Parallel Query Processing

Parallelizing relational query operators

Alvin Cheung
Aditya Parameswaran

Reading: R & G Chapter 22.1-4

Parallelizing Relational Operators

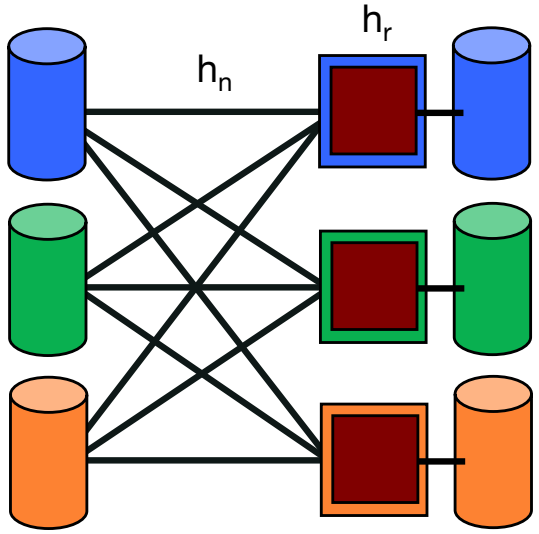
- Joins
- Sort
- Aggregation
- Group by

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
  - Writing joined tuples to disk is independent, hence parallel

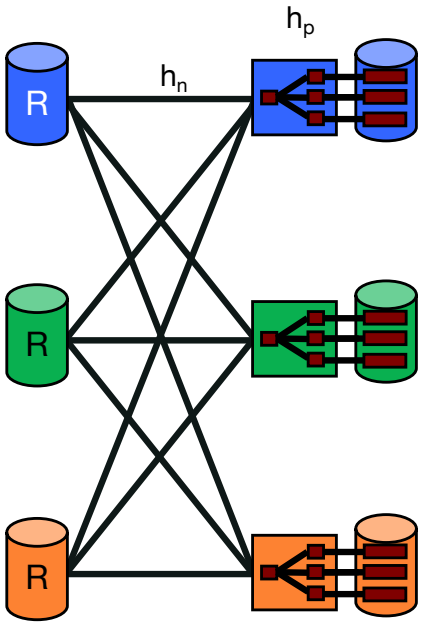
- Note: there is a variation that has *no waiting*: both tables stream
  - Wilschut and Apers' "Symmetric" or "Pipeline" hash join
  - Requires more memory space

