CPT-S 415

Big Data

Yinghui Wu

EME B45

CPT-S 415
Big Data

MapReduce

- ✓ MapReduce model
- ✓ MapReduce for relational operators
- ✓ MapReduce for graph querying

MapReduce

MapReduce

✓ A programming model with two primitive functions:

✓ Map: <k1, v1> → list (k2, v2)

✓ Reduce: <k2, list(v2)> → list (k3, v3)

✓ Input: a list <k1, v1> of key-value pairs

✓ Map: applied to each pair, computes key-value pairs <k2, v2>

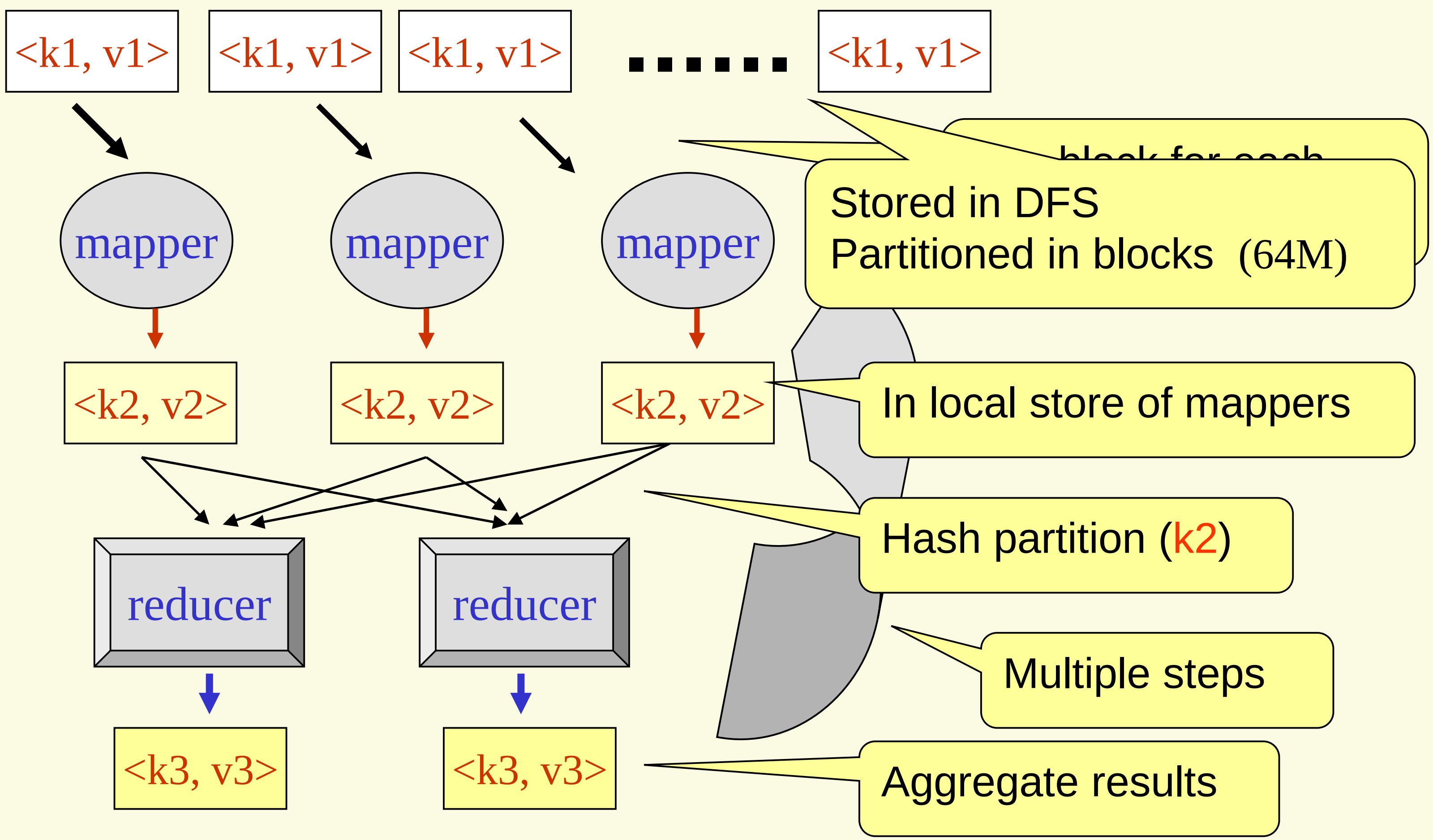
- The intermediate key-value pairs are hash-partitioned based on k2. Each partition (k2, list(v2)) is sent to a reducer

✓ Reduce: takes a partition as input, and computes key-value pairs <k3, v3>

The process may reiterate – multiple map/reduce steps

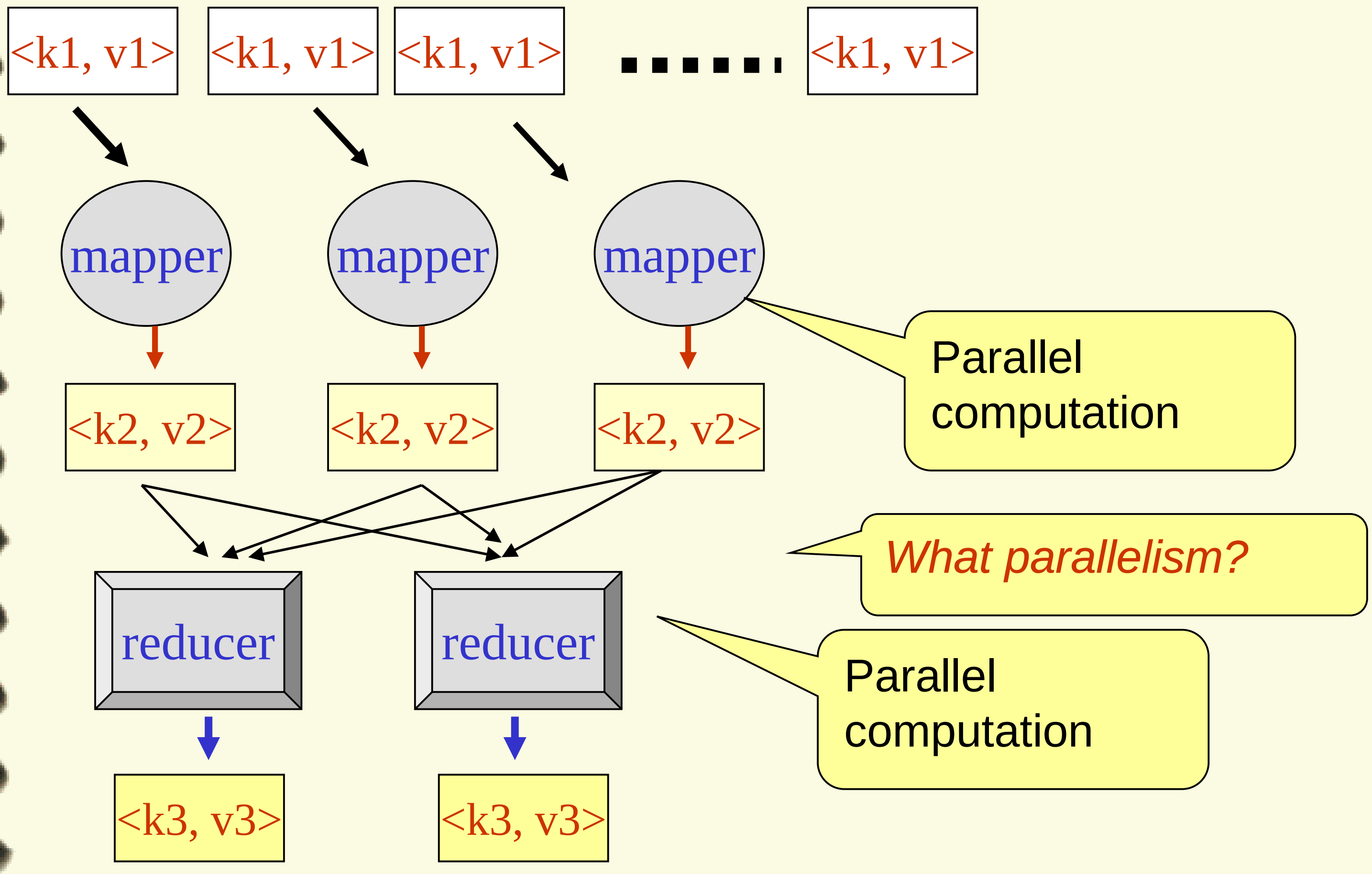
How doe

Architecture (Hadoop)



