Data Analytics

Dan Ports, CSEP 552

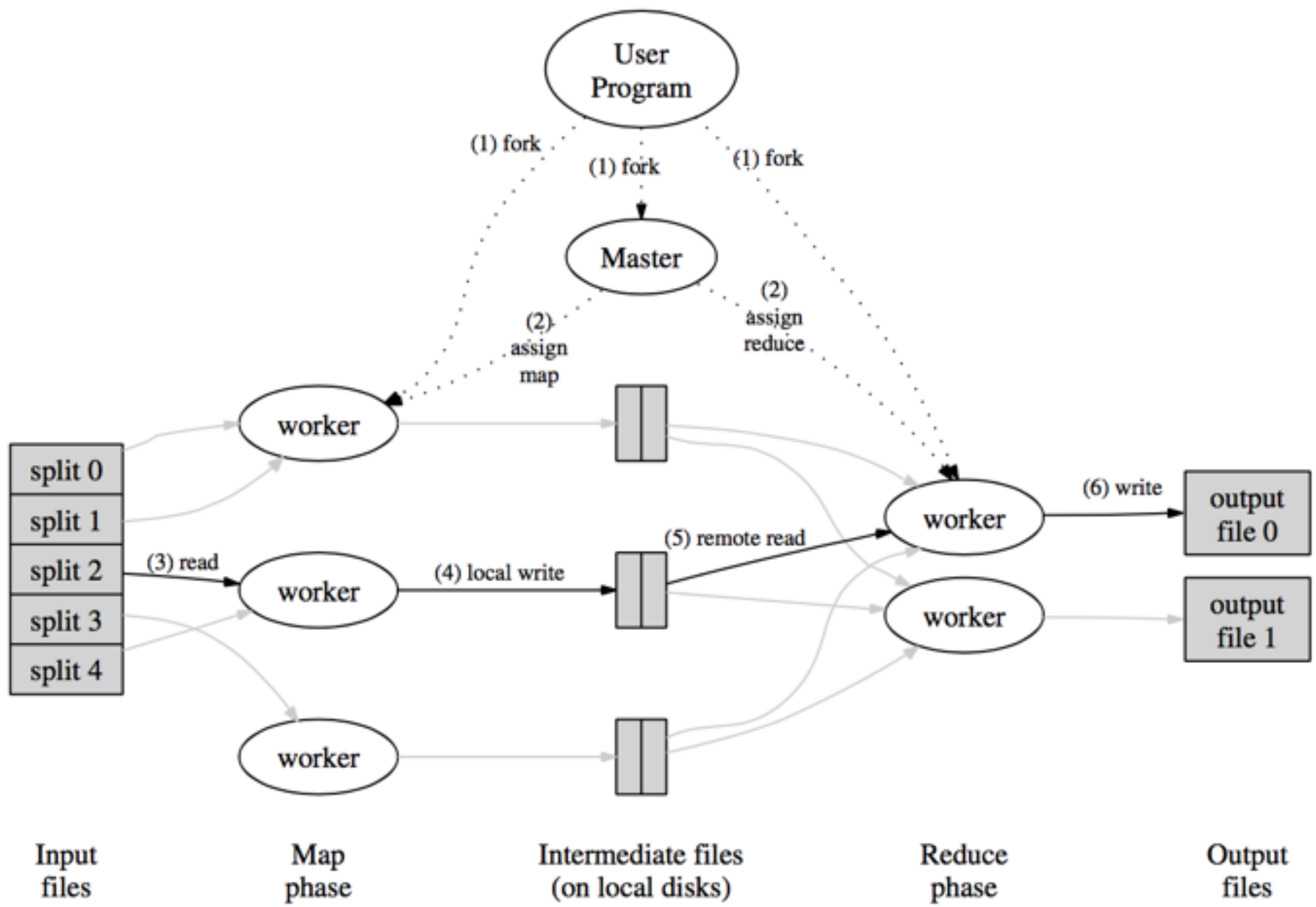
Today

- MapReduce
    - is it a major step backwards?
    - beyond MapReduce: Dryad
- Other data analytics systems:
    - Machine learning: GraphLab
    - Faster queries: Spark

MapReduce Model

- input is stored as a set of key-value pairs (k,v)
- programmer writes map function
  map(k,v) -> list of (k2, v2) pairs
  gets run on every input element
- hidden shuffle phase:
  group all (k2, v2) pairs with the same key
- programmer writes reduce function
  reduce(k2, set of values) -> output pairs (k3,v3)

MapReduce implementation



MapReduce article

- Mike Stonebraker (Berkeley -> MIT)
  - built one of first relational DBs (Ingres) &
many subsequent systems:
Postgres, Mariposa, Aurora, C-Store, H-Store, ..
  - many startups: Illustra, Streambase, Vertica, VoltDB
  - 2014 Turing award
- David DeWitt (Wisconsin -> Microsoft)
  - parallel databases, database performance

Discussion

- Is MapReduce a major step backwards?
- Are database researchers incredibly bitter?
- Are systems researchers ignorant of 50 years of database work?

