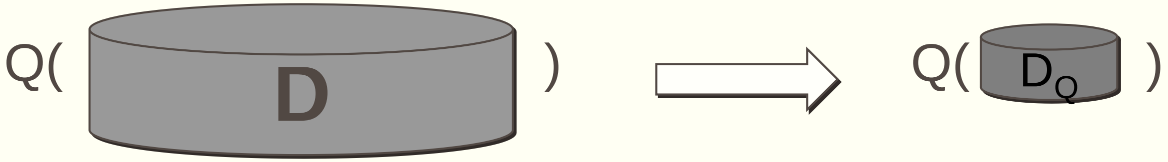
Distributed Query Processing

How to Make Big Data Small?

- **Input**: A class $\mathcal{Q}$ of queries
- **Question**: Can we effectively find, given queries $Q \in \mathcal{Q}$ and any (possibly big) data D, a small D_Q such that $Q(D) = Q(D_Q)$?



- Data synopsis
- Boundedly evaluable queries
- Query answering using views
- Incremental evaluation
- Distributed Query Processing

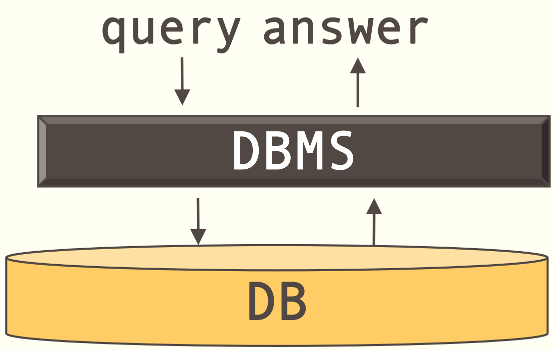
Parallel DBMS

- Why parallel DBMS?
- Architectures
- Parallelism
  - Intra-query Parallelism
  - Inter-query Parallelism
  - Intra-operation Parallelism
  - Inter-operation Parallelism

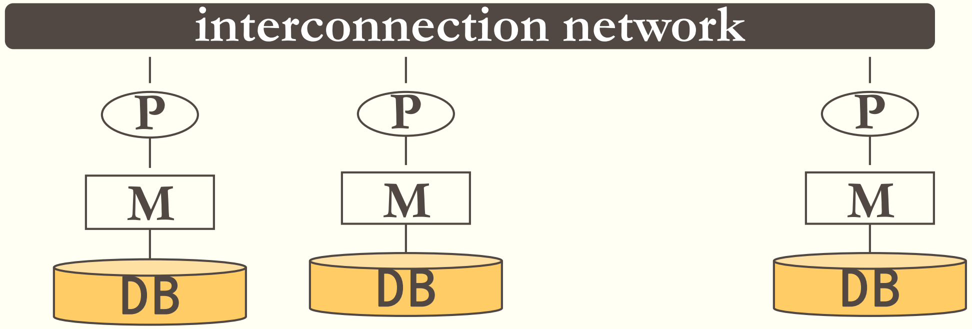
Performance of a DBMS

- Throughput:
  - The number of tasks finished in a given time interval
- Response Time (Latency):
  - The amount of time to finish a single task from the time it is submitted
- Can we do better given more resources (CPU, disk, memory, …)?
- Parallel DBMS: exploring parallelism
  - Divide a big problem into many smaller ones to be solved in parallel

