Session B: Hadoop

Hadoop At Yahoo! (Some Statistics)

- 25,000 + machines in 10+ clusters
- Largest cluster is 3,000 machines
- 3 Petabytes of data (compressed, unreplicated)
- 1000+ users
- 100,000+ jobs/week

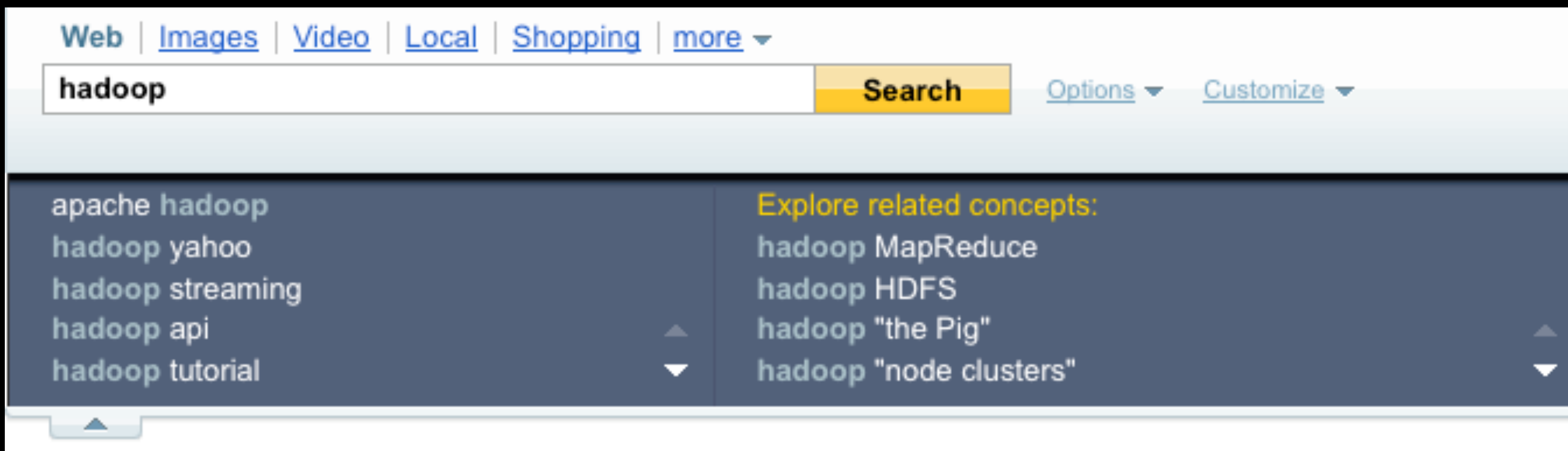
Sample Applications

- Data analysis is the inner loop of Web 2.0
  - Data ⇒ Information ⇒ Value
- Log processing: reporting, buzz
- Search index
- Machine learning: Spam filters
- Competitive intelligence

Prominent Hadoop Users

- Yahoo!
- A9.com
- EHarmony
- Facebook
- Fox Interactive Media
- IBM
- Quantcast
- Joost
- Last.fm
- Powerset
- New York Times
- Rackspace

Yahoo! Search Assist



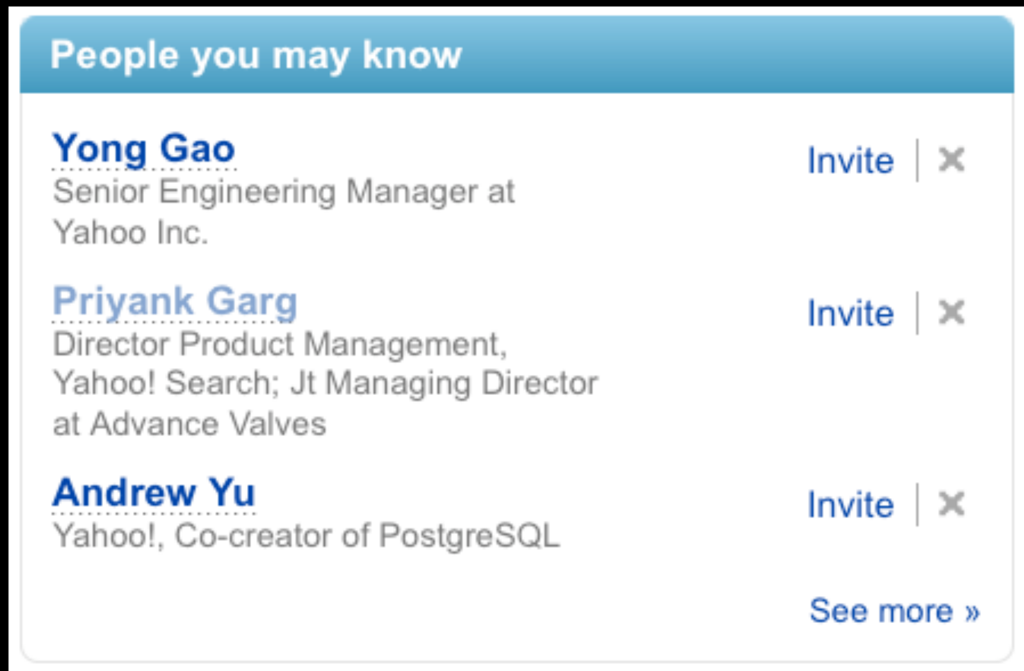
Search Assist

- Insight: Related concepts appear close together in text corpus
- Input: Web pages
  - 1 Billion Pages, 10K bytes each
  - 10 TB of input data
- Output: List(word, List(related words))

Search Assist

```
// Input: List(URL, Text)
foreach URL in Input :
    Words = Tokenize(Text(URL));
    foreach word in Tokens :
        Insert (word, Next(word, Tokens)) in Pairs;
        Insert (word, Previous(word, Tokens)) in Pairs;
// Result: Pairs = List (word, RelatedWord)
Group Pairs by word;
// Result: List (word, List(RelatedWords)
foreach word in Pairs :
    Count RelatedWords in GroupedPairs;
// Result: List (word, List(RelatedWords, count))
foreach word in CountedPairs :
    Sort Pairs(word, *) descending by count;
    choose Top 5 Pairs;
// Result: List (word, Top5(RelatedWords))
```

You Might Also Know



You Might Also Know

- Insight: You might also know Joe Smith if a lot of folks you know, know Joe Smith
  - if you don't know Joe Smith already
- Numbers:
  - 100 MM users
  - Average connections per user is 100

You Might Also Know

```
// Input: List(UserName, List(Connections))

foreach u in UserList : // 100 MM
    foreach x in Connections(u) : // 100
        foreach y in Connections(x) : // 100
            if (y not in Connections(u)) :
                Count(u, y)++; // 3 Trillion Iterations
    Sort (u,y) in descending order of Count(u,y);
    Choose Top 3 y;
    Store (u, {y0, y1, y2}) for serving;
```

Performance

- 101 Random accesses for each user
  - Assume 1 ms per random access
  - 100 ms per user
- 100 MM users
  - 100 days on a single machine

Map & Reduce

- Primitives in Lisp (& Other functional languages) 1970s
- Google Paper 2004
  - http://labs.google.com/papers/mapreduce.html

Map

Output_List = Map (Input_List)

Square (1, 2, 3, 4, 5, 6, 7, 8, 9, 10) =

(1, 4, 9, 16, 25, 36,49, 64, 81, 100)

Reduce

Output_Element = Reduce (Input_List)

Sum (1, 4, 9, 16, 25, 36,49, 64, 81, 100) = 385

Parallelism

- Map is inherently parallel
  - Each list element processed independently
- Reduce is inherently sequential
  - Unless processing multiple lists
- Grouping to produce multiple lists

