INGI2145: CLOUD COMPUTING (Fall 2015)

Beyond MapReduce: In-memory processing, Streaming

5 November 2015

Certain lecture slides adapted from Upenn NETS212 by A. Haeberlen, Z. Ives
Reproduced with permission

MapReduce: Not for Every Task

- MapReduce greatly simplified large-scale data analysis on unreliable clusters of computers
  - Brought together many traditional CS principles
    - functional primitives; master/slave; replication for fault tolerance
  - Hadoop adopted by many companies
  - Affordable large-scale batch processing for the masses

But increasingly people wanted more!

MapReduce: Not for Every Task

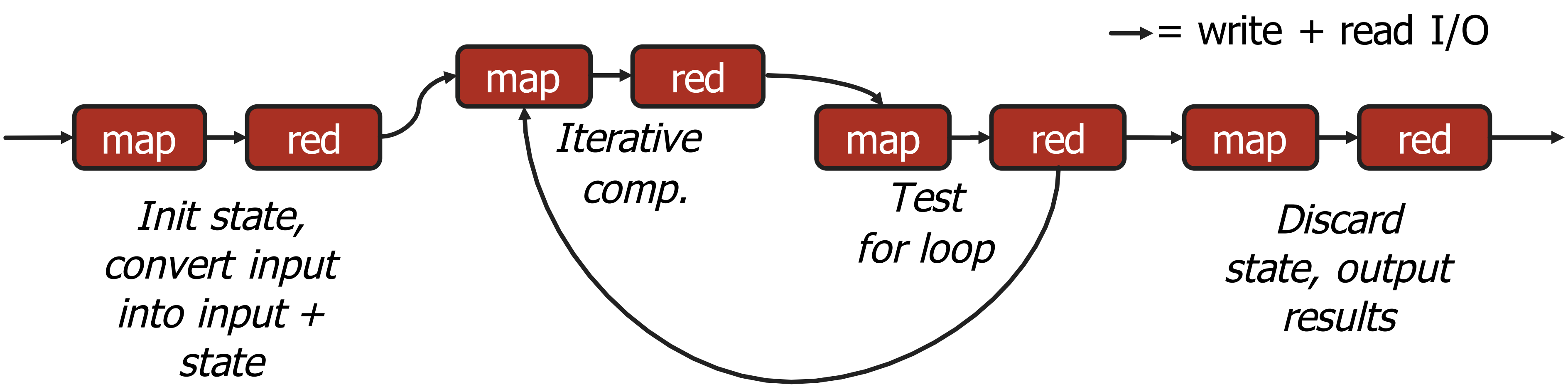
But increasingly people wanted more:

- More complex, multi-stage applications
- More interactive ad-hoc queries
- Process live data at high throughput and low latency

Which are not a good fit for MapReduce…

MapReduce for Iterative Computation

- MapReduce is essentially functional
- Expressing iterative algorithms as chains of Map/Reduce requires passing the entire state and doing a lot of network and disk I/O
  - Recall all between-stage results are materialized to reliable and distributed storage (HDFS)



MapReduce for Ad-hoc Queries

- MapReduce specifically designed for batch operations over large amounts of data
- New analysis task means writing a new MapReduce program
  - Tedious thing to do with languages such as Java
  - Programming interface is not familiar to traditional data analysts with SQL skills
- Getting results incurs development effort!

Plan for today

- Beyond MapReduce ✓
- Abstractions for iterative batch-processing
  - Pregel: Bulk Synchronous Parallel for Graphs ✓
  - Spark: In-Memory Resilient Distributed Datasets ← NEXT
- Stream processing
  - Storm: One-record at a time
  - Spark Streaming: Micro-batching

Spark: Resilient Distributed Datasets

- Let’s think of just having a big block of RAM, partitioned across machines…
  - And a series of operators that can be executed in parallel across the different partitions

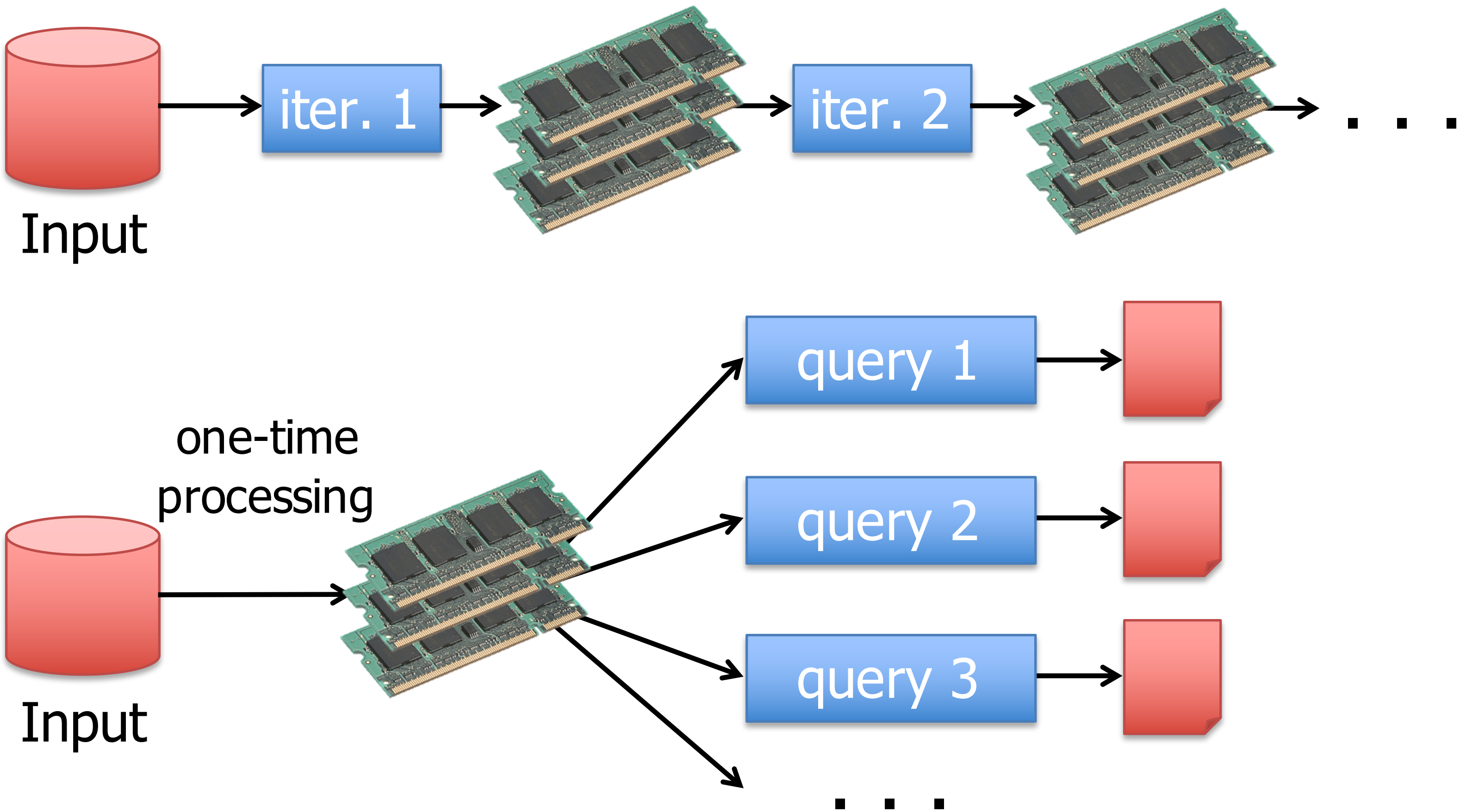
- That’s basically Spark
  - A distributed memory abstraction that is both fault-tolerant and efficient

