Outline

- Introduction
- Background
- Distributed Database Design
- Database Integration
- Semantic Data Control
- Distributed Query Processing
- Multidatabase Query Processing
- Distributed Transaction Management
- Data Replication
- Parallel Database Systems
    - Data placement and query processing
    - Load balancing
    - Database clusters
- Distributed Object DBMS
- Peer-to-Peer Data Management
- Web Data Management
- Current Issues

The Database Problem

- Large volume of data ⇒ use disk and large main memory
- I/O bottleneck (or memory access bottleneck)

  Speed(disk) << speed(RAM) << speed(microprocessor)
- Predictions

  Moore's law: processor speed growth (with multicore): 50 % per year

  DRAM capacity growth : 4 · every three years

  Disk throughput : 2 · in the last ten years
- Conclusion : the I/O bottleneck worsens

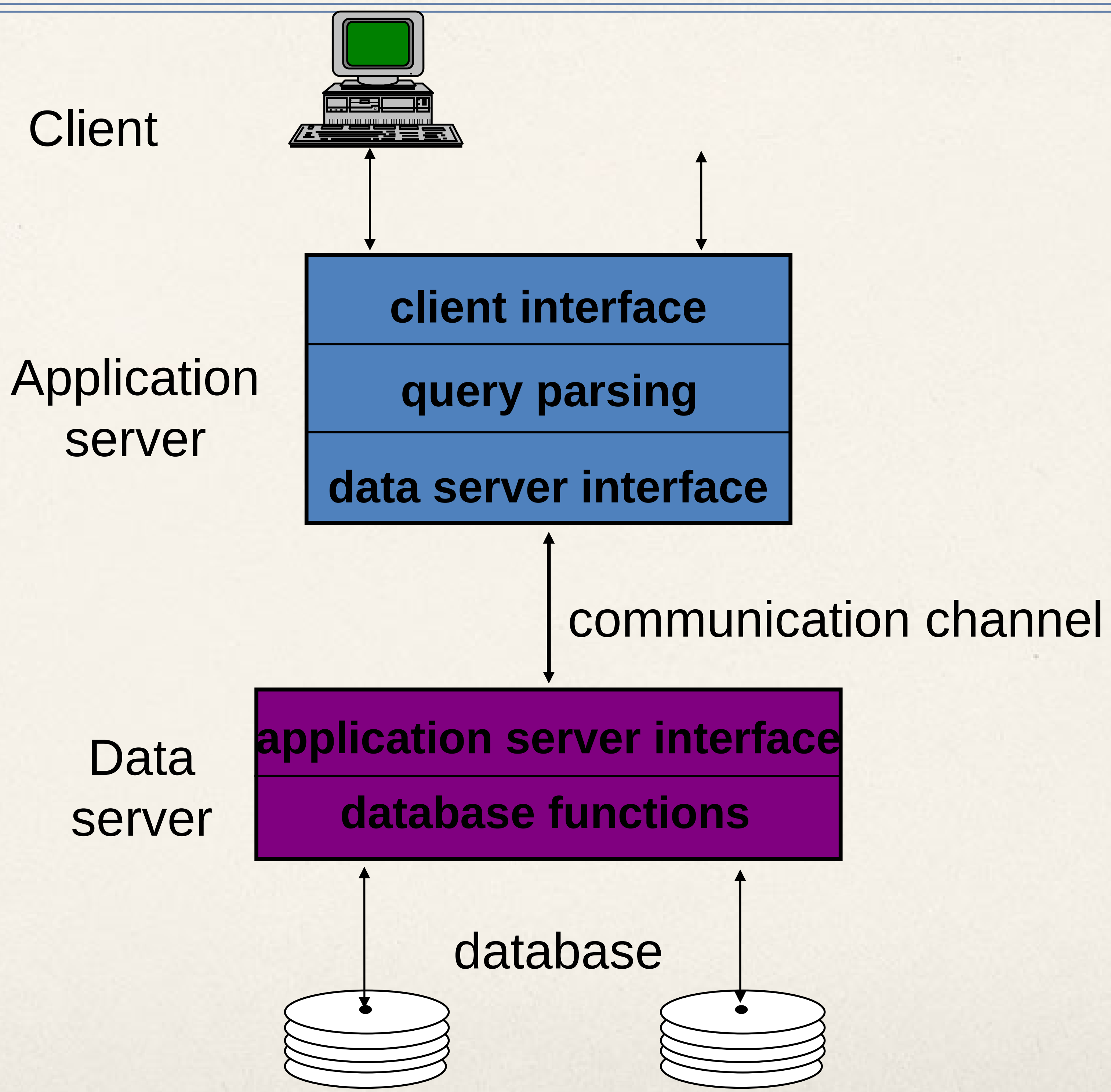
The Solution

- Increase the I/O bandwidth
  - Data partitioning
  - Parallel data access
- Origins (1980's): *database machines*
  - Hardware-oriented ⇒ bad cost-performance ⇒ failure
  - Notable exception : ICL's CAFS Intelligent Search Processor
- 1990's: same solution but using standard hardware components integrated in a multiprocessor
  - Software-oriented
  - Standard essential to exploit continuing technology improvements

Multiprocessor Objectives

- High-performance with better cost-performance than mainframe or vector supercomputer
- Use many nodes, each with good cost-performance, communicating through network
  - Good cost via high-volume components
  - Good performance via bandwidth
- Trends
  - Microprocessor and memory (DRAM): off-the-shelf
  - Network (multiprocessor edge): custom
- The real chalenge is to parallelize applications to run with good *load balancing*

Data Server Architecture



Objectives of Data Servers

- Avoid the shortcomings of the traditional DBMS approach
  - Centralization of data and application management
  - General-purpose OS (not DB-oriented)
- By separating the functions between
  - Application server (or host computer)
  - Data server (or database computer or back-end computer)

Data Server Approach: Assessment

- Advantages
  - Integrated data control by the server (black box)
  - Increased performance by dedicated system
  - Can better exploit parallelism
  - Fits well in distributed environments
- Potential problems
  - Communication overhead between application and data server
    - High-level interface
  - High cost with mainframe servers

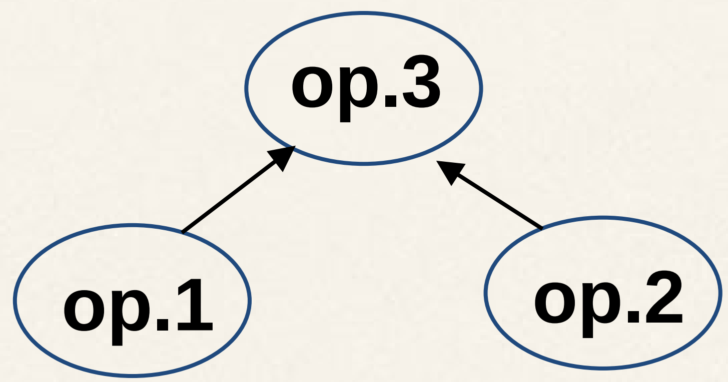
Parallel Data Processing

- Three ways of exploiting high-performance multiprocessor systems:
  1. Automatically detect parallelism in sequential programs (e.g., Fortran, OPS5)
  2. Augment an existing language with parallel constructs (e.g., C*, Fortran90)
  3. Offer a new language in which parallelism can be expressed or automatically inferred
- Critique
  1. Hard to develop parallelizing compilers, limited resulting speed-up
  2. Enables the programmer to express parallel computations but too low-level
  3. Can combine the advantages of both (1) and (2)

