Generalizing Map-Reduce

The Computational Model

Map-Reduce-Like Algorithms

Computing Joins

Overview

- There is a new computing environment available:
  - Massive files, many compute nodes.
- Map-reduce allows us to exploit this environment easily.
- But not everything is map-reduce.
- What else can we do in the same environment?

Files

- Stored in dedicated file system.
- Treated like relations.
  - Order of elements does not matter.
- Massive *chunks* (e.g., 64MB).
- Chunks are replicated.
- Parallel read/write of chunks is possible.

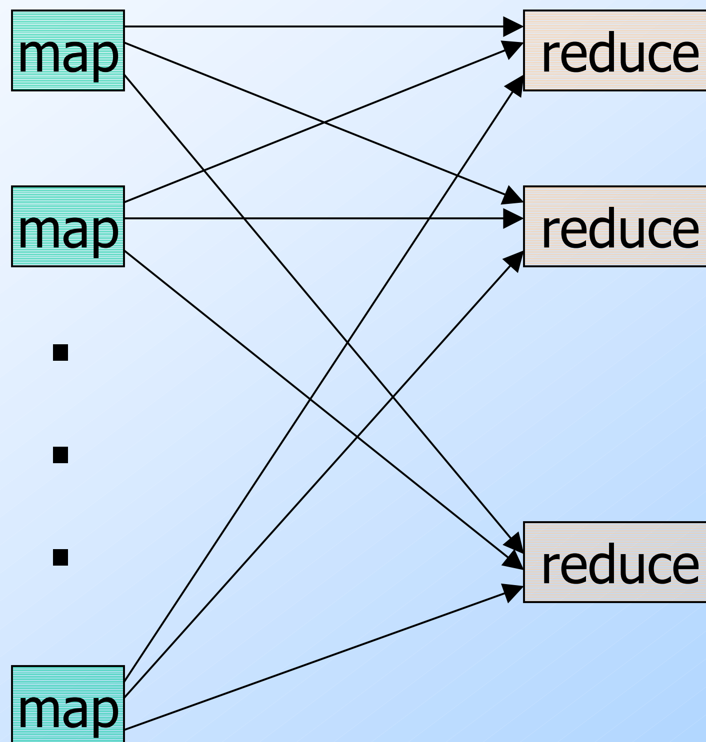
Processes

- Each process operates at one node.
- "Infinite" supply of nodes.
- Communication among processes can be via the file system or special communication channels.
  - Example: Master controller assembling output of Map processes and passing them to Reduce processes.

Algorithms

- An algorithm is described by an acyclic graph.
  1. A collection of processes (nodes).
  2. Arcs from node *a* to node *b*, indicating that (part of) the output of *a* goes to the input of *b*.

Example: A Map-Reduce Graph



Algorithm Design

- Goal: Algorithms should exploit as much parallelism as possible.
- To encourage parallelism, we put a limit s on the amount of input or output that any one process can have.
  - s could be:
    - What fits in main memory.
    - What fits on local disk.
    - No more than a process can handle before cosmic rays are likely to cause an error.

Cost Measures for Algorithms

1. *Communication cost* = total I/O of all processes.
2. *Elapsed communication cost* = max of I/O along any path.
3. (*Elapsed*) *computation costs* analogous, but count only running time of processes.

Example: Cost Measures

- For a map-reduce algorithm:
  - Communication cost = input file size + 2 $\times$ (sum of the sizes of all files passed from Map processes to Reduce processes) + the sum of the output sizes of the Reduce processes.
  - Elapsed communication cost is the sum of the largest input + output for any map process, plus the same for any reduce process.

What Cost Measures Mean

- Either the I/O (communication) or processing (computation) cost dominates.
  - Ignore one or the other.
- Total costs tell what you pay in rent from your friendly neighborhood cloud.
- Elapsed costs are wall-clock time using parallelism.