Spark: Resilient Distributed Datasets

- Restricted form of distributed shared memory
  - Immutable, partitioned collections of records
  - Can only be built through *coarse-grained* deterministic transformations (map, filter, join, …)
  - They are called Resilient Distributed Datasets (RDDs)

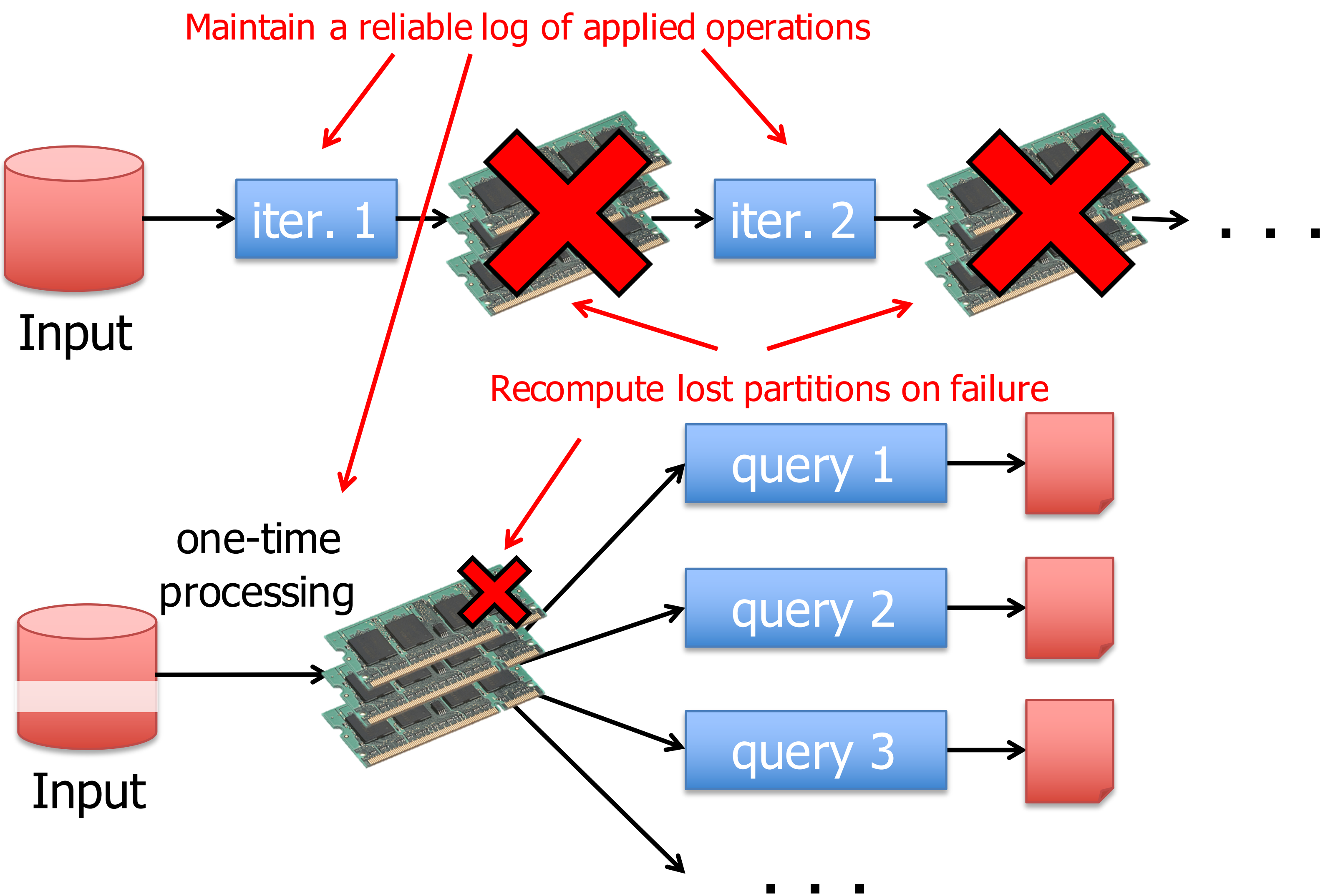
- Efficient fault recovery using *lineage*
  - Log one operation to apply to many elements
  - Recompute lost partitions on failure
  - No cost if nothing fails

