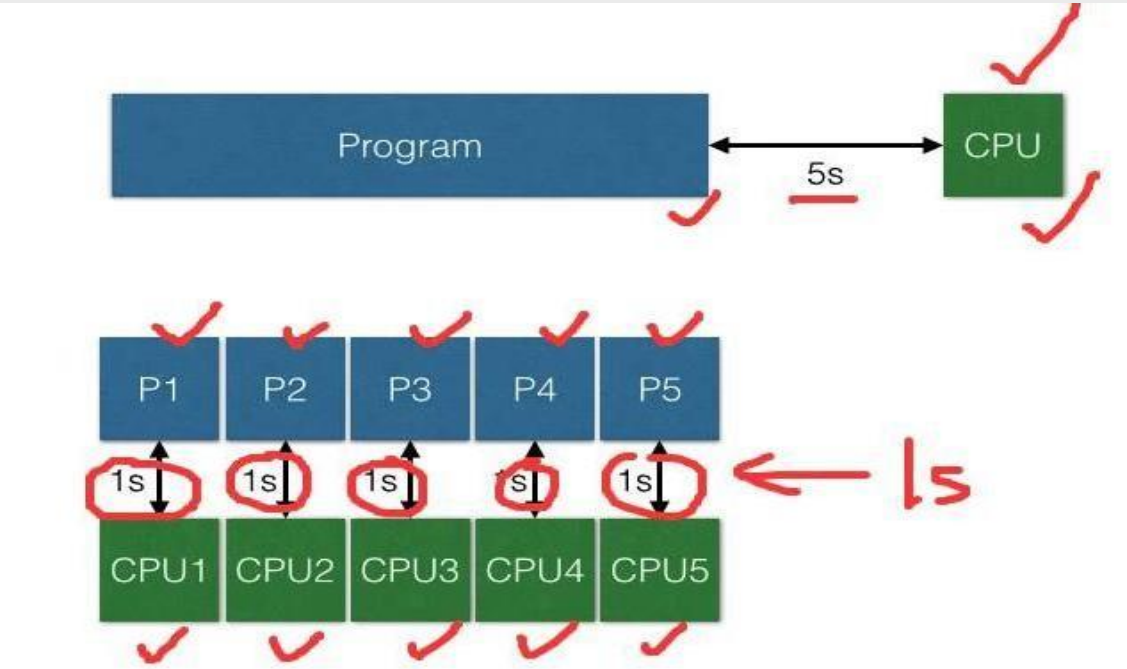
Parallel Databases

Chapter 21: Parallel Databases

- Introduction
- I/O Parallelism
- Interquery Parallelism
- Intraquery Parallelism
- Intraoperation Parallelism
- Interoperation Parallelism
- Design of Parallel Systems

Parallel Systems

- Parallel database systems consist of multiple processors and multiple disks connected by a fast interconnection network.
- The driving force behind parallel database systems is the demands of applications that have to query extremely large databases or that have to process an extremely large number of transactions per second
- A **coarse-grain parallel** machine consists of a small number of powerful processors
- A **massively parallel** or **fine grain parallel** machine utilizes thousands of smaller processors.



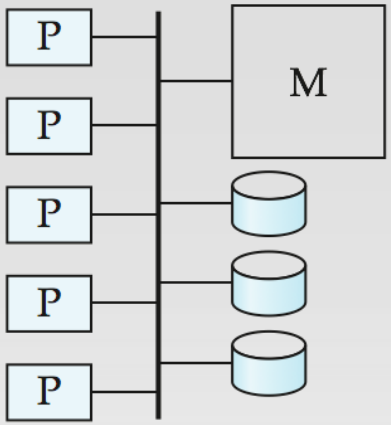
Parallel Systems

- Two main performance measures:
  - **throughput** --- the number of tasks that can be completed in a given time interval
  - **response time** --- the amount of time it takes to complete a single task from the time it is submitted

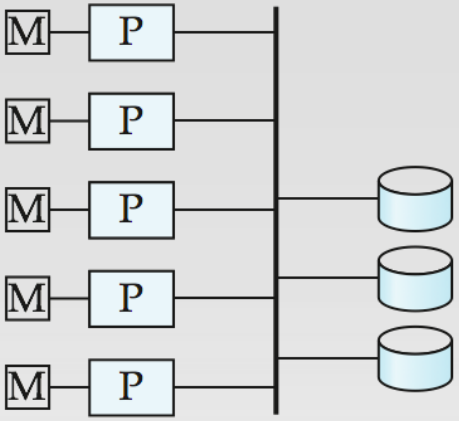
Parallel Database Architectures

- **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

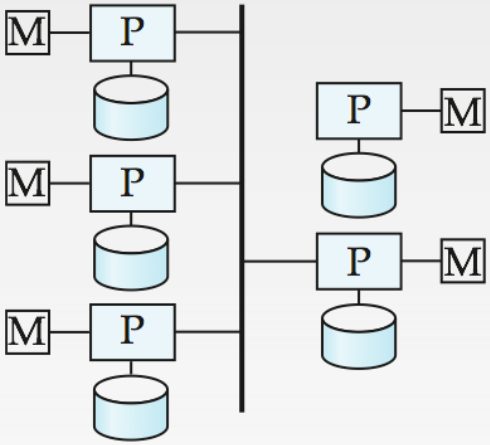
Parallel Database Architectures



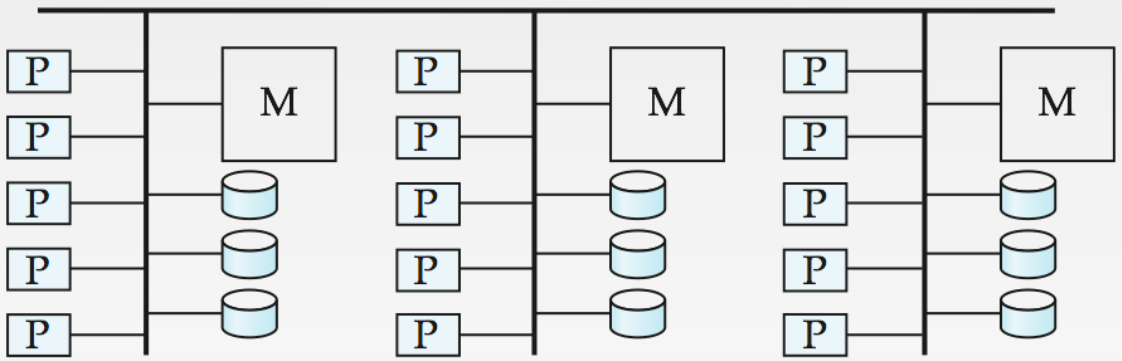
(a) shared memory



(b) shared disk



(c) shared nothing



(d) hierarchical

Shared Memory

- Processors and disks have access to a common memory, typically via a bus or through an interconnection network.
- 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 interconnection network becomes a bottleneck
- Widely used for lower degrees of parallelism.

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
  - Architecture provides 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.
- Examples: IBM Sysplex and DEC clusters (now part of Compaq) running Rdb (now Oracle Rdb) were early commercial users
- Downside: bottleneck now occurs at interconnection to the disk subsystem when the database makes a large number of accesses to disks.
- Shared-disk systems can scale to a somewhat larger number of processors, but communication between processors is slower.

Shared Nothing

- Node consists of a processor, memory, and one or more disks. Processors at one node communicate with another processor at another node using an interconnection network. A node functions as the server for the data on the disk or disks the node owns.
- Examples: Teradata, Tandem, Oracle-n CUBE
- Less I/O Overhead: Data accessed from local disks (and local memory accesses) do not pass through interconnection network, thereby minimizing the interference of resource sharing. Only queries, accesses to nonlocal disks, and result relations pass through the network.
- Shared-nothing multiprocessors can be scaled up to thousands of processors without interference.
- Main drawback: 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 of the system could be a shared-memory system with a few processors.
- In figure above, a hierarchical architecture with shared-memory nodes connected together in a shared-nothing architecture.

Why Parallel Database ?

- Parallel machines are becoming quite common and affordable
  - Prices of microprocessors, memory and disks have dropped sharply
  - Recent desktop computers feature multiple processors and this trend is projected to accelerate
- Databases are growing increasingly large
  - large volumes of transaction data are collected and stored for later analysis.
  - multimedia objects like images are increasingly stored in databases
- Large-scale parallel database systems increasingly used for:
  - storing large volumes of data
  - processing time-consuming decision-support queries
  - providing high throughput for transaction processing

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.
- Thus, databases naturally lend themselves to parallelism.

I/O Parallelism

- Refers to, reducing 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.
- Partitioning techniques (number of disks = *n*): The same set of operations is performed on tuples partitioned across the disks. The set of operations are done in parallel on partitions of the data.

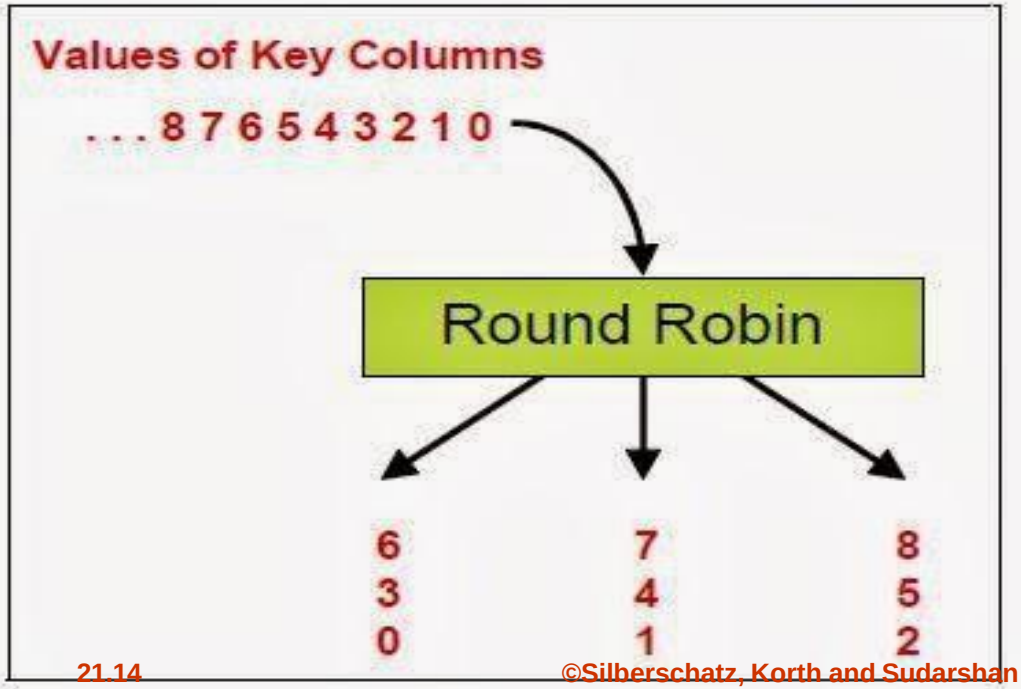
I/O Parallelism

Partitioning Techniques:

1. Round robin
2. Hash
3. Range

1. Round-robin:

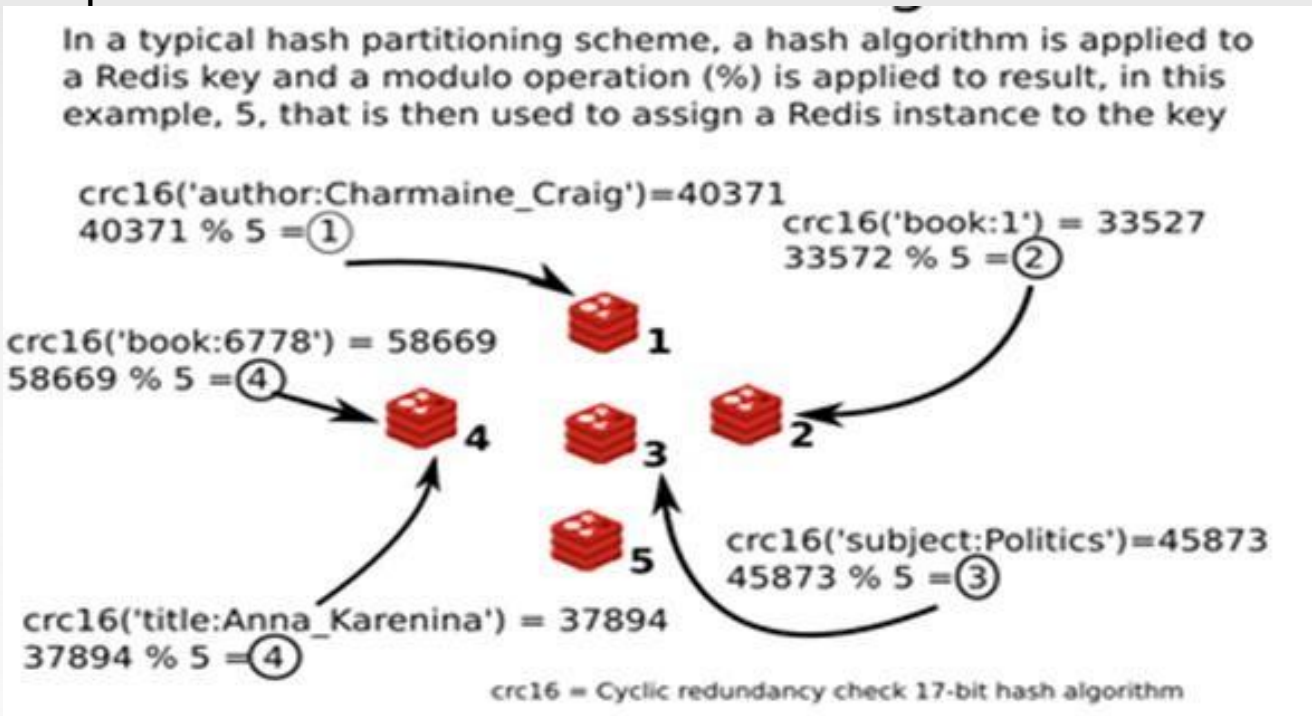
Send the i^{th} tuple inserted in the relation to disk $i \bmod n$.



I/O Parallelism

2. Hash partitioning:

- Uses a hash function to distribute the records into many disks based on the input values (partitioning attribute values).
- Choose one or more attributes as the partitioning attributes.
- Choose hash function *h* with range 0…*n* - 1
- Let *i* denote result of hash function *h* applied tothe partitioning attribute value of a tuple. Send tuple to disk *i*.



I/O Parallelism (Cont.)

3. Range partitioning:

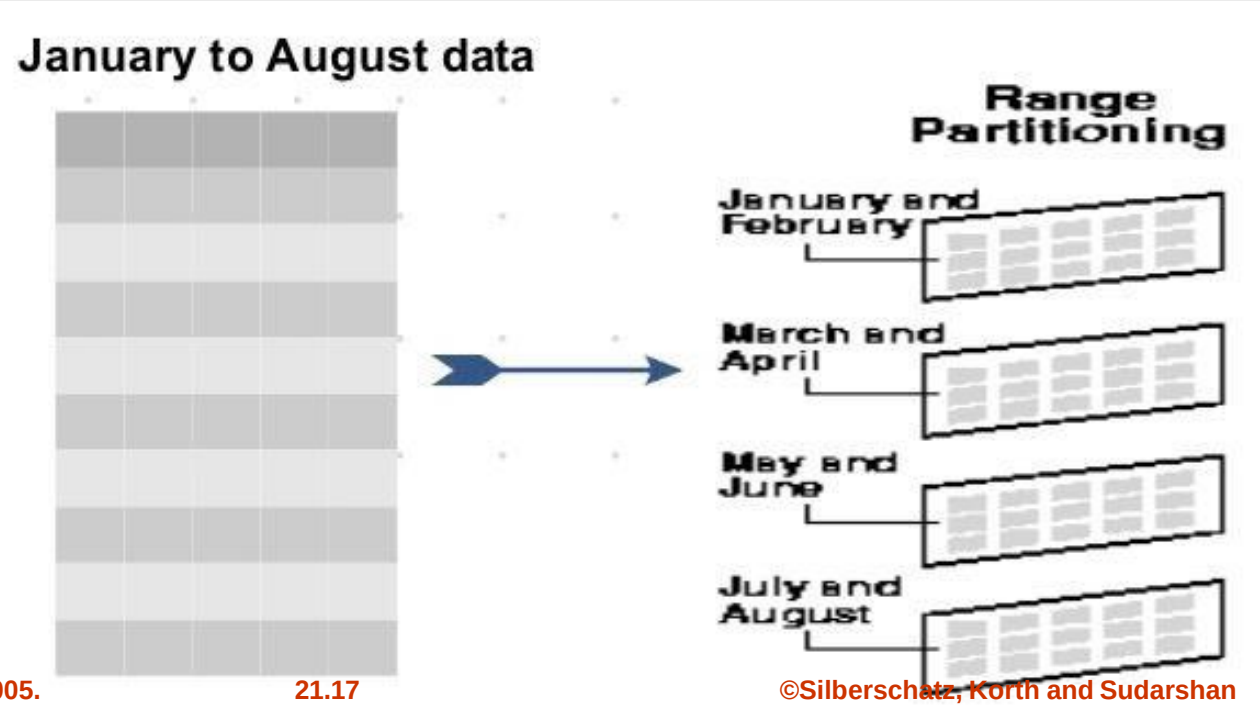
- Range partitioning strategy partitions the data based on the partitioning attributes values.
- We need to find set of range vectors on which we are about to partition.
- For example, the records with Salary range 100 to 5000 will be in disk 1, 5001 to 10000 in disk 2, and so on.

