Parallel Databases

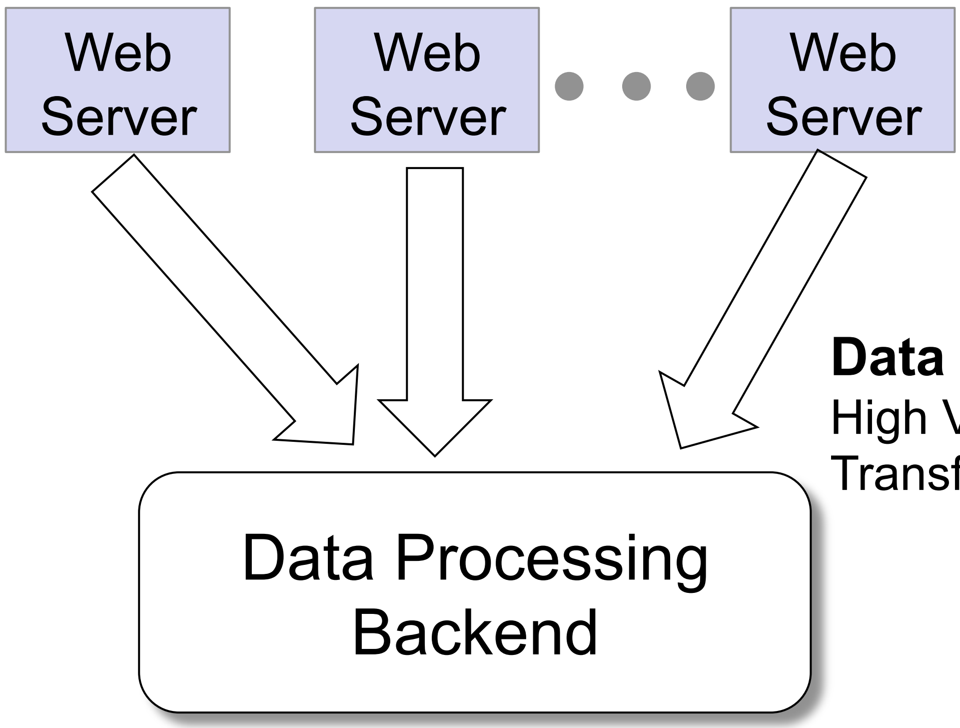
Prof. Yanlei Diao

Slides Courtesy of R. Ramakrishnan and J. Gehrke

A Data Warehouse for A Social Network

User profiles:
100 Million users
Each with profile, pics, postings,…

Click Streams:
1 billion rows/day
5-10 TB/day



Data Loading:
High Volume + Transformation

Quick lookups and updates:
Update your own profile, read friends' profiles, write msgs,…

Analysis Queries:
Ad targeting, fraud detection, resource provisioning…

Fun (Old) Numbers about Facebook

500 million active users

9.5% Internet traffic

>30,000 servers

Initial software:
PHP + MySQL cluster
+ Memcached

One of the largest MySQL cluster

>4.5 billion msgs/day
>15 TB click logs/day

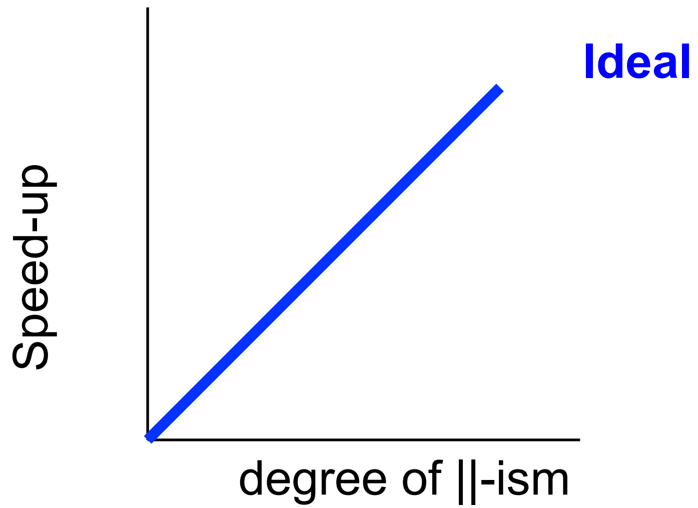
Stores >20 billion photos, and serves 1 million img/sec.

