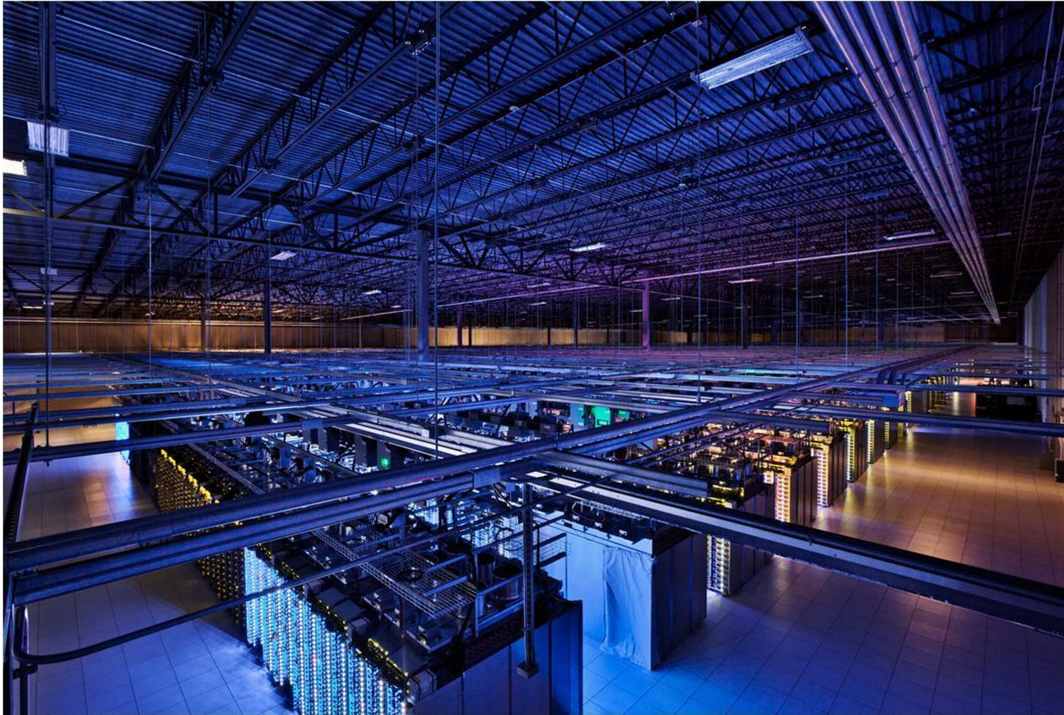
CWI
Centrum Wiskunde & Informatica

Large-Scale Data Engineering

Frameworks Beyond MapReduce



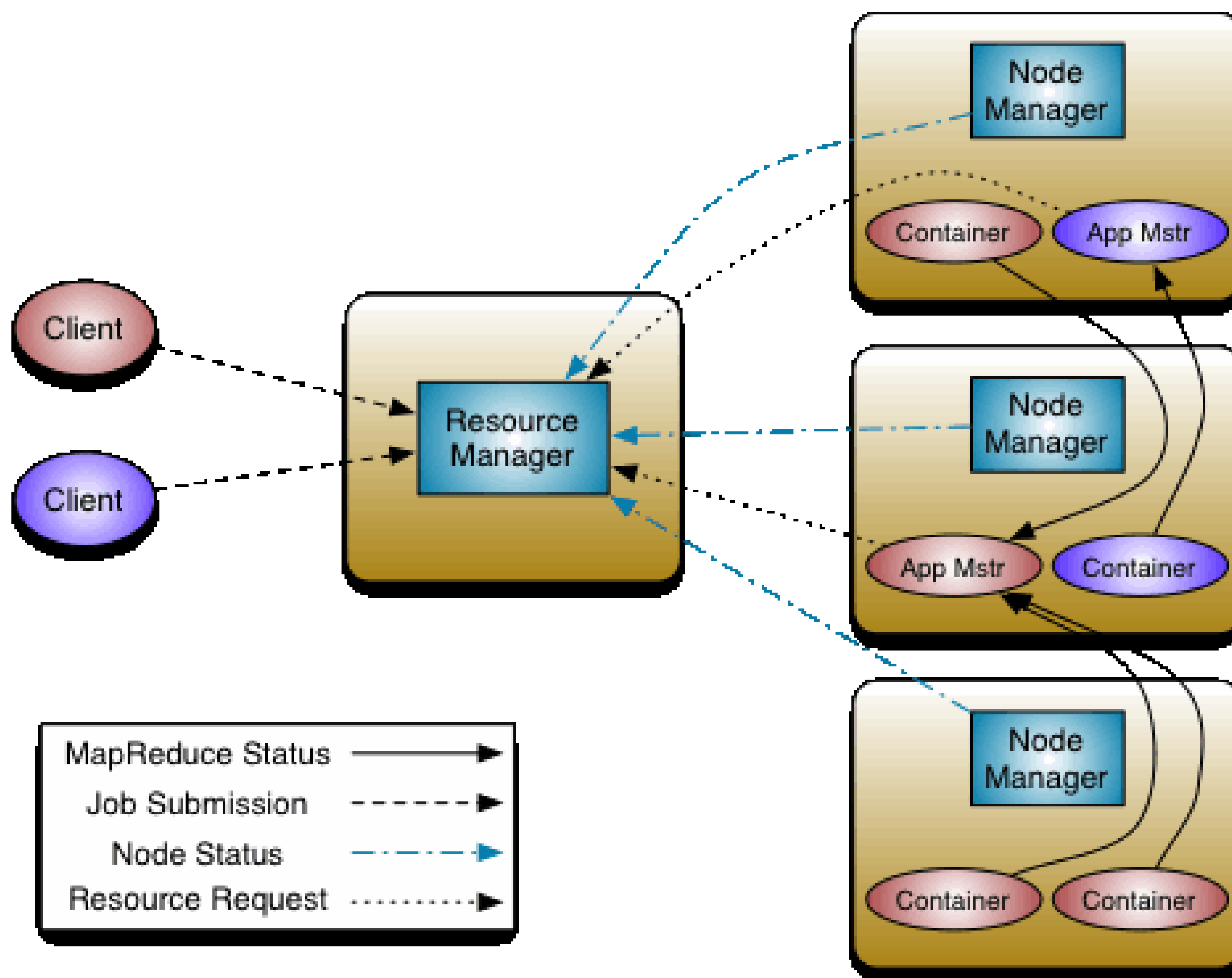
event.cwi.nl/lsde

THE HADOOP ECOSYSTEM

YARN: Hadoop version 2.0

- Hadoop limitations:
  - Can only run MapReduce
  - What if we want to run other distributed frameworks?
- YARN = Yet-Another-Resource-Negotiator
  - Provides API to develop any generic distribution application
  - Handles scheduling and resource request
  - MapReduce (MR2) is one such application in YARN

YARN: architecture



The Hadoop Ecosystem

fast in-memory processing

graph analysis

machine learning

MLIB

graphX

SparkSQL

Spark

cascading

data querying

HCATALOG

Impala

APACHE DRILL

APACHE GIRAPH

hadoop Map Reduce

HDFS

YARN

The Hadoop Ecosystem

- **Basic services**
  - HDFS = Open-source GFS clone originally funded by Yahoo
  - MapReduce = Open-source MapReduce implementation (Java,Python)
  - YARN = Resource manager to share clusters between MapReduce and other tools
  - HCATALOG = Meta-data repository for registering datasets available on HDFS (Hive Catalog)
  - Cascading = Dataflow tool for creating multi-MapReduce job dataflows (Driven = GUI for it)
  - Spark = new in-memory MapReduce++ based on Scala (avoids HDFS writes)
- **Data Querying**
  - Pig = Relational Algebra system that compiles to MapReduce
  - Hive = SQL system that compiles to MapReduce (Hortonworks)
  - Impala, or, Drill = efficient SQL systems that do *not* use MapReduce (Cloudera,MapR)
  - SparkSQL = SQL system running on top of Spark
- **Graph Processing**
  - Giraph = Pregel clone on Hadoop (Facebook)
  - GraphX = graph analysis library of Spark
- **Machine Learning**
  - Okapi = Giraph –based library of machine learning algorithms (graph-oriented)
  - Mahout = MapReduce-based library of machine learning algorithms
  - MLib = Spark –based library of machine learning algorithms

CWI

Centrum Wiskunde & Informatica

HIGH-LEVEL WORKFLOWS

HIVE & PIG

Need for high-level languages

- Hadoop is great for large-data processing!
  - But writing Java/Python/… programs for everything is verbose and slow
  - Cumbersome to work with multi-step processes
  - “Data scientists” don’t want to / can not 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

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

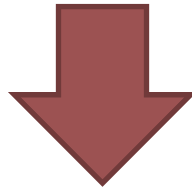
```
SELECT s.word, s.freq, k.freq FROM shakespeare s
  JOIN bible k ON (s.word = k.word) WHERE s.freq >= 1 AND k.freq >= 1
  ORDER BY s.freq DESC LIMIT 10;

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

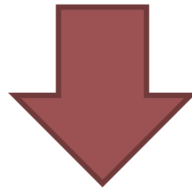
Hive: behind the scenes

