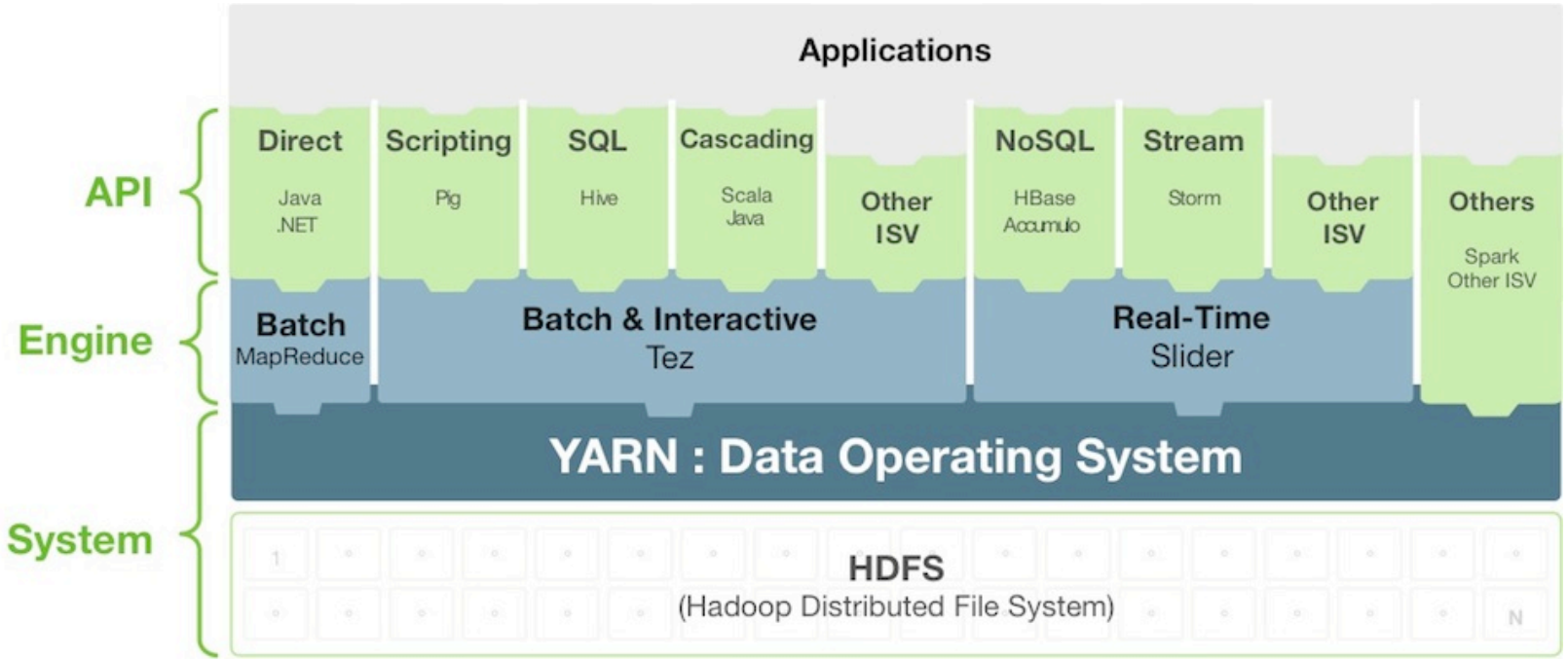
COSI 129a

Introduction to Big Data Analysis

Fall 2016

Hadoop Software beyond MR

Other Application on HDFS



Today

- Apache Pig: high-level language for data analysis
- Apache Hive: SQL on Hadoop
- Apache HBase: NoSQL database on Hadoop
- Cassandra: NoSQL database
- Spark: in memory data processing (more from Liuba)

Pig

Need for High-Level Languages

- Hadoop is great for large-data processing!
  - But writing Java programs for everything is slow
  - Not everyone wants to (or can) write Java code
- Solution: higher-level data processing languages
  - Pig: Pig Latin is a bit like Perl (scripting language)

What is PIG?

- High level scripting language used with Hadoop
- Platform for analyzing large data sets
  - High level language for expressing data analysis flows
- Pig-Latin: SQL-ish scripting language
- Pig scripts are translated to Map/Reduce jobs that are run on the Apache Hadoop cluster
- Used by Yahoo!, Twitter, Netflix, etc…

Why Pig?

- Map Reduce requires programmers to think in terms of map and reduce functions
  - Used by Java programmers
- Pig provides high-level language
  - Trivial to parallelize simple but "embarrassingly parallel" data analysis tasks
  - Complex tasks (multiple data transformations) can be encoded as (easy to write/understand/maintain) data flow sequences
  - Easily used by data scientists, analysts, statisticians

What can you do with Pig?

- Join Datasets
- Sort Datasets
- Filter
- Define Data Types
- Group By
- User-Defined Functions

Example Data Analysis Task

Find users who tend to visit "good" pages.

Visits

user	url	time
Amy	www.cnn.com	8:00
Amy	www.crap.com	8:05
Amy	www.myblog.com	10:00
Amy	www.flickr.com	10:05
Fred	cnn.com/index.htm	12:00

⋮

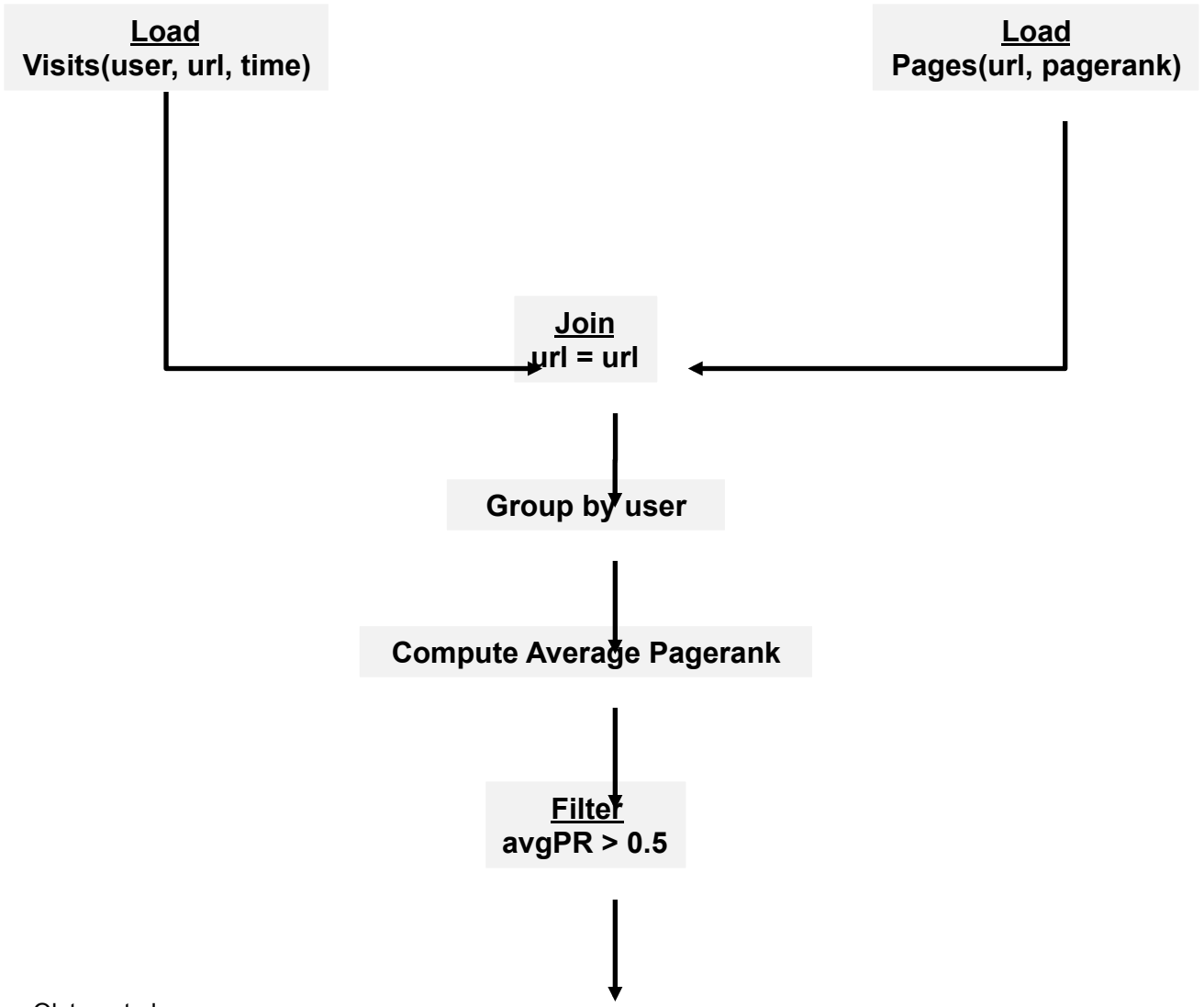
Pages

url	pagerank
www.cnn.com	0.9
www.flickr.com	0.9
www.myblog.com	0.7
www.crap.com	0.2

⋮

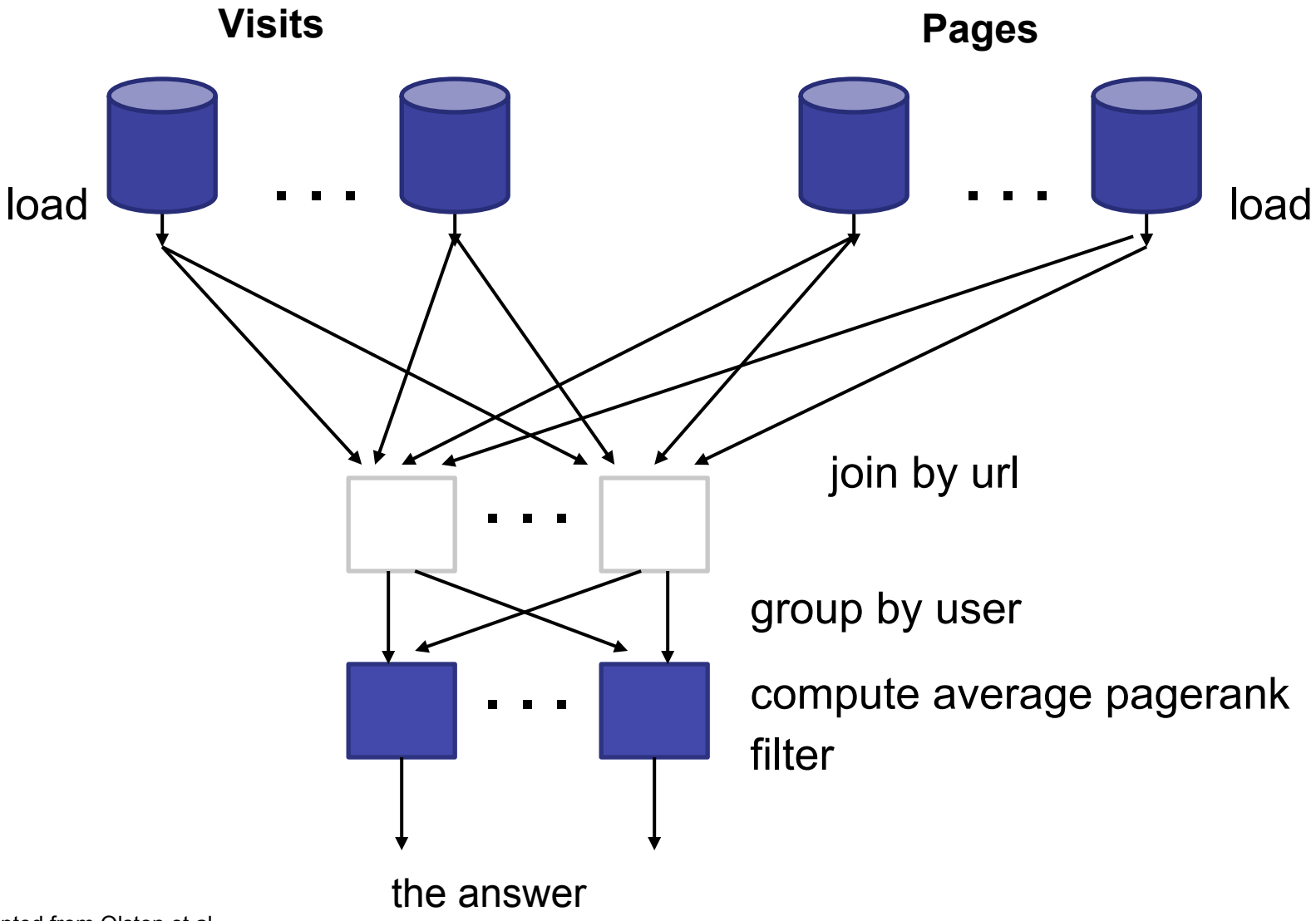
Pig Slides adapted from Olston et al.

Conceptual Dataflow



Pig Slides adapted from Olston et al.

System-Level Dataflow



Pig Slides adapted from Olston et al.

