INGI2145: CLOUD COMPUTING (Fall 2015)

Beyond MapReduce: Higher-level languages, Graphs

22 October 2015

Certain lecture slides adapted from Upenn NETS212 by A. Haeberlen, Z. Ives
Reproduced with permission

MapReduce: Not for Every Task

- MapReduce greatly simplified large-scale data analysis on unreliable clusters of computers
  - Brought together many traditional CS principles
    - functional primitives; master/slave; replication for fault tolerance
  - Hadoop adopted by many companies
  - Affordable large-scale batch processing for the masses

But increasingly people wanted more!

MapReduce: Not for Every Task

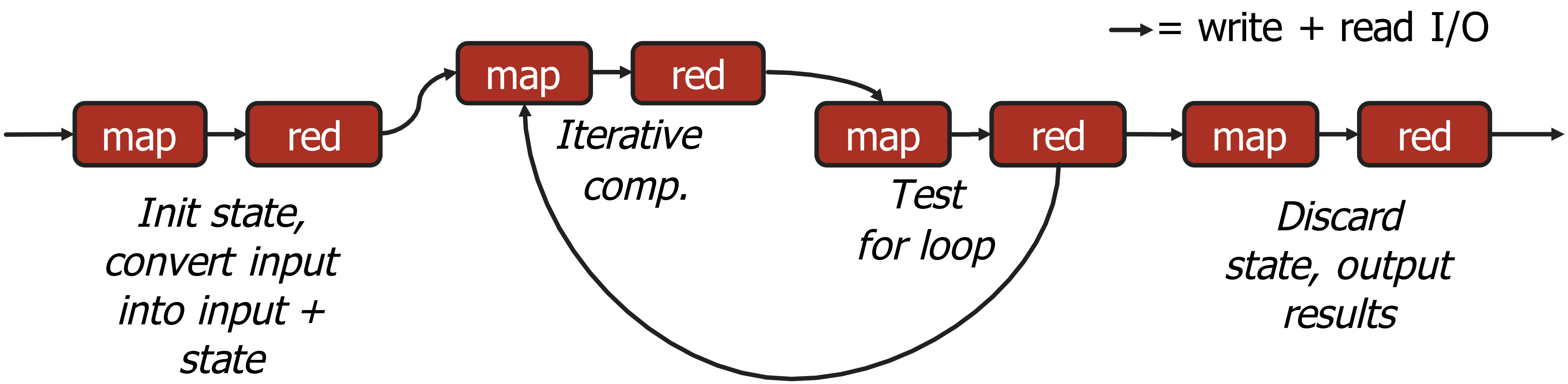
But increasingly people wanted more:

- More complex, multi-stage applications
- More interactive ad-hoc queries
- Process live data at high throughput and low latency

Which are not a good fit for MapReduce…

MapReduce for Iterative Computation

- MapReduce is essentially functional
- Expressing iterative algorithms as chains of Map/Reduce requires passing the entire state and doing a lot of network and disk I/O
  - Recall all between-stage results are materialized to reliable and distributed storage (HDFS)



MapReduce for Ad-hoc Queries

- MapReduce specifically designed for batch operations over large amounts of data
- New analysis task means writing a new MapReduce program
  - Tedious thing to do with languages such as Java
  - Programming interface is not familiar to traditional data analysts with SQL skills

- Getting results incurs development effort!

Plan for today

- Beyond MapReduce
- Higher-level languages for Hadoop
  - Hive Query Language NEXT
  - Pig and Pig Latin
- Abstractions for iterative batch-processing
  - Pregel: Bulk Synchronous Parallel for Graphs

Hive: SQL on top of Hadoop

- SQL is a higher-level language than MapReduce
  - Problem: Company may have lots of people with SQL skills, but few with Java/MapReduce skills

- Can we "bridge the gap" somehow?

```
SELECT a.campaign_id, count(*), count(DISTINCT b.user_id)
FROM dim_ads a JOIN impression_logs b ON(b.ad_id=a.ad_id)
WHERE b.dateid = '2008-12-01'
GROUP BY a.campaign_id
```

- Idea: SQL frontend for MapReduce
  - Abstract delimited files as tables (give them schemas)
  - Compile (approximately) SQL to MapReduce jobs!

Recall: Database Mgmt System

- An abstract storage system
  - Provides access to tables, organized however the database administrator and the system have chosen
- Relational data model
  - Schema formally describes fields, data types, and constraints
- A declarative processing model
  - Query language: SQL or similar
  - We describe <u>what</u> we want to store or compute, not <u>how</u> it should be done
  - More general than (single-pass) MapReduce
- A strong consistency and durability model
  - Transactions with ACID properties

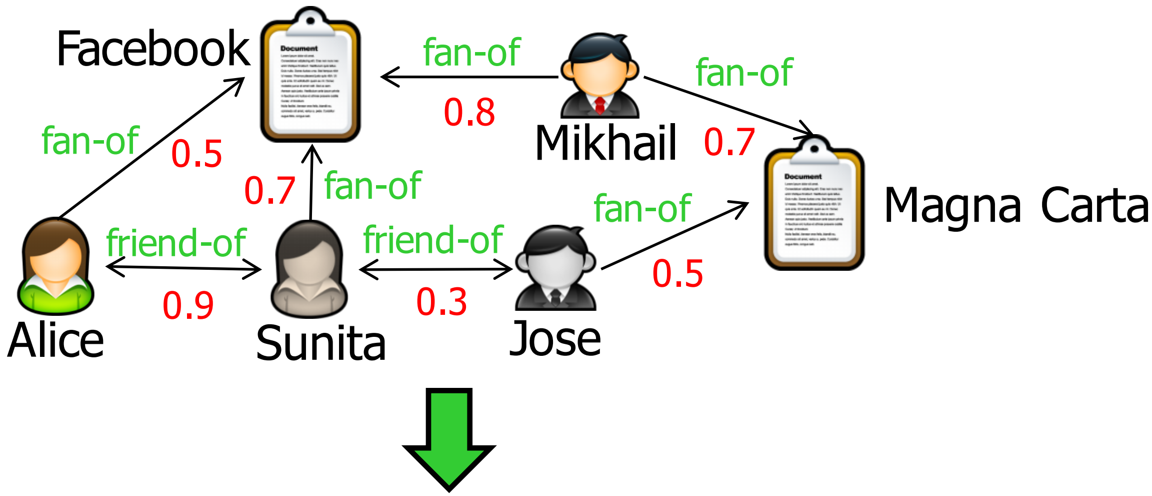
Roles of a DBMS

- Online transaction processing (OLTP)
  - Workload: Mostly updates
  - Examples: Order processing, flight reservations, banking, …
- Online analytic processing (OLAP)
  - Workload: Mostly queries
  - Aggregates data on different axes; often step towards mining
- May well have combinations of both

