Parallel Query Processing

Alvin Cheung

Aditya Parameswaran

R&G Chapters
22.1-22.4

A little history

- Relational revolution
  - 1970’s
  - Single machine: declarative set-oriented primitives

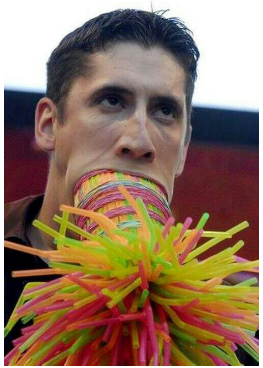
- Parallel relational database systems
  - 1980’s
  - Insight: can parallelize declarative queries!
  - Multiple commodity machines

- “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 4): ~200,000 sec = ~2.31 days

Why Parallelism? Cont.



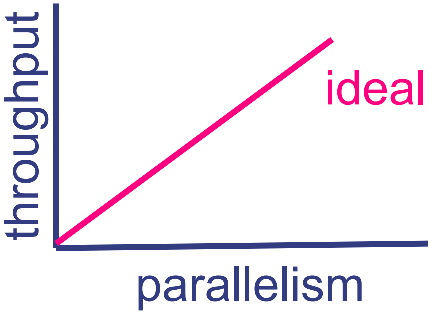
- Scan 100TB
  - At 0.5 GB/sec (see lec 4):
    ~200,000 sec = ~2.31 days
- Run it 100-way parallel:
  - 2,000 sec = 33 minutes
- 1 big problem = many small problems
  - Trick: make them independent
  - Each proceeds at their own pace
  - Thankfully, most rel. operators are amenable to this

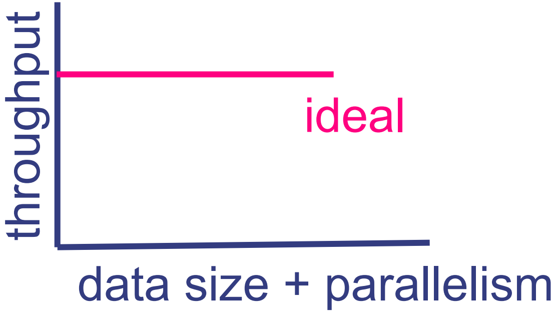
Two Metrics to Shoot For

- Recall: throughput = txns/sec supported

- Speed-up
  - Increase HW
  - Fix workload

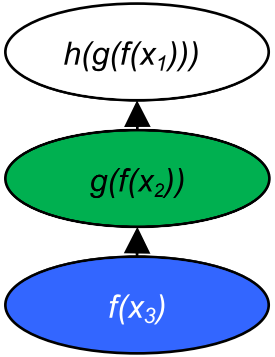
- Scale-up
  - Increase HW
  - Increase workload





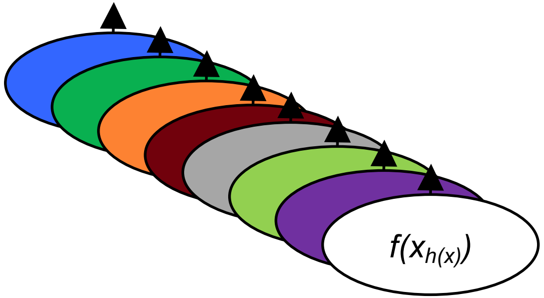
Roughly 2 Kinds of Parallelism

⬬ : 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



Partition
scales up to amount of data

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

Can be combined!

We'll get more refined soon.

Particularly Easy for Databases!

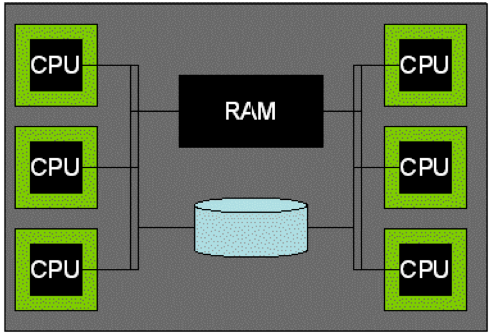
- Lots of Data & Parallelizable Operations:
    - Batch operations on sets of data (relations)
    - Pre-existing divide-and-conquer algorithms
    - Natural pipelining w/ iterator model

