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
- SSD linked to controller (500-800 MB/s)!
- 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*

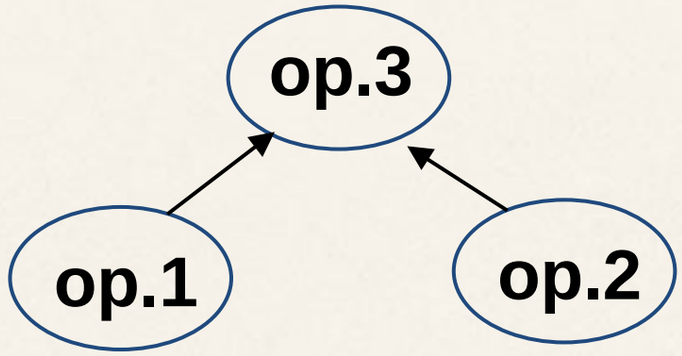
Parallel Data Processing

- 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

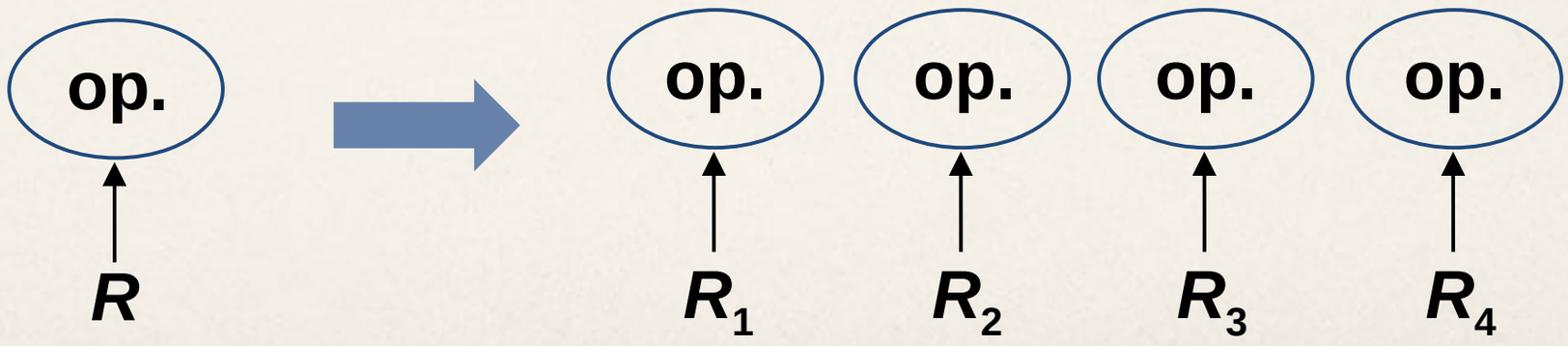
- Inter-operation

  ➡️p operations of the same query in parallel



- Intra-operation

  ➡️The same op in parallel



Parallel DBMS

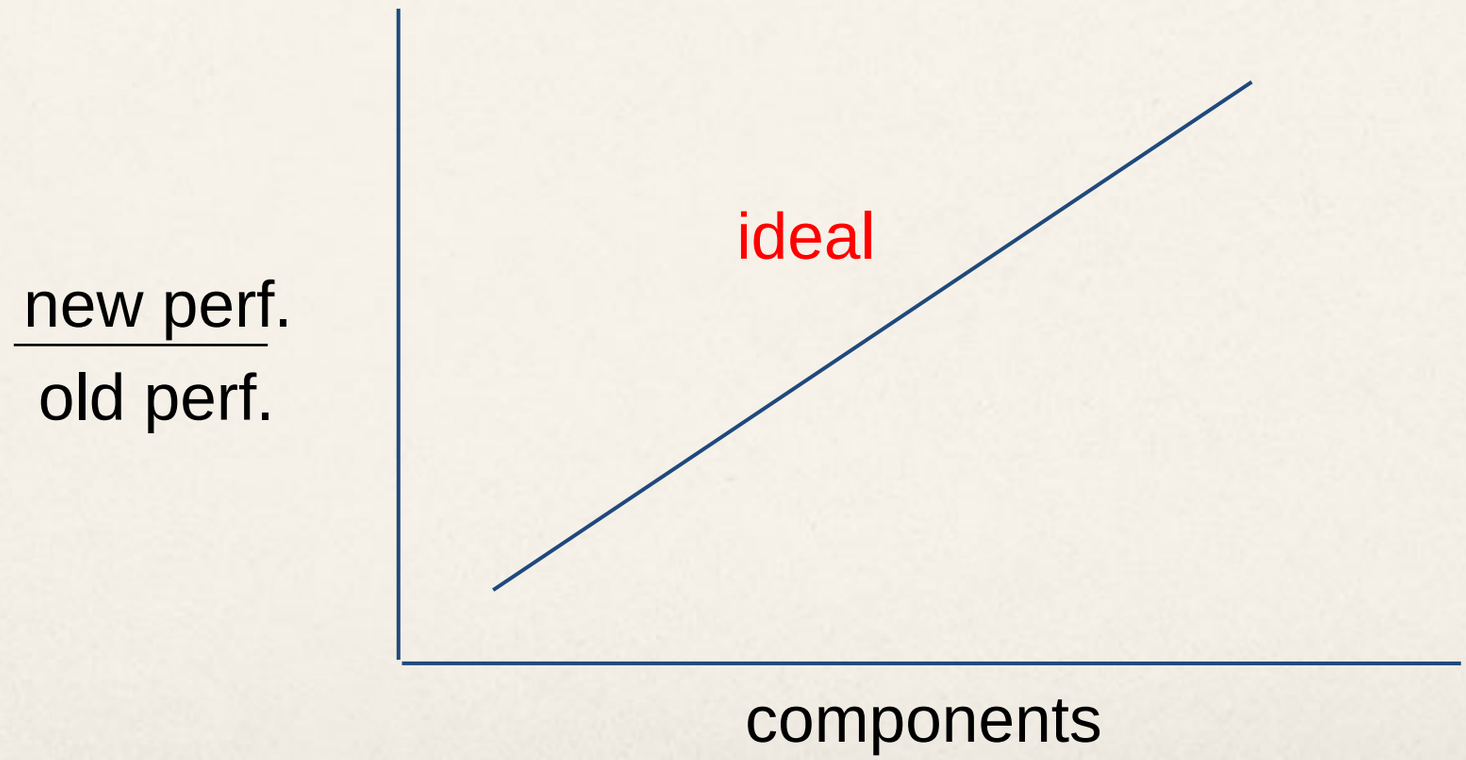
- Loose definition: a DBMS implemented on a tighly coupled multiprocessor
- Alternative extremes
  - ➡️Straighforward porting of relational DBMS (the software vendor edge)
  - ➡️New hardware/software combination (the computer manufacturer edge)
- Naturally extends to distributed databases with one server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

