Hadoop:
A Cursory Introduction

Erik Scott
escott@renci.org

What is Hadoop?

- An Open-Source reimplementation of Google's Map/Reduce paper

- A **Distributed Filesystem** and a **Runtime** for implementing Map/Reduce algorithms

Hadoop's goals, quickly stated

- Massive Parallelism – 4 nodes to 9000 nodes.
- Does the housekeeping for us – starts jobs, distributes workload evenly, handles failures.
- Gives up generality in exchange for simple, uniform programming problem.
- ...But still rich enough to do a **lot** of useful work.

Map/Reduce

- Grossly simplified:
  - First, do some sort of transformation on every item to process
  - Then, collapse all those results into useful, smaller aggregates.
  - You've probably seen this in a Survey Of Languages course – Map/Reduce dates from the early Lisp days.

Map/Reduce Definitions: Map()

- Map (key1, value1) -> list (key2, value2)
- Or in prosaic terms, “Take a whole bunch of values (“value1” items), and optionally their keys (“key1” items) and add some result of that to a list of new keys and values (“key2 and value2”) items.
- Strictly speaking, this is optional – it's OK to deliberately ignore records (ex: throw out obviously bad data)

Map/Reduce Definitions: Reduce()

- Reduce (key2, list(values2)) -> list(values3)
- In other words, “For every unique value of key2, take all of the associated “values2” items and compute some value “values3”, and save that result as a record in the final result.

Reduce(), cont.

- Hadoop allows a Reduce() function to emit a key and a value. Google's paper uses only the value, formally, but their examples have cases where both are used.
- Reduce(key2, list(values2)) -> list (key3, value3)
- You're allowed to emit empty keys for key3, so it can be just a plain list.

We desperately need an example!

Given a file containing two columns (roomNumber and studentID), produce a file containing the number of students in each room.

```
Broughton1401     30040
Dabney201         48410
Broughton1401     31066
Broughton1401     48410
TompkinsG113      48410
Dabney201         30040
```

The Map() step

For each record <key1, value1>, emit the key back out and a “1”. (This is practically idiomatic: if you see people emitting a “1”, that usually means they're counting something.)

```
Broughton1401     1
Dabney201         1
Broughton1401     1
Broughton1401     1
TompkinsG113      1
Dabney201         1
```

Then the Hadoop runtime does some sorting and shuffling...

The previous list gets combined by keys, and the associated values get strung out into lists:

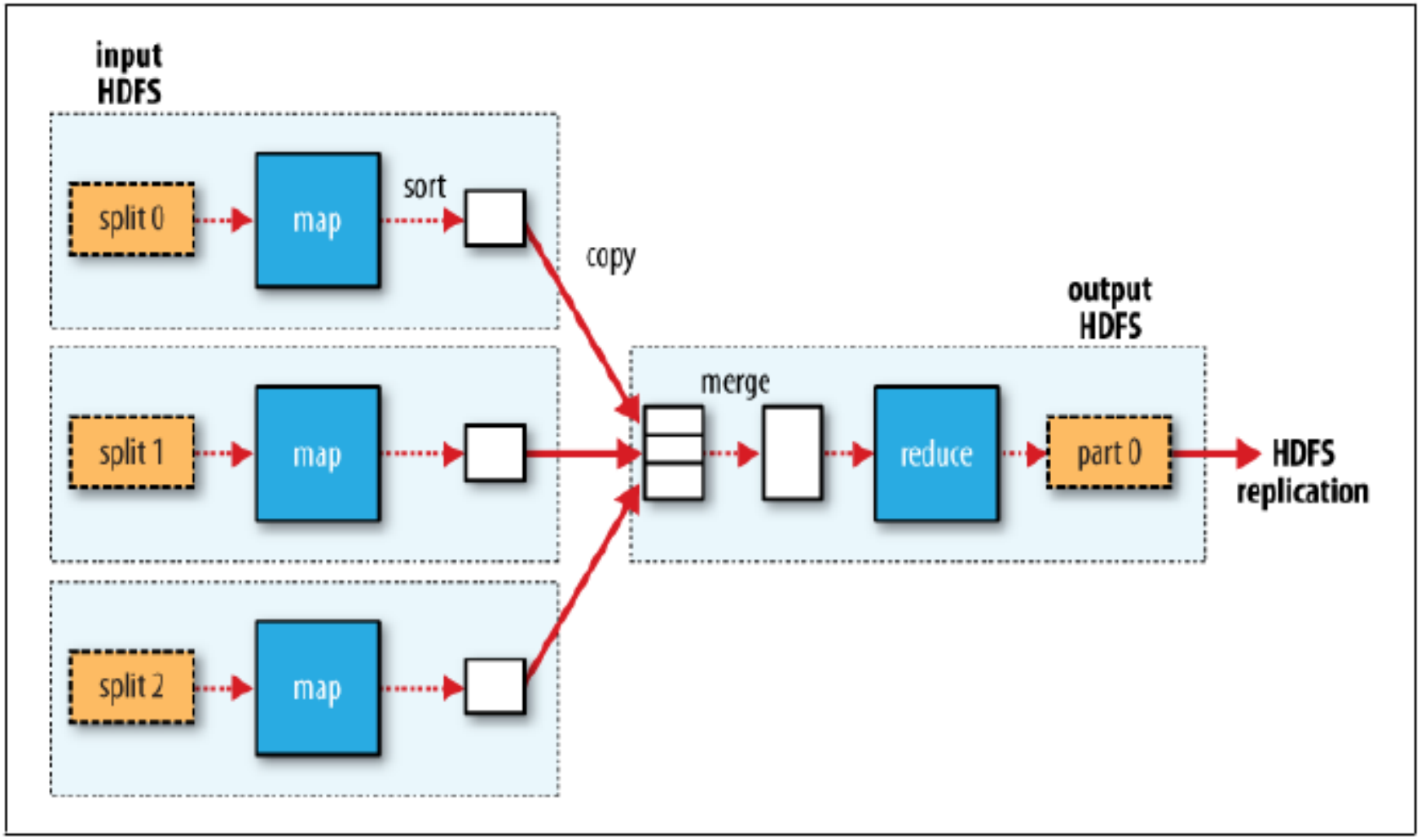
```
Broughton1401    1,1,1
Dabney201        1,1
TompkinsG113     1
```

Finally, we Reduce()

The Reduce(key2, list(value2)) method is called, and it in turn emits rows of the form <key2, value3>, where value3 is the number of list items in list(value2).