- 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

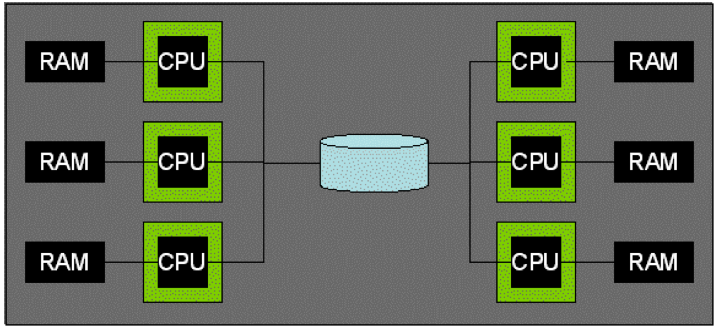
- These insights emerged from the parallel dbms work in the 80’s
    - Reimagined or re-learned in the context of “Big Data” in the 2000s

Parallel Architectures

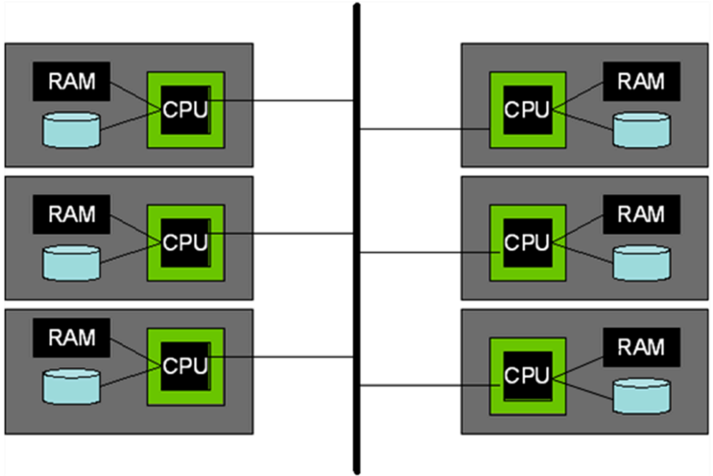
Shared Memory
(Similar to modern computers)



Shared Disk
(Usually w/ some networked file system)



Shared Nothing
(cluster)

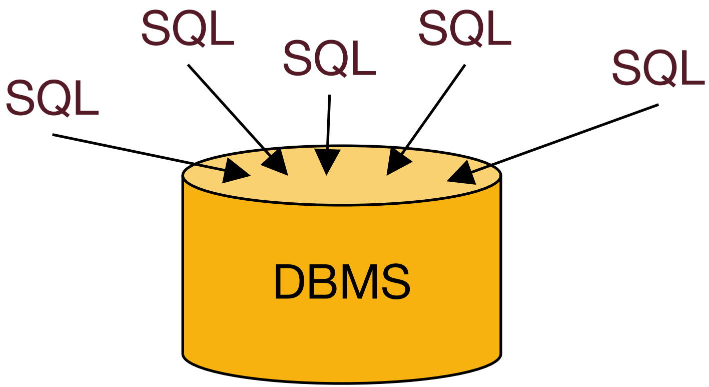


Shared Nothing

- 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

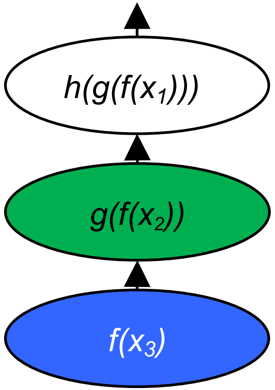
Kinds of Query Parallelism: Inter vs. Intra

- 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)

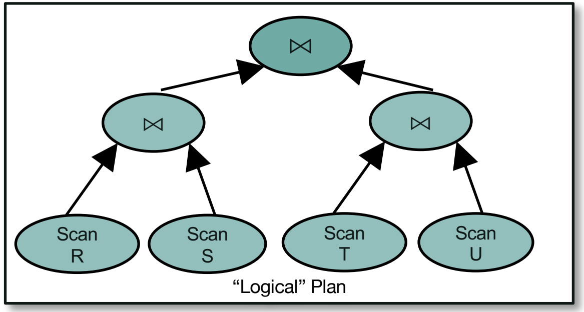


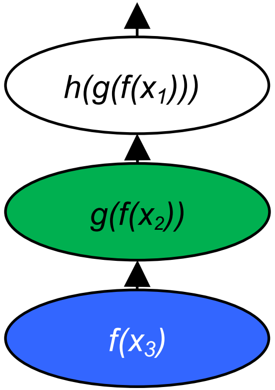
Same tuples get processed by different operators “in a pipeline”

