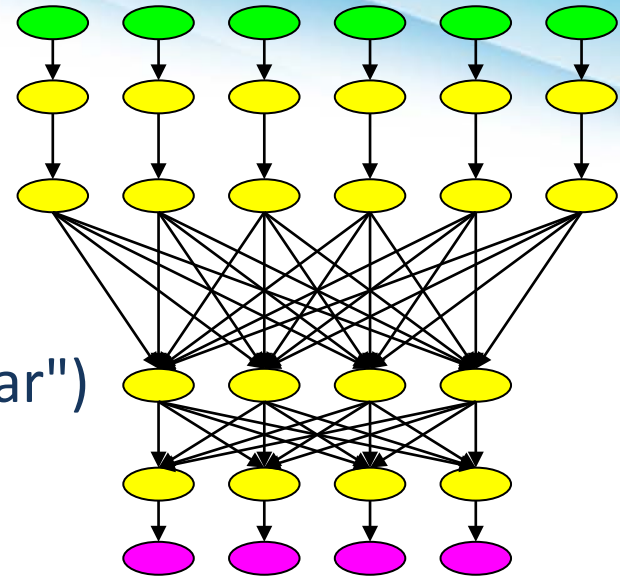


The World of Big Data Tools

- Computing Models:
 - MapReduce: divide-and-conquer data flow
 - For common data processing applications.
 - DAG: direct acyclic graph data flow
 - For more general applications.
 - Graph: specific problems for graph algorithms, such as shortest path, travelling salesman problem, etc.
 - Getting more popular for analyzing social network data.
 - It is also a BSP
 - BSP(Bulk Synchronous Parallel): processing involving synchronization points.
 - Commonly seen from iterated algorithm with data dependency

DAG Computing Model

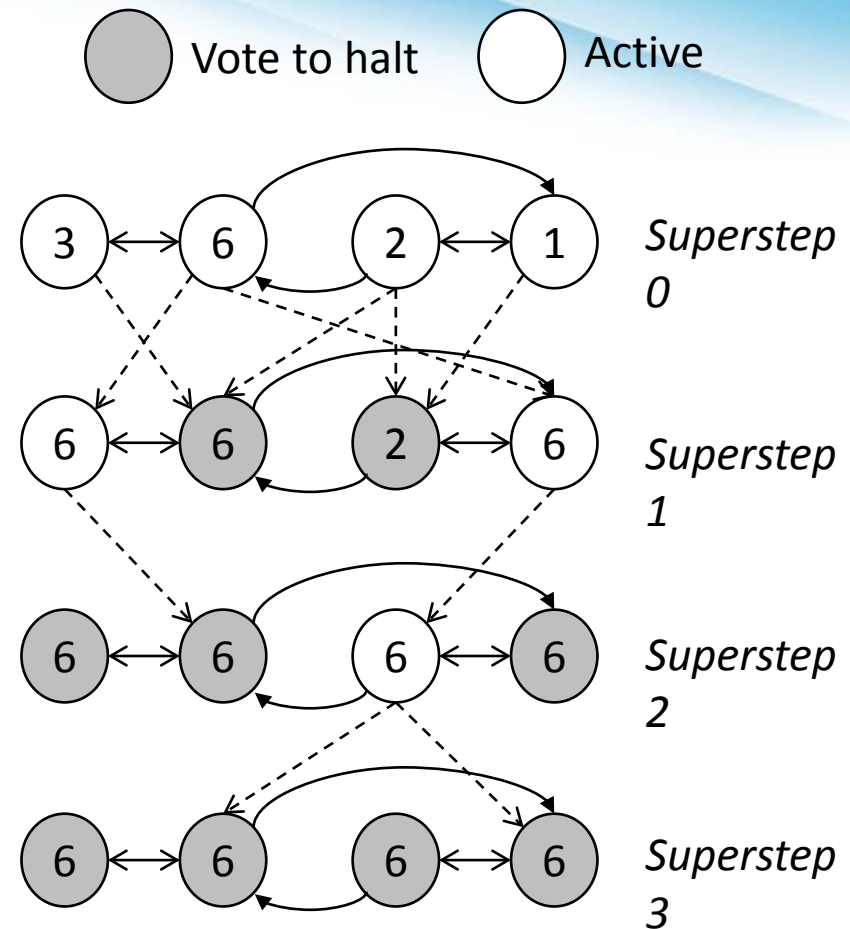
```
var logentries =  
  from line in logs  
  where !line.StartsWith("#")  
  select new LogEntry(line);  
var user =  
  from access in logentries  
  where access.user.EndsWith("@\"\\ulfar\"")  
  select access;  
var accesses =  
  from access in user  
  group access by access.page into pages  
  select new UserPageCount("ulfar", pages.Key, pages.Count());  
var htmAccesses =  
  from access in accesses  
  where access.page.EndsWith(".htm")  
  orderby access.count descending  
  select access;
```



Synchronous Computation Model

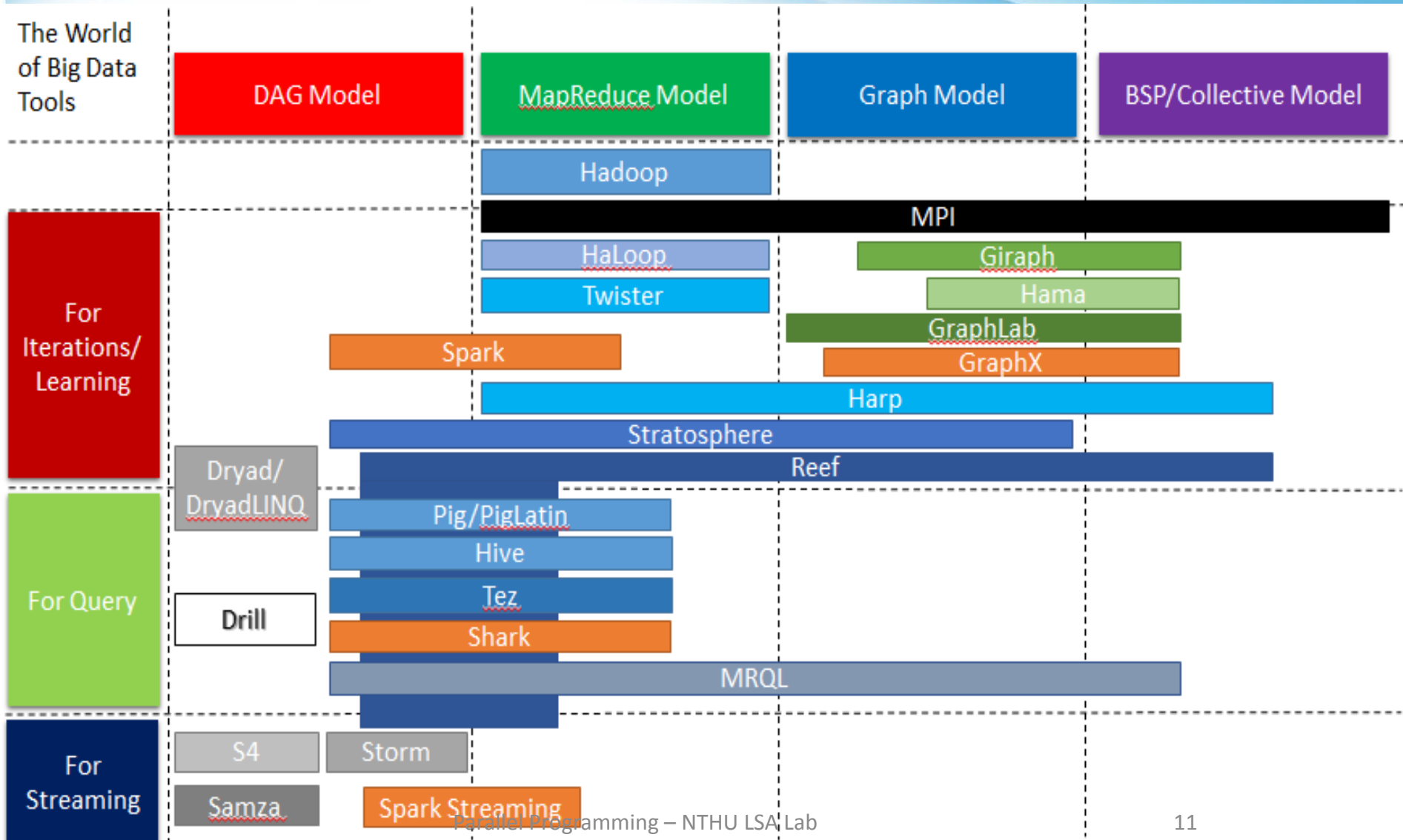
- Superstep as iteration
- Vertex state machine:
Active and Inactive, vote to halt
- Message passing between vertices
- User defines the computing task on each vertex

```
Vertex() {
  i_val := val
  for each message m
    if m > val then val := m
  if i_val == val then
    vote_to_halt
  else
    for each neighbor v
      send_message(v, val)
}
```

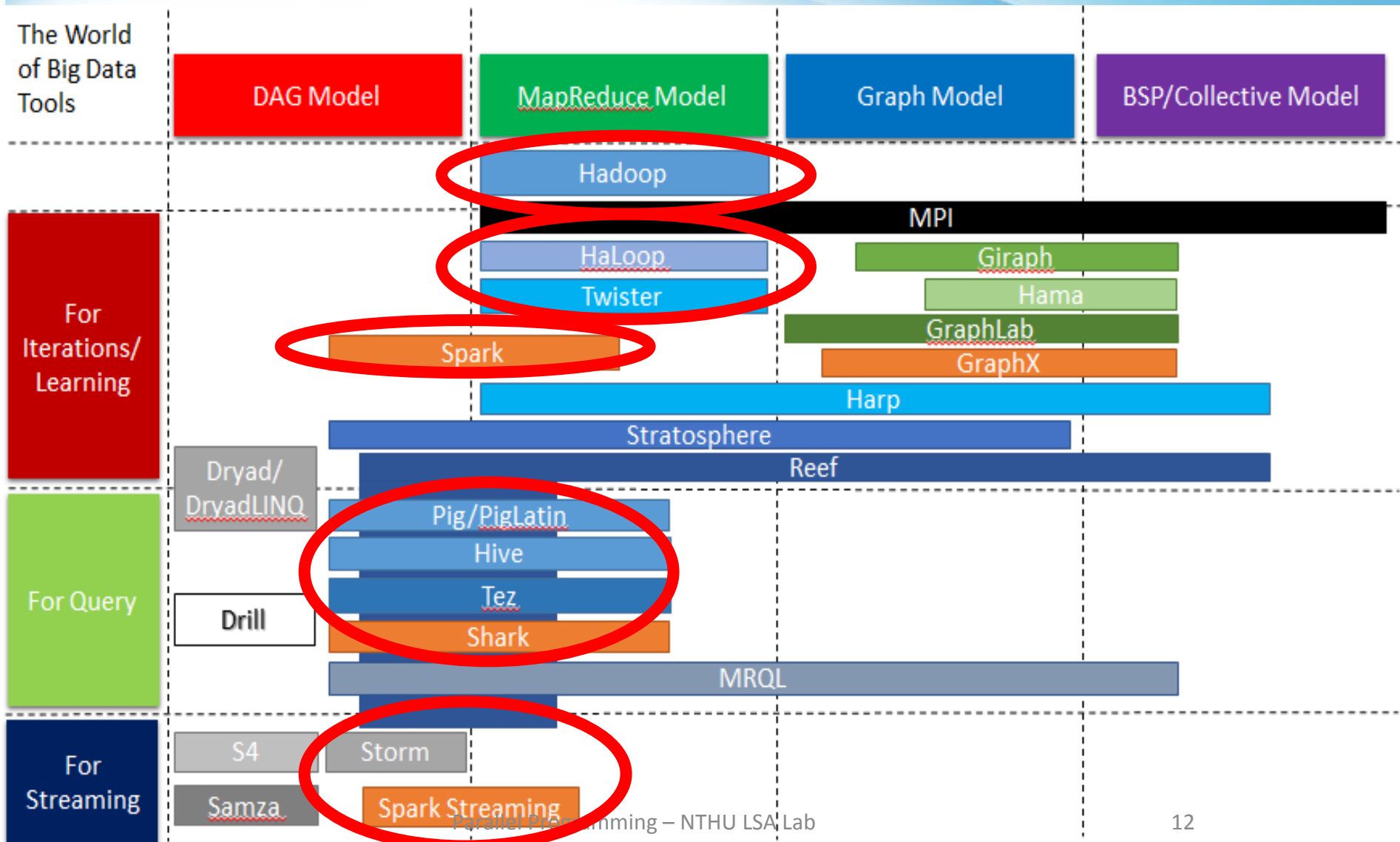


Maximum Value Example

Big Picture



Big Picture



Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica
From UC Berkeley

SPARK: LOW LATENCY, MASSIVELY PARALLEL PROCESSING FRAMEWORK

Motivation & Objectives

- Motivation:
 - Data reuse is frequent in iterative and interactive data processing
 - MapReduce only support acyclic workflow
- Objectives:
 - Utilize **DSM** (Distributed Shared Memory) in data processing to enable **in-memory computing**
 - Allow users to explicitly **cache dataset in memory across machines** and reuse it in multiple MapReduce-like parallel operations.
 - Retain the **scalability and fault tolerance** property like MapReduce

Overview

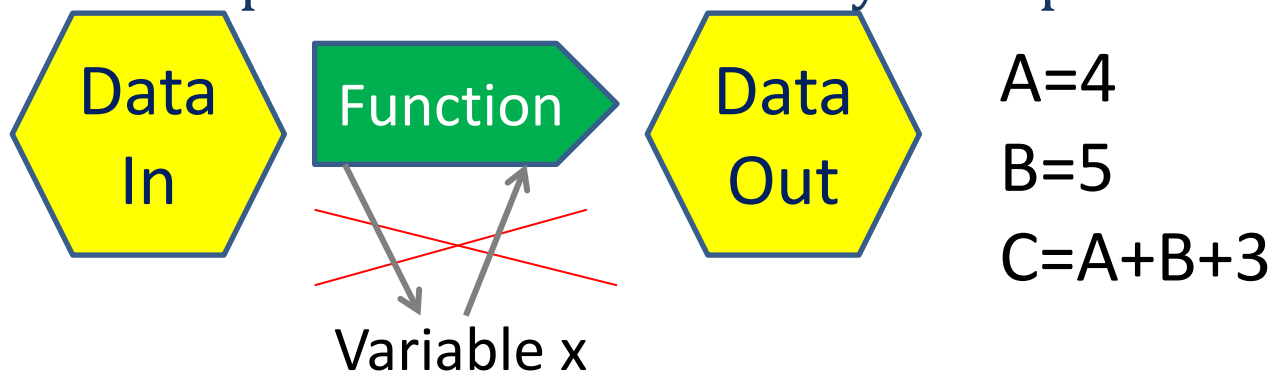
- Spark introduces an data abstraction called **Resilient Distributed Datasets (RDDs)**:
 - RDD is a *read-only collection* of objects partitioned across a set of machines need not to consider the issue of sync, nobody can write it
 - RDD **can be rebuilt** if a partition is lost using the “*lineage*” technique
- Spark is integrated into a general programming language called “**scala**”
 - Pure-bred **O.O language**: every variable/dataset is an object and every operation is a method-call
 - Seamless **Java interpreter**
 - Programmer specify operations to **transform dataset**
 - Operations are **parallelized and executed by Spark**

Functional Programming

- Immutable data + function = functional programming
 - Theoretical foundation based on Alonzo Church's **Lambda Calculus**.
 - A style of building the structure and elements of computer programs—that treats **computation as the evaluation of mathematical functions** and **avoids changing-state and mutable data**
 - It is **referential transparency**: the output value of a function **depends only on the arguments** that are input to the function.
- Languages:
 - Haskell, Erlang, Scala, Lisp, Scheme, F#

Referential Transparency

- An expression is said to be referentially transparent if it can be replaced with its value without changing the behavior of a program (in other words, yielding a program that has the same effects and output on the same input).
- While in mathematics all function applications are referentially transparent, in programming this is not always the case.
- If all functions involved in the expression are pure functions, then the expression is referentially transparent.



