INFSCI 2750: Cloud Computing

Lecture 4: MapReduce

Dr. Balaji Palanisamy
Associate Professor
School of Computing and Information
University of Pittsburgh
bpalan@pitt.edu

Slides Courtesy: Prof. Keke Chen, Wright State University
Tanenbaum & Van Steen, Distributed Systems: Principles and Paradigms, 2e, (c) 2007

Pig Latin: A Not-So-Foreign Language For Data Processing

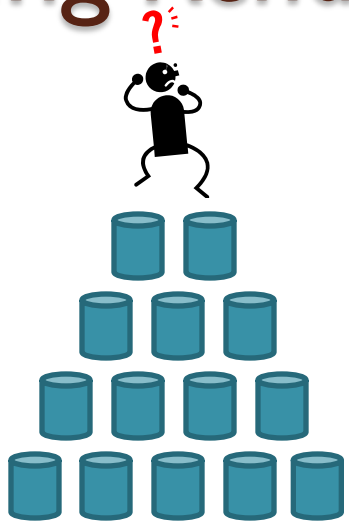
Chris Olston Benjamin Reed

Utkarsh Srivastava

Ravi Kumar Andrew Tomkins

Y! Research

Data Processing *Renaissance*



- Internet companies swimming in data
  - E.g. TBs/day at Yahoo!
- Data analysis is “inner loop” of product innovation
- Data analysts are skilled programmers

Data Warehousing …?

Scale

Often not scalable enough

$ $ $ $

Prohibitively expensive at web scale

- Up to $200K/TB

SQL

- Little control over execution method
- Query optimization is hard
  - Parallel environment
  - Little or no statistics
  - Lots of UDFs

New Systems For Data Analysis

- Map-Reduce 
- Apache Hadoop   . . .
- ~~Dryad~~ 

The Map-Reduce Appeal

Scale

Scalable due to simpler design

- Only parallelizable operations
- No transactions

$

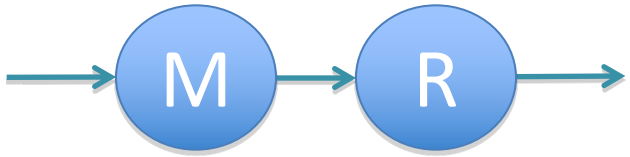
Runs on cheap commodity hardware

~~SQL~~

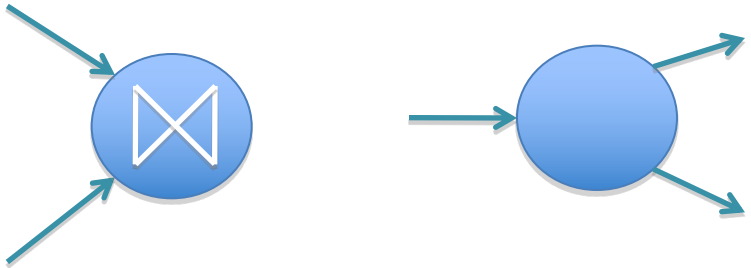
Procedural Control- a processing “pipe”

Disadvantages

1. Extremely rigid data flow

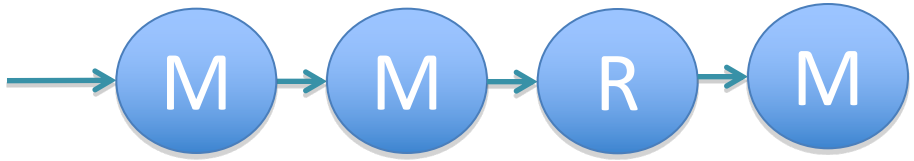


Other flows constantly hacked in



Join, Union

Split



Chains

2. Common operations must be coded by hand
   - Join, filter, projection, aggregates, sorting, distinct
3. Semantics hidden inside map-reduce functions
   - Difficult to maintain, extend, and optimize

Outline

- Map-Reduce and the need for Pig Latin
- Pig Latin example
- Salient features
- Implementation

Example Data Analysis Task

Find the top 10 most visited pages in each category

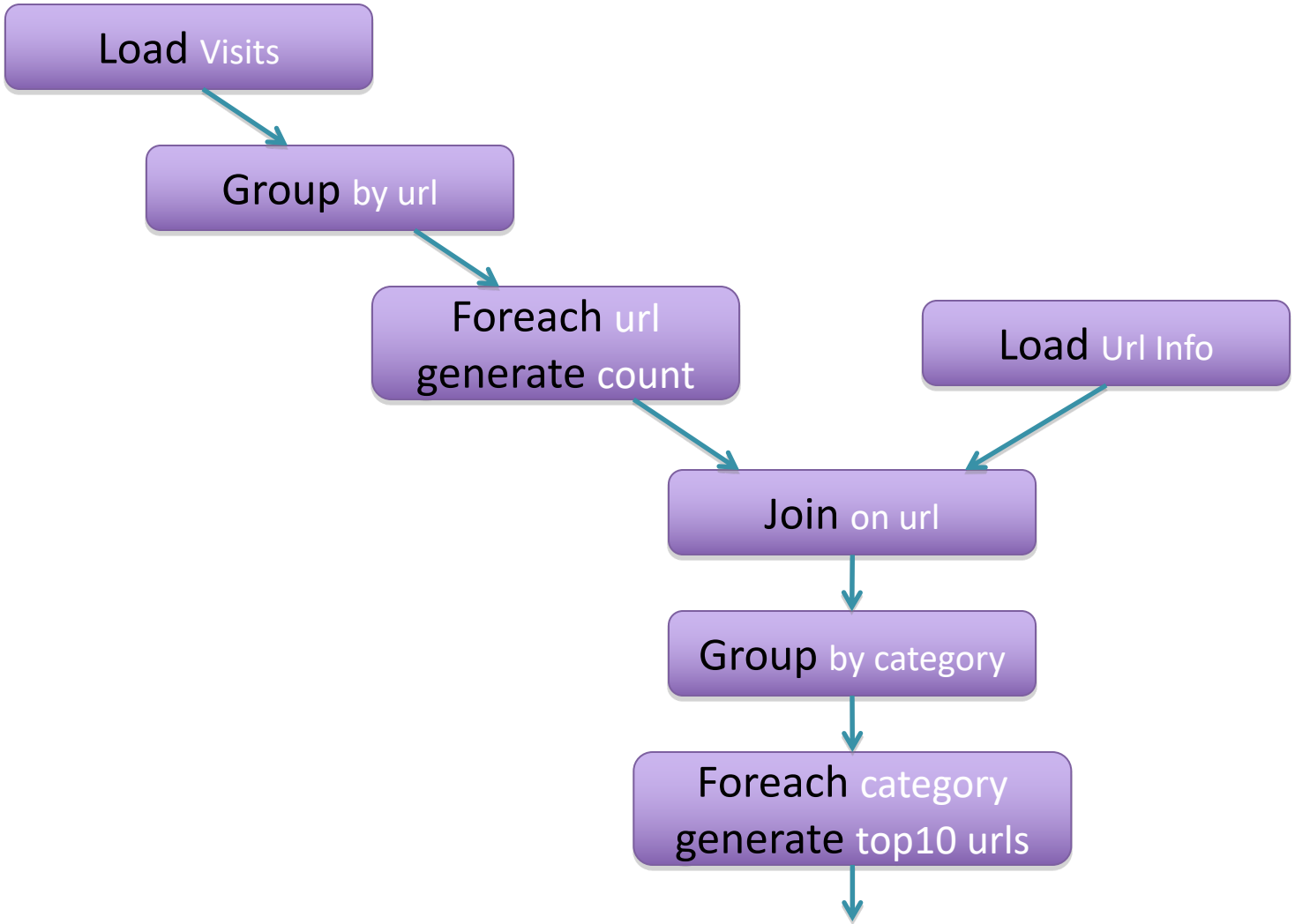
Visits

User	Url	Time
Amy	cnn.com	8:00
Amy	bbc.com	10:00
Amy	flickr.com	10:05
Fred	cnn.com	12:00

Url Info

Url	Category	PageRank
cnn.com	News	0.9
bbc.com	News	0.8
flickr.com	Photos	0.7
espn.com	Sports	0.9