In-Memory Data Sharing



Contents from M. Zaharia talk: Resilient Distributed Datasets, NSDI'12

Efficient Fault Recovery via Lineage

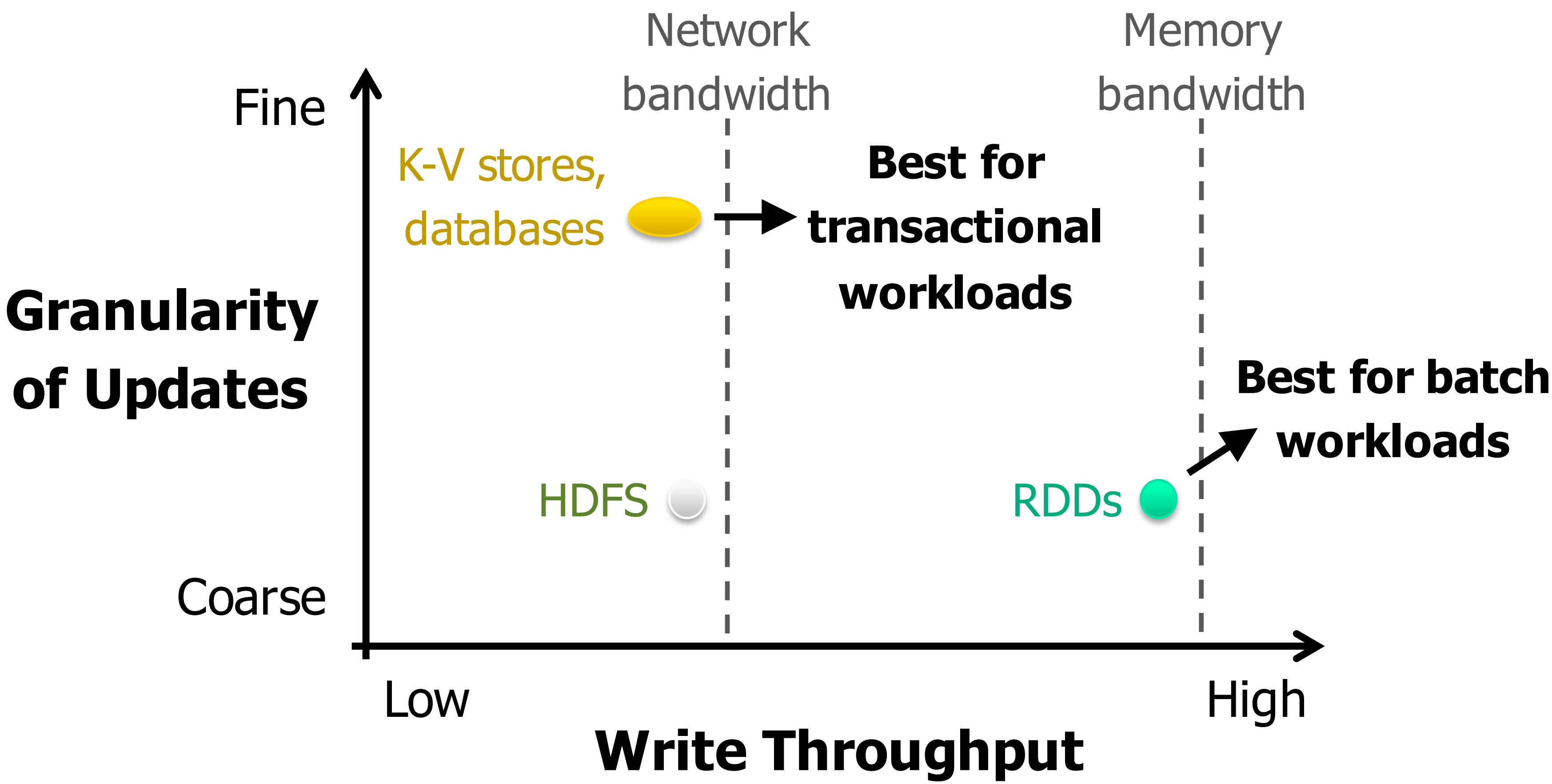


Contents from M. Zaharia talk: Resilient Distributed Datasets, NSDI'12

Generality of RDDs

- Despite their restrictions, RDDs can express many parallel algorithms
  - These naturally *apply the same operation to many items*
- Unify many programming models
  - *Data flow models:* MapReduce, Dryad, SQL, …
  - *Specialized models* for iterative apps: BSP (Pregel), iterative MapReduce (Haloop), bulk incremental, …
- Support *new apps* that these models don't

Tradeoff Space



Spark Programming Interface

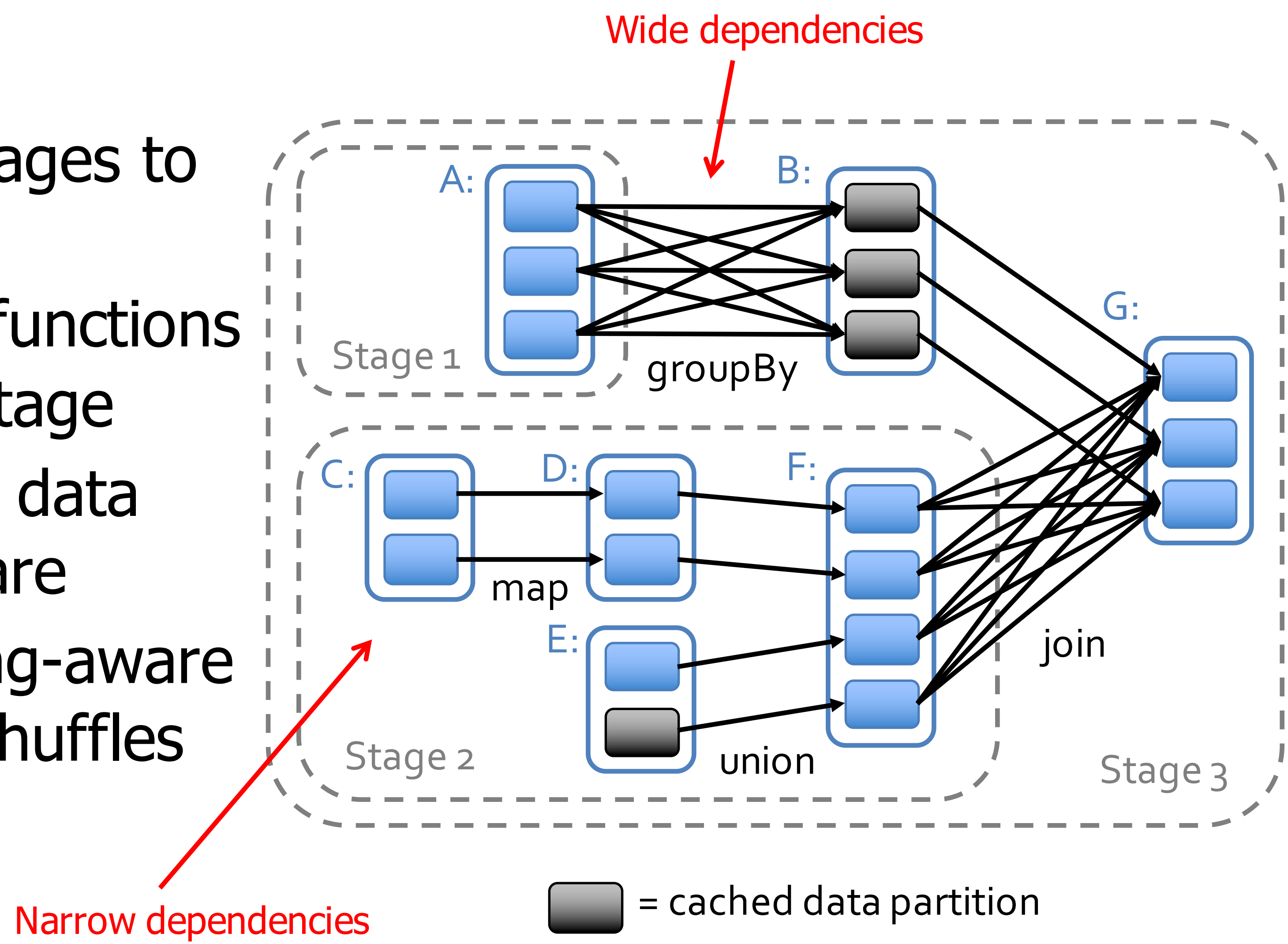
- Language-integrated API in Scala
- Usable interactively from Scala interpreter
- Provides:
  - Resilient distributed datasets (RDDs)
  - Operations on RDDs: deterministic *transformations* (build new RDDs), *actions* (compute and output results)
  - Control of each RDD's *partitioning* (layout across nodes) and *persistence* (storage in RAM, on disk, etc)

Spark Operations

Transformations (define a new RDD)	map filter sample groupByKey reduceByKey sortByKey	flatMap union join cogroup cross mapValues
Actions (return a result to driver program)	collect reduce count save lookupKey take	

Task Scheduler

- DAG of stages to execute
- Pipelines functions within a stage
- Locality & data reuse aware
- Partitioning-aware to avoid shuffles

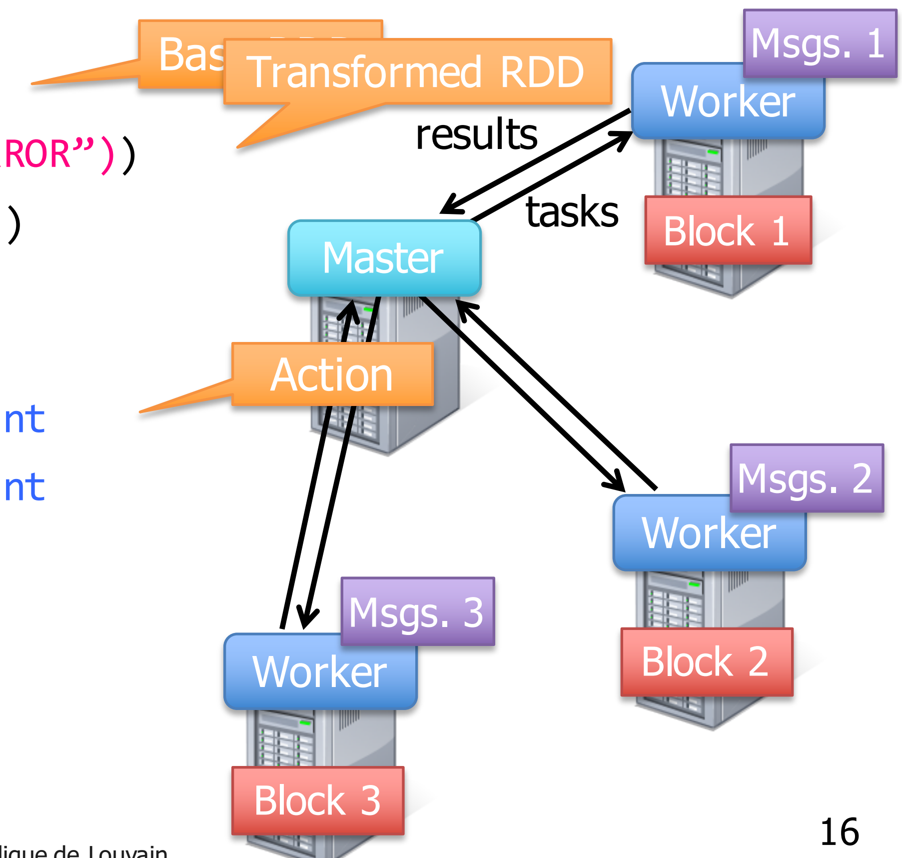


Example: Log Mining

- 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))
messages.persist()