Traditional DBMS

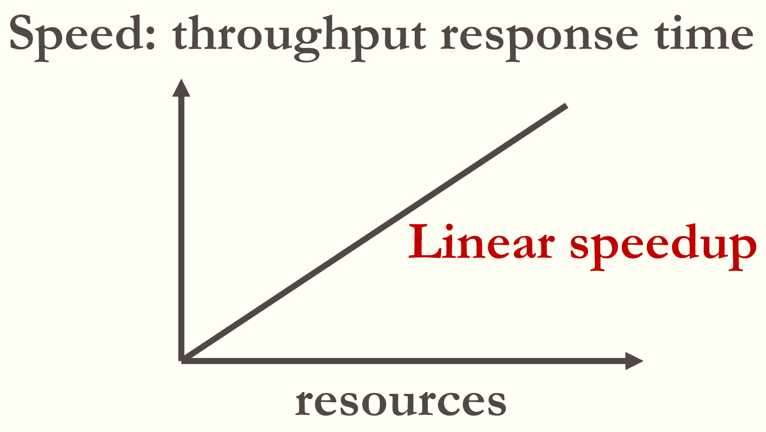


Parallel DBMS



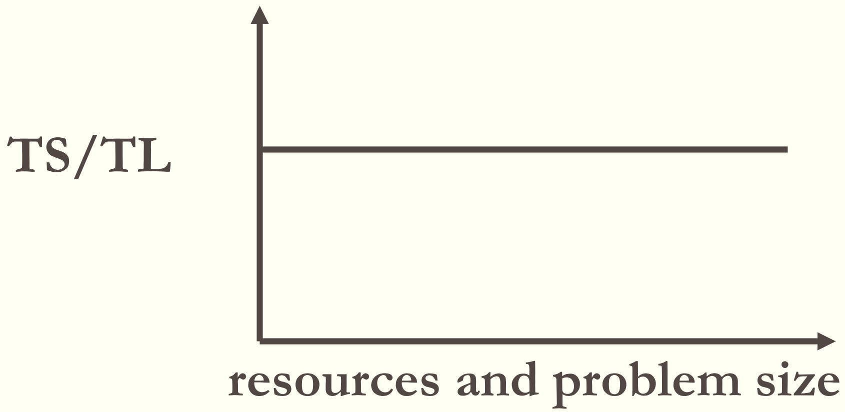
Degree of Parallelism: Speedup

- Speedup: TS/TL, for a given task
  - TS: time taken by a traditional DBMS
  - TL: time taken by a parallel DBMS with more resources
  - TS/TL: Ideally, more resources mean proportionally less time for a task
- Linear speedup:
  - The speedup is N while the parallel system has N times resources of the traditional system
  - Can we do better?



Degree of Parallelism: Scaleup

- Scaleup: TS/TL
  - Factor that expresses how much more work can be done in the same time period by a larger system
  - A task Q and a task Q_N, which is N times bigger than Q
  - A DBMS M_S and a parallel DBMS M_L, N times more resources
  - TS: time taken by a traditional DBMS
  - TL: time taken by a parallel DBMS with more resources
  - Linear scaleup if $\frac{TS}{TL} = 1$
    - The time is constant if the resource increases in proportion to increase in problem size



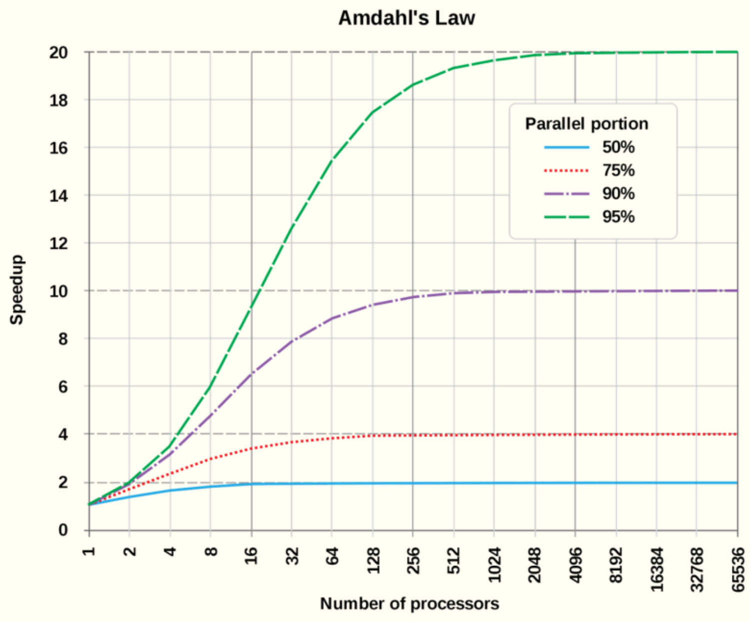
Why can't it be better than linear scaleup/speedup?

- Startup costs: initializing each process
- Interference: competing for shared resources (network, disk, memory or locks)
- Skew:
  - It is difficult to divide a task into exactly equal-sized parts
  - The response time (latency) is determined by the largest part
- Amdahl's law:

$$\text{Speedup} = \frac{1}{f + \frac{1-f}{s}}$$

f: "sequential fraction" of the program

s: Amount of parallel resources



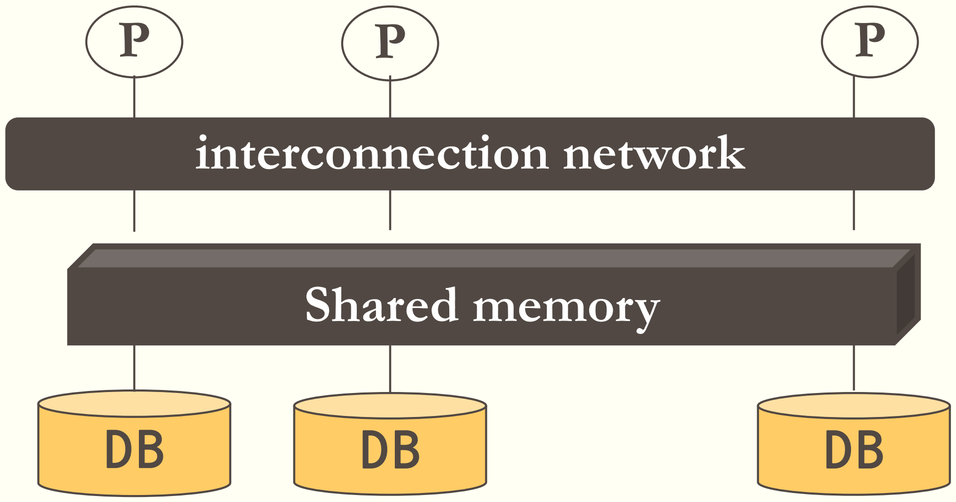
Why Parallel DBMS

- Improve Performance
- Almost died 25 years ago, but with renewed interests
  - Big Data: Data collected from the web
  - Decision Support Queries: Costly on large data
  - Hardware has become much cheaper
- Improve reliability and availability
- MapReduce