Example: Pipeline Parallelism

Intra Query – Inter-operator Part 2

- Intra-query
  - Inter-operator

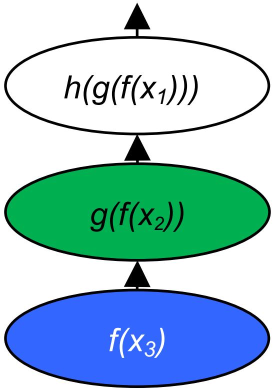




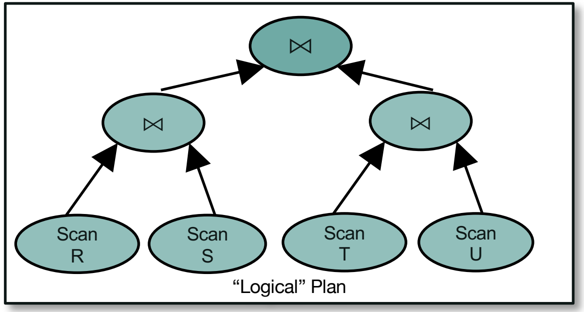
Pipeline Parallelism

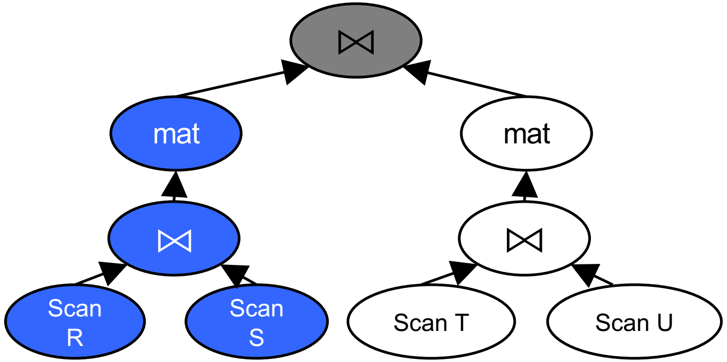
Intra Query - Inter-Operator Part 3

- Intra-query
  - Inter-operator



Pipeline Parallelism





Bushy (Tree) Parallelism

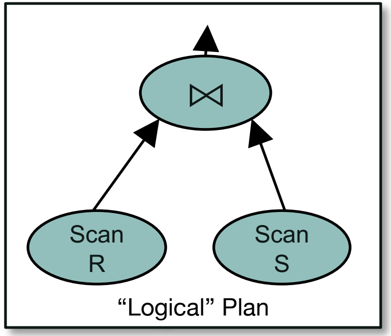
Same tuples get processed by different operators "in a pipeline"

vs.

Different components of the plan proceed in parallel on different data

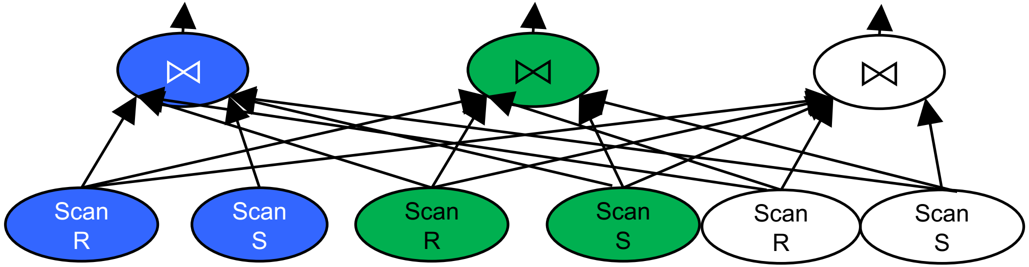
Intra Query – Intra-Operator

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



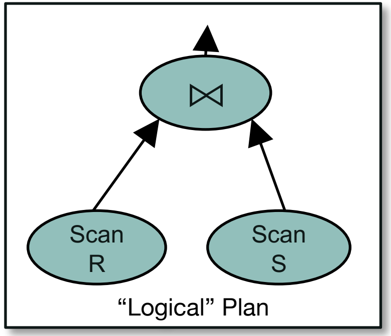
Kinds of Query Parallelism, cont.