Search Assist Map

```
// Input: http://hadoop.apache.org

Pairs = Tokenize_And_Pair ( Text ( Input ) )
```

```
Output = {
(apache, hadoop) (hadoop, mapreduce) (hadoop, streaming)
(hadoop, pig) (apache, pig) (hadoop, DFS) (streaming,
commandline) (hadoop, java) (DFS, namenode) (datanode,
block) (replication, default)...
}
```

Search Assist Reduce

```
// Input: GroupedList (word, GroupedList(words))

CountedPairs = CountOccurrences (word, RelatedWords)
```

```
Output = {
(hadoop, apache, 7) (hadoop, DFS, 3) (hadoop, streaming,
4) (hadoop, mapreduce, 9) ...
}
```

Issues with Large Data

- Map Parallelism: Splitting input data
  – Shipping input data
- Reduce Parallelism:
  – Grouping related data
- Dealing with failures
  – Load imbalance

Apache Hadoop

- January 2006: Subproject of Lucene
- January 2008: Top-level Apache project
- Latest Version: 0.20.x
- Stable Version: 0.18.x
- Major contributors: Yahoo!, Facebook, Powerset

Apache Hadoop

- Reliable, Performant Distributed file system
- MapReduce Programming framework
- Sub-Projects: HBase, Hive, Pig, Zookeeper, Chukwa
- Related Projects: Mahout, Hama, Cascading, Scribe, Cassandra, Dumbo, Hypertable, KosmosFS

Problem: Bandwidth to Data

- Scan 100TB Datasets on 1000 node cluster
  - Remote storage @ 10MB/s = 165 mins
  - Local storage @ 50-200MB/s = 33-8 mins
- Moving computation is more efficient than moving data
  - Need visibility into data placement

Problem: Scaling Reliably

- Failure is not an option, it's a rule !
  - 1000 nodes, MTBF < 1 day
  - 4000 disks, 8000 cores, 25 switches, 1000 NICs, 2000 DIMMS (16TB RAM)
- Need fault tolerant store with reasonable availability guarantees
  - Handle hardware faults transparently

Hadoop Goals

- Scalable: Petabytes (10^{15} Bytes) of data on thousands on nodes
- Economical: Commodity components only
- Reliable
  - Engineering reliability into every application is expensive

HDFS

- Data is organized into files and directories
- Files are divided into uniform sized blocks (default 64MB) and distributed across cluster nodes
- HDFS exposes block placement so that computation can be migrated to data

HDFS

- Blocks are replicated (default 3) to handle hardware failure
- Replication for performance and fault tolerance (Rack-Aware placement)
- HDFS keeps checksums of data for corruption detection and recovery

HDFS

- Master-Worker Architecture
- Single NameNode
- Many (Thousands) DataNodes

HDFS Master (NameNode)

- Manages filesystem namespace
- File metadata (i.e. “inode”)
- Mapping inode to list of blocks + locations
- Authorization & Authentication
- Checkpoint & journal namespace changes

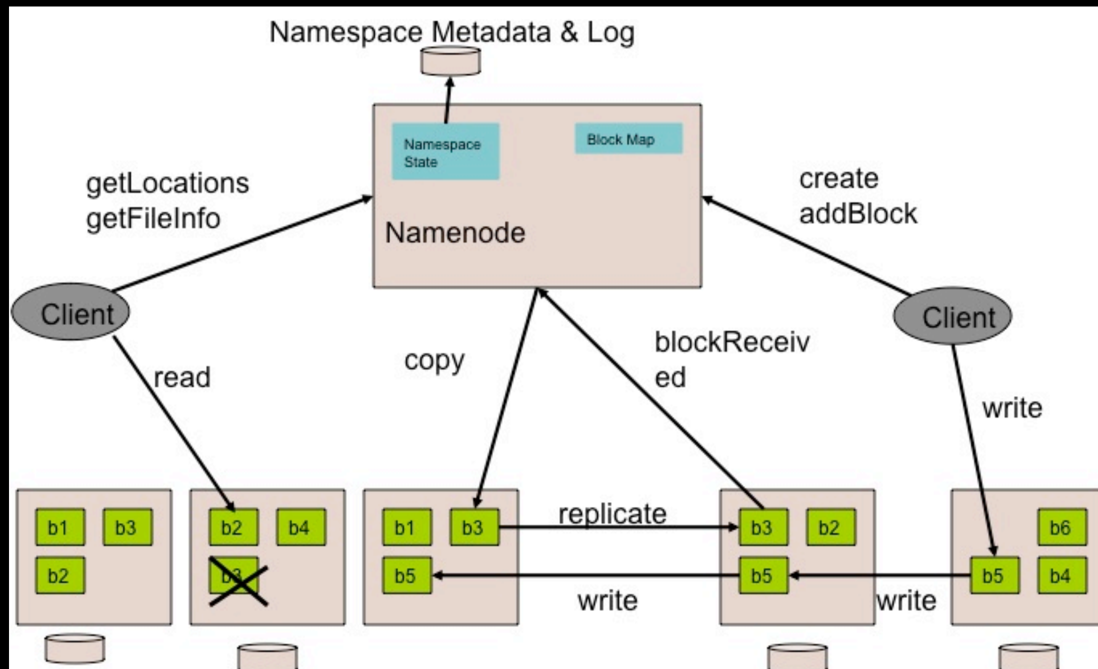
Namenode

- Mapping of datanode to list of blocks
- Monitor datanode health
- Replicate missing blocks
- Keeps ALL namespace in memory
- 60M objects (File/Block) in 16GB

Datanodes

- Handle block storage on multiple volumes & block integrity
- Clients access the blocks directly from data nodes
- Periodically send heartbeats and block reports to Namenode
- Blocks are stored as underlying OS's files

HDFS Architecture



Replication

- A file's replication factor can be changed dynamically (default 3)
- Block placement is rack aware
- Block under-replication & over-replication is detected by Namenode
- Balancer application rebalances blocks to balance datanode utilization

Accessing HDFS

```
hadoop fs [-fs <local | file system URI>] [-conf <configuration file>]
[-D <property=value>] [-ls <path>] [-lsr <path>] [-du <path>]
[-dus <path>] [-mv <src> <dst>] [-cp <src> <dst>] [-rm <src>]
[-rmr <src>] [-put <localsrc> ... <dst>] [-copyFromLocal <localsrc> ... <dst>]
[-moveFromLocal <localsrc> ... <dst>] [-get [-ignoreCrc] [-crc] <src> <localdst>
[-getmerge <src> <localdst> [addnl]] [-cat <src>]
[-copyToLocal [-ignoreCrc] [-crc] <src> <localdst>] [-moveToLocal <src> <localdst>]
[-mkdir <path>] [-report] [-setrep [-R] [-w] <rep> <path/file>]
[-touchz <path>] [-test -[ezd] <path>] [-stat [format] <path>]
[-tail [-f] <path>] [-text <path>]
[-chmod [-R] <MODE[,MODE]... | OCTALMODE> PATH...]
[-chown [-R] [OWNER][:[GROUP]] PATH...]
[-chgrp [-R] GROUP PATH...]
[-count[-q] <path>]
[-help [cmd]]
```

HDFS Java API

```
// Get default file system instance
fs = Filesystem.get(new Configuration());
// Or Get file system instance from URIfs =
Filesystem.get(URI.create(uri),
                    new Configuration());
// Create, open, list, … OutputStream out =
fs.create(path, …);
InputStream in = fs.open(path, …);
boolean isDone = fs.delete(path, recursive);
FileStatus[] fstat = fs.listStatus(path);
```