Systems vs Databases

- two generally separate streams of research
- distributed systems are relevant to both
    - much distributed systems research follows from OS community, including MapReduce
- (I have worked on both)

The database tradition

- Top-down design
- Most important: define the right semantics first
  - e.g., relational model and abstract language (SQL)
  - e.g., concurrency properties (serializability)
- …then figure out how to implement them
  - usually in a general purpose system
  - making them fast comes later
- Provide general interfaces for *users*

The OS tradition

- Bottom-up design
- Most important: engineering elegance
  - simple, narrow interfaces
  - clean, efficient implementations
  - performance and scalability first-class concerns
- Figuring out the semantics is secondary
- Provide tools for *programmers* to build systems

- Where does MapReduce fit into this?

- Does this help explain the critique?

MapReduce Critiques

- Not as good as a database interface
  - no schema; uses imperative language instead of declarative
- Poor implementation: no indexes, can’t scale
- Not novel
- Missing DB features & incompatible with existing DB tools
  - loading, indexing, transactions, constraints, etc

- Is MapReduce even a database?
- Is this an apples-to-oranges comparison?
- Should Google have built a scalable database instead of MR?

MapReduce vs DBs

- Maybe not that far off?
- Languages atop MapReduce for simplified (either declarative or imperative) queries:
  - Sawzall (Google); Pig (Yahoo), Hive (Facebook)
  - often involve adding schema to data

(My) lessons from MapReduce

- Specializing the system to focus on a particular type of processing makes the problem tractable
- Map/reduce functional model supports writing easier parallel code
(though so does the relational DB model!)
- Fault-tolerance is easy when computations are idempotent and stateless: just reexecute!

Non-lesson

- The map and reduce phases are not fundamental
- Don't need to follow the pattern
  input -> map -> shuffle -> reduce -> output
- Some computations can't be expressed in this model
- but could generalize MapReduce to handle them

Example

- 1. score webpages by words they contain
  2. score webpages by # of incoming links
  3. combine the two scores
  4. sort by combined score

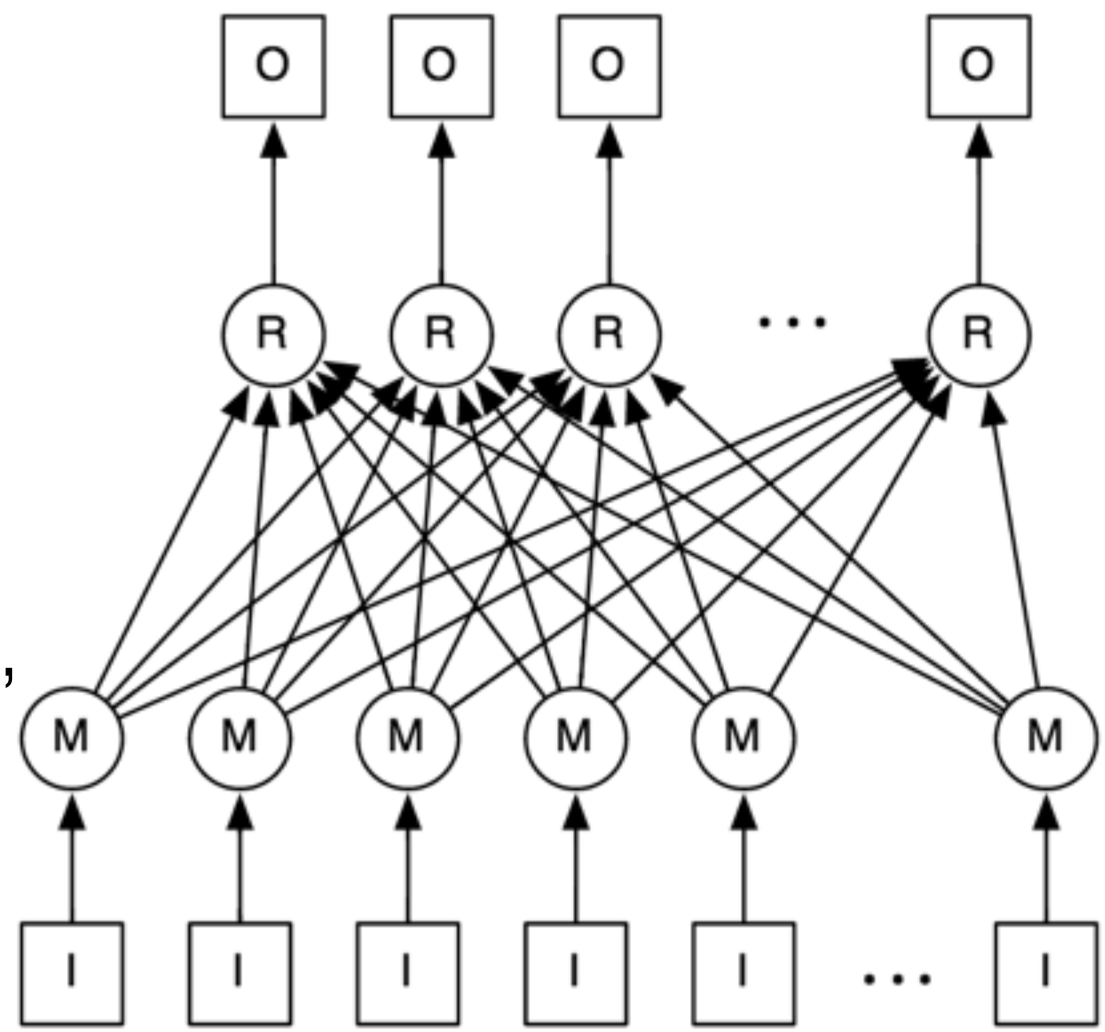
- would require multiple MR runs, probably 1 per step