The database approach

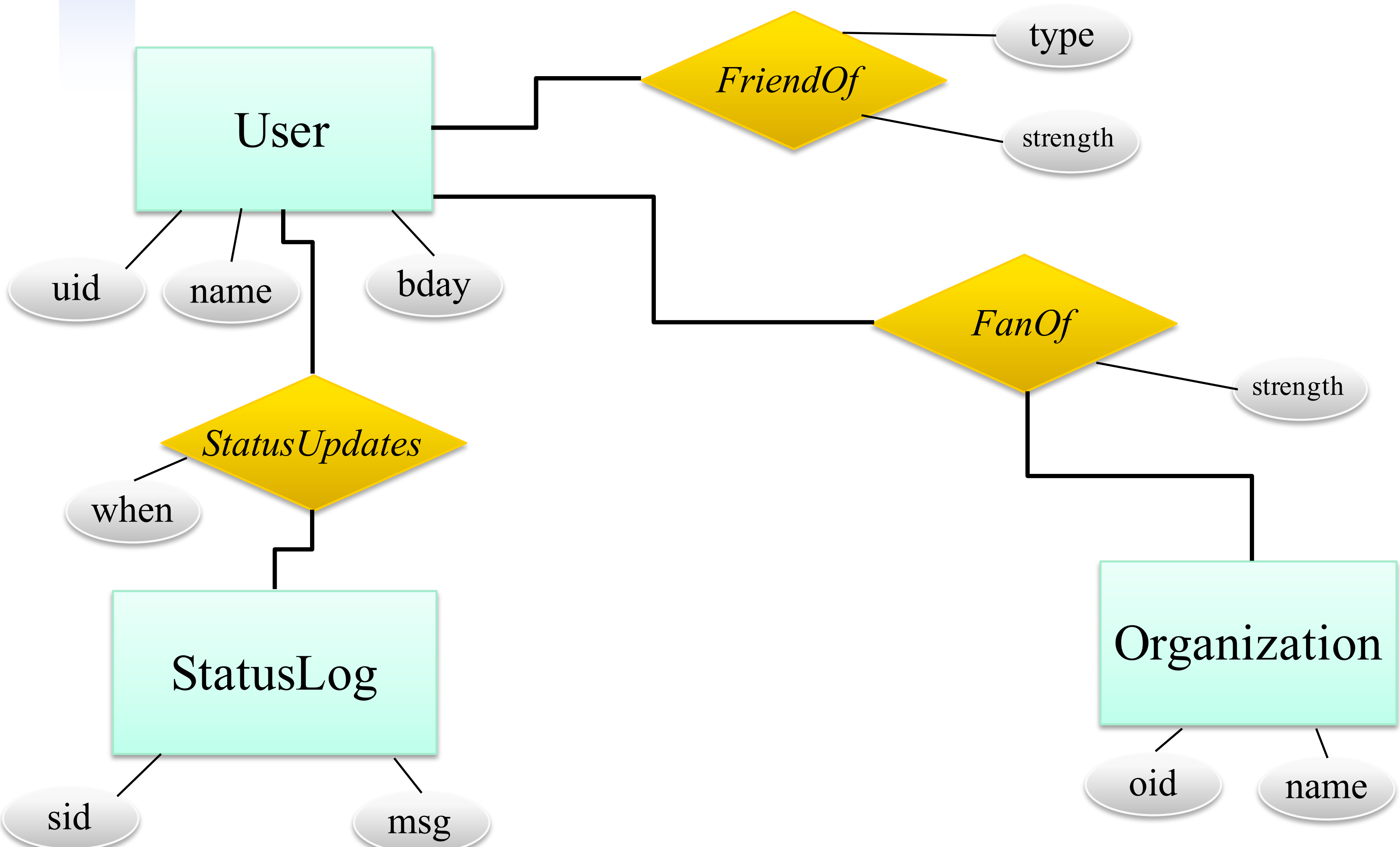
- Idea: User should work at a level close to the specification – not the implementation
  - A logical model of the data – a schema
    - Basically like class definitions, but also includes relationships, constraints
  - This will help us form a physical representation, i.e., a set of tables
    - Applications stay the same even if the platform changes
- Computations are specified as queries
  - Again, in terms of logical operations
  - Gets mapped into a query evaluation plan

Recall: Our (simplistic) social network



(Alice, fan-of, 0.5, Facebook)
(Alice, friend-of, 0.9, Sunita)
(Jose, fan-of, 0.5, Magna Carta)
(Jose, friend-of, 0.3, Sunita)
(Mikhail, fan-of, 0.8, Facebook)
(Mikhail, fan-of, 0.7, Magna Carta)
(Sunita, fan-of, 0.7, Facebook)
(Sunita, friend-of, 0.9, Alice)
(Sunita, friend-of, 0.3, Jose)

Logical schema with entity-relationship



Some example tables

User

uid	name	bdate
1	alice	1-1-80
2	jose	1-1-70
3	sunita	6-1-75

FriendOf

uid	fid	strength	type
1	3	0.9	fr
2	3	0.3	fr

StatusUpdates

uid	sid	when
1	1	10/1
2	17	11/1

FanOf

uid	oid	strength
1	99	0.5
2	100	0.5
3	99	0.7

StatusLog

sid	post
1	In Rome
17	Drank a latte

Organization

oid	name
99	Facebook
100	Magna Carta

Recap: Databases

- A more abstract view of the data
  - Schema formally describes fields, data types, and constraints
  - Relational model: Data is stored in tables
  - Declarative: We describe <u>what</u> we want to store or compute, not <u>how</u> it should be done
  - The implementation (a database management system, or DBMS) takes care of the details

- Much higher-level than MapReduce
  - This has both pros and cons

Basics of querying in SQL

- At its core, a database query language consists of manipulations of sets of tuples
  - We bind variables to the tuples within a table, perform tests on each value, and then construct an output set
  - Java:
    ```
    ArrayList<String> output …
    for (u : Table<User>) { output.add(u.name); }
    ```
  - Map/Reduce:
    ```
    public void map(LongWritable k, User v)
      { context.write(new Text(v.name)); }
    ```
  - SQL:
    ```
    SELECT U.name FROM User U
    ```

The SQL standard form

- Each block computes a set/bag of tuples
- A block looks like this:

SELECT [DISTINCT] {T_1.attrib, …, T_2.attrib}
FROM {relation} T_1, {relation} T_2, …
WHERE {predicates}