Hadoop MapReduce

- Record = (Key, Value)
- Key : Comparable, Serializable
- Value: Serializable
- Input, Map, Shuffle, Reduce, Output

Seems Familiar ?

```
cat /var/log/auth.log* | \
grep “session opened” | cut -d’ ‘ -f10 | \
sort | \
uniq -c > \
~/userlist
```

Map

- Input: (Key_1, $Value_1$)
- Output: List(Key_2, $Value_2$)
- Projections, Filtering, Transformation

Shuffle

- Input: List(Key_2, $Value_2$)
- Output
  - Sort(Partition(List(Key_2, List($Value_2$))))
- Provided by Hadoop

Reduce

- Input: List(Key_2, List($Value_2$))
- Output: List(Key_3, $Value_3$)
- Aggregation

Example: Unigrams

- Input: Huge text corpus
  - Wikipedia Articles (40GB uncompressed)
- Output: List of words sorted in descending order of frequency

Unigrams

```
$ cat ~/wikipedia.txt | \
sed -e 's/ /\n/g' | grep . | \
sort | \
uniq -c > \
~/frequencies.txt

$ cat ~/frequencies.txt | \
# cat | \
sort -n -k1,1 -r |
# cat > \
~/unigrams.txt
```

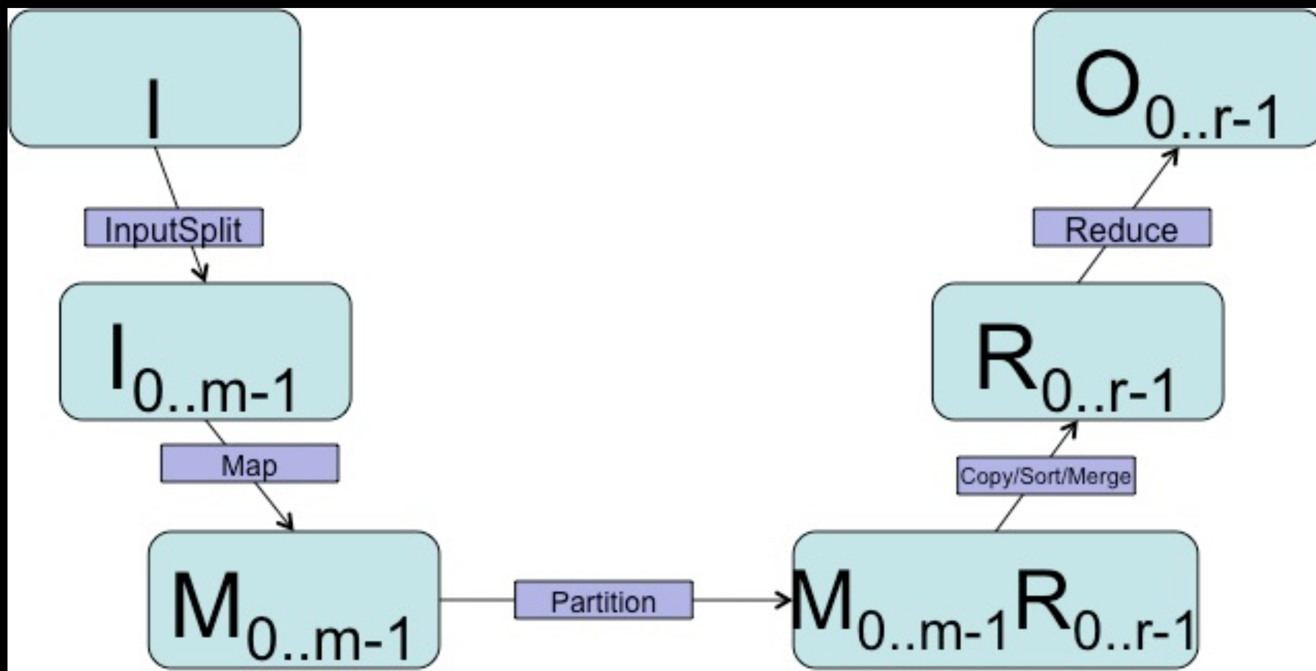
MR for Unigrams

```
mapper (filename, file-contents):
  for each word in file-contents:
    emit (word, 1)

reducer (word, values):
  sum = 0
  for each value in values:
    sum = sum + value
  emit (word, sum)
```

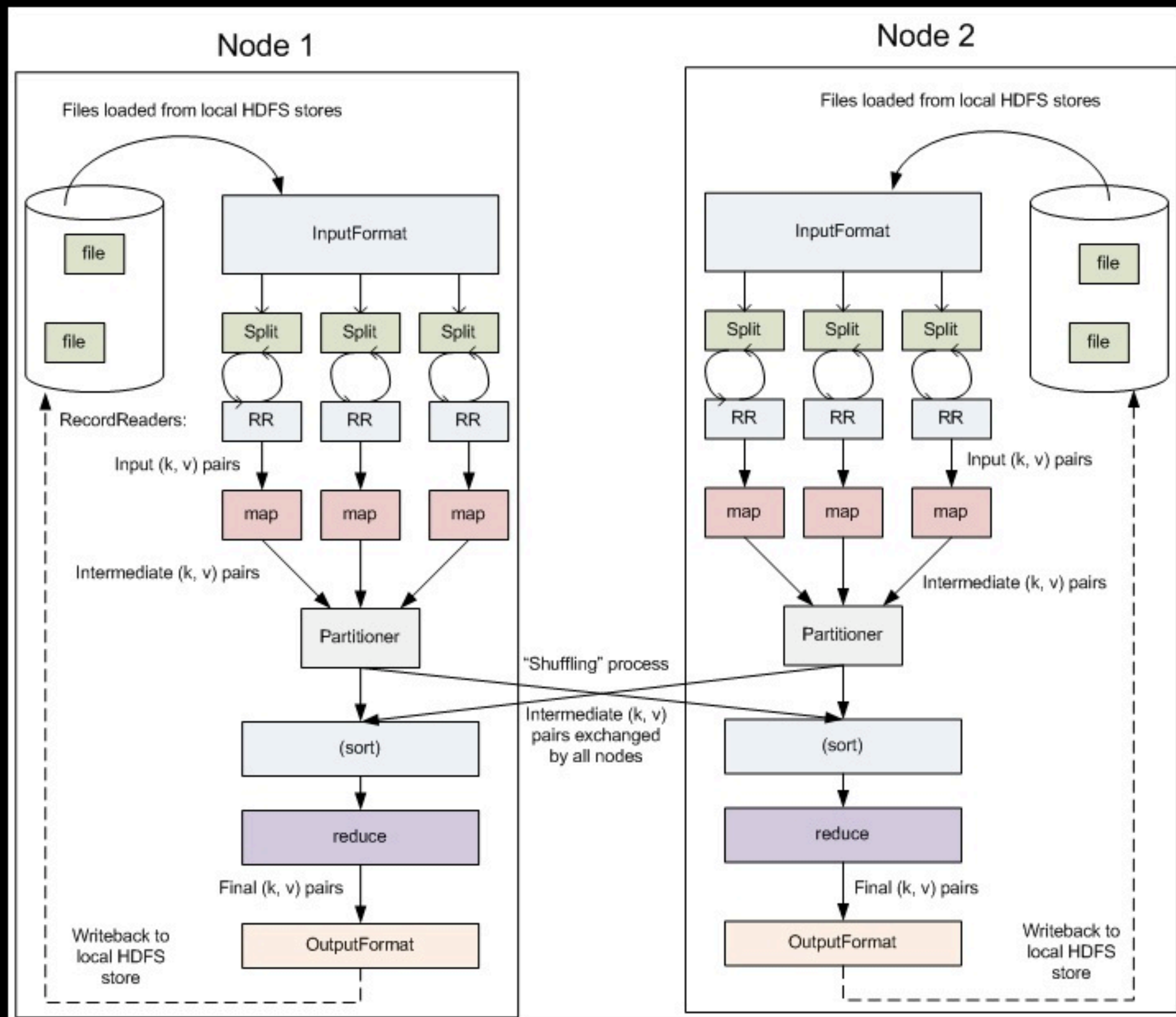
MR for Unigrams

```
mapper (word, frequency):
  emit (frequency, word)

reducer (frequency, words):
  for each word in words:
    emit (word, frequency)
```