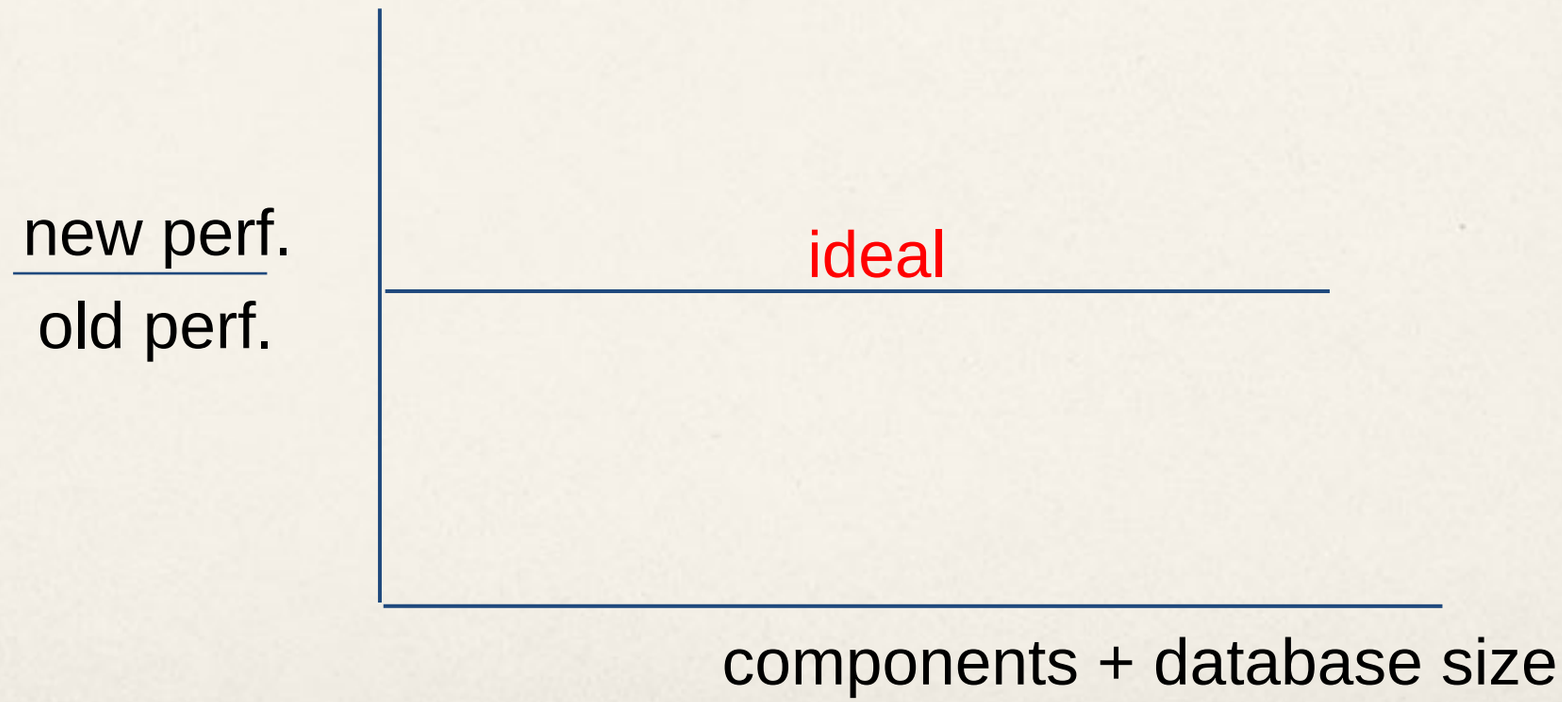
Linear Speed-up

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



Linear Scale-up

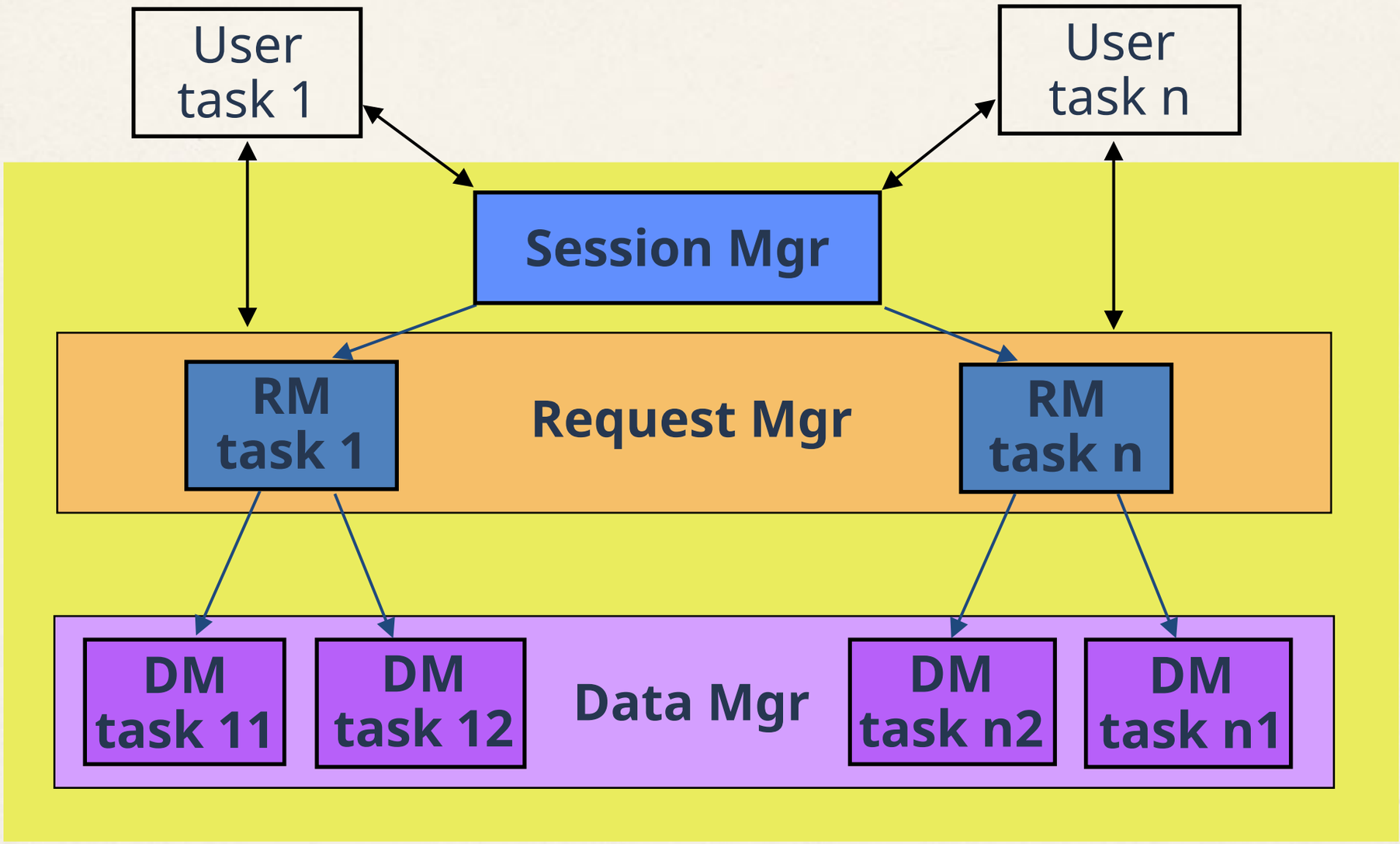
Sustained performance for a linear increase of database size and proportional increase of the system components.



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



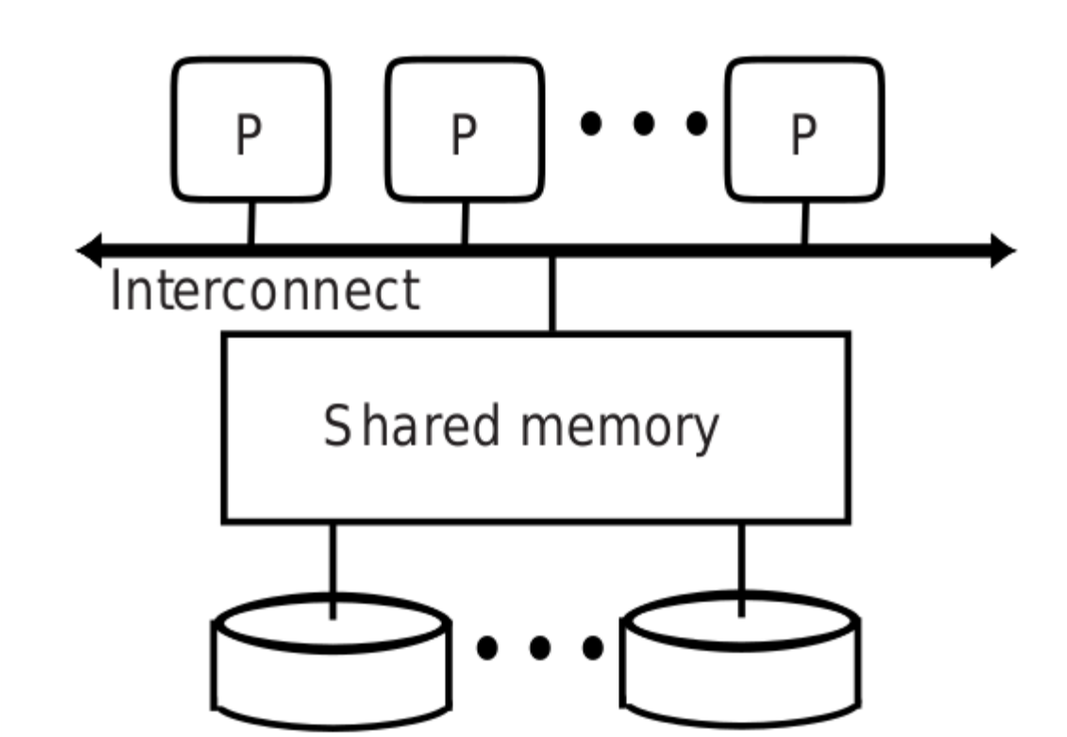
Parallel DBMS Functions

- 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

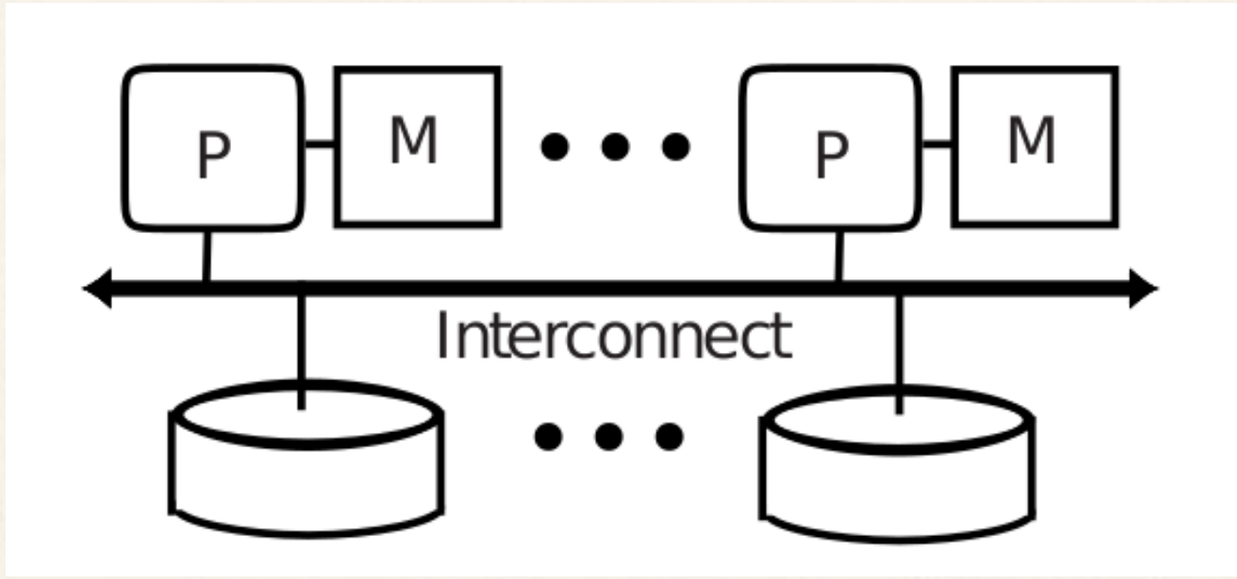


DBMS on symmetric multiprocessors (SMP)
Prototypes: XPRS, Volcano, DBS3
+ Simplicity, load balancing, fast communication
- Network cost, low extensibility

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



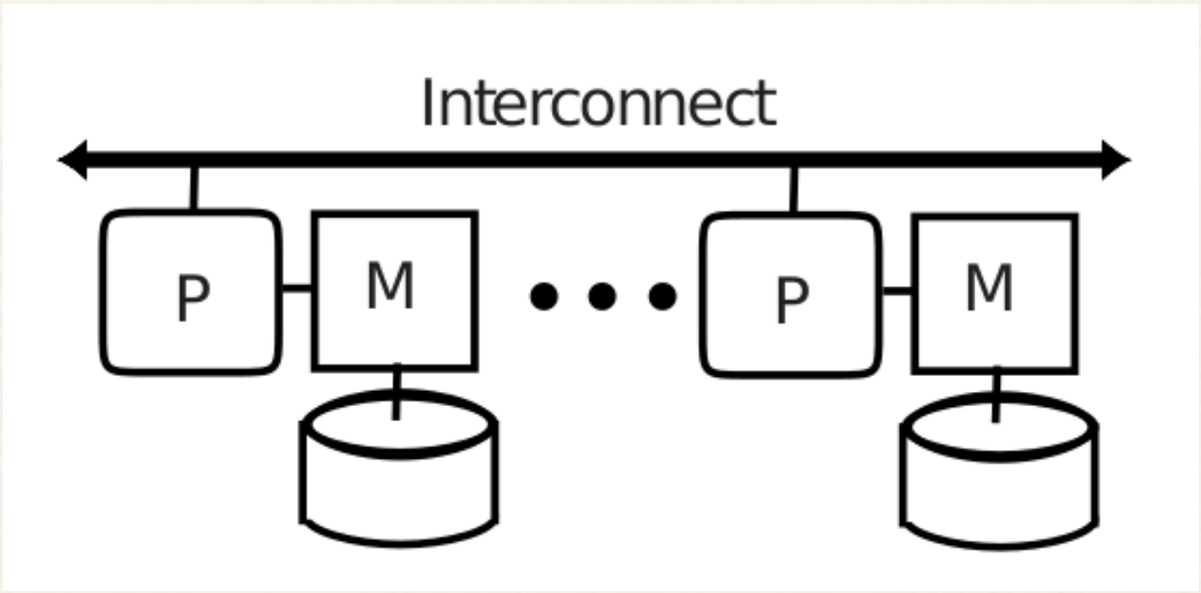
Origins : DEC's VAXcluster, IBM's IMS/VS Data Sharing
Used first by Oracle with its Distributed Lock Manager
Now used by most DBMS vendors

+ 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.
  - ➡️ Network-attached storage (NAS) and storage-area network (SAN)
    - 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 advantages: 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

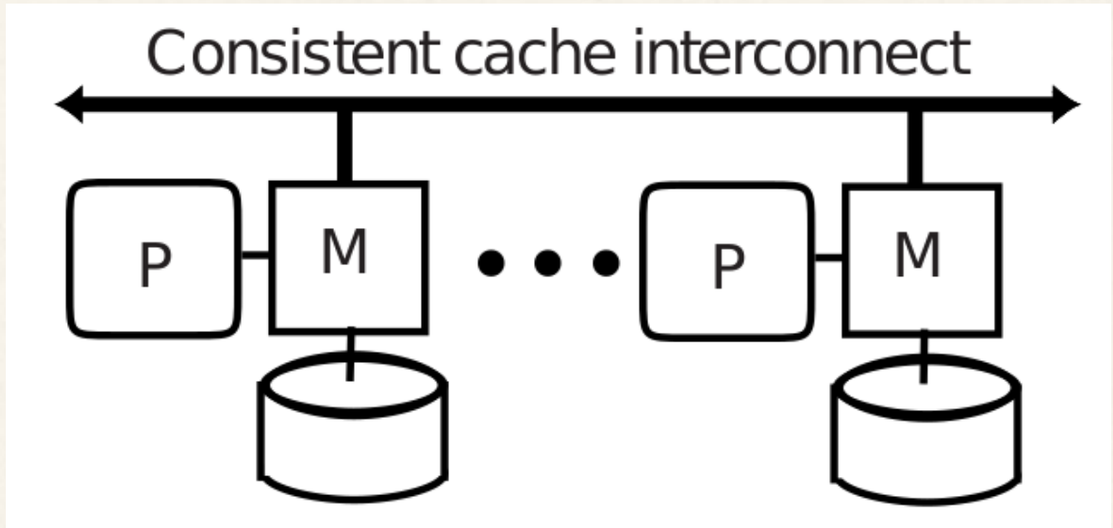
- NUMA = single address space on distributed physical memory

