CS150A Database

Wenjie Wang
School of Information Science and Technology
ShanghaiTech University
Dec. 13, 2024

Today:
- Parallel Query Processing:

Readings:
- Database Management Systems (DBMS), Chapter 22

Acknowledgement: Joe Hellerstein@UCBerkeley's scourse notes

Review: Refinement

Refinement

- Remove Redundance: Functional Dependencies
- FD
  - Formally:An FD $X \rightarrow Y$ holds over relation schema R if, for every allowable instance r of R: $t1 \in r$, $t2 \in r$, $\pi_X(t1) = \pi_X(t2) \Rightarrow \pi_Y(t1) = \pi_Y(t2)$
  - Key/Super Key/Candidate Key
  - F+ = closure of F:
    - the set of all FDs that are implied by F.
    - includes “trivial dependencies”
    - **Armstrong's Axioms**

Example

- **Contracts(cid,sid,jid,did,pid,qty,value),** and:
  - C is the key: C $\rightarrow$ CSJDPQV
  - Proj (J) purchases each part (P) using single contract (C): JP $\rightarrow$ C
  - Dept (D) purchases at most 1 part (P) from a supplier (S): SD $\rightarrow$ P

- **Problem: Prove that SDJ is a key for Contracts**
- JP $\rightarrow$ C, C $\rightarrow$ CSJDPQV
  - Imply JP $\rightarrow$ CSJDPQV
  - (by transitivity) (shows that JP is a key)
- SD $\rightarrow$ P
  - implies SDJ $\rightarrow$ JP (by augmentation)
- SDJ $\rightarrow$ JP, JP $\rightarrow$ CSJDPQV
  - imply SDJ $\rightarrow$ CSJDPQV
  - (by transitivity) (shows that SDJ is a key).

- Q: can you now infer that SD $\rightarrow$ CSDPQV

Refinement

- Remove Redundance: Functional Dependencies
- FD
  - Formally:An FD $X \rightarrow Y$ holds over relation schema R if, for every allowable instance r of R: $t1 \in r$, $t2 \in r$, $\pi_X(t1) = \pi_X(t2) \Rightarrow \pi_Y(t1) = \pi_Y(t2)$
  - Key/Super Key/Candidate Key
  - F+ = closure of F:
    - the set of all FDs that are implied by F.
    - includes "trivial dependencies"
    - **Armstrong's Axioms**
  - Compute attribute closure of X (denoted X+) wrt F.

Attribute Closure (example)

$R = \{A, B, C, D, E\}$

$F = \{ B \rightarrow CD, D \rightarrow E, B \rightarrow A, E \rightarrow C, AD \rightarrow B \}$

- **Is $B \rightarrow E$ in F^+ ?**
  $B+ = \{B, C, D, E, \ldots\}$
  … Yep!

- **Is D a key for R?**
  $D^+ = \{D, E, C\}$
  … Nope!

- **Is AD a key for R?**
  $AD^+ = \{A, D, E, C, B\}$
  …Yep!

- **Is AD a *candidate* key for R?**
  $A^+ = \{A\}$ $D^+ = \{D, E, C\}$
  …Yes!

- **Is ADE a candidate key for R?**
  No!

Refinement

- Remove Redundance: Functional Dependencies
- Functional Dependencies
- Normal Form: Boyce-Codd Normal Form (BCNF)
  - R is in BCNF if the only non-trivial FDs over R are key constraints.
- Decomposition of a Relation Scheme into BCNF
  - There are three potential problems to consider:

    1) May be ***impossible*** to reconstruct the original relation! (Lossiness)
    - Decomposition of R into X and Y is ***lossless-join*** w.r.t. a set of FDs F if, for every instance *r* that satisfies F: $\pi_X(r) \bowtie \pi_Y(r) = r$
    - Theorem: The decomposition of R into X and Y is lossless with respect to F if and only if the closure of F contains: $X \cap Y \rightarrow X$, or $X \cap Y \rightarrow Y$

    2) Dependency checking may require joins.
    - Decomposition of R into X and Y is dependency preserving if $(F_X \cup F_Y)^+ = F^+$

    3) Some queries become more expensive.

Dependency Preservation: Notes

- Critical to consider F^+ in the definition:
  - ABC, $A \to B$, $B \to C$, $C \to A$, decomposed into AB and BC.
  - Is this dependency preserving? Is $C \to A$ preserved?????
- Well… F^+ contains $F \cup \{A \to C, B \to A, C \to B\}$, so…
  - $F_{AB} \supseteq \{A \to B, B \to A\}$; $F_{BC} \supseteq \{B \to C, C \to B\}$
  - So, $(F_{AB} \cup F_{BC})^+ \supseteq \{B \to A, C \to B\}$
  - Hence $(F_{AB} \cup F_{BC})^+ \supseteq \{C \to A\}$

$$\mathbf{(F_X \cup F_Y)^+ = F^+}$$

Decomposition into BCNF

- Consider relation R with FDs F.
- If X → Y violates BCNF, decompose R into **R - Y and XY** (guaranteed to be loss-less).
  - Repeated application of this idea will give us a collection of relations that are in BCNF
  - Lossless join decomposition, and guaranteed to terminate.

- e.g., CSJDPQV, key C, JP → C, SD → P, J → S
  - {contractid, supplierid, projectid,deptid,partid, qty, value}
  - To deal with SD → P, decompose into SDP, CSJDQV.
  - To deal with J → S, decompose CSJDQV into JS and CJDQV
  - So we end up with: SDP, JS, and CJDQV

- Note: several dependencies may cause violation of BCNF.
- The order in which we “deal with” them could lead to very different sets of relations!

BCNF and Dependency Preservation

- In general, **there may not be a dependency preserving decomposition into BCNF.**
- E.g., CSZ, CS → Z, Z → C
  - Can't decompose while preserving 1st FD; not in BCNF.
- Similarly, decomposition of CSJDPQV into SDP, JS and CJDQV is not dependency preserving (w.r.t. the FDs **JP** → **C**, SD → P and J → S).
  - However, it is a lossless join decomposition.
  - In this case, adding JPC to the collection of relations gives us a dependency preserving decomposition.
    - but JPC tuples are stored only for checking the f.d. (***Redundancy!***)

A little history

- Relational revolution
  - declarative set-oriented primitives
  - 1970's
- Parallel relational database systems
  - on commodity hardware
  - 1980's
- Big Data: MapReduce, Spark, etc.
  - scaling to thousands of machines and beyond
  - 2005-2015

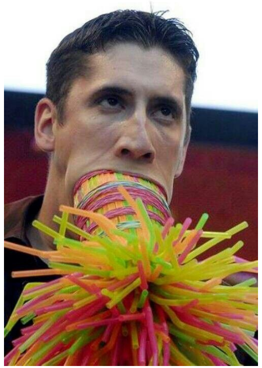
Review: Parallel Query Processing

Why Parallelism?