I/O Parallelism (Cont.)

Range partitioning (contd..):

- 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 $v_i \leq v_{i+1}$ 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.



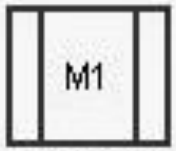
I/O Parallelism (Cont.)

Range partitioning (contd..):

Range : <=14000

Emp_ID	EName	Salary
E102	Kumar	10000
E103	Madhan	5000
E101	Jack	6000
E112	Kesav	10500
E123	Ram	5000
E121	Guhan	7500

Temp_Employee0



Range : >14000 and <=24000

Emp_ID	EName	Salary
E105	Meena	15000
E113	Maddy	15500
E115	Megha	18000

Temp_Employee1

Range : >24000

Emp_ID	EName	Salary
E122	Maya	30000
E111	Ramya	26000
E125	Steve	25000

Temp_Employee2

Comparison of Partitioning Techniques

- Once a relation has been partitioned among several disks, we can retrieve it in parallel, using all the disks. Similarly, when a relation is being partitioned, it can be written to multiple disks in parallel
- Evaluate how well partitioning techniques support the following types of data access:

1. Scanning the entire relation.
2. Locating a tuple associatively – **point queries**.
   - E.g., $r.A = 25$, seek tuples that have a specified value for a specific attribute.
3. Locating all tuples such that the value of a given attribute lies within a specified range – **range queries**.
   - E.g., $10 \leq r.A < 25$.

Comparison of Partitioning Techniques(Cont.)

- Examples:

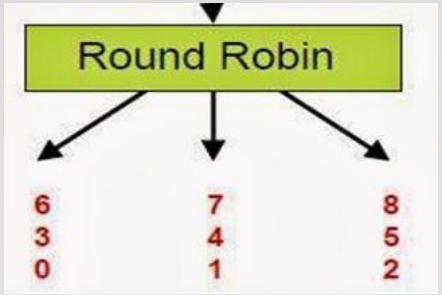
A. SELECT * FROM Employee WHERE Emp_ID = ‘E101’; (Assume Emp_ID is Primary key)

B. SELECT * FROM Employee WHERE EName = ‘Murugan’; (Assume EName is non-key attribute)

C. SELECT * FROM Employee ORDER BY Phone;

D. SELECT * FROM Employee WHERE Salary BETWEEN 10000 AND 20000;

Comparison of Partitioning Techniques (Cont.)



Round robin:

- Advantages

Ex. SELECT * FROM Employee ORDER BY Phone;

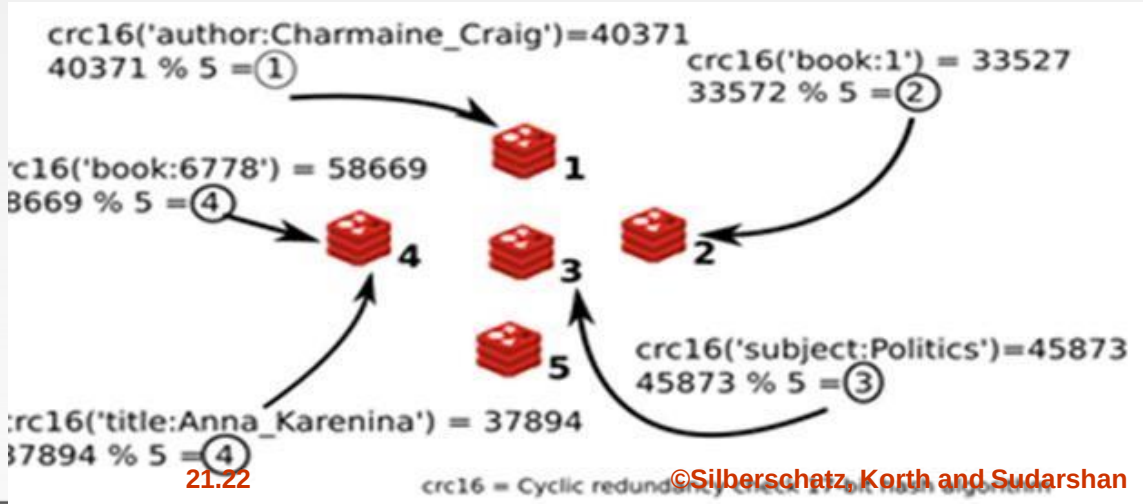
  - Best suited for sequential scan of entire relation on each query.
  - All disks have almost an equal number of tuples; retrieval work is thus well balanced between disks.

- Disadvantages:
  - Range queries and point queries are difficult to process because the distribution is not based on attribute; it was actually based on the record position,
    - For Range queries, all the values falls in the given range may be in all the disks.
    - For Point Queries, cannot identify the exact disk where the records with the specified values would be stored. For Query B, one has to search in all disks.
    - As per ex; Query A, B and D cannot be answered efficiently

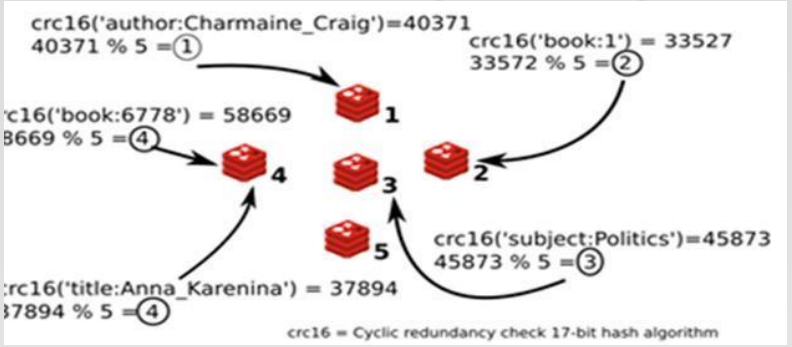
Comparison of Partitioning Techniques(Cont.)

Hash partitioning:

- Best suited for point queries based on the partitioning attribute.
  - Ex: if a relation is partitioned on the telephone_number attribute , then query = —Findthe record of the employee with *telephone number = 555-3333"*
  - Applying the partitioning hash function to 555-3333 and then searching that disk.
- Hash partitioning is also useful for sequential scans of the entire relation.
- Hash-based partitioning is also not well suited for answering range queries.



Comparison of Partitioning Techniques(Cont.)



Hash partitioning:

Advantages

- Good for sequential access

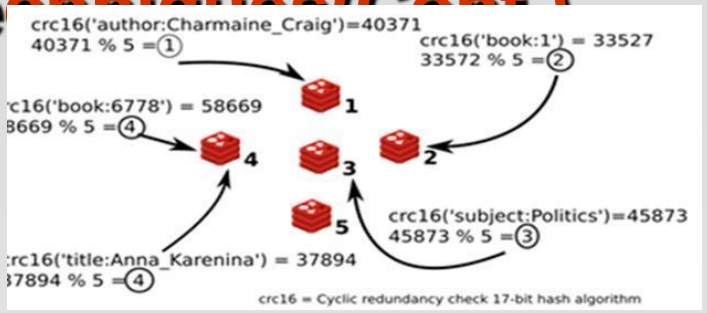
  Ex: SELECT * FROM Employee ORDER BY Phone;

  - The data are available in all the disks, sorting can be done in parallel
  - Retrieval work is then well balanced between disks.
- Good for point queries on partitioning attribute

  Ex: SELECT * FROM Employee WHERE EName = ‘Murugan’;

  - The similar values could be found in one location only
  - 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

Comparison of Partitioning Techniques(Cont.)



Hash partitioning:

- Disadvantages:
  - (prev. point contd..) - Hard to answer point queries on non-partitioning attributes. Consider the following query;

    Ex: SELECT * FROM Employee WHERE EName = 'Murugan';

    - If table is partitioned using Salary as the partitioning attribute using Hash partitioning. Hence, EName is the non-partitioning attribute. So, the queries with Salary in the WHERE clause can be efficiently executed at one processor, all the other queries which do not have Salary in WHERE clause must be executed at the all the processors as we do not know in which partition the value 'Murugan' would be available.
  - No clustering, the data distributed unevenly. so difficult to answer range queries. Hence, to answer Query D, we have to use all the processors.
    - Ex: SELECT * FROM Employee WHERE Salary BETWEEN 10000 AND 20000; (not efficient)

Comparison of Partitioning Techniques (Cont.)

Range partitioning:

Advantages

- Well balanced data distribution is possible, only if good partitioning vector is chosen. One has to choose the partitioning vector which would at least near equally into many disks.
- Good for sequential access and can be done in parallel using all the processors. Query C can be executed efficiently.
  - Ex. SELECT * FROM Employee ORDER BY Phone;
- Good for point queries on partitioning attribute: only one disk needs to be accessed.

Comparison of Partitioning Techniques (Cont.)

Range partitioning:

Advantages (contd..)

- For range queries on partitioning attribute, according to the range given in the query, one to a few disks may need to be accessed. Query D can be executed efficiently
  - Remaining disks are available for other queries.
  - Good if result tuples are from one to a few blocks.
- Higher throughput and good response time.

Comparison of Partitioning Techniques (Cont.)

Range partitioning:

Disadvantages

- If the query specifies large range values, i.e, if the result consists of large number of records in very few disks among the available disk, an I/O bottleneck may be created on those disks.
- This is called Execution skew. If the same query executed in Round-robin or Hash partitioned disks, we could get better performance compared to Range partitioning.
  - Data Skew: The no. of tuples vary widely across the partitions.
  - Execution Skew: Due to data skew, more time is spent to execute an operation on partitions that contains large no. of tuples.

Comparison of Partitioning Techniques (Cont.)

Range partitioning:

Examples:

ENo	EName	DOB	Designation	Salary	
E101	Kumar	14-JUN-1980	Lecturer	10002	R1
E102	Melvin	12-MAR-1975	AP	23000	R2
E103	Ram	10-MAR-1978	AP	22051	R3
E107	Jacob	28-MAY-1970	Professor	30011	R4
E109	Nisha	10-AUG-1981	Lecturer	9031	R5
E105	Shaik	20-DEC-1971	Professor	29002	R6
E104	Vinay	15-SEP-1982	Attender	4004	R7
E106	Menon	30-AUG-1978	AP	21016	R8

Table 1 - Employee

- For above ex, following vector as **range vector**; v [5000, 10000, 25000]
- These vector represents 4 partitions, viz, 5000 and less, 10000 and less, 25000 and less, and the other salary values more than 25000.

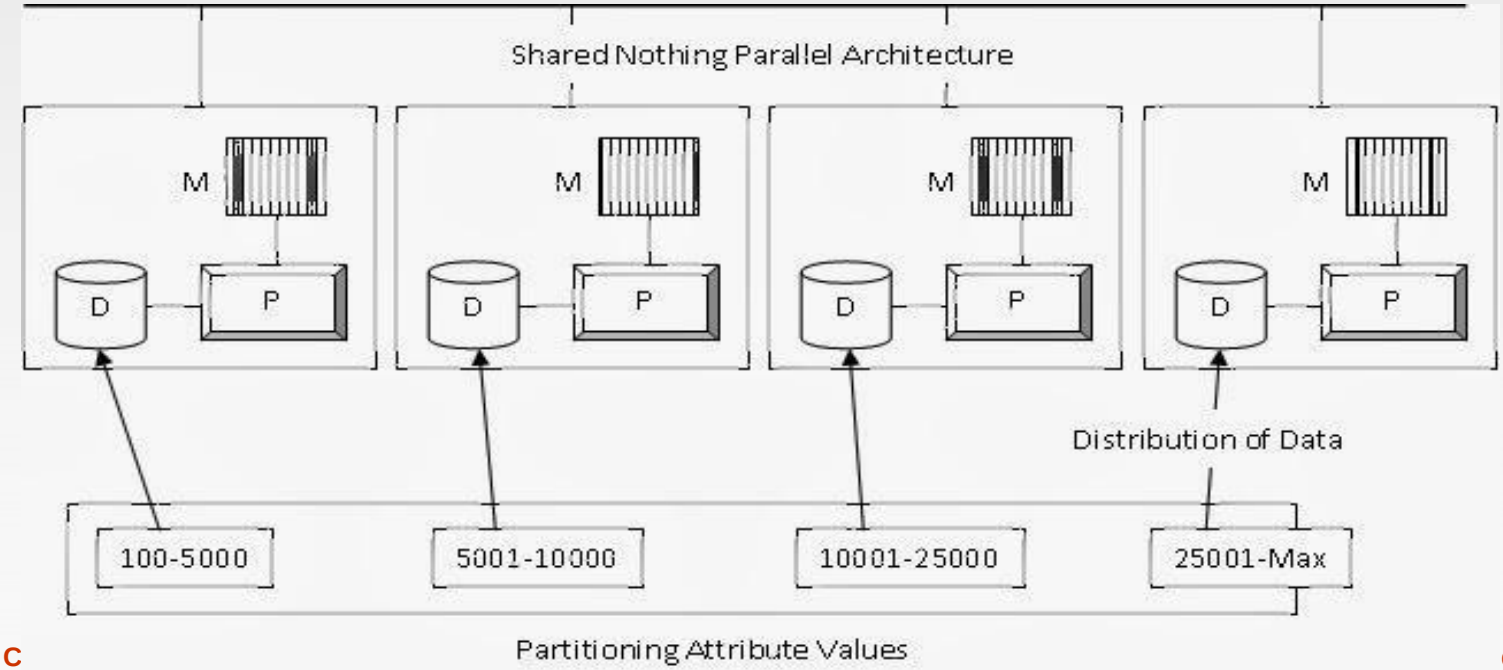
Comparison of Partitioning Techniques (Cont.)

Range partitioning:

Examples:

- For above ex, following vector as **range vector**; v [5000, 10000, 25000]
- For this range vector, data will be distributed as follows;
  - Disk 0 : R7 (only record 7 is <=5000)
  - Disk 1 : R5 (only record 5 is >5000 and <=10000)
  - Disk 2 : R2, R3, R8 (the range >10000 and <=25000)
  - Disk 3 : R4, R6 (the range >25000)

Execution Skew



Comparison of Partitioning Techniques (Cont.)

Range partitioning:

W.r.t. Example:

- **Advantages:**
  - Good for sequential scan. For example, the query,
    - SELECT * FROM Employee ORDER BY Phone; can be executed efficiently by all the processors in parallel.
  - Point queries can be executed efficiently. For example, the query,
    - SELECT * FROM Employee WHERE Salary = 12000; need to be executed by only one of all the available processors.
    - The reason is, the salary value 12000, in our case, belongs to Disk 2. Hence, we need to scan only the records at Disk 2.

Comparison of Partitioning Techniques (Cont.)

Range partitioning:

W.r.t. Example:

- **Advantages:**
  - Range queries can be executed efficiently compared to other partitioning techniques. For example, the query,
    - SELECT * FROM Employee WHERE Salary BETWEEN 10001 AND 20000;
    - This query needs one to few disks to be scanned based on the values specified in the WHERE clause. For our query, we need to scan only Disk 2. Because, the range 10001 and 20000 stored in Disk 2 only.

Comparison of Partitioning Techniques (Cont.)

Range partitioning:

W.r.t. Example:

- **Disadvantages:**
  - Execution Skew might occur. For example, consider the query,
    - SELECT * FROM Employee WHERE Salary BETWEEN 10001 AND 26000;
    - This range falls in Disk 2 and 3. Unfortunately, because of the bad range vectors, disk 2 and 3 have more number of records compared to any other disks. Hence, though the query needs to be executed at Processor 2 and 3, it consumes more time as there are more number of records.

Partitioning a Relation across Disks

- If a relation contains only a few tuples which will fit into a single disk block, then assign the relation to a single disk.
- Large relations are preferably partitioned across all the available disks.
- If a relation consists of *m* disk blocks and there are *n* disks available in the system, then the relation should be allocated **min**(*m*,*n*) disks.

Handling of Skew

- The distribution of tuples to disks may be **skewed** — that is, some disks have many tuples, while others may have fewer tuples.
- **Types of skew:**
  - **Attribute-value skew.**
    - All the tuples with the same value for the partitioning attribute end up in the same partition.
    - Can occur with range-partitioning and hash-partitioning.
  - **Partition skew**.
    - With range-partitioning, badly chosen partition vector may assign too many tuples to some partitions and too few to others.
    - Less likely with hash-partitioning if a good hash-function is chosen.

