Big Data Infrastructure

Session 9: Beyond MapReduce — Dataflow Languages

Jimmy Lin
University of Maryland
Monday, April 6, 2015

Today's Agenda

- What's beyond MapReduce?
  - SQL on Hadoop
  - Dataflow languages
- Past and present developments

A Major Step Backwards?

- MapReduce is a step backward in database access:
  - Schemas are good ✔
  - Separation of the schema from the application is good ✔
  - High-level access languages are good ?
- MapReduce is poor implementation
  - Brute force and only brute force (no indexes, for example) ✔
- MapReduce is not novel
- MapReduce is missing features
  - Bulk loader, indexing, updates, transactions…
- MapReduce is incompatible with DMBS tools ?

Source: Blog post by DeWitt and Stonebraker

Need for High-Level Languages

- Hadoop is great for large-data processing!
  - But writing Java programs for everything is verbose and slow
  - Data scientists don't want to write Java
- Solution: develop higher-level data processing languages
  - Hive: HQL is like SQL
  - Pig: Pig Latin is a bit like Perl

Hive and Pig

- Hive: data warehousing application in Hadoop
  - Query language is HQL, variant of SQL
  - Tables stored on HDFS with different encodings
  - Developed by Facebook, now open source
- Pig: large-scale data processing system
  - Scripts are written in Pig Latin, a dataflow language
  - Programmer focuses on data transformations
  - Developed by Yahoo!, now open source
- Common idea:
  - Provide higher-level language to facilitate large-data processing
  - Higher-level language "compiles down" to Hadoop jobs

facebook®

Jeff Hammerbacher, Information Platforms and the Rise of the Data Scientist.
In, *Beautiful Data*, O'Reilly, 2009.

"On the first day of logging the Facebook clickstream, more than 400 gigabytes of data was collected. The load, index, and aggregation processes for this data set really taxed the Oracle data warehouse. Even after significant tuning, we were unable to aggregate a day of clickstream data in less than 24 hours."

Hive: Example

- Hive looks similar to an SQL database
- Relational join on two tables:
  - Table of word counts from Shakespeare collection
  - Table of word counts from the bible

```
SELECT s.word, s.freq, k.freq FROM shakespeare s
  JOIN bible k ON (s.word = k.word) WHERE s.freq >= 1 AND k.freq >= 1
  ORDER BY s.freq DESC LIMIT 10;
```

the	25848	62394
I	23031	8854
and	19671	38985
to	18038	13526
of	16700	34654
a	14170	8057
you	12702	2720
my	11297	4135
in	10797	12445
is	8882	6884

Source: Material drawn from Cloudera training VM

Hive: Behind the Scenes

```
SELECT s.word, s.freq, k.freq FROM shakespeare s
  JOIN bible k ON (s.word = k.word) WHERE s.freq >= 1 AND k.freq >= 1
  ORDER BY s.freq DESC LIMIT 10;
```

(Abstract Syntax Tree)

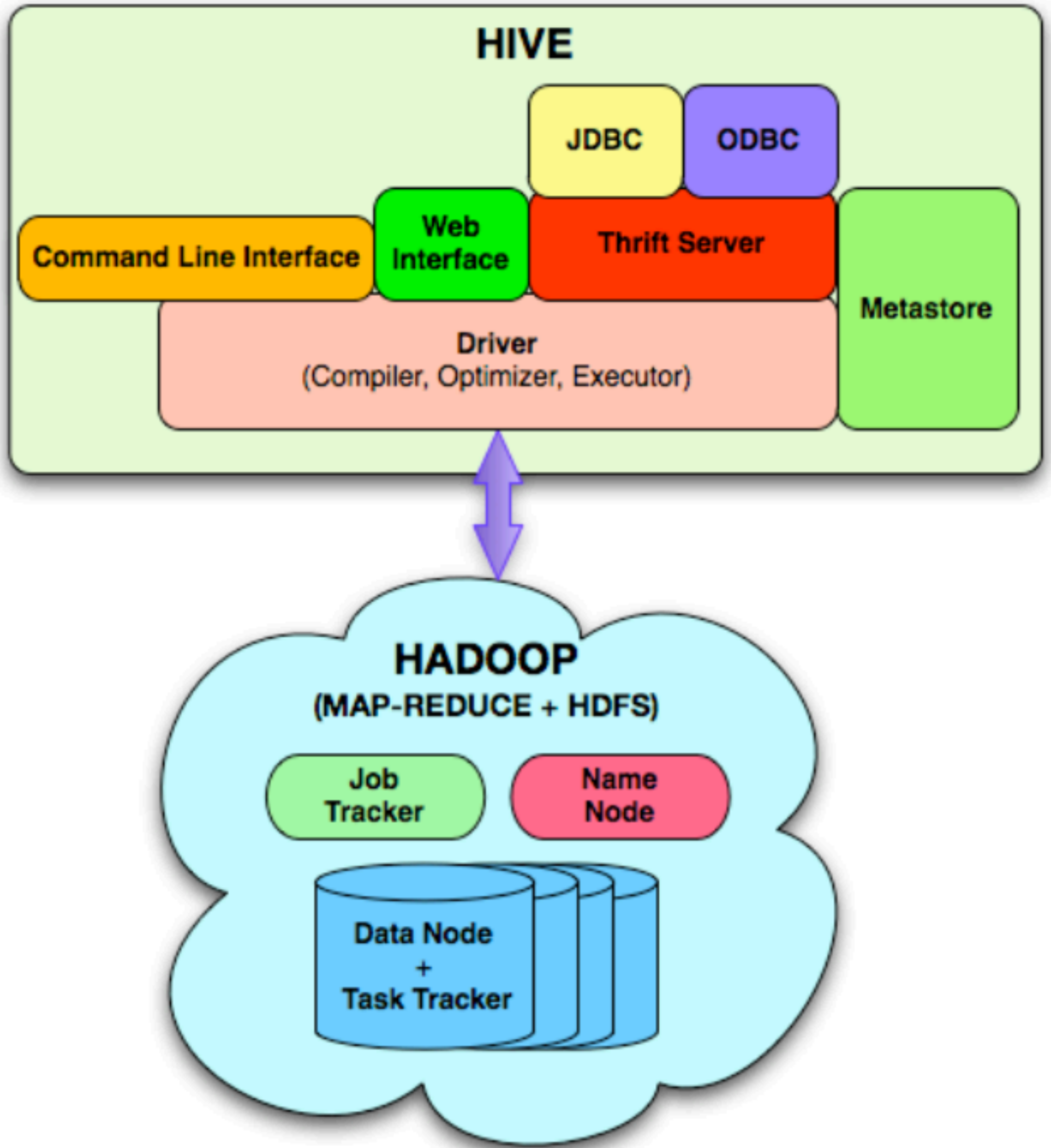
(TOK_QUERY (TOK_FROM (TOK_JOIN (TOK_TABREF shakespeare s) (TOK_TABREF bible k) (= (. (TOK_TABLE_OR_COL s) word) (. (TOK_TABLE_OR_COL k) word)))) (TOK_INSERT (TOK_DESTINATION (TOK_DIR TOK_TMP_FILE)) (TOK_SELECT (TOK_SELEXPR (. (TOK_TABLE_OR_COL s) word)) (TOK_SELEXPR (. (TOK_TABLE_OR_COL s) freq)) (TOK_SELEXPR (. (TOK_TABLE_OR_COL k) freq))) (TOK_WHERE (AND (>= (. (TOK_TABLE_OR_COL s) freq) 1) (>= (. (TOK_TABLE_OR_COL k) freq) 1))) (TOK_ORDERBY (TOK_TABSORTCOLNAMEDESC (. (TOK_TABLE_OR_COL s) freq))) (TOK_LIMIT 10)))

(one or more of MapReduce jobs)

Hive: Behind the Scenes

```
STAGE DEPENDENCIES:
  Stage-1 is a root stage
  Stage-2 depends on stages: Stage-1
  Stage-0 is a root stage

STAGE PLANS:
  Stage: Stage-1
    Map Reduce
      Alias -> Map Operator Tree:
        s
          TableScan
            alias: s
            Filter Operator
              predicate:
                  expr: (freq >= 1)
                  type: boolean
              Reduce Output Operator
                key expressions:
                      expr: word
                      type: string
                sort order: +
                Map-reduce partition columns:
                      expr: word
                      type: string
                tag: 0
                value expressions:
                      expr: freq
                      type: int
                      expr: word
                      type: string
        k
          TableScan
            alias: k
            Filter Operator
              predicate:
                  expr: (freq >= 1)
                  type: boolean
              Reduce Output Operator
                key expressions:
                      expr: word
                      type: string
                sort order: +
                Map-reduce partition columns:
                      expr: word
                      type: string
                tag: 1
                value expressions:
                      expr: freq
                      type: int
      Reduce Operator Tree:
        Join Operator
          condition map:
               Inner Join 0 to 1
          condition expressions:
            0 {VALUE._col0} {VALUE._col1}
            1 {VALUE._col0}
          outputColumnNames: _col0, _col1, _col2
          Filter Operator
            predicate:
                expr: ((_col0 >= 1) and (_col2 >= 1))
                type: boolean
            Select Operator
              expressions:
                    expr: _col1
                    type: string
                    expr: _col0
                    type: int
                    expr: _col2
                    type: int
              outputColumnNames: _col0, _col1, _col2
              File Output Operator
                compressed: false
                GlobalTableId: 0
                table:
                    input format: org.apache.hadoop.mapred.SequenceFileInputFormat
                    output format: org.apache.hadoop.hive.ql.io.HiveSequenceFileOutputFormat

  Stage: Stage-2
    Map Reduce
      Alias -> Map Operator Tree:
        hdfs://localhost:8022/tmp/hive-training/364214370/10002
            Reduce Output Operator
              key expressions:
                    expr: _col1
                    type: int
              sort order: -
              tag: -1
              value expressions:
                    expr: _col0
                    type: string
                    expr: _col1
                    type: int
                    expr: _col2
                    type: int
      Reduce Operator Tree:
        Extract
          Limit
            File Output Operator
              compressed: false
              GlobalTableId: 0
              table:
                  input format: org.apache.hadoop.mapred.TextInputFormat
                  output format: org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat

  Stage: Stage-0
    Fetch Operator
      limit: 10
```

