Data Intensive Systems (DIS)
KBH-SW7 E25

2. Parallel Databases

AALBORG UNIVERSITY

Agenda

- Introduction
- Parallel Database Architectures
- IO Parallelism and Partitioning
- Other Types of Parallelism

Parallel Databases

- A parallel DBMS runs across multiple processors and is designed to execute operations in parallel, whenever possible.
- A parallel DBMS links a number of smaller machines to achieve the same throughput as expected from a single large machine.
- Parallel databases improve processing and IO speeds by using multiple CPUs and disks in parallel.
  - Processors are *tightly* coupled and constitutes a single database system in *a single location*.
  - Data is often partitioned among different disks to enable parallel retrieval and increase the throughput.

Distributed Databases

- A Distributed database is defined as a *logically related* collection of shared data that is *physically distributed* over a computer network on *different sites*.
  - The data is split and replicated across different sites.
- Homogeneous distributed databases
  - Same software/schema on all sites, data may be partitioned among sites
  - Goal: provide a view of a single database, hiding details of distribution
- Heterogeneous distributed databases
  - Different software/schema on different sites
  - Goal: integrate existing databases to provide useful functionality

Agenda

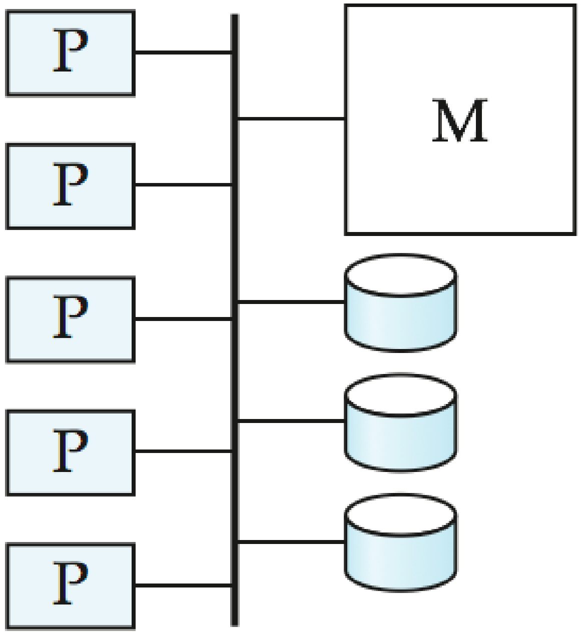
- Introduction
- Parallel Database Architectures
  - Shared Nothing
  - Shared Memory
  - Shared Disk
  - Hierarchical
- IO Parallelism and Partitioning
- Other Types of Parallelism

Parallel Database Architectures

- **Shared memory** -- processors share a common memory
- **Shared disk** -- processors share a common disk
- **Shared nothing** -- processors share neither a common memory nor common disk
- **Hierarchical** -- hybrid of the above architectures

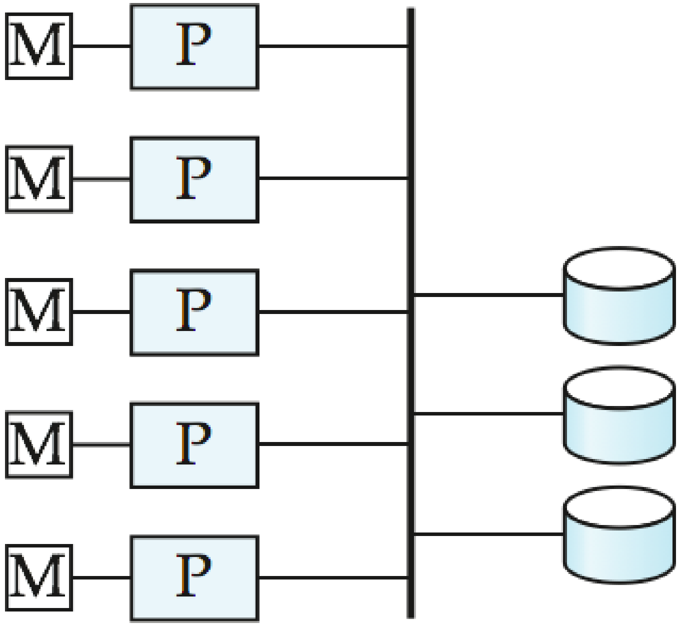
Shared Memory

- Processors (and disks) have access to a common memory, typically via a bus or a high-speed LAN.
- Extremely efficient communication between processors — data in shared memory can be accessed by any processor without having to move it using software.
- Downside – architecture is not scalable beyond 32 or 64 processors since the bus or the LAN becomes a bottleneck.
- Widely used for lower degrees of parallelism (4 to 8).



Shared Disk

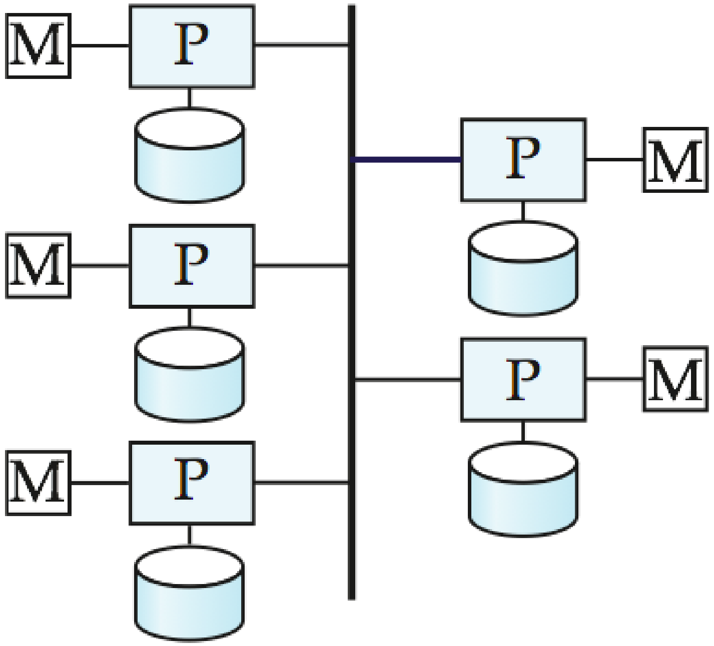
- All processors can directly access all disks via an interconnection network, but the processors have private memories.
  - The memory bus is not a bottleneck
  - A degree of **fault-tolerance**: if a processor fails, the other processors can take over its tasks since the database is resident on disks that are accessible from all processors.
- Downside: bottleneck now occurs at interconnection to the disk subsystem.
- Shared-disk systems can scale to a somewhat larger number of processors, but communication between processors is slower.



AALBORG UNIVERSITET

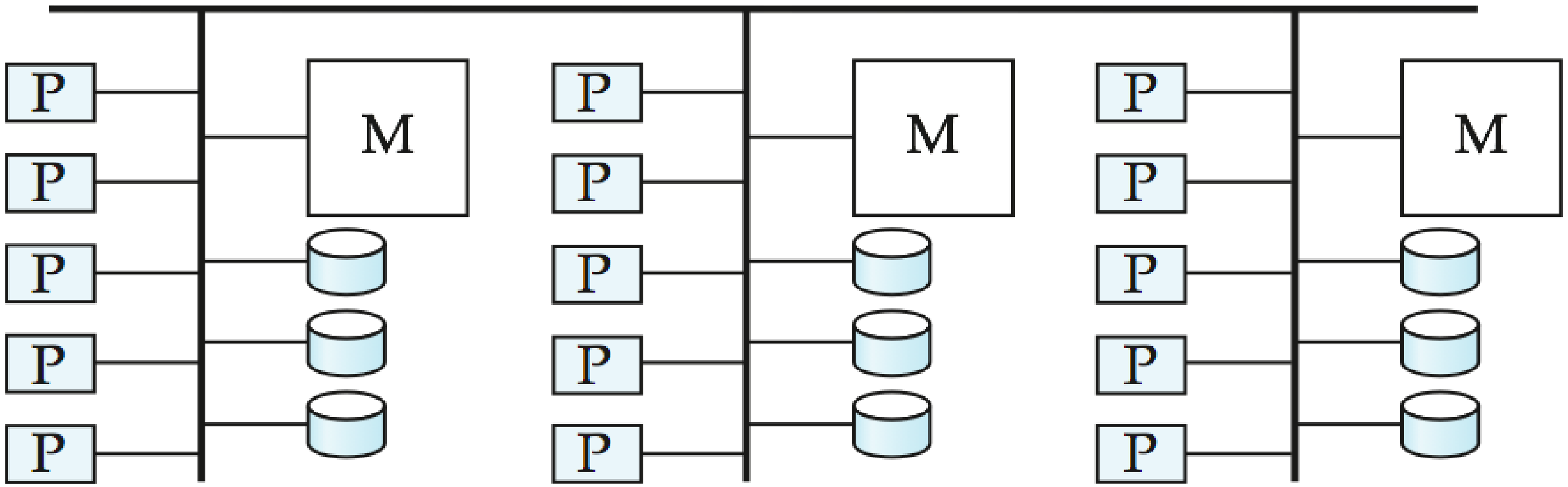
Shared Nothing

- Each node has a processor, memory, and one or more disks. Processor at a node communicates with processors at other nodes using an interconnection network.
- A node = a server for the data on its own disks.
- Data accessed from local disks (or memory) do not pass through the network, minimizing the interference of resource sharing.
- Shared-nothing multiprocessors can be scaled up to thousands of processors without interference.
- Main drawbacks:
  - cost of communication and non-local disk access
  - sending data involves software interaction at both ends.