- step 3 has 2 inputs; MR supports only one

- MR requires writing output & intermed results to disk

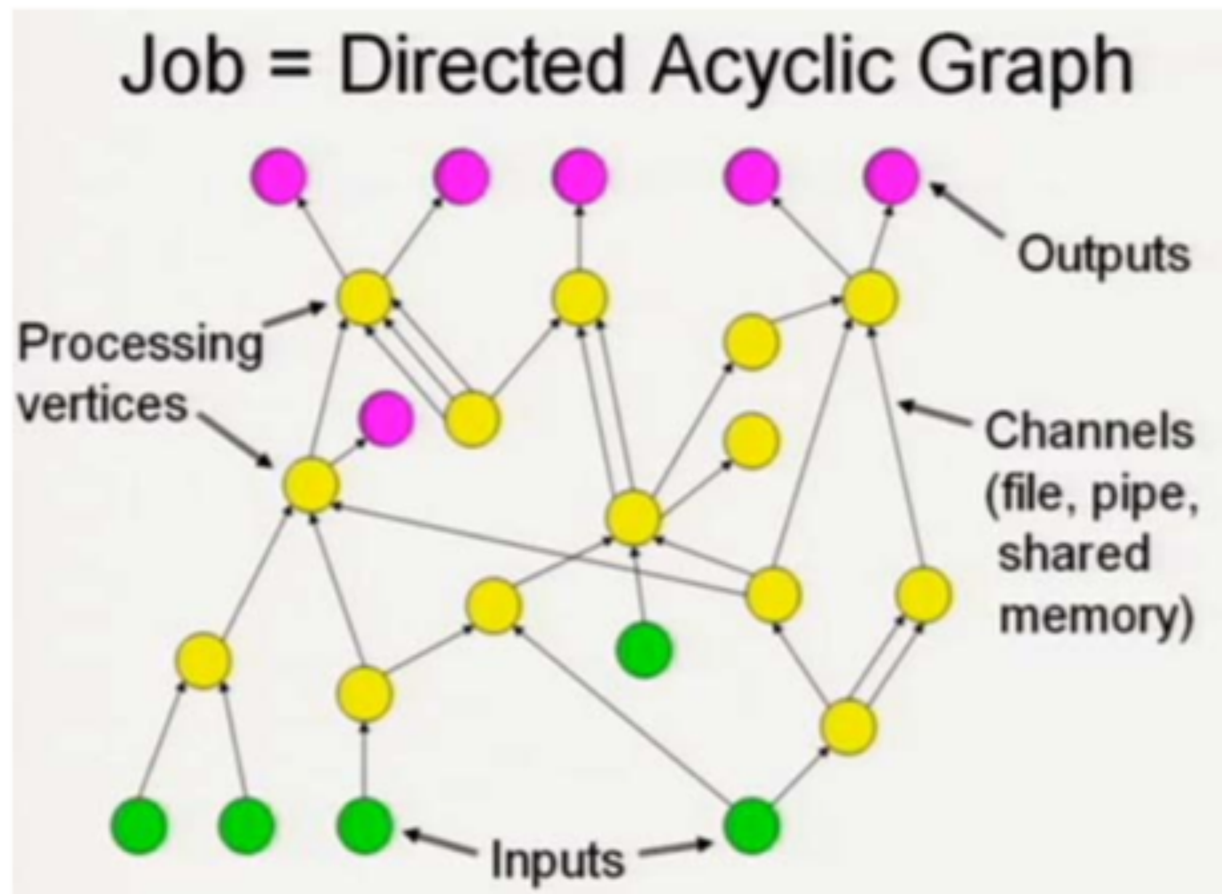
Dryad

- MSR system that generalizes MapReduce
- Observation: MapReduce computation can be visualized as a DAG
  - vertexes are inputs, outputs, or computation workers
  - edges are communication channels