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



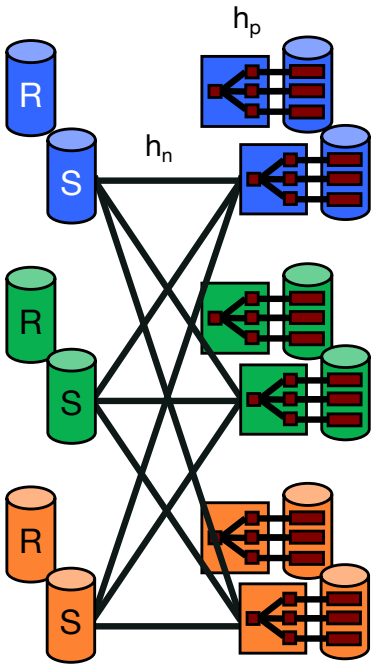
Parallel Grace Hash Join Pass 1

- Pass 1 is like hashing earlier



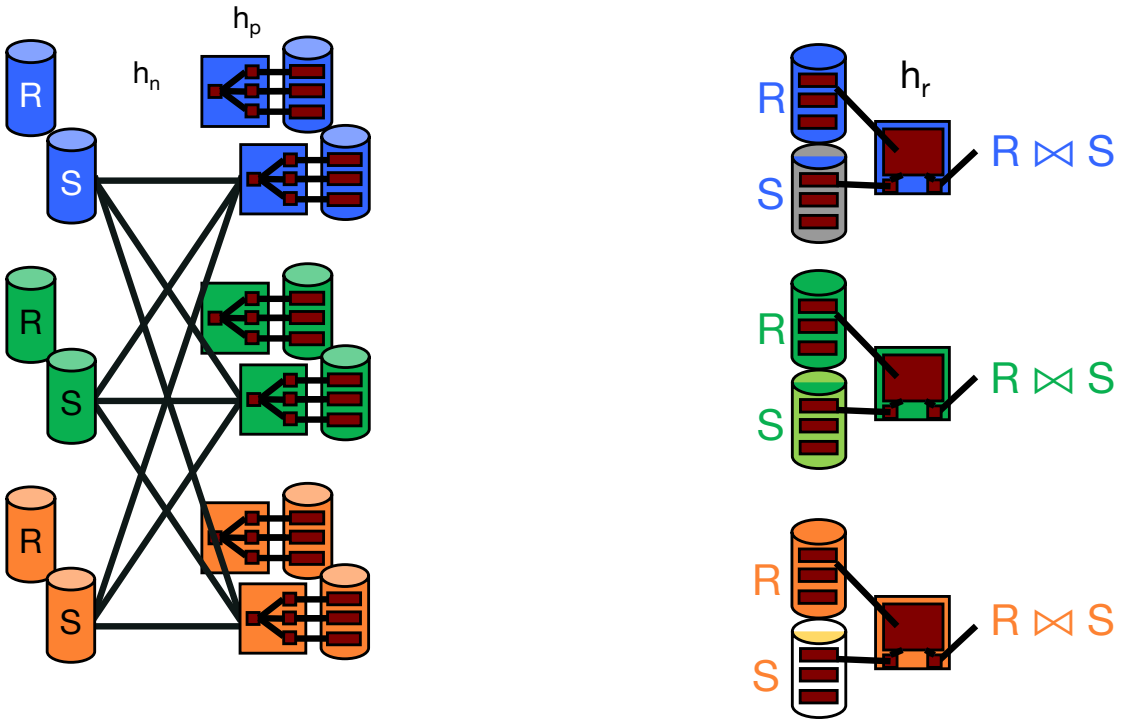
Parallel Grace Hash Join Pass 1 cont

- 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



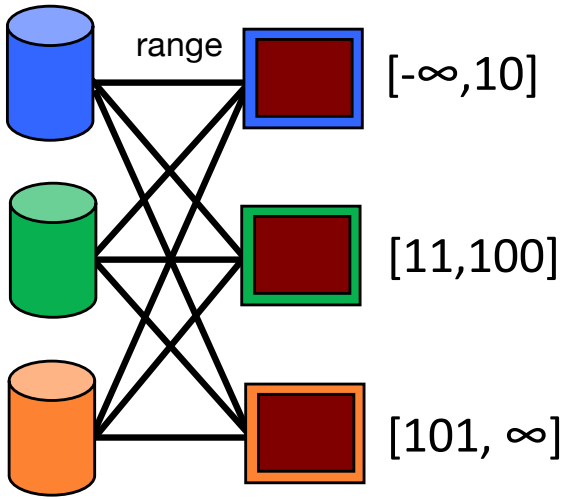
Parallel Grace Hash Join

- 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

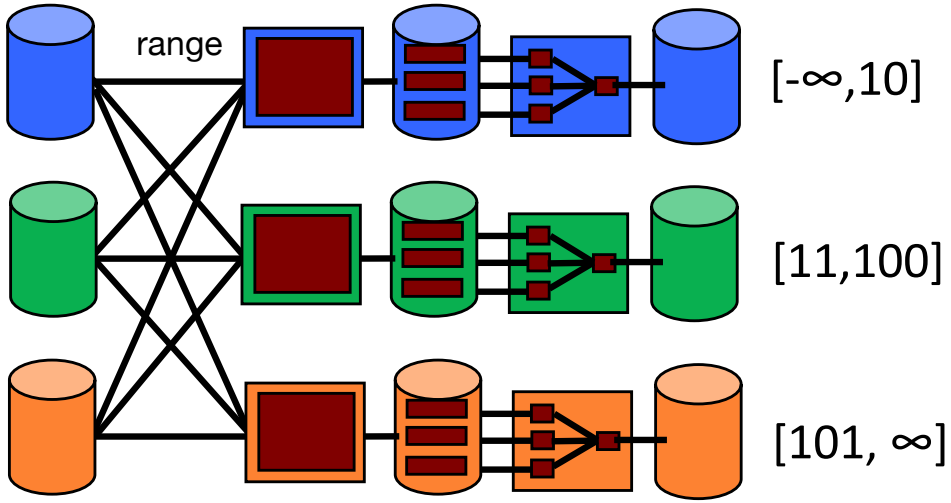
Parallel Sorting Pass 0

- 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, ∞]).