GROUP BY {T_1.attrib, …, T_2.attrib}
HAVING {predicates}
ORDER BY {T_1.attrib, …, T_2.attrib}

Multiple table variables in SQL

- Recall from a couple of slides back:

  ```
  SELECT U.name
  FROM User U
  ```

  returns (name) tuples
- We can compute all combinations of possible values (Cartesian product of tuples) as:

  ```
  SELECT U.name, U2.name
  FROM User U, User U2
  ```

- Or we can compute a union of tuples as:

  ```
  (SELECT U.name FROM User U)
  UNION
  (SELECT O.name FROM Organization O)
  ```

The basic operations

- So far, we've seen how to combine tables
- Let's see some more sophisticated operations:
  - Filtering
  - Remapping / renaming / reorganizing
  - Intersecting
  - Sorting
  - Aggregating

Filtering and remapping

- Filtering is very easy – simply add a test in the WHERE clause:

```
SELECT *
FROM User
WHERE name LIKE 'j%'
```

  (Note *, LIKE, %)

- We can also reorder, rename, and project:

```
SELECT name, uid AS id
FROM User
WHERE name LIKE 's%'
```

Intersection and join

- True intersection – "same kind" of tuples:

```
(SELECT U.name FROM User U)
INTERSECT
(SELECT O.name FROM Organization O)
```

- Join – merge tuples from different table variables when they satisfy a condition:

```
SELECT U.name, S.post
FROM User U, StatusUpdates P, StatusLog S
WHERE U.uid = P.uid AND P.sid = S.sid
```

- If the attribute names are the same:

```
SELECT U.name, S.post
FROM User U NATURAL JOIN StatusUpdates SU
NATURAL JOIN StatusLog S
```

Sorting

- Output order is arbitrary in SQL
- Unless you specifically ask for it:

```
SELECT *
FROM USER U
ORDER BY name
```

```
SELECT *
FROM USER U
ORDER BY name DESC
```

Aggregating on a key: Group By

- What if we wanted to compute the average friendship strength per organization?
  - Need to group the tuples in FanOf by 'oid', then average

- This can be done with Group By:

SELECT {group-attribs}, {aggregate-op} (attrib)
FROM {relation} T_1, {relation} T_2, ...
WHERE {predicates}
GROUP BY {group-list}

- Built-in aggregation operators:
  - AVG, COUNT, SUM, MAX, MIN
  - DISTINCT keyword for AVG, COUNT, SUM

Example: Group By

- Recall the k-means algorithm
  - Suppose we want to compute the new centroids for a set of points, and we already have the points as a table PointGroups(PointID, GroupID, X, Y)

```
SELECT P.GroupID, AVG(P.X), AVG(P.Y)
FROM PointGroups P
GROUP BY P.GroupID
```

- Can also write aggregation, e.g., in C, Java
  - Example: Oracle's Java Stored Procedures
  - Basically like the Reduce function!

Composition

- The results of SQL are tables
  - Hence you can query the results of a query!

- Let's do k-means in SQL:

```
SELECT PG.GroupID, AVG(PG.X), AVG(PG.Y)
FROM (
  SELECT P.ID, P.X, P.Y,
     ARGMIN(dist(P.X, P.Y, G.X, G.Y), G.ID),
     MIN(dist(P.X, P.Y, G.X, G.Y))
  FROM POINTS P, GROUPS G
  GROUP BY P.ID
) AS PG
GROUP BY PG.GroupID
```

Recap: Querying with SQL

- We have seen SQL constructs for:
  - Projection and remapping/renaming (SELECT)
  - Cartesian product (FROM x, y, z, …; NATURAL JOIN)
  - Filtering (WHERE)
  - Set operations (UNION, INTERSECT)
  - Aggregation (GROUP BY + MIN, MAX, AVG, …)
  - Sorting (ORDER BY)
  - Composition (SELECT … FROM (SELECT … FROM …))
- Not a complete list - SQL has more features!

Hive

- A data warehouse infrastructure built on top of Hadoop for providing data summarization, query and analysis
- Hive Query Language (HQL) – similar to SQL
  - Suitable for processing structured data
  - Create a table structure on top of HDFS
  - Queries are compiled in to MapReduce jobs
- Not designed for OLTP!
  - Updating records or transactions are not supported

Example: WordCount

```
CREATE TABLE doc (line STRING);
LOAD DATA LOCAL INPATH 'text.txt' INTO TABLE doc;

CREATE TABLE wordcount AS
SELECT word, count(1) AS count
FROM (SELECT EXPLODE(SPLIT(line, '\s')) AS word FROM doc) words
GROUP BY word
ORDER BY count DESC, word ASC;
```

Plan for today

- Beyond MapReduce
- Higher-level languages for Hadoop
  - Hive Query Language
  - Pig and Pig Latin

    NEXT
- Abstractions for iterative batch-processing
  - Pregel: Bulk Synchronous Parallel for Graphs

Towards Pig #1: Beyond relations?

- The relational data model allows us to have arbitrary numbers of relations
  - Each with its own schema that includes arbitrary numbers of attributes
- But: No nested tables!
  - These would be converted into multiple tables by 1NF normalization
  - Hence SQL has no nested collections at all, (sets, lists, bags...)
- Can we add support for these?

Towards Pig #2: Programming model

- Hadoop MapReduce:
  - file-oriented, procedural
  - regularized "pipeline" – map, combine, shuffle, reduce
  - arbitrary Java functions at each step
- SQL:
  - random access-storage-oriented (DBMS controls storage)
  - compositional, tuple-collection-oriented query model
  - declarative queries are automatically optimized
  - can accommodate Java functions, but not naturally
  - Hive: SQL queries → file-oriented Map/Reduce
- Is there something in between?
  - Declarative is nice, but many data analysts are 'entrenched' procedural programmers...
  - Pig and Pig Latin!

rigid dataflow

opacity

custom code even for very common operations

what about "procedural programmers"?