Parallel Databases 101

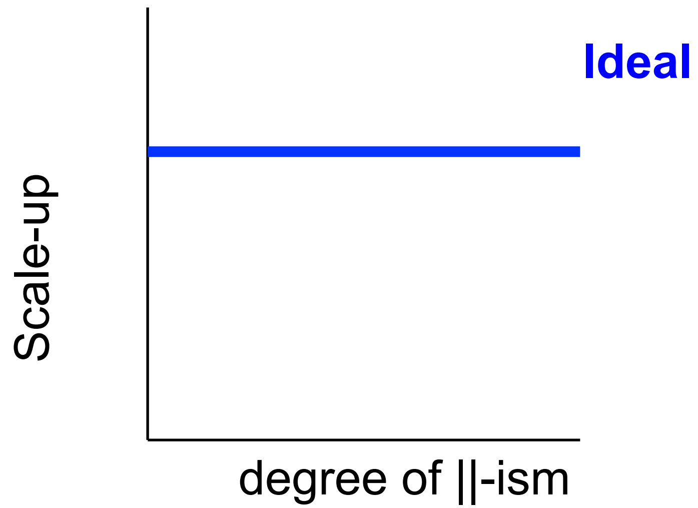
- Rise of parallel databases: late 80' s
- Architecture: shared-nothing systems
  - A number of nodes connected by fast Ethernet switches
  - But used special-purpose hardware (costly, slow to evolve)
  - Small scale (hence did not focus on fault tolerance)
- Typical systems
  - Gamma: U. of Wisconsin Madison
  - TeraData: Wal-Mart's 7.5TB sales data in hundreds of machines
  - Tandem
  - IBM / DB2
  - Informix…

Some Parallel (||) Terminology

- Speed-Up
  - Holds the problem size, grows the system
  - Reports serial time/||-time
  - Ideally, linear

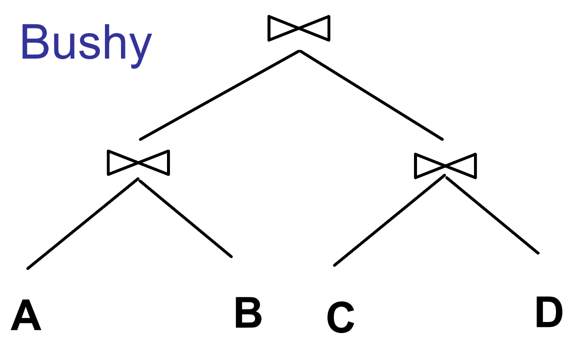


- Scale-Up
  - Grows both the system and the problem, reports running time
  - Ideally, constant



Different Types of DBMS ||-ism

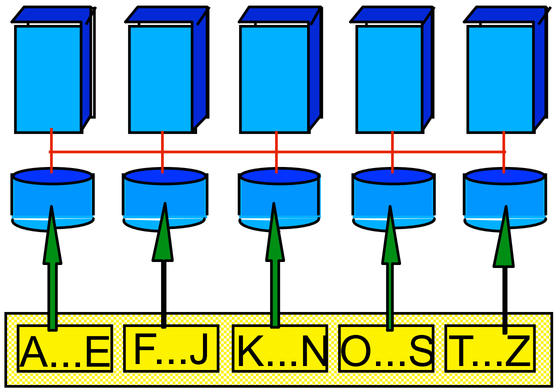
- Partitioned (data) parallelism
  - Partition data over all nodes, get the nodes working to compute a given operation (scan, sort, join)
- Pipelined parallelism
  - A chain of operators O_1, O_2, …,O_k run in parallel, with O_1 working on tuple t_n, O2 on t_(n-1), … O_k on t_(n-k+1)
  - Can run these operators on different nodes
  - Some operators break pipelining, e.g. sort, hash
- Independent operators
  - Consider bushy query plans
  - A join B, C join D are independent
- We'll focus on partitioned ||-ism