  ➡️Eases application portability

  ➡️Extensibility

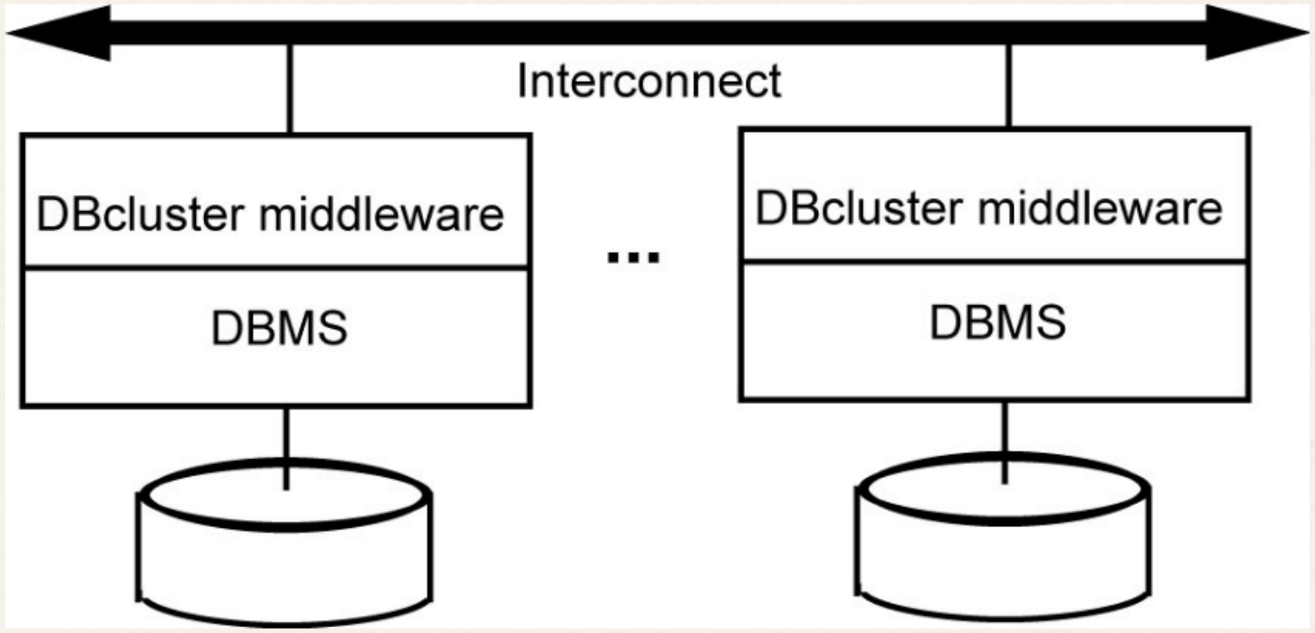
- The most successful NUMA is Cache Coherent NUMA (CC-NUMA)

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., Ethernet, Infiniband, Omni-Path, Fibre Channel) with high bandwidth (up to 400 Gb/s)

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

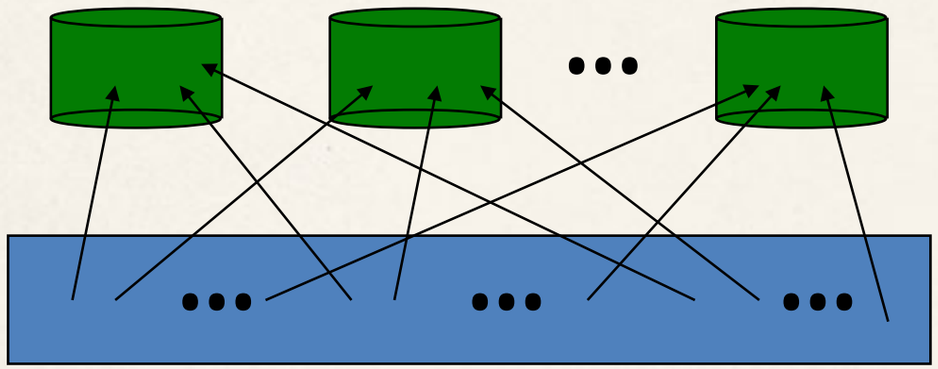
  ➡️B-tree index

  - Supports range queries but large index

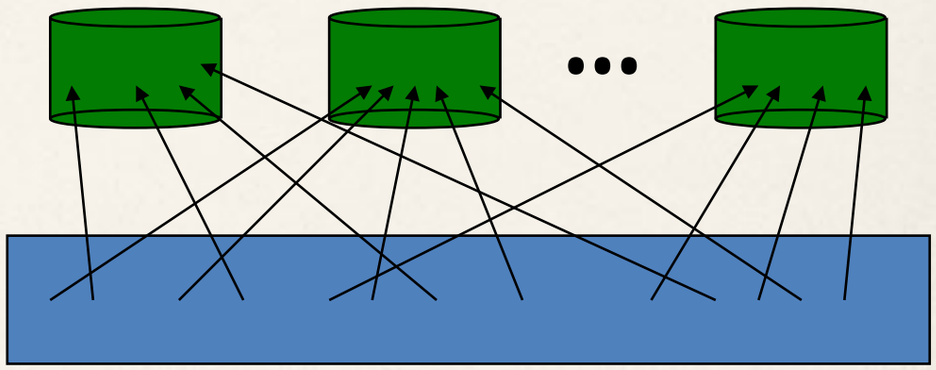
  ➡️Hash function

  - Only exact-match queries but small index

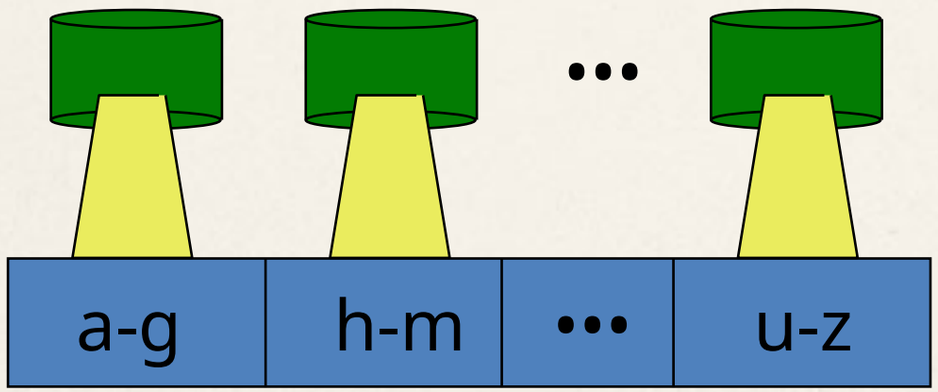
Partitioning Schemes



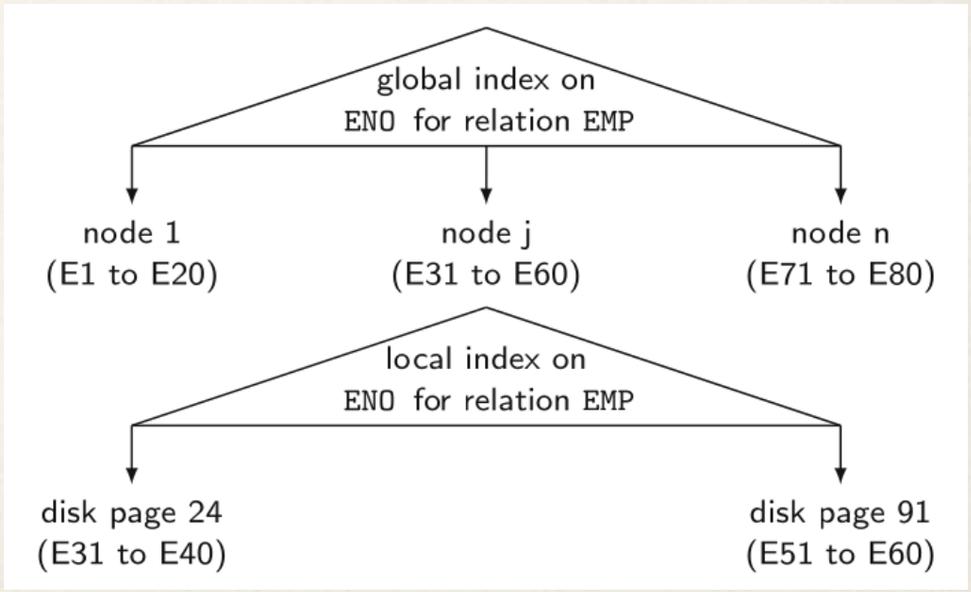
Round-Robin



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

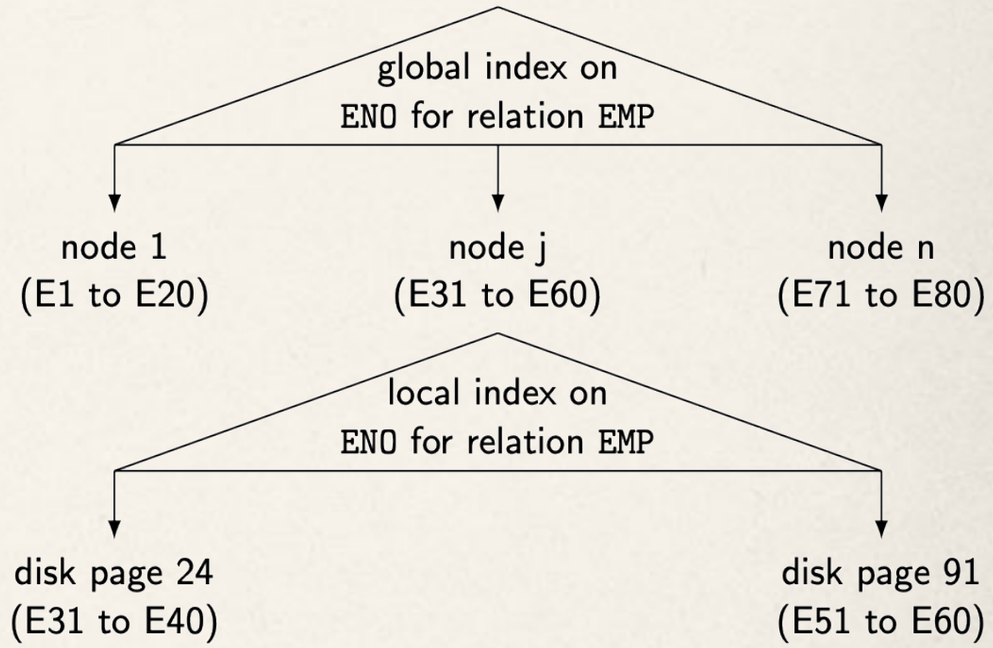
Node	1	2	3	4
Primary copy	R_1	R_2	R_3	R_4
Backup copy	$r_{2.3}$ $r_{3.2}$	$r_{1.1}$ $r_{3.2}$	$r_{1.2}$ $r_{2.1}$	$r_{1.3}$ $r_{2.2}$ $r_{3.1}$