Pig Latin and Pig

- Pig Latin: a compositional, collections-oriented dataflow language
  - Oriented towards parallel data processing & analysis
  - Think of it as a more procedural SQL-like language with nested collections
    - Emphasizes user-defined functions, esp. those that have nice algebraic properties (unlike SQL)
    - Supports external data from files (like Hive)
  - By Chris Olston et al. at Yahoo! Research
    - http://www.tomkinshome.com/site_media/papers/papers/ORS+08.pdf
- Pig: the runtime system

Pig Latin: Basic constructs

- Collection-valued expressions whose results get assigned to variables
  - A program does a series of assignments in a dataflow
  - It gets compiled down to a sequence of MapReduces
    - Similar to Hive, but Pig Latin has its own query language (not SQL)

- Basic SQL-like operations are explicitly specified:
  - `load … as` [HDFS scan]
  - Remapping: `foreach … generate` [Map]
  - Filtering: `filter by` [Map]
  - Intersecting: `join` [Reduce]
  - Aggregating: `group by` [Reduce]
  - Sorting: `order` [Shuffle]
  - `store` [HDFS store]

Simple example: Face detection

- Each expression creates a named collection
  - load collections from files
  - process them (e.g., per tuple) using a user-defined function
  - store the results into files

```
I = load '/mydata/images' using ImageParser()
    as (id, image);
F = foreach I generate id, detectFaces(image);
store F into '/mydata/faces';
```

Example: Session Classification

- Goal: Find web sessions that end on the 'best' page (i.e., the page with the highest PageRank)
  - We need to join two tables, and then compare the final rank in the sequence to the other ranks

Visits

User	URL	Time
Alice	www.cnn.com	7:00
Alice	www.digg.com	7:20
Alice	www.social.com	10:00
Alice	www.flickr.com	10:05
Joe	www.cnn.com/index.htm	12:00

⋮

Pages

URL	PageRank
www.cnn.com	0.9
www.flickr.com	0.9
www.social.com	0.7
www.digg.com	0.2

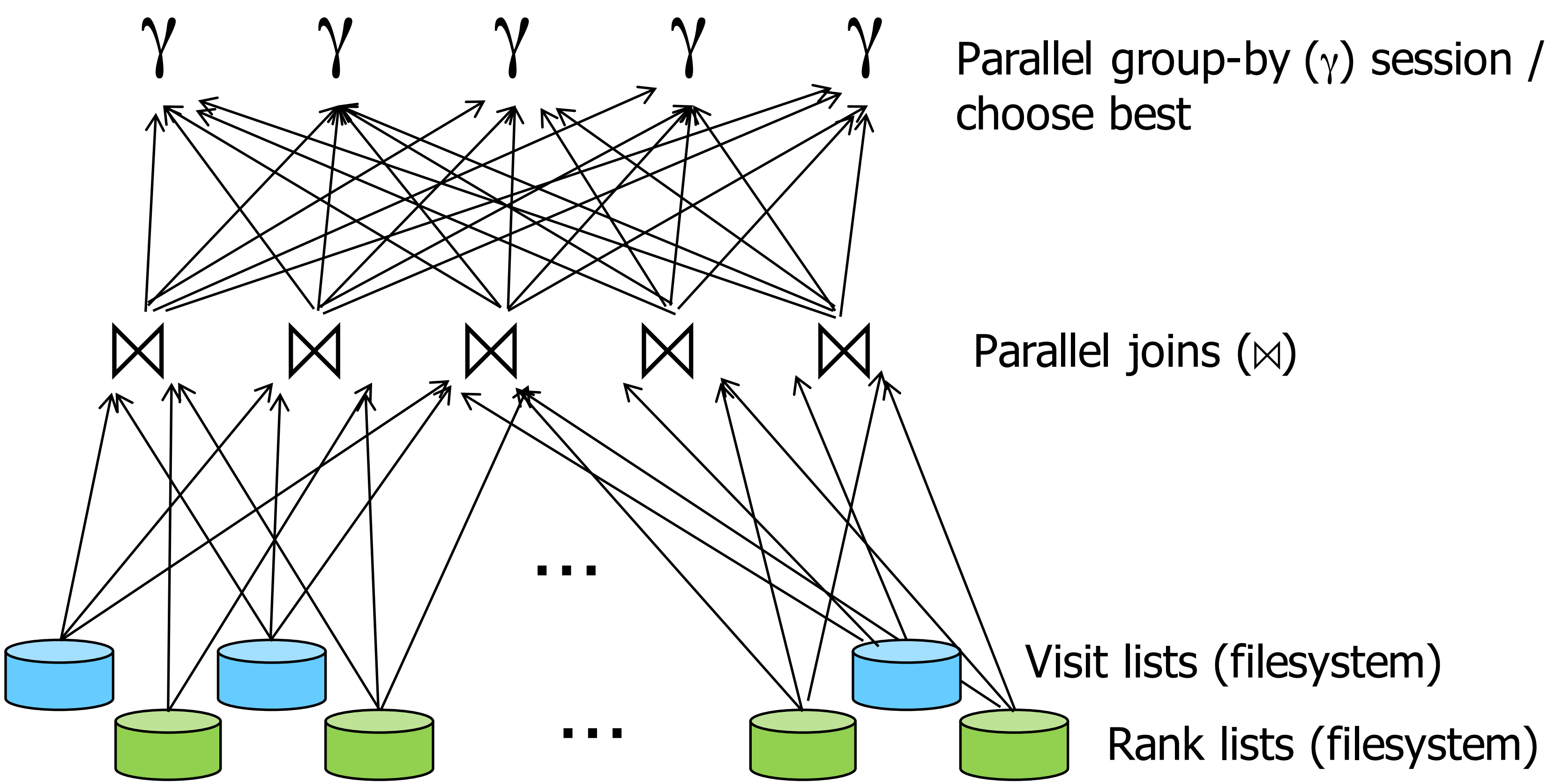
⋮

The computation in Pig Latin

```
Visits = load '/data/visits' as (user, url, time);
Visits = foreach Visits generate user, Canonicalize(url),
  time;

Pages = load '/data/pages' as (url, pagerank);

         VP = join Visits by url, Pages by url;
 UserVisits = group VP by user;
   Sessions = foreach UserVisits generate
                flatten(FindSessions(*));
HappyEndings = filter Sessions by BestIsLast(*);

       store HappyEndings into '/data/happy_endings';
```

What does this query compile to?