Handling Skew in Range-Partitioning

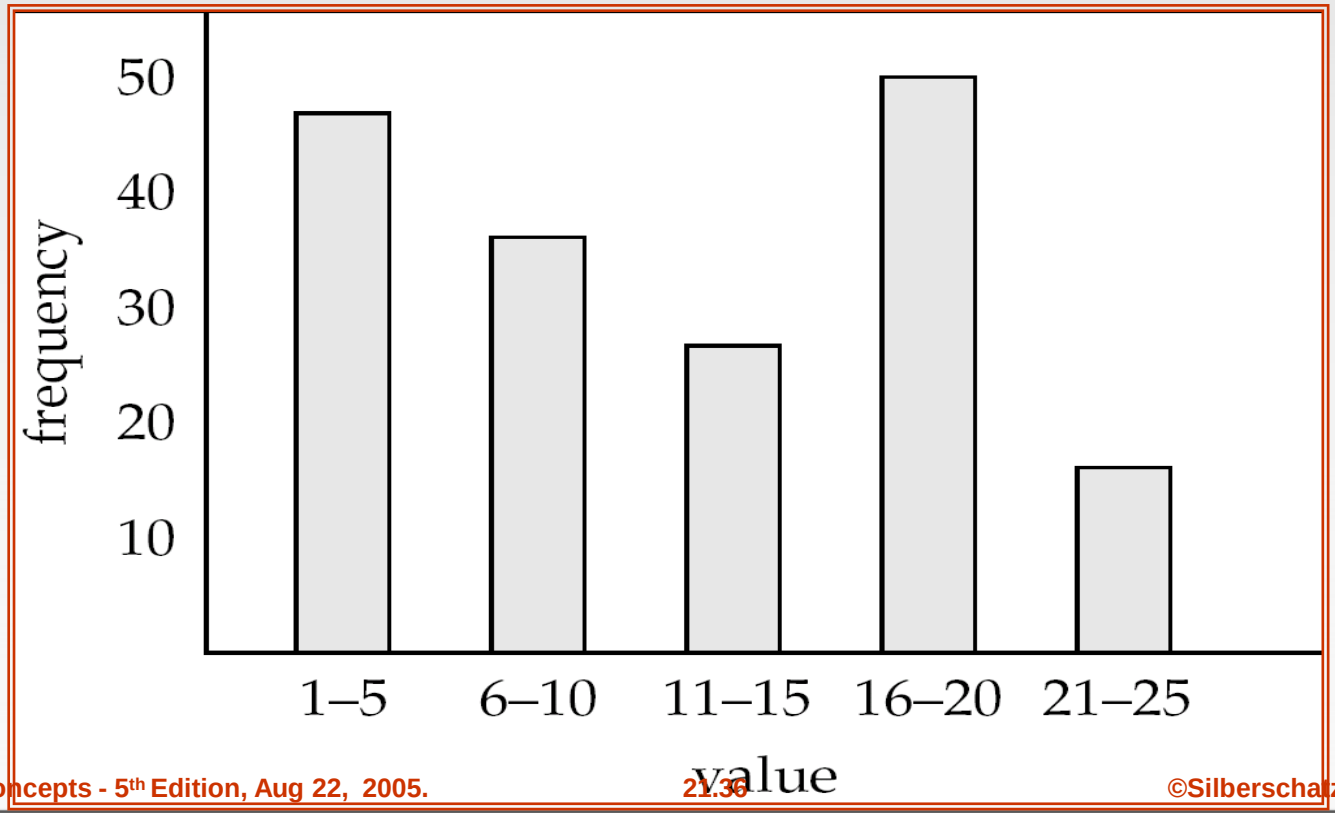
First Technique: **Balanced partitioning vector**

- To create a balanced partitioning vector (assuming partitioning attribute forms a key of the relation):
  - Sort the relation on the partitioning attribute.
  - Construct the partition vector by scanning the relation in sorted order as follows.
    - After every $1/n^{th}$ of the relation has been read, the value of the partitioning attribute of the next tuple is added to the partition vector.
  - *n* denotes the number of partitions to be constructed.
  - Duplicate entries or imbalances can result if duplicates are present in partitioning attributes.
- Alternative technique based on **histograms** used in practice

Handling Skew using Histograms

Second Technique: **Histogram**

- Balanced partitioning vector can be constructed from histogram in a relatively straightforward fashion
  - Assume uniform distribution within each range of the histogram
- Histogram can be constructed by scanning relation, or sampling (blocks containing) tuples of the relation

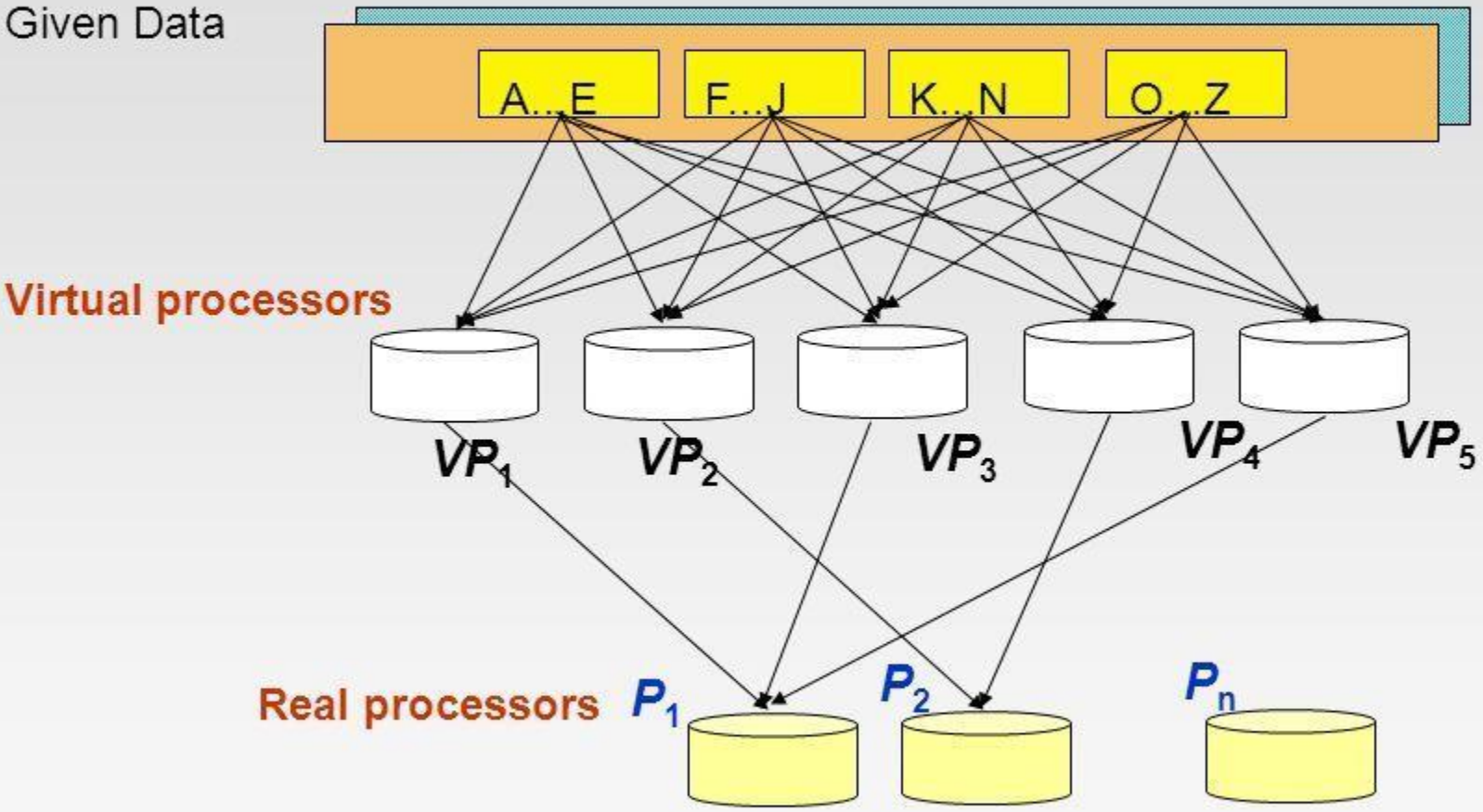


Handling Skew Using Virtual Processor Partitioning

Third Technique: **Virtual processor partitioning**

- Skew in range partitioning can be handled elegantly using **virtual processor partitioning**:
  - create a large number of partitions (say 10 to 20 times the number of processors)
  - Assign virtual processors to partitions either in round-robin fashion or based on estimated cost of processing each virtual partition
- Basic idea:
  - If any normal partition would have been skewed, it is very likely the skew is spread over a number of virtual partitions
  - Skewed virtual partitions get spread across a number of processors, so work gets distributed evenly!

Virtual Processor Partitioning



Query Parallelism

Query Parallelism

Executing database query/queries in parallel.

The concept of parallelism can be exploited in executing mulitple database queries in parallel.

Intra-query parallelism

Execution of a single query in parallel by dividing the workload among various processors

Example:

Consider the following query;

```
SELECT * FROM Emp, Dept WHERE Emp.dno = Dept.dno;
```

Intra-query parallelism is about, "how would we perform the JOIN operation of the given query in parallel using multiple processors"

Inter-query parallelism

Execution of multiple queries in parallel by dividing the workload like each individual query is assigned with individual processors etc.

Example:

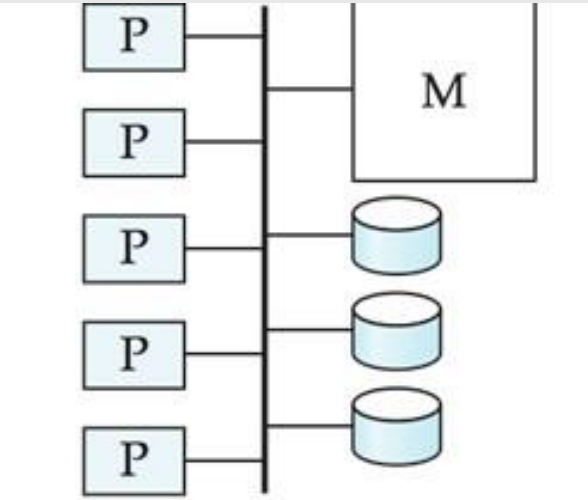
Consider the following queries;

```
SELECT * FROM Emp;

SELECT * FROM Dept WHERE mgrname = 'Steve';

SELECT Furniture_Name, Cost FROM Furniture;
```

Inter-query parallelism is about, "how would we execute all the above queries simultaneously by using parallel servers, so that each transaction need not wait for the other to complete".

Interquery Parallelism

- Queries/transactions execute in parallel with one another.
- Increases transaction throughput; used primarily to scale up a transaction processing 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.



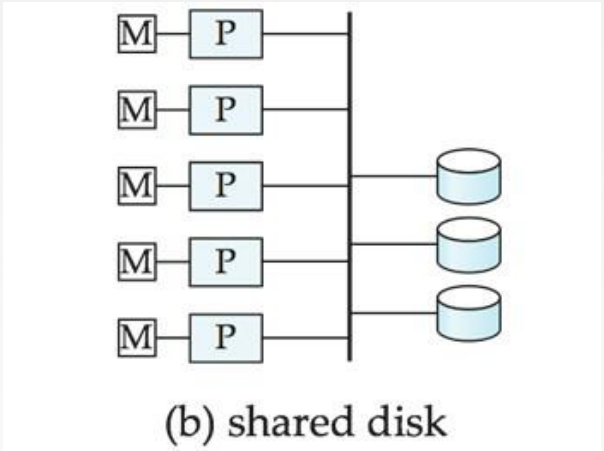
(a) shared memory

Interquery Parallelism

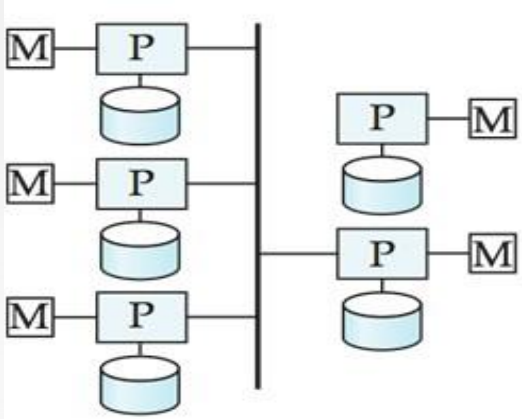
- More complicated to implement on shared-disk or shared-nothing architectures

Issues:

- Locking and logging must be coordinated by passing messages between processors.
- A parallel database system must also ensure that two processors do not update the same data independently at the same time.
- **Cache-coherency** has to be maintained — reads and writes of data in buffer must find latest version of data.
  - The problem of ensuring that the version is the latest is known as the **cache-coherency problem.**



(b) shared disk



(c) shared nothing

Interquery Parallelism

Brief about the next slide:

- Cache coherency protocol for Shared disk:
  - Simple
  - Complex
- Cache coherency protocol for Shared Nothing:

Explained in the next slide……

Cache Coherency Protocol

- Example of a cache coherency protocol for shared disk systems:
  - Before any read or write access to a page, a transaction locks the page in shared or exclusive mode, as appropriate. Immediately after the transaction obtains either a shared or exclusive lock on a page, it also reads the most recent copy of the page from the shared disk.
  - Before a transaction releases an exclusive lock on a page, it flushes the page to the shared disk; then, it releases the lock.
- More complex protocols with fewer disk reads/writes exist. – directly reading a recent version of page from buffer pool of some processor (if it is currently holding and updating)
- Cache coherency protocols for shared-nothing systems are similar. Each database page is assigned a *home* processor. Requests to fetch the page or write it to disk are sent to the home processor.

Query Parallelism – 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.
  - **Interoperation Parallelism** – execute the different operations in a query expression in parallel.

  the first form 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

Query Parallelism – Intra query Parallelism

Intra-query Parallelism

Execution of a single query in parallel by dividing the workload among various processors.

Inter-operation parallelism

Execution of different operations of same query in parallel

Example:

SELECT * FROM Emp WHERE Salary>5000 ORDER BY Name;

In this example, we perform the following two operations;

* SCAN Emp table for Salary > 5000
* Sort all the Emp records on Name attribute

Now, we may use two processors, one to scan the table, and the otehr to perform sorting.

Intra-operation parallelism

Execution of single operation of a query in parallel.

Example:

SELECT * FROM Customer ORDER BY CName;

Let us assume that we have 10 million customers. For such a huge number of records sorting would consume a considerable amount of time. Hence, we would divide all the customer records into n parallel servers and each can perform sorting on the assigned records.

Parallel Processing of Relational Operations- SORT and JOIN

- Our discussion of parallel algorithms assumes:
  - *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.

Intraoperation Parallelism- Parallel Sort

Since relational operations work on relations containing large sets of tuples

- parallelize the operations by executing them in parallel on different subsets

-the number of tuples in a relation can be large, the degree of parallelism is potentially enormous

Range-Partitioning Sort

- Choose processors P_0, ..., P_m, where $m \leq n$ -1 to do sorting.
- Create range-partition vector with m entries, on the sorting attributes
- 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.
- Each processor P_i sorts its partition of the relation locally.
- Each processors executes same operation (sort) in parallel with other processors, without any interaction with the others (**data parallelism**).

Intraoperation Parallelism- Parallel Sort

Range-Partitioning Sort

- **Assumptions:**
  - Assume n processors, P_0, P_1, …, P_{n-1} and n disks D_0, D_1, …, D_{n-1}.
  - Disk D_i is associated with Processor P_i.
  - Relation R is partitioned into R_0, R_1, …, R_{n-1} using Round-robin technique or Hash Partitioning technique or Range Partitioning technique (if range partitioned on some other attribute other than sorting attribute)

- **Objective:**
  - Our objective is to sort a relation (table) R_i that resides on n disks on an attribute A in parallel.

Intraoperation Parallelism- Parallel Sort

Range-Partitioning Sort

- **Steps:**
  - ***Step 1:***
    - Partition the relations R_i on the sorting attribute A at every processor using a range vector **v**.
    - Send the partitioned records which fall in the i^{th} range to Processor P_i where they are temporarily stored in D_i.
  - ***Step 2:***
    - Sort each partition locally at each processor P_i.
    - And, send the sorted results for merging with all the other sorted results which is trivial process.
- **Point to note:**
  - Range partition must be done using a good range-partitioning vector. Otherwise, skew might be the problem.

Intraoperation Parallelism- Parallel Sort

Range-Partitioning Sort

- Consider the following relation schema Employee;
  - **Employee (Emp_ID, EName, Salary)**
- Assume that relation ***Employee is permanently partitioned using Round-robin technique*** into 3 disks D_0, D_1, and D_2 which are associated with processors P_0, P_1, and P_2.
- At processors P_0, P_1, and P_2, the relations are named Employee0, Employee1 and Employee2 respectively. This initial state is given in Figure 1.

Employee0

Emp_ID	EName	Salary
E102	Kumar	10000
E103	Madhan	5000
E101	Jack	6000
E105	Meena	15000

Employee1

Emp_ID	EName	Salary
E112	Kesav	10500
E113	Maddy	15500
E111	Ramya	26000
E115	Megha	18000

Employee2

Emp_ID	EName	Salary
E122	Maya	30000
E123	Ram	5000
E121	Guhan	7500
E125	Steve	25000

Intraoperation Parallelism- Parallel Sort

Range-Partitioning Sort

- Assume that the following sorting query is initiated.
  - **SELECT * FROM Employee ORDER BY Salary;**
- As already said, the table Employee is not partitioned on the sorting attribute Salary. Then, the Range-Partitioning technique works as follows;
- **Step 1:**
  - At first, identify a range vector v on the Salary attribute.
  - The range vector is of the form **v[v0, v1, …, vn-2]**.
  - For given example, let us assume the following range vector;
    - **v[14000, 24000]**
  - This range vector represents 3 ranges, range 0 (14000 and less), range 1 (14001 to 24000) and range 2 (24001 and more).
  - Redistribute the relations Employee0, Employee1 and Employee2 using these range vectors into 3 disks temporarily.