MapReduce Code

```
import java.io.IOException;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.Writable;
import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.KeyValueTextInputFormat;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.RecordReader;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.mapred.SequenceFileInputFormat;
import org.apache.hadoop.mapred.SequenceFileOutputFormat;
import org.apache.hadoop.mapred.TextInputFormat;
import org.apache.hadoop.mapred.jobcontrol.Job;
import org.apache.hadoop.mapred.jobcontrol.JobControl;
import org.apache.hadoop.mapred.lib.IdentityMapper;

public class MRExample {
    public static class LoadPages extends MapReduceBase
        implements Mapper<LongWritable, Text, Text, Text> {

        public void map(LongWritable k, Text val,
                OutputCollector<Text, Text> oc,
                Reporter reporter) throws IOException {
            // Pull the key out
            String line = val.toString();
            int firstComma = line.indexOf(',');
            String key = line.substring(0, firstComma);
            String value = line.substring(firstComma + 1);
            Text outKey = new Text(key);
            // Prepend an index to the value so we know which file
            // it came from.
            Text outVal = new Text("1" + value);
            oc.collect(outKey, outVal);
        }
    }
    public static class LoadAndFilterUsers extends MapReduceBase
        implements Mapper<LongWritable, Text, Text, Text> {

        public void map(LongWritable k, Text val,
                OutputCollector<Text, Text> oc,
                Reporter reporter) throws IOException {
            // Pull the key out
            String line = val.toString();
            int firstComma = line.indexOf(',');
            String value = line.substring(firstComma + 1);
            int age = Integer.parseInt(value);
            if (age < 18 || age > 25) return;
            String key = line.substring(0, firstComma);
            Text outKey = new Text(key);
            // Prepend an index to the value so we know which file
            // it came from.
            Text outVal = new Text("2" + value);
            oc.collect(outKey, outVal);
        }
    }
    public static class Join extends MapReduceBase
        implements Reducer<Text, Text, Text, Text> {

        public void reduce(Text key,
                Iterator<Text> iter,
                OutputCollector<Text, Text> oc,
                Reporter reporter) throws IOException {
            // For each value, figure out which file it's from and
store it
            // accordingly.
            List<String> first = new ArrayList<String>();
            List<String> second = new ArrayList<String>();

            while (iter.hasNext()) {
                Text t = iter.next();
                String value = t.toString();
                if (value.charAt(0) == '1')
first.add(value.substring(1));
                else second.add(value.substring(1));
                reporter.setStatus("OK");
            }

            // Do the cross product and collect the values
            for (String s1 : first) {
                for (String s2 : second) {
                    String outval = key + "," + s1 + "," + s2;
                    oc.collect(null, new Text(outval));
                    reporter.setStatus("OK");
                }
            }
        }
    }
    public static class LoadJoined extends MapReduceBase
        implements Mapper<Text, Text, Text, LongWritable> {

        public void map(
                Text k,
                Text val,
                OutputCollector<Text, LongWritable> oc,
                Reporter reporter) throws IOException {
            // Find the url
            String line = val.toString();
            int firstComma = line.indexOf(',');
            int secondComma = line.indexOf(',', firstComma);
            String key = line.substring(firstComma, secondComma);
            // drop the rest of the record, I don't need it anymore,
            // just pass a 1 for the combiner/reducer to sum instead.
            Text outKey = new Text(key);
            oc.collect(outKey, new LongWritable(1L));
        }
    }
    public static class ReduceUrls extends MapReduceBase
        implements Reducer<Text, LongWritable, WritableComparable,
Writable> {

        public void reduce(
                Text key,
                Iterator<LongWritable> iter,
                OutputCollector<WritableComparable, Writable> oc,
                Reporter reporter) throws IOException {
            // Add up all the values we see

            long sum = 0;
            while (iter.hasNext()) {
                sum += iter.next().get();
                reporter.setStatus("OK");
            }

            oc.collect(key, new LongWritable(sum));
        }
    }
    public static class LoadClicks extends MapReduceBase
        implements Mapper<WritableComparable, Writable, LongWritable,
Text> {

        public void map(
                WritableComparable key,
                Writable val,
                OutputCollector<LongWritable, Text> oc,
                Reporter reporter) throws IOException {
            oc.collect((LongWritable)val, (Text)key);
        }
    }
    public static class LimitClicks extends MapReduceBase
        implements Reducer<LongWritable, Text, LongWritable, Text> {

        int count = 0;
        public void reduce(
            LongWritable key,
            Iterator<Text> iter,
            OutputCollector<LongWritable, Text> oc,
            Reporter reporter) throws IOException {

            // Only output the first 100 records
            while (count < 100 && iter.hasNext()) {
                oc.collect(key, iter.next());
                count++;
            }
        }
    }
    public static void main(String[] args) throws IOException {
        JobConf lp = new JobConf(MRExample.class);
        lp.setJobName("Load Pages");
        lp.setInputFormat(TextInputFormat.class);
        lp.setOutputKeyClass(Text.class);
        lp.setOutputValueClass(Text.class);
        lp.setMapperClass(LoadPages.class);
        FileInputFormat.addInputPath(lp, new
Path("/user/gates/pages"));
        FileOutputFormat.setOutputPath(lp,
            new Path("/user/gates/tmp/indexed_pages"));
        lp.setNumReduceTasks(0);
        Job loadPages = new Job(lp);

        JobConf lfu = new JobConf(MRExample.class);
        lfu.setJobName("Load and Filter Users");
        lfu.setInputFormat(TextInputFormat.class);
        lfu.setOutputKeyClass(Text.class);
        lfu.setOutputValueClass(Text.class);
        lfu.setMapperClass(LoadAndFilterUsers.class);
        FileInputFormat.addInputPath(lfu, new
Path("/user/gates/users"));
        FileOutputFormat.setOutputPath(lfu,
            new Path("/user/gates/tmp/filtered_users"));
        lfu.setNumReduceTasks(0);
        Job loadUsers = new Job(lfu);

        JobConf join = new JobConf(MRExample.class);
        join.setJobName("Join Users and Pages");
        join.setInputFormat(KeyValueTextInputFormat.class);
        join.setOutputKeyClass(Text.class);
        join.setOutputValueClass(Text.class);
        join.setMapperClass(IdentityMapper.class);
        join.setReducerClass(Join.class);
        FileInputFormat.addInputPath(join, new
Path("/user/gates/tmp/indexed_pages"));
        FileInputFormat.addInputPath(join, new
Path("/user/gates/tmp/filtered_users"));
        FileOutputFormat.setOutputPath(join, new
Path("/user/gates/tmp/joined"));
        join.setNumReduceTasks(50);
        Job joinJob = new Job(join);
        joinJob.addDependingJob(loadPages);
        joinJob.addDependingJob(loadUsers);

        JobConf group = new JobConf(MRExample.class);
        group.setJobName("Group URLs");
        group.setInputFormat(KeyValueTextInputFormat.class);
        group.setOutputKeyClass(Text.class);
        group.setOutputValueClass(LongWritable.class);
        group.setOutputFormat(SequenceFileOutputFormat.class);
        group.setMapperClass(LoadJoined.class);
        group.setCombinerClass(ReduceUrls.class);
        group.setReducerClass(ReduceUrls.class);
        FileInputFormat.addInputPath(group, new
Path("/user/gates/tmp/joined"));
        FileOutputFormat.setOutputPath(group, new
Path("/user/gates/tmp/grouped"));
        group.setNumReduceTasks(50);
        Job groupJob = new Job(group);
        groupJob.addDependingJob(joinJob);

        JobConf top100 = new JobConf(MRExample.class);
        top100.setJobName("Top 100 sites");
        top100.setInputFormat(SequenceFileInputFormat.class);
        top100.setOutputKeyClass(LongWritable.class);
        top100.setOutputValueClass(Text.class);
        top100.setOutputFormat(SequenceFileOutputFormat.class);
        top100.setMapperClass(LoadClicks.class);
        top100.setCombinerClass(LimitClicks.class);
        top100.setReducerClass(LimitClicks.class);
        FileInputFormat.addInputPath(top100, new
Path("/user/gates/tmp/grouped"));
        FileOutputFormat.setOutputPath(top100, new
Path("/user/gates/top100sitesforusers18to25"));
        top100.setNumReduceTasks(1);
        Job limit = new Job(top100);
        limit.addDependingJob(groupJob);

        JobControl jc = new JobControl("Find top 100 sites for users
18 to 25");
        jc.addJob(loadPages);
        jc.addJob(loadUsers);
        jc.addJob(joinJob);
        jc.addJob(groupJob);
        jc.addJob(limit);
        jc.run();
    }
}
```

Pig Slides adapted from Olston et al.

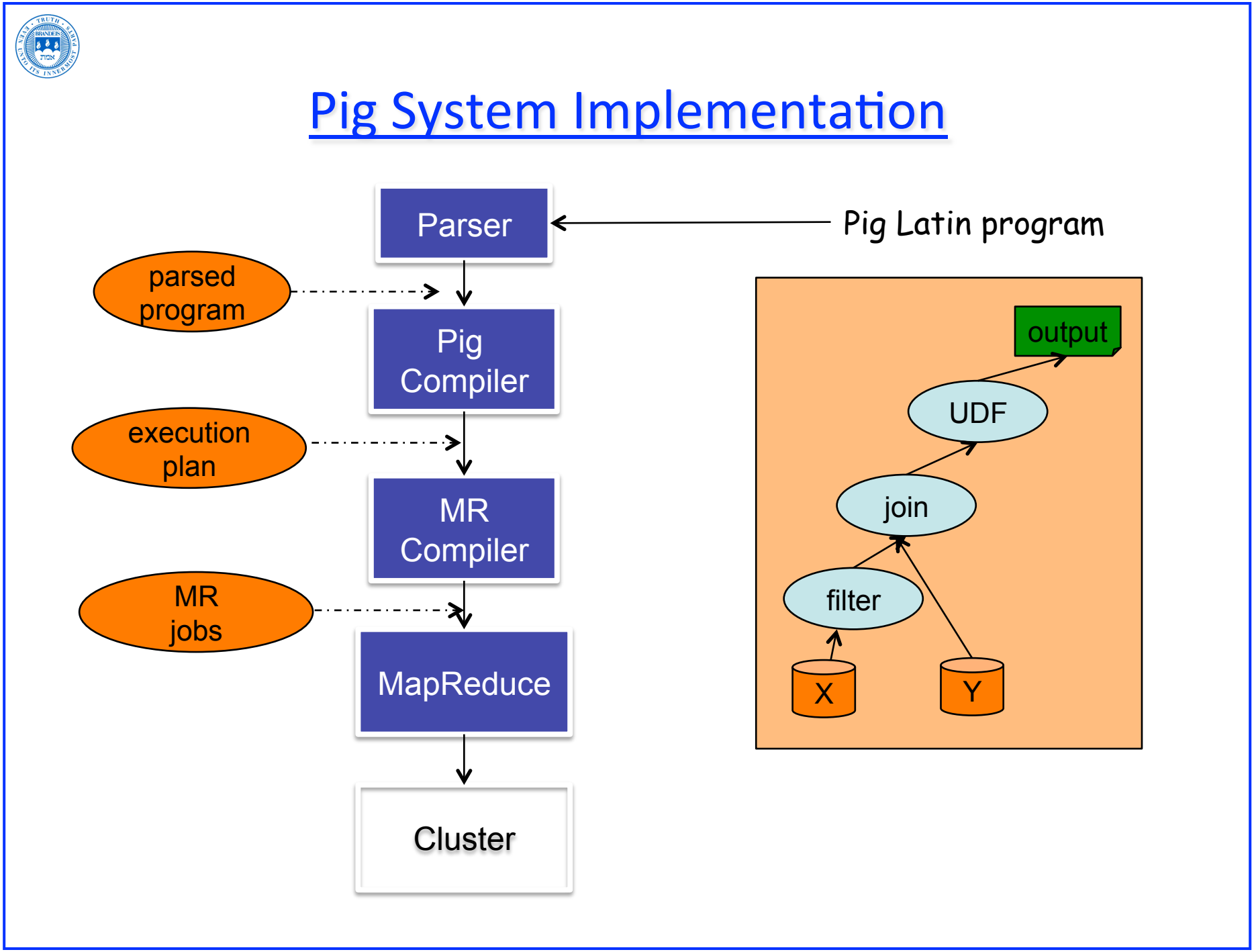
Pig Latin Script

```
Visits = load    '/data/visits' as (user, url, time);
Pages = load    '/data/pages' as (url, pagerank);

VP = join    Visits by url, Pages by url;
UserVisits = group   VP by user;
UserPageranks = foreach UserVisits generate user,
AVG(VP.pagerank) as avgpr;
GoodUsers = filter  UserPageranks by avgpr > '0.5';

store   GoodUsers into '/data/good_users';
```

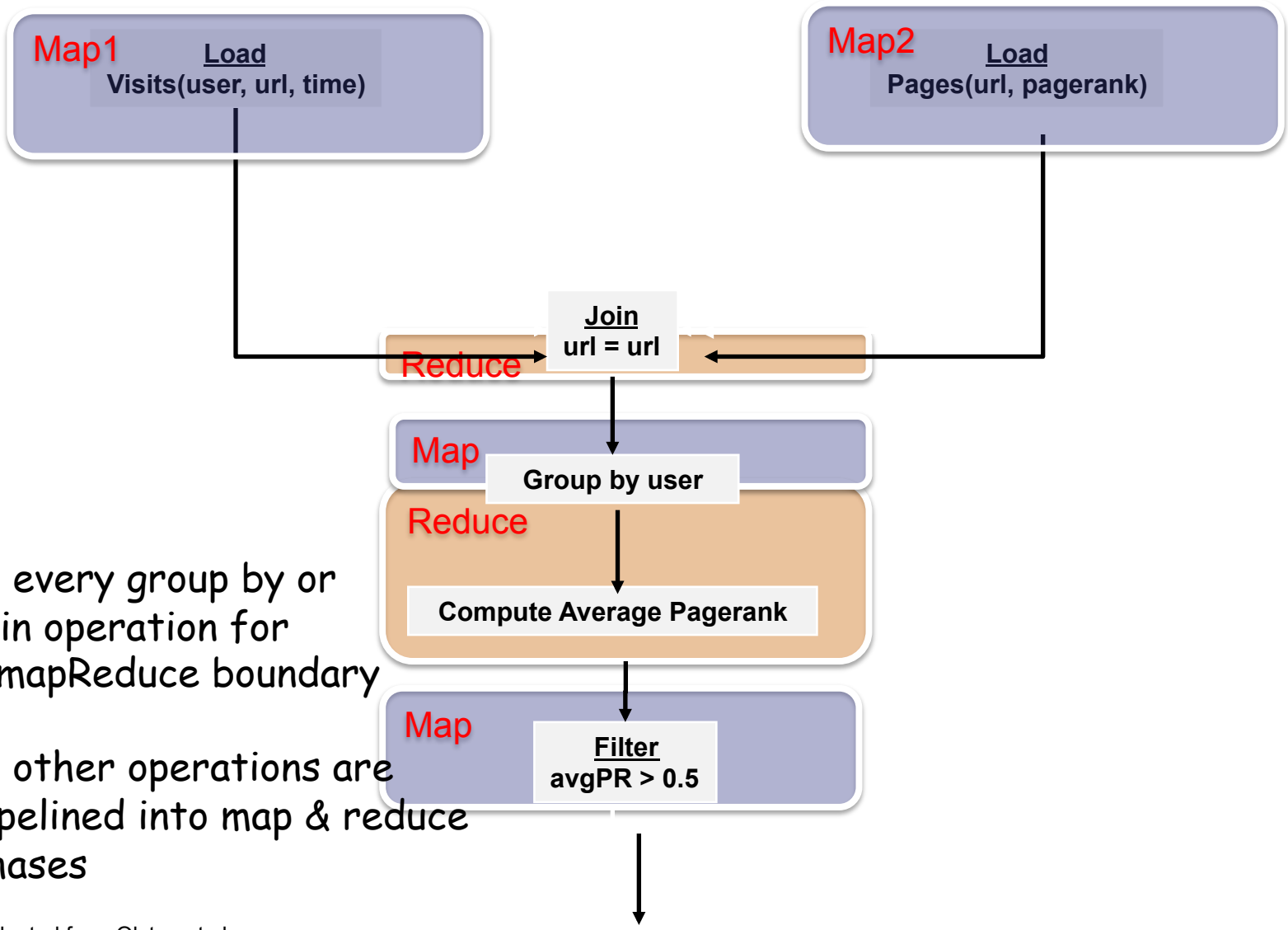
Pig Slides adapted from Olston et al.