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



Contents from M. Zaharia talk: Resilient Distributed Datasets, NSDI'12

Example: Word Count

MapReduce way

```
public static class WordCountMapClass extends MapReduceBase
  implements Mapper<LongWritable, Text, Text, IntWritable> {
  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();
  public void map(LongWritable key, Text value,
                  OutputCollector<Text, IntWritable> output,
                  Reporter reporter) throws IOException {
    String line = value.toString();
    StringTokenizer itr = new StringTokenizer(line);
    while (itr.hasMoreTokens()) {
      word.set(itr.nextToken());
      output.collect(word, one);
    }
  }
}
public static class WorkdCountReduce extends MapReduceBase
  implements Reducer<Text, IntWritable, Text, IntWritable> {
  public void reduce(Text key, Iterator<IntWritable> values,
                     OutputCollector<Text, IntWritable> output,
                     Reporter reporter) throws IOException {
    int sum = 0;
    while (values.hasNext()) {
      sum += values.next().get();
    }
    output.collect(key, new IntWritable(sum));
  }
}
```

Spark way

```
val spark = new SparkContext(master, appName, [sparkHome], [jars])
val file = spark.textFile("hdfs://...")
val counts = file.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey((x, y) => x + y)
counts.saveAsTextFile("hdfs://...")
```

Example: Word Count

```
val spark = new SparkContext(master, appName, [sparkHome], [jars])
val file = spark.textFile("hdfs://...")
val counts = file.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey((x, y) => x + y)
counts.saveAsTextFile("hdfs://...")
```

Example: PageRank

- Iterative computation

$$PageRank(p) = (1-d) + d \sum_{b \in B_p} \frac{1}{N(b)} PageRank(b)$$

```
var links = // RDD of (url, neighbors) pairs
var ranks = // RDD of (url, rank) pairs

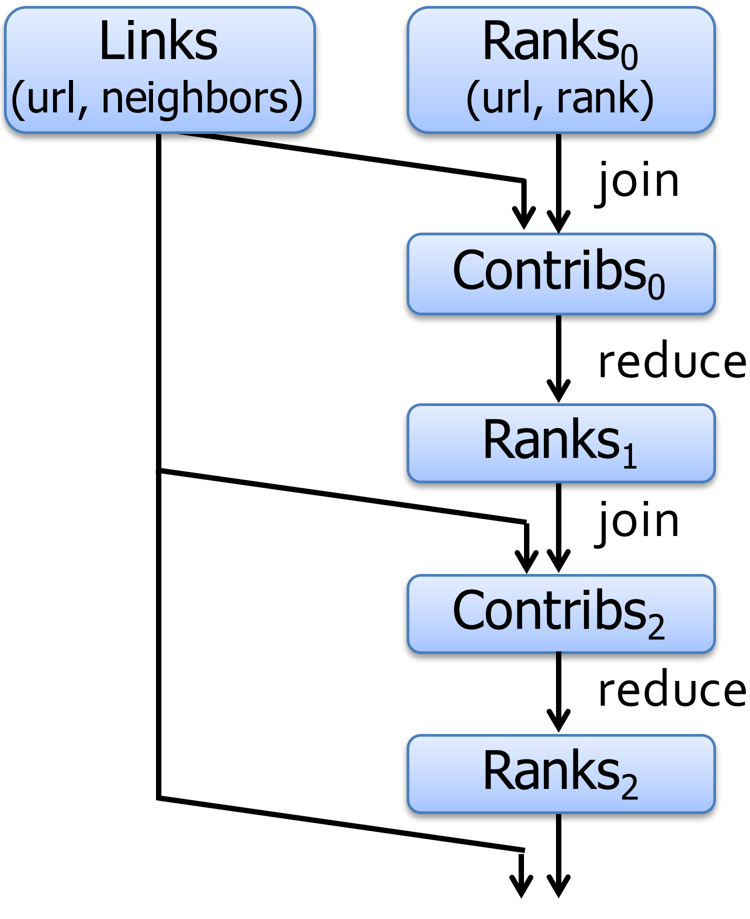
for (i <- 1 to ITERATIONS) {
  contribs = links.join(ranks).flatMap {
    (url, (links, rank)) =>
      links.map(dest => (dest, rank/links.size))
  }
  ranks = contribs.reduceByKey((x, y) => x + y)
                  .mapValues(sum => 0.85*sum + 0.15/N)
}
```

RDDs are immutable

contribs and ranks are new RDDs!