Data-based Parallelism

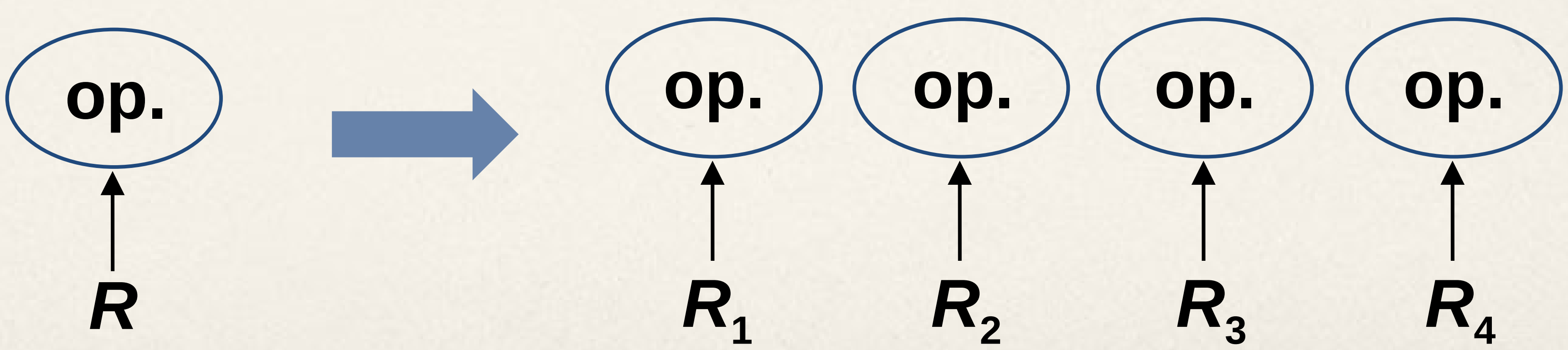
- Inter-operation

  p operations of the same query in parallel



- Intra-operation

  The same op in parallel



Parallel DBMS

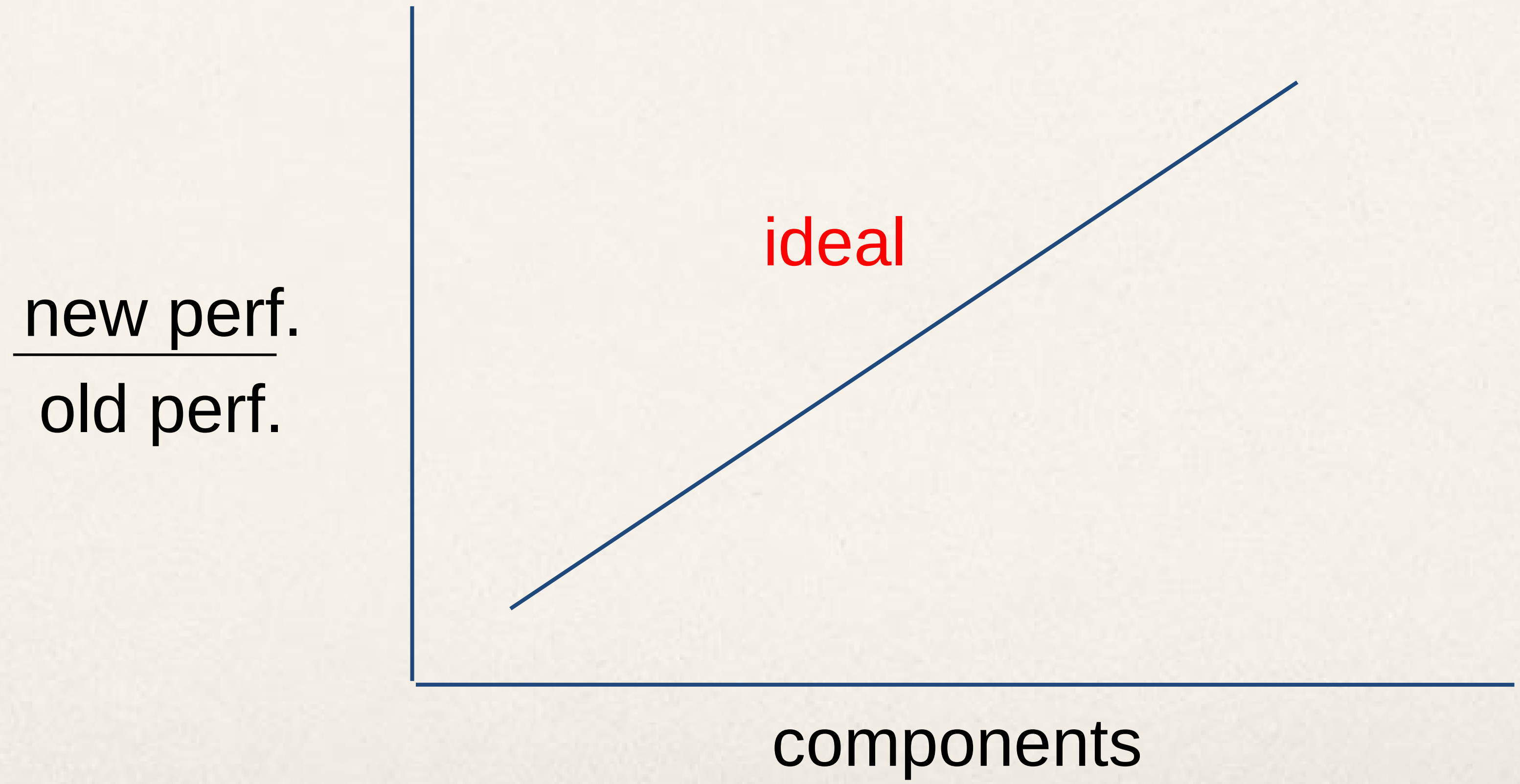
- Loose definition: a DBMS implemented on a tighly coupled multiprocessor
- Alternative extremes
  - Straighforward porting of relational DBMS (the software vendor edge)
  - New hardware/software combination (the computer manufacturer edge)
- Naturally extends to distributed databases with one server per site

Parallel DBMS - Objectives

- Much better cost / performance than mainframe solution
- High-performance through parallelism
  - High throughput with inter-query parallelism
  - Low response time with intra-operation parallelism
- High availability and reliability by exploiting data replication
- Extensibility with the ideal goals
  - Linear speed-up
  - Linear scale-up

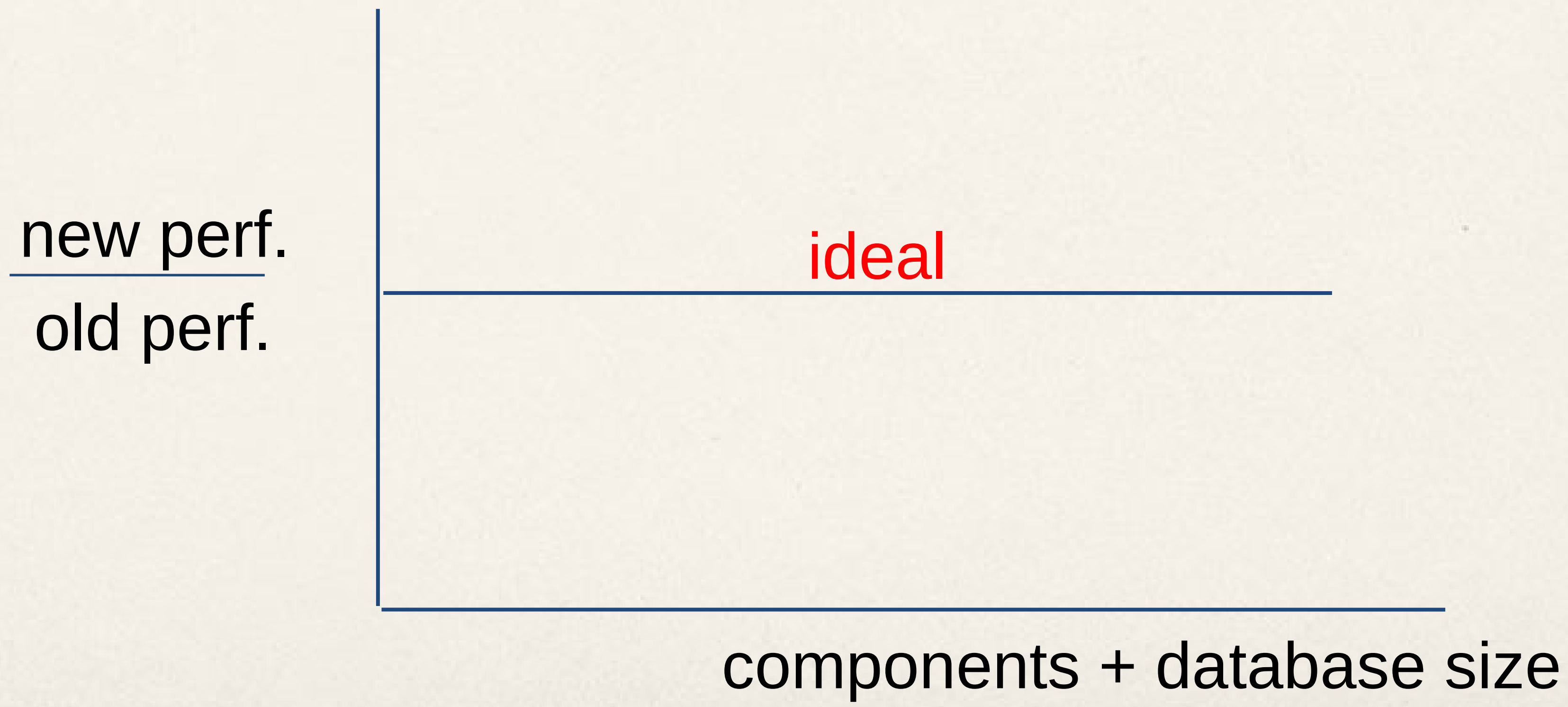
Linear Speed-up

Linear increase in performance for a constant DB size and proportional increase of the system components (processor, memory, disk)



Linear Scale-up

Sustained performance for a linear increase of database size and proportional increase of the system components.



Barriers to Parallelism

- Startup

  The time needed to start a parallel operation may dominate the actual computation time

- Interference

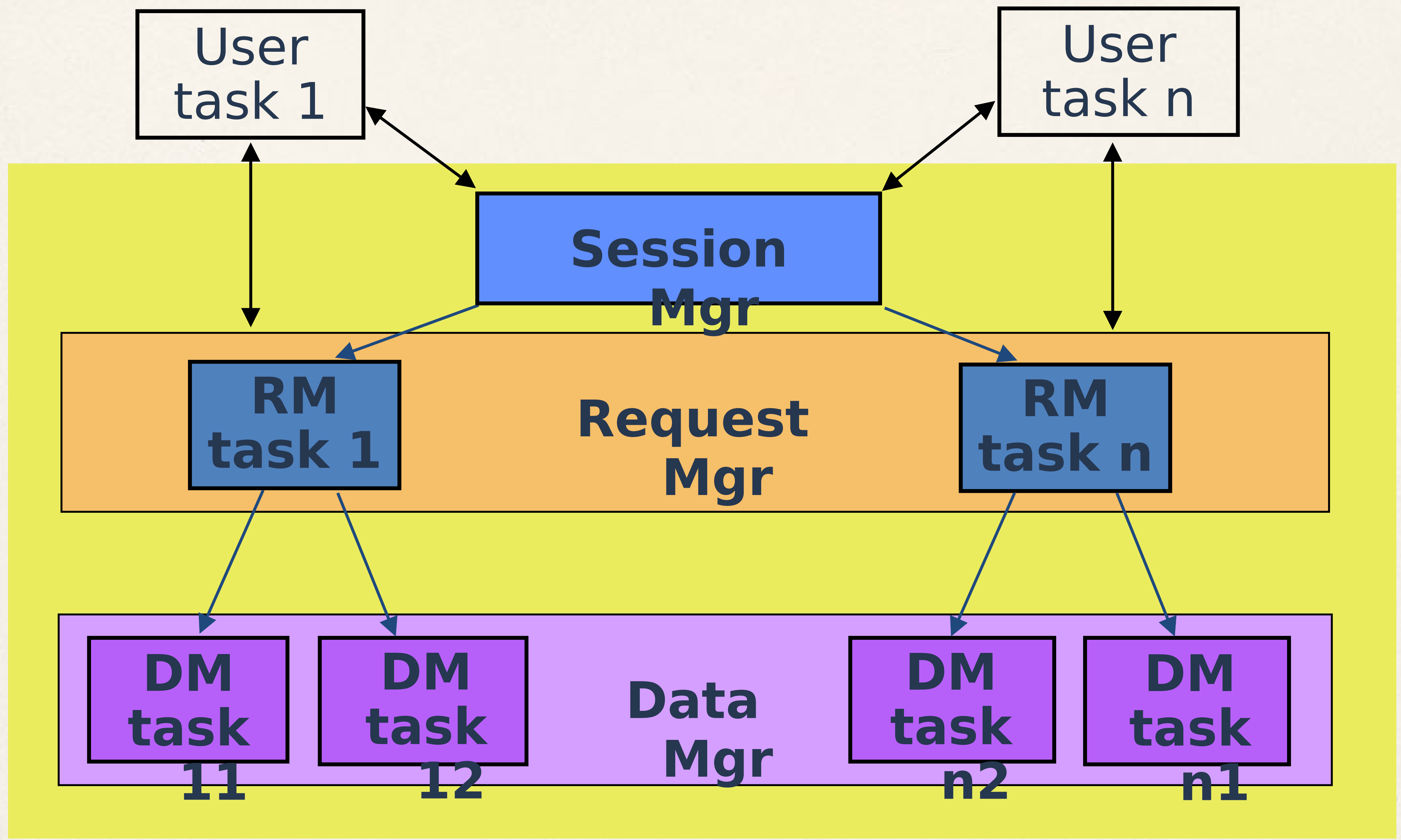
  When accessing shared resources, each new process slows down the others (hot spot problem)