Pig Latin Nodes

- No need to import data into database
  - Pig Latin works directly on files
- Schemas are optional and can be assigned dynamically
  - Load 'data/visits' as (user, url, time);
- Can call User Defined Functions (UDFs) in every construct like Load, Store, Group, Filter, Foreach
  - `foreach UserVisits generate user, AVG(VP.pagerank)`


Pig System Implementation
Parser
Pig Latin program
parsed program
Pig Compiler
execution plan
MR Compiler
MR jobs
MapReduce
Cluster
output
UDF
join
filter
X
Y

Conceptual Dataflow



- every group by or join operation for a mapReduce boundary
- other operations are pipelined into map & reduce phases

Pig Slides adapted from Olston et al.

Hive

Motivation

- Hadoop supports data-intensive distributed applications.

However…

  - Map-reduce hard to program (users know sql/bash/ python).
  - No schema.

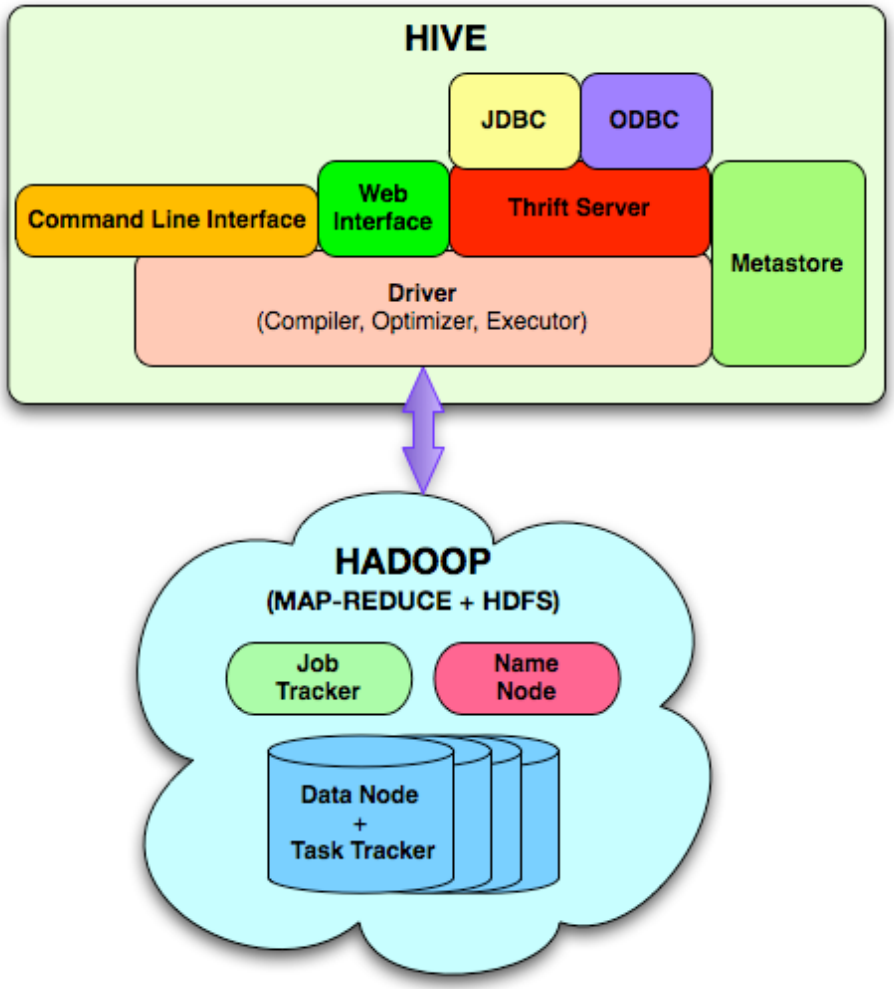
Hive: SQL on Hadoop

- Data warehouse infrastructure on top of HDFS
- Designed for data summarization, query and analysis
- Projects structure on unstructured data and queries the data
- Developed initially by Facebook
- Used now by Netflix, Amazon, etc

Hive Features

- HiveSQL: SQL-like language for data querying
  - Queries are translated to Map/Reduce jobs
- Maintains support for Map/Reduce programs
- Provides indexes
- Stores metadata on a relational DB
  - Derby DB but this can be customized
- Operates on compressed data
- Supports UDFs for processing dates, strings

Hive architecture (from the paper)



Data model

- Hive structures data into well-understood database concepts such as: tables, rows, cols, partitions
- It supports primitive types: integers, floats, doubles, and strings
- Hive also supports:
  - associative arrays: map<key-type, value-type>
  - Lists: list<element type>

Examples – Data Definition Operations

- **CREATE TABLE** sample (foo INT, bar STRING) **PARTITIONED BY** (ds STRING);
- **SHOW TABLES** '.*s';
- **DESCRIBE** sample;
- **ALTER TABLE** sample **ADD COLUMNS** (new_col INT);
- **DROP TABLE** sample;

Examples – Data Manipulation Operations

- **LOAD DATA LOCAL INPATH** './sample.txt' **OVERWRITE INTO TABLE** sample **PARTITION** (ds='2012-02-24');

- **LOAD DATA INPATH** '/user/falvariz/hive/sample.txt' **OVERWRITE INTO TABLE** sample **PARTITION** (ds='2012-02-24');

SELECTS and FILTERS

- **SELECT** foo **FROM** sample **WHERE** ds='2012-02-24';
- **INSERT OVERWRITE DIRECTORY** '/tmp/hdfs_out' **SELECT** * **FROM** sample **WHERE** ds='2012-02-24';
- **INSERT OVERWRITE LOCAL DIRECTORY** '/tmp/hive-sample-out' **SELECT** * **FROM** sample;

Aggregations and Groups

- **SELECT MAX**(foo) **FROM** sample;
- **SELECT** ds, **COUNT**(*), **SUM**(foo) **FROM** sample **GROUP BY** ds;
- **FROM** sample s **INSERT OVERWRITE TABLE** bar **SELECT** s.bar, count(*) WHERE s.foo > 0 **GROUP BY** s.bar;

Pig Example

Load files “users” and “clicks”

```
Users             = load 'users' as (name, age, ipaddr);
Clicks            = load 'clicks' as (user, url, value);
ValuableClicks    = filter Clicks by value > 0;
UserClicks        = join Users by name, ValuableClicks by user;
Geoinfo           = load 'geoinfo' as (ipaddr, dma);
UserGeo           = join UserClicks by ipaddr, Geoinfo by ipaddr;
ByDMA             = group UserGeo by dma;
ValuableClicksPerDMA = foreach ByDMA generate group, COUNT(UserGeo);
store ValuableClicksPerDMA into 'ValuableClicksPerDMA';
```

Pig Example

```
Users           = load 'users' as (name, age, ipaddr);
Clicks          = load 'clicks' as (user, url, value);
ValuableClicks  = filter Clicks by value > 0;
UserClicks      = join Users by name, ValuableClicks by user;
Geoinfo         = load 'geoinfo' as (ipaddr, dma);
UserGeo         = join UserClicks by ipaddr, Geoinfo by ipaddr;
ByDMA           = group UserGeo by dma;
ValuableClicksPerDMA = foreach ByDMA generate group, COUNT(UserGeo);
store ValuableClicksPerDMA into 'ValuableClicksPerDMA';
```

Filter clicks with no value

Pig Example

```
Users          = load 'users' as (name, age, ipaddr);
Clicks         = load 'clicks' as (user, url, value);
ValuableClicks = filter Clicks by value > 0;
UserClicks     = join Users by name, ValuableClicks by user;
Geoinfo        = load 'geoinfo' as (ipaddr, dma);
UserGeo        = join UserClicks by ipaddr, Geoinfo by ipaddr;
ByDMA          = group UserGeo by dma;
ValuableClicksPerDMA = foreach ByDMA generate group, COUNT(UserGeo);
store ValuableClicksPerDMA into 'ValuableClicksPerDMA';
```

Join users with their clicks

Pig Example

```
Users           = load 'users' as (name, age, ipaddr);
Clicks          = load 'clicks' as (user, url, value);
ValuableClicks  = filter Clicks by value > 0;
UserClicks      = join Users by name, ValuableClicks by us
Geoinfo         = load 'geoinfo' as (ipaddr, dma);
UserGeo         = join UserClicks by ipaddr, Geoinfo by ipaddr;
ByDMA           = group UserGeo by dma;
ValuableClicksPerDMA = foreach ByDMA generate group, COUNT(UserGeo);
store ValuableClicksPerDMA into 'ValuableClicksPerDMA';
```

Load “geoinfo” file (DMA: designated marketing area)

Pig Example

```
Users            = load 'users' as (name, age, ipaddr);
Clicks           = load 'clicks' as (user, url, value);
ValuableClicks   = filter Clicks by value > 0;
UserClicks       = join Users by name, ValuableClicks by user;
Geoinfo          = load 'geoinfo' as (ipaddr, dma);
UserGeo          = join UserClicks by ipaddr, Geoinfo by ipaddr;
ByDMA            = group UserGeo by dma;
ValuableClicksPerDMA = foreach ByDMA generate group, COUNT(UserGeo);
store ValuableClicksPerDMA into 'ValuableClicksPerDMA';
```

Join users/clicks with their geo data based on their ip

Pig Example

```
Users           = load 'users' as (name, age, ipaddr);
Clicks          = load 'clicks' as (user, url, value);
ValuableClicks  = filter Clicks by value > 0;
UserClicks      = join Users by name, ValuableClicks by user;
Geoinfo         = load 'geoinfo' as (ipaddr, dma);
UserGeo         = join UserClicks by ipaddr, Geoinfo by ipad
ByDMA           = group UserGeo by dma;
ValuableClicksPerDMA = foreach ByDMA generate group, COUNT(UserGeo);
store ValuableClicksPerDMA into 'ValuableClicksPerDMA';
```

Group data based on marketing area

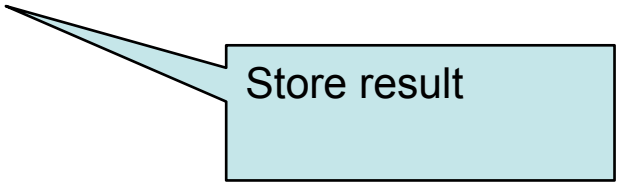
Pig Example

```
Users            = load 'users' as (name, age, ipaddr);
Clicks           = load 'clicks' as (user, url, value);
ValuableClicks   = filter Clicks by value > 0;
UserClicks       = join Users by name, ValuableClicks by user;
Geoinfo          = load 'geoinfo' as (ipaddr, dma);
UserGeo          = join UserClicks by ipaddr, Geoinfo by ipaddr;
ByDMA            = group UserGeo by dma;
ValuableClicksPerDMA = foreach ByDMA generate group, COUNT(UserGeo);
store ValuableClicksPerDMA into 'ValuableClicksPerDMA';
```

Count # clicks per marketing area

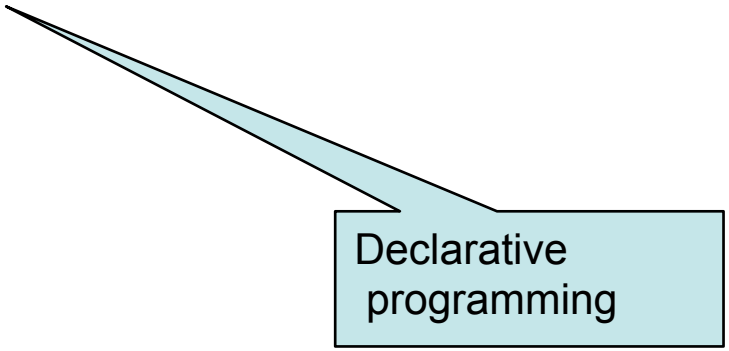
Pig Example

```
Users            = load 'users' as (name, age, ipaddr);
Clicks           = load 'clicks' as (user, url, value);
ValuableClicks   = filter Clicks by value > 0;
UserClicks       = join Users by name, ValuableClicks by user;
Geoinfo          = load 'geoinfo' as (ipaddr, dma);
UserGeo          = join UserClicks by ipaddr, Geoinfo by ipaddr;
ByDMA            = group UserGeo by dma;
ValuableClicksPerDMA = foreach ByDMA generate group, COUNT(UserGeo);
store ValuableClicksPerDMA into 'ValuableClicksPerDMA';
```



Hive Example (same as Pig example)

```
insert into ValuableClicksPerDMA
select dma, count(*)
from geoinfo join (
select name, ipaddr
from users join clicks on (users.name = clicks.user)
where value > 0;
) using ipaddr
group by dma;
```



HBase

HBase: Overview

- HBase is an **open-source**, **distributed**, **column-oriented** database built on top of HDFS
  - Scales horizontally to 1,000s of commodity servers and petabytes of indexed storage.
- Designed to operate on top of the Hadoop distributed file system (HDFS)
  - Offers scalability, fault tolerance, and high availability
- Designed for storing large quantities of sparse data
  - Small amounts of information caught within a large collection of empty or unimportant data
    - E..g, finding the 50 largest items in a group of 2 billion records

What is HBase?