Data Partitioning Schemes

Partitioning a table:

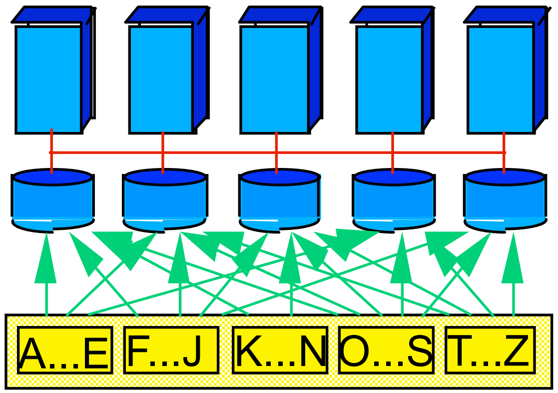
Range



Good for seq. scan, associative search, sorting

Can have data skew

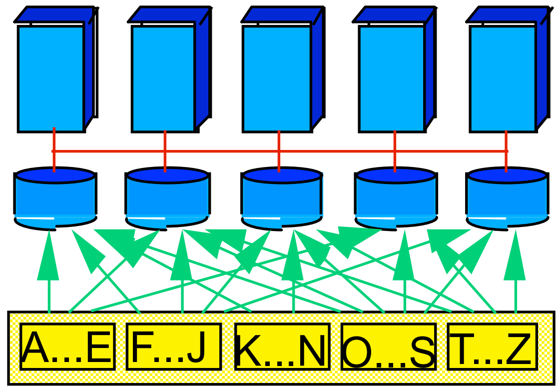
Hash



Good for seq. scan, equality search, equijoins if they match the hash attr

Bad for range search, or operations that do not match the hash attr;
Can also have data skew

Round Robin



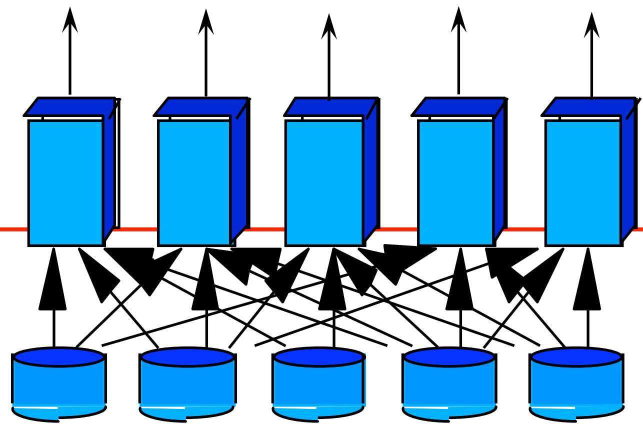
Good for seq. scan

Useless for other query operations

Parallel Scans and Associative Accesses

- Scan in parallel, and merge.
- Selection may not require all sites for range or hash partitioning.
  - Want to restrict selection to a few nodes, or restrict "small" queries to a few nodes.
  - Indexes can be built at each partition.
  - What happens during data inserts and lookups?