Architecture: Shared Memory

- Efficient Communication
  - Via data in memory, accessible by all
- Not Scalable:
  - Shared memory and network become bottleneck – interference
  - Not scalable beyond 32/64 processors
  - Adding memory cache to each processor?
    - Cache coherence problem when data is updated

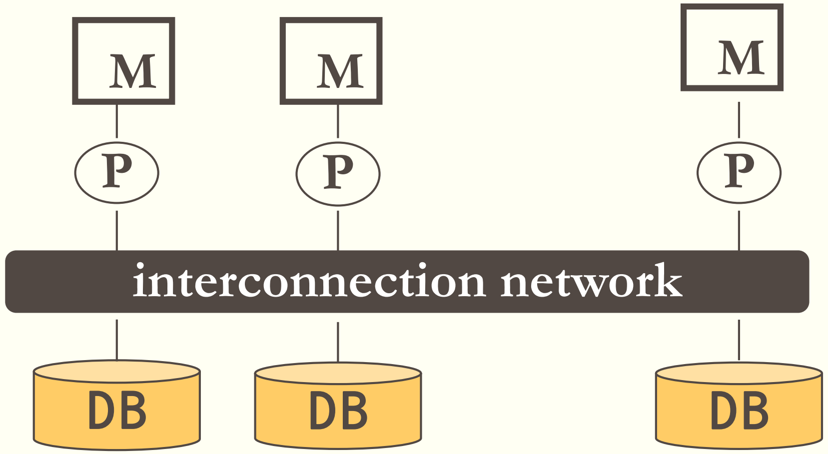
Informix (9 nodes)



Shared Disk

- Fault tolerance:
  - If a processor fails, the other can take over, since the database is resident on disk
- Scalability:
  - Better than shared memory: memory is no longer a bottleneck
  - But disk subsystem is a bottleneck
- Interference:
  - All I/O to go through a single network
  - Not scalable beyond a couple of hundred processors

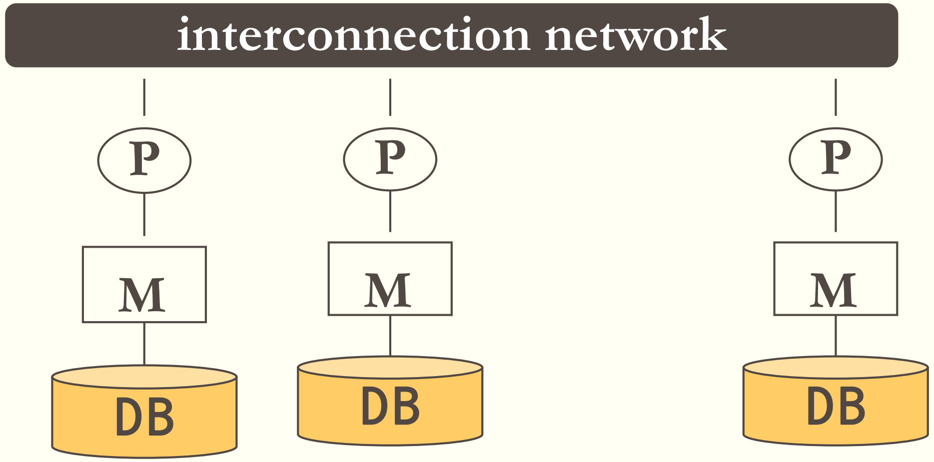
Oracle RDB (170 nodes)



Shared Nothing

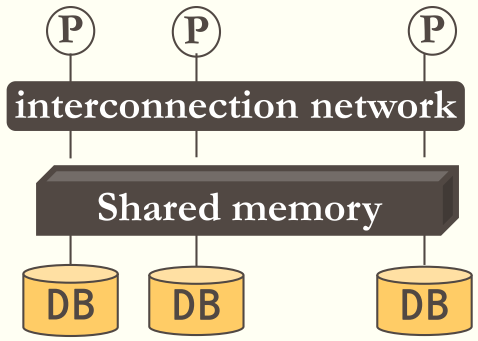
- Scalable:
  - Only queries and result relations pass through the network
- Communication costs and access to non-local disks
  - Sending data involves software interaction at both ends