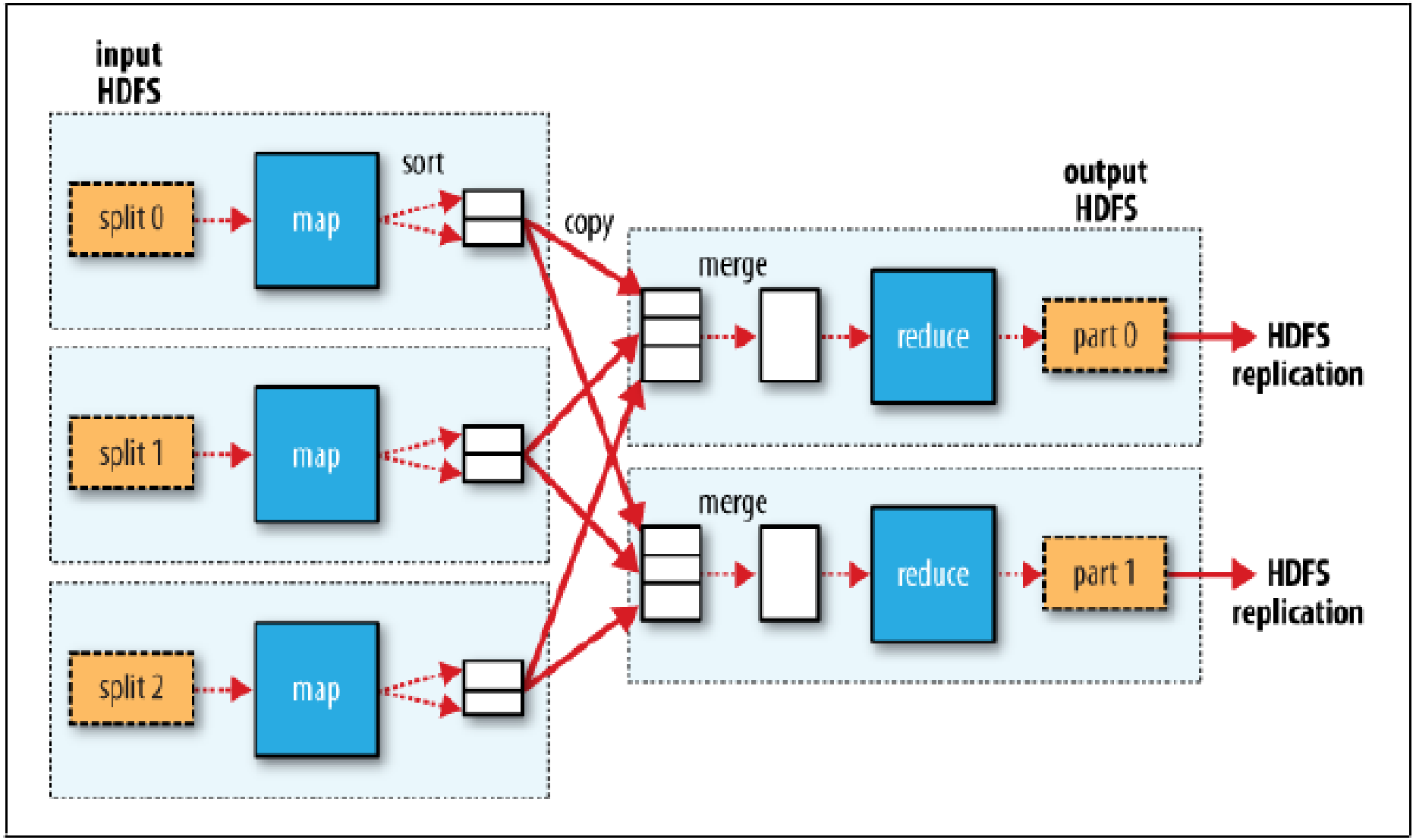
```
Broughton1401   3
Dabney201       2
TompkinsG113    1
```

Dataflow (for one reducer)



From Hadoop: The Definitive Guide, by Tom White

Dataflow for many reducers



From Hadoop: The Definitive Guide, by Tom White

Some Implications of Map/Reduce

- Each input record is processed fully independently of any other record.
- Because of this, input data can be processed in parallel. If we have as many compute nodes as input records, we could theoretically run every Map() for every record in parallel.
- Because the Shuffle/Sort groups items by key, and guarantees that every record with a given key value gets coalesced into one input <key2,list(value2)> pair, we can often have huge parallelism in reducers, as well.

There are some drawbacks, though:

- Every record has to be mapped independently of every other record, and we don't even know the order they will run in.
  - This means sliding windows, signal processing, or particularly complex models, for instance, have to be pushed back into the reduce step (unless we're clever and denormalize our data rather severely)
  - Acting on two independent inputs can be done by holding one in RAM, if it's small enough.
- The Shuffle/Sort step is implemented as an optional hash followed by a merge sort, and for simple mappers the Shuffle/Sort can take longer than the map.