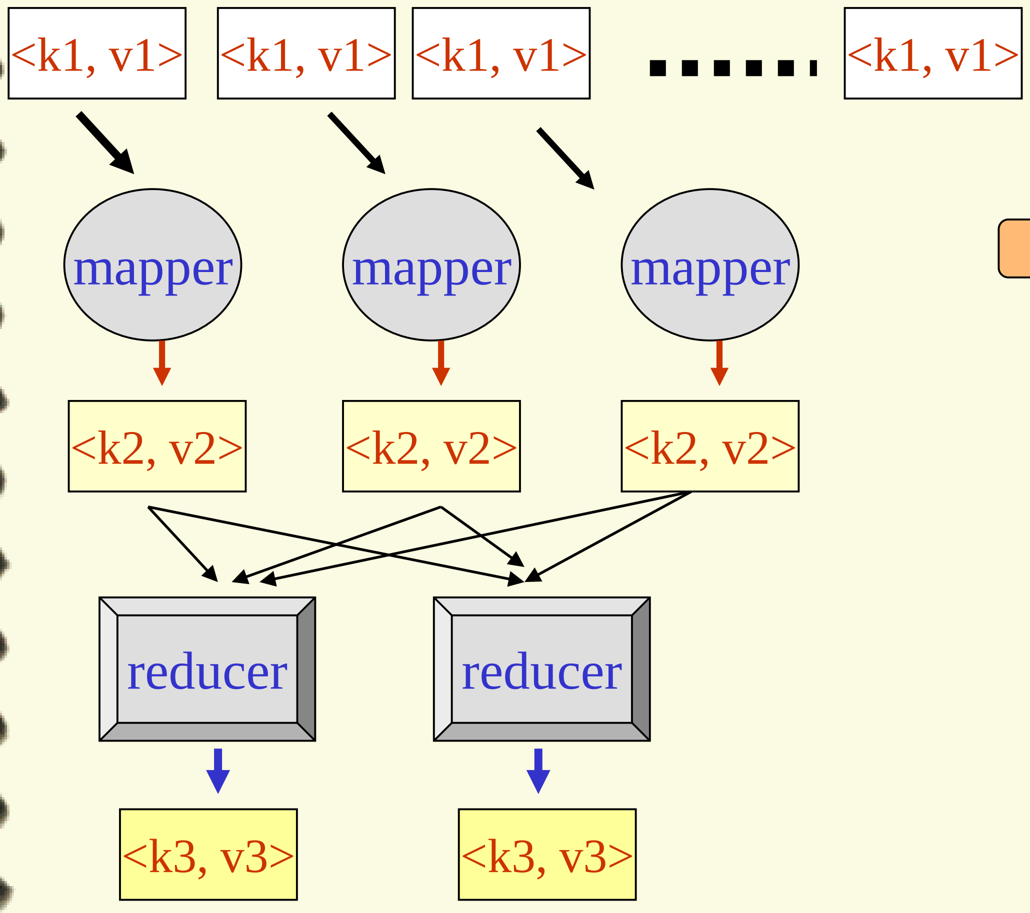
No need to worry about how the data is stored and sent

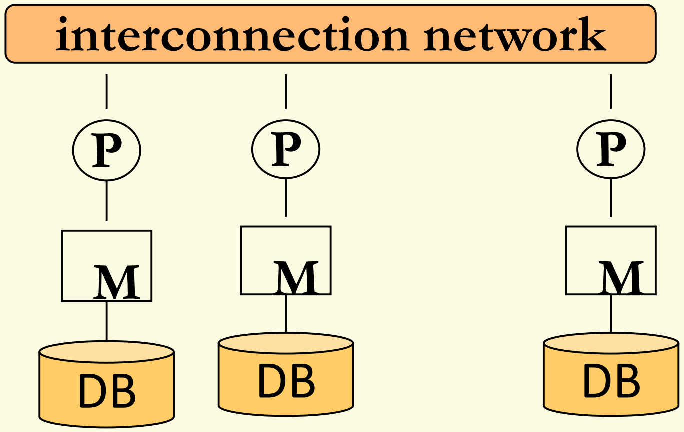
Connection with parallel database systems



Data partitioned parallelism

Parallel database systems





Restricted query languages: only two primitive

MapReduce implementation of relational operators

Projection Π_A R

not necessarily a key of R

Input: for each tuple t in R, a pair (key, value), where value = t

✓ Map(key, t)

- emit (t.A, t.A)

Apply to each input tuple, in parallel; emit new tuples with projected attributes

✓ Reduce(hkey, hvalue[ ])

- emit(hkey, hkey)

the reducer is not necessary; but it eliminates duplicates. Why?

Mappers: processing each tuple in parallel

Selection

Selection σ_C R

Input: for each tuple t in R, a pair (key, value), where value = t

- ✓ Map(key, t)
  - if C(t)
    - then emit (t, “1”)

Apply to each input tuple, in parallel; select tuples that satisfy condition C

- ✓ Reduce(hkey, hvalue[ ])
  - emit(hkey, hkey)

Reducers: eliminate duplicates

Union

A mapper is assigned chunks from either R1 or R2

Union R1 ∪ R2

Input: for each tuple t in R1 and s in R2, a pair (key, value)

- ✓ Map(key, t)
  - emit (t, “1”)

A mapper just passes an input tuple to a reducer

- ✓ Reduce(hkey, hvalue[ ])
  - emit(hkey, hkey)

Reducers simply eliminate duplicates

Map: process tuples of R1 and R2 uniformly

Set difference

distinguishable

Set difference R1 – R2

Input: for each tuple t in R1 and s in R2, a pair (key, value)

✓ Map(key, t)
- if t is in R1
  - then emit(t, “1”)
  - else emit(t, “2”)

tag each tuple with its source

✓ Reduce(hkey, hvalue[ ])
- if only “1” appears in the list hvelue
  - then emit(hkey, hkey)

Why does it work?

Reducers do the checking

Join Algorithms in MapReduce

- ✓ Reduce-side join
- ✓ Map-side join
- ✓ In-memory join
  - Striped variant
  - Memcached variant

Reduce-side join

Natural R1 $\bowtie_{R1.A = R2.B}$ R2, where R1[A, C], R2[B, D]

Input: for each tuple t in R1 and s in R2, a pair (key, value)

- ✓ Map(key, t)
  - if t is in R1
    - then emit(t.[A], ("1", t[C]))
    - else emit(t.[B], ("2", t.[D]))

Hashing on join attributes

- ✓ Reduce(hkey, hvalue[ ])
  - for each ("1", t[C]) and each ("2", s[D]) in the list hvalue

How to implement R1 $\bowtie$ R2 $\bowtie$ R3?

Nested loop

Reduce-side join (based on hash partitioning)

Map-side join

Recall $R1 \bowtie_{R1.A = R2.B} R2$

- ✓ Partition R1 and R2 into n partitions, by the same partitioning function in R1.A and R2.B, via either range or hash partitioning
- ✓ Compute $R^i1 \bowtie_{R1.A = R2.B} R^i2$ locally at processor i
- ✓ Merge the local results

Partitioned join

Map-side join:

- ✓ Input relations are partitioned and sorted based on join keys
- ✓ Map over R1 and read from the corresponding partition of R2

Merge join

- ✓ map(key, t)
  - read R^i2
  - for each tuple s in relation R^i2
    - if t[A] = s[B] then emit((t[A], t[C], s[D]), t[A])

Limitation: sort order and partitioning

In-memory join

Recall $R1 \bowtie_{R1.A < R2.B} R2$

- ✓ Partition R1 into n partitions, by any partitioning method, and distribute it across n processors
- ✓ Replicate the other relation R2 across all processors
- ✓ Compute $R^j1 \bowtie_{R1.A < R2.B} R2$ locally at processor j
- ✓ Merge the local results

Fragment and replicate join

Broadcast join

- ✓ A smaller relation is broadcast to each node and stored in its local memory
- ✓ The other relation is partitioned and distributed across mappers

- ✓ map(key, t)
  - for each tuple s in relation R2 (local)
    - if t[A] = s[B] then emit((t[A], t[C], s[D]), t[A])

Limitation: memory

Aggregation

R(A, B, C), compute sum(B) group by A

- ✓ Map(key, t)
  - emit (t[A], t[B])

Grouping: done by MapReduce framework

- ✓ Reduce(hkey, hvalue[ ])
  - sum := 0;
  - for each value s in the list hvalue
    - sum := sum + 1;
  - emit(hkey, sum)

Compute the aggregation for each group

Leveraging the MapReduce framework

Practice: validation of functional dependencies

- A functional dependency (FD) defined on schema R: $X \rightarrow Y$
  - For any instance D of R, D satisfies the FD if for any pair of tuples t and t', if t[X] = t'[X], then t[Y] = t'[Y]
  - Violations of the FD in D:

  { t | there exists t' in D, such that t[X] = t'[X], but t[Y] $\neq$ t'[Y] }

- Develop a MapReduce algorithm to find all the violations of the FD in D

  key

  - Map: for each tuple t, add it to list (t[X], t)
  - Reduce: find all tuples t such that there exists t', with but t[Y] $\neq$ t'[Y]; add such tuples to list (1, t)

Transitive closures

- ✓ The transitive closure TC of a relation R[A, B]
  - R is a subset of TC
  - if (a, b) and (b, c) are in TC, then (a, c) is in TC

  That is,
  - TC(x, y) :- R(x, y);
  - TC(x, z) :- TC(x, y), TC(y, z).
- ✓ Develop a MapReduce algorithm that given R, computes TC

  A fixpoint computation
  - How to determine when to terminate?
  - How to minimize redundant computation?

: *Write a MapReduce algorithm*

A naïve MapReduce algorithm

Given R(A, B), compute TC

Initially, the input relation R

- ✓ Map((a, b), value)
  - emit (a, (“r”, b)); emit(b, (“l”, a));

Iteration: the output of reducers becomes the input of mappers in the next round of MapReduce computation

- ✓ Reduce(b, hvalue[ ])
  - for each (“l”, a) in hvalue
    - for each (“r”, c) in hvalue
      - emit(a, c); emit(b, c);
    - emit(a, b);

One round of recursive computation:
- ✓ Apply the transitivity rule
- ✓ Restore (a, b), (b, c). Why?

Termination?

A MapReduce algorithm

Given R(A, B), compute TC

- ✓ Map((a, b), value)
  - emit (a, (“r”, b)); emit(b, (“l”, a));

- ✓ Reduce(b, hvalue[ ])
  - for each (“l”, a) in hvalue
    - for each (“r”, c) in hvalue
      - emit(a, c); emit(b, c);
    - emit(a, b);

Termination: when the intermediate result no longer changes

✓ controlled by a non-MapReduce driver

How to improve it?
Course project

Naïve: not very efficient. Why?