Data Flow


Load Visits
Group by url
Foreach url generate count
Load Url Info
Join on url
Group by category
Foreach category generate top10 urls

In Pig Latin

```
visits       = load '/data/visits' as (user, url, time);
gVisits      = group visits by url;
visitCounts  = foreach gVisits generate url, count(visits);

urlInfo      = load '/data/urlInfo' as (url, category, pRank);
visitCounts  = join visitCounts by url, urlInfo by url;

gCategories = group visitCounts by category;
topUrls = foreach gCategories generate
  top(visitCounts,10);

store topUrls into '/data/topUrls';
```

Outline

- Map-Reduce and the need for Pig Latin
- Pig Latin example
- Salient features
- Implementation

Quick Start and Interoperability

```
visits      = load '/data/visits' as (user, url, time);
gVisits     = group visits by url;
visitCounts = foreach gVisits generate url, count(urlVisits);

urlInfo     = load '/data/urlInfo' as (url, category, pRank);

visitCounts = join visitCounts by url, urlInfo by url;
gCategorie                                          y;
topUrls =
  top(visit                                         );

store topUrls into '/data/topUrls';
```

Operates directly over files

Quick Start and Interoperability

```
visits       = load '/data/visits' as (user, url, time);
gVisits      = group visits by url;
visitCounts  = foreach gVisits generate url, count(urlVisits);

urlInfo      = load '/data/urlInfo' as (url, category, pRank);

visitCounts = join visitCounts by url, urlInfo by url;
gCategorie
topUrls =
  top(visit

store topUrls into '/data/topUrls';
```

Schemas optional;
Can be assigned dynamically

User-Code as a First-Class Citizen

```
visits      = load '/data/visits' as (user, url, time);
gVisits
visitCo                                    , count(urlVisits);

urlInfo                                    ategory, pRank);

visitCounts  = join visitCounts by url, urlInfo by url;
gCategories = group visitCounts by category;
topUrls = foreach gCategories generate
   top(visitCounts,10);

store topUrls into '/data/topUrls';
```

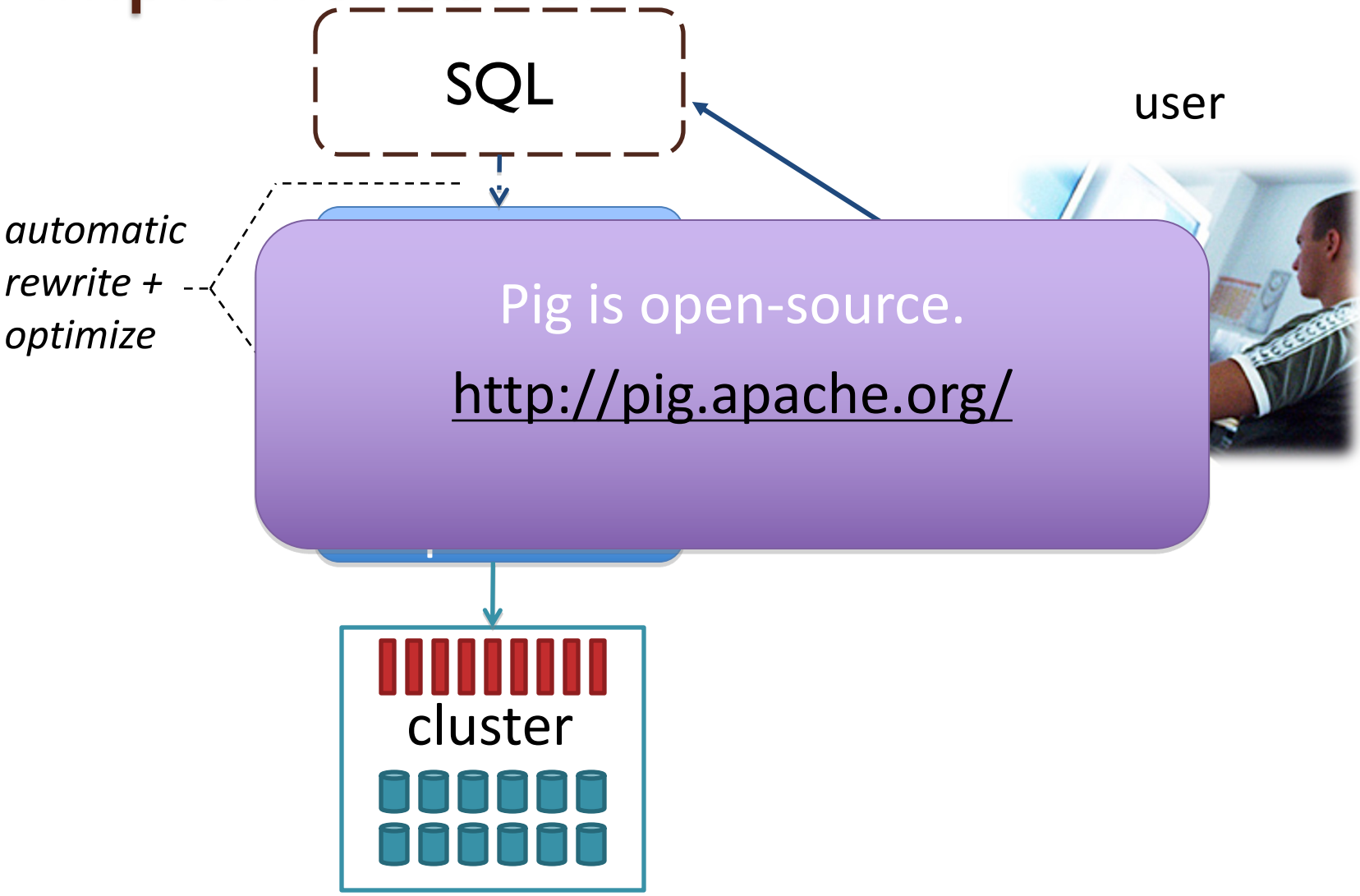
User-defined functions (UDFs) can be used in every construct