- Intra-query
  - Intra-operator



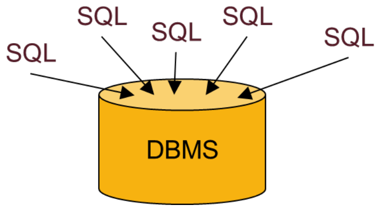
Partition Parallelism

Same operations in parallel on different data partitions

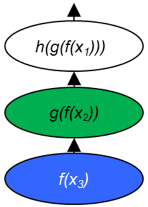


Summary: Kinds of Parallelism

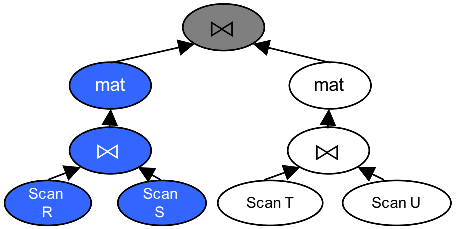
- Inter-Query



- Intra-Query
  - Inter-Operator

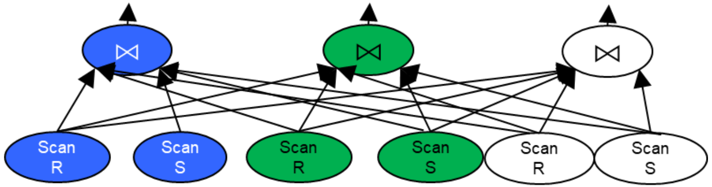


Pipeline Parallelism



Bushy (Tree) Parallelism

  - Intra-Operator (partitioned)

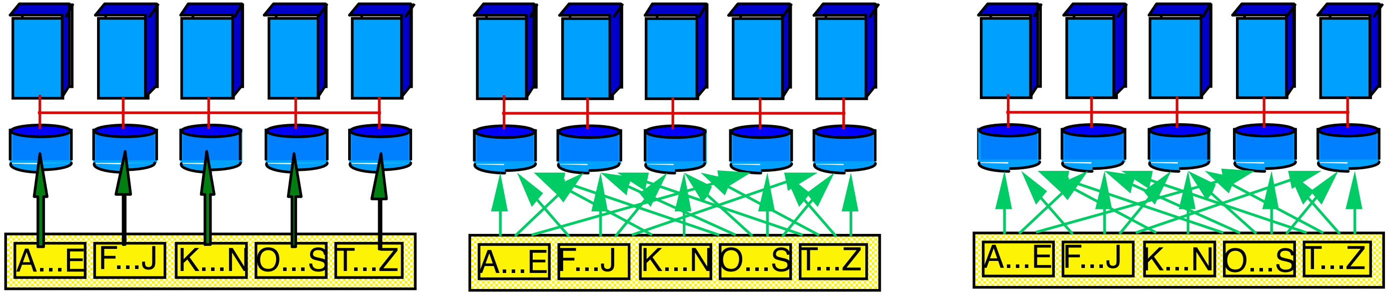


Partition Parallelism

INTRA-OPERATOR PARALLELISM

Data Partitioning

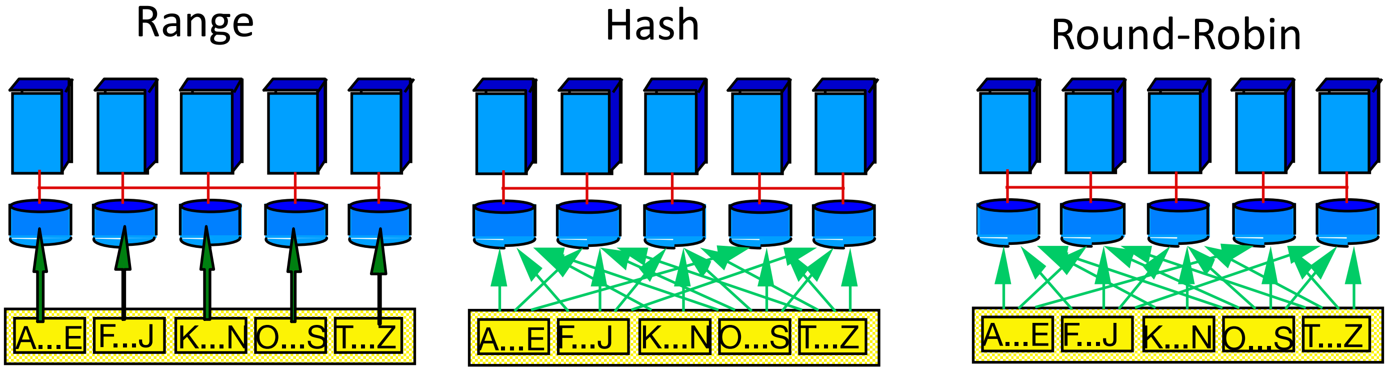
- How to partition a table across disks/machines
  - A bit like coarse-grained indexing!