Advantages of MapReduce

✓ Simple: one only needs to define two functions

no need to worry about how the data is stored, distributed and how the operations are scheduled

✓ scalability: a large number of low end machines

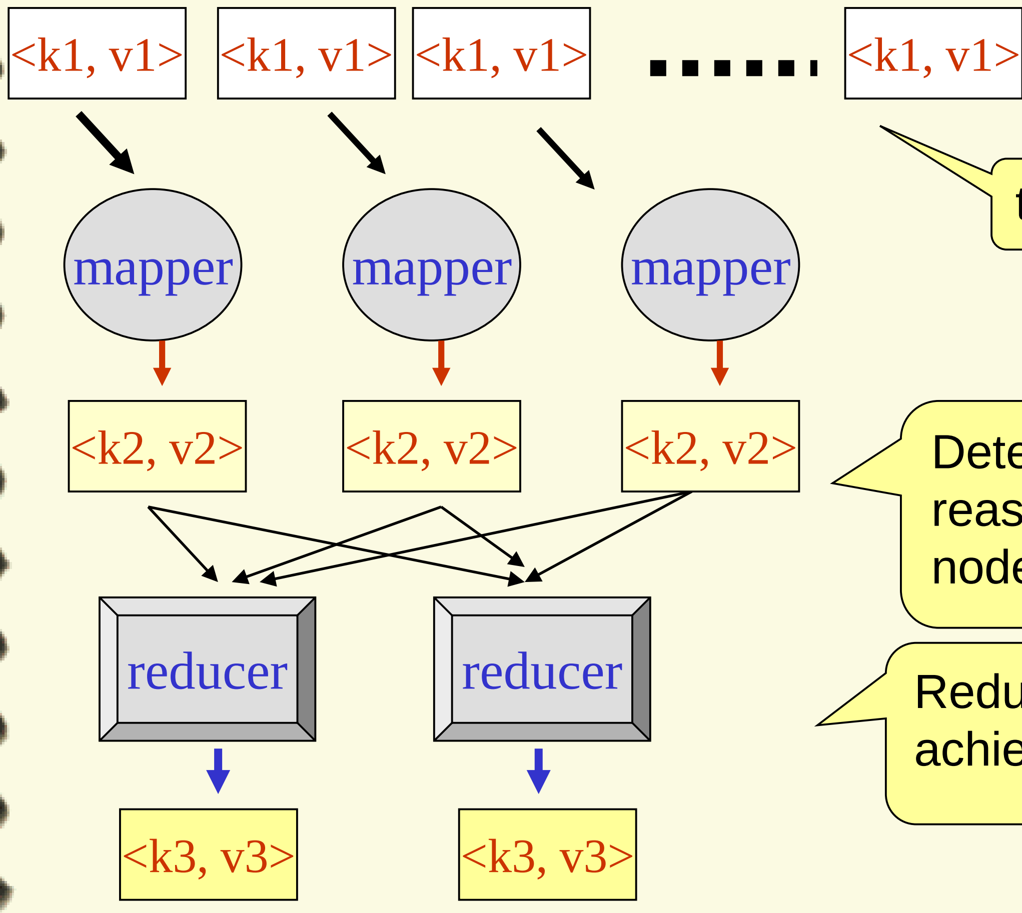
- scale out (scale horizontally): adding a new computer to a distributed software application; lost-cost "commodity"
- scale up (scale vertically): upgrade, add (costly) resources to a single node

✓ independence: it can work with various storage layers (e.g., Bigtable)

✓ flexibility: independent of data models or schema

Fault tolerance: why?

Fault tolerance



triplicated

Detecting failures and reassigning the tasks of failed nodes to healthy nodes

Redundancy checking to achieve load balancing

Able to handle an average of 1.2 failures per analysis job

MapReduce platforms

- ✓ Apache Hadoop, used by Facebook, Yahoo, …
  - Hive, Facebook, HiveQL (SQL)
  - PIG, Yahoo, Pig Latin (SQL like)
  - SCOPE, Microsoft, SQL
- ✓ NoSQL
  - Cassandra, Facebook, CQL (no join)
  - HBase, Google, distributed BigTable
  - MongoDB, document-oriented (NoSQL)
- ✓ Distributed graph query engines
  - Pregel, Google
  - TAO: Facebook,
  - GraphLab, machine learning and data mining
  - Neo4j, Neo Tech; Trinity, Microsoft; HyperGraphDB (knowledge)

A vertex-centric model

Study some of these

MapReduce Algorithms: Graph Processing

MapReduce algorithms

Input: query Q and graph G

Output: answers Q(G) to Q in G

```
map(key: node,  value: (adjacency-list, others) )
      {computation;
       emit (mkey, mvalue)
      }
```

Match rkey, rvalue when multiple iterations of MapReduce are needed

Match mkey, mvalue

```
reduce(key:  __ ,  value: list[value] )
      { …
      emit (rkey, rvalue)
      }
```

compatibility

BFS for distance queries

Dijkstra’s algorithm for distance queries

✓ Distance: single-source shortest-path problem

- Input: A directed weighted graph G, and a node s in G
- Output: The lengths of shortest paths from s to all nodes in G

✓ Dijkstra (G, s, w):

1. for all nodes v in V do
   a. d[v] $\leftarrow \infty$; $\leftarrow$
2. d[s] $\leftarrow$ 0; Que $\leftarrow$ V;
3. while Que is nonempty do
   a. u $\leftarrow$ ExtractMin(Que);
   b. for all nodes v in adj(u) do
      a) if d[v] > d[u] + w(u, v) then … u, v);

Use a priority queue Que; w(u, v): weight of edge (u, v); d(u): the distance from s to u

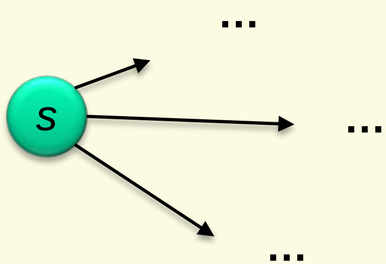
Extract one with the minimum d(u)

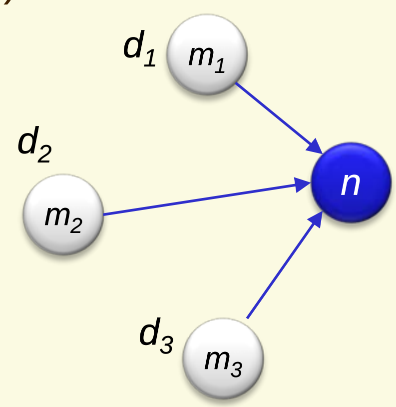
MapReduce?

O(|V| log|V| + |E|).

Finding the Shortest Path

- Consider simple case of equal edge weights: solution to the problem can be defined inductively
- Intuition:
  - Define: *b* is reachable from *a* if *b* is on adjacency list of *a*
  - DistanceTo(s) = 0
  - For all nodes *p* reachable from *s*,
    DistanceTo(p) = 1
  - For all nodes *n* reachable from some other set of nodes *M*, DistanceTo(n) = 1 + min(DistanceTo(m), $m \in M$)





From Intuition to Algorithm

Input: graph G, represented by adjacency lists

- ✓ Node N:
  - Node id: nid n
  - N.distance: from start node s to N
  - N.AdjList: [(m, w(n, m))], node id and weight of edge (n, m)
- ✓ Key: node id n
- ✓ Value of node N:
  - N.Distance: from start node s to n got so far
  - N.AdjList
- Initialization: for all n, N.Distance = ∞

key-values pairs

From Intuition to Algorithm

- Mapper:
  - $\forall m \in$ adjacency list: emit (m, d + w(n, m)))
- Sort/Shuffle
  - Groups distances by reachable nodes
- Reducer:
  - Selects minimum distance path for each reachable node
  - Additional bookkeeping needed to keep track of actual path

Mapper

Map (nid n, node value N)

- d ← N.distance;
- emit(nid n, N);
- for each (m, w) in N.AdjList do
  - emit(nid m, d + w(n, m));

Why?

Revise distance of m via n

Parallel processing

- ✓ all nodes are processed in parallel, each by a mapper
- ✓ for each node m adjacent to n, emit a revised distance via n
- ✓ emit (nid n, N): preserve graph structure for iterative processing

Data-partitioned parallelism

Reducer

Group by node id
Each d in list is either
✓ a distance to m from a predecessor
✓ or node M

Reduce (nid m, list[d1,d

- $d_{min} \leftarrow \infty$;
- for all d in list do
  - if IsNode(d)
    - then M $\leftarrow$ d;
    - else if d < d_{min}
      - then $d_{min} \leftarrow$ d;
- M.distance $\leftarrow d_{min}$;
- emit (nid m, node M);

Always be there. Why?