- 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

Parallel DBMS – Functional Architecture



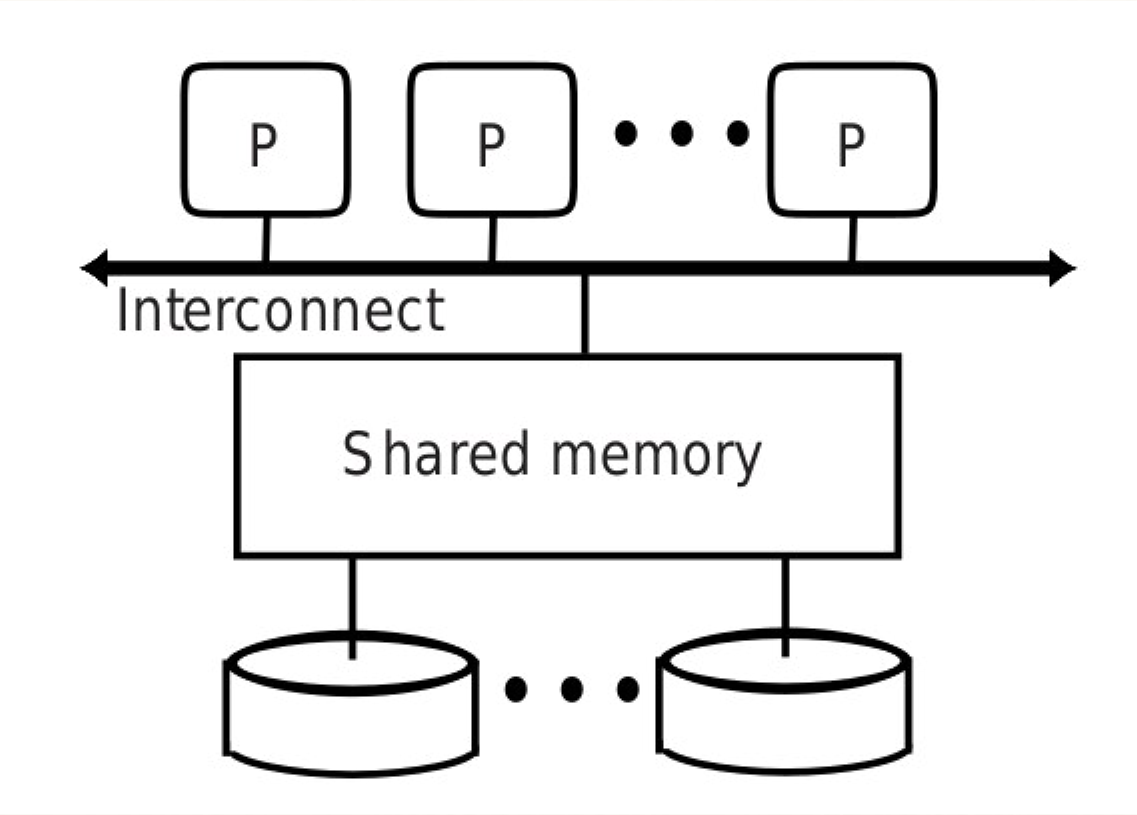
Parallel DBMS Functions

- Session manager
  - Host interface
  - Transaction monitoring for OLTP
- Request manager
  - Compilation and optimization
  - Data directory management
  - Semantic data control
  - Execution control
- Data manager
  - Execution of DB operations
  - Transaction management support
  - Data management

Parallel System Architectures

- Multiprocessor architecture alternatives
  - Shared memory (SM)
  - Shared disk (SD)
  - Shared nothing (SN)
- Hybrid architectures
  - Non-Uniform Memory Architecture (NUMA)
  - Cluster

Shared-Memory



DBMS on symmetric multiprocessors (SMP)

Prototypes: XPRS, Volcano, DBS3

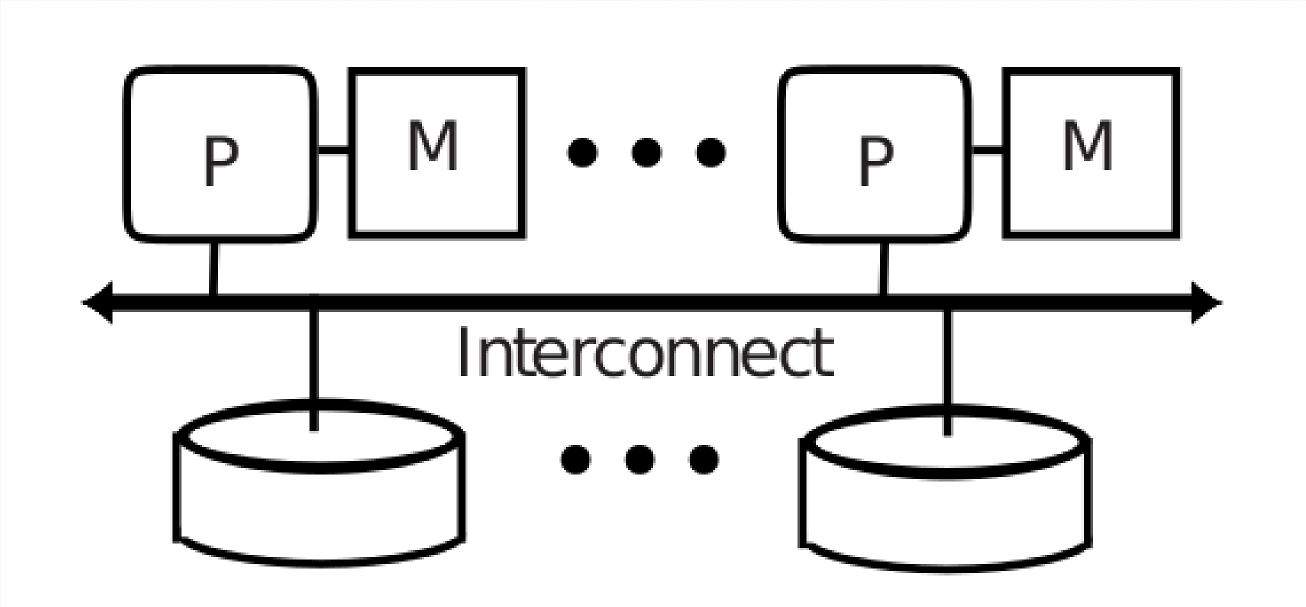
+ Simplicity, load balancing, fast communication

- Network cost, low extensibility

Shared-Memory

- Meta-information (directory) and control information (e.g., lock tables) can be shared by all processors
- Inter-query parallelism comes for free
- Intra-query parallelism requires some parallelization but remains rather simple
- Load balancing is easy to achieve
    - Allocating each new task to the least busy processor.
- Shared-memory has three problems: high cost, limited extensibility and low availability
    - Interconnect requires fairly complex hardware
    - With faster processors (even with larger caches), conflicting accesses to the shared-memory increase rapidly and degrade performance
    - Extensibility is limited to a few tens of processors, typically up to 16

Shared-Disk



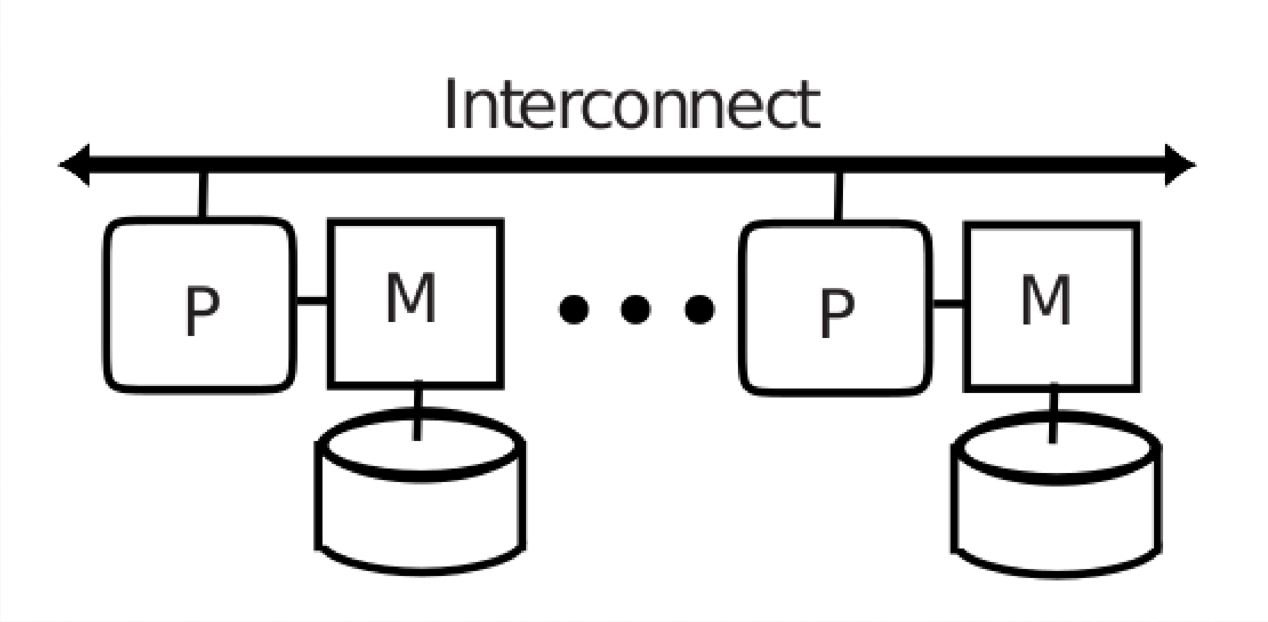
Origins : DEC's VAXcluster, IBM's IMS/VS Data Sharing
Used first by Oracle with its Distributed Lock Manager
Now used by most DBMS vendors

+ network cost, extensibility, migration from uniprocessor
- complexity, potential performance problem for cache coherency

Shared-Disk

- Any processor has access to any disk unit through the interconnect but exclusive access to its main memory.
  - Each processor-memory node is under the control of its own OS
  - Global cache consistency is needed
  - This is typically achieved using a distributed lock manager
- Shared-disk has a number of advantages:
  - Lower cost, high extensibility (up to 100), load balancing, availability (owns memory), and easy migration from centralized systems.
  - Cost of the interconnect is significantly less than with shared-memory
- Shared-disk suffers from higher complexity & potential perform. problems.
  - Distributed locking and two-phase commit.

