Parallel Query Processing

Alvin Cheung

Fall 2022

Reading: R&G Ch 22.1-22.4

A little history

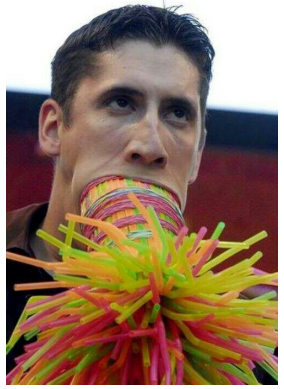
- Relational revolution
  - 1970's
  - Single machine: declarative set-oriented primitives
- Parallel relational database systems
  - 1980's
  - Insight: can parallelize declarative queries!
  - Multiple commodity machines
- "Big Data": MapReduce, Spark, etc.
  - Mid 2000's and continuing
  - From multiple machines to thousands of machines and beyond

Why Parallelism?

- Scan 100TB
  - At 0.5 GB/sec (see lec 3 on files):
    ~200,000 sec = ~2.31 days

Why Parallelism? Cont.

- Scan 100TB
  - At 0.5 GB/sec (see lec 3 on files):
    ~200,000 sec = ~2.31 days
- Run it 100-way parallel:
  - 2,000 sec = 33 minutes
- 1 big problem = many small problems
  - Trick: make them independent
  - Each proceeds at their own pace
  - Thankfully, most rel. operators are amenable to this

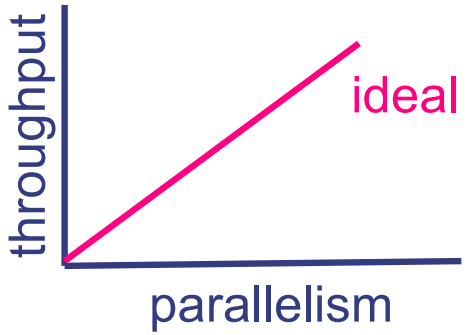


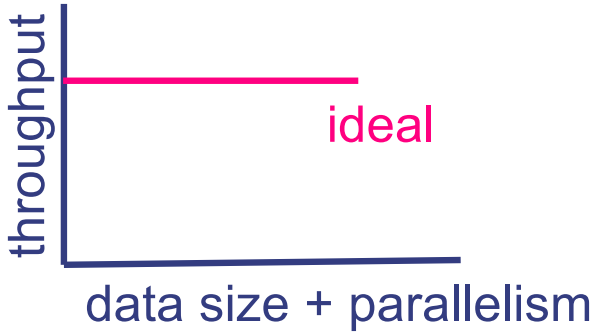
Two Metrics to Shoot For

- Recall: throughput = txns/sec supported

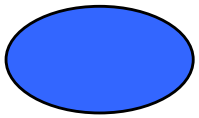
- Speed-up
  - Increase HW
  - Fix workload
- Scale-up
  - Increase HW
  - Increase workload

parallelism

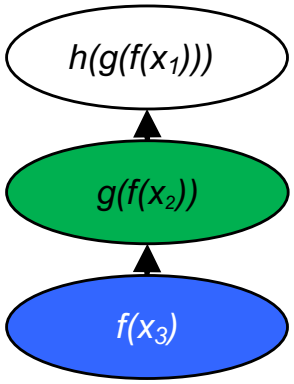




Roughly 2 Kinds of Parallelism



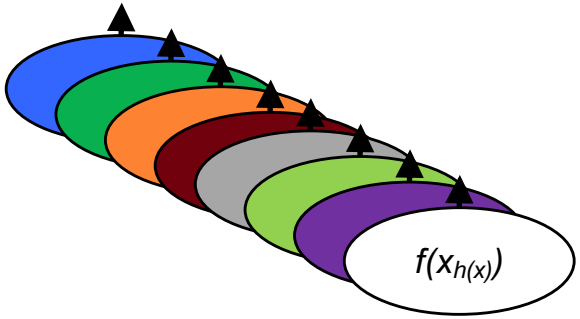
: any sequential program,
e.g. a relational operator



Pipeline
scales up to pipeline depth

Each program performs different operations on different data items in parallel

MIMD



Partition
scales up to amount of data

Each program applies the same operation on different data items in parallel

SIMD

Can be combined!

We'll get more refined soon.

Particularly Easy for Databases!

- Lots of Data & Parallelizable Operations:
  - Batch operations on sets of data (relations)
  - Pre-existing divide-and-conquer algorithms
  - Natural pipelining w/ iterator model

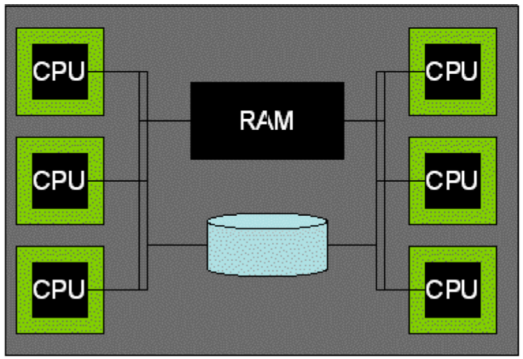
- Declarative languages
  - Can adapt the parallelism strategy to the task and the hardware
  - All without changing the program (i.e., SQL)!
    - Codd's Physical Data Independence

- These insights emerged from the parallel dbms work in the 80's
  - Reimagined or re-learned in the context of "Big Data" in the 2000s

Parallel Architectures

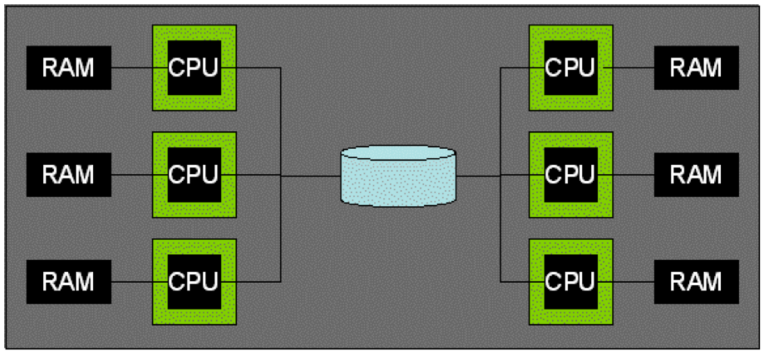
Shared Memory

(Similar to modern computers)



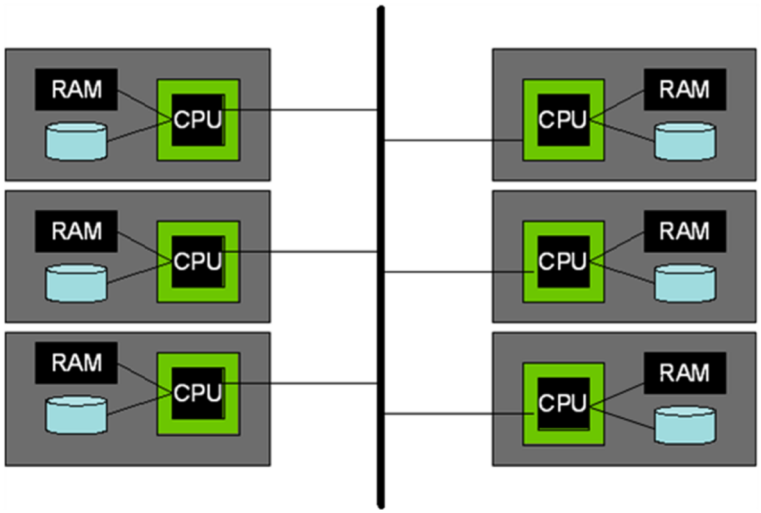
Shared Disk

(Usually w/ some networked file system)



Shared Nothing

(cluster)



cloud computing!