Imperative vs. Functional Programming

- **imperative (procedure) programming** is a programming paradigm that describes computation in terms of **statements that change a program state**.

Procedure programming	Functional programming
Everything is done in a specific order	Order of evaluation is usually undefined
Execution of a routine may have side effects	<ul style="list-style-type: none">• Must be stateless. i.e. No operation can have side effects• Always returns the same output for a given input
Tends to emphasize implementing solutions in a linear fashion	Good fit for parallel execution

About SCALA

- Scala = “Scalable Language”
- High-level language for the JVM
 - Combine **object oriented** and **functional programming** with a powerful static type system and expressive syntax
- Interoperates with Java
 - Can use any Java class
 - Can be called from Java code
- Upsurge in adoption since 2010 to handle massive parallel data processing problem

<http://www.scala-lang.org/>

https://twitter.github.io/scala_school/

SCALA: Function Definition

```
def add(a:Int, b:Int): Int = a+ b
```

```
val m:Int =add(1,2)
```

```
Println(m)
```

- Scala is a “statically typed language”
 - We define “add” to be a function which accepts two parameters of type Int and return a value of type Int.
 - “m” is defined as a variable of type Int.

SCALA: Function Definition


```
def func(a:Int, b:Int): Int = {  
    a + 1  
    b - 2  
    a*b  
}  
val p:Int fun(1,2)  
println(m)
```

- There is no explicit “return” statement! The value of the last expression in the body is automatically returned.

SCALA: Type Inference

```
def add(a:Int, b:Int) = a+ b  
val m =add(1,2)  
Println(m)
```

```
def add(a, b) = a+ b  
val m =add(1,2)  
Println(m)
```



Scala does NOT infer type
of function parameters

- The return type of the function and the type of variable “m” is not specified. Scala “**infers**” that automatically because it is a statically typed language.

SCALA: Expression Oriented Programming

```
val i = 3
val p = if(i > 0) -1 else -2
val q = if(true) "hello" else "world"
println(p)
println(q)
```

- Unlike languages like C/JAVA, almost **everything in Scala is an “expression” that returns a value!**
- Rather than programming with “statements”, we program with “expressions”

SCALA: Functions return functions

```
def fun():Int => Int = {  
    def sqr(x: Int): Int= x*x  
    sqr  
}  
val f = func();  
println(f(10));
```

- “def fun():Int => Int “ says “fun” is a function which does not take any argument and returns a function which maps an Int to an Int.

SCALA: Lazy val's

```
def hello() = {  
    println("hello")  
    10  
}  
lazy val a = hello()
```

- “hello” is NOT printed by the program because the expression which assigns a value to a “lazy” val is executed only when that lazy val is used somewhere in the code!

Basic Data Structures

- List: List of elements
 - `List(1, 1, 2) → {1, 1, 2}`
- Set: Sets have no duplicates
 - `Set(1, 1, 2) → {1, 2}`
- Tuple:
 - Groups together simple logical collections of items
 - Values have accessors that are named by their position and is 1-based rather than 0-based.
 - E.g.: `val hostPort = ("localhost", 80)`
`hostPort._1 // localhost`
`hostPort._2 // 80`

SCALA: Data collections

<code>val list = List(1, 2, 3)</code>	
<code>list.foreach(x => println(x))</code>	<code>//print 1, 2, 3</code>
<code>list.foreach(println)</code>	<code>//same</code>
<code>list.map(x=>x + 2)</code>	<code>// return new list (3,4,5)</code>
<code>list.map(_ + 2)</code>	<code>// same</code>
<code>list.filter(x=> x%2 == 1)</code>	<code>// return new list (1,3)</code>
<code>list.filter(_ %2 == 1)</code>	<code>// same</code>
<code>list.reduce((x, y)=>x+y)</code>	<code>// => 6</code>
<code>list.reduce(_ + _)</code>	<code>// same</code>

- All of these leave the list unchanged as it is immutable

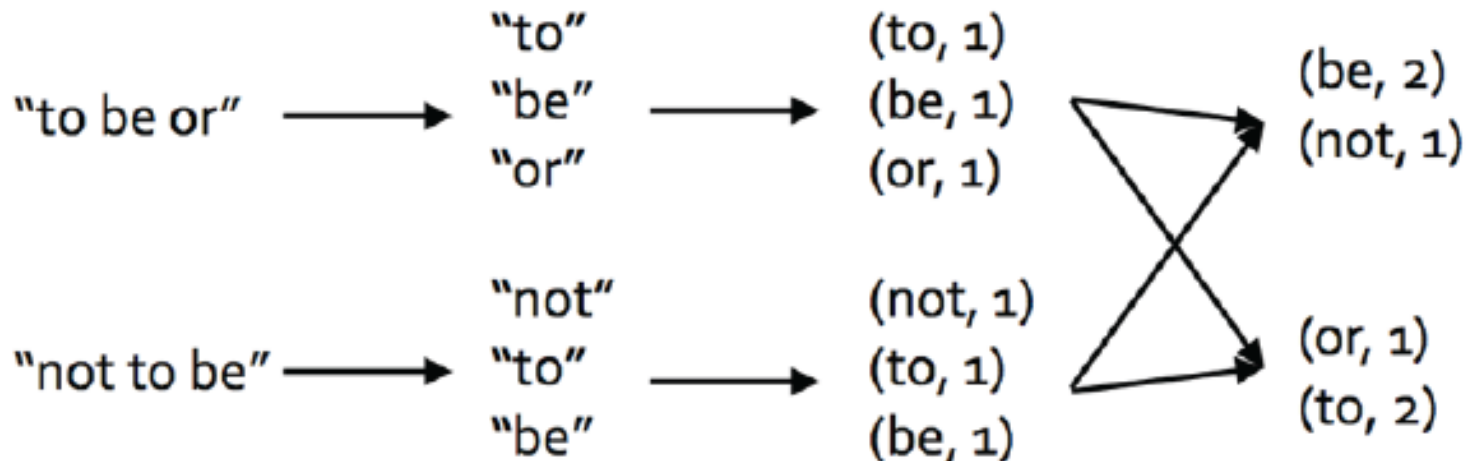
Functional methods on collections

- There are a lot of methods on Scala collections, just google Scala Seq or <http://www.scala-lang.org/api/2.10.4/index.html#scala.collection.Seq>

Method on Seq[T]	Explanation
map(f: T=>U): Seq[U]	Each element is result of f
flatMap(f: T=>Seq[U]): Seq[U]	One to many mapping
filter(f: T=>Boolean): Seq[T]	Keep elements passing f
exists(f: T=>Boolean): Boolean	True if one element passes f
forall(f: T=>Boolean): Boolean	True if all elements pass f
reduce(f: (T,T) => T): T	Merge elements using f
groupBy(f: T=>K): Map[K,List[T]]	Group elements by f
sortBy(f: T=>K): Seq[T]	Sort elements

Word Count Example

- `val lines = sc.textFile("hamlet.txt")!`
- `val counts = lines.flatMap(line => line.split(" ")).
map(word => (word, 1)).
reduceByKey(_ + _)`



Spark: RDDs

Resilient distributed datasets (RDDs)

- **Immutable, partitioned collections** of objects
- **Created through parallel *transformations*** (map, filter, groupBy, join, ...) on data in stable storage
- Can be ***cached*** for efficient reuse
- RDDs are lazy and ephemeral. That is, partitions of a dataset are **materialized (i.e. computed)** on demand when they are used in a parallel operation

Actions on RDDs

- Count, reduce, collect, save, ...

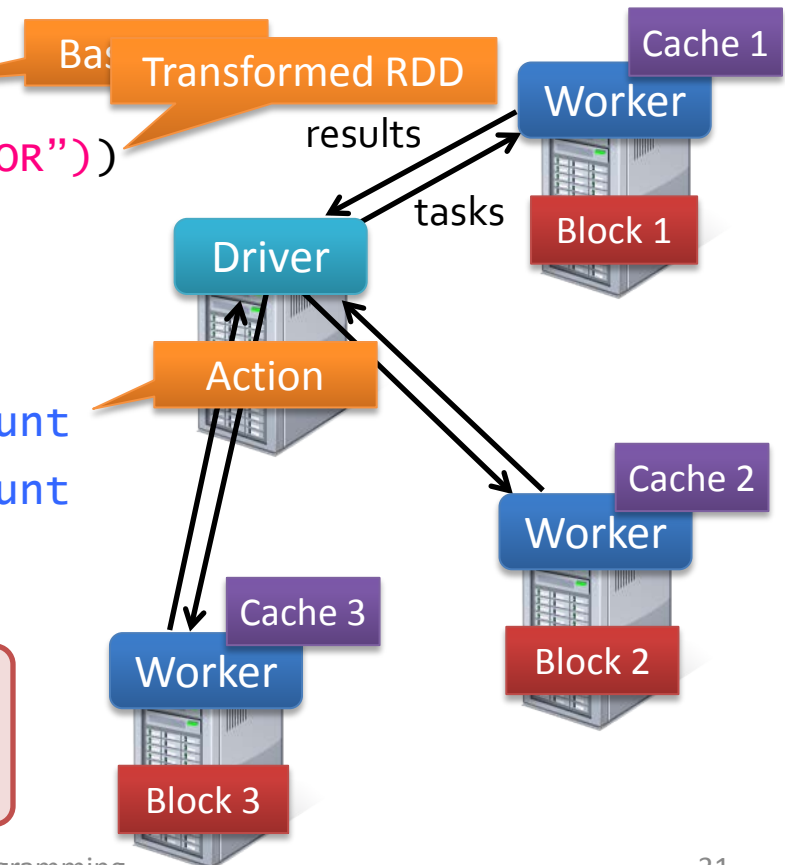
Example: Log Mining

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

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
cachedMsgs = messages.cache()