Minimum distance so far

Update M.distance for this round

list for m:

✓ distances from all predecessors so far

✓ Node M: must exist (from Mapper)

Current M.distance: minimum from all predecessors

Iterations and termination

Each MapReduce iteration advances the “known frontier” by one hop

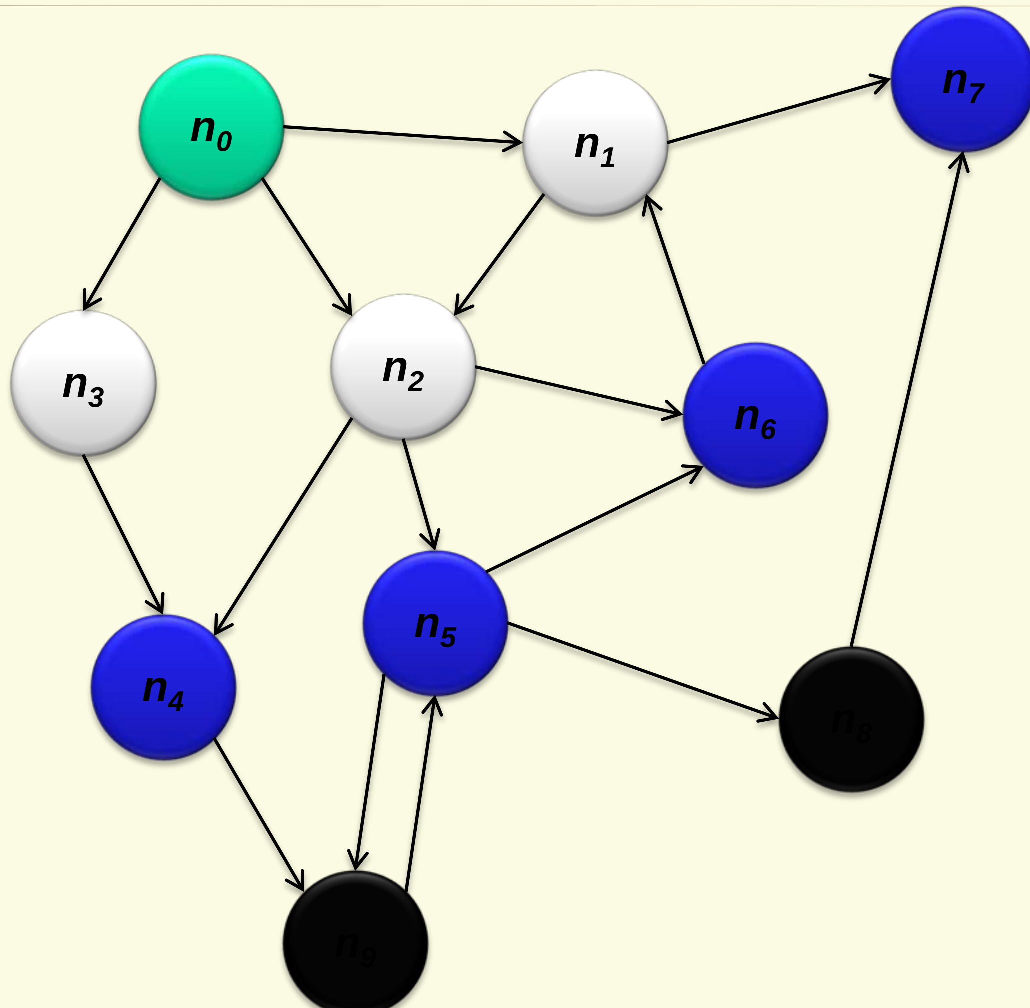
- ✓ Subsequent iterations include more and more reachable nodes as frontier expands
- ✓ Multiple iterations are needed to explore entire graph

Termination: when the intermediate result no longer changes

- ✓ controlled by a non-MapReduce driver
  - ✓ Use a flag – inspected by non-MapReduce driver

Termination control

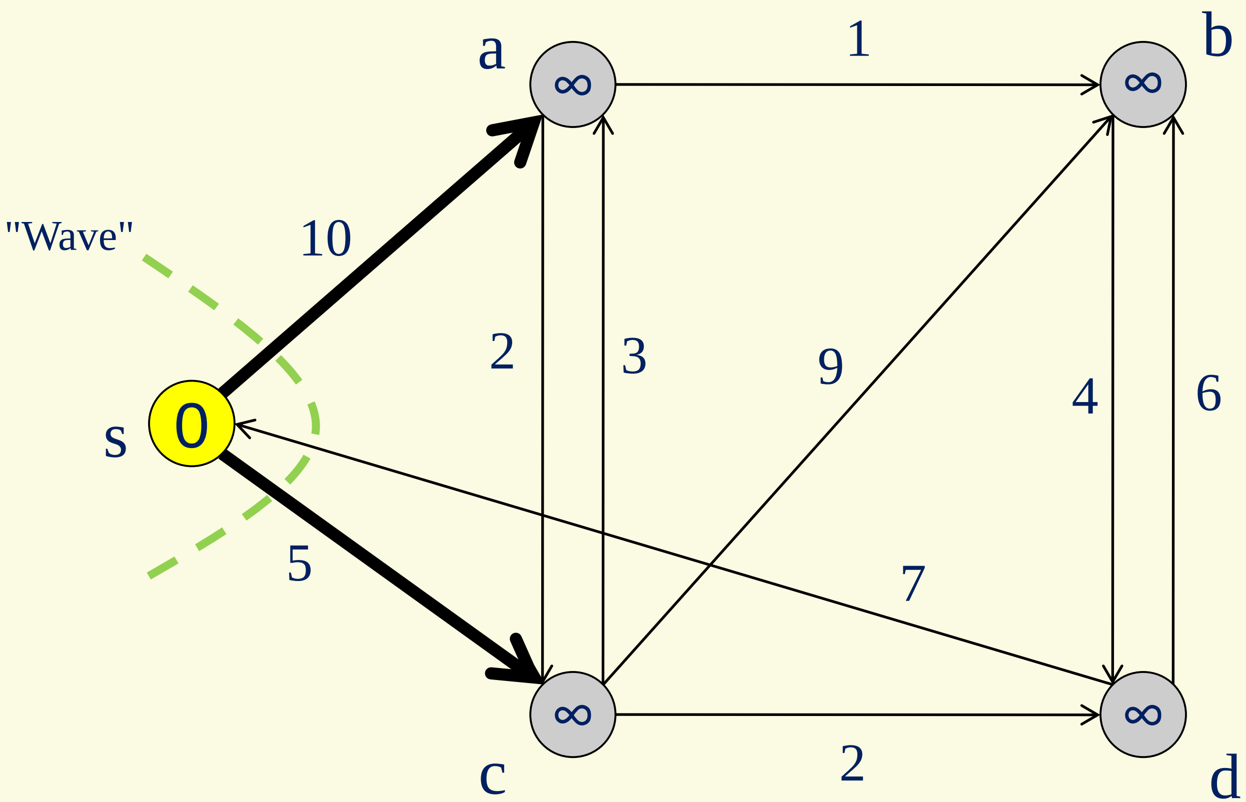
Visualizing Parallel BFS



Source: Wikipedia (Wave)

Iteration 0: Base case

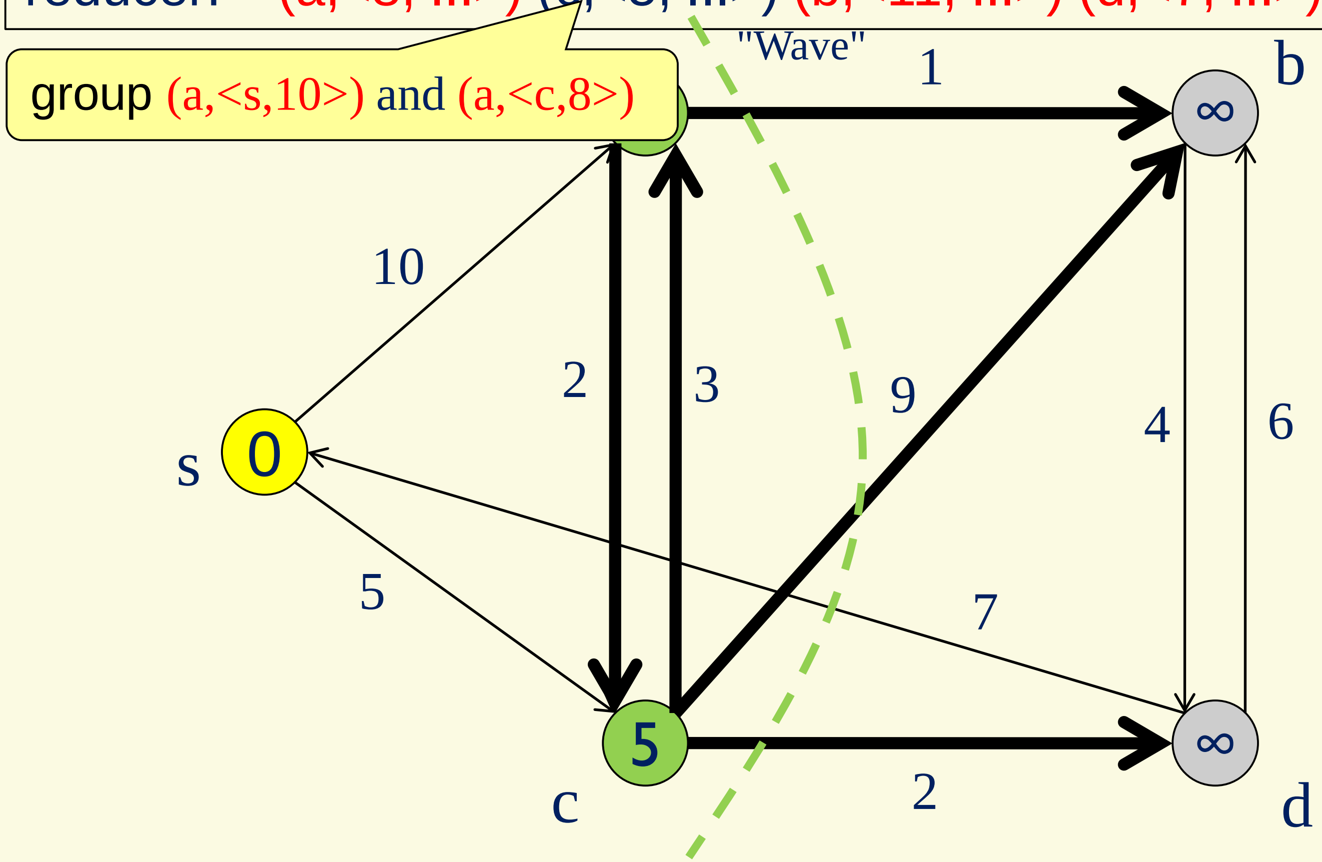
mapper: (a,<s,10>) (c,<s,5>) edges
reducer: (a,<10, ...>) (c,<5, ...>)