Hierarchical

- Combines characteristics of shared-memory, shared-disk, and shared-nothing architectures.
- Top level is a shared-nothing architecture – nodes connected by an interconnection network, and do not share disks or memory with each other.
- Each node can be a shared-memory system with a few processors.
  - Alternatively, each node could be a shared-disk system, and each of the systems sharing a set of disks could be a shared-memory system.



AALBORG UNIVERSITET

Agenda

- Introduction
- Parallel Database Architectures
- IO Parallelism and Partitioning
  - Round-robin
  - Hash
  - Range
- Other Types of Parallelism

Parallelism in Databases

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

I/O Parallelism

- Reduce the time required to retrieve relations from disk by partitioning the relations on *multiple disks*.
- **Horizontal partitioning**
  - Tuples of a relation are divided among many disks such that *each tuple* resides on one disk.
  - Mostly used in parallel databases
- Horizontal partitioning techniques
  - Round-robin
  - Hashing partitioning
  - Range partitioning

id	Name	Age	email

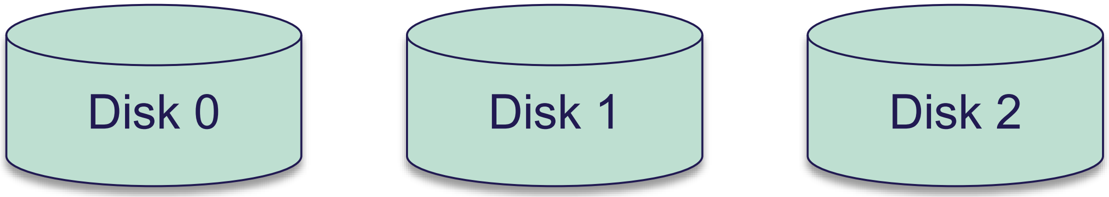
Data Partitioning w.r.t *n* Disks

- **Round-robin**
  - Send the i^{th} tuple in the relation to disk i mod n.
- **Hash partitioning**
  - Choose one or more attributes as the partitioning attributes.
  - Choose hash function h with range $0 \ldots n - 1$
  - Let i denote the result of hash function h applied to the partitioning attribute value of a tuple. Send tuple to disk i.
- **Range partitioning**
  - Choose an attribute as the partitioning attribute.
  - A partitioning vector $[v_0, v_1, ..., v_{n-2}]$ is chosen.
  - Let v be the partitioning attribute value of a tuple. Tuples such that $v_i \leq v < v_{i+1}$ go to disk $i + 1$. Tuples with $v < v_0$ go to disk 0 and tuples with $v \geq v_{n-2}$ go to disk n-1.
    - E.g., with a partitioning vector [5,11], a tuple with partitioning attribute value of 2 will go to disk 0, a tuple with value 8 will go to disk 1, while a tuple with value 20 will go to disk2.

Example of Data Partitioning

R	x	y	z
t_1	1	1	…
t_2	2	4	…
t_3	15	6	
t_4	6	6	
t_5	7	2	…
t_6	9	3	…
t_7	12	4	
t_8	5	1	
t_9	8	3	…

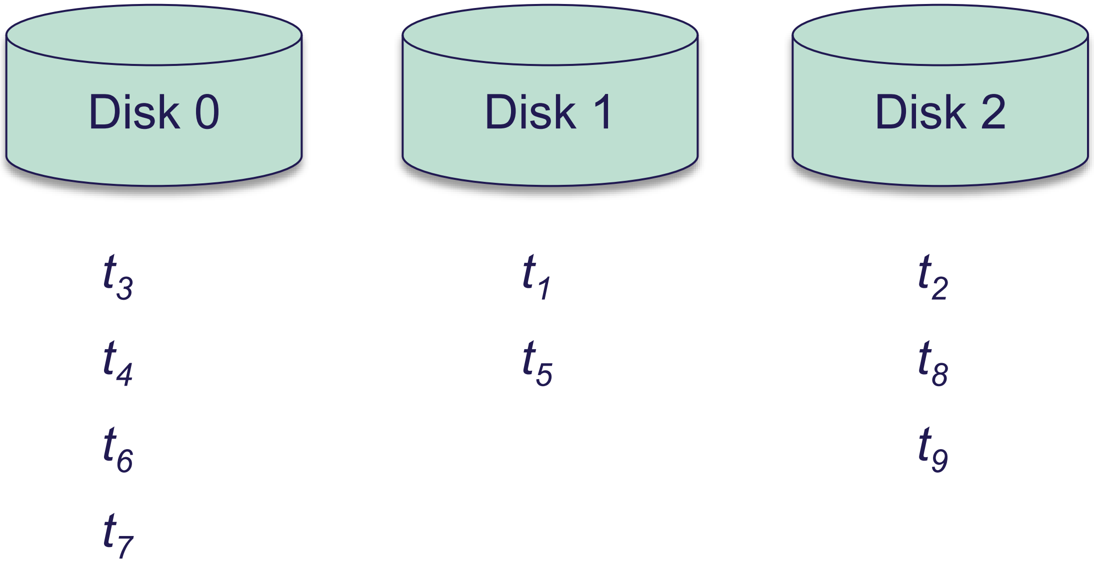
By Round-Robin



t_1	t_2	t_3
t_4	t_5	t_6
t_7	t_8	t_9

R	*x*	*y*	*z*
t_1	1	1	…
t_2	2	4	…
t_3	15	6	
t_4	6	6	
t_5	7	2	…
t_6	9	3	…
t_7	12	4	
t_8	5	1	
t_9	8	3	…

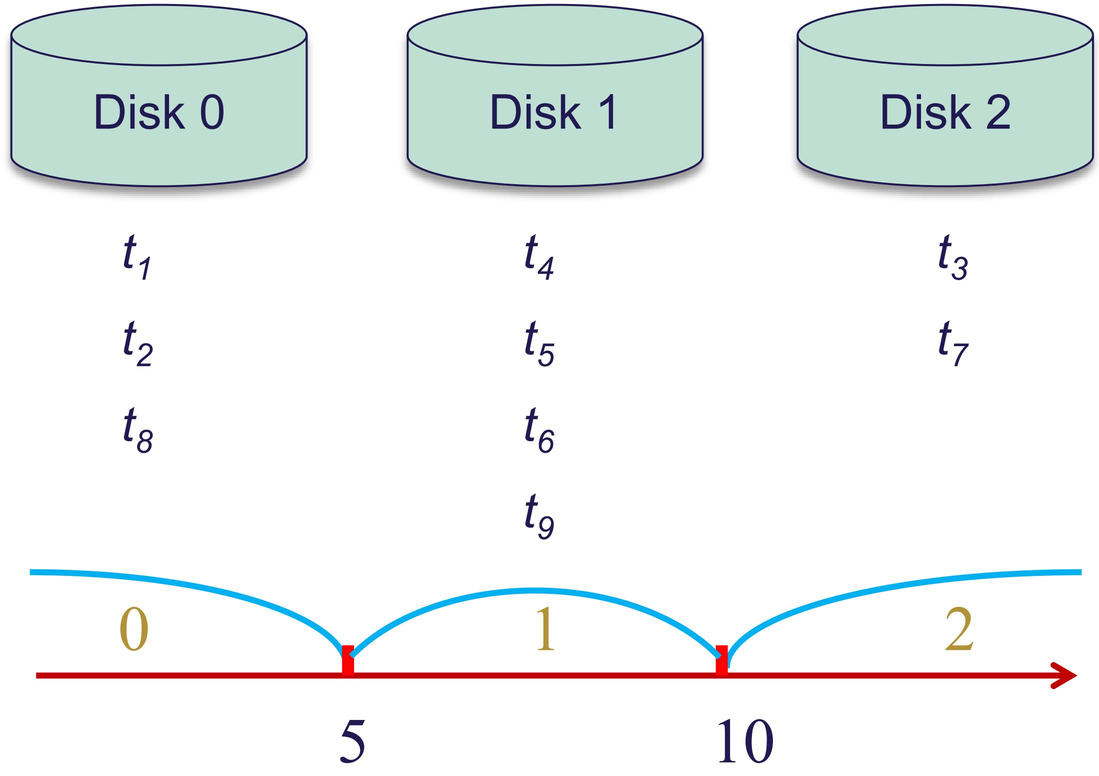
By Hash



- Hash function: $h(x) = x \; mod \; 3$

R	x	y	z
t_1	1	1	…
t_2	2	4	…
t_3	15	6	
t_4	6	6	
t_5	7	2	…
t_6	9	3	…
t_7	12	4	
t_8	5	1	
t_9	8	3	…

By Range



- 0: $x \leq 5$; 1: $5 < x \leq 10$; 2: $x > 10$

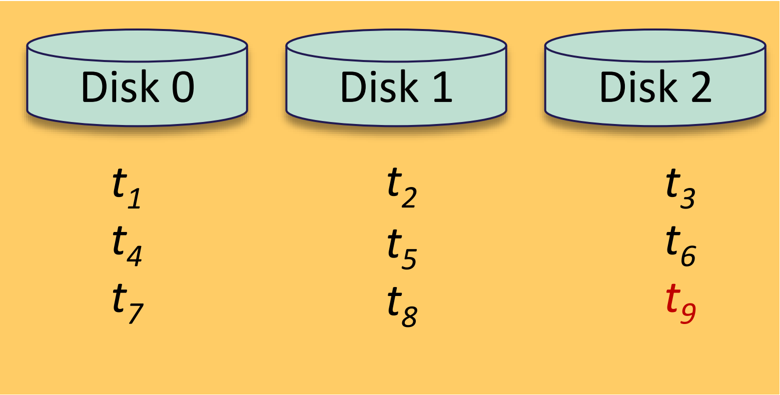
R	x	y	z
t_1	1	1	…
t_2	2	4	…
t_3	15	6	
t_4	6	6	
t_5	7	2	…
t_6	9	3	…
t_7	12	4	
t_8	5	1	
t_9	8	3	…

Queries

- How to process queries against a partitioned relation?
- Three types of queries
  - Query-1: Find all tuples that have $x = 8$.
    - A *point* query issued on the partitioning attribute.
  - Query-2: Find all tuples that have $a < x \leq b$.
    - A *range* query on the partitioning attribute.
  - Query-3: Find all tuples that have $a < y \leq b$.
    - A *range* query **NOT** on the partitioning attribute.