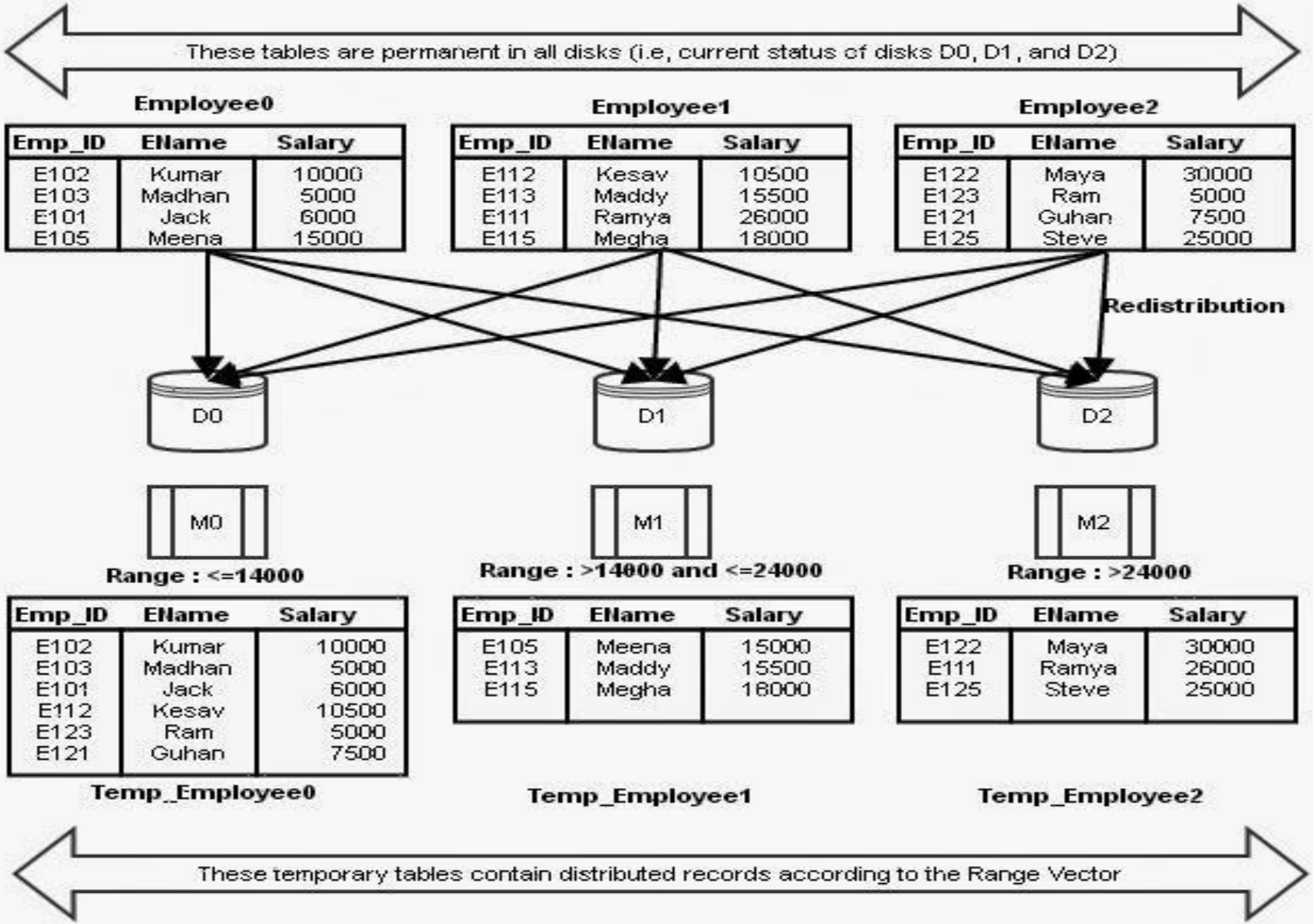
Intraoperation Parallelism- Parallel Sort

Range-Partitioning Sort

- **Step 1 (contd..):**
  - After this distribution, disk 0 holds range 0, disk 1 holds range 1, disk 2 holds range 2
  - Figure 2 links to all the disks from all the relations. Temp_Employee0, Temp_Employee1, and Temp_Employee2, are the relations after successful redistribution, stored temporarily in disks D_0, D_1, and D_2

Figure 2 on next slide…..

Intraoperation Parallelism- Parallel Sort



Intraoperation Parallelism- Parallel Sort

Range-Partitioning Sort

- **Step 2:**
  - All the processors sort the data assigned to them in ascending order of Salary individually.
  - The process of performing the same operation in parallel on different sets of data is called ***Data Parallelism***.
  - Final Result:
    - After the processors completed the sorting, simply collect the data from different processors and merge them.
    - This merge process is straightforward as we have data already sorted for every range.
    - Hence, collecting sorted records from partition 0, partition 1 and partition 2 and merging them will give us final sorted output.

Intraoperation Parallelism- Parallel Sort (Cont.)

Parallel External Sort-Merge

- Assume the relation has already been partitioned among disks D_0, ..., D_{n-1} (in whatever manner).
- Each processor P_i locally sorts the data on disk D_i.
- The sorted runs on each processor are then merged to get the final sorted output.
- Parallelize the merging of sorted runs as follows:
  - The sorted partitions at each processor P_i are range-partitioned across the processors P_0, ..., P_{m-1}.
  - Each processor P_i performs a merge on the streams as they are received, to get a single sorted run.
  - The sorted runs on processors P_0,..., P_{m-1} are concatenated to get the final result.

Intraoperation Parallelism- Parallel Sort (Cont.)

Parallel External Sort-Merge

- **Assumptions:**
  - Assume n processors, P_0, P_1, …, P_{n-1} and n disks D_0, D_1, …, D_{n-1}.
  - Disk D_i is associated with Processor P_i.
  - Relation R is partitioned into R_0, R_1, …, R_{n-1} using Round-robin technique or Hash Partitioning technique or Range Partitioning technique (partitioned on any attribute)

- **Objective:**
  - To sort a relation (table) R_i on an attribute A in parallel where R resides on n disks.

Intraoperation Parallelism- Parallel Sort (Cont.)

Parallel External Sort-Merge

- **Steps:**
  - **Step 1:** Sort the relation partition R_i which is stored on disk D_i on the sorting attribute of the query.
  - **Step 2:** Identify a range partition vector v and range partition every R_i into processors, P_0, P_1, …, P_{n-1} using vector v.
  - **Step 3:** Each processor P_i performs a merge on the incoming range partitioned data from every other processors (The data are actually transferred in order. That is, all processors send first partition into P0, then all processors sends second partition into P1, and so on).
  - **Step 4:** Finally, concatenate all the sorted data from different processors to get the final result.

- **Point to note:**
  - Range partition must be done using a good range-partitioning vector. Otherwise, skew might be the problem.

Intraoperation Parallelism- Parallel Sort (Cont.)

Parallel External Sort-Merge

Example:

- Consider the following relation schema Employee;
  - **Employee (Emp_ID, EName, Salary)**
- Assume that relation ***Employee is permanently partitioned using Round-robin technique*** into 3 disks D_0, D_1, and D_2 which are associated with processors P_0, P_1, and P_2.
- At processors P_0, P_1, and P_2, the relations are named Employee0, Employee1 and Employee2 respectively.
  - This initial state is given in Figure 1.

Employee0

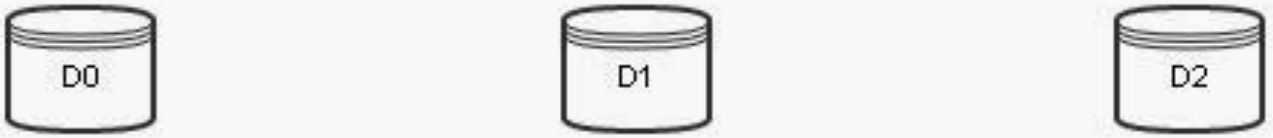
Emp_ID	EName	Salary
E102	Kumar	10000
E103	Madhan	5000
E101	Jack	6000
E105	Meena	15000

Employee1

Emp_ID	EName	Salary
E112	Kesav	10500
E113	Maddy	15500
E111	Ramya	26000
E115	Megha	18000

Employee2

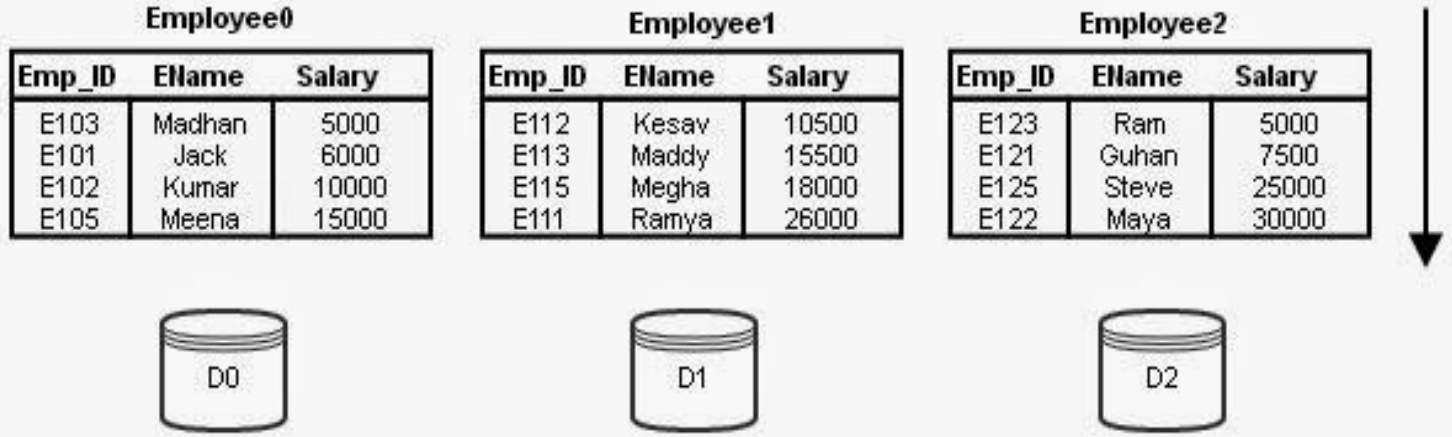
Emp_ID	EName	Salary
E122	Maya	30000
E123	Ram	5000
E121	Guhan	7500
E125	Steve	25000



Intraoperation Parallelism- Parallel Sort (Cont.)

Parallel External Sort-Merge

- Assume that the following sorting query is initiated.
  - SELECT * FROM Employee ORDER BY Salary;
- The table Employee is not partitioned on the sorting attribute Salary. Then, the Parallel External Sort-Merge technique works as follows;
- **Step 1:**
  - Sort the data stored in every partition (every disk) using the ordering attribute Salary. (Sorting of data in every partition is done temporarily).
  - At this stage every $Employee_i$ contains salary values of range minimum to maximum. The partitions sorted in ascending order is shown below, in Figure 2

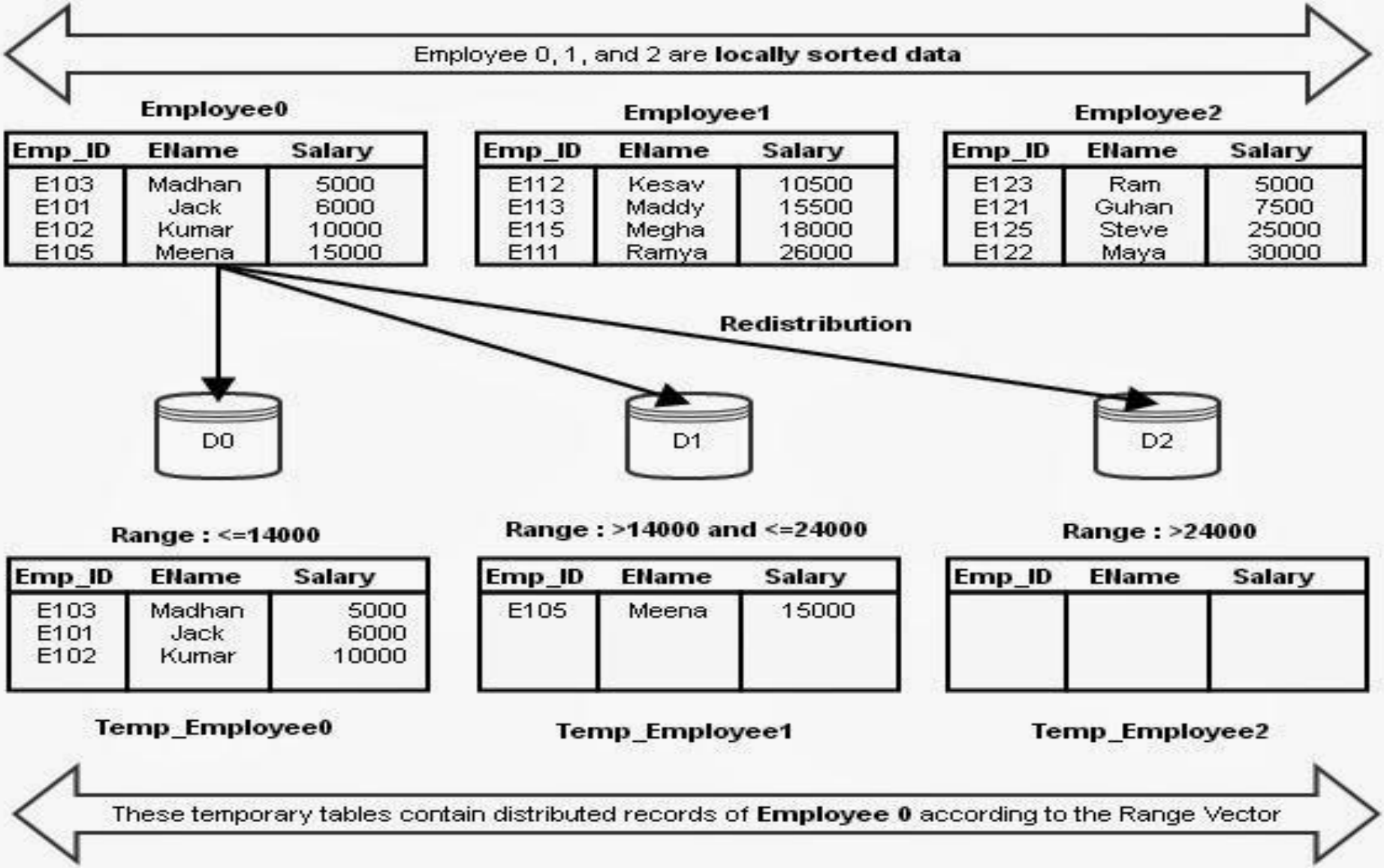


Intraoperation Parallelism- Parallel Sort (Cont.)

Parallel External Sort-Merge

- **Step 2:**
  - Identify a range vector v on the Salary attribute. The range vector is of the form **v[v0, v1, …, vn-2]**. For our example, let us assume the following range vector;
    - v[14000, 24000]
  - This range vector represents 3 ranges, range 0 (14000 and less), range 1 (14001 to 24000) and range 2 (24001 and more).
  - Redistribute every partition (Employee0, Employee1 and Employee2) using these range vectors into 3 disks temporarily.
  - Status of Temp_Employee 0, 1, and 2 after distributing Employee 0 is given in Figure 3. (next slide…)

Intraoperation Parallelism- Parallel Sort (Cont.)