Teradata: 400 nodes
IBM SP2/DB2:128 nodes
Informix SP2: 48 nodes



Architecture Summary

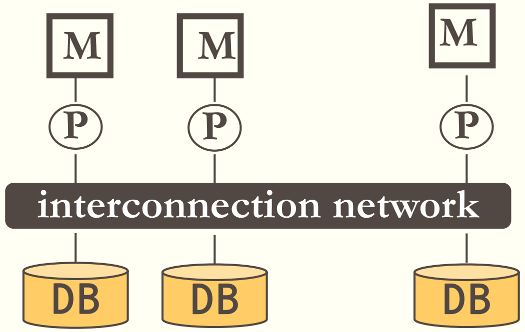
Shared Memory (SMP)



Easy to program
Expensive to build
Difficult to scaleup

Informix, RedBrick
Sequent, SGI, Sun
scale: 9 nodes

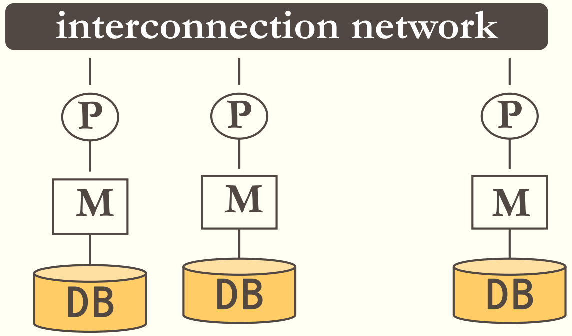
Shared Disk



Better scalability
Fault tolerance

VMScluster,
Oracle (170 nodes)
DEC Rdb (24 nodes)

Shared Nothing (network)



Hard to program
Cheap to build
Easy to scaleup

Teradata:	400 nodes
Tandem:	110 nodes
IBM/SP2/DB2:	128 nodes
Informix/SP2	48 nodes

Architectures of Parallel DBMS

- Tradeoffs
  - Scalability
  - Communication Speed
  - Cache Coherence

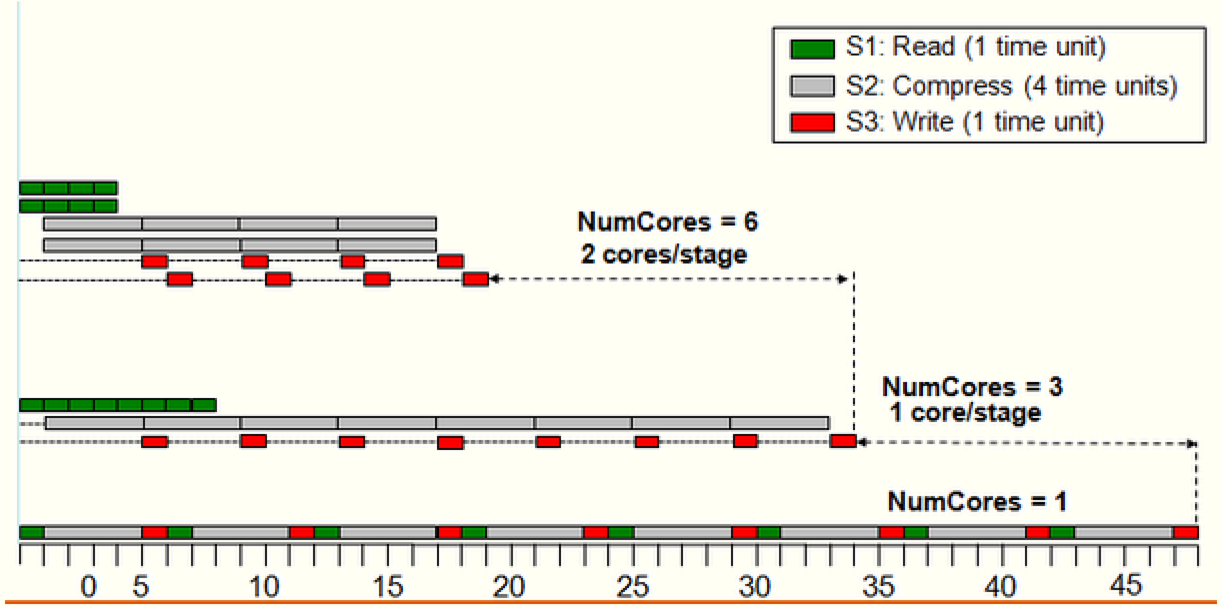
Pipelined

- The output of operation A is consumed by another operation B, before A has produced the entire output
  - Many machines, each doing one step in a multi-step process
  - May lead to increase in response time
- Does not scale up well when
  - The computation does not provide sufficiently long chain to provide a high degree of parallelism
  - Relational operators do not produce output until all inputs have been accessed (blocking)
  - A’s computation cost is much higher than that of B

