

Parallel DBMSs

Instructor: Peter Baumann
email: pbaumann@constructor.university
tel: -3178
office: room 88, Research 1

Overview

- Motivation
- Data partitioning
- Query operator parallelization
- Skew
- Optimization

Parallelization: Principle

- Goal
 - Improve performance by executing multiple operations in parallel
 - More processors →
each query faster / same speed on more data / more transactions per second / ...
- In LAN: $\text{cost}(\text{network}) \ll \text{cost}(\text{disk IO})$
- Key challenge
 - overhead & contention can kill performance

Parallelization Variants

■ Pipeline parallelism

- many machines each doing one step in a multi-step process



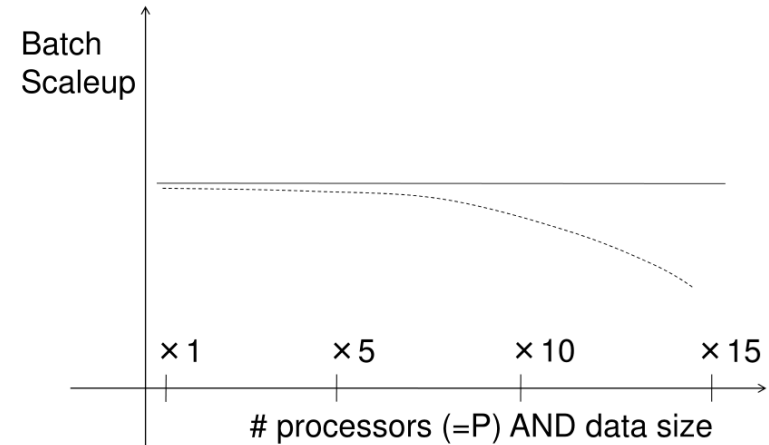
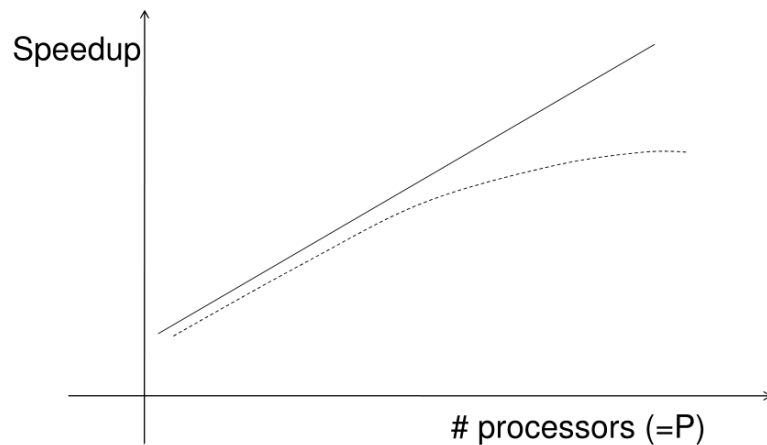
■ Partition parallelism

- many machines doing the same thing to different pieces of data



Speedup & Scaleup

- Speedup: faster
- Scaleup: do more
- Linear vs non-linear (sub-linear)



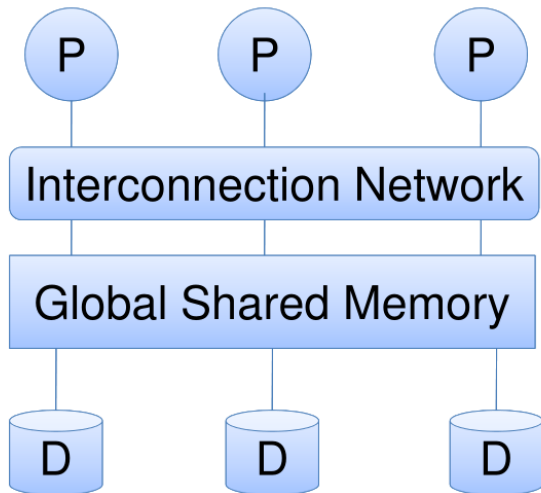
Challenges to Linear Speedup & Scaleup

- Startup cost
 - Cost of starting an operation on many processors
- Interference
 - Contention for resources between processors
- Skew
 - Slowest processor becomes the bottleneck
- Blocking operations
 - Can continue only once all results are seen: sort, top-k, aggregation, ...