R	x	y	z
t_1	1	1	…
t_2	2	4	…
t_3	15	6	
t_4	6	6	
t_5	7	2	…
t_6	9	3	…
t_7	12	4	
t_8	5	1	
t_9	8	3	…

Query-1: Find all tuples that have $x = 8$



- Under Round-Robin:
  - All disks need to be searched.
  - In each disk, if there is no index, then all buckets need to be searched.
- In the example,

Worst case: Each tuple in a different local bucket.

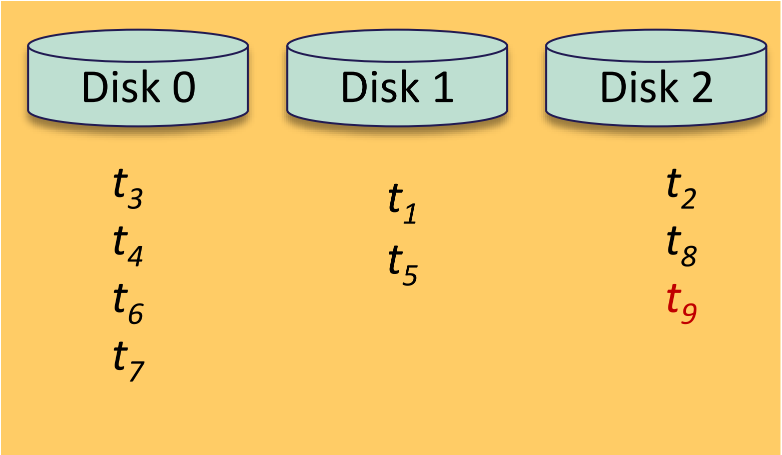
	Without Local Index	With Local Index
Disks searched	3	3
Response time	3	1
Total time	9	3

Query-1: Find all tuples that have $x = 8$

- **Under Round-Robin**
  - **All *n* disks** need to be searched. In each disk, if there is no index, then **all *m* buckets** need to be searched.
- In general,

	Without Local Index	With Local Index
Disks searched	n	n
Response time	m	1
Total time	$n \times m$	n

Query-1: Find all tuples that have $x = 8$



- **Under Hash**
  - Only 1 disk needs to be searched.
    In each disk, if there is no index,
    then all buckets need to be searched.
- In the example,

	Without Local Index	With Local Index
Disks searched	1	1
Response time	3	1
Total time	3	1

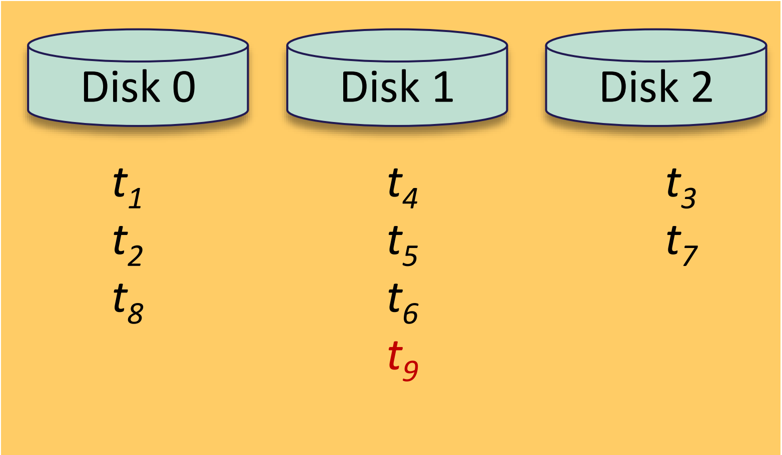
Query-1: Find all tuples that have $x = 8$

- Under Hash
  - **Only 1 disk** needs to be searched. In each disk, if there is no index, then **all *m* buckets** need to be searched.

- In general,

	Without Local Index	With Local Index
Disks searched	1	1
Response time	m	1
Total time	m	1

Query-1: Find all tuples that have $x = 8$



- **Under Range**
  - Only 1 disk needs to be searched.
    In each disk, if there is no index,
    then all buckets need to be searched.
- In the example,

	Without Local Index	With Local Index
Disks searched	1	1
Response time	4	1
Total time	4	1

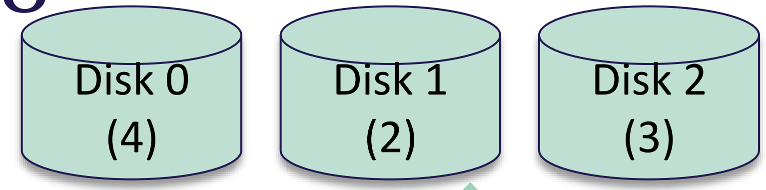
Query-1: Find all tuples that have $x = 8$

- **Under Range**
  - **Only 1 disk** needs to be searched. In each disk, if there is no index, then **all *m* buckets** need to be searched.

- In general,

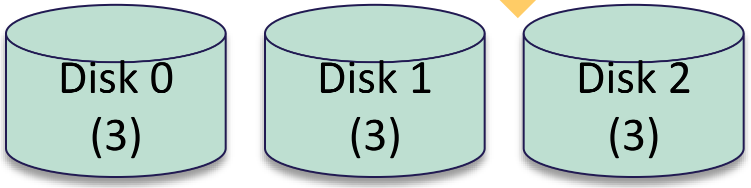
	Without Local Index	With Local Index
Disks searched	1	1
Response time	m	1
Total time	m	1

Query-2: Find all tuples with $5 < x \leq 8$



- Without local index
- In the example,

	range	hash	Round robin
Disks searched	1	3	3
Response time	4	4	3
Total time	4	9	9



Query-2: Find all tuples with $5 < x \leq 8$

- With local index
- In the example,

D0{6, 9, 12, 15} D1{1, 7} D2{2, 5, 8}

	range	hash	Round robin
Disks searched	1	3	3
Response time	3	1	1
Total time	3	3	3

D0{1, 6, 12}
D1{2, 5, 7}
D2{8, 9, 15}

Query-2: Find all tuples with $2 < y \leq 5$

- Without local index
- In the example,

	range	hash	Round robin
Disks searched	3	3	3
Response time	4	4	3
Total time	9	9	9

Query-2: Find all tuples with $2 < y \leq 5$

D0{3, 4, 6, 6} D1{1, 2} D2{1, 3, 4}

- With local index on y
- In the example,

	range	hash	Round robin
Disks searched	3	3	3
Response time	2	2	2
Total time	4	4	4

D0{1, 1, 4}
D1{2, 3, 3, 6}
D2{4, 6}

D0{1, 4, 6}
D1{1, 2, 4}
D2{3, 3, 6}

AALBORG UNIVERSITET

Pros and Cons of Round Robin Partitioning

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

Pros and Cons of Hash Partitioning

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

Pros and Cons of Range Partitioning

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

Agenda

- Introduction
- Parallel Database Architectures
- IO Parallelism and Partitioning
- Other Types of Parallelism
  - Interquery Parallelism
  - Intraquery Parallelism
    - Intraoperation Parallelism
    - Interoperation Parallelism

Interquery Parallelism

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

Intraquery Parallelism

- Execution of a **single query** in parallel on multiple processors/disks; important for speeding up long-running queries.
- Two complementary forms of intraquery parallelism:
  - **Intraoperation Parallelism** – parallelize the execution of each individual operation in the query. Different subsets of a relation is processed in parallel.
  - **Interoperation Parallelism** – parallelize the execution of different operations. Each operation processes the whole relation without parallelism.
- **Intraoperation Parallelism** scales better with increasing parallelism because *the number of tuples* processed by each operation is typically more than *the number of operations* in a query.

Parallel Algorithms for Relational Operations

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

Parallel Algorithms

- Parallel Sort
  - Range-Partitioning Sort
  - Parallel External Sort-Merge
- Parallel Join
  - Partitioned Parallel Join
  - Fragment-and-Replicate Join (Asymmetric and Symmetric)
  - Partitioned Parallel Hash-Join
  - Parallel Nested-Loop Join

Range-Partitioning Sort

Choose processors P_0, ..., P_m, where $m \le n$ -1 to do sorting.