cachedMsgs.filter(_.contains("foo")).count
cachedMsgs.filter(_.contains("bar")).count
. . .
```

Result: scaled to 1 TB data in 5-7 sec
(vs 170 sec for on-disk data)

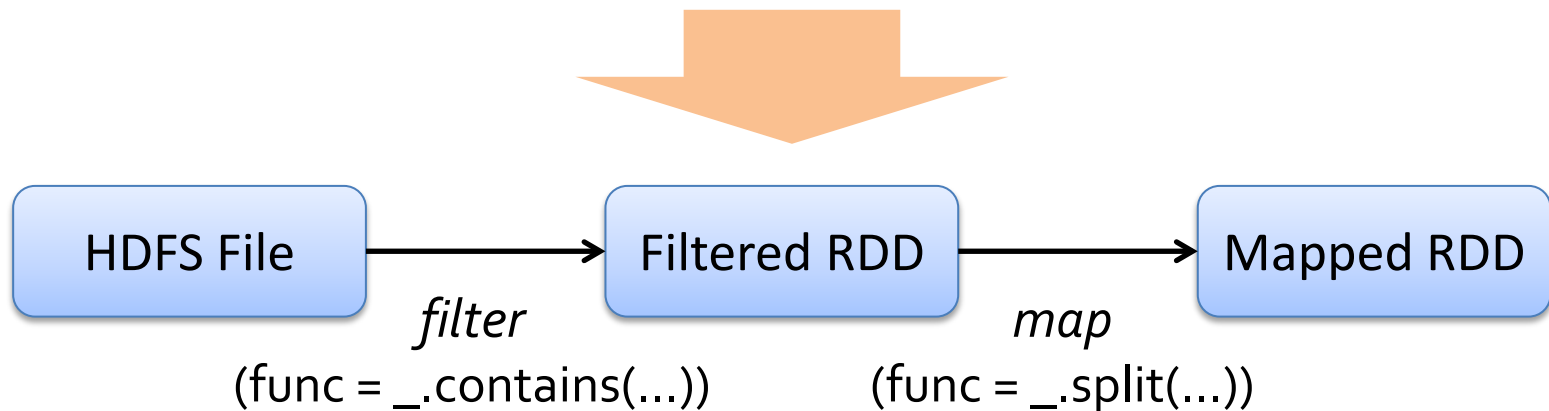


RDD Fault Tolerance

RDDs maintain *lineage* information that can be used to reconstruct lost partitions

Ex:

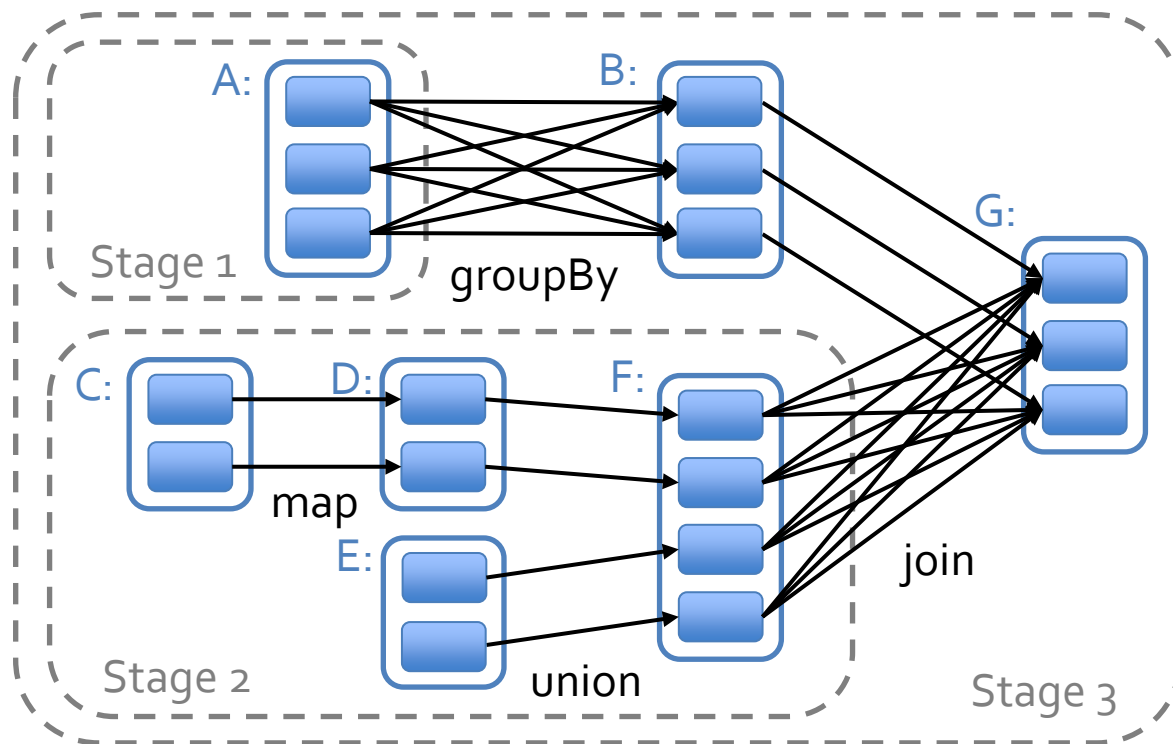
```
messages = textFile(...).filter(_.startsWith("ERROR"))  
                        .map(_.split('\t')(2))
```



*If lineage is too long, it could cause stack overflow (out of memory) problem during execution

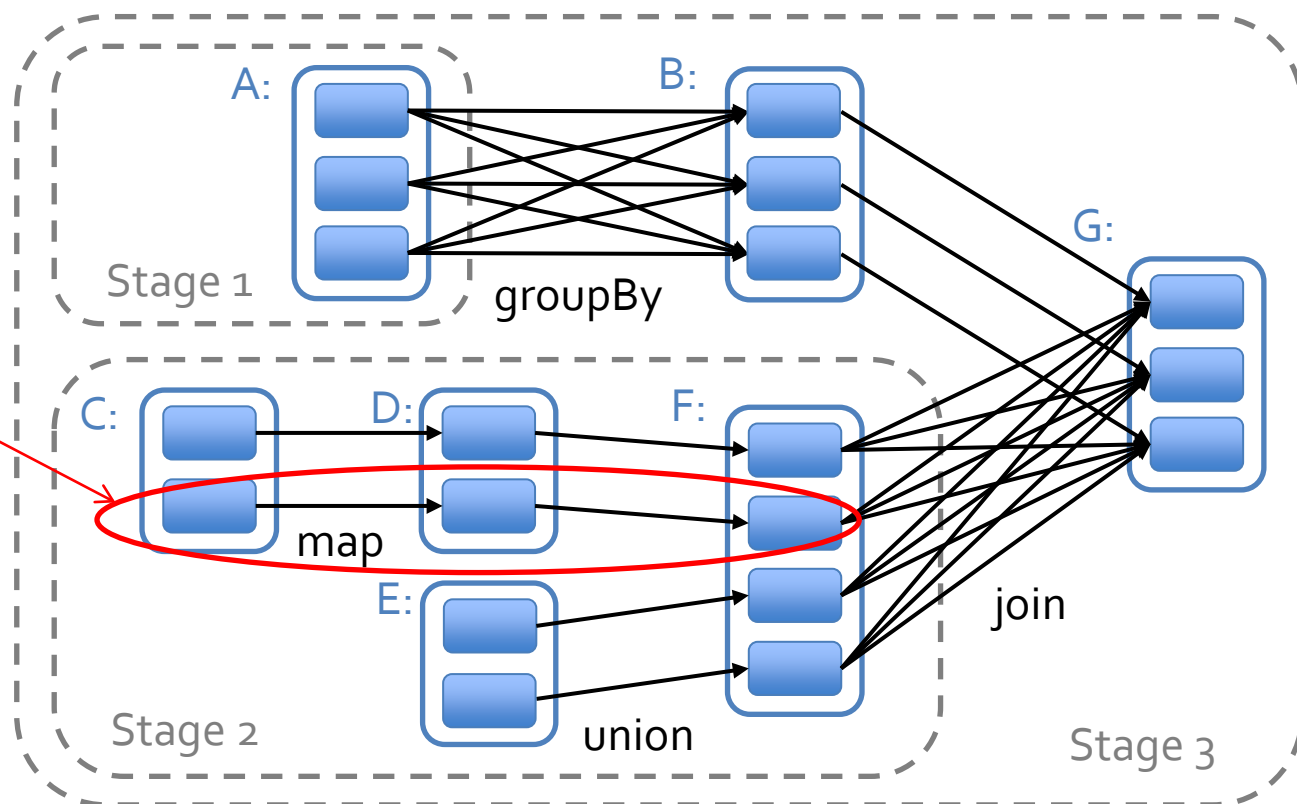
Spark Scheduler

- Whenever a user runs an action (e.g., count or save) on an RDD, the scheduler examines that RDD's lineage graph to build a DAG of stages to execute



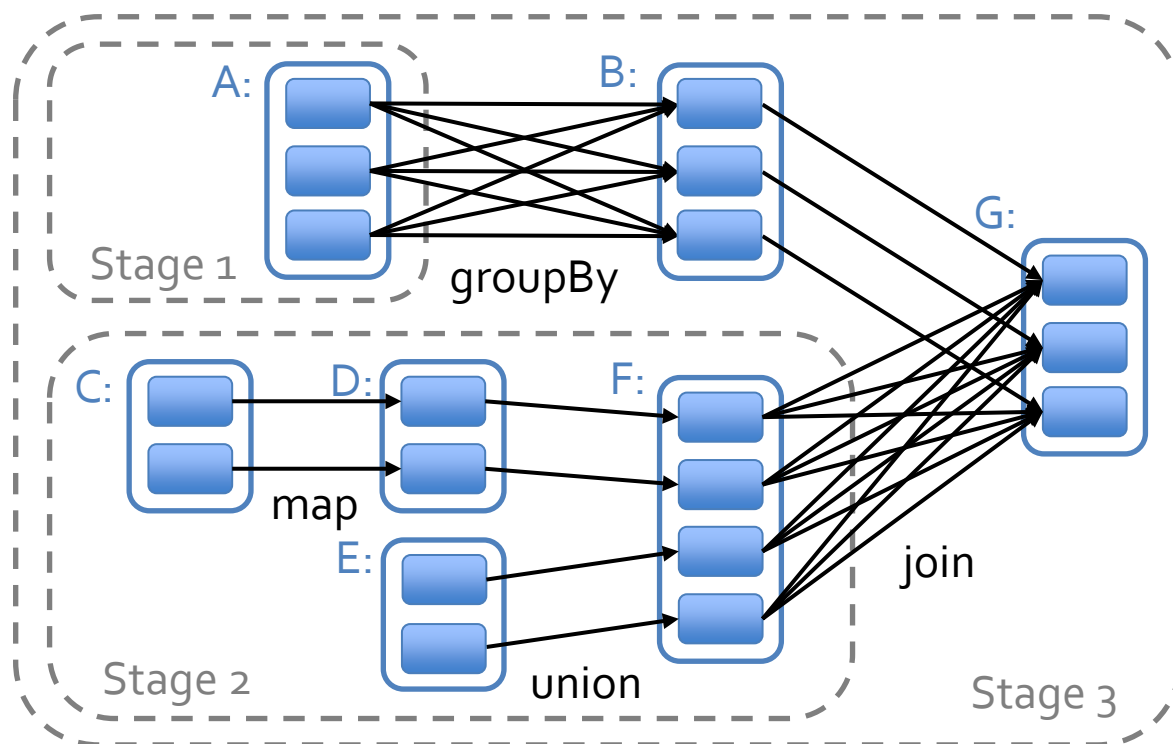
Spark Scheduler

- Each stage contains as many pipelined transformations with **narrow dependencies** as possible



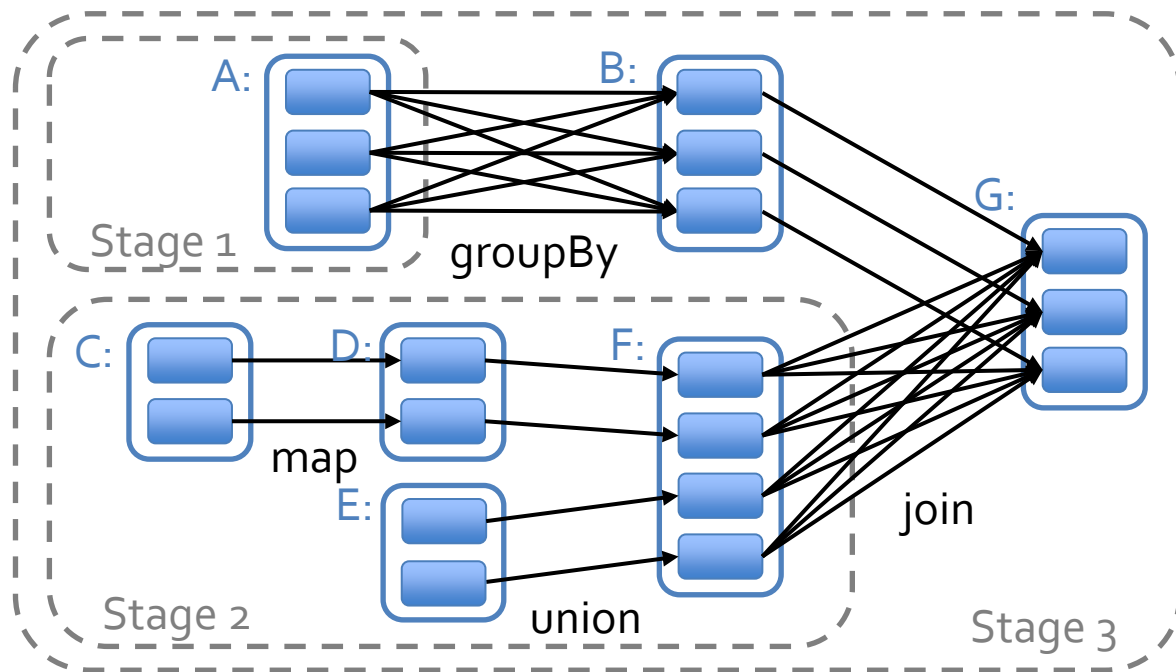
Spark Scheduler

- Our scheduler assigns tasks to machines based on data **locality of the cached data (not the file in disks)**



Spark Scheduler

- If a task fails, we re-run it on another node as long as its stage's parents are still available.
- We do not yet tolerate scheduler failures, though replicating the RDD lineage graph would be straightforward



Performance Comparison

- HadoopBinMem: A Hadoop deployment that **converts the input data into a low-overhead binary format** in the first iteration to eliminate text parsing in later ones, and **stores it in an in-memory HDFS instance**.
- Spark beats HadoopBinMem by 20X
 - Overhead of HDFS
 - Deserialization cost to convert binary records to usable in-memory Java objects

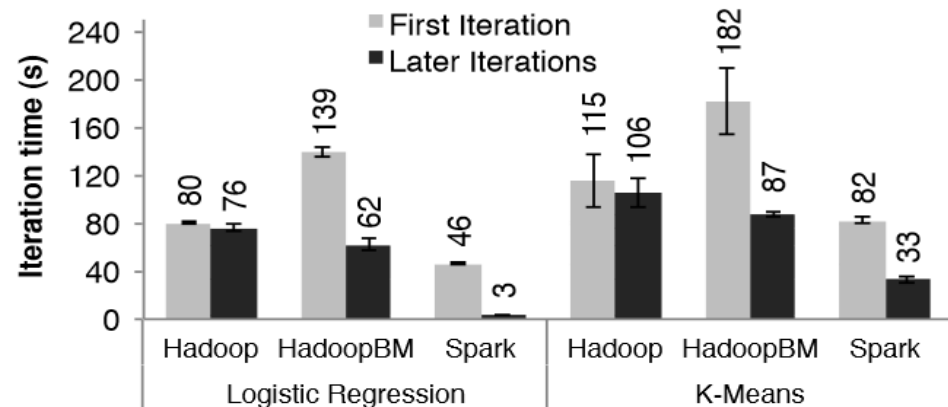


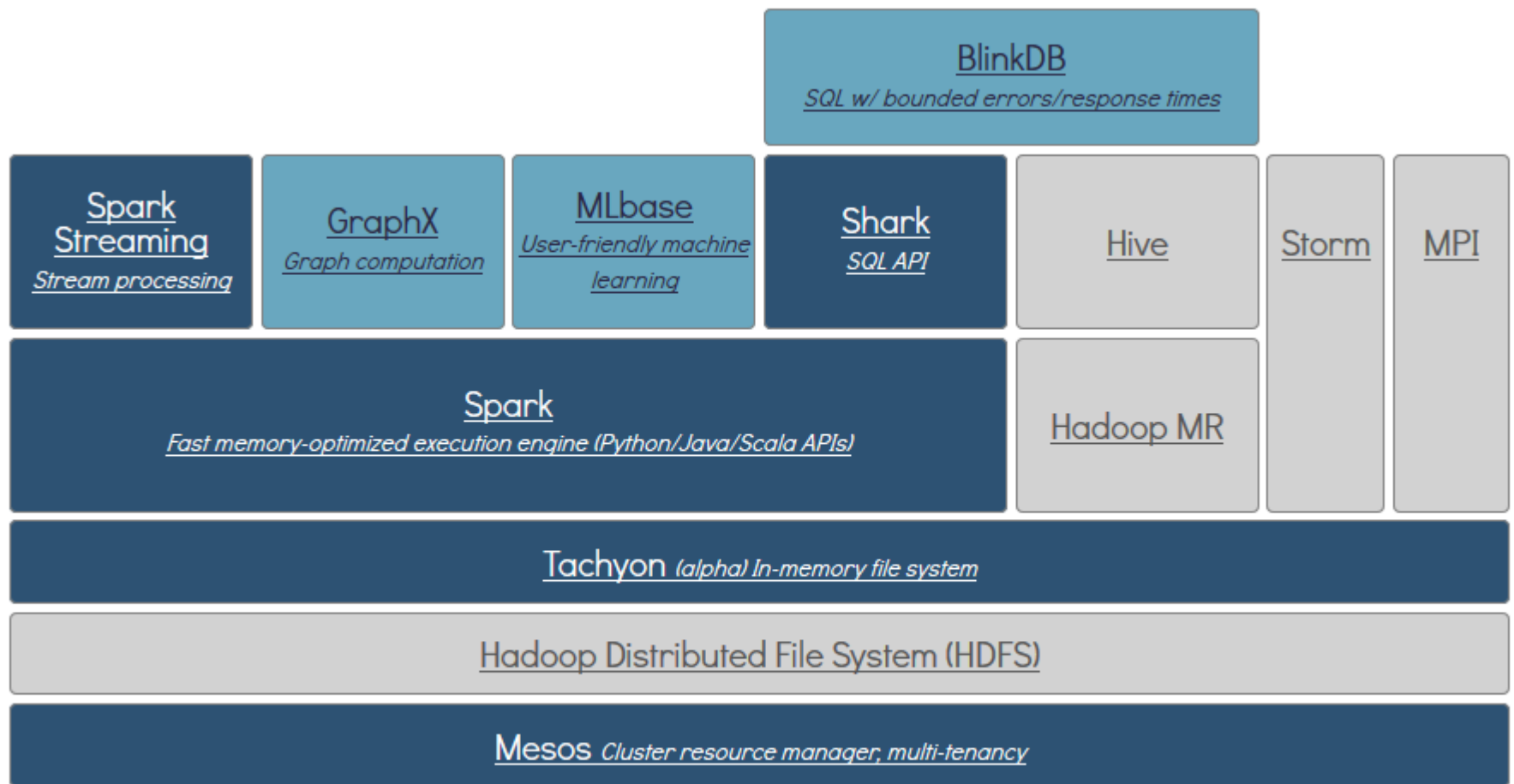
Figure 7: Duration of the first and later iterations in Hadoop, HadoopBinMem and Spark for logistic regression and k-means using 100 GB of data on a 100-node cluster.

Frameworks Built on Spark

- Pregel on Spark (Bagel)
 - Google message passing model for graph computation
 - 200 lines of code
- Hive on Spark (Shark)
 - 3000 lines of code
 - Compatible with Apache Hive
 - ML operators in Scala
- Spark Streaming
 - Small batch streaming



BDAS: *the Berkeley Data Analytics Stack*



■ Supported Release ■ In Development ■ Related External Project

Slides from Tathagata Das (TD)

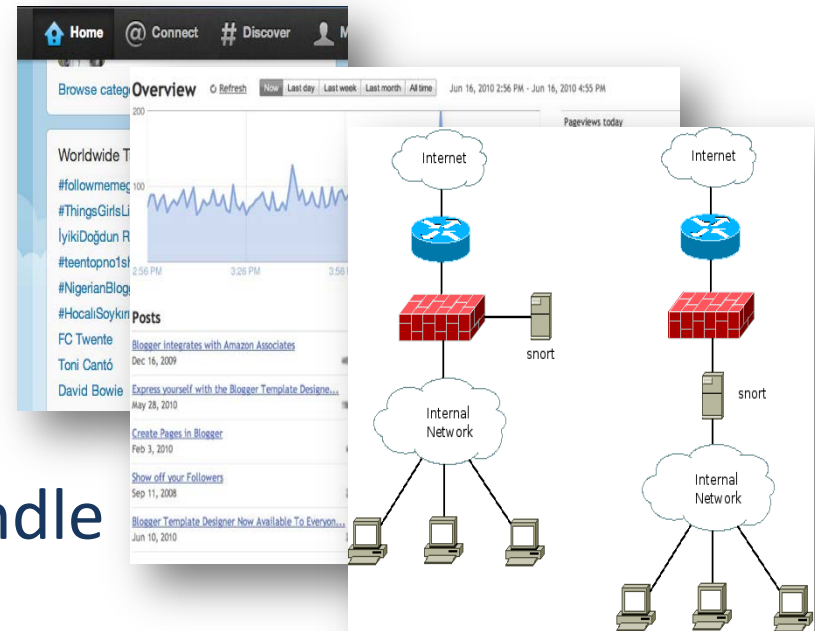
Core committer on Apache Spark

Lead developer on Spark Streaming

SPARK STREAMING

Motivation

- Many important applications must process large streams of live data and provide results in near-real-time
 - Social network trends
 - Website statistics
 - Intrusion detection systems
 - etc.
- Require large clusters to handle workloads
- Require latencies of few seconds



How to Process Big Streaming Data



Scales hundreds of nodes

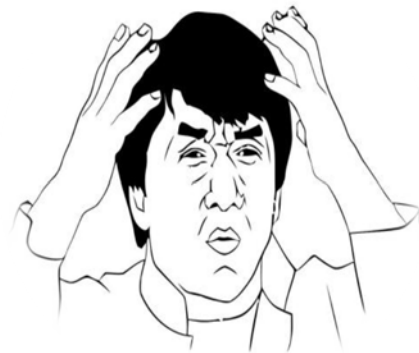
Achieves low latency

Efficiently recover from failures

Integrates with batch and interactive processing

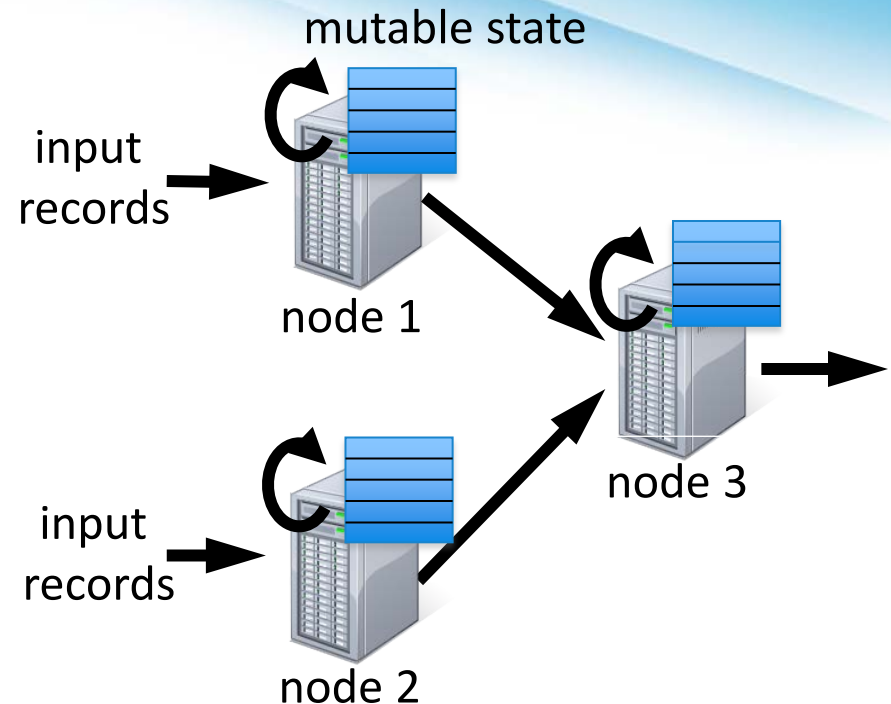
What people have been doing?

- Build two stacks --- one for batch, one for streaming
 - Often both process same data
- Existing frameworks cannot do both
 - Either, stream processing of 100s of MB/s with low latency
 - Or, batch processing of TB of data with big latency
- Extremely painful to maintain two stacks
 - Different programming models
 - Doubles implementation effort
 - Doubles operational cost



Stateful Stream Processing

- Traditional streaming systems have an event-driven **record-at-a-time** processing model
 - Each node has mutable state
 - For each record, update state & send new records
- State is lost if node dies!
- Making stateful stream processing be fault-tolerant is challenging



Existing Streaming Systems

- Storm
 - Replays record if not processed by a node
 - Processes each record *at least once*
 - May update mutable state twice!
 - Mutable state can be lost due to failure!
- Trident – Use transactions to update state
 - Processes each record *exactly once*
 - Per state transaction updates slow

What is Spark Streaming?

- Framework for large scale stream processing
 - Scales to 100s of nodes
 - Can achieve second scale latencies
 - Integrates with Spark's batch and interactive processing
 - Provides a simple batch-like API for implementing complex algorithm
 - Can absorb live data streams from Kafka, Flume, ZeroMQ, etc.



Discretized Stream Processing

Run a streaming computation as a **series of very small, deterministic batch jobs**

live data stream

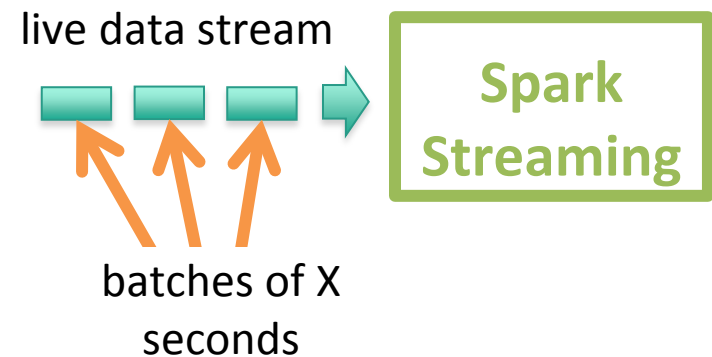


**Spark
Streaming**

Discretized Stream Processing

Run a streaming computation as a **series of very small, deterministic batch jobs**

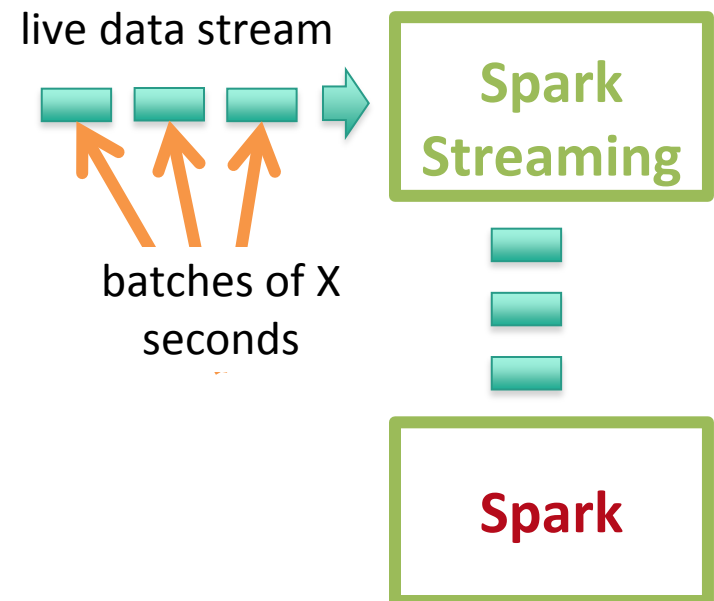
- Chop up the live stream into batches of X seconds



Discretized Stream Processing

Run a streaming computation as a **series of very small, deterministic batch jobs**

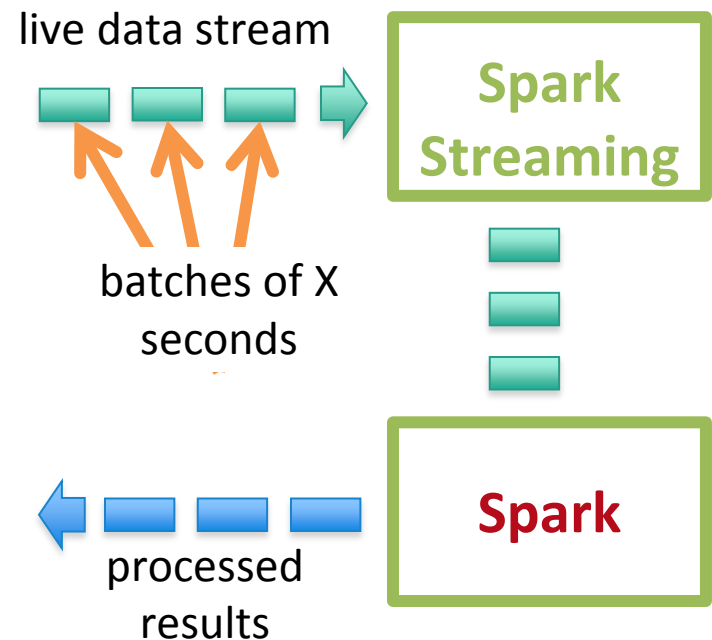
- Chop up the live stream into batches of X seconds
- Spark treats each batch of data as RDDs and processes them using RDD operations



Discretized Stream Processing

Run a streaming computation as a **series of very small, deterministic batch jobs**

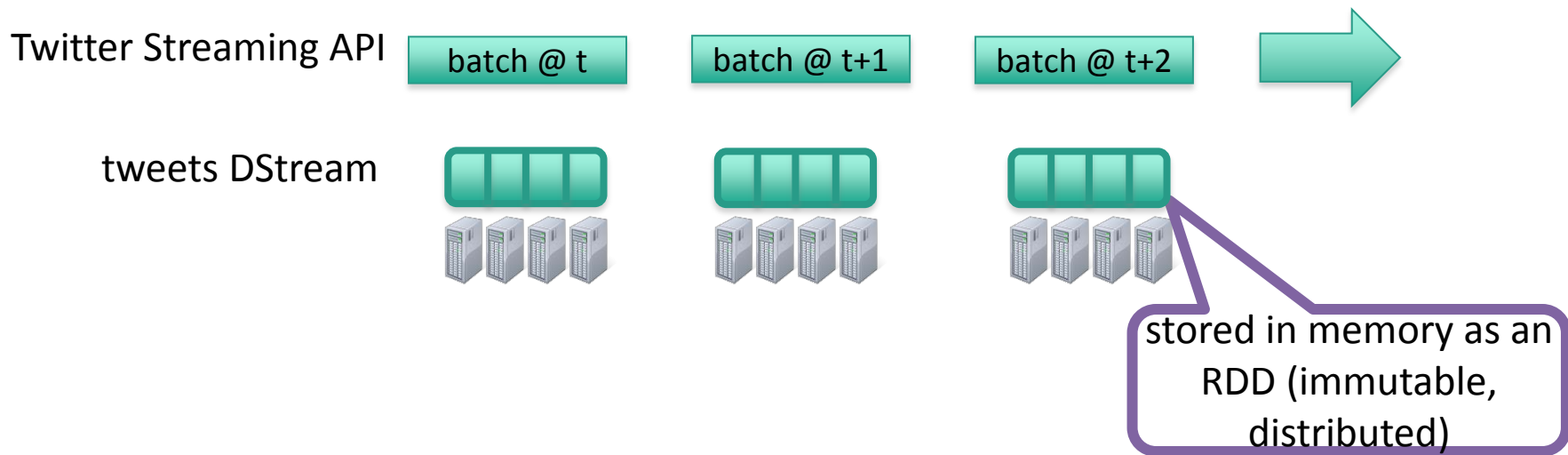
- Chop up the live stream into batches of X seconds
- Spark treats each batch of data as RDDs and processes them using RDD operations
- Finally, the processed results of the RDD operations are returned in batches



Example 1 – Get hashtags from Twitter

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
```