Contents from M. Zaharia talk: Resilient Distributed Datasets, NSDI'12

Optimizing Placement



- `links` & `ranks` repeatedly joined
- Can *co-partition* them (e.g. hash both on URL) to avoid shuffles
- Can also use app knowledge, e.g., hash on DNS name

- ```
  links = links.partitionBy(
                  new URLPartitioner())
  ```
  ```

Programming Models Implemented on Spark

- RDDs can express many existing parallel models
  - **MapReduce, DryadLINQ**
  - **Pregel** graph processing
  - **Iterative MapReduce**
  - **SQL**: Hive on Spark (Shark)

All are based on coarse-grained operations

- Enables apps to efficiently *intermix* these models

Spark Summary

- Global aggregate computations that produce program state
  - compute the count() of an RDD, compute the max diff, etc.
- Loops!
  - Spark makes it much easier to do multi-stage MapReduce
- Built-in abstractions for some other common operations like joins

- See also Apache Crunch / Google FlumeJava for a very similar approach

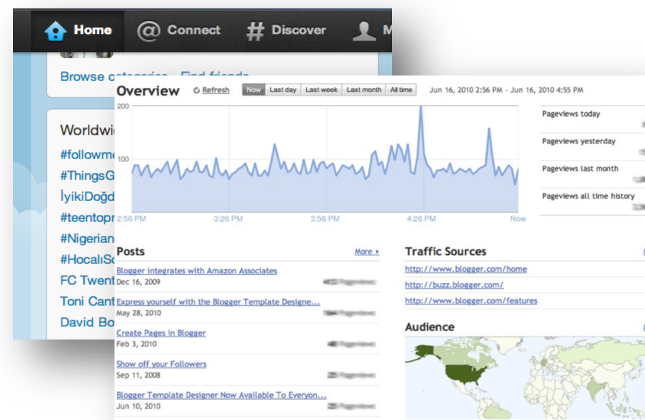
Plan for today

- Beyond MapReduce
- Abstractions for iterative batch-processing
  - Pregel: Bulk Synchronous Parallel for Graphs
  - Spark: In-Memory Resilient Distributed Datasets
- Stream processing
  - Storm: One-record at a time NEXT
  - Spark Streaming: Micro-batching

Stream Processing

- Many important applications must process large streams of live data and provide results in near-real-time
  - Social network trends
  - Website statistics
  - Ad impressions

  …

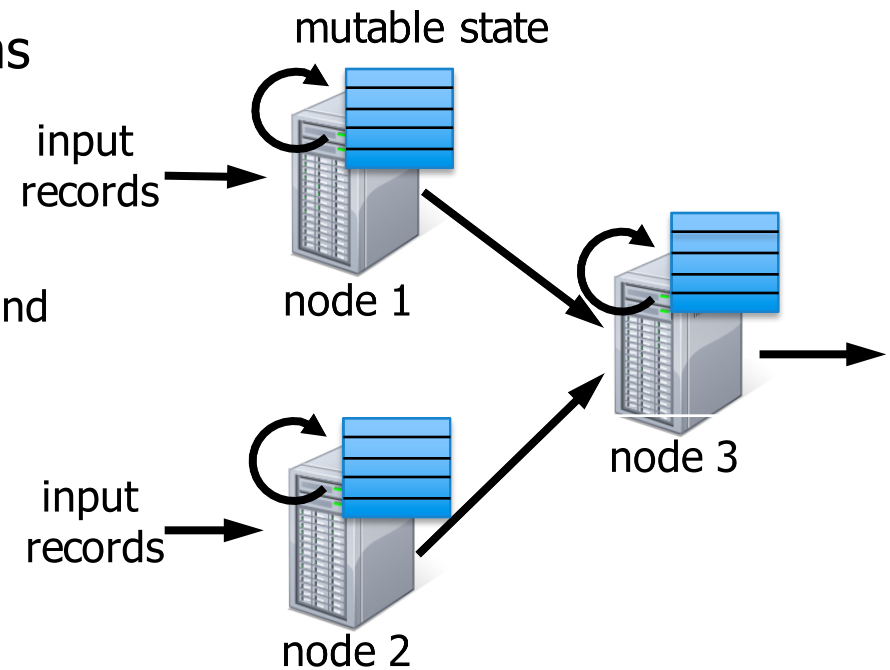


- Distributed stream processing framework is required to
  - Scale to large clusters (100s of machines)
  - Achieve low latency (few seconds)

Contents from T. Das talk: Spark Streaming, AMPCamp'13

Stateful Stream Processing

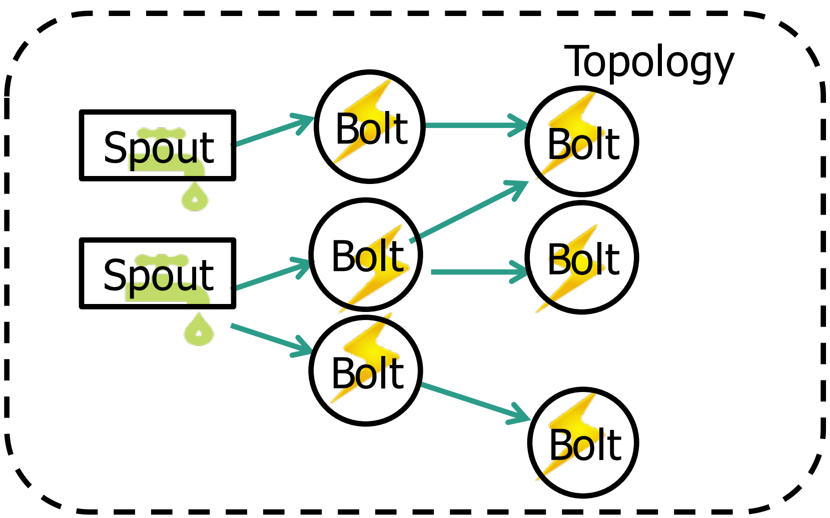
- Traditional streaming systems have a record-at-a-time processing model
  - Each node has mutable state
  - For each record, update state and send new records
- State is lost if node dies!
- Making stateful stream processing be fault-tolerant is challenging



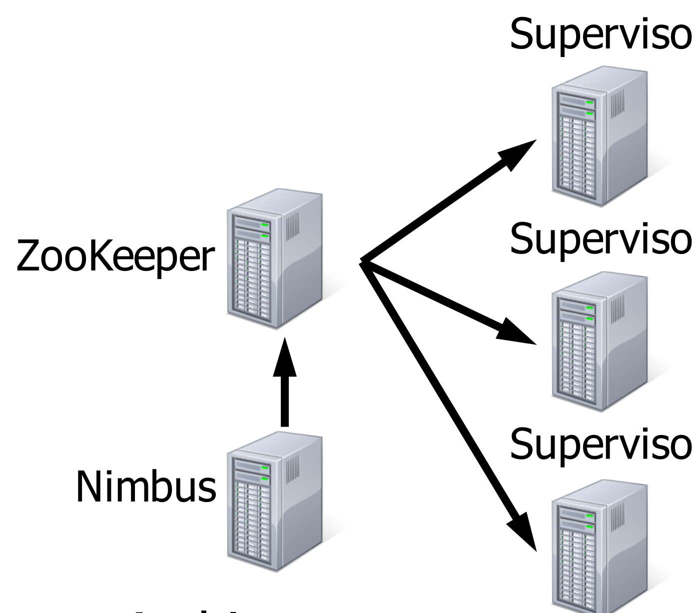
Contents from T. Das talk: Spark Streaming, AMPCamp'13

Apache Storm

- Framework for distributed stream processing
- Provides: Stream Partitioning + Fault Tolerance + Parallel Execution



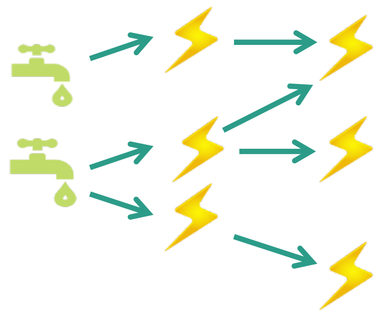
Programming Model



Architecture

Abstractions in Storm

Topology



Arbitrarily complex multi-stage stream computation

Stream

Unbounded sequence of tuples

Spout

Source of streams

Bolt

Process input streams and produce new streams

Holds most computation logic

Storm Cluster Architecture

- A storm cluster has three sets of nodes:

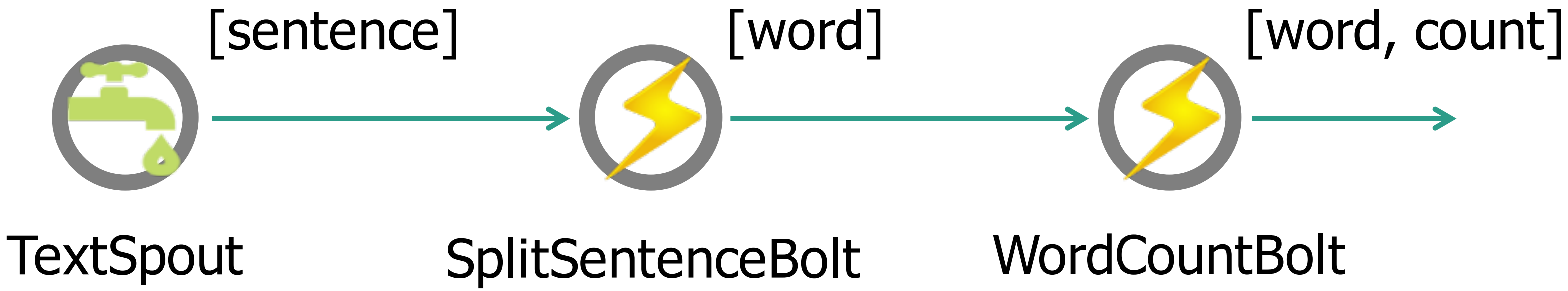
- Nimbus node (master node)
  - Similar to the Hadoop JobTracker
  - Distributes code, launches workers across the cluster
  - Monitors computation and reallocates workers as needed
- ZooKeeper nodes – (coordinate the cluster)
  - Will discuss ZooKeeper in detail in a later lecture
- Supervisor nodes
  - Start and stop workers according to signals from Nimbus

Fault Tolerance

- If a supervisor node fails, Nimbus reassigns that node's task to other nodes in the cluster
- Any tuples sent to a failed node will time out and be replayed
  - Delivery guarantee dependent on a reliable data source
    - It can replay a message if processing fails at any point
- Storm can guarantee that every tuple will be process at least once or at most once, but not exactly once
  - Exactly once guarantee requires a durable data source that can replay any message or set of messages given the necessary selection criteria

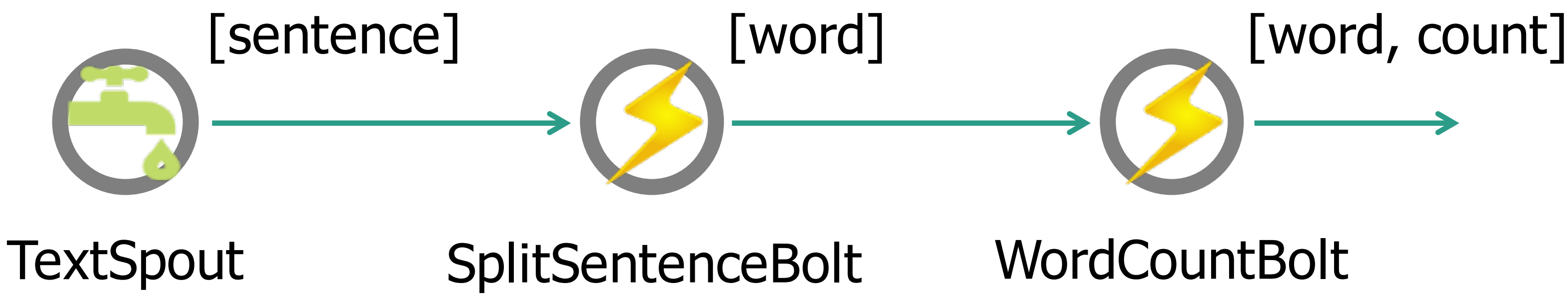
Contents from P. T. Goetz talk: Apache Storm and Spark Streaming Compared

Example: Word Count




