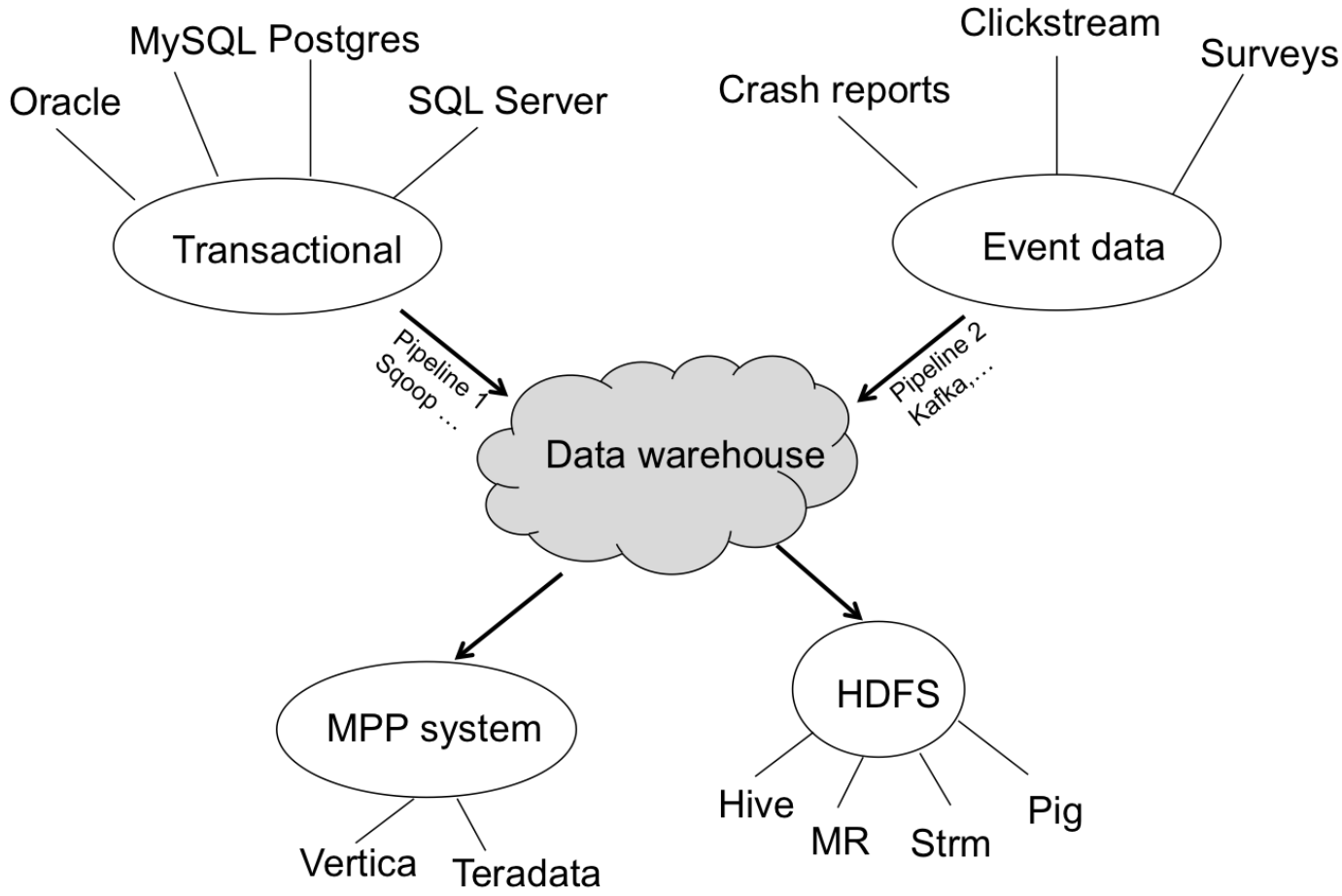
Introduction to Hadoop for data scientists

Rohan Kekatpure

Intuit, Inc

rohan_kekatpure@intuit.com

June 26, 2015



Types of data

- Transactional data
- Event data

Modern data warehouses house transactional as well as event data

Transactional data

Who did what transaction when and where?