Iteration 1

mapper: (a,<s,10>) (c,<s,5>) (a,<c,8>) (c,<a,12>) (b,<a,11>)
(b,<c,14>) (d,<c,7>) edges

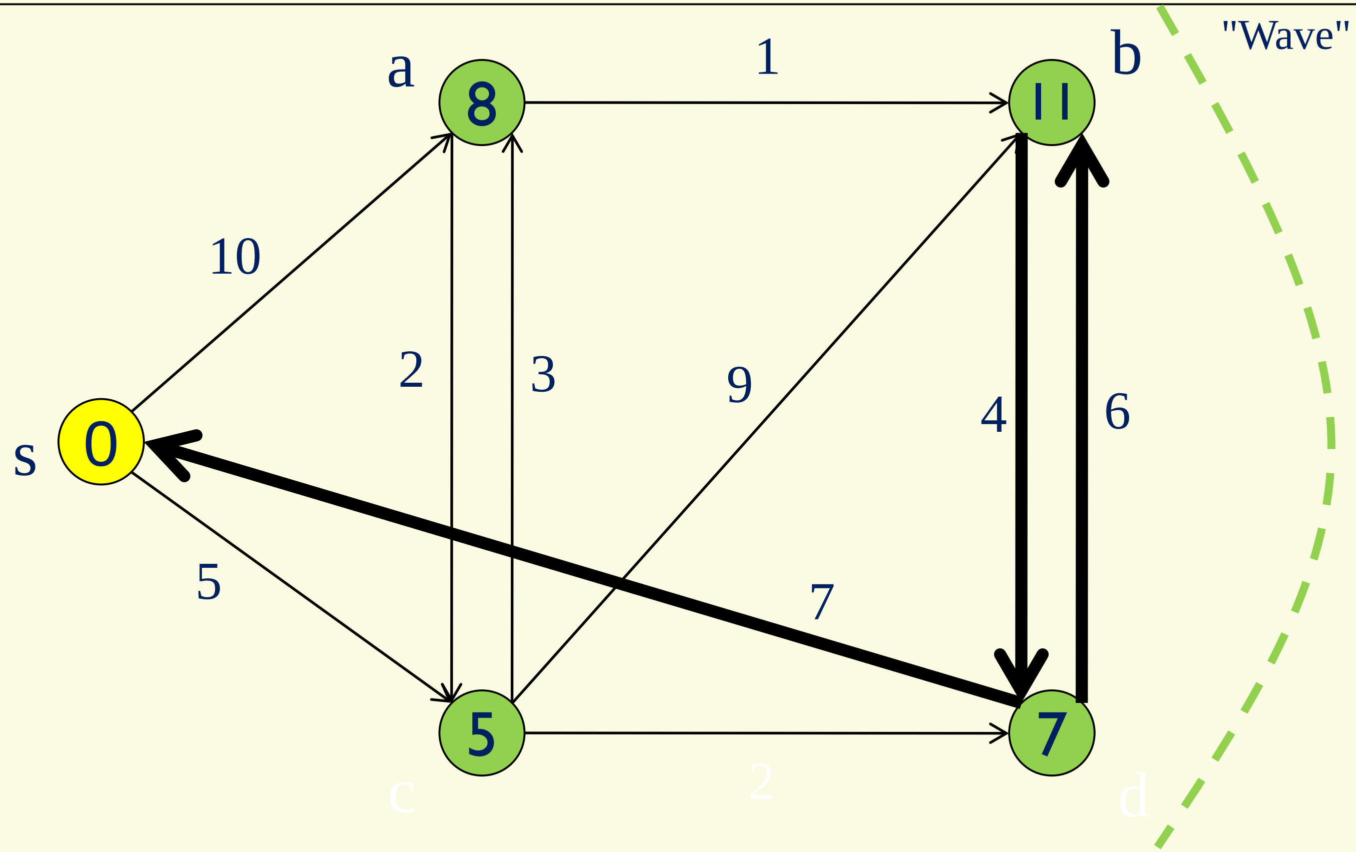
reducer: (a,<8, ...>) (c,<5, ...>) (b,<11, ...>) (d,<7, ...>)



Iteration 2

mapper: (a,<s,10>) (c,<s,5>) (a,<c,8>) (c,<a,12>) (b,<a,11>) (b,<c,14>) (d,<c,7>) (b,<d,13>) (d,<b,15>) edges

reducer: (a,<8>) (c,<5>) (b,<11>) (d,<7>)

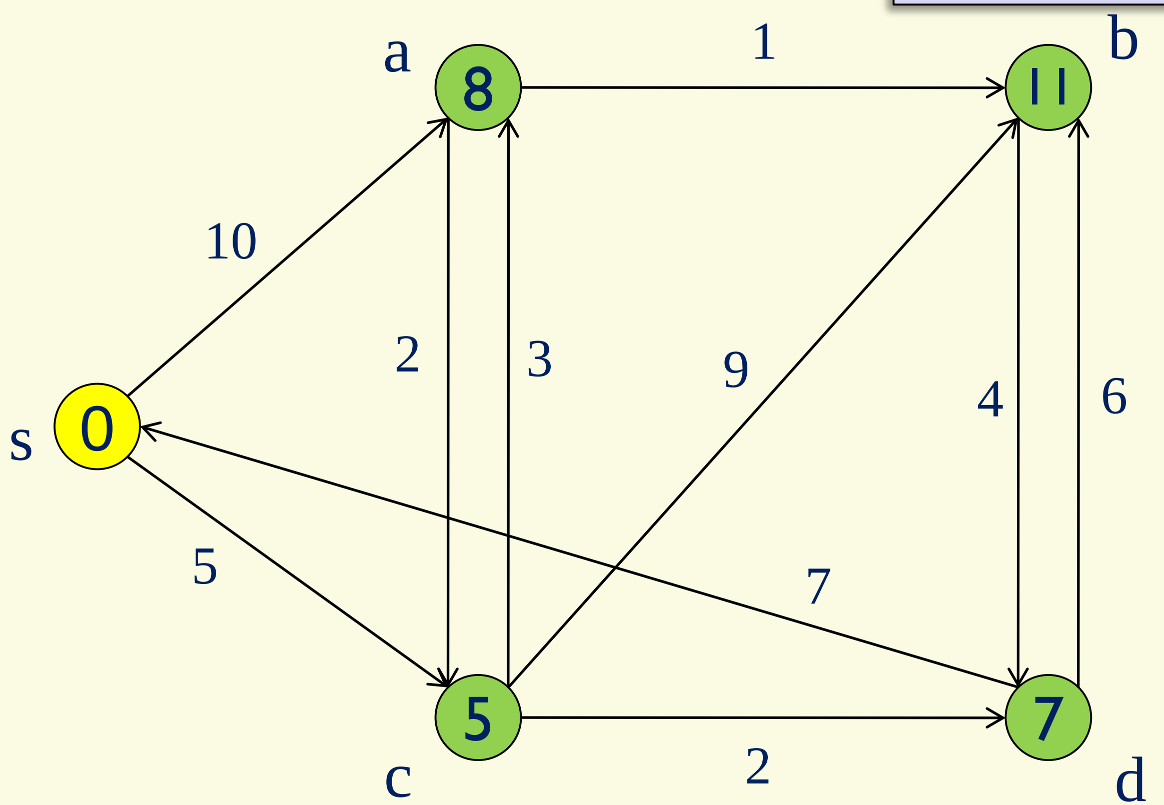


Iteration 3

mapper: (a,<s,10>) (c,<s,5>) (a,<c,8>) (c,<a,9>) (b,<a,11>)
(b,<c,14>) (d,<c,7>) (b,<d,13>) (d,<b,15>) edges

reducer: (a,<8>) (c,<5>) (b,<11>) (d,<7>)

No change: Convergence!



Efficiency?

MapReduce explores all paths in parallel

Each MapReduce iteration advances the “known frontier” by one hop

- Redundant work, since useful work is only done at the “frontier”

Dijkstra’s algorithm is more efficient

- At any step it only pursues edges from the minimum-cost path inside the frontier

skew

Any other sources of inefficiency?

A closer look

Data partitioned parallelism

- ✓ Local computation at each node in mapper, in parallel: attributes of the node, adjacent edges and local link structures
- ✓ Propagating computations: traversing the graph; this may involve iterative MapReduce

Tips:

- ✓ Adjacency lists
- ✓ Local computation in mapper;
- ✓ Pass along partial results via outlinks, keyed by destination node;
- ✓ Perform aggregation in reducer on inlinks to a node
- ✓ Iterate until convergence: controlled by external “driver”
- ✓ pass graph structures between iterations

Needs a way to test for convergence

PageRank

PageRank

The likelihood that page v is visited by a random walk:

$$\alpha\ (1/|V|)\ +\ (1-\alpha)\ \Sigma\_(u \in L(v))\ P(u)/C(u)$$

random jump

following a link from other pages

- ✓ Recursive computation: for each page v in G,
  - compute P(v) by using P(u) for all $u \in L(v)$

until

  - converge: no changes to any P(v)
  - after a fixed number of iterations

How to speed it up?