- Parallel evaluation is really a Map-Map/Reduce/Reduce chain:



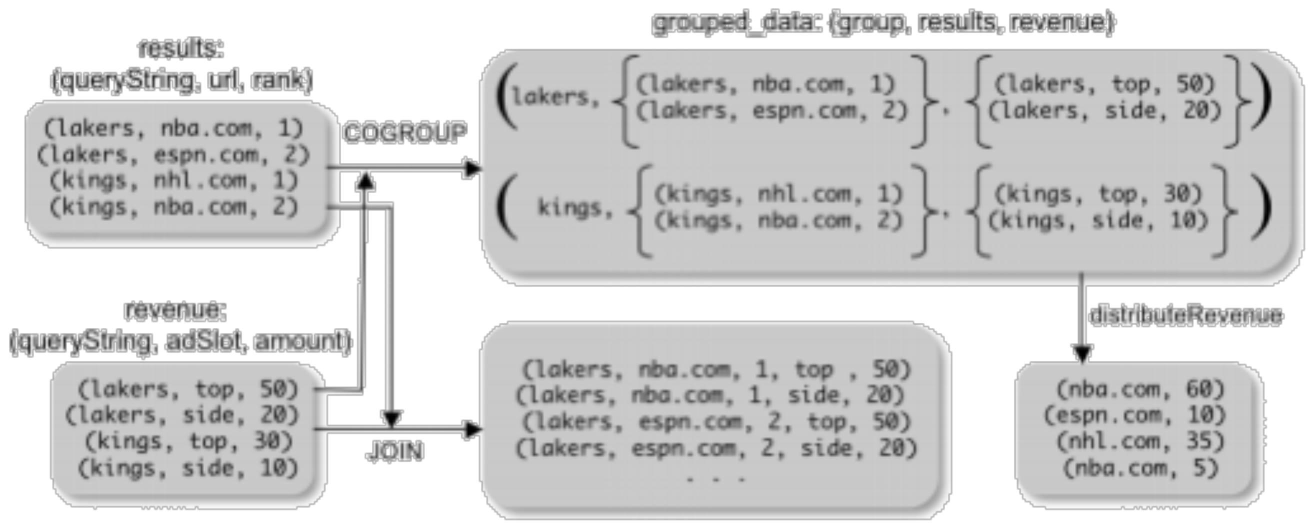
Pig Latin features

- Record-oriented transformations
  - Can work over, create nested collections
  - (Resembles Nested Relational variants of SQL)
- Basic operators expose parallelism; user-defined operators may not
- Operations are explicit, not declarative
  - Unlike SQL

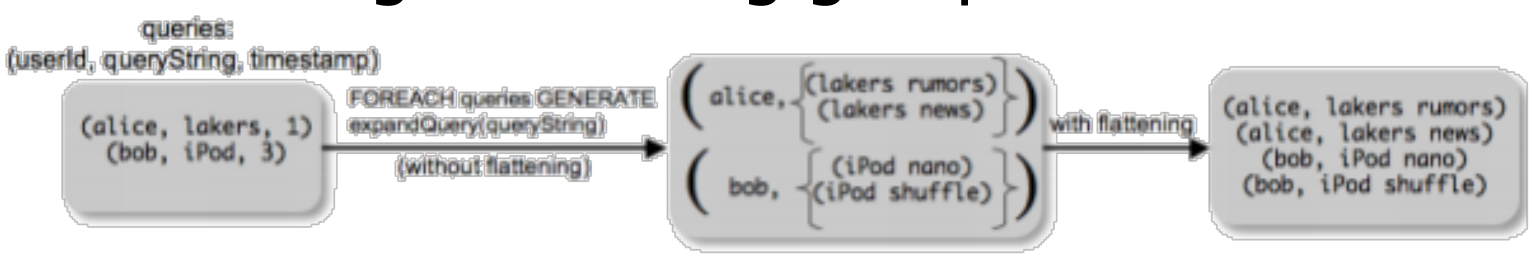
operators:	binary operators:
• FILTER	• JOIN
• FOREACH … GENERATE	• COGROUP
• GROUP	• UNION

Nesting: COGROUP & FLATTEN

- Cogrouping: nesting groups into columns



- Flattening: unnesting groups



Pig Latin vs. MapReduce

- MapReduce combines 3 primitives:
  process records → create groups → process groups

```
a = FOREACH input GENERATE flatten(Map(*));
b = GROUP a BY $0;
c = FOREACH b GENERATE Reduce(*);
```

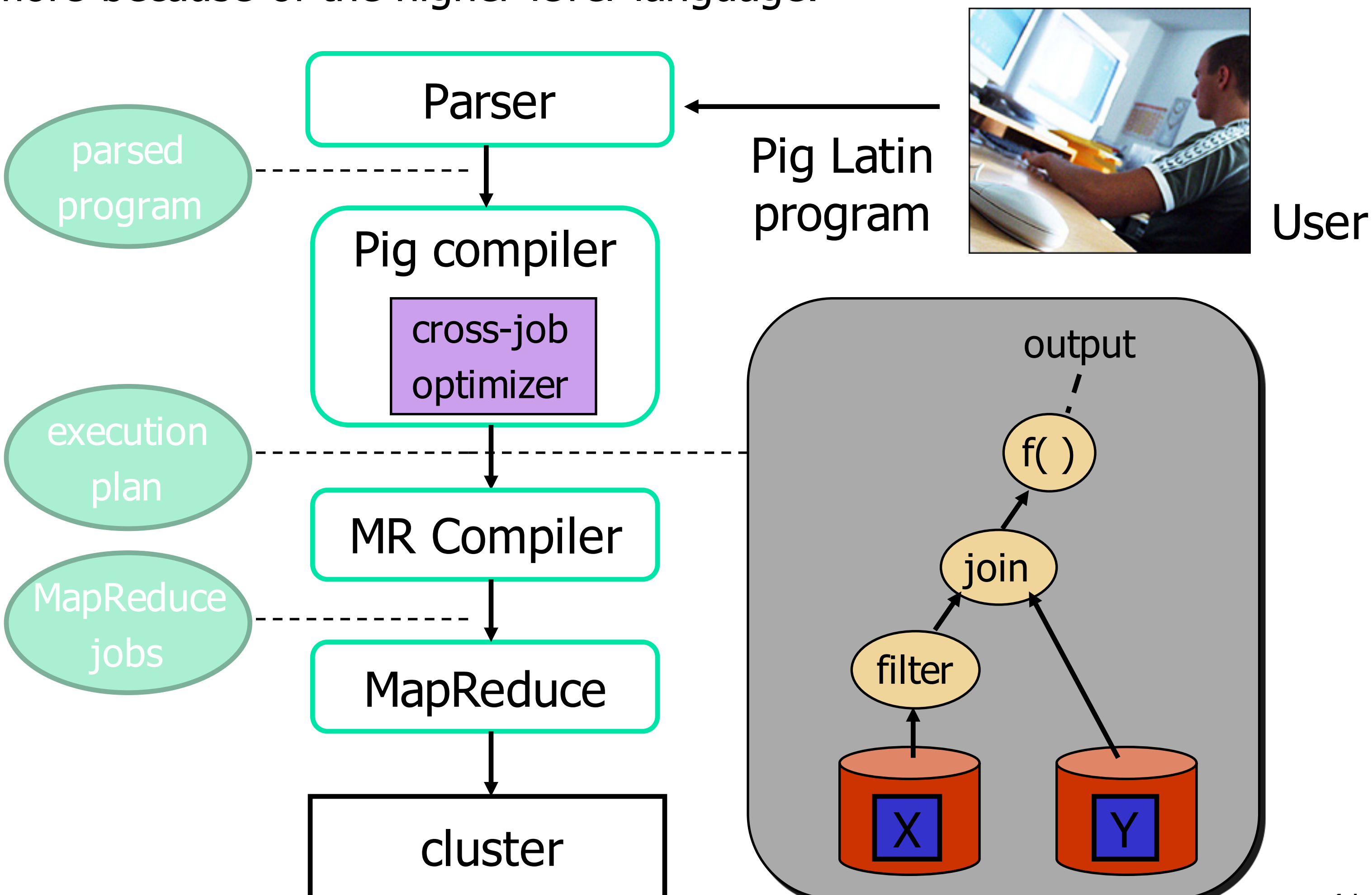
- In Pig, these primitives are:
  - explicit
  - independent
  - fully composable
- Pig adds primitives for:
  - filtering tables
  - projecting tables
  - combining 2 or more tables