- Dimension (DIM) tables for who, what, when, where
- Fact (FACT) tables for transactions

cust_id	cust_name	cust_street	...
C001	John	123 Hope St.	...

Table: DIM_CUSTOMER

prod_id	prod_name	prod_price	...
P001	Electric shaver	$99.00	...

Table: DIM_PRODUCT

store_id	store_street	store_zip	...
S001	San Antonio Rd	94043	...

Table: DIM_STORE

trans_id	prod_id	cust_id	store_id	...
T001	P001	C001	S001	...

Table: FACT_SALE

Transactional data

Transactional data tends to be

- normalized (no repetitions, star schema, . . .)
- row oriented
- optimized for fast look up and simple queries
- optimized for quick updates, inserts, deletes
- less optimized for complex analytic queries
- stored in MySQL, Oracle, SQL Server, . . .

Transactional $\rightarrow$ Analytic

- Denormalize (introduce repetition)
- Aggregate
- Join
- Store in analytic databases (Vertica, Netezza, Teradata, . . .)

Example

List all transactions for top 10 most popular products in all stores

Event data

Event data is

- Data about events generated by humans and machines
- Clickstream, server logs, crash-reports, ...

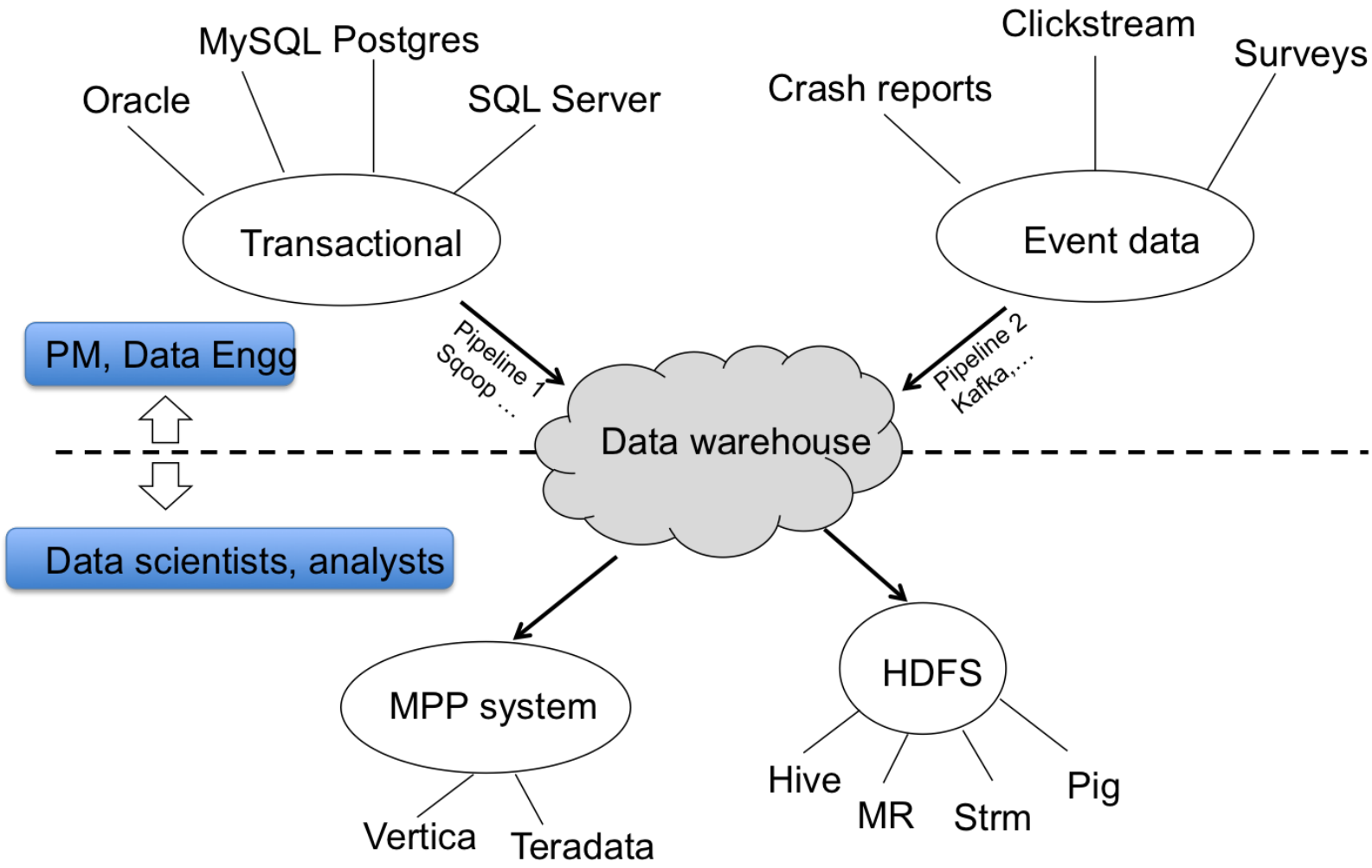
Event data tends to be

- schemaless, free-form
- textual, JSON
- streaming
- large, redundant
- painful to organize into tables
- better suited for graph-based organization

Event $\rightarrow$ Analytical

- Parse JSON
- Define schema
- Create Hive tables
- Hive $\rightarrow$ Vertica, Netezza
- (or) MapReduce jobs

Working with Hadoop



HDFS basic operations

```
# Create HDFS dir, optionally with a path
$ hadoop fs -mkdir -[p] <hfdsdir>

# Local >> HDFS
$ hadoop fs -put <localfile> <hdfsdir>
