Principles of Distributed Database Systems

TS. Phan Thị Hà

Outline

- Introduction
- Distributed and Parallel Database Design
- Distributed Data Control
- Distributed Query Processing
- Distributed Transaction Processing
- Data Replication
- Database Integration – Multidatabase Systems
- Parallel Database Systems
- Peer-to-Peer Data Management
- Big Data Processing
- NoSQL, NewSQL and Polystores
- Web Data Management

Outline

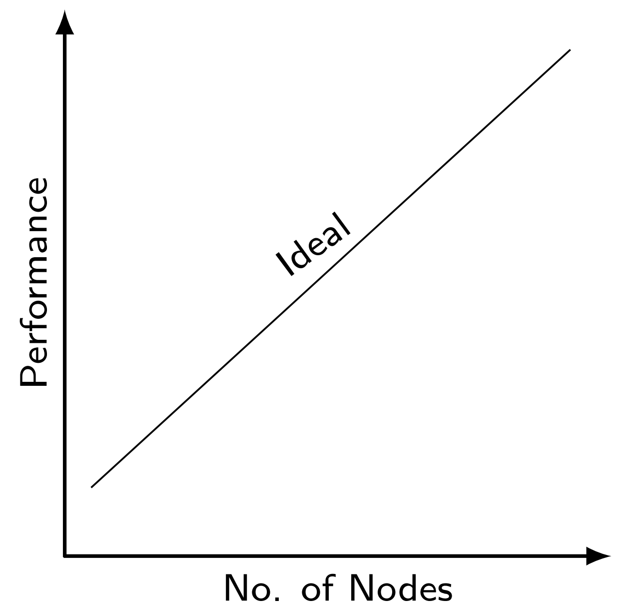
- Parallel Database Systems
  - Parallel Architectures
  - Data Placement
  - Query Processing
  - Load Balancing and Fault-tolerance
  - Database Clusters

Objectives of Parallel Systems

- High-performance using parallelism
  - High throughput for OLTP loads
    - Lots of short transactions, which update a few data
  - Low response time for OLAP queries
  - Large queries, which read lots of data
- High availability and reliability through data replication and failover
- Extensibility and scalability by adding ressources
  - Processors, memory, disk, network

Extensibility

- Ideal: linear speed-up
  - Linear increase of performance by growing the components
    - For a fixed database size and load



Speed-up Limits

- Hardware/software
  - As we add more resources, arbitration conflicts increase
    - E.g. Access to the bus by processors
- Application
  - Only part of a program can be parallelized
  - Recall: Amdahl's law that gives the maximum speed-up
    - *Seq* = fraction of code that cannot be parallelized

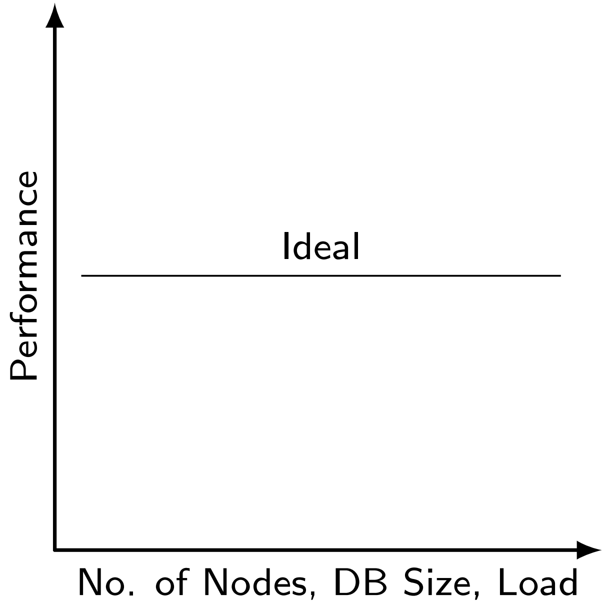
$$\frac{1}{Seq + \dfrac{1 - Seq}{NbProc}}$$

Examples

- Seq=0, NbProc=4 => speed-up= 4
- Seq=30%, NbProc=4 => speed-up= 2,1
- Seq=30%, NbProc=8 => speed-up= 2,5

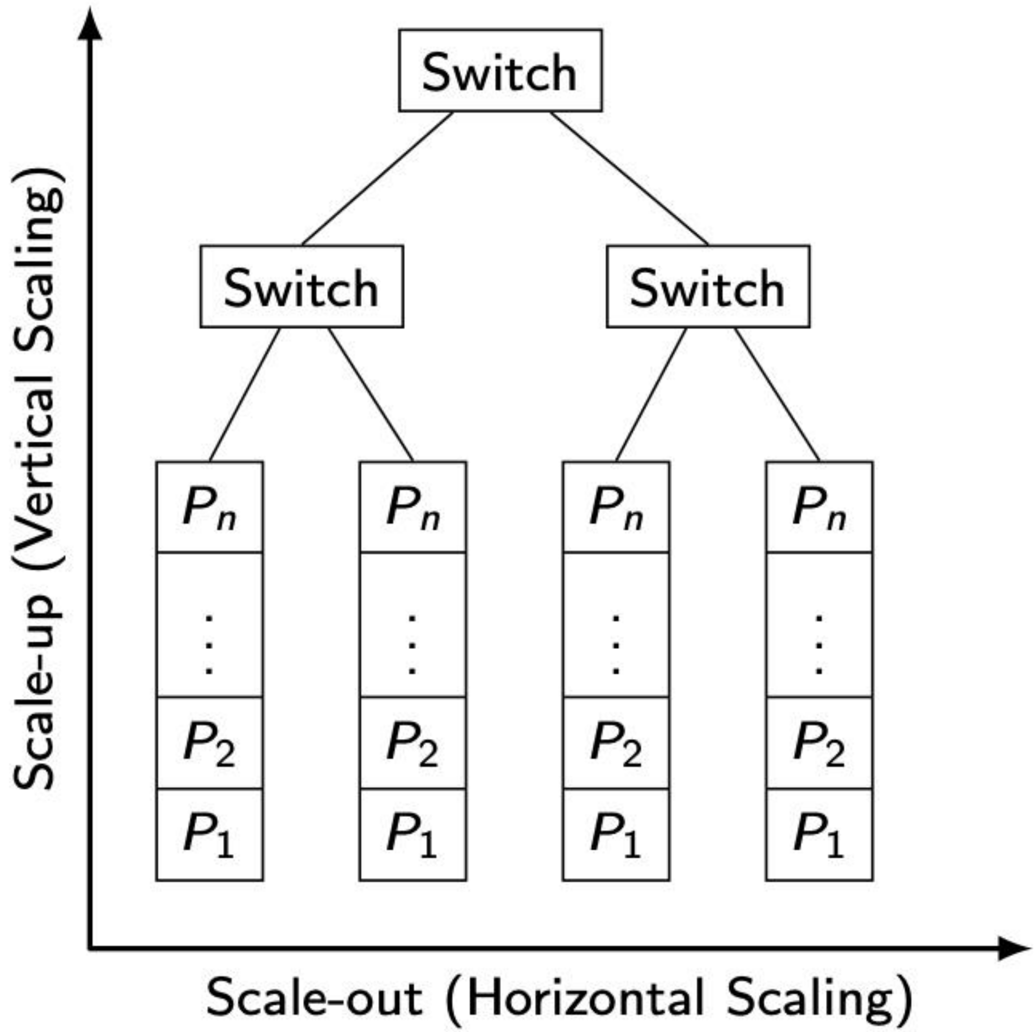
Scalability

- Ideal: linear scale-up
  - Sustained performance for a linear increase of database size and load
  - By proportional increase of components