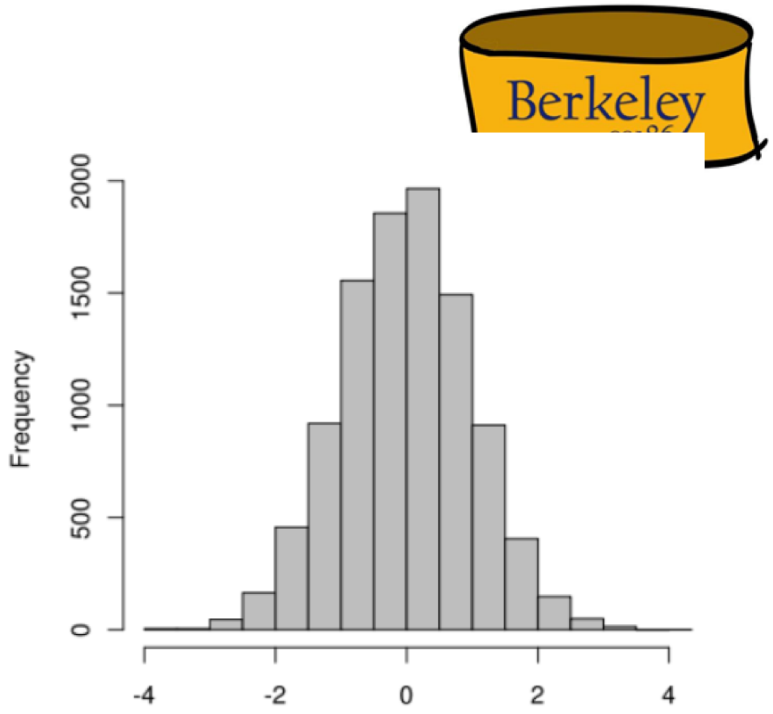
Parallel Sorting Passes 1-n

- 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?



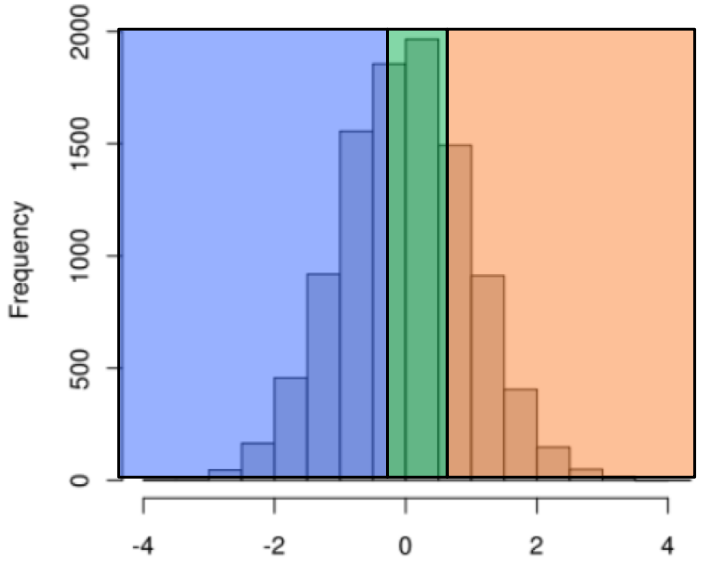
Range partitioning

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



Range partitioning cont.

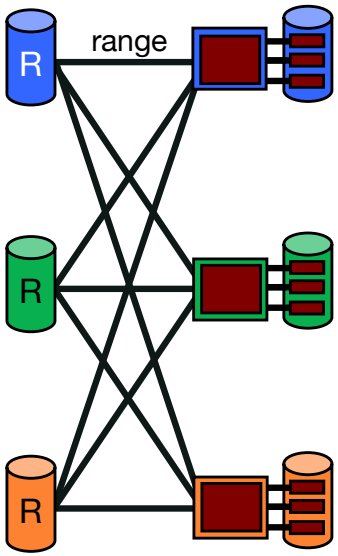
- Would be easy if data small
- In general, can sample the input relation prior to shuffling, pick splits based on sample
- Note: Random sampling can be tricky to implement in a query pipeline; simpler if you materialize first.