$ hadoop fs -copyFromLocal <localfile> <hdfsdir>

# HDFS >> local
$ hadoop fs -get <hdfsfile> <localfile>
$ hadoop fs -copyToLocal <hdfsfile> <localfile>

# HDFS <> HDFS
$ hadoop fs -cp <hdfsfile1> <hdfsfile2>
$ hadoop fs -mv <hdfsfile1> <hdfsfile2>

# Append to an existing HDFS file
$ hadoop fs -appendToFile <localfile> <hdfsfile>

# Output raw file contents
$ hadoop fs -cat <hdfspathpattern>
$ hadoop fs -cat user/rkekatpure/sandbox/*/*

# Output contents as plain text
$ hadoop fs -text <hdfspathpattern>
```

Try it!

```
# Try the above commands with sample data
cp /home/rkekatpure/tutorials/hadooptut/schdist.txt .
```

Hive

- Tabular view of text files
  - Files are in HDFS
  - Can be compressed or plain text
- SQL-like queries on those files
- Query execution engine is MapReduce, not SQL
- Tables are partitioned by key (e.g. hdfs://year/month/day/hour)
- Not suited for real time or transactional

Some tips for effective queries

- Restrict queries to partitions, whenever possible
  - `select * from <table> where year='2015' and month='06'`
  - Often makes impossible queries feasible
- Reduce intermediate data transfer volume
  - `set mapred.compress.map.output=true;`
  - `set mapred.compress.output=true;`
- Enable parallelism: perform independent MR steps concurrently
  - `set hive.exec.parallel=true;`
- Process multiple rows simultaneously
  - `set hive.vectorized.execution.enabled=true;`
- Play with query execution engines
  - `set hive.execution.engine=tez;`
  - (default)`set hive.execution.engine=mr;`
- Use sampling instead of full table scan
  - `select * from <table> tablesample(bucket 1 out of 100 on <col>);`

For more...