1. Create range-partition vector with m entries, on the sorting attributes
2. Redistribute the relation using range partitioning
   - All tuples that lie in the i^{th} range are sent to processor P_i
   - P_i stores the tuples it received temporarily on disk D_i.
   - This step requires I/O and communication overhead.
3. Each processor P_i sorts its own partition of the relation locally.
   - Each processors executes same operation (sort) in parallel with other processors, without any interaction with the others (**data parallelism**).
4. Final merge operation is trivial: range-partitioning ensures that,
   for $1 \le i < j \le m$, the key values in processor P_i are all less than the key values in P_j.

Range-Partitioning Sort

Employee

Employee_ID	Employee_Name	Salary
E101	Andy	1000
E102	Bob	750
E103	Christian	400
E104	Adam	600
E105	Bill	1500
E106	John	300
E107	Helle	1200
E108	Jimmy	350
E109	Will	800
E110	Tommy	1150

SELECT * FROM Employee ORDER BY Salary

Range-Partition

[500, 900]

Employee1

D1

Employee_ID	Employee_Name	Salary
E103	Christian	400
E106	John	300
E108	Jimmy	350

Employee2

D2

Employee_ID	Employee_Name	Salary
E102	Bob	750
E104	Adam	600
E109	Will	800

Employee3

D3

Employee_ID	Employee_Name	Salary
E101	Andy	1000
E105	Bill	1500
E107	Helle	1200
E110	Tommy	1150

Range-Partitioning Sort

Employee1 (D1)

Employee_ID	Employee_Name	Salary
E103	Christian	400
E106	John	300
E108	Jimmy	350

P1

Employee1 (D1)

Employee_ID	Employee_Name	Salary
E106	John	300
E108	Jimmy	350
E103	Christian	400

Employee2 (D2)

Employee_ID	Employee_Name	Salary
E102	Bob	750
E104	Adam	600
E109	Will	800

P2

Employee2 (D2)

Employee_ID	Employee_Name	Salary
E104	Adam	600
E102	Bob	750
E109	Will	800

Employee3 (D3)

Employee_ID	Employee_Name	Salary
E101	Andy	1000
E105	Bill	1500
E107	Helle	1200
E110	Tommy	1150

P3

Employee3 (D3)

Employee_ID	Employee_Name	Salary
E101	Andy	1000
E110	Tommy	1150
E107	Helle	1200
E105	Bill	1500

Range-Partitioning Sort

Employee1

D1

Employee_ID	Employee_Name	Salary
E106	John	300
E108	Jimmy	350
E103	Christian	400

Employee2

D2

Employee_ID	Employee_Name	Salary
E104	Adam	600
E102	Bob	750
E109	Will	800

Employee3

D3

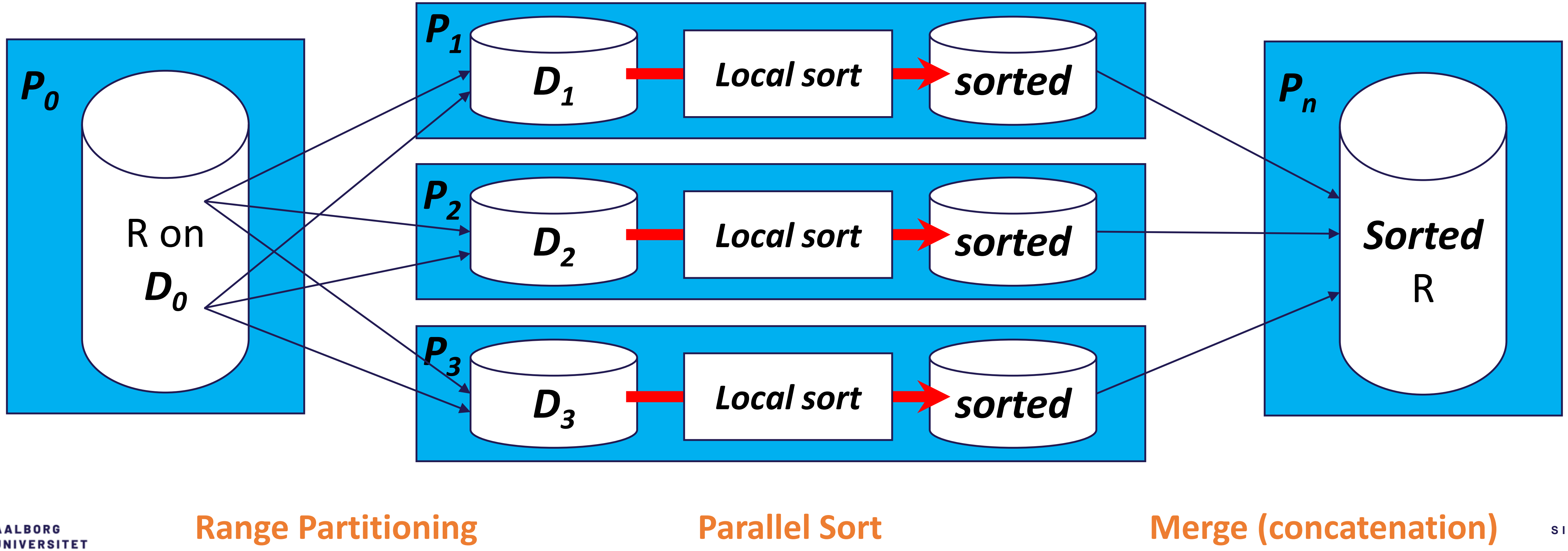
Employee_ID	Employee_Name	Salary
E101	Andy	1000
E110	Tommy	1150
E107	Helle	1200
E105	Bill	1500

Employee

Employee_ID	Employee_Name	Salary
E106	John	300
E108	Jimmy	350
E103	Christian	400
E104	Adam	600
E102	Bob	750
E109	Will	800
E101	Andy	1000
E110	Tommy	1150
E107	Helle	1200
E105	Bill	1500

Illustration of Range-Partitioning Sort

- Suppose the relation R to sort is on P_0, and we need the sorted result on P_n.



Parallel External Sort-Merge

- Assume the relation has already been partitioned among disks D_0, ..., $D_{n\text{-}1}$ (in *whatever* manner).

1. Each processor P_i locally sorts the data on disk D_i to run R_i .
2. Use a range-partitioning vector to partition each sorted run R_i into processors P_0, ..., $P_{n\text{-}1}$.
   - Each P_i transfers the data *in order*; but all processors do so *in parallel*.
   - Each processor sends the first partition to P_0, then the second partition to P_1, and so on so forth.
3. Each processor P_i performs a merge on the incoming range-partitioned data from every other processor.
   - The merges to get all sorted runs are parallelized.
4. The sorted data on processors P_0,..., $P_{n\text{-}1}$ are concatenated to get the final result.

Parallel External Sort-Merge

Employee

Employee_ID	Employee_Name	Salary
E101	Andy	1000
E102	Bob	750
E103	Christian	400
E104	Adam	600
E105	Bill	1500
E106	John	300
E107	Helle	1200
E108	Jimmy	350
E109	Will	800
E110	Tommy	1150

Round-Robin

Employee1

D1

Employee_ID	Employee_Name	Salary
E101	Andy	1000
E104	Adam	600
E107	Helle	1200
E110	Tommy	1150

Employee2

D2

Employee_ID	Employee_Name	Salary
E102	Bob	750
E105	Bill	1500
E108	Jimmy	350

Employee3

D3

Employee_ID	Employee_Name	Salary
E103	Christian	400
E106	John	300
E109	Will	800

Parallel External Sort-Merge

D1

Employee1

Employee_ID	Employee_Name	Salary
E101	Andy	1000
E104	Adam	600
E107	Helle	1200
E110	Tommy	1150

P1 →

Employee1

Employee_ID	Employee_Name	Salary
E104	Adam	600
E101	Andy	1000
E110	Tommy	1150
E107	Helle	1200

D1

D2

Employee2

Employee_ID	Employee_Name	Salary
E102	Bob	750
E105	Bill	1500
E108	Jimmy	350

P2 →

Employee2

Employee_ID	Employee_Name	Salary
E108	Jimmy	350
E102	Bob	750
E105	Bill	1500

D2

D3

Employee3

Employee_ID	Employee_Name	Salary
E103	Christian	400
E106	John	300
E109	Will	800

P3 →

Employee3

Employee_ID	Employee_Name	Salary
E106	John	300
E103	Christian	400
E109	Will	800

D3

Parallel External Sort-Merge

Employee1

	Employee_ID	Employee_Name	Salary
	E104	Adam	600
D1	E101	Andy	1000
	E110	Tommy	1150
	E107	Helle	1200

Employee2

	Employee_ID	Employee_Name	Salary
	E108	Jimmy	350
D2	E102	Bob	750
	E105	Bill	1500

Employee3

	Employee_ID	Employee_Name	Salary
	E106	John	300
D3	E103	Christian	400
	E109	Will	800

P1

[500, 900]
Range-Partition

Employee1

Employee_ID	Employee_Name	Salary	
			D1

Employee2

Employee_ID	Employee_Name	Salary	
E104	Adam	600	D2

Employee3

Employee_ID	Employee_Name	Salary	
E101	Andy	1000	
E110	Tommy	1150	D3
E107	Helle	1200	