Pipelined Parallelism: Compress a File

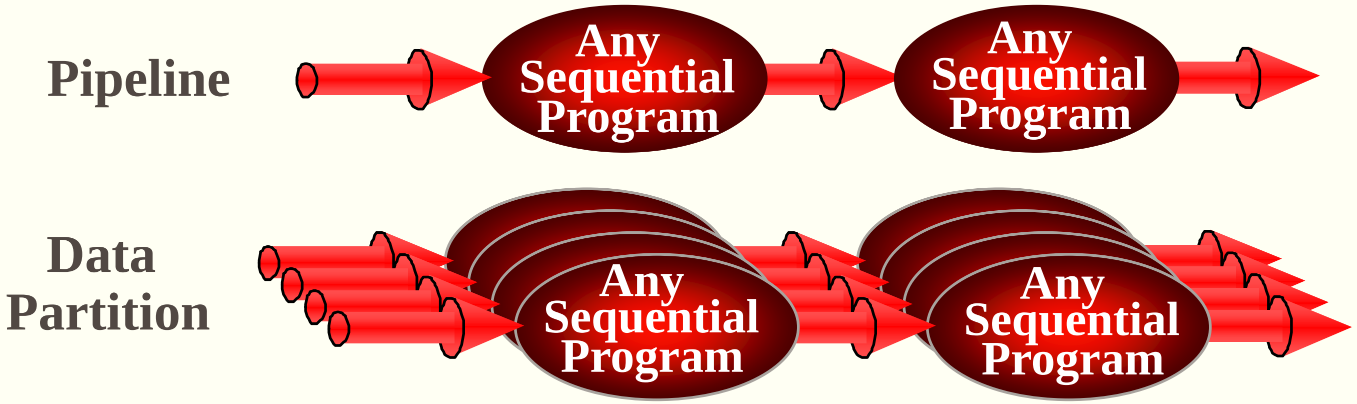
```
while(!done){
    Read block from file;
    Compress the block;
    Write block to file;
}
```

```
while(!program end){
    If work is available in the
    in-Queue of thread
    Compress the block;
    Write block to file;
}
```



Data partitioned Parallelism

- Many machines performing the same operation on different pieces of data (similar to SIMD)



Partitioning in RDMS

- Partition a relation and distribute it to different processors
  - Maximize processing at each individual processor
  - Minimize data shipping
- Query types
  - Scan a relation
  - Point access $r.A = v$
  - Range Queries $v < r.A < v'$

Partitioning Strategy

- Assume N disks, a relation R
- Round Robin
  - Send the j-th tuple of R to disk number $j \ mod \ N$
  - Even distribution: Good for scanning
  - Not good for equal joins (point queries) and range queries (all disk have to be involved for the search)
- Range Partitioning:
  - Partitioning attribute A, vector $[v_1, \ldots, v_{n-1}]$
  - Send tuple t to disk j if $t[A]$ in $[v_{j-1}, v_j]$
  - Good for point and range queries on partitioning attributes (using only a few disks, while leaving the others free)
  - Execution skew: distribution may not be even, and all operations occur in one or few partitions (scanning)

Partitioning Strategies (Cont'd)

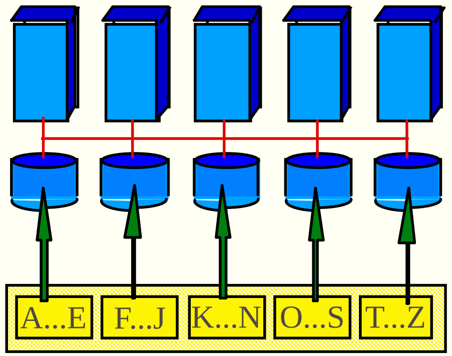
- Assume N disks, a relation R
- Hash Partitioning:
  - Hash function $f(t)$ in the range of integer $[0, N-1]$
  - Send tuple t to disk $f(t)$
  - Good for point queries on partitioning attributes, and sequential scanning if the hash function is even
  - No good for point queries on non-partitioning attributes and range queries

Question: how to partition $R1(A, B)$: $\{(i, i+1)\}$, with 5 processors?

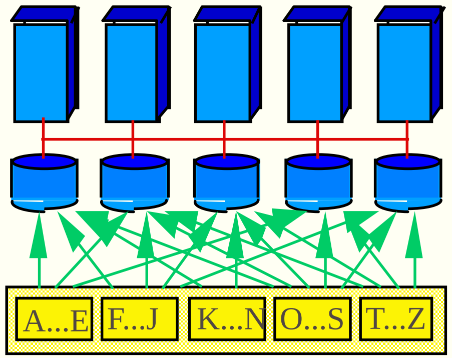
- Round-robin
- Range partitioning: partitioning attribute A
- Hash partitioning