Hive Architecture



Hive Implementation

- Metastore holds metadata
  - Databases, tables
  - Schemas (field names, field types, etc.)
  - Permission information (roles and users)
- Hive data stored in HDFS
  - Tables in directories
  - Partitions of tables in sub-directories
  - Actual data in files


Pig!
Source: Wikipedia (Pig)

Pig: Example

Task: Find the top 10 most visited pages in each category

Visits

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

⋮

Url Info

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

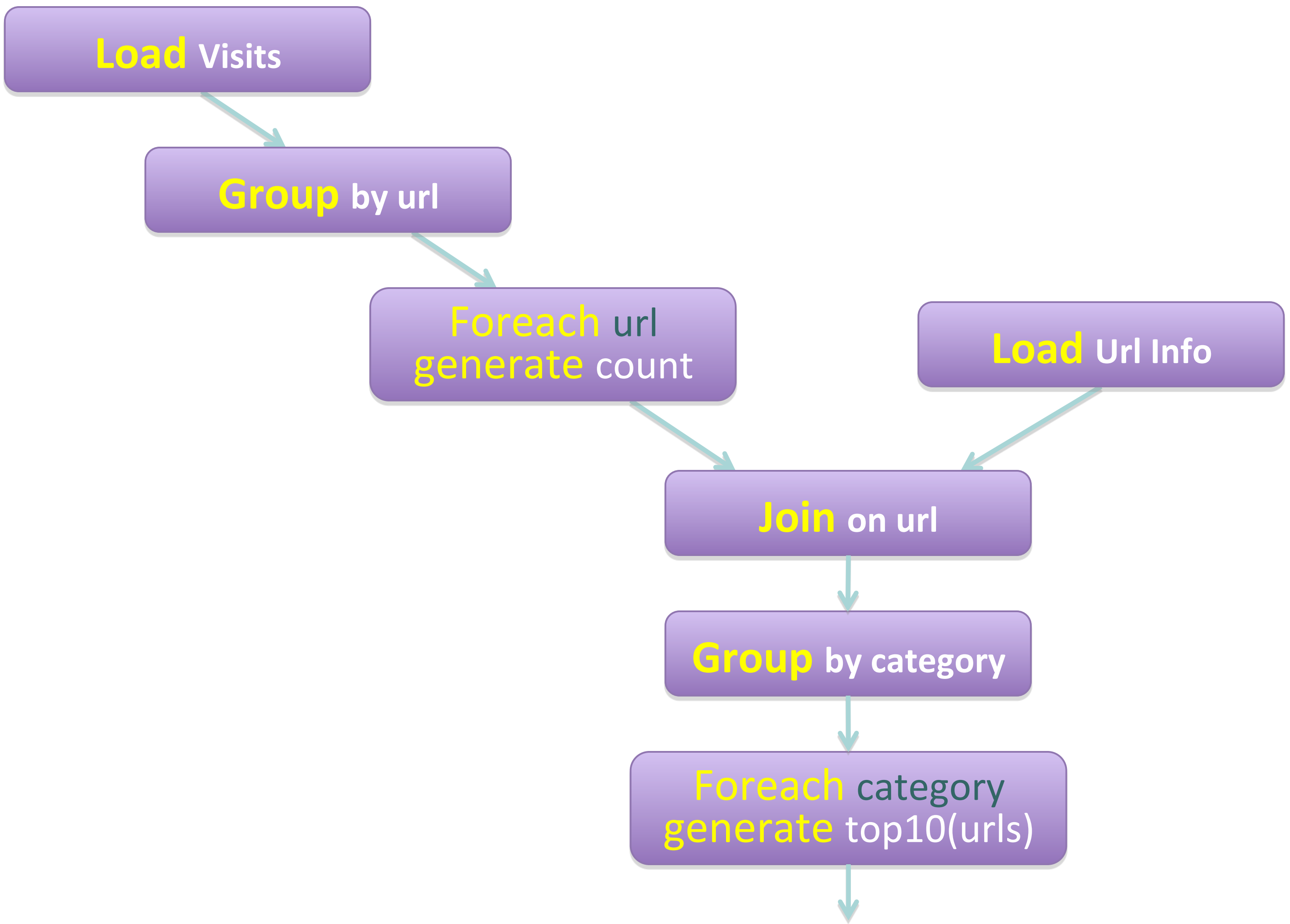
⋮

Pig Slides adapted from Olston et al. (SIGMOD 2008)

Pig Script

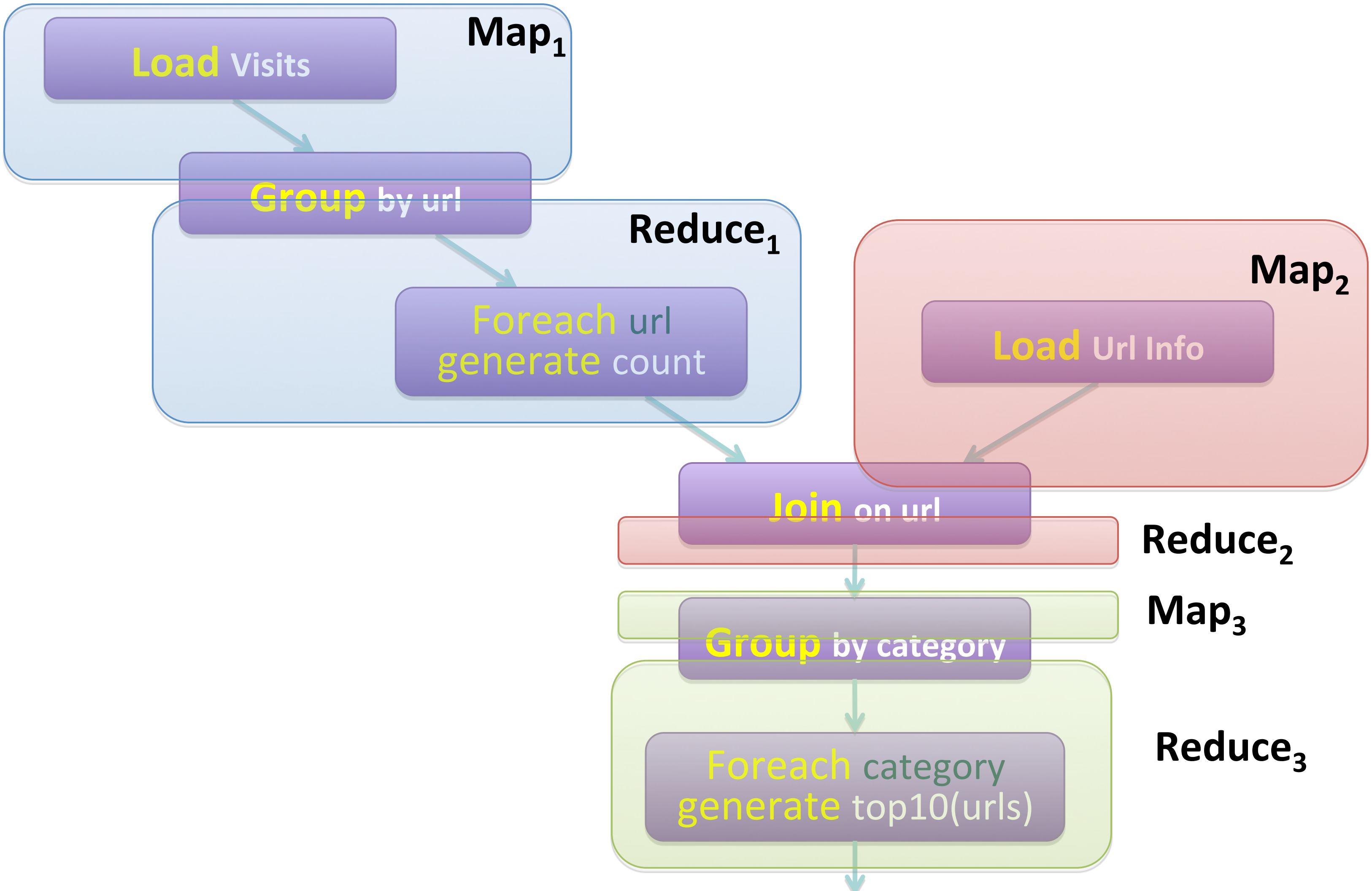
```
visits = load '/data/visits' as (user, url, time);
gVisits = group visits by url;
visitCounts = foreach gVisits generate url, count(visits);
urlInfo = load '/data/urlInfo' as (url, category, pRank);
visitCounts = join visitCounts by url, urlInfo by url;
gCategories = group visitCounts by category;
topUrls = foreach gCategories generate top(visitCounts,10);

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

Pig Query Plan



Pig Slides adapted from Olston et al. (SIGMOD 2008)

Pig Script in Hadoop



Pig Slides adapted from Olston et al. (SIGMOD 2008)

Pig: Basics

- Sequence of statements manipulating relations (aliases)
- Data model
  - atoms
  - tuples
  - bags
  - maps
  - json

Pig: Common Operations

- LOAD: load data
- FOREACH … GENERATE: per tuple processing
- FILTER: discard unwanted tuples
- GROUP/COGROUP: group tuples
- JOIN: relational join

Pig: GROUPing