- Scan 100TB
  - At 0.5 GB/sec (see lec 4): ~200,000 sec = ~2.31 days

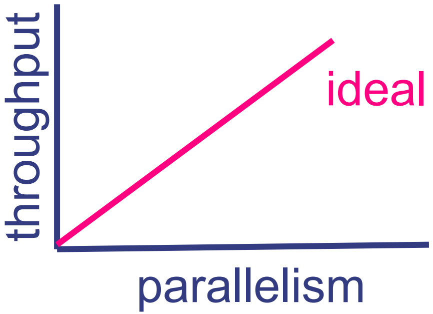
Why Parallelism? Cont.

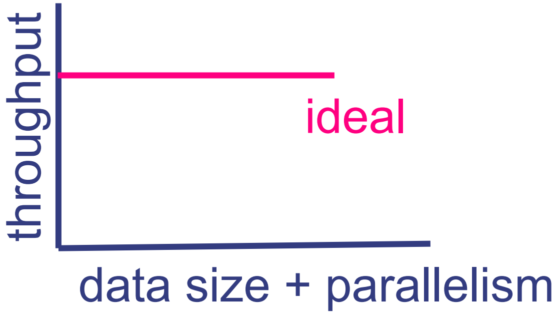
- Scan 100TB
  - At 0.5 GB/sec (see lec 4): ~200,000 sec = ~2.31 days
- Run it 100-way parallel:
  - 2,000 sec = 33 minutes
- 1 big problem = many small problems
  - Trick: make them independent



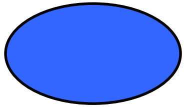
Two Metrics to Shoot For

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

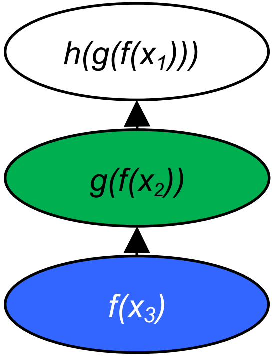




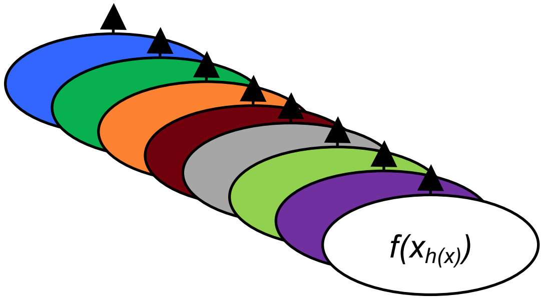
Roughly 2 Kinds of Parallelism



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



Pipeline
scales up to pipeline depth



Partition
scales up to amount of data

We'll get more refined soon.

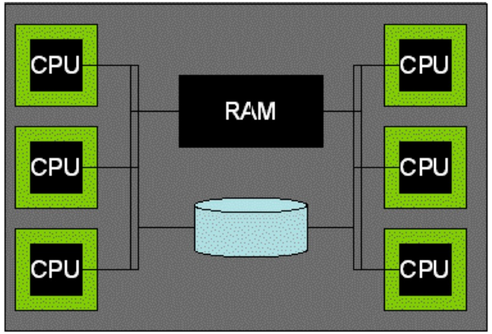
Easy for us to say!

- Lots of Data:
  - Batch operations
  - Pre-existing divide-and-conquer algorithms
  - Natural pipelining

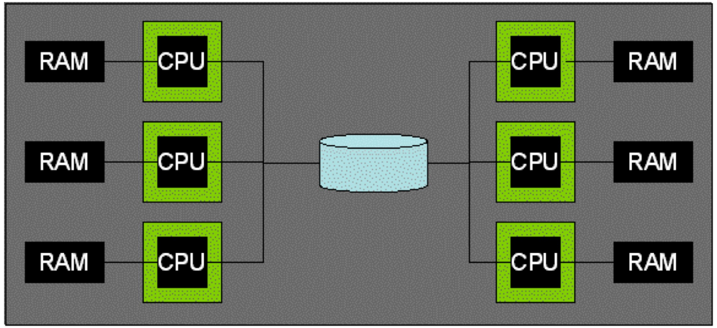
- Declarative languages
  - Can adapt the parallelism strategy to the task and the hardware
  - All without changing the program!
    - Codd’s Physical Data Independence

Parallel Architectures

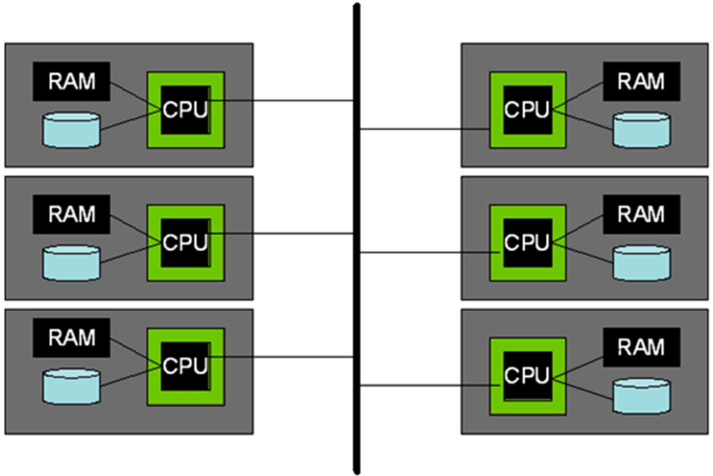
Shared Memory



Shared Disk



Shared Nothing
(cluster)