Recap: Pig Latin

- A dataflow language that compiles to MapReduce
  - Borrows many of the elements of SQL, but eliminates the reliance on declarative optimization
  - Incorporates primitives for nested collections
- Quite successful:
  - As of 2008: 25% of Yahoo Map/Reduce jobs from Pig
  - Part of the Hadoop standard distribution

Pig system implementation

- Let's briefly look at the Pig implementation, and how it can do a bit more because of the higher-level language:



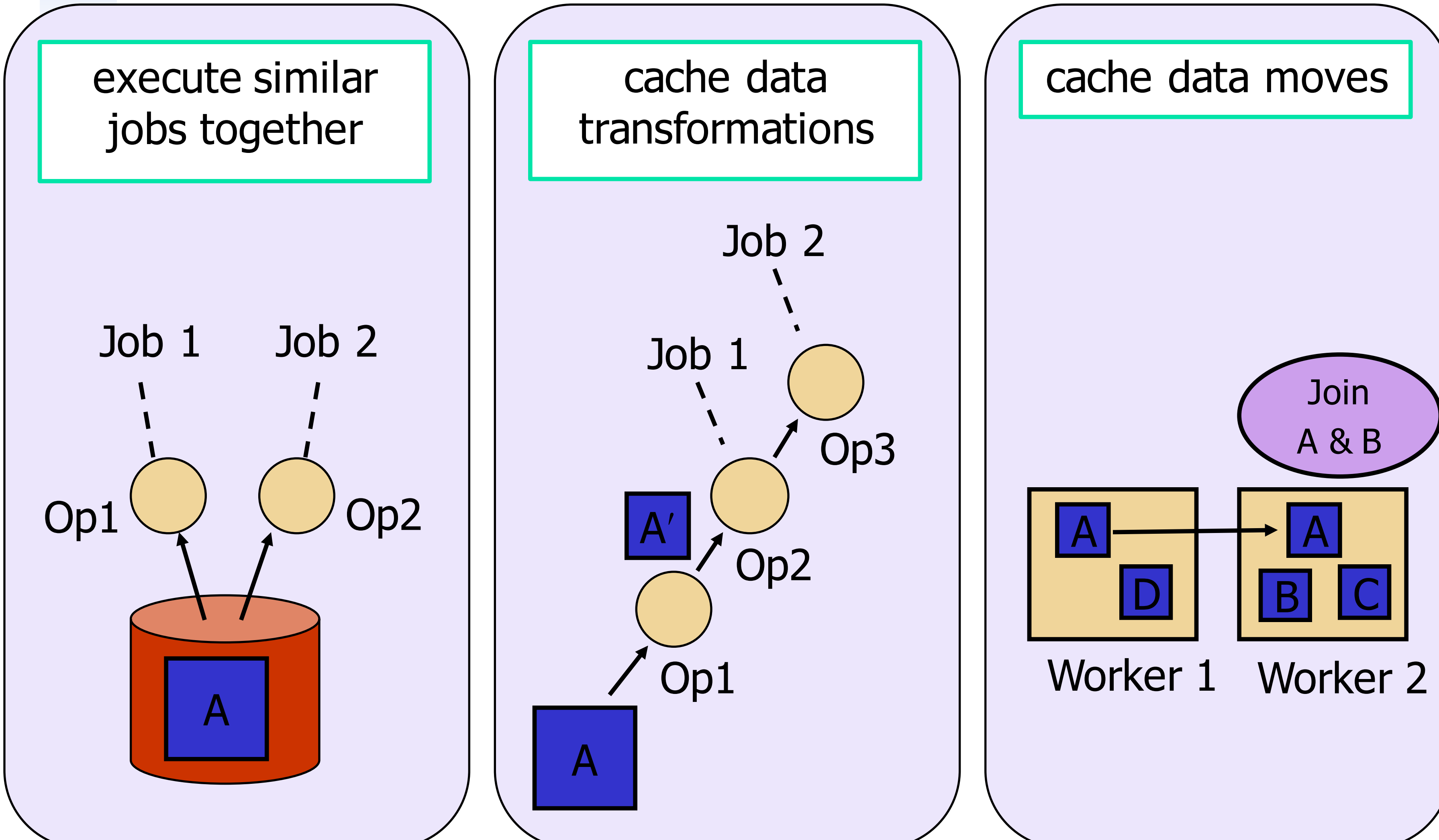
Key issue: Minimizing redundancy

- Popular tables
  - web crawl
  - search log

- Popular transformations
  - eliminate spam pages
  - group pages by host
  - join web crawl with search log

- Goal: Minimize redundant work

Work-sharing techniques



Recap: Pig and Pig Latin

- Somewhere between a programming language and a DBMS
- Allows distributed programming with explicit parallel dataflow operators
- Supports explicit management of nested collections
- Runtime system does caching and batching

Plan for today

- Beyond MapReduce ✓
- Higher-level languages for Hadoop
  - Hive Query Language ✓
  - Pig and Pig Latin ✓
- Abstractions for iterative batch-processing ← NEXT
  - Pregel: Bulk Synchronous Parallel for Graphs

New Abstractions Needed

Much of the mismatch stems from the lack of shared global state

Complex applications and interactive queries both need one thing that MapReduce lacks

- Efficient primitives for data sharing

What If We Could Remember?