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

Data Partitioning

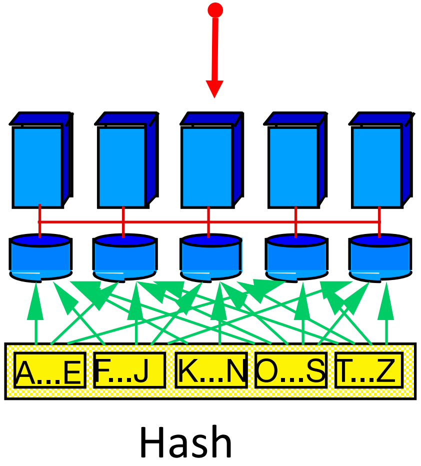
- How to partition a table across disks/machines
  - A bit like coarse-grained indexing!



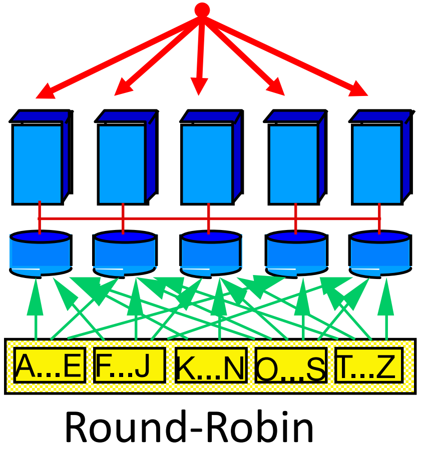
  - Shared nothing particularly benefits from "good" partitioning
    - Reduces network traffic
    - Better if operations are “localized” to certain nodes
  - Indexes can be built at each partition
    - E.g., a B+tree at each node

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
- Otherwise or if we use round-robin partitioning
  - Have to broadcast lookup (to all nodes)



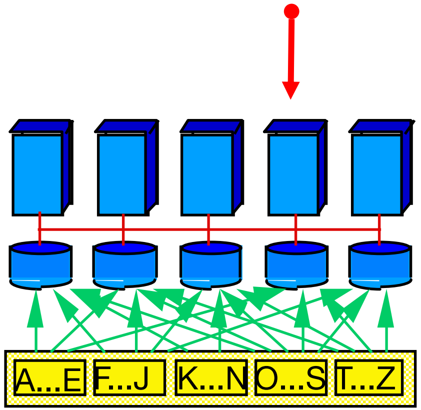
Hash



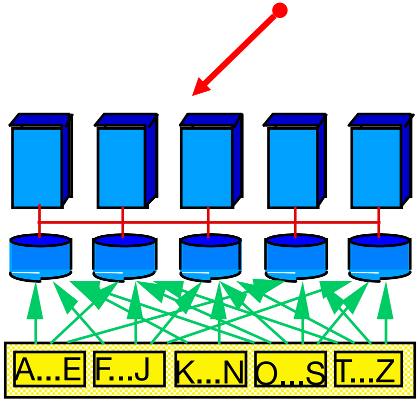
Round-Robin

What about Insert?