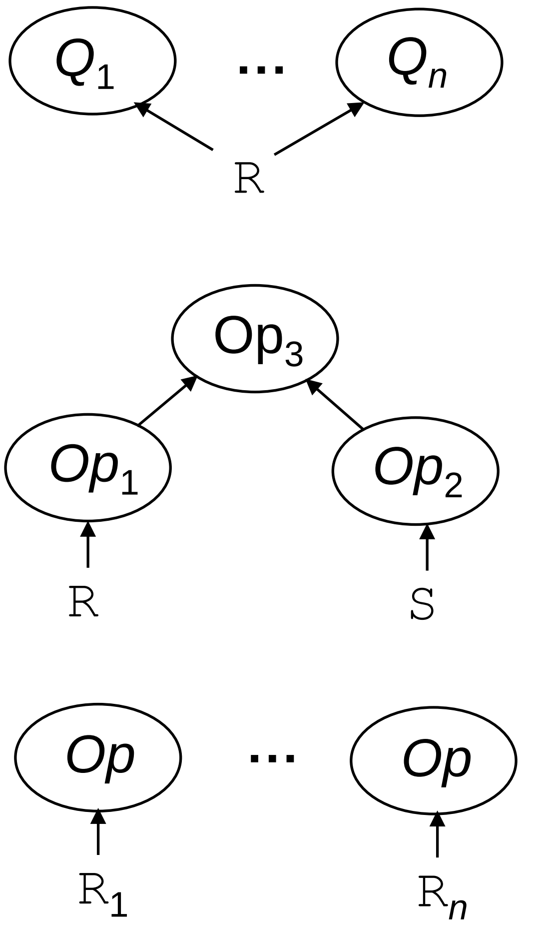
Vertical vs Horizontal Scaleup

- Typically in a computer cluster



Data-based Parallelism

- Inter-query
  - Different queries on the same data
  - For concurrent queries
- Inter-operation
  - Different operations of the same query on different data
  - For complex queries
- Intra-operation
  - The same operation on different data
  - For large queries



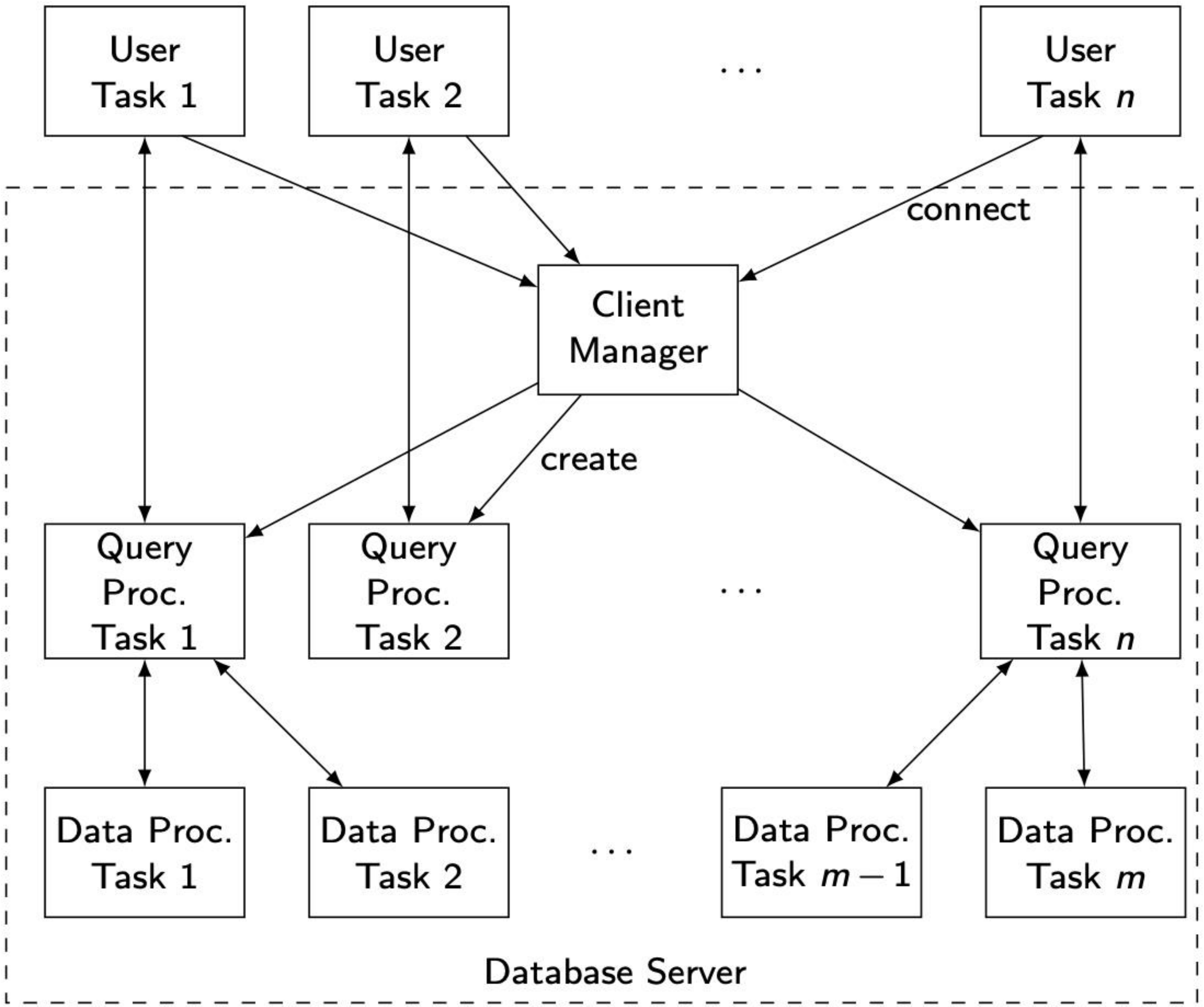
Barriers to Parallelism

- Startup
  - The time needed to start a parallel operation may dominate the actual computation time
- Interference
  - When accessing shared resources, each new process slows down the others (hot spot problem)
- Skew
  - The response time of a set of parallel processes is the time of the slowest one
- Parallel data management techniques intend to overcome these barriers

Outline

- Parallel Database Systems
  - Parallel Architectures
  - Data Placement
  - Query Processing
  - Load Balancing and Fault-tolerance
  - Database Clusters

Parallel DBMS – general architecture



Parallel DBMS Functions

- Client manager
  - Support for client interactions and user sessions
  - Load balancing
- Query processor
  - Compilation and optimization
  - Catalog and metadata management
  - Semantic data control
  - Execution control
- Data processor
  - Execution of DB operations
  - Transaction management
  - Data management

Parallel System Architectures

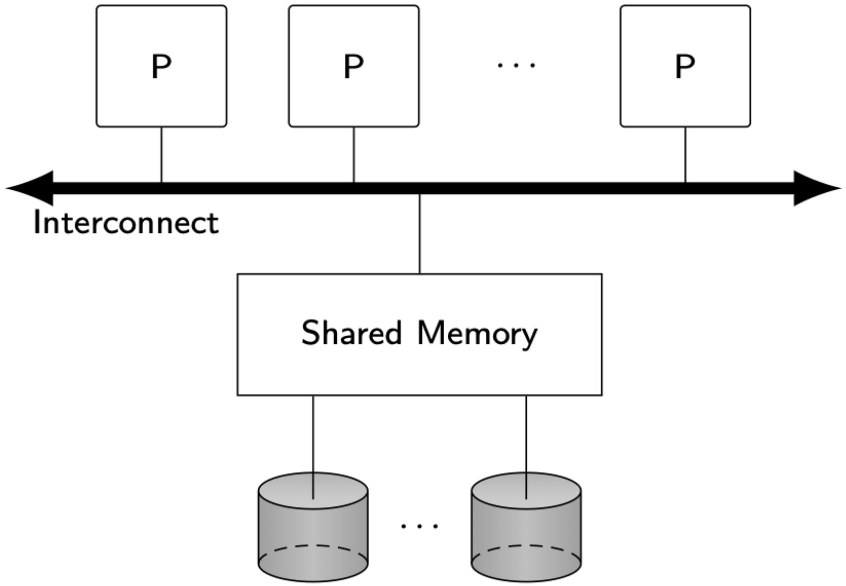
- Shared memory (SM)
  - Uniform Memory Architecture (UMA)
  - Non-Uniform Memory Architecture (NUMA)
- Shared disk (SD)
- Shared nothing (SN)

UMA

- Physical memory shared by all processors
  - Symmetric multiprocessor (SMP) or multicore processor
  - Constant access time
- Examples
  - XPRS, Volcano, DBS3
- Assessment