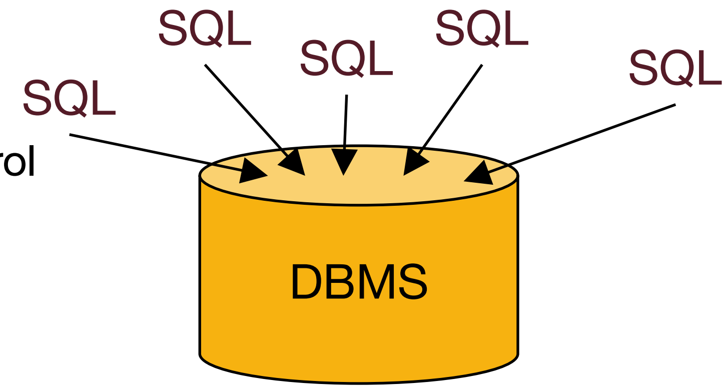


Shared Nothing

- 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
    - Control it in SW

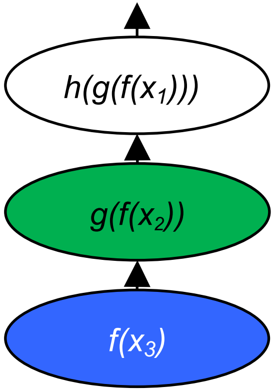
Kinds of Query Parallelism

- Inter-query (parallelism across queries)
  - Each query runs on a separate processor
    - Single thread (no parallelism) per query
  - Does require parallel-aware concurrency control



Intra Query – Inter-operator

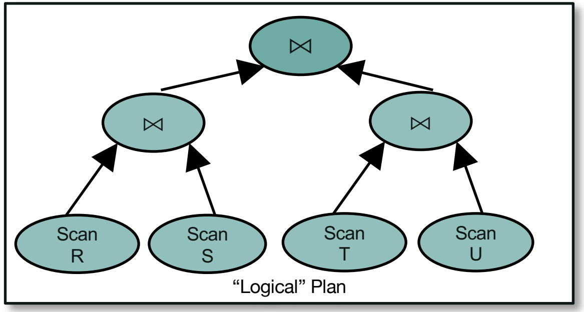
- Intra-query (within a single query)
  - Inter-operator (between operators)

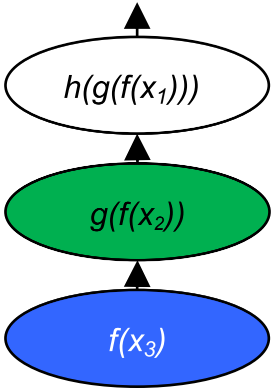


Pipeline Parallelism

Intra Query – Inter-operator Part 2

- Intra-query
  - Inter-operator

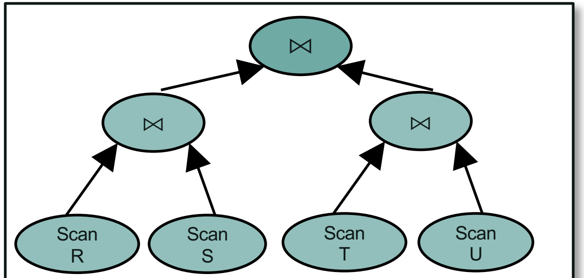




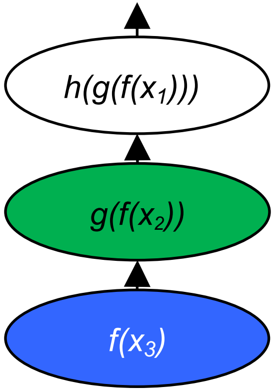
Pipeline Parallelism

Intra Query - Inter-Operator Part 3

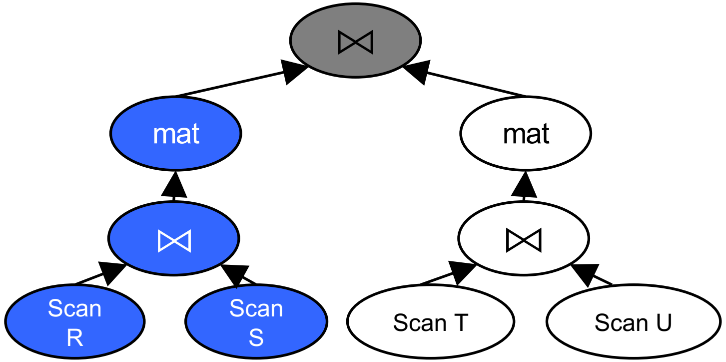
- Intra-query
  - Inter-operator



“Logical” Plan



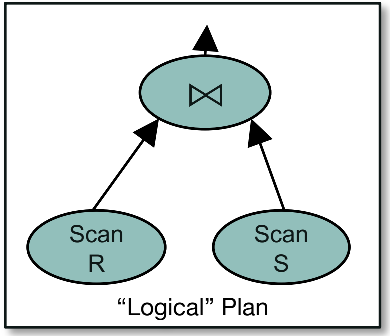
Pipeline Parallelism



Bushy (Tree) Parallelism

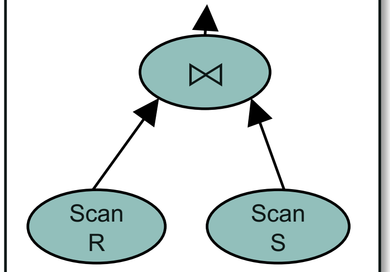
Intra Query – Intra-Operator

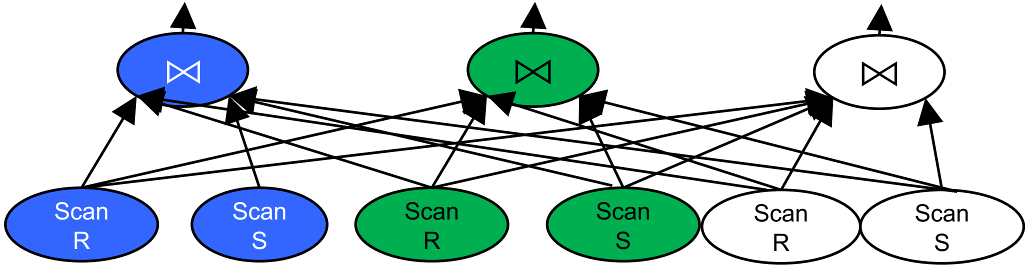
- Intra-query
  - Intra-operator (within a single operator)



Kinds of Query Parallelism, cont.

