INGI2145: CLOUD COMPUTING (Fall 2014)

Beyond MapReduce

30 October 2014

2002-2004: Lucene and Nutch

- Early 2000s: Doug Cutting develops two open-source search projects:
  - Lucene: Search indexer
    - Used e.g., by Wikipedia
  - Nutch: A spider/crawler (with Mike Carafella)

- Nutch
  - Goal: Web-scale, crawler-based search
  - Written by a few part-time developers
  - Distributed, 'by necessity'
  - Demonstrated 100M web pages on 4 nodes, but true 'web scale' still very distant

2004-2006: GFS and MapReduce

- 2003/04: GFS, MapReduce papers published
  - Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung: "The Google File System", SOSP 2003
  - Jeffrey Dean and Sanjay Ghemawat: "MapReduce: Simplified Data Processing on Large Clusters", OSDI 2004
  - Directly addressed Nutch's scaling issues

- GFS & MapReduce added to Nutch
  - Two part-time developers over two years (2004-2006)
  - Crawler & indexer ported in two weeks
  - Ran on 20 nodes at IA and UW
  - Much easier to program and run, scales to several 100M web pages, but still far from web scale

2006-2008: Yahoo

- 2006: Yahoo hires Cutting
  - Provides engineers, clusters, users, ...
  - Big boost for the project; Yahoo spends tens of M$
  - Not without a price: Yahoo has a slightly different focus (e.g., security) than the rest of the project; delays result
- Hadoop project split out of Nutch
  - Finally hit web scale in early 2008
- Cutting is now at Cloudera
  - Startup; started by three top engineers from Google, Facebook, Yahoo, and a former executive from Oracle
  - Has its own version of Hadoop; software remains free, but company sells support and consulting services
  - Was elected chairman of Apache Software Foundation

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

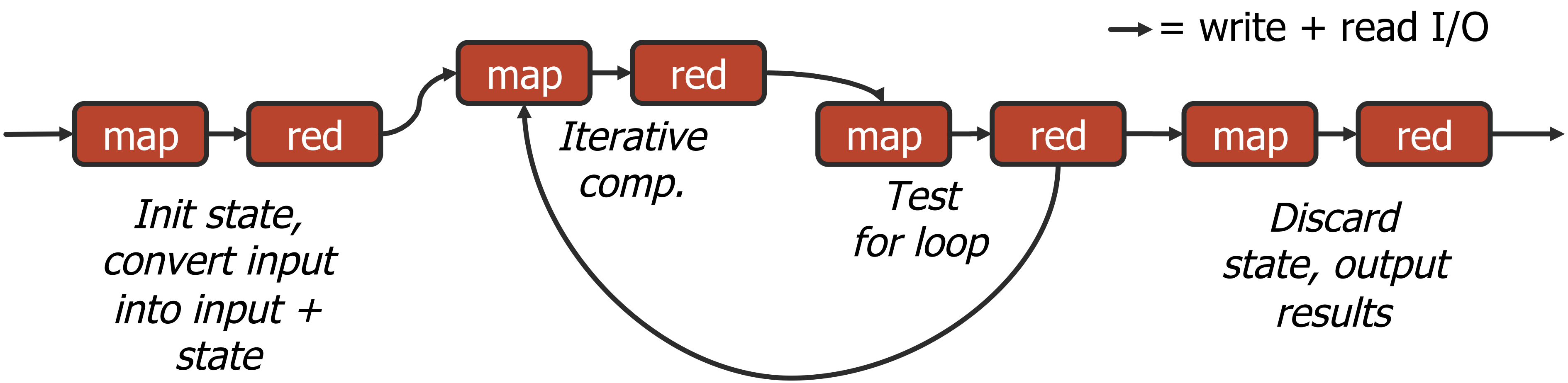
But increasingly people wanted more:

- More complex, multi-stage applications
- More interactive ad-hoc queries
- Process live data at high txput 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 NEXT
  - Pregel: Bulk Synchronous Parallel for Graphs
  - Spark: In-Memory Resilient Distributed Datasets
- Higher-level languages for Hadoop
  - Hive Query Language
  - Pig and Pig Latin
- Stream processing
  - Storm: One-record at a time
  - Spark Streaming: Micro-batching

New Abstractions Needed

Much of the mismatch stems from the lack of shared global state

Complex applications and interactive queries both need one thing that MapReduce lacks

- Efficient primitives for data sharing

What If We Could Remember?

Suppose we were to change things entirely:

- A set of machines
- … each with a *partition* of a dataset, stored in memory

- Computation consists of *sending updates* from one portion to another

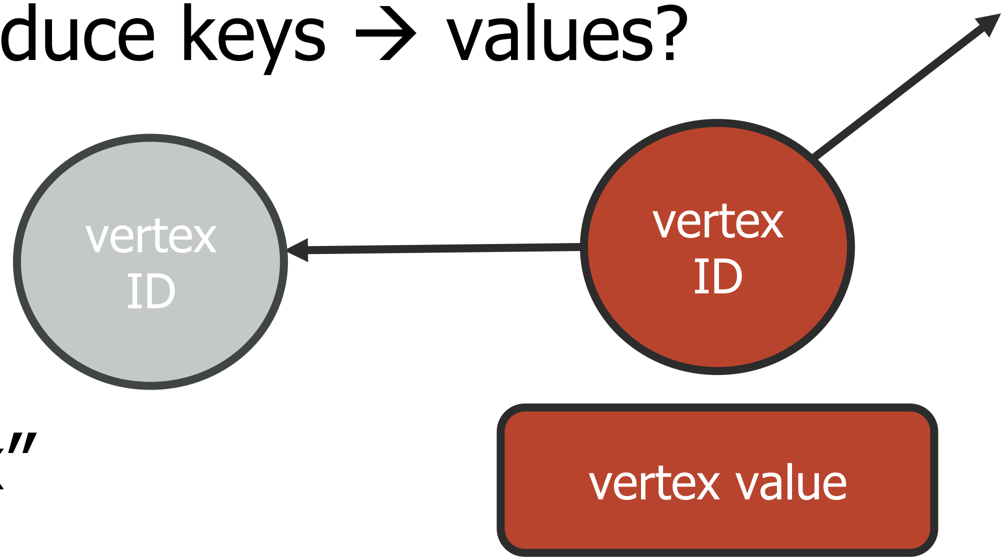
Let’s look at two versions of this

Pregel: Bulk Synchronous Parallel

Let's slightly rethink the MapReduce model for processing **graphs**

- Vertices
- "Edges" are really messages

Compare to MapReduce keys → values?



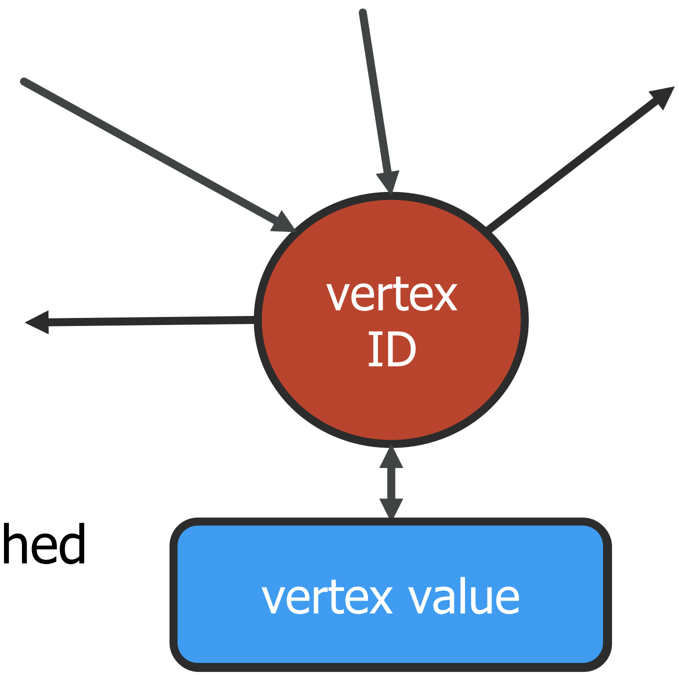
"Think like a vertex"

The Basic Pregel Execution Model

A sequence of *supersteps*, for each vertex V

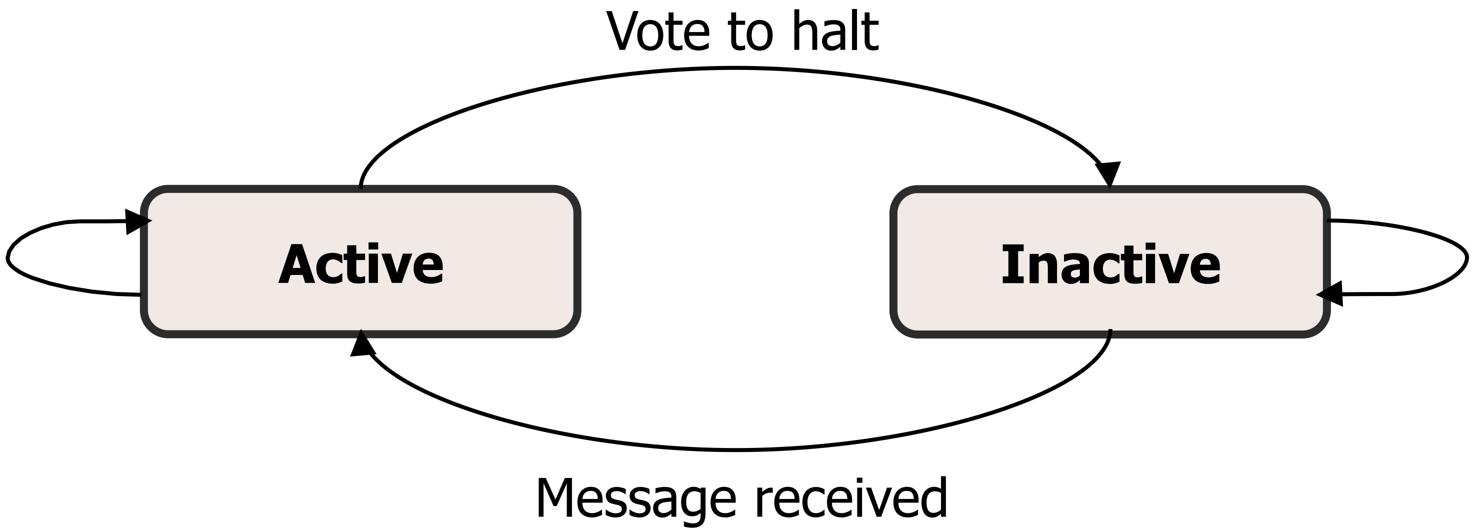
At superstep S:

- Compute in parallel at each V
  - Read messages sent to V in superstep S-1
  - Update value / state
  - Optionally change topology
- Send messages
- Synchronization
  - Wait till all communication is finished

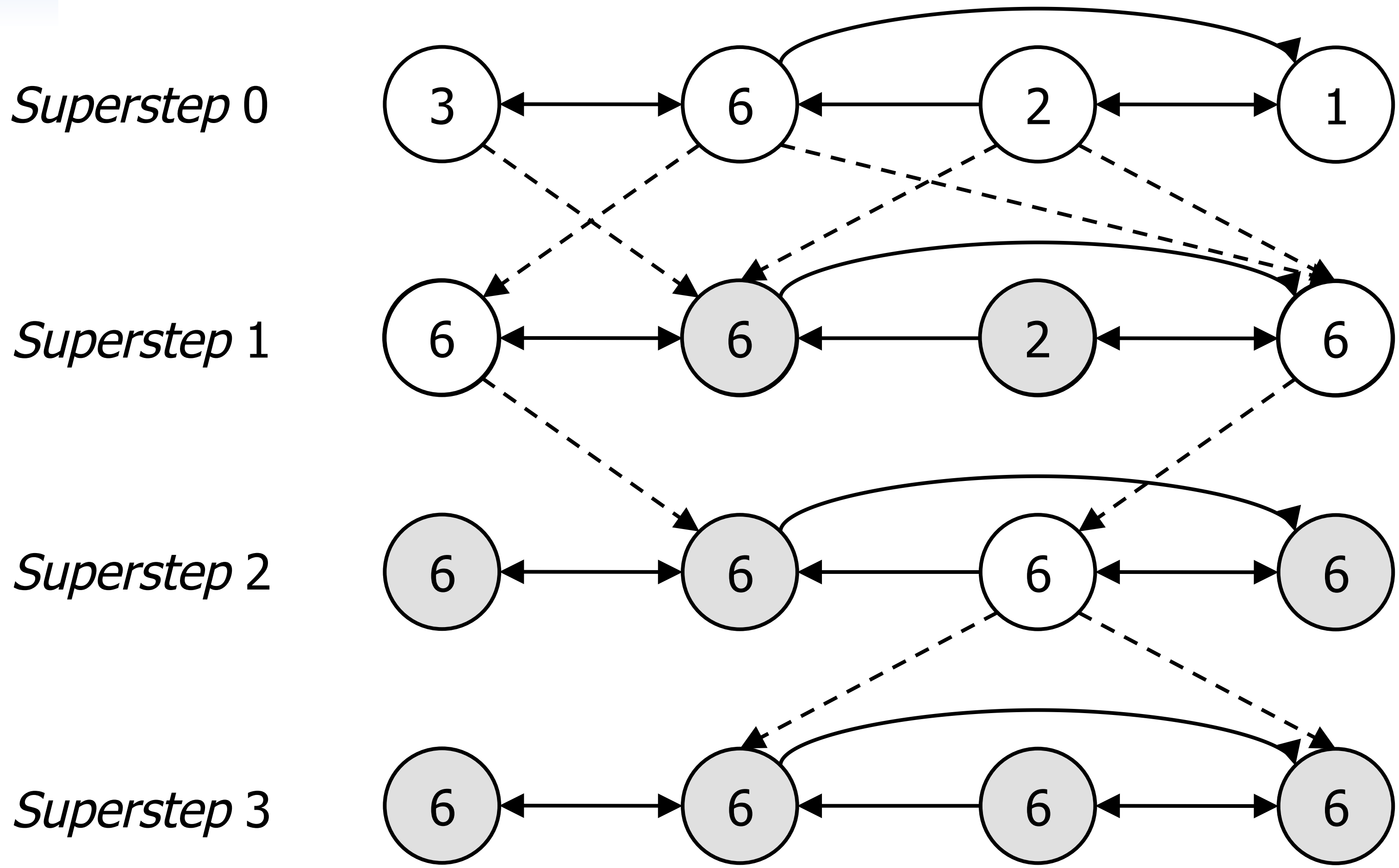


Termination Test

- Based on every vertex voting to halt
  - Once a vertex deactivates itself it does no further work unless triggered externally by receiving a message
- Algorithm terminates when all vertices are simultaneously inactive



Example: Find Maximum Value



Pregel Summary

- Bulk Syncronous Parallel – sequence of synchronized supersteps
  - Abstraction originally invented by Leslie Valliant in the ’80s
- Consider the nodes to have state (memory) that carries from superstep to superstep
- Connections to MapReduce model?
- See also Apache Hama, Giraph, Graph.lab

Plan for today

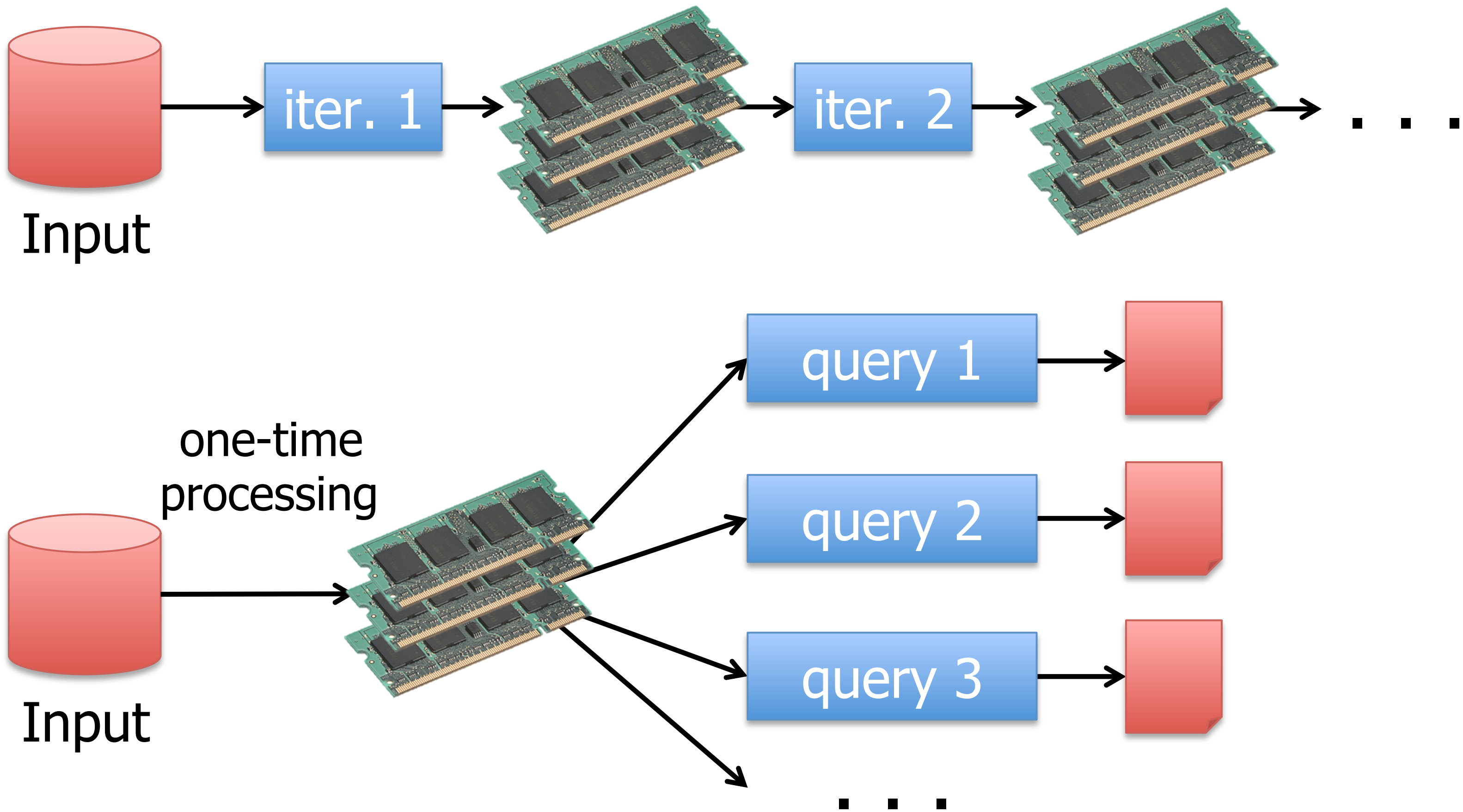
- Beyond MapReduce
- Abstractions for iterative batch-processing
  - Pregel: Bulk Synchronous Parallel for Graphs
  - Spark: In-Memory Resilient Distributed Datasets NEXT
- Higher-level languages for Hadoop
  - Hive Query Language
  - Pig and Pig Latin
- Stream processing
  - Storm: One-record at a time
  - 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