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

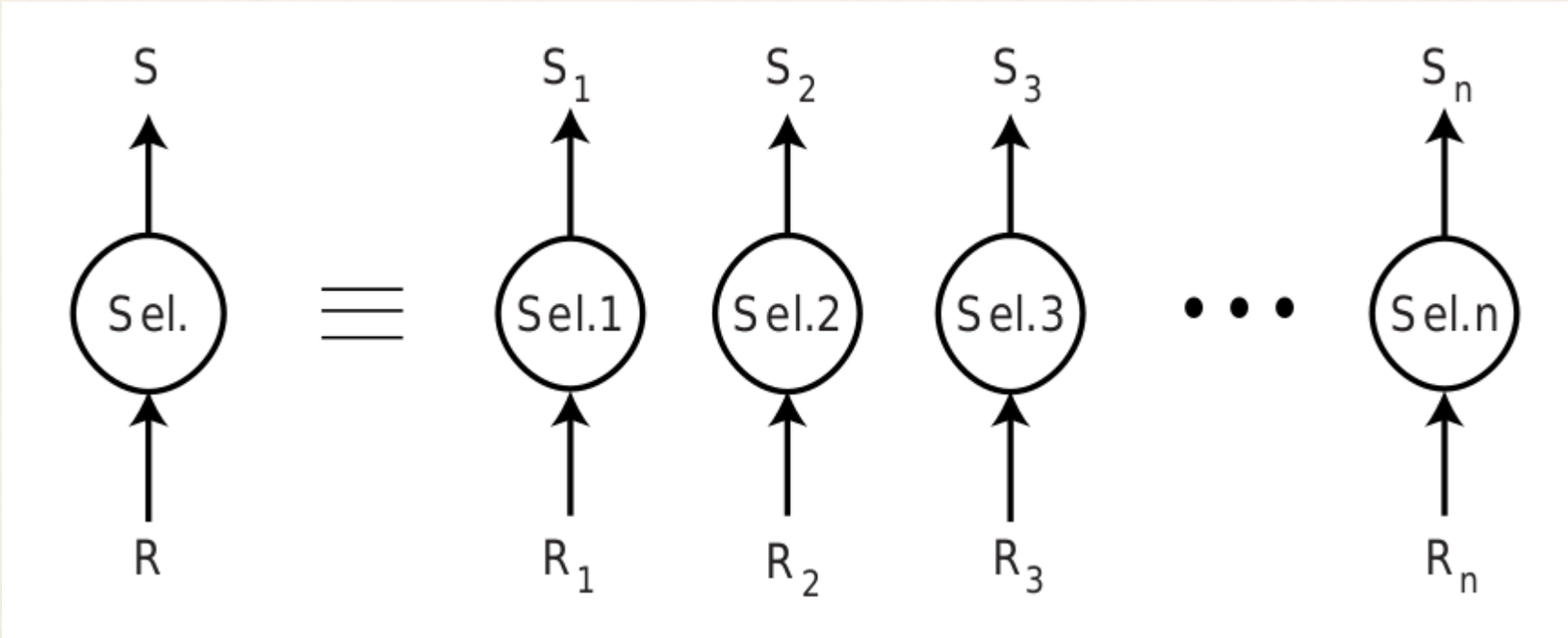
- Inter-query parallelism
  - ➡️ Parallel execution of multiple queries generated by concurrent transactions, in order to increase the transactional throughput.
- Intra-query parallelism
  - ➡️ Within a query: inter-operator and intra-operator parallelism
  - ➡️ Inter-operator parallelism is obtained 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

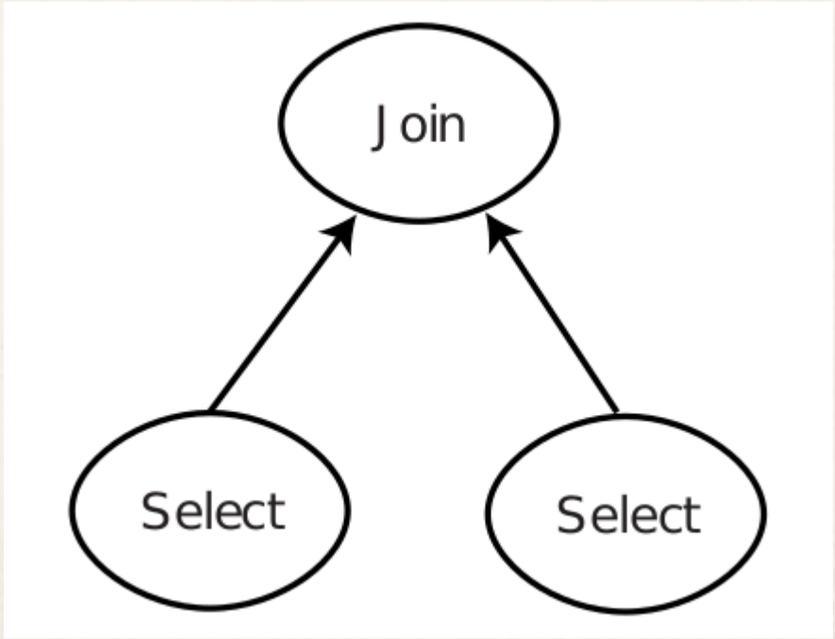
- If the relation is partitioned on the select attribute, partitioning properties can be used to eliminate some select instances
- In order to have independent joins, each bucket of the first relation R may be joined to the entire relation S

  ➡️ 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

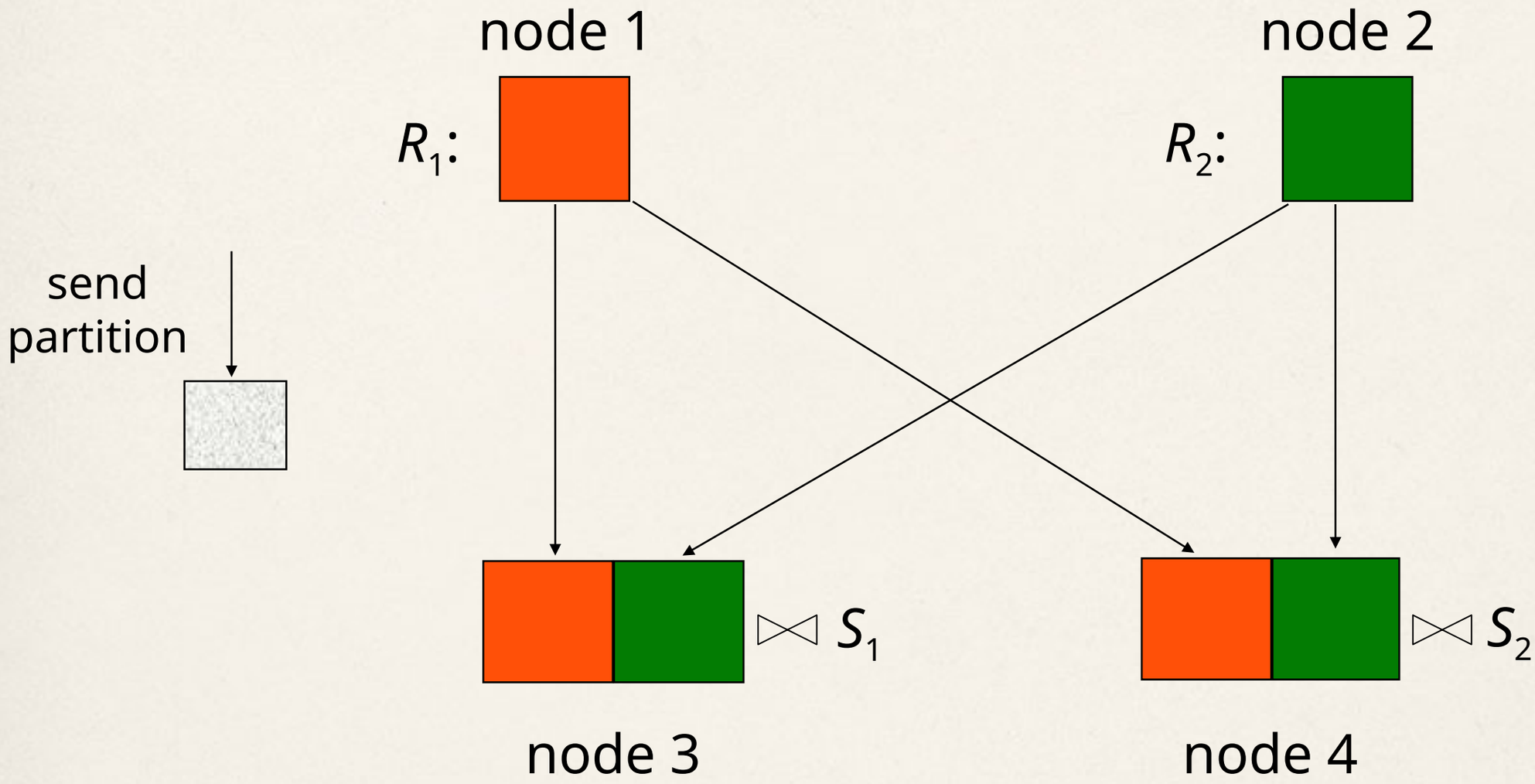
- Three basic algorithms for intra-operator parallelism
  - ➡️Parallel nested loop join: no special assumption
  - ➡️Parallel associative join: one relation is declustered on join attribute and equi-join
  - ➡️Parallel hash join: equi-join
- They also apply to other complex operators such as duplicate 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

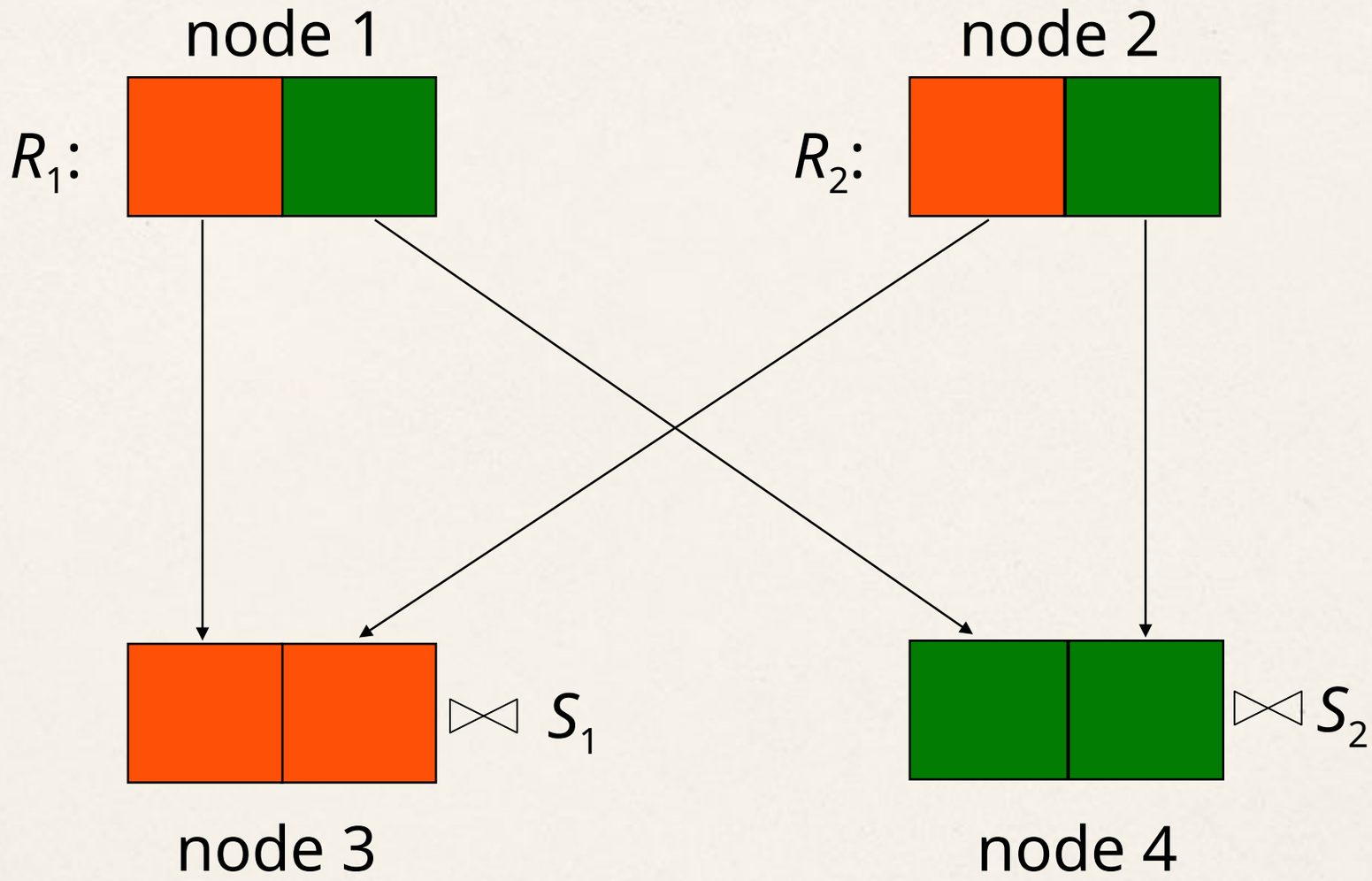
- PNL composes the Cartesian product of relations R and S in parallel
  - ➡️ Arbitrarily complex join predicates may be supported
- Join result is produced at the S-nodes (Distributed INGRES)
- In the first phase, each fragment of R is sent and replicated at each node containing a fragment of S (there 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