Actual Code for First Example

```
public class StudentsPerRoom {
public static void main(String[] args) throws IOException {
    if (args.length != 2) {
      System.err.println("Usage: StudentsPerRoom <input path> <output path>");
      System.exit(-1);
    }

    JobConf conf = new JobConf(StudentsPerRoom.class);
    conf.setJobName("StudentsPerRoom");

    FileInputFormat.addInputPath(conf, new Path(args[0]));
    FileOutputFormat.setOutputPath(conf, new Path(args[1]));

    conf.setMapperClass(StudentsPerRoomMapper.class);
    conf.setReducerClass(StudentsPerRoomReducer.class);

    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);

    JobClient.runJob(conf);
  }
}
```

First Example Mapper

```
public class StudentsPerRoomMapper extends MapReduceBase implements Mapper<LongWritable,
Text, Text, IntWritable> {
  private Text room = new Text();
  private final static IntWritable one = new IntWritable(1);


  public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> output, Reporter
reporter)
      throws IOException {


  String line =value.toString();
  StringTokenizer tokenizer = new StringTokenizer(line);


  if (tokenizer.hasMoreTokens()) {
  room.set(tokenizer.nextToken());
  output.collect(room, one);
  }
  }
}
```

First Example Reducer

```
public class StudentsPerRoomReducer 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));
  }
}
```

Hadoop Streaming

- Writing Map/Reduce jobs in Java is very flexible, but not terribly productive.
- Wouldn't it be nice if we could write Map() and Reduce() functions in friendlier languages, like Python? Or Shell Scripts? Or FORTRAN?
- Hadoop Streaming lets you write Map and Reduce functions that read from standard input and write to standard output.

Streaming Mapper, in Python

```
#!/usr/bin/python
import re
import sys

for line in sys.stdin:
    vals = line.strip()
    key = vals.split(' ')[0]
    print key, "\t", "1"
```

Streaming Reducer

```
#!/usr/bin/python
import re
import sys

(last_key, sum_val) = (None, 0)
for line in sys.stdin:
    (key, val) = line.strip().split("\t")
    if last_key and last_key != key:
        print last_key, "\t", sum_val
        (last_key, sum_val) = (key, int(val))
    else:
        (last_key, sum_val) = (key, sum_val + int(val))

if last_key:
    print last_key, "\t", sum_val
```

A little bit about HDFS

- Hadoop Distributed Filesystem – slow writes, sequential writes only, at most one thread writing, no editing of the file later, but BRUTALLY fast reads.
- Not a mounted filesystem, like a thumb drive, but a remote filesystem that acts a more like an FTP site.

Using HDFS

- `hadoop fs -ls`
- `hadoop fs -put` *`localFile hdfsFile`*
- `hadoop fs -get` *`hdfsFile localFile`*
- `hadoop fs -mkdir` *`hdfsDirName`*
- etc.
- `hadoop fs` `#` no arguments, prints online help.

HDFS Performance

- Distribution across the network is used to increase the chances that a Map() function will run on the same node that the Datanode for its block is on.
- In other words, increases the odds of moving the computation to the data (fast), instead of moving data to the computation (slow)
- Reducers? Not too much we can to help there.

HDFS Performance

- MacBook Pro, 2.5” 320 gig 5400 rpm disk drive: **40 megabytes/sec**
- 34 node Hadoop cluster with 10,000 rpm 600 gigabyte 2.5” SAS drives: **500 megabytes/sec**
- 300 node Hadoop cluster with 5400 rpm 40 gigabyte 2.5” laptop-grade drives: **1.25 gigabytes/sec**.
  - Or, reads 100+ million rows per second

Hadoop at Scale

- Lousy for small jobs – you lose the first few seconds just starting up Java and waiting for classes to load.
- Very much a batch-oriented system. You **could** use it for interactive work, but there is better stuff for that.
- 2000 nodes is considered a practical upper limit, though 9000 has been demonstrated.

Apache Pig:

A package for Relational Algebra on top of Hadoop

Motivation

- Pig is a software package for doing Relational Algebra very quickly.
- Resembles a Relational Database (MySQL, Postgres, Oracle...)
- A good way to very rapidly write Map/Reduce workflows without resorting to Java.