MR Dataflow


I
InputSplit
$I_{0..m-1}$
Map
$M_{0..m-1}$
Partition
$M_{0..m-1}R_{0..r-1}$
Copy/Sort/Merge
$R_{0..r-1}$
Reduce
$O_{0..r-1}$

Pipeline Details



Hadoop Streaming

- Hadoop is written in Java
  - Java MapReduce code is “native”
- What about Non-Java Programmers ?
  - Perl, Python, Shell, R
  - grep, sed, awk, uniq as Mappers/Reducers
- Text Input and Output

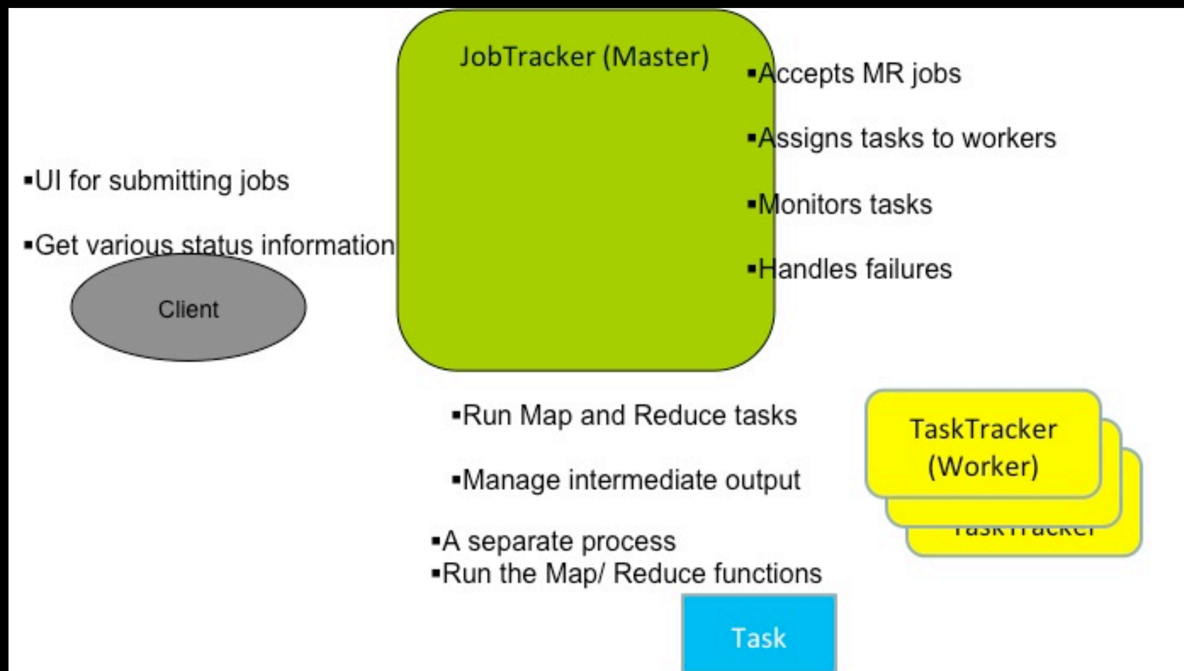
Hadoop Streaming

- Thin Java wrapper for Map & Reduce Tasks
- Forks actual Mapper & Reducer
- IPC via *stdin, stdout, stderr*
- *Key.toString() \t Value.toString() \n*
- Slower than Java programs
  - Allows for quick prototyping / debugging

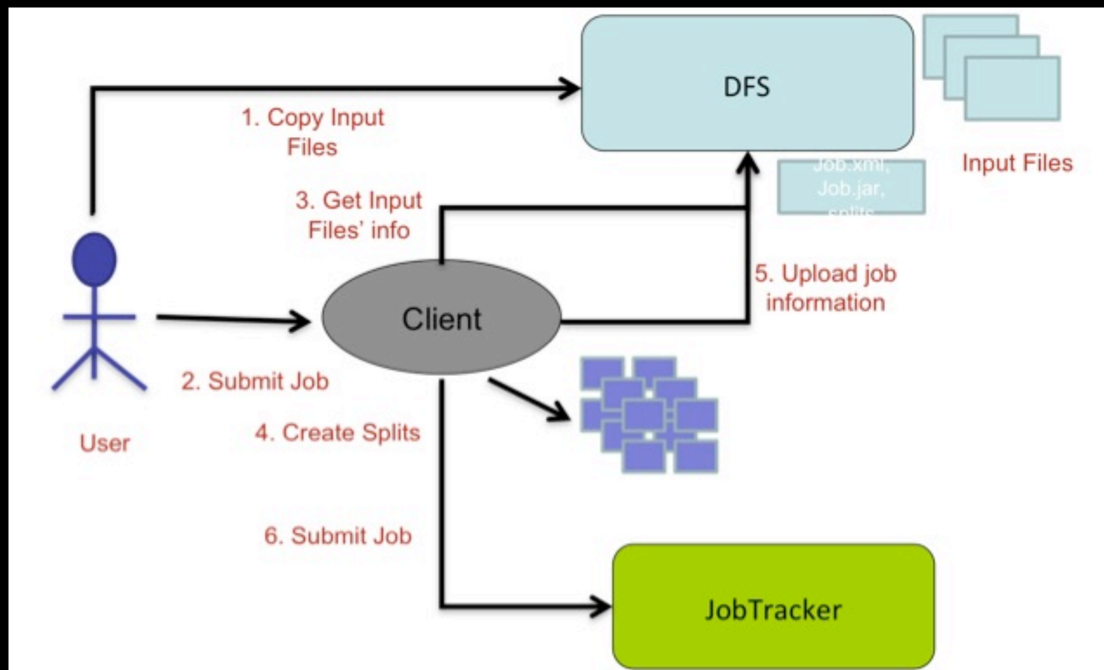
Hadoop Streaming

```
$ bin/hadoop jar hadoop-streaming.jar \
      -input in-files -output out-dir \
      -mapper mapper.sh -reducer reducer.sh

# mapper.sh

sed -e 's/ /\n/g' | grep .

# reducer.sh

uniq -c | awk '{print $2 "\t" $1}'
```