Shared-Nothing



Used by Teradata, IBM, Sybase, Microsoft for OLAP

Prototypes: Gamma, Bubba, Grace, Prisma, EDS

+ Extensibility, availability

- Complexity, difficult load balancing

Shared-Nothing

- Each processor has exclusive access to its main memory and disk unit(s)
  - Each node can be viewed as a local site (with its own database and software) in a distributed database system.
  - Most solutions of DDBMS may be reused: fragmentation, transaction management and query processing
  - Architecture is often called Massively Parallel Processor (MPP), opposed to SMP
- Shared-nothing +-s: lower cost, high extensibility, high availability
  - Shared-disk that requires a special interconnect, not shared-nothing
  - Careful partitioning of the data on multiple disks => almost linear speedup and linear scaleup for simple workloads
- SN is much more complex to manage than either SM or SD
  - Distributed DB functions assuming large numbers of nodes; load balancing is more difficult (=>partitioning); adding new nodes?

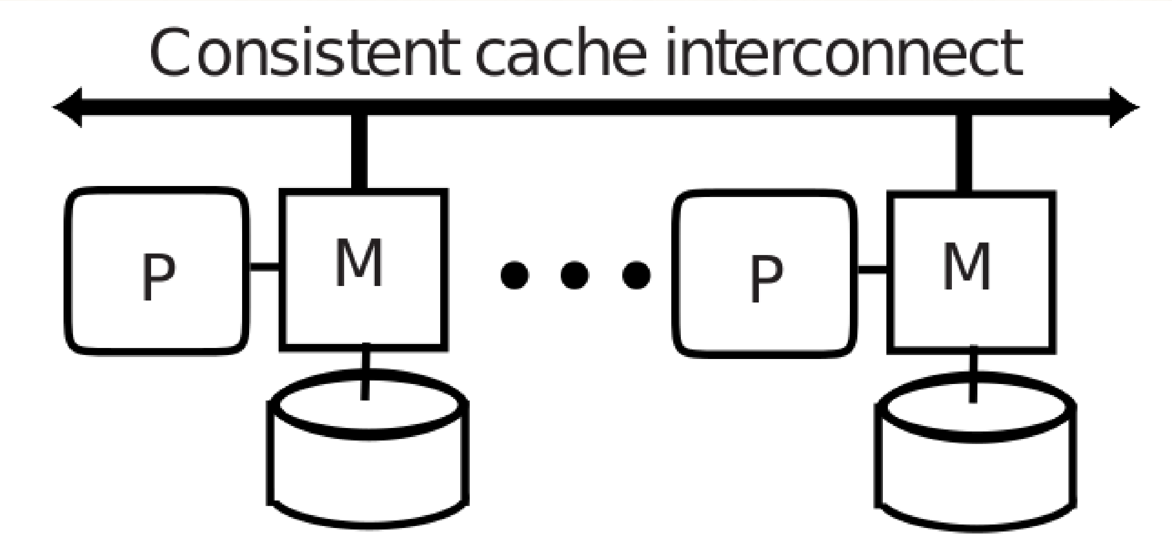
Hybrid Architectures

- Various possible combinations of the three basic architectures are possible to obtain different trade-offs between cost, performance, extensibility, availability, etc.
- Hybrid architectures try to obtain the advantages of different architectures:
  - efficiency and simplicity of shared-memory
  - extensibility and cost of either shared disk or shared nothing
- 2 main kinds: NUMA and cluster

NUMA

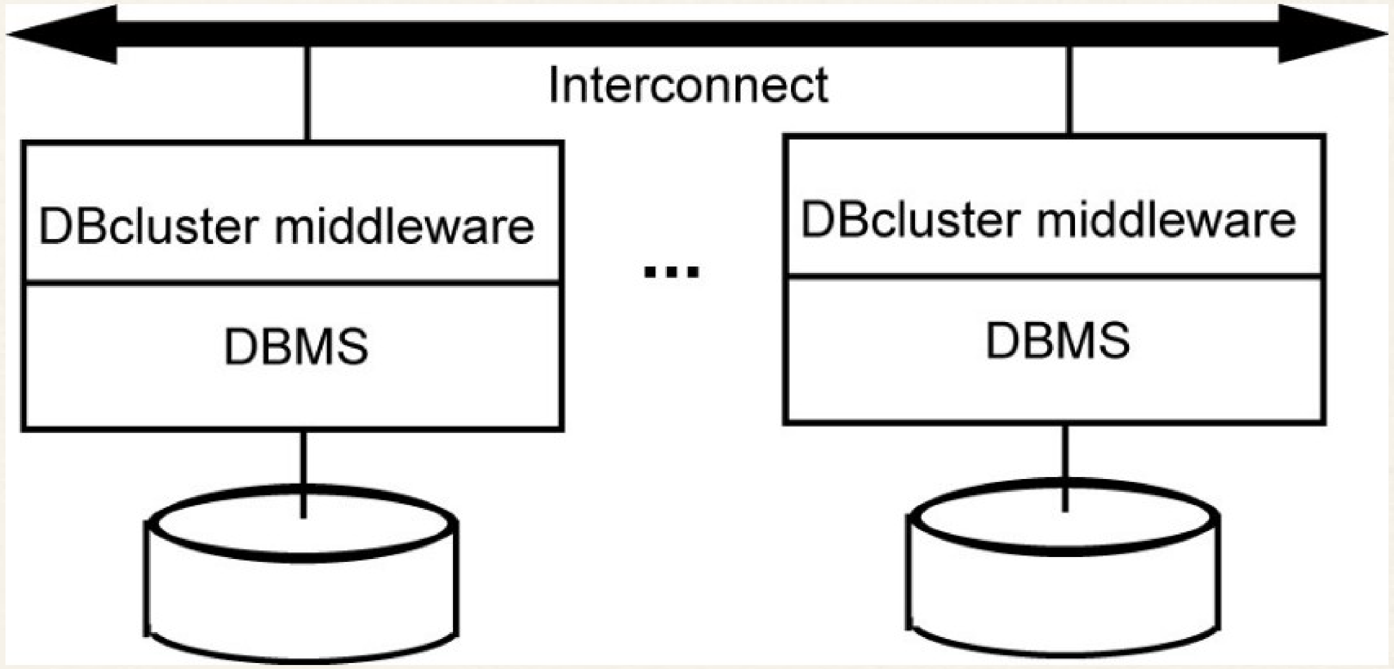
- Shared-Memory vs. Distributed Memory
  - Mixes two different aspects : addressing and memory
    - Addressing: single address space vs multiple address spaces
    - Physical memory: central vs distributed
- NUMA = single address space on distributed physical memory
  - Eases application portability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

CC-NUMA



- Principle
  - Main memory distributed as with shared-nothing
  - However, any processor has access to all other processors' memories
- Similar to shared-disk, different processors can access the same data in a conflicting update mode, so global cache consistency protocols are needed.
  - Cache consistency done in hardware through a special consistent cache interconnect
    - Remote memory access very efficient, only a few times (typically between 2 and 3 times) the cost of local access

Cluster



- Combines good load balancing of SM with extensibility of SN
- Server nodes: off-the-shelf components
    - From simple PC components to more powerful SMP
    - Yields the best cost/performance ratio
    - In its cheapest form,
- Fast standard interconnect (e.g., Myrinet and Infiniband) with high bandwidth (Gigabits/sec) and low latency

Cluster

- Set of independent server nodes interconnected to share resources and form a single system
  - “clustered” resources: disk or software such as data management services
  - off-the-shelf components: PC components, SMP-s, ...
  - Interconnect: local network, fast standard interconnects for clusters
  - Compared to a distributed system: geographically concentrated and made of homogeneous nodes
- There are two main technologies to share disks in a cluster: network-attached storage (NAS) and storage-area network (SAN).
  - NAS is a dedicated device to shared disks over TCP/IP and NFS (low throughput)
  - SAN gives similar functionality with lower-level interface
  - Block-based protocol: easier to manage cache consistency (block-based)
  - SAN provides high data throughput and can scale up to large numbers of nodes

Comparison

- SN cluster can yield best cost/performance and extensibility
  - But adding or replacing cluster nodes requires disk and data reorganization
- SD cluster avoids such reorganization but requires disks to be globally accessible by the cluster nodes
- Small configuration (20P): SM can provide the highest performance because of better load balancing
- Shared-disk and shared-nothing architectures outperform shared-memory in terms of extensibility.
- Some years ago, shared-nothing was the only choice for high-end systems.
- Recent progress in disk connectivity technologies such as SAN make SD a viable alternative
- SD is now the preferred architecture for OLTP applications
- OLAP databases that are typically very large and mostly read-only, SN is used
- NUMA and cluster, can combine the efficiency and simplicity of SM and the extensibility and cost of either SD or SN.
- Using standard PCs and interconnects, clusters provide a better cost/ performance ratio, and, using SN, they can scale up to very large configurations

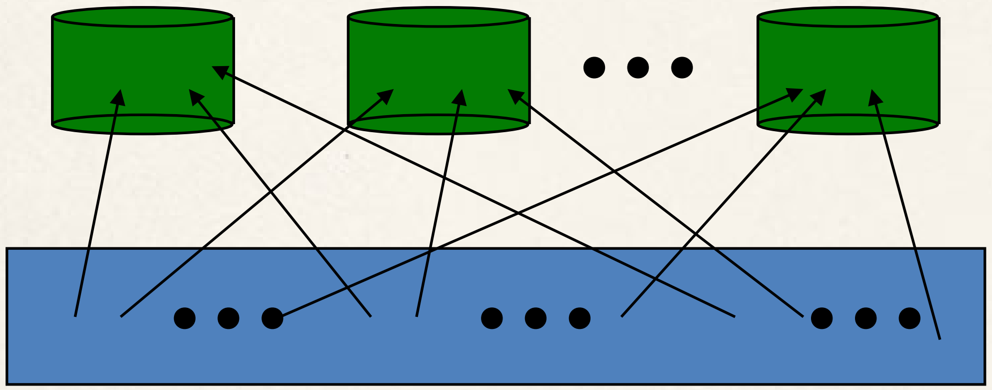
Parallel DBMS Techniques

- Data placement
  - Physical placement of the DB onto multiple nodes
  - Static vs. Dynamic
- Parallel data processing
  - Select is easy
  - Join (and all other non-select operations) is more difficult
- Parallel query optimization
  - Choice of the best parallel execution plans
  - Automatic parallelization of the queries and load balancing
- Transaction management
  - Similar to distributed transaction management