```
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

Pig: example

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

Visits

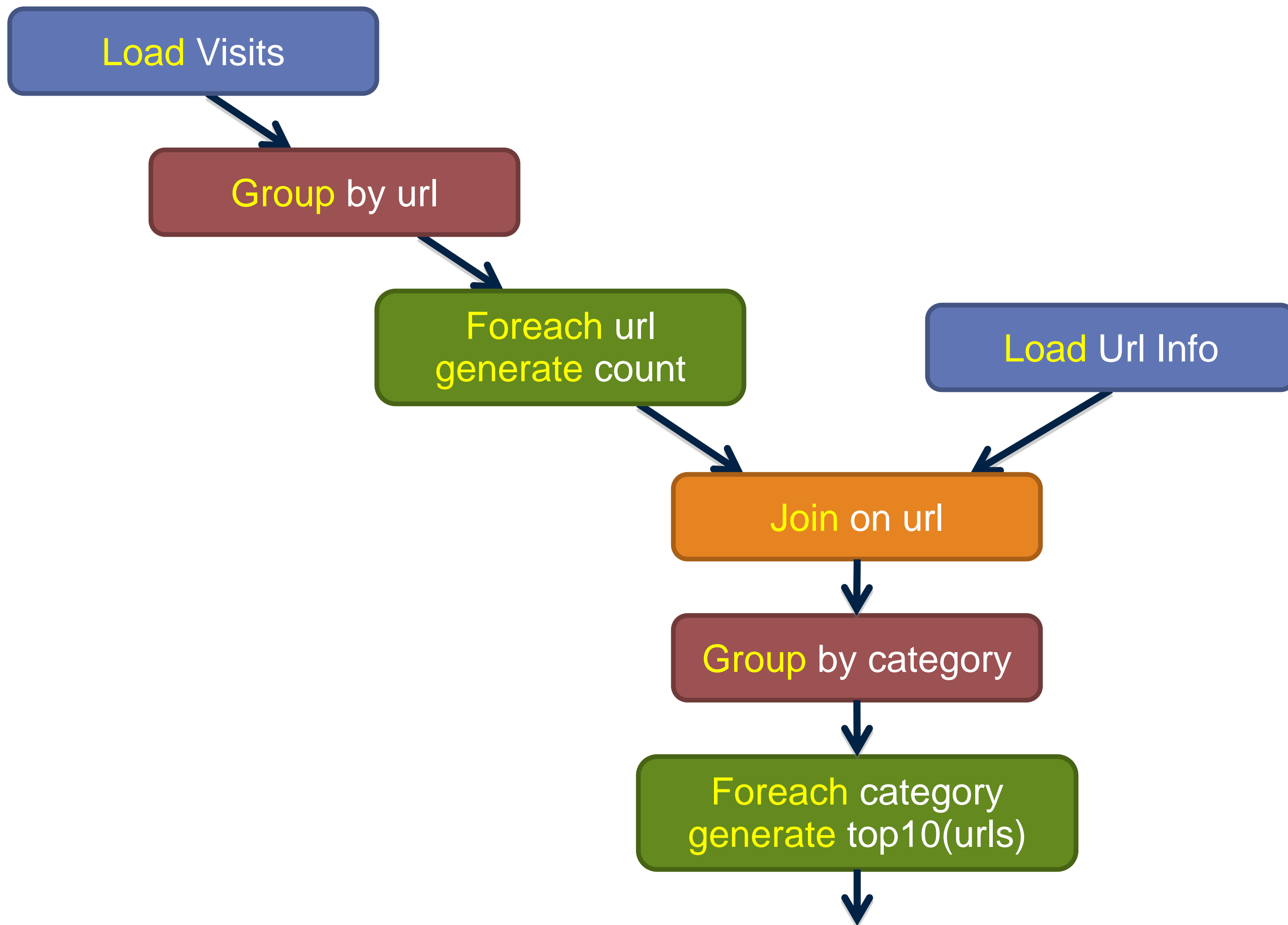
User	Url	Time
Amy	cnn.com	8:00
Amy	bbc.com	10:00
Amy	flickr.com	10:05
Fred	cnn.com	12:00

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 query plan



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

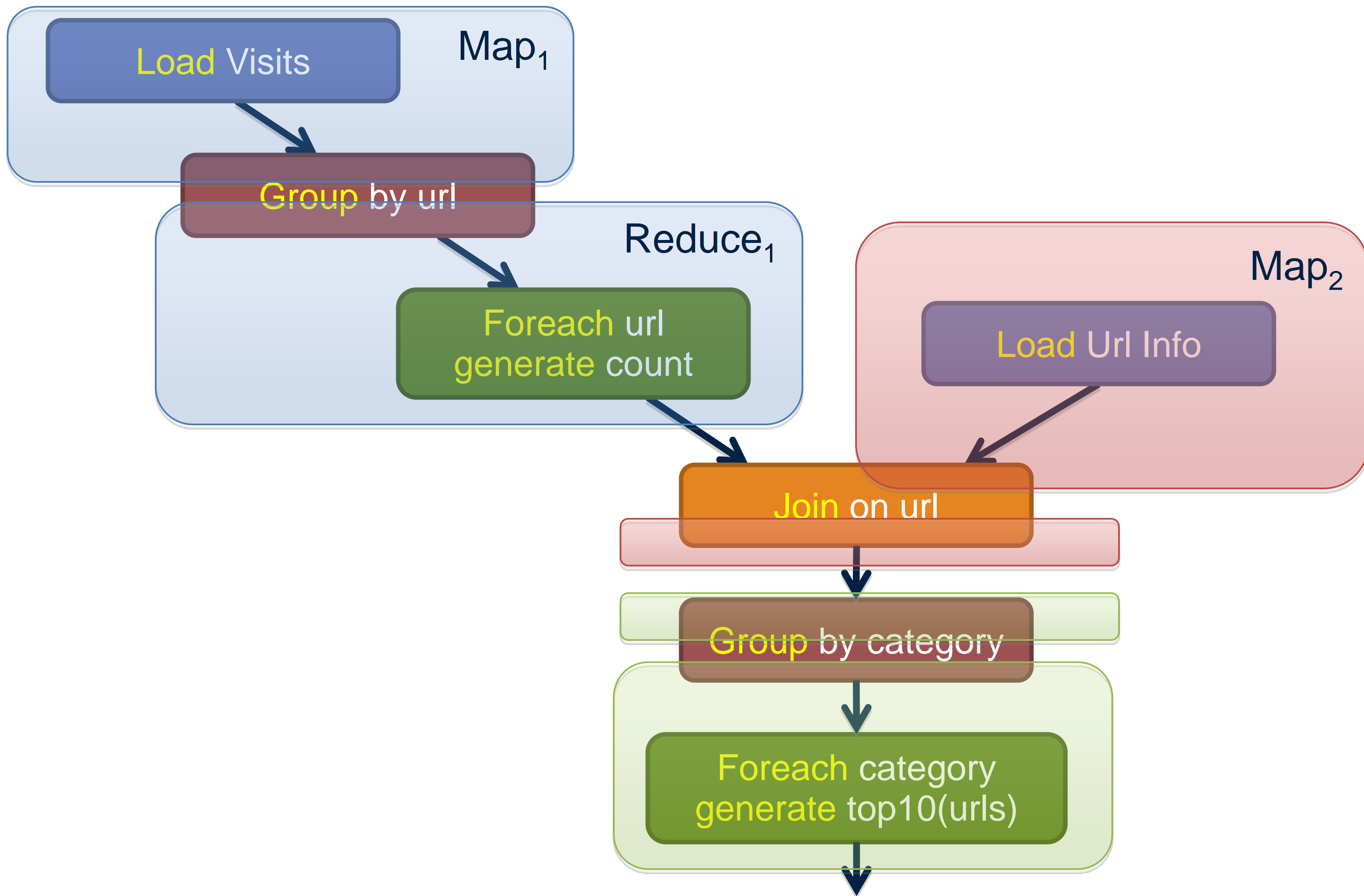
Pig script

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

Pig query plan



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

Digging further into Pig: basics

- Sequence of statements manipulating relations (aliases)
- Data model
  - Scalars (int, long, float, double, chararray, bytearray)
  - Tuples (ordered set of fields)
  - Bags (collection of tuples)

Pig: common operations

- Loading/storing data
  - LOAD, STORE
- Working with data
  - FILTER, FOREACH, GROUP, JOIN, ORDER BY, LIMIT, …
- Debugging
  - DUMP, DESCRIBE, EXPLAIN, ILLUSTRATE

Pig: LOAD/STORE data

```
A = LOAD 'data' AS (a1:int,a2:int,a3:int);

STORE A INTO 'data2’;
STORE A INTO 's3://somebucket/data2';
```

Pig: FILTER data

```
X = FILTER A BY a3 == 3;

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

Pig: FOREACH

```
X = FOREACH A GENERATE a1, a2;

X = FOREACH A GENERATE a1+a2 AS f1:int;
```

Pig: ORDER BY / LIMIT

```
X = LIMIT A 2;

(1,2,3)
(4,2,1)


X = ORDER A BY a1;

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

CWI
Centrum Wiskunde & Informatica

Pig: GROUPing

```
G = GROUP A BY a1;

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

Bags

Pig: Dealing with grouped data

```
G = GROUP A BY a1;
R = FOREACH G GENERATE group, COUNT(A);

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

Pig: Dealing with grouped data

```
G = GROUP A BY a1;
R = FOREACH G GENERATE group, SUM(A.a3);

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

Pig: Dealing with grouped data