Architectures for Parallel Databases

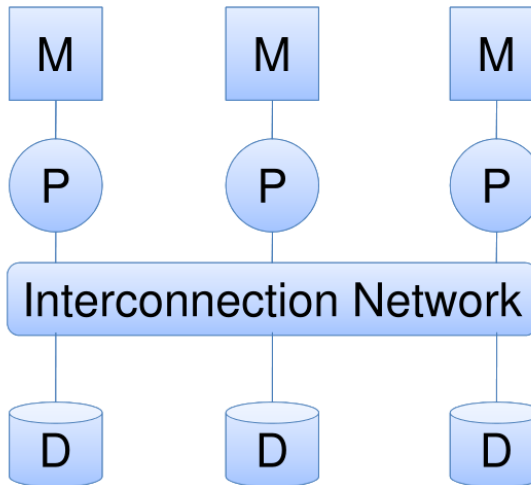
Shared **memory**

Sequent, SGI, Sun, NEC



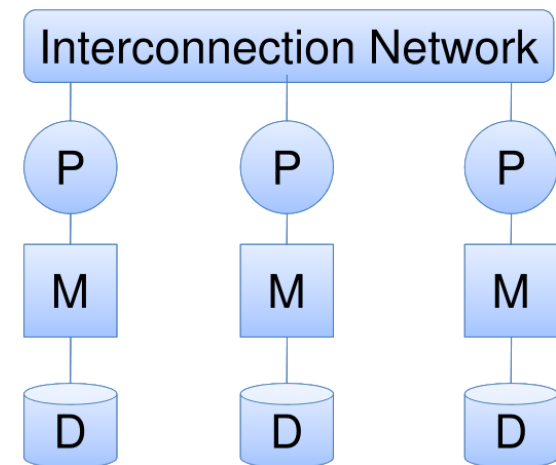
Shared **disk**

VMSccluster, Sysplex



Shared **nothing**

Tandem, Teradata, SP2



most **scalable**

- minimizes interference by minimizing resource sharing
- commodity hardware

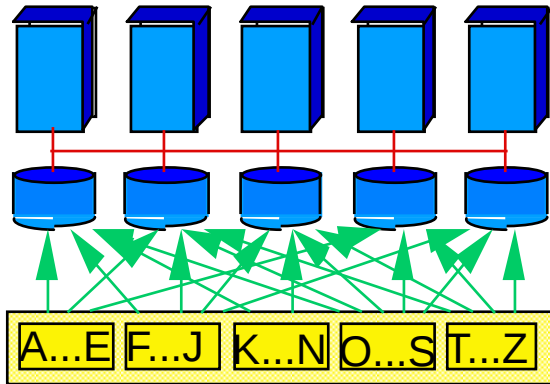
most **difficult**

Data Placement: How to Partition?

- Partitioning always necessary: tuples assigned to set of disks / processors
 - Static or during query

Round Robin

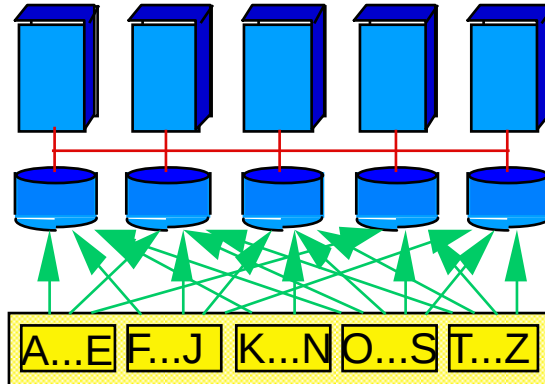
tuple $t_i \rightarrow \text{chunk } (i \bmod P)$



😊 balance load, full scan
☹ range queries

Hash partitioning

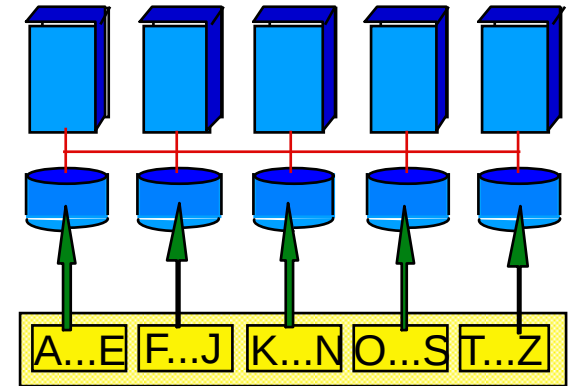
tuple $t \rightarrow \text{chunk } h(t.A) \bmod P$



😊 equijoins, point queries, full scan; ☹ range queries

Range partitioning

tuple $t \rightarrow \text{chunk } i \text{ if } v_{i-1} < t.A < v_i$



😊 equijoins, range queries, group-by

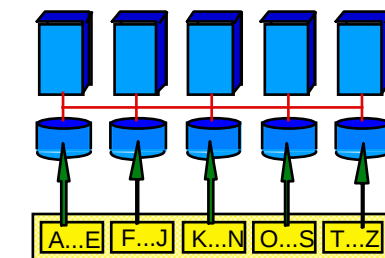
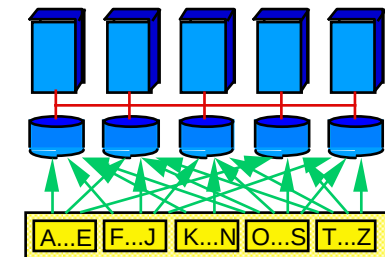
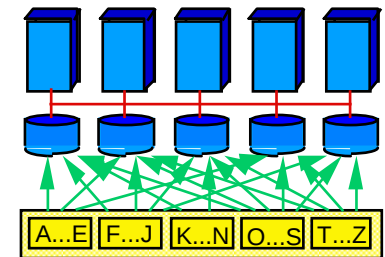
Partition vector =
list of switch points $[v_1; \dots; v_p]$

|| of Query Operators

- Discussion assumes:
 - read-only queries
 - shared-nothing architecture
 - n processors, $P_0, \dots, P_{n-1}$, and n disks $D_0, \dots, D_{n-1}$,
where disk D_i is associated with processor P_i
- Will look at filter, sort, join
- PS: Shared-nothing architectures can be efficiently simulated on shared-memory and shared-disk systems