10 tips for effective hive queries

Hive operations

- Get table information
  - `describe [formatted] <tablename>;`
  - Partition information, file format, file location, schema, ...
- Hive from bash shell
  - `hive -e 'select * from <table>;'`
  - `hive -f <file.hql>`
  - Complex query logic spanning multiple databases or tables
  - List all hive tables in all hive databases

```
for DB in $(hive -e 'show databases'); do
    for TBL in $(hive -e 'use $DB; show tables'); do
        echo ${DB}.${TBL}
        hive -e 'select count(*) from ${DB}.${TBL};'
    done
done
```

- Can use Bash or Python for downstream processing
  - `hive -f <file.hql> | awk -F"\t" ...  | sort ...`

Hive internal tables

- Managed by Hive
- Data deleted if internal tables are dropped

Create internal table

```
# Generate HiveQL for internal table creation
$ ./create_schema_int.sh

# Execute the generated HiveQL
$ hive -f schdist_schema_int.hql

# Drop the table
$ hive -e 'drop table hadooptut_schdist_int;'

# Check if the file still exists
$ hadoop fs -ls
/user/hive8/warehouse/hadooptut_schdist_int

# What do you see?
```

Hive external tables

- Hive is a view
- Data persists after external tables are dropped

Create external table

```
# Generate HiveQL for external table creation
$ ./create_schema_ext.sh

# Execute the generated HiveQL
$ hive -f schdist_schema_ext.hql

# Now drop the table
$ hive -e 'drop table hadooptut_schdist_ext;'

# Check if the file still exists
$ hadoop fs -ls hdtutorial/schools

# What do you see?
```

Python UDFs in Hive

- UDF = User defined function
- Allows non-standard operations (non-standard = not supported natively in Hive)
- Invoked during `select` statement

Steps

In Python:

- Write your Python UDF to consume data from standard input (`stdin`)
- Assume `tab` delimited items passed to `stdin`
- Do your work in the UDF
- Write results to `stdout`
- Make python script executable (permissions and shebang)

In Hive:

- Register your UDF
- Use it in Hive `select transform(...)` statement

Python UDFs in Hive

udf_string.py

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

def add_dashes(word):
    return "-".join(c for c in word)

for line in sys.stdin:
    name, city = map(str, line.strip().split("\t"))
    print "\%s, \%s" %(name, city)
    #print "\%s:\%s" %(add_dashes(name), city.capitalize())
```

test_udf_string.hql

```
add file /home/rkekatpure/tutorials/hadooptut/udf_string.py;
set mapred.job.queue.name=exp_dsa;

select transform(hd.name, hd.mcity)
using 'python udf_string.py' as ts
from hadooptut_schdist_ext hd
limit 100;
```

Python UDFs in Hive

udf_timestamp.py

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

for line in sys.stdin:
    time_units = map(int, line.strip().split("\t"))
    dt = datetime.datetime(*time_units)
    print int(dt.strftime("\%s"))
```

test_udf_timestamp.hql

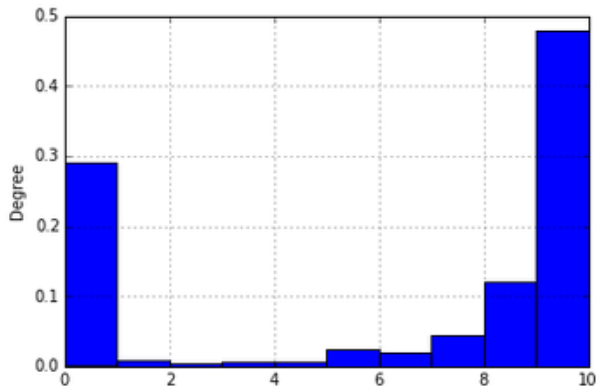
```
add file   /home/rkekatpure/tutorials/hadooptut/udf_timestamp.py;
set mapred.job.queue.name=exp_dsa;
use hive_database;

select transform(s.year, s.month, s.day, s.hour)
using 'python udf_timestamp.py' as ts
from sbg_us_clickstream_mobile s
where year='2015' and month='06' and day='02'
limit 100;
```