```
G = GROUP A BY a1;
R = FOREACH G {
        O = ORDER A BY a2;
        L = LIMIT O 1;
        GENERATE FLATTEN(L);
}
```

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

G

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

Pig: JOINs

```
A1 = LOAD 'data' AS (a1:int,a2:int,a3:int);
A2 = LOAD 'data' AS (a1:int,a2:int,a3:int);
J = JOIN A1 BY a1, A2 BY a3;

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

Pig: DESCRIBE (Show Schema)

```
DESCRIBE A;
A: {a1: int,a2: int,a3: int}
```

Pig: ILLUSTRATE (Show Lineage)

```
G = GROUP A BY a1;
R = FOREACH G GENERATE group, SUM(A.a3);
ILLUSTRATE R;
```

```
---------------------------------------------------
| A    | a1:int    | a2:int    | a3:int    |
---------------------------------------------------
|      | 8         | 4         | 3         |
|      | 8         | 3         | 4         |
---------------------------------------------------
-----------------------------------------------------------------------------------
| G    | group:int    | A:bag{:tuple(a1:int,a2:int,a3:int)}             |
-----------------------------------------------------------------------------------
|      | 8            | {}                                              |
|      | 8            | {}                                              |
-----------------------------------------------------------------------------------
---------------------------------------
| R    | group:int    | :long    |
---------------------------------------
|      | 8            | 7        |
---------------------------------------
```

Pig: DUMP (careful!)

```
DUMP A;
(1,2,3)
(4,2,1)
(8,3,4)
(4,3,3)
(7,2,5)
(8,4,3)
```

Centrum Wiskunde & Informatica

OK.. Live Demo

http://demo.gethue.com → Query Editors → Pig

```
lines = LOAD '/user/hue/pig/examples/data/midsummer.txt' as (text:CHARARRAY);
words = FOREACH lines GENERATE FLATTEN(TOKENIZE(text,' '));
grouped = GROUP words BY token;
counted = FOREACH grouped GENERATE group,COUNT(words) AS cnt;
filtered = FILTER counted BY cnt > 40;
ordered = ORDER filtered BY cnt;
DUMP ordered;
EXPLAIN ordered;
DESCRIBE ordered;
```

Pig: EXPLAIN (Execution plan)

EXPLAIN R;

```
Map Plan
G: Local Rearrange[tuple]{int}(false)
|
|---R: New For Each(false,false)[bag]
    |
    |---Pre Combiner Local Rearrange[tuple]{Unknown}
        |
        |---A: New For Each(false,false,false)[bag]
            |
            |---A: Load(file:///Users/hannes/data:org.apache.pig.builtin.PigStorage)

Combine Plan
G: Local Rearrange[tuple]{int}(false)
|
|---R: New For Each(false,false)[bag]
    |
    |---G: Package(CombinerPackager)[tuple]{int}
Reduce Plan
R: Store(fakefile:org.apache.pig.builtin.PigStorage)
|
|---R: New For Each(false,false)[bag]
    |
    |---G: Package(CombinerPackager)[tuple]{int}
Global sort: false
```

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();
```

Iterative computation