Dryad

- Arbitrary programmer-specified graphs
- inputs, outputs = set of typed items
- edges are channels (TCP, pipe, temp file)
- intermediate processing vertexes can have several inputs and outputs



Dryad implementation

- Similar to MapReduce
  - vertices are stateless, deterministic computations
  - no cycles means that after a failure, can just rerun a vertex's computation
  - if its inputs are lots, rerun upstream vertices (transitively)

Programming Dryad

- Don’t want programmers to directly write graphs
- also built DryadLINQ, an API that integrates with programming languages (e.g., C#)

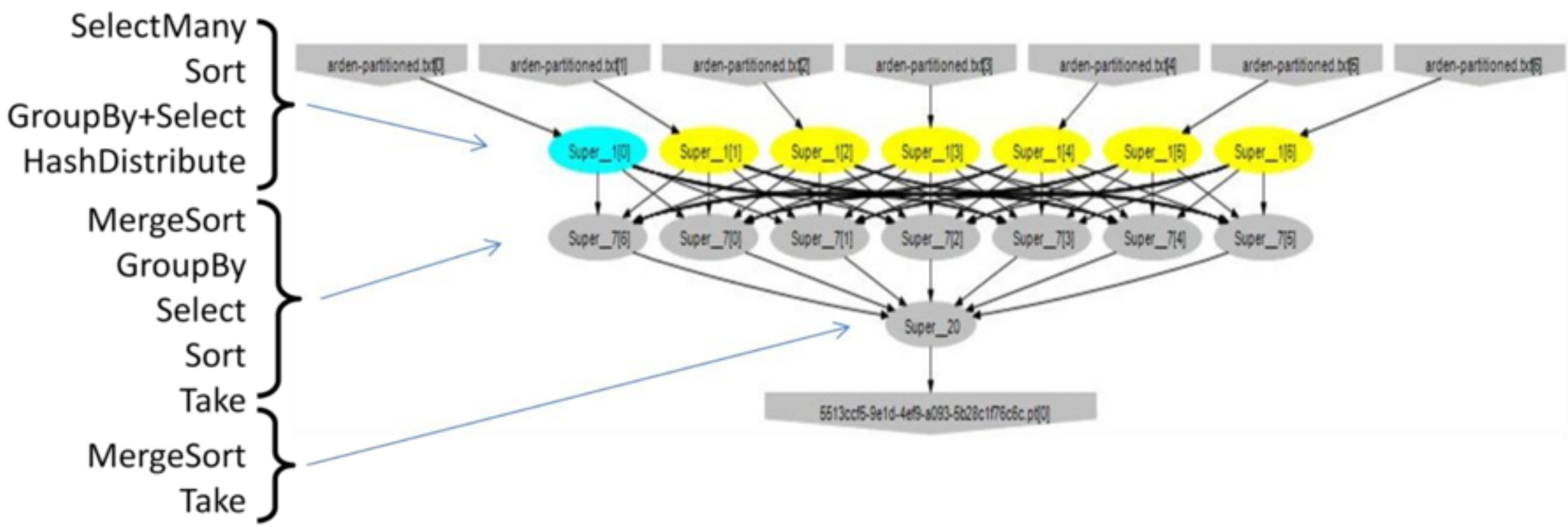
DryadLINQ example

- Word frequency: count occurrences of each word, return top 3

```
public static IQueryable<Pair> Histogram(input, k){
  var words = input.SelectMany(x => x.Split(' '));
  var groups = words.GroupBy(x => x);
  var counts = groups.Select(x => new Pair(x.Key, x.Count()));
  var ordered = counts.OrderByDescending(x => x.Count);
  var top = ordered.Take(k);
  return top;
}
```

table	"A line of words of wisdom"
SelectMany	["A", "line", "of", "words", "of", "wisdom"]
GroupBy	[["A"], ["line"], ["of", "of"], ["words"], ["wisdom"]]
Select	[{"A", 1}, {"line", 1}, {"of", 2}, {"words", 1}, {"wisdom", 1}]
OrderByDescending	[{"of", 2}, {"A", 1}, {"line", 1}, {"words", 1}, {"wisdom", 1}]
Take(3)	[{"of", 2}, {"A", 1}, {"line", 1}]

DryadLINQ example



Machine Learning: GraphLab

- ML and data mining are hugely popular areas now!
  - clustering, modeling, classification, prediction
- Need to run these algorithms on huge data sets
- Means that we need to run them on distributed systems
- Need new distributed systems abstractions