DStream: a sequence of RDD representing a stream of data

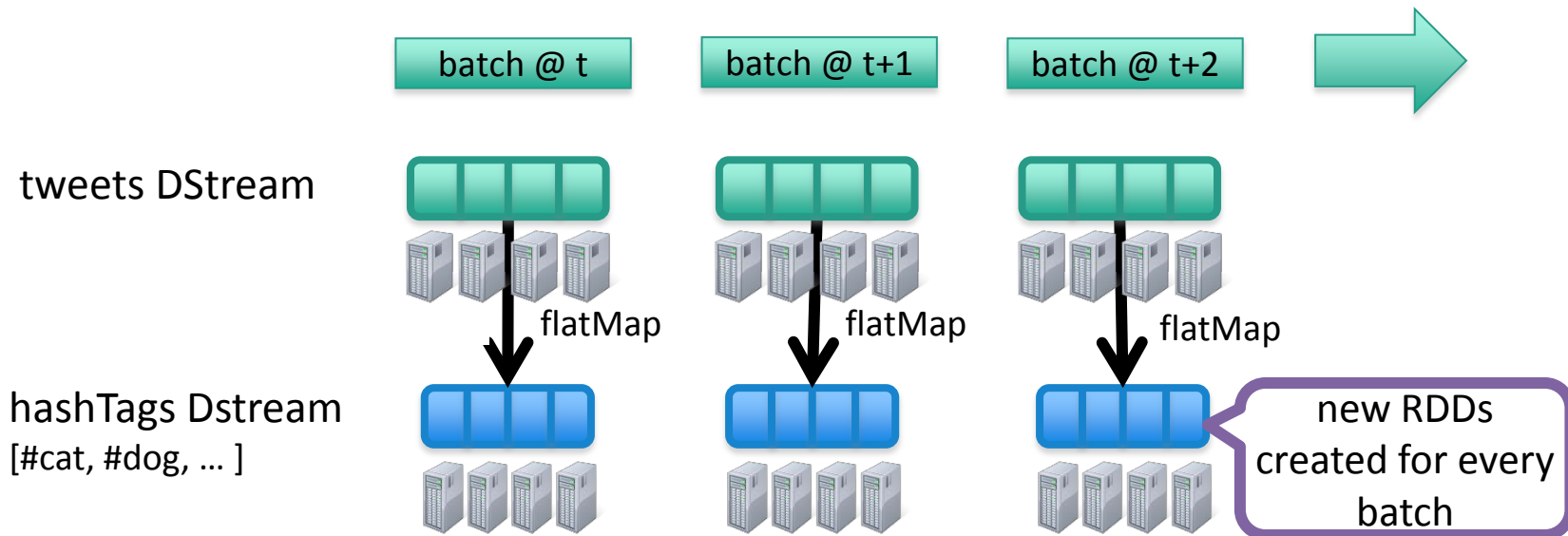


Example 1 – Get hashtags from Twitter

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)  
val hashTags = tweets.flatMap (status => getTags(status))
```

new DStream

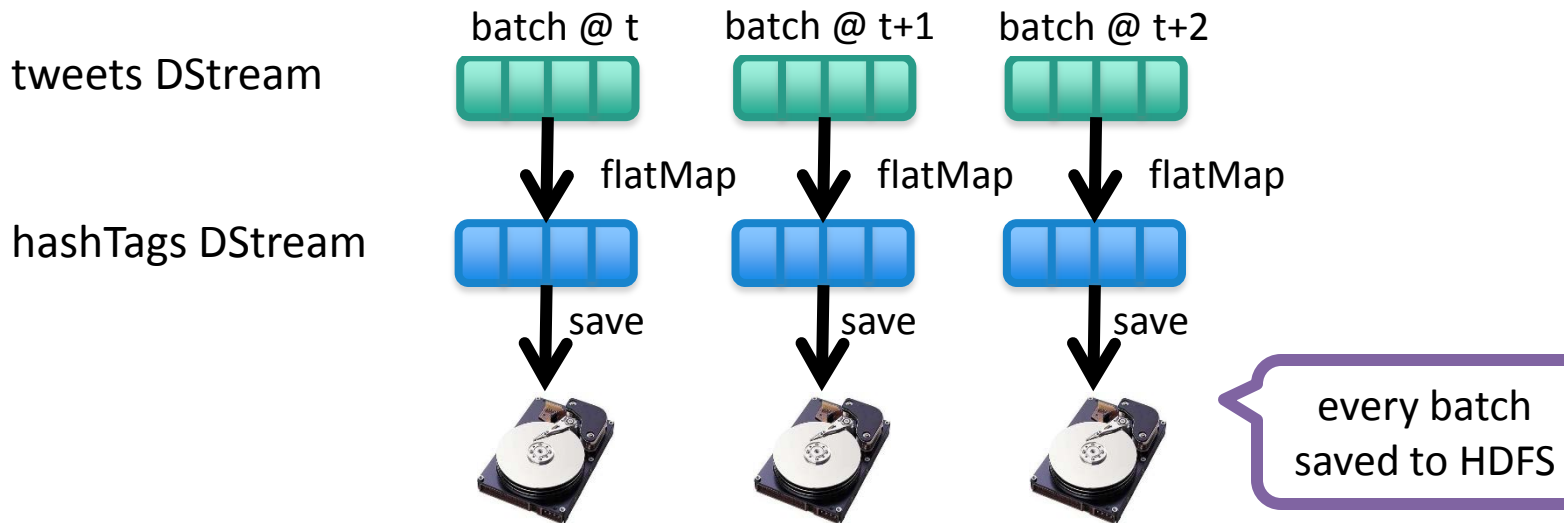
transformation: modify data in one Dstream to create another DStream



Example 1 – Get hashtags from Twitter

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)  
val hashTags = tweets.flatMap (status => getTags(status))  
hashTags.saveAsHadoopFiles("hdfs://...")
```