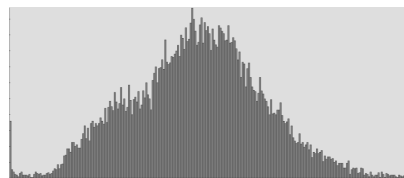
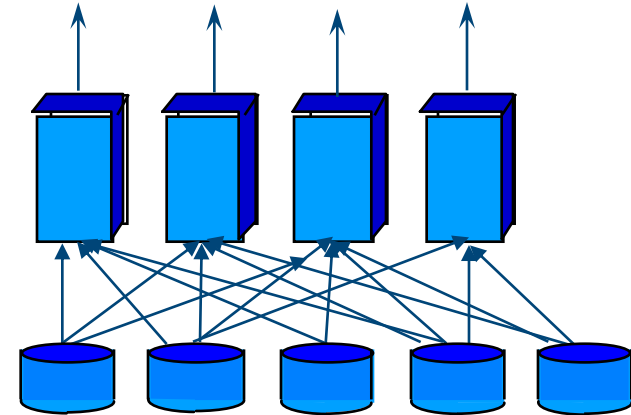
|| Filter

- How is work distribution among processors?
 - Point query $\sigma_{A=v}(R)$, range query $\sigma_{v1 < A < v2}(R)$
 - Load balancing
- Round robin: **all** servers do the work
- Hash partition:
 - **One** server for $\sigma_{A=v}(R)$
 - **All** servers for $\sigma_{v1 < A < v2}(R)$
- Range partition: **one** server does the work



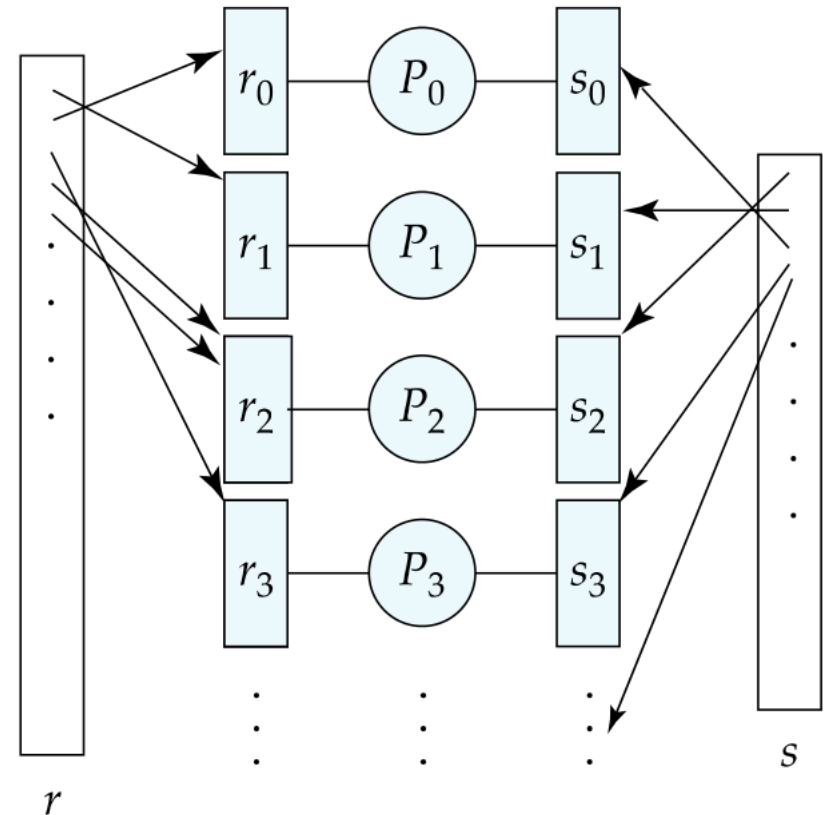
|| Sort-Merge with Range-Partitioning

- Choose **partitioning vector**
- **Scan** table in parallel, **range-partition** as you go
- Each processor: **sort** partition locally
 - All execute same operation in parallel, no interaction
 - Can create local index
- Final **merge** operation (trivial: concatenation of sorted partial results)
 - range-partitioning ensures global sortedness
- Problem: **skew** – more later