Parallel Sorting

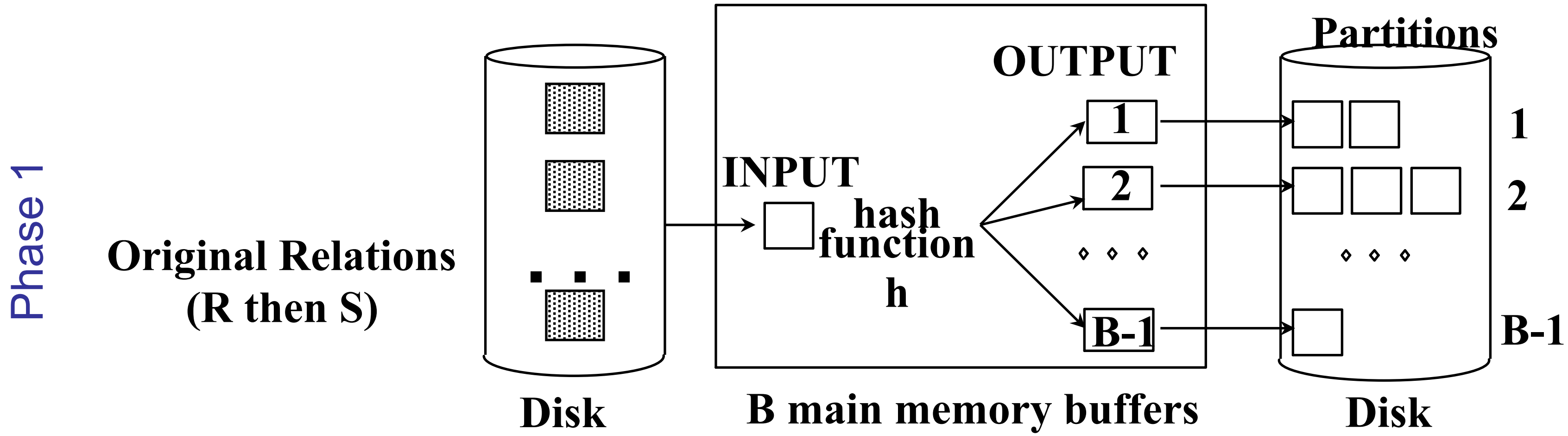


- Some record in the history:
  - 8.5 Gb/minute, shared-nothing; Datamation benchmark in 2.41 secs (UCB students! http://now.cs.berkeley.edu/NowSort/)
- Idea:
  - Scan in parallel, and range-partition as you go.
  - As tuples come into each node, begin "local" sorting
  - Resulting data is sorted, and range-partitioned.
  - Problem: ***skew!***
  - Solution: "sample" the data at start to determine partition points.