Parallel External Sort-Merge

Employee1

D1

Employee_ID	Employee_Name	Salary
E104	Adam	600
E101	Andy	1000
E110	Tommy	1150
E107	Helle	1200

Employee2

D2

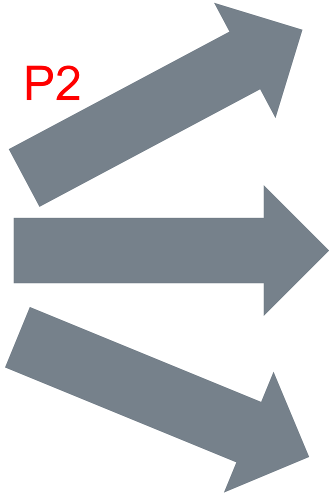
Employee_ID	Employee_Name	Salary
E108	Jimmy	350
E102	Bob	750
E105	Bill	1500

Employee3

D3

Employee_ID	Employee_Name	Salary
E106	John	300
E103	Christian	400
E109	Will	800

P2



[500, 900]
Range-Partition

Employee1

D1

Employee_ID	Employee_Name	Salary
E108	Jimmy	350

Employee2

D2

Employee_ID	Employee_Name	Salary
E104	Adam	600
E102	Bob	750

Employee3

D3

Employee_ID	Employee_Name	Salary
E101	Andy	1000
E110	Tommy	1150
E107	Helle	1200
E105	Bill	1500

Parallel External Sort-Merge

D1

Employee1

Employee_ID	Employee_Name	Salary
E104	Adam	600
E101	Andy	1000
E110	Tommy	1150
E107	Helle	1200

D2

Employee2

Employee_ID	Employee_Name	Salary
E108	Jimmy	350
E102	Bob	750
E105	Bill	1500

D3

Employee3

Employee_ID	Employee_Name	Salary
E106	John	300
E103	Christian	400
E109	Will	800

[500, 900]

Range-Partition

P3

Employee1

Employee_ID	Employee_Name	Salary
E108	Jimmy	350
E106	John	300
E103	Christian	400

Employee2

Employee_ID	Employee_Name	Salary
E104	Adam	600
E102	Bob	750
E109	Will	800

Employee3

Employee_ID	Employee_Name	Salary
E101	Andy	1000
E110	Tommy	1150
E107	Helle	1200
E105	Bill	1500

Parallel External Sort-Merge

Employee1

Employee_ID	Employee_Name	Salary
E108	Jimmy	350
E106	John	300
E103	Christian	400

P1

Employee1

Employee_ID	Employee_Name	Salary
E106	John	300
E108	Jimmy	350
E103	Christian	400

Employee2

Employee_ID	Employee_Name	Salary
E104	Adam	600
E102	Bob	750
E109	Will	800

P2

Employee2

Employee_ID	Employee_Name	Salary
E104	Adam	600
E102	Bob	750
E109	Will	800

Employee3

Employee_ID	Employee_Name	Salary
E101	Andy	1000
E110	Tommy	1150
E107	Helle	1200
E105	Bill	1500

P3

Employee3

Employee_ID	Employee_Name	Salary
E101	Andy	1000
E110	Tommy	1150
E107	Helle	1200
E105	Bill	1500

Parallel External Sort-Merge

Employee1

Employee_ID	Employee_Name	Salary
E106	John	300
E108	Jimmy	350
E103	Christian	400

Employee2

Employee_ID	Employee_Name	Salary
E104	Adam	600
E102	Bob	750
E109	Will	800

Employee3

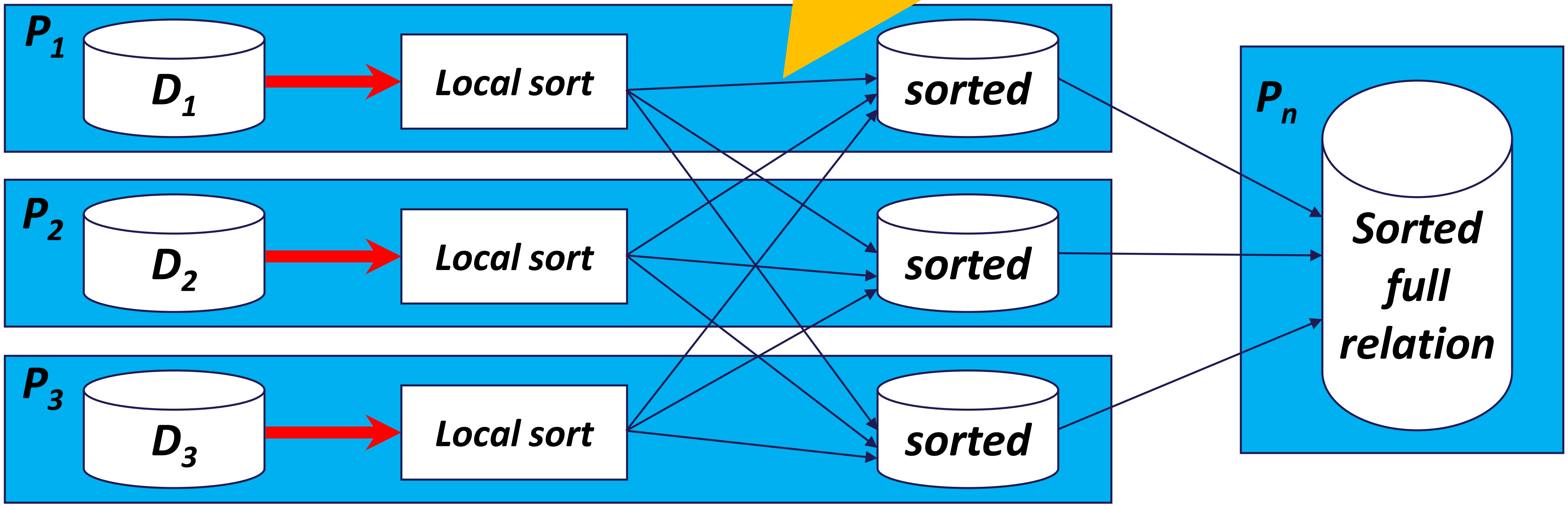
Employee_ID	Employee_Name	Salary
E101	Andy	1000
E110	Tommy	1150
E107	Helle	1200
E105	Bill	1500

Employee

Employee_ID	Employee_Name	Salary
E106	John	300
E108	Jimmy	350
E103	Christian	400
E104	Adam	600
E102	Bob	750
E109	Will	800
E101	Andy	1000
E110	Tommy	1150
E107	Helle	1200
E105	Bill	1500

Illustration of Parallel External Sort-Merge

- All processors send the 1st partition into P_1, then all send the 2nd partition to P_2, and so on so forth.
- Each P_i does a merge upon receiving the data from the others, making sure the local dataset is sorted.



Assume the relation has already been partitioned

Parallel Sort | Range Partitioning | Merge (concatenation)

Parallel Join

- The join operation requires pairs of tuples to be tested to see if they satisfy the join condition, and if they do, the pair is added to the join output.
- Parallel join algorithms attempt to split the pairs to be tested over several processors. Each processor then computes part of the join locally.
- In a final step, the results from each processor can be collected together to produce the final result.

Partitioned Parallel Join

- For **equi-joins** and **natural joins**, it is possible to *partition* the two input relations across the processors, and compute the join locally at each processor.
- Let *r* and *s* be the input relations, and we want to compute $r \bowtie_{r.A=s.B} s$.

1. Relations *r* and *s* each are partitioned into *n* partitions, denoted as $r_0, r_1, ..., r_{n-1}$ and $s_0, s_1, ..., s_{n-1}$.
   - Can use either **range partitioning** or **hash partitioning**.
   - *r* and *s* must be partitioned on their join attributes *r*.A and *s*.B, using the ***same*** range-partitioning vector or hash function.
2. Partitions r_i and s_i are sent to processor P_i.
3. Each processor P_i locally computes $r_i \bowtie_{ri.A=si.B} s_i$.
   - Any of the standard join methods can be used.
4. The final result is the union of all local results.