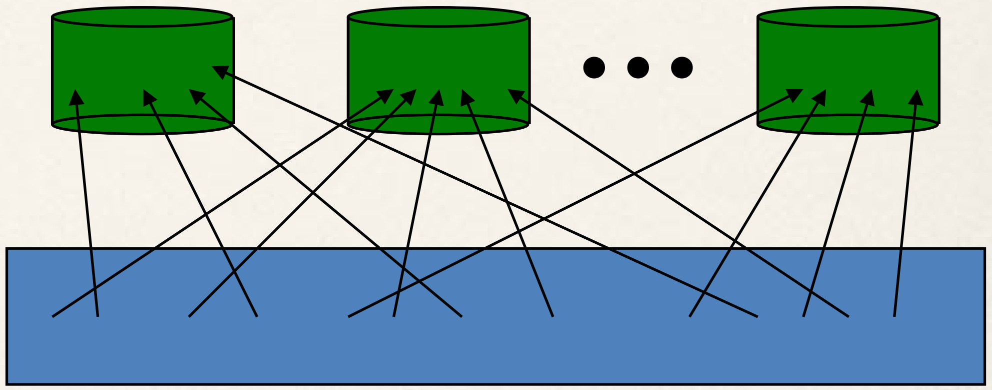
Data Partitioning

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

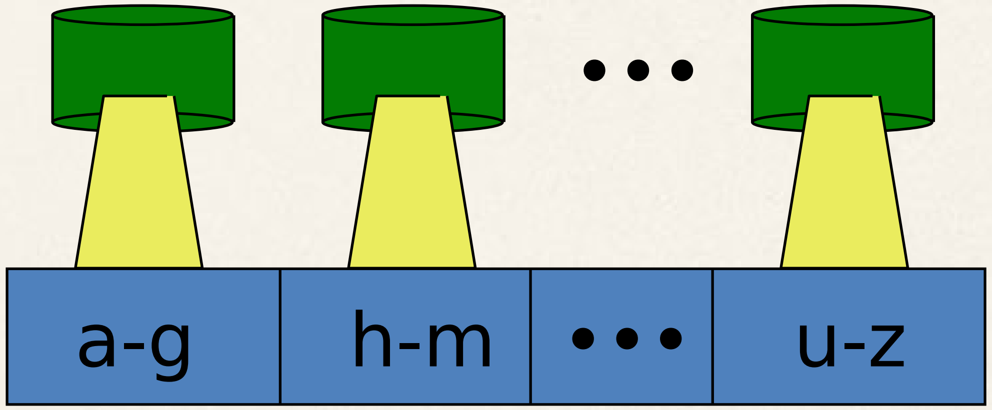
Partitioning Schemes



Round-Robin



Hashing



Interval

Variable partitioning

- Compromise between clustering and full partitioning
  - Full partitioning has obvious performance advantages
  - Highly parallel execution might cause a serious performance overhead
  - Full partitioning is not appropriate for small relations
- Number of nodes over which a relation is fragmented, is a function of the size and access frequency of the relation
  - Changes in data distribution may result in reorganization
  - Periodic reorganizations for load balancing are essential
  - Such reorganizations should remain transparent to compiled programs that run on the database server
  - Compiled programs should remain independent of data location

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

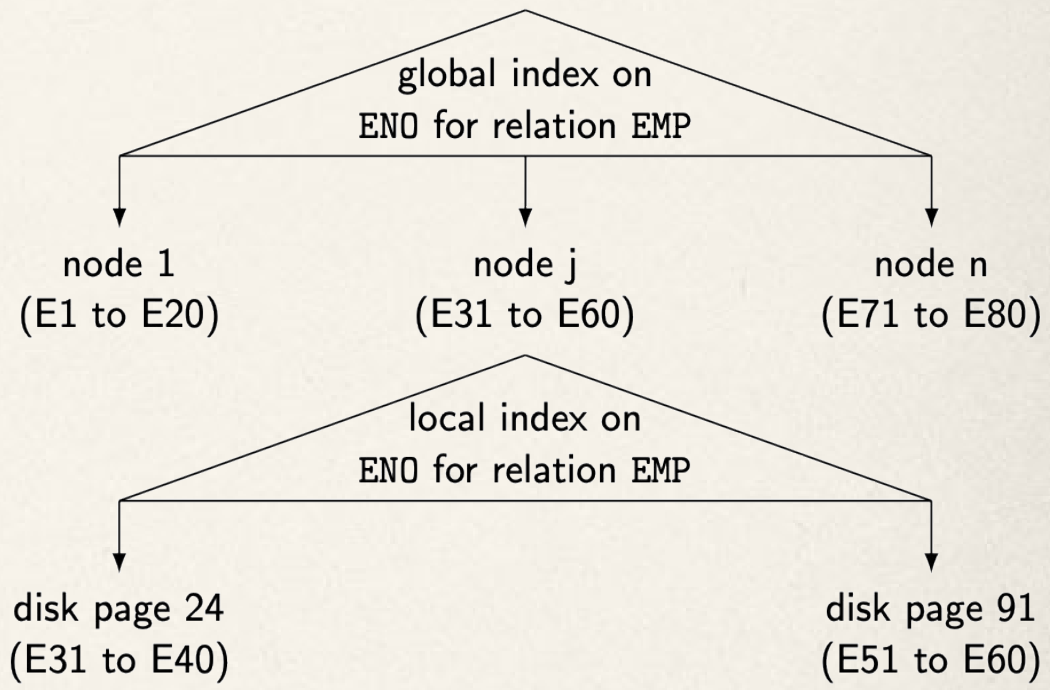
Node	1	2	3	4
Primary copy	R_1	R_2	R_3	R_4
Backup copy		$r_{1.1}$	$r_{1.2}$	$r_{1.3}$
	$r_{2.3}$		$r_{2.1}$	$r_{2.2}$
	$r_{3.2}$	$r_{3.2}$		$r_{3.1}$

Chained Partitioning

Node	1	2	3	4
Primary copy	R_1	R_2	R_3	R_4
Backup copy	r_4	r_1	r_2	r_3

Placement Directory

- Performs two functions
  - F_1 (relname, placement attval) = lognode-id
  - F_2 (lognode-id) = phynode-id
- The global index indicates the placement of a relation onto a set of nodes.
  - Major clustering on the relation name and a minor clustering on some attribute of the relation.
  - The index structure can be based on hashing or on a B-tree like organization.
  - B-tree allows range queries.
- In addition, each node has its local index (to access disk pages)



Parallel Query Processing

- Transform queries into execution plans that can be efficiently executed in parallel

  Exploiting parallel data placement and the various forms of parallelism offered by high-level queries

1) Forms of query parallelism

2) Basic parallel algorithms for data processing

Query Parallelism

- Inter-query parallelism

  Parallel execution of multiple queries generated by concurrent transactions, in order to increase the transactional throughput.

- Intra-query parallelism

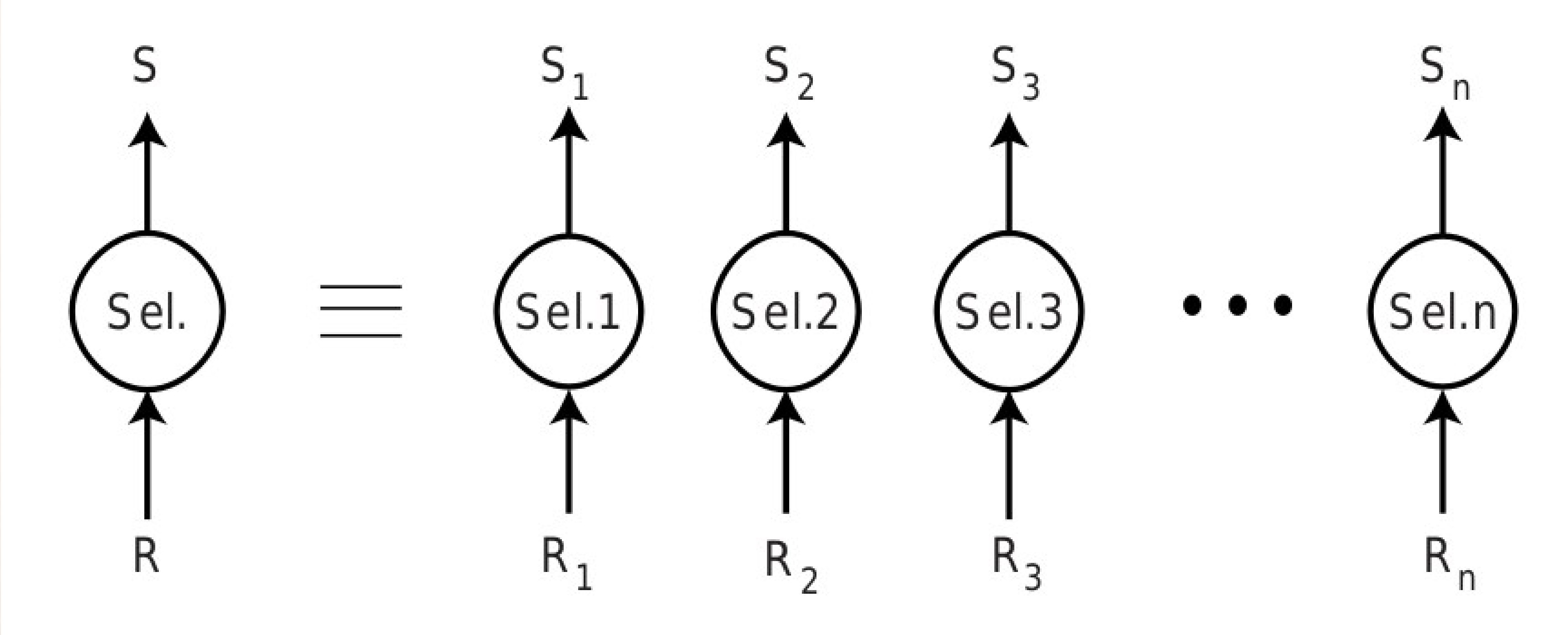
  Within a query: inter-operator and intra-operator parallelism

  Inter-operator parallelism is obtained by executing in parallel several operators of the query tree on several processors

  Intra-operator parallelism, the same operator is executed by many processors, each one working on a subset of the data

Intra-operator Parallelism

- Decomposition of one operator in a set of independent sub-operators, called operator instances
  - Static and/or dynamic partitioning of relations
    - Each operator instance processes one relation partition, also called a bucket
    - Operator decomposition frequently benefits from the initial partitioning of the data