A MapReduce algorithm

Input: graph G, represented by adjacency lists

- ✓ Node N:
  - Node id: nid n
  - N.rank: the current rank
  - N.AdjList: [m], node id
- ✓ Key: node id n
- ✓ Value of node N:
  - rank: a rank of a node
  - Node N (id, AdjList, etc)
- ✓ Simplified version: Σ_(u $\in$ L(v)) P(u)/C(u)

Assume that $\alpha = 0$

Mapper

Map (nid n, node N)

P(u)/C(u)

- p ← N.rank/|N.AdjList|;
- emit(nid n, N);
- for each m in N.AdjList do
  - emit(nid m, p);

Pass rank to neighbors

Parallel processing

- ✓ all nodes are processed in parallel, each by a mapper
- ✓ Pass PageRank at n to successors of n
- ✓ emit (nid: n, N): preserve graph structure for iterative processing

Local computation in mapper

Reducer

Reduce (nid m, list)

- s ← 0;
- for all p in list do
  - if IsNode(p)
    - then M ← p;
    - else s ← s + p;
- M.rank ← s;
- emit (nid m, node M);

Recover graph structure

Sum up

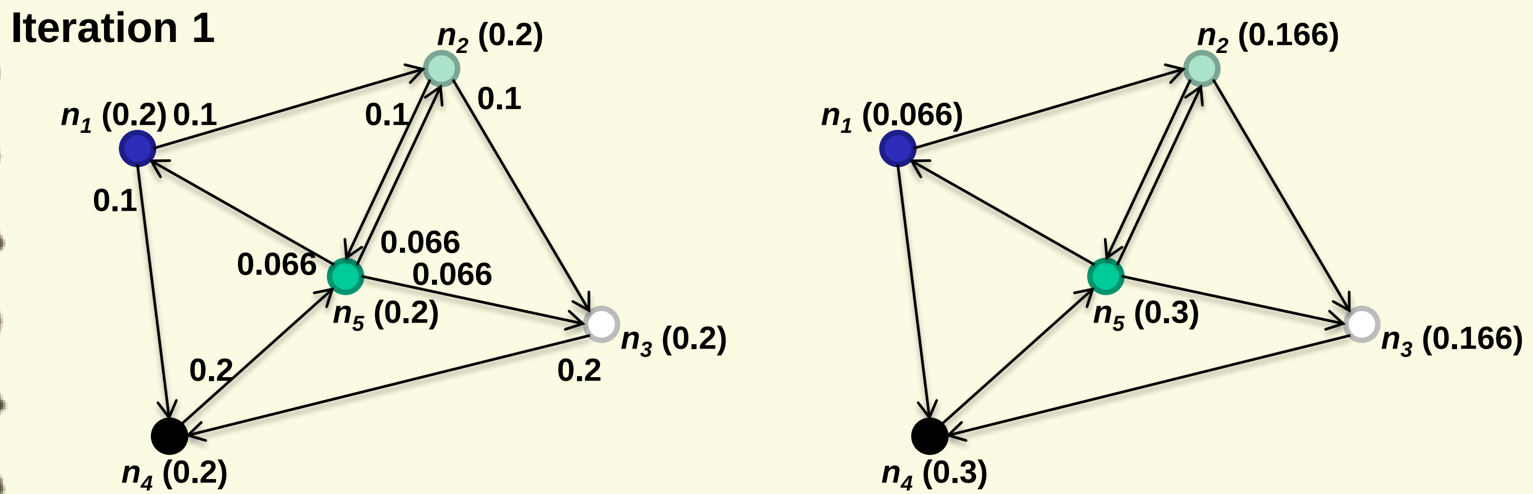
With updated M.rank for this round

✓ list for m: P(u)/C(u) from all predecessors of m

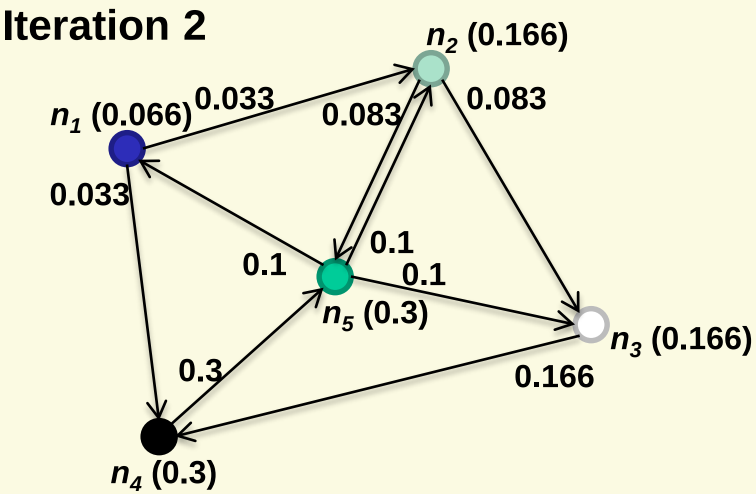
✓ m.rank at the end: Σ_(u ∈ L(v)) P(u)/C(u)

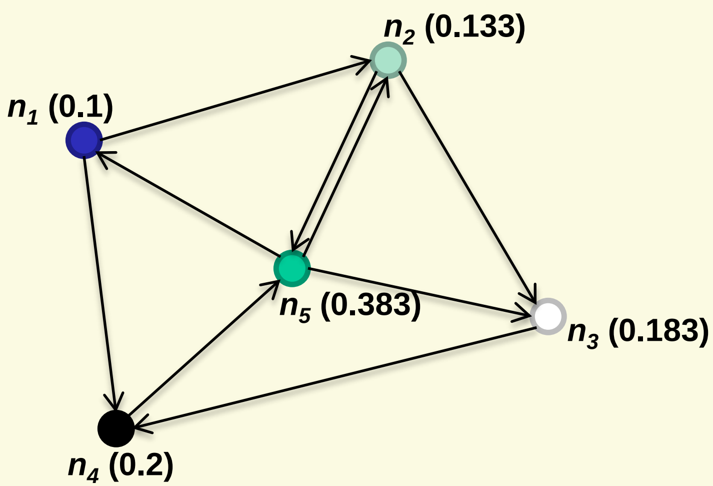
Aggregation in reducer

Sample PageRank Iteration (1)

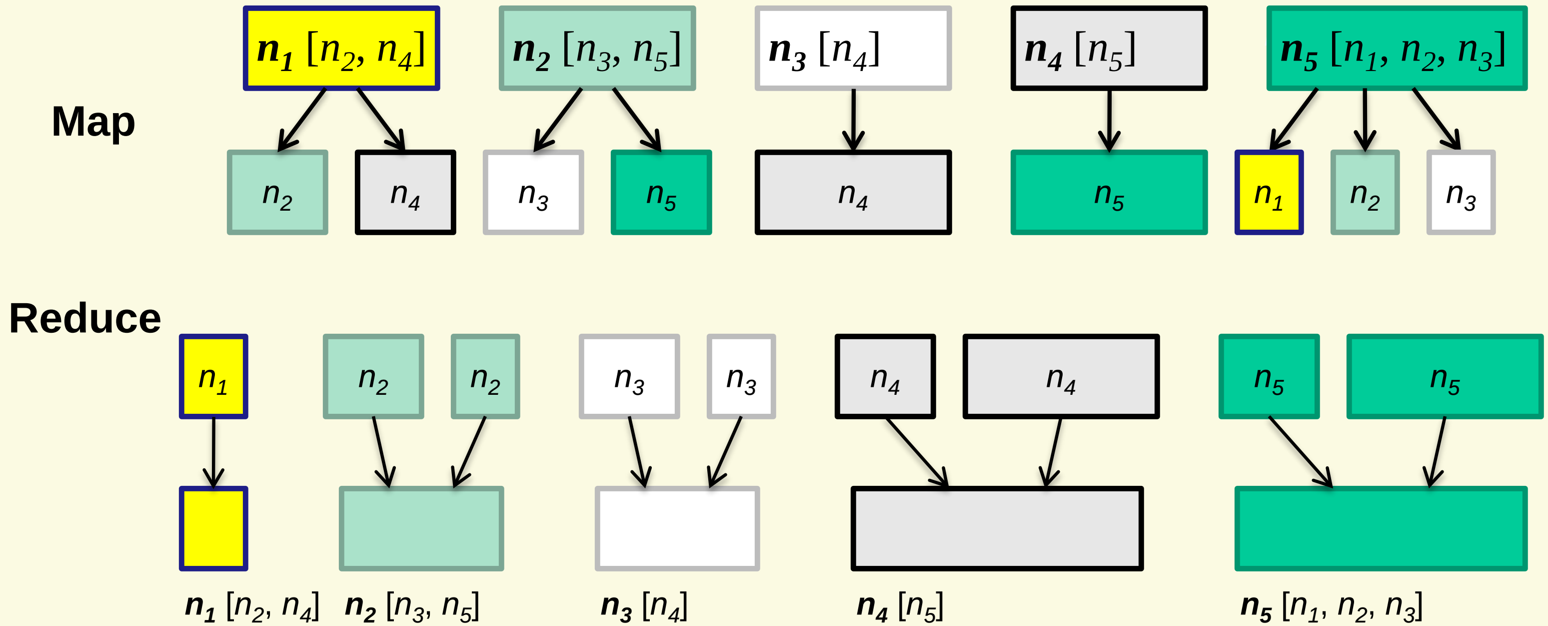


Sample PageRank Iteration (2)





PageRank in MapReduce



Termination control: external driver

Source: Wikipedia (Wave)

Keyword search

Distinct-root trees

- ✓ Input: A list $Q = (k_1, \ldots, k_m)$ of keywords, a directed graph G, and a positive integer D
- ✓ Output: distinct trees that match Q bounded by D