Join By Map-Reduce

- Our first example of an algorithm in this framework is a map-reduce example.
- Compute the natural join $R(A,B) \bowtie S(B,C)$.
- R and S each are stored in files.
- Tuples are pairs (a,b) or (b,c).

Map-Reduce Join – (2)

- Use a hash function *h* from B-values to 1..*k*.
- A Map process turns input tuple R(a,b) into key-value pair (b,(a,R)) and each input tuple S(b,c) into (b,(c,S)).

Map-Reduce Join – (3)

- Map processes send each key-value pair with key *b* to Reduce process h(b).
  - Hadoop does this automatically; just tell it what *k* is.
- Each Reduce process matches all the pairs (b,(a,R)) with all (b,(c,S)) and outputs (a,b,c).

Cost of Map-Reduce Join

- Total communication cost = $O(|R|+|S|+|R \bowtie S|)$.
- Elapsed communication cost = $O(s)$.
  - We're going to pick k and the number of Map processes so I/O limit s is respected.
- With proper indexes, computation cost is linear in the input + output size.
  - So computation costs are like comm. costs.

Three-Way Join

- We shall consider a simple join of three relations, the natural join $R(A,B) \bowtie S(B,C) \bowtie T(C,D)$.
- One way: cascade of two 2-way joins, each implemented by map-reduce.
- Fine, unless the 2-way joins produce large intermediate relations.

Example: Large Intermediate Relations

- A = "good pages"; B, C = "all pages"; D = "spam pages."
- R, S, and T each represent links.
- 3-way join = "path of length 3 from good page to spam page.
- R ⋈ S = paths of length 2 from good page to any; S ⋈ T = paths of length 2 from any page to spam page.

Another 3-Way Join

- Reduce processes use hash values of entire S(B,C) tuples as key.
- Choose a hash function h that maps B- and C-values to k buckets.
- There are k^2 Reduce processes, one for each (B-bucket, C-bucket) pair.