- Intra-query
  - Intra-operator



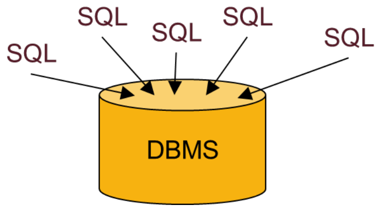


“Logical” Plan

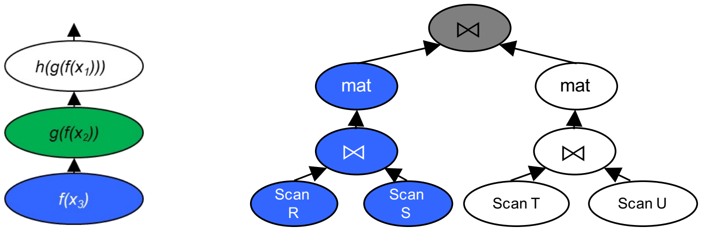
Partition Parallelism

Summary: Kinds of Parallelism

- Inter-Query

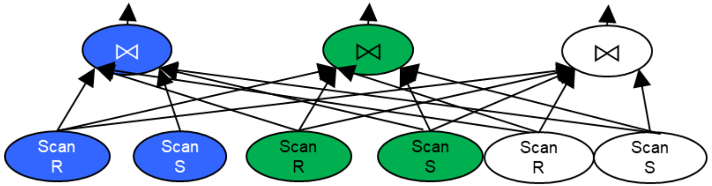


- Intra-Query
  - Inter-Operator



Pipeline Parallelism

  - Intra-Operator (partitioned)

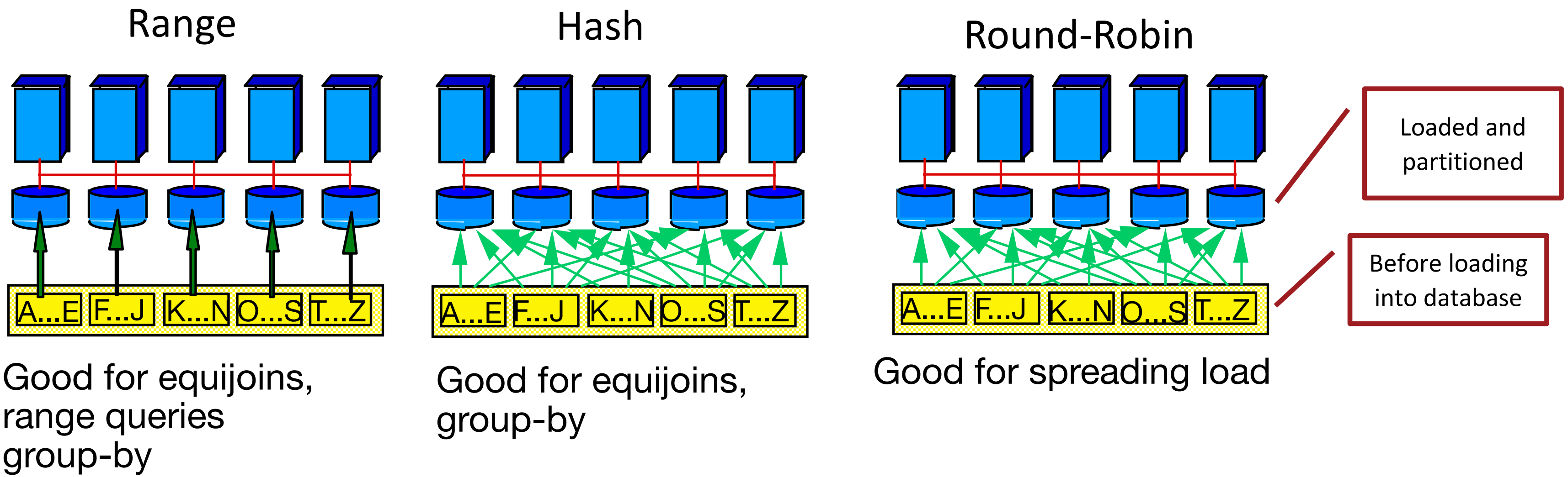


Partition Parallelism

INTRA-OPERATOR PARALLELISM

Data Partitioning

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



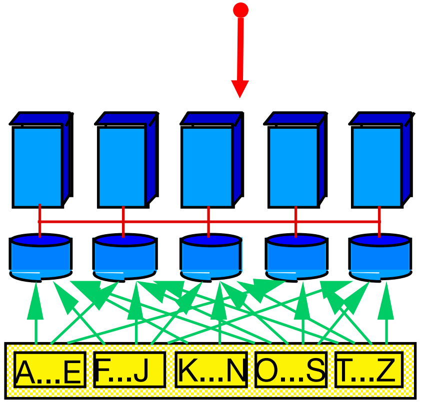
- Shared nothing particularly benefits from "good" partitioning

Parallel Scans

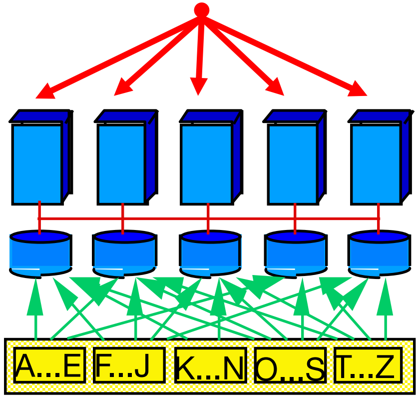
- Scan in parallel, merge (concat) output
- σ_p : skip entire sites that have no tuples satisfying p
  - range or hash partitioning
- Indexes can be built at each partition
- Q: Do indexes differ in the different data partitioning schemes?

Lookup by key

- Data partitioned on function of key?
  - Great! Route lookup only to relevant node
- Otherwise
  - Have to broadcast lookup (to all nodes)



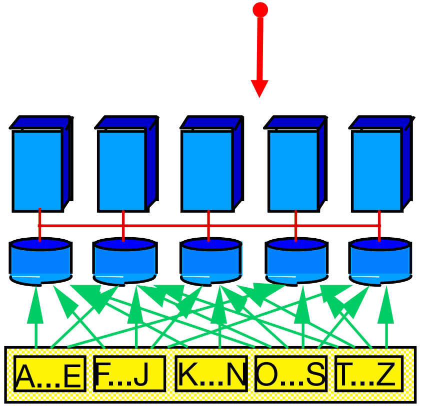
Hash



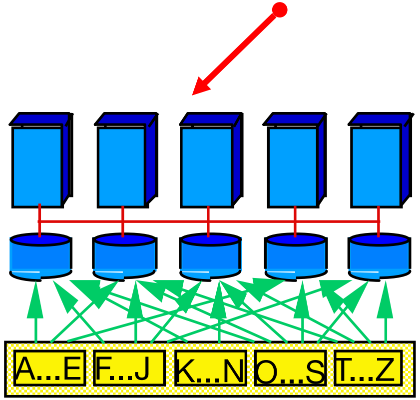
Round-Robin

What about Insert?