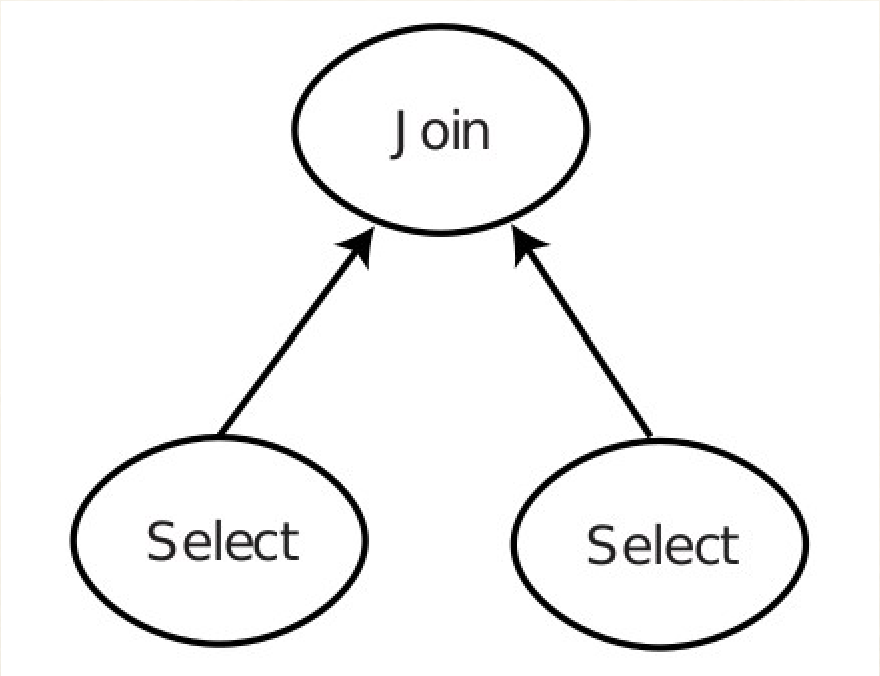


Intra-operator Parallelism

- If the relation is partitioned on the select attribute, partitioning properties can be used to eliminate some select instances
- In order to have independent joins, each bucket of the first relation R may be joined to the entire relation S
    - S needs to be broadcast to each site of R buckets (inefficient!)
- If R and S are partitioned by hashing on the join attribute and if the join is an equijoin, then we can partition the join into independent joins
- Partitioning function (hash, range, round robin) is independent of the local algorithm (e.g., nested loop, hash, sort merge) used to process the join operator

Inter-operator Parallelism

- Two forms of inter-operator parallelism can be exploited
  - Pipeline parallelism, several operators with a producer-consumer link are executed in parallel
    - Advantage: intermediate result is not materialized
  - Independent parallelism is achieved when there is no dependency between the operators that are executed in parallel



Join Processing

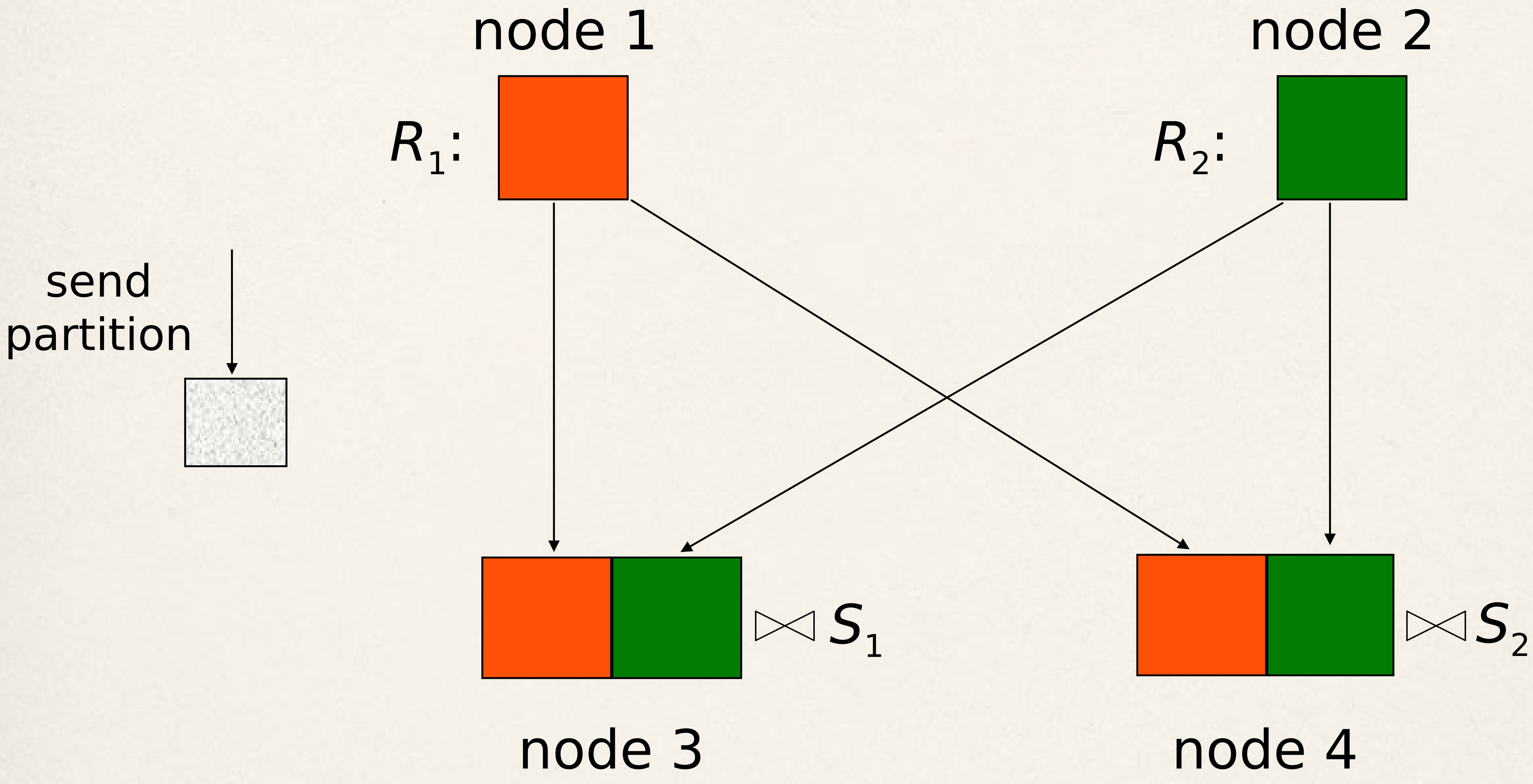
- Three basic algorithms for intra-operator parallelism
  - Parallel nested loop join: no special assumption
  - Parallel associative join: one relation is declustered on join attribute and equi-join
  - Parallel hash join: equi-join
- They also apply to other complex operators such as duplicate elimination, union, intersection, etc. with minor adaptation

Parallel Nested Loop Join

Algorithm 14.1: PNL Algorithm

Input: $R_1, R_2, \ldots, R_m$: fragments of relation R;
$S_1, S_2, \ldots, S_n$: fragments of relation S;
JP: join predicate
Output: $T_1, T_2, \ldots, T_n$: result fragments
begin
    for *i from* 1 *to m in parallel* **do** {send R entirely to each S-node}
        send R_i to each node containing a fragment of S
    for *j from* 1 *to n in parallel* **do** {perform the join at each S-node}
        $R \leftarrow \bigcup_{i=1}^{m} R_i$; {receive R_i from R-nodes; R is fully replicated at S-nodes}
        $T_j \leftarrow R \bowtie_{JP} S_j$
end

Parallel Nested Loop Join



Parallel Nested Loop Join

- PNL composes the Cartesian product of relations R and S in parallel
  - Arbitrarily complex join predicates may be supported
- Join result is produced at the S-nodes (Distributed INGRES)
- In the first phase, each fragment of R is sent and replicated at each node containing a fragment of S (there are n such nodes)
- In the second phase, each Sj-node receives relation R entirely, and locally joins R with fragment Sj .
  - This phase is done in parallel by n nodes
  - Depending on the local join algorithm, join processing may or may not start as soon as data are received (NL algorithm)

Parallel Associative Join

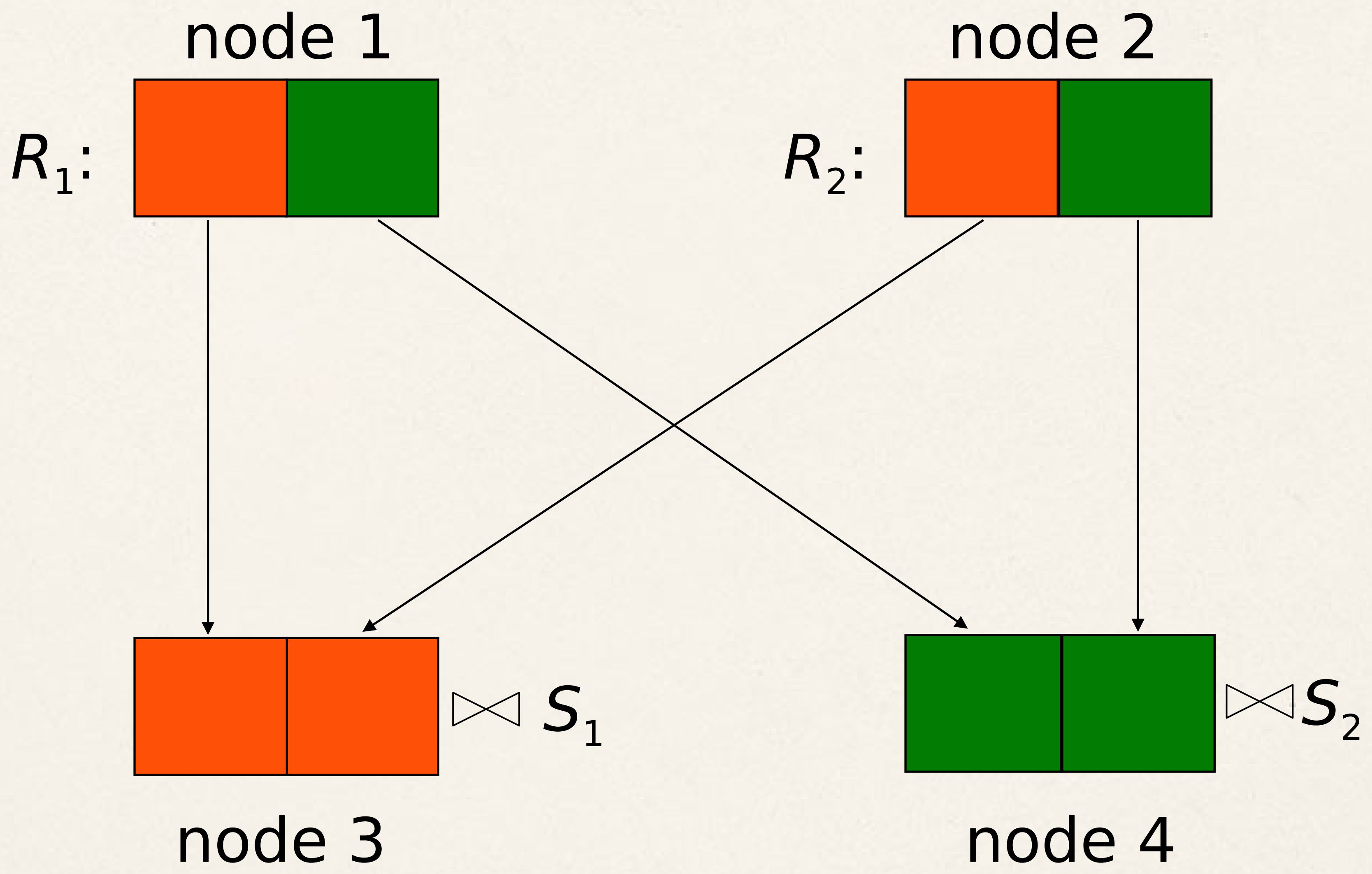
Algorithm 14.2: PAJ Algorithm

Input: $R_1, R_2, \ldots, R_m$: fragments of relation R;
$S_1, S_2, \ldots, S_n$: fragments of relation S;
JP: join predicate
Output: $T_1, T_2, \ldots, T_n$: result fragments
begin