Projection

- Project (sname, rating) onto S2:

```
grunt> A = load 's2' using PigStorage(':');
grunt> B = foreach A generate $1, $2;
grunt> dump B;
(yuppy,9)
(lubber,8)
(guppy,5)
(rusty,10)
```

Selection

- Select rows where rating > 8 from S2

```
C = filter A by $2 > 8;
Dump C;
(28,yuppy,9,35.0)
(58,rusty,10,35.0)
```

Compounding selection and projection

- Another ex: $p_{sname,rating}(s_{rating>8}(S2))$

```
A = load 's2' using PigStorage(':') as (sid, sname, rating, age);
B = filter A by rating > 8;
C = foreach B generate sname, rating;
dump C;

(yuppy,9)
(rusty,10)
```

Union

- Union of S1 and S2 – note duplicates!

```
A = load 's1' using PigStorage(':') as (sid, sname, rating, age);
B = load 's2' using PigStorage(':') as (sid, sname, rating, age);
C = union A, B;
dump C;
(22,dustin,7,45.0)
(31,lubber,8,55.5)
(58,rusty,10,35.0)
(28,yuppy,9,35.0)
(31,lubber,8,55.5)
(44,guppy,5,35.0)
(58,rusty,10,35.0)
```

Must explicitly remove duplicates

- DISTINCT

```
D = distinct C;
dump D;
(22,dustin,7,45.0)
(28,yuppy,9,35.0)
(31,lubber,8,55.5)
(44,guppy,5,35.0)
(58,rusty,10,35.0)
```

Cross Product

Cross product of S1 and R1:

```
S1 = load 's1' using PigStorage(':')  as (sid:int, sname:chararray, rating:int, age:float);
R1 = load 'r1' using PigStorage(':') as (sid:int, bid:int, day:chararray);
CROSSPROD = CROSS S1, R1;
dump CROSSPROD;
```

```
(22,dustin,7,45.0,22,101,19961010)
(22,dustin,7,45.0,58,103,19961112)
(31,lubber,8,55.5,22,101,19961010)
(31,lubber,8,55.5,58,103,19961112)
(58,rusty,10,35.0,22,101,19961010)
(58,rusty,10,35.0,58,103,19961112)
```

Equijoin

```
S1 = load 's1' using PigStorage(':')  as
(sid:int, sname:chararray, rating:int,
age:float);
R1 = load 'r1' using PigStorage(':') as
(sid:int, bid:int, day:chararray);
EJ = join S1 by sid, R1 by sid;
dump EJ;
```

```
(22,dustin,7,45.0,22,101,19961010)
(58,rusty,10,35.0,58,103,19961112)
```

Special Extras in Pig

- Not strictly relational: relations can contain non-atomic data (even other relations).
  - See GROUP, FLATTEN
- Can, like any modern database, execute user-defined functions (“UDF”s), in this case written in Java or Python
- Can execute user-defined Java map and reduce functions inside the Hadoop framework, so these UDFs can run very quickly in parallel.

Non First Normal Database

- Items need not be the atomic types supported by pig (int, long, float, double, chararray, bytearray, boolean), but in fact can be a:
  - Tuple – an embedded “row” within a cell.
    - ('Poland', 2, 0.66)
  - Bag – a collection of tuples (more or less a “relation”, if you relax the first normal requirement)
    - {('Poland', 2), ('Greece'), ('3.14')}
  - Map – an associative array
    - ['UNC'#'Chapel Hill, NC','NCSU'#'Raleigh, NC']

Coming up next...

- Gee, what if we could combine the brute-force parallelism of Hadoop with the elegance, productivity, and convenience of a declarative programming language like SQL?
- Well then you'd have Apache Hive...

Hive:
A Relational Database With a Mostly SQL-like Language

Motivation

- Hadoop requires Java knowledge
  - Yes, there are workarounds, but incomplete
- Pig is better
  - But the language is still arcane, and just enough like SQL to really mess you up.
- Facebook wanted something big and cheap like Hadoop, but usable by analysts who only knew SQL.