- Data partitioned on function of key?
  - Route insert to relevant node
- Otherwise
  - Route insert to *any* node



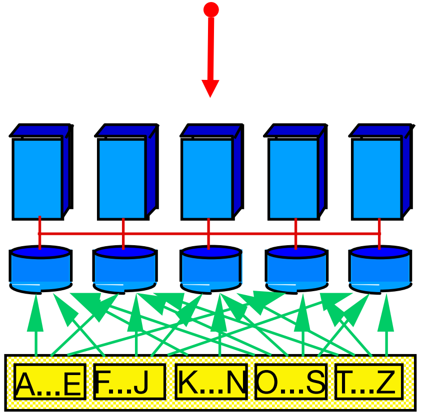
Hash



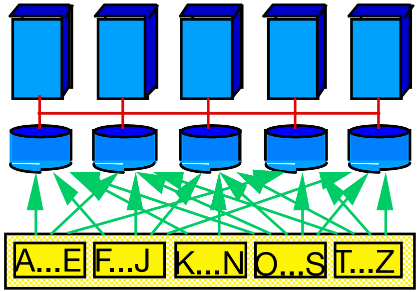
Round-Robin

Insert to Unique Key?

- Data partitioned on function of key?
  - Route to relevant node
    - And reject if already exists



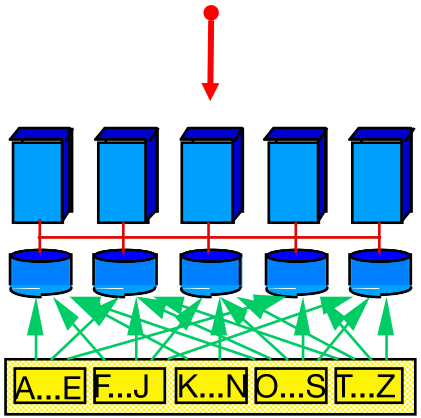
Hash



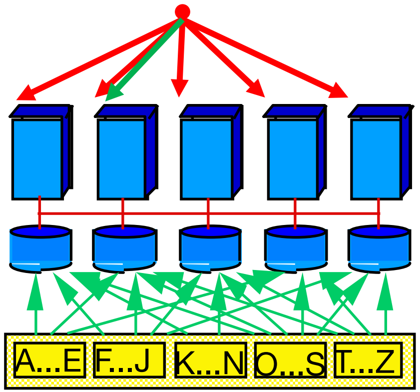
Round-Robin

Insert to Unique Key cont.

- Otherwise
  - Broadcast lookup
  - Collect responses
  - If not exists, insert anywhere
    - Else reject

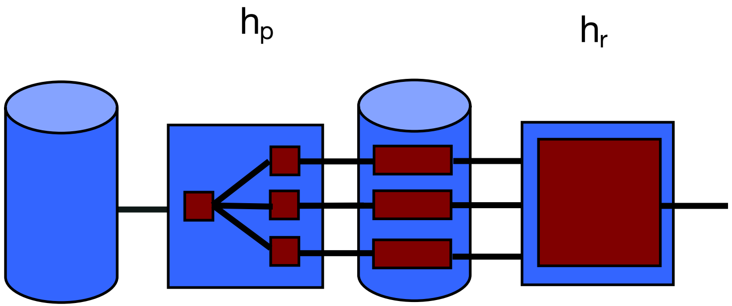


Hash



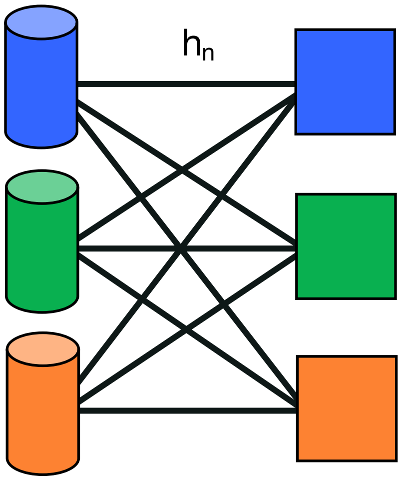
Round-Robin

Remember Hashing?



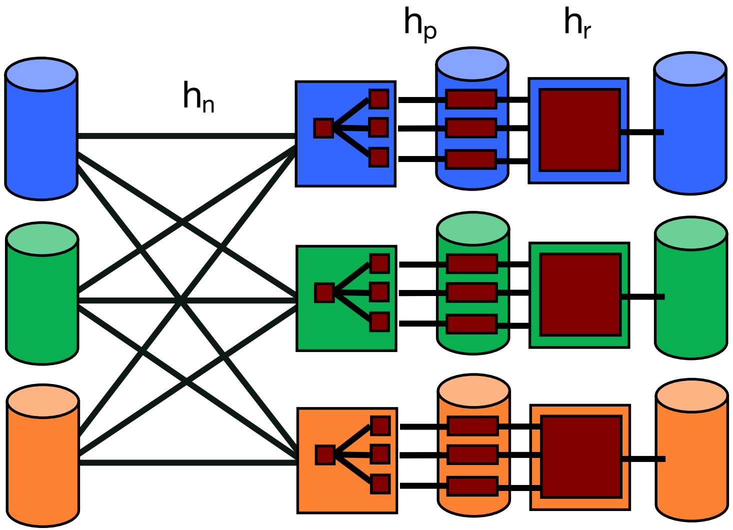
Parallelize me! Hashing

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



Parallelize me! Hashing Part 2

- Receivers proceed with phase 1 in a pipeline
  as data streams in
  - from local disk and network



Nearly same as single-node hashing

Near-perfect speed-up, scale-up!
Streams through phase 1, during which time every component works at its top speed, no waiting.

Have to wait to start phase 2.