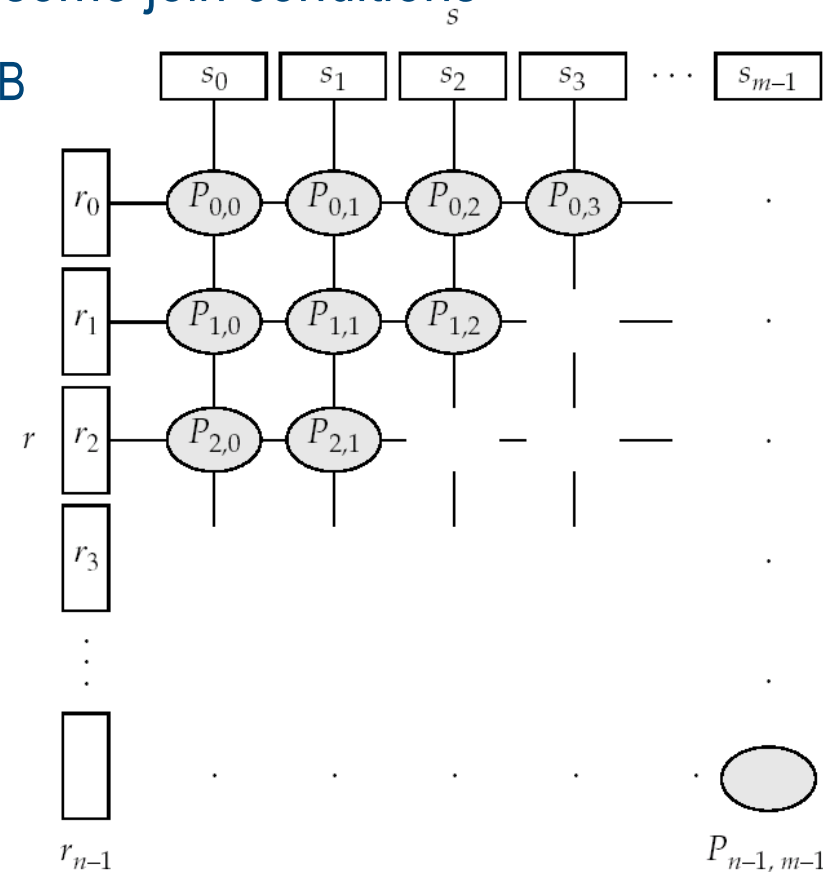
Partitioned Join

- For Equi-Join $R \bowtie_{R.A=S.B} S$:
 - partition input relations, distribute
 - compute join partitions
 - recollect
- Partition R, S on join attrs $R.A$ & $S.B$
 - No need to sort
 - Range, hash partitioning all fine
- Corresponding partitions R_i & $S_i \rightarrow$ processor P_i ,
- P_i **locally** computes $R_i \bowtie_{R_i.A=S_i.B} S_i$
 - Any standard join method



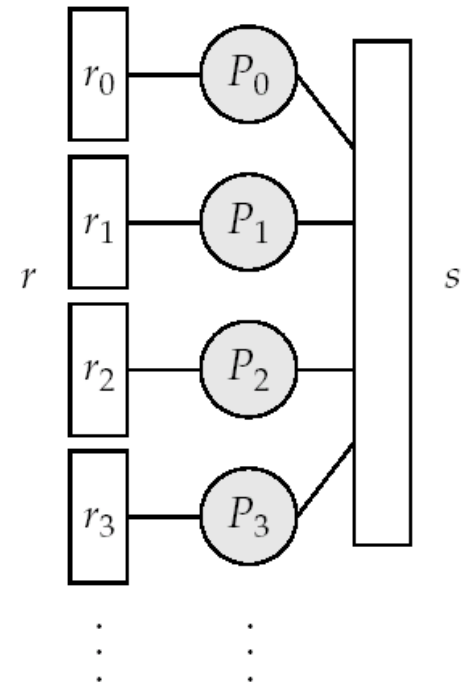
Fragment-and-Replicate Join

- Observation: Partitioning not possible for some join conditions
 - Ex: non-equijoin conditions, such as $R.A > S.B$
- fragment & replicate



Fragment-and-Replicate Join

- Observation: Partitioning not possible for some join conditions
 - Ex: non-equijoin conditions, such as $R.A > S.B$
- fragment & replicate
- Special case: **asymmetric fragment-and-replicate**
 - R partitioned; any partitioning technique can be used
 - **small** S replicated across all processors