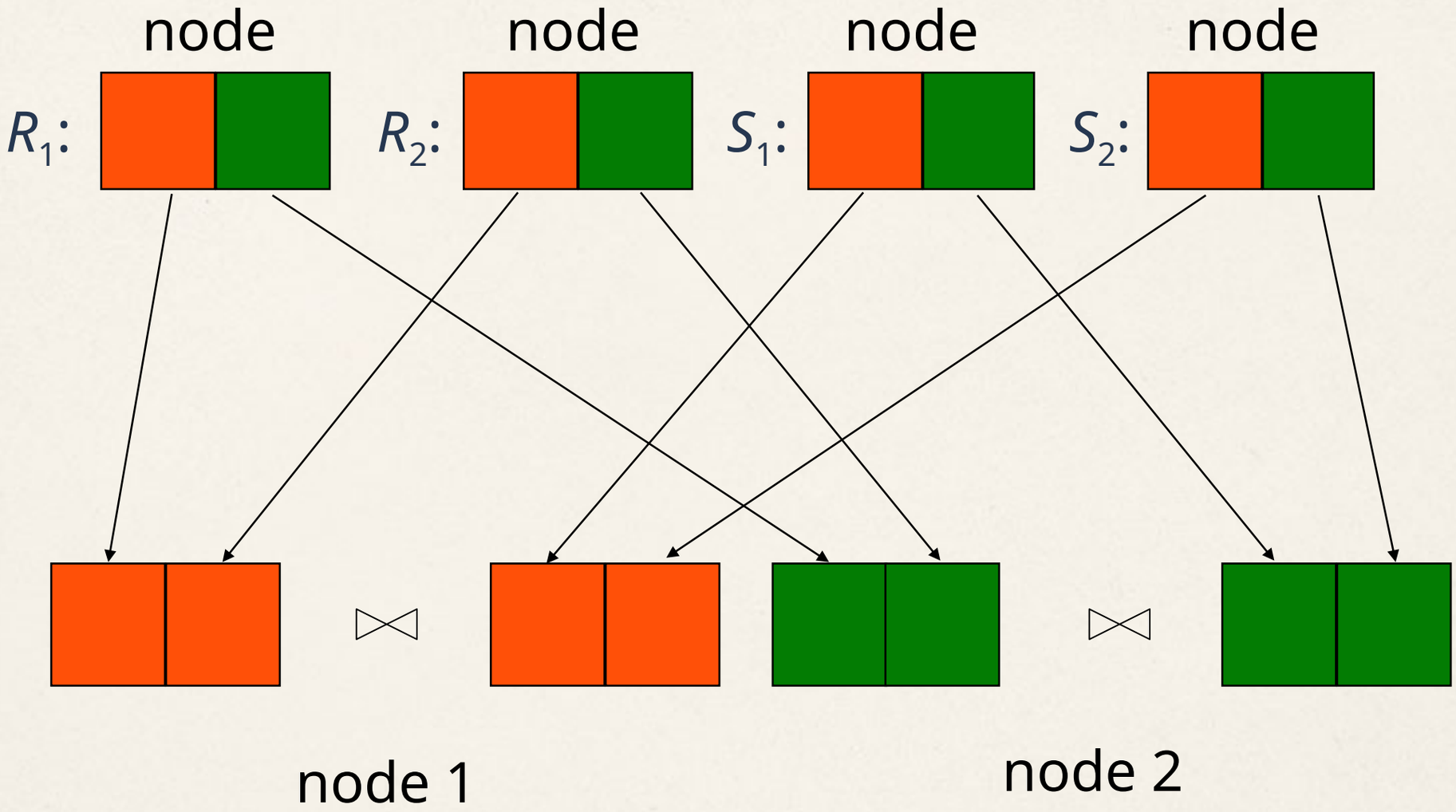
- In the first phase, relation R is sent associatively to the S-nodes based on the hash function h applied to attribute A
  - ➡️ Tuple of R with hash value v is sent only to the S-node that contains tuples with hash value v
  - ➡️ Tuples of R get distributed but not replicated across the S-nodes
- In 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

- Also called (recently) *canonical 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 associatively 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

R2 — Site2

A	B
5	17
4	22

S1 — Site3

A	C
3	55
2	44

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

T3 — SiteN+1

A	B
5	17

T4 — SiteN+1

A	C
3	55
5	48

SiteN

A	B	C
2	10	44
2	10	76

SiteN+1

A	B	C
5	17	48

New hardware

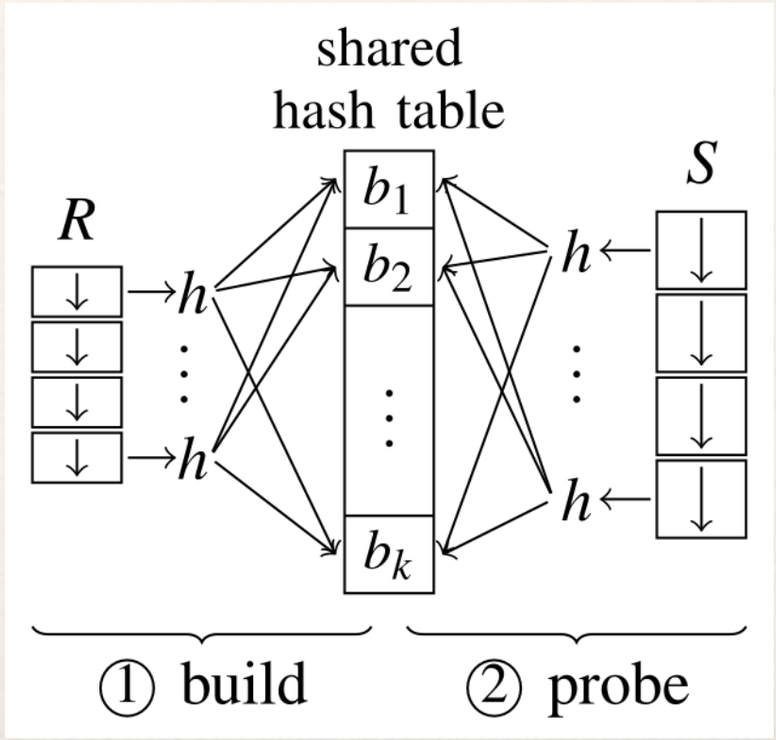
- Modern processors provide parallelism at various levels
  - Instruction parallelism via super scalar execution
  - **Data-level parallelism** by extended support for SIMD; and
  - **Thread-level parallelism** through multiple cores and simultaneous multi-threading (SMT).
- Two main types of join algorithms
  - **Hash Join** (abbr. HJ)
  - **Sort-Merge Join** (abbr. SMJ)
- Lines of arguments
  - Main-memory parallel joins should be hardware-conscious
    - fine tuning the algorithm to the underlying architecture (SIMD curr. not good enough)
  - Join algorithms can be made efficient while remaining hardware-oblivious
    - less portable and less robust to, e.g., data skew
  - SMJ is already better than HJ join and can be efficiently implemented without SIMD

Parallel Hash Joins

- No partitioning hash join
- Partitioned hash join
- Radix hash join
- Parallel radix hash join

No Partitioning (Hash) Join

- Balkesen, et al., ICDE 2013 (T. Ozsu)
- A direct parallel version of the canonical hash join
  - It does not depend on any hardware-specific parameters
- NPJ Algorithm (R⋈S)
  - Both input relations are divided into equi-sized portions that are assigned to a number of worker threads
  - Build phase: R workers populate shared hash table that all workers can access
  - After synchron. barrier: worker threads in probe phase read from S and concurrently find matching join pairs from R
- Concurrent insertions into hash table must be synchronized
  - Latches (contention?)
  - Compare-and-Swap (CAS)
  - Probe: RO mode