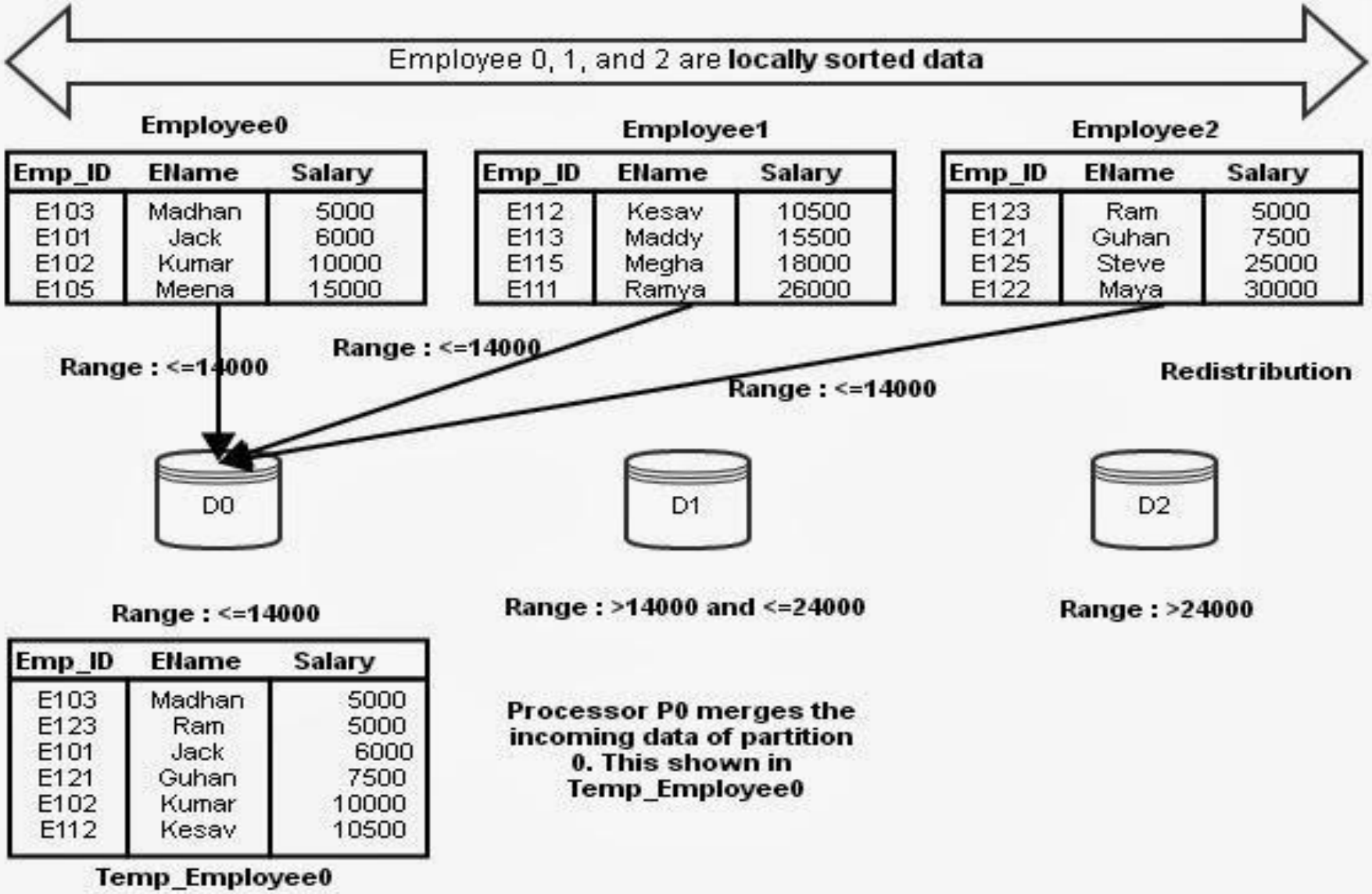


Intraoperation Parallelism- Parallel Sort (Cont.)

Parallel External Sort-Merge

- **Step 3:**
  - Actually, the above said distribution is executed at all processors in parallel such that processors P0, P1, and P2 are sending the first partition of Employee 0, 1, and 2 to disk 0.
  - Upon receiving the records from various partitions, the receiving processor P0 merges the sorted data. This is shown in Figure 4. (next slide..)

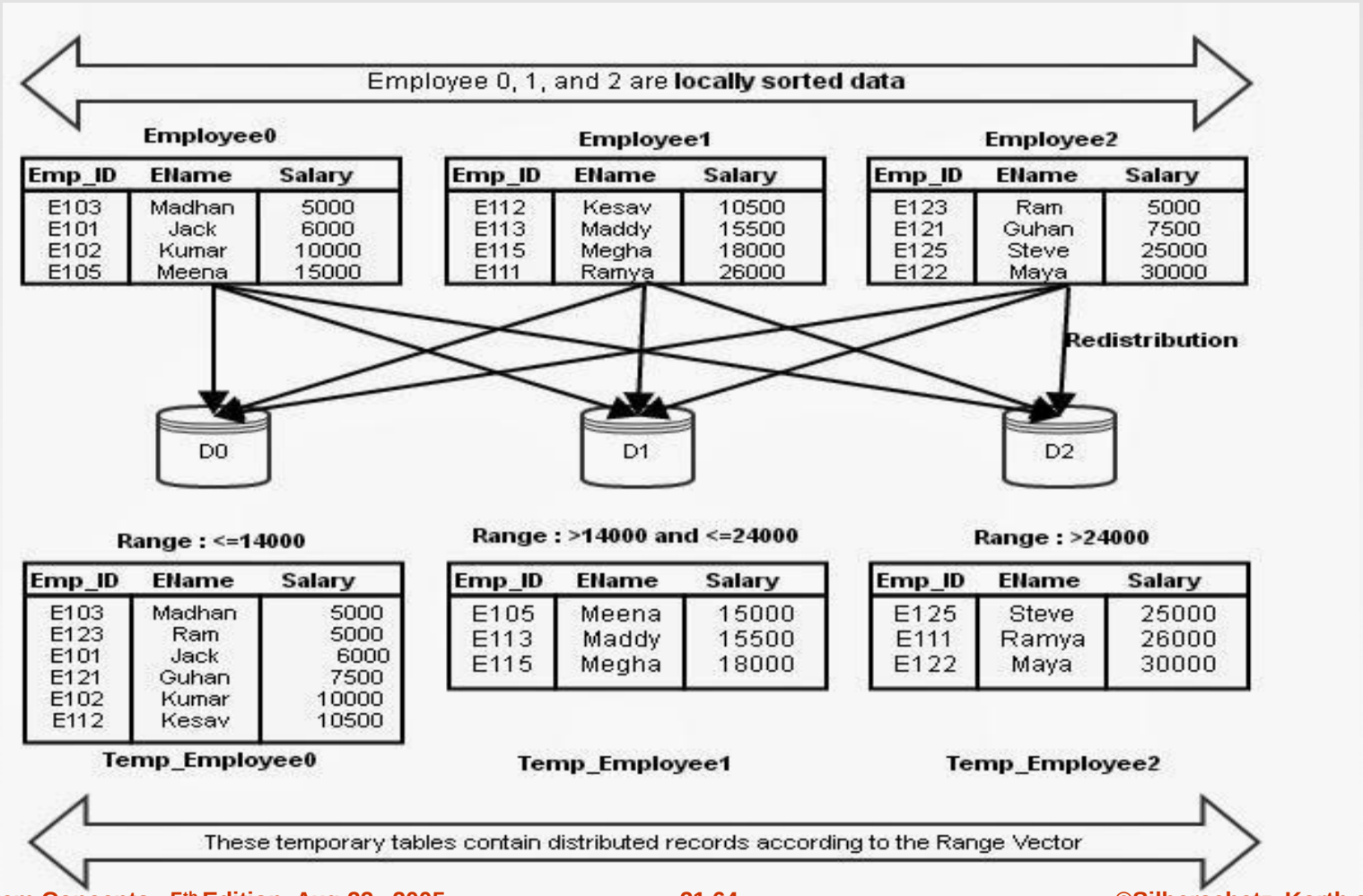
Intraoperation Parallelism- Parallel Sort (Cont.)



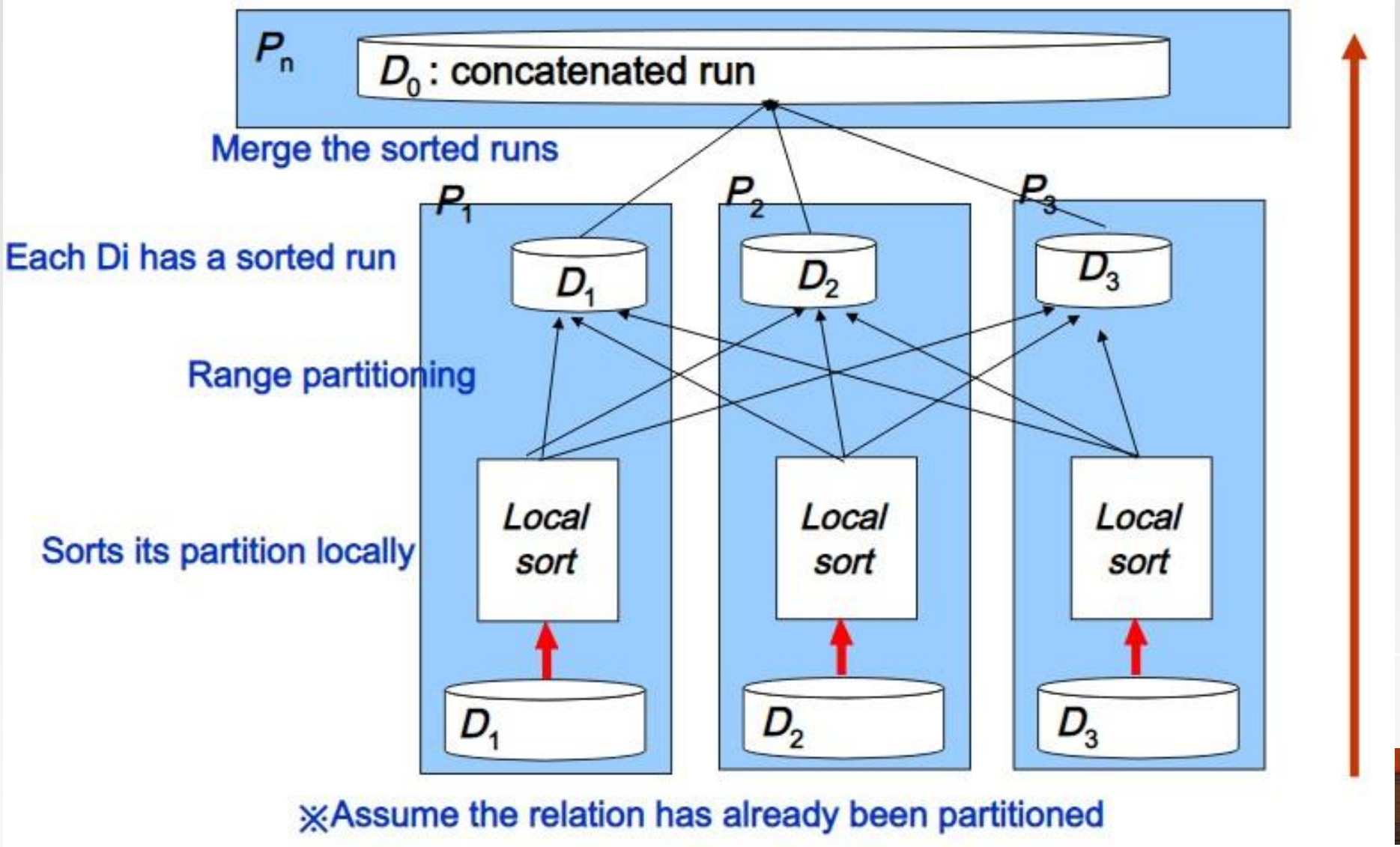
Intraoperation Parallelism- Parallel Sort (Cont.)

Parallel External Sort-Merge

- The above said process is done at all processors for different partitions. The final version of Temp_Employee 0, 1, and 2 are shown in Figure 5.



Parallel External Sort-Merge - Figure



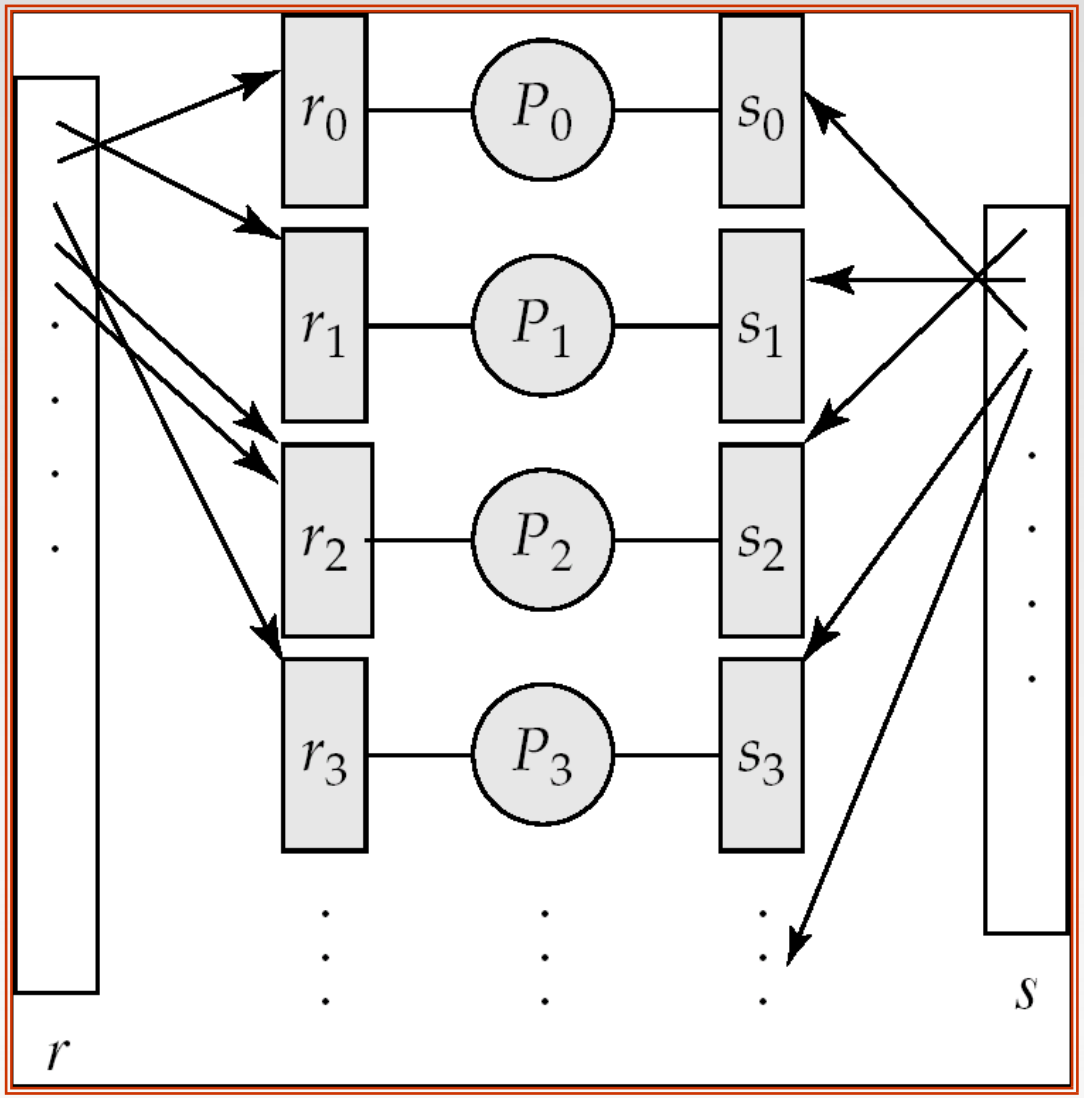
Intraoperation Parallelism- Parallel Join

- 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.

Intraoperation Parallelism- Parallel Join

- Works for equi-joins and natural joins
- Let *r* and *s* be the input relations,
  - compute $r \bowtie_{r.A=s.B} s$.
- *r* and *s* each are partitioned into *n* partitions, denoted $r_0, r_1, \ldots, r_{n-1}$ and $s_0, s_1, \ldots, s_{n-1}$.
  - 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.
- Partitions r_i and s_i are sent to processor P_i,
- Each processor P_i locally computes $r_i \bowtie_{ri.A=si.B} s_i$. Any of the standard join methods can be used.

Intraoperation Parallelism- Parallel Join



Intraoperation Parallelism- Parallel Join

Partitioned Join

With Example….

- The relational tables that are to be joined gets partitioned on the joining attributes of both tables using same partitioning function to perform Join operation in parallel.
- Assume that relational tables r and s need to be joined using attributes r.A and s.B.
- The system partitions both r and s using same partitioning technique into n partitions
- r is partitioned into $\mathbf{r_0, r_1, r_2, \ldots, r_{n-1}}$ and s is partitioned into $\mathbf{s_0, s_1, s_2, \ldots, s_{n-1}}$. Then, the system sends partitions $\mathbf{r_i}$ and $\mathbf{s_i}$ into processor $\mathbf{P_i}$, where the join is performed locally.
- Only joins such as Equi-Joins and Natural Joins can be performed using Partitioned join technique.
  - Equi-Join or Natural Join is done between two tables using an equality condition such as r.A = s.B.
  - The tuples which are satisfying this condition, i.e, same value for both A and B, are joined together and will end up in the same partition

Intraoperation Parallelism- Parallel Join

With Example….

RegNo	SName	Gen	Phone
1	Sundar	M	9898786756
3	Karthik	M	8798987867
4	John	M	7898886756
2	Ram	M	9897786776

Table 1 - STUDENT

RegNo	Courses
4	Database
2	Database
3	Data Structures
1	Multimedia

Table 2 – COURSES_REGD

Let us assume the following;

- The RegNo attributes of tables STUDENT and COURSES_REGD are used for joining.
- Tuples are not ordered. They are stored in random order on RegNo.
- Partition the tables on RegNo attribute using Hash Partition.
  - We have 2 disks and we need to partition the relational tables into two partitions (possibly equal). Hence, n is 2.

Intraoperation Parallelism- Parallel Join

Partitioned Join With Example….

- The hash function is, h(RegNo) = (RegNo mod n) = (RegNo mod 2).
- The tables STUDENT and COURSES_REGD partitioned into $Disk_0$ and $Disk_1$ as stated below.
- RegNo values of both tables STUDENT and COURSES_REGD are sent to same partitions.
  - join can be performed locally at every processor in parallel.