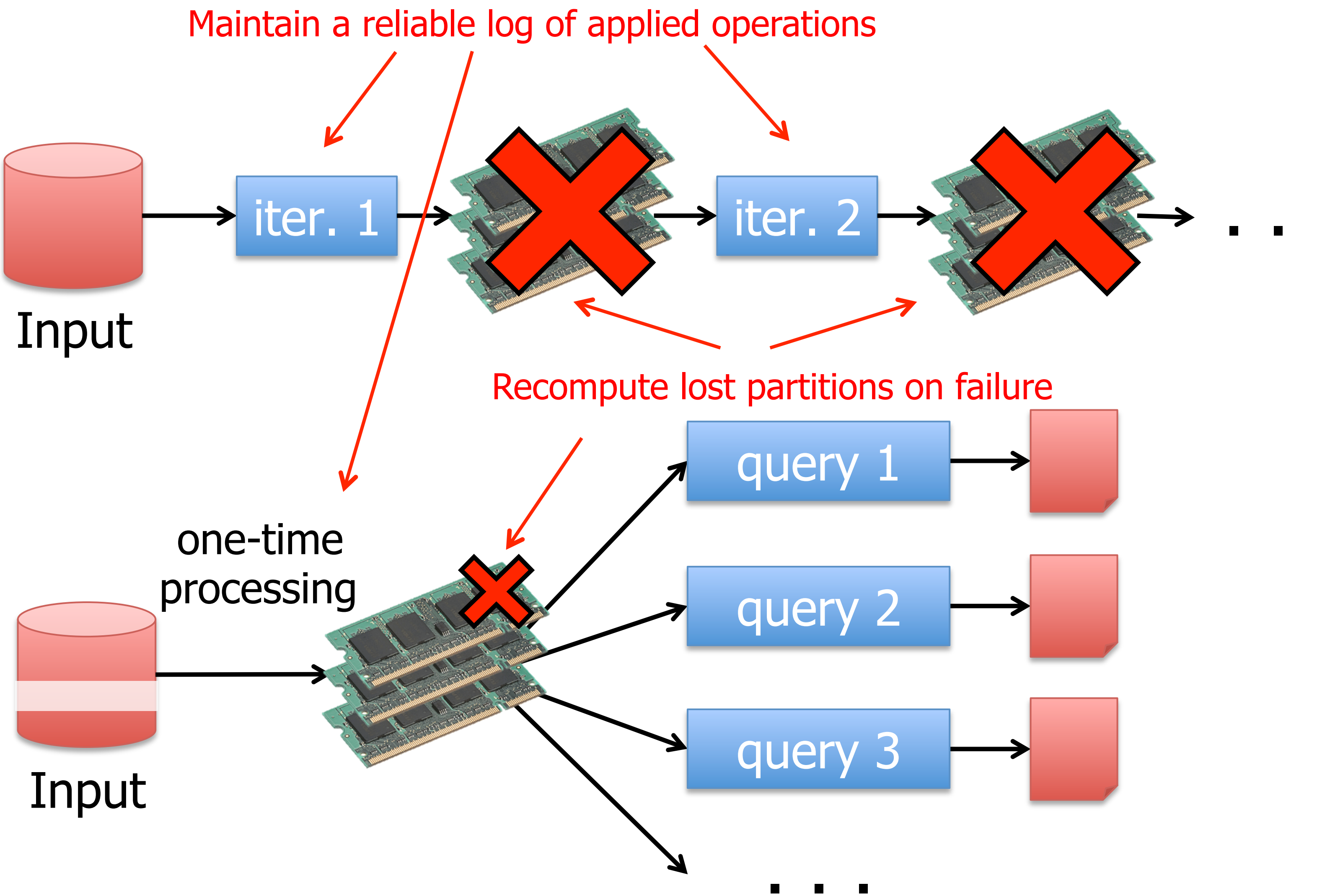
- That's basically Spark
  - A distributed memory abstraction that is both fault-tolerant and efficient
  - Spark programs are written by defining *coarse-grained deterministic* functions to be called over *immutable* collections of records
  - Automatically rebuilt on failure

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

Programming Interface

- Resilient distributed datasets (RDDs)
  - Immutable collections of records spread across a cluster, stored in RAM or on disk
- RDDs can only be built through coarse-grained deterministic **transformations** executed in parallel on the cluster

map	flatMap	filter	union
sample	join	groupByKey	cogroup
reduceByKey	cross	sortByKey	mapValues

- Programs use **actions** to output results

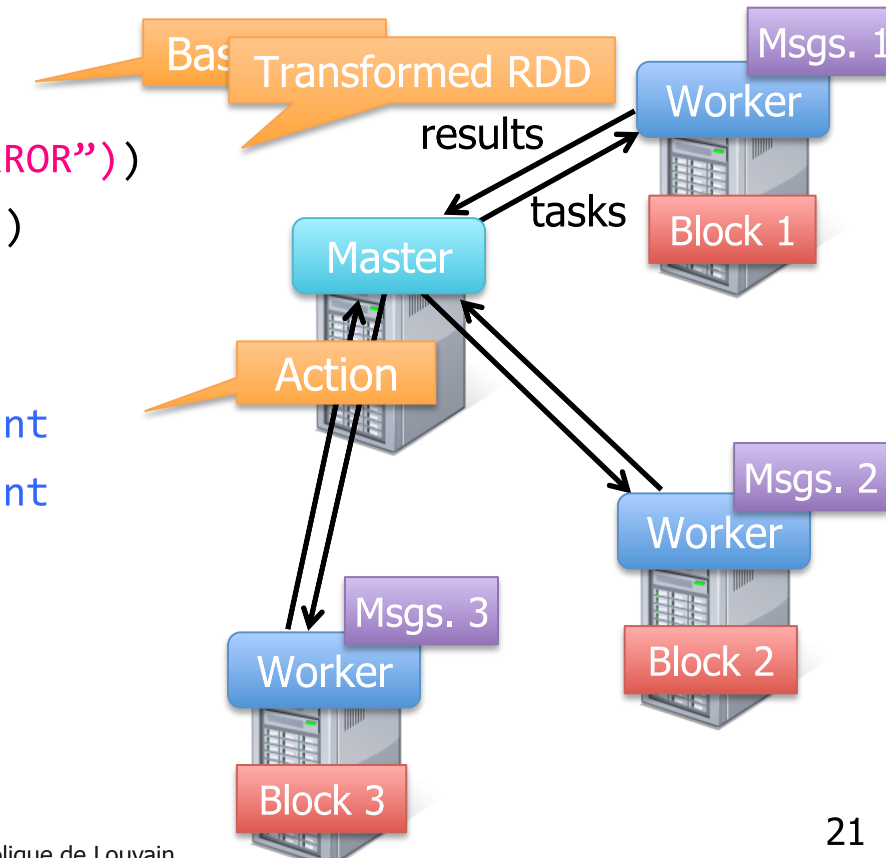
collect	reduce	count	save	lookupKey

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

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(_ + _)
counts.saveAsTextFile("hdfs://...")
```

Example: Simplified PageRank

- Iterative computation

$$PageRank(p) = \sum_{b \in B(p)} \frac{1}{N(b)} PageRank(b)$$

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

for (i <- 1 to ITERATIONS) {
  contribs = graph.join(ranks).flatMap {
    case (url, (links, rank)) =>
      links.map(dest => (dest, rank/links.size))
  }
  ranks = contribs.reduceByKey(_ + _)
}
```

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

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
- Higher-level languages for Hadoop
  - Hive Query Language NEXT
  - Pig and Pig Latin
- Stream processing
  - Storm: One-record at a time
  - Spark Streaming: Micro-batching

Hive: SQL on top of Hadoop

- SQL is a higher-level language than MapReduce
  - Problem: Company may have lots of people with SQL skills, but few with Java/MapReduce skills

- Can we "bridge the gap" somehow?

```
SELECT a.campaign_id, count(*), count(DISTINCT b.user_id)
FROM dim_ads a JOIN impression_logs b ON(b.ad_id=a.ad_id)
WHERE b.dateid = '2008-12-01'
GROUP BY a.campaign_id
```

- Idea: SQL frontend for MapReduce
  - Abstract delimited files as tables (give them schemas)
  - Compile (approximately) SQL to MapReduce jobs!

Recall: Database Mgmt System

- An abstract storage system
  - Provides access to tables, organized however the database administrator and the system have chosen
- Relational data model
  - Schema formally describes fields, data types, and constraints
- A declarative processing model
  - Query language: SQL or similar
  - We describe what we want to store or compute, not how it should be done
  - More general than (single-pass) MapReduce
- A strong consistency and durability model
  - Transactions with ACID properties

Roles of a DBMS

- Online transaction processing (OLTP)
  - Workload: Mostly updates
  - Examples: Order processing, flight reservations, banking, …
- Online analytic processing (OLAP)
  - Workload: Mostly queries
  - Aggregates data on different axes; often step towards mining
- May well have combinations of both

- Stream / Web
  - Today not all of the data really needs to be in a database – it can be on the network!