```
#!/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
```

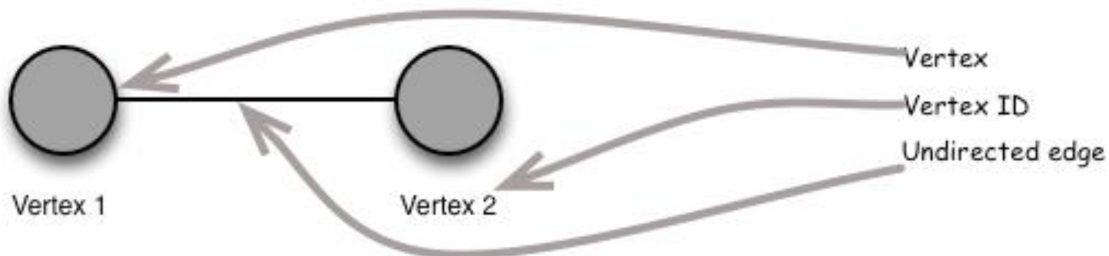
Uuuugly!

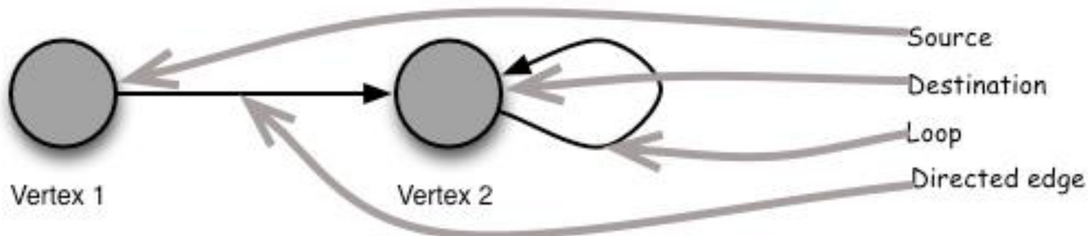
From: http://techblug.wordpress.com/2011/07/29/pagerank-implementation-in-pig/

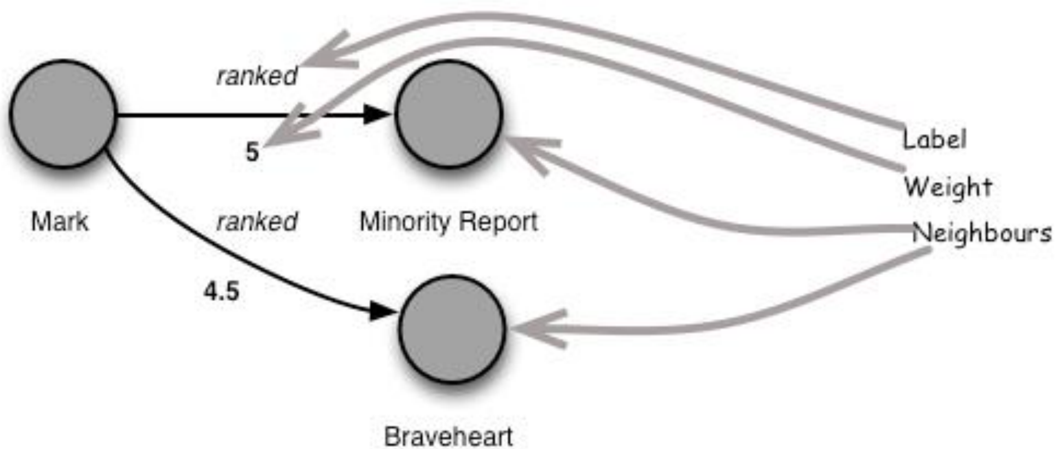
GOOGLE PREGEL &

GIRAPH: LARGE-SCALE GRAPH PROCESSING ON HADOOP

Graphs are Simple







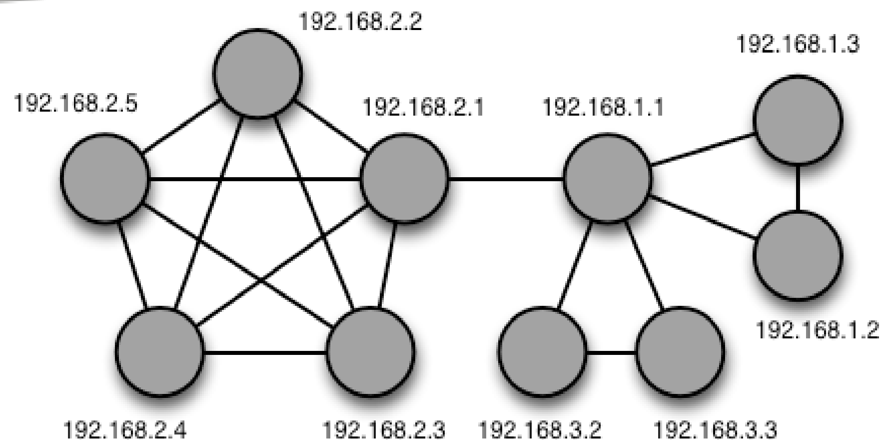
A Computer Network

192.168.2.2
192.168.2.1
192.168.1.1, 192.168.3.1
192.168.1.3
192.168.2.5
192.168.1.2
192.168.3.3
192.168.3.2
192.168.2.4
192.168.2.3

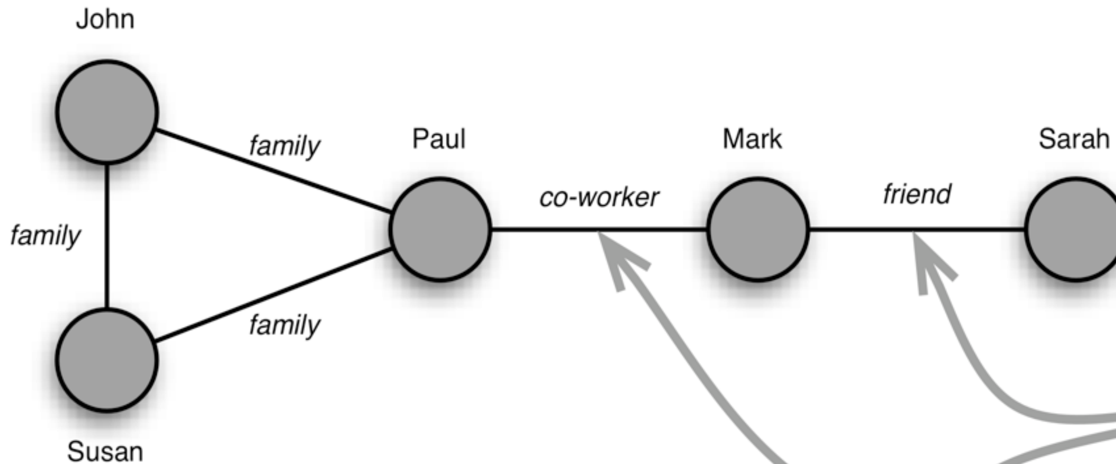
Desktop
Switch
Router
WiFi Access point
Tablet
Phone
Server

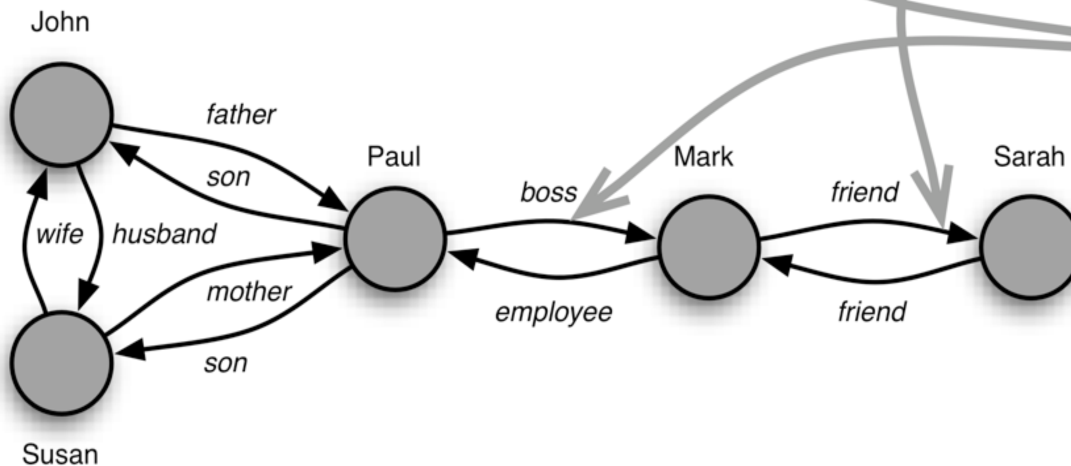
Note

1. There are three networks: servers, desktops and mobile.
2. They are connected through two routers/firewalls.
3. We ignored the switches and the access point in the graph.
4. Router 192.168.1.1 has two interfaces but we used one as vertex ID.



A Social Network

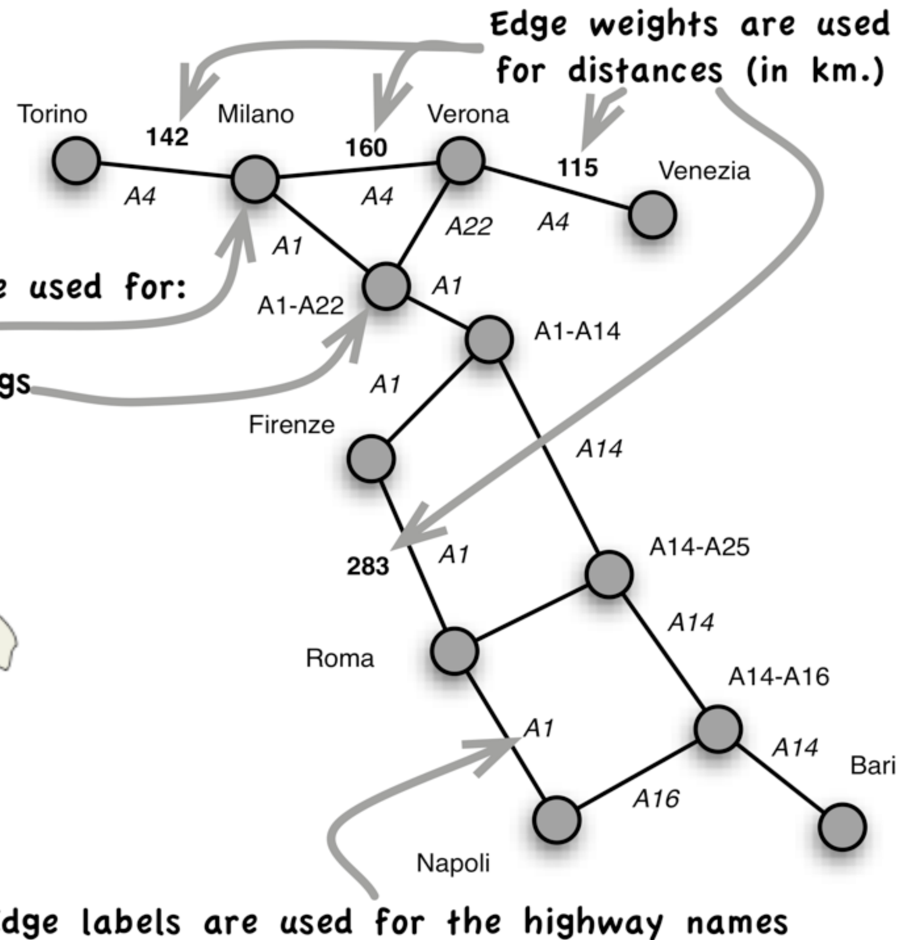




Note