Partition 0

RegNo	SName	Gen	Phone
4	John	M	7898886756
2	Ram	M	9897786776

STUDENT_0

RegNo	Courses
4	Database
2	Database

COURSES_REGD_0

Partition 1

RegNo	SName	Gen	Phone
1	Sundar	M	9898786756
3	Karthik	M	8798987867

STUDENT_1

RegNo	Courses
3	Data Structures
1	Multimedia

COURSES_REGD_1

Intraoperation Parallelism- Parallel Join

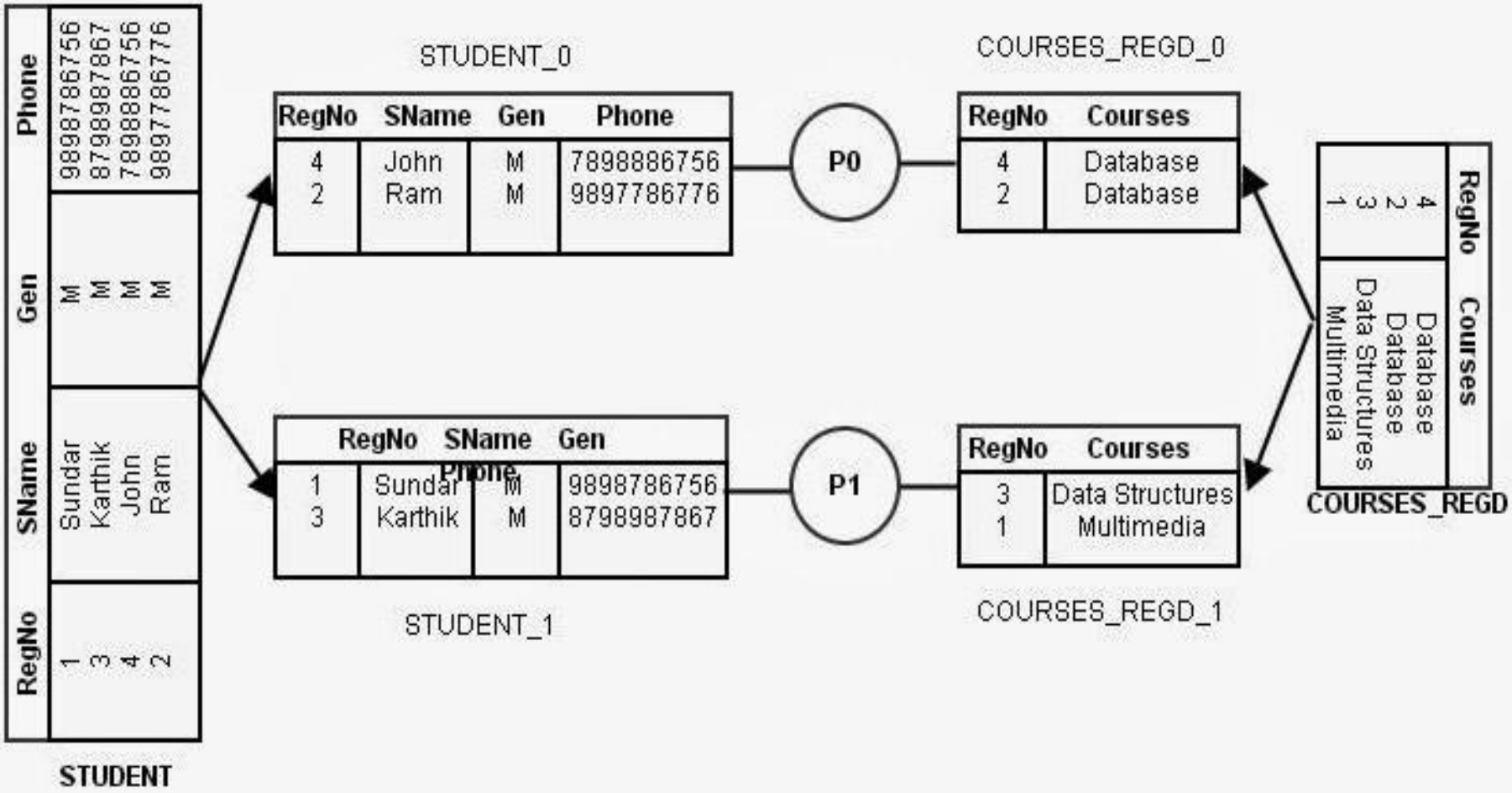
Partitioned Join

With Example….

- No. of Comparisons:
  - only 4 (2 Student records X 2 Courses_regd records) comparisons need to be done in every partition
  - Hence, we need total of 8 comparisons in partitioned join against 16 (4 X 4) in conventional join.

The above discussed process is shown in Figure 1. (next slide..)

Intraoperation Parallelism- Parallel Join



Intraoperation Parallelism- Parallel Join

Partitioned Join

Points to note:

- There are only two ways of partitioning the relations,
  - Range partitioning on the join attributes or
  - Hash partitioning on the join attributes.
- Only equi-joins and natural joins can be performed in parallel using Partitioned Join technique. Non-equi-joins cannot be performed with this method.
- After successful partitioning, the records at every processor can be joined locally using any of the joining techniques hash join, merge join, or nested loop join.
- If Range partitioning technique is used to partition the relations into n processors, **Skew** may present a special problem.
- The number of comparisons between relations are well reduced in partitioned join parallel technique.

Parallel Join

Which Join methods should be used for following Join Operation?

- Equi-join is of the form, - Partitioned Join
  - **SELECT columns FROM list_of_tables WHERE table1.column = table2.column;**
- whereas, Non-equi join is of the form, - Fragment and Replicate Join
  - **SELECT columns FROM list_of_tables WHERE table1.column < table2.column;**
    - (Or, any other operators >, <>, <=, >= etc. in the place of < in the above example)

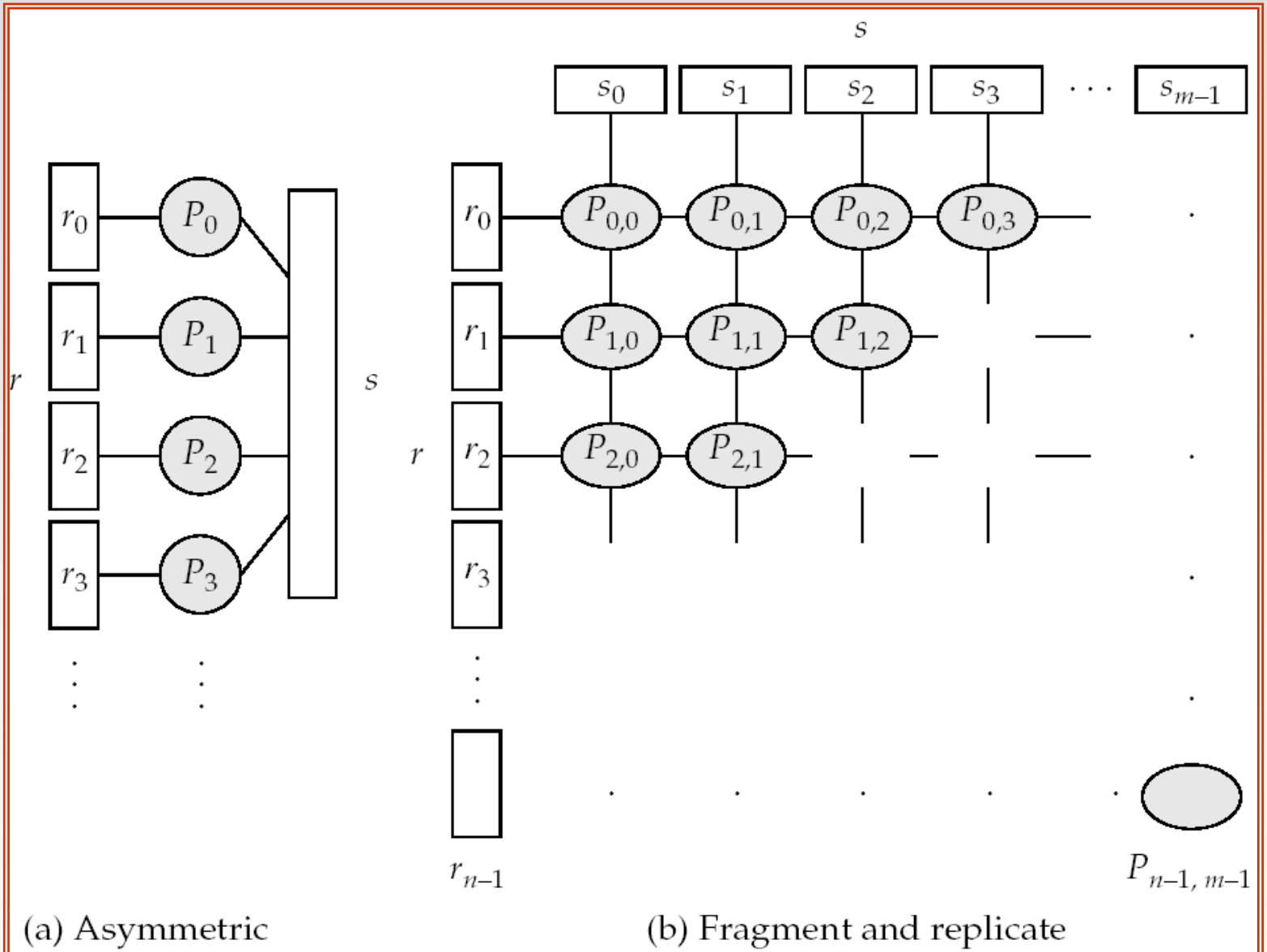
Fragment-and-Replicate Join

- **What does fragment and replicate mean?**
  - Fragment means partitioning a table either horizontally or vertically (Horizontal and Vertical fragmentation).
  - Replicate means duplicating the relation, i.e, generating similar copies of a table.
    - This join is performed by fragmenting and replicating the tables to be joined.
- Consider a join condition as given below;
  - $r \bowtie_{r.a>s.b} s$
    - all records of r join with some records of s and vice versa.

Fragment-and-Replicate Join

- Special case – **asymmetric fragment-and-replicate** (when one of the relation is smaller)
  - 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.

Depiction of Fragment-and-Replicate Joins



(a) Asymmetric fragment and replicate

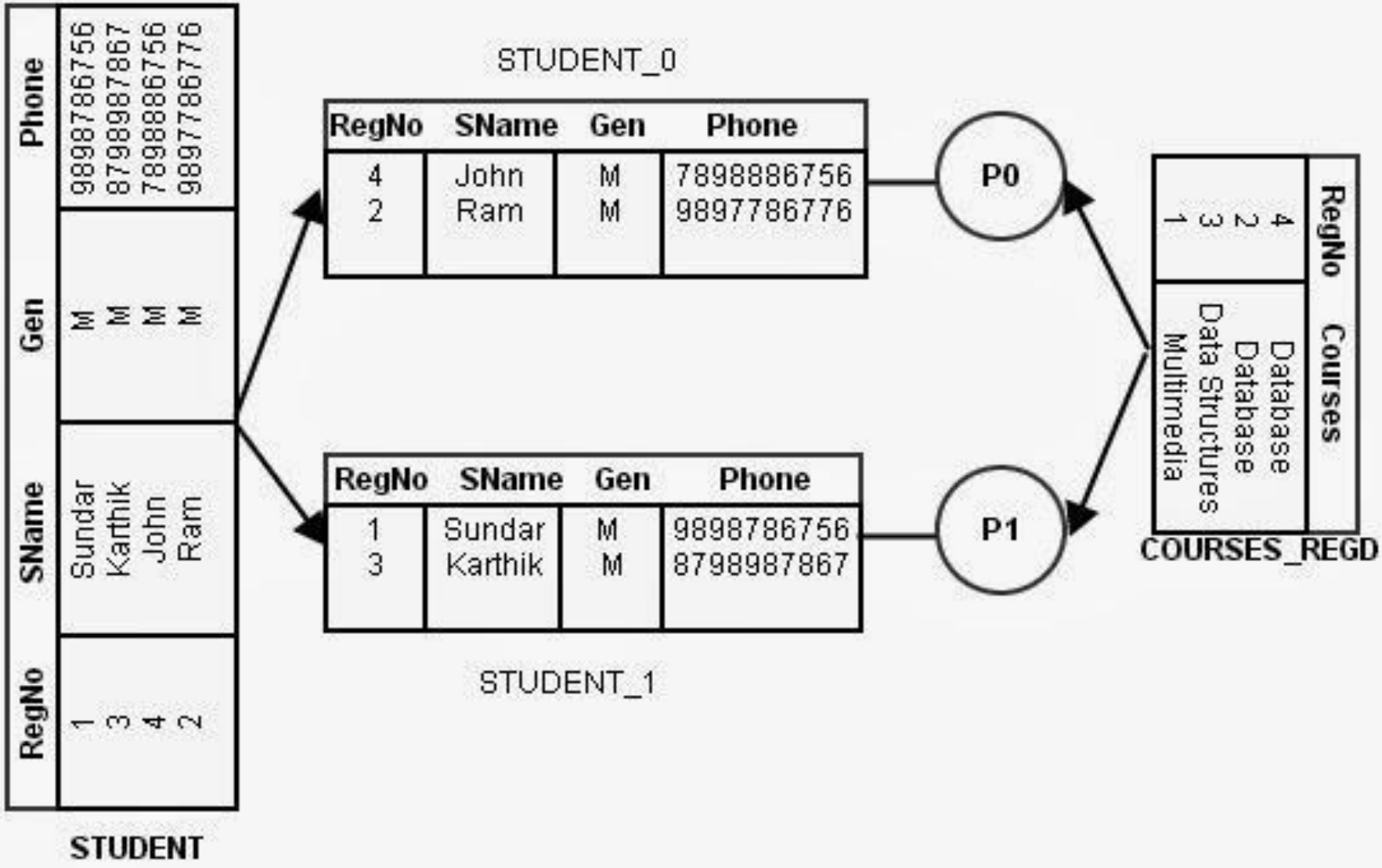
(b) Fragment and replicate

Fragment-and-Replicate Join (Cont.)

with example:

- **Asymmetric Fragment and Replicate Join (Special Case)**
  - The system fragments table r into n fragments such that r_0, r_1, r_2, .., r_{n-1},
    - where, r is one of the tables that is to be joined
    - n represents the number of processors.
    - Any partitioning technique, round-robin, hash or range partitioning could be used to partition the relation.
  - The system replicates the other table, say s into n processors.
  - Thus,
    - r_0 and s in processor P_0,
    - r_1 and s in processor P_1,
    - r_2 and s in processor P_2, ..,
    - r_{n-1} and s in processor P_{n-1}.
    - The processor Pi is performing the join locally on r_i and s.

Asymmetric Fragment and Replicate Join



Fragment-and-Replicate Join (Cont.)

- **Asymmetric Fragment and Replicate Join**

  Points to note:

  - Non-equal join can be performed in parallel.
  - If one of the relations to be joined is already partitioned into n processors, this technique is best suited, because we need to replicate the other relation.
  - Unlike in Partitioned Join, any partitioning techniques can be used.
  - If one of the relations to be joined is very small, and if it can fit into memory. The technique performs better.

Fragment-and-Replicate Join (Cont.)