Automatic Data Partitioning

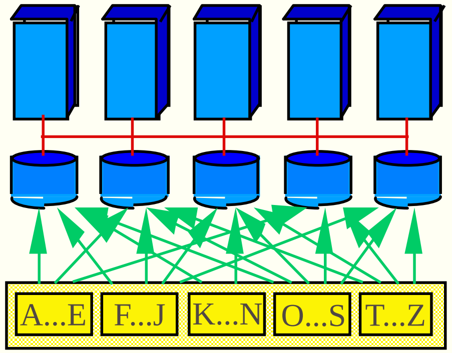
- Partitioning a table



Good for group-by, range queries, and also equip-join



Good for equijoins



Good to spread load;
Most flexible
Not good for equi-join and range

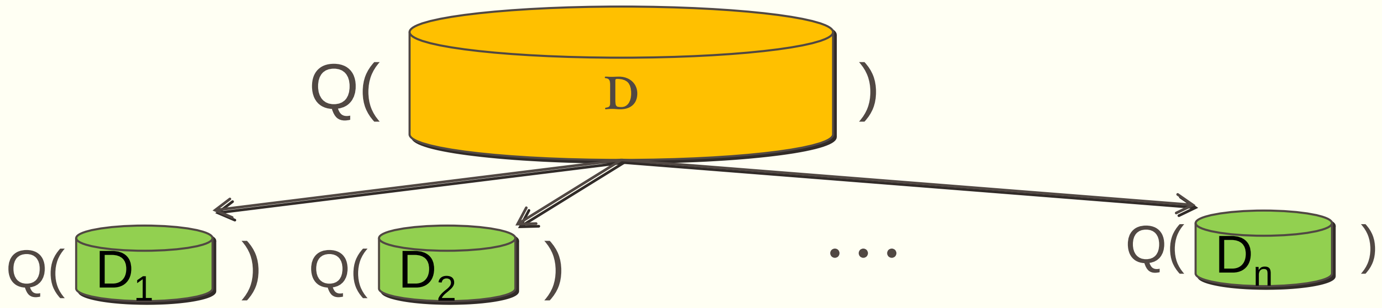
Shared-disk and -memory less sensitive to partitioning,
Shared nothing benefits from "good" partitioning

Inter-query Vs. Intra-query Parallelism

- Inter-query:
  - Different queries or transactions execute in parallel
  - Improve transaction throughput
  - Easy on shared-memory: Traditional DBMS tricks will do
  - Shared-nothing/Disk: Cache coherence problem
    - Ensure that each processor has the latest version of the data in its buffer pool
    - Flush updated pages to shared disk before releasing the lock
- Intra-Query
  - A single query in parallel on multiple processors
  - Speed up single complex long running queries
  - Interoperation: operator tree
  - Intraoperation: parallelize the same operation on different sets of the same relations: Parallel sorting, Parallel join; Selection; Projection; Aggregation

Parallel Query Answering

- Given data D, and n processors $S_1, S_2, \dots, S_n$
  - D is partitioned into fragments $(D_1, D_2, \dots, D_n)$
  - D is distributed to n processors: D_i is stored at S_i
- Each processor S_i processes operations for a query on its local fragment D_i, in parallel



Relational Operators

- Projection $\pi_A R$
- Selection $\sigma_C R$
- Join $R_1 \bowtie_C R_2$
- Union $R_1 \cup R_2$
- Set Difference $R_1 - R_2$
- Group by and Aggregation
    - Max, min, count, average, …

Intra-operation Parallelism: Projection

- Projection $\pi_A R$, where R is partitioned across n processors
  - Read tuples of R at all processors involved, in parallel
  - Conduct projection on tuples
  - Merge local results – to eliminate duplicate elimination (via sorting?)

Intra-Operation Parallelism: Selection

- Selection $\sigma_C R$, where R is partitioned across n processors
- If A is the partitioning attribute
  - Point query $C: r.A = v$