  \+ Simplicity, load balancing, fast communication

  \- Network cost, low extensibility

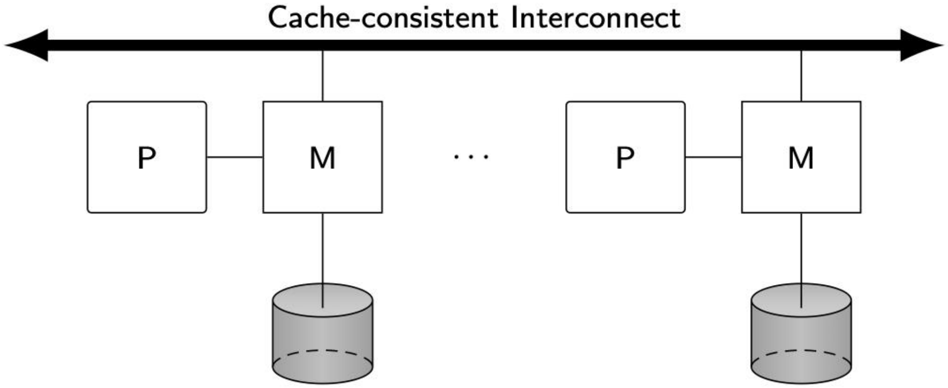


NUMA

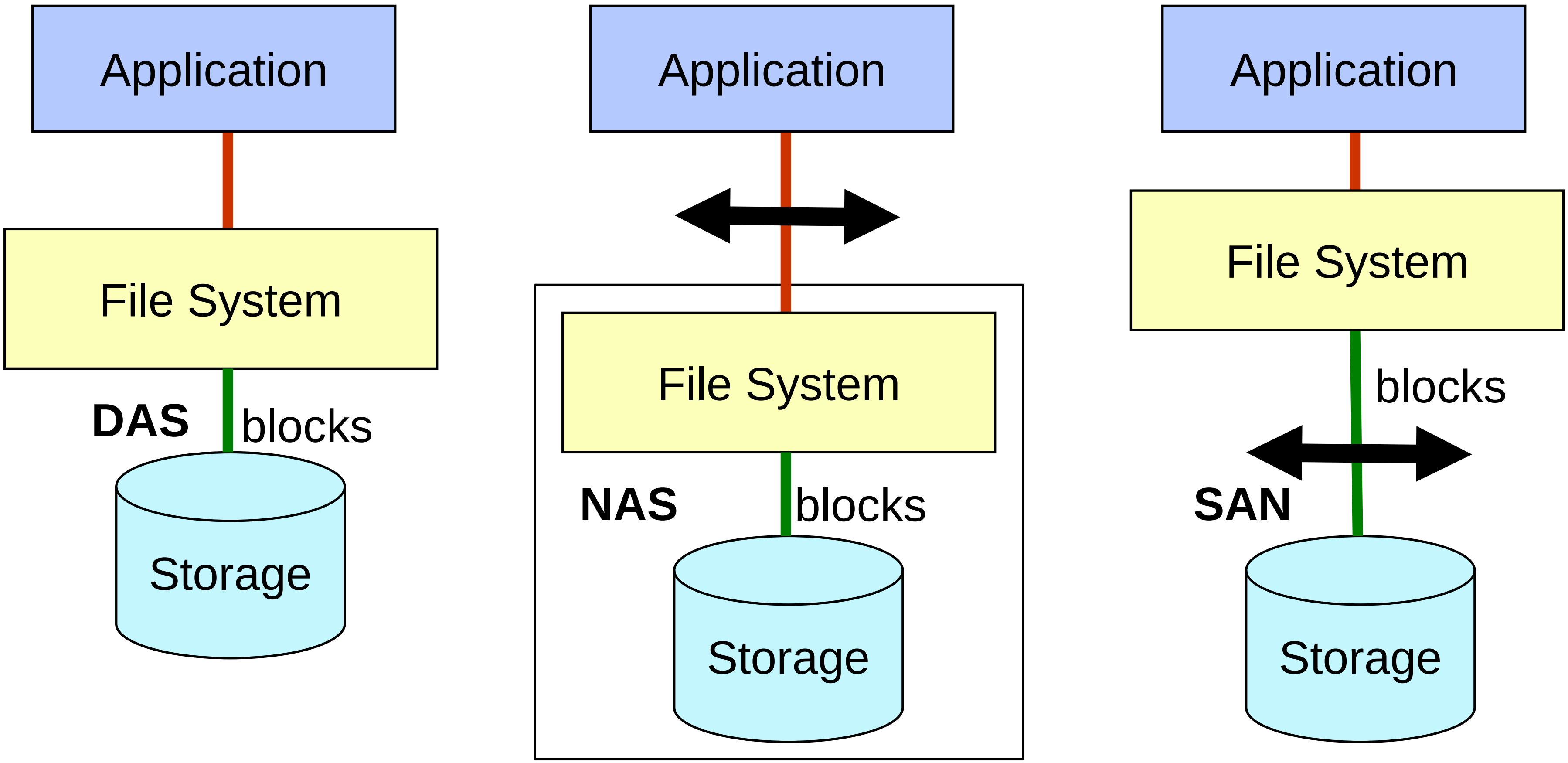
- Shared-memory vs. distributed memory
  - Mixes two different aspects : addressing and memory
    - Addressing: single address space vs multiple address spaces
    - Physical memory: central vs distributed
- NUMA = single address space on distributed physical memory
  - Eases application portability
  - Extensibility
- The most successful NUMA is Cache Coherent NUMA (CC-NUMA)

CC-NUMA

- Principle
  - Main memory physically distributed (as in shared-nothing)
  - However, any processor has access to all other processors' memories
- Cache consistency
  - Special consistent cache interconnect
    - Remote memory access very efficient, only a few times (typically between 2 and 3 times) the cost of local access
  - Exploit Remote Direct Memory Access (RDMA)
    - Now provided by low latency cluster interconnects such as Infiniband and Myrinet



Storage: DAS vs NAS vs SAN



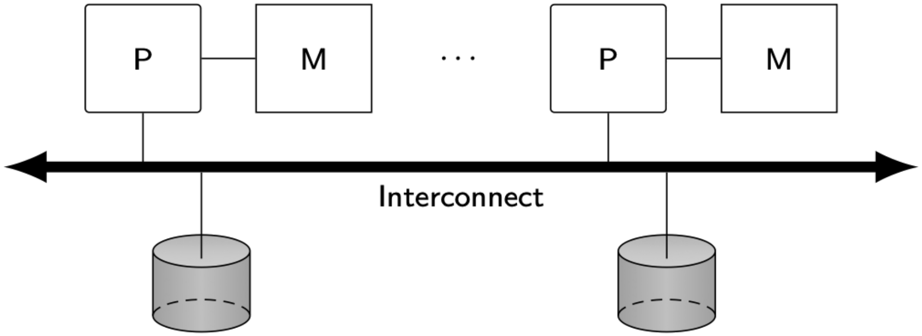
Direct Attached Storage

Network Attached Storage

Storage Area Network

Shared-Disk

- Shared disk, private memory
  - SAN
  - Cache coherency
- Examples
  - Oracle RAC et Exadata
  - IBM PowerHA
- Assessment
  - \+ Simplicity for admin.
  - \- Network cost (SAN), scalability

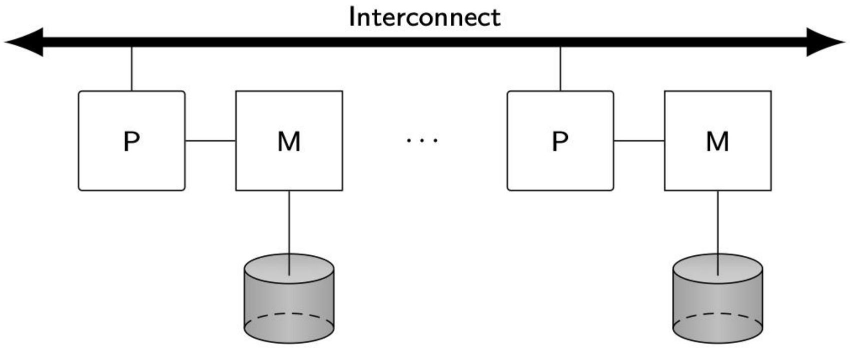


Shared-Nothing

- No sharing of either disk or memory
  - Data partitioning
- Examples
  - DB2 DPF, SQL Server Parallel DW, Teradata, MySQLcluster
  - NoSQL, NewSQL
- Assessment

  \+ Scalability, cost/performance

  \- Complex (distributed updates)



Parallel DBMS Techniques

- Data placement
  - Data partitioning
  - Replication
- Parallel query processing
  - Parallel algorithms for relational operators
  - Query optimization
  - Load balancing
- Transaction management
  - Similar to distributed transaction management
    - But need to scale up to many nodes
  - Fault-tolerance and failover