Partitioned Join

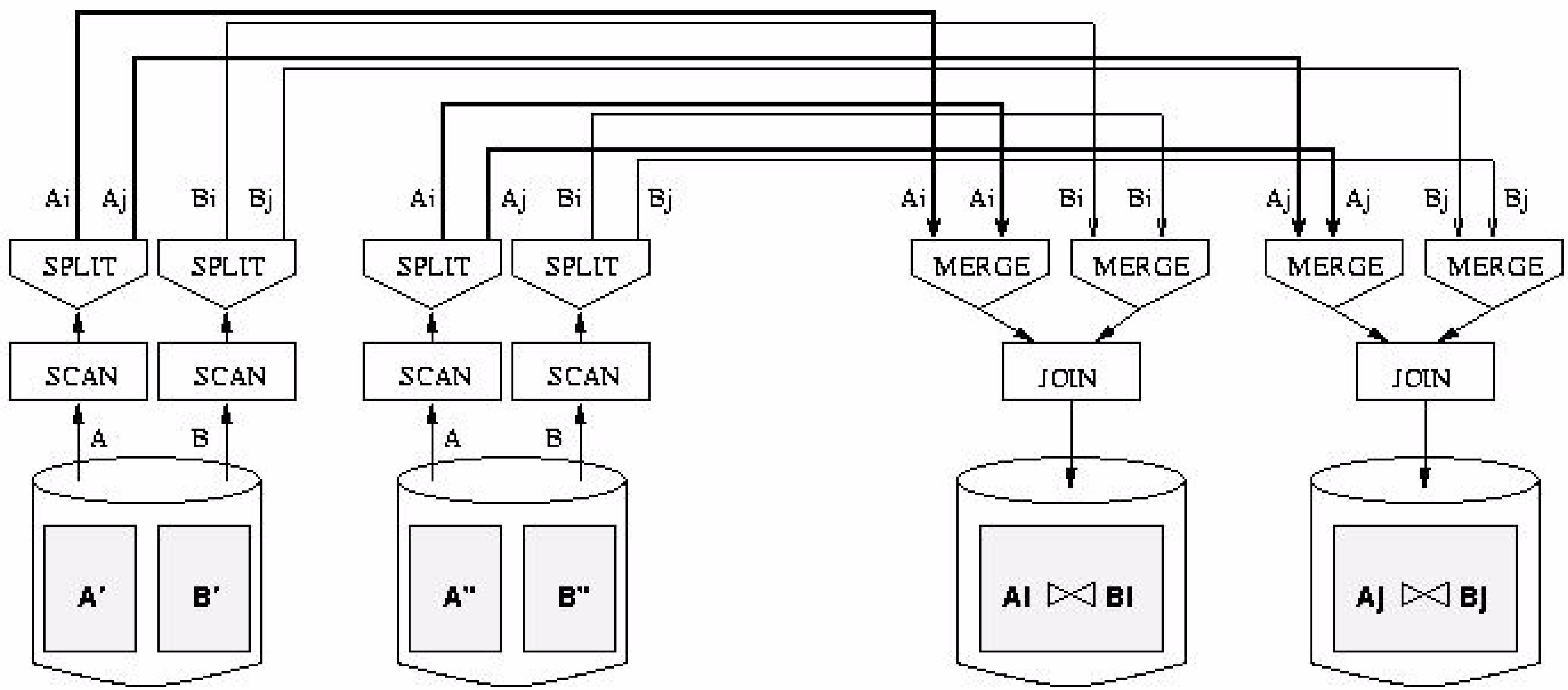
- For equi-joins, *partition* the two input relations across all nodes, and compute the join locally at each processor.
- Can use either *range partitioning* or *hash partitioning*., on the join attribute
  - R and S each are partitioned into n partitions, denoted R_0, R_1, ..., R_{n-1} and S_0, S_1, ..., S_{n-1}.
  - Partitions R_i and S_i are sent to node i.
  - Each node locally computes the join using any method.

Parallel Hash Join



- In first phase, partitions get distributed to different nodes:
  - A good hash function *automatically* distributes work evenly!
- Do second phase at each node.
- Almost always the winner for equi-join.