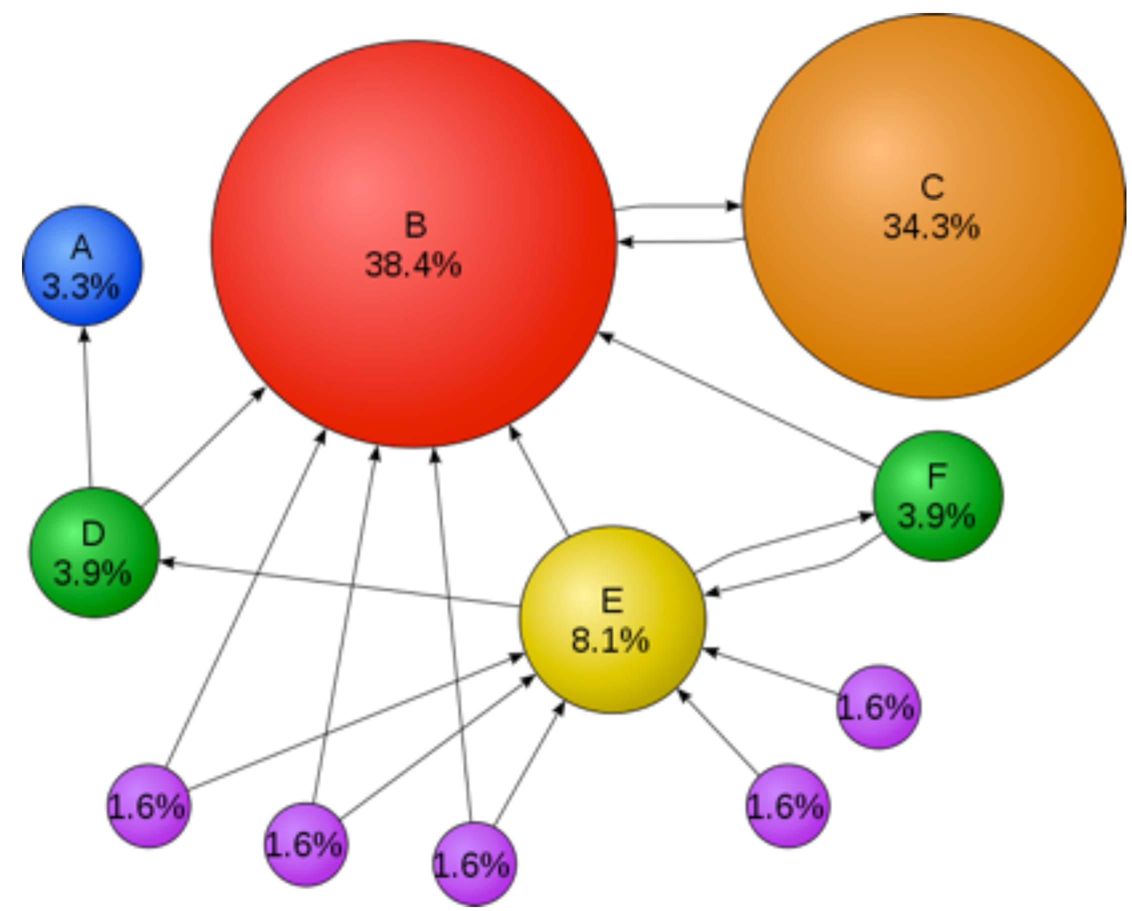
Example: PageRank

- Assign a score to each webpage
- Update the score:

$$\text{PageRank of site} = \sum \frac{\text{PageRank of inbound link}}{\text{Number of links on that page}}$$

PageRank Formula

- Repeat until converged



What’s the right abstraction?

- Message-passing & threads? (MPI/pthreads)
  - leaves all the hard work to the programmer!
    fault tolerance, load balancing, locking, races
- MapReduce?
  - fails when there are computational dependencies in data (Dryad can help)
  - fails when there is an iterative structure
    - rerun until it converges? programmer has to deal with this!
- GraphLab: computational model for graphs

Why graphs?

- most ML/DM applications are amenable to graph structuring
- ML/DM is often about dependencies between data
  - represent each data item as a vertex
  - represent each dependency between two pieces of data as an edge

Graph representation

- graph = vertices + edges, each with data
- graph structure is static, data is mutable
- update function for a vertex
  $f(v, S_v) \rightarrow (S_v, T)$
    - S_v is the scope of vertex v:
      the data stored in v *and* all adjacent vertexes + edges
    - vertex function can update any data in scope
    - T: output a new list of vertices that need to be rerun

Synchrony

- GraphLab model allows asynchronous computation
- synchronous = all parameters are updated simultaneously using values from previous time step
  - requires a barrier before next round; straggler problem
  - iterated MapReduce works like this
- asynchronous = continuously update parameters, always using most recent input values
  - adapts to differences in execution speed
  - supports dynamic computation:
    in PageRank, some nodes converge quickly; stop rerunning them!