Outline

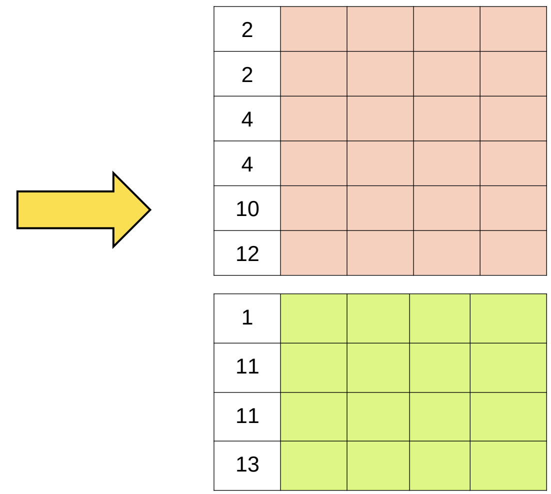
- Parallel Database Systems
  - Parallel Architectures
  - Data Placement
  - Query Processing
  - Load Balancing and Fault-tolerance
  - Database Clusters

Data Partitioning

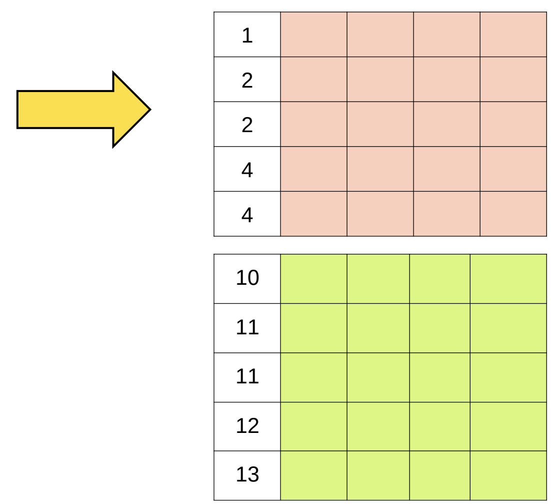
- Each relation is divided in *n* partitions (subrelations), where *n* is a function of relation size and access frequency
- Horizontal partitioning (see Chapter 2)
  - Round-robin
    - Maps *i*-th element to node *i* mod *n*
    - Simple but only exact-match queries
  - Hash
    - Only exact-match queries but small index
  - Range
    - Supports range queries but large index

Partitioning Functions

- Hashing
  - (*k*,*v*) assigned to node *h(k)*
  - Exact match queries
  - Problem with skew distribution



- Range
  - (*k*,*v*) to the node that holds *k*'s interval
  - Exact match and range queries
  - Needs an index on key



Replicated Data Partitioning

- High-availability requires data replication
  - Simple solution is mirrored disks
    - Hurts load balancing when one node fails
  - More elaborate solutions achieve load balancing
    - Interleaved partitioning (Teradata)
    - Chained partitioning (Gamma)

Interleaved Partitioning

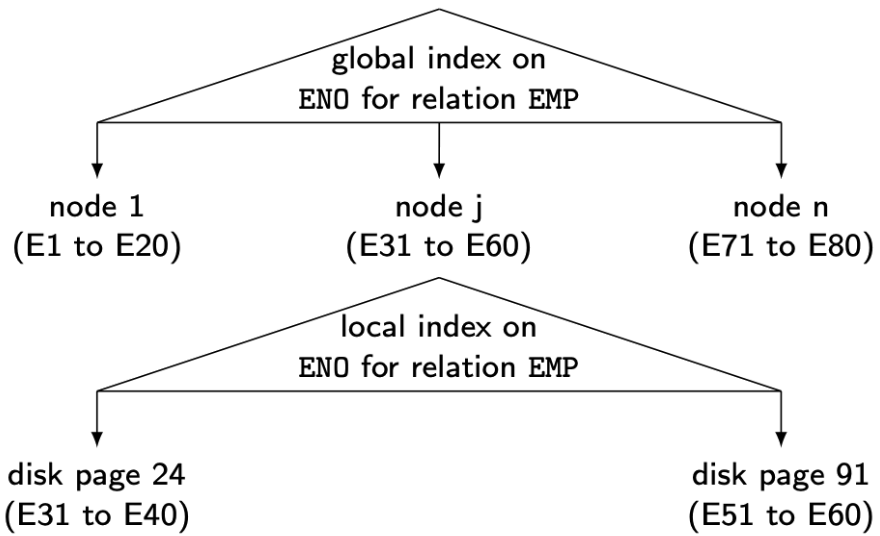
Node	1	2	3	4
Primary copy	R_1	R_2	R_3	R_4
Backup copies	$R_{2,1}$ $R_{3,1}$ $R_{4,1}$	$R_{1,1}$ $R_{3,2}$ $R_{4,2}$	$R_{1,2}$ $R_{2,2}$ $R_{4,3}$	$R_{1,3}$ $R_{2,3}$ $R_{3,3}$

Chained Partitioning

Node	1	2	3	4
Primary copy	R_1	R_2	R_3	R_4
Backup copies	R_4 R_3	R_1 R_4	R_2 R_1	R_3 R_2

Placement Directory

- Performs two functions
  - F_1 (relname, placement attval) = lognode-id
  - F_2 (lognode-id) = phynode-id
- Global index on placement att. available at each node
- In addition, each node has its local index (to access disk pages)



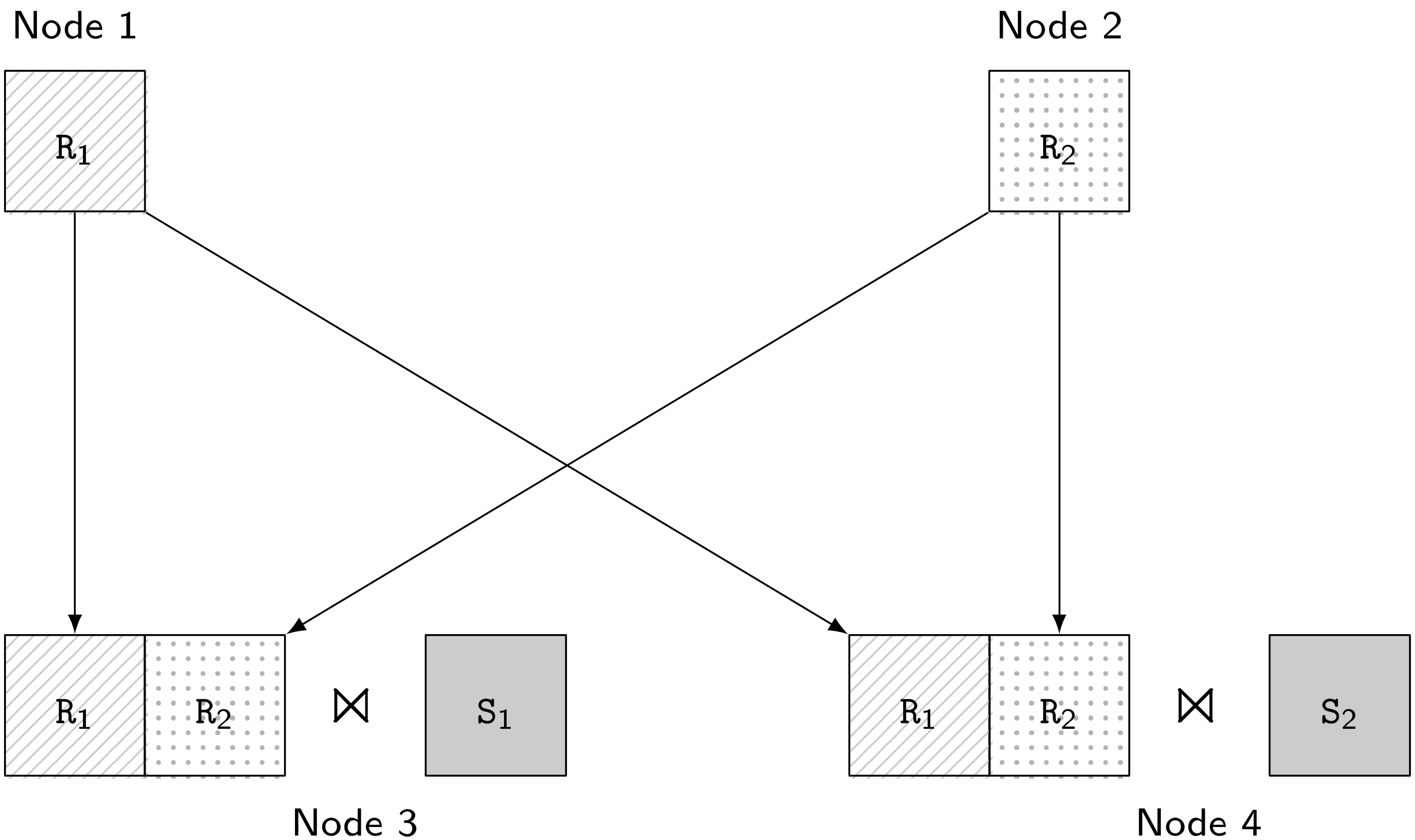
Outline

- Parallel Database Systems
  - Parallel Architectures
  - Data Placement
  - Query processing
  - Load Balancing and Fault-tolerance
  - Database Clusters