Hive + IPython

Hive usage through iPython notebook

In [7]:
```
import pandas as pd
import matplotlib.pylab as pl
import re
%matplotlib inline

# Compose Hive query string
qry_str = "hive -S -e 'set hive.cli.print.header=true; \
select * from survey.survey_responses limit 10000' "

# Embed the query in SSH, make sure to ignore stderr
ssh_qry = ''' ssh iac-prod "%s" 2>/dev/null ''' % (qry_str)

# Execute the query in Command line
results = !$ssh_qry

# Create table (list of lists) by splitting on TAB
table = [row.split("\t") for row in results]

# Create data frame
df = pd.DataFrame(table[1:], columns=table[0])
```

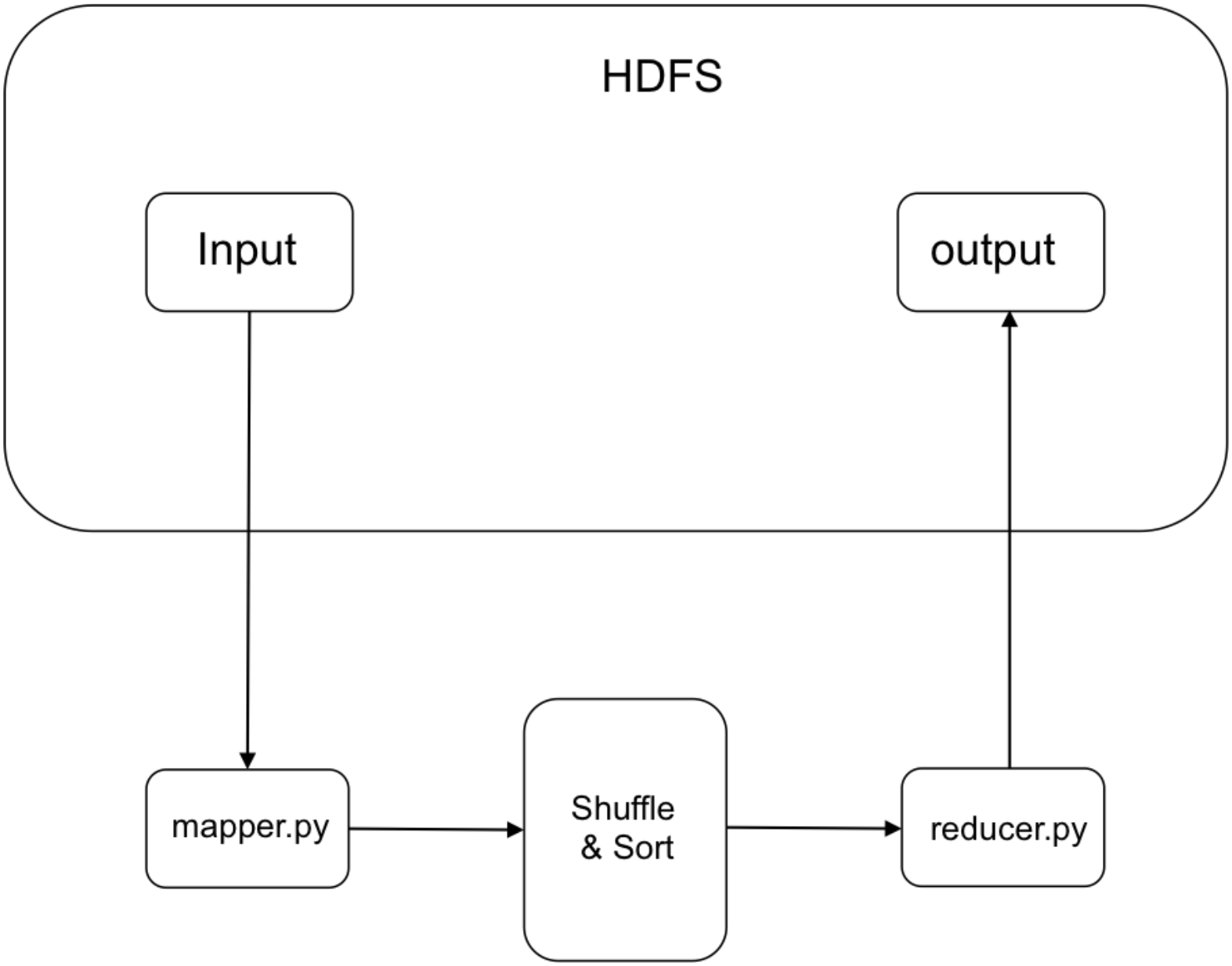
In [8]:
```
# Plot histogram for answer_0
df["answer_0"].apply(lambda x: int(x) if re.match("^[0-9]+$", x) else 0).plot(kind="hist", normed=True, bins=10)
pl.show()
```