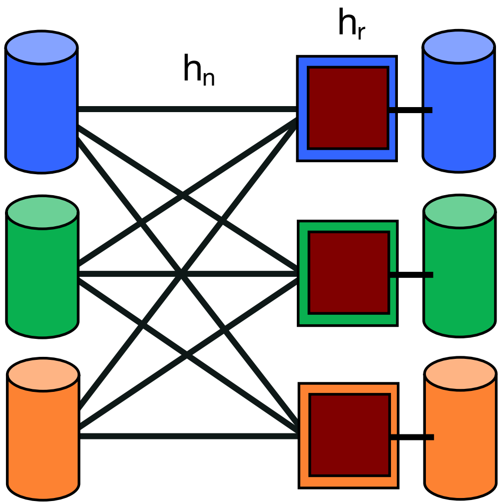
Hash Join?

- Hmmm….

If you have enough machines… Naïve parallel hash join

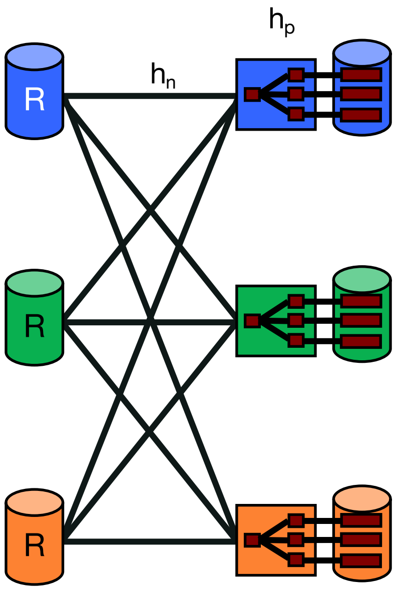
- Phase 1: *shuffle* each table across machines (h_n)
  - Parallel scan streaming out to network
  - **Wait** for building relation to finish
  - Then stream probing relation through it
- Receivers proceed with naïve hashing in a pipeline *as probe data streams in*
  - from local disk and network
  - Writes are 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



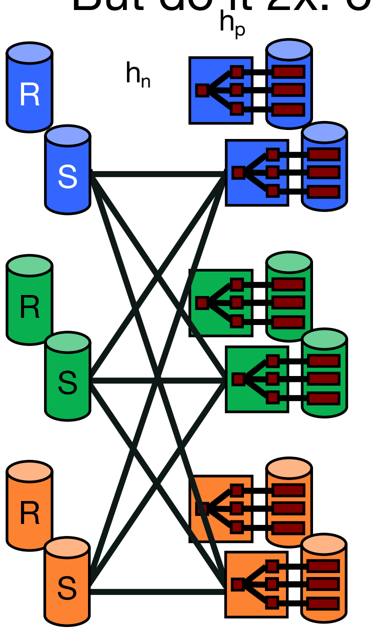
Parallel Grace Hash Join Pass 1

- Pass 1 is like hashing above



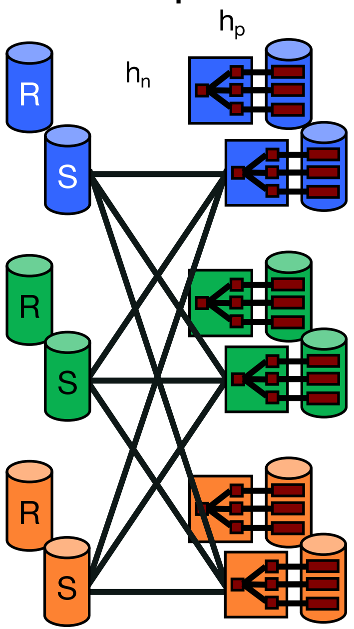
Parallel Grace Hash Join Pass 1 cont

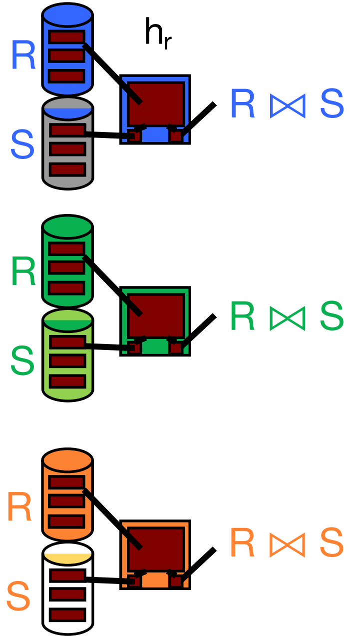
- Pass 1 is like hashing above
  - But do it 2x: once for each relation being joined



Parallel Grace Hash Join Pass 2

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





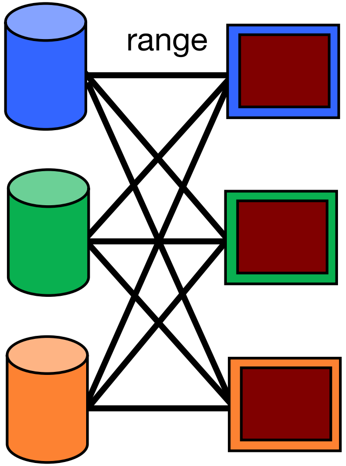
Parallel Grace Hash Join

- 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

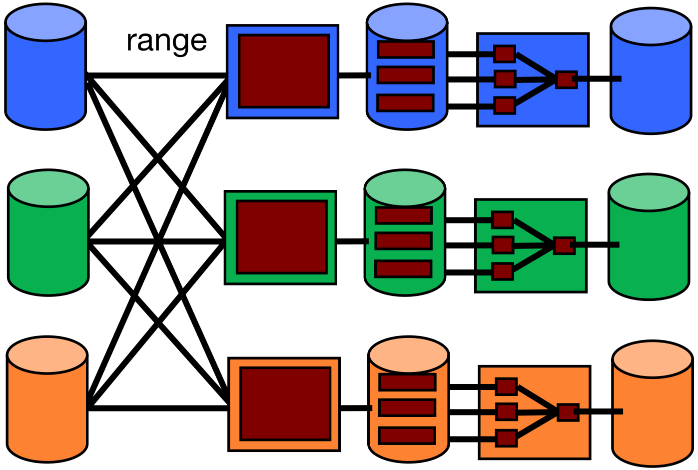
Parallelize me! Sorting Pass 0

- 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, ∞]).