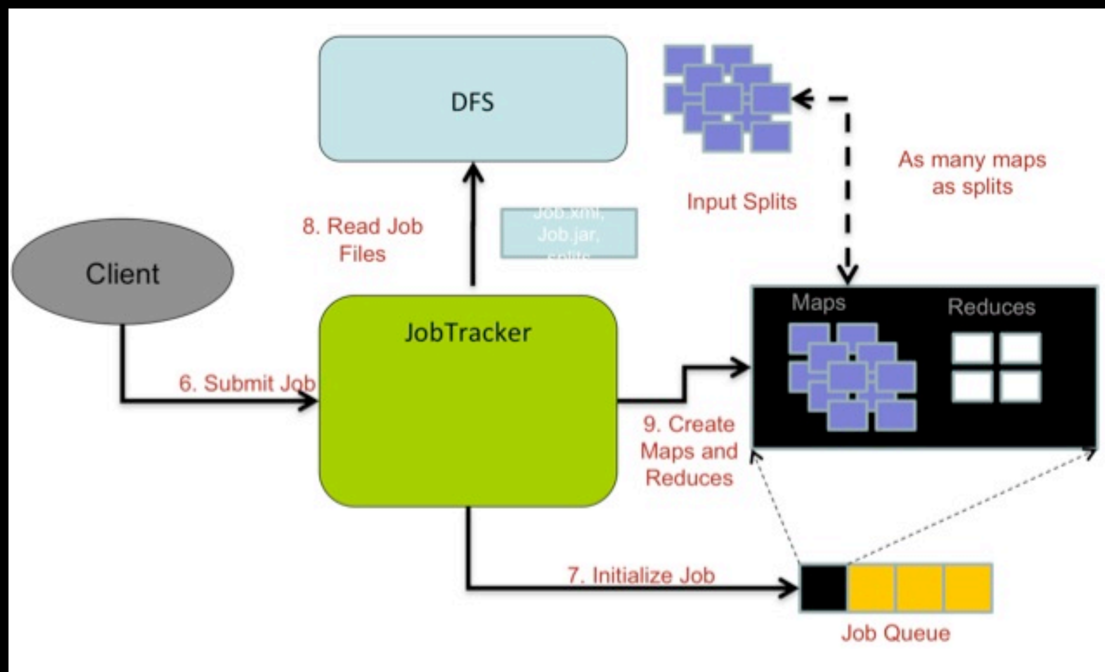
MR Architecture



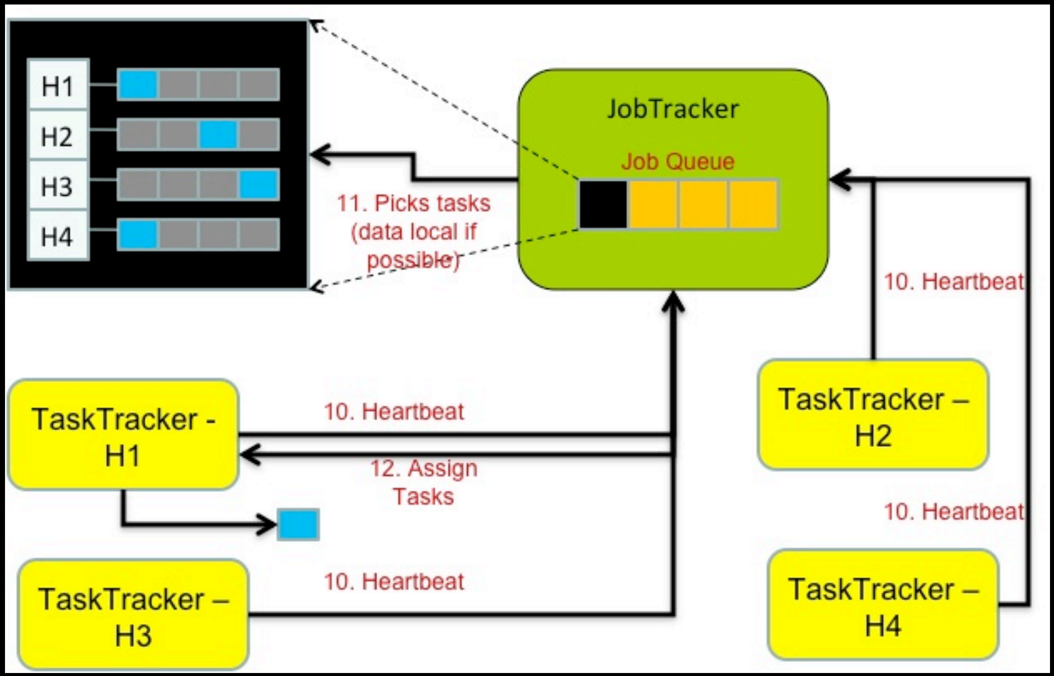
Job Submission



Initialization

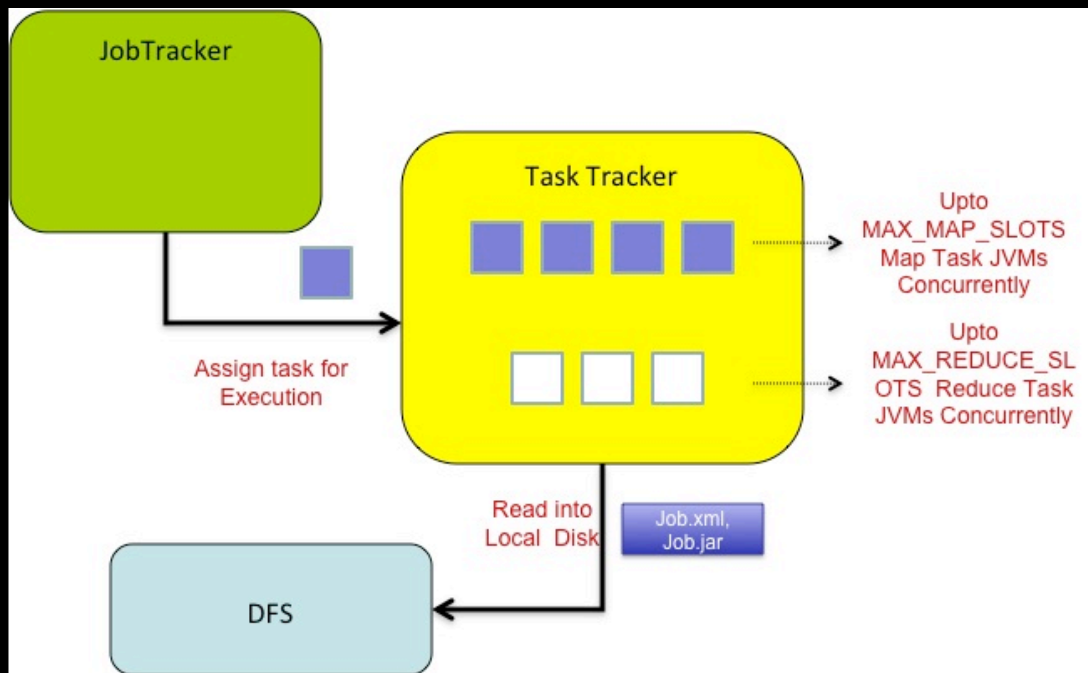


Scheduling

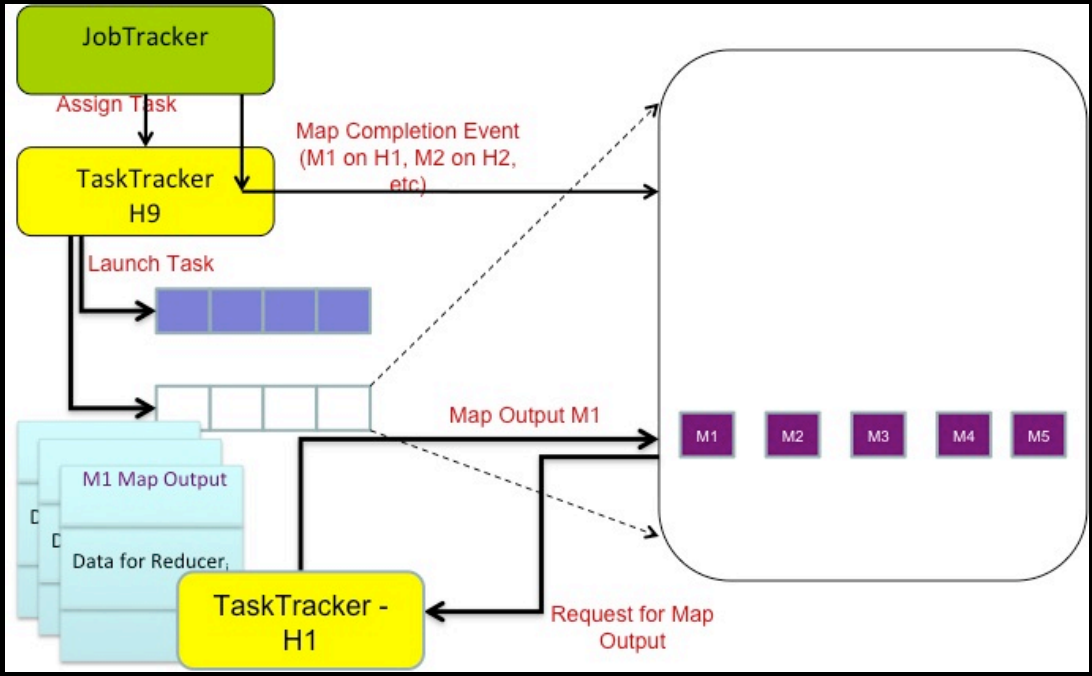


H1
H2
H3
H4
JobTracker
Job Queue
11. Picks tasks (data local if possible)
10. Heartbeat
TaskTracker - H1
10. Heartbeat
12. Assign Tasks
TaskTracker – H2
10. Heartbeat
TaskTracker – H3
10. Heartbeat
TaskTracker – H4

Execution


JobTracker
Task Tracker
Upto MAX_MAP_SLOTS Map Task JVMs Concurrently
Upto MAX_REDUCE_SLOTS Reduce Task JVMs Concurrently
Assign task for Execution
Read into Local Disk
Job.xml, Job.jar
DFS

Reduce Task



JobTracker
Assign Task
TaskTracker
H9
Map Completion Event
(M1 on H1, M2 on H2,
etc)
Launch Task
Map Output M1
M1
M2
M3
M4
M5
M1 Map Output
Data for Reducer
TaskTracker -
H1
Request for Map
Output