Hive in a Nutshell

- Stores tables as human-readable flat files, delimited by “ctrl-A” characters.
- Performs relational algebra mostly like Pig
- Stays much closer to the relational model
- Tuples usually contain atomic values, but can contain structures, maps, or arrays.
- Front end supports a query parser much like SQL, a socket interface for remote connections (even JDBC), and actual support for real metadata.

A worked example in Hive

- Create Tables
- Load data
- Execute Queries
- Save Results

CREATE TABLE

```
CREATE TABLE enrollment (classID INT, studentID INT);


CREATE TABLE classes (classid INT, course STRING,
section INT);


SHOW TABLES;
OK
enrollment
classes
Time taken: 0.059 seconds, Fetched: 2 row(s)
```

Loading

load data local inpath '/home/escott/projects/hiveEx/enrollment.tab' overwrite into table enrollment;

load data local inpath '/home/escott/projects/hiveEx/classes.tab' overwrite into table classes;

Selects

```
select * from classes;
OK
7       ece201  1
8       ece202  1
9       ece204  13
Time taken: 0.038 seconds, Fetched: 3 row(s)
```

SELECT with a JOIN

SELECT studentID, course, section
FROM enrollment e
JOIN classes c on (e.classID = c.classID);

OK

4 ece201 1

22 ece201 1

13 ece202 1

19 ece202 1

19 ece204 13

Time taken: 4.528 seconds, Fetched: 5 row(s)

Saving results (and outer join)

```
CREATE TABLE results (studentID INT,
course STRING, section INT);


INSERT OVERWRITE TABLE results select
studentID, course, section from
enrollment e JOIN classes c on
(e.classID = c.classID);
```

...and seeing the results

select * from results;

OK

4 ece201 1

22 ece201 1

13 ece202 1

19 ece202 1

19 ece204 13

Time taken: 0.044 seconds, Fetched: 5 row(s)

Aggregate Functions

- Works just like you expect...

```
SELECT max(classID) from enrollment;
OK
9
```

Aggregate Functions

- COUNT(), COUNT(DISTINCT)
- SUM()
- AVG()
- MIN()
- MAX()

Other Useful Stuff You Can Do With Hadoop

Apache Mahout

- A package of “machine learning” algorithms for Hadoop

Mahout Algorithm Families

- Recommender Documentation
- Restricted Boltzmann Machines
- K-Means Clustering
- Fuzzy K-Means
- Canopy Clustering
- Mean Shift Clustering

- Hierarchical Clustering
- Dirichlet Process Clustering
- Latent Dirichlet Allocation
- Collocations
- Dimensional Reduction
- Expectation Maximization

- Gaussian Discriminative Analysis
- Independent Component Analysis
- Principal Components Analysis
- Bayesian
- Locally Weighted Linear Regression

- Logistic Regression Page: Neural Network
- Hidden Markov Models
- Random Forests
- Perceptron and Winnow

- Support Vector Machines
- Parallel Frequent Pattern Mining
- Boosting
- Collaborative Filtering with ALS-WR
- Itembased Collaborative Filtering

- Minhash Clustering
- Online Passive Aggressive
- Online Viterbi
- Parallel Viterbi
- RowSimilarityJob
- Spectral Clustering
- Stochastic Singular Value Decomposition
- Top Down Clustering

Apache Giraph

- Graph Traversal and Manipulation
- Developed at Facebook
- More of a programmer's library than a sit-down interactive tool
- The big, open-source player in parallel distributed graph databases is Neo4J. It doesn't use Hadoop...

Apache HBase

- Very high rate of insertions and selections
- For interactive use, not as well suited for batch processing
- Based strongly on Google's BigTable

IBM's BigSheets

- An interface between Pig and Excel
- ...hence it looks like a spreadsheet.
- ...but has some computational muscle behind it.

SAS's newly-announced partnership

- SAS to work with Hortonworks to connect SAS to Hadoop.
- Press releases make it sound like adding Hadoop (Pig? Hive?) as a provider for SAS/Access.

R

- Revolution Analytics' commercial package to integrate R and Hadoop.
- RHadoop – open source, free. Installation is... “tedious”.