1. A symmetric relationship is substituted by two directed edges.
2. A relationship does not have to be substituted by two edges, but e.g. by a more specific one.

Maps are Graphs as well



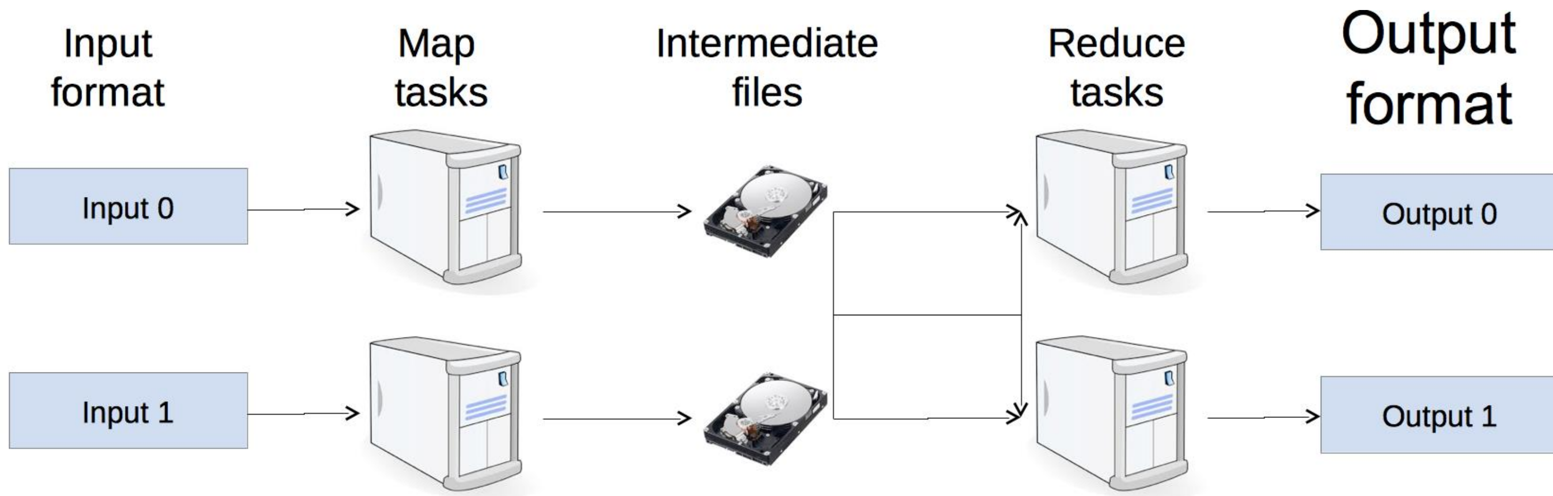
Graphs are nasty.

- Each vertex depends on its neighbours, recursively.
- Recursive problems are nicely solved iteratively.

PageRank in MapReduce

- Record: < v_i, pr, [v_j, ..., v_k] >
- Mapper: emits < v_j, pr / #neighbours >
- Reducer: sums the partial values

MapReduce DataFlow



- Each job is executed N times
- Job bootstrap
- Mappers send PR values and structure
- Extensive IO at input, shuffle & sort, output

Pregel: computational model

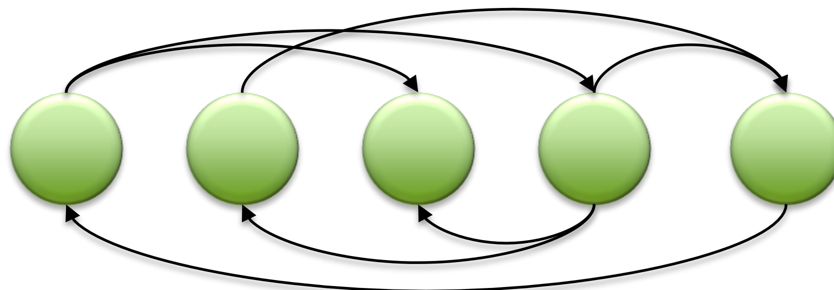
- Based on Bulk Synchronous Parallel (BSP)
  - Computational units encoded in a directed graph
  - Computation proceeds in a series of supersteps
  - Message passing architecture
- Each vertex, at each superstep:
  - Receives messages directed at it from previous superstep
  - Executes a user-defined function (modifying state)