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

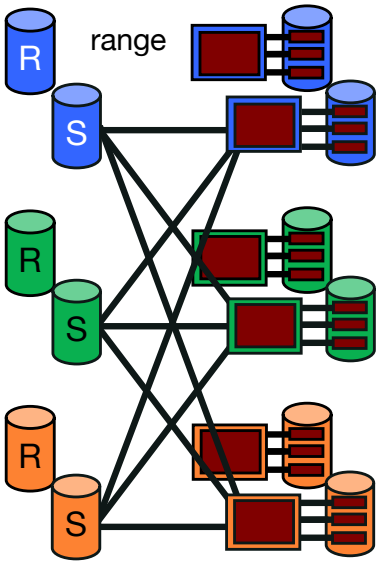
Parallel Sort-Merge Join

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



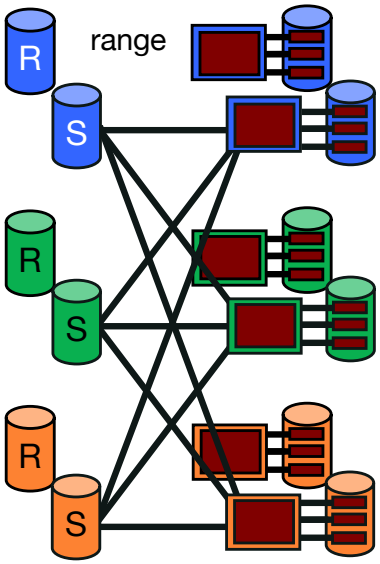
Parallel Sort-Merge Join Pass 0…n-1

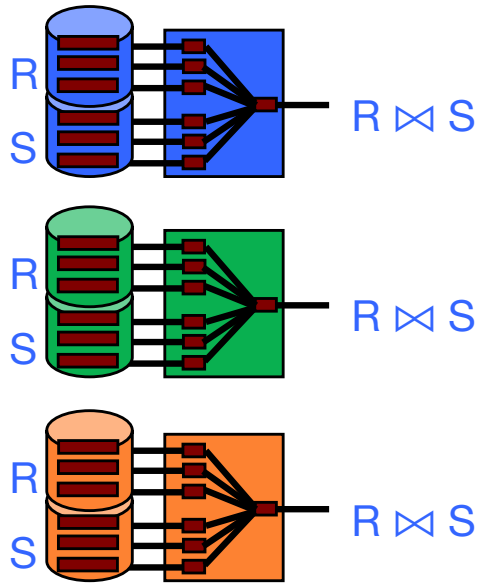
- Pass 0 .. n-1 are like parallel sorting above
  - But do it 2x: once for each relation, with same ranges
  - Note: this picture is a 2-pass sort (n=1); this is pass 0



Pass n

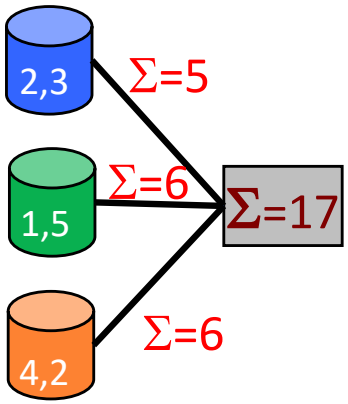
- Pass 0 .. n-1 are like parallel sorting above
  - But do it 2x: once for each relation, with same ranges
- Pass n: merge join partitions locally on each node





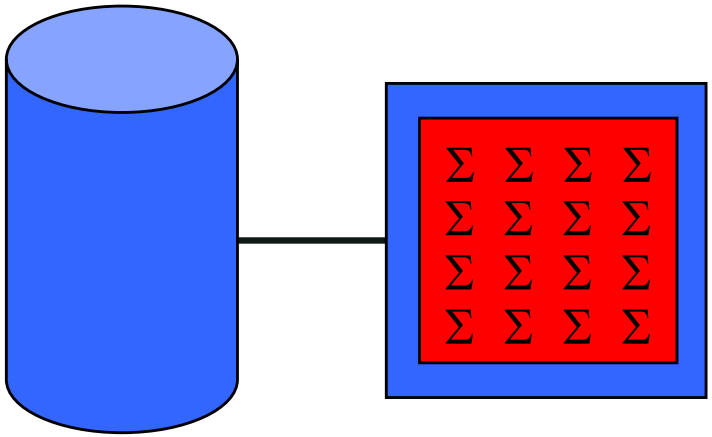
Parallel Aggregates

- 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...