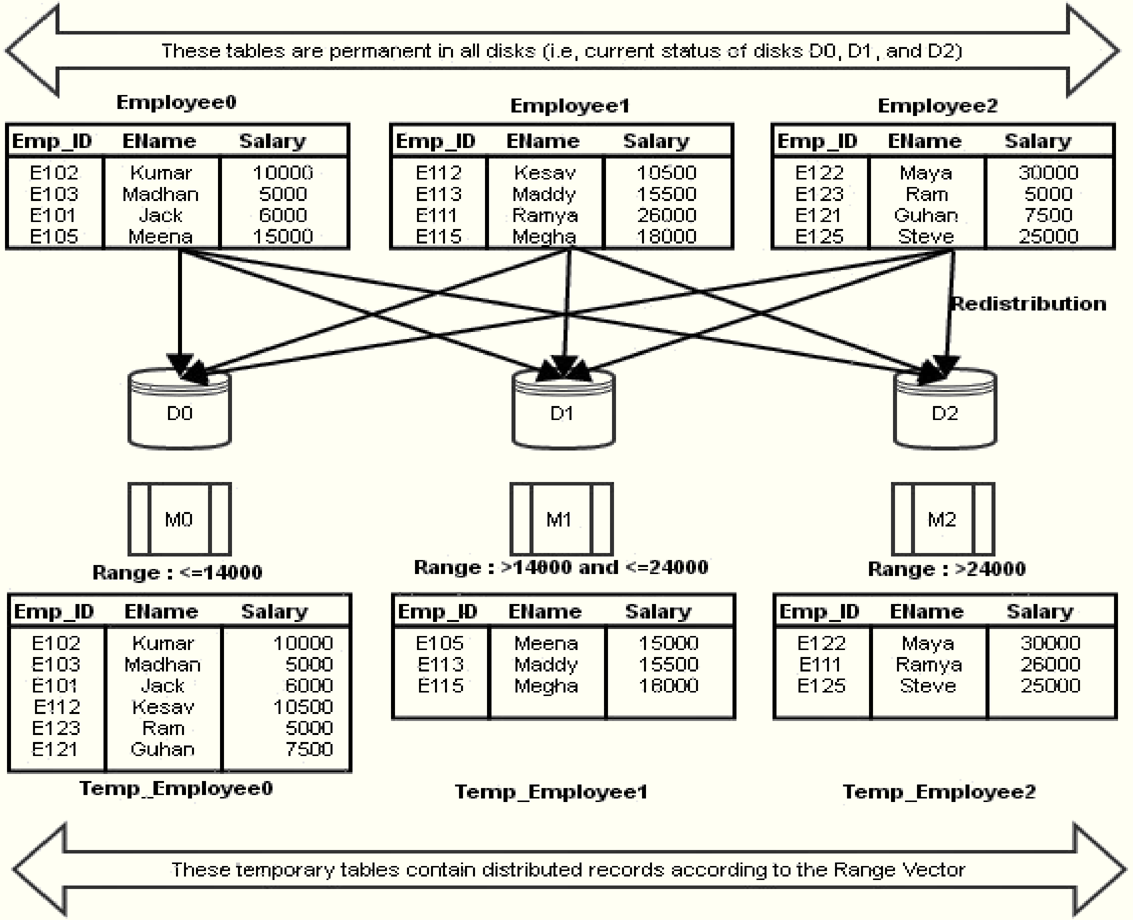
    A single processor that holds $r.A = v$ is involved
  - Range query $C: v \leq r.A \leq v'$

    Only processors whose partition overlaps with the range are involved
- If A is not the partitioning attribute
  - Compute selection at each individual processor
  - Merge local result

Intra-Operation Parallelism: Sort

- Sort R on attribute A, where R is partitioned across n processors
- If A is the partitioning attribute: Range-partitioning
  - Sort each partition
  - Concatenate the result
- If A is not the partitioning attribute: Range-partitioning sort
  - Range partitioning R based on A, redistributed the tuples in R
  - Every processor works in parallel: read tuples and send them to corresponding processors
  - Each processor sorts its new partition locally when the tuple come in
  - Merge local results
- Issue: skew
- Solution: sample the data to determine the partitioning vector

Example: Parallel Sort

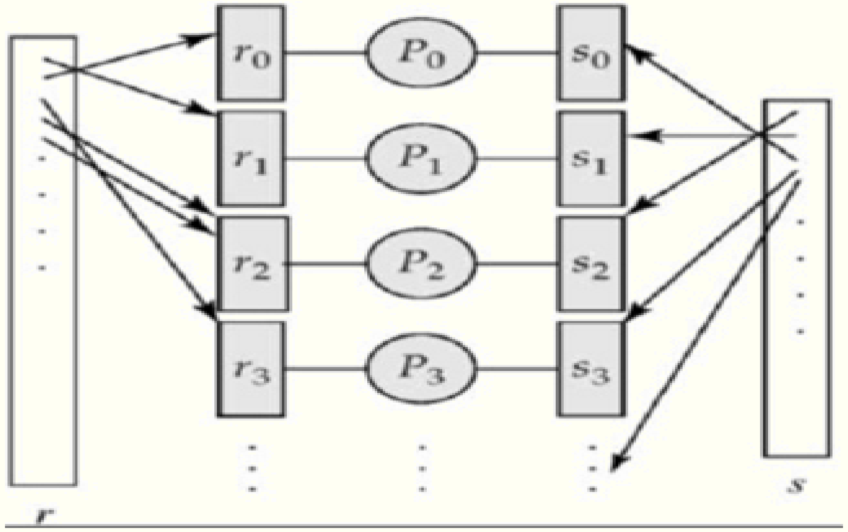


Intra-Operation Parallelism: Join

- Partitioned join: for equi-joins and natural joins
- Fragment-and replication join: inequality
- Partitioned parallel hash-join: equal or natural join
  - where R1, R2 are too large to fit in memory
  - Almost always the winner for equi-joins