Dataflow Network for Parallel Join

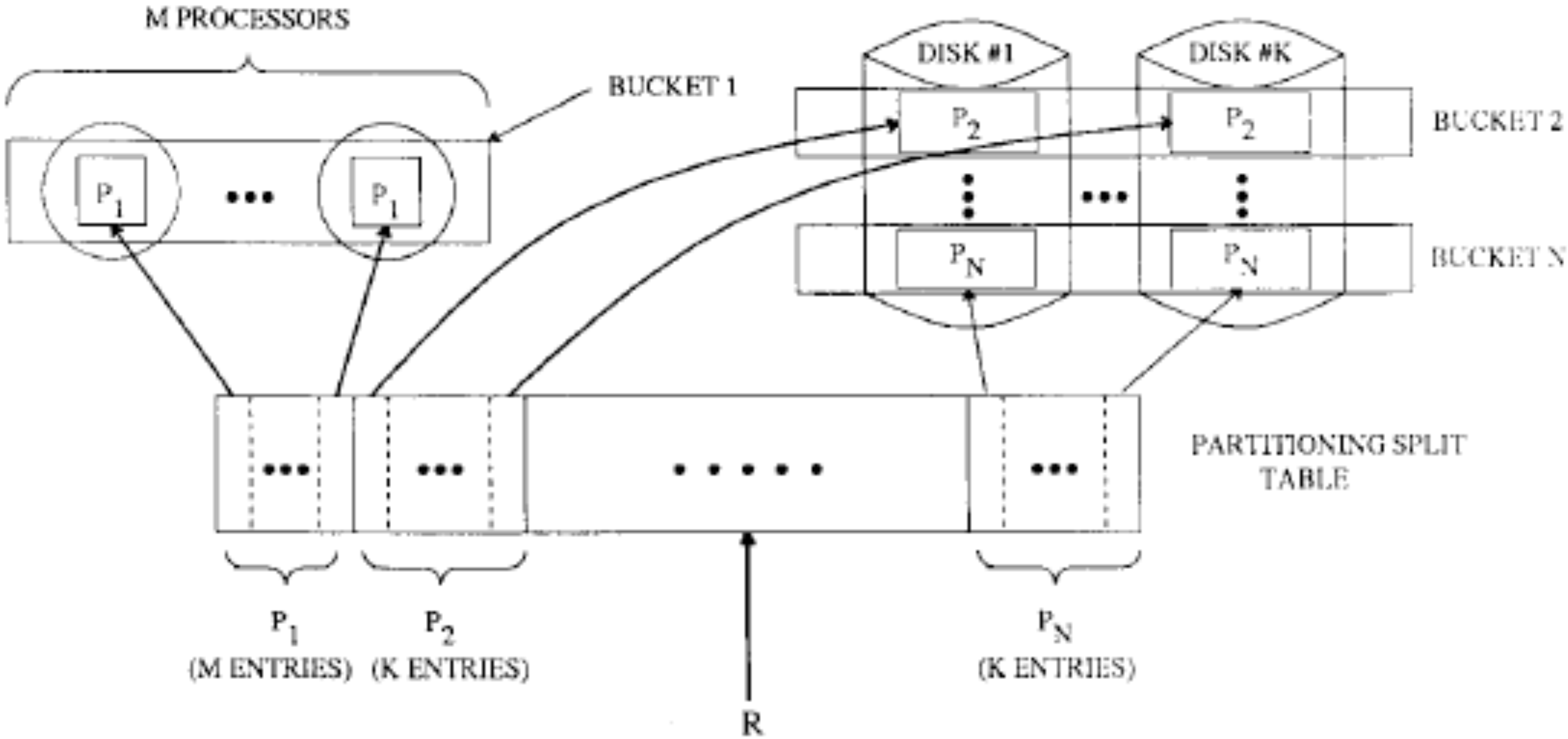


- Good use of split/merge makes it easier to build parallel versions of sequential join code.

Parallel Hybrid Hash Join

- Run the analysis of hybrid hash join as before, but using the aggregate memory B^*, to determine the number of *logical* buckets N; hash fn h_1 maps tuples to buckets.
- The in-memory join, $R_0 \triangleright\triangleleft S_0$, is partitioned over M processors using hash fn h_2.
- Disk resident buckets, R_i and S_i (i>0), are partitioned over K disks using h_3.
- For each subsequent bucket i (i>0), partition $R_i \triangleright\triangleleft S_i$ over M processors using h_2.