  - Emits messages to other vertices (for the next superstep)
- Termination:
  - A vertex can choose to deactivate itself
  - Is “woken up” if new messages received
  - Computation halts when all vertices are inactive

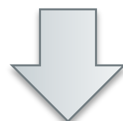
Source: Malewicz et al. (2010) Pregel: A System for Large-Scale Graph Processing. SIGMOD.

CWI
Centrum Wiskunde & Informatica

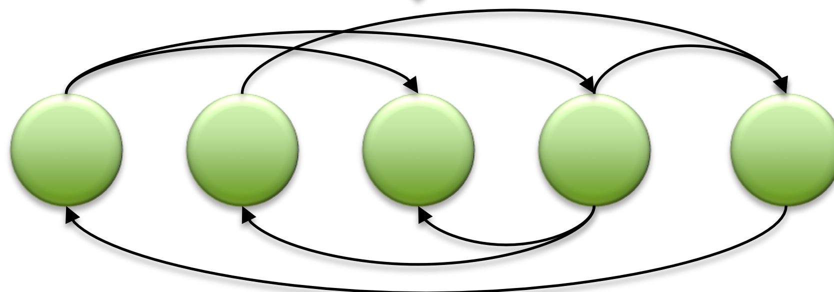
Pregel

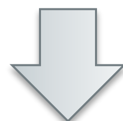
superstep *t*



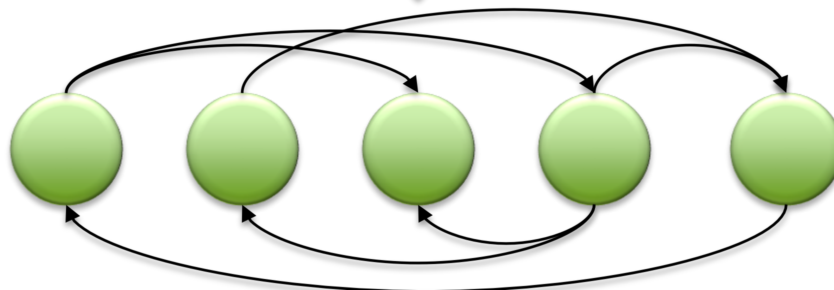


superstep *t+1*





superstep *t+2*



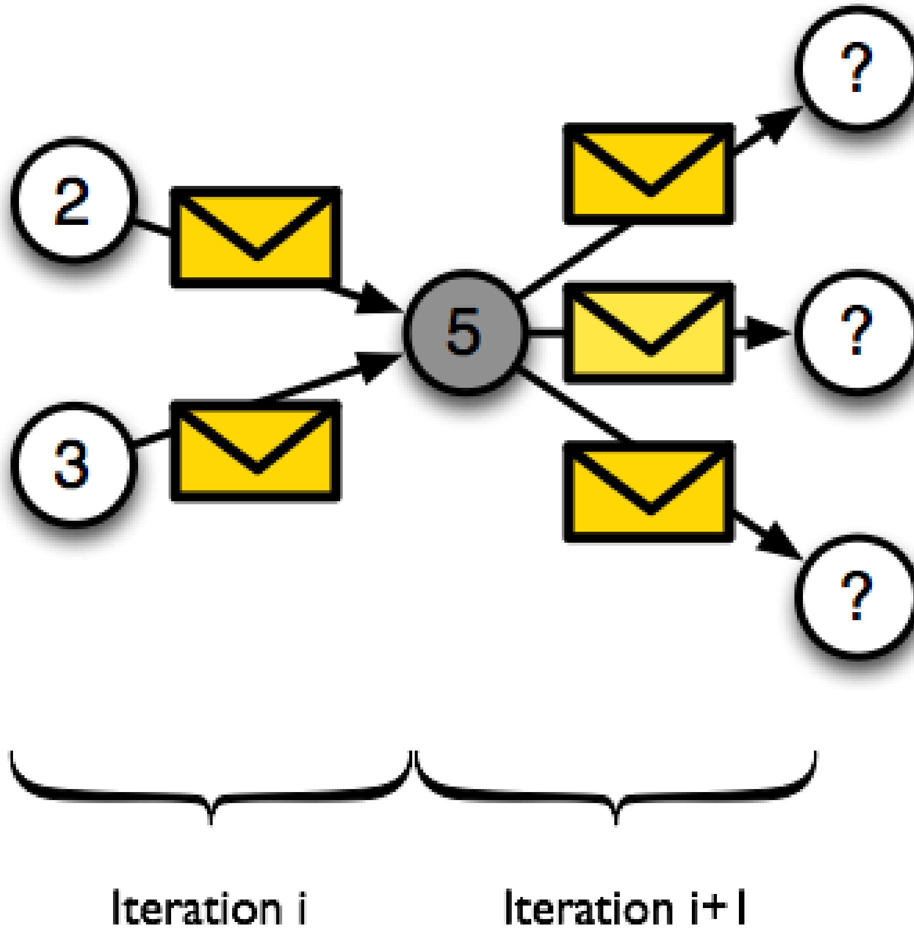
Source: Malewicz et al. (2010) Pregel: A System for Large-Scale Graph Processing. SIGMOD.

Pregel: implementation

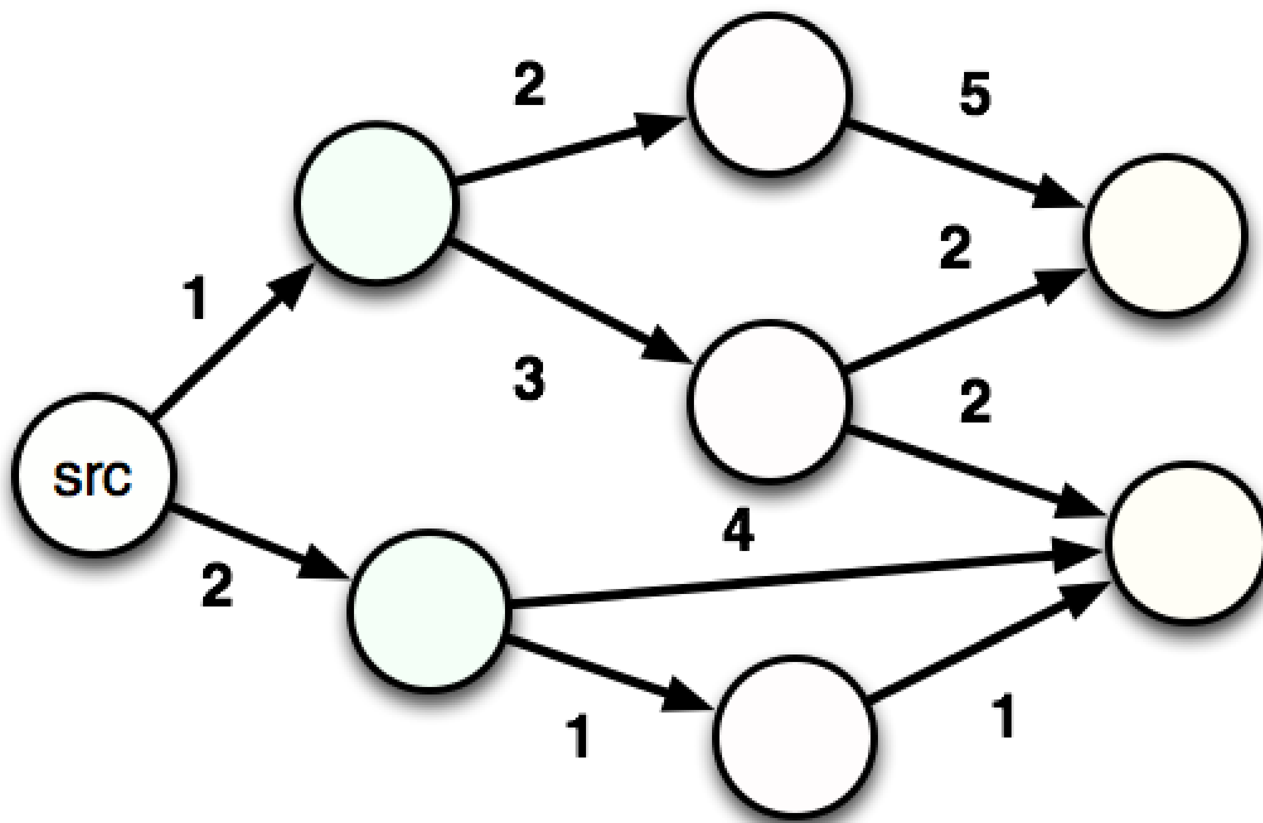
- Master-Slave architecture
  - Vertices are hash partitioned (by default) and assigned to workers
  - Everything happens in memory
- Processing cycle
  - Master tells all workers to advance a single superstep
  - Worker delivers messages from previous superstep, executing vertex computation