Graph processing correctness

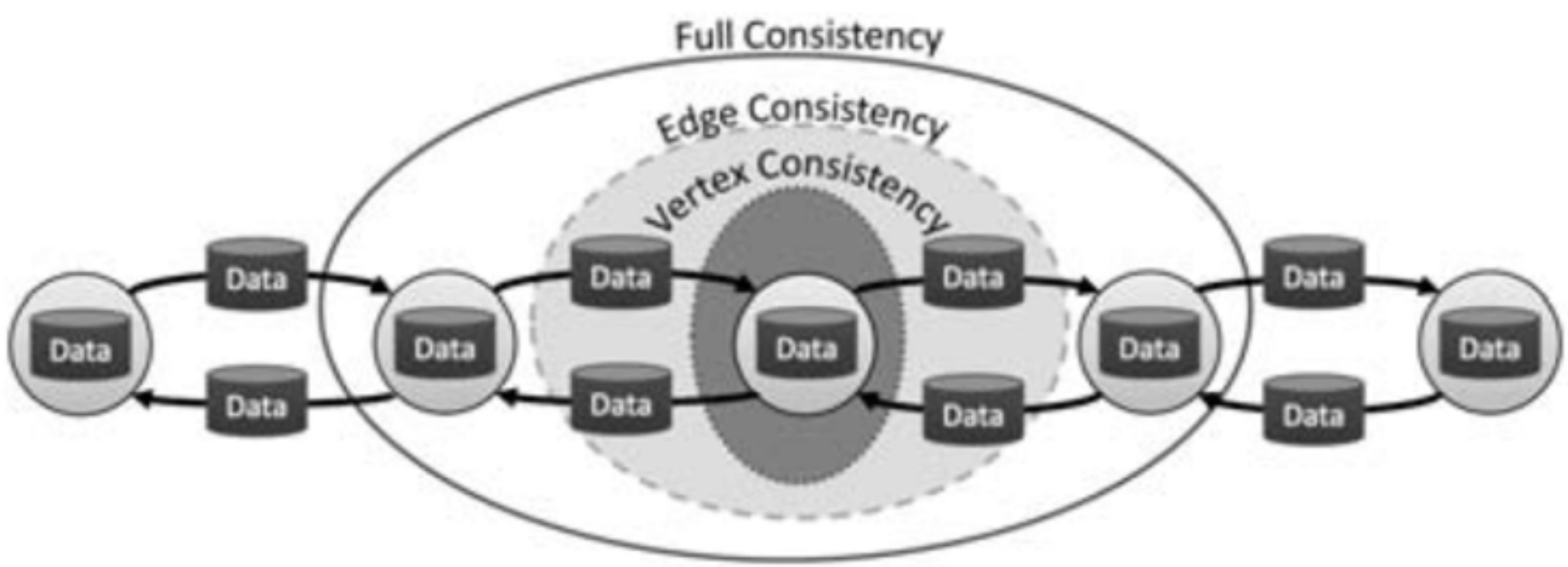
- Is asynchronous processing OK?
- Depends on the algorithm
  - some require total synchrony
  - usually ok to compute asynchronously as long as there's consistency
  - sometimes it's even ok to run without locks at all
- Serializability: same results as though we picked a sequential order of vertexes and each ran their update function in sequence

GraphLab implementation

- 3 versions
  - single machine, multicore shared memory
  - Distributed GraphLab (this paper)
  - PowerGraph (distributed, optimized for power-law graphs)

Single-machine GraphLab

- Maintain queue of vertices to be updated, run update functions on these in parallel
- Ensuring serializability involves locking the scope of a vertex update function
- Weaker versions for optimizations: reduced scope



Making GraphLab distributed

- Partition the graph across machines w/ edge cut
  - partition boundary is set of edges =>
    each vertex is on exactly one machine
  - except we need "ghost vertices" to compute:
    cached copies of vertices stored on neighbors
- Consistency problem:
  keep the ghost vertices up to date
- Partitioning controls load balancing
  - want same number of vertices per partition (=> computation)
  - want same number of ghosts (=> network load for cache updates)

Locking in GraphLab

- Same general idea as single-machine but now distributed!
- Enforcing consistency model requires acquiring locks on vertex scope
- If need to acquire lock on edge or vertex on boundary, need to do it on all partitions (ghosts) involved
- What about deadlock?
  - usual DB answer is to detect deadlocks and roll back
  - GraphLab uses a canonical ordering of lock acquisition instead