- Load, Store
- Group, Filter, Foreach

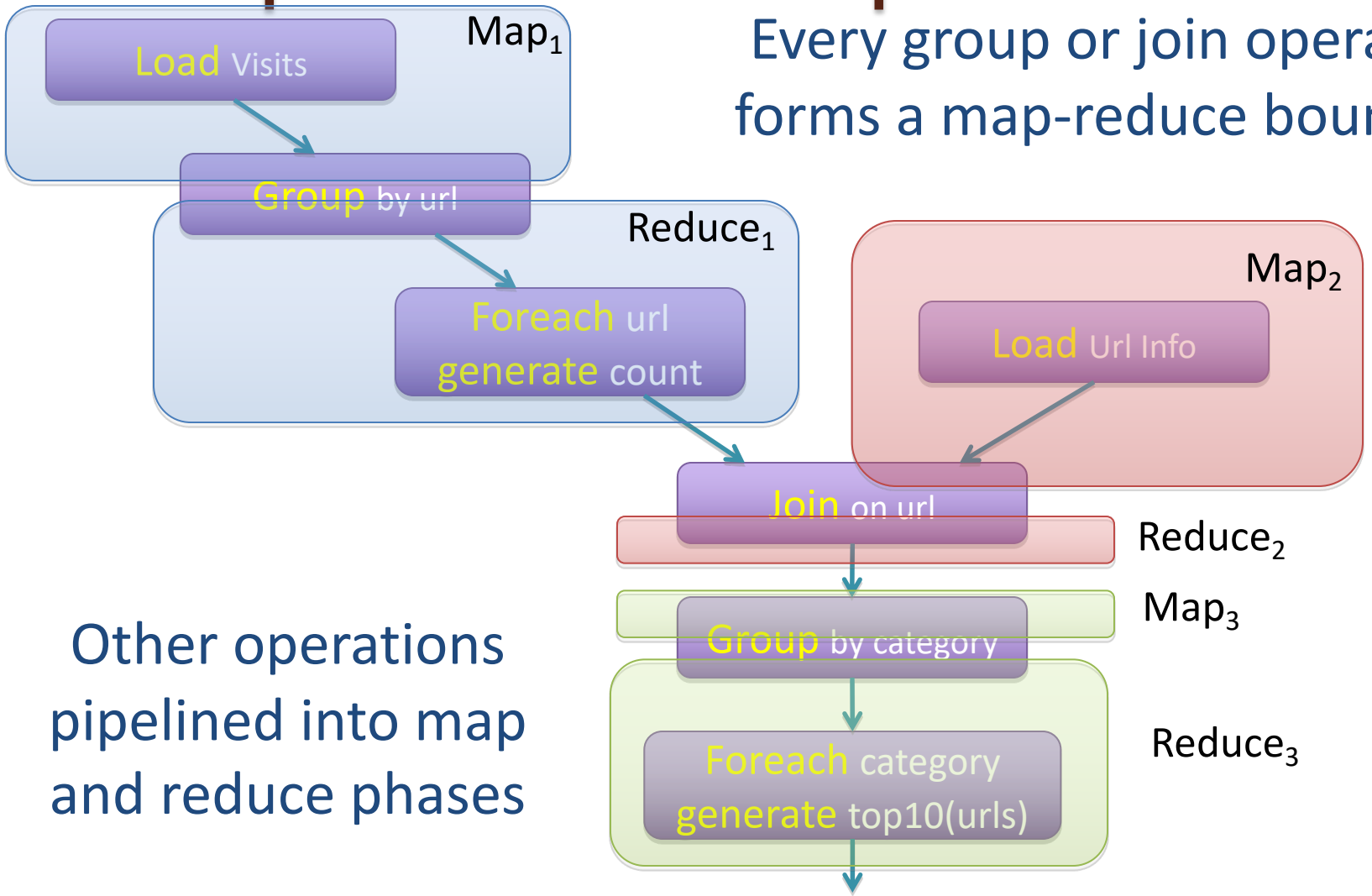
Outline

- Map-Reduce and the need for Pig Latin
- Pig Latin example
- Novel features
- Implementation

Implementation



Compilation into Map-Reduce



Every group or join operation forms a map-reduce boundary

Other operations pipelined into map and reduce phases

Compilation

- Pig system does two tasks:
  - Builds a Logical Plan from a Pig Latin script
    - Supports execution platform independence
    - No processing of data performed at this stage
  - Compiles the Logical Plan to a Physical Plan and Executes
    - Convert the Logical Plan into a series of Map-Reduce statements to be executed (in this case) by Hadoop Map-Reduce

Compilation

- Building a Logical Plan
  - Verify input files and bags referred to are valid
  - Create a logical plan for each bag(variable) defined

Compilation

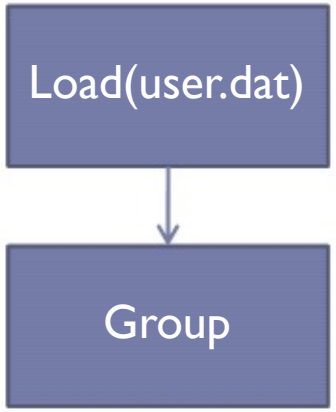
- Building a Logical Plan Example

```
A = LOAD 'user.dat' AS (name, age, city);
B = GROUP A BY city;
C = FOREACH B GENERATE group AS city,
       COUNT(A);
D = FILTER C BY city IS 'kitchener'
       OR city IS 'waterloo';
STORE D INTO 'local_user_count.dat';
```

Compilation

- Building a Logical Plan Example

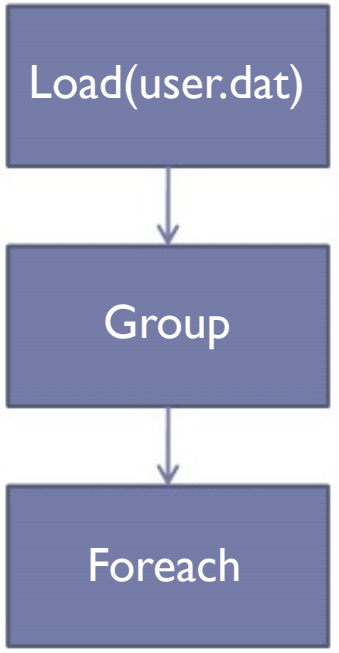
```
A = LOAD 'user.dat' AS (name, age, city);
B = GROUP A BY city;
C = FOREACH B GENERATE group AS city,
       COUNT(A);
D = FILTER C BY city IS 'kitchener'
       OR city IS 'waterloo';
STORE D INTO 'local_user_count.dat';
```



Compilation

- Building a Logical Plan Example

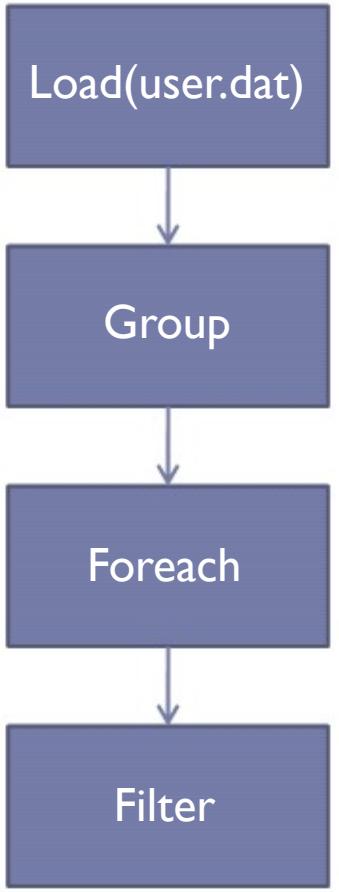
```
A = LOAD 'user.dat' AS (name, age, city);
B = GROUP A BY city;
C = FOREACH B GENERATE group AS city,
      COUNT(A);
D = FILTER C BY city IS 'kitchener'
      OR city IS 'waterloo';
STORE D INTO 'local_user_count.dat';
```



Compilation

- Building a Logical Plan Example

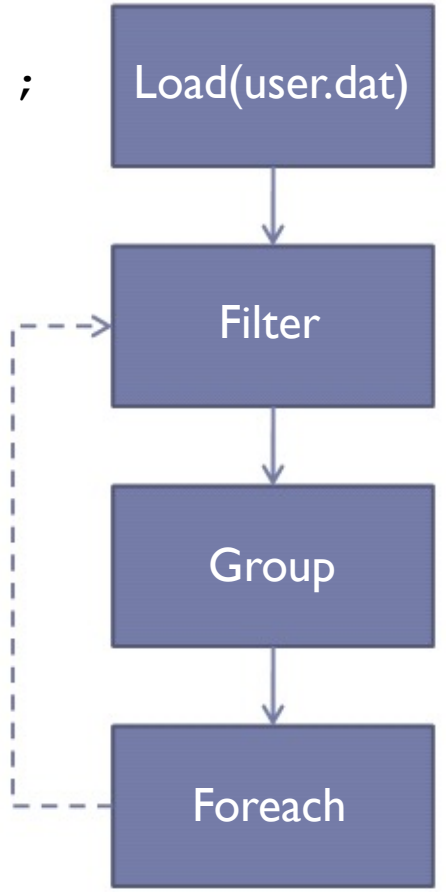
```
A = LOAD 'user.dat' AS (name, age, city);
B = GROUP A BY city;
C = FOREACH B GENERATE group AS city,
       COUNT(A);
D = FILTER C BY city IS 'kitchener'
       OR city IS 'waterloo';
STORE D INTO 'local_user_count.dat';
```



Compilation

- Building a Logical Plan Example

```
A = LOAD 'user.dat' AS (name, age, city);
B = GROUP A BY city;
C = FOREACH B GENERATE group AS city,
       COUNT(A);
D = FILTER C BY city IS 'kitchener'
       OR city IS 'waterloo';
STORE D INTO 'local_user_count.dat';
```