Setting up passwordless SSH

```
# On your local machine
$ ssh-keygen -t rsa

# Execute and enter password when prompted
$ ssh username@remotehost "mkdir ~/.ssh"

#
$ cat ~/.ssh/id_rsa.pub | \
    ssh username@remotehost \
    "cat >> ~/.ssh/authorized_keys"
```

MapReduce using Python streaming



Python mapper

- Solve standard word-count problem
- `cp /home/rkekatpure/tutorials/hadooptut/big.txt .`
- Push to `streaming/` dir in your HDFS home
- Create `<your-home>/streaming/output` dir in your HDFS home

hdstreaming_mapper.py

```
#!/usr/bin/python -O

import sys
import string
sys.path.append(".")

def standardize(word):
    translations = string.maketrans("", "")
    deletions = string.punctuation + string.whitespace
    return word.translate(translations, deletions).lower()

for line in sys.stdin:
    words = line.strip().split()
    for word in words:
        print "%s\t%d" %(standardize(word), 1)
```

Python reducer

hdstreaming_reducer.py

```
#!/usr/bin/python -O

import sys
import string
sys.path.append(".")

current_word = ""
current_count = 0

for line in sys.stdin:
    try:
        word, count = line.strip().split("\t", 1)
        if word == current_word:
            current_count += 1
        else:
            print "%s\t%s" % (current_word, current_count)
            current_count = 1
            current_word = word
    except ValueError:
        pass
```

MapReduce driver

pymapred.sh

```
#!/bin/bash
HADOOP_STREAMING_JAR_PATH="/opt/cloudera/parcels/CDH/lib/\
hadoop-0.20-mapreduce/contrib/streaming"
PROJECT_HOME="/home/rkekatpure/tutorials/hadooptut"
HDFS_INPUT_PATH="/user/rkekatpure/streaming"
HDFS_OUTPUT_PATH="/user/rkekatpure/streaming/output"

hadoop fs -rm -r "$HDFS_OUTPUT_PATH"

hadoop jar "$HADOOP_STREAMING_JAR_PATH/hadoop-streaming.jar" \
-Dmapreduce.job.queuename=exp_dsa \
-file "$PROJECT_HOME/hdstreaming_mapper.py" \
-file "$PROJECT_HOME/hdstreaming_reducer.py" \
-mapper hdstreaming_mapper.py \
-reducer hdstreaming_reducer.py \
-input "$HDFS_INPUT_PATH/*" \
-output "$HDFS_OUTPUT_PATH"
```

Next steps

- Streaming model
- Streaming with complex and packged python code
- Machine learning models in streaming mode
- Hadoop internals
- DFS interface and implementation
- Hadoop optimizations
- Choice of # mappers and reducers
- Many more...