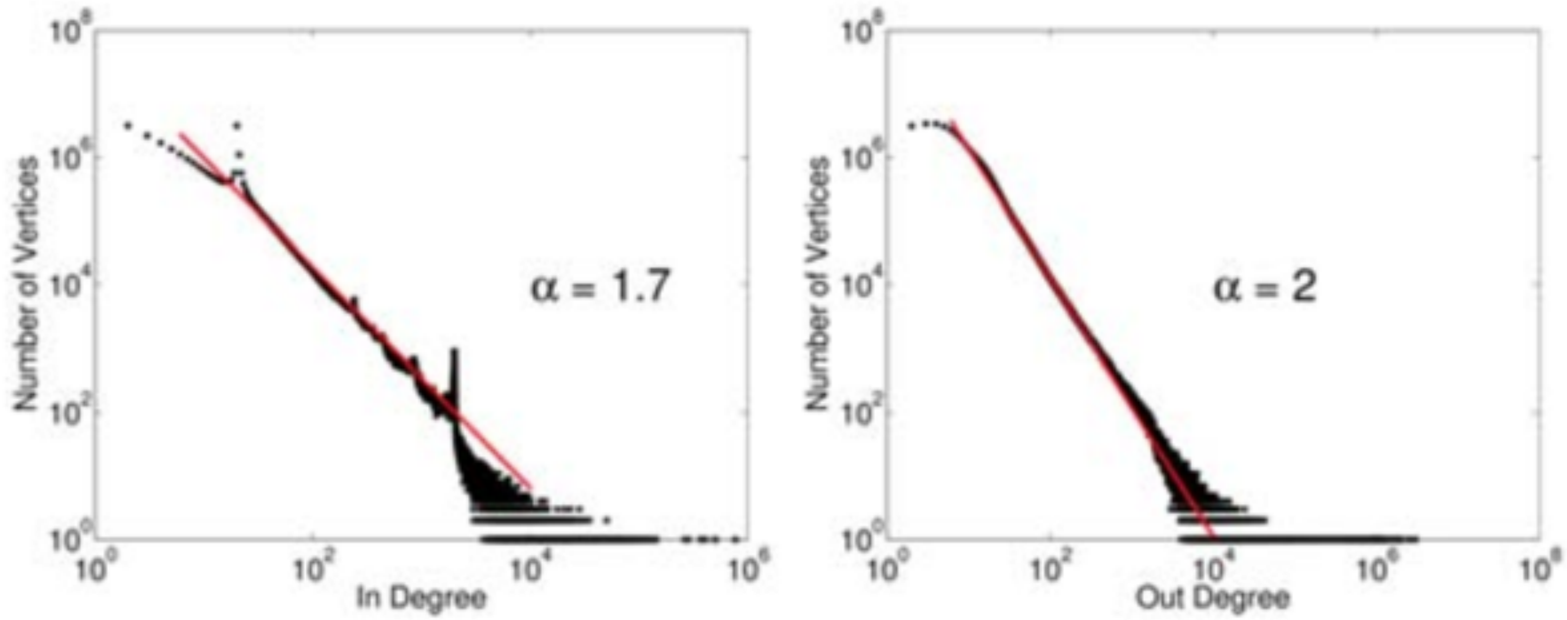
Fault-tolerance

- MapReduce answer isn't good enough:
  workers have state so we can't just reassign their task
- Take periodic, globally consistent snapshots
  - Chandy-Lamport snapshot algorithm!

Challenge: power-law graphs

- Many graphs are not uniform!

- Power-law: a few popular vertices with ***many*** edges, many unpopular vertices with a few edges



(a) Twitter In-Degree (b) Twitter Out-Degree

- Problem for GraphLab: edge cuts are hugely imbalanced

PowerGraph: later version

- First improvement:
  partition by cutting *vertices* instead of edges
  - each edge is in one partition, vertices can be in multiple
  - high-degree vertices are split over many partitions
- Second: parallelize update function (new API)
  - each server computes its “local” change to a split vertex,
    e.g., PageRank computation from other pages on that server
    then accumulate and apply the partial updates
- Third: better algorithm for fair partitioning

Spark

- Framework for large-scale distributed computation
- Designed for to support interactive applications not just batch processing
- Relatively recent (2012) but used widely:
  IBM, Yahoo, Baidu, Groupon, …
  Apache project, 1000+ contributors

Spark motivation

- Want a general framework for distributed computations
- MapReduce isn't enough
  - too inflexible, can't handle iteration, etc
  - can't do interactive queries, only batch processing
- Argument: MR can't handle complex interactive queries because the only way to share data across jobs is to store it in stable storage

Spark challenge

- Store intermediate data in a way that’s both fault-tolerant and efficient
  - want it to be in-memory because that’s 10-100x faster than writing to disk / network FS
  - enable reusing intermediate results between different computations
  - but in-memory data can be lost on failure!