    {we assume that JP is $R.A = S.B$ and relation S is fragmented according to the function $h(B)$}
    for *i from* 1 *to m in parallel* **do** {send R associatively to each S-node}
        $R_{ij} \leftarrow$ apply $h(A)$ to R_i $(j = 1, \ldots, n)$
    for *j from* 1 *to n in parallel* **do**
        send R_{ij} to the node storing S_j
    for *j from* 1 *to n in parallel* **do** {perform the join at each S-node}
        $R_j \leftarrow \bigcup_{i=1}^{m} R_{ij}$; {receive only the useful subset of R}
        $T_j \leftarrow R_j \bowtie_{JP} S_j$

end

Parallel Associative Join



Parallel Associative Join

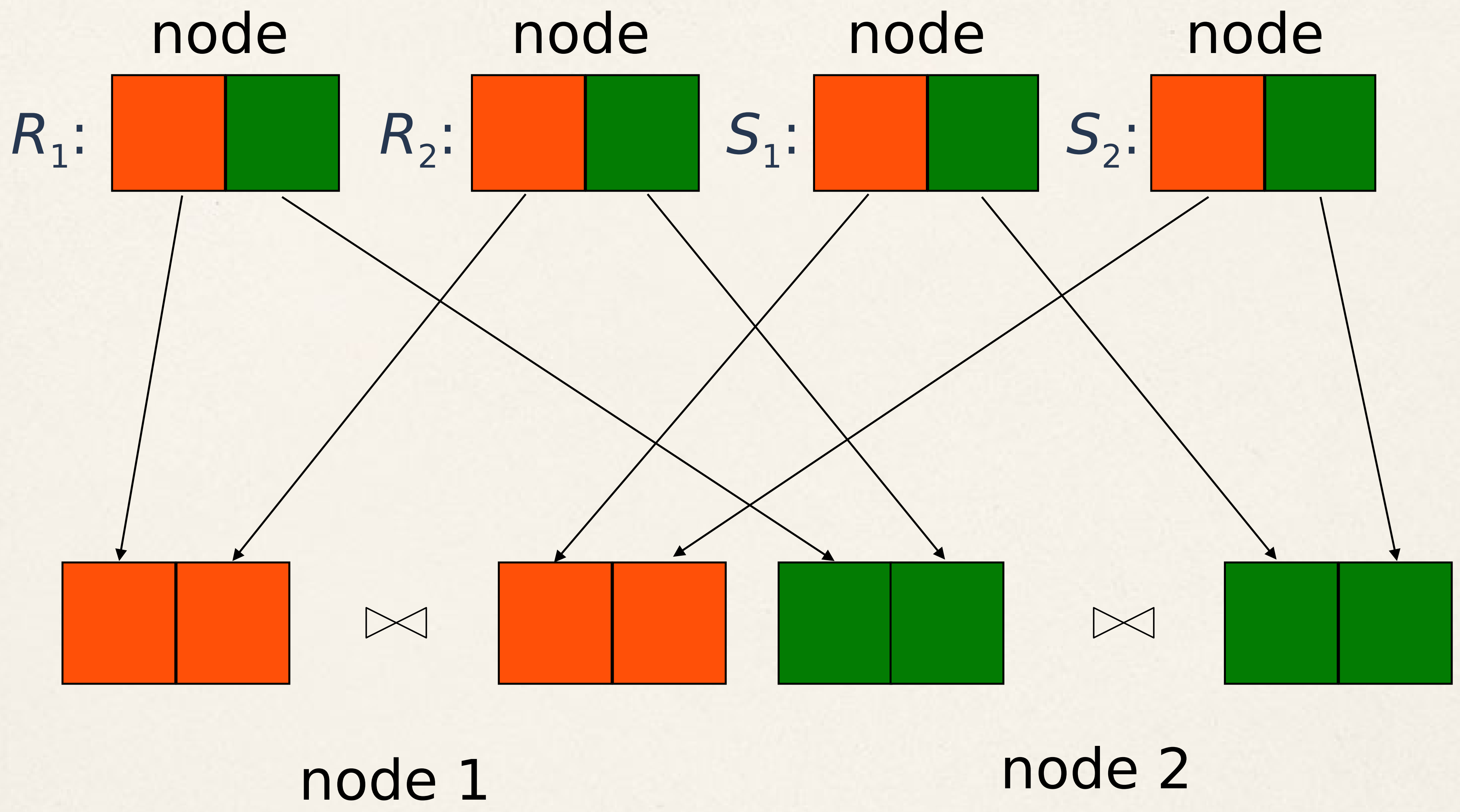
- In the first phase, relation R is sent associatively to the S-nodes based on the hash function h applied to attribute A
  - Tuple of R with hash value v is sent only to the S-node that contains tuples with hash value v
  - Tuples of R get distributed but not replicated across the S-nodes
- In the second phase, each Sj-node receives in parallel from the different R-nodes the relevant subset of R (i.e., Rj) and joins it locally with the fragments Sj
  - Rj joined locally with the fragments Sj
  - It depends on the local join if Sj-node starts immediately

Parallel Hash Join

Algorithm 14.3: PHJ Algorithm

Input: $R_1, R_2, \ldots, R_m$: fragments of relation R ;
$S_1, S_2, \ldots, S_n$: fragments of relation S ;
JP: join predicate $R.A = S.B$;
h: hash function that returns an element of $[1, p]$
Output: $T_1, T_2, \ldots, T_p$: result fragments
begin
  {Build phase}
  for *i from* 1 *to m in parallel* **do**
    $R_{ij} \leftarrow$ apply $h(A)$ to R_i $(j = 1, \ldots, p)$; {hash R on A)} ;
    send R_{ij} to node j
  for *j from* 1 *to p in parallel* **do**
    $R_j \leftarrow \bigcup_{i=1}^{m} R_{ij}$ {receive from R-nodes}
  {Probe phase}
  for *i from* 1 *to n in parallel* **do**
    $S_{ij} \leftarrow$ apply $h(B)$ to S_i $(j = 1, \ldots, p)$; {hash S on B)} ;
    send S_{ij} to node j
  for *j from* 1 *to p in parallel* **do** {perform the join at each of the p nodes}
    $S_j \leftarrow \bigcup_{i=1}^{n} S_{ij}$; {receive from S-nodes} ;
    $T_j \leftarrow R_j \bowtie_{JP} S_j$
end

Parallel Hash Join



Parallel Hash Join

- Generalization of the parallel associative join algorithm

    Does not require any particular partitioning of the operand relations

    The basic idea is to partition relations R and S into the same number p of mutually exclusive sets

    Each individual join (Ri JOIN Si) is done in parallel, and the join result is produced at p nodes

- A build phase and a probe phase

    The build phase hashes R on the join attribute, sends it to the target p nodes that build a hash table for the incoming tuples

The probe phase sends S assThe probe phase sends S associatively to the target p nodes that probe the hash table for each incoming tupleociatively to the target p nodes that probe the hash table for each incoming tuple

    After the hash tables have been built for R, the S tuples can be sent and processed in pipeline by probing the hash tables

Example: Parallel Hash Join

- R1 and R2 are fragments of a table R; S1 and S2 be fragments of S.
- Show the steps in the computation of the distributed hash join R $\bowtie_A$ S.
- Hash function h(A) = (A mod 2)+N.
- Set N so that Sites 1-4 are used.

R1 — Site1

A	B
2	10
6	14

S1 — Site3

A	C
3	55
2	44

R2 — Site2

A	B
5	17
4	22

S2 — Site4

A	C
5	48
2	76

T1 — SiteN

A	B
2	10
6	14
4	22

T2 — SiteN

A	C
2	44
2	76

SiteN

A	B	C
2	10	44
2	10	76

T3 — SiteN+1

A	B
5	17

T4 — SiteN+1

A	C
3	55
5	48

SiteN+1

A	B	C
5	17	48

Parallel Hash Join

- Generalization of the parallel associative join algorithm
    - Does not require any particular partitioning of the operand relations
    - The basic idea is to partition relations R and S into the same number p of mutually exclusive sets
    - Each individual join (Ri JOIN Si) is done in parallel, and the join result is produced at p nodes
- A build phase and a probe phase
    - The build phase hashes R on the join attribute, sends it to the target p nodes that build a hash table for the incoming tuples

The probe phase sends S assThe probe phase sends S associatively to the target p nodes that probe the hash table for each incoming tupleociatively to the target p nodes that probe the hash table for each incoming tuple

- After the hash tables have been built for R, the S tuples can be sent and processed in pipeline by probing the hash tables

Parallel Query Optimization

- Parallel query optimization exhibits similarities with distributed query processing.
- Taking advantage of both
  - intra-operator and inter-operator parallelism.
- A parallel query optimizer can be seen as three components:
  - a search space, a cost model, and a search strategy.