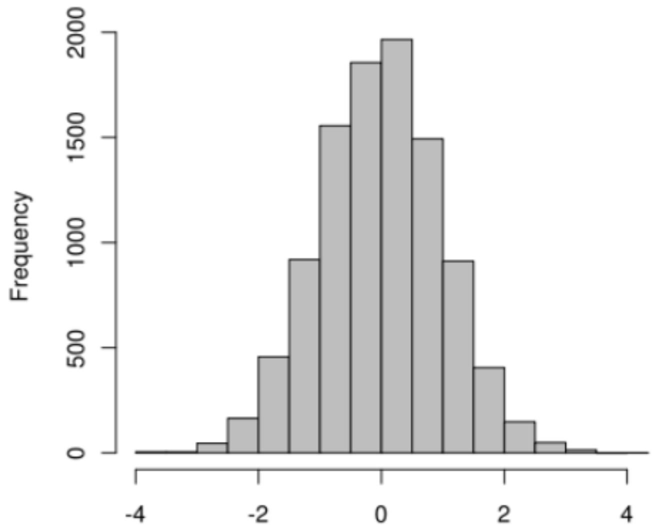
Parallelize me! Sorting Pass 1-n

- 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 are the same #pages?!**
  - **i.e. avoid data skew?**



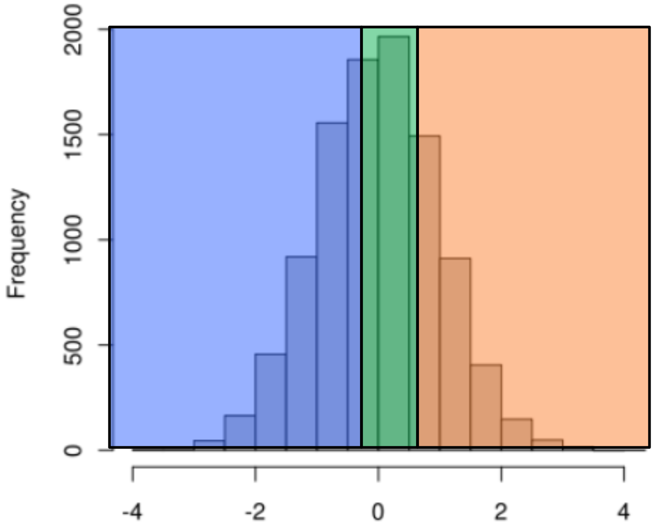
Range partitioning

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



Range partitioning cont.

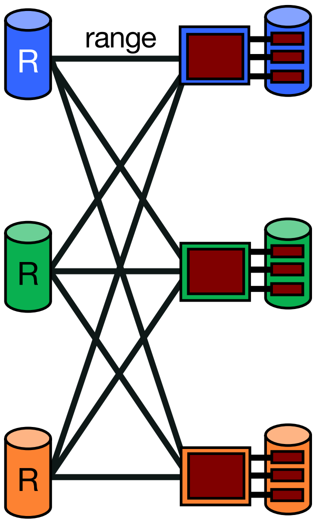
- Would be easy if data small
- In general, can sample the input relation prior to shuffling, pick splits based on sample
- Note: Random sampling can be tricky to implement in a query pipeline; simpler if you materialize first.



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

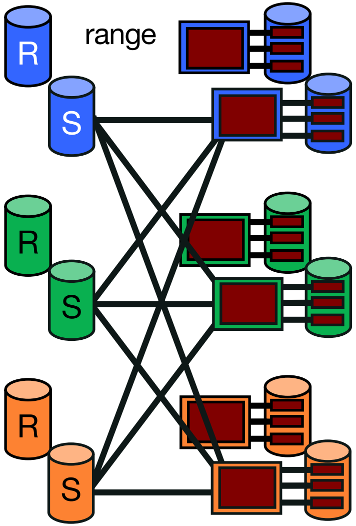
Parallel Sort-Merge Join

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



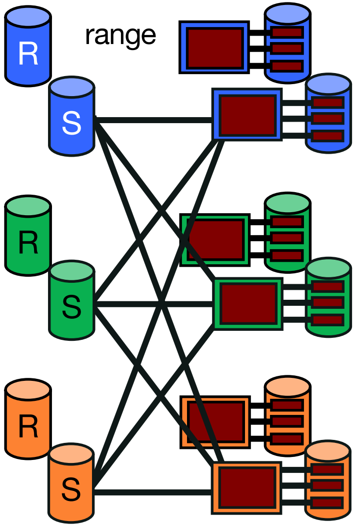
Parallel Sort-Merge Join Pass 0…n-1

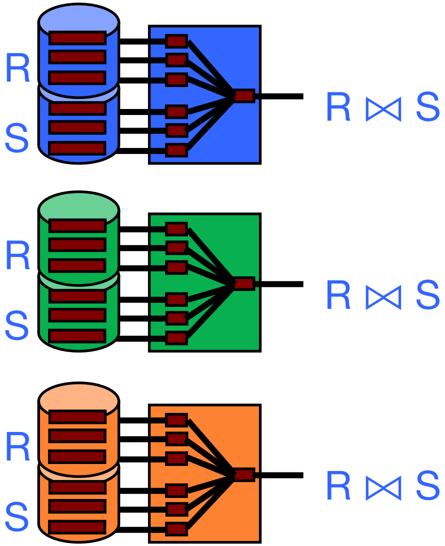
- Pass 0 .. n-1 are like parallel sorting above
  - But do it 2x: once for each relation, with same ranges
  - Note: this picture is a 2-pass sort (n=1); this is pass 0



Pass n (with optimization)

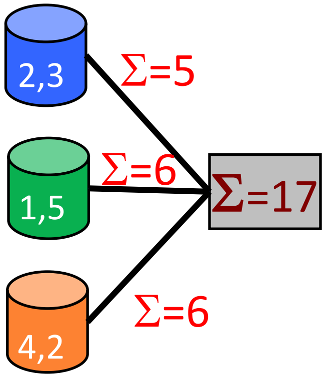
- Pass 0 .. n-1 are like parallel sorting above
  - But do it 2x: once for each relation, with same ranges
- Pass n: merge join partitions locally on each node





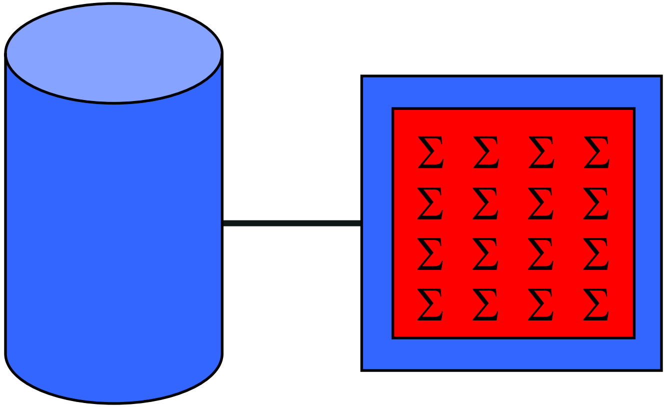
Parallel Aggregates

- 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...