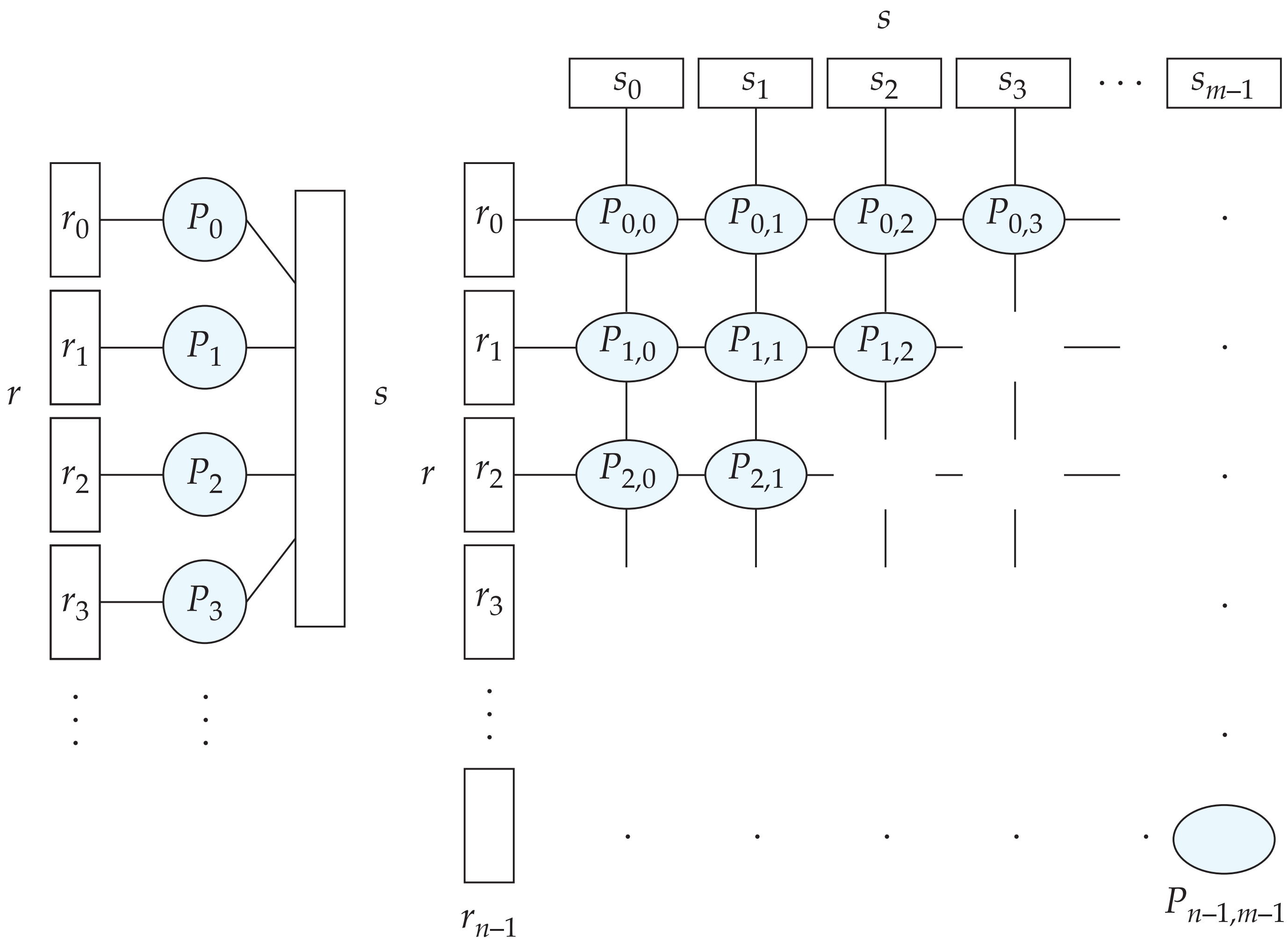
Parallel Hybrid Hash Join



Fragment-and-Replicate Join

- Partitioning not possible for some join conditions
  - E.g., non-equijoin conditions, such as R.A > S.B.
- Use the **fragment and replicate** technique
  - 1) Special case: only one relation is partitioned
    - *R* is partitioned; any partitioning technique can be used.
    - The other relation, *S*, is replicated across all the nodes.
    - Node *i* then locally computes the join of R_i with all of S using any join technique.
    - Works well when *S* is small.
  - 2) General case: both *R* and *S* are partitioned
    - Need to replicate all *R* partitions or all *S* partitions
    - Depicted on the next slide

Illustrating Fragment-and-Replicate Join



(a) Asymmetric fragment and replicate

(b) Fragment and replicate

Parallel Aggregates

- For each aggregate function, need a decomposition:
  - **Distributive:** **count**(S) = Σ **count**(s(i)), ditto for **sum**()
  - **Algebraic:** **avg**(S) = (Σ **sum**(s(i))) / Σ **count**(s(i))
  - **Holistic**: e.g., median, quantiles
- For group-by aggregation:
  - How would you implement parallel group by?
  - How do you add aggregation to each group?

Questions