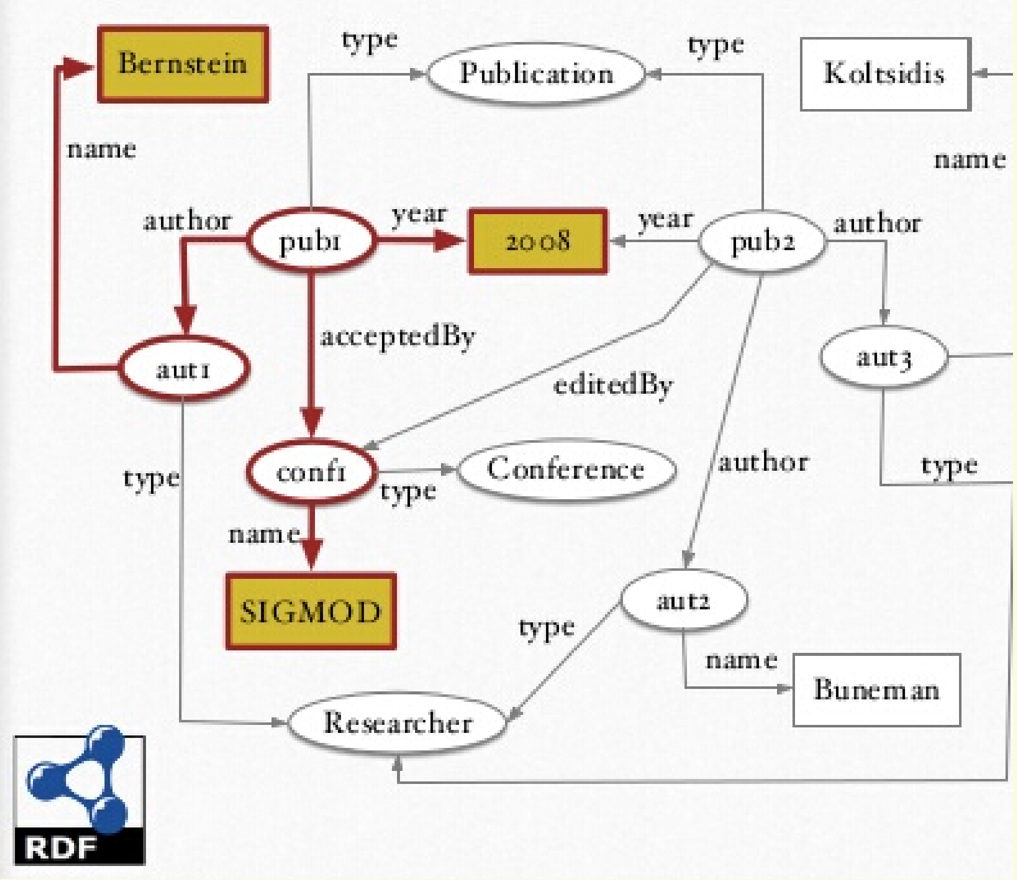
- ✓ Match: a subtree $T = (r, (k_1, p_1, d_1(r, p_1)), \ldots, (k_m, p_m, d_m(r, p_m))$ of G such that
  - each keyword k_i in Q is contained in a leaf p_i of T
  - p_i is closest to r among all nodes that contain k_i
  - the distance from the root r of T to the lead does not exceed D

A simplified version

$k \geq d_j(r, p_j)$: k iterations (termination condition)

Searching citation network

Input Keyword: **Bernstain, 2008, SIGMOD**



An MapReduce algorithm

Input: graph G, represented by adjacency lists

- ✓ Node N:
  - Node id: nid n
  - N.$((K_1, P_1, D_1), \ldots, (K_m, P_m, D_m)$: representing $(n, (k_1, p_1, d_1(n, p_1)), \ldots, (k_m, p_m, d_m(n, p_m))$
  - N.AdjList: [m], node id

- ✓ Key: node id n
- ✓ Preprocessing: N.$((K_1, P_1, D_1), \ldots, (K_m, P_m, D_m)$:
  - $P_1 = \perp$ and $D_m = \infty$ if N does not contain k_m
  - $P_1 = n$ and $D_m = 0$ otherwise

Preprocessing: can be done in MapReduce itself

Mapper

Map (nid n, node N)

- emit(nid n, N);
- for each m in N.AdjList do
  - emit(nid n, (M.(K_1, P_1, D_1+1), …, (K_m, P_m, D_m+1));

m is the node id of node M

Local computation:

✓ Shortcut one node

✓ One hop forward

Contrast this to, e.g., PageRank

Pass information from successors

Reducer

Reduce (nid n, list)

- for i from 1 to m do
  - $p_i \leftarrow$ N. P_i; $d_i \leftarrow$ N. d_i;
- for i from 1 to m do
  - $S_i \leftarrow$ the set of all M.(K_i, P_i, D_i) in list
  - $d_i \leftarrow$ the smallest M.D_i; $p_i \leftarrow$ the corresponding M. D_i;
- for i from 1 to m do
  - N.$P_i \leftarrow p_i$;   N.$D_i \leftarrow d_i$;
- emit (nid n, node N);

N: the node represented by n; must be in list

Group by keyword k_i

Pick the one with the shortest distance to n

✓ Invariant: in iteration j, N.((K_1, P_1, D_1), …, (K_m, P_m, D_m) represents (n, (k_1, p_1, d_1(n, p_1)), …, (k_m, p_m, d_m(n, p_m))

Shortest distances within j hops

Termination and post-processing

Termination: after D iterations, for a given positive integer D

Post-processing: upon termination, for each node n, where

N.((K_1, P_1, D_1), …, (K_m, P_m, D_m)

- If no $P_i = \perp$ for i from 1 to m, then

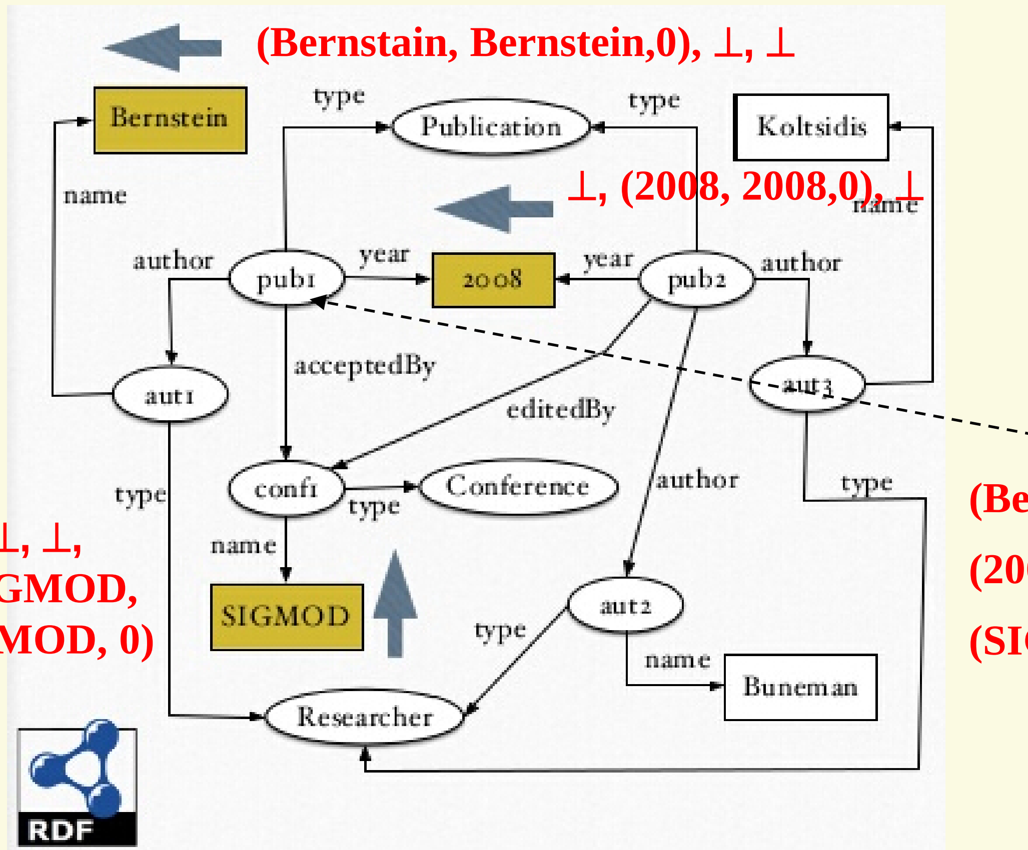
  N.((K_1, P_1, D_1), …, (K_m, P_m, D_m) represents a valid match

  (n, (k_1, p_1, d_1(n, p_1)), …, (k_m, p_m, d_m(n, p_m))

A different way of passing information during traversal

Searching citation network

Input Keyword: **Bernstain, 2008, SIGMOD**



⊥, ⊥,
(SIGMOD,
SIGMOD, 0)

(Bernstain, Bernstein,2)

(2008, 2008,1)

(SIGMOD, SIGMOD,2)

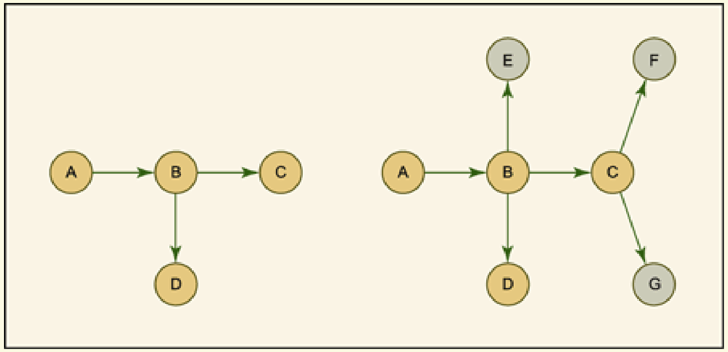
Graph pattern matching by subgraph isomorphism

Graph pattern matching by subgraph isomorphism

- Input: a query Q and a data graph G,
- Output: all the matches of Q in G, i.e, all subgraphs of G that are isomorphic to Q

a bijective function *f* on nodes:

$(u,u') \in Q$ iff $(f(u), f(u')) \in G$



MapReduce?