Fragment and Replicate Join (General Case) with example:

- If the relations that are to be joined are large, and the joins is non-equal then we need to use Fragment-and-Replicate Join. It works as follows;
  - The system fragments table r into m fragments such that r_0, r_1, r_2, .., r_{m-1},
  - and s into n fragments such that s_0, s_1, s_2, .., s_{n-1} .
  - Any partitioning technique, round-robin, hash or range partitioning could be used to partition the relations.
  - We need at least m*n processors to perform join.
  - ***compare every tuple of one relation with every tuple of other relation. That is the records of r_0 partition should be compared with all partitions of s, and the records of partition s_0 should be compared with all partitions of r.***

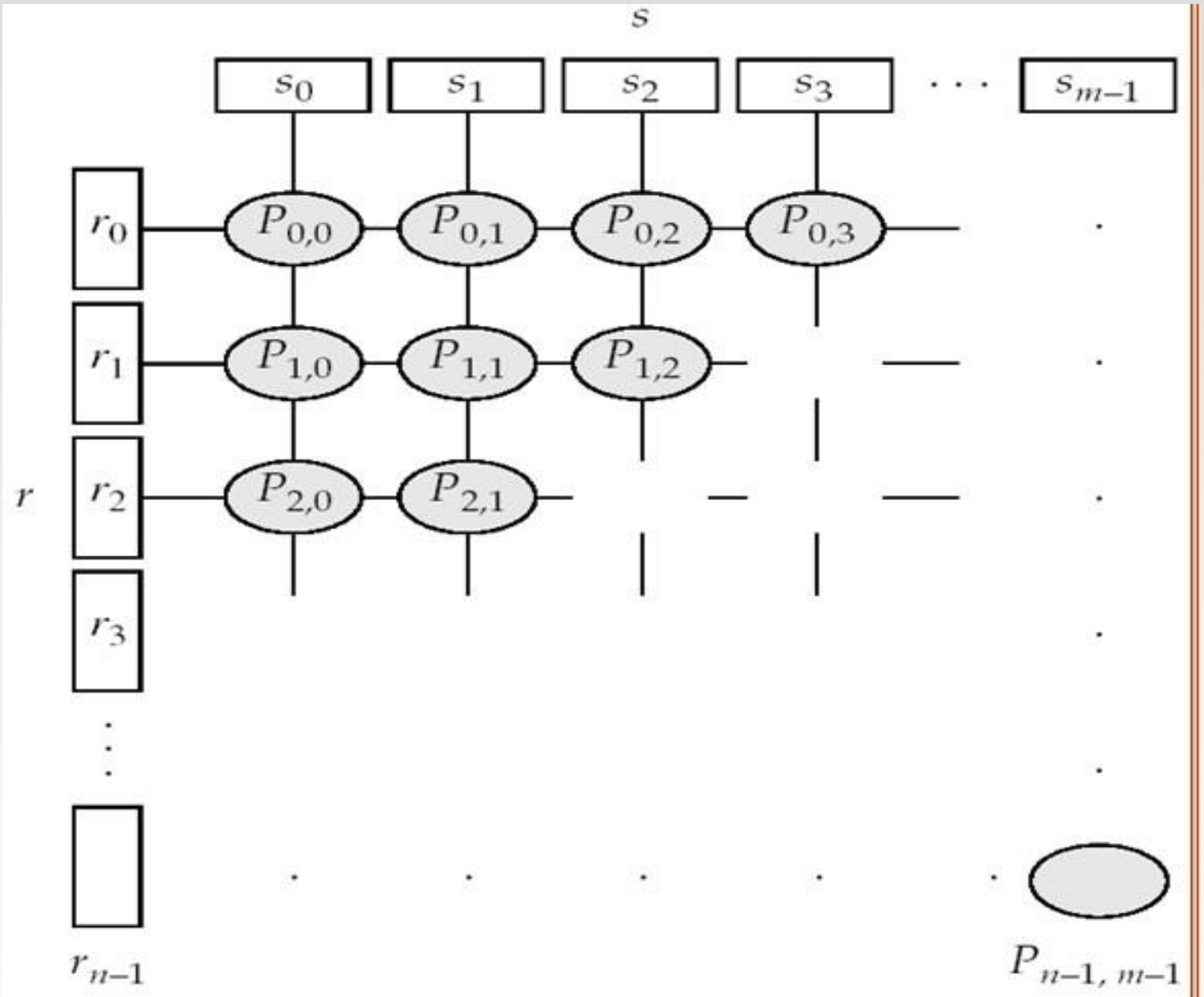
Fragment-and-Replicate Join (Cont.)

Fragment and Replicate Join with example:

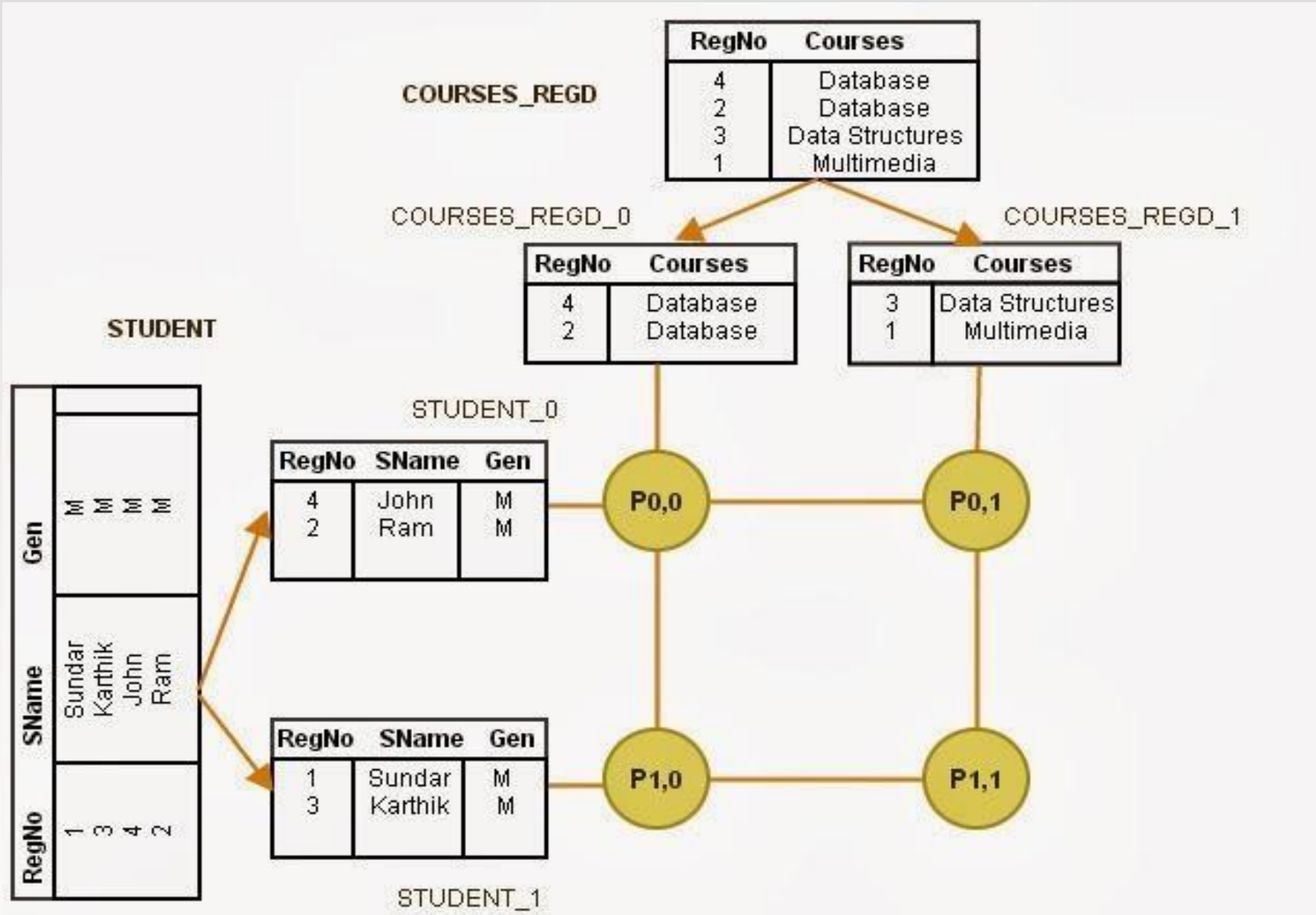
Hence, the data distribution is done as follows;

- assume that we have processors $P_{0,0}$, $P_{0,1}$, …, $P_{0,n-1}$, $P_{1,0}$, $P_{1,1}$, …, $P_{m-1,n-1}$. Thus, processor $P_{i,j}$ performs the join of r_i with s_j.
- To ensure the comparison of every partition of r with every other partition of s,
  - replicate r_i with the processors, $P_{i,0}$, $P_{i,1}$, $P_{i,2}$, …, $P_{i,n-1}$, where 0, 1, 2, …, n-1 are partitions of s.
  - This replication ensures the comparison of every r_i with complete s.
- To ensure the comparison of every partition of s with every other partition of r,
  - replicate s_i with the processors, $P_{0,i}$, $P_{1,i}$, $P_{2,i}$, …, $P_{m-1,i}$, where 0, 1, 2, …, m-1 are partitions of r.
  - This replication ensures the comparison of every s_i with complete r.
- $P_{i,j}$ computes the join locally to produce the join result.

Depiction of Fragment-and-Replicate Joins



Fragment-and-Replicate Joins Example



Fragment-and-Replicate Join (Cont.)

Points to note:

- 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 has a higher cost than partitioning, since one of the relations (for asymmetric fragment-and-replicate) or both relations (for general fragment-and-replicate) have to 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

First understand Sequential Hash Join

-- depicted in the next slide…

Sequential Hash-Join

- Hash join is proposed for performing joins that are Natural joins or Equi-joins.
- **Simple Hash Join:**
  - **Input:** The list of tables to be joined, say r and s and the joining attributes r.A and s.B of tables r and s.
  - **Output:** Joined relation.
- **Steps:**
  - Identify a hash function
  - Identify the smaller relation among the relations that are to be joined, say r and s.
  - Partition the smaller relation, say r, into the identified buckets using the join attribute of r.
    - r is partitioned into available buckets as r_0, r_1, r_2, …, r_9.
    - This step is called ***Building Phase*** (sometimes ***Partitioning Phase***).

Sequential Hash-Join

- **Steps (contd..):**
  - Partition the relation s into the identified buckets using the join attribute of s.
  - s is partitioned into available buckets as s_0, s_1, s_2, …, s_9.
    - This step is called ***Probing Phase*** as the records of s probe for the matching records in the appropriate buckets.
  - Finally the result can be collected from different buckets and submitted as result.
- **Example:** Consider the following two tables;

EmpID	EName	DeptNo
E101	Kumar	2
E103	Wess	1
E107	Terry	1
E102	Gowri	2
E110	Morgan	3
E111	Sachin	3

Table 1 - Employee

DeptNo	DName	DLocation
1	Design	Chennai
2	Production	Chennai
3	Administration	Bangalore

Table 2 – Department

Sequential Hash-Join

- **Steps (contd..):**
  - Step 1: Hash function looks like the one given below;
    - ***h(Hash_key) = Hash_key_value mod n***
  - where n is the number of partitions (or buckets).
  - Let choose the DeptNo attribute as hash key and partition the relation into 3 partitions such as 0, 1, and 2.
  - hash function will look like as follows;
    - ***h(DeptNo) = DeptNo mod 3***
  - Step 2: The Department table is the smaller table.
  - Step 3: Partition Department using the hash function.
    - The first record of Department will go into Bucket 1, second record into Bucket 2, and third record into Bucket 0.
    - Figure on next slide…

Sequential Hash-Join

DeptNo	DName	DLocation
1	Design	Chennai
2	Production	Chennai
3	Administration	Bangalore

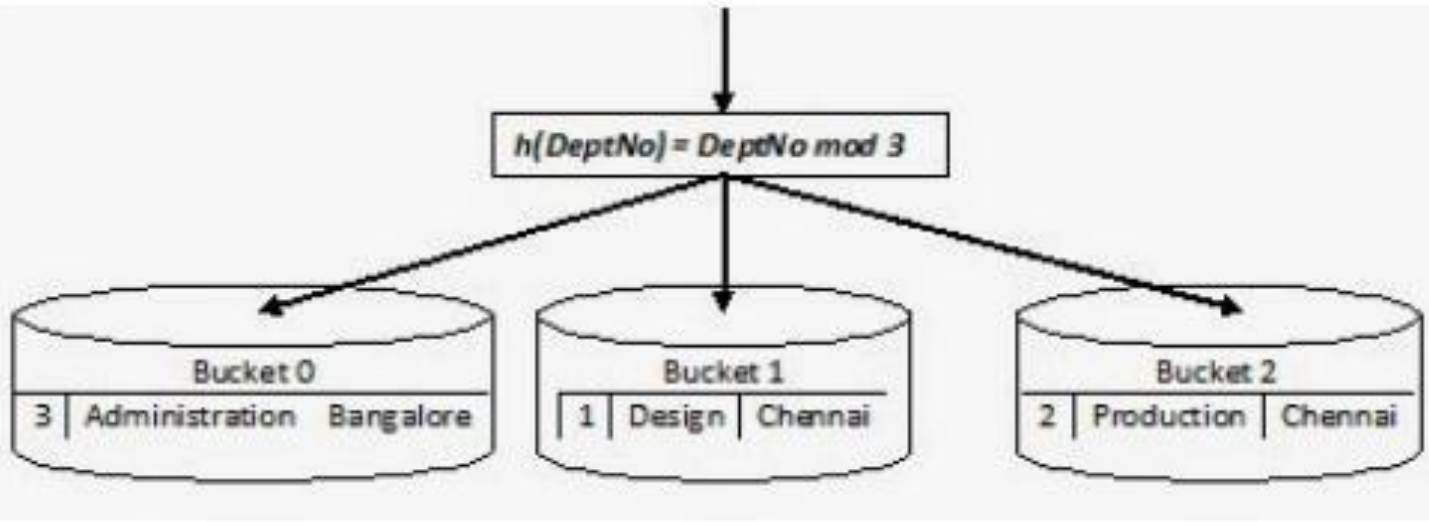


Figure 1 - Hashing of Department table

Sequential Hash-Join

- **Steps (contd..):**
  - Step 4: Now partition the larger relation Employee using the same hash function. That is use the following hash function;
    - ***h(DeptNo) = DeptNo mod 3***
  - The figure given below shows the partitioning of Employee table into 3 buckets.

EmpID	EName	DeptNo
E101	Kumar	2
E103	Wess	1
E107	Terry	1
E102	Gowri	2
E110	Morgan	3
E111	Sachin	3

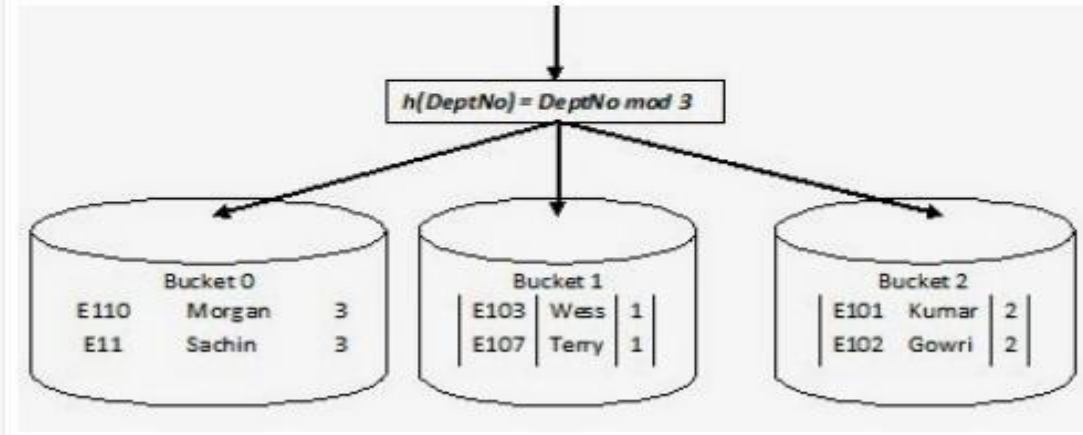
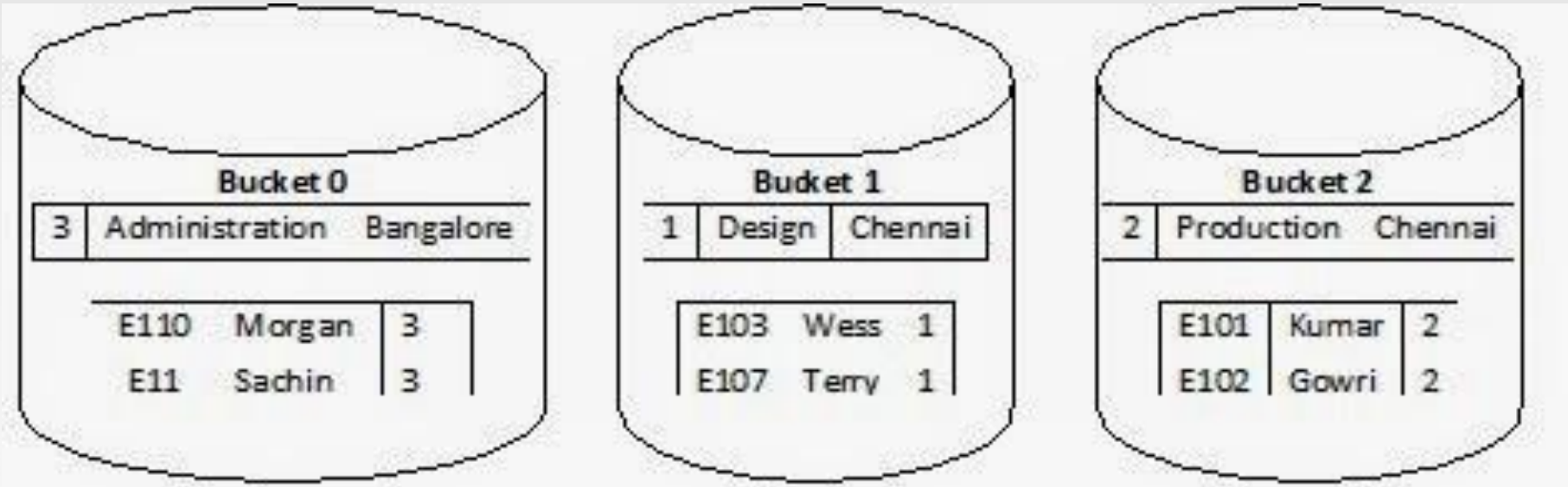


Figure 2 - Hashing of Employee table

Sequential Hash-Join

- **Steps (contd..):**
  - After successful partitioning, hash buckets will look the figure given below;
    - in this figure, the first table shows 0th partition of Department table, and the second one shows the 0th partition of Employee table.