```
A = LOAD 'myfile.txt’ AS (f1: int, f2: int, f3: int);

    (1, 2, 3)
    (4, 2, 1)
    (8, 3, 4)
    (4, 3, 3)
    (7, 2, 5)
    (8, 4, 3)
```

```
X = GROUP A BY f1;

    (1, {(1, 2, 3)})
    (4, {(4, 2, 1), (4, 3, 3)})
    (7, {(7, 2, 5)})
    (8, {(8, 3, 4), (8, 4, 3)})
```

Pig: COGROUPing

A:

```
(1, 2, 3)
(4, 2, 1)
(8, 3, 4)
(4, 3, 3)
(7, 2, 5)
(8, 4, 3)
```

B:

```
(2, 4)
(8, 9)
(1, 3)
(2, 7)
(2, 9)
(4, 6)
(4, 9)
```

```
X = COGROUP A BY f1, B BY $0;
```

```
(1, {(1, 2, 3)}, {(1, 3)})
(2, {}, {(2, 4), (2, 7), (2, 9)})
(4, {(4, 2, 1), (4, 3, 3)}, {(4, 6),(4, 9)})
(7, {(7, 2, 5)}, {})
(8, {(8, 3, 4), (8, 4, 3)}, {(8, 9)})
```

Pig UDFs

- User-defined functions:
  - Java
  - Python
  - JavaScript
  - Ruby
- UDFs make Pig arbitrarily extensible
  - Express “core” computations in UDFs
  - Take advantage of Pig as glue code for scale-out plumbing

PageRank in Pig

```
previous_pagerank = LOAD '$docs_in' USING PigStorage()
  AS (url: chararray, pagerank: float,
      links:{link: (url: chararray)});

outbound_pagerank = FOREACH previous_pagerank
  GENERATE pagerank / COUNT(links) AS pagerank,
  FLATTEN(links) AS to_url;

new_pagerank =
    FOREACH ( COGROUP outbound_pagerank
    BY to_url, previous_pagerank BY url INNER )
    GENERATE group AS url,
       (1 - $d) + $d * SUM(outbound_pagerank.pagerank) AS pagerank,
       FLATTEN(previous_pagerank.links) AS links;

STORE new_pagerank INTO '$docs_out' USING PigStorage();
```

From: http://techblug.wordpress.com/2011/07/29/pagerank-implementation-in-pig/

Oh, the iterative part too...

```
#!/usr/bin/python
from org.apache.pig.scripting import *
P = Pig.compile(""" Pig part goes here """)