NP-complete

An MapReduce algorithm

Input: Q, graph G, represented by adjacency lists

- ✓ Node N:
  - Node id: nid n
  - $N.G_d$: the subgraph of G rooted at n, consisting of nodes within d hops of n
  - N.AdjList: [m], node id

d: the radius of Q

- ✓ Key: node id n
- ✓ Preprocessing: for each node n, computes $N.G_d$
  - A MapReduce algorithm of d iterations
  - adjacency lists are only used in the preprocessing step

Two MapReduce steps: preprocessing, and computation

Algorithm

Invoke any algorithm for subgraph isomorphism: VF2, Ullman

Map (nid n, node N)

- compute all matches S of Q in N.G_d
- emit(1, S);

not necessary; just to eliminate duplicates

reduce (1, list)

- M ← the union of all sets in list
- emit(M, 1);

Yes, data locality

✓ Show the correctness? All and only isomorphic mappings?

✓ Parallel scalability? The more processors, the faster?

Yes, as long as the number of processors does not exceed the number of nodes of G

Lot of redundant computations

Just a conceptual level evaluation

Pitfalls of MapReduce

- ✓ No schema: schema-free
- ✓ No index: index-free
- ✓ A single dataflow: a single input and a single output
- ✓ No high-level languages: no SQL
- ✓ No support for incremental computation: redundant computation
- ✓ The MapReduce model does not provide a mechanism to maintain global data structures that can be accessed and updated by all mappers and reducers
- ✓ Low efficiency: I/O optimization, utilization, no pipelining, Map/Reduce bottleneck; no specific execution plan; batch nature.

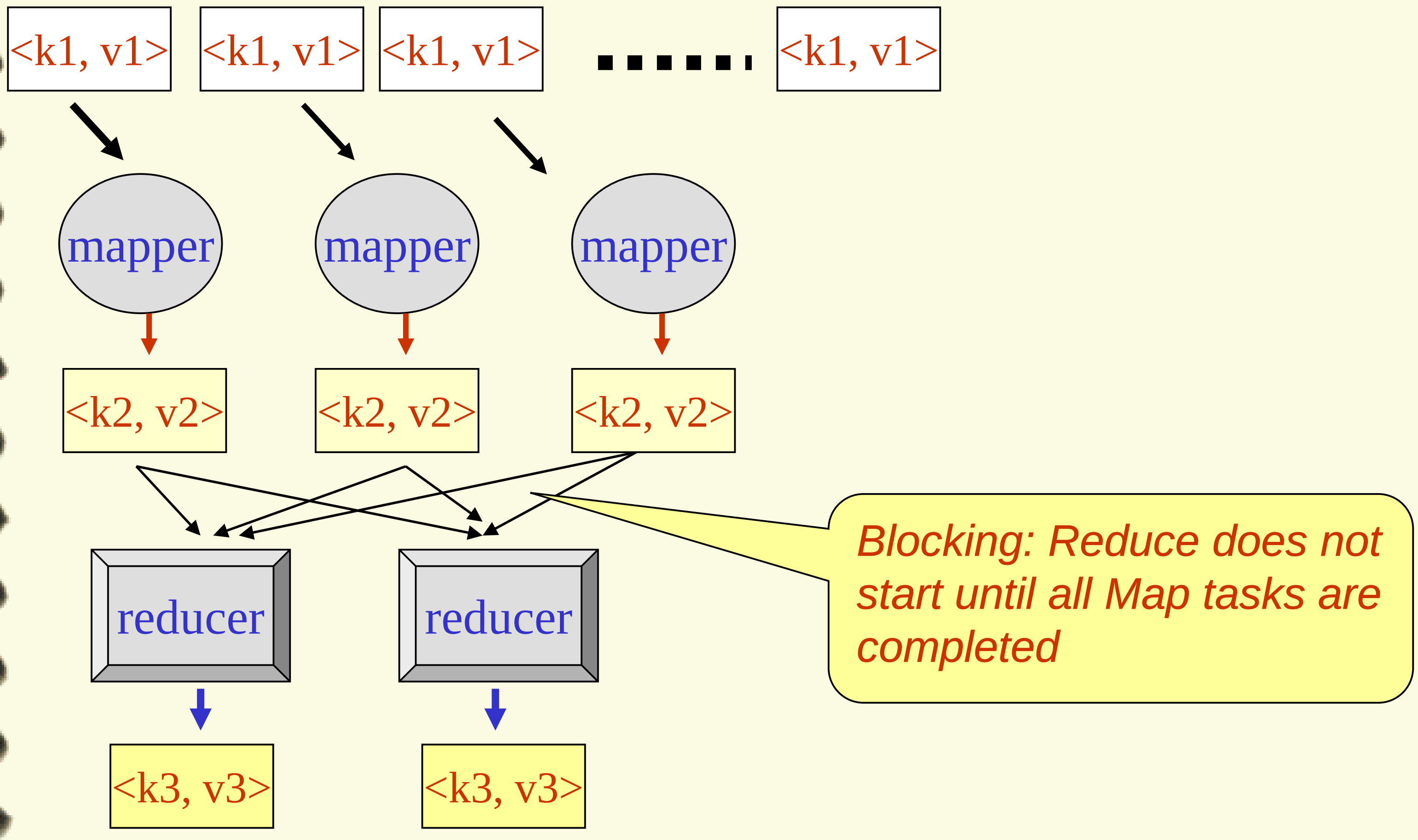
Why bad?

Inefficient to do join

Functional programming

Why low efficiency?

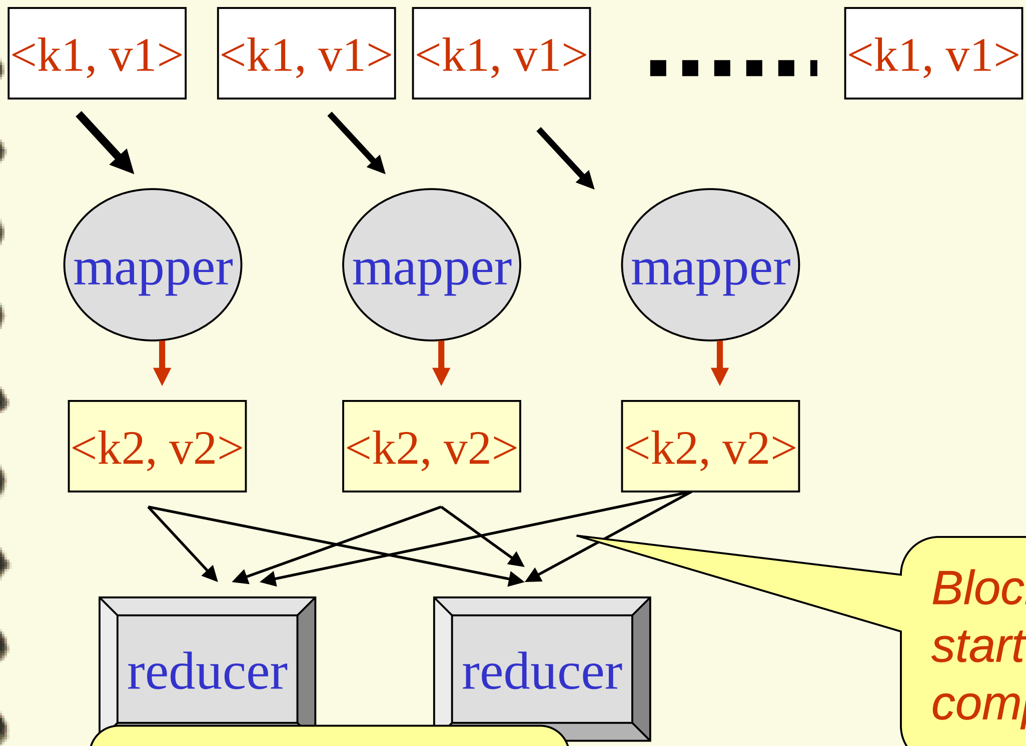
Inefficiency of MapReduce



Despite these, MapReduce is popular in industry

Parallel models beyond MapReduce

Inefficiency of MapReduce



Blocking: Reduce does not start until all Map tasks are completed

Intermediate results shipping: all to all

Write to disk and read from disk in each step, although the data does not change in loops

Other reasons?

The need for parallel models beyond MapReduce

- MapReduce:
  - Inefficiency: blocking, intermediate result shipping (all to all); write to disk and read from disk in each step, even for invariant data in a loop
  - Does not support iterative graph computations:
    - External driver
    - No mechanism to support global data structures that can be accessed and updated by all mappers and reducers
  - Support for incremental computation?
  - Have to re-cast algorithms in MapReduce, hard to reuse existing (incremental) algorithms
  - General model, not limited to graphs

Can we do better for graph algorithms?

Summing up

Summary and review

- What is the MapReduce framework?
- How to develop graph algorithms in MapReduce?
  - Graph representation
  - Local computation in mapper
  - Aggregation in reducer
  - Termination
- Graph algorithms in MapReduce may not be efficient. Why?
- Develop your own graph algorithms in MapReduce. Give correctness proof, complexity analysis and performance guarantees for your algorithms

Papers for you to review

- W. Fan, F. Geerts, and F. Neven. *Making Queries Tractable on Big Data with Preprocessing*, VLDB 2013
- Y. Tao, W. Lin. X. Xiao. Minimal MapReduce Algorithms (MMC) http://www.cse.cuhk.edu.hk/~taoyf/paper/sigmod13-mr.pdf
- L. Qin, J. Yu, L. Chang, H. Cheng, C. Zhang, Xuemin Lin: Scalable big graph processing in MapReduce. SIGMOD 2014. http://www1.se.cuhk.edu.hk/~hcheng/paper/SIGMOD2014qin.pdf
- W. Lu, Y. Shen, S. Chen, B. Ooi: Efficient Processing of k Nearest Neighbor Joins using MapReduce. PVLDB 2012. http://arxiv.org/pdf/1207.0141.pdf
- V. Rastogi, A. Machanavajjhala, L. Chitnis, A. Sarma: Finding connected components in map-reduce in logarithmic rounds. ICDE 2013http://arxiv.org/pdf/1203.5387.pdf