output operation: to push data to external storage




Java Example

Scala

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)  
val hashTags = tweets.flatMap (status => getTags(status))  
hashTags.saveAsHadoopFiles("hdfs://...")
```

Java

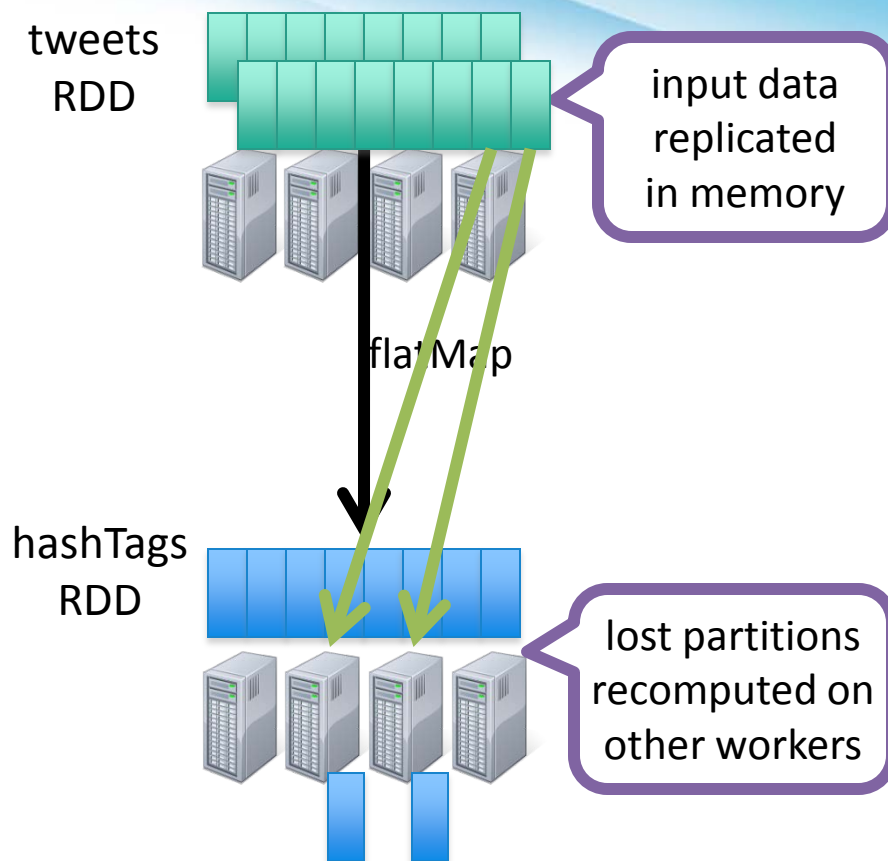
```
JavaDStream<Status> tweets = ssc.twitterStream(<Twitter username>,  
<Twitter password>)  
JavaDStream<String> hashTags = tweets.flatMap(new Function<...> { })  
hashTags.saveAsHadoopFiles("hdfs://...")
```



Function object to define the transformation

Fault-tolerance

- RDDs remember the sequence of operations that created it from the original fault-tolerant input data
- Batches of input data are replicated in memory of multiple worker nodes, therefore fault-tolerant
- Data lost due to worker failure, can be recomputed from input data

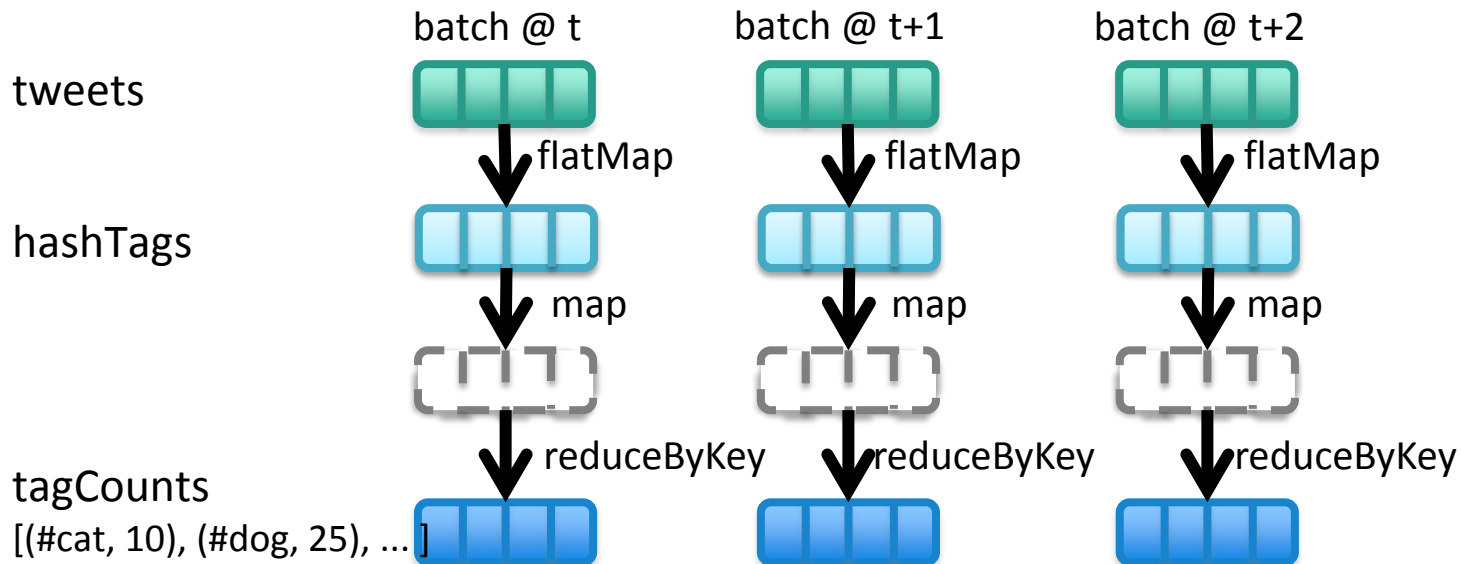


Key concepts

- **DStream** – sequence of RDDs representing a stream of data
 - Twitter, HDFS, Kafka, Flume, ZeroMQ, Akka Actor, TCP sockets
- **Transformations** – modify data from on DStream to another
 - Standard RDD operations – map, countByValue, reduce, join, ...
 - Stateful operations – window, countByValueAndWindow, ...
- **Output Operations** – send data to external entity
 - saveAsHadoopFiles – saves to HDFS
 - foreach – do anything with each batch of results

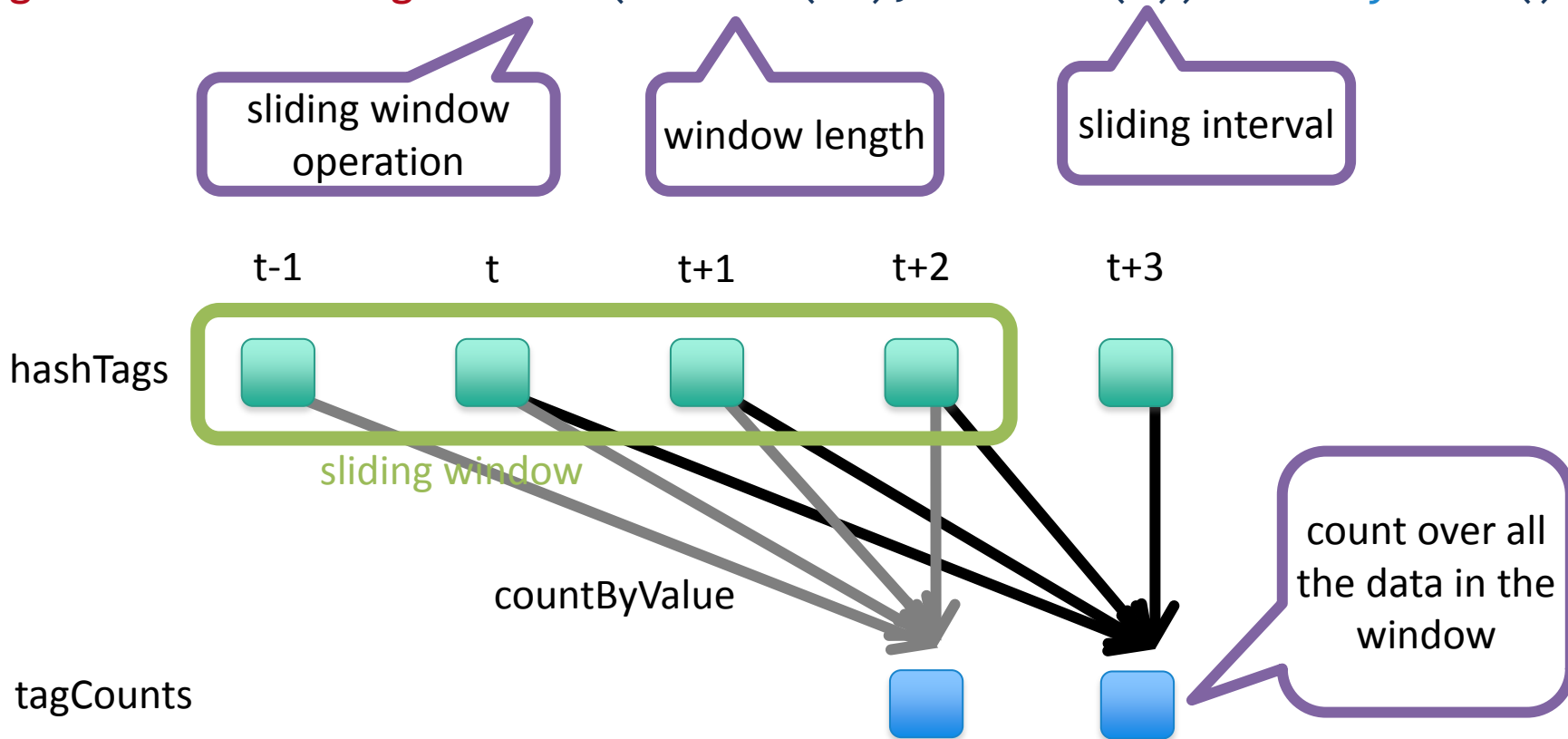
Example 2 – Count the hashtags

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)  
val hashTags = tweets.flatMap(status => getTags(status))  
val tagCounts = hashTags.countByValue()
```

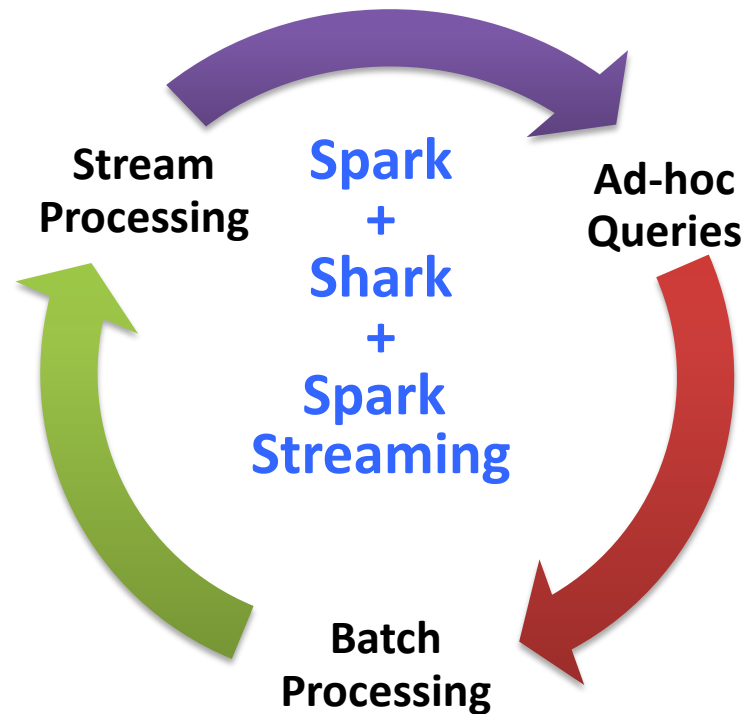


Example 3 – Count the hashtags over last 10 mins

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)  
val hashTags = tweets.flatMap (status => getTags(status))  
val tagCounts = hashTags.window(Minutes(10), Seconds(1)).countByValue()
```



Vision - one stack to rule them all



Spark program vs Spark Streaming program

Spark Streaming program on Twitter stream

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)  
val hashTags = tweets.flatMap (status => getTags(status))  
hashTags.saveAsHadoopFiles("hdfs://...")
```

Spark program on Twitter log file

```
val tweets = sc.hadoopFile("hdfs://...")  
val hashTags = tweets.flatMap (status => getTags(status))  
hashTags.saveAsHadoopFile("hdfs://...")
```

Vision - one stack to rule them all

- Explore data interactively using Spark Shell / PySpark to identify problems
- Use same code in Spark stand-alone programs to identify problems in production logs
- Use similar code in Spark Streaming to identify problems in live log streams

```
$ ./spark-shell
scala> val file = sc.hadoopFile("smallLogs")
...
scala> val filtered = file.filter(_.contains("ERROR"))
...
scala> val mapped = file.map(...)
```

```
object ProcessProductionData {
  def main(args: Array[String]) {
    val sc = new SparkContext(...)
    val file = sc.hadoopFile("productionLogs")
    val filtered =
      file.filter(_.contains("ERROR"))
    val mapped = file.map(...)
    ...
  }
}
```

```
object ProcessLiveStream {
  def main(args: Array[String]) {
    val sc = new StreamingContext(...)
    val stream = sc.kafkaStream(...)
    val filtered =
      file.filter(_.contains("ERROR"))
    val mapped = file.map(...)
    ...
  }
}
```

Alpha Release with Spark 0.7

- Integrated with Spark 0.7
 - Import **spark.streaming** to get all the functionality
- Both Java and Scala API
- Give it a spin!
 - Run locally or in a cluster

References

- Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. 2010. HaLoop: efficient iterative data processing on large clusters. *Proc. VLDB Endow.* 3, 1-2 (September 2010), 285-296.
- Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing (HotCloud'10)*. USENIX Association, Berkeley, CA, USA, 10-10.
- Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.* 41, 3 (March 2007), 59-72.
- http://www.slideshare.net/pramode_ce/introduction-to-functional-programming-with-scala

Backup Slides

Yingyi Bu, Bill Howe Magdalena Balazinska Michael D. Ernst
FROM UC Irvin & Univ. of Washington

HALOOP: EFFICIENT ITERATIVE DATA PROCESSING ON LARGE CLUSTERS

Motivation

- The MapReduce framework does not directly support these iterative data analysis applications
- Issues:
 - Invariant data must be re-loaded and re-processed at each iteration, wasting I/O, network bandwidth, and CPU resources
 - An extra termination detection job which cause overhead in terms of scheduling extra tasks, reading extra data from disk, and moving data across the network

Contributions

- A modified version of Hadoop for iterative jobs
 - New Programming Model and Architecture for Iterative Programs
 - Allow user to specify termination condition and invariant data
 - Enable Caching
 - Reducer input cache
 - Reducer output cache
 - Map input cache
 - Loop-Aware Task Scheduling
 - Maximize the effectiveness of cache

Programming Model

- Iterative computing that has the code construct

$$R_{i+1} = R_0 \cup (R_i \bowtie L)$$

- R_0 is an initial result ; L is a invariant relation
 - Program is terminated after a *fixpoint* is reached:

$$R_{i+1} = R_i \text{ or } R_{i+1} - R_i < \delta$$

- PageRank example

$$MR_1 \left\{ \begin{array}{l} T_1 = R_i \bowtie_{url=url_source} L \\ T_2 = \gamma_{url, rank, \frac{rank}{COUNT(url_dest)} \rightarrow new_rank} (T_1) \\ T_3 = T_2 \bowtie_{url=url_source} L \end{array} \right.$$

$$MR_2 \left\{ \begin{array}{l} R_{i+1} = \gamma_{url_dest \rightarrow url, SUM(new_rank) \rightarrow rank} (T_3) \end{array} \right.$$

url_source	url_dest
www.a.com	www.b.com
www.a.com	www.c.com
www.c.com	www.a.com
www.e.com	www.d.com
www.d.com	www.b.com
www.c.com	www.e.com
www.e.com	www.c.com
www.a.com	www.d.com

Linkage Table L

url	rank
www.a.com	1.0
www.b.com	1.0
www.c.com	1.0
www.d.com	1.0
www.e.com	1.0

Initial Rank Table R_0

url	rank
www.a.com	2.13
www.b.com	3.89
www.c.com	2.60
www.d.com	2.60
www.e.com	2.13

Rank Table R_3

New Functions

- ***AddMap*** and ***AddReduce*** express a loop body that consists of more than one MapReduce step. An integer argument indicates the order of the step.
- ***SetFixedPointThreshold*** sets a bound on the distance between one iteration and the next.
- ***ResultDistance*** function calculates the distance between two out value sets sharing the same out key.
- ***SetMaxNumOfIterations*** provides further control of the loop termination condition.