Hive

- A data warehouse infrastructure built on top of Hadoop for providing data summarization, query and analysis
- Hive Query Language (HQL) – similar to SQL
  - Suitable for processing structured data
  - Create a table structure on top of HDFS
  - Queries are compiled in to MapReduce jobs
- Not designed for OLTP!
  - Updating records or transactions are not supported

Example: WordCount

```
CREATE TABLE doc (line STRING);
LOAD DATA LOCAL INPATH 'text.txt' INTO TABLE doc;

CREATE TABLE wordcount AS
SELECT word, count(1) AS count
FROM (SELECT EXPLODE(SPLIT(line, '\s')) AS word FROM doc) words
GROUP BY word
ORDER BY count DESC, word ASC;
```

Plan for today

- Beyond MapReduce
- Abstractions for iterative batch-processing
  - Pregel: Bulk Synchronous Parallel for Graphs
  - Spark: In-Memory Resilient Distributed Datasets
- Higher-level languages for Hadoop
  - Hive Query Language
  - Pig and Pig Latin NEXT
- Stream processing
  - Storm: One-record at a time
  - Spark Streaming: Micro-batching

Towards Pig #1: Beyond relations?

- The relational data model allows us to have arbitrary numbers of relations
  - Each with its own schema that includes arbitrary numbers of attributes
- But: No nested tables!
  - These would be converted into multiple tables by 1NF normalization
  - Hence SQL has no nested collections at all, (sets, lists, bags...)
- Can we add support for these?

Towards Pig #2: Programming model

- Hadoop MapReduce:
  - file-oriented, procedural
  - regularized "pipeline" – map, combine, shuffle, reduce
  - arbitrary Java functions at each step
- SQL:
  - random access-storage-oriented (DBMS controls storage)
  - compositional, tuple-collection-oriented query model
  - declarative queries are automatically optimized
  - can accommodate Java functions, but not naturally
  - Hive: SQL queries → file-oriented Map/Reduce
- Is there something in between?
  - Declarative is nice, but many data analysts are 'entrenched' procedural programmers...
  - Pig and Pig Latin!

rigid dataflow

opacity

custom code even for very common operations

what about "procedural programmers"?

Pig Latin and Pig

- Pig Latin: a compositional, collections-oriented dataflow language
  - Oriented towards parallel data processing & analysis
  - Think of it as a more procedural SQL-like language with nested collections
    - Emphasizes user-defined functions, esp. those that have nice algebraic properties (unlike SQL)
    - Supports external data from files (like Hive)
  - By Chris Olston et al. at Yahoo! Research
    - http://www.tomkinshome.com/site_media/papers/papers/ORS+08.pdf
- Pig: the runtime system

Pig Latin: Basic constructs

- Collection-valued expressions whose results get assigned to variables
  - A program does a series of assignments in a dataflow
  - It gets compiled down to a sequence of MapReduces
    - Similar to Hive, but Pig Latin has its own query language (not SQL)

- Basic SQL-like operations are explicitly specified:
  - `load … as` [HDFS scan]
  - Remapping: `foreach … generate` [Map]
  - Filtering: `filter by` [Map]
  - Intersecting: `join` [Reduce]
  - Aggregating: `group by` [Reduce]
  - Sorting: `order` [Shuffle]
  - `store` [HDFS store]

Simple example: Face detection

- Each expression creates a named collection
  - load collections from files
  - process them (e.g., per tuple) using a user-defined function
  - store the results into files

```
I = load ‘/mydata/images’ using ImageParser()
    as (id, image);
F = foreach I generate id, detectFaces(image);
store F into ‘/mydata/faces’;
```

Example: Session Classification

- Goal: Find web sessions that end on the 'best' page (i.e., the page with the highest PageRank)
  - We need to join two tables, and then compare the final rank in the sequence to the other ranks

Visits

User	URL	Time
Alice	www.cnn.com	7:00
Alice	www.digg.com	7:20
Alice	www.social.com	10:00
Alice	www.flickr.com	10:05
Joe	www.cnn.com/index.htm	12:00

⋮

Pages

URL	PageRank
www.cnn.com	0.9
www.flickr.com	0.9
www.social.com	0.7
www.digg.com	0.2

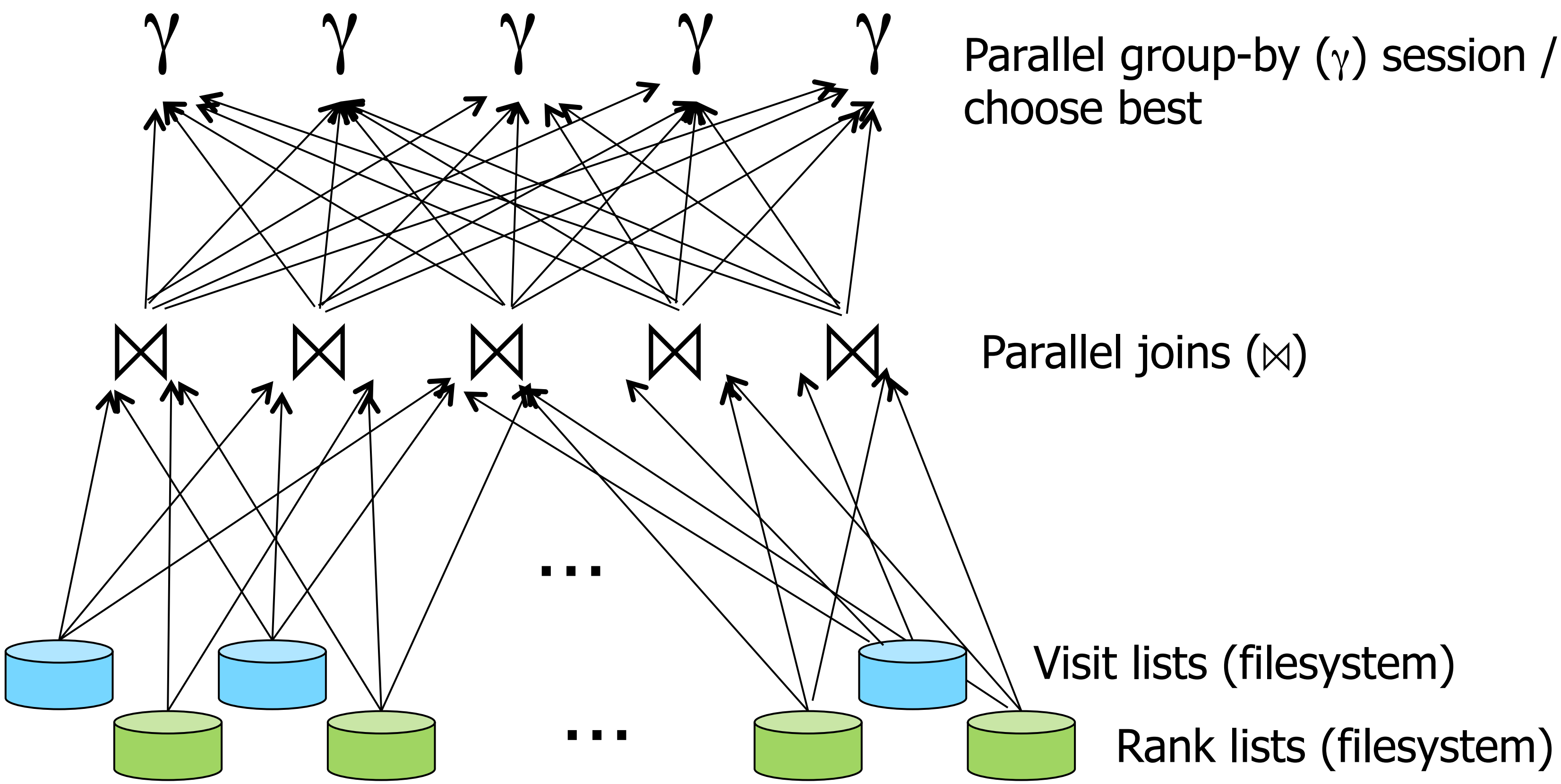
⋮

The computation in Pig Latin

```
Visits = load '/data/visits' as (user, url, time);
Visits = foreach Visits generate user, Canonicalize(url),
  time;

Pages = load '/data/pages' as (url, pagerank);

         VP = join Visits by url, Pages by url;
 UserVisits = group VP by user;
   Sessions = foreach UserVisits generate
                flatten(FindSessions(*));
HappyEndings = filter Sessions by BestIsLast(*);

      store HappyEndings into '/data/happy_endings';
```

What does this query compile to?