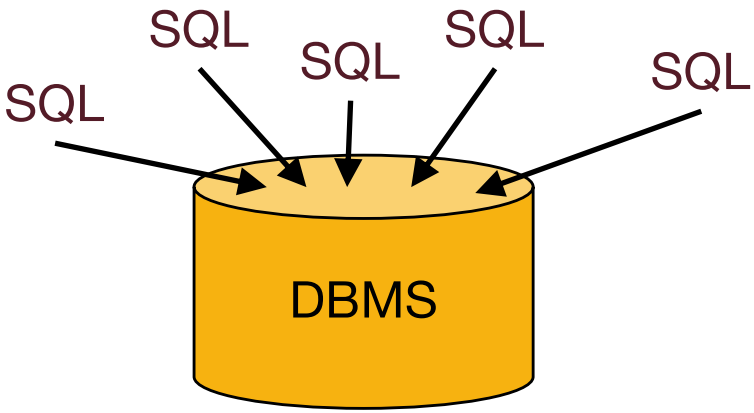
Shared Nothing

- We will focus on Shared Nothing here
  - It's the most common
    - DBMS, web search, big data, machine learning, …
  - Runs on commodity hardware
  - Scales up with data
    - Just keep putting machines on the network!
  - Does not rely on HW to solve problems
    - Good for helping us understand what's going on

Kinds of Query Parallelism: Inter vs. Intra

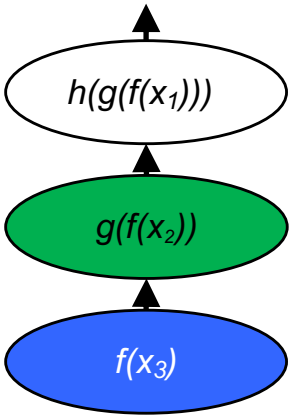
i.e. many queries running on Amazon

- Inter-query (parallelism across queries)
  - Each query runs on a separate processor
    - Single thread (no parallelism) per query
  - Does require parallel-aware concurrency control
    - Will discuss later
  - If many are read-only, easily get good performance



Intra Query – Inter-operator

- Intra-query (within a single query)
  - Inter-operator (between operators)

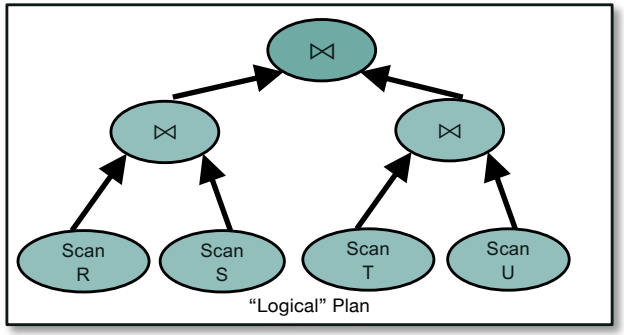


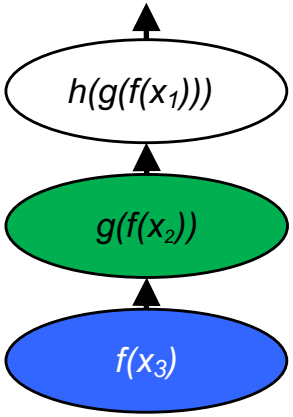
Same tuples get processed by different operators "in a pipeline"

Example: Pipeline Parallelism

Intra Query – Inter-operator Part 2

- Intra-query
  - Inter-operator



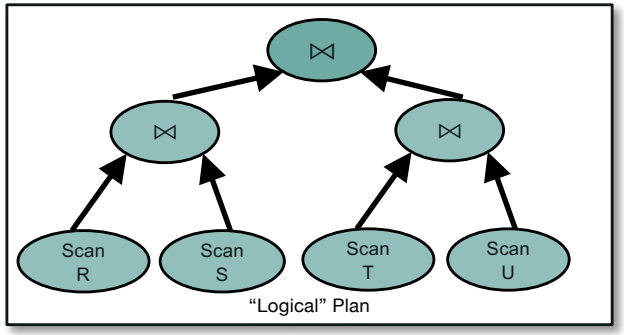


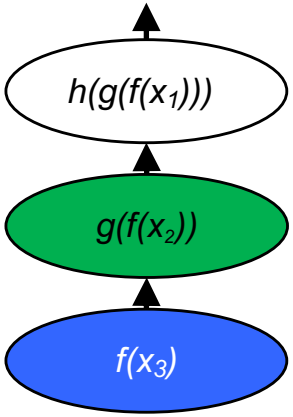
Pipeline Parallelism

Same query, diff operators

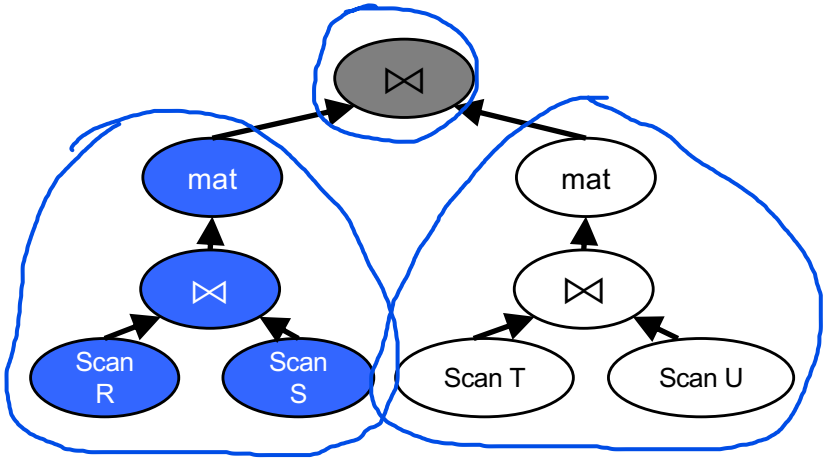
Intra Query - Inter-operator Part 3

- Intra-query
  - Inter-operator





Pipeline Parallelism



Bushy (Tree) Parallelism

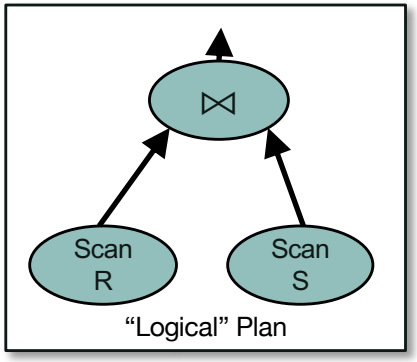
Same tuples get processed by different operators "in a pipeline"

vs.