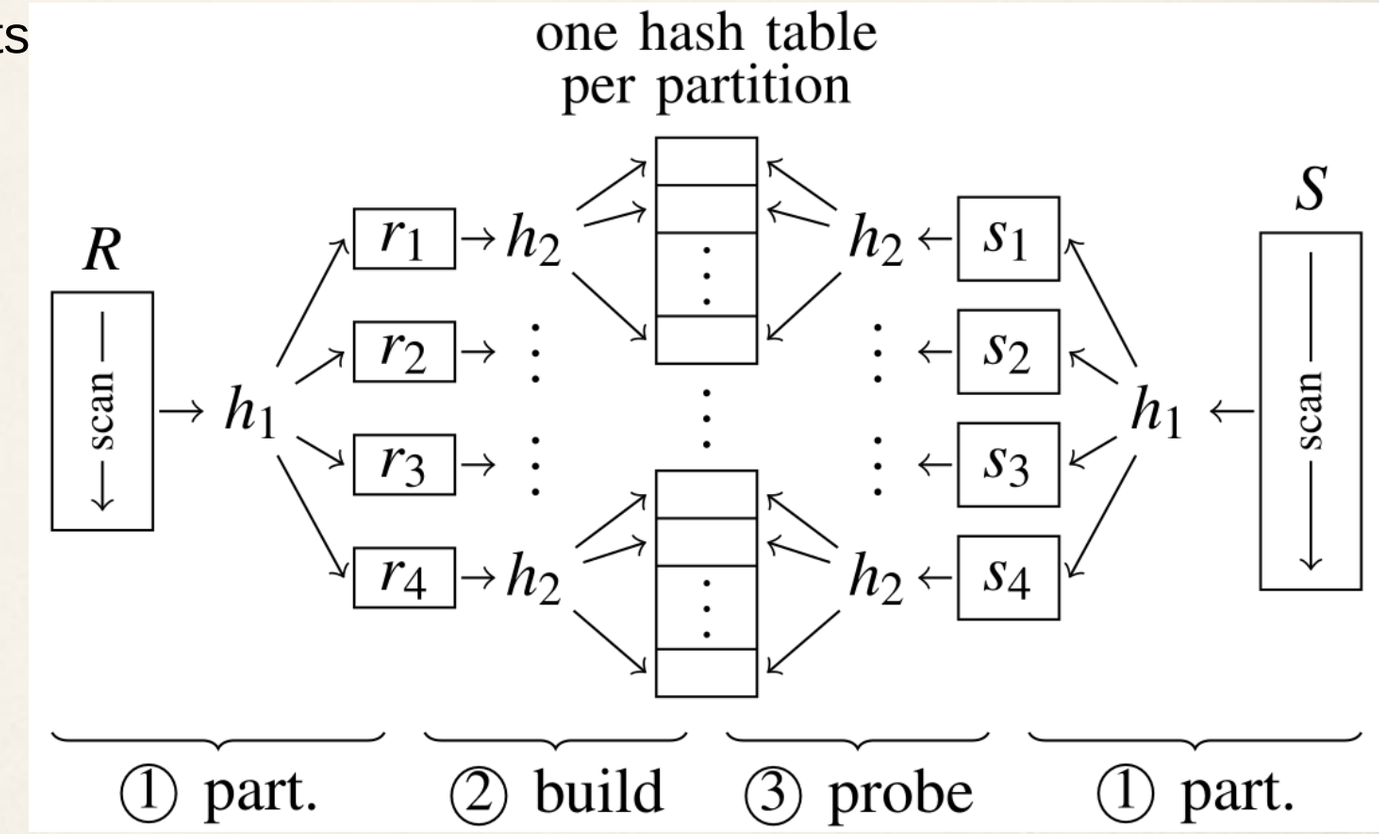


Observations: Cache and TLB

- Shatdal, et al., VLDB, 1994; Manegold, et al., IEEE TKDE, 2002
- Avoiding cache misses during build and probe phases
- Hashing results in cache misses (while accessing memory)
  - When the HT is larger than cache, almost every access to HT results in a cache miss
  - Therefore, partitioning HT into **cache-sized chunks** reduces cache misses and improves performance
- Effects of translation look-aside buffer (TLB) during partitioning phase
  - TLB is a memory cache that stores the recent translations of virtual memory to physical memory (located in MMU, in CPU)
  - Partitions typically reside on different memory pages with a separate entry for virtual memory mapping (TLB entry) required for each partition.
  - # of TLB entries is an upper bound on # of partitions that can be created or accessed efficiently at the same time
  - This led to multi-pass partitioning, now a standard component of radix join

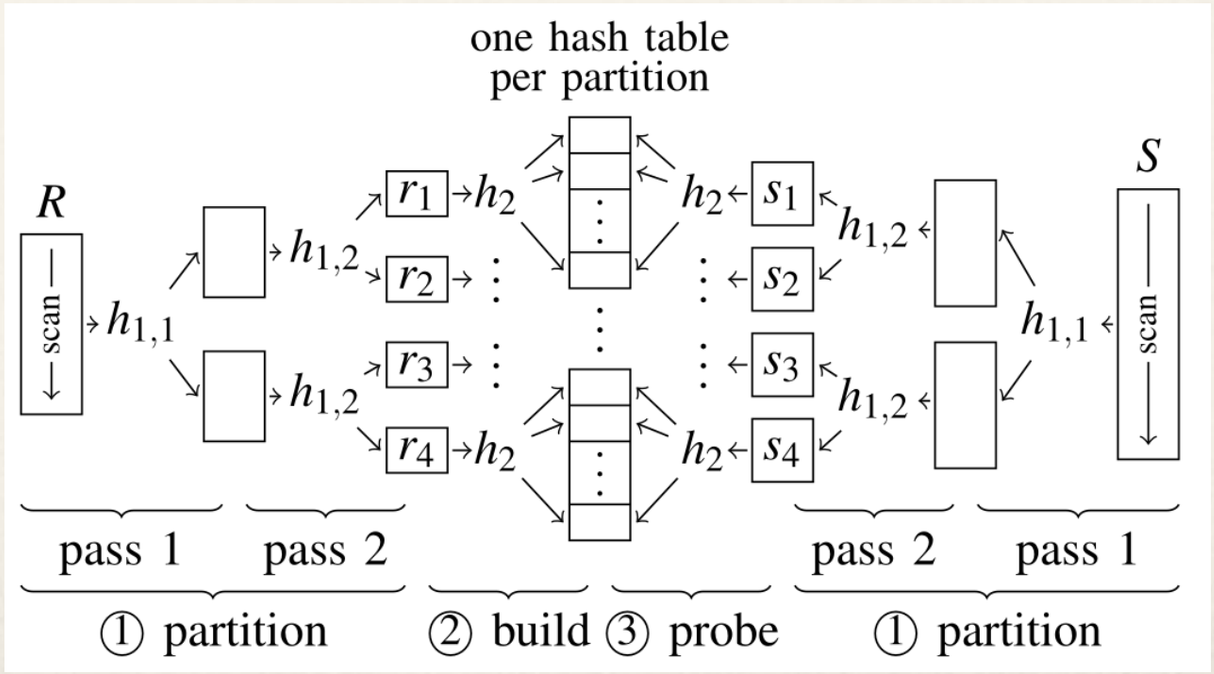
Partitioned Hash Join

- Build phase: input rels R and S divided into partitions r_i and s_j
  - R and S tuples are divided up using hash partitioning (h1) on their **join key values** (consequences? $r_i \bowtie s_j = \emptyset$ $(i \neq j)$)
    - Parts of R may be assigned to threats
  - A separate HT is created for each r_i partition
    - A separate thread is created for each partition r_i
    - No synchronisation needed for writing HTs
  - Hash tables for partitions r_i now **fit into CPU cache**!
- Probe phase:
  - s_j partitions are scanned and
  - Respective hash tables are probed for matching tuples using h2
    - One thread is used for one partition s_i



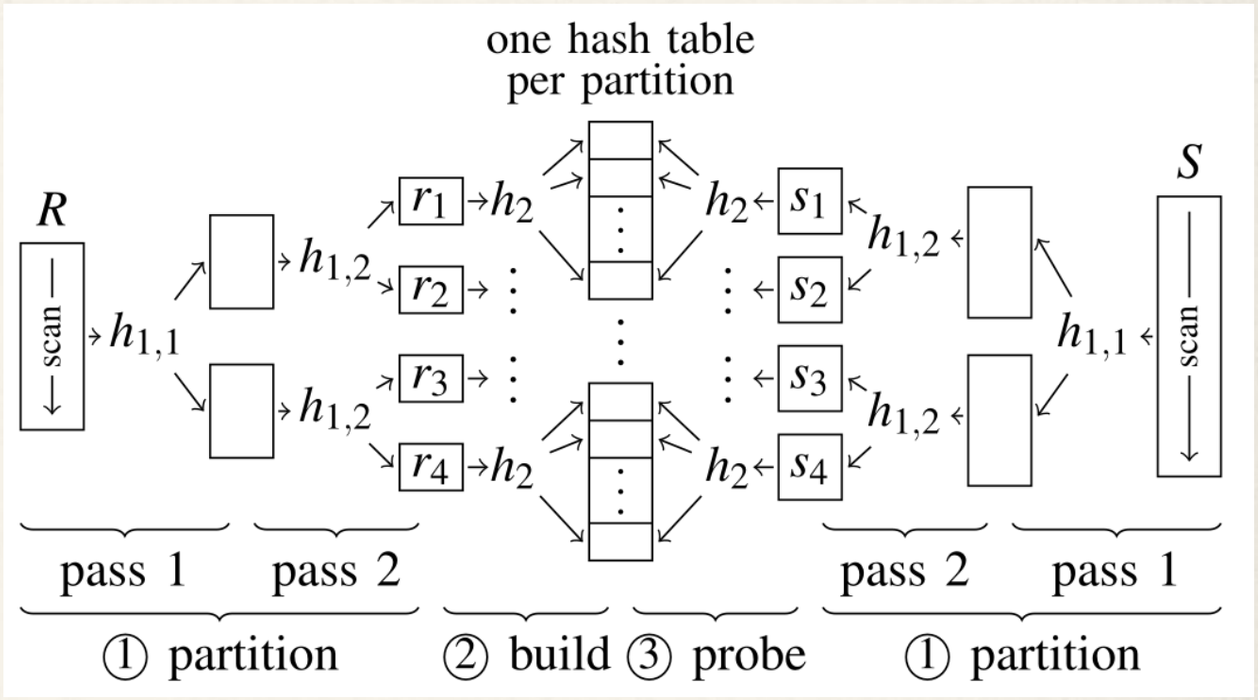
Radix Hash Join

- Radix partitioning
  - Excessive TLB misses can be avoided by partitioning input data in **multiple passes**
    - ∀ pass j, partitions produced by preceding pass j−1 are refined
    - Partitioning fan-out never exceeds the hardware limit given by the number of TLB entries
    - Precomputing output memory ranges of each target partition by building histograms
  - Each pass looks at different set of bits from hash function h1 ==> radix partitioning
    - 2-3 passes sufficient to create partitions without problems with TLB limitations
    - Different targeted memory regions ==> no need for further synchronization!



Radix Hash Join

- Radix join R⋈S
  - Both inputs are partitioned using two-pass radix partitioning
  - **Built phase:**
    - Hash tables are then built over each r_i partition of input table R.
  - **Probe phase:**
    - All s_i partitions are scanned and respective r_i partitions probed for join matches.

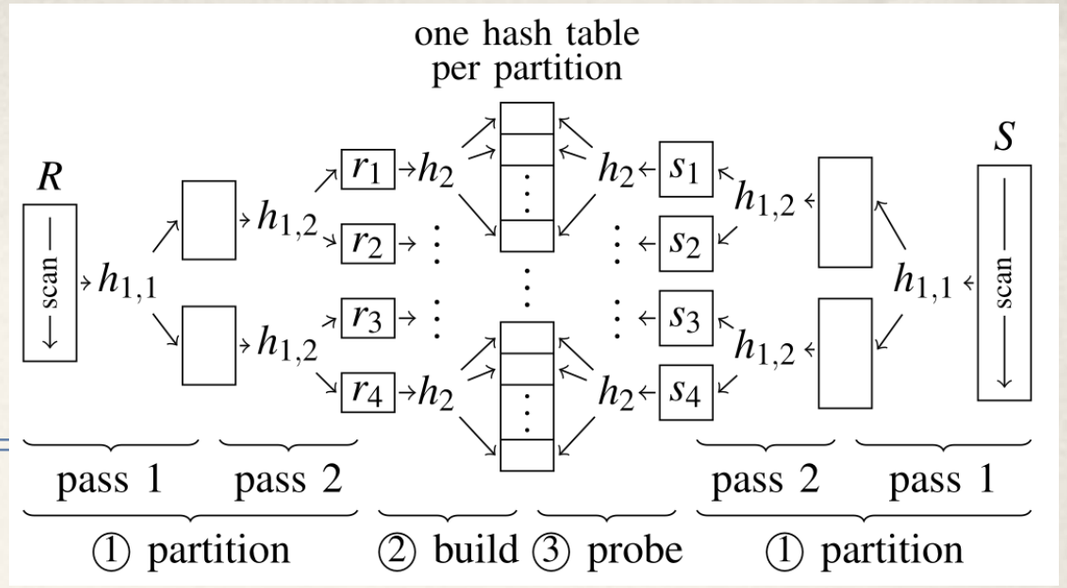


- Maximum "fanout" per pass is restricted by TLB size
  - log(|R|) (or log(|S|)) passes are necessary
- Complexity of radix hash join is O((|R| + |S|) log |R|) // R is typically smaller
- Hardware parameters:

  1) Maximum fanout per radix pass is limited by # TLB entries;

  2) Resulting partition size = roughly the size of the system's CPU cache.