Join Processing

- Two basic algorithms for intra-operator parallelism
  - Parallel nested loop join: no special assumption
  - Parallel hash join: equijoin
- They also apply to other complex operators such as duplicate elimination, union, intersection, etc. with minor adaptation

Parallel Nested Loop Join

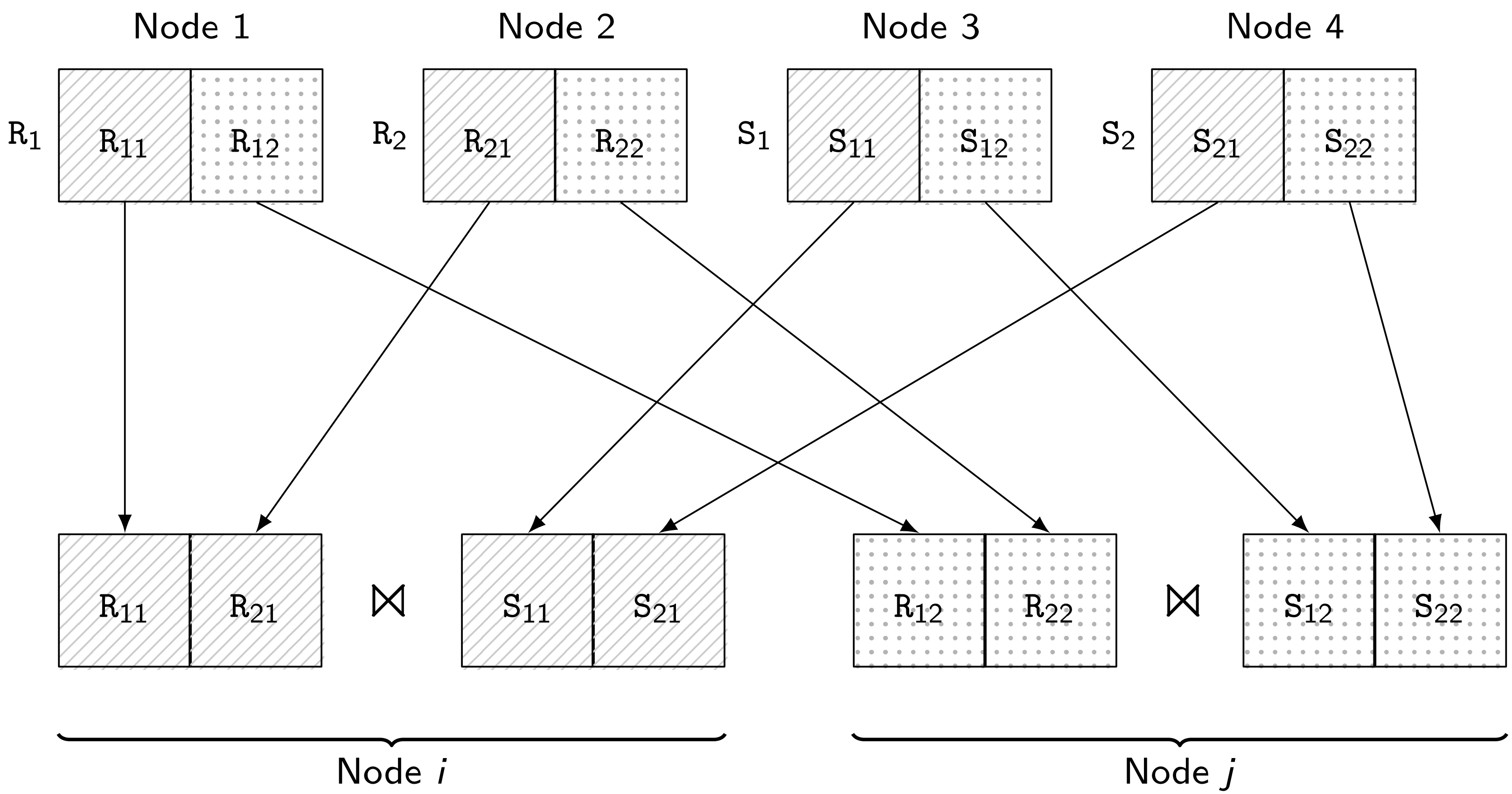


$$R \bowtie S = \bigcup_{i=1}^{n} (R \bowtie S_i)$$

Parallel Nested Loop Join - implementation

- Two nested loops
  - One relation is chosen as the inner relation (e.g. R), the other relation as the outer relation (e.g. S)
- First phase
  - Each fragment of R is sent and replicated at each node that contains a fragment of S (there are n such nodes)
- Second phase
  - Each S-node *j* receives relation R entirely, and locally joins R with its fragment of S
- Pipelining
  - As soon as R tuples are received, they can be processed in pipeline by accessing the S tuples, e.g. through an index

Parallel Hash Join



$$R \bowtie S = \bigcup_{i=1}^{p} (R_i \bowtie S_i)$$

Parallel Hash Join - implementation

- Build phase
  - Hashes $\mathbb{R}$ used as inner relation, on the join attribute
  - Sends it to the target *p* nodes that build a hash table for the incoming tuples
- Probe phase
  - Sends $\mathbb{S}$, the outer relation, associatively to the target *p* nodes
  - Each target node probes the hash table for each incoming tuple
- Pipelining
  - As soon as the hash tables have been built for $\mathbb{R}$, the $\mathbb{S}$ tuples can be sent and processed in pipeline by probing the hash tables

Variants

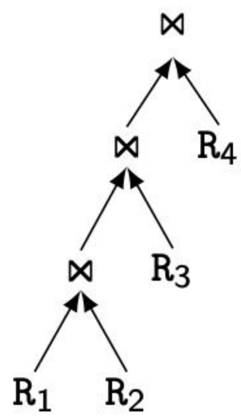
- To exploit large main memories and multicore
- Symmetric hash join
  - The traditional build and probe phases of the basic hash join algorithm are simply interleaved, using two hash tables
  - When a tuple arrives
    - It is used to probe the hash table corresponding to the other relation and find matching tuples
    - Then, it is inserted in its corresponding hash table so that tuplesof the other relation arriving later can be joined
- Ripple join
  - Generalization of the nested loop join algorithm where the roles of inner and outer relation continually alternate during query execution

Parallel Query Optimization

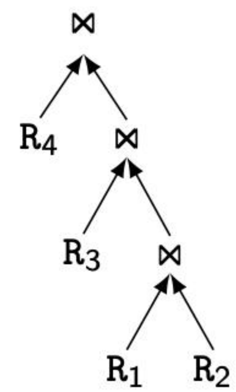
- The objective is to select the “best” parallel execution plan for a query using the following components
- Search strategy
  - Dynamic programming for small search space
  - Randomized for large search space
- Search space
  - Models alternative execution plans as operator trees
  - Different tree shapes
- Cost model (abstraction of execution system)
  - Physical schema info. (partitioning, indexes, etc.)
  - Statistics and cost functions