Partitioned Parallel Join

Student

Semester	Student_ID	Student_Name	Gender
1	S101	Adam	M
3	S105	Christian	M
4	S107	Helle	F
2	S110	Thomas	M
3	S103	Kim	F

Course

Semester	Course_ID	Course_Name
4	C1	Data-intensive Systems
2	C2	Data Mining
3	C6	Advanced Algorithms
1	C11	Machine Intelligence

- Relation: **Student** and **Course**
- Joining on **Semester**
- No of disks = 2 ⇒ No of partitions = 2
- Partition: Hash partition on **Semester** by (Semester *mod* 2)
- Apply hash partition on both relations

Partitioned Parallel Join

D0

Student0

Semester	Student_ID	Student_Name	Gender
4	S107	Helle	F
2	S110	Thomas	M

Course0

Semester	Course_ID	Course_Name
4	C1	Data-intensive Systems
2	C2	Data Mining

D1

Student1

Semester	Student_ID	Student_Name	Gender
1	S101	Adam	M
3	S105	Christian	M
3	S103	Kim	F

Course1

Semester	Course_ID	Course_Name
3	C6	Advanced Algorithms
1	C11	Machine Intelligence

Partitioned Parallel Join

D0

P0

Student0

Semester	Student_ID	Student_Name	Gender
4	S107	Helle	F
2	S110	Thomas	M

Course0

Semester	Course_ID	Course_Name
4	C1	Data-intensive Systems
2	C2	Data Mining

$r0 \bowtie_{r.A=s.B} s0.$

D1

P1

Student1

Semester	Student_ID	Student_Name	Gender
1	S101	Adam	M
3	S105	Christian	M
3	S103	Kim	F

Course1

Semester	Course_ID	Course_Name
3	C6	Advanced Algorithms
1	C11	Machine Intelligence

$r1 \bowtie_{r.A=s.B} s1.$

Partitioned Parallel Join

D0

Semester	Student_ID	Student_Name	Gender	Course_ID	Course_Name
4	S107	Helle	F	C1	Data-intensive Systems
2	S110	Thomas	M	C2	Data Mining

D1

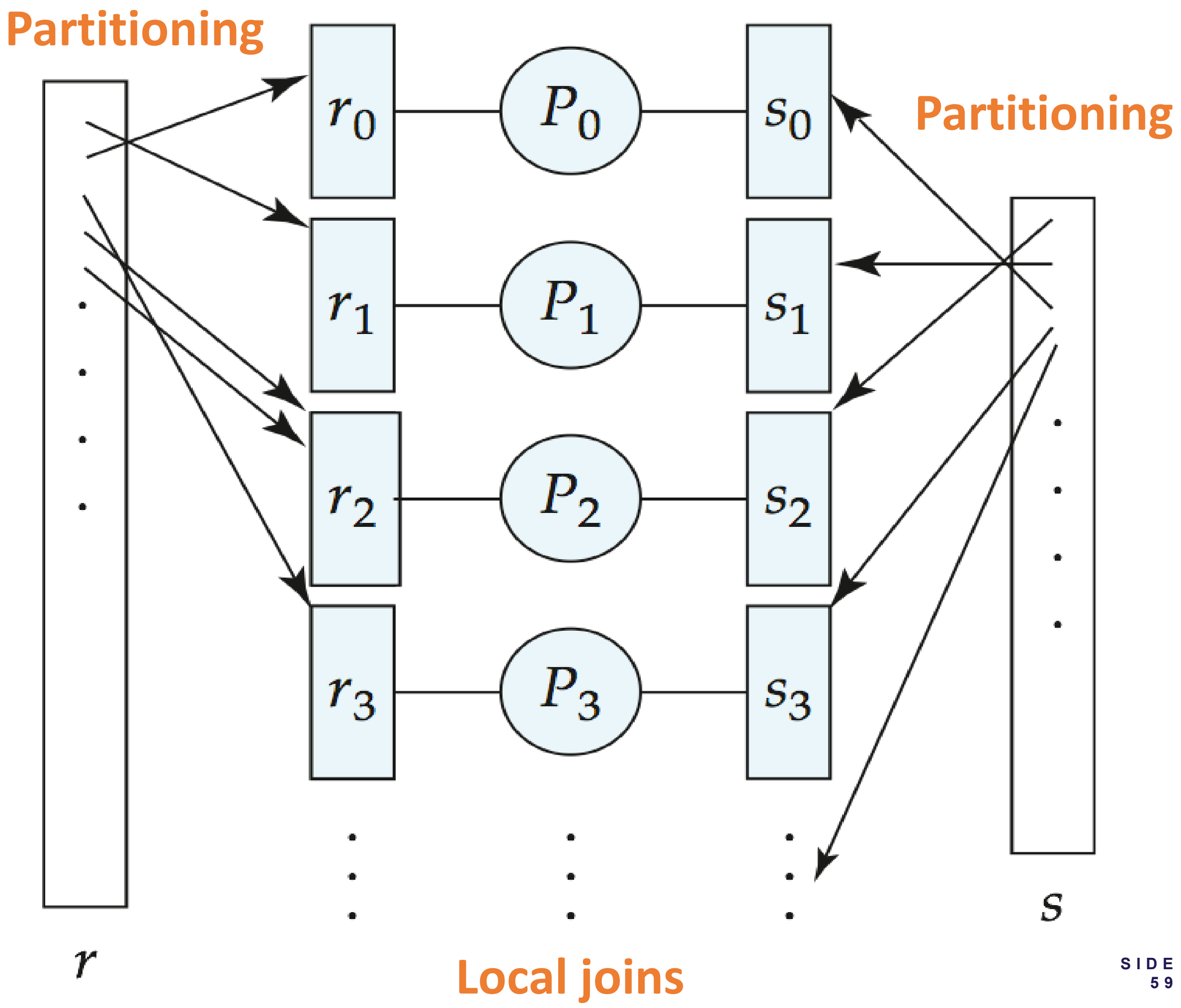
Semester	Student_ID	Student_Name	Gender	Course_ID	Course_Name
1	S101	Adam	M	C11	Machine Intelligence
3	S105	Christian	M	C6	Advanced Algorithms
3	S103	Kim	F	C6	Advanced Algorithms

Partitioned Parallel Join

Semester	Student_ID	Student_Name	Gender	Course_ID	Course_Name
4	S107	Helle	F	C1	Data-intensive Systems
2	S110	Thomas	M	C2	Data Mining
1	S101	Adam	M	C11	Machine Intelligence
3	S105	Christian	M	C6	Advanced Algorithms
3	S103	Kim	F	C6	Advanced Algorithms

Illustration of Partitioned Parallel Join

- Equi-join or natural join
- Partition each relation
  - The *same* range partitioning or hash partitioning
    - Why the same?
    - Why not round-robin?
- Each processor receives 'matching' subsets of the two relations
- Parallelism
  - The two partitioning operations
  - All local joins



Fragment-and-Replicate Join

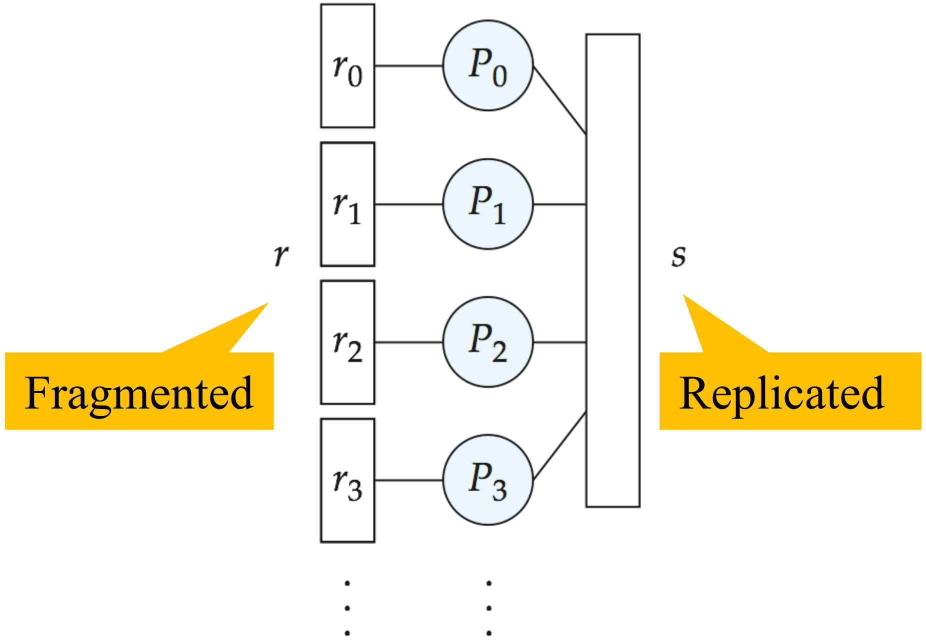
- Partitioning not possible for some join conditions
  - E.g., *non-equijoin* conditions, such as r.A > s.B.
- For joins where partitioning is not applicable, parallelization can be accomplished by **fragment and replicate** technique
  - To be explained on next slide
- Special case – **asymmetric fragment-and-replicate**:
  - One of the relations, say *r*, is partitioned.
    - Any partitioning technique can be used.
  - The other relation, *s*, is replicated across *all* the processors.
  - Processor P_i then locally computes the join of r_i with all of s using any join technique.