Mapping for 3-Way Join

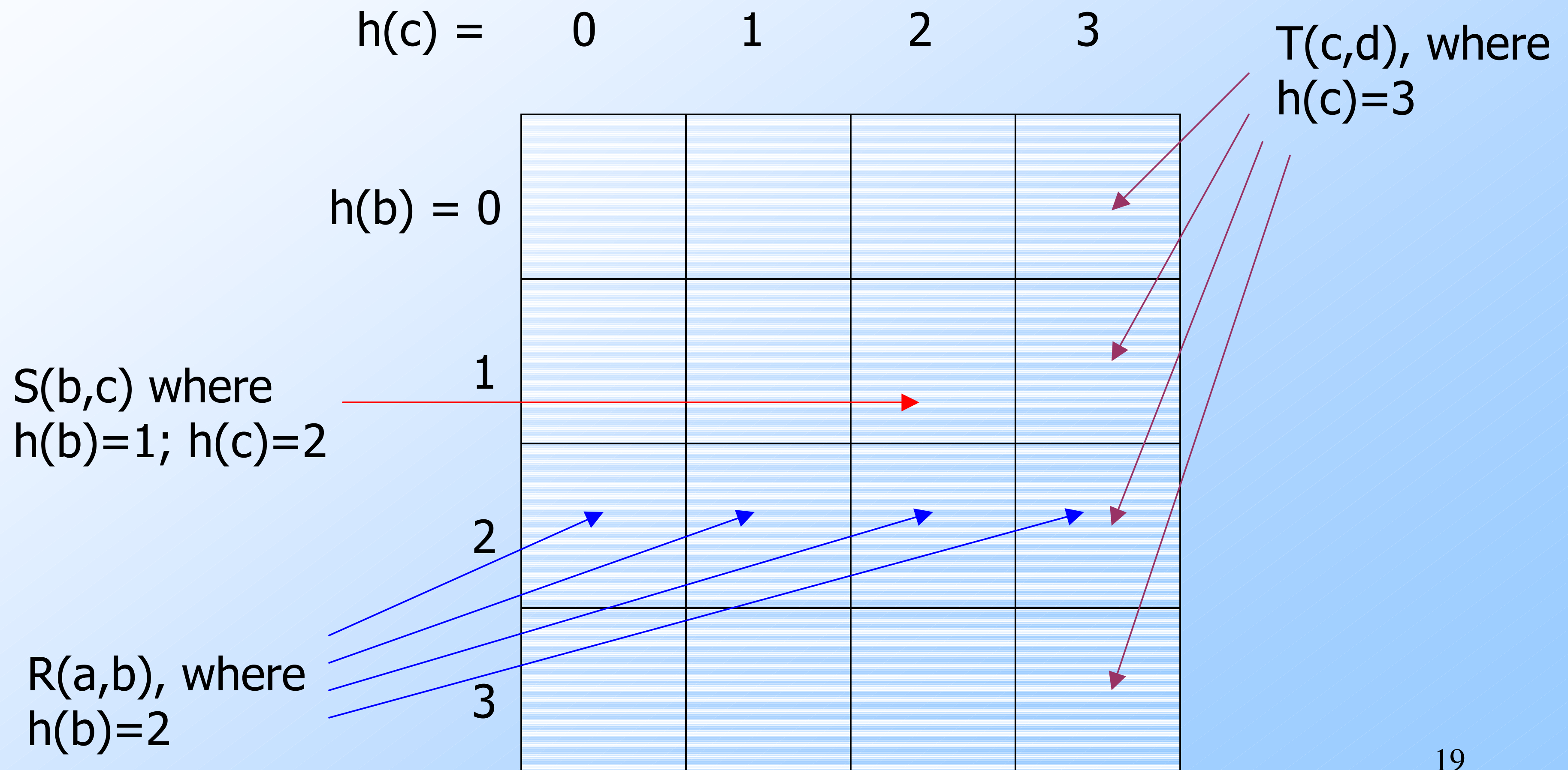
- We map each tuple S(b,c) to ((h(b), h(c)), (S, b, c)).
- We map each R(a,b) tuple to ((h(b), y), (R, a, b)) for all $y = 1, 2,\ldots,k$.
- We map each T(c,d) tuple to ((x, h(c)), (T, c, d)) for all $x = 1, 2,\ldots,k$.

Aside: even normal map-reduce allows inputs to map to several key-value pairs.

Keys

Values

Assigning Tuples to Reducers



Job of the Reducers

- Each reducer gets, for certain B-values *b* and C-values *c* :
  1. All tuples from R with B = *b*,
  2. All tuples from T with C = *c*, and
  3. The tuple S(b,c) if it exists.
- Thus it can create every tuple of the form (a, b, c, d) in the join.

3-Way Join and Map-Reduce

- This algorithm is not exactly in the spirit of map-reduce.
- While you could use the hash-function h in the Map processes, Hadoop normally does the hashing of keys itself.

3-Way Join/Map-Reduce – (2)

- But if you Map to attribute values rather than hash values, you have a subtle problem.
- Example: R(a, b) needs to go to all keys of the form (b, y), where *y* is any C-value.
  - But you don't know all the C-values.

Semijoin Option

- A possible solution: first semijoin – find all the C-values in S(B,C).
- Feed these to the Map processes for R(A,B), so they produce only keys (b, y) such that y is in $\pi_C(S)$.
- Similarly, compute $\pi_B(S)$, and have the Map processes for T(C,D) produce only keys (x, c) such that x is in $\pi_B(S)$.

Semijoin Option – (2)

- Problem: while this approach works, it is not a map-reduce process.
- Rather, it requires three layers of processes:
  1. Map S to $\pi_B(S)$, $\pi_C(S)$, and S itself (for join).
  2. Map R and $\pi_B(S)$ to key-value pairs and do the same for T and $\pi_C(S)$.
  3. Reduce (join) the mapped R, S, and T tuples.