New Functions

- ***SetIterationInput*** associates an input source with a specific iteration. It will be the input for the “first” mapreduce in a loop body
- ***AddStepInput*** associates an additional input source with an intermediate map-reduce pair in the loop body. The output of preceding map-reduce pair is always in the input of the next mapreduce pair.
- ***AddInvariantTable*** specifies an input table (an HDFS file) that is **loop-invariant**. During job execution, HaLoop will cache this table on cluster nodes.

Architecture

- In HaLoop, a **user program specifies loop settings** and the **framework controls the loop execution**
- But in Hadoop, it is the **application's responsibility to control the loops**

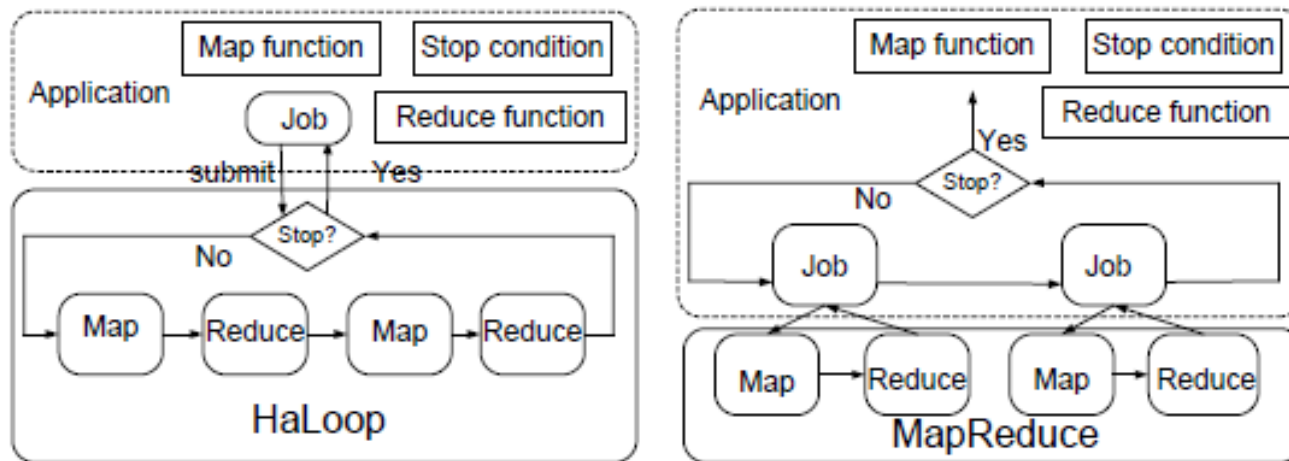


Figure 4: Boundary between an iterative application and the framework (HaLoop vs. Hadoop). HaLoop knows and controls the loop, while Hadoop only knows jobs with one map-reduce pair.

Descendant Query Example

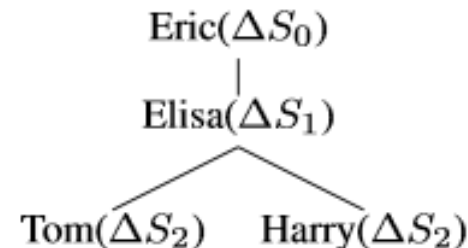
- Descendant query example: Given the social network relation, who is within two friend-hops from Eric?
 - Loop body contains 2 MapReduce jobs
 - Friend table is invariant

Friend table

name1	name2
Tom	Bob
Tom	Alice
Elisa	Tom
Elisa	Harry
Sherry	Todd
Eric	Elisa
Todd	John
Robin	Edward

Loop body

$$\begin{aligned}
 MR_1 & \left\{ \begin{aligned} T_1 &= \Delta S_i \bowtie_{\Delta S_i.name2=F.name1} F \\ T_2 &= \pi_{\Delta S_i.name1, F.name2}(T_1) \end{aligned} \right. \quad \rightarrow \text{Find friend in next hop} \\
 MR_2 & \left\{ \begin{aligned} T_3 &= \bigcup_{0 \leq j \leq (i-1)} \Delta S_j \\ \Delta S_{i+1} &= \delta(T_2 - T_3) \end{aligned} \right. \quad \rightarrow \text{Aggregate results from each iteration}
 \end{aligned}$$



Result generating tree

name1	name2
Eric	Elisa
Eric	Tom
Eric	Harry

Final Result Table

Descendant Query Example

- Iteration1; Step1: extend neighbor by one hop using JOIN**

IterationInput

Input: int iteration

```

1: if iteration==1 then
2:   return  $F \cup \Delta S_0$ ;
3: else
4:   return  $\Delta S_{\text{iteration}-1}$ 
5: end if
    
```

Map_Join

Input: Key k, Value v, int iteration

```

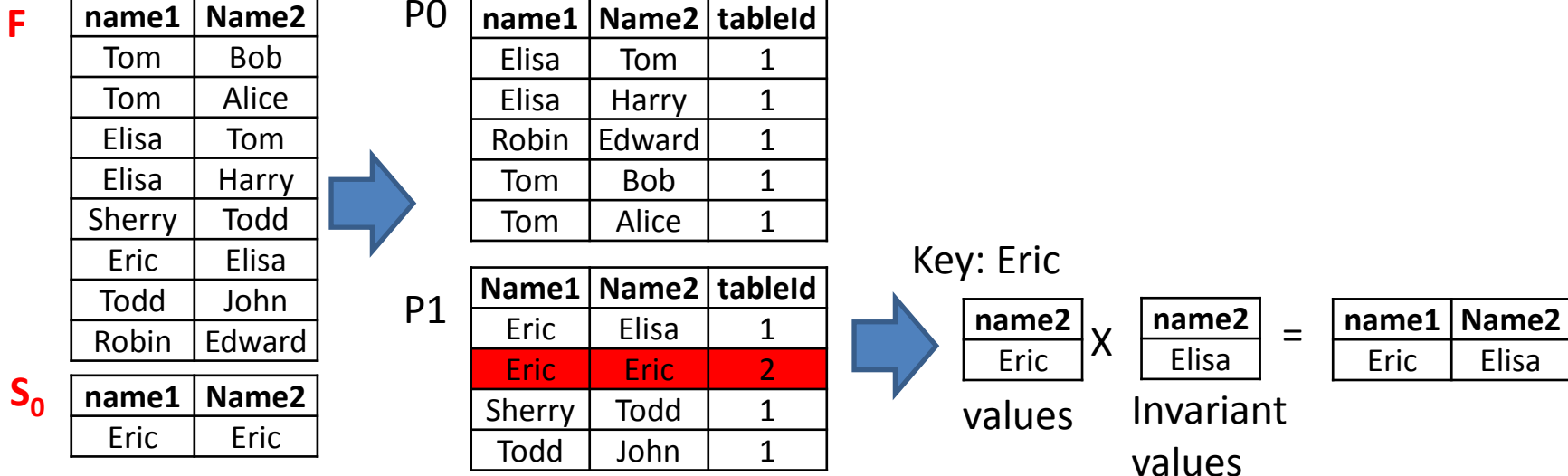
1: if v from F then
2:   Output(v.name1, v.name2, #1);
3: else
4:   Output(v.name2, v.name1, #2);
5: end if
    
```

Reduce_Join

Input: Key key, Set values, Set invariantValues, int iteration

```

1: Output(Product(values, invariantValues));
    
```



Input from files

Input to reducers

Output from reducers

Descendant Query Example

- Iteration1; Step2 → output distinct friend list

Map_Distinct

Input: Key k, Value v, int iteration

1: Output(v.name1, v.name2, iteration);

Reduce_Distinct

Input: Key key, Set values, int iteration

```
1: for name in values do
2:   if (name.iteration < iteration) then
3:     set_old.add(name);
4:   else set_new.add(name);
5: end for
6: Output(Product(key, Distinct(set_new-set_old)));
```

Output from
step1 reducer

name1	Name2
Eric	Elisa



name1	Name2	Iteration
Eric	Elisa	1



Key: Eric

set_new:

name1	Name2
Eric	Elisa

set_old: none

Output: **S₁**

name1	Name2
Eric	Elisa

Input to mappers

Input to reducers

Output from reducers

Descendant Query Example

- Iteration2; Step1: extend neighbor by one hop using JOIN**

IterationInput

Input: int iteration

```

1: if iteration==1 then
2:   return  $F \cup \Delta S_0$ ;
3: else
4:   return  $\Delta S_{\text{iteration}-1}$ 
5: end if
    
```

F

name1	Name2
Tom	Bob
Tom	Alice
Elisa	Tom
Elisa	Harry
Sherry	Todd
Eric	Elisa
Todd	John
Robin	Edward

S₁

name1	Name2
Eric	Elisa

Map_Join

Input: Key k, Value v, int iteration

```

1: if v from F then
2:   Output(v.name1, v.name2, #1);
3: else
4:   Output(v.name2, v.name1, #2);
5: end if
    
```

P0

name1	Name2	tableId
Elisa	Tom	1
Elisa	Harry	1
Elisa	Eric	2
Robin	Edward	1
Tom	Bob	1
Tom	Alice	1

P1

Name1	Name2	tableId
Eric	Elisa	1
Sherry	Todd	1
Todd	John	1

Reduce_Join

Input: Key key, Set values, Set invariantValues, int iteration

```

1: Output(Product(values, invariantValues));
    
```

Key: Elisa

name2
Eric

x

name2
Tom
Harry

=

name1	Name2
Eric	Tom
Eric	Harry

values

Invariant
Values
(read from
local cache)

= cached data

Descendant Query Example

- Iteration2; Step2: output distinct friend list**

Map_Distinct

Input: Key k, Value v, int iteration

1: Output(v.name1, v.name2, iteration);

StepInput

Input: int step, int iteration

1: if step==2 then
 2: return $\bigcup_{0 \leq j \leq (\text{iteration}-1)} \Delta S_j$
 3: end if

Output from
step1 reducer

name1	Name2
Eric	Tom
Eric	Harry



name1	Name2	Iteration
Eric	Tim	2
Eric	Harry	2



S₁

name1	Name2	Iteration
Eric	Elisa	1

Reduce_Distinct

Input: Key key, Set values, int iteration

1: for name in values do
 2: if (name.iteration < iteration) then
 3: set_old.add(name);
 4: else set_new.add(name);
 5: end for
 6: Output(Product(key, Distinct(set_new-set_old)));

set_new:

name1	Name2
Eric	Tim
Eric	Harry

set_old:

name1	Name2
Eric	Elisa

Output :

S₂

name1	Name2
Eric	Tim
Eric	Harry
Eric	Elisa

Descendant Query Example

Main

```
1: Job job = new Job();
2: job.AddMap(Map_Join, 1);
3: job.AddReduce(Reduce_Join, 1);
4: job.AddMap(Map_Distinct, 2);
5: job.AddReduce(Reduce_Distinct, 2);
6: job.SetDistanceMeasure(ResultDistance);
7: job.SetInput(IterationInput);
8: job.AddInvariantTable(#1);
9: job.SetFixedPointThreshold(1);
10: job.SetMaxNumOfIterations(2);
11: job.SetReducerInputCache(true);
12: job.AddStepInput(StepInput);
13: job.Submit();
```

Reducer Input Cache

- Purpose:
 - if a table is specified to be loop-invariant, the data in the table is cached across all reducers.
- Benefit in later iterations:
 - File does not to be re-loaded from FS
 - Data can be read directly from local disk for reducer
 - Invariant data does not need to be reprocessed or transferred by mapper

Reducer Input Cache

- Descendant query example:
 - Friend table (F) is invariant
 - All friend relations are hashed to 2 reducers
 - Reducer collects all relations, and produce join results
 - All data with tableID #1 will be cached at reducer node
 - M1, M2 don't need to reload/reprocess/transfer data in table F

Mapper Input

name1	name2
Tom	Bob
Tom	Alice
Elisa	Tom
Elisa	Harry

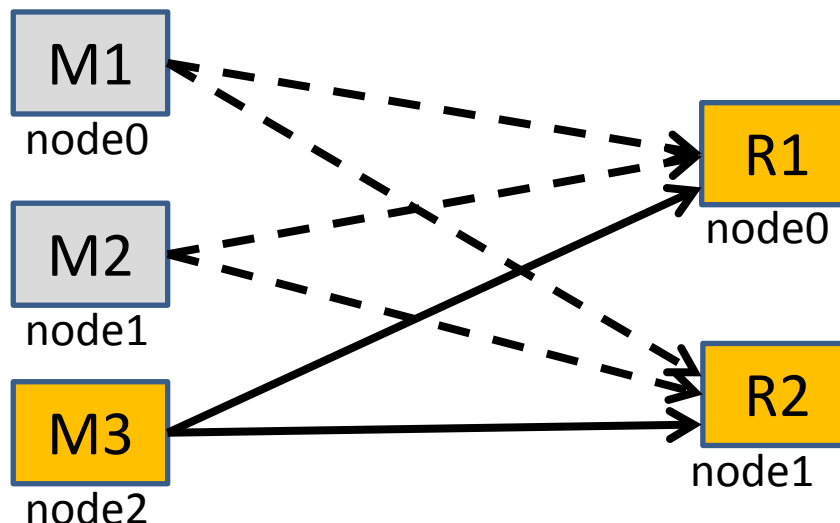
(a) F -split0

name1	name2
Sherry	Todd
Eric	Elisa
Todd	John
Robin	Edward

(b) F -split1

name1	name2
Eric	Eric

(c) ΔS_0 -split0



Reducer Input

name1	name2	table ID
Elisa	Tom	#1
Elisa	Harry	#1
Robin	Edward	#1
Tom	Bob	#1
Tom	Alice	#1