- Non-relational, distributed database
- Column-oriented
- Multi-dimensional
- High availability
- High performance

Background: Column Families

- Image a relational database that holds personal information

PERSON TABLE

PersonID	Name	BirthDate	Gender	Address	...
1	H. Houdini	1926-10-31	M	Budapest, Hungary	
2	D. Copper	1956-09-16	M	New Jersey, USA	
3	Merlin	1136-12-03	F	Stonehenge, England	
...	...	...	...	...	
500,000,000	F. Cadillac	1964-01-07	M	Nevada, USA	

Figure 1 - Census Data

- Query: "How many men were born each year?"
  - We have to read all the columns of each row
- Column-store DBs solve this problem
  - Store each column separately so that operations for one column of the entire tables are significantly quicker that the traditional row-store DBs
  - Problem: Getting all the data for a single person is very expensive
    - We must fetch data from numerous places on the disk

Column Families

- Columns of related data are grouped together within one table

PERSON TABLE

row key	personal_data		demographic		...
PersonID	**Name**	**Address**	**BirthDate**	**Gender**	...
1	H. Houdini	Budapest, Hungary	1926-10-31	M	
2	D. Copper	New Jersey, USA	1956-09-16	M	
3	Merlin	Stonehenge, England	1136-12-03	F	
...	...	...	...	...	
500,000,000	F. Cadillac	Nevada, USA	1964-01-07	M	

Figure 2 - Census Data in Column Families

- Hive supports column families
  - Only fetch the column families of the columns required by a query
  - Stores columns in a family together on the disk

Tables and Column Families

Tables are broken into groupings called Column Families.

Column Family "contactinfo"	Column Family "profilephoto"

Group data frequently accessed together and compress it

Group photos with different settings

Rows and Columns

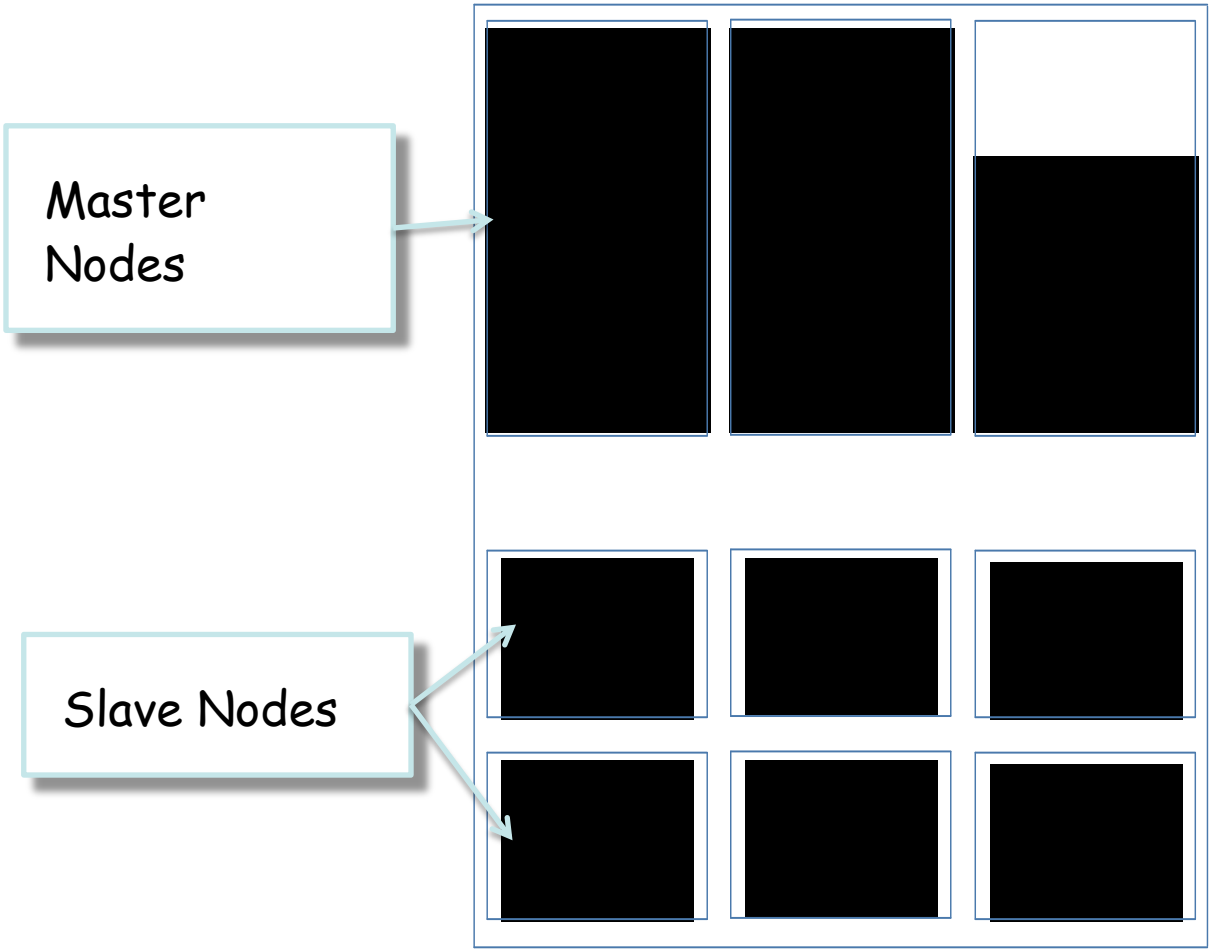
No storage penalty for unused columns

Row key	Column Family "contactinfo"		Column Family "profilephoto"
adupont	fname: Andre	lname: Dupont	
jsmith	fname: John	lname: Smith	image: <smith.jpg>

Row keys identify a row

Each Column Family can have many columns

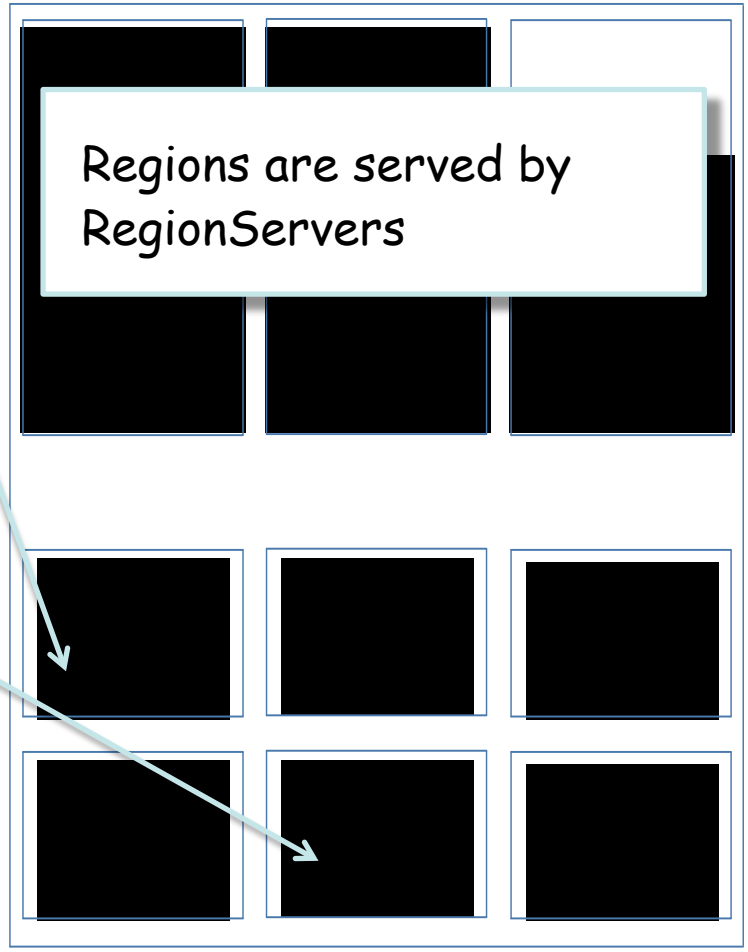
Daemon Locations

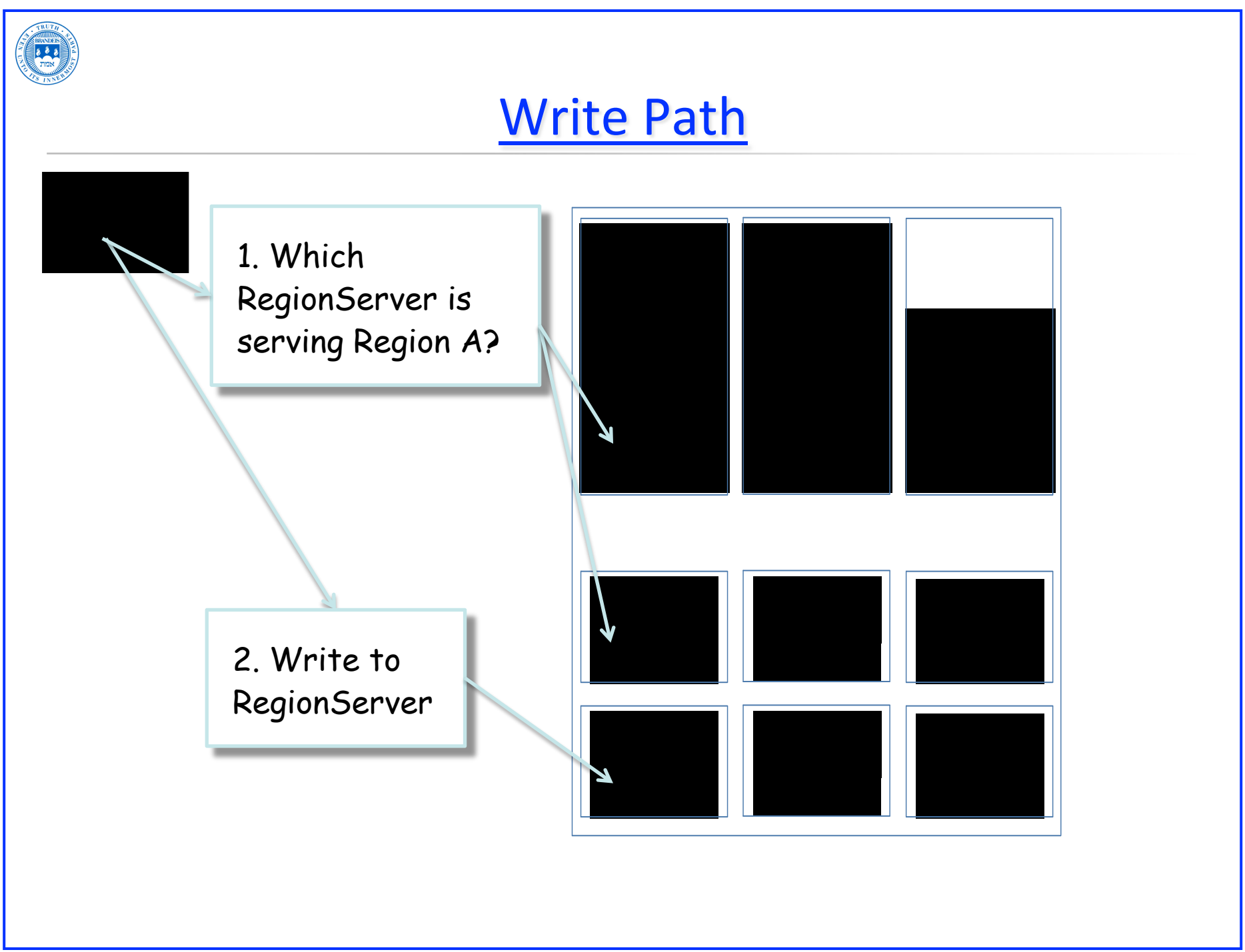


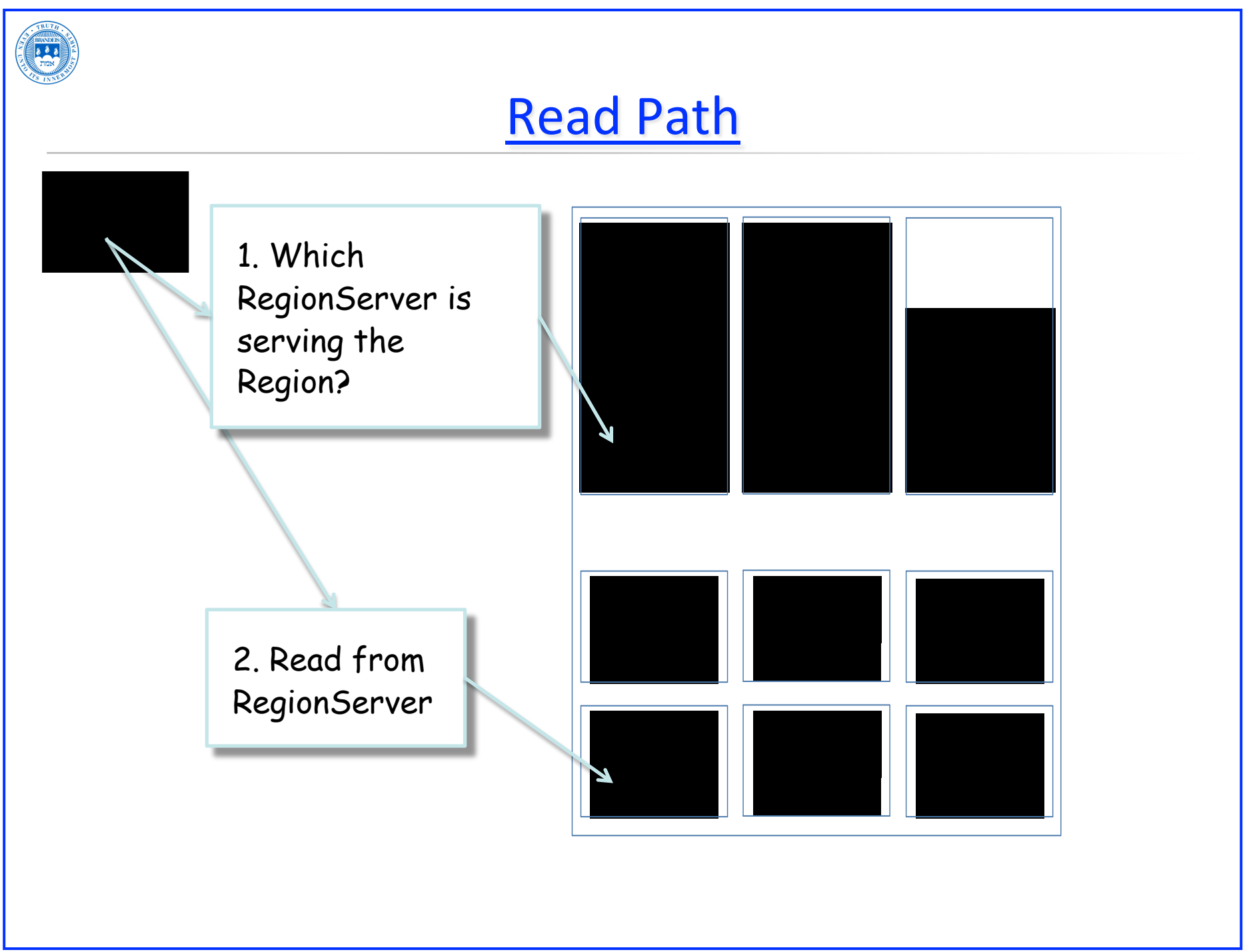
Regions

Row key	Column Family "contactinfo"	
adupont	fname: Andre	lname: Dupont

Row key	Column Family "contactinfo"	
mrossi	fname: Mario	lname: Rossi
zstevens	fname: Zack	lname: Stevens