- Parallel evaluation is really a Map-Map/Reduce/Reduce chain:



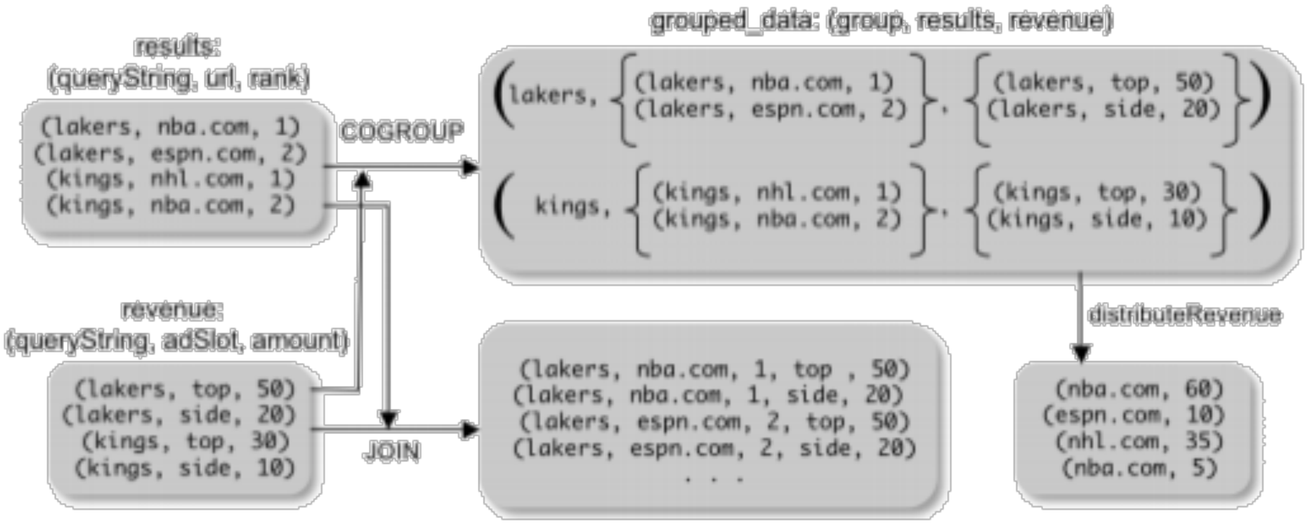
Pig Latin features

- Record-oriented transformations
  - Can work over, create nested collections
  - (Resembles Nested Relational variants of SQL)
- Basic operators expose parallelism; user-defined operators may not
- Operations are explicit, not declarative
  - Unlike SQL

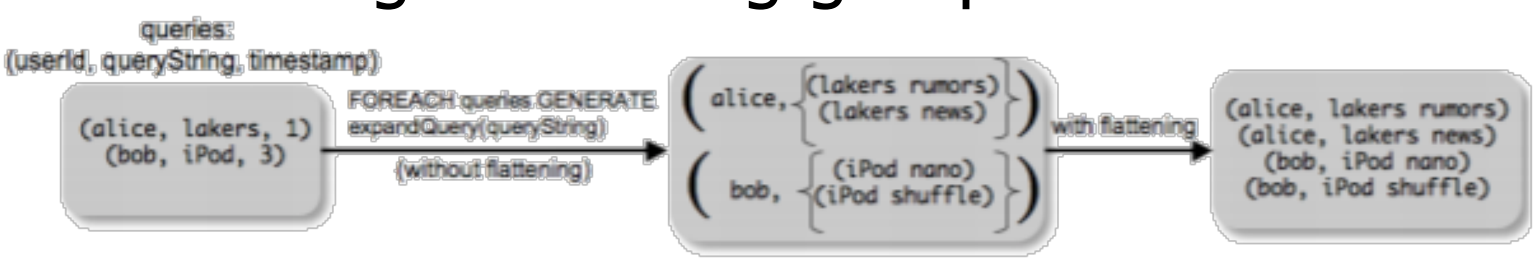
operators:	binary operators:
• FILTER	• JOIN
• FOREACH … GENERATE	• COGROUP
• GROUP	• UNION

Nesting: COGROUP & FLATTEN

- Cogrouping: nesting groups into columns



- Flattening: unnesting groups



Pig Latin vs. MapReduce

- MapReduce combines 3 primitives:
  process records → create groups → process groups

```
a = FOREACH input GENERATE flatten(Map(*));
b = GROUP a BY $0;
c = FOREACH b GENERATE Reduce(*);
```

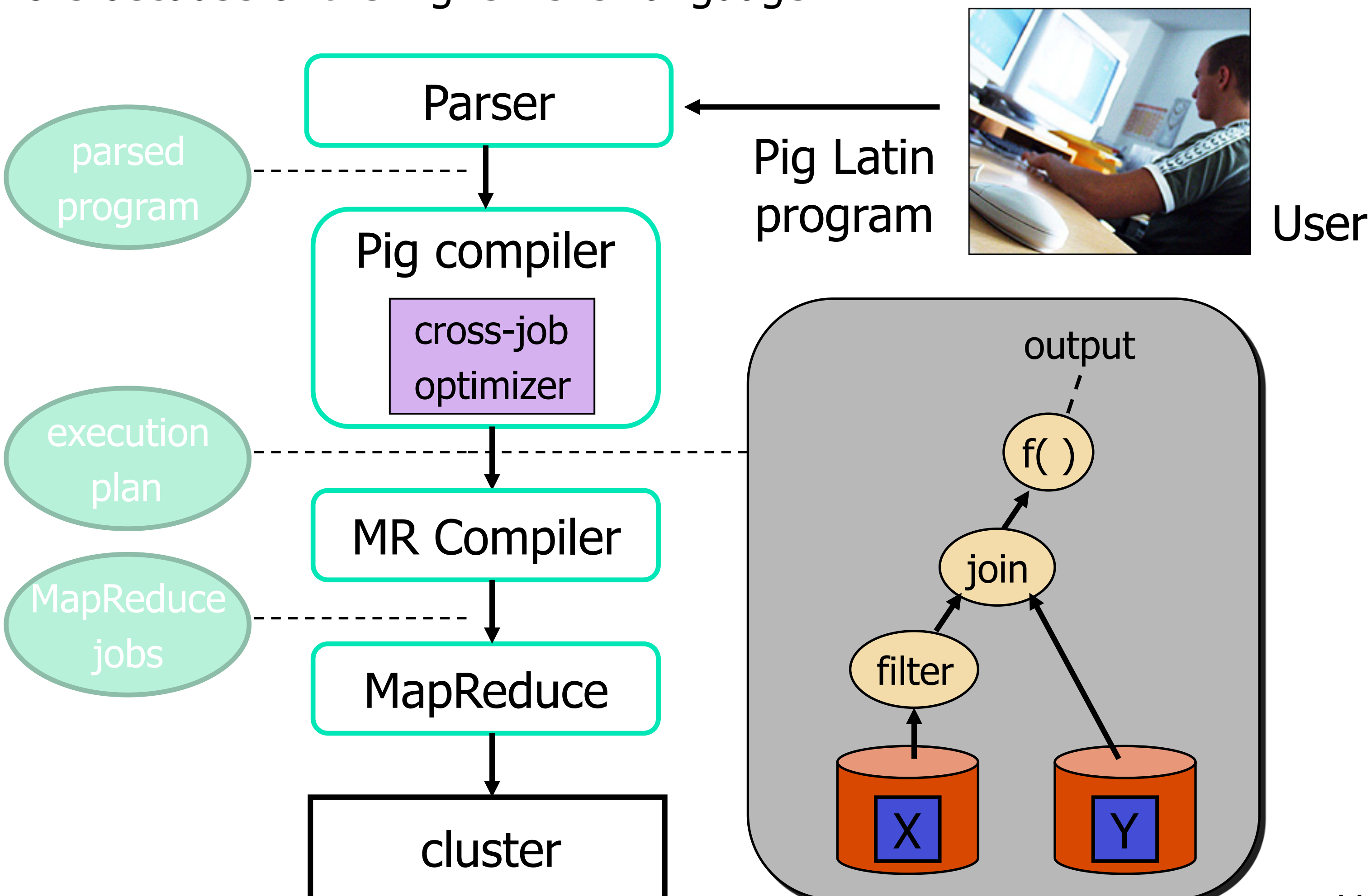
- In Pig, these primitives are:
  - explicit
  - independent
  - fully composable
- Pig adds primitives for:
  - filtering tables
  - projecting tables
  - combining 2 or more tables

Recap: Pig Latin

- A dataflow language that compiles to MapReduce
  - Borrows many of the elements of SQL, but eliminates the reliance on declarative optimization
  - Incorporates primitives for nested collections
- Quite successful:
  - As of 2008: 25% of Yahoo Map/Reduce jobs from Pig
  - Part of the Hadoop standard distribution

Pig system implementation

- Let’s briefly look at the Pig implementation, and how it can do a bit more because of the higher-level language:

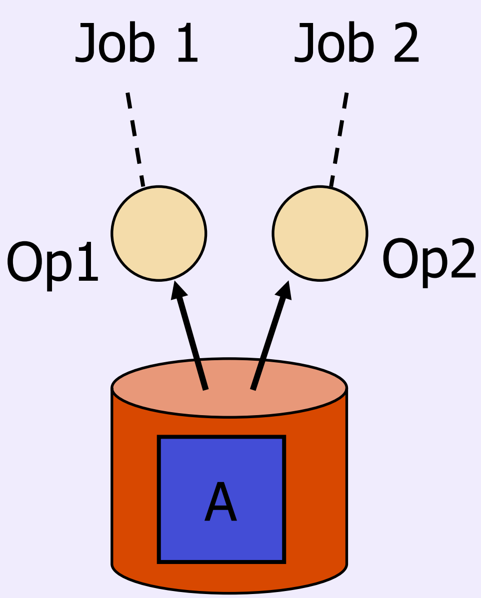


Key issue: Minimizing redundancy

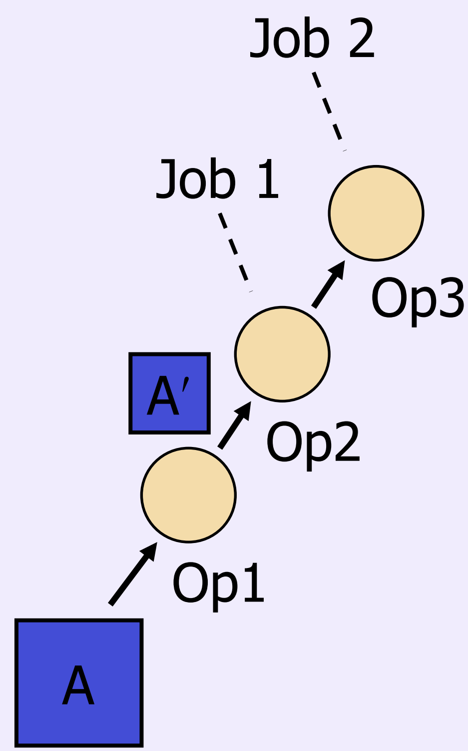
- Popular tables
  - web crawl
  - search log
- Popular transformations
  - eliminate spam pages
  - group pages by host
  - join web crawl with search log
- Goal: Minimize redundant work