Suppose we were to change things entirely:

- A set of machines
- … each with a *partition* of a dataset, stored in memory
- Computation consists of *sending updates* from one portion to another

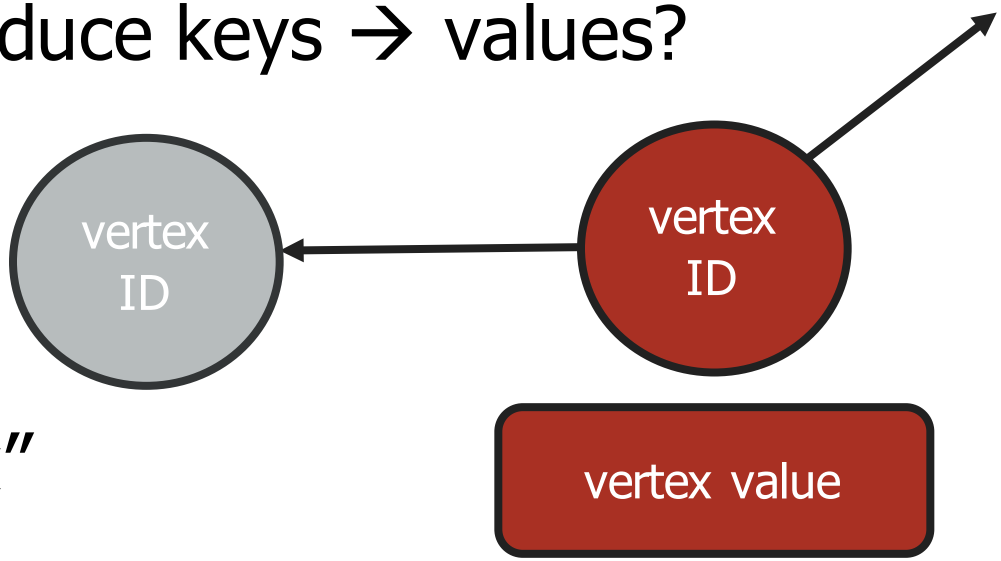
Let's look at two versions of this

Pregel: Bulk Synchronous Parallel

Let's slightly rethink the MapReduce model for processing **graphs**

- Vertices
- "Edges" are really messages

Compare to MapReduce keys → values?



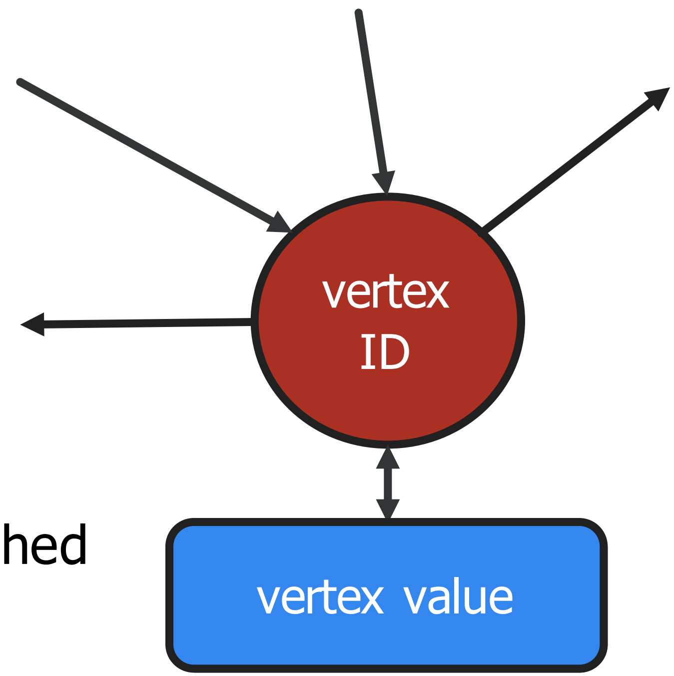
"Think like a vertex"

The Basic Pregel Execution Model

A sequence of *supersteps*, for each vertex V

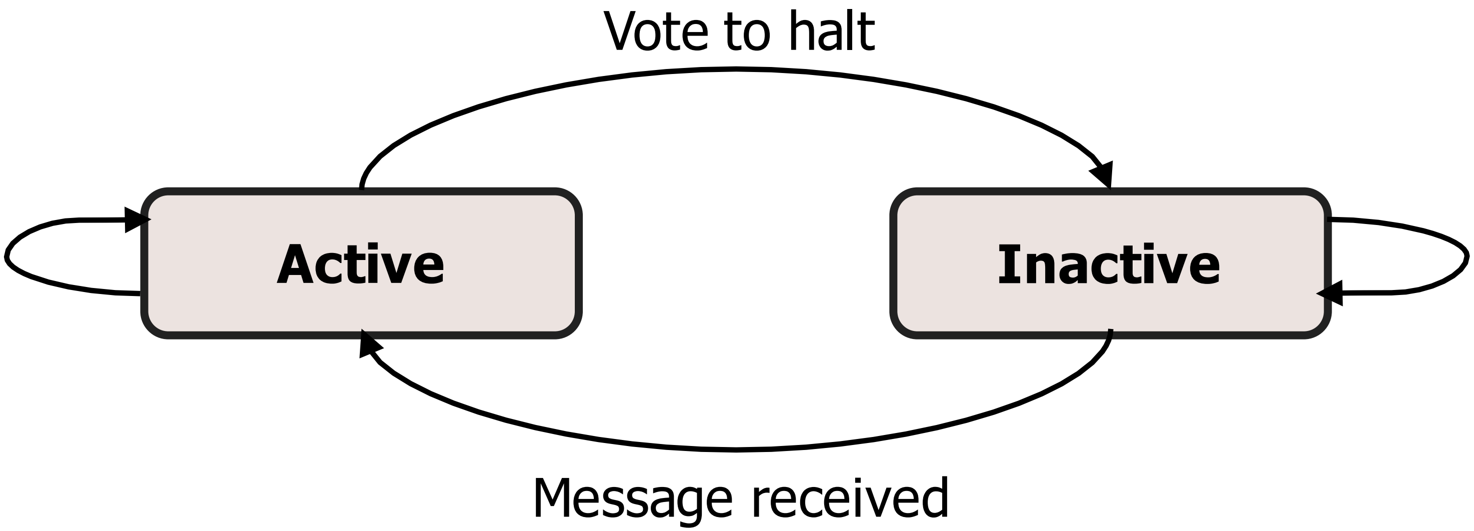
At superstep S:

- Compute in parallel at each V
  - Read messages sent to V in superstep S-1
  - Update value / state
  - Optionally change topology
- Send messages
- Synchronization
  - Wait till all communication is finished