Different components of the plan proceed in parallel on different data

Intra Query – Intra-Operator

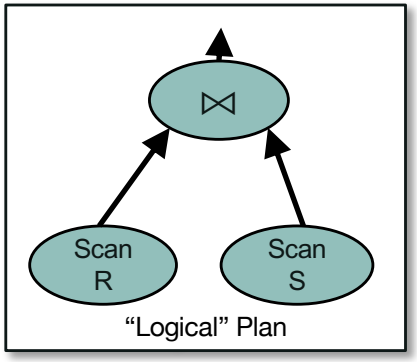
- Intra-query
  - Intra-operator (within a single operator)

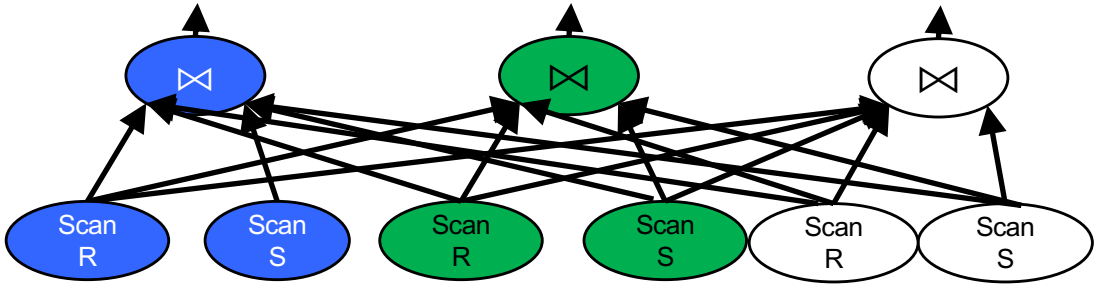


Different queries, same operator

Kinds of Query Parallelism, cont.

- Intra-query
  - Intra-operator



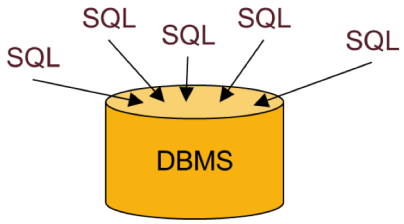


Partition Parallelism

Same operations in parallel
on different data partitions

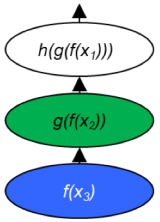
Summary: Kinds of Parallelism

- Inter-Query

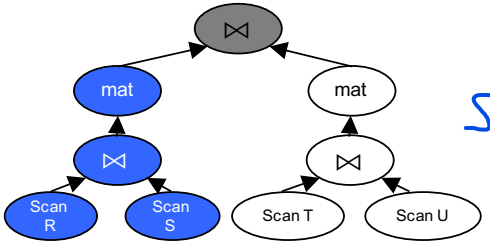


different queries

- Intra-Query
  - Inter-Operator



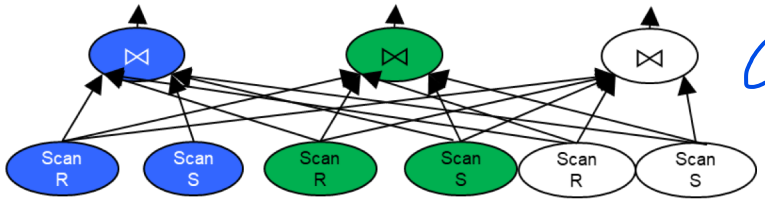
Pipeline Parallelism



Bushy (Tree) Parallelism

same query, diff operators

  - Intra-Operator (partitioned)



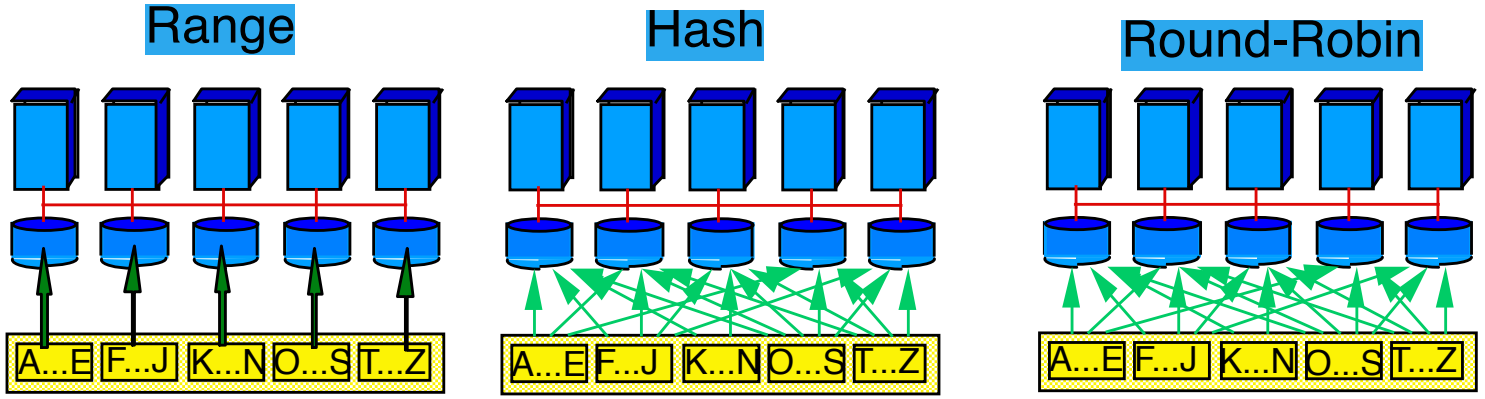
Partition Parallelism

diff queries, same Operators

INTRA-OPERATOR PARALLELISM

Data Partitioning

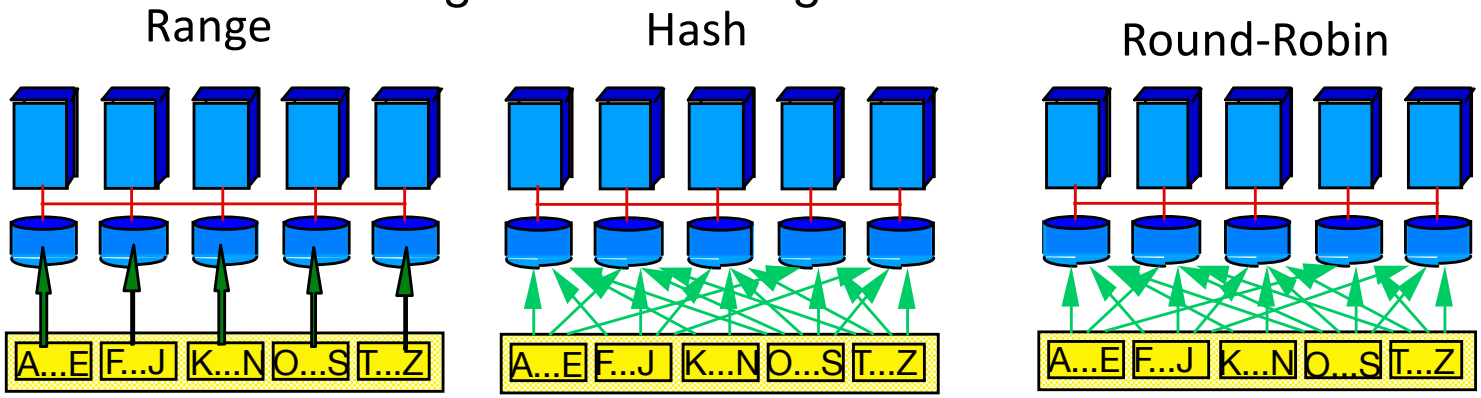
- How to partition a table across disks/machines
  - A bit like coarse-grained indexing!