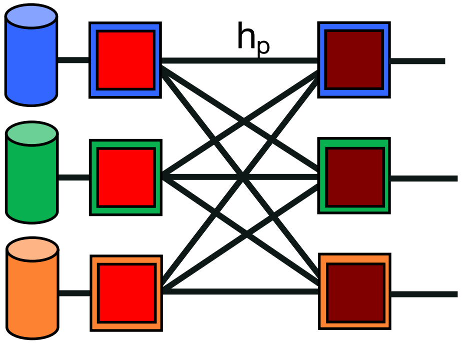
Parallel GroupBy

- 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;`



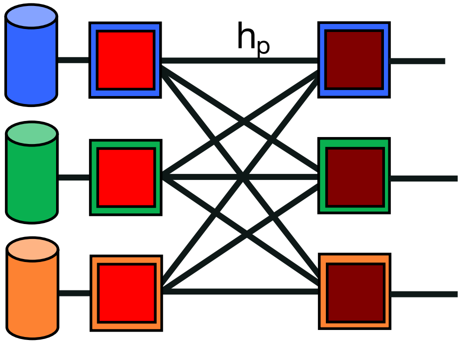
Parallel GroupBy, Cont.

- 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 aggs by a hash function $h_p(k_i)$
  - Compute global aggs 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

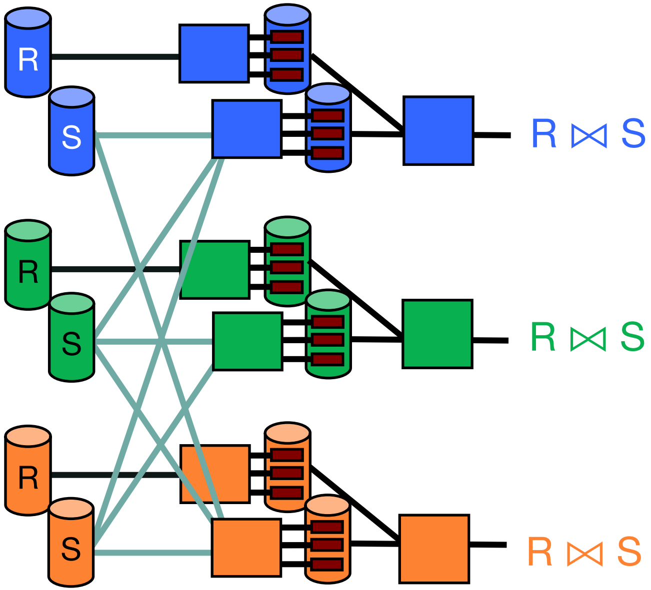


Joins: Bigger picture

- Alternatives:
  - Symmetric shuffle
    - What we did so far
  - Asymmetric shuffle
  - Broadcast join

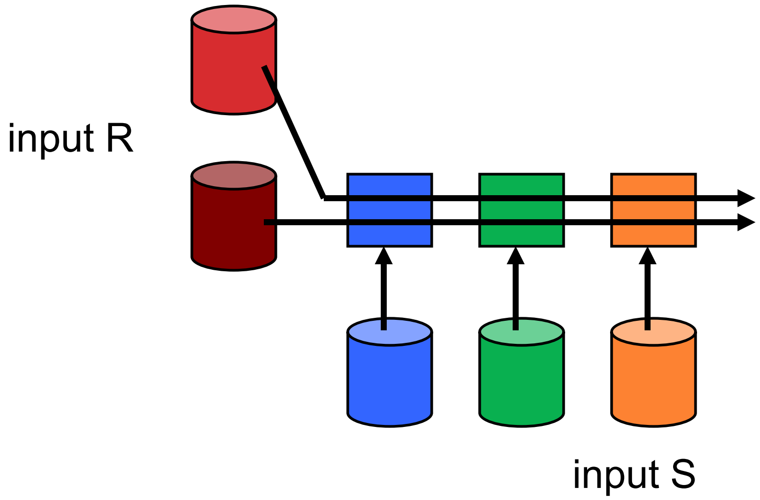
Join: One-sided shuffle

- 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.

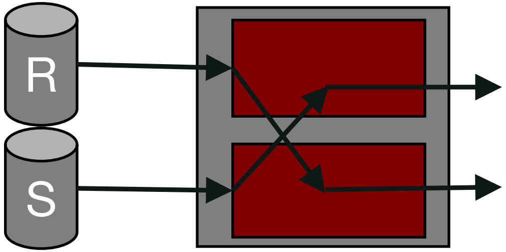


What are “pipeline breakers”?

- 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!

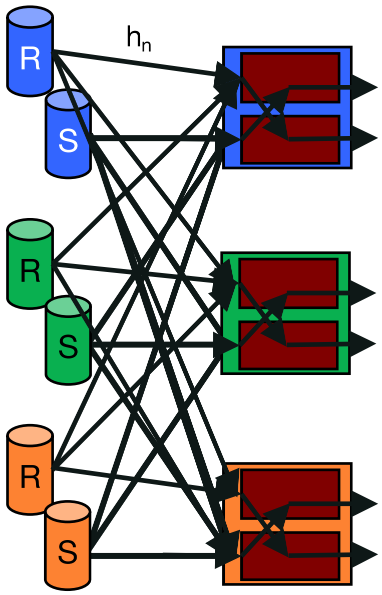


Symmetric (Pipeline) Hash Join cont

- 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!

Extensions

- Parallel Symmetric Hash Join
  - Straightforward—part of the original proposal
  - Just add a streaming partitioning phase up front
  - As in naïve hash join
- Out-of-core Symmetric Hash Join
  - Quite a bit trickier. See the [X-Join paper](#).
- Non-blocking sort-merge join
  - See the [Progressive Merge Join](#) paper



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 tricky stuff related to concurrency control
- Intra-op, Inter-op, & Inter-query parallelism all possible.

Parallel DBMS Summary, Part 2

- Data layout choices important!
- 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

- Transactions require introducing some new protocols
  - distributed deadlock detection
  - two-phase commit (2PC)
- 2PC not great for availability, latency
  - single failure stalls the whole system
  - transaction commit waits for the slowest worker
- More on this in subsequent lectures