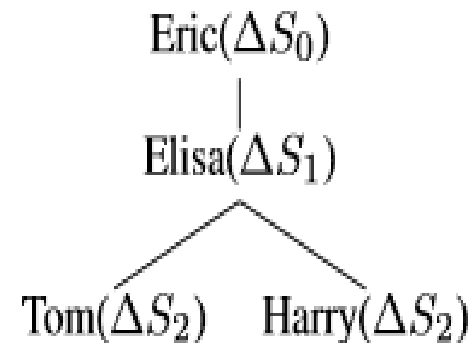
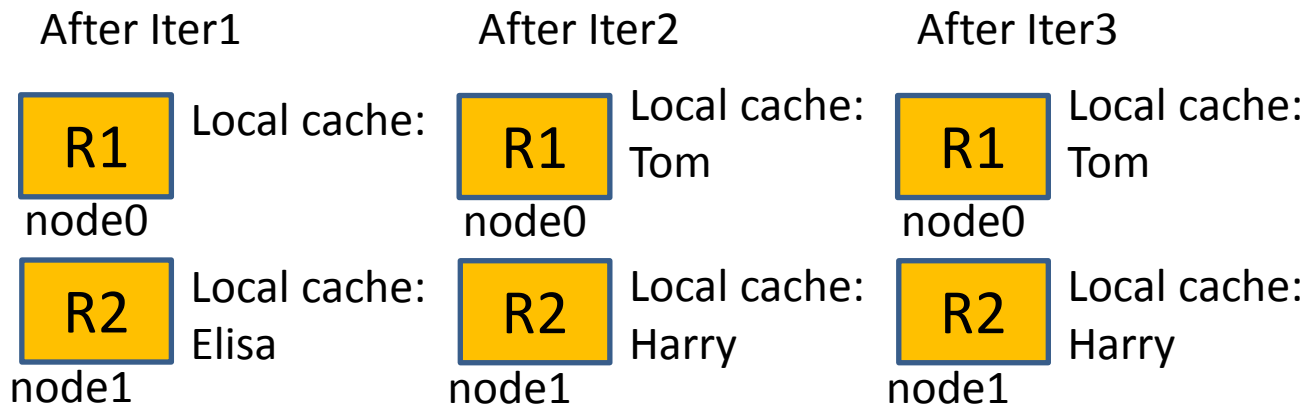
(a) partition 0

name1	name2	table ID
Eric	Elisa	#1
Eric	Eric	#2
Sherry	Todd	#1
Todd	John	#1

(b) partition 1

Reducer Output Cache

- Purpose:
 - Stores the most recent local output on each reducer.
- Benefit:
 - Reducers evaluate the termination condition by comparing to their **local cache file** in parallel
 - But, output with the same key must be produced by the same reducer in each iteration (e.g. tom must output by R1)
- Descendant query example:



Mapper Input Cache

- Purpose:
 - In the first iteration, if a mapper performs a non-local read on an input split, the split will be cached in the local disk of the mapper's physical node.
- Benefit:
 - Avoid non-local data reads in mappers during non-initial iterations
 - But mapper input cache shows only limited improvement because the rate of data-local mappers is already around 70%–95% in real system

Loop-Aware Task Scheduling

- Why need loop-aware task scheduling?
 - Cache is **locally** stored by mappers or reducers from the previous iteration
 - Nodes do not aware cache content from other nodes.
 - To re-use the cache, the task responding for the same key in the next iteration must be scheduled to the same node
- Principal of loop-aware scheduling
 - The scheduler must record the task scheduling location after each iteration, and the partition of invariant table assigned to each task
 - Scheduler attempts to schedule the task responsible for the same partition data to the same physical node

Loop-Aware Task Scheduling

Task Scheduling

Input: Node node

// The current iteration's schedule; initially empty

Global variable: Map<Node, List<Partition>> current

// The previous iteration's schedule

Global variable: Map<Node, List<Partition>> previous

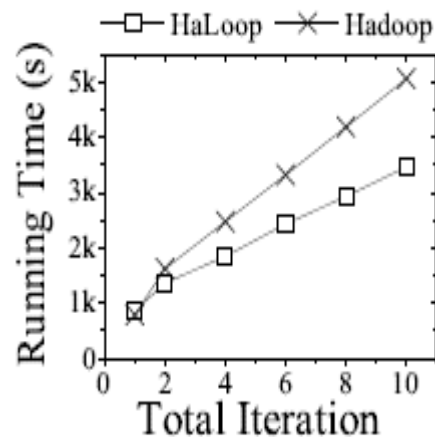
```
1: if iteration == 0 then
2:   Partition part = hadoopSchedule(node);
3:   current.get(node).add(part);
4: else
5:   if node.hasFullLoad() then
6:     Node substitution = findNearestIdleNode(node);
7:     previous.get(substitution).addAll(previous.remove(node));
8:     return;
9:   end if
10:  if previous.get(node).size() > 0 then
11:    Partition part = previous.get(node).get(0);
12:    schedule(part, node);
13:    current.get(node).add(part);
14:    previous.remove(part);
15:  end if
16: end if
```

The partitions that are cached on each node in the current and previous iteration

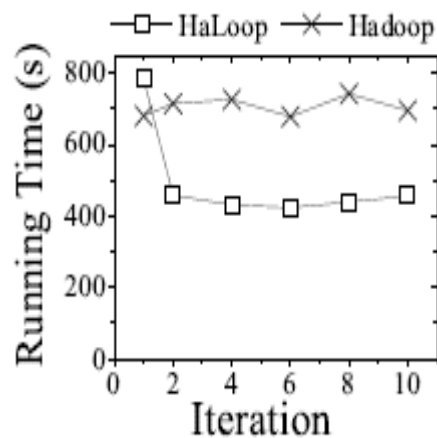
Use default scheduling in first iteration

Schedule a task that has been placed on the same node

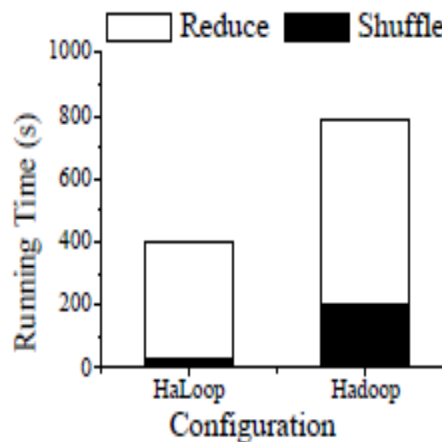
Performance Comparison for PageRank



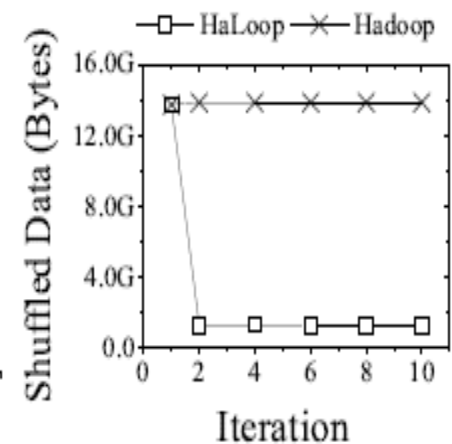
(a) Overall Performance



(b) Join Step



(c) Cost Distribution



(d) Shuffled Bytes

- Improvement increases with more iterations
- Less shuffle data in later iterations due to reducer input cache

Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly.
2007. Dryad: distributed data-parallel programs from sequential building
blocks. *SIGOPS Oper. Syst. Rev.* 41, 3 (March 2007), 59-72.

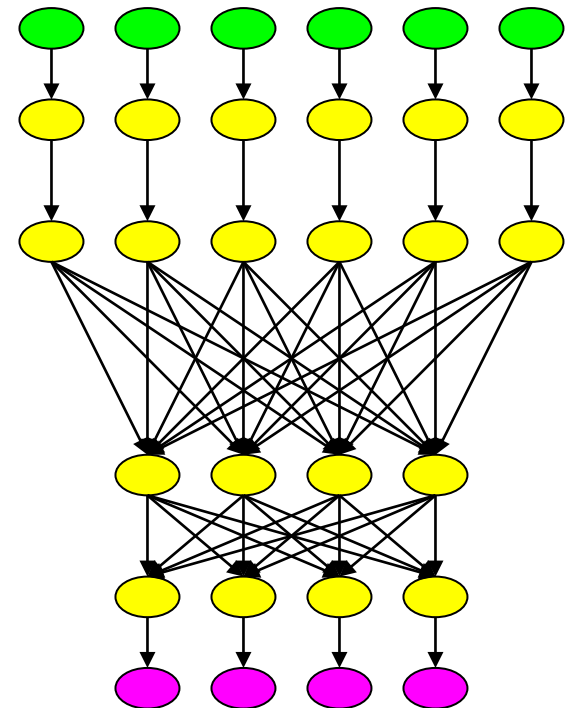
From Microsoft

DRYAD: DISTRIBUTED DATA-PARALLEL PROGRAMS FROM SEQUENTIAL BUILDING BLOCKS

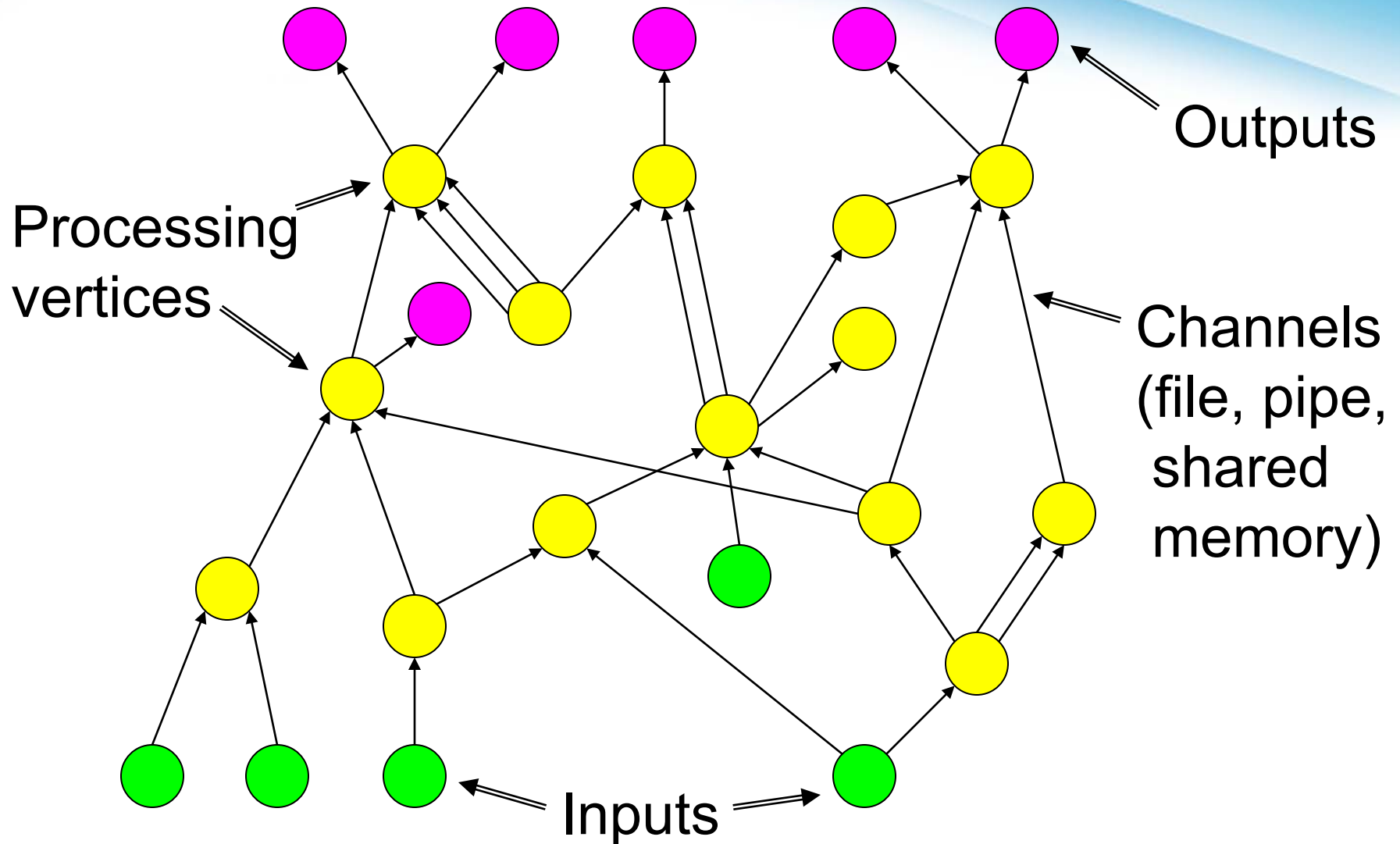
Motivation & Goals

- Many programs can be represented as a distributed execution graph
- Dryad will run them for you

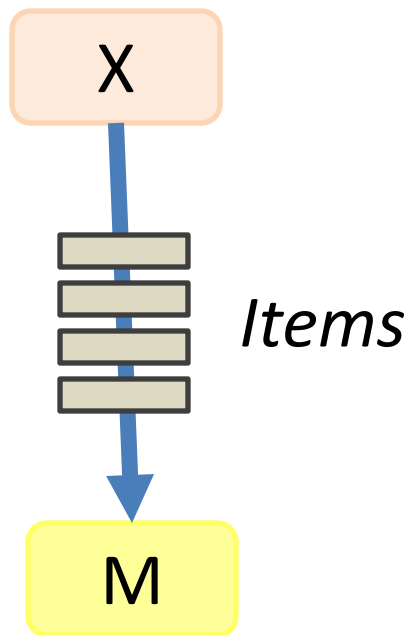
```
var logentries =  
  from line in logs  
  where !line.StartsWith("#")  
  select new LogEntry(line);  
var user =  
  from access in logentries  
  where access.user.EndsWith("@\ulfar")  
  select access;  
var accesses =  
  from access in user  
  group access by access.page into pages  
  select new UserPageCount("\ulfar", pages.Key, pages.Count());  
var htmAccesses =  
  from access in accesses  
  where access.page.EndsWith(".htm")  
  orderby access.count descending  
  select access;
```