	Range	Hash	Round-Robin
Locality of access	most	medium	least
Balanced load	least	medium	most

Data Partitioning

- How to partition a table across disks/machines
  - A bit like coarse-grained indexing!

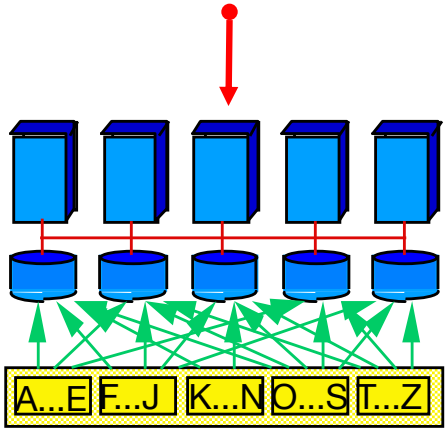


  - Shared nothing particularly benefits from "good" partitioning
    - Reduces network traffic
    - Better if operations are “localized” to certain nodes
  - Indexes can be built at each partition
    - E.g., a B+tree at each node

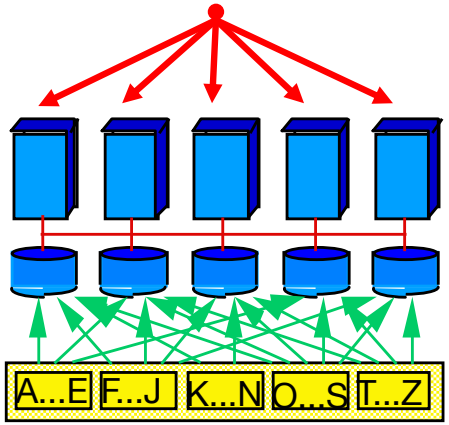
b/c multiple machines w/ multiple disks

Lookup by key

- Q: Which scheme would work best? Range/Hash/Round-Robin
- Data partitioned on function of key?
  - Great! Route lookup only to relevant node
    - Applies to both hash and range-based partitioning good
- Otherwise or if we use round-robin partitioning bad
  - Have to broadcast lookup (to all nodes)



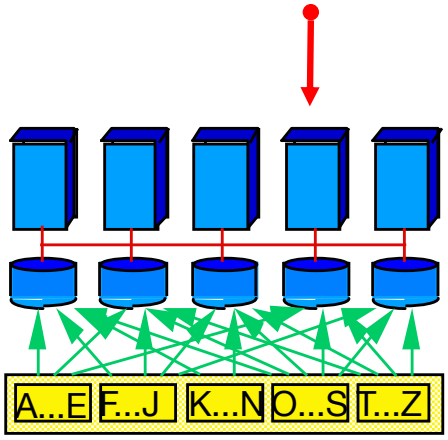
Hash / Range



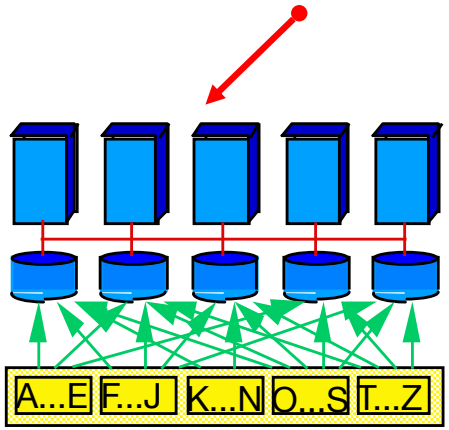
Round-Robin

What about Insert?

- Route to relevant node
  - As before, applies to both hash and range-based partitioning
- Otherwise
  - Route insert to *any* node



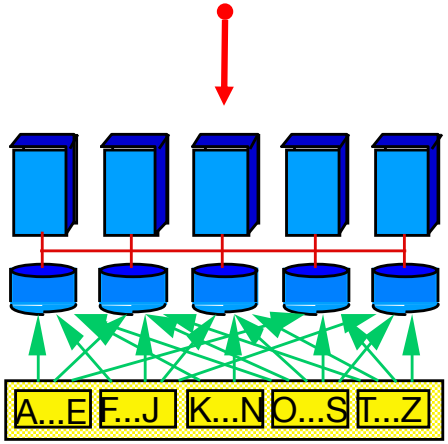
Hash / Range



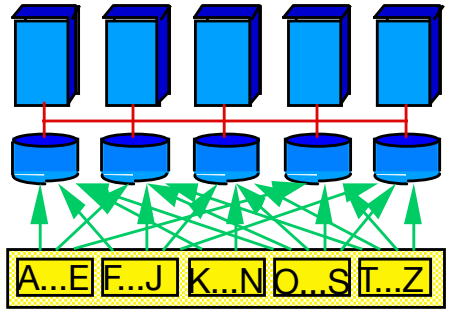
Round-Robin

Insert to Unique Key?

- Data partitioned on function of key?
  - Route to relevant node
    - And reject if already exists
  - Again, applies to both hash and range-based



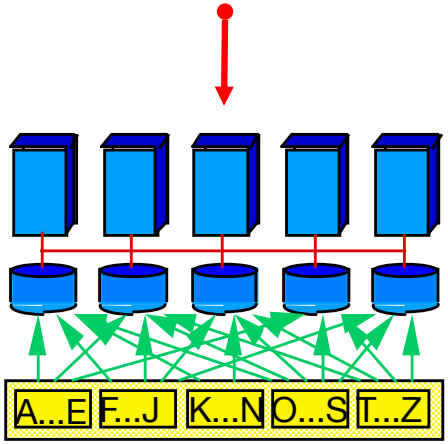
Hash / Range



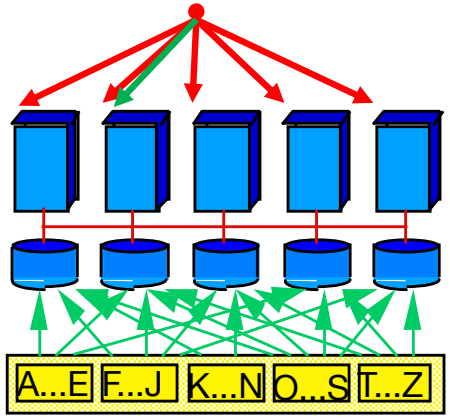
Round-Robin

Insert to Unique Key cont.