params = { ‘d’: ‘0.5’, ‘docs_in’: ‘data/
pagerank_data_simple’ }

for i in range(10):
   out = "out/pagerank_data_" + str(i + 1)
   params["docs_out"] = out
   Pig.fs("rmr " + out)
   stats = P.bind(params).runSingle()
   if not stats.isSuccessful():
      raise ‘failed’
   params["docs_in"] = out
```

From: http://techblug.wordpress.com/2011/07/29/pagerank-implementation-in-pig/

What's next?

Imapala

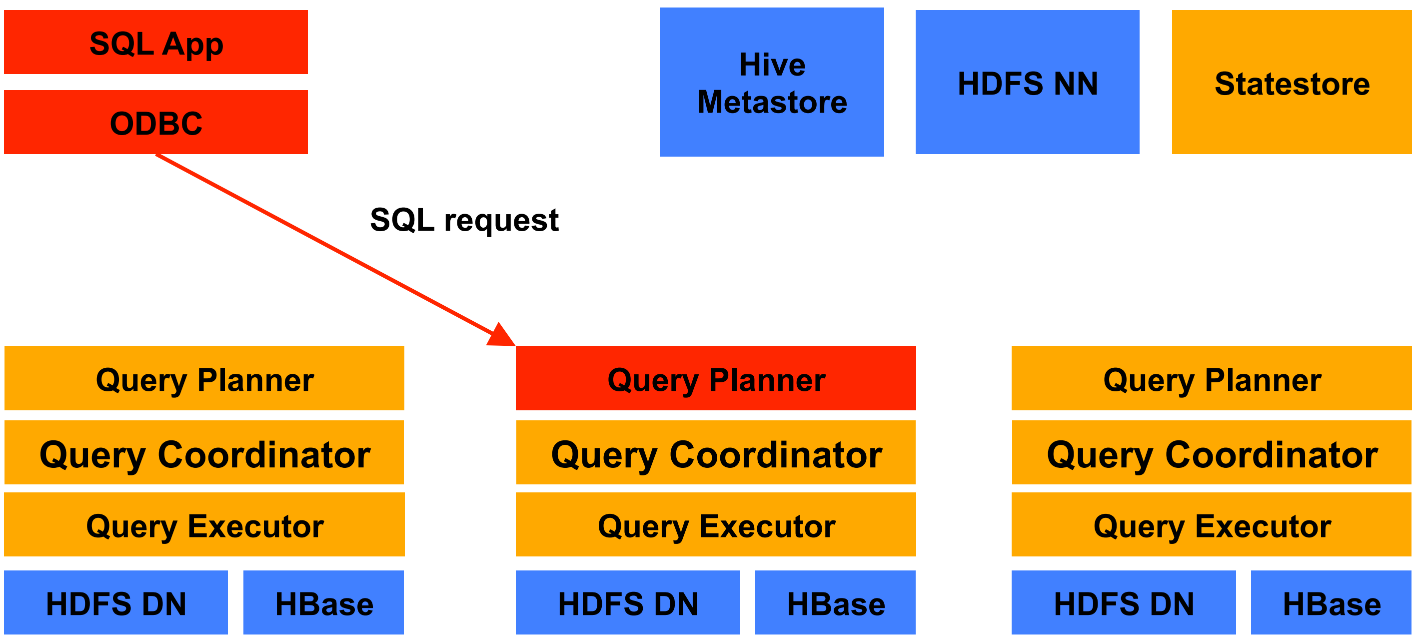
- Open source analytical database for Hadoop
- Tight integration with HDFS and Parquet format
- Released October 2012

Impala Architecture

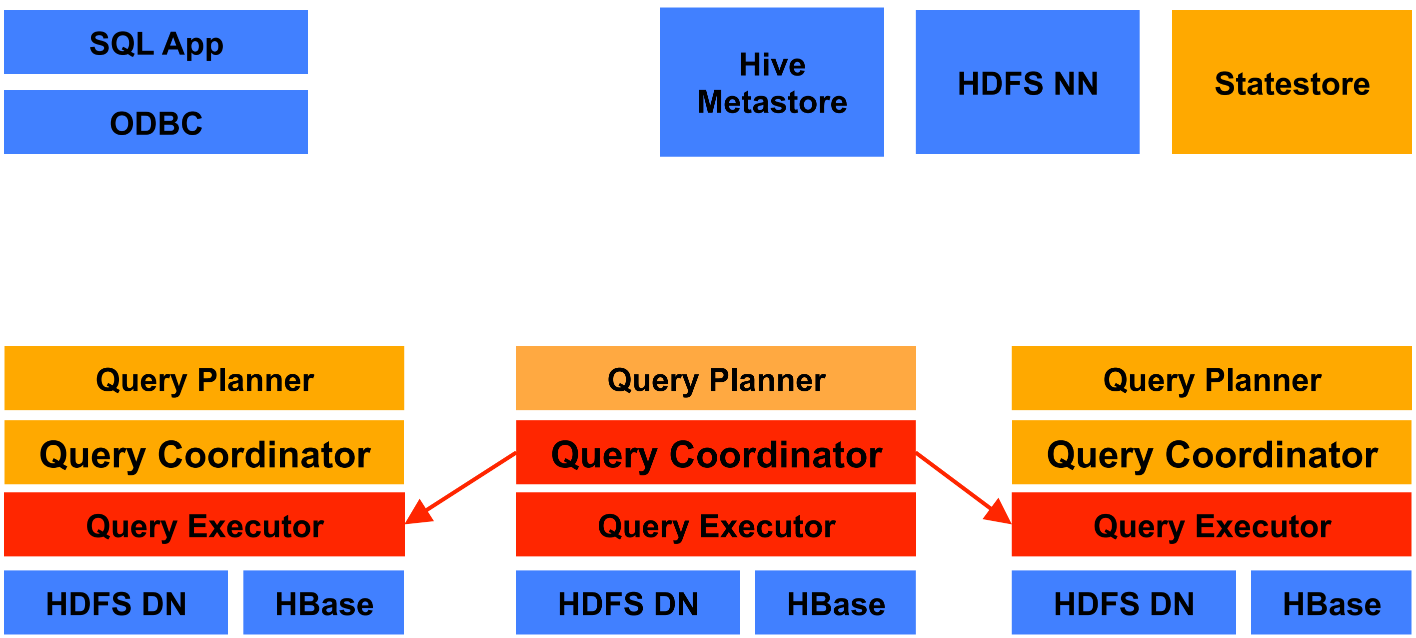
- Impala daemon (impalad)
  - Handles client requests
  - Handles internal query execution requests
- State store daemon (statestored)
  - Provides name service and metadata distribution

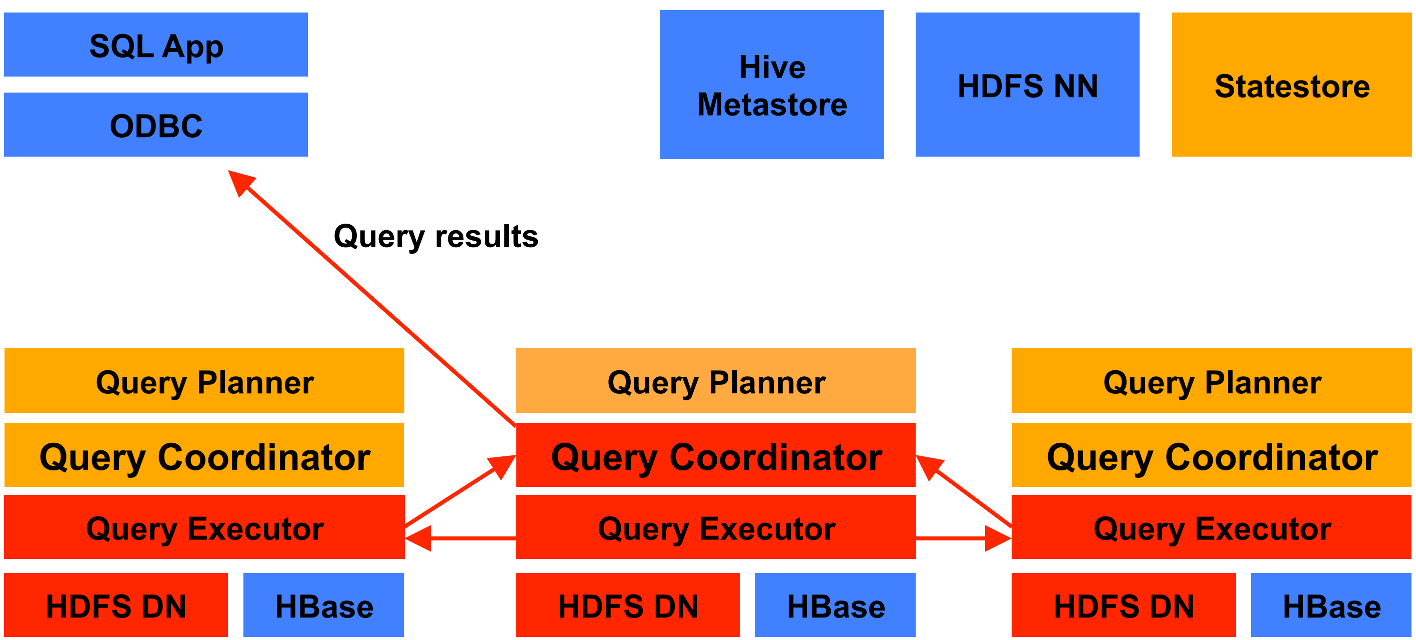
Impala Query Execution

1. Request arrives
2. Planner turns request into collections of plan fragments
3. Coordinator initiates execution on remote impala daemons
4. Intermediate results are streamed between executors
5. Query results are streamed back to client



SQL App
ODBC
Hive Metastore
HDFS NN
Statestore
SQL request
Query Planner
Query Coordinator
Query Executor
HDFS DN
HBase
Query Planner
Query Coordinator
Query Executor
HDFS DN
HBase