Write Path
1. Which RegionServer is serving Region A?
2. Write to RegionServer


Read Path
1. Which RegionServer is serving the Region?
2. Read from RegionServer

No SQL Means No SQL

- Data is not accessed over SQL
- You must:
  - Create your own connections
  - Keep track of the type of data in a column
  - Give each row a key
  - Access a row by its key

Types of Access

- Gets
  - Gets a row's data based on the row key
- Puts
  - Upserts a row with data based on the row key
- Scans
  - Finds all matching rows based on the row key
  - Scan logic can be increased by using filters

Gets

```
1 Get g = new Get(ROW_KEY_BYTES);
2 Result r= table.get(g);
3 byte[] byteArray =
  r.getValue(COLFAM_BYTS,COLDESC_BYTS);
4 String columnValue =
  Bytes.toString(byteArray);
```

Puts

```
1 Put p = new
  Put(Bytes.toBytes(ROW_KEY_BYTES);
2 p.add(COLFAM_BYTES, COLDESC_BYTES,
  Bytes.toBytes("value"));
3
4 table.put(p);
```

No SQL Means No SQL

- Designing schemas for HBase requires an in-depth knowledge
- Schema Design is ‘data-centric’ not ‘relationship-centric’
- You design around how data is accessed
- Row keys are engineered

Row Keys

- A row key is more than the glue between two tables
- Engineering time is spent just on constructing a row key
  - Contents of a row key vary by access pattern
  - Often made up of several pieces of data

Schema Design

- Schema design does not start in an ERD
- Access pattern must be known and ascertained
- Denormalize to improve performance
  - Fewer, bigger tables

When to Consider Not Using HBase?

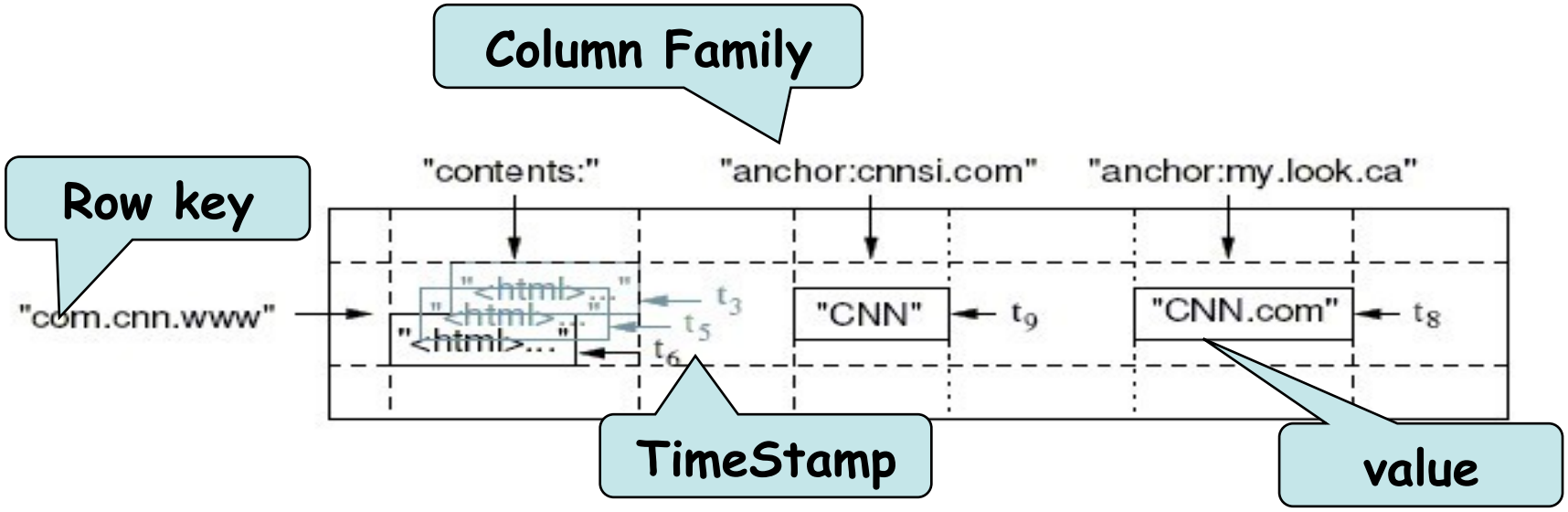
- Only use with Big Data problems
- Read straight through files
- Write all at once or append new files
  - Not random reads or writes
- Access patterns of the data are ill-defined

HBase Is Not …

- Limited atomicity and transaction support.
  - HBase supports multiple batched mutations of single rows only.
  - Data is unstructured and untyped.
- No accessed or manipulated via SQL.
  - Programmatic access via Java, REST, or Thrift APIs.
  - Scripting via JRuby.

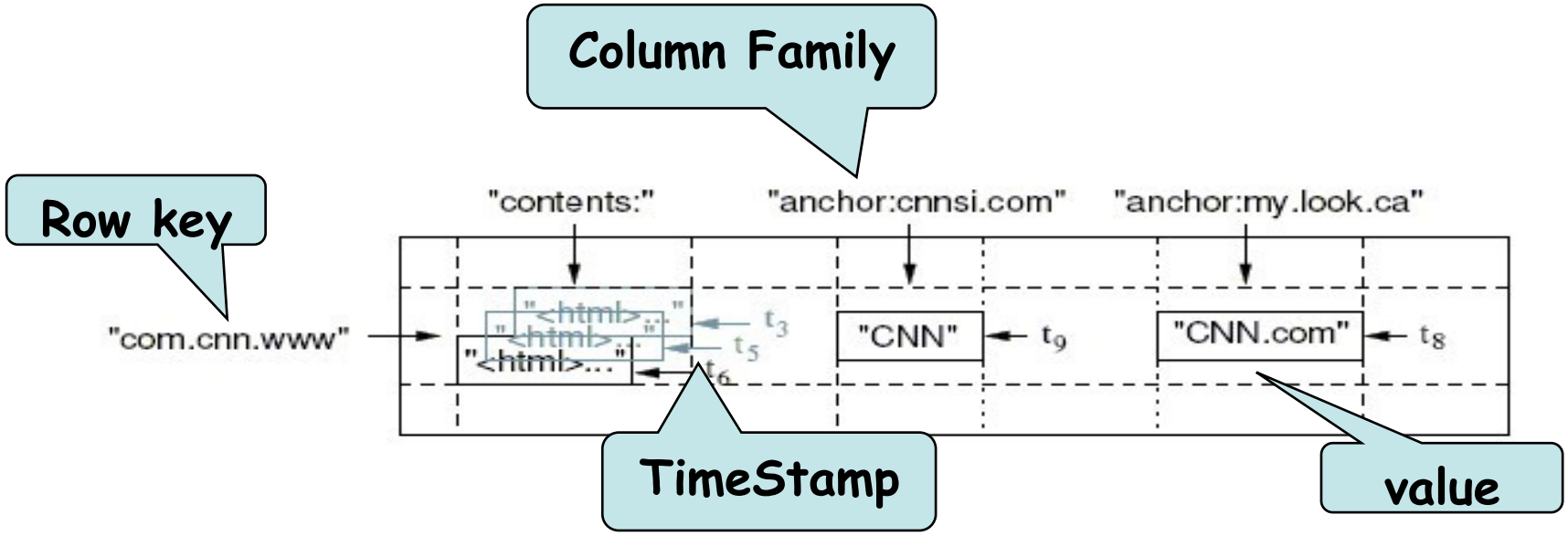
HBase: Data Model

- A sparse, multi-dimensional sort map
  - {row, column, timestamp}->value
- Column = Column Family



Data Model in Hive

- Tables are sorted by Row
- Table schema only defines it's *column families.*
  - Each family consists of any number of columns
  - Each column consists of any number of versions
  - Columns within a family are sorted and stored together



Example: Facebook Inbox Search

- Get top N message IDs for a specific user & word
- Store data as <userID, word, message ID> -> offset of word in message
  - UserID as key
  - words as column
  - Message ID as the timestamp
  - Offset is the value

User1:hi:17->offset1

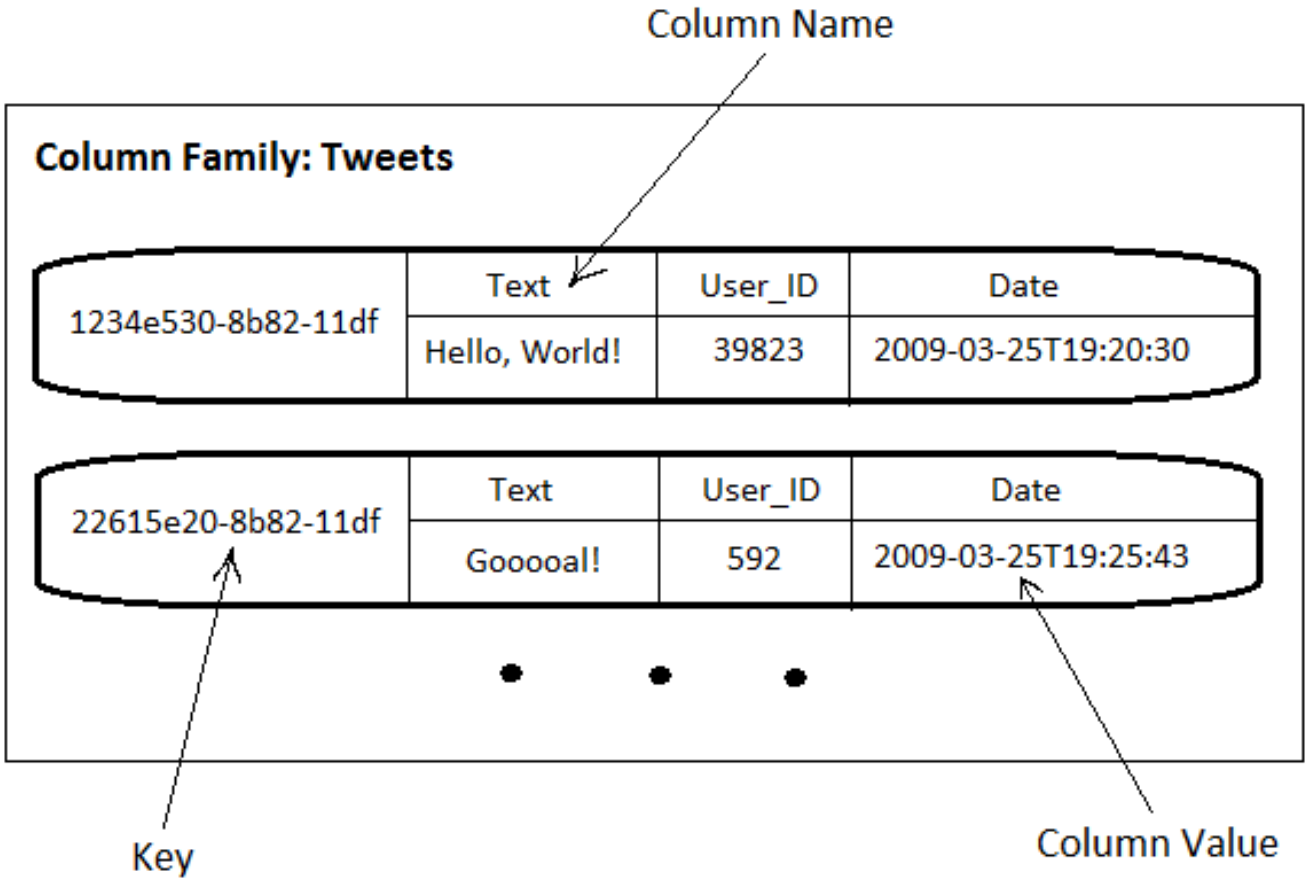
User1:hi:18->offset2

User1:hello:16->offset3

User1:hello:21->offset4

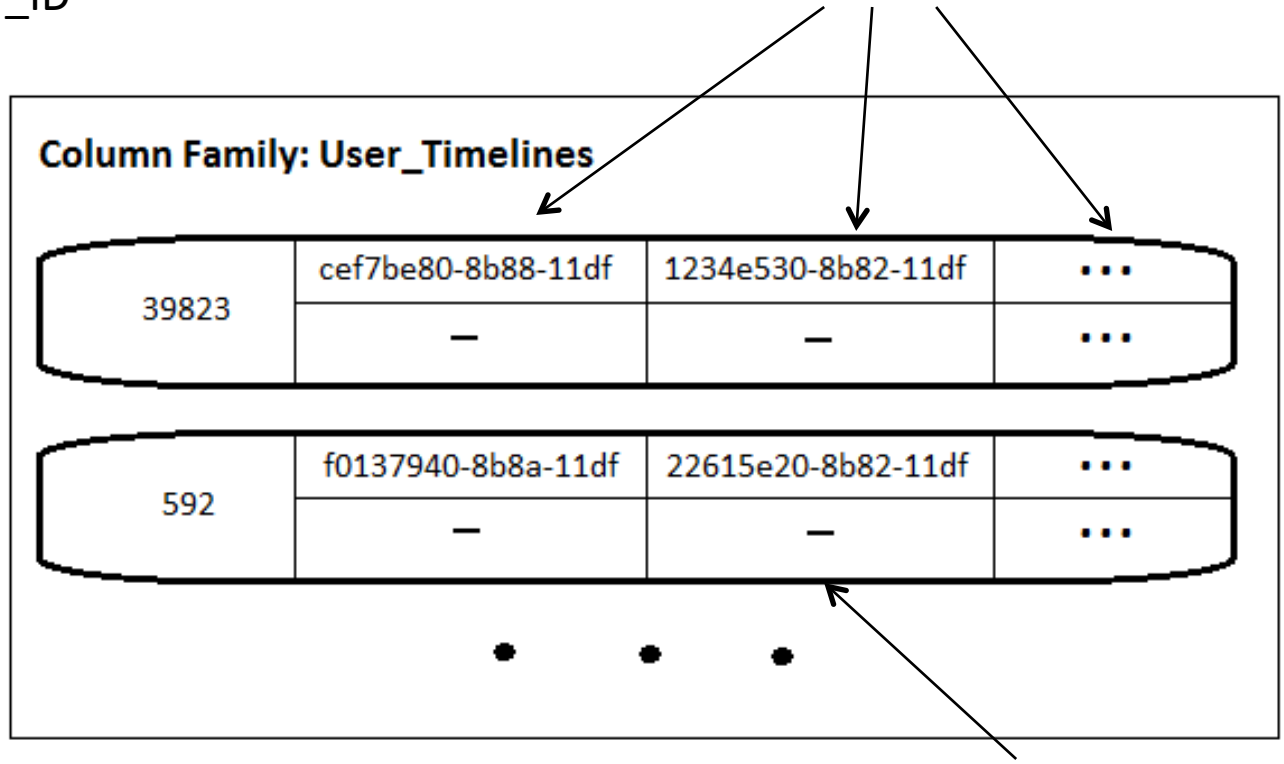
User2:……..