```
TextSpout implements IRichSpout {
nextTuple() {
  while ((str = reader.readLine()) != null)
    collector.emit(new Values(str), str);
}
[…]
}
```

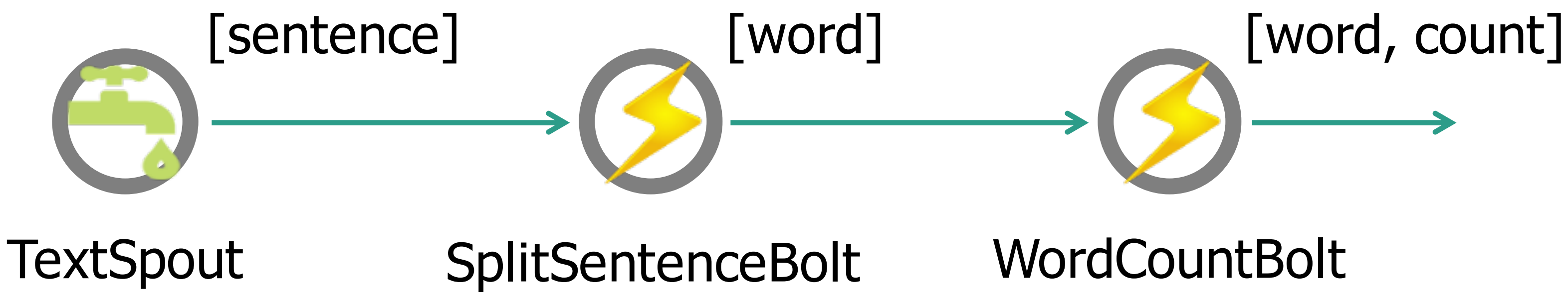
Example: Word Count




```
class SplitSentenceBolt implements
IRichBolt {
execute(Tuple input) {
  String sentence = input.getString(0);
  String[] words = sentence.split(" ");
  for (String word: words) {
    collector.emit(new Values(word));
  }
  collector.ack(input);
}
[…] }
```

```
class WordCounterBolt implements
IRichBolt {
Map<String, Integer> counters;
execute(Tuple input) {
  String str = input.getString(0);
  if(!counters.containsKey(str))
    counters.put(str, 1);
  else {
    Integer c = counters.get(str) + 1;
    counters.put(str, c);
  }
  collector.ack(input);
}
[…] }
```

Example: Word Count




```
public class WordCountTopology {
[…] main(String[] args) throws Exception {
  Config config = new Config();
  config.setDebug(true);
  TopologyBuilder builder = new TopologyBuilder();
  builder.setSpout("textspout", new LineReaderSpout());
  builder.setBolt("splitsentence", new WordSpitterBolt()).shuffleGrouping("textspout");
  builder.setBolt("word-count", new WordCounterBolt()).shuffleGrouping("splitsentence");
  LocalCluster cluster = new LocalCluster();
  cluster.submitTopology("WordCountTopology", config, builder.createTopology());
  Thread.sleep(10000);
  cluster.shutdown();
} }
```

Plan for today

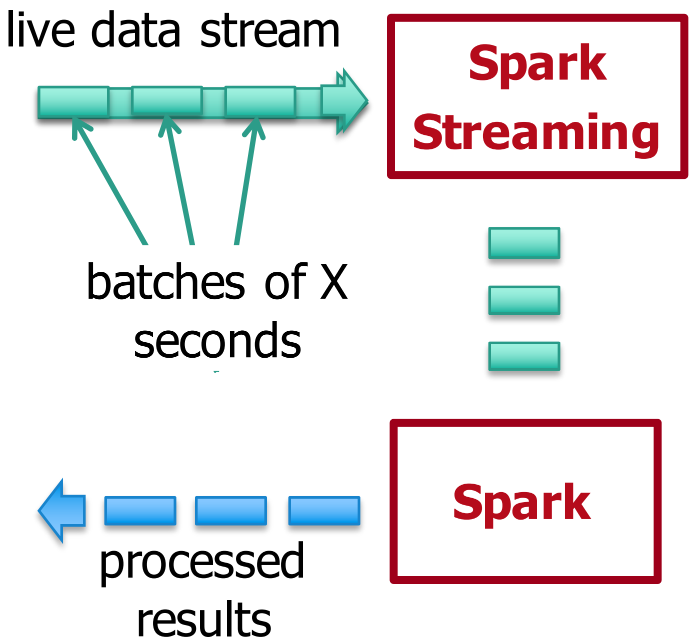
- Beyond MapReduce
- Abstractions for iterative batch-processing
  - Pregel: Bulk Synchronous Parallel for Graphs
  - Spark: In-Memory Resilient Distributed Datasets
- Stream processing
  - Storm: One-record at a time
  - Spark Streaming: Micro-batching

NEXT

Spark Streaming

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

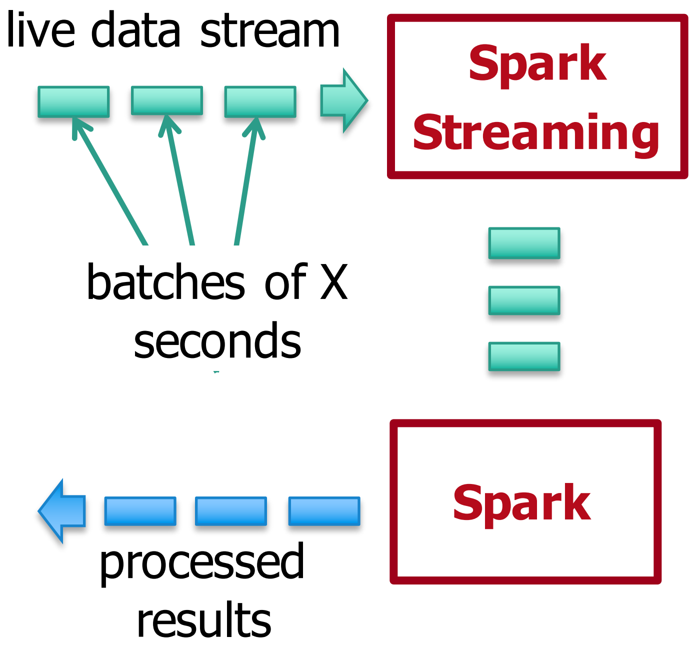


Contents from T. Das talk: Spark Streaming, AMPCamp'13

Spark Streaming

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

- Batch sizes as low as ½ second, latency of about 1 second
- Potential for combining batch processing and streaming processing in the same system

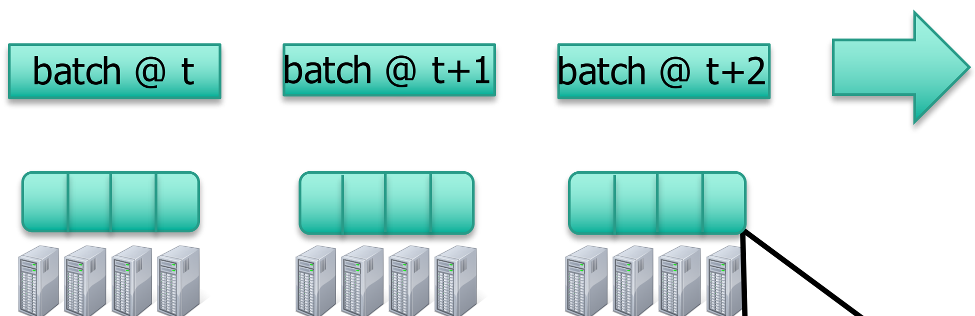


Contents from T. Das talk: Spark Streaming, AMPCamp'13

Example: Get hashtags from Twitter