Work-sharing techniques

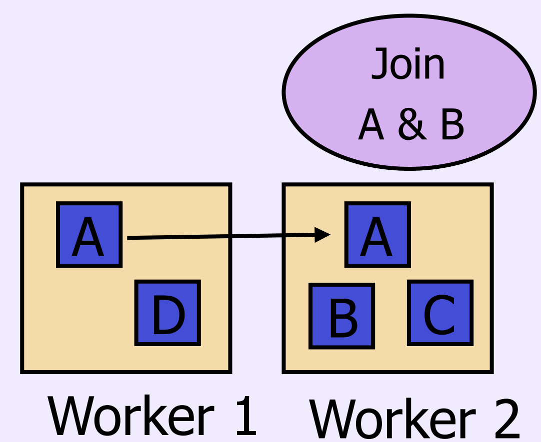
execute similar jobs together



cache data transformations



cache data moves



Recap: Pig and Pig Latin

- Somewhere between a programming language and a DBMS

- Allows distributed programming with explicit parallel dataflow operators

- Supports explicit management of nested collections

- Runtime system does caching and batching

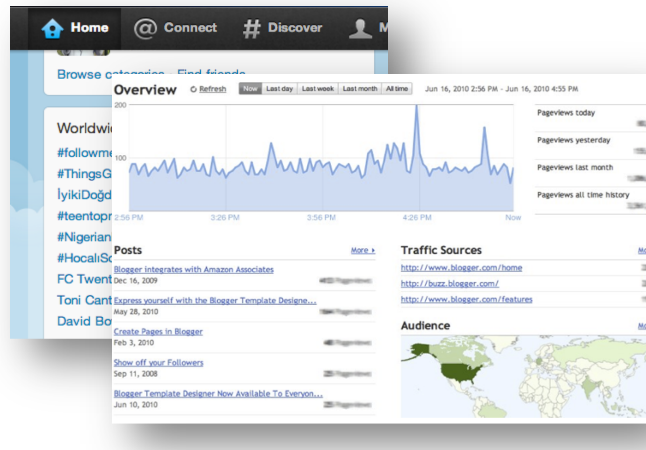
Plan for today

- Beyond MapReduce
- Abstractions for iterative batch-processing
  - Pregel: Bulk Synchronous Parallel for Graphs
  - Spark: In-Memory Resilient Distributed Datasets
- Higher-level languages for Hadoop
  - Hive Query Language
  - Pig and Pig Latin
- Stream processing NEXT
  - Storm: One-record at a time
  - 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

  …

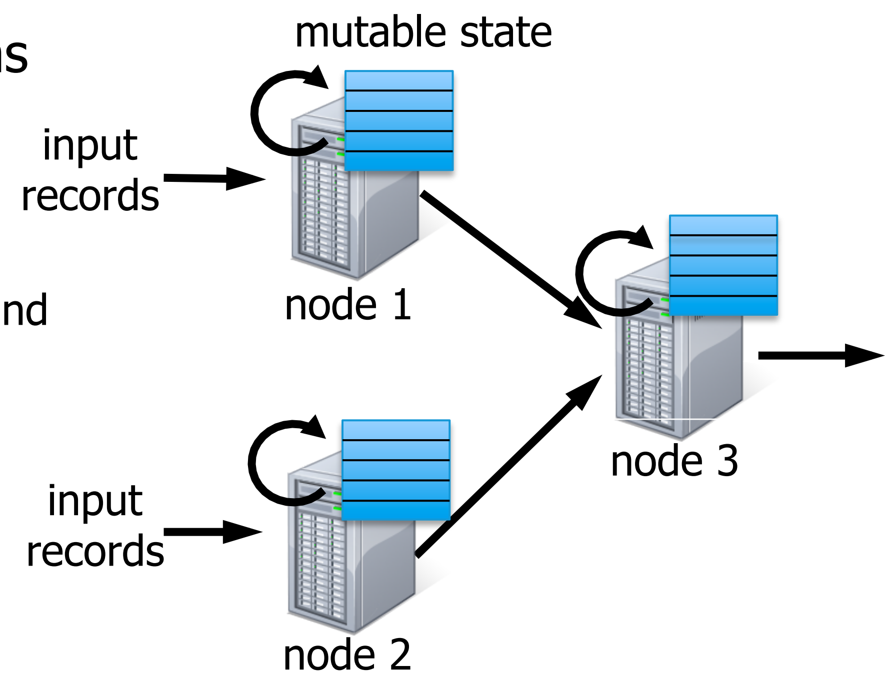


- Distributed stream processing framework is required to
  - Scale to large clusters (100s of machines)
  - Achieve low latency (few seconds)

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

Stateful Stream Processing

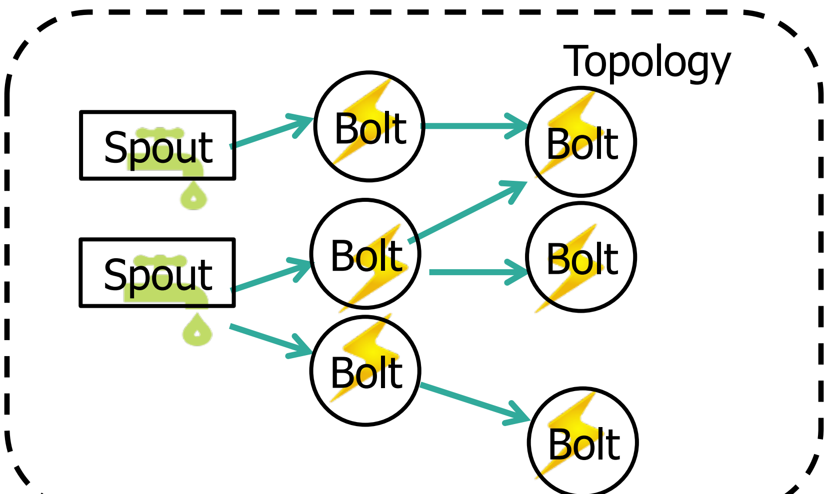
- Traditional streaming systems have a record-at-a-time processing model
  - Each node has mutable state
  - For each record, update state and send new records
- State is lost if node dies!
- Making stateful stream processing be fault-tolerant is challenging



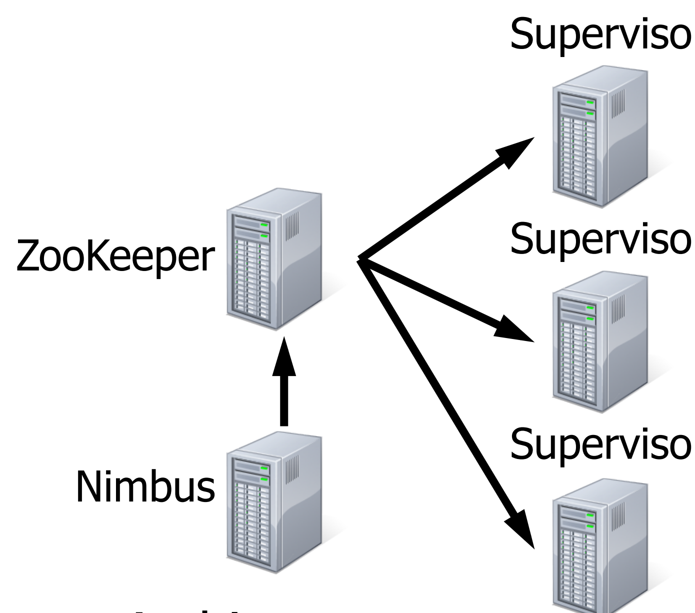
Contents from T. Das talk: Spark Streaming, AMPCamp'13

Apache Storm

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



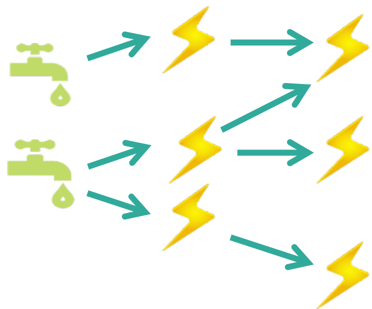
Programming Model



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:

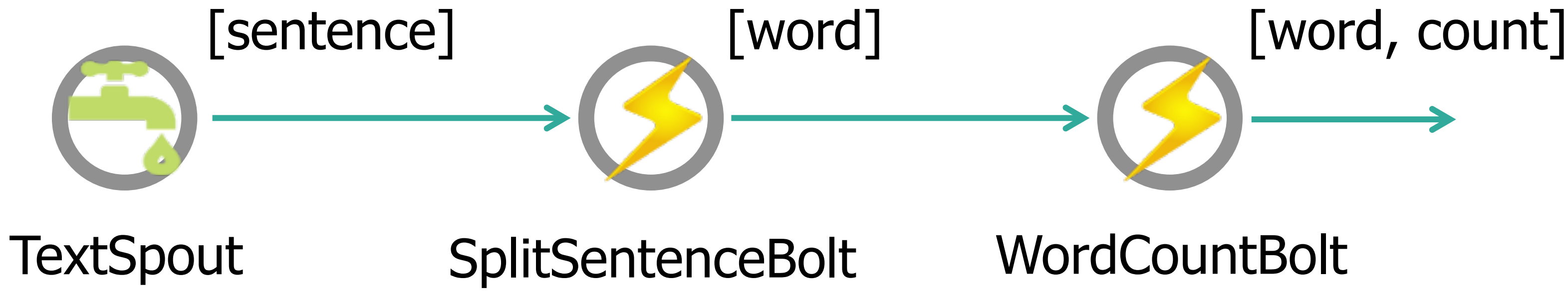
- Nimbus node (master node)
  - Similar to the Hadoop JobTracker
  - Distributes code, launches workers across the cluster
  - Monitors computation and reallocates workers as needed
- ZooKeeper nodes – (coordinate the cluster)
  - Will discuss ZooKeep in detail in a later lecture
- Supervisor nodes
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

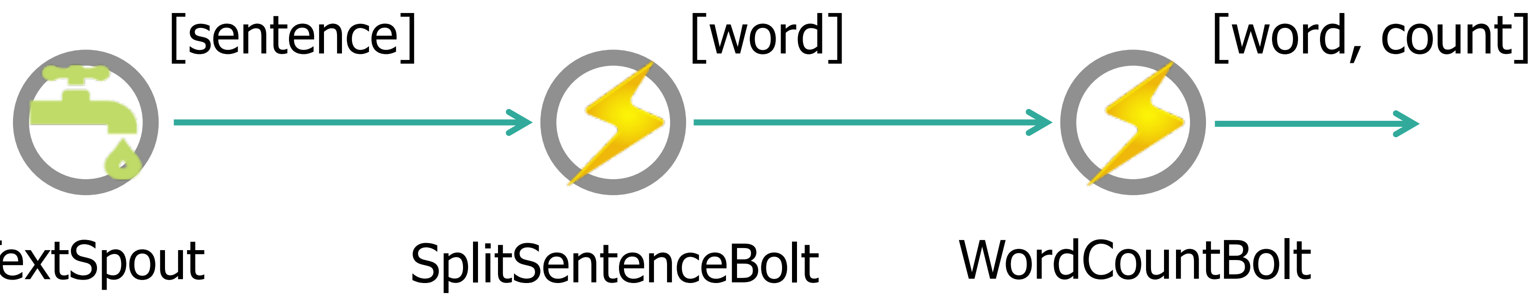
Contents from P. T. Goetz talk: Apache Storm and Spark Streaming Compared

Example: Word Count




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

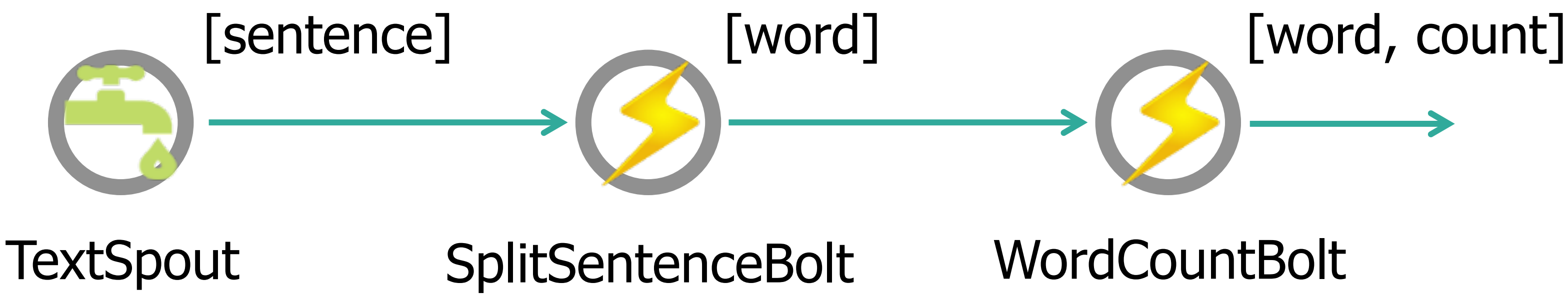
Example: Word Count




```
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();
} }
```

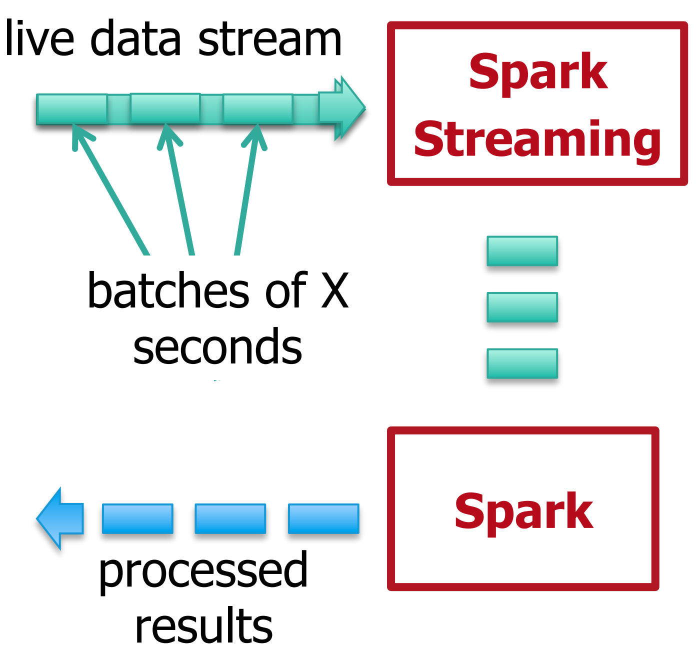
Plan for today

- Beyond MapReduce
- Abstractions for iterative batch-processing
  - Pregel: Bulk Synchronous Parallel for Graphs
  - Spark: In-Memory Resilient Distributed Datasets
- Higher-level languages for Hadoop
  - Hive Query Language
  - Pig and Pig Latin
- Stream processing
  - Storm: One-record at a time
  - Spark Streaming: Micro-batching NEXT