Sequential Hash-Join

- **Steps (contd..):**
  - The major advantage is that, with conventional join technique, the comparison requires,
    - 6 records X 3 records = 18 comparisons
  - With hash join, we need
    - (1 record X 2 records in Bucket 0)+ (1 X 2 in Bucket 1)+ (1 X 2 in Bucket 2) = 6 comparisons
- **Points to note:**
  - Only Equi-join or Natural Join can be performed using Hash join
  - Simple hash join assumed one of the relations as small relation, i.e, the whole relation can fit into memory.
  - Smaller relation is chosen in the building phase.
  - Only a single pass through the tables is required. That is one time scanning of relation is required.
  - Hash function should be chosen such that it should not cause skewness.

Partitioned Parallel Hash-Join

Parallelizing partitioned hash join:

- Assume s is smaller than r and therefore s is chosen as the build 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.
- 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.
- 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. *(Cont.)*

Partitioned Parallel Hash-Join (Cont.)

- Once the tuples of *s* have been distributed, the larger relation *r* is redistributed across the *m* processors using the hash function h_1
  - Let r_i denote the tuples of relation *r* that are sent to processor P_i.
- 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).
- Each processor P_i executes the build and probe phases of the hash-join algorithm on the local partitions r_i and *s* of *r* and *s* to produce a partition of the final result of the hash-join.
- Note: Hash-join optimizations can be applied to the parallel case
  - e.g., the hybrid hash-join algorithm can be used to cache some of the incoming tuples in memory and avoid the cost of writing them and reading them back in.

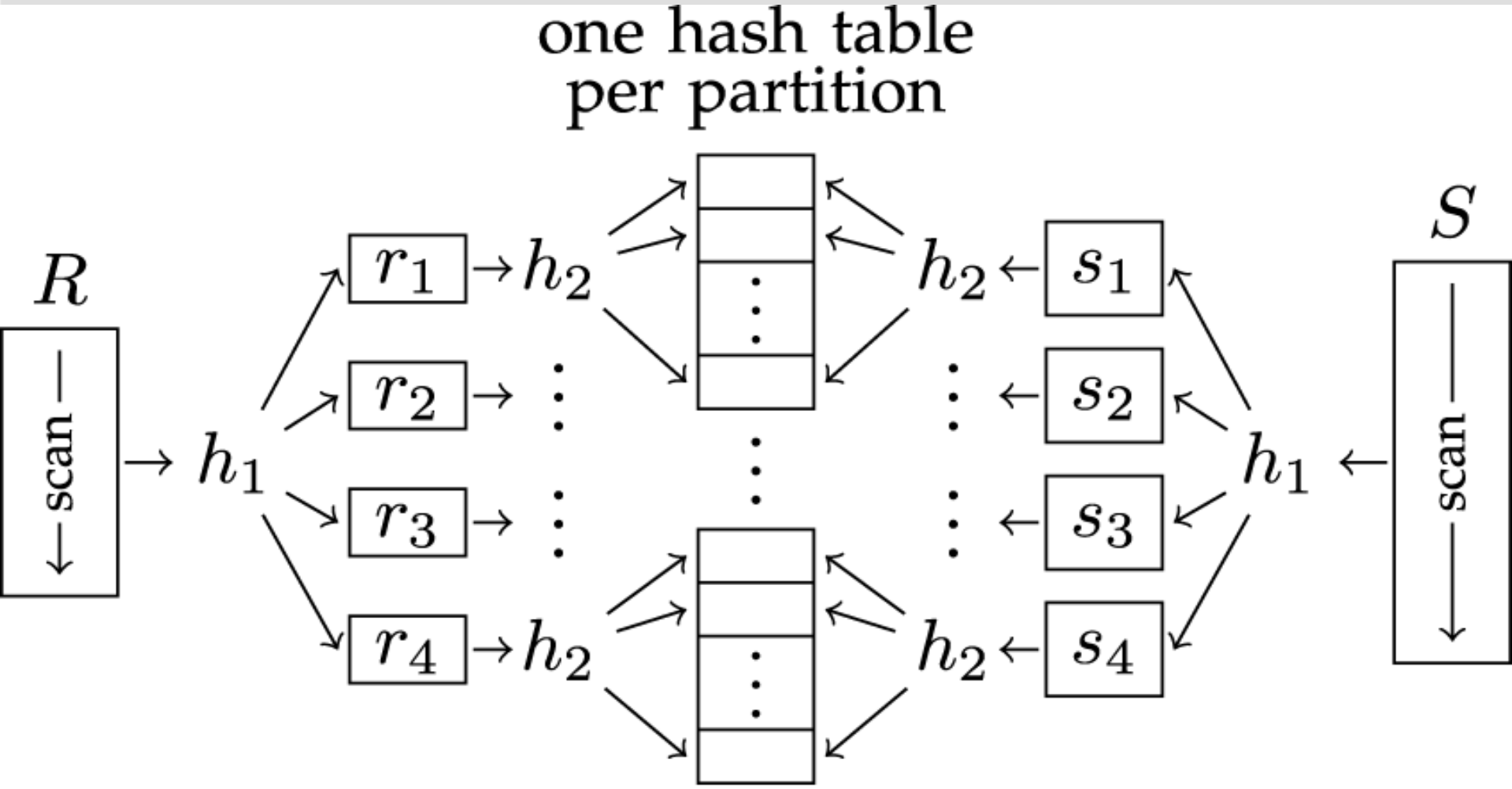
Partitioned Parallel Hash-Join (Cont.)

In brief:

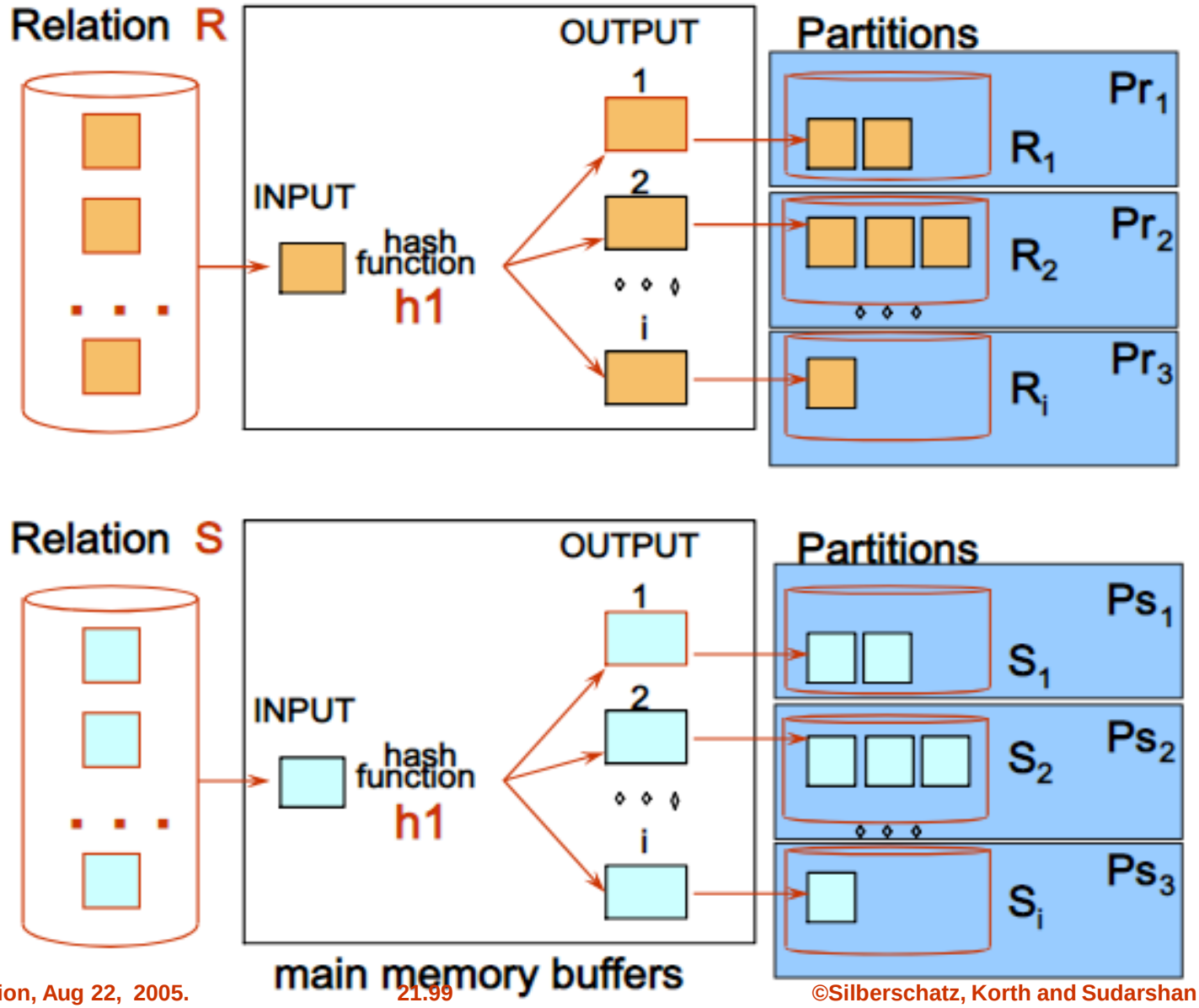
- Relation r and s: assume s is smaller,
  - Apply hash h1 to s, and partition across processors
  - Each partition is re-partitioned using hash h2

- Take relation r,
  - Apply hash h1 to r, and partition across processors
  - Each partition of r is re-partitioned using hash h2 and probe for appropriate bucket in s

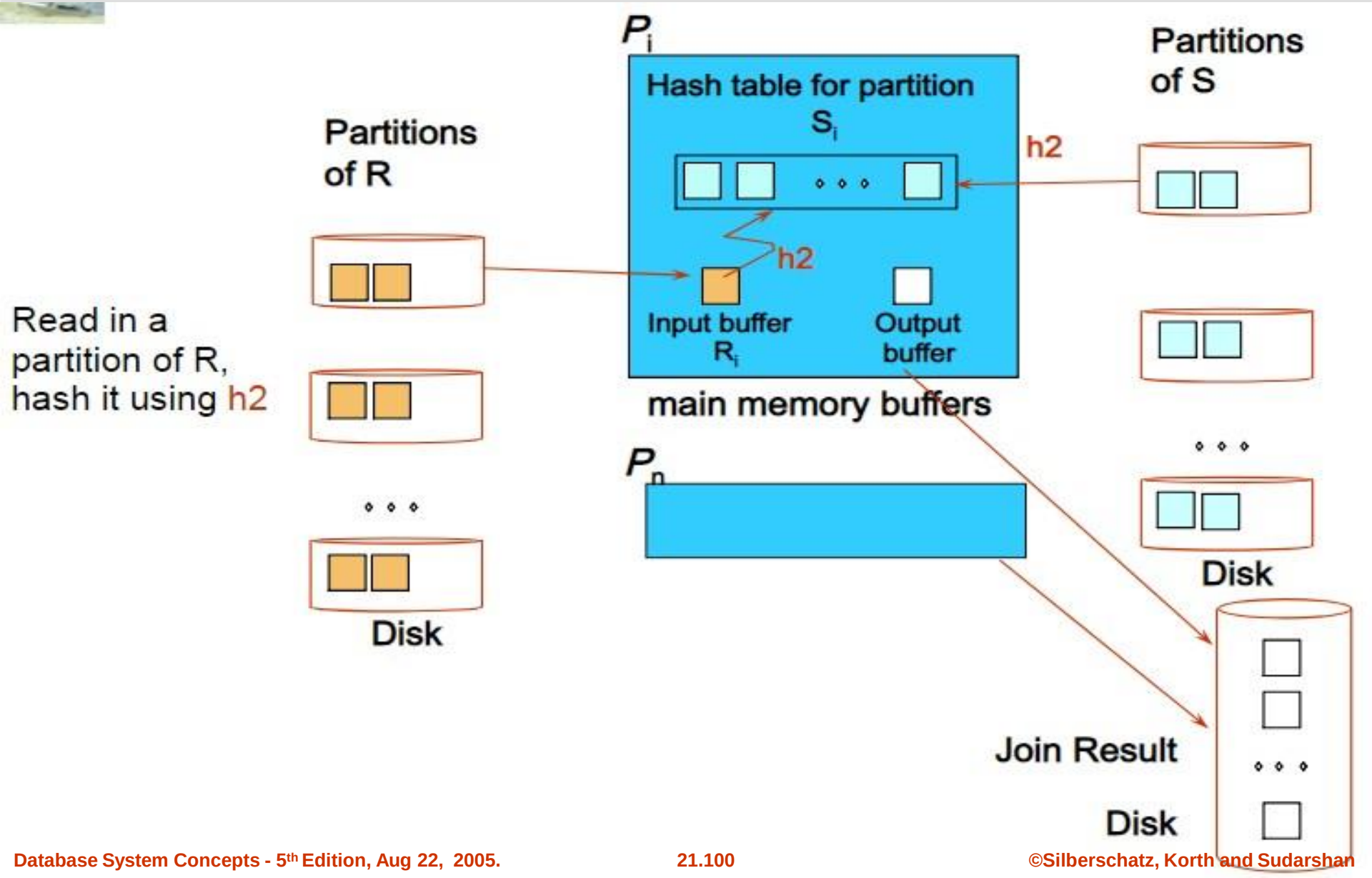
Partitioned Parallel Hash-Join (Cont.) – Figure 1



Partitioned Parallel Hash-Join (Cont.) – Figure 2 (a)



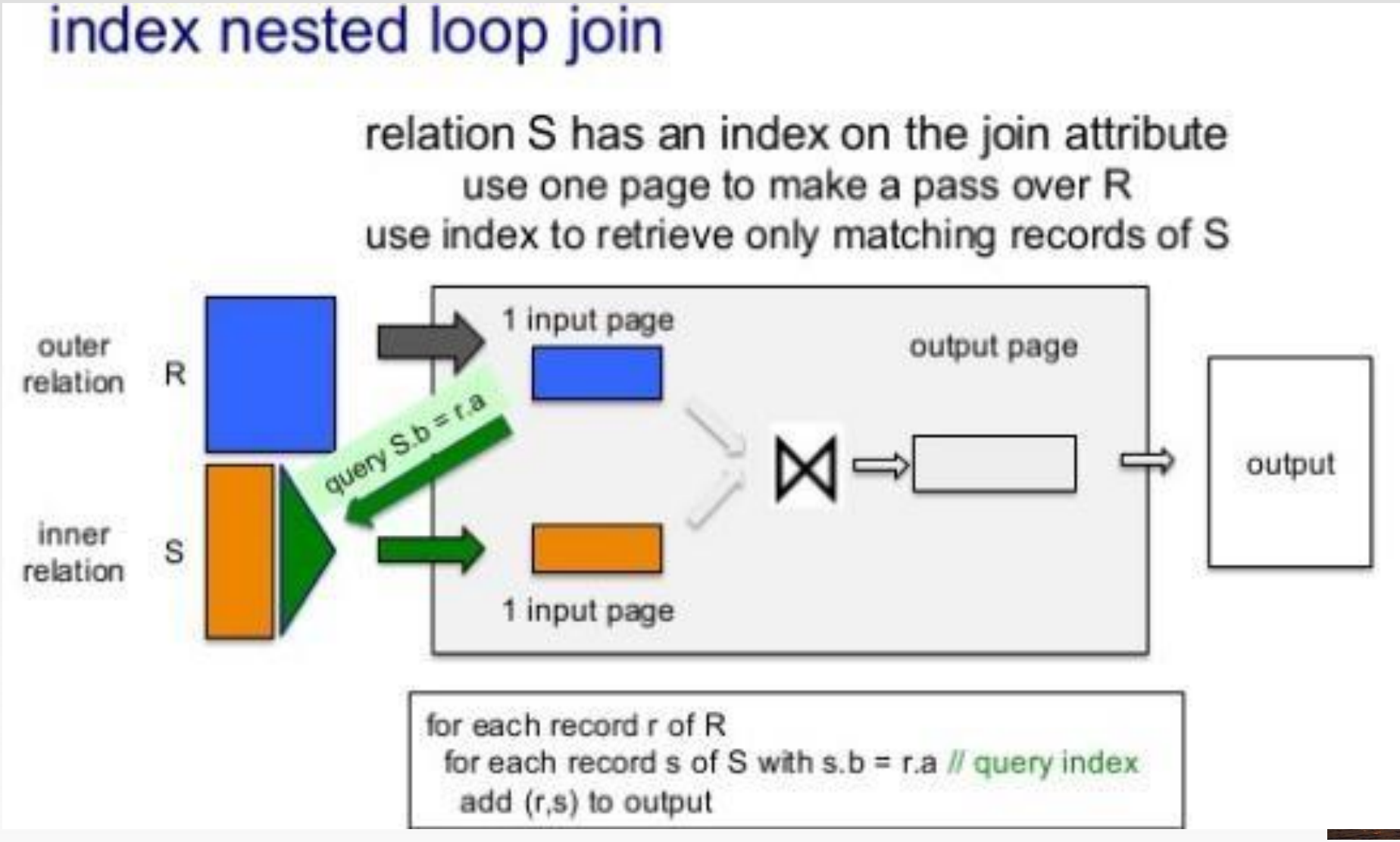
Partitioned Parallel Hash-Join (Cont.) – Figure 2 (b)