Example: Twitter



Cassandra Data Model
Example: Twitter Database

Column Names: Tweet_IDs, of type Time UUID, ordered by time of posting
Key: User_ID



Column Values
(empty in this example)

When to use HBase?

- Storing large amounts of data
- Have variable schema where each row is slightly different
  - Many columns with most values being null
- Have structured and unstructured data
- Don't need full RDBMs capabilities (transactions, joins, etc)

Cassandra

Definition of Cassandra

Apache Cassandra™ is a free

Distributed…
High performance…
Extremely scalable…
Fault tolerant (i.e. no single point of failure)…

post-relational database solution. Cassandra can serve as both real-time datastore (the “system of record”) for online/ transactional applications, and as a read-intensive database for business intelligence systems.

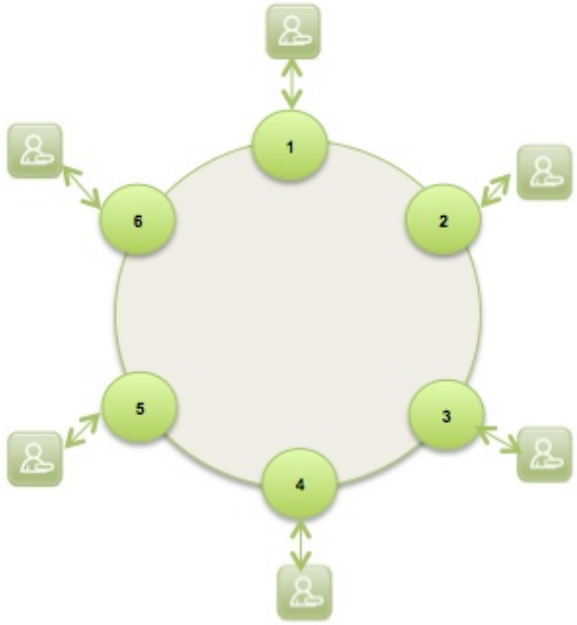
The History of Cassandra

Bigtable

Dynamo

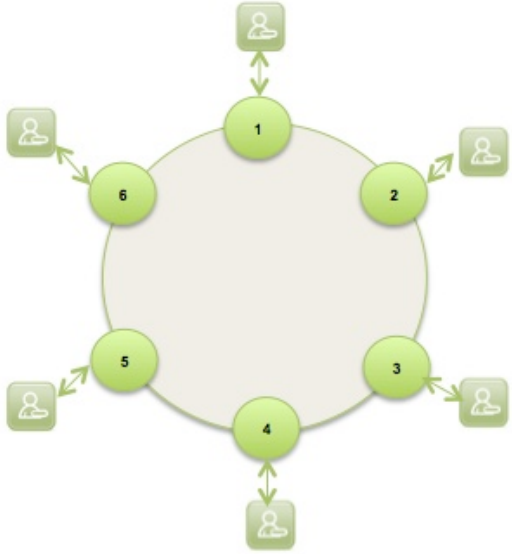
Architecture Overview

- Cassandra was designed with the understanding that system/ hardware failures can and do occur
- Peer-to-peer, distributed system
  - All nodes the same
- Data partitioned among all nodes in the cluster
- Custom data replication to ensure fault tolerance
- Read/Write-anywhere design



Architecture Overview

- Each node communicates with each other through the Gossip protocol, which exchanges information across the cluster every second
- A commit log is used on each node to capture write activity. Data durability is assured
- Data also written to an in-memory structure (memtable) and then to disk once the memory structure is full (an SStable)

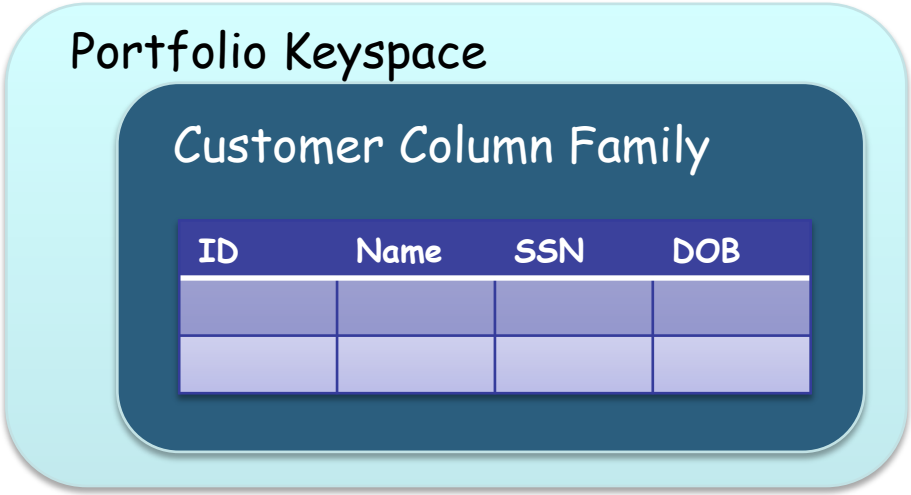


Gossip-based Communication

- Every node has to know important information about every other node (e.g., membership, failures).
- Such information is communicated among the nodes using a **gossip-based protocol**.
  - Every second, each node randomly chooses a peer node and starts a state exchange round with it.
  - State is disseminated in O(log N) rounds, where N is the number of nodes in the cluster.

Architecture Overview

- The schema used in Cassandra is a row-oriented, column structure
- A keyspace is akin to a database in the RDBMS world
- A column family is similar to an RDBMS table but is more flexible/ dynamic
- A row in a column family is indexed by its key. Other columns may be indexed as well

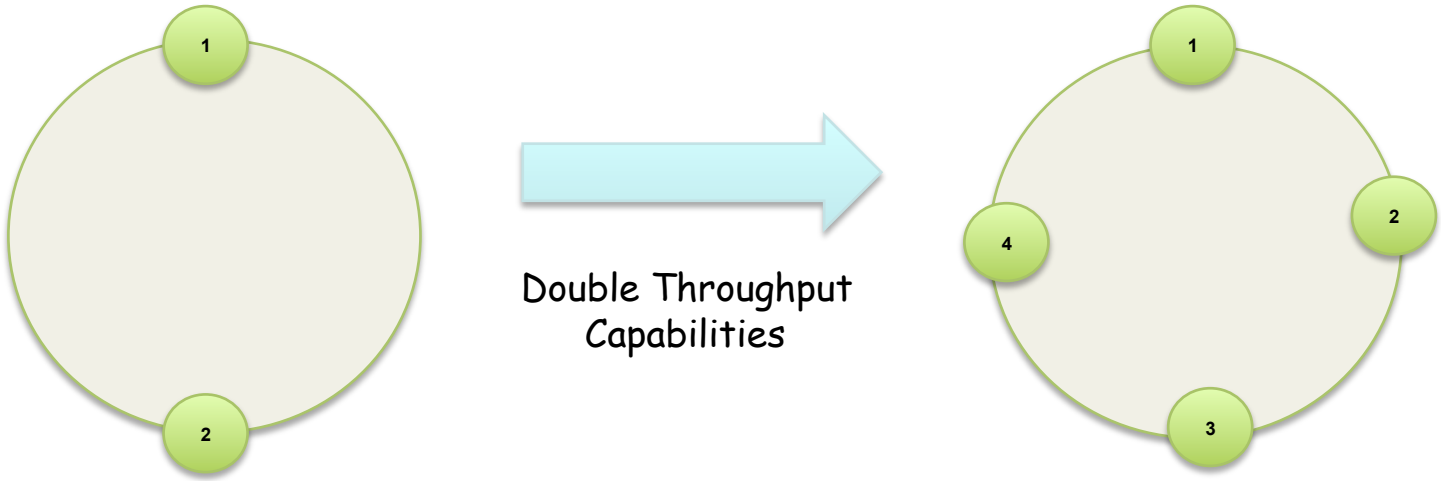


Why Cassandra?

- Gigabyte to Petabyte scalability
- Linear performance gains through adding nodes
- No single point of failure
- Easy replication / data distribution
- Multi-data center and Cloud capable
- No need for separate caching layer
- Tunable data consistency
- Flexible schema design
- Data Compression
- CQL language (like SQL)
- Support for key languages and platforms
- No need for special hardware or software

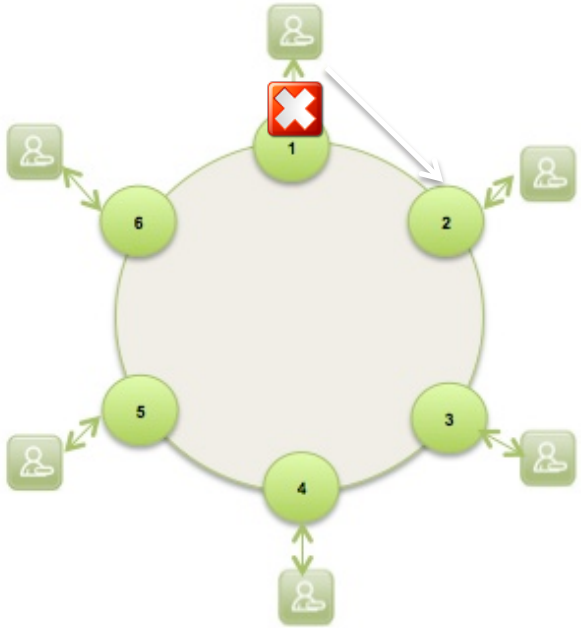
Big Data Scalability

- Capable of comfortably scaling to petabytes
- New nodes = Linear performance increases
- Add new nodes online



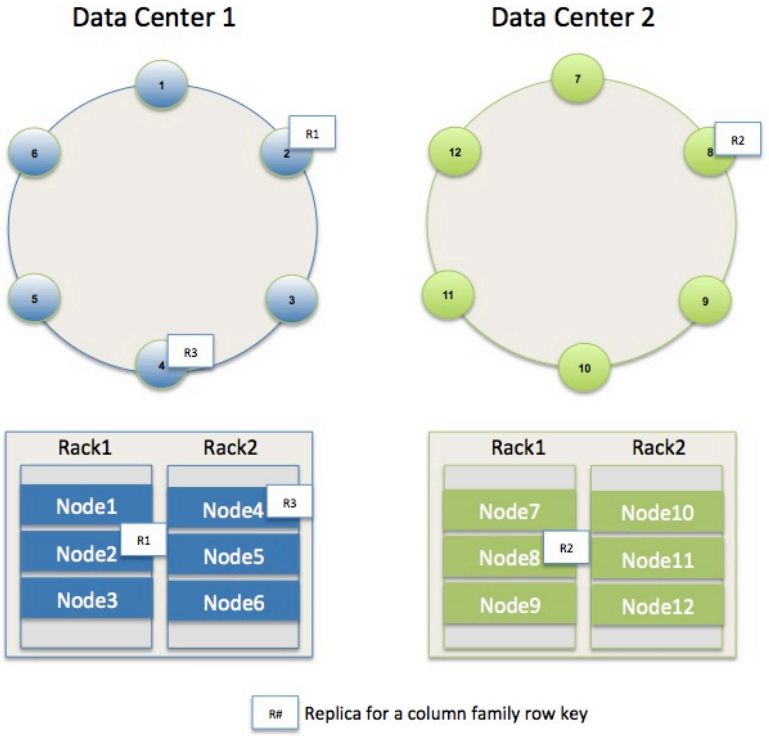
No Single Point of Failure

- All nodes the same
- Customized replication affords tunable data redundancy
- Read/write from any node
- Can replicate data among different physical data center racks



Easy Replication / Data Distribution

- Transparently handled by Cassandra
- Multi-data center capable



Core Functionality of Cassandra

- Handle read/write requests.
- A read/write request for a key gets routed to any node in the Cassandra cluster
- The node then determines the replicas for this particular key.
  - For writes: The system routes requests to the replicas and waits for a "quorum" of replicas to acknowledge the write completion.
  - For reads: Based on the consistency guarantees required by the client, the system routes to the closest replica or to all replicas and waits for a "quorum".

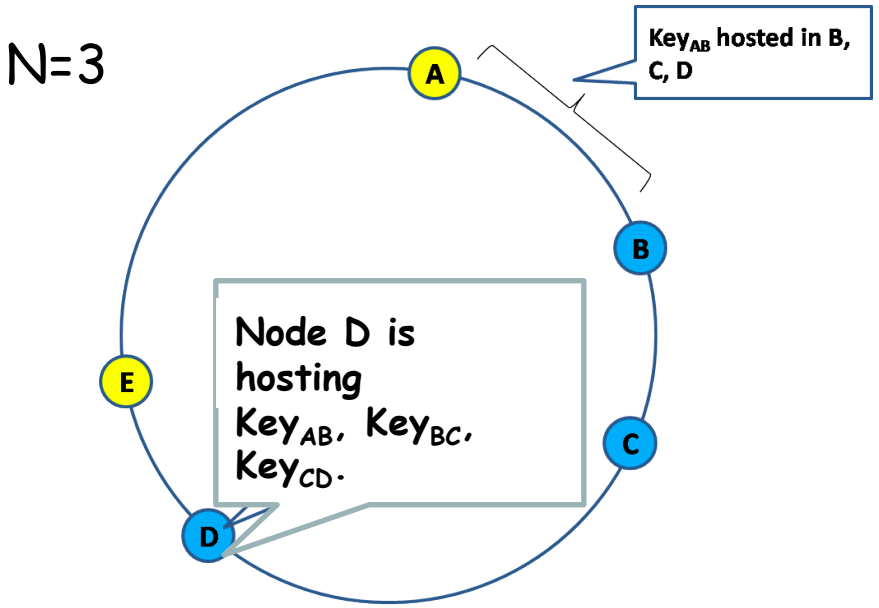
Data Replication in Cassandra

- Data replication is key to achieving **high availability** and **durability**.
  - It also helps improve performance by providing the means for spreading the workload across multiple replicas.