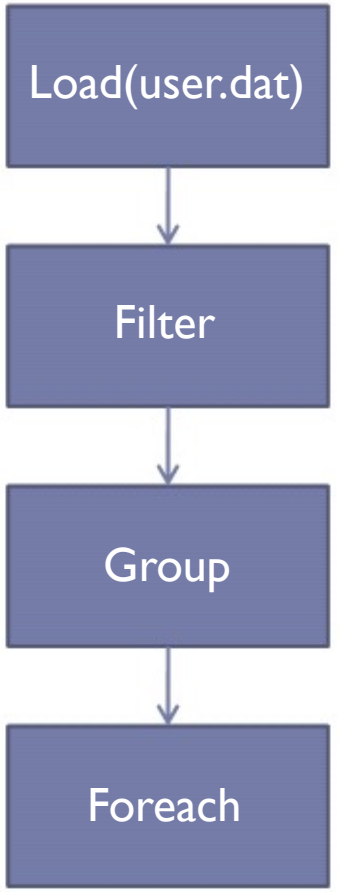


Compilation

- Building a Physical Plan

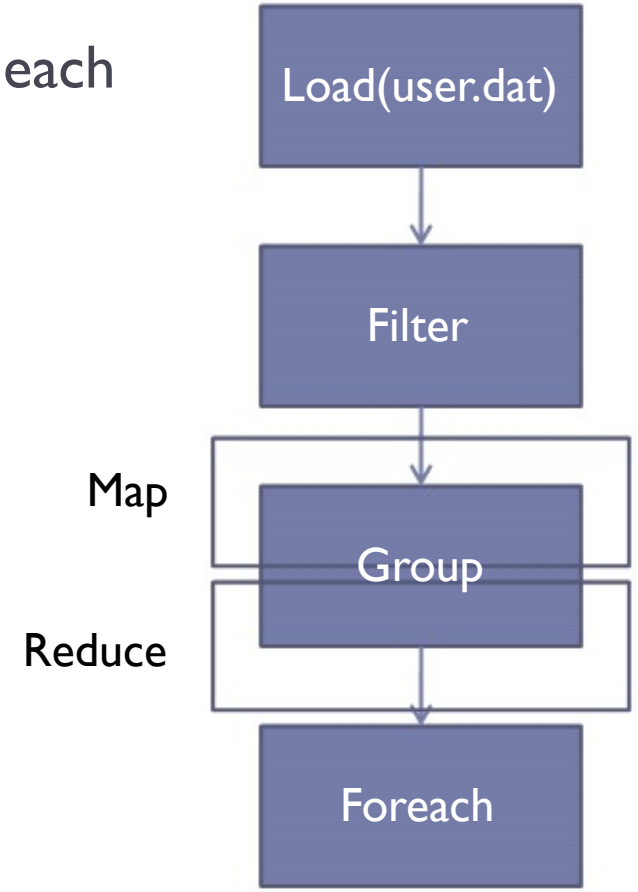
```
A = LOAD 'user.dat' AS (name, age, city);
B = GROUP A BY city;
C = FOREACH B GENERATE group AS city,
       COUNT(A);
D = FILTER C BY city IS 'kitchener'
       OR city IS 'waterloo';
STORE D INTO 'local_user_count.dat';
```

Only happens when output is specified by STORE or DUMP



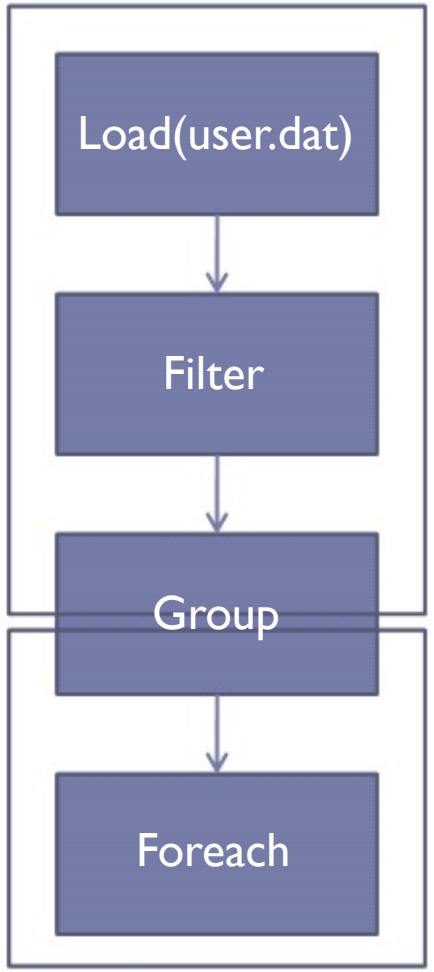
Compilation

- Building a Physical Plan
  - Step 1: Create a map-reduce job for each COGROUP



Compilation

- Building a Physical Plan
  - Step 1: Create a map-reduce job for each COGROUP
  - Step 2: Push other commands into the map and reduce functions where possible
  - May be the case certain commands require their own map-reduce job (ie: ORDER needs separate map-reduce jobs)

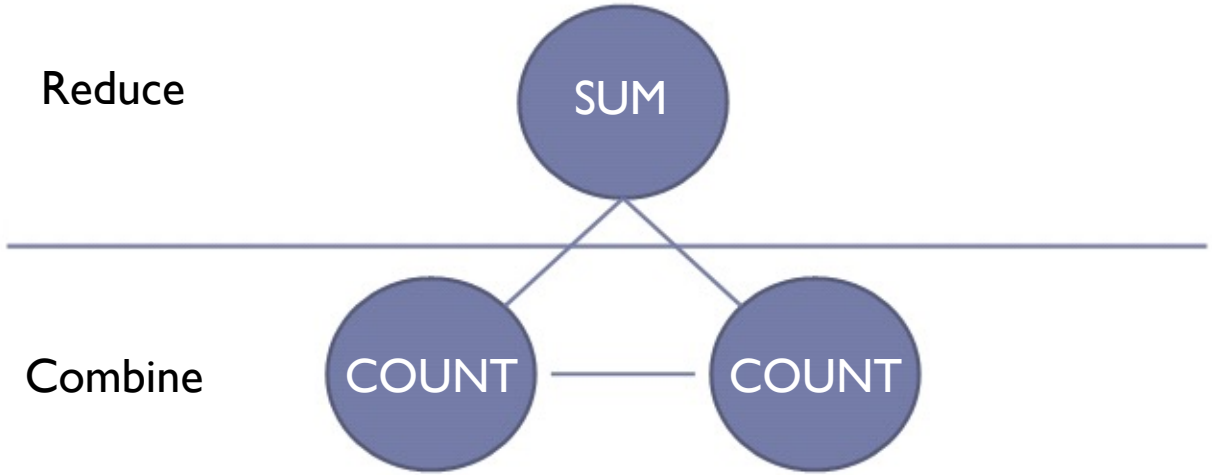


Compilation

- Efficiency in Execution
  - Parallelism
    - Loading data - Files are loaded from HDFS
    - Statements are compiled into map-reduce jobs

Compilation

- Efficiency with Nested Bags
  - Why this works:
    - COUNT is an algebraic function; it can be structured as a tree of sub-functions with each leaf working on a subset of the data



Compilation

- Efficiency with Nested Bags
  - Pig provides an interface for writing algebraic UDFs so they can take advantage of this optimization as well.
- Inefficiencies
  - Non-algebraic aggregate functions (ie: MEDIAN) need entire bag to materialize; may cause a very large bag to spill to disk if it doesn"t fit in memory
  - Every map-reduce job requires data be written and replicated to the HDFS (although this is offset by parallelism achieved)

Hive

- Developed by facebook (open source)
- Mimic SQL language
- Built on hadoop/mapreduce