Abstraction: RDDs

- immutable collection of records, partitioned
- only two ways to create a RDD
  - access dataset on stable storage
  - transformation of existing RDD (map, join, etc)
- Creation is lazy, just specifies a plan for computing
- Actions, e.g., storing result, cause RDD to be materialized

Example: PageRank

1. Start each page with a rank of 1
2. On each iteration, update each page's rank to

$$\Sigma_{i \in neighbors}\, rank_i \,/\, |neighbors_i|$$

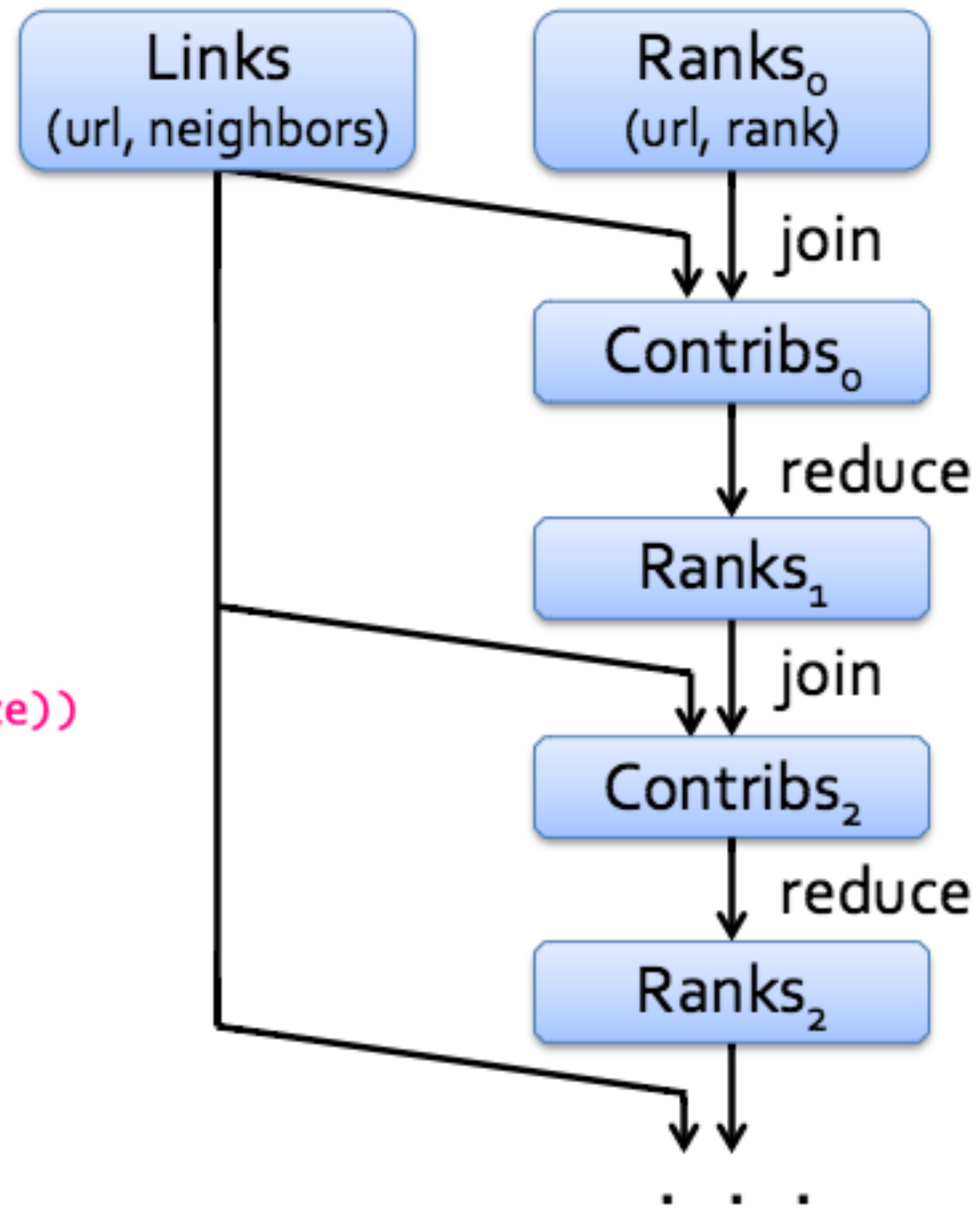
```
links = // RDD of (url, neighbors) pairs
ranks = // RDD of (url, rank) pairs

for (i <- 1 to ITERATIONS) {
  ranks = links.join(ranks).flatMap {
    (url, (links, rank)) =>
      links.map(dest => (dest, rank/links.size))
  }.reduceByKey(_ + _)
}
```

PageRank RDDs

```
links = // RDD of (url, neighbors) pairs
ranks = // RDD of (url, rank) pairs

for (i <- 1 to ITERATIONS) {
  ranks = links.join(ranks).flatMap {
    (url, (links, rank)) =>
      links.map(dest => (dest, rank/links.size))
  }.reduceByKey(_ + _)
}
```



RDDs

- RDDs are represented as
  - **list of parent RDDs**
  - **function to compute result from them**
  - partitioning scheme
  - computation placement hint
  - list of partitions for the RDD

Failure recovery in Spark

- Spark only makes one in-memory copy of a newly computed RDD partition! (by default)
    - if it fails, data is gone!
- Scheduler detects machine failure and schedules recomputation
    - will need to recursively compute all partitions it depends on, until one of them is found
- Checkpointing is optional
    - user can ask Spark scheduler to make some RDD persistent
    - expensive, but means that failure won't have to recompute everything