- Otherwise (e.g., round-robin or partitioning on diff attribute)
  - Broadcast lookup
  - Collect responses
  - If not exists, insert at appropriate place
    - Else reject



Hash / Range

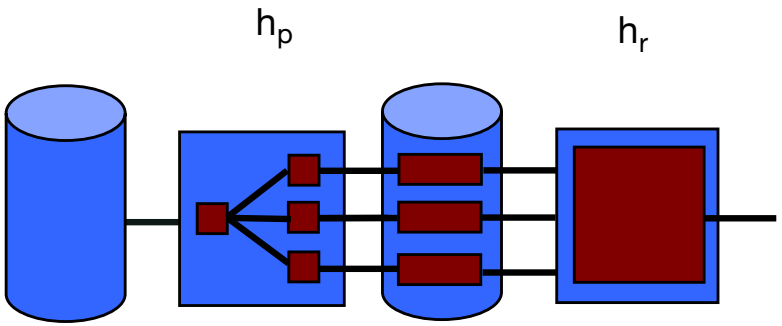


Round-Robin

Parallel Scans

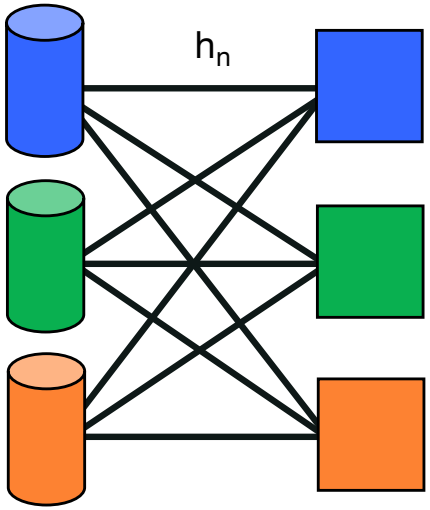
- Scan in parallel, merge (concat) output
- σ_p : skip entire sites that have no tuples satisfying p
  - Range or hash partitioning on attributes involved in *p* benefits from this
  - Round-robin does not, nor does partitioning on other attributes

Remember Hashing?



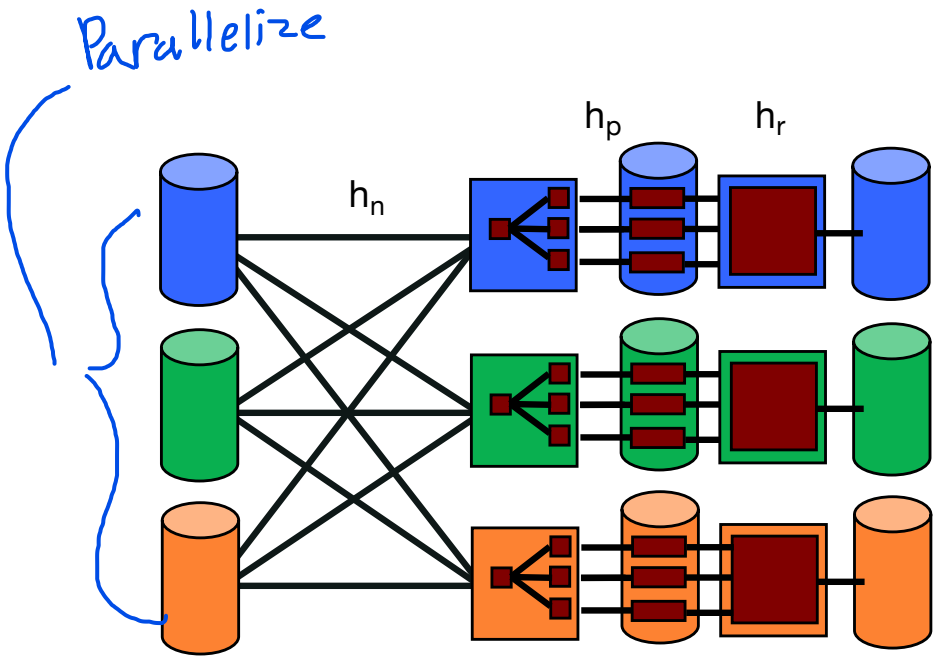
Parallelize me! Hashing

- Phase 1: shuffle data across machines (h_n)
  - streaming out to network as it is scanned
  - which machine for this record?
    - use (yet another) independent hash function h_n



Parallelize me! Hashing Part 2

- Receivers proceed with phase 1 in a pipeline as data streams in



Nearly same as single-node hashing

Near-perfect speed-up, scale-up! Streams through phase 1, with no waiting

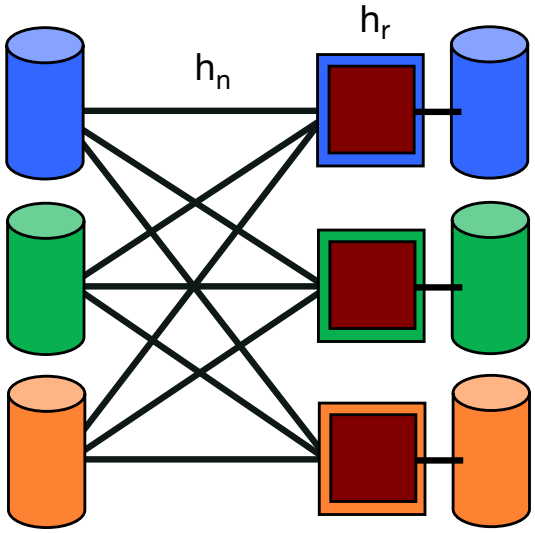
Have to wait for stragglers to start phase 2, accounting for skew if needed

Parallelizing Relational Operators

- Joins
- Sort
- Aggregation
- Group by

Naïve parallel hash join R ⋈ S

- Phase 1: Shuffle R across machines using h_n
  - Parallel scan streaming tuples out to network
  - Build in memory hash table using h_r
  - Wait for hash table building to finish
- Phase 2: Stream probing relation S across machines using h_n
  - Probe hash table for R tuple matches (Last R+W)
  - Writing joined tuples to disk is independent, hence parallel



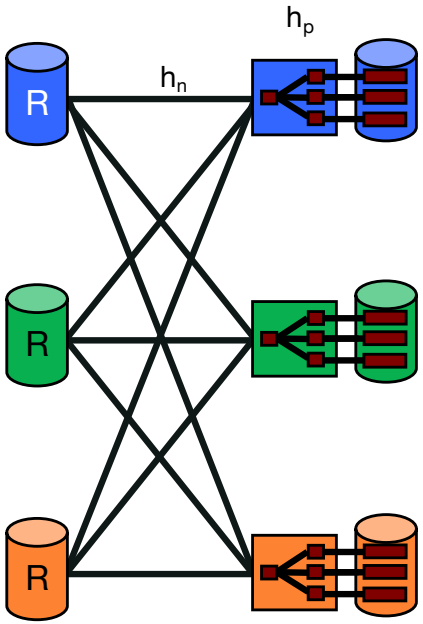
- Note: there is a variation that has *no waiting*: both tables stream
  - Wilschut and Apers' "Symmetric" or "Pipeline" hash join
  - Requires more memory space

- What if we don't have enough memory to build full hash table on R?