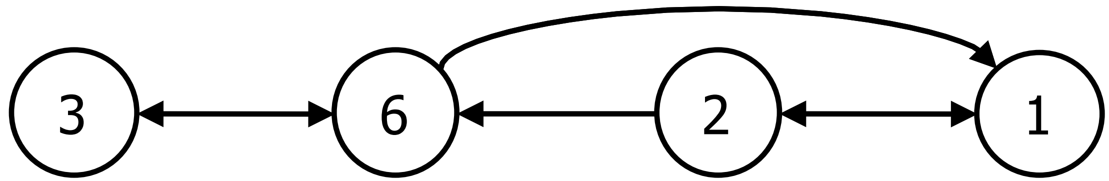


Termination Test

- Based on every vertex voting to halt
  - Once a vertex deactivates itself it does no further work unless triggered externally by receiving a message
- Algorithm terminates when all vertices are simultaneously inactive

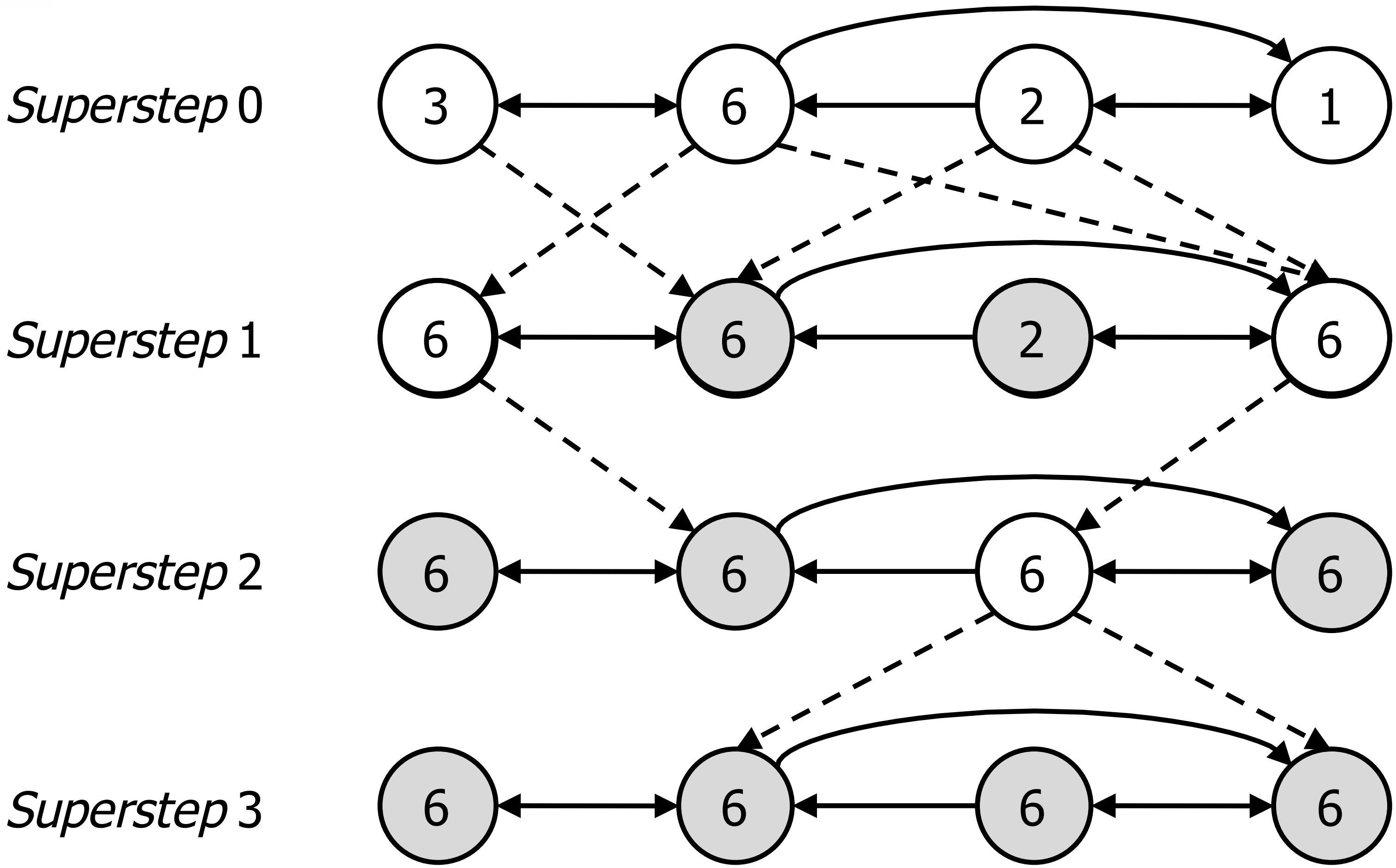


Example: Find Maximum Value



- Each vertex contains an integer value
- Idea: propagate the largest value to every vertex
  - At superstep 0, start by propagating value to all neighbors
  - In each superstep, any vertex that has learned a larger value from its messages sends it to all its neighbors; otherwise vote to halt
  - Terminates when no further vertices change in a superstep

Example: Find Maximum Value



Recall: PageRank

- Each page i is given a rank x_i
- Goal: Assign the x_i such that the rank of each page is governed by the ranks of the pages linking to it:

$$x_i = \sum_{j \in B_i} \frac{1}{N_j} x_j$$

Rank of page i

Every page j that links to i

Number of links out from page j

Rank of page j

How do we compute the rank values?

Naïve PageRank Algorithm Restated

- Let
  - N(p) = number outgoing links from page p
  - B(p) = number of back-links to page p

$$PageRank(p) = \sum_{b \in B(p)} \frac{1}{N(b)} PageRank(b)$$

  - Each page b distributes its importance to all of the pages it points to (so we scale by 1/N(b))
  - Page p's importance is increased by the importance of its back set
  - Iterate till convergence or some number of iterations

PageRank in Pregel

```
void Compute(messages) {
 if (superstep() >= 1) {
    sum = 0;
    foreach (msg in messages)
      sum += msg->Value();
    value = 0.15 / NumVertices() + 0.85 * sum;
    SetValue(value);
  }
  if (superstep() < 30) {
    n = GetNumOfOutEdges();
    SendMessageToAllNeighbors(GetValue() / n);
  } else {
    VoteToHalt();
  }
}
```

Pregel Summary

- Bulk Synchronous Parallel – sequence of synchronized supersteps
  - Abstraction originally invented by Leslie Valliant in the '80s
- Consider the nodes to have state (memory) that carries from superstep to superstep
- Connections to MapReduce model?
- See also Apache Hama, Giraph, Graph.lab

Stay tuned

http://www.flickr.com/photos/3dking/2573905313/sizes/l/in/photostream/

Next time you will learn about:

Beyond MapReduce – In-memory processing, Streaming