Operator Trees

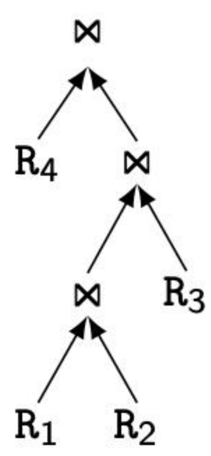
Left-deep



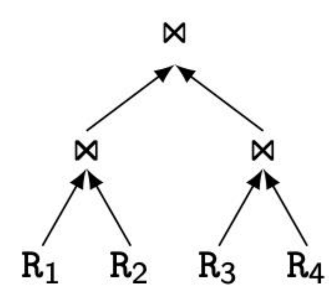
Right-deep



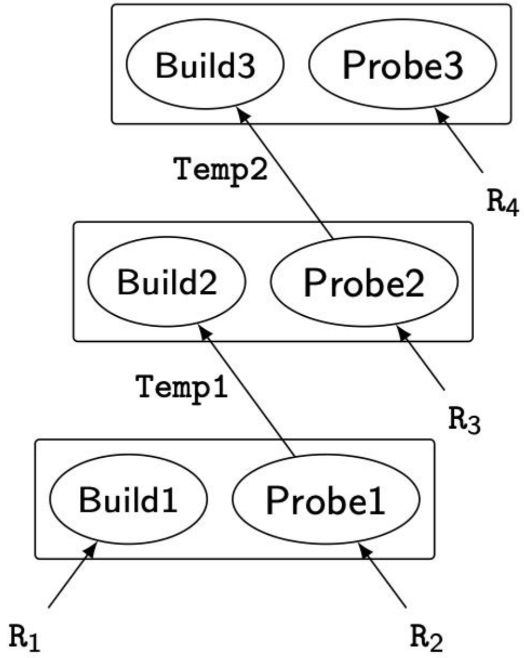
Zig-zag



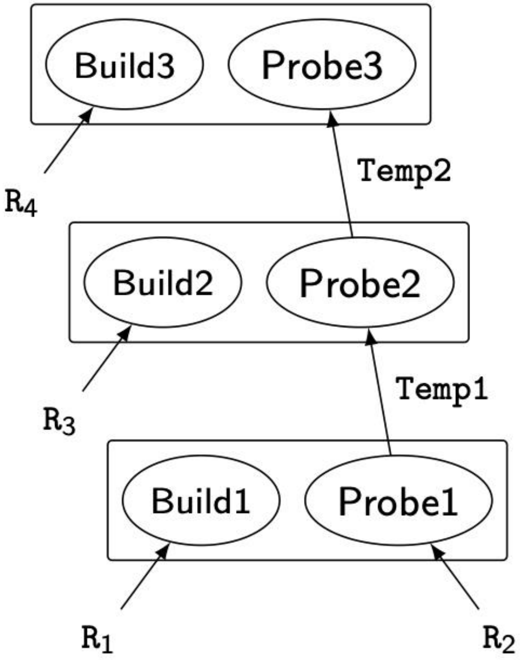
Bushy



Equivalent Hash-Join Trees with Different Scheduling

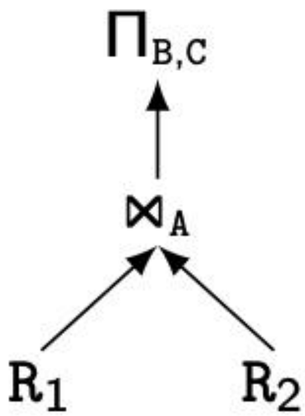


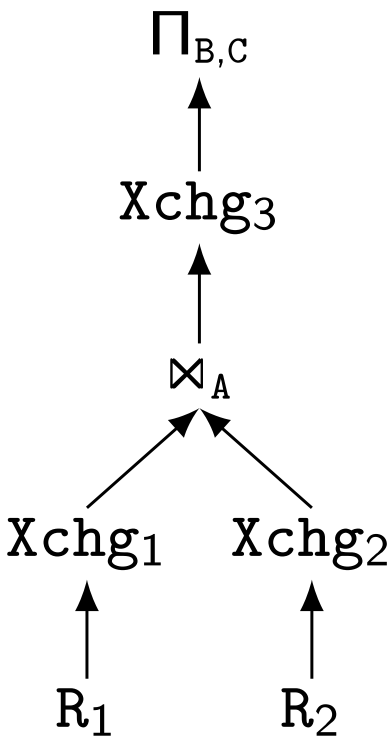
No pipeline



Pipeline of $\mathtt{R}_2$, $\mathtt{Temp}_1$ and $\mathtt{Temp}_2$

Operator Tree with Exchange Operator





$\texttt{Xchg}_1$: partition by $h(\texttt{A})$

$\texttt{Xchg}_2$: partition by $h(\texttt{A})$

$\texttt{Xchg}_3$: partition by $h(\texttt{B}, \texttt{C})$

Cost Model

- Total time
  - Similar to distributed query optimization
  - Adds all CPU, I/O and com. Costs
- Response time
  - More involved as it must take into account pipelining
  - Schedule plan *p* in phases *ph*

$$restTime(p) = \sum_{ph \in p} \Big(\max_{Op \in ph} \big(respTime(Op) + pipeDelay(Op)\big) + storeDelay(ph) \Big)$$

where

- $pipeDelay(Op)$ is the time necessary for the operator *Op* to deliver the first result tuples
- $storeDelay(ph)$ is the time necessary to store the result of phase *ph*

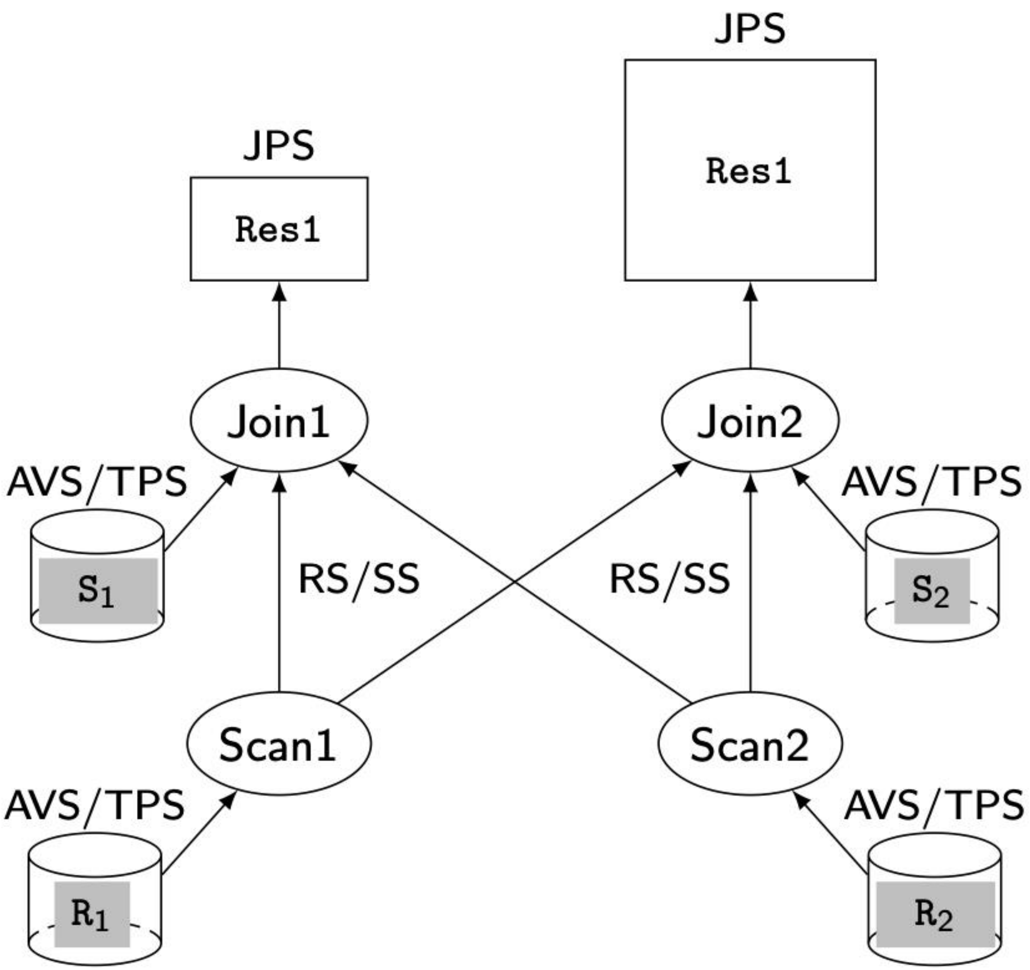
Outline

- Parallel Database Systems
  - Parallel Architectures
  - Data Placement
  - Query Processing
  - Load Balancing and Fault-tolerance
  - Database Clusters