Parallel Grace Hash Join Pass 1

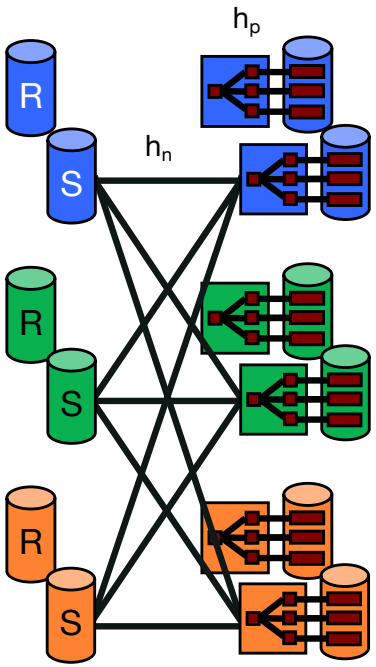
- Pass 1 is like hashing earlier

Naive



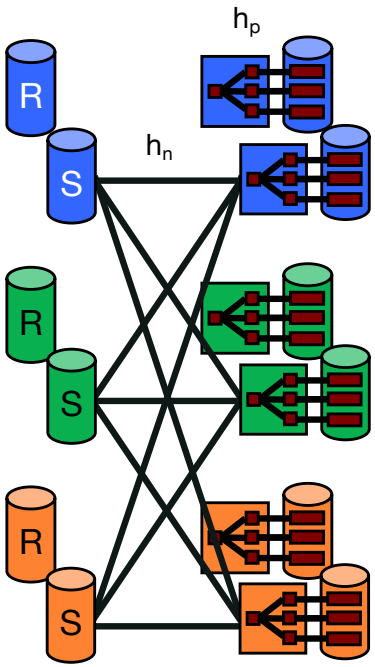
Parallel Grace Hash Join Pass 1 cont

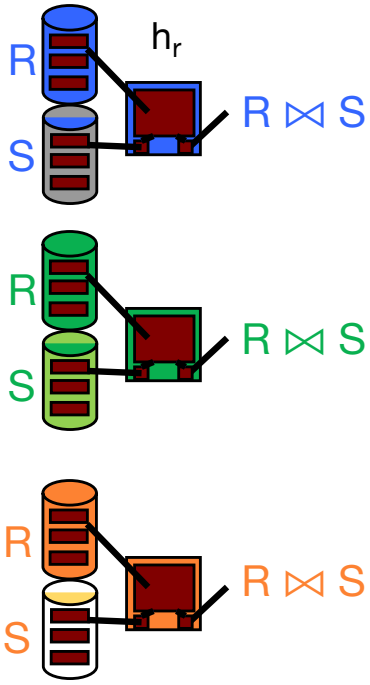
- Pass 1 is like hashing above
  - But do it 2x: once for each relation being joined



Parallel Grace Hash Join Pass 2

- Pass 2 is local Grace Hash Join per node
  - Complete independence across nodes





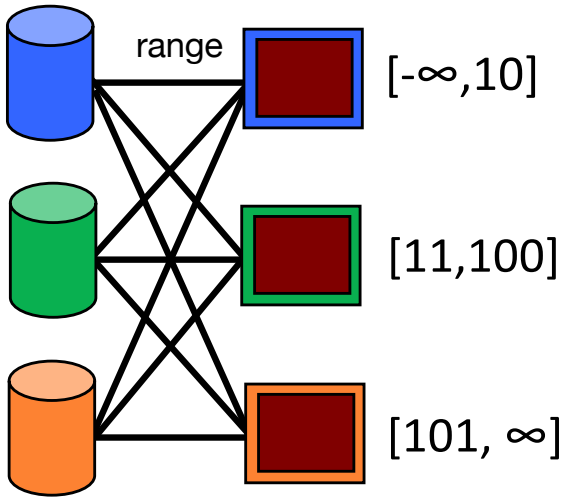
Parallel Grace Hash Join

- Pass 1: parallel streaming
  - Stream building and probing tables through shuffle/partition
- Pass 2 is local Grace Hash Join per node
  - Complete independence across nodes in Pass 2
- Near-perfect speed-up, scale-up!
- Every component works at its top speed
  - Only waiting is for Pass 1 to end.

- Note: there is a variant that has no waiting
  - Urhan's Xjoin, a variant of symmetric hash

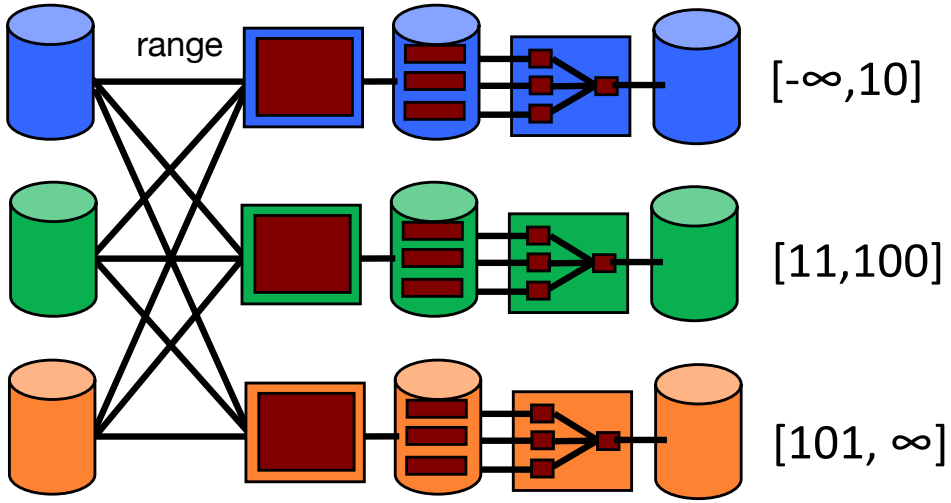
Parallel Sorting Pass 0

- Pass 0: shuffle data across machines
  - streaming out to network as it is scanned
  - which machine for this record?
    Split on value range (e.g. [-∞,10], [11,100], [101, ∞]).



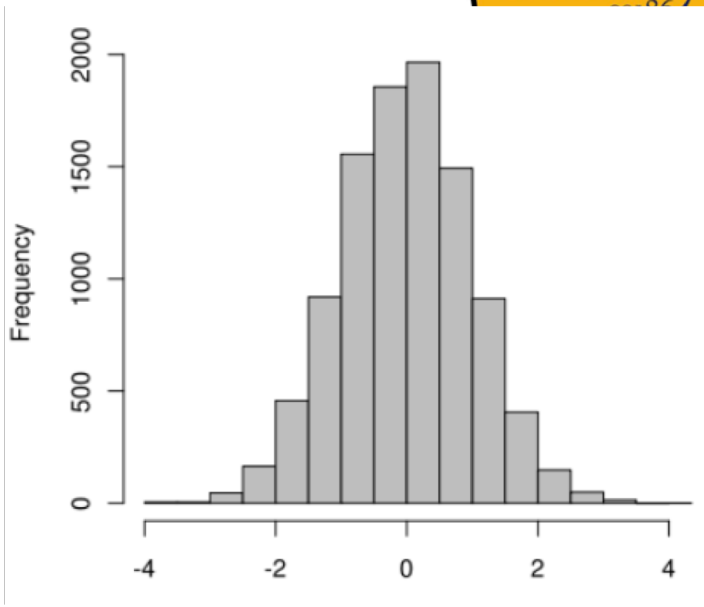
Parallel Sorting Passes 1-n

- Receivers proceed with pass 0 as the data streams in
- Passes 1–n done independently as in single-node sorting
- A Wrinkle: How to ensure ranges have the same #pages?
  - i.e. avoid data skew?



