JACOBS UNIVERSITY

Parallel DBMSs

Not: Ramakrishnan/Gehrke Chapter 22, Part A

But: Silberschatz, Korth, Sudarshan: Database System Concepts, 5th edition

See also: http://www.research.microsoft.com/research/BARC/Gray/PDB95.ppt

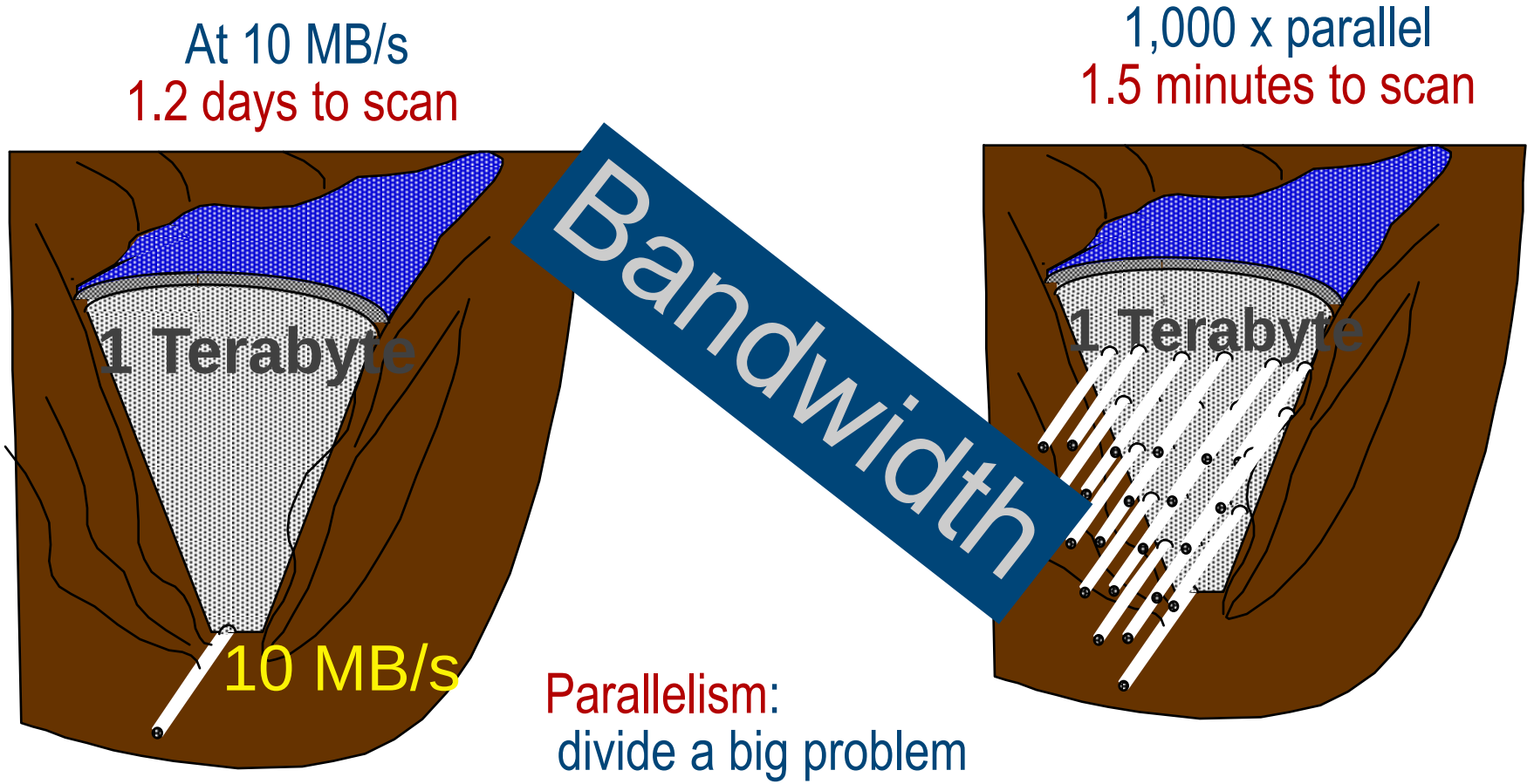
Roadmap

JACOBS UNIVERSITY

- Introduction
- I/O Parallelism
- Interquery Parallelism
- Intraquery Parallelism
- Intraoperation Parallelism
- Interoperation Parallelism
- Query Optimization
- Design of Parallel Systems

Why Parallel Access To Data?

JACOBS UNIVERSITY



Parallelism:
divide a big problem
into many smaller ones
to be solved in parallel.

Introduction

JACOBS UNIVERSITY

- Parallel machines are becoming quite common and affordable
  - Prices of microprocessors, memory and disks have dropped sharply
  - Typically today multi-core processors
  - Clouds
- Databases are growing tremendously – "Big Data"
  - large volumes of transaction data collected & stored for later analysis
  - images etc. increasingly stored in databases
- Manifold applications need speedup:
  - large volumes of data
  - complex decision-support queries
  - high throughput for transaction processing

DBMS: The || Success Story

JACOBS UNIVERSITY

- DBMSs are maybe the most successful application of parallelism
  - Teradata, Tandem vs. Thinking Machines, ...
  - Every major DBMS vendor has some || server
  - Workstation manufacturers now depend on || DB server sales
- Reasons for success:
  - Bulk-processing (= partition ||-ism)
  - Natural pipelining
  - Inexpensive hardware can do the trick
  - Users/app-programmers don't need to think in ||

Parallelism in Databases

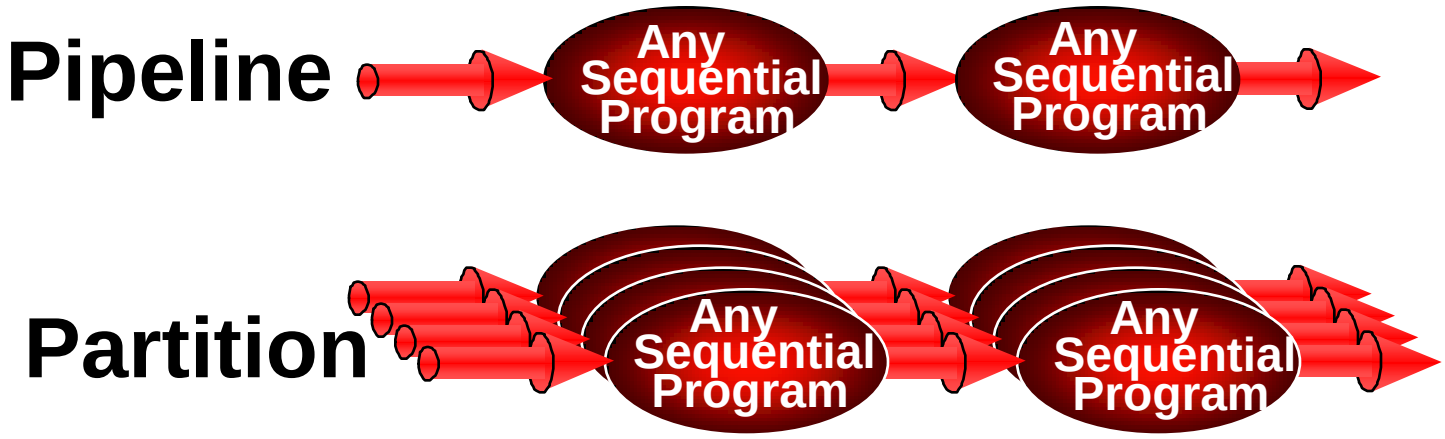
JACOBS UNIVERSITY

- Data can be partitioned across multiple disks for parallel I/O
- Individual relational operations can be executed in parallel
  - (e.g., sort, join, aggregation)
  - data can be partitioned and each processor can work independently on its own partition
- Queries expressed in high level language (SQL / relational algebra)
  - makes parallelization easier
- Different queries can be run in parallel with each other
  - Concurrency control takes care of conflicts
- *Thus, databases naturally lend themselves to parallelism*

Parallel DBMS: Intro

JACOBS UNIVERSITY

- Parallelism is natural to DBMS processing
  - 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
  - Both are natural in DBMS!

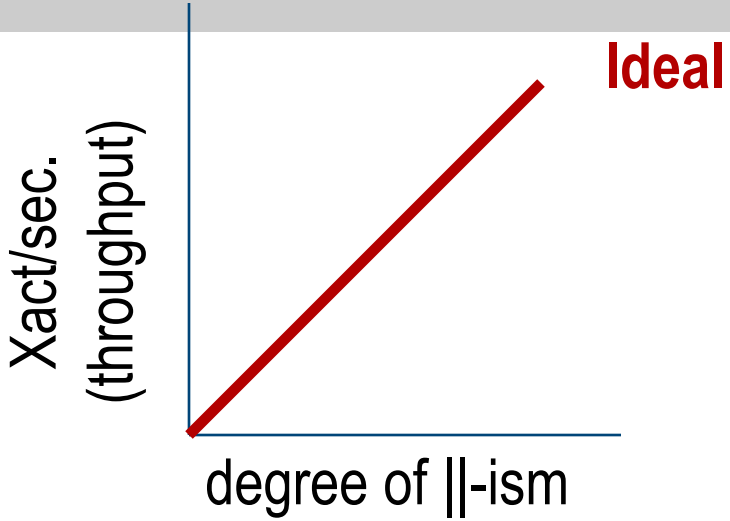


outputs split N ways, inputs merge M ways

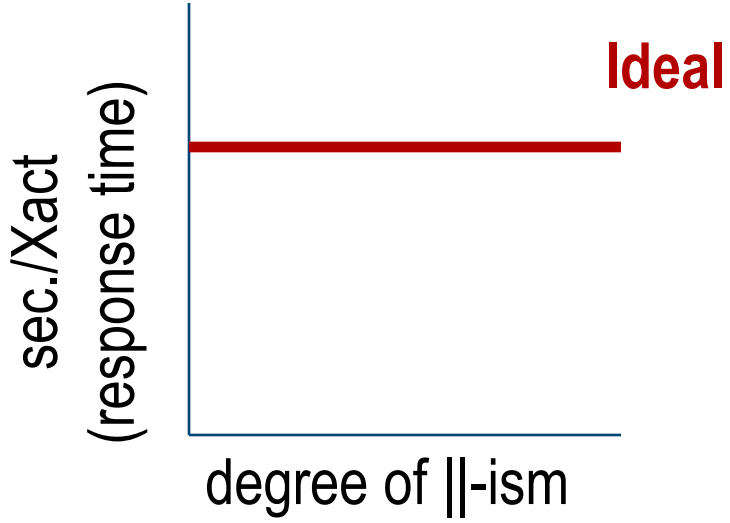
Some || Terminology

JACOBS UNIVERSITY

- Speed-Up
  - More resources means proportionally less time for given amount of data



- Scale (up/out)
  - If resources increased in proportion to increase in data size, time is constant



Scaling

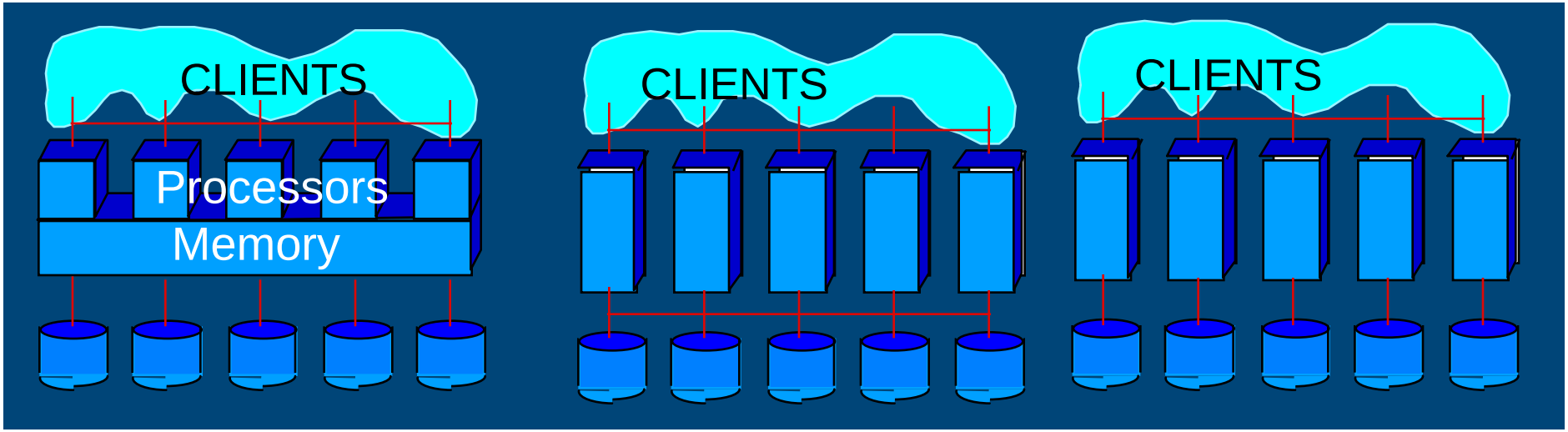
JACOBS UNIVERSITY

- scale horizontally (scale out) = add nodes
  - More iron / more VMs
  - Ex: 1 Web server system →3
  - Cheap hw, parallel algorithms, high-performance interconnects such as Gigabit Ethernet, InfiniBand, Myrinet
  - Increases demand for efficient management & maintenance of multiple nodes
- scale vertically (scale up) = add resources to node
  - CPUs, memory, …
  - use virtualization more effectively (more resources!)
  - Ex: increase number of Apache daemon processes

Architecture Issue: Shared What?

JACOBS UNIVERSITY

Shared Memory (SMP)	Shared Disk	Shared Nothing (network)



Easy to program Expensive to build Difficult to scale up		Hard to program Cheap to build Easy to scale up
Sequent, SGI, Sun, NEC	VMScluster, Sysplex	Tandem, Teradata, SP2

Roadmap

JACOBS UNIVERSITY

- Introduction
- I/O Parallelism
- Interquery Parallelism
- Intraquery Parallelism
- Intraoperation Parallelism
- Interoperation Parallelism
- Query Optimization
- Design of Parallel Systems

I/O Parallelism

JACOBS UNIVERSITY

- Reduce disk access time by partitioning relations on multiple disks
- Horizontal partitioning = tuples of relation divided among many disks such that each tuple resides on one disk
  - Vertical p.: store columns separately
- Several partitioning techniques, will inspect 3 (n disks):
- Round-robin:
  - Send ith tuple inserted in relation to disk (i mod n)
- Hash partitioning:
  - Choose one or more attributes as partitioning attributes, hash function h with range 0…n - 1
  - Let i = result of hash function h applied to partitioning attribute value
  - Send tuple to disk i

I/O Parallelism (Cont.)

JACOBS UNIVERSITY

- Range partitioning:
  - Choose attribute as the partitioning attribute
  - Choose partitioning vector [v_0, v1, ..., v_{n-2}]
  - Let v be the partitioning attribute value of a tuple
    Tuples i with $v_i \leq v_{i+1}$ go to disk i+1
    Tuples i with $v_i < v_0$ go to disk 0
    Tuples i with $v_i \geq v_{n-2}$ go to disk n-1
  - Ex: partitioning vector [5,11]
    - tuple with partitioning attribute value of 2 $\rightarrow$ disk 0
    - tuple with value 8 $\rightarrow$ disk 1
    - tuple with value 20 $\rightarrow$ disk 2

Comparison of Partitioning Techniques

JACOBS UNIVERSITY

- Let's evaluate how well partitioning techniques support the following types of data access:
- Scanning the entire relation
- Locating a tuple associatively – point queries
  - Ex: r.A = 25
- Locating all tuples such that the value of a given attribute lies within a specified range – range queries
  - Ex: $10 \leq r.A < 25$

Comparison: Round Robin

JACOBS UNIVERSITY

- Advantages
  - Best suited for sequential scan of entire relation on each query
  - All disks have almost equal number of tuples;
    retrieval work thus well balanced between disks
- Range queries difficult to process
  - No clustering → tuples are scattered across all disks

Comparison of Hash Partitioning

- Good for sequential access
  - Assuming hash function is good, and partitioning attributes form a key, tuples will be equally distributed between disks
  - Retrieval work is then well balanced between disks
- Good for point queries on partitioning attribute
  - Can lookup single disk, leaving others available for answering other queries
  - Index on partitioning attribute can be local to disk, making lookup and update more efficient
- No clustering, so difficult to answer range queries

Comparison: Range Partitioning

JACOBS UNIVERSITY

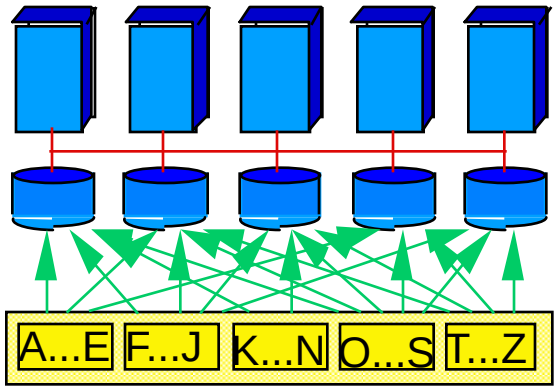
- Provides data clustering by partitioning attribute value,hence:
- Good for sequential access
- Good for point queries on partitioning attribute: only one disk accessed
- For range queries on partitioning attribute, one to a few disks may need to be accessed
  - Remaining disks are available for other queries
  - Good if result tuples are from one to a few blocks
  - If many blocks are to be fetched, they are still fetched from one to a few disks
    → potential disk parallelism wasted
    - *Example of execution skew*

Summary: Data Partitioning

JACOBS UNIVERSITY

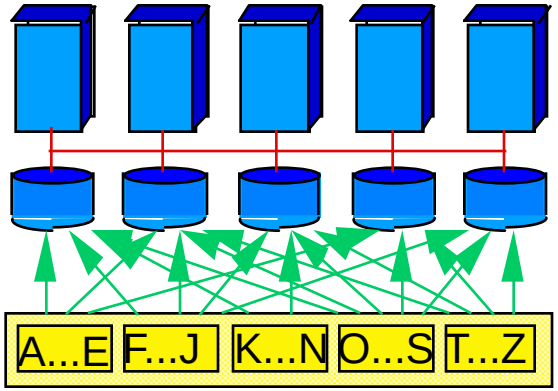
Partitioning a table:

Round Robin



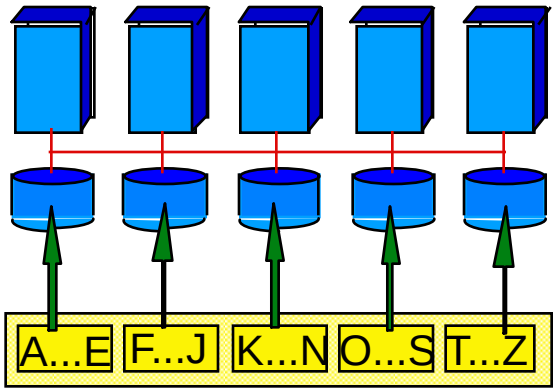
Good to spread load

Hash



Good for equijoins

Range



Good for equijoins,
range queries, group-by

Shared disk and memory less sensitive to partitioning,
Shared nothing benefits from "good" partitioning

Thumb Rules: Partitioning across Disks

JACOBS UNIVERSITY

- relation contains only few tuples, fitting into single disk block
  → assign relation to a single disk
- large relation
  → partition across all available disks
- relation of m disk blocks, n disks available
  → allocate min(m,n) disks

Skew

JACOBS UNIVERSITY

- distribution of tuples to disks may be skewed
  = some disks have many tuples, while others may have fewer tuples
- 2 Types of skew: attribute-value, partition
- Attribute-value skew
  - Some values appear in the partitioning attributes of many tuples;
    all tuples with same value for partitioning attribute end up in the same partition
  - Can occur with range-partitioning and hash-partitioning
- Partition skew
  - With range-partitioning, badly chosen partition vector may assign too many tuples to some partitions and too few to others
  - Less likely with hash-partitioning if a good hash-function is chosen

Handling Skew in Range-Partitioning

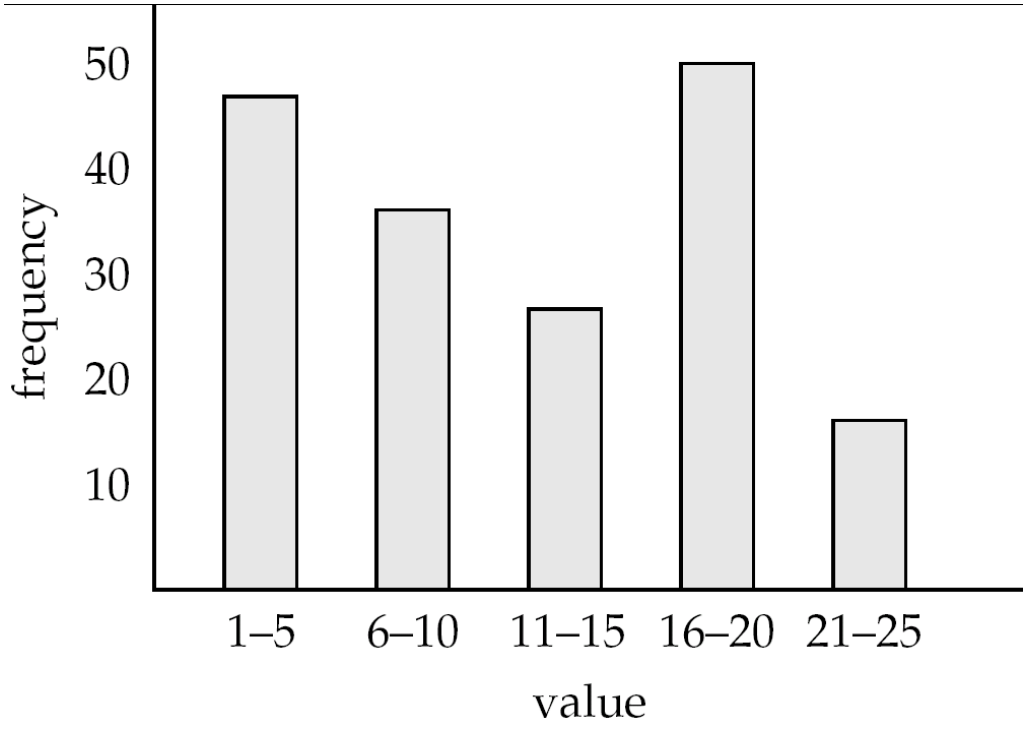
JACOBS UNIVERSITY

- To create a balanced partitioning vector (assuming partitioning attribute forms a key of the relation):
  - Sort relation on partitioning attribute
  - Let n: number of partitions to be constructed.
    Construct partition vector by scanning relation in sorted order as follows:
    After every 1/nth of relation has been read, value of partitioning attribute of next tuple is added to the partition vector
- Duplicate entries or imbalances can result if duplicates are present in partitioning attributes!
- Alternative technique based on histograms used in practice

Handling Skew using Histograms

JACOBS UNIVERSITY

- Balanced partitioning vector can be constructed from histogram in a relatively straightforward fashion
  - Assume uniform distribution within each range of the histogram
- Histogram can be constructed by
  - scanning complete relation
  - or sampling (blocks containing) tuples



Skew: Virtual Processor Partitioning

JACOBS UNIVERSITY

- Skew in range partitioning can be handled elegantly using virtual processor partitioning:
  - create large number of partitions (say 10 to 20 times the number of processors)
  - Assign virtual processors to partitions either in round-robin fashion or based on estimated cost of processing each virtual partition
- Basic idea:
  - If any normal partition would have been skewed, it is very likely the skew is spread over a number of virtual partitions
  - Skewed virtual partitions get spread across a number of processors, so work gets distributed evenly!

Roadmap

JACOBS UNIVERSITY

- Introduction
- I/O Parallelism
- Interquery Parallelism
- Intraquery Parallelism
- Intraoperation Parallelism
- Interoperation Parallelism
- Query Optimization
- Design of Parallel Systems

Interquery Parallelism

JACOBS UNIVERSITY

- Interquery Parallelism = Queries/TAs execute in parallel with one another
- Increases transaction throughput
  - used primarily to scale up transaction processing system to larger number of transactions per second
- Easiest form of parallelism to support particularly in shared-memory || db
  - because even sequential database systems support concurrent processing
- More complicated to implement on shared-disk or shared-nothing architectures
  - Locking & logging must be coordinated by passing messages between processors
  - Data in a local buffer may have been updated at another processor
  - Cache-coherence! — reads & writes of data in buffer must find latest version of data

Cache Coherency Protocol

JACOBS UNIVERSITY

- Example of a cache coherency protocol for shared disk systems:
  - Before reading/writing to a page,
    page must be locked in shared/exclusive mode
  - On locking a page, page must be read from disk
  - Before unlocking a page,
    page must be written to disk if it was modified

is this costly?

- More complex protocols with fewer disk reads/writes exist
- Cache coherency protocols similar for shared-nothing systems
  - Each database page assigned a home processor
  - Requests to fetch / write page sent to home processor

Roadmap

JACOBS UNIVERSITY

- Introduction
- I/O Parallelism
- Interquery Parallelism
- Intraquery Parallelism
- Intraoperation Parallelism
- Interoperation Parallelism
- Query Optimization
- Design of Parallel Systems

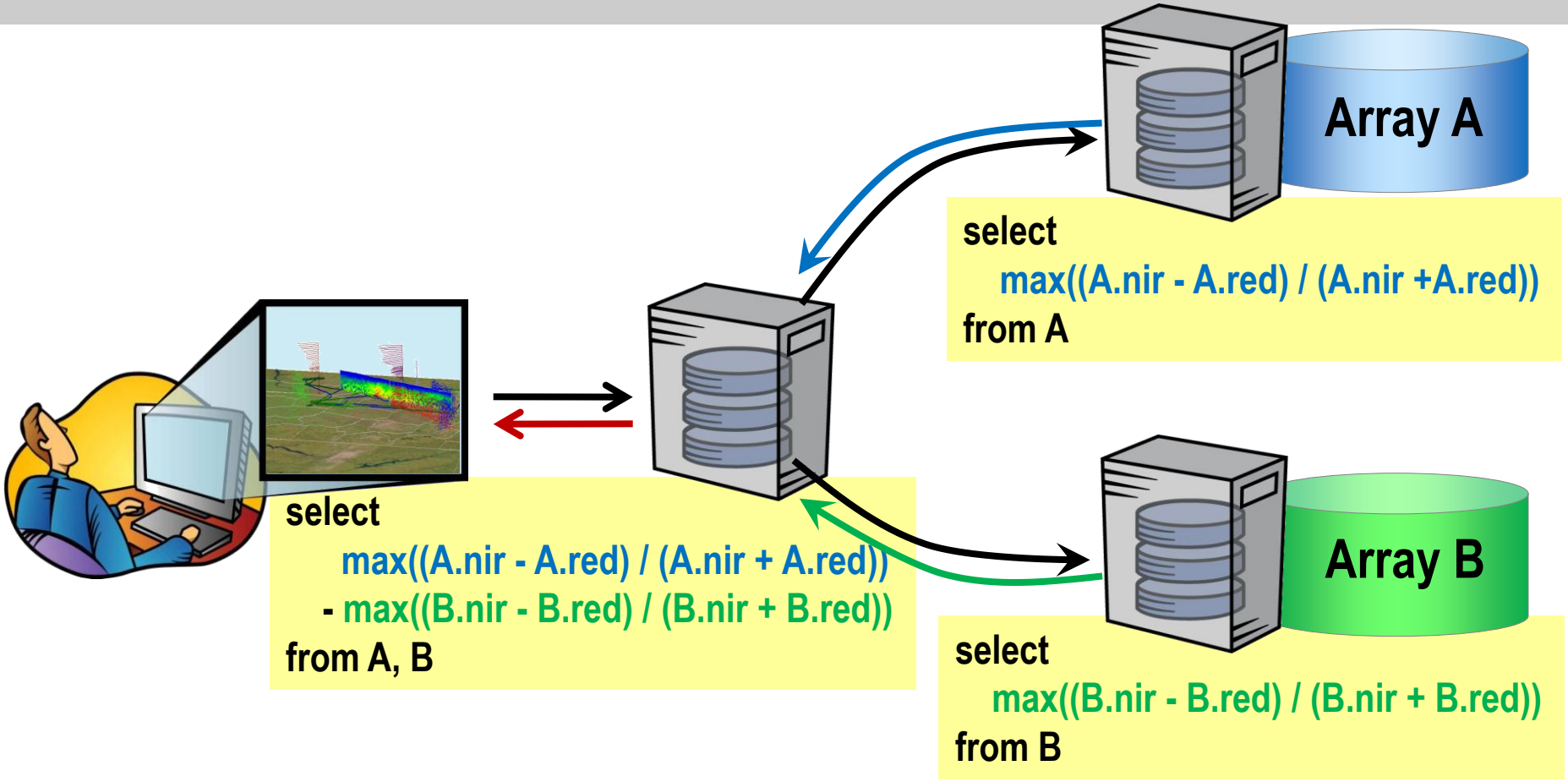
Intraquery Parallelism

JACOBS UNIVERSITY

- Intraquery Parallelism
  = execution of a single query in parallel on multiple processors/disks
  - important for speeding up long-running queries
- Two complementary forms (details see subseq):
  - Intra-operation Parallelism
    – parallelize the execution of each individual operation in the query
  - Inter-operation Parallelism
    – execute the different operations in a query expression in parallel
- first form scales better with increasing parallelism
  - # tuples processed by operation $>_{typically}$ # operations in a query

Inset: || QP in Array Databases

JACOBS UNIVERSITY



Roadmap

JACOBS UNIVERSITY

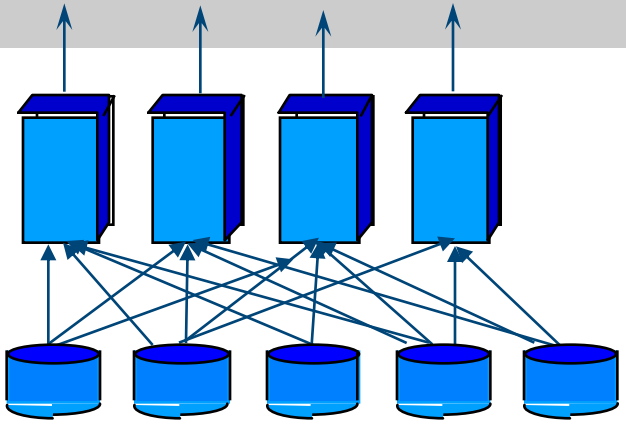
- Introduction
- I/O Parallelism
- Interquery Parallelism
- Intraquery Parallelism
- Intraoperation Parallelism
- Interoperation Parallelism
- Query Optimization
- Design of Parallel Systems

Parallel Processing of Rel. Operations

JACOBS UNIVERSITY

- discussion assumes:
  - read-only queries
  - shared-nothing architecture
  - n processors, P_0, ..., P_{n-1}, and n disks D_0, ..., D_{n-1}, where disk D_i is associated with processor P_i
- If a processor has multiple disks they can simply simulate a single disk D_i
- Shared-nothing architectures can be efficiently simulated on shared-memory and shared-disk systems
  - Algorithms for shared-nothing systems can thus be run on shared-memory and shared-disk systems
  - However, some optimizations may be possible

|| Sort: Range-Partitioning



- Choose partitioning vector
- Scan in parallel, and range-partition as you go (m entries)
- Each processor: sort locally
  - All execute same operation in parallel, without any interaction (= data parallelism)
- Final merge operation (trivial)
  - range-partitioning ensures global sortedness
- Problem: skew
  - Solution: “sample” data upfront for “good” partition points

|| Join

JACOBS UNIVERSITY

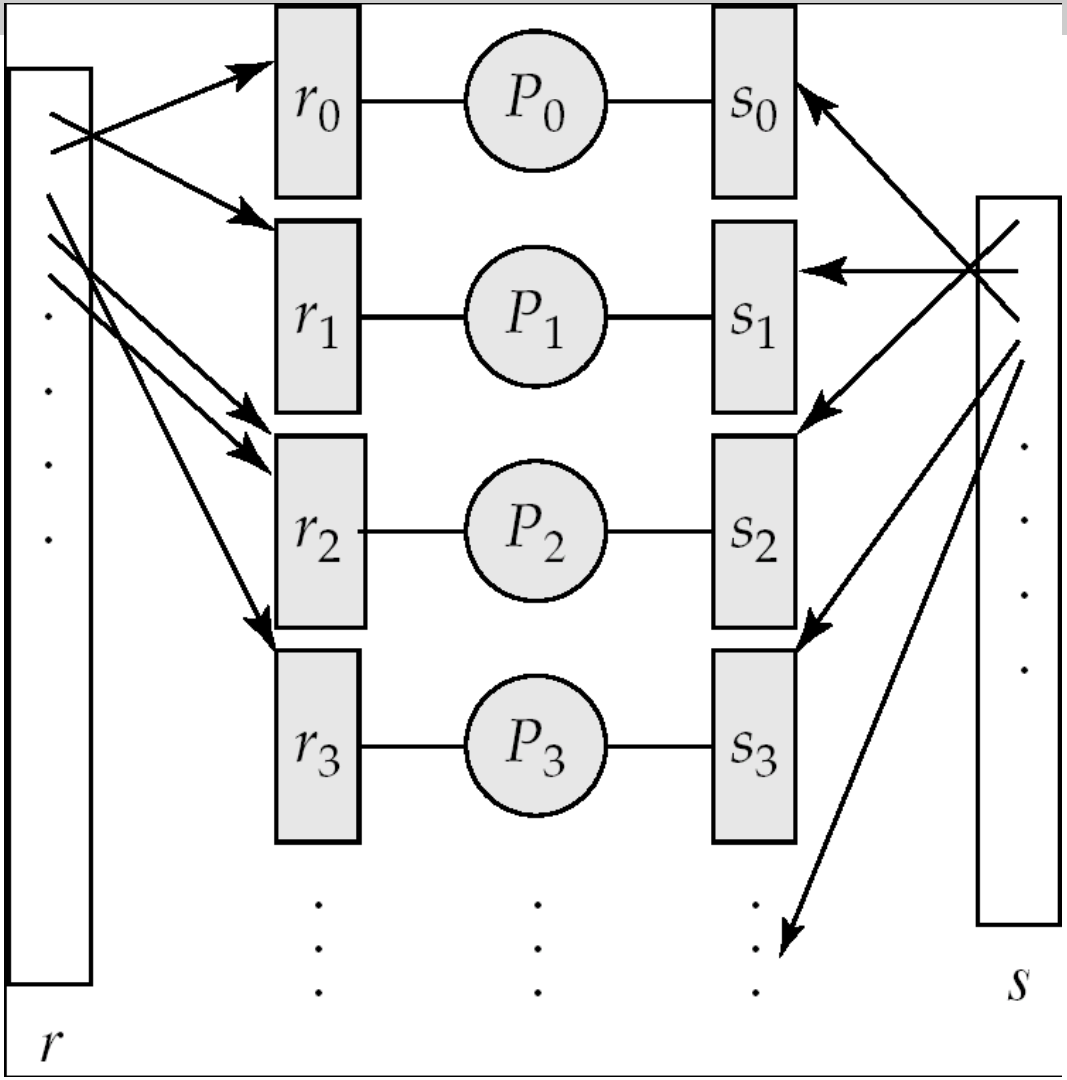
- join operation requires pairs of tuples to be inspected
  - test if they satisfy join condition
  - if they do, pair is added to join output
- Idea: split pairs over several processors
  - Each processor then computes part of the join locally
  - Finally, collect results from each processor $\rightarrow$ final result

Partitioned Join

JACOBS UNIVERSITY

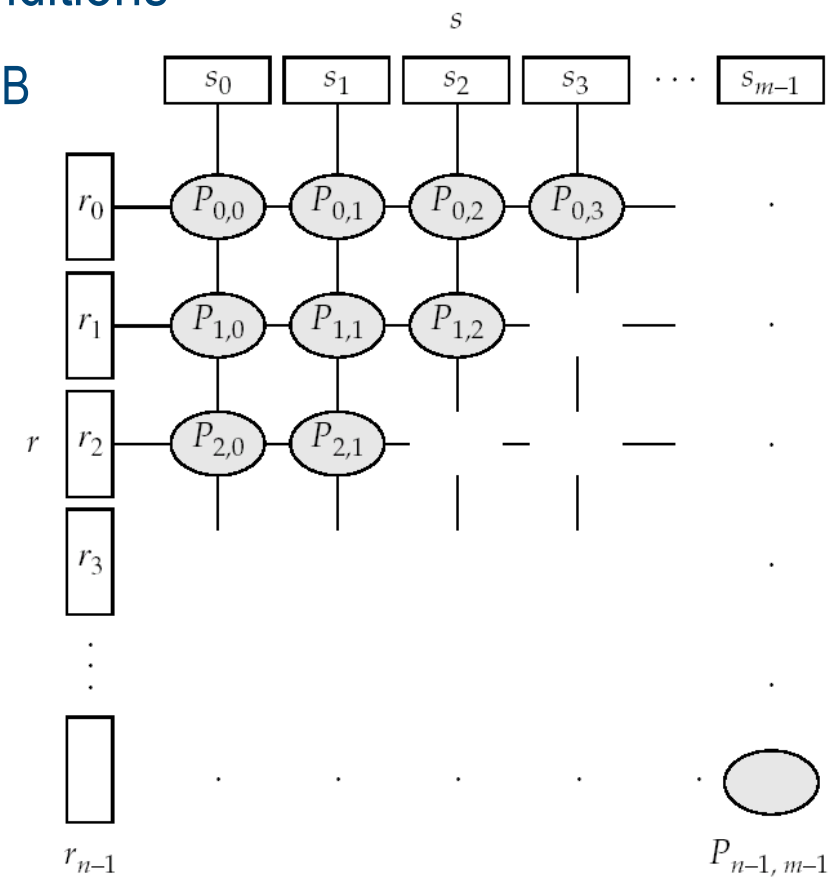
- For equi-joins and natural joins: possible to partition input relations across processors, and compute join locally at each processor
- Let R and S be the input relations; want to compute $R \bowtie_{R.A=S.B} S$
- R, S each partitioned into n partitions, denoted $R_0,...,R_{n-1}$ and $S_0,...,S_{n-1}$
  - range or hash partitioning
- R, S must be partitioned on their join attributes R.A and S.B
  - using same range-partitioning vector or hash function
- Partitions R_i and S_i sent to processor P_i,
- Each processor P_i locally computes $R_i \bowtie_{Ri.A=Si.B} S_i$
  - Any standard join method

Partitioned Join (Cont.)



Fragment-and-Replicate Join

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

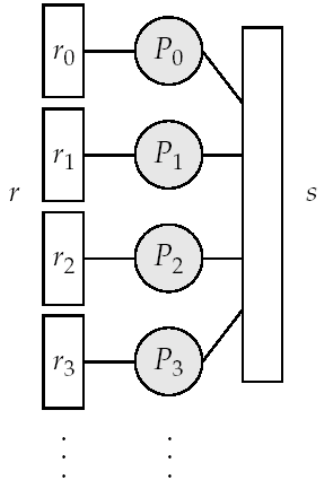


(b) Fragment and replicate

Fragment-and-Replicate Join

JACOBS UNIVERSITY

- Partitioning not possible for some join conditions
  - Ex: non-equijoin conditions, such as R.A > S.B
- fragment and replicate technique
- Special case: asymmetric fragment-and-replicate
  - R partitioned; any partitioning technique can be used
  - S replicated across all processors
  - Processor P_i locally computes join of R_i with all of S (any join technique)



(a) Asymmetric fragment and replicate

Parallel Nested-Loop Join

JACOBS UNIVERSITY

- *Assume that*
  - *|S| << |R| and R stored by partitioning*
  - *index on a join attribute of relation R at each of the partitions of relation R*
- Use asymmetric fragment-and-replicate
  - with relation S being replicated, and using existing partitioning of relation R
- Each processor P_j holding an S partition reads tuples of S stored in D_j, replicates tuples to every other P_i
  - At the end of this phase, relation S is replicated at all sites that store tuples of relation R
- Each processor P_i performs an indexed nested-loop join of S with ith partition of relation R

Other Relational Operations

JACOBS UNIVERSITY

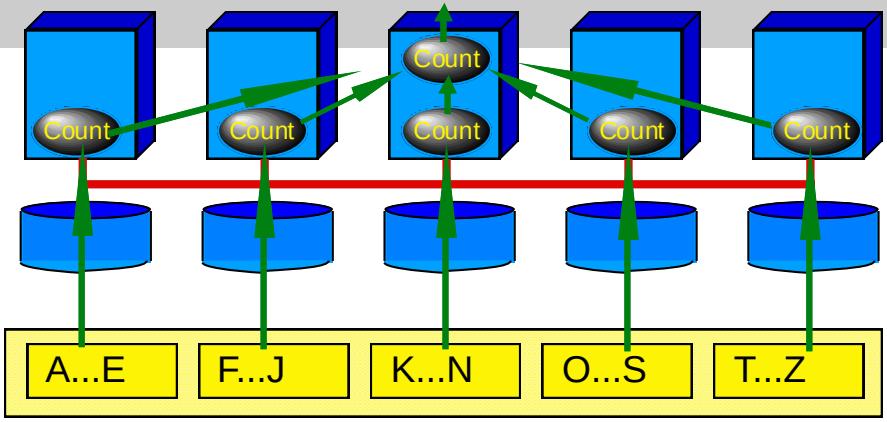
- Selection $\sigma_\theta(r)$
  - If θ is $a_i = v$, where a_i is an attribute and v a value:
    If R is partitioned on a_i then the selection is performed at a single processor
  - If θ is $L <= a_i <= U$ (i.e., θ is a range selection), relation range-partitioned on a_i:
    Selection performed at each processor whose partition overlaps with specified range of values
  - In all other cases:
    selection performed in parallel at all processors
- Projection
  - without duplicate elimination: possible in parallel while tuples read from disk
  - If required, use any duplicate elimination technique

Other Relational Operations (Cont.)

JACOBS UNIVERSITY

- Duplicate elimination
  - Perform by using either of the parallel sort techniques; eliminate duplicates as soon as they are found during sorting
  - Can also partition tuples + perform duplicate elimination locally at each processor

Aggregation



- For each aggregate function, establish decomposition:
  - **count**(S) = Σ **count**(s(i)) -- ditto for **sum**()
  - **avg**(S) = (Σ **sum**(s(i))) / Σ **count**(s(i))
  - and so on...
- For groups:
  - Sub-aggregate groups close to the source
  - Pass each sub-aggregate to its group's site
    - *Chosen via hash function*

Cost of Parallel Evaluation of Operations

- no skew in partitioning, and no overhead due to parallel evaluation: expected speed-up will be 1/n
- skew & overheads taken into account, time taken by a parallel operation can be estimated as

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

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)

Roadmap

JACOBS UNIVERSITY

- Introduction
- I/O Parallelism
- Interquery Parallelism
- Intraquery Parallelism
- Intraoperation Parallelism
- Interoperation Parallelism
- Query Optimization
- Design of Parallel Systems

Interoperator ||: Pipelining

JACOBS UNIVERSITY

- Here: Pipelined parallelism
- Consider a join of four relations: $r1 \bowtie r2 \bowtie r3 \bowtie r4$
- Set up a pipeline that computes the three joins in parallel
  - Let P1 be assigned the computation of temp1 = $r1 \bowtie r2$
  - P2 be assigned the computation of temp2 = $temp1 \bowtie r3$
  - P3 be assigned the computation of $temp2 \bowtie r4$
- Each of these operations can execute in parallel, sending result tuples it computes to the next operation even as it is computing further results
  - Needs pipelineable join evaluation algorithm
    - *e.g. indexed nested loop join*

Limiting Factors for Pipeline ||

JACOBS UNIVERSITY

- Pipeline || avoids writing intermediate results to disk
- Useful with small number of processors,
  does not scale up well with more processors
  - One reason: pipeline chains do not attain sufficient length
- Cannot pipeline blocking operators
  - I.e., which do not produce output until all inputs have been accessed
  - (e.g. aggregate and sort)
- Little speedup obtained for the frequent cases of skew in which one operator's execution cost is much higher than the others

Interoperator ||: Independent P.

JACOBS UNIVERSITY

- Now: Independent parallelism
- Consider join of four relations: $r1 \bowtie r2 \bowtie r3 \bowtie r4$
  - Let P1 be assigned the computation of temp1 = $r1 \bowtie r2$,
    P2 be assigned the computation of temp2 = $r3 \bowtie r4$,
    P3 be assigned the computation of $temp1 \bowtie temp2$
  - P1 and P2 can work independently in parallel
  - P3 has to wait for input from P1 and P2
  - Can pipeline output of P1 and P2 to P3, combining independent parallelism and pipelined parallelism
- Does not provide a high degree of parallelism
  - useful with a lower degree of parallelism; less useful in a highly parallel system

Roadmap

JACOBS UNIVERSITY

- Introduction
- I/O Parallelism
- Interquery Parallelism
- Intraquery Parallelism
- Intraoperation Parallelism
- Interoperation Parallelism
- Query Optimization
- Design of Parallel Systems

Query Optimization

JACOBS UNIVERSITY

- Query optimization in || databases significantly more complex than in sequential databases; *open research!*
- Cost models more complicated
  - partitioning costs; skew and resource contention; etc.
- When scheduling execution tree in parallel system, must decide:
  - How to parallelize each operation, how many processors to use for it?
  - What operations to pipeline? what operations to execute independently in parallel? what operations to execute sequentially? …one after the other
- Difficult: Determining resources needed for each operation
  - E.g., allocating more processors than optimal can result in high communication overhead
- Avoid long pipelines
  - final operation may wait a lot for inputs, while holding precious resources

Query Optimization (Cont.)

JACOBS UNIVERSITY

- | parallel evaluation plans | >> | sequential evaluation plans |
  - heuristics needed for choosing parallel plans
- Heuristic I: just parallelize every operation across all processors
  - No pipelining & inter-operation pipelining
  - Finding best plan much easier: use standard optimization technique, but with new cost model
  - Ex: Volcano parallel database, popularized exchange-operator model
    - *exchange operator introduced into query plans to partition and distribute tuples*
    - *each operation works independently on local data on each processor, in parallel with other copies of the operation*
- Heuristic II: First choose most efficient sequential plan and then choose how to best parallelize operations in that plan
  - Can explore pipelined parallelism as an option
- Choosing good physical organization (partitioning technique) important

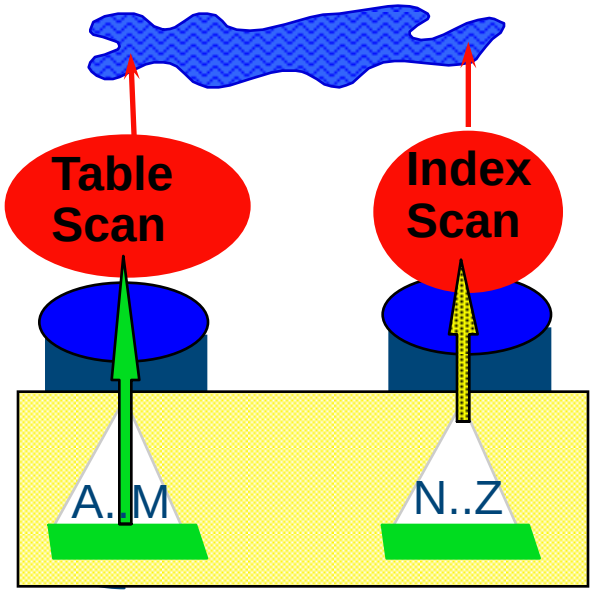
What's Wrong With That?

JACOBS UNIVERSITY

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

```
SELECT *
FROM telephone_book
WHERE name < "NoGood"
```

  - Table partitioned
    with local secondary 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 secondary index



Roadmap

JACOBS UNIVERSITY

- Introduction
- I/O Parallelism
- Interquery Parallelism
- Intraquery Parallelism
- Intraoperation Parallelism
- Interoperation Parallelism
- Query Optimization
- Design of Parallel Systems

Design of Parallel Systems

JACOBS UNIVERSITY

Some issues in the design of parallel systems:

- Parallel loading of data from external sources needed
  - handle large volumes of incoming data
  - Index updates?
- Resilience to failure of processors or disks
  - Probability higher in parallel system
  - Availability: Operation (perhaps degraded) in spite of isolated failures
  - Redundancy: extra copy of every (?) data item at another processor

Design of Parallel Systems (Cont.)

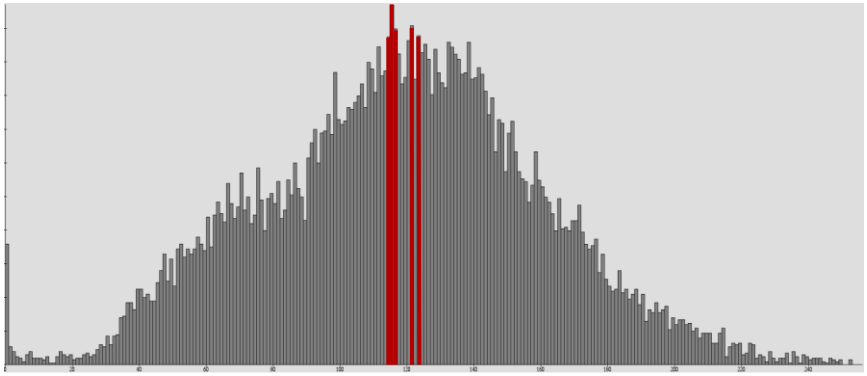
JACOBS UNIVERSITY

- On-line **re**organization of data and schema changes
  - Ex: index construction on TB database can take hours/days even on parallel system
    - *Need to allow other processing (insertions/deletions/updates) to be performed on relation even as index is being constructed*
  - Basic idea: index construction tracks changes and ``catches up'' on changes at end
- On-line **re**partitioning and schema changes
  - concurrently with other processing

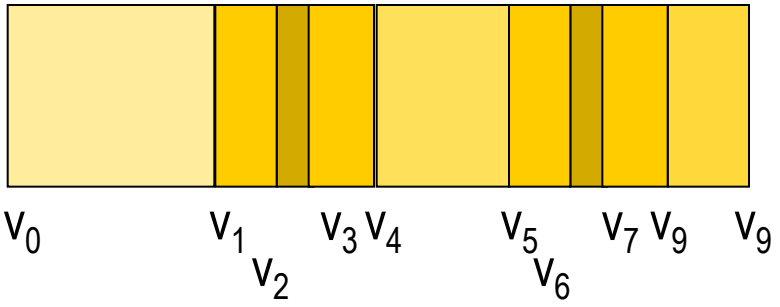
Histograms in Practice

JACOBS UNIVERSITY

- Usually two 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

JACOBS UNIVERSITY

- 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

JACOBS UNIVERSITY

- 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

JACOBS UNIVERSITY

- 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

JACOBS UNIVERSITY

- 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