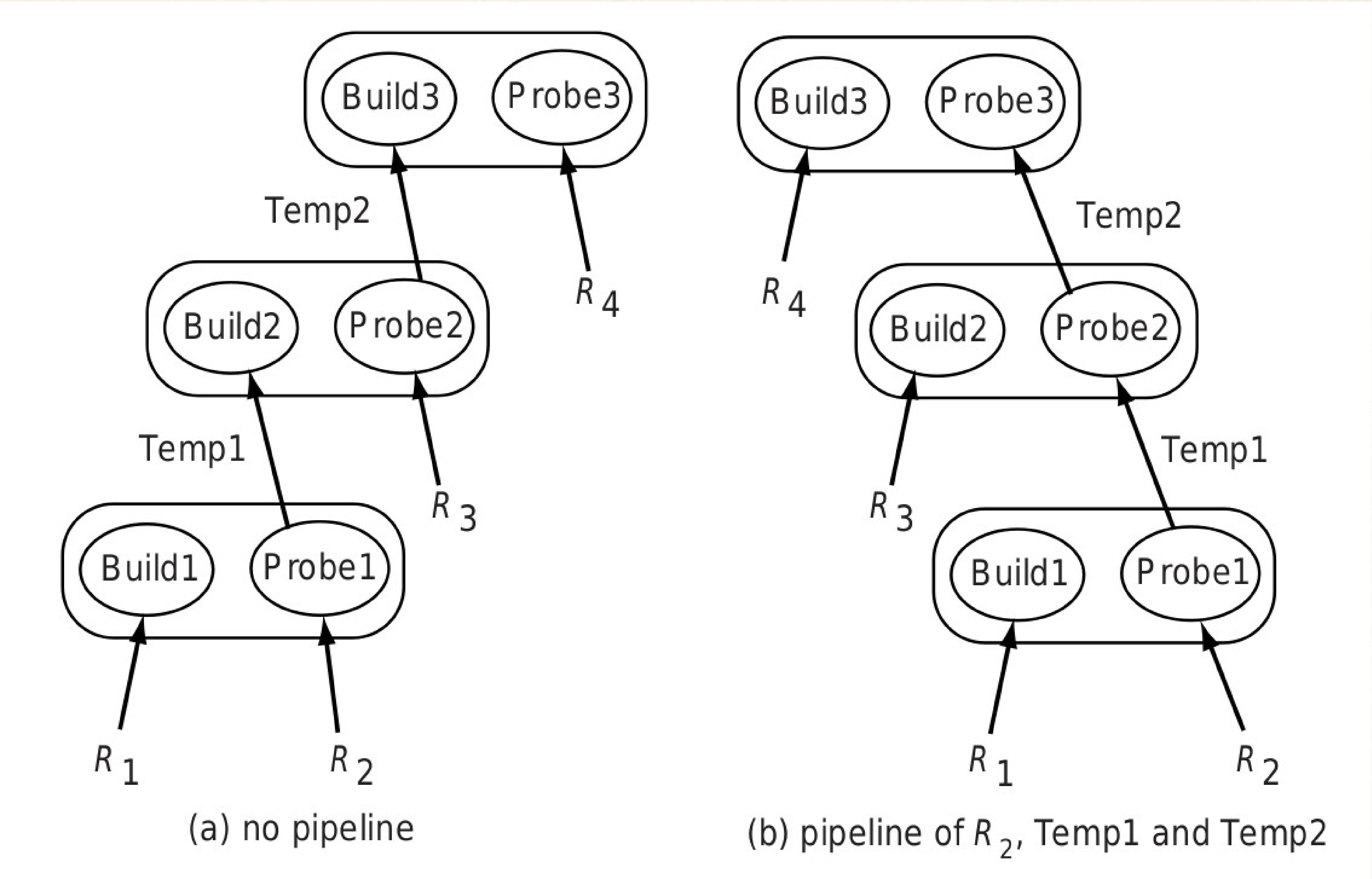
Parallel Query Optimization

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

Search space

- Execution plans are abstracted by means of operator trees
  - Annotations indicate additional execution aspects
  - Algorithm of each operator
  - Pipelined execution (flow of tuples, intermediate results not materialized)
  - One operand is stored (e.g., parallel hash join algorithm in the build phase)
  - Pipeline and stored annotations constrain the scheduling of execution plans
  - Splitting an operator tree into non-overlapping sub-trees, corresponding to execution phases

Two hash-join trees with a different scheduling

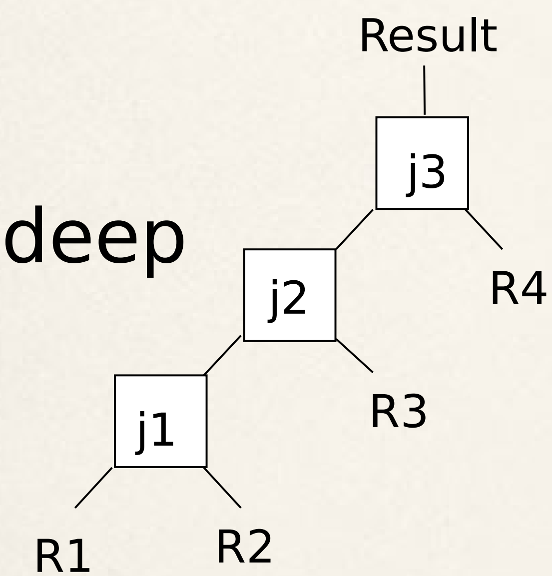


Search space

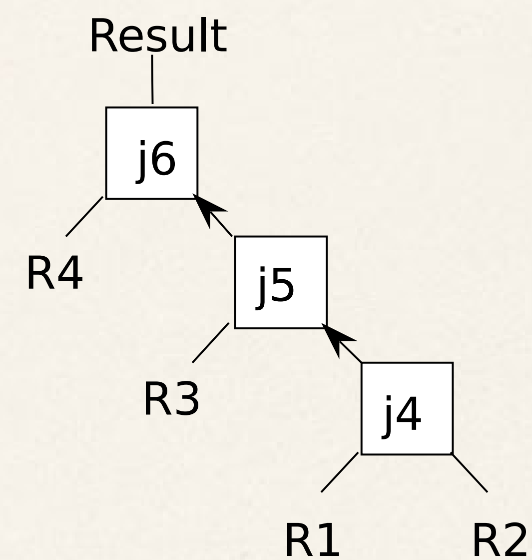
- Set of nodes where a relation is stored is called its home.
  - The home of an operator is the set of nodes where it is executed
  - The home of an operator must be the home of its operands in order for the operator to access its operands
  - For binary operators such as join, this might imply repartitioning one of the operands.
  - Annotations to indicate repartitioning.
- Four operator trees that represent execution plans for a three-way join.
  - Linear or bushy trees
  - Right-deep trees express full pipelining while left-deep trees express full materialization of all intermediate results

Execution Plans as Operator Trees

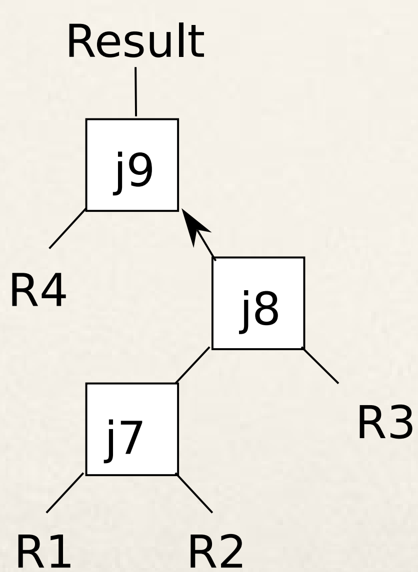
Left-deep



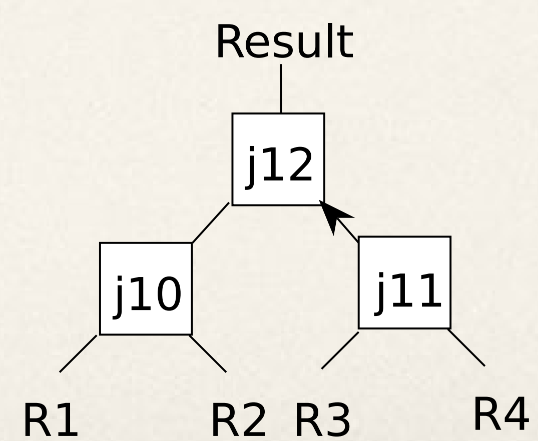
Right-deep



Zig-zag



Bushy



Search space

- Long right-deep trees are more efficient then corresponding left-deep

  … but tend to consume more memory to store left-hand side relations

- Bushy trees are the only ones to allow independent parallelism

  … and some pipeline parallelism

  Independent parallelism is useful when the relations are partitioned on disjoint homes.

- Zigzag trees are intermediate formats between left-deep and right-deep trees

  Sometimes outperform right-deep trees due to a better use of main memory

  Use right-deep or zigzag trees when relations are partially fragmented on disjoint homes and intermediate relations are rather large.

  When intermediate relations are small, pipelining is not very efficient because it is difficult to balance the load between the pipeline stages.

Search Strategy

- Research problems
  - No ad-hoc solutions and dynamic optimization?
  - Improve the cost function; it is always an estimation
  - Problems with skew; hard to find good solutions
  - Properties of cost function; well-designed cost function
- Exhaustive search
  - Possible for small number of joins in relational parallel systems
  - Exhaustive join reordering useful for simple and very specific query languages (e.g., document search)
- Dynamic programming
  - Many instances of the dynamic programming
  - Dynamic programming “by the book”
    - Very complex environment; hard to nail down a clean implementation

Search Strategy

- Bottom-up dynamic programming
  - Start with the optimal access to relations and build the plan in a bottom-up fashion
- Memoization
  - A variant of dynamic programming storing best approaches for subqueries
- Problems with cost estimation function
  - Cost function is an estimation; very hard to compute precisely
  - Cost function as heuristics
    - Monotonicity of the cost function? Structure, properties of cost function?
  - Open problems?

Search Strategy

- Heuristic-based Optimization
  - See dynamic optimization in distributed databases
  - Push down all selections and projections
  - Select the smallest intermediate result first
  - Select if enough physical memory available
  - More insight in cost function (structure, math.properties, …)
    - Gives more possibilities to use heuristics!

Cost Model

- Cost model is responsible for estimating the cost of a given execution plan.
  - Architecture-dependent and architecture-independent
- Architecture-independent
  - Operator algorithms, e.g., nested loop for join and sequential access for select.
- Architecture-dependent
  - Data repartitioning and memory consumption
- The total time of a plan
  - Add CPU, I/O and communication cost components as in distributed query optimization.

Main Products

Vendor	Product	Architecture	Platforms
IBM	DB2 Pure Scale DB2 Database Partitioning Feature (DPF)	SD SN	AIX on SP Linux on cluster
Microsoft	SQL Server SQL Server 2008 R2 Parallel Data Warehouse	SD SN	Windows on SMP and cluster
Oracle	Real Application Cluster Exadata Database Machine	SD	Windows, Unix, Linux on SMP and cluster
NCR	Teradata	SN Bynet network	NCR Unix and Windows
Oracle	MySQL	SN	Linux Cluster