Session C: Pig

What is Pig?

- System for processing large semi-structured data sets using Hadoop MapReduce platform
- Pig Latin: High-level procedural language
- Pig Engine: Parser, Optimizer and distributed query execution

Pig vs SQL

- Pig is procedural (How)
- Nested relational data model
- Schema is optional
- Scan-centric analytic workloads
- Limited query optimization

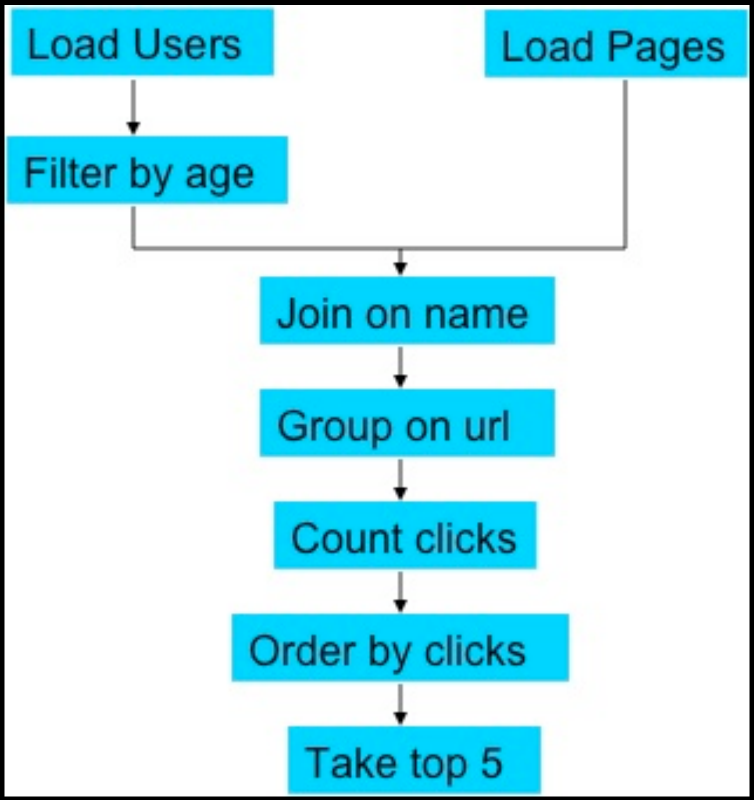
- SQL is declarative
- Flat relational data model
- Schema is required
- OLTP + OLAP workloads
- Significant opportunity for query optimization

Pig vs Hadoop

- Increase programmer productivity
- Decrease duplication of effort
- Insulates against Hadoop complexity
  - Version Upgrades
  - *JobConf* configuration tuning
  - Job Chains

Example

- Input: User profiles, Page visits
- Find the top 5 most visited pages by users aged 18-25