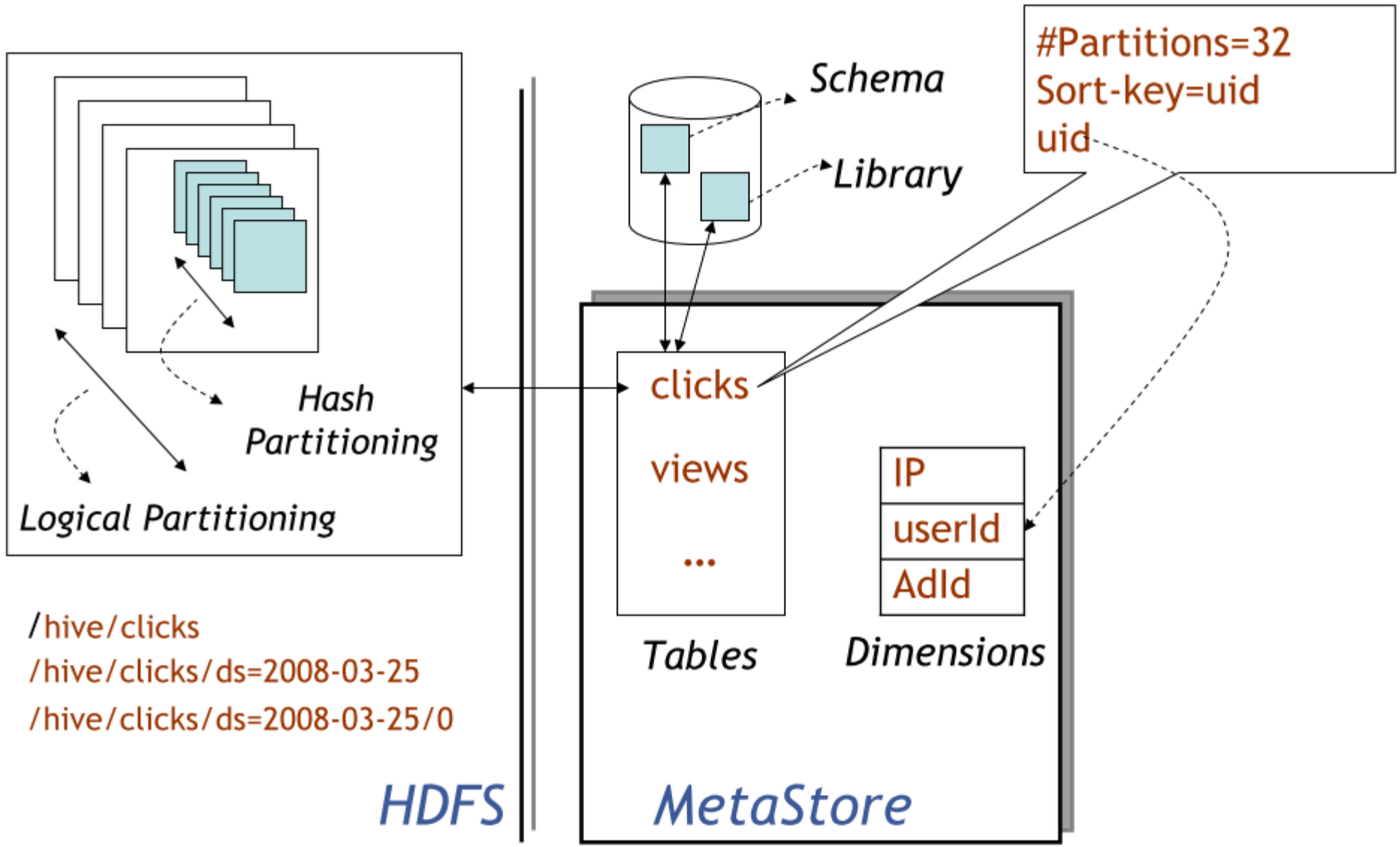
Hive data model: table etc.

- Table
  - Similar to DB table
  - stored in hadoop directories
  - Builtin compression, serialization/deserialization
- Partitions
  - Groups in the table
  - Subdirectory in the table directory
- Buckets
  - Files in the partition directory
  - Key (column) based partition
    - /table/partition/bucket1

Hive data model: Column type

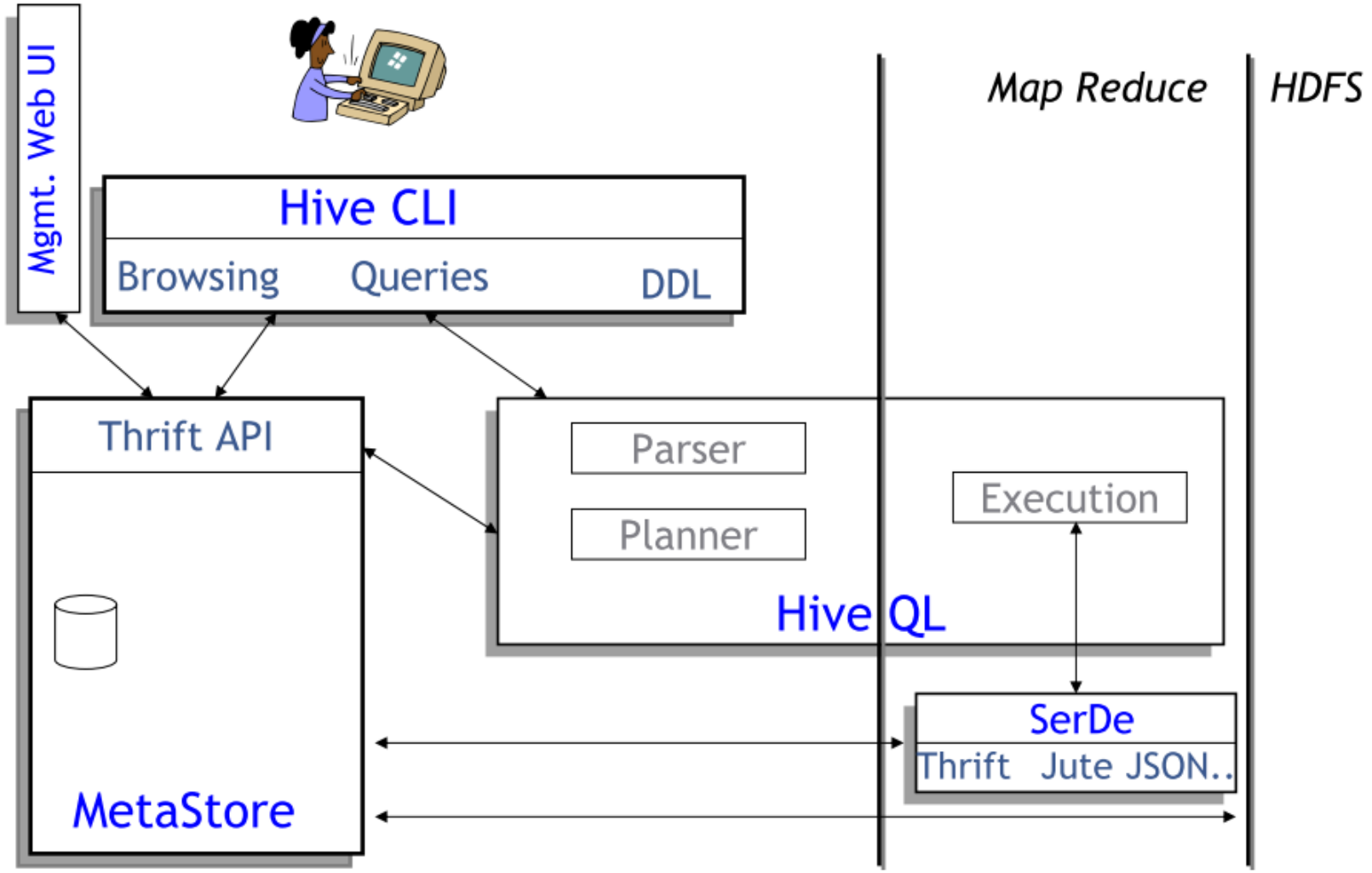
- integers, floating point numbers, generic strings, dates and booleans
- nestable collection types: array and map.