- Cassandra largely borrows from Dynamo's data replication techniques.
  - Replicate each data item at N hosts, where N is the **replication factor** configured "per-instance".
  - In the ring, each node is responsible for the replication of data items in its range (locally store them + replicate them at N-1 other nodes in the ring).

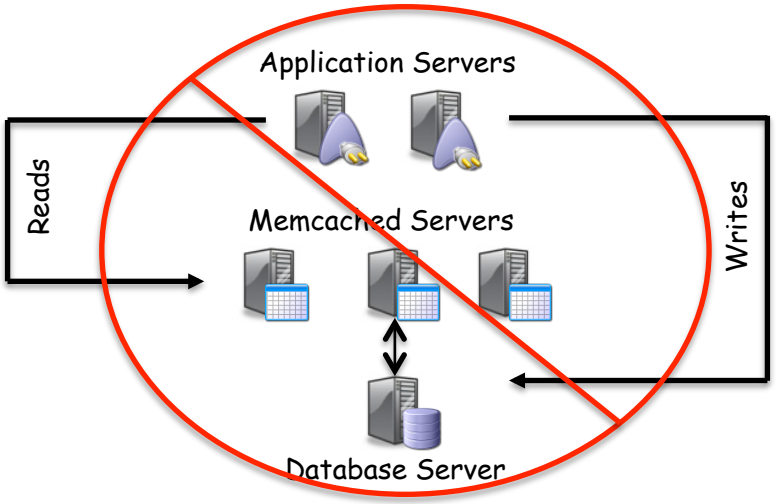
Data Replication in Cassandra

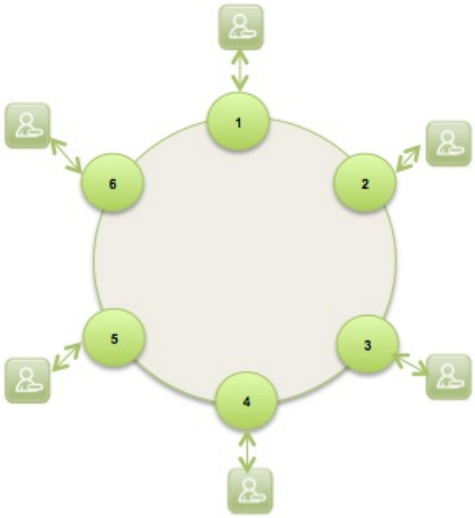
- Applications can choose from multiple replication policies: Rack Unaware, Rack Aware, Datacenter Aware.
- The basic replication policy borrowed from Dynamo is called "Rack Unaware": Store replicas on the subsequent N-1 nodes on the ring.
- Example:



No Need for Caching Software

- Peer-to-peer architecture removes need for special caching layer and the programming that goes with it
- The database cluster uses the memory from all participating nodes to cache the data assigned to each node
- No irregularities between a memory cache and database are encountered



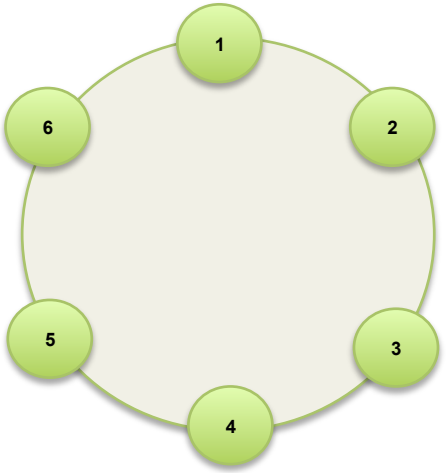


Tunable Data Consistency

- Choose between strong and eventual consistency (All to any node responding) depending on the need
- Can be done on a per-operation basis, and for both reads and writes
- Handles Multi-data center operations

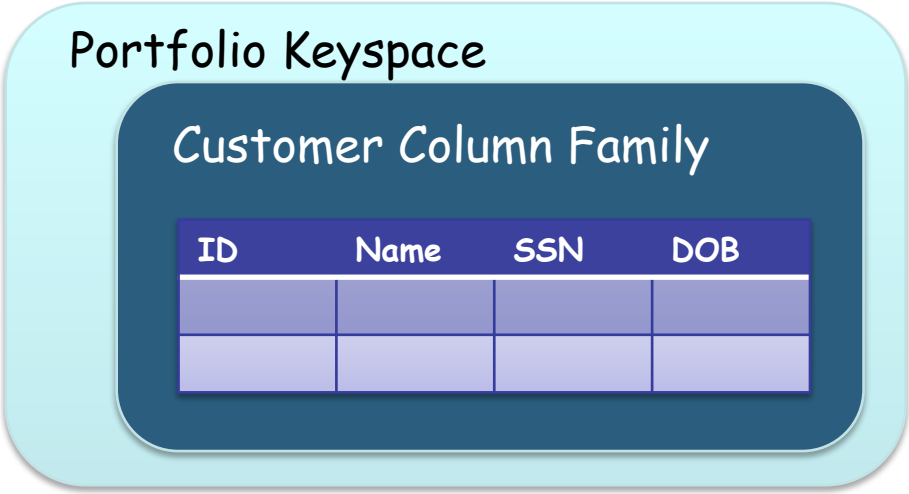
Writes
- Any
- One
- Quorum
- All

Reads
- One
- Quorum
- All



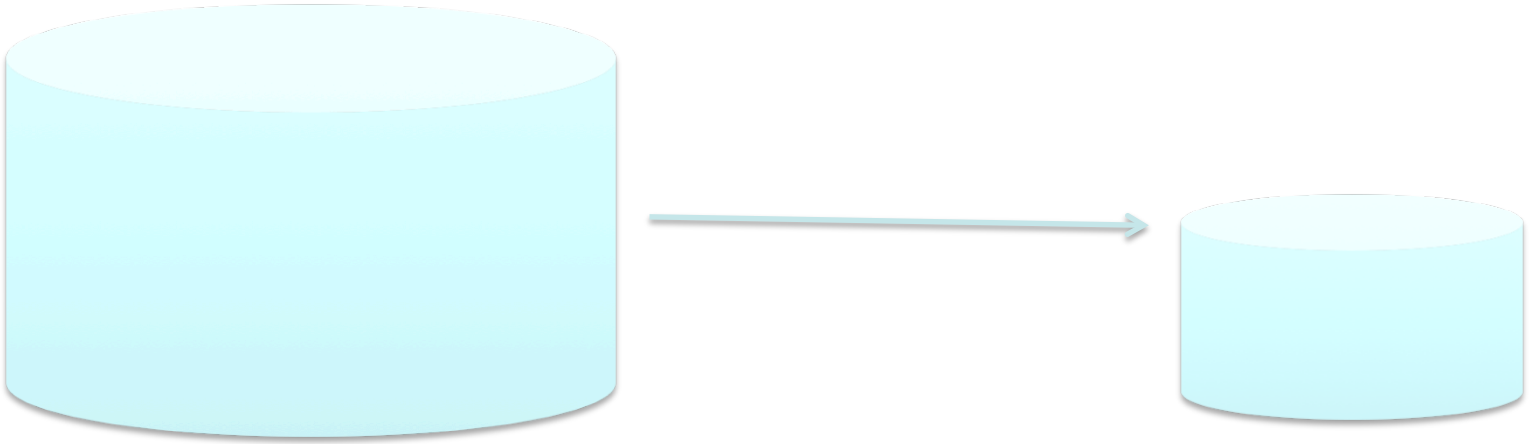
Flexible Schema

- Dynamic schema design allows for much more flexible data storage than rigid RDBMS
- Handles structured, semi-structured, and unstructured data. Counters also supported
- No offline/downtime for schema changes
- Supports primary and secondary indexes



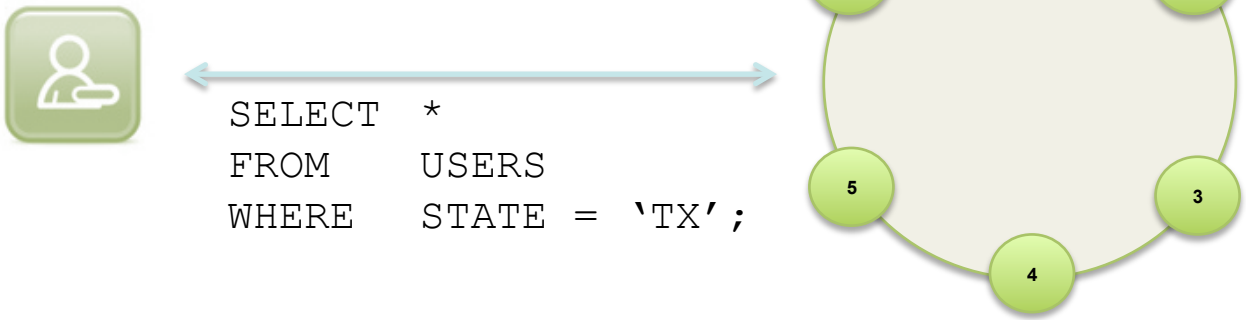
Data Compression

- Uses Google's Snappy data compression algorithm
- Compresses data on a per column family level
- Internal tests at DataStax show up to 80%+ compression of raw data
- No performance penalty (and some increases in overall performance due to less physical I/O)!



CQL Language

- Very similar to RDBMS SQL syntax
- Create objects via DDL (e.g. CREATE…)
- Core DML commands supported: INSERT, UPDATE, DELETE
- Query data with SELECT



Combining Cassandra with Hadoop

- Cassandra: online operations
- Hadoop: offline analytics
- Hadoop on top of Cassandra: analyze data in Cassandra without having to move data to Hadoop

Who's Using Cassandra?

http://www.datastax.com/cassandrausers#all

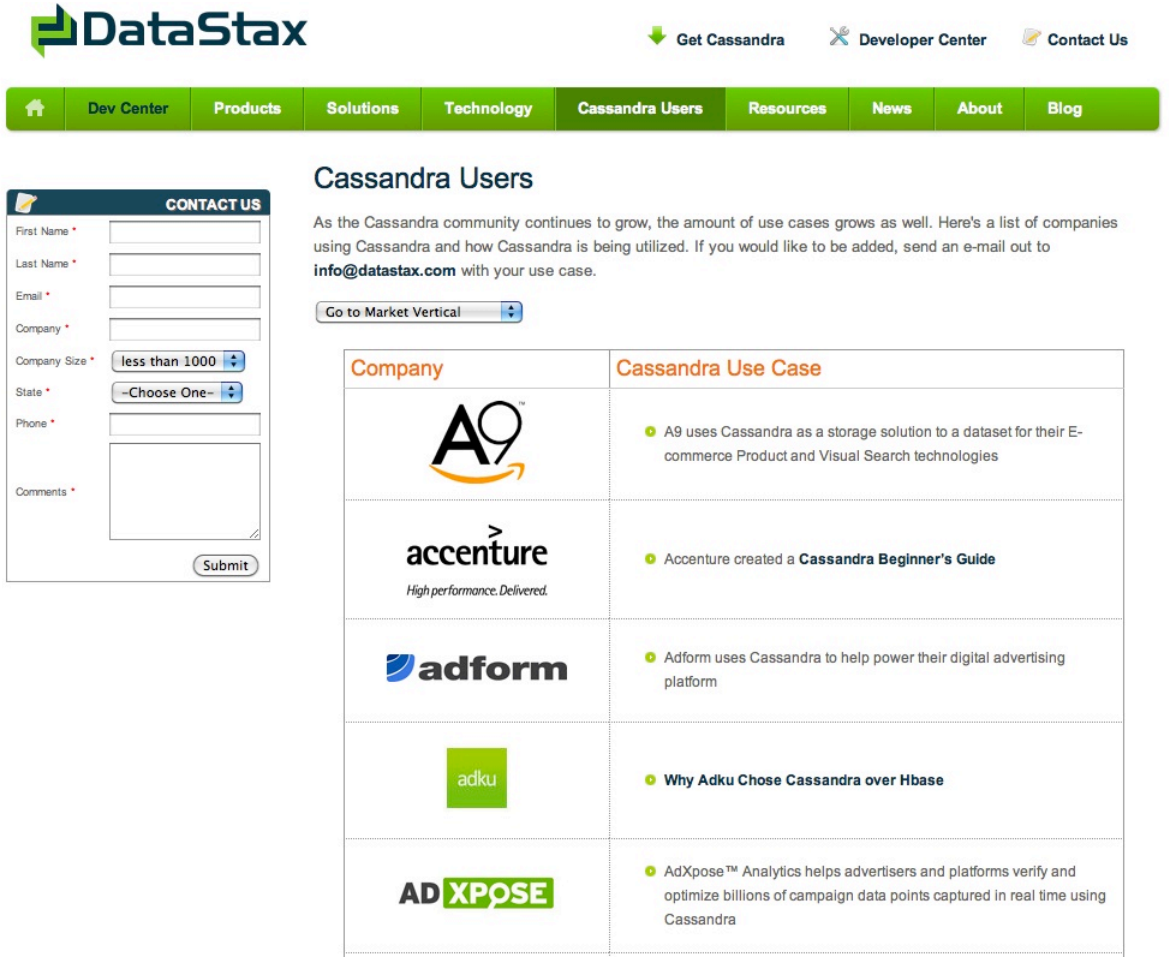
NETFLIX

twitter

cisco

hp invent

UBS

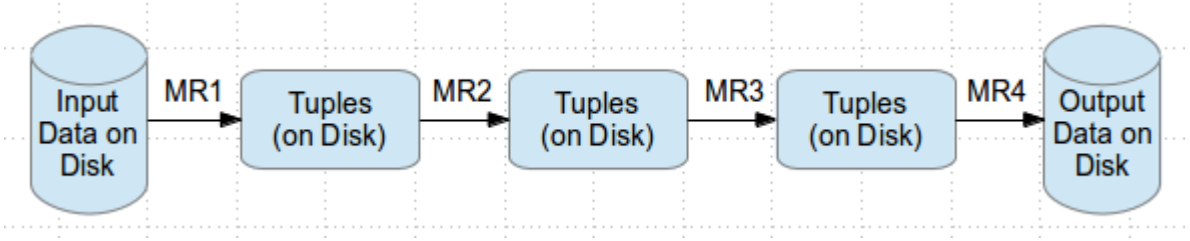


Spark

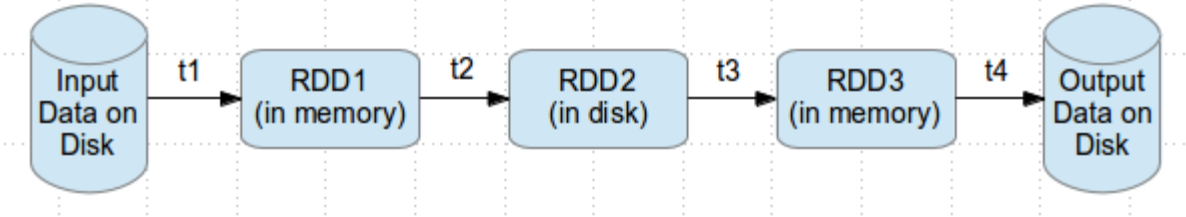
Why Spark ?