```
val tweets = ssc.twitterStream()
```

DStream: a sequence of RDDs representing a stream of data

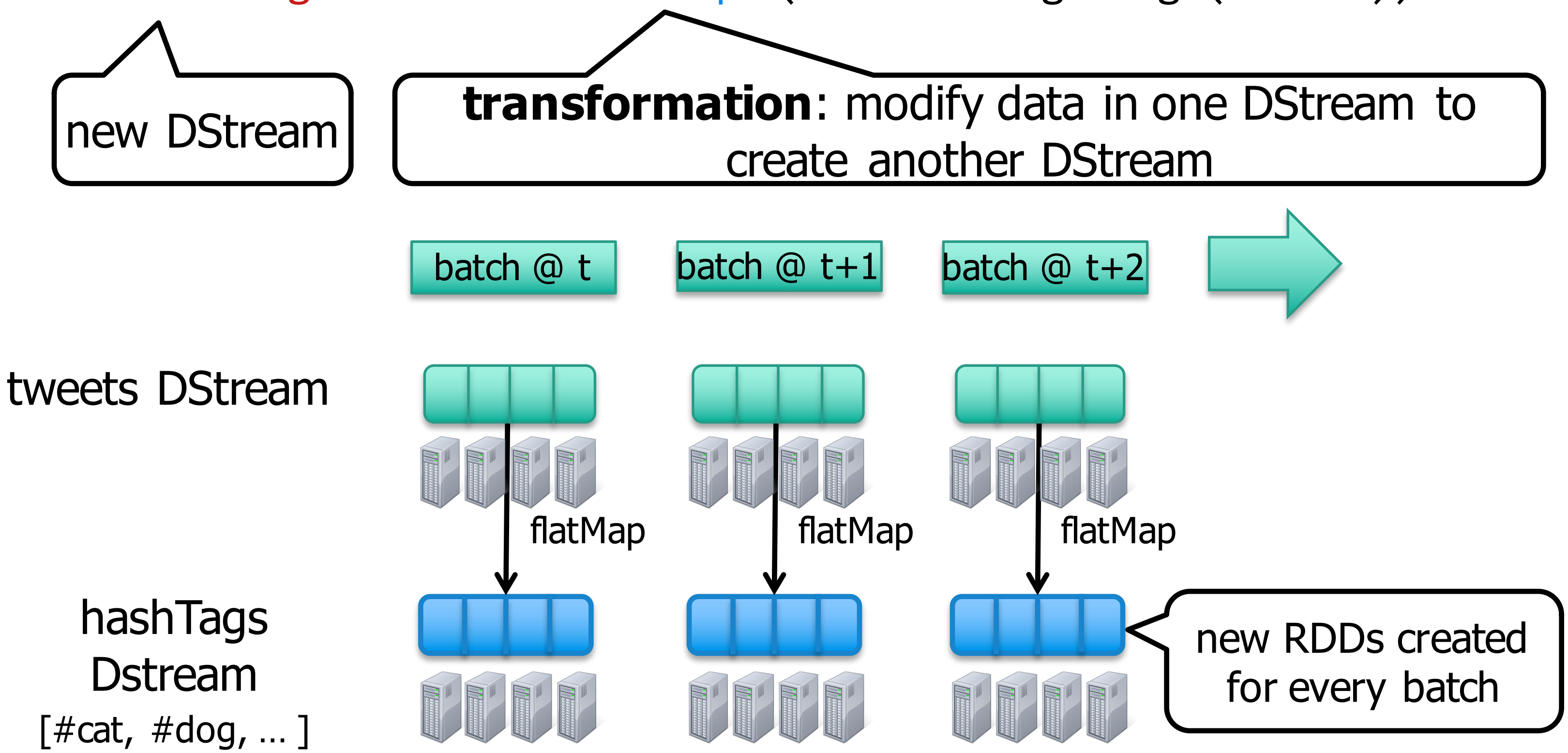


stored in memory as an RDD (immutable, distributed)

Contents from T. Das talk: Spark Streaming, AMPCamp'13

Example: Get hashtags from Twitter

```
val tweets = ssc.twitterStream()
val hashTags = tweets.flatMap (status => getTags(status))
```

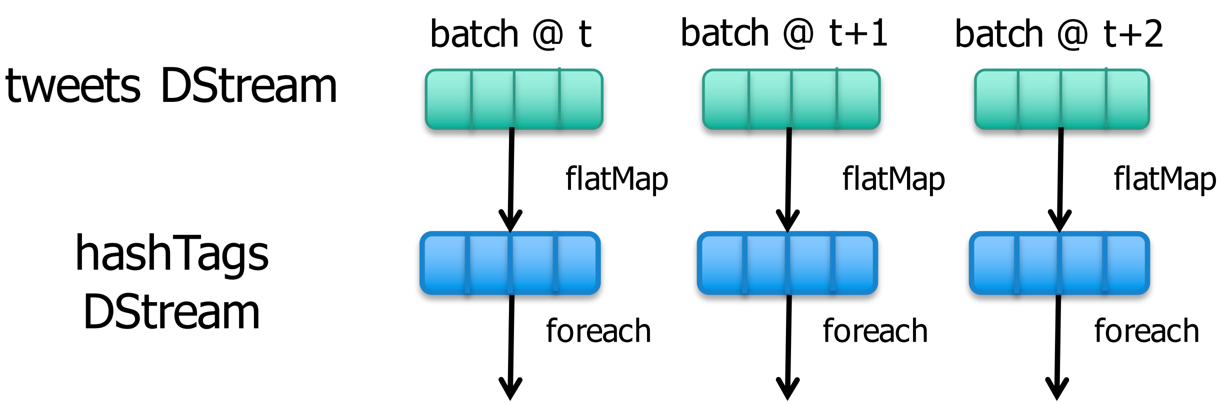


Contents from T. Das talk: Spark Streaming, AMPCamp'13

Example: Get hashtags from Twitter