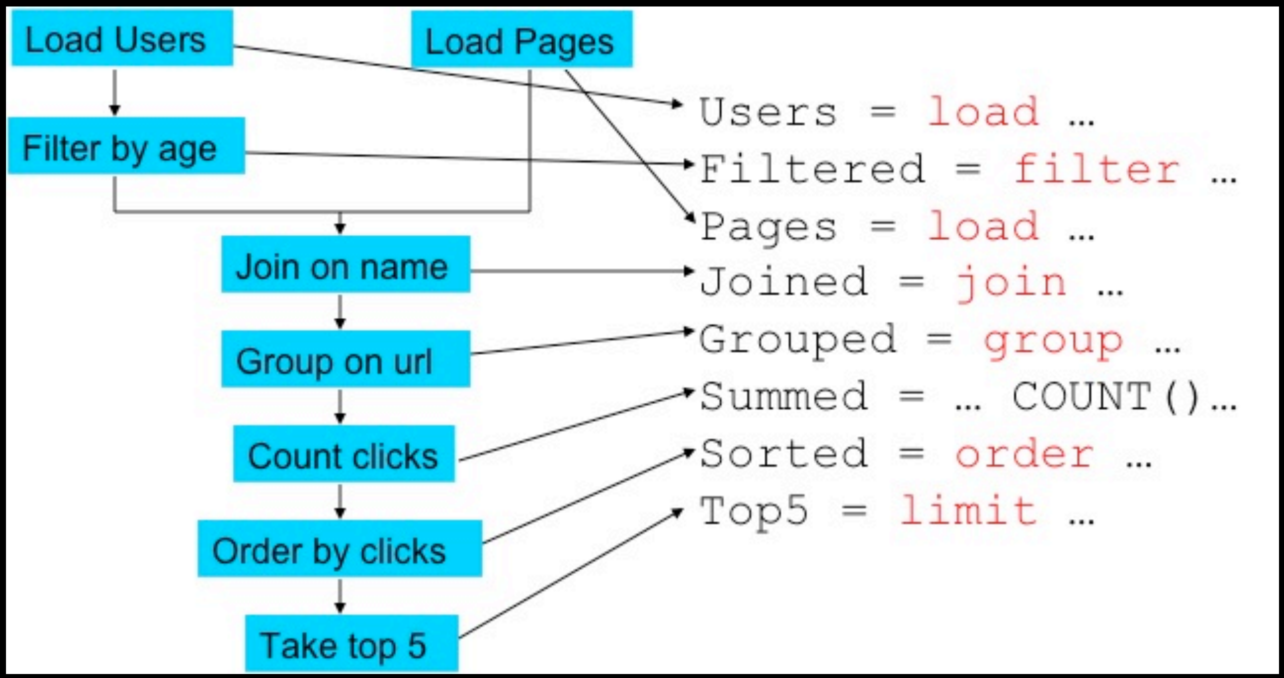


In Native Hadoop

In Pig

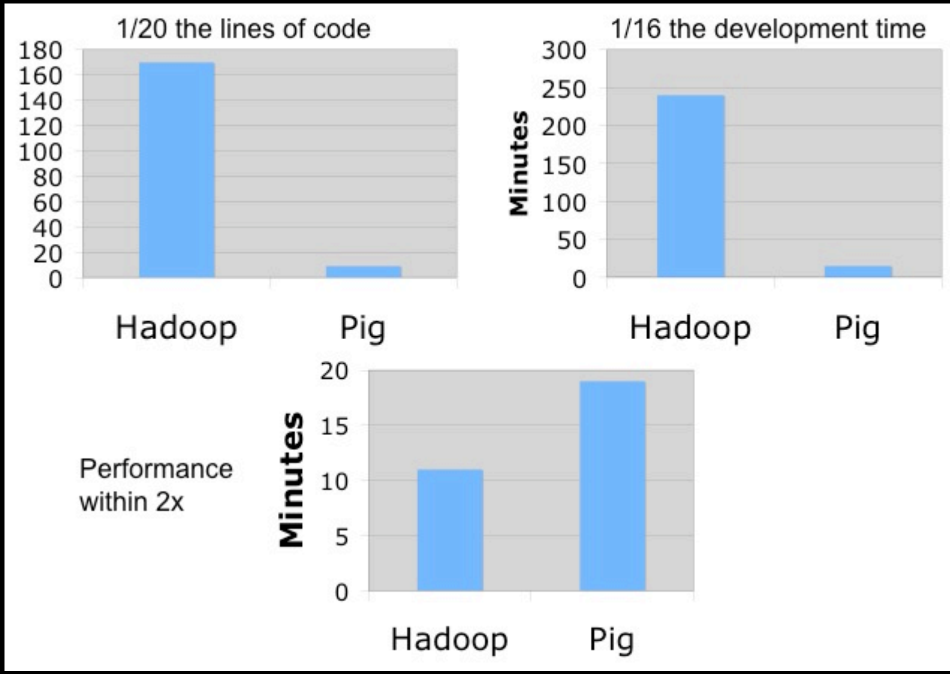
```
Users = load ‘users’ as (name, age);
Filtered = filter Users by age >= 18 and age <= 25;
Pages = load ‘pages’ as (user, url);
Joined = join Filtered by name, Pages by user;
Grouped = group Joined by url;
Summed = foreach Grouped generate group,

           COUNT(Joined) as clicks;
Sorted = order Summed by clicks desc;
Top5 = limit Sorted 5;
store Top5 into ‘top5sites’;
```

Natural Fit



Load Users
Load Pages
Filter by age
Join on name
Group on url
Count clicks
Order by clicks
Take top 5
Users = load …
Filtered = filter …
Pages = load …
Joined = join …
Grouped = group …
Summed = … COUNT()…
Sorted = order …
Top5 = limit …

Comparison



1/20 the lines of code
180
160
140
120
100
80
60
40
20
0
Hadoop
Pig
1/16 the development time
Minutes
300
250
200
150
100
50
0
Hadoop
Pig
Performance within 2x
Minutes
20
15
10
5
0
Hadoop
Pig

Flexibility & Control

- Easy to plug-in user code
- Metadata is not mandatory
- Pig does not impose a data model on you
- Fine grained control
- Complex data types

Pig Data Types

- Tuple: Ordered set of fields
  - Field can be simple or complex type
  - Nested relational model
- Bag: Collection of tuples
  - Can contain duplicates
- Map: Set of (key, value) pairs

Simple data types

- *int* : 42
- *long* : 42L
- *float* : 3.1415f
- *double* : 2.7182818
- *chararray* : UTF-8 String
- *bytearray* : blob

NULL

- Same as SQL: unknown or non-existent
- Loader inserts NULL for empty data
- Operations can produce NULL
  - divide by 0
  - dereferencing a non-existent map key

Expressions

```
A = LOAD ‘data.txt AS
  (f1:int , f2:{t:(n1:int, n2:int)}, f3: map[] )
```

```
A =
{
    (1,                        -- A.f1 or A.$0
    { (2, 3), (4, 6) },        -- A.f2 or A.$1
    [ ‘yahoo’#’mail’ ]         -- A.f3 or A.$2
}
```

Counting Word Frequencies

- Input: Large text document
- Process:
  - Load the file
  - For each line, generate word tokens
  - Group by word
  - Count words in each group

Load

```
myinput = load '/user/milindb/text.txt'
     USING TextLoader() as (myword:chararray);
```

```
(program program)
(pig pig)
(program pig)
(hadoop pig)
(latin latin)
(pig latin)
```

Tokenize

```
words = FOREACH myinput GENERATE FLATTEN(TOKENIZE(*));
```

```
(program) (program) (pig) (pig) (program) (pig) (hadoop)
(pig) (latin) (latin) (pig) (latin)
```

Group

```
grouped = GROUP words BY $0;
```

```
(pig, {(pig), (pig), (pig), (pig), (pig)})
(latin, {(latin), (latin), (latin)})
(hadoop, {(hadoop)})
(program, {(program), (program), (program)})
```

Count

```
counts = FOREACH grouped GENERATE group, COUNT(words);
```

```
(pig, 5L)
(latin, 3L)
(hadoop, 1L)
(program, 3L)
```

Store

```
store counts into '/user/milindb/output'
      using PigStorage();
```

```
pig     5
latin   3
hadoop  1
program 3
```

Example: Log Processing

```
-- use a custom loader
Logs = load ‘apachelogfile’ using
    CommonLogLoader() as (addr, logname,
    user, time, method, uri, p, bytes);
-- apply your own function
Cleaned = foreach Logs generate addr,
    canonicalize(url) as url;
Grouped = group Cleaned by url;
-- run the result through a binary
Analyzed = stream Grouped through
    ‘urlanalyzer.py’;
store Analyzed into ‘analyzedurls’;
```

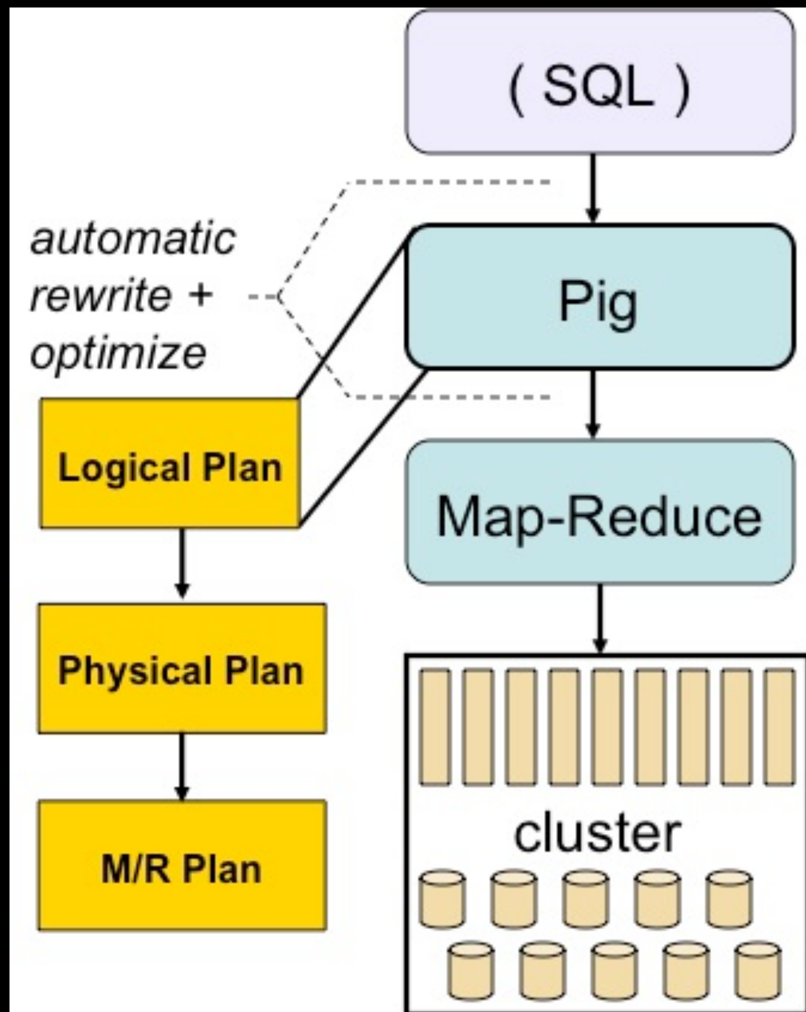
Schema on the fly

```
-- declare your types
Grades = load ‘studentgrades’ as
    (name: chararray, age: int,
    gpa: double);
Good = filter Grades by age > 18
    and gpa > 3.0;
-- ordering will be by type
Sorted = order Good by gpa;
store Sorted into ‘smartgrownups’;
```