- Most of Machine Learning Algorithms are iterative because each iteration can improve the results
- With Disk based approach each iteration's output is written to disk making it slow

Hadoop execution flow



Spark execution flow



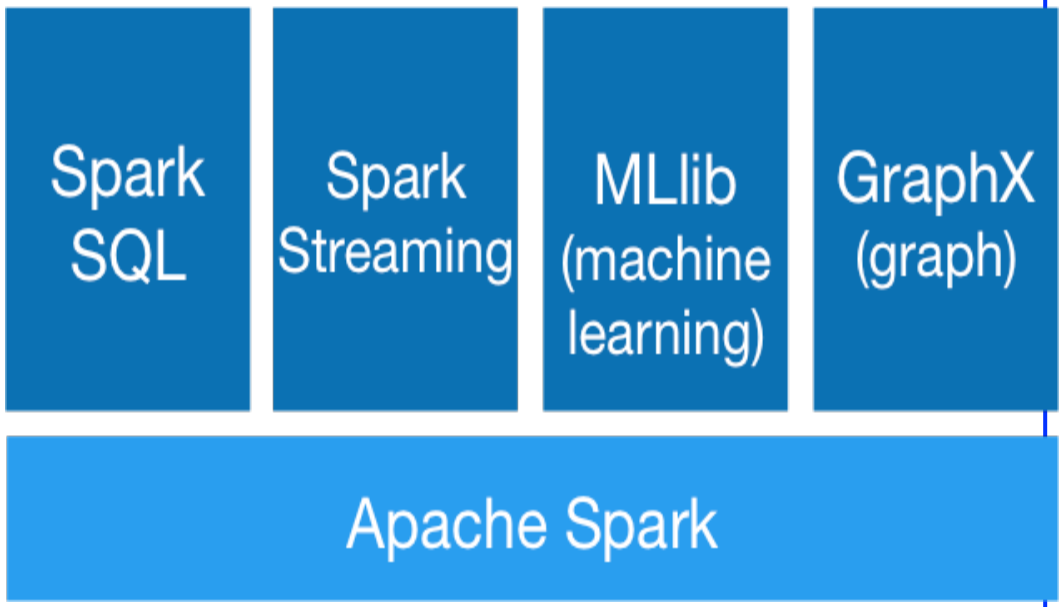
http://www.wiziq.com/blog/hype-around-apache-spark/

About Apache Spark

- Initially started at UC Berkeley in 2009
- Fast and general purpose cluster computing system
- 10x (on disk) - 100x (In-Memory) faster
- Most popular for running *Iterative Machine Learning Algorithms.*
- Provides high level APIs in
  - Java
  - Scala
  - Python
- Integration with Hadoop and its eco-system and can **read existing data.**
- http://spark.apache.org/

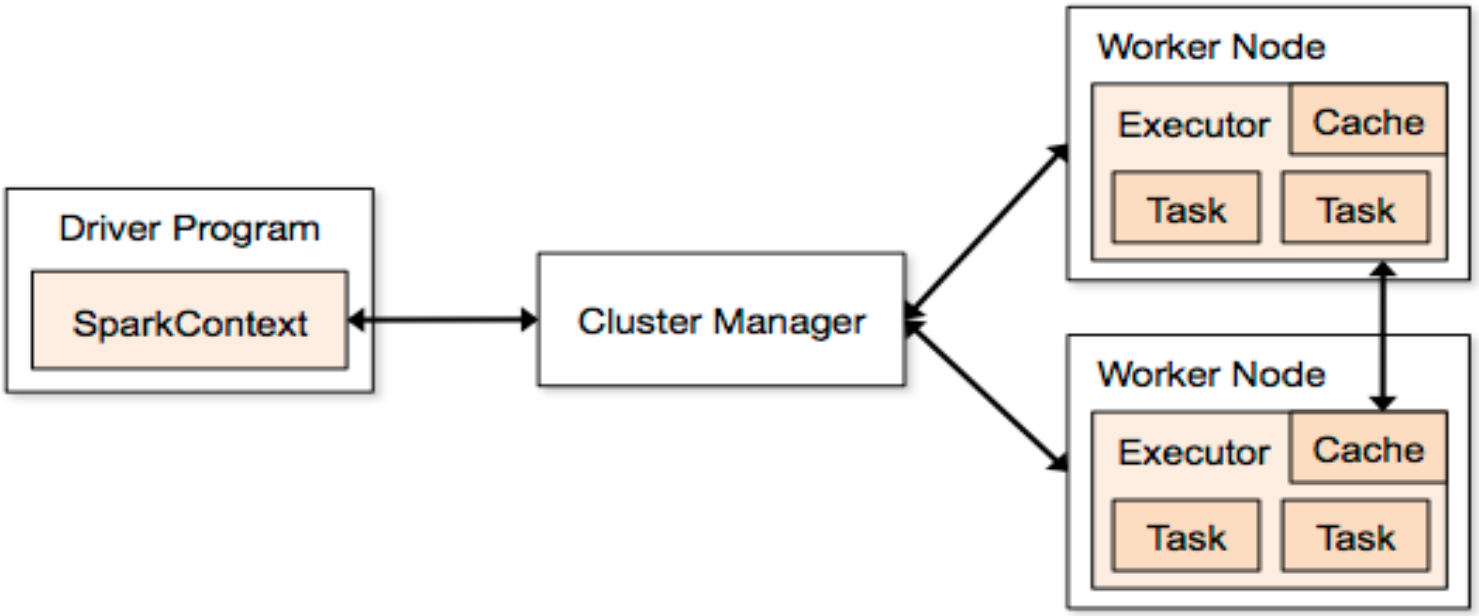
Spark Stack

- Spark SQL
  - For SQL and unstructured data processing
- MLib
  - Machine Learning Algorithms
- GraphX
  - Graph Processing
- Spark Streaming
  - stream processing of live data streams



http://spark.apache.org

Execution Flow



http://spark.apache.org/docs/latest/cluster-overview.html

Terminology

- **Application Jar**
  - User Program and its dependencies except Hadoop & Spark Jars bundled into a Jar file
- **Driver Program**
  - The process to start the execution (main() function)
- **Cluster Manager**
  - An external service to manage resources on the cluster (standalone manager, YARN, Apache Mesos)
- **Deploy Mode**
  - **cluster** : Driver inside the cluster
  - **client** : Driver outside of Cluster

Terminology (contd.)

- **Worker Node :** Node that run the application program in cluster
- **Executor**
  - Process launched on a worker node, that runs the Tasks
  - Keep data in memory or disk storage
- **Task :** A unit of work that will be sent to executor
- **Job**
  - Consists multiple tasks
  - Created based on a Action
- **Stage :** Each Job is divided into smaller set of tasks called Stages that is sequential and depend on each other
- **SparkContext :**
  - represents the connection to a Spark cluster, and can be used to create RDDs, accumulators and broadcast variables on that cluster.

Resilient Distributed Dataset (RDD)

- Resilient Distributed Dataset (RDD) is a basic Abstraction in Spark
- Immutable, Partitioned collection of elements that can be operated in parallel
- Basic Operations
  - map
  - filter
  - persist
- Multiple Implementation
  - PairRDDFunctions : RDD of Key-Value Pairs, groupByKey, Join
  - DoubleRDDFunctions : Operation related to double values
  - SequenceFileRDDFunctions : Operation related to SequenceFiles
- RDD main characteristics:
  - A list of partitions
  - A function for computing each split
  - A list of dependencies on other RDDs
  - Optionally, a Partitioner for key-value RDDs (e.g. to say that the RDD is hash-partitioned)
  - Optionally, a list of preferred locations to compute each split on (e.g. block locations for an HDFS file)
- Custom RDD can be also implemented (by overriding functions)

Cluster Deployment

- Standalone Deploy Mode
  - simplest way to deploy Spark on a private cluster
- Amazon EC2
  - EC2 scripts are available
  - Very quick launching a new cluster
- Hadoop YARN

Storm: Stream Processing on Hadoop

Storm: In a nutshell

- Stream-based real-time processing
  - Map Reduce is designed for batch processing
- Developed as standalone application
- Used by Twitter

Storm Features

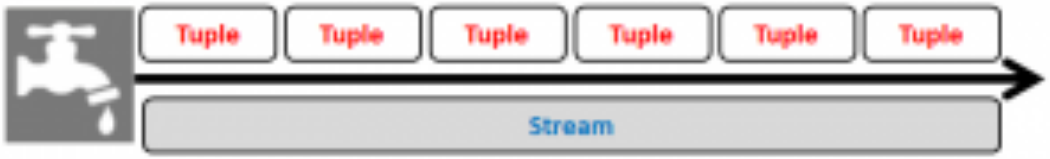
- Scalable and robust
  - No persistent layer
- Guarantees no data loss
- Fault-tolerant
- Programming language agnostic

Topology

- Tuples
  - Ordered list of elements
  - (“user”, “link”, “event”, “10/3/2014, 10:23am”)
- Stream
  - Unbounded sequence of streams

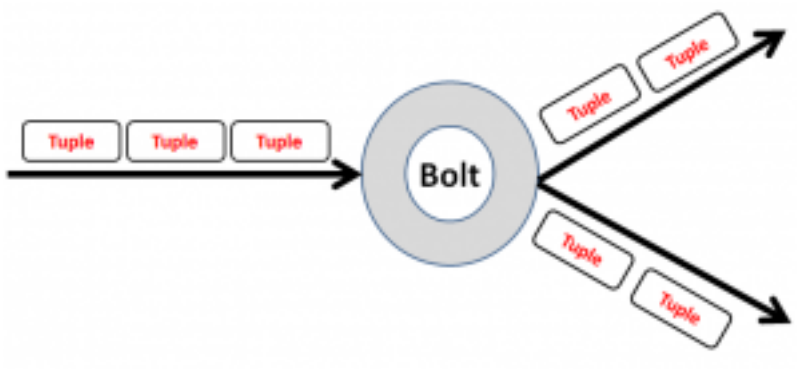
Spouts

- Source of streams
- Example
  - Read from logs, API calls, event data, queues,…



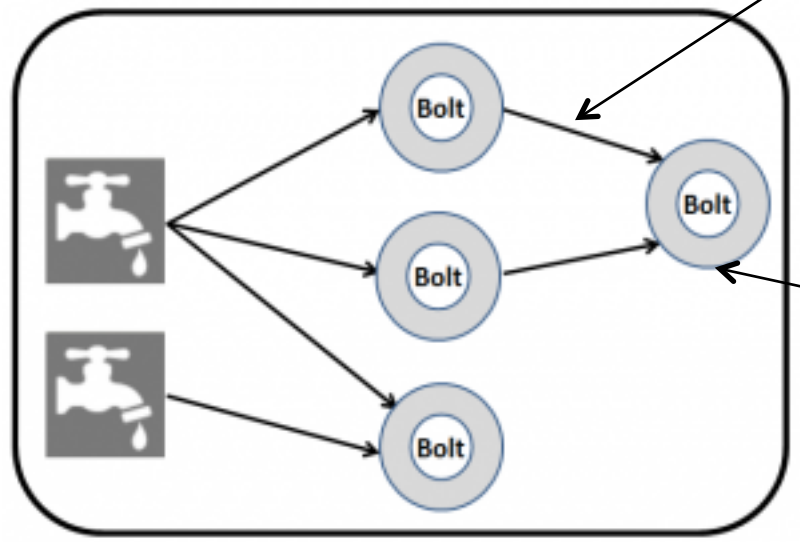
Bolts

- Processes input streams and produces new streams
- Example:
  - Stream joins, filter, aggregation



Topology

- A directed graph of spouts and bolts
  - E.g., computation of stream of trending topics from a stream of tweets



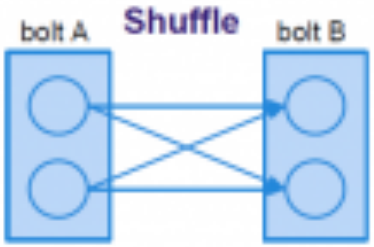
Subscription to streams

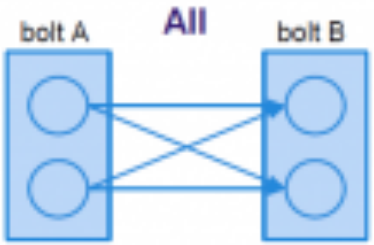
Nodes are executed in parallel (Storm decides assignments to cluster machines)

Stream Grouping

- Specifies how tuples are sent between processing components
  - How the stream will be partitioned across the bolt's tasks

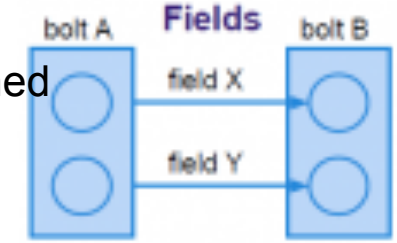
Tuples are shuffled but equally divided

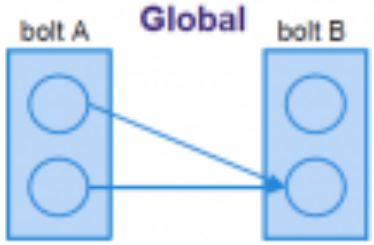




Tuples are replicated

Tuples partitioned based on field values





All tuples go to a single bolt

Other Application on HDFS