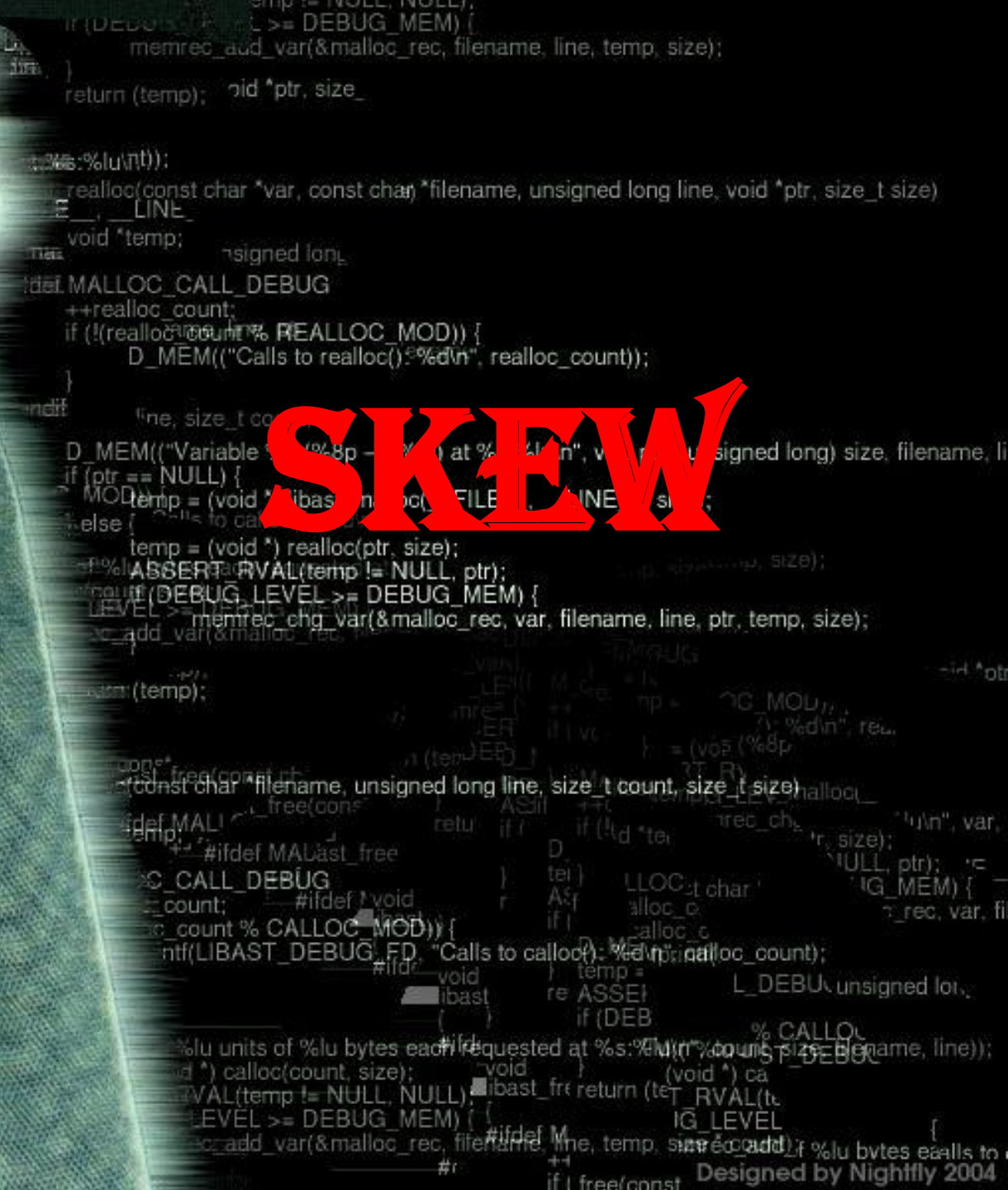
Cost of || Evaluation

- no skew in partitioning, no || overhead: **expected speed-up is $1/n$**
- skew & overheads taken into account, || time estimate:

$$T_{\text{part}} + \max(T_0, \dots, T_{n-1}) + T_{\text{asm}}$$

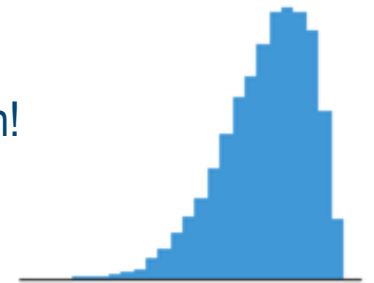
where:

- T_{part} time for partitioning the relations
- T_{asm} time for assembling the results
- T_i time taken for operation at processor P_i
(needs to be estimated taking into account skew and time wasted in contentions)

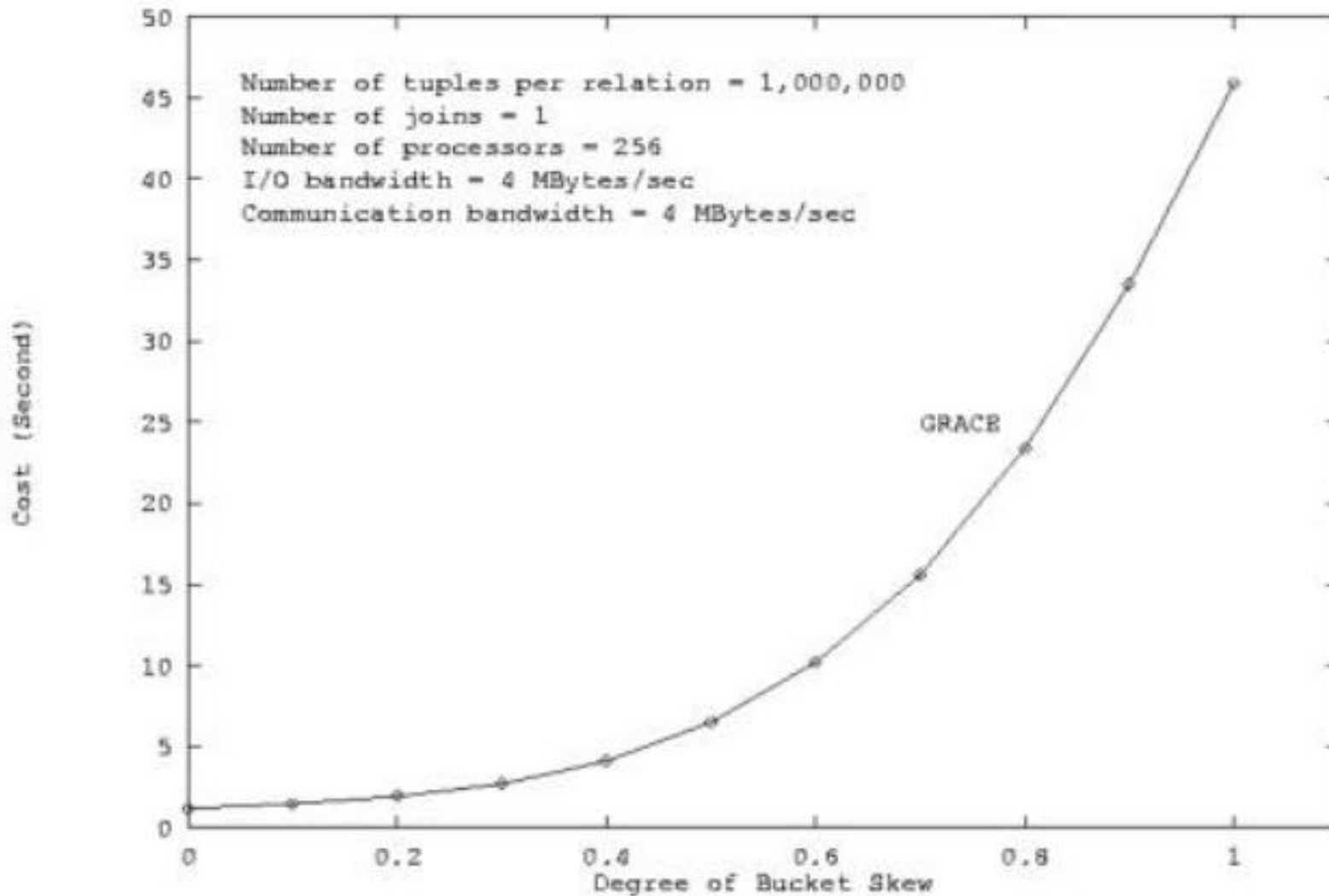


Skew

- distribution of tuples to disks may be **skewed**
= some disks have many tuples, while others may have fewer tuples
- **Attribute-value skew**
 - Many tuples share same values, few distinct values;
all tuples with same value for partitioning attribute end up in same partition!
 - Affects hash-partitioning, range-partitioning
- Consequence: **Partition skew**
 - Range-partitioning: bad partition vector → too many tuples to some partitions, too few to others
 - Less likely with hash-partitioning if hash-function good



Skew Kills || Performance



Handling Skew in Range-Partitioning

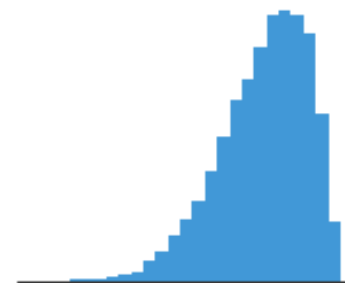
- Method for a balanced partitioning vector
 - Sort relation on partitioning attribute
 - Scan relation in sort order
 - After every $1/n^{\text{th}}$ of relation: add attribute value of next tuple to partition vector
- Drawbacks:
 - Imbalance possible if duplicates in partitioning attributes
 - Best for initial table load; frequent updates may change=disturb distribution
 - Table scan expensive
- Alternative: histograms ↩

Histograms

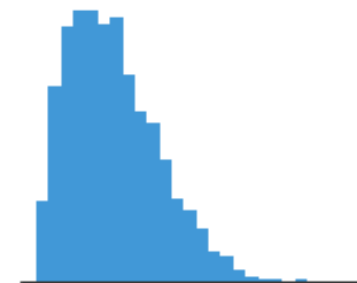
- Helps finding balanced partitioning vector
- Histogram can be constructed by
 - **scanning** complete relation
 - *expensive*
 - **sampling**
 - *Accuracy?*
 - *Over time, with updates?*



symmetric, unimodal



skew left



skew right



uniform



bimodal

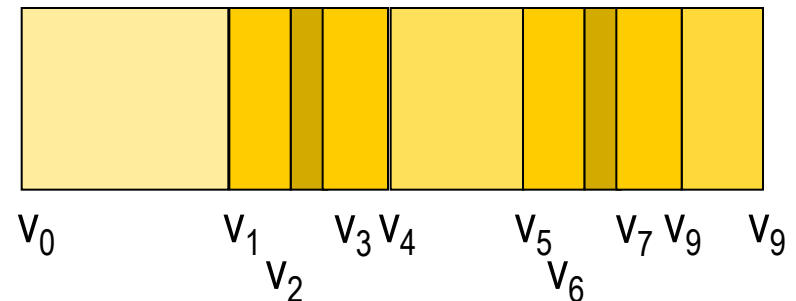
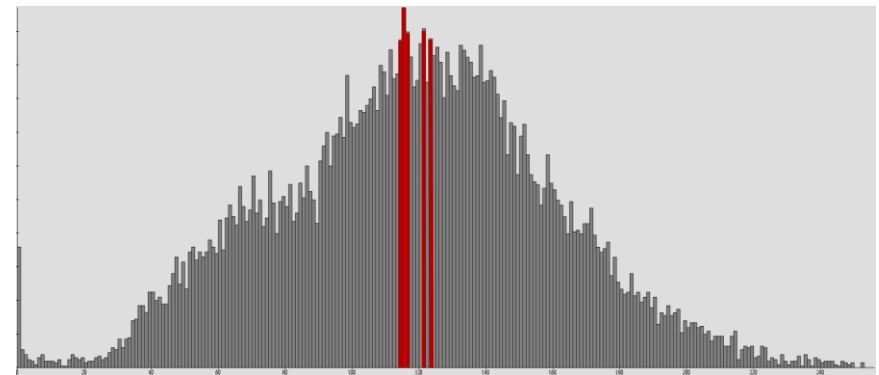


multimodal

[Atlassian]

Histograms Types

- Two main types of histograms:
- frequency histogram
 - (attribute value, frequency) pairs for N most frequent attribute values
 - optimizer estimates selectivity of equality predicates
- quantile histogram
 - = equidepth range histogram
 - optimizer estimates selectivity of range predicates



Histograms in Practice: Oracle

- **single histogram**, can act as either frequency histogram or equidepth histogram
 - frequency version used when number of unique values of attribute is low
 - **switches** to equidepth histogram if domain is large and number of unique values crosses a threshold
- Default threshold value is 75
 - will be number of buckets in equidepth histogram
- Oracle provides view, *all_tab_histogram*, to read histogram information

Histograms in Practice: DB2

- **quantile** histogram
 - 20 buckets by default to approximate data distribution
 - stored in system table *SYSIBM.SYSCOLDIST*
- **frequency** histogram
 - Top 10 by default, can be specified by DBA
 - used to estimate selectivity of equality predicates

Histograms in Practice: MS SQL Server

- mix of **frequency** and **equidepth** histogram
 - frequency of bucket boundaries + number of tuples in bucket
 - number of buckets can go up to 200
- Histograms by default generated with sampling
- stored procedure *DBCC SHOW STATISTICS* extracts histogram information

Histograms in Practice: PostgreSQL

- mixture of **end biased** and **equidepth** histograms
- Histograms stored in relation *pg_stats* catalog table
 - most frequent values stored as an array in the *most_common_vals* column
 - equi-depth histogram stored as two arrays:
 - *frequency of corresponding buckets*
 - *bounds of the buckets*
- 10 buckets by default

Different Approach: Virtual Partitioning

- create **large** number of partitions
 - say, 10x to 20x number of disks / processors
- Assign virtual processors to partitions
 - round-robin or based on cost estimate
- Basic idea:
 - If any normal partition skewed, this skew spread over several virtual partitions
 - Skewed virtual partitions spread across several processors, so work distributed evenly

Taxonomy for Parallel Query Evaluation

- So far: looked at operators – big picture?

- **Inter-query ||**

- 1 query → 1 processor

- **Intra-query ||:**

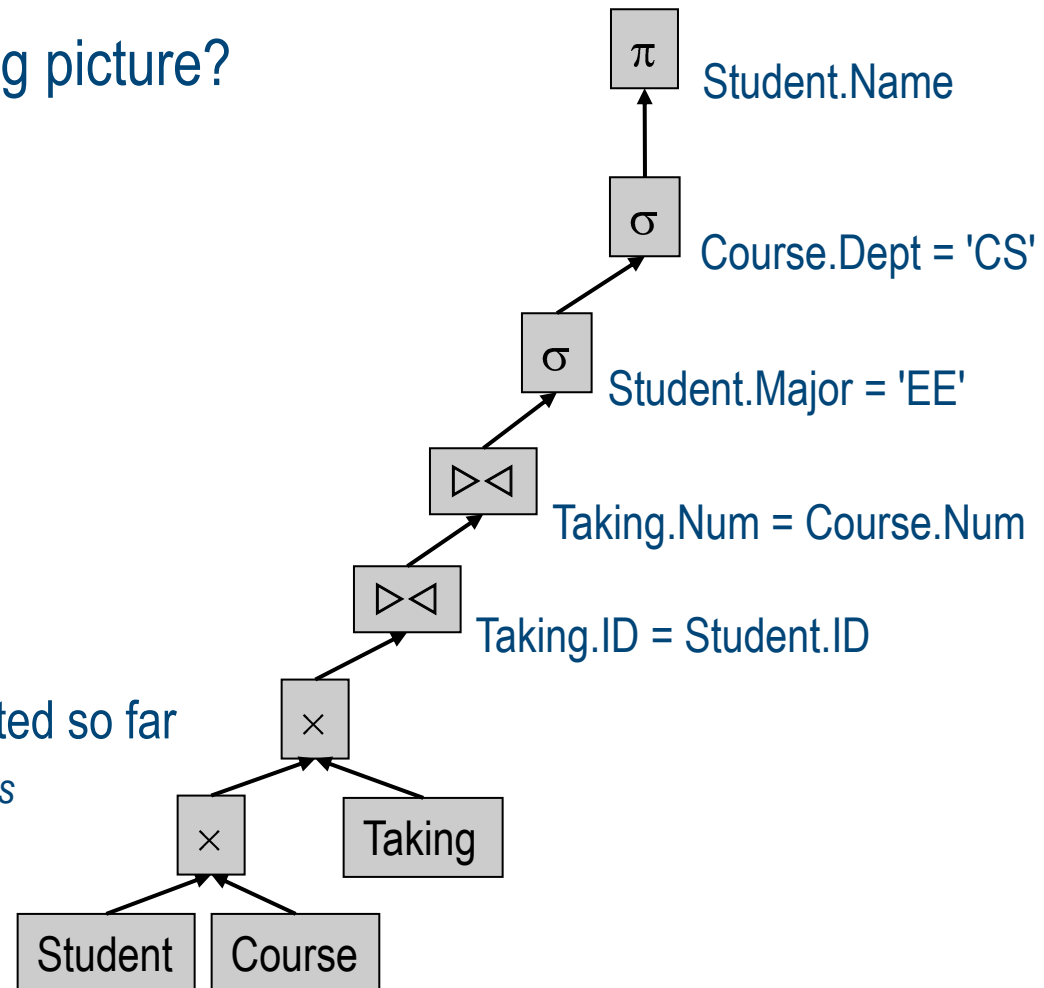
- **Inter-operator ||**

- *query runs on multiple processors*
- *operator runs on one processor*

- **Intra-operator ||**

← inspected so far

- *operator runs on multiple processors*
- *most scalable*



Interquery Parallelism

- Queries/transactions execute in parallel with one another
 - Increases transaction throughput; used primarily for larger #TAs per second
- Easiest ||
- **Locking** & logging coordinated by passing messages between processors
 - Data in local buffer may have been updated at another processor
- **Cache-coherency** challenging: buffer reads and writes need latest version
 - Simple cache coherency protocol for shared disk systems:
Lock page; read page from disk; write page if modified; unlock page
 - Each page has home processor, all page requests sent to home processor

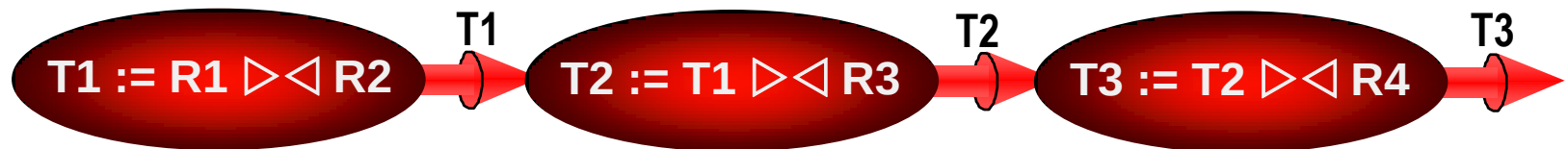
Intra-Query Parallelism

- 1 query $\rightarrow$ n processors/disks;
 - important for long-running queries
- Two complementary forms:
- **Inter-operator** || – execute query operations in parallel, aka “pipelining”
- **Intra-operator** || – parallelize execution of each individual operation in query

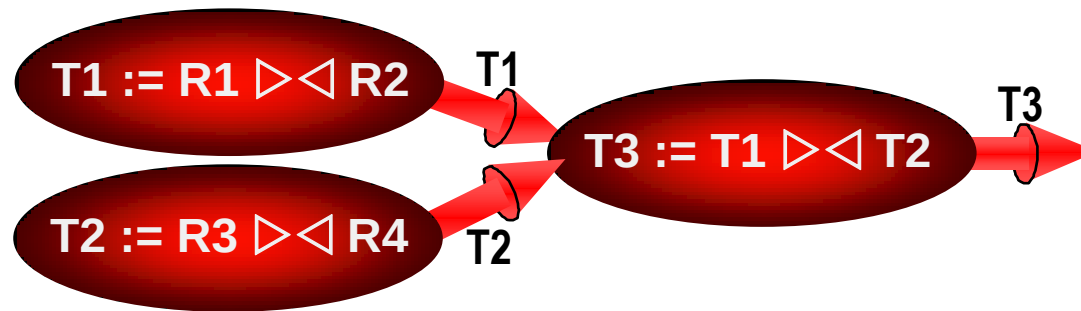
Inter-Operator Parallelism

- Execute query operations in parallel

- Ex: pipelining of $R1 \triangleright \triangleleft R2 \triangleright \triangleleft R3 \triangleright \triangleleft R4$



- Even better:



- Tuple streams

→ avoid (disk) storage of large intermediate tables

- Drawbacks:

- Useful with small #processors, not for #procs >> #ops
- Not possible to parallelize blocking operations (e.g., aggregate, sort)
- Skew: cost of operators can vary significantly

Intra-Operator Parallelism

- parallelize execution of each individual operation in query
 - See earlier examples
- Scales better with increasing parallelism
 - #tuples processed by operation typically $\gg$ #operations in query

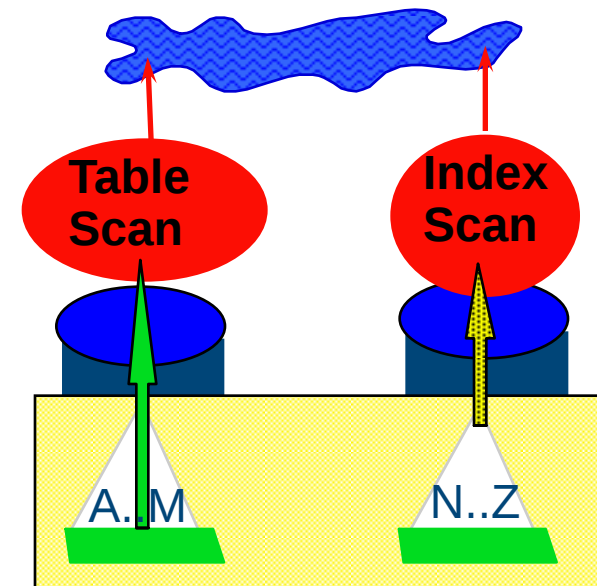
|| Query Optimization

- Query optimization in || databases **significantly more complex** than in sequential databases; *ongoing research!*
 - | parallel evaluation plans | >> | sequential evaluation plans |
- Cost models more complicated
 - How to parallelize each operation, how many processors to use? What operations to pipeline? what operations to execute independently in parallel? what operations to execute sequentially? ...etc.
- **Heuristic I:** parallelize every operation across all processors (MapReduce!)
- **Heuristic II:** choose most efficient sequential plan, parallelize that plan
- Critical:
 - good physical organization (partitioning technique)
 - Good resource need estimate

What's Wrong With That?

- Best serial plan != Best || plan! ...why?
- Trivial counter example:
 - This query:


```
SELECT *
FROM telephone_book
WHERE name < "NoGood"
```
 - Table partitioned with local index at two nodes
 - Range query addresses **all** of node 1 and **1%** of node 2
- Assessment:
 - Node 1 should best do a **scan** of its partition, Node 2 should best use **index**



Distributed Databases

- **Parallel** database system:
 - One DB server environment (cloud, data center), stores all data
 - Typically: processing nodes + Storage-Area Network (SAN) + fast network
- **Distributed** database system:
 - Data stored across several geographically remote sites → slow, failing network
 - each site managed by independent DB server
 - Distributed transactions
- **Failures** to be expected always
 - More hardware → more failure probability
 - Replication

Summary

- Parallel processing boosts performance
 - Massive research done, continuing
- Challenges:
 - Data placement, data skew
 - Parallel bulk load, data maintenance (updates, index), online repartitioning, ...
 - Complex optimization
- Even more challenging: distributed query processing
 - Independent nodes; failures; ...