```
val tweets = ssc.twitterStream()
val hashTags = tweets.flatMap (status => getTags(status))
hashTags.foreach(hashTagRDD => { ... })
```

foreach: do whatever you want with the processed data

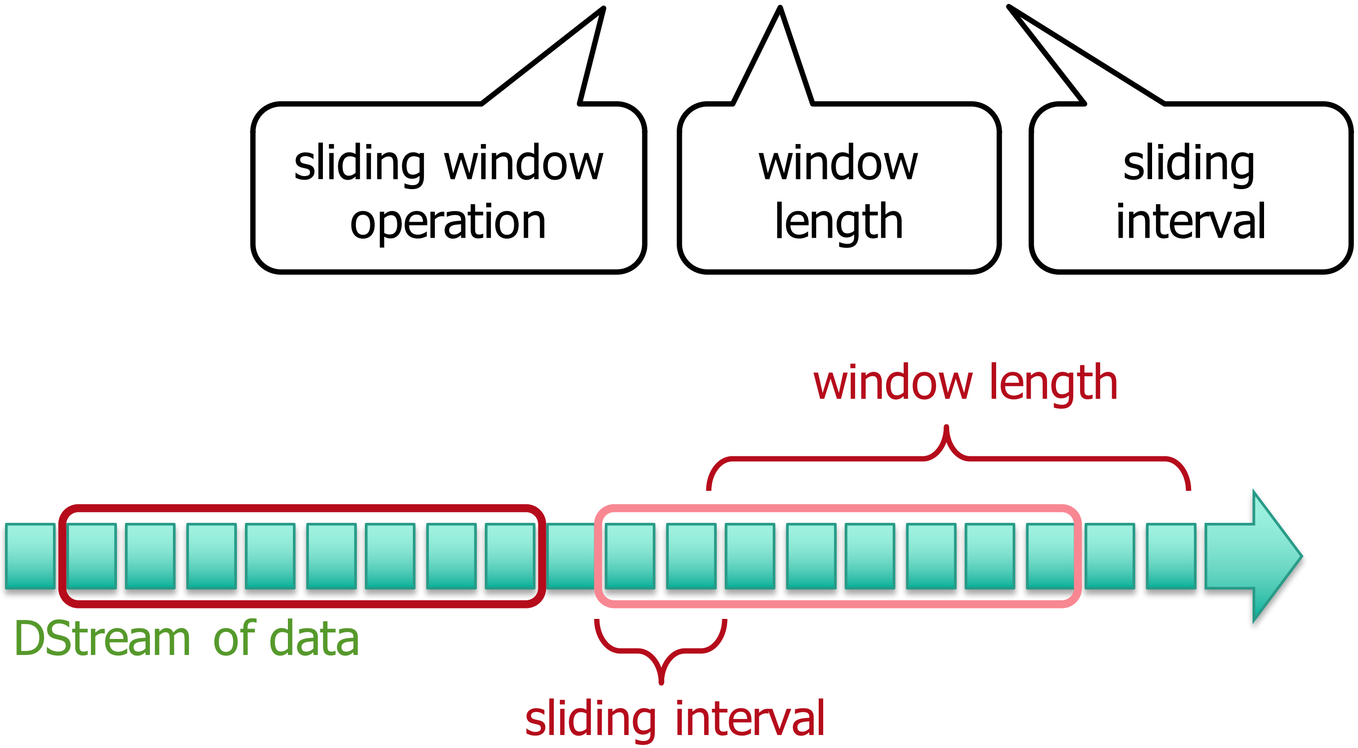


Write to database, update analytics UI, do whatever you want

Contents from T. Das talk: Spark Streaming, AMPCamp'13

Window-based Transformations

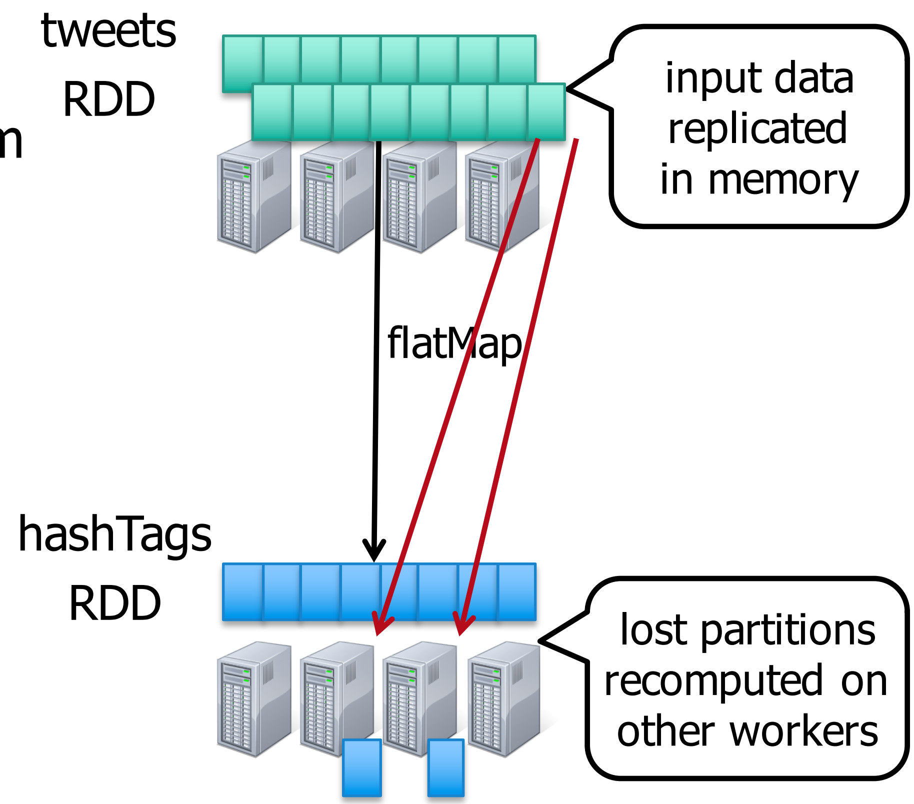
```
val tweets = ssc.twitterStream()
val hashTags = tweets.flatMap (status => getTags(status))
val tagCounts = hashTags.window(Min(1), Sec(5)).countByValue()
```



Contents from T. Das talk: Spark Streaming, AMPCamp'13

Fault Tolerance

- RDDs remember the operations that created them
- Batches of input data are replicated in memory for fault-tolerance
- Data lost due to worker failure, can be recomputed from replicated input data



Contents from T. Das talk: Spark Streaming, AMPCamp'13

Streaming Summary

- Stream processing of large amounts of live data is an important requirement
  - Desirable to have both high throughput (100s of MB/s) and low latency (~s)
- Combine the efficiency of in-memory distributed processing of Spark with stream processing model
  - Key is to break down processing in small batches
  - Storm has a second API (Trident) for micro-batch processing
- Also an advantage: use and maintain a single software stack for both processing models

Stay tuned

http://www.flickr.com/photos/3dking/2573905313/sizes/l/in/photostream/

Next time you will learn about:

Cloud networks