Radix Hash Join



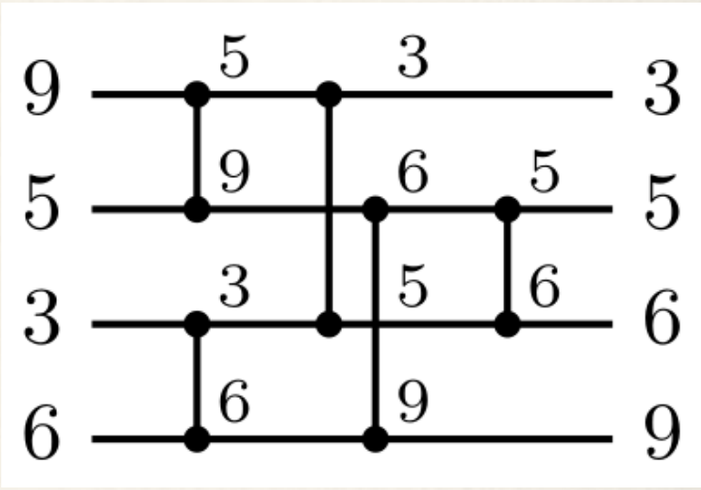
- Mapping tasks to threads
  - Dividing R and S into sub-relations that are assigned to individual threads
    - During the first pass, all threads create a shared set of partitions
    - Number of partitions in this set is limited by hardware parameters and typically small
    - They are accessed by potentially many execution threads, creating a **contention problem**
    - For each thread a dedicated range is reserved within **each output partition**
      - Threads scan input relations and write to selected memory regions
  - To this end, both input relations are scanned twice
    - 1st scan: computes a set of histograms over the input data
    - 2nd scan: each thread pre-computes the exclusive location, where it writes its output
- Final partitions used in built and probe phases **fit into CPU cache**
  - In building phase, threads are building the hash tables for each particular partition
    - No contention since there are **different memory regions** for different HTs
  - In probing phase, many execution threads write the result table
    - No contention since each threat joins exactly two partitions (s_i and r_i)

Sort-Merge Join

- Sort R and S on their join keys and then merge sorted R' and S'.
  - Sorting R and S is the **dominant cost**
  - Usually, merge sort is used for the sort phase of SM join.
- New hardware is used
  - Single Instructions on vectors (multiple scalars)
  - SIMD registers in novel CPUs (typically, 4x4 SIMD registers)
  - There are many SIMD instructions that work on vectors

Sorting Networks

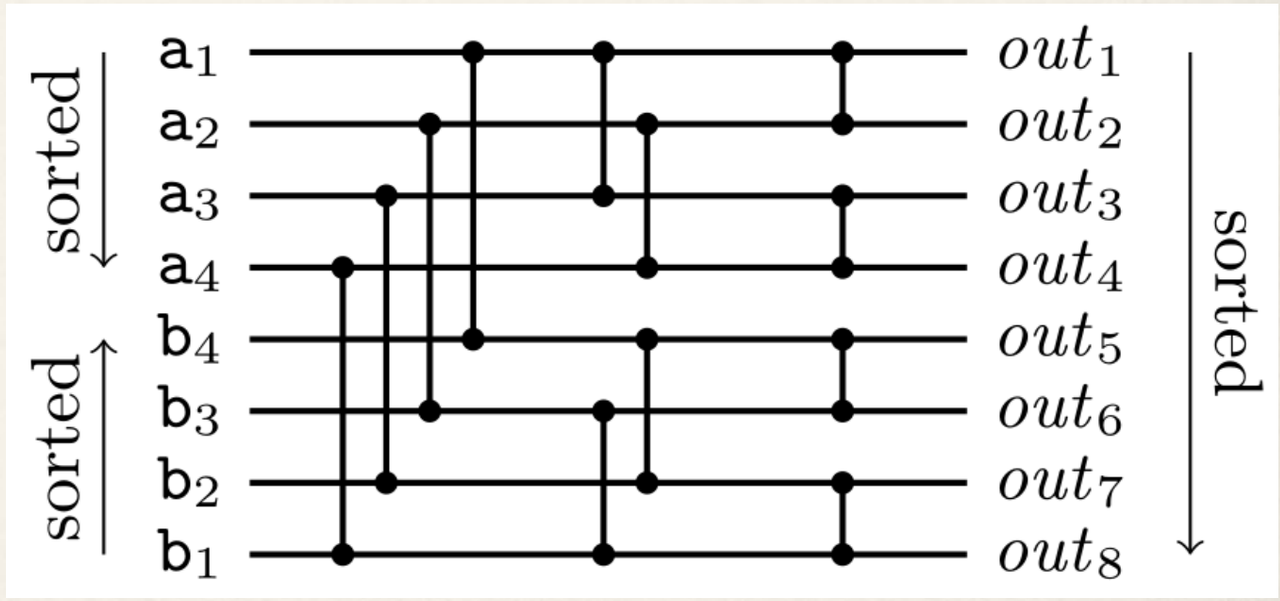
- Knuth's notation of sorting networks
  - Input ⟨9, 5, 3, 6⟩
  - Comparator emits the smaller value at the top and larger value on the bottom
  - Traversing comparators from left to right
- Comparators can be implemented with min/max operators only
  - Five comparators from the above figure is converted into sequence of 10 min/max operations (no branching, fast!)
- Sorting networks are appealing because they be accelerated through SIMD instructions
  - k=4: 4 SIMD vector registers with 4 elements
  - If vectors are loaded in SIMD registers then they need to be transposed first (expensive shuffle instructions)
  - 4 vectors are sorted in the time used for one in CPU without SIMD (speedup 2.7 because of shuffle ops)




```
e = min(a, b)
f = max(a, b)
g = min(c, d)
h = max(c, d)
i = max(e, g)
j = min(f, h)
w = min(e, g)
x = min(i, j)
y = max(i, j)
z = max(f, h)
```

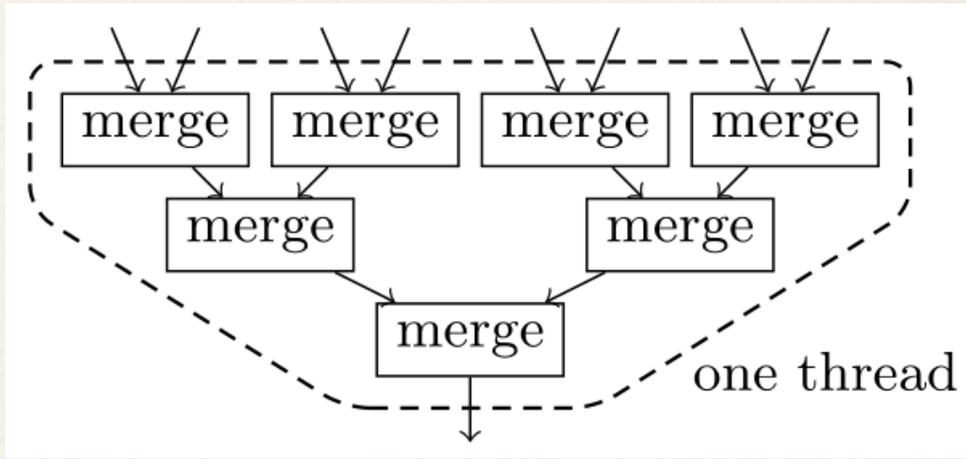
Merging Sorted Runs

- Bitonic Merge Networks
  - Merging phase of SM join also benefits from SIMD acceleration
  - Networks that combine two pre-sorted inputs into an sorted output
    - This allows building larger networks
- A network that combines two input lists of size four
  - Using min and max operations, and shuffle operations in between comparators
- Merging Larger Lists
  - For larger input sizes, merge networks scale poorly - $O(N \log^2 N)$
  - Small merge networks can be used as a kernel within a merging algorithm for larger lists



Balancing Computation & Bandwidth

- Accesses to off-chip memory makes sorting sensitive to the characteristics of the memory interface.
- 8-wide bitonic merge implementation requires 36 assembly instructions per 8 tuples being merged (29 CPU cycles)
  - With a clock frequency of 2.4 GHz, this corresponds to a memory bandwidth 10.6 GB/s for a single merging thread
  - This is more than existing interfaces support (+ 8 cores per CPU)
  - Out-of-cache merging is thus severely bound by the memory bandwidth
- Merging more than 2 runs at once => Memory bandwidth demand reduced
  - **Multi-way merging** saves round-trips to memory (memory bandwidth)
  - Multi-way merging is implemented with multiple two-way merge units
- Two-way merge units are connected by queues
  - FIFO queues are sized such that all queues together still fit into CPU cache