Fragment-and-Replicate Join: General Case

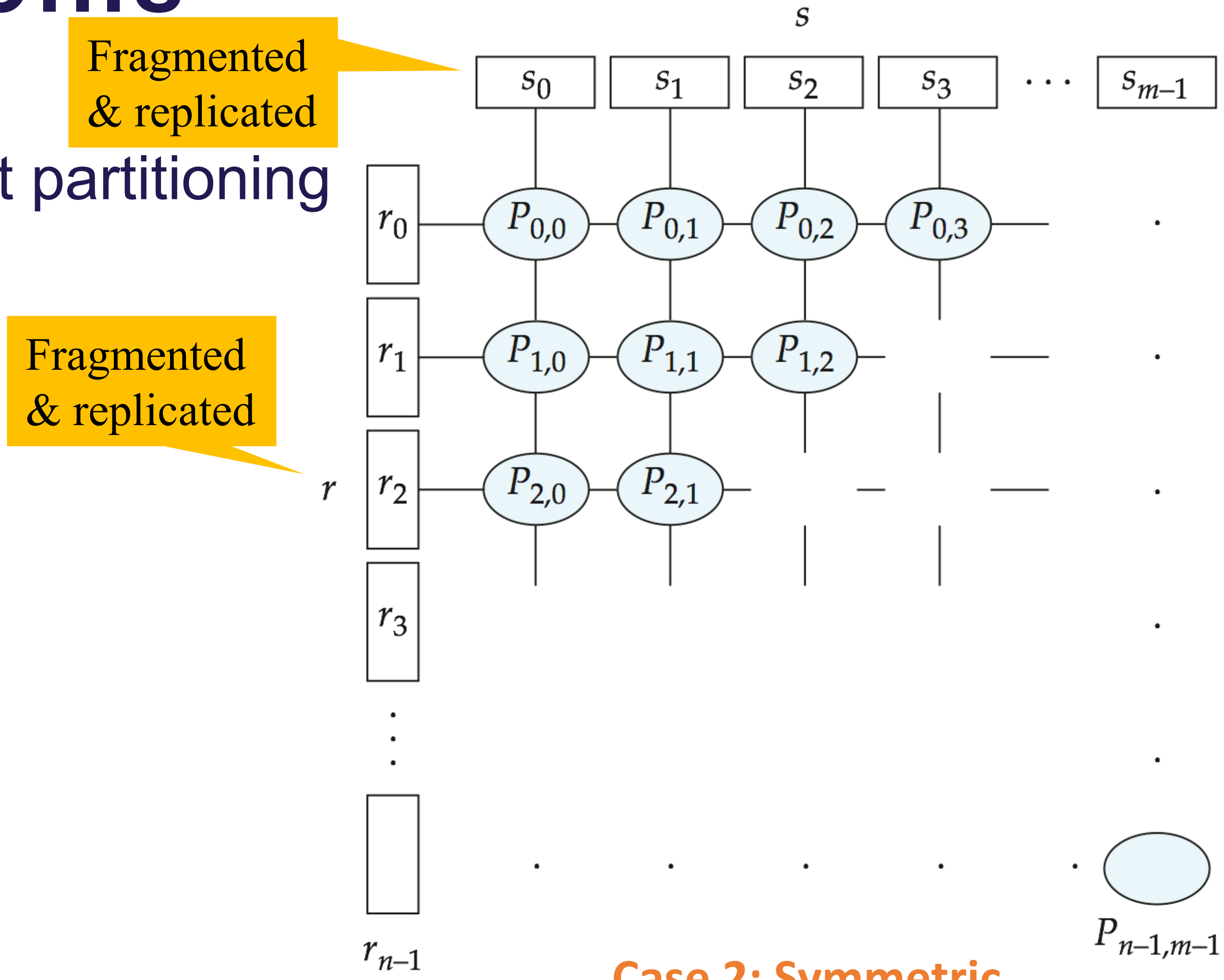
- General case: reduces the sizes of the relations at each processor.
  - r is partitioned into n partitions: $r_0, r_1, \ldots, r_{n-1}$; s is partitioned into m partitions: $s_0, s_1, \ldots, s_{m-1}$.
  - Any partitioning technique may be used.
  - There must be at least m*n processors.
  - Label the processors as $P_{0,0}, P_{0,1}, \ldots, P_{0,m-1}, P_{1,0}, \ldots, P_{n-1m-1}$.
  - $P_{i,j}$ computes the join of r_i with s_j. In order to do so, r_i is replicated to $P_{i,0}, P_{i,1}, \ldots, P_{i,m-1}$, while s_j is replicated to $P_{0,j}, P_{1,j}, \ldots, P_{n-1,j}$
  - Any join technique can be used at each processor $P_{i,j}$.

Illustration of Fragment-and-Replicate Joins

- Join conditions don’t support partitioning
- Two cases



Case 1: Asymmetric



Case 2: Symmetric

Fragment-and-Replicate Join

Person

id	Name	rooms	min_price	max_price	apartment_id
1	Anne	2	40,000	1,50,000	2
2	John	1	20,000	50,000	2
3	Michael	2	2,00,000	3,00,000	6
4	Oliver	4	30,000	1,00,000	7

Apartment

id	rooms	price	city
1	2	30,000	Houston
2	2	45,000	Dallas
3	3	1,25,000	Chicago
4	5	2,45,000	Los Angeles
5	4	3,40,000	San Jose
6	4	2,20,000	San Diego
7	1	36,000	Cleveland

```
SELECT Name, min_price, max_price, price, city
FROM person JOIN apartment ON apartment.id !=person.apartment_id
AND price BETWEEN min_price AND max_price;
```

Fragment-and-Replicate Join

id	Name	rooms	min_price	max_price	apartment_id
1	Anne	2	40,000	1,50,000	2
2	John	1	20,000	50,000	2
3	Michael	2	2,00,000	3,00,000	6
4	Oliver	4	30,000	1,00,000	7

Fragmentation

id	Name	rooms	min_price	max_price	apartment_id
2	John	1	20,000	50,000	2
4	Oliver	4	30,000	1,00,000	7

P0

id	Name	rooms	min_price	max_price	apartment_id
1	Anne	2	40,000	1,50,000	2
3	Michael	2	2,00,000	3,00,000	6

P1

Replication

id	rooms	price	city
1	2	30,000	Houston
2	2	45,000	Dallas
3	3	1,25,000	Chicago
4	5	2,45,000	Los Angeles
5	4	3,40,000	San Jose
6	4	2,20,000	San Diego
7	1	36,000	Cleveland

Fragment-and-Replicate Join

Person

id	Name	rooms	min_price	max_price	apartment_id
1	Anne	2	40,000	1,50,000	2
2	John	1	20,000	50,000	2
3	Michael	2	2,00,000	3,00,000	6
4	Oliver	4	30,000	1,00,000	7

Apartment

id	rooms	price	city
1	2	30,000	Houston
2	2	45,000	Dallas
3	3	1,25,000	Chicago
4	5	2,45,000	Los Angeles
5	4	3,40,000	San Jose
6	4	2,20,000	San Diego
7	1	36,000	Cleveland

```
SELECT Name, min_price, max_price, price, city
FROM person JOIN apartment ON apartment.id !=person.apartment_id
AND price BETWEEN min_price AND max_price;
```

Name	min_price	max_price	price	city
John	20,000	50,000	30,000	Houston
John	20,000	50,000	36,000	Cleveland
Anne	40,000	1,50,000	1,25,000	Chicago
Michael	2,00,000	3,00,000	2,45,000	Los Angeles
Oliver	30,000	1,00,000	45,000	Dallas
Oliver	30,000	1,00,000	30,000	Houston

Fragment-and-Replicate Join

id	rooms	price	city
1	2	30,000	Houston
2	2	45,000	Dallas
3	3	1,25,000	Chicago
4	5	2,45,000	Los Angeles
5	4	3,40,000	San Jose
6	4	2,20,000	San Diego
7	1	36,000	Cleveland

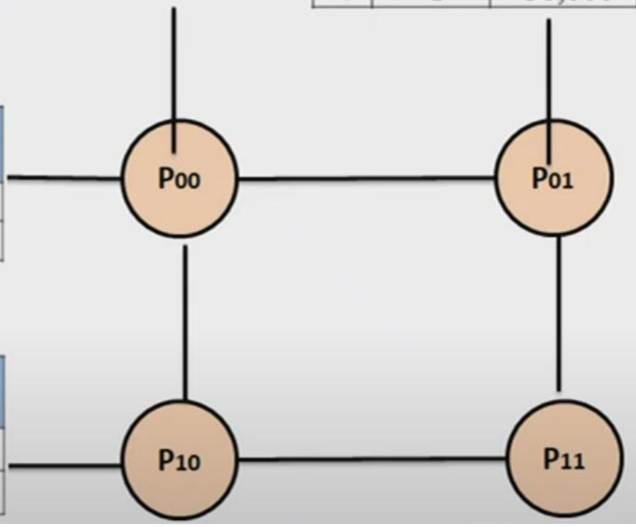
id	rooms	price	city
2	2	45,000	Dallas
4	5	2,45,000	Los Angeles
6	4	2,20,000	San Diego

id	rooms	price	city
1	2	30,000	Houston
3	3	1,25,000	Chicago
5	4	3,40,000	San Jose
7	1	36,000	Cleveland

id	Name	rooms	min_price	max_price	apartment_id
1	Anne	2	40,000	1,50,000	2
2	John	1	20,000	50,000	2
3	Michael	2	2,00,000	3,00,000	6
4	Oliver	4	30,000	1,00,000	7

id	Name	rooms	min_price	max_price	apartment_id
2	John	1	20,000	50,000	2
4	Oliver	4	30,000	1,00,000	7

id	Name	rooms	min_price	max_price	apartment_id
1	Anne	2	40,000	1,50,000	2
3	Michael	2	2,00,000	3,00,000	6



Fragment-and-Replicate Join

Person

id	Name	rooms	min_price	max_price	apartment_id
1	Anne	2	40,000	1,50,000	2
2	John	1	20,000	50,000	2
3	Michael	2	2,00,000	3,00,000	6
4	Oliver	4	30,000	1,00,000	7

Apartment

id	rooms	price	city
1	2	30,000	Houston
2	2	45,000	Dallas
3	3	1,25,000	Chicago
4	5	2,45,000	Los Angeles
5	4	3,40,000	San Jose
6	4	2,20,000	San Diego
7	1	36,000	Cleveland

```
SELECT Name, min_price, max_price, price, city
FROM person JOIN apartment ON apartment.id !=person.apartment_id
AND price BETWEEN min_price AND max_price;
```

Name	min_price	max_price	price	city
John	20,000	50,000	30,000	Houston
John	20,000	50,000	36,000	Cleveland
Anne	40,000	1,50,000	1,25,000	Chicago
Michael	2,00,000	3,00,000	2,45,000	Los Angeles
Oliver	30,000	1,00,000	45,000	Dallas
Oliver	30,000	1,00,000	30,000	Houston

Fragment-and-Replicate Join: Notes

- Both versions of fragment-and-replicate work with *any* join condition, since every tuple in *r* can be tested with *every* tuple in *s*.
- Usually, this join has a higher cost than partitioned parallel join, since one of the relations (*for asymmetric fragment-and-replicate*) or both relations (*for general fragment-and-replicate*) must be replicated.
- Sometimes asymmetric fragment-and-replicate is preferable even though partitioning could be used.
  - E.g., say *s* is small, and *r* is large and already partitioned. It may be cheaper to replicate *s* across all processors, rather than repartition *r* and *s* on the join attributes.