Data Model


#Partitions=32
Sort-key=uid
uid
Schema
Library
Hash
Partitioning
Logical Partitioning
clicks
views
...
IP
userId
AdId
Tables
Dimensions
/hive/clicks
/hive/clicks/ds=2008-03-25
/hive/clicks/ds=2008-03-25/0
HDFS
MetaStore

Architecture

Metastore stores the schema of databases. It uses non HDFS data store



Query processing

- Steps (similar to DBMS)
  - Parse
  - Semantic analyzer
  - Logical plan generator (algebra tree)
  - Optimizer
  - Physical plan generator (to mapreduce jobs)

Operations: DDL and DML

- HiveQL: SQL like, with slightly different syntax
- User defined filtering and aggregation functions
  - Java only
- Map/reduce plugin for streaming process
  - Implemented with any language

Example

```
FROM (SELECT a.status, b.school, b.gender
      FROM status_updates a JOIN profiles b
           ON (a.userid = b.userid and
               a.ds='2009-03-20' )
      ) subq1
INSERT OVERWRITE TABLE gender_summary
                       PARTITION(ds='2009-03-20')
SELECT subq1.gender, COUNT(1) GROUP BY subq1.gender
INSERT OVERWRITE TABLE school_summary
                       PARTITION(ds='2009-03-20')
SELECT subq1.school, COUNT(1) GROUP BY subq1.school
```

More query examples

- Multi table inserts

```
FROM ad_impressions_stg imps
  INSERT INTO ad_legals/ds=2008-03-08 select imps.* where imps.legal = 1
  INSERT INTO ad_non_legals/ds=2008-03-08 select imps.* where imps.legal = 0
```

- Joins

```
FROM ad_impressions imps, ad_dimensions ads
  INSERT INTO ad_legals_joined select imps.*, ads.campaignid
                  JOIN ON(imps.adid, ads.adid)
                  WHERE imps.legal = 1
```

Query examples

- Group By

```
FROM ad_legals_joined imps
    INSERT INTO hdfs://hadoop001:9000/user/ads/adid_uu_summary
                select imps.adid, count_distinct(imps.uid)
                group by(imps.adid)
    INSERT INTO hdfs://hadoop001:9000/user/ads/campaignid_uu_summary
                select imps.campaign_id, count_distinct(imps.uid)
                group by(imps.campaignid)
```

Spark

In-Memory Cluster Computing for
Iterative and Interactive Applications

Matei Zaharia, Mosharaf Chowdhury, Justin Ma,
Michael Franklin, Scott Shenker, Ion Stoica

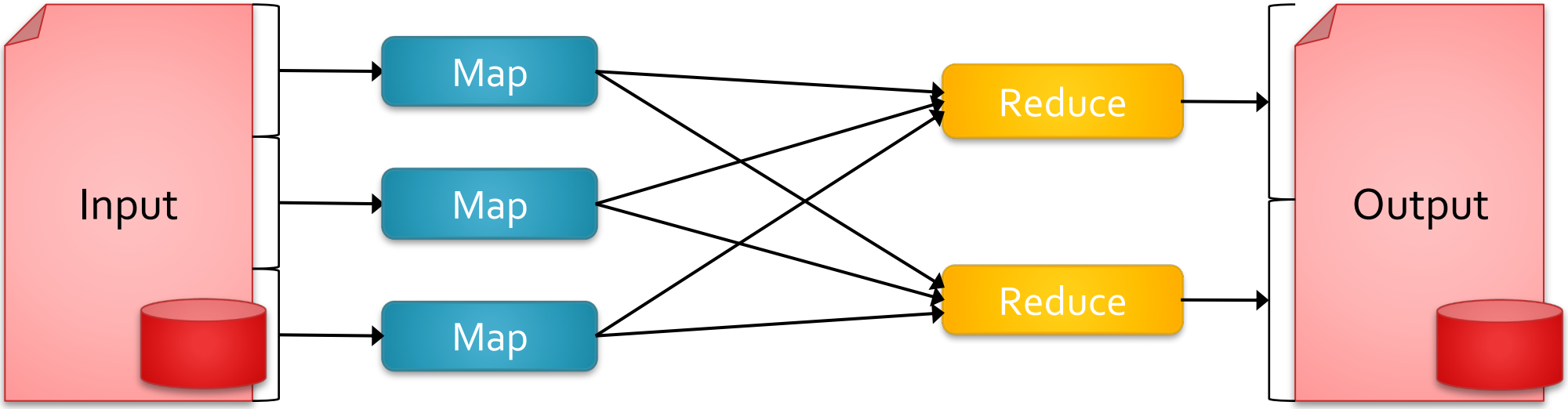
Background

- Commodity clusters have become an important computing platform for a variety of applications
  - **In industry:** search, machine translation, ad targeting, …
  - **In research:** bioinformatics, NLP, climate simulation, …
- High-level cluster programming models like MapReduce power many of these apps
- *Theme of this work: provide similarly powerful abstractions for a broader class of applications*

Motivation

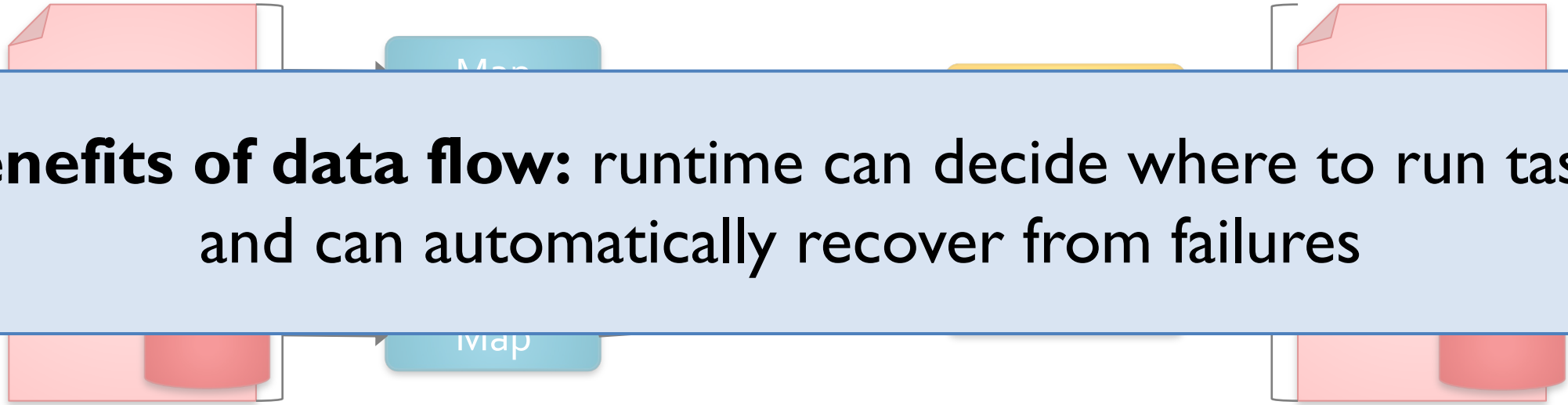
Current popular programming models for clusters transform data flowing from stable storage to stable storage

E.g., MapReduce:



Motivation

- Current popular programming models for clusters transform data flowing from stable storage to stable storage
- E.g., MapReduce:



Benefits of data flow: runtime can decide where to run tasks and can automatically recover from failures

Motivation

- Acyclic data flow is a powerful abstraction, but is not efficient for applications that repeatedly reuse a *working set* of data:
  - **Iterative** algorithms (many in machine learning)
  - **Interactive** data mining tools (R, Excel, Python)
- Spark makes working sets a first-class concept to efficiently support these apps

Spark Goal

- Provide distributed memory abstractions for clusters to support apps with working sets
- Retain the attractive properties of MapReduce:
  - Fault tolerance (for crashes & stragglers)
  - Data locality
  - Scalability

Solution: augment data flow model with "resilient distributed datasets" (RDDs)

Generality of RDDs

- We conjecture that Spark's combination of data flow with RDDs unifies many proposed cluster programming models
  - *General data flow models:* MapReduce, Dryad, SQL
  - *Specialized models for stateful apps:* Pregel (BSP), HaLoop (iterative MR), Continuous Bulk Processing
- Instead of specialized APIs for one type of app, give user first-class control of distrib. datasets

Outline

- Spark programming model
- Example applications
- Implementation
- Demo
- Future work