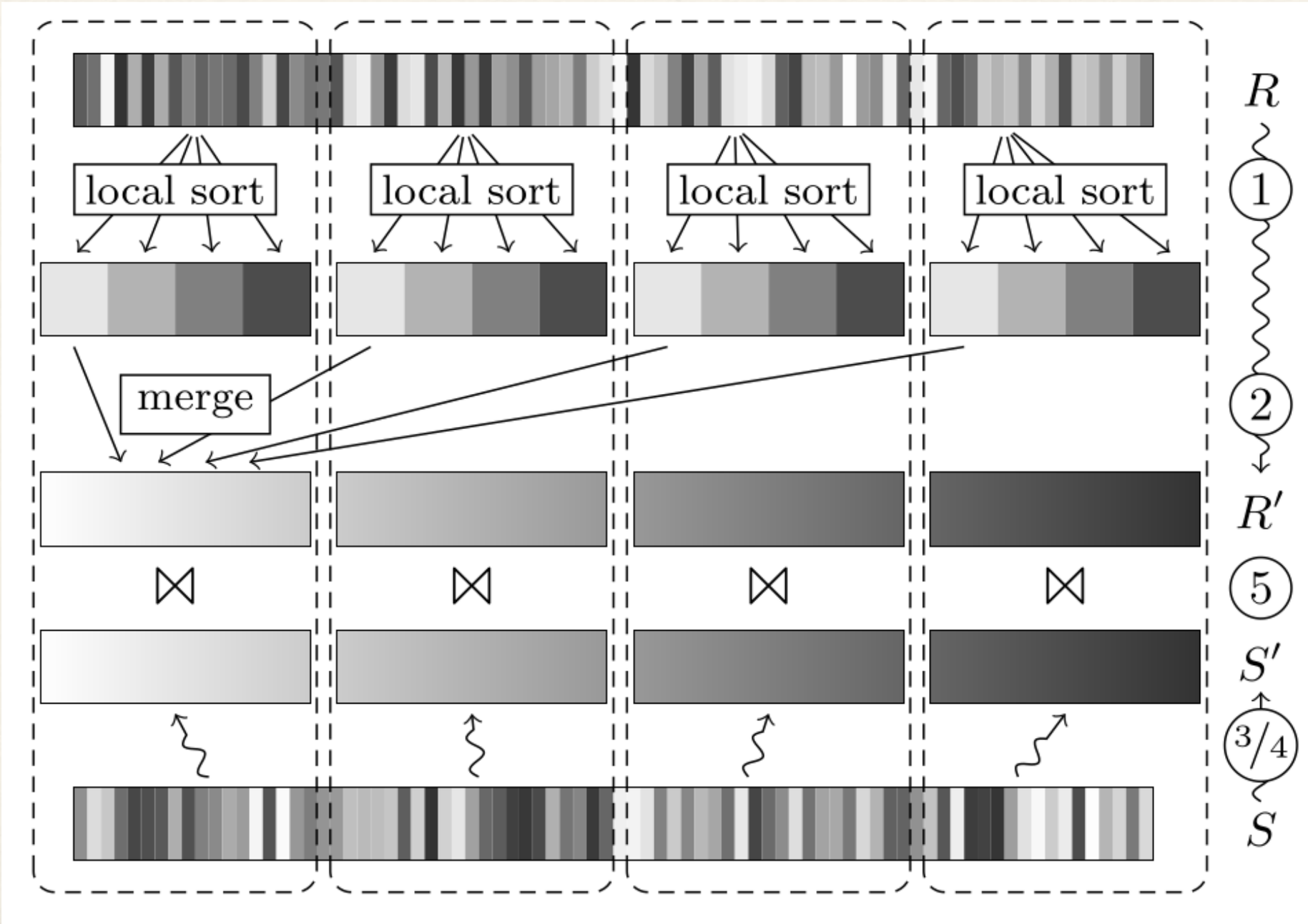
Sorting and Memory Hierarchy

- Sorting using Advanced Vector Extensions (AVX) in CPUs
  - **AVX sorting algorithm** (2.5 - 3 times faster then C++ sort)
- Cache hierarchies in modern hardware require separating the overall sorting into several phases to optimize cache access
  - In-register sorting, with runs that fit into (SIMD) CPU registers
  - In-cache sorting, where runs can still be held in a CPU-local cache
  - Out-of-cache sorting, once runs exceed cache sizes
- In-register sorting
  - Run-generation using sorting networks (initial sorted runs)
- In-cache sorting
  - Initial runs are merged until runs can no longer be contained within CPU caches
  - Multi-way merging using a hierarchy of Bitonic Merge Networks
- Out-of-Cache Sorting
  - Continues merging until the data is fully sorted
  - Memory references will have to be fetched from off-chip memory

m-way Sort-Merge Join

- Highly parallel sort-merge join
  - Balkesen, et al., Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited, VLDB, 2013 (T. Ozsu)
  - Relies on both data and thread parallelism
  - Data parallelism based on SIMD extensions to standard ISA
  - Carefully optimized toward NUMA
- General description of the algorithm
  - Initially, input relations R and S are equally distributed across NUMA regions
  - 1st phase: each thread is assigned its NUMA-local chunk of R and all the threads range-partition their local chunks in parallel
    - Allowing threads in the subsequent phases to work independently without any synchronization.
    - Partitioning fan-out is usually on the order of the number of threads (64–128)
    - Then each local partition is sorted locally (NUMA-local) using the AVX sorting algorithm
    - Different threads can sort different partitions independently

m-way Sort-Merge Join



m-way Sort-Merge Join

- General description of the algorithm
  - 2nd phase:
    - The only phase that requires shuffling data between NUMA regions (thru interconnect bandwidth)
    - **Multi-way merging** presented before overlaps data transfer and merging (they are in balance)
    - Outcome of this phase is a globally sorted copy of R, indicated as R'
  - 3rd and 4th phase:
    - The same steps 1 and 2 are applied to relation S!
  - 5th phase:
    - R' and S' are stored in the NUMA-local memory of each thread
    - Each thread concurrently evaluates the join between NUMA-local sorted runs using a single-pass merge join

m-pass and mpsm SM Joins

- Competitors of m-way join
- Sort-Merge Join Algorithm – **m-pass**
  - Algorithm differs solely in Phase 2
  - Instead of applying a multi-way merge for merging NUMA-remote runs, m-pass applies successive two-way bitonic merging
  - First iteration of merging of sorted runs is done as data is transferred to local memory
    - The number of runs reduces to 1/2 of the initial total number of runs.
  - The rest of the merging done in local memory, using multi-pass merging technique
- Massively Parallel Sort-Merge Join – **mpsm**
  - mpsm first globally range-partitions relation R
    - Different ranges of R are assigned to different NUMA-regions/threads
  - Next, each thread independently sorts its partition, resulting in a globally-sorted R'
  - S is sorted only partially
    - Each thread sorts its NUMA-local chunk of S without a prior partitioning
  - Last phase: a run of R must be merge-joined with all the NUMA-remote runs of relation S

Other join algorithms

- 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 tuples of 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

- 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

- 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.

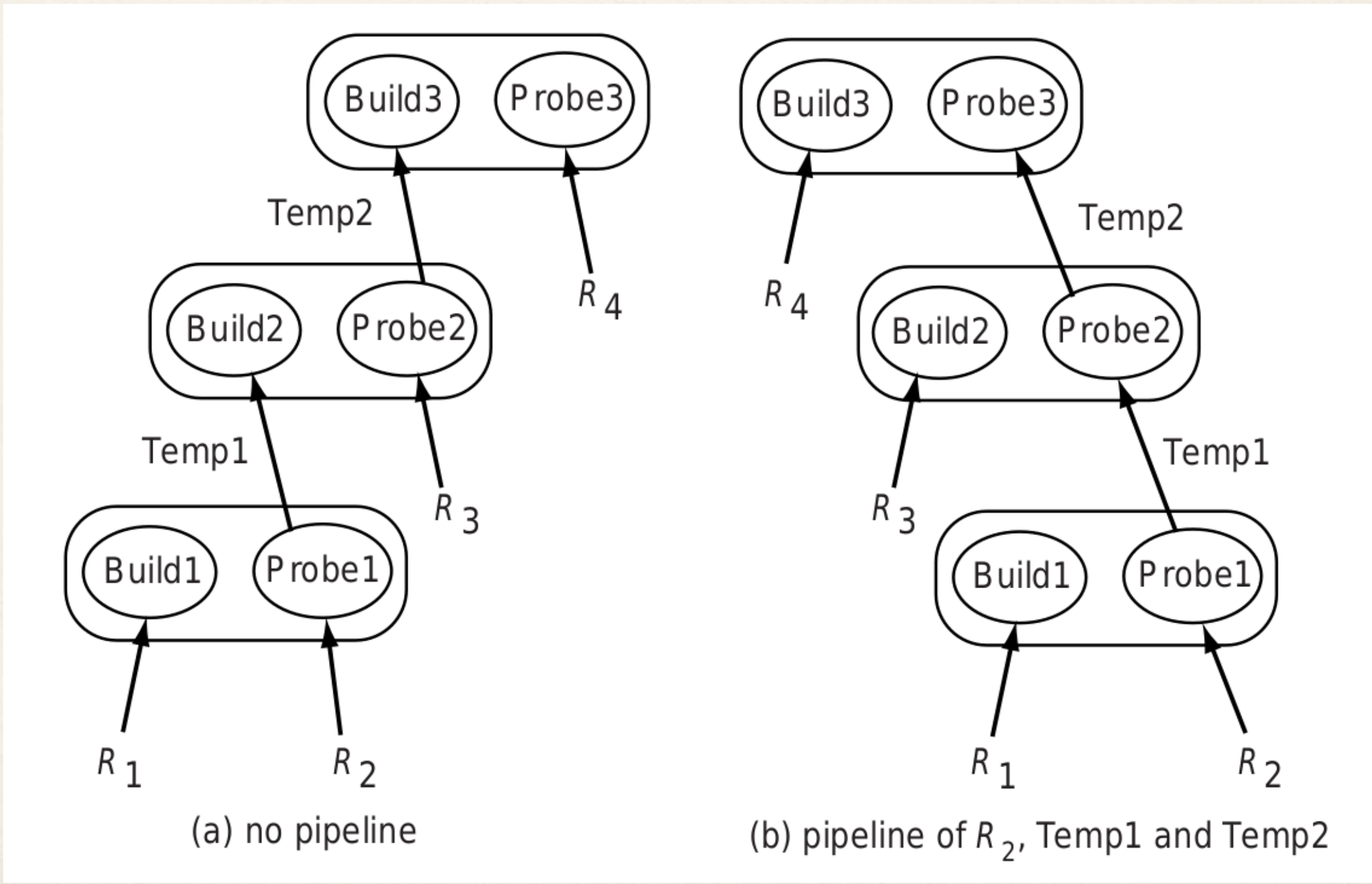
Parallel Query Optimization

- 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

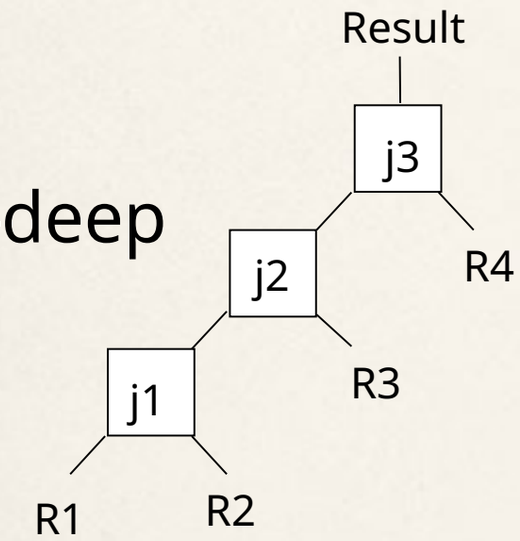


Search space

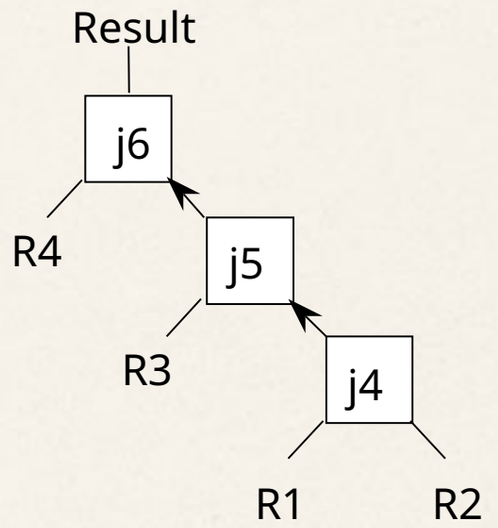
- Set of nodes where a relation is stored is called its home.
  - ➡️ The home of an operator is the set of nodes where it is executed
  - ➡️ The home of an operator must be the home of its operands in order for the operator to access its operands
  - ➡️ For binary operators such as join, this might imply 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

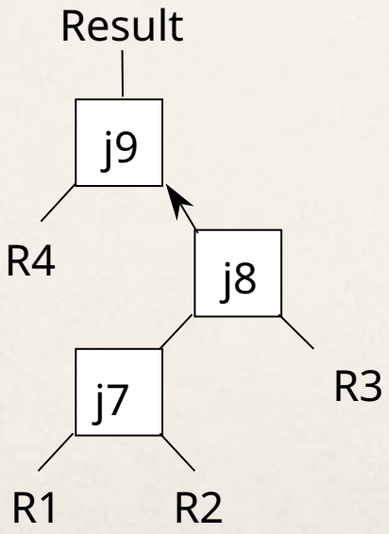
Left-deep



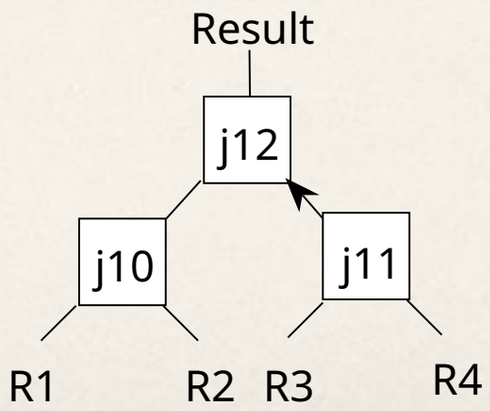
Right-deep



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