Spark Streming

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

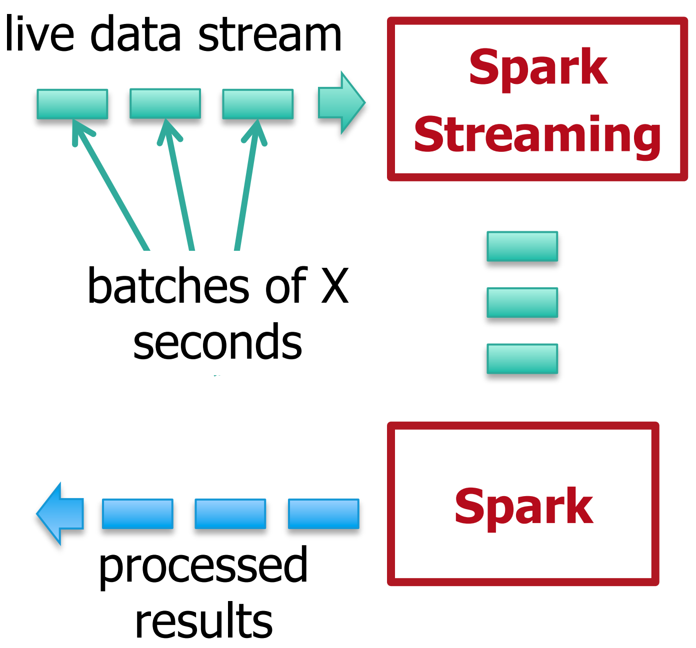


Contents from T. Das talk: Spark Streaming, AMPCamp'13

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

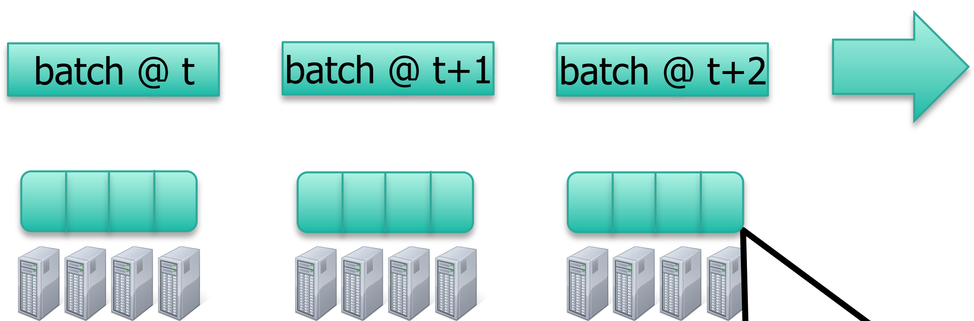
Example: Get hashtags from Twitter

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

DStream: a sequence of RDDs representing a stream of data

Twitter Streaming API

tweets DStream

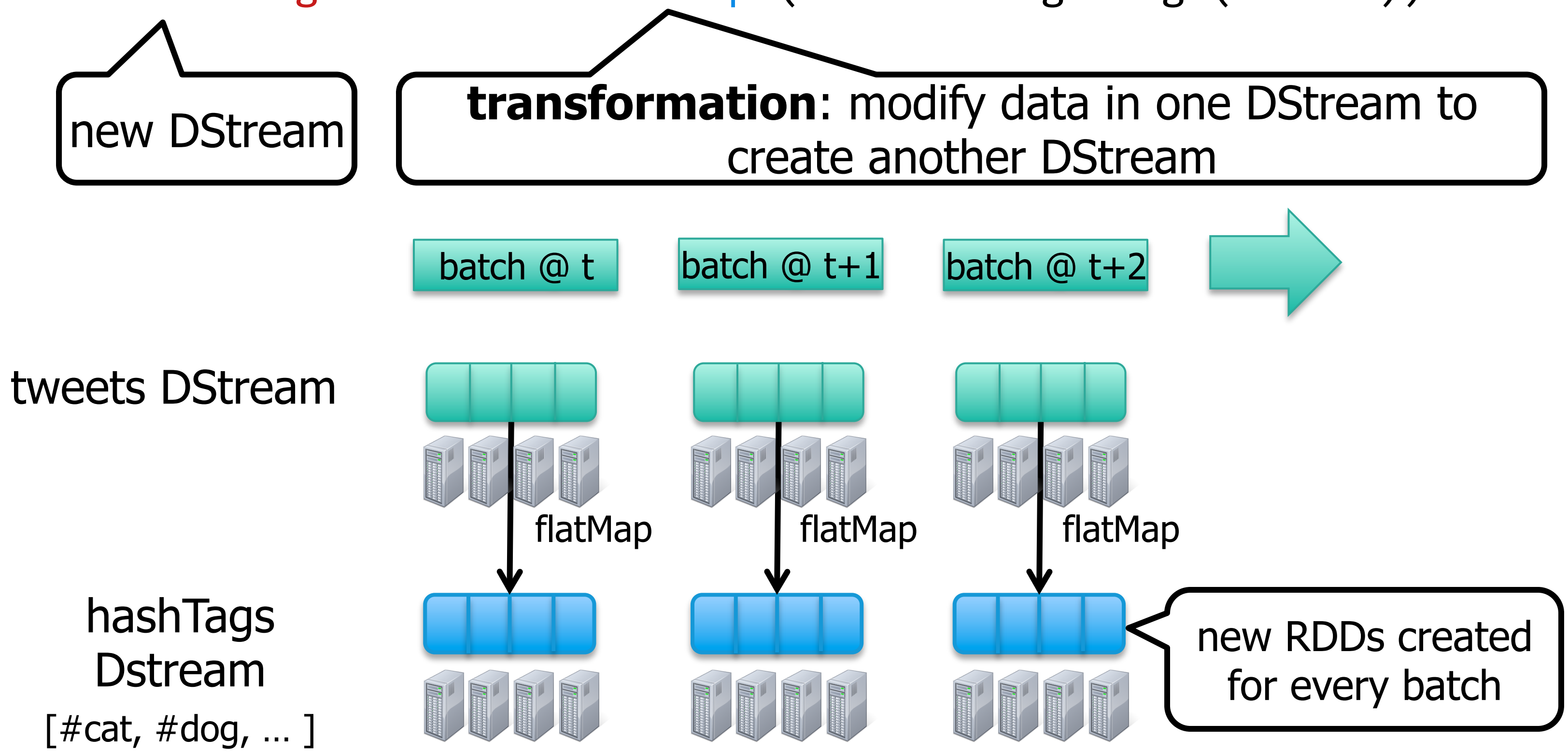


stored in memory as an RDD (immutable, distributed)

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))
```

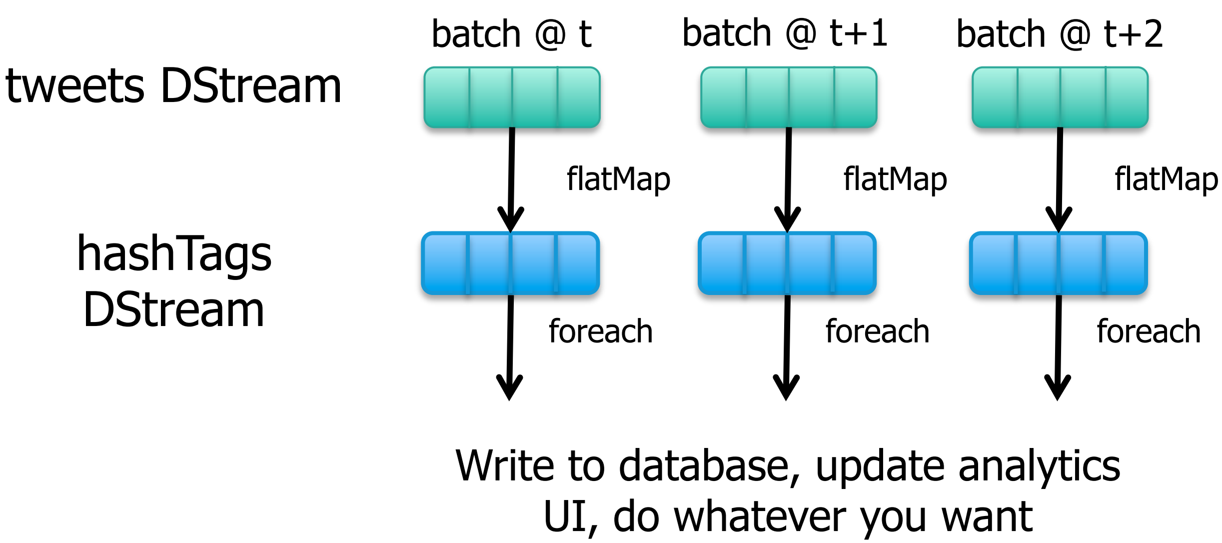


Contents from T. Das talk: Spark Streaming, AMPCamp'13

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



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

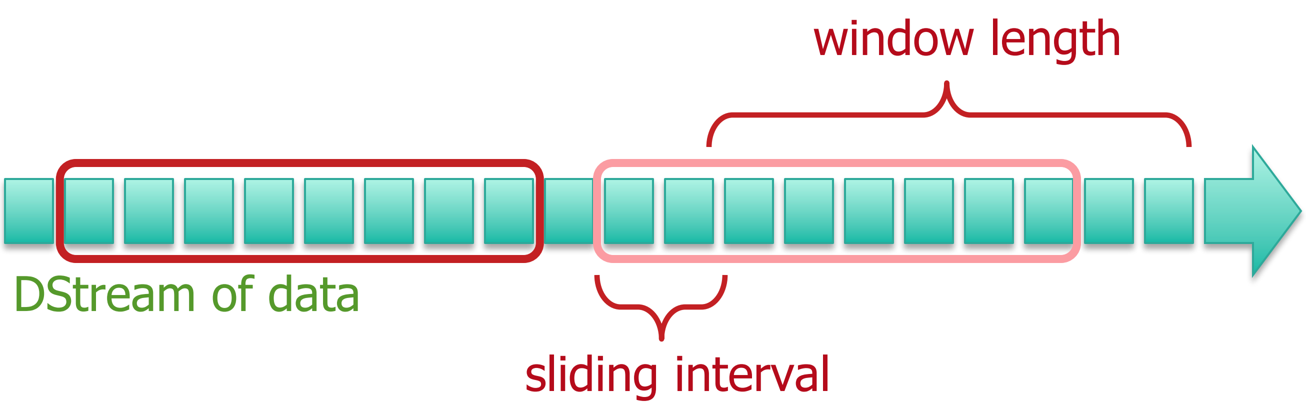
Window-based Transformations

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

sliding window operation

window length

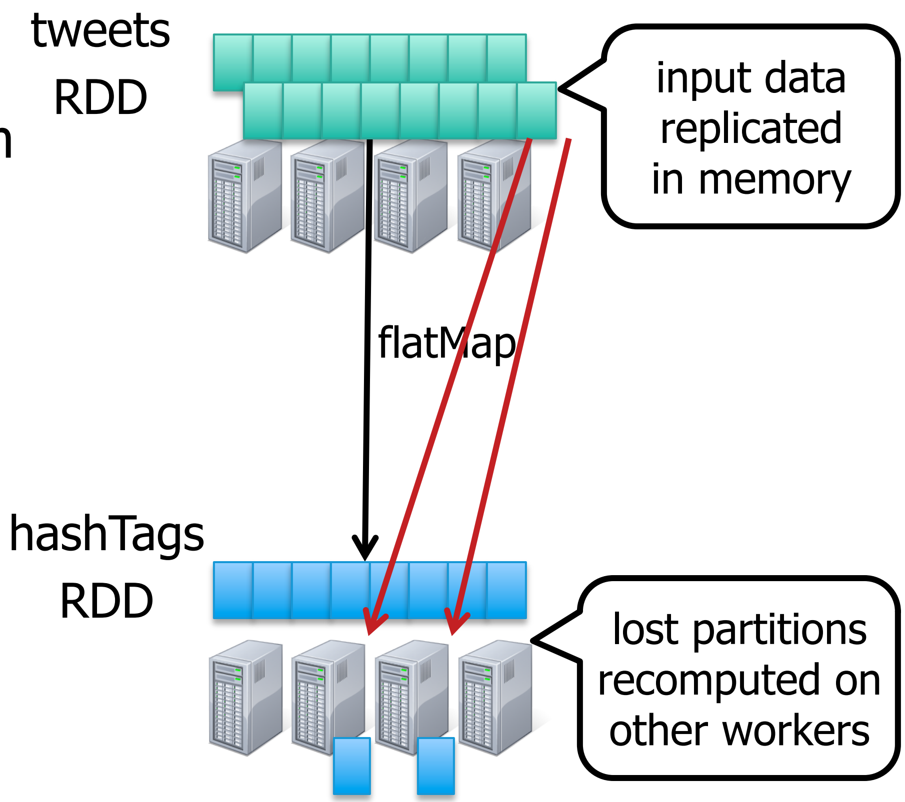
sliding interval



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 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 storage