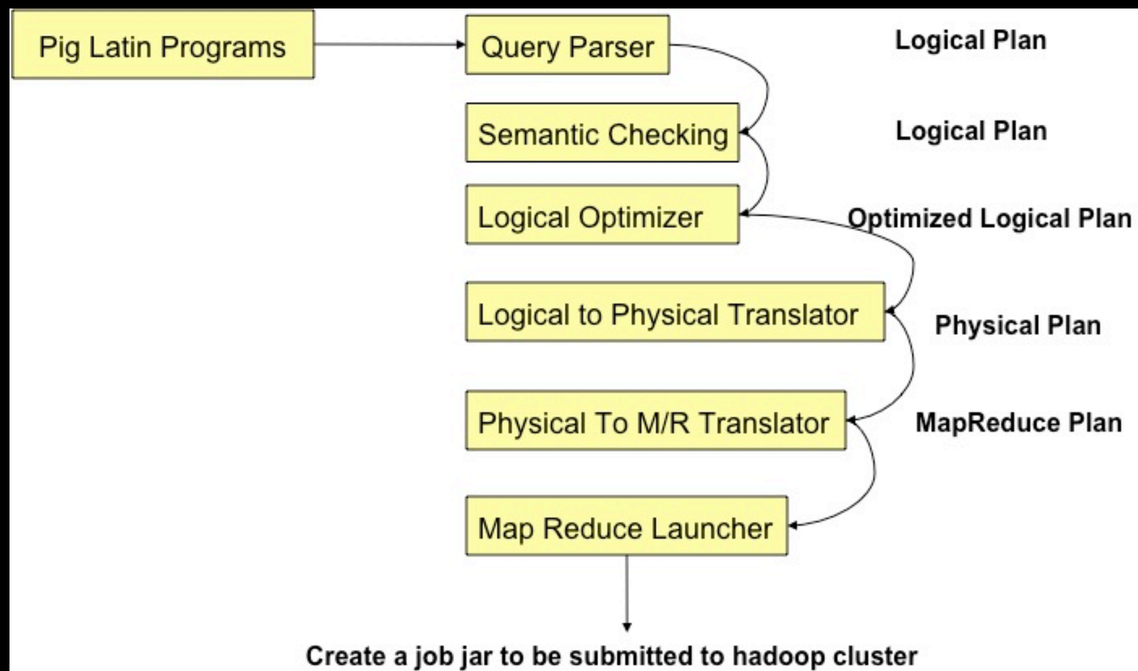
Nested Data

```
Logs = load 'weblogs' as (url, userid);
Grouped = group Logs by url;
-- Code inside {} will be applied to each
-- value in turn.
DisinctCount = foreach Grouped {
    Userid = Logs.userid;
    DistinctUsers = distinct Userid;
    generate group, COUNT(DistinctUsers);
}
store DistinctCount into 'distinctcount';
```

Pig Architecture



Pig Frontend



Logical Plan

- Directed Acyclic Graph
  - Logical Operator as Node
  - Data flow as edges
- Logical Operators
  - One per Pig statement
  - Type checking with Schema

Pig Statements

Load	Read data from the file system
Store	Write data to the file system
Dump	Write data to *stdout*

Pig Statements

Foreach..Generate	Apply expression to each record and generate one or more records
Filter	Apply predicate to each record and remove records where false
Stream..through	Stream records through user-provided binary

Pig Statements

Group/CoGroup	Collect records with the same key from one or more inputs
Join	Join two or more inputs based on a key
Order..by	Sort records based on a key

Physical Plan

- Pig supports two back-ends
  - Local
  - Hadoop MapReduce
- 1:1 correspondence with most logical operators
  - Except Distinct, Group, Cogroup, Join etc

MapReduce Plan

- Detect Map-Reduce boundaries
  - Group, Cogroup, Order, Distinct
- Coalesce operators into Map and Reduce stages
- *Job.jar* is created and submitted to Hadoop *JobControl*

Lazy Execution

- Nothing really executes until you request output
- Store, Dump, Explain, Describe, Illustrate
- Advantages
  - In-memory pipelining
  - Filter re-ordering across multiple commands

Parallelism

- Split-wise parallelism on Map-side operators
- By default, 1 reducer
- PARALLEL keyword
  - group, cogroup, cross, join, distinct, order

Running Pig

```
$ pig
grunt > A = load ‘students’ as (name, age, gpa);
grunt > B = filter A by gpa > ‘3.5’;
grunt > store B into ‘good_students’;
grunt > dump A;
(jessica thompson, 73, 1.63)
(victor zipper, 23, 2.43)
(rachel hernandez, 40, 3.60)
grunt > describe A;
A: (name, age, gpa )
```

Running Pig

- Batch mode
  - $ pig myscript.pig
- Local mode
  - $ pig –x local
- Java mode (embed pig statements in java)
  - Keep pig.jar in the class path

SQL to Pig

SQL	Pig
`...FROM MyTable...`	`A = LOAD ‘MyTable’ USING PigStorage(‘\t’) AS (col1:int, col2:int, col3:int);`
`SELECT col1 + col2, col3 ...`	`B = FOREACH A GENERATE col1 + col2, col3;`
`...WHERE col2 > 2`	`C = FILTER B by col2 > 2;`

SQL to Pig

SQL	Pig

`SELECT col1, col2, sum(col3)`
`FROM X GROUP BY col1, col2`
	`D = GROUP A BY (col1, col2)`
	`E = FOREACH D GENERATE`
	`FLATTEN(group), SUM(A.col3);`

`...HAVING sum(col3) > 5`	`F = FILTER E BY $2 > 5;`
`...ORDER BY col1`	`G = ORDER F BY $0;`

SQL to Pig

SQL	Pig
`SELECT DISTINCT col1` `from X`	`I = FOREACH A GENERATE col1;` `J = DISTINCT I;`

`SELECT col1,`
`count(DISTINCT col2)`
`FROM X GROUP BY col1`
	`K = GROUP A BY col1;`
	`L = FOREACH K {`
	`      M = DISTINCT A.col2;`
	`      GENERATE FLATTEN(group), count(M);`
	`    }`

SQL to Pig

SQL	Pig

`SELECT A.col1, B.col3 FROM A JOIN B USING (col1)`	`N = JOIN A by col1 INNER, B by col1 INNER;`
	`O = FOREACH N GENERATE A.col1, B.col3;`
	`-- Or`
	`N = COGROUP A by col1 INNER, B by col1 INNER;`
	`O = FOREACH N GENERATE flatten(A), flatten(B);`
	`P = FOREACH O GENERATE A.col1, B.col3`