Job = Directed Acyclic Graph



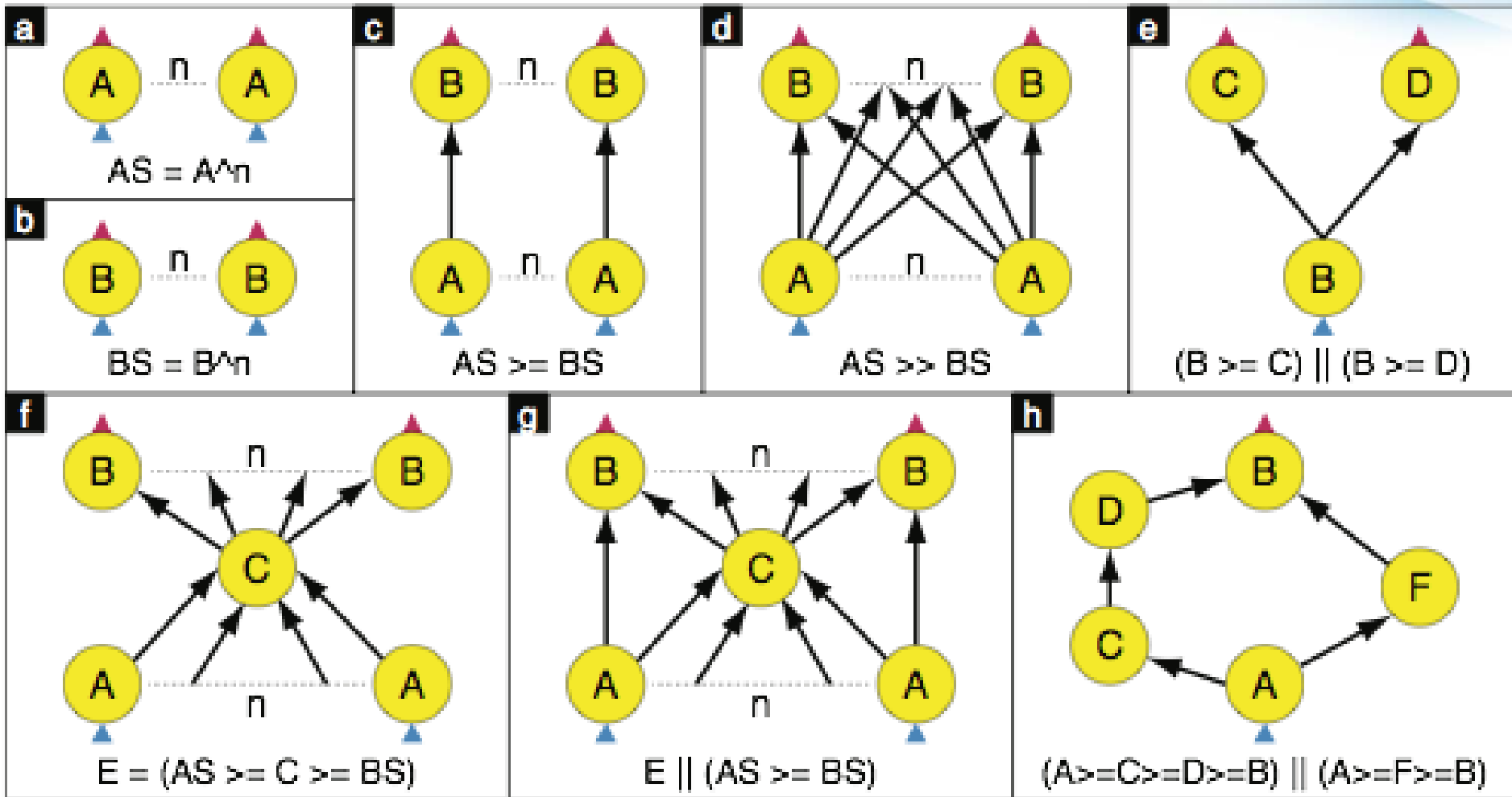
Channels



Finite Streams of items

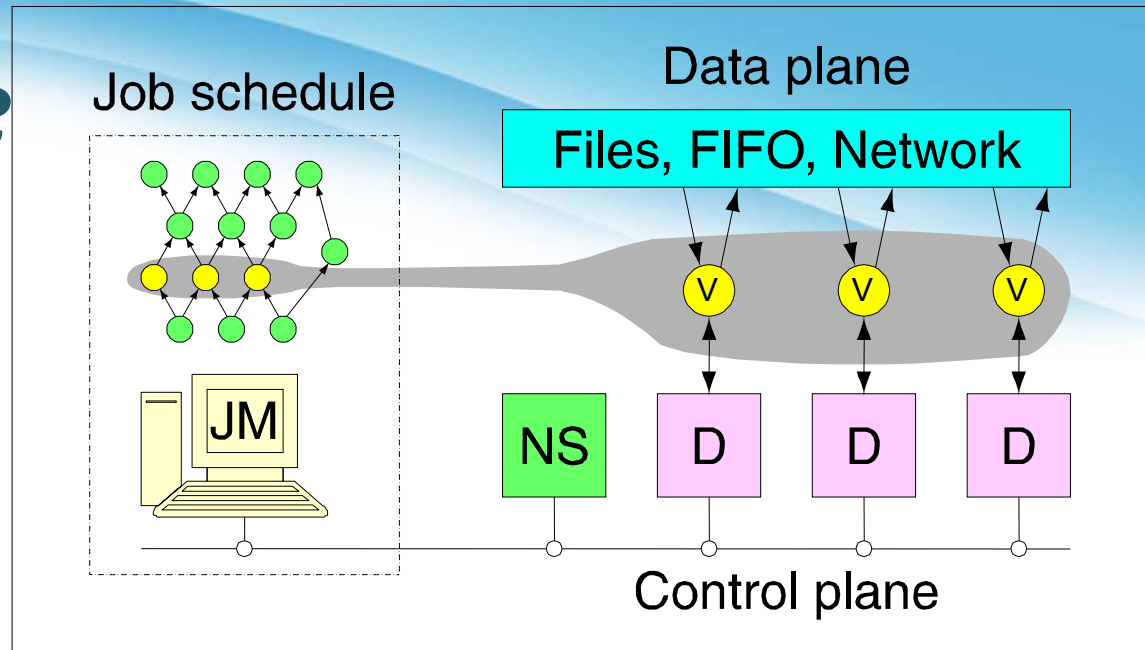
- distributed filesystem files
(persistent)
- SMB/NTFS files
(temporary)
- TCP pipes
(inter-machine)
- memory FIFOs
(intra-machine)

Operators of Graph Descr. Language



Architecture

- Services
 - Name server
 - Daemon
- Job Manager
 - Centralized coordinating process
 - User application to construct graph
 - Linked with Dryad libraries for scheduling vertices
- Vertex executable
 - Dryad libraries to communicate with JM
 - User application sees channels in/out
 - Arbitrary application code, can use local FS

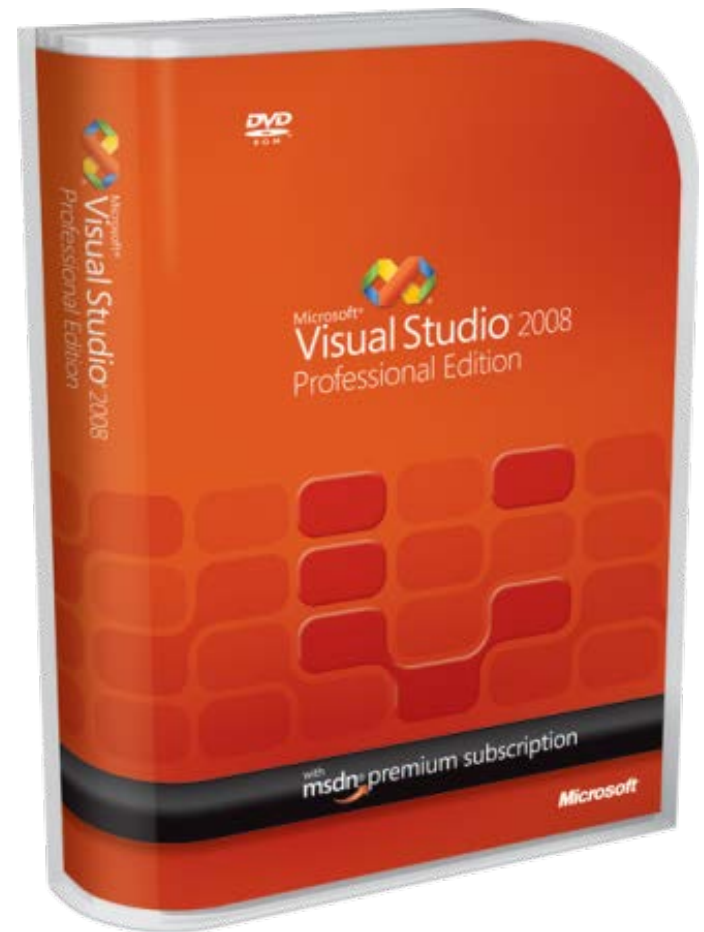


Scheduler state machine

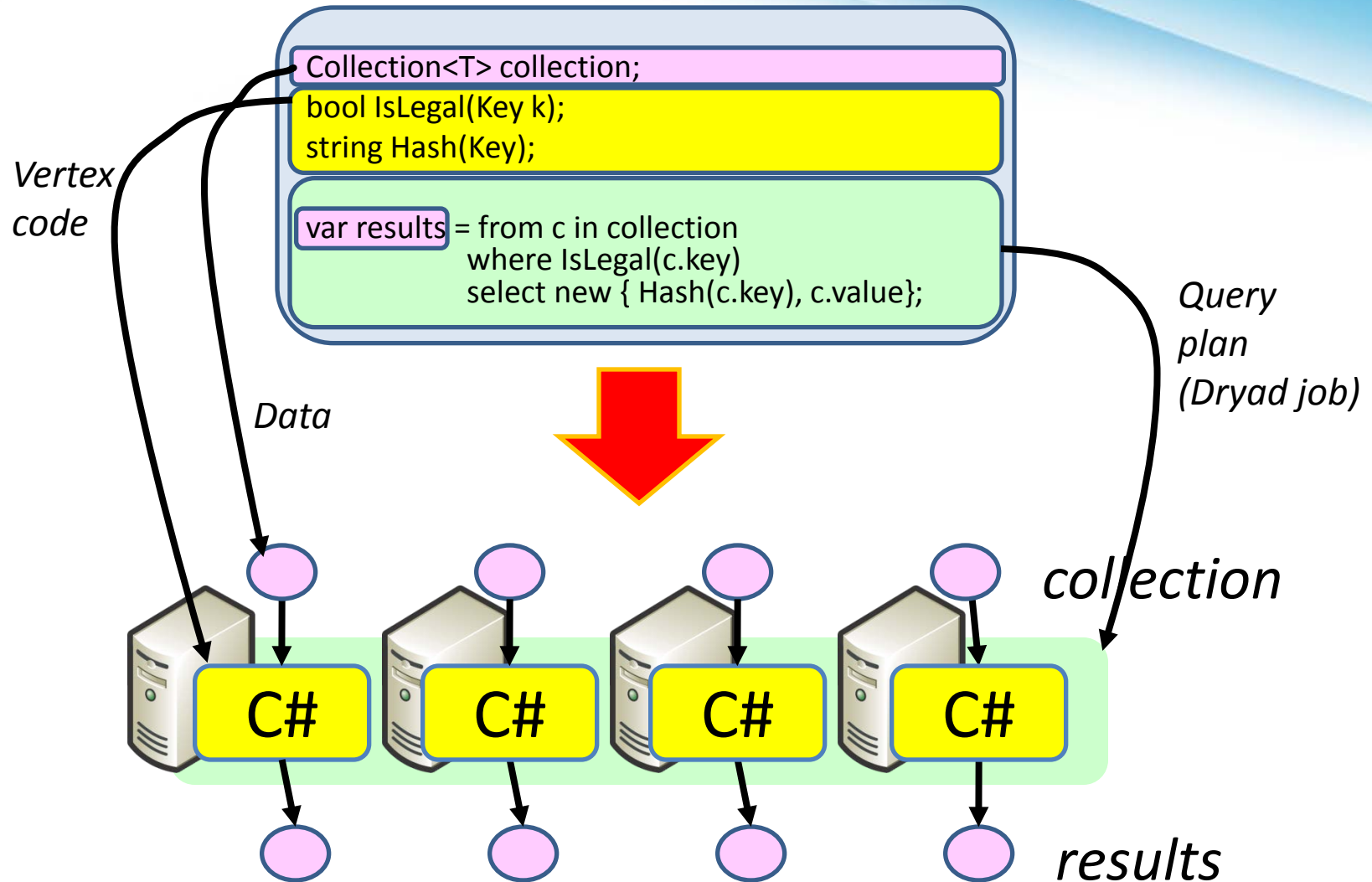
- Scheduling is independent of semantics
 - Vertex can run anywhere once all its inputs are ready
 - Constraints/hints **place it near its inputs**
 - Fault tolerance
 - If A fails, run it again
 - If A's inputs are gone, run upstream vertices again (recursively)
 - If A is slow, run another copy elsewhere and use output from whichever finishes first (**speculative execution**)

DryadLINQ

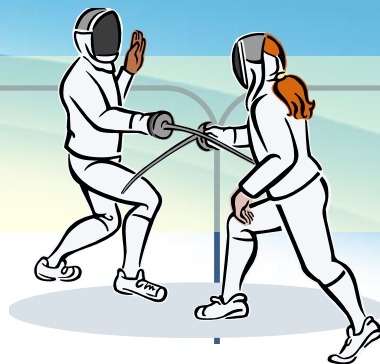
- Declarative programming
- Integration with Visual Studio
- Integration with .Net
- Type safety
- Automatic serialization
- Job graph optimizations
 - static
 - dynamic
- Conciseness



DryadLINQ = LINQ + Dryad



Dryad



Map-Reduce

- Many similarities

- Execution layer
- Job = arbitrary DAG
- Plug-in policies
- Program=graph gen.

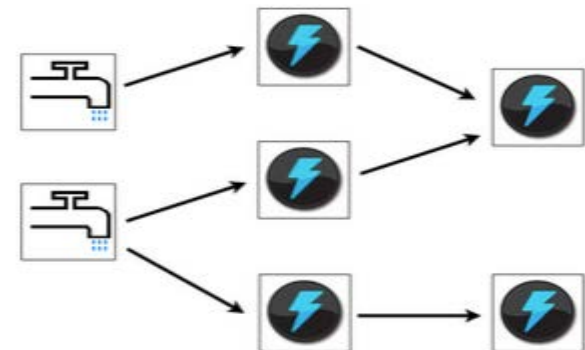
- Complex (↑features)
- New (< 2 years)
- Still growing
- Internal

- Exe + app. model
- Map+sort+reduce
- Few policies
- Program=map+reduce

- Simple
- Mature (> 4 years)
- Widely deployed
- Hadoop

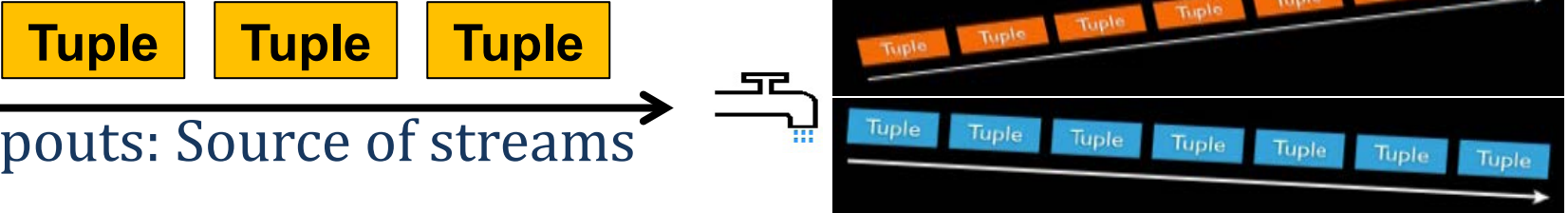
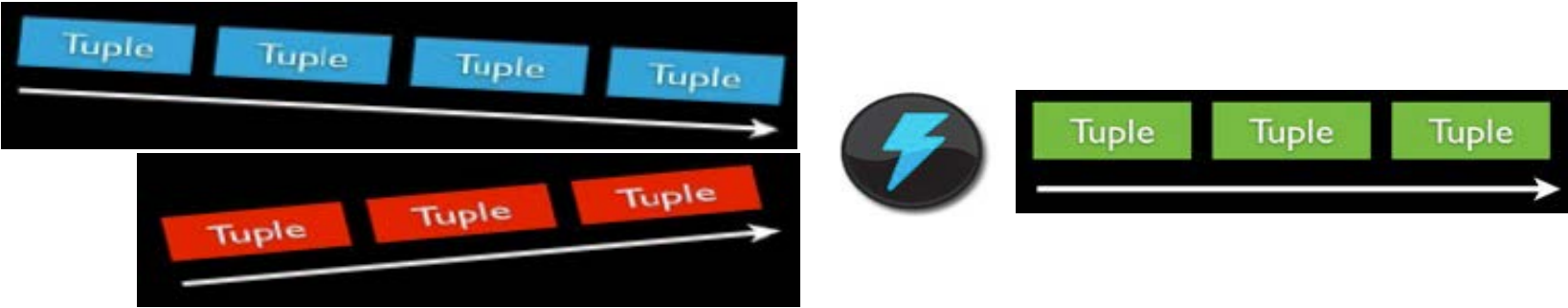
Storm

- Developed by BackType which was acquired by **Twitter**
- **A workflow management and execution platform**
 - For real-time unbounded streams of data: Unlike Hadoop and other batch processing systems
 - Allow users to specify the topology: message queues between components are managed by the system
 - Extremely **robust and fault-tolerant**: guarantee no data loss
 - Scalable: each component can be ran on multiple servers
 - **Programming language agnostic**: can use any programming language



Storm Components

- Tuple: Core unit of data (immutable set of key/value pair)
- Stream: Unbounded sequence of tuples

- Spouts: Source of streams

- Bolts: Processes input streams and produces new streams


Storm Components

- Topology: Spouts and bolts execute as many tasks across the cluster