  - Messages sent asynchronously (in batches)
  - Worker notifies master of number of active vertices
- Fault tolerance
  - Checkpointing
  - Heartbeat/revert

Source: Malewicz et al. (2010) Pregel: A System for Large-Scale Graph Processing. SIGMOD.

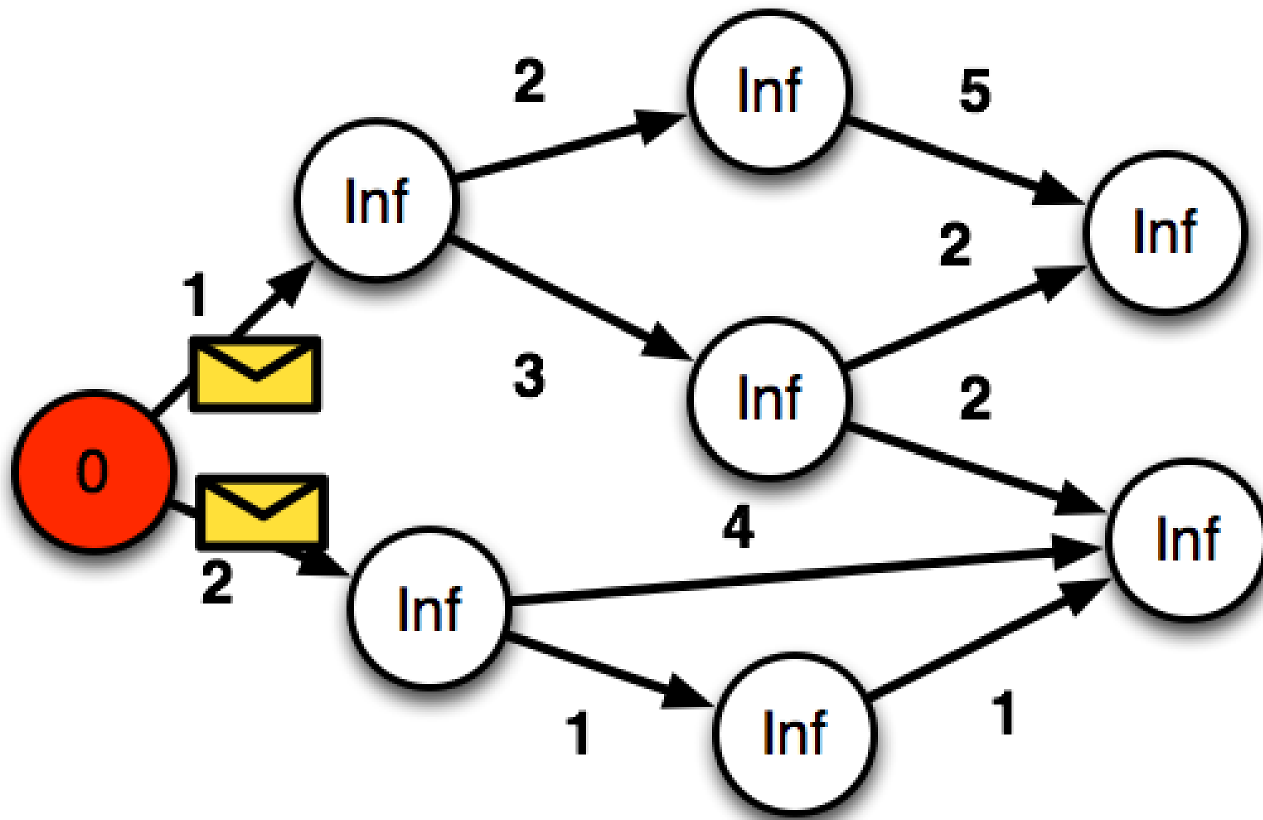
Vertex-centric API



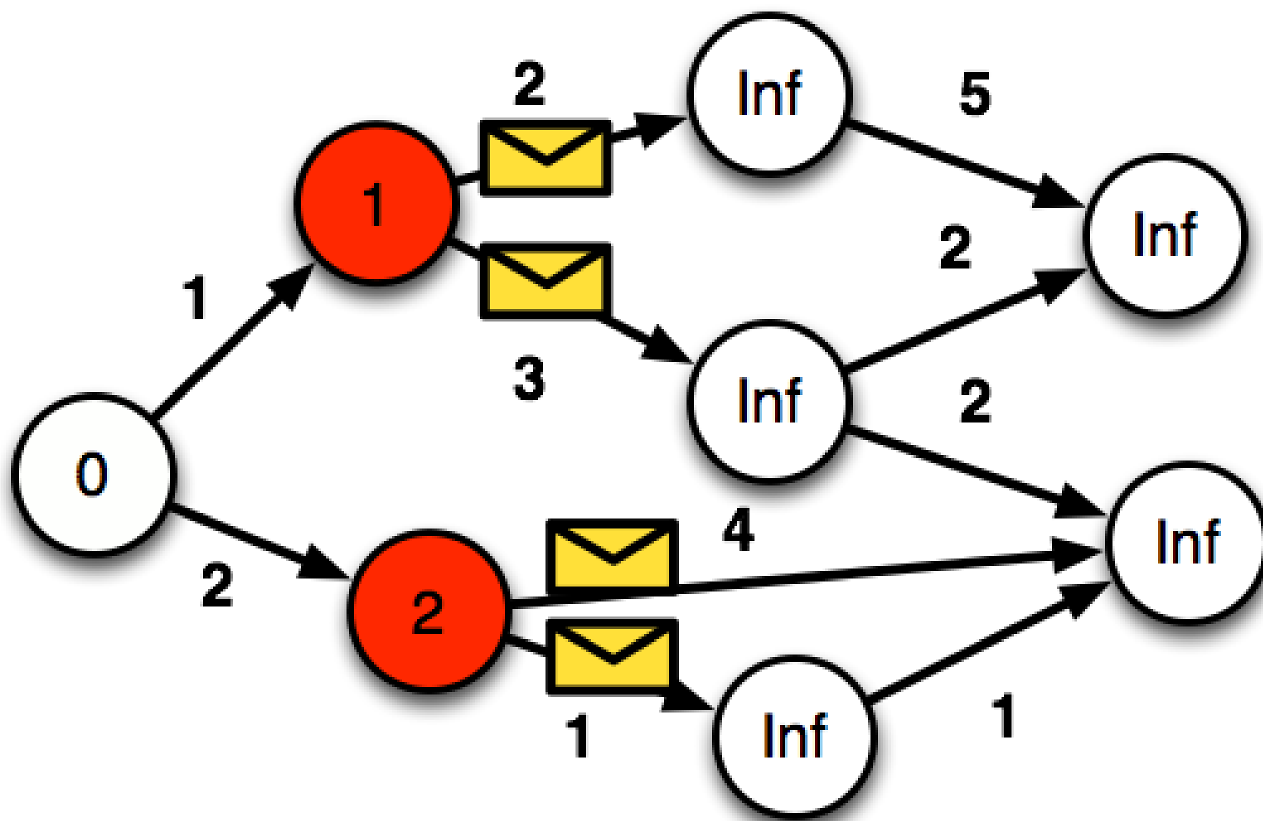
Shortest Paths



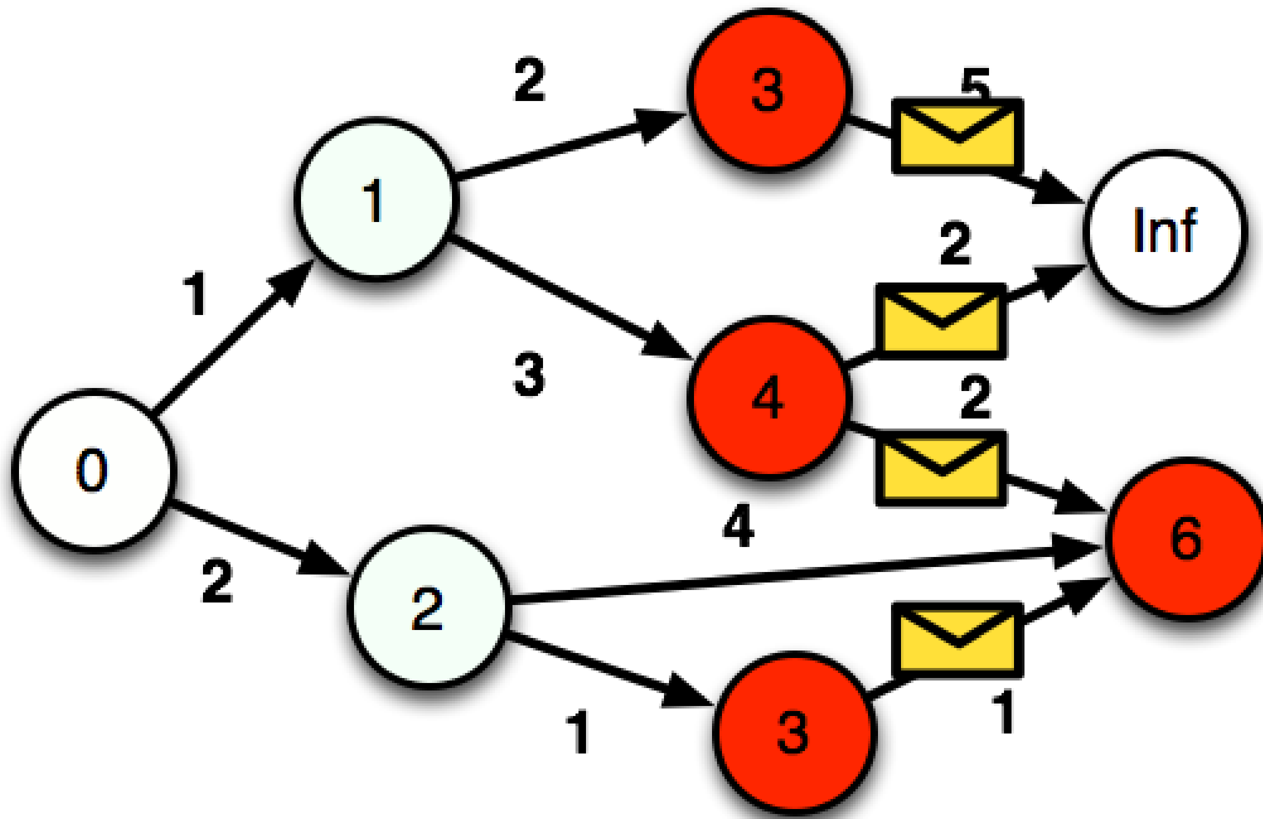
Shortest Paths



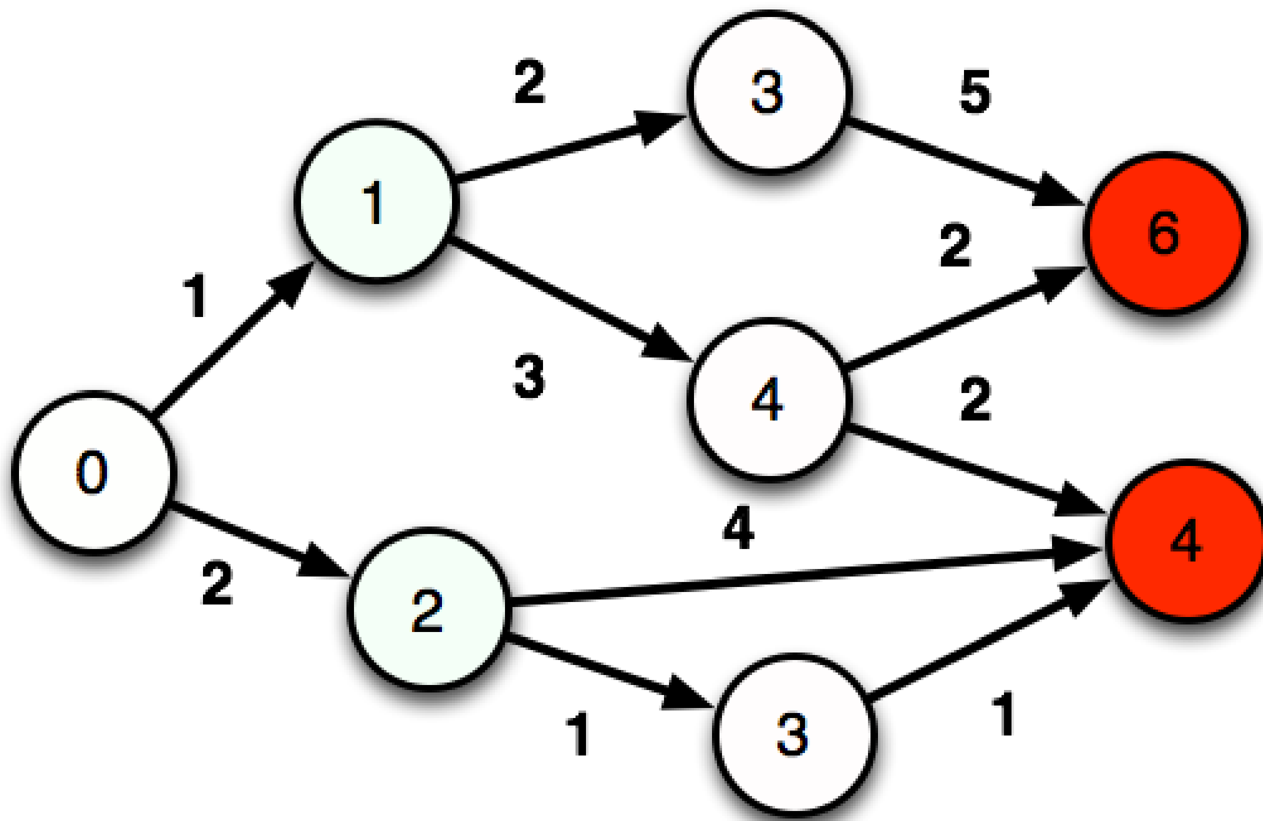
Shortest Paths



Shortest Paths



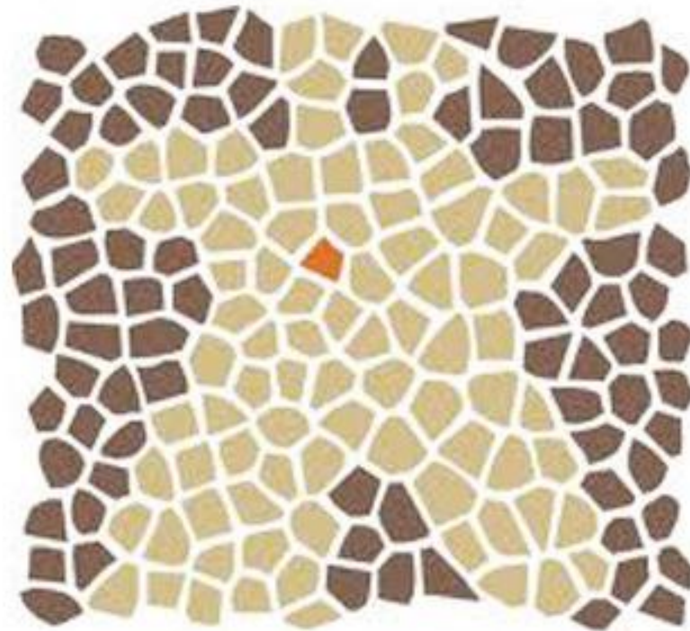
Shortest Paths



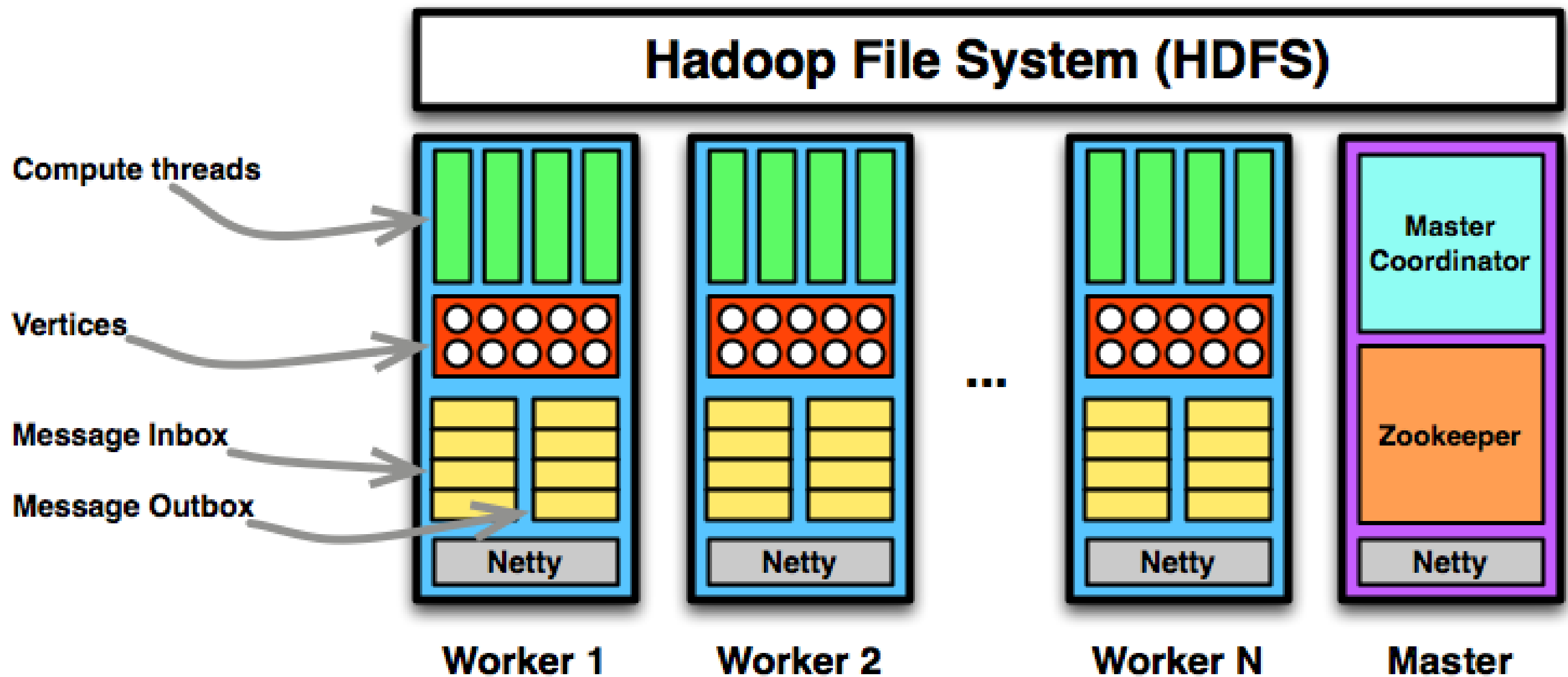
CWI
Centrum Wiskunde & Informatica

Shortest Paths

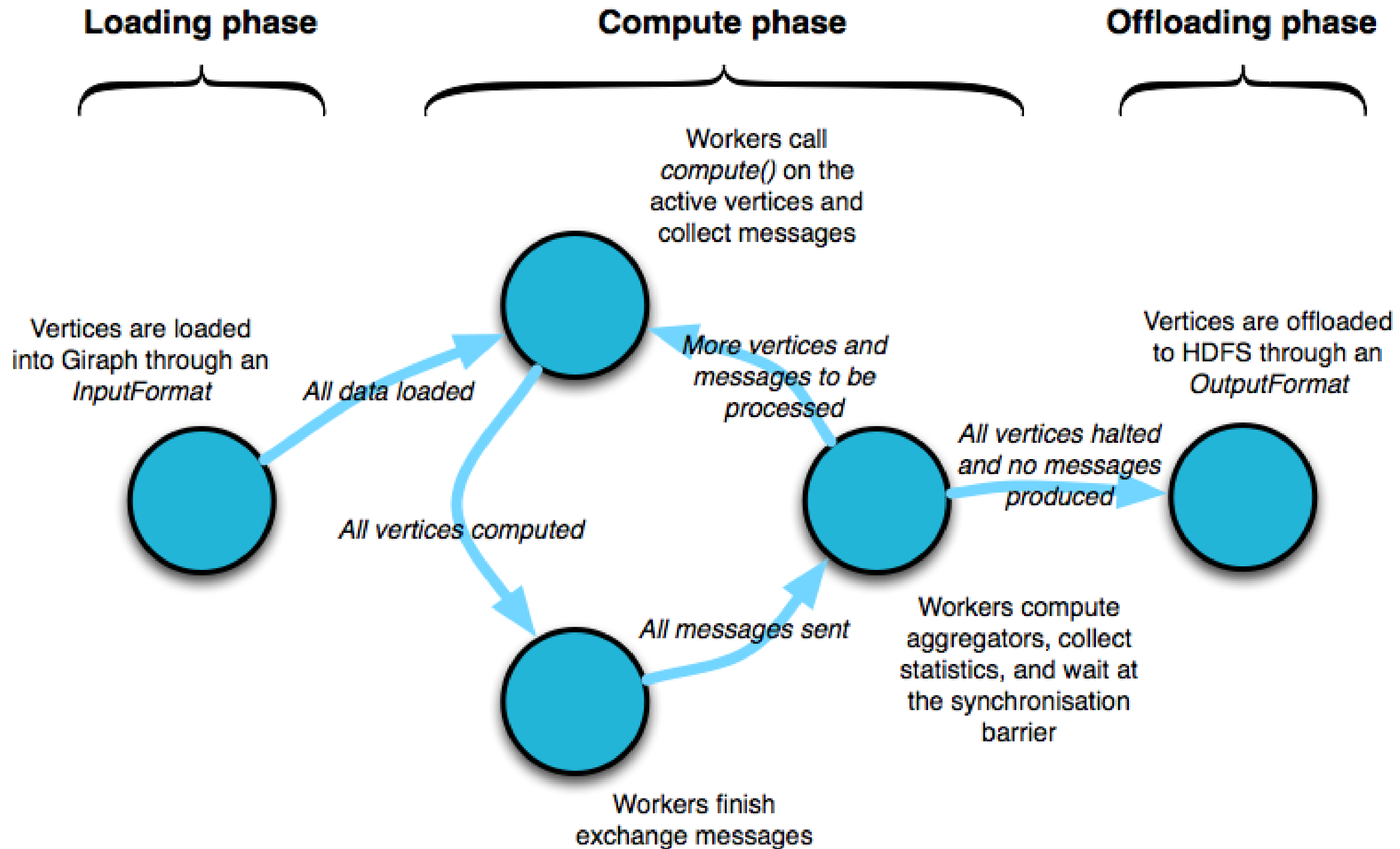
```
def compute(vertex, messages):
    minValue = Inf    # float('Inf')
    for m in messages:
        minValue = min(minValue, m)
    if minValue < vertex.getValue():
        vertex.setValue(minValue)
        for edge in vertex.getEdges():
            message = minValue + edge.getValue()
            sendMessage(edge.getTargetId(), message)
    vertex.voteToHalt()