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



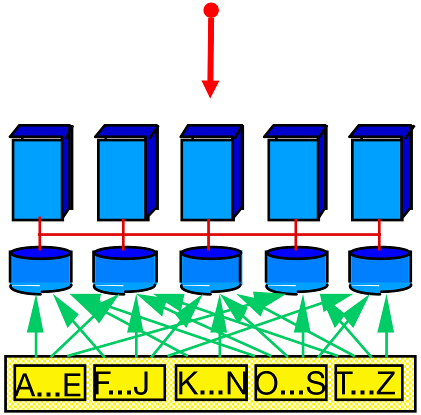
Hash



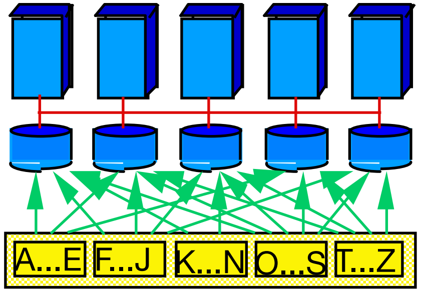
Round-Robin

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



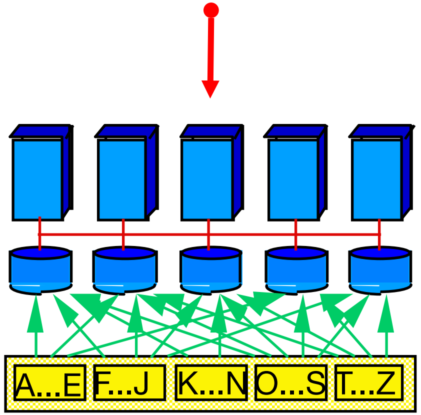
Hash



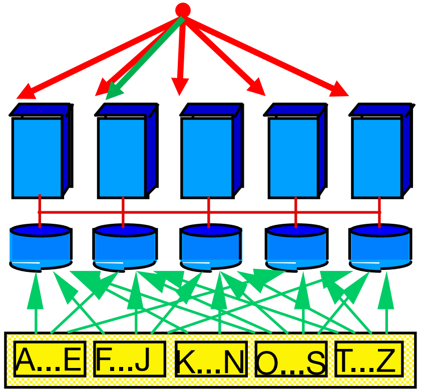
Round-Robin

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

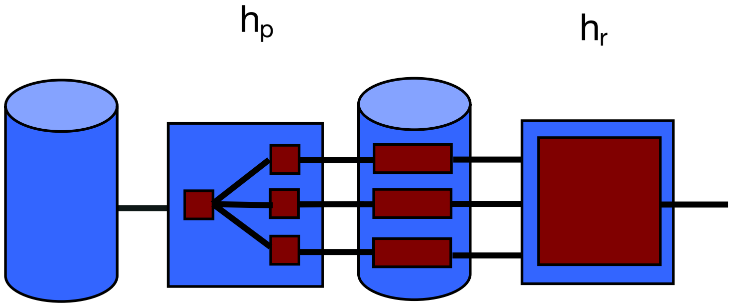


Round-Robin

Parallel Scans

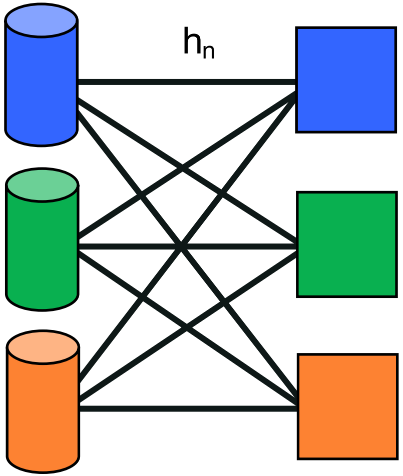
- Scan in parallel, merge (concat) output
- σ_p : skip entire sites that have no tuples satisfying p
    - Range or hash partitioning on attributes involved in *p* benefits from this
    - Round-robin does not, nor does partitioning on other attributes

Remember Hashing?



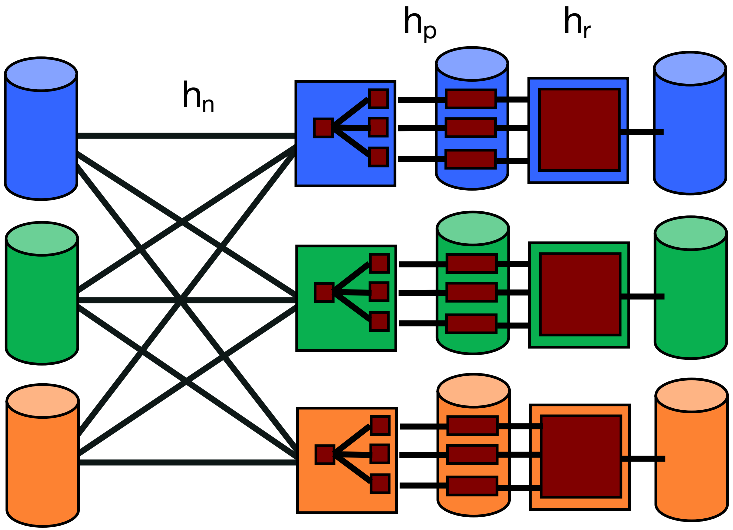
Parallelize me! Hashing

- 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