Load Balancing

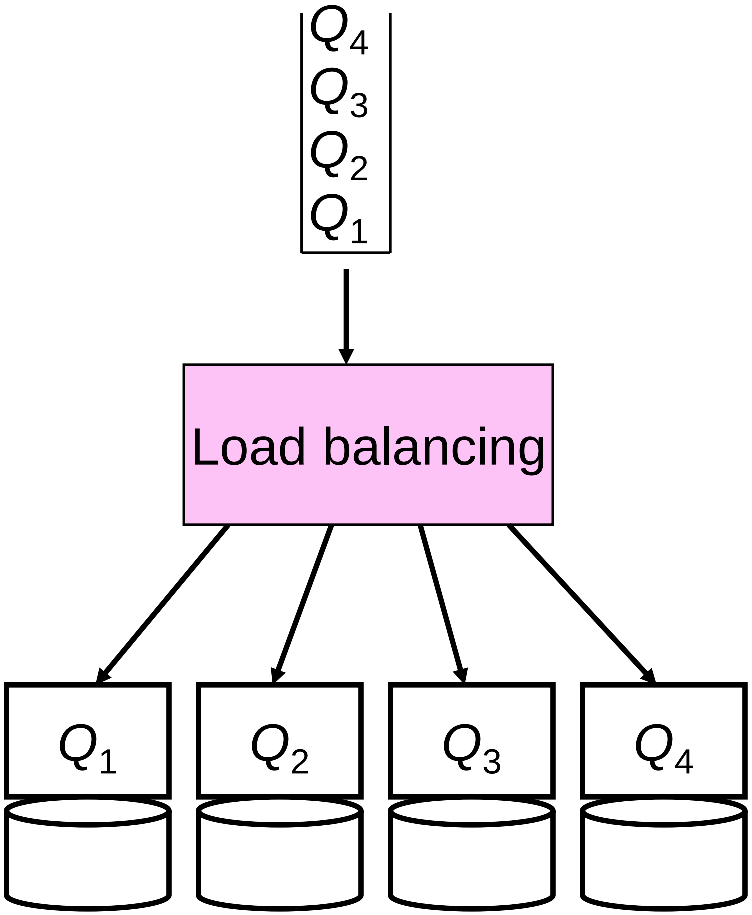
- Problems arise for intra-operator parallelism with *skewed* data distributions
  - Attribute data skew (AVS)
  - Tuple placement skew (TPS)
  - Selectivity skew (SS)
  - Redistribution skew (RS)
  - Join product skew (JPS)
- Solutions
  - Sophisticated parallel algorithms that deal with skew
  - Dynamic processor allocation (at execution time)

Data Skew Example



Load Balancing

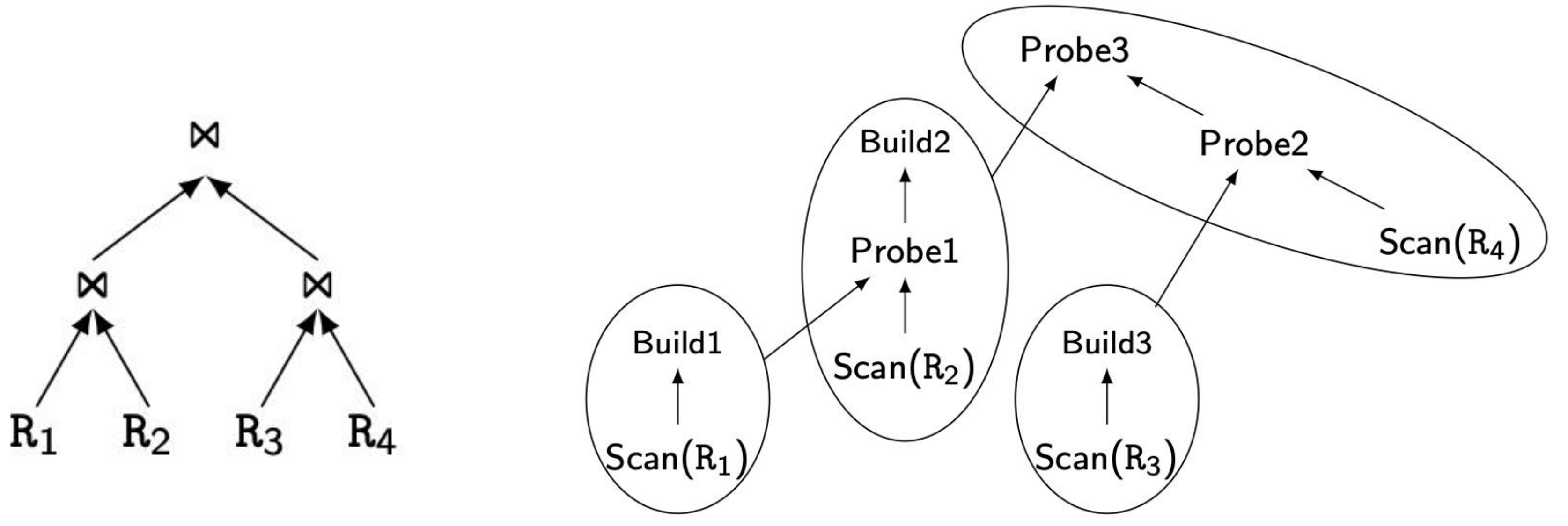
- Choose the node to execute *Q*
  - round robin
  - The least loaded
    - Need to get load information
- Failover
  - In case a node N fails, N's queries are taken over by another node
    - Requires a copy of N's data or SD
- In case of interference
  - Data of an overloaded node are replicated to another node



Dynamic Processing (DP) Execution Model

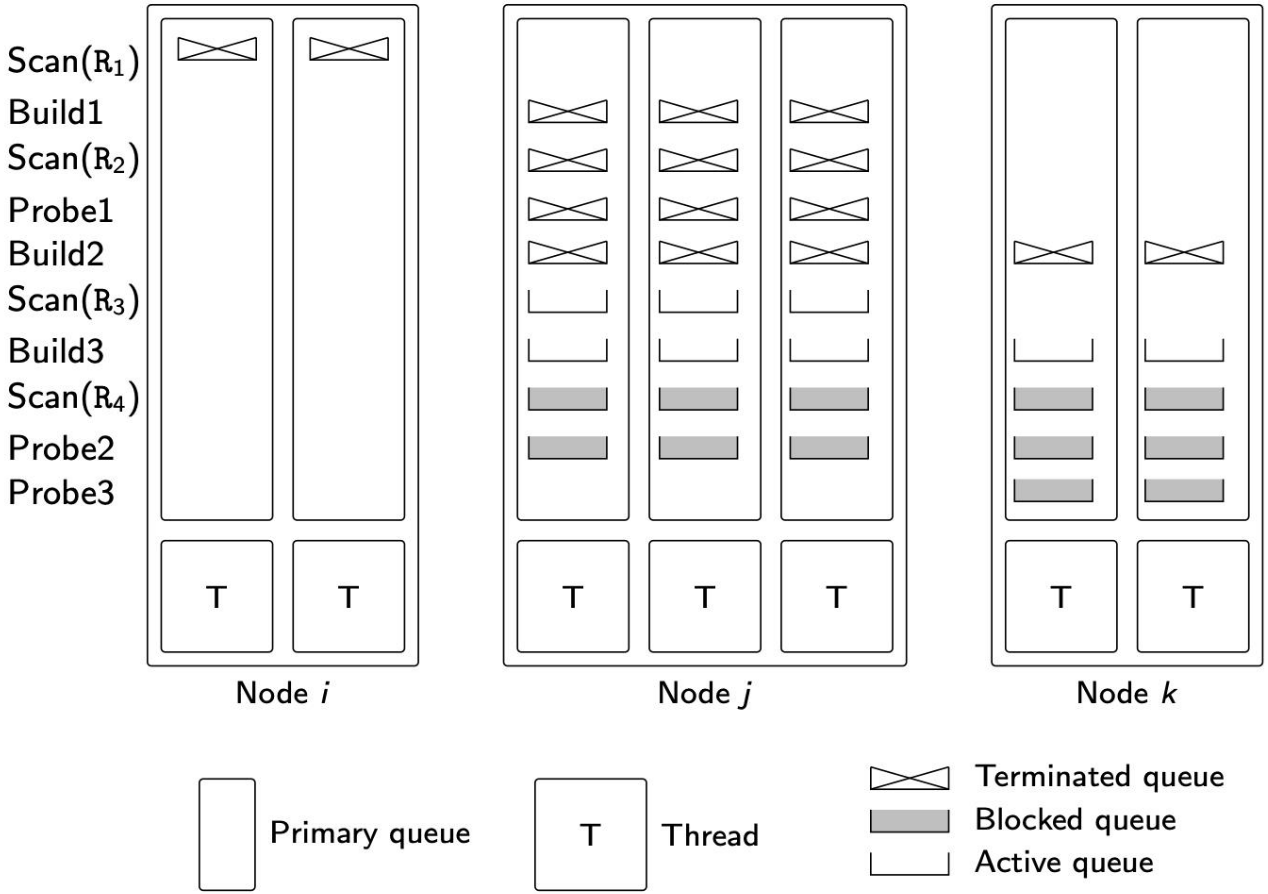
- A query is decomposed into selfcontained units of sequential processing, each of which can be carried out by any processor
- Intuitively, a processor can migrate horizontally (intraoperator parallelism) and vertically (interoperator parallelism) along the query operators
- The parallel execution plan as produced by the optimizer includes operator scheduling constraints that express a partial order among the operators of the query