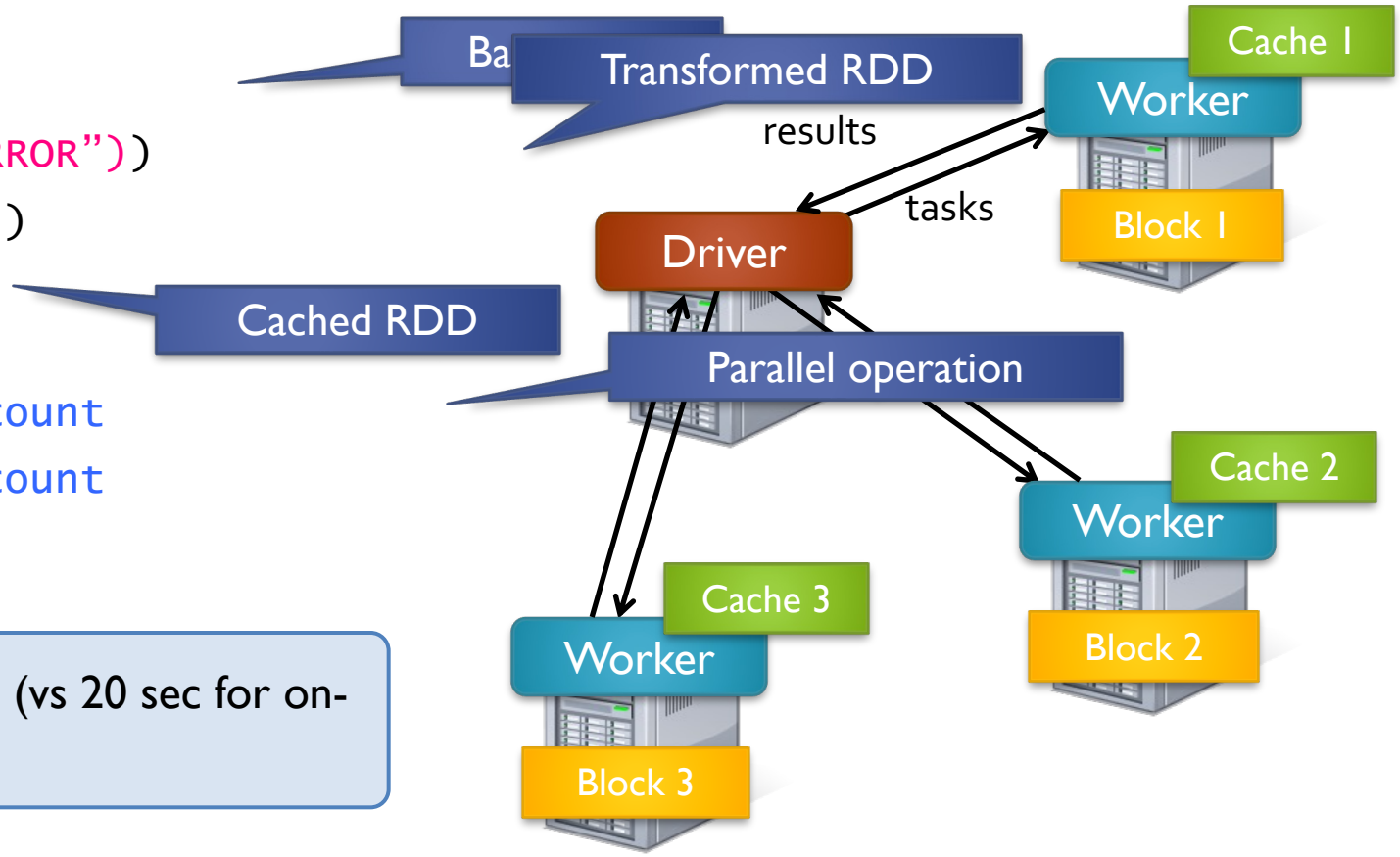
Programming Model

- Resilient distributed datasets (RDDs)
  - Immutable collections partitioned across cluster that can be rebuilt if a partition is lost
  - Created by transforming data in stable storage using data flow operators (map, filter, group-by, …)
  - Can be *cached* across parallel operations
- Parallel operations on RDDs
  - Reduce, collect, count, save, …
- Restricted shared variables
  - Accumulators, broadcast variables

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: full-text search of Wikipedia in <1 sec (vs 20 sec for on-disk data)

RDDs in More Detail

- An RDD is an immutable, partitioned, logical collection of records
  - Need not be materialized, but rather contains information to rebuild a dataset from stable storage
- Partitioning can be based on a key in each record (using hash or range partitioning)
- Built using bulk transformations on other RDDs
- Can be cached for future reuse

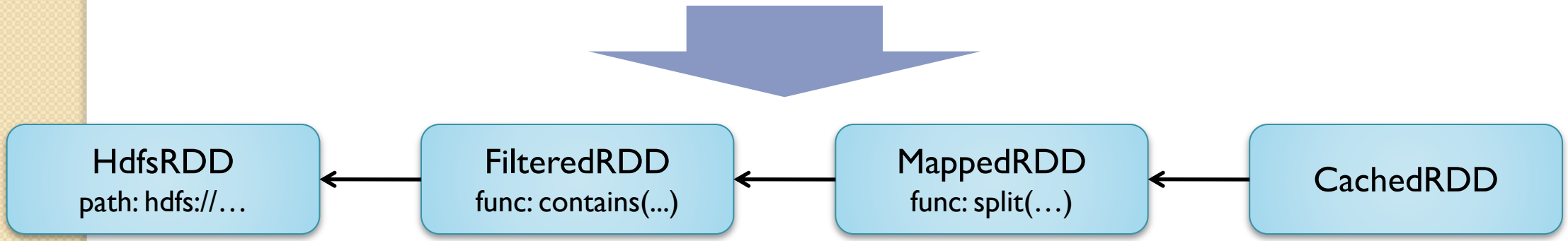
RDD Operations

Transformations (define a new RDD)	**Parallel operations** (return a result to driver)
map	reduce
filter	collect
sample	count
union	save
groupByKey	lookupKey
reduceByKey	…
join	
cache	
…	

RDD Fault Tolerance

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

```
cachedMsgs = textFile(...).filter(_.contains("error"))
                          .map(_.split('\t')(2))
                          .cache()
```



Benefits of RDD Model

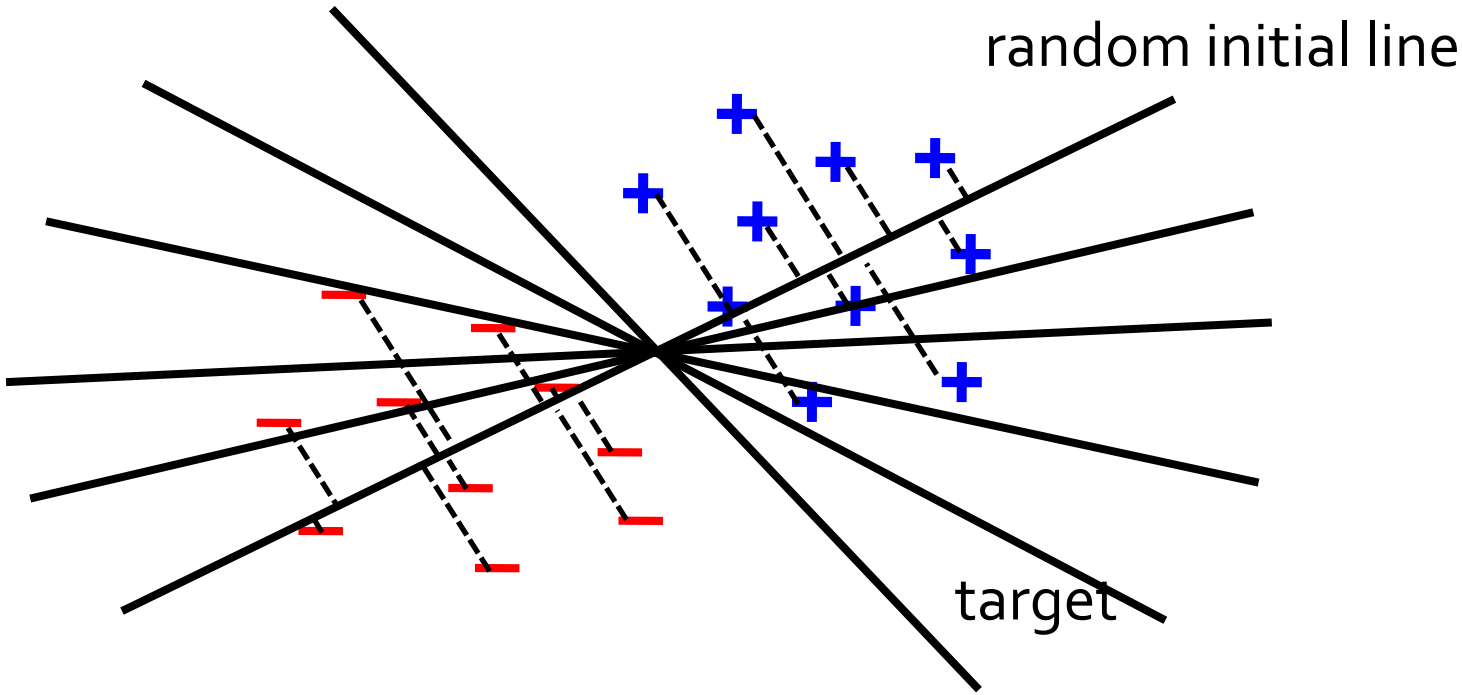
- Consistency is easy due to immutability
- Inexpensive fault tolerance (log lineage rather than replicating/checkpointing data)
- Locality-aware scheduling of tasks on partitions
- Despite being restricted, model seems applicable to a broad variety of applications