```


APACHE
GIRAPH

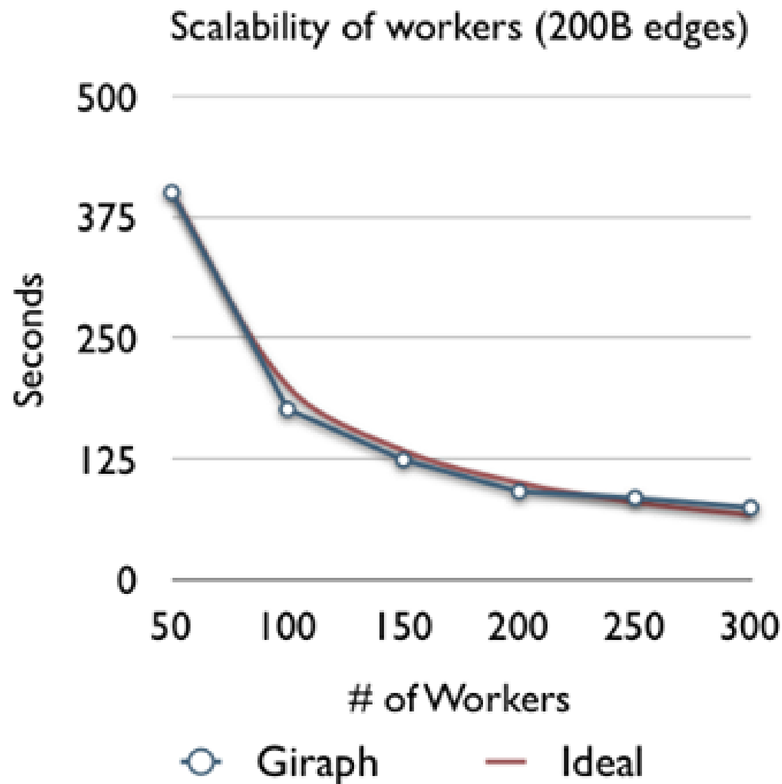
Giraph Architecture

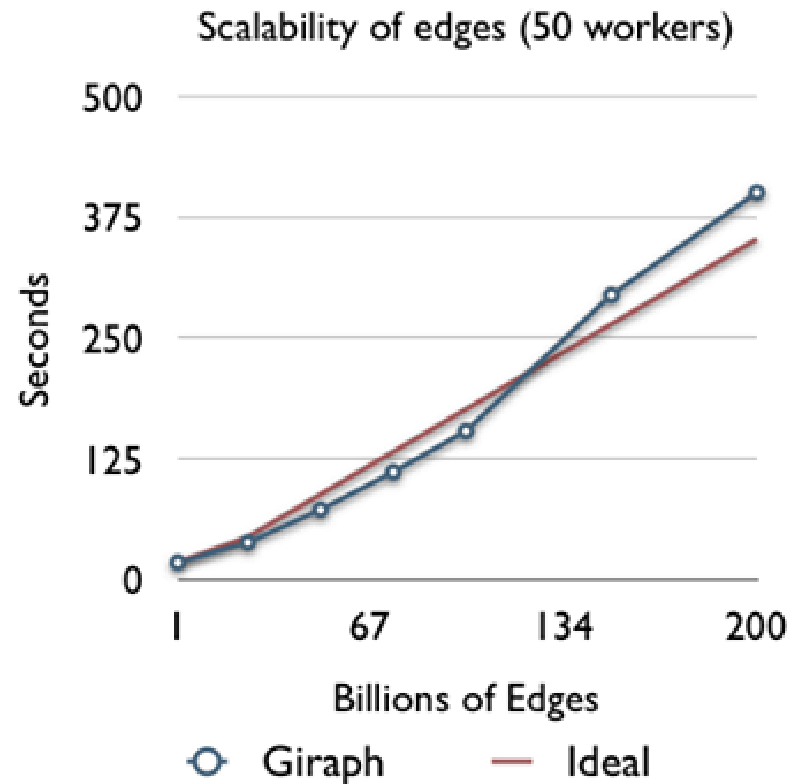


Giraph Job Lifetime



Giraph Scales





ref: https://www.facebook.com/notes/facebook-engineering/scaling-apache-giraph-to-a-trillion-edges/10151617006153920

Giraph Machine Learning: Okapi

- Apache Mahout for graphs
- Graph-based recommenders: ALS, SGD, SVD++, etc.
- Graph analytics: Graph partitioning, Community Detection, K-Core, etc.

Summary

- The Hadoop Ecosystem
  - Focused today on Pig and Giraph
  - The others will be discussed in coming sesions
- Pig
  - Higher abstraction level than MapReduce (Relational Algebra)
  - One Pig script compiles into multiple MapReduce jobs
  - Allows easy integration of User Defined Functions (UDFs)
- Giraph
  - Many analysis problems revove around graphs or networks
  - Algorithms are often iterative (multi-job) ➔ a pain in MapReduce
  - Vertex-centric programming model:
    - Who to send messages to (halt if none)
    - How to compute new vertex state from messages