Query Planner
Query Coordinator
Query Executor
HDFS DN
HBase


SQL App
ODBC
Hive Metastore
HDFS NN
Statestore
Query Planner
Query Coordinator
Query Executor
HDFS DN
HBase
Query Planner
Query Coordinator
Query Executor
HDFS DN
HBase
Query Planner
Query Coordinator
Query Executor
HDFS DN
HBase



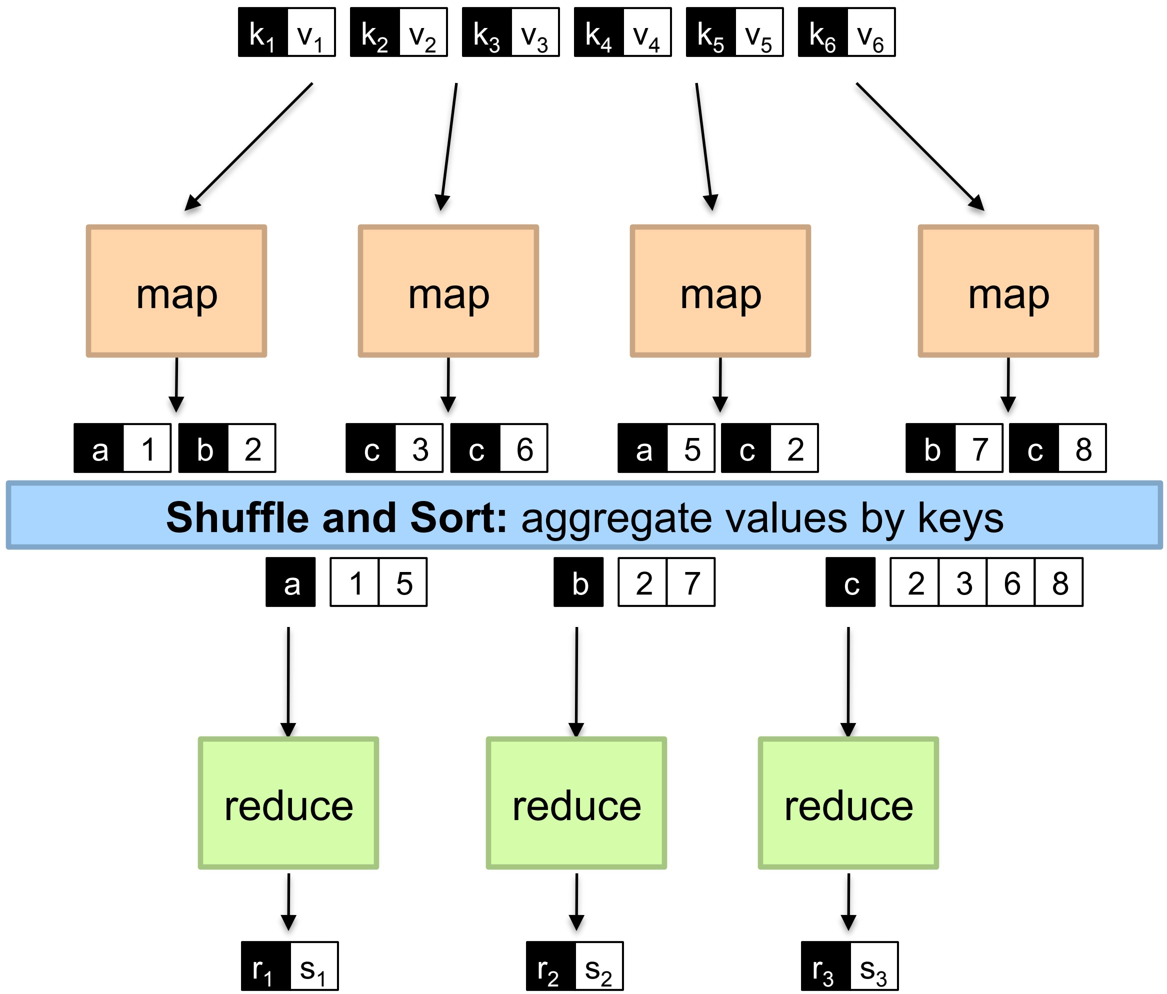
SQL App
ODBC
Hive Metastore
HDFS NN
Statestore
Query results
Query Planner
Query Coordinator
Query Executor
HDFS DN
HBase
Query Planner
Query Coordinator
Query Executor
HDFS DN
HBase
Query Planner
Query Coordinator
Query Executor
HDFS DN
HBase

Impala Execution Engine

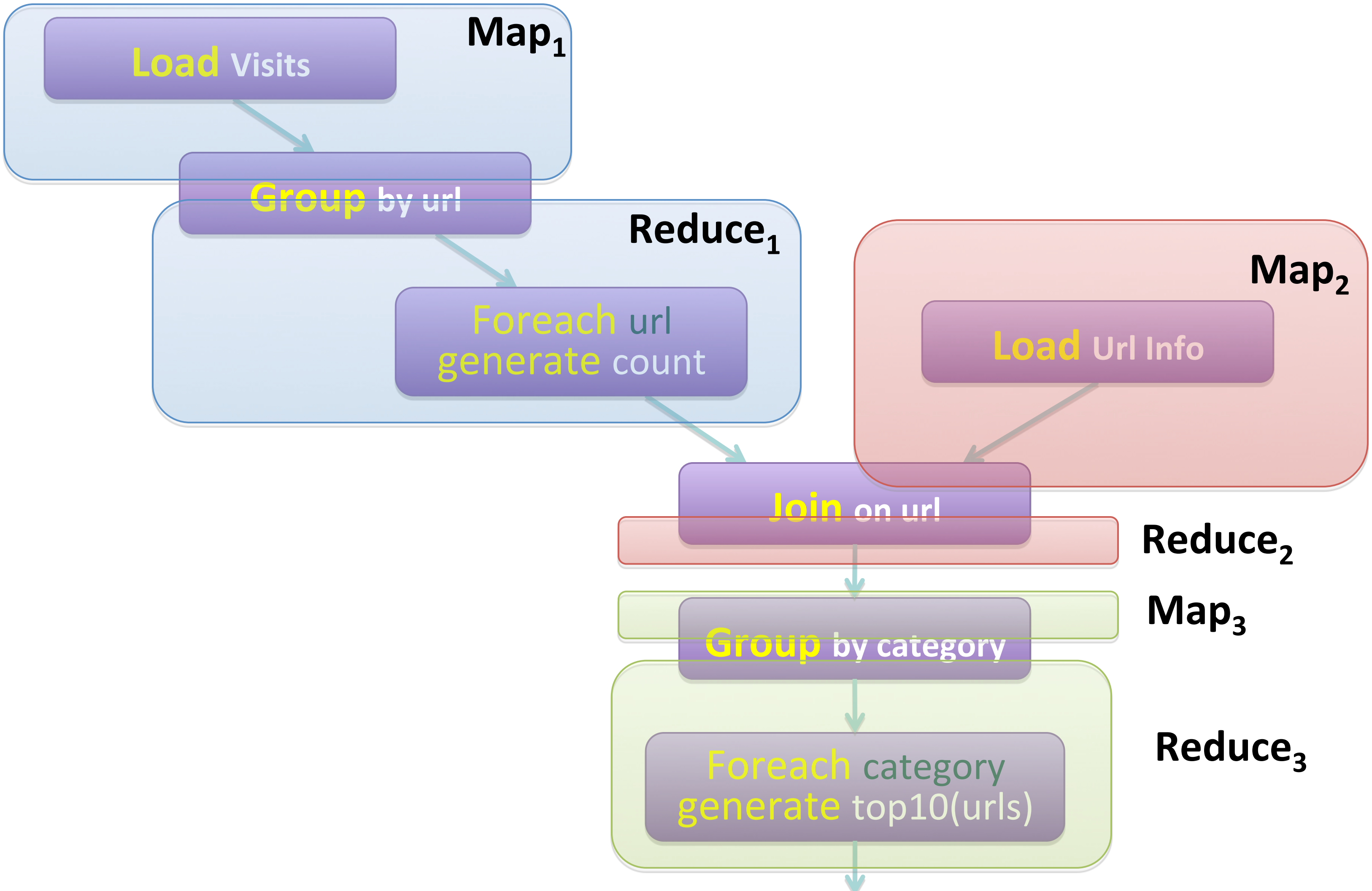
- Written in C++
- Runtime code generation for “big loops” (via LLVM)


The datacenter *is* the computer!
What's the instruction set?
Source: Google

Answer?

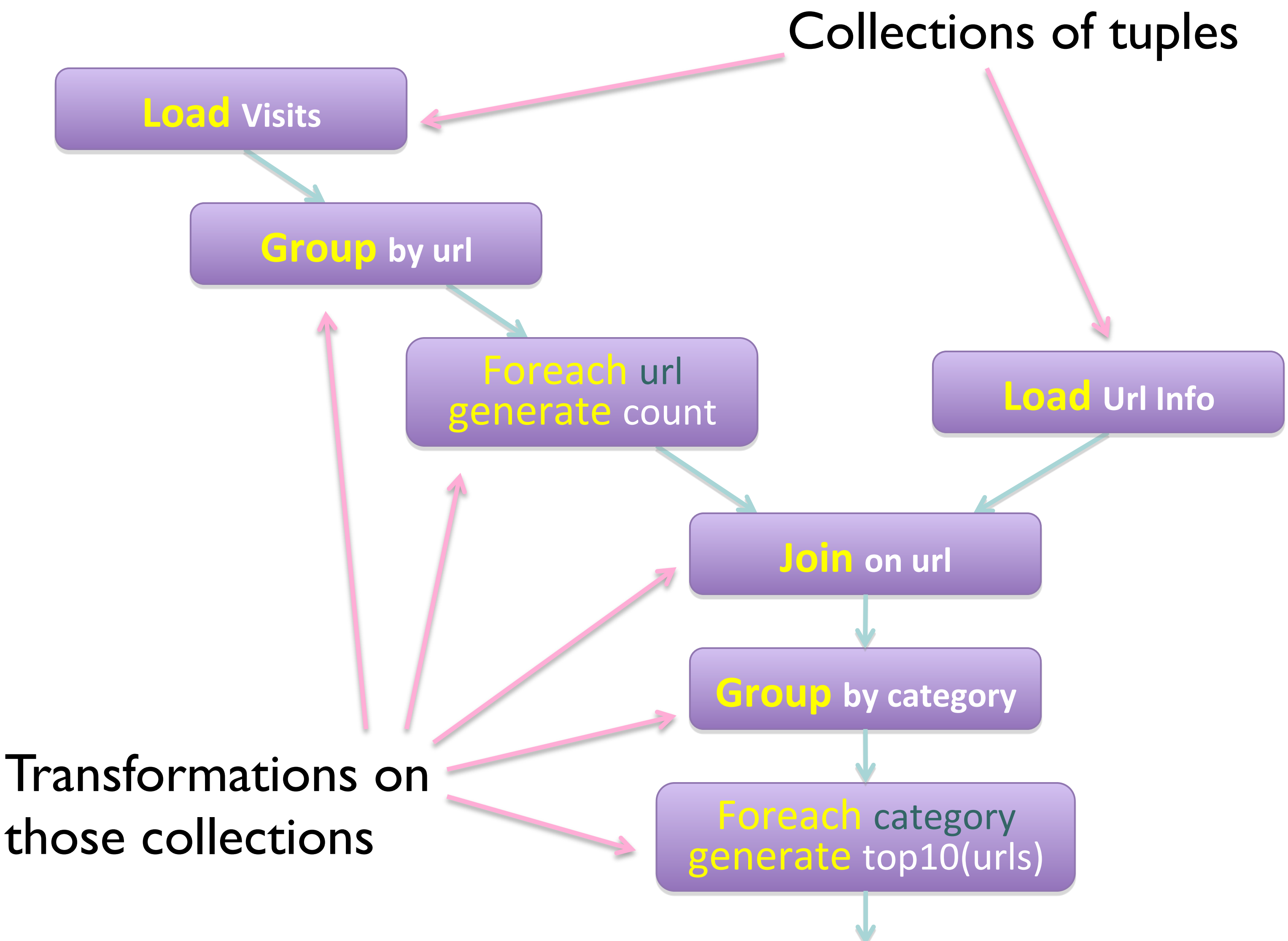

k_1 v_1 k_2 v_2 k_3 v_3 k_4 v_4 k_5 v_5 k_6 v_6
map
map
map
map
a 1 b 2
c 3 c 6
a 5 c 2
b 7 c 8
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 3 6 8
reduce
reduce
reduce
r_1 s_1
r_2 s_2
r_3 s_3

Answer?



Pig Slides adapted from Olston et al. (SIGMOD 2008)

Generically, what is this?

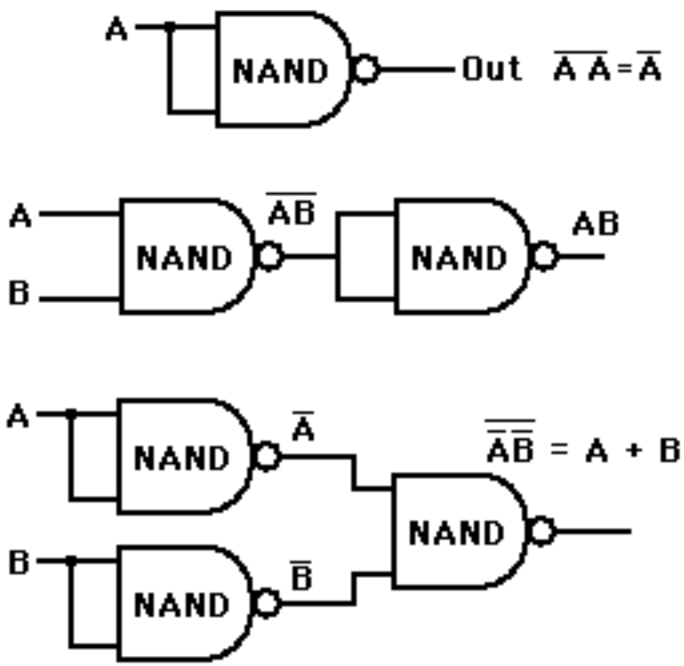


Pig Slides adapted from Olston et al. (SIGMOD 2008)

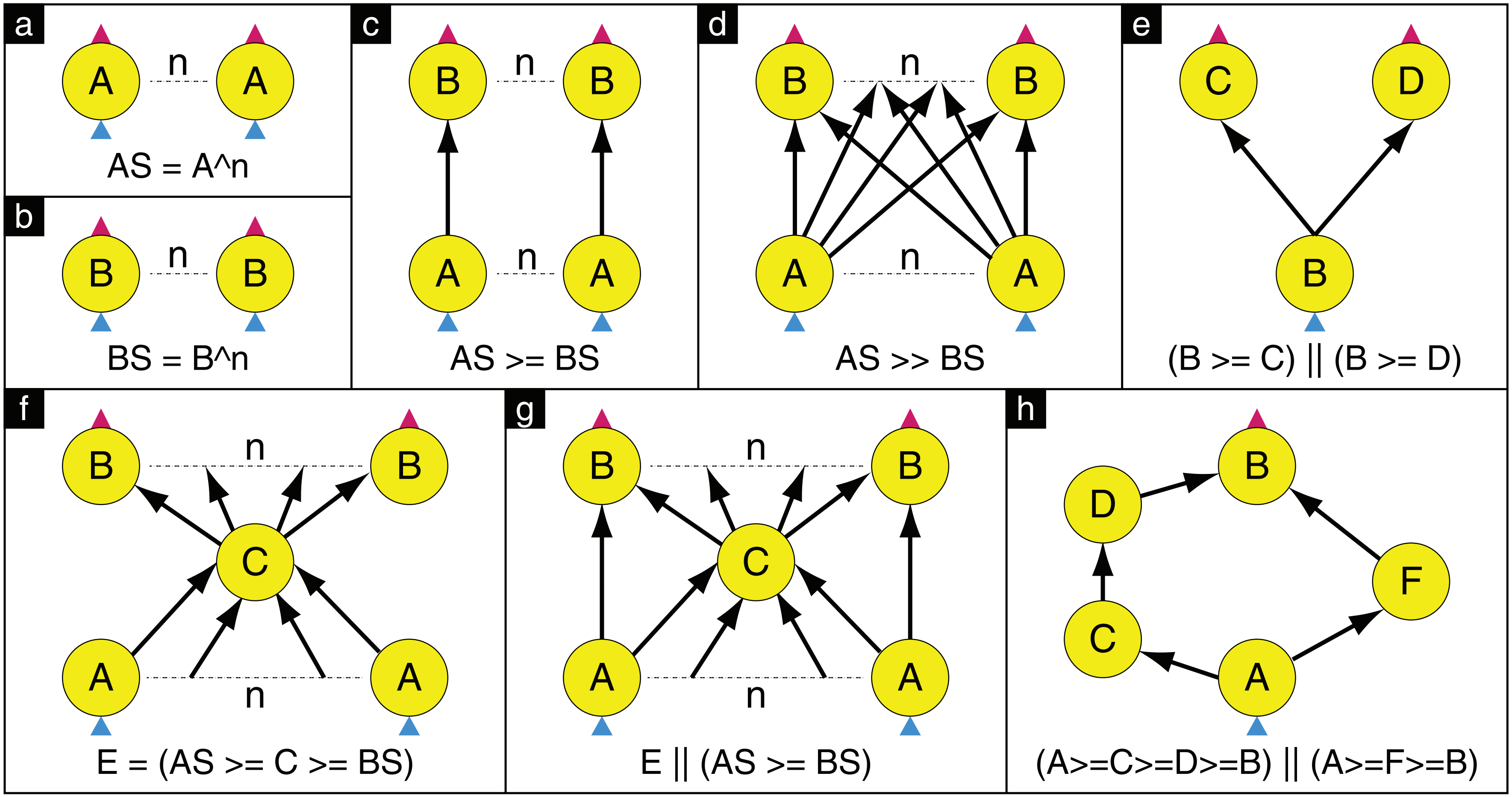
Dataflows

- Comprised of:
  - Collections of records
  - Transformations on those collections
- Two important questions:
  - What are the logical operators?
  - What are the physical operators?

Analogy: NAND Gates are universal

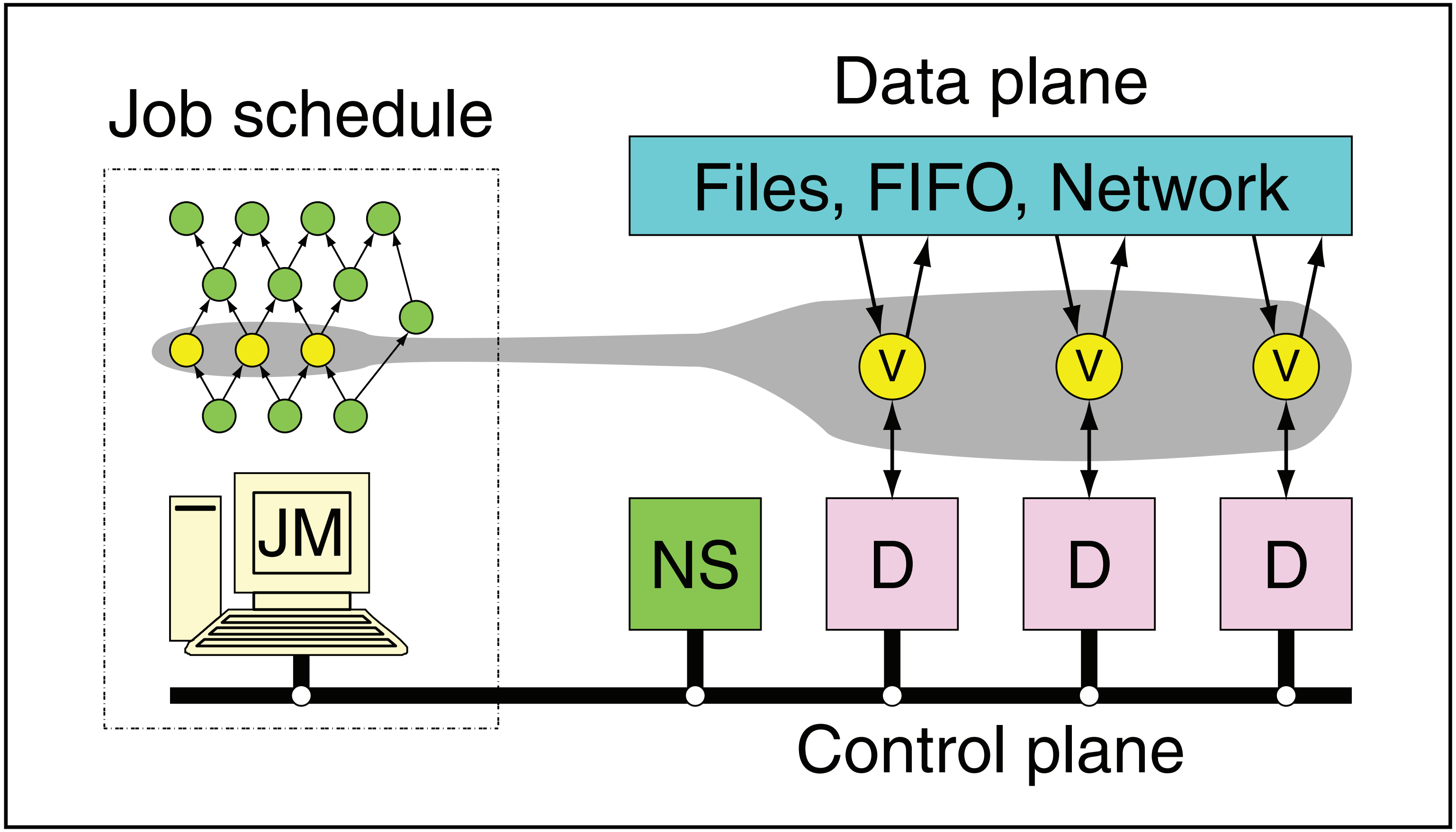


Dryad: Graph Operators



Source: Isard et al. (2007) Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. EuroSys.

Dryad: Architecture



The Dryad system organization. The job manager (JM) consults the name server (NS) to discover the list of available computers. It maintains the job graph and schedules running vertices (V) as computers become available using the daemon (D) as a proxy. Vertices exchange data through files, TCP pipes, or shared-memory channels. The shaded bar indicates the vertices in the job that are currently running.

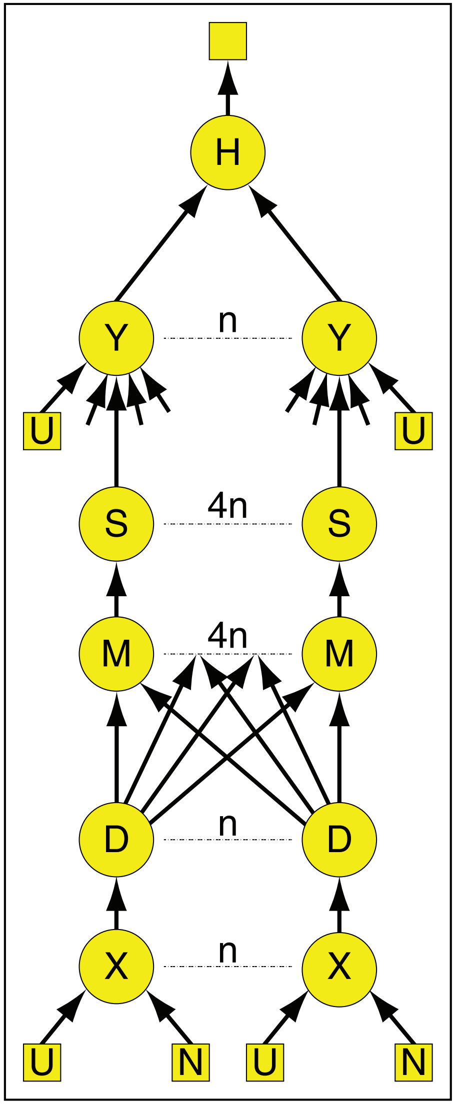
Source: Isard et al. (2007) Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. EuroSys.

Dryad: Cool Tricks

- Channel: abstraction for vertex-to-vertex communication
  - File
  - TCP pipe
  - Shared memory
- Runtime graph refinement
  - Size of input is not known until runtime
  - Automatically rewrite graph based on invariant properties

Source: Isard et al. (2007) Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. EuroSys.

Dryad: Sample Program