Outline

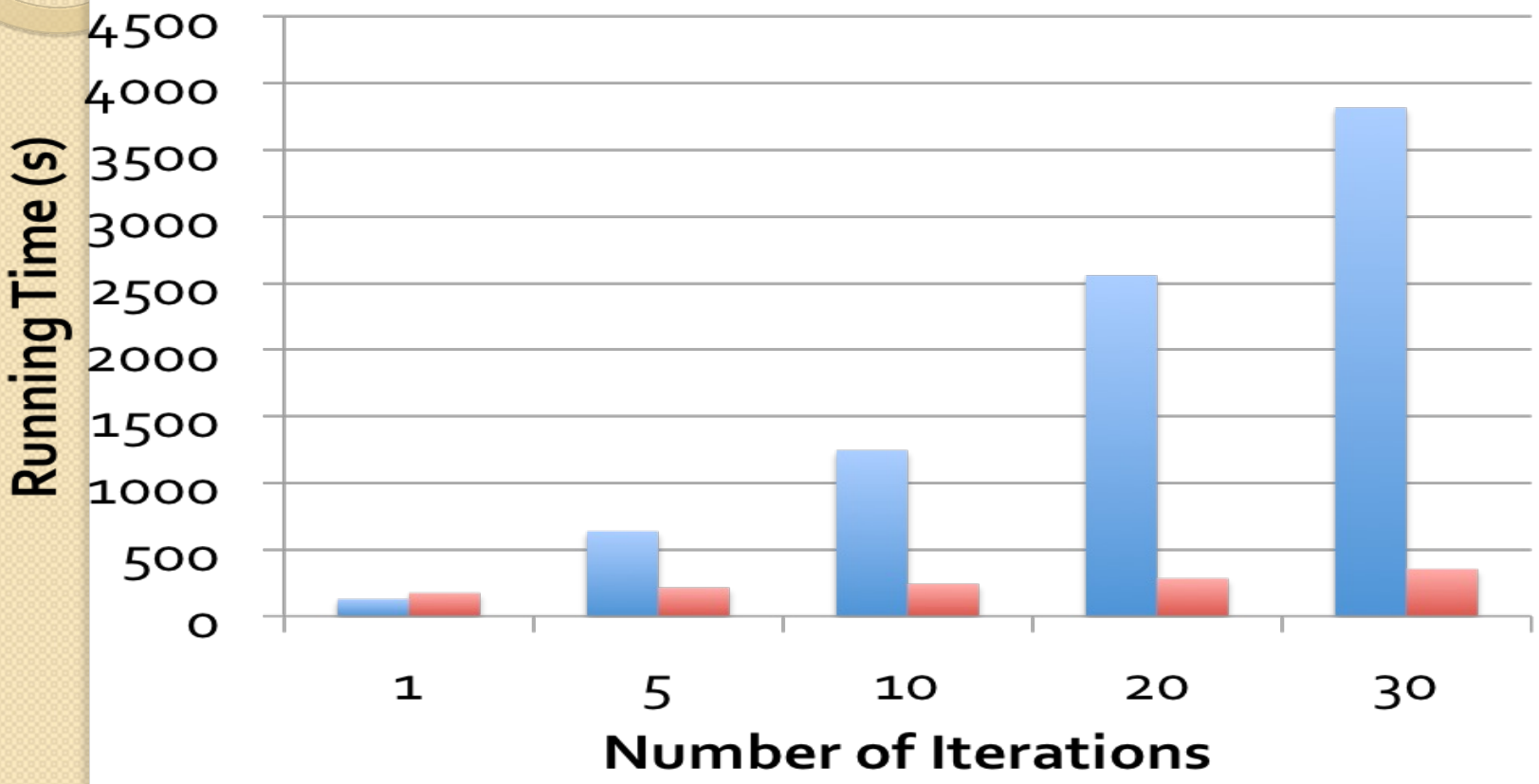
- Spark programming model
- Example applications
- Implementation
- Demo
- Future work

Example: Logistic Regression

- Goal: find best line separating two sets of points



Logistic Regression Performance



127 s / iteration

Hadoop

Spark

first iteration 174 s
further iterations 6 s

Example: MapReduce

- MapReduce data flow can be expressed using RDD transformations

```
res = data.flatMap(rec => myMapFunc(rec))
          .groupByKey()
          .map((key, vals) => myReduceFunc(key, vals))
```

Or with combiners:

```
res = data.flatMap(rec => myMapFunc(rec))
          .reduceByKey(myCombiner)
          .map((key, val) => myReduceFunc(key, val))
```

Word Count in Spark

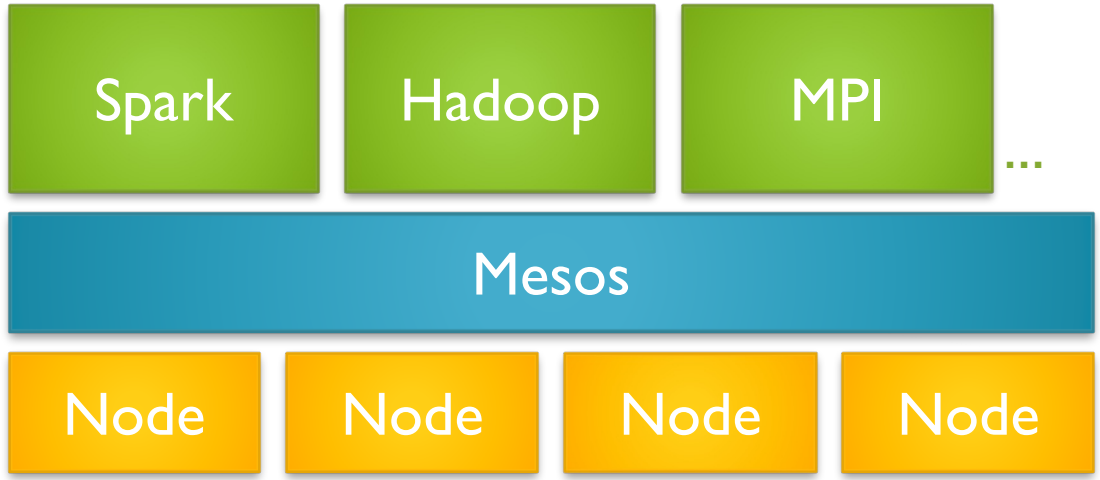
```
val lines = spark.textFile(“hdfs://...”)

val counts = lines.flatMap(_.split(“\\s”))
                  .reduceByKey(_ + _)

counts.save(“hdfs://...”)
```

Other Spark Applications

- Twitter spam classification
- Traffic prediction
- K-means clustering
- Alternating Least Squares matrix factorization
- In-memory OLAP aggregation on Hive data

Overview

- Spark runs on the Mesos cluster manager [NSDI 11], letting it share resources with Hadoop & other apps
- Can read from any Hadoop input source (e.g. HDFS)



- ~6000 lines of Scala code thanks to building on Mesos