Partitioned Parallel Hash Join

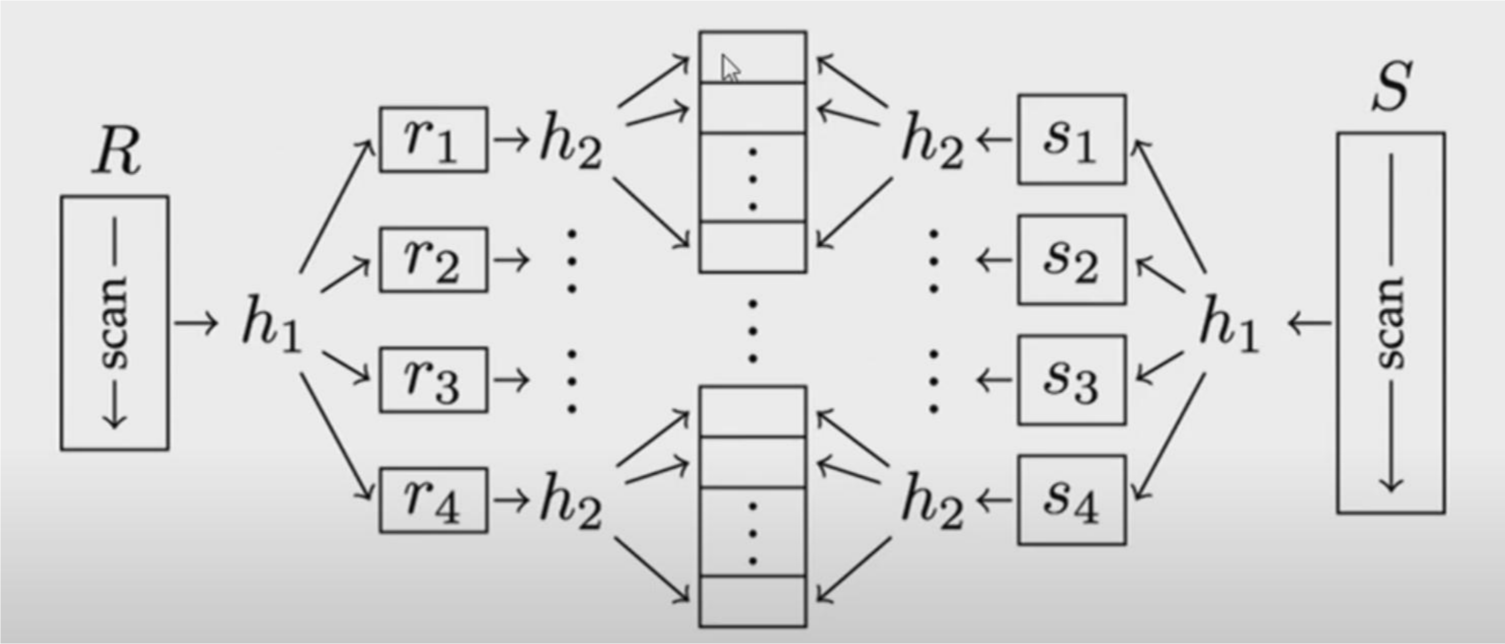
- Assume s is smaller than r.
- Therefore, s is chosen as the *build* relation, and r *probe* relation.
- A hash function h_1 takes the join attribute value of each tuple in s and maps this tuple to one of the n processors.

1. Each processor P_i reads the tuples of s that are on its disk D_i and sends each tuple to the appropriate processor based on hash function h_1. Let s_i denote the tuples of relation s that are sent to processor P_i.
2. As tuples of relation s are received at the destination processors, they are partitioned further using another hash function, h_2, which is used to compute the hash-join locally. *(to be continued)*

Partitioned Parallel Hash Join, cont.

3. Once the tuples of *s* have been distributed, the larger relation *r* is also distributed across the *m* processors using hash function h_1
   - Let r_i denote the tuples of relation *r* that are sent to processor P_i.
4. As the *r* tuples are received at the destination processors, they are repartitioned using the function h_2
   - (just as the probe relation is partitioned in the sequential hash-join algorithm).
5. Each processor P_i executes the build and probe phases of the hash-join algorithm on the local partitions r_i and s_i of *r* and *s* to produce a partition of the final result of the hash-join.

Partitioned Parallel Hash Join, cont.



Example of Partitioned Parallel Hash Join

- $R \bowtie_{R.A=T.A} T$

Hash partitioning:
$h_1(A) = A \bmod 3$
$h_2(A) = A \bmod 4$

R	A	B
	1	a
	2	b
	4	a
	7	a
	8	a
	9	b

T	A	C
	1	x
	2	x
	3	x
	7	z
	8	w
	9	y

P₁

R1: (1, a), (4, a), (7, a)

T1: (1, x), (7, z)

$R1 \bowtie T1$: (1, a, x), (7, a, z)

P₂

R2: (2, b), (8, a)

T2: (2, x), (8, w)

$R2 \bowtie T2$: (2, b, x), (8, a, w)

P₃

R3: (9, b)

T3: (3, x), (9, y)

$R3 \bowtie T3$: (9, b, y)

Parallel local joins (each node uses a local hash function h_2)

Parallel Nested-Loop Join

- Assume that
  - relation *s* is much smaller than relation *r*
  - relation *r* is stored by partitioning
  - (*optional*) there is an index on a join attribute of relation *r* at each of its partitions
- Use asymmetric fragment-and-replicate, with relation *s* being replicated, and using the existing partitioning of relation *r*.

1. Each processor P_j where a partition of relation *s* is stored reads the tuples of relation *s* stored in D_j and replicates the tuples to every other processor P_i.
   - At the end of this phase, relation *s* is replicated at all sites that store tuples of relation *r*.
2. Each processor P_i performs an (indexed) nested-loop join of relation *s* with the i^{th} (indexed) partition of relation *r*.

Example of Parallel Nested-Loop Join

- R $\triangleright\triangleleft_{R.B=S.B}$ S
- |R|>>|S|

Fragmented

R	A	B
	1	a
	2	b
	4	a
	7	a
	8	a
	9	b

Replicated

S	B	C
	a	x
	b	x
	a	y
	d	z
	a	w
	c	y

R1 on P_1: (1, a), (7, a)

R2 on P_2: (2, b), (8, a)

R3 on P_3: (4, a), (9, b)

S: (a, x), (b, x), (a, y), (d, z), (a, w), (c, y)

Parallel local nested-loop joins

R1 $\triangleright\triangleleft$ S

(1, a, x), (7, a, x)
(1, a, y), (7, a, y)
(1, a, w), (7, a, w)

R2 $\triangleright\triangleleft$ S

(8, a, x), (2, b, x)
(8, a, y), (8, a, w)

R3 $\triangleright\triangleleft$ S

(4, a, x), (9, b, x)
(4, a, y), (4, a, w)

Summary

- Parallel Database Architectures
  - Shared Nothing
  - Shared Memory
  - Shared Disk
  - Hierarchical
- IO Parallelism and Partitioning
  - Round-robin partitioning
  - Hash partitioning
  - Range partitioning
- Other Types of Parallelism
  - Parallel sort algorithms
  - Parallel join algorithms

Readings

- Mandatory readings (for both Lectures 2 and 3)
  - A. Silberschatz, H. F. Korth, S. Sudarshan: Database System Concepts (7th edition), McGraw-Hill. Chapter 20, 21, 22, 23.
    - Among these chapters, the following sections are *optional*:
      21.3, 21.5, 22.6, 22.7, 22.8, 23.4, 23.5, 23.6, 23.7, 23.8

Exercises

1. We want to count *each* hashtag that appears in the ID-Hashtag file (in Moodle). If you have a shared-nothing cluster with m processors, how can that help to speed up the counting? Describe your data partitioning strategy and counting algorithm.
2. If we only want to know the frequency of a given specific hashtag, how can you make use of the shared-nothing architecture? Describe your data partitioning strategy and counting algorithm.

For each exercises:

- Suppose one of the m processors is designated as the 'master' node
  - It has the original data file, and it needs to store the final result.
  - It can 'tell' all other nodes what to do in some form of message.
- You're encouraged to also write codes (in Python or another language) to simulate the parallelism of your algorithmic solutions.