DP Example



Join tree with 4 pipelined chains

DP Example – snapshot of execution

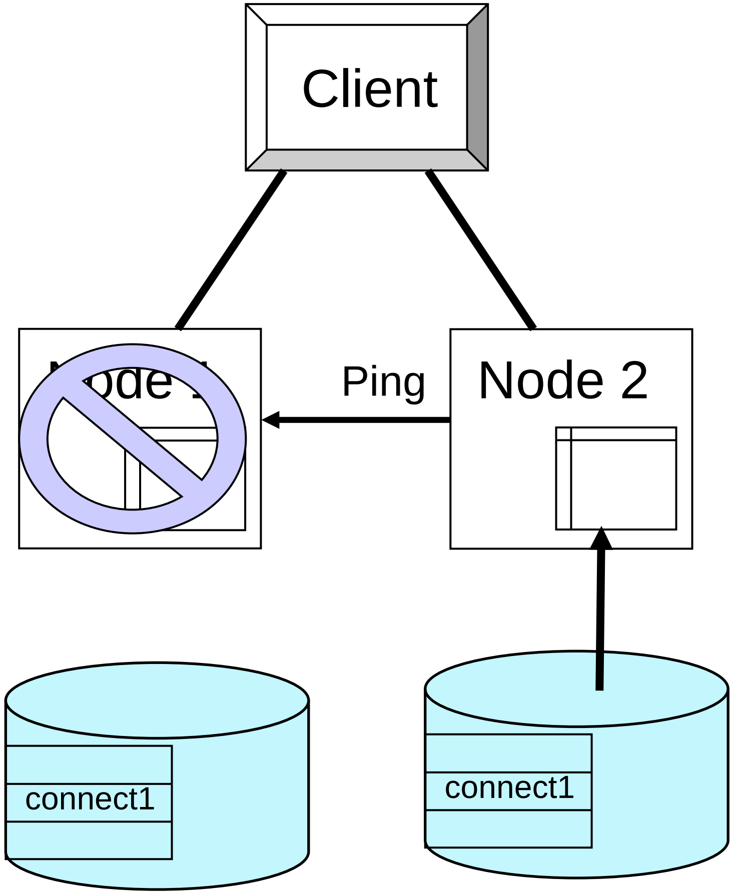


Replication and Failover

- Replication
  - The basis for fault-tolerance and availability
- Failover
  - On a node failure, another node detects and recovers the node's tasks

=>

Savepoints for long queries



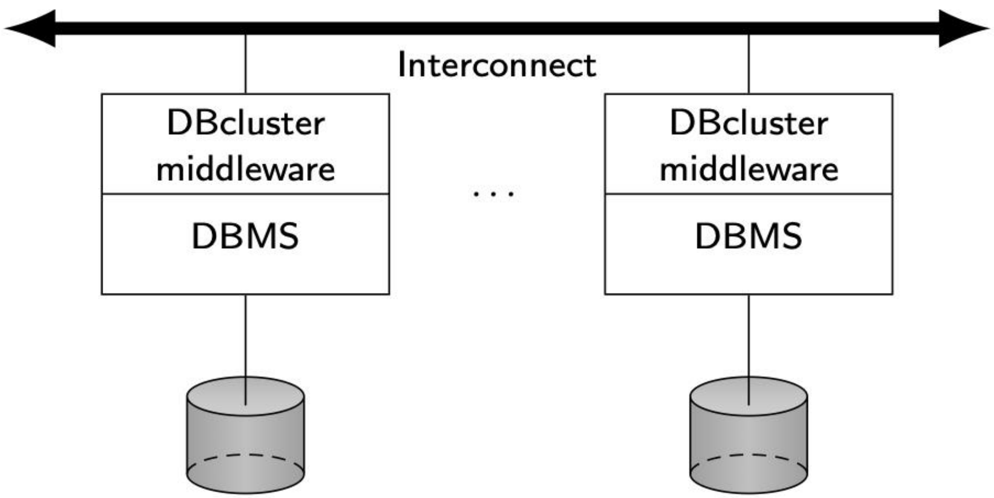
Outline

- Parallel Database Systems
  - Parallel Architectures
  - Data Placement
  - Query Processing
  - Load Balancing and Fault-tolerance
  - Database Clusters

Motivations

- Database cluster = cluster of autonomous databases, each managed by an off-the-shelf DBMS
- Major difference with a parallel DBMS is the use of a “black-box” DBMS at each node
  - In general, same DBMS
- Since the DBMS source code is not necessarily available and cannot be changed to be “cluster-aware,” parallel data management capabilities must be implemented via middleware
- Examples: MySQL or PostgreSQL clusters

Database Cluster Architecture



- Middleware has several software layers
  - Transaction load balancer, replication manager, query processor, and fault-tolerance manager

Replication

- The fast interconnect and communication system can be exploited to support one-copy serializability
- Example with Preventive Replication
  - Uses FIFO reliable multicast (simple and efficient)
  - Principle
    - Each incoming transaction T has a chronological timestamp ts(T) = C, and is multicast to all other nodes where there is a copy.
    - At each node, a time delay is introduced before starting the execution of T. This delay corresponds to the upper bound of the time needed to multicast a message
    - When the delay expires, all transactions that may have committed before C are guaranteed to be received and executed before T , following the timestamp order (i.e., total order)

Load Balancing

- Query load balancing is easy
  - Since all copies are mutually consistent, any node that stores a copy of the data, e.g., the least loaded node, can be chosen at runtime by a conventional load balancing strategy
- Transaction load balancing
  - The total cost of transaction execution at all nodes may be high
  - By relaxing consistency, lazy replication can better reduce transaction execution cost and thus increase performance of both queries and transactions

Query Processing

- Interquery parallelism is naturally obtained as a result of load balancing and replication
  - Useful to increase the throughput of transaction-oriented applications and, to some extent, to reduce the response time of transactions and queries
- Intraquery parallelism is essential to further reduce response times of OLAP queries
  - More difficult than data partitioning because of black-box DBMSs
  - 2 solutions
    - Physical partitioning
    - Virtual partitioning

Physical Partitioning

- Similar to data partioning in distributed databases (see Chap. 2) except that the objective is to increase intraquery parallelism, not locality of reference
- Thus, depending on the query and relation sizes, the degree of partitioning should be much finer
- Under uniform data distribution, yields good intraquery parallelism and outperforms interquery parallelism
- However, it is static and thus very sensitive to data skew conditions and the variation of query patterns that may require periodic repartitioning

Virtual Partitioning

- Uses full replication (each relation is replicated at each node).
- Simple virtual partitioning (SVP)
  - Virtual partitions are dynamically produced for each query and intraquery parallelism is obtained by sending subqueries to different virtual partitions
  - To produce the different subqueries, the query processor adds predicates to the incoming query in order to restrict access to a subset of a relation, i.e., a virtual partition
  - Then, each DBMS that receives a subquery is forced to process a different subset of data items
  - Finally, the partitioned result needs to be combined by an aggregate query

SVP Example

- Consider

```
SELECT PNO, AVG(DUR)
FROM   WORKS
WHERE  SUM(DUR) > 200
GROUP BY PNO
```

- A generic subquery on a virtual partition is obtained by adding to *Q*'s where clause the predicate

```
AND PNO >= 'P1' AND PNO < 'P2'
```

SVP Example

- By binding `['P1', 'P2']` to *n* subsequent ranges of `PNO` values, we obtain *n* subqueries, each for a different node on a different virtual partition of `WORKS`
- The **`AVG`**`(DUR)` operation must be rewritten as **`SUM`**`(DUR)`, **`COUNT`**`(DUR)` in the subquery
- Finally, to obtain the correct result for **`AVG`**`(DUR)`, the composition query must perform
  - **`SUM`**`(DUR)/`**`SUM`**`(`**`COUNT`**`(DUR))`

  over the *n* partial results

SVP Assessment

- Great flexibility for node allocation during query processing since any node can be chosen for executing a subquery
- Not all kinds of queries can benefit from SVP
  - Only queries without subqueries that access a fact table are ok
  - Queries with a subquery must be converted to have no subquery
  - Any other queries cannot benefit
- Limitations
  - Determining the best virtual partitioning attributes and value ranges can be difficult
  - Dependent on the underlying DBMS query capabilities
- Solution: Adaptive Virtual Partitioning