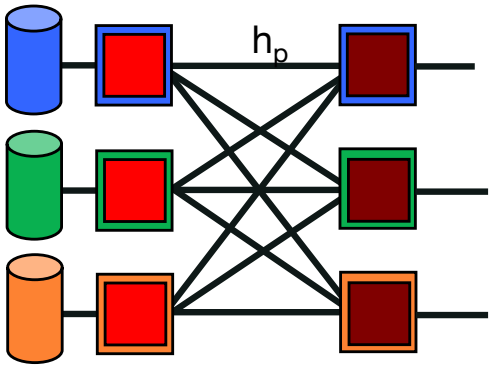
Parallel GroupBy

- 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;`



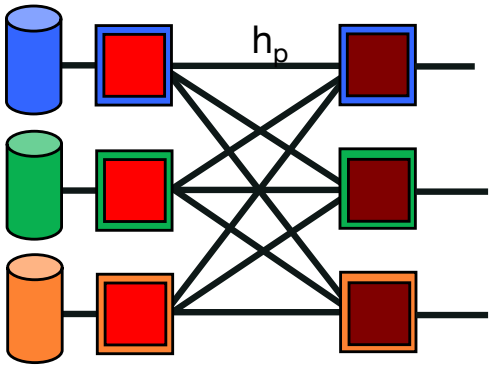
Parallel GroupBy, Cont.

- 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



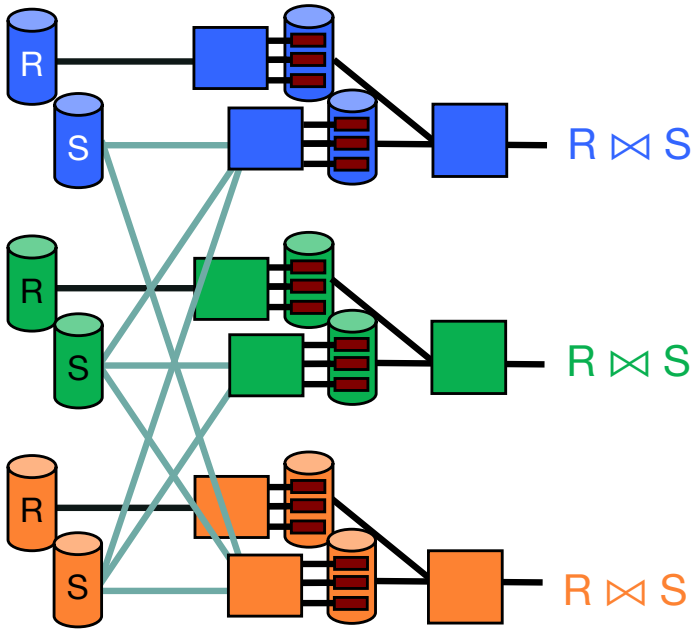
Joins: Bigger picture

- Symmetric shuffle
  - What we did so far

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

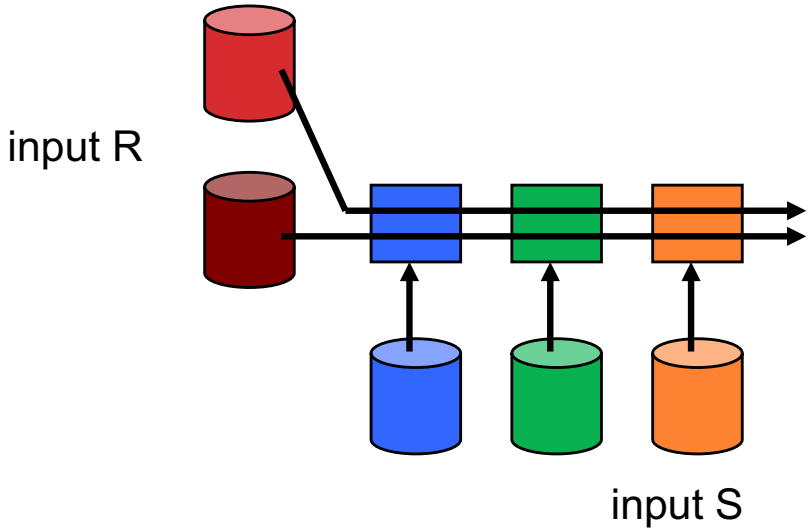
Join: Asymmetric / One-sided shuffle

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



“Broadcast” Join

- 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.

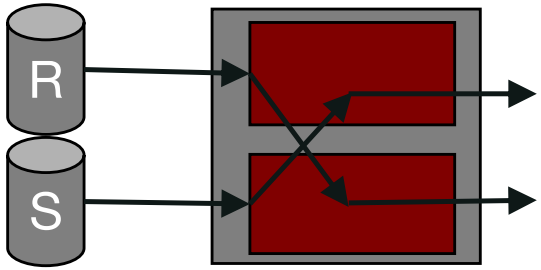


What are “pipeline breakers”?

- 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