```
GraphBuilder XSet = moduleX^N;
GraphBuilder DSet = moduleD^N;
GraphBuilder MSet = moduleM^(N*4);
GraphBuilder SSet = moduleS^(N*4);
GraphBuilder YSet = moduleY^N;
GraphBuilder HSet = moduleH^1;

GraphBuilder XInputs = (ugriz1 >= XSet) || (neighbor >= XSet);
GraphBuilder YInputs = ugriz2 >= YSet;

GraphBuilder XToY = XSet >= DSet >> MSet >= SSet;
for (i = 0; i < N*4; ++i)
{
    XToY = XToY || (SSet.GetVertex(i) >= YSet.GetVertex(i/4));
}

GraphBuilder YToH = YSet >= HSet;
GraphBuilder HOutputs = HSet >= output;

GraphBuilder final = XInputs || YInputs || XToY || YToH || HOutputs;
```

Source: Isard et al. (2007) Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. EuroSys.

DryadLINQ

- LINQ = Language INtegrated Query
  - .NET constructs for combining imperative and declarative programming
- Developers write in DryadLINQ
  - Program compiled into computations that run on Dryad

Sound familiar?

Source: Yu et al. (2008) DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. OSDI.

DryadLINQ: Word Count

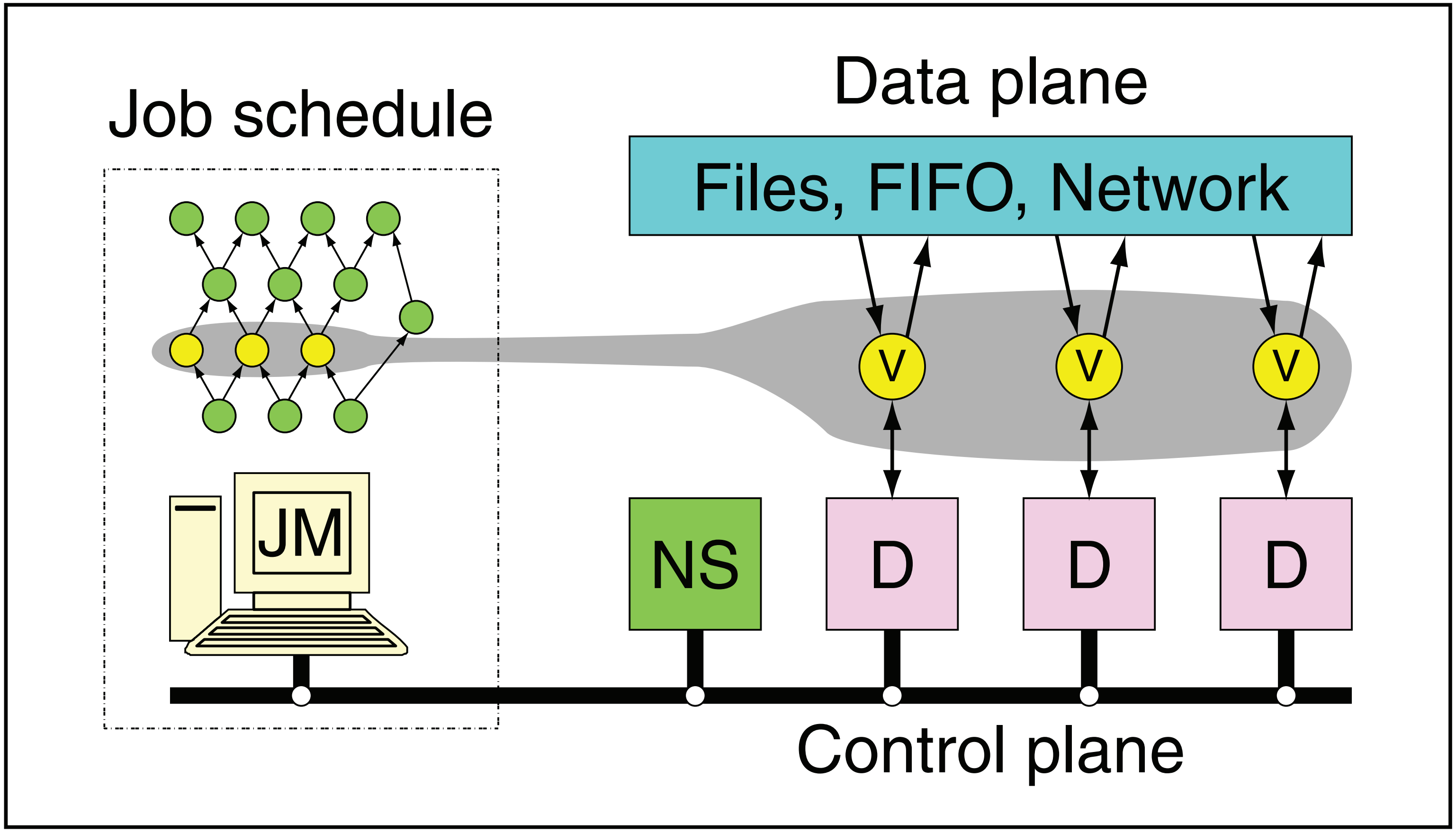
```
PartitionedTable<LineRecord> inputTable =
  PartitionedTable.Get<LineRecord>(uri);

IQueryable<string> words = inputTable.SelectMany(x => x.line.Split(' '));
IQueryable<IGrouping<string, string>> groups = words.GroupBy(x => x);
IQueryable<Pair> counts = groups.Select(x => new Pair(x.Key, x.Count()));
IQueryable<Pair> ordered = counts.OrderByDescending(x => x.Count);
IQueryable<Pair> top = ordered.Take(k);
```

Compare:

```
a = load ’file.txt' as (text: chararray);
b = foreach a generate flatten(TOKENIZE(text)) as term;
c = group b by term;
d = foreach c generate group as term, COUNT(b) as count;