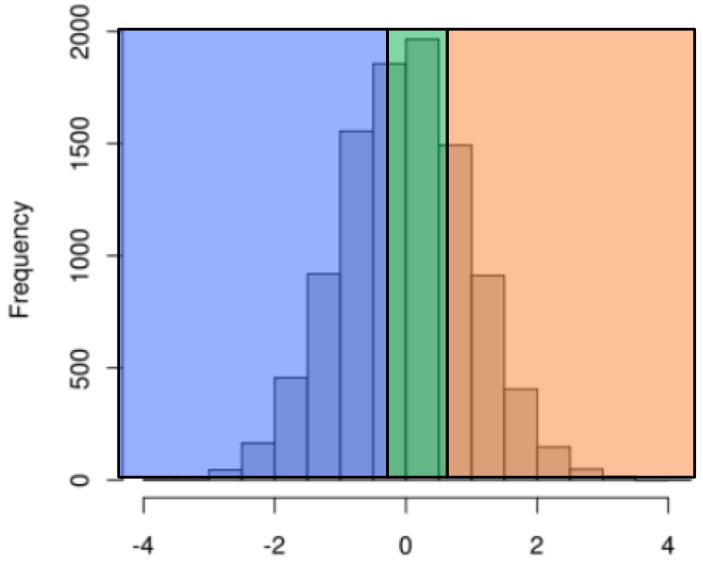
Range partitioning

- Goal: equal frequency per machine
- Note: ranges often don't divide x axis evenly
- How to choose?



Range partitioning cont.

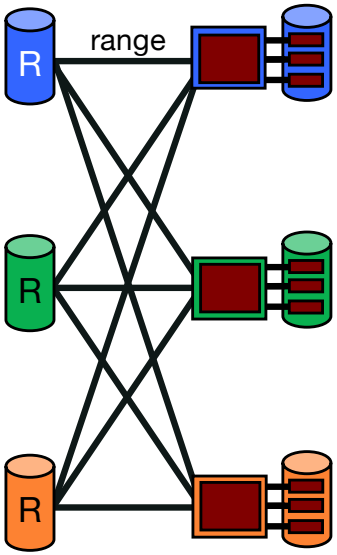
- Easy for small data
- In general, can sample the input relation prior to shuffling, pick splits based on sample
- Note: Random sampling can be tricky to implement in a query pipeline; simpler if we materialize the (intermediate) table first.



How to sample a database table?
Advanced topic, we will not discuss in this class.

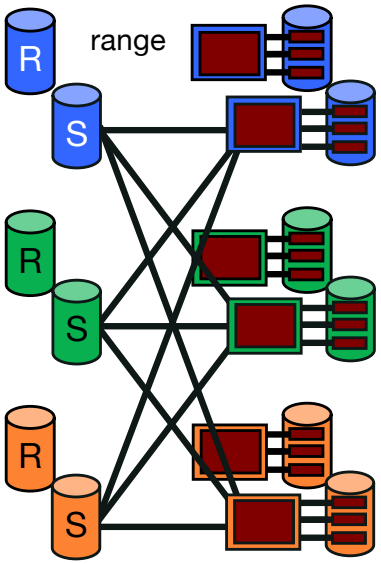
Parallel Sort-Merge Join

- Pass 0 .. n-1 are like parallel sorting above
- Note: this picture is a 2-pass sort (n=1); this is pass 0



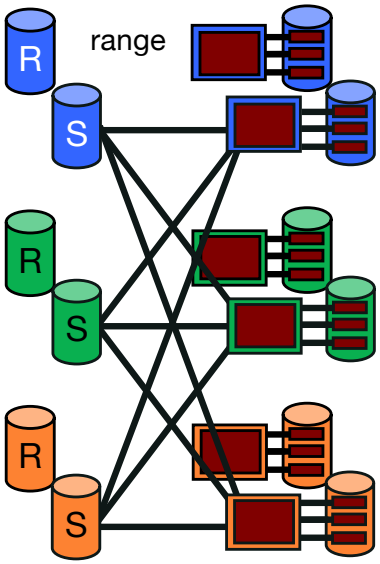
Parallel Sort-Merge Join Pass 0…n-1

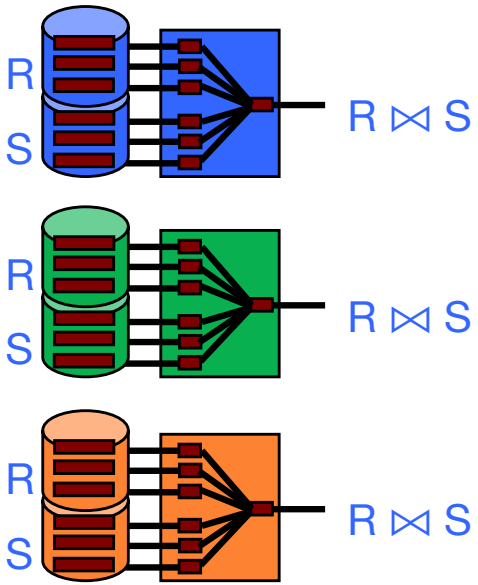
- Pass 0 .. n-1 are like parallel sorting above
  - But do it 2x: once for each relation, with same ranges
  - Note: this picture is a 2-pass sort (n=1); this is pass 0



Pass n

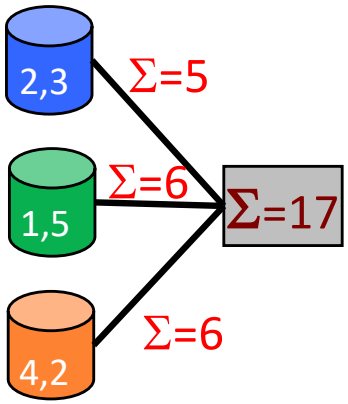
- Pass 0 .. n-1 are like parallel sorting above
  - But do it 2x: once for each relation, with same ranges
- Pass n: merge join partitions locally on each node





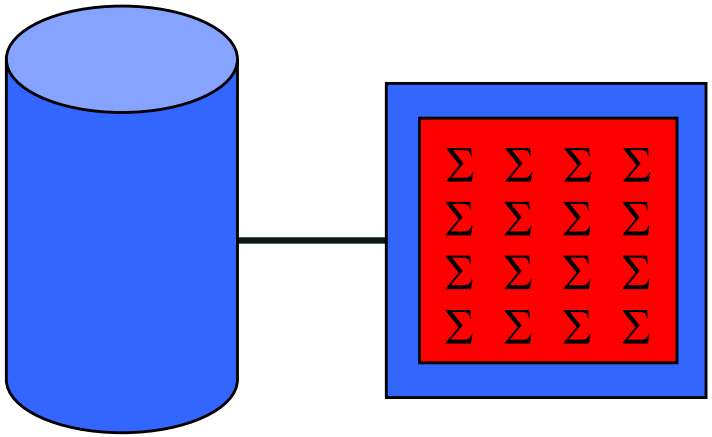
Parallel Aggregates

- Hierarchical aggregation
- For each aggregate function, need a global/local decomposition:
  - **sum**(S) = Σ Σ (s)
  - **count** = Σ **count** (s)
  - **avg**(S) = (Σ Σ (s)) / Σ **count** (s)
  - etc…



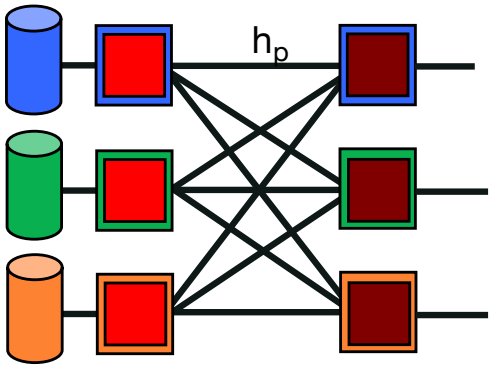
Parallel GroupBy

- Naïve Hash Group By
  - Local aggregation: in hash table keyed by group key k_i keep local agg_i
    - E.g. `SELECT SUM(price) group by cart;`



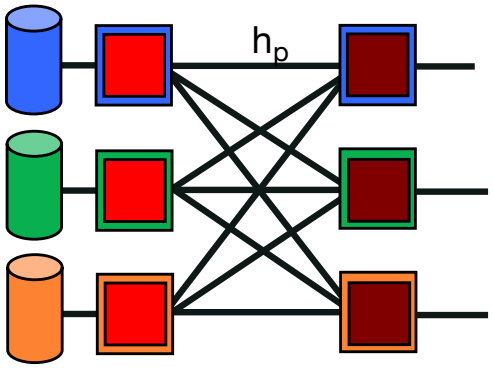
Parallel GroupBy, Cont.

- Naïve Hash Group By
  - Local aggregation: in hash table keyed by group key k_i keep local agg_i
    - For example, k is major, agg is (avg(gpa), count(*))
  - Shuffle local aggregates by a hash function $h_p(k_i)$
  - Compute global aggregates for each key k_i



Parallel Aggregates/GroupBy Challenge!

- Exercise:
  - Figure out parallel 2-pass GraceHash-based scheme to handle # large of groups
  - Figure out parallel Sort-based scheme



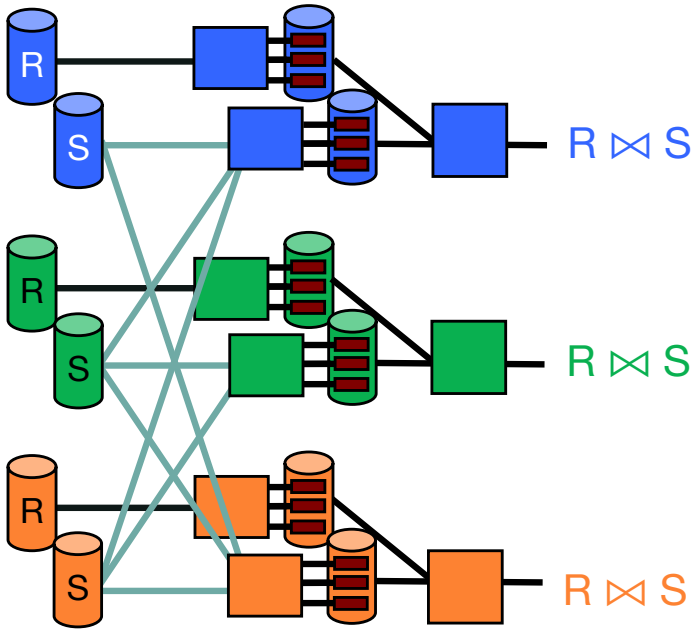
Joins: Bigger picture

- Symmetric shuffle
  - What we did so far

- Alternatives:
  - Asymmetric shuffle
  - Broadcast join
  - Pipeline (non-blocking) join

Join: Asymmetric / One-sided shuffle

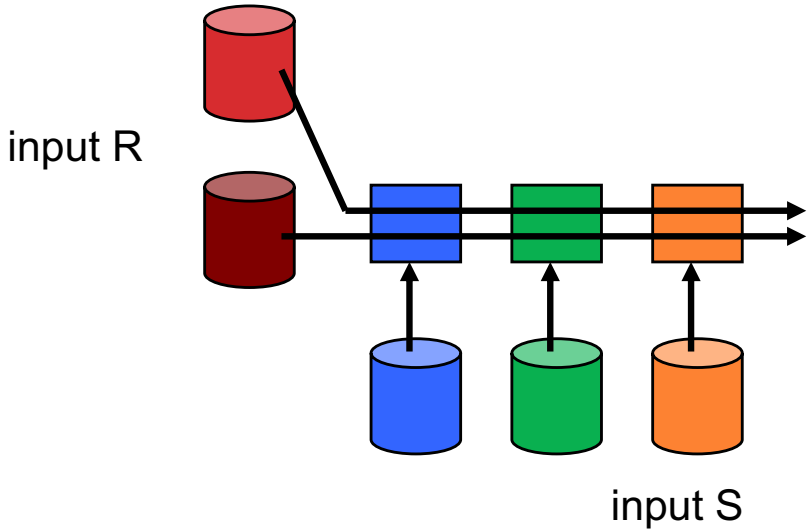
- If R already suitably partitioned, just partition S, then run local join at every node and union results.



"Broadcast" Join

broadcast a table to all the machines

- If R is small, send it to all nodes that have a partition of S.
- Do a local join at each node (using any algorithm) and union results.

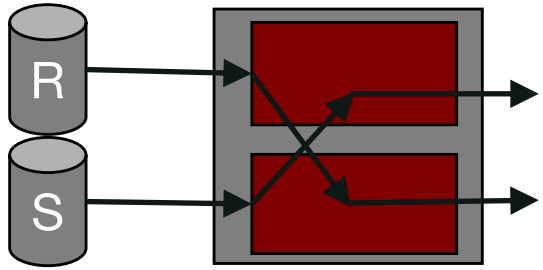


What are “pipeline breakers”?

- Sort
  - Hence sort-merge join can’t start merging until sort is complete
- Hash build
  - Hence Grace hash join can’t start probing until hashtable is built
- Is there a join scheme that pipelines?

Symmetric (Pipeline) Hash Join

- Single-phase, streaming
- Each node allocates two hash tables, one for each side
- Upon arrival of a tuple of R:
  - Build into R hashtable by join key
  - Probe into S hashtable for matches and output any that are found
- Upon arrival of a tuple of S:
  - Symmetric to R!



Symmetric (Pipeline) Hash Join cont

- Why does it work?
  - Each output tuple is generated exactly once: when the second part arrives
- Streaming!
  - Can always pull another tuple from R or S, build, and probe for outputs
  - Useful for stream query engines!

Parallel DBMS Summary

- Parallelism natural to query processing:
  - Both pipeline and partition
- Shared-Nothing vs. Shared-Mem vs. Shared Disk
  - Shared-mem easiest SW, costliest HW.
    - Doesn't scale indefinitely
  - Shared-nothing cheap, scales well, harder to implement.
  - Shared disk a middle ground
    - For updates, introduces icky stuff related to concurrency control
- Intra-op, Inter-op, & Inter-query parallelism all possible.

Parallel DBMS Summary, Part 2

- Most DB operations can be done partition-parallel
  - Sort. Hash.
  - Sort-merge join, hash-join.
- Complex plans.
  - Allow for pipeline-parallelism, but sorts, hashes block the pipeline.
  - Partition parallelism achieved via bushy trees.

Parallel DBMS Summary, Part 3

- What about running transactions on parallel databases?
  - Distributed locks?
  - Distributed deadlock detection?
  - We need new protocols
- More on this in subsequent lectures