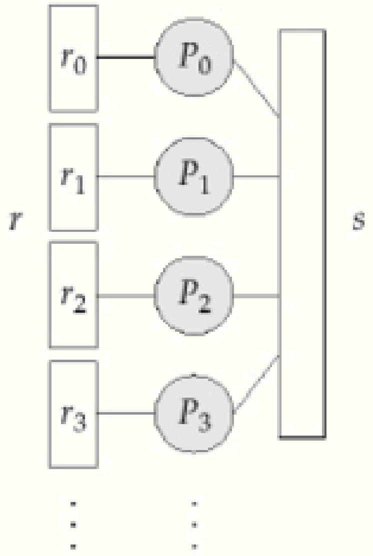
Partitioned Join

- $R_1 \bowtie_{R_1.A=R_2.B} R_2$
  - Partition R_1 and R_2 into n partitions, by the same partitioning function in $R_1.A$ and $R_2.B$, via either
    - range partitioning, or
    - hash partitioning
  - Compute $R_1^i \bowtie_{R_1.A=R_2.B} R_2^i$ locally at processor i
  - Merge the local results
- Question: how to perform partitioned join on the following, with 2 processors?
  - R1(A, B): {(1, 2), (3, 4), (5, 6)}
  - R2(B, C): {(2, 3), {3, 4)}



Fragment and Replicate join

- $R_1 \bowtie_{R_1.A<R_2.B} R_2$
  - Partition R_1 into n partitions, by any partitioning method, and distribute it across n processors
  - Replicate the other relation R_2 across all processors
  - Compute $R_1^j \bowtie_{R_1.A<R_2.B} R_2$ locally at processor j
  - Merge the local results
- Question: how to perform fragment and replicate join on the following, with 2 processors?
  - R1(A, B): {(1, 2), (3, 4), (5, 6)}
  - R2(B, C): {(2, 3), {3, 4)}



Partitioned Parallel Hash Join

- $R_1 \bowtie_{R_1.A=R_2.B} R_2$
  - Hash partitioning R_1 and R_2 using hash function h on partitioning attributes $R_1.A$ and $R_2.B$, respectively
  - For $i \in [1, k]$, process the join of i-th partition $R_1^i \bowtie R_2^i$, with hash join

Intra-Operation Parallelism: Aggregation

- Aggregate on the Attribute B of R, grouping on A
- Decomposition:
  - $count(S) = \sum count(S_i)$; similar for sum
  - $avg(S) = sum(S)/count(S)$
- Strategy 1:
  - Range partitioning R based on A: redistribute the tuples in R
  - Each processor computes sub-aggregate (data parallelism)
  - Merge local results as above
- Strategy 2:
  - Each processor computes sub-aggregate (data parallelism)
  - Range partitioning local results based on A, redistribute partial results
  - Compose the local results

Example: Aggregation

- Describe a good processing strategy to parallelize the following query
- SELECT branch-name, avg(balance)
  FROM account
  GROUP BY branch-name
- The schema for the account is

  account(account-id, branch-name, balance)

- Strategy:
  - Range or hash partition account by using branch-name as the partitioning attribute. This creates table $account_j$ at each site j.
  - At each site j, compute $\frac{sum(account_j)}{count(account_j)}$;
  - output $\frac{sum(account_j)}{count(account_j)}$: the union of these partial results is the final query answer

Inter-Operation Parallelism

- Consider $R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4$
- Pipelined:
  - $Temp1 \leftarrow R_1 \bowtie R_2$
  - $Temp2 \leftarrow R_3 \bowtie Temp1$
  - $Result \leftarrow R_4 \bowtie Temp2$
- Independent
  - $Temp1 \leftarrow R_1 \bowtie R_2$
  - $Temp2 \leftarrow R_3 \bowtie R_4$
  - $Result \leftarrow Temp1 \bowtie Temp2$ (Pipelined Stage)

Cost Model

- Cost model: partitioning, skew, resource contention, scheduling
  - Partitioning: Tpart
  - Cost of assembling local answers: Tasm
  - Skew: max(T0, …, Tn)
  - Estimation: Tpart + Tasm + max(T0, …, Tn)

  May also include startup costs and contention for resources (in each Tj)
- Query optimization: find the “best” parallel query plan
  - Heuristic 1: parallelize all operations across all processors -- partitioning, cost estimation (Teradata)
  - Heuristic 2: best sequential plan, and parallelize operations -- partition, skew, … (Volcano parallel machine)

Practice: Validation of Functional Dependencies

- Develop a parallel algorithm that given a relation D and an Functional dependency FD, computes all the violations
  - Partitioned Join
  - Partitioned and replication Join
- Question: what can we do if we are given a set of FDs to validate?

Practice: Implement Set Difference

- Develop a parallel algorithm that R1 and R2, compute R1-R2, by using:
  - partitioned join
  - partitioned and replicated
- Questions: what can we do if the relations are too large to fit in memory?