store d into 'cnt';
```

Compare and contrast…

What happened to Dryad?



The Dryad system organization. The job manager (JM) consults the name server (NS) to discover the list of available computers. It maintains the job graph and schedules running vertices (V) as computers become available using the daemon (D) as a proxy. Vertices exchange data through files, TCP pipes, or shared-memory channels. The shaded bar indicates the vertices in the job that are currently running.

Source: Isard et al. (2007) Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. EuroSys.

Spark

- One popular answer to "What's beyond MapReduce?"
- Open-source engine for large-scale batch processing
  - Supports generalized dataflows
  - Written in Scala, with bindings in Java and Python
- Brief history:
  - Developed at UC Berkeley AMPLab in 2009
  - Open-sourced in 2010
  - Became top-level Apache project in February 2014
  - Commercial support provided by DataBricks

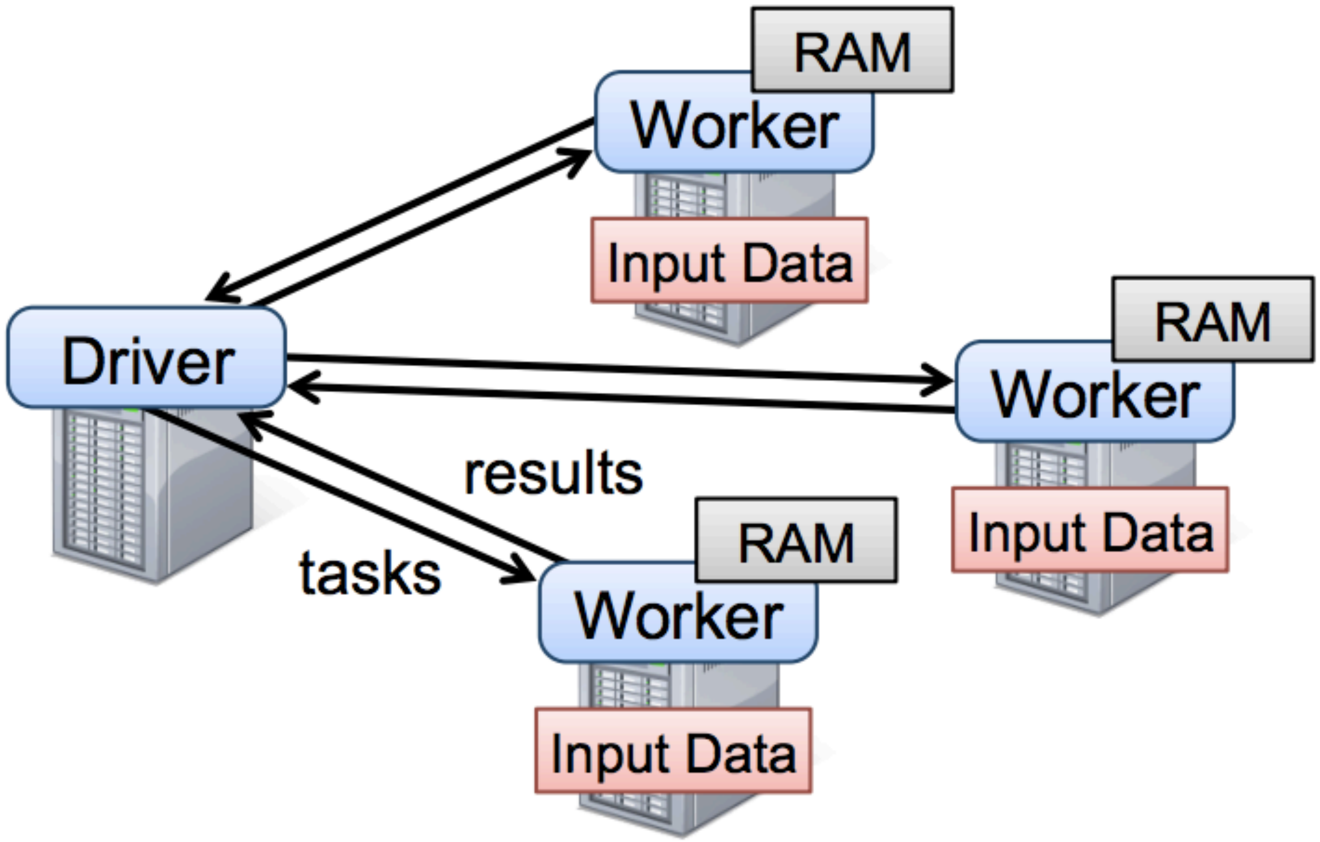
Resilient Distributed Datasets (RDDs)

- RDD: Spark "primitive" representing a collection of records
  - Immutable
  - Partitioned (the *D* in RDD)
- Transformations operate on an RDD to create another RDD
  - Coarse-grained manipulations only
  - RDDs keep track of *lineage*
- Persistence
  - RDDs can be materialized in memory or on disk
  - OOM or machine failures: What happens?
- Fault tolerance (the *R* in RDD):
  - RDDs can *always* be recomputed from stable storage (disk)

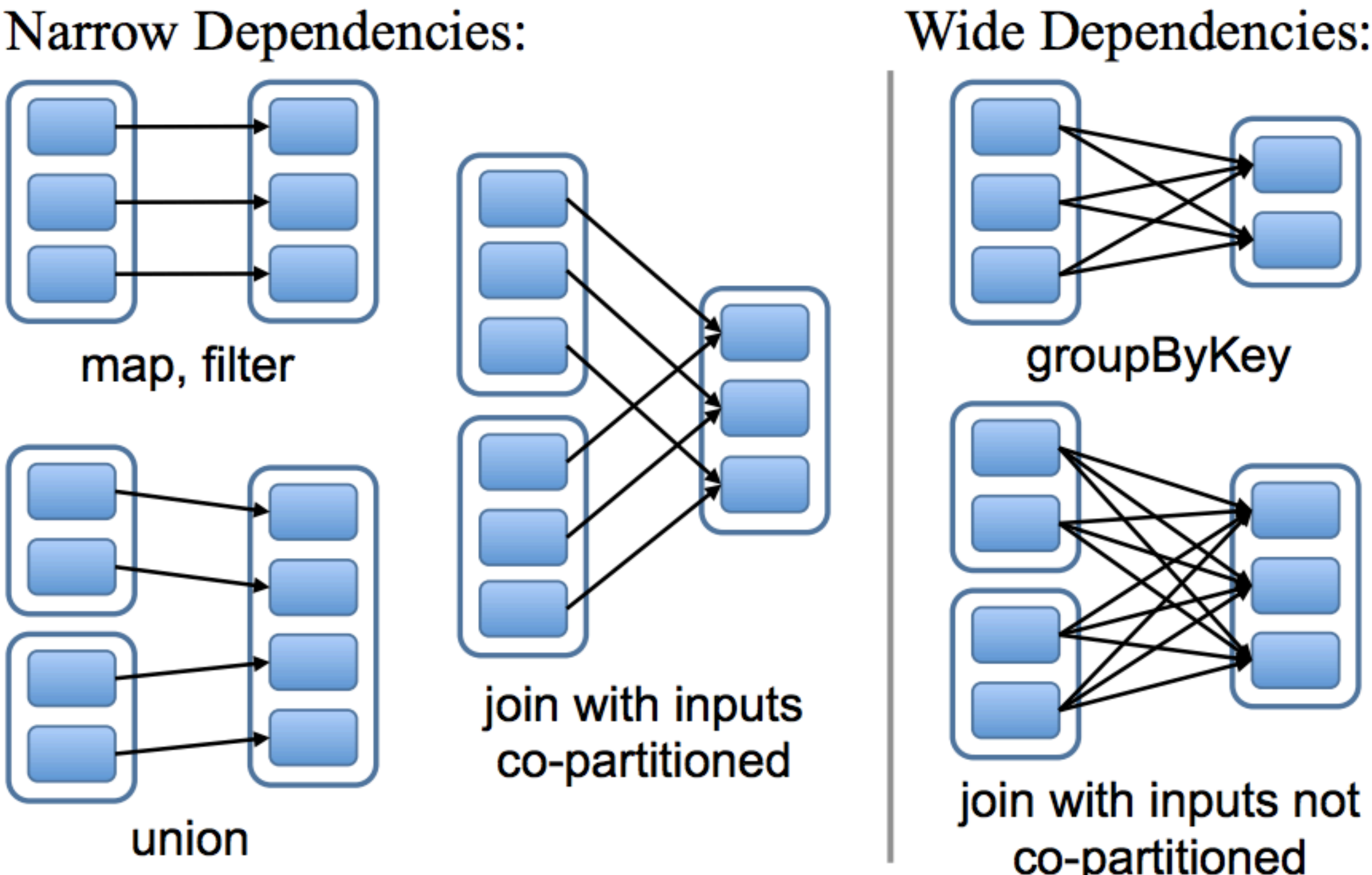
Operations on RDDs

- Transformations (lazy):
  - map
  - flatMap
  - filter
  - union/intersection
  - join
  - reduceByKey
  - groupByKey
  - …
- Actions (actually trigger computations)
  - collect
  - saveAsTextFile/saveAsSequenceFile
  - …

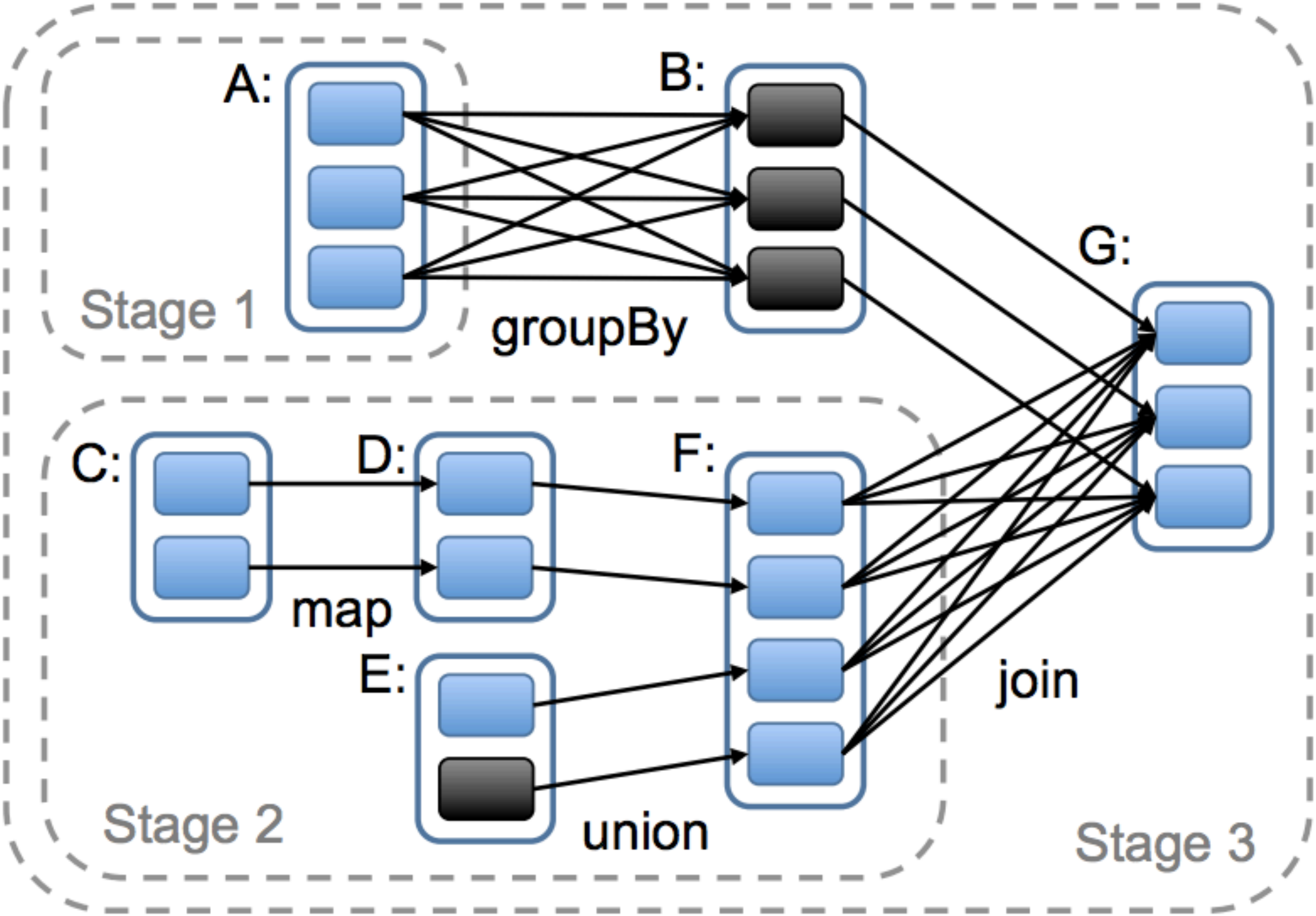
Spark Architecture



Spark Physical Operators



Spark Execution Plan



Today’s Agenda

- What’s beyond MapReduce?
  - SQL on Hadoop
  - Dataflow languages
- Past and present developments

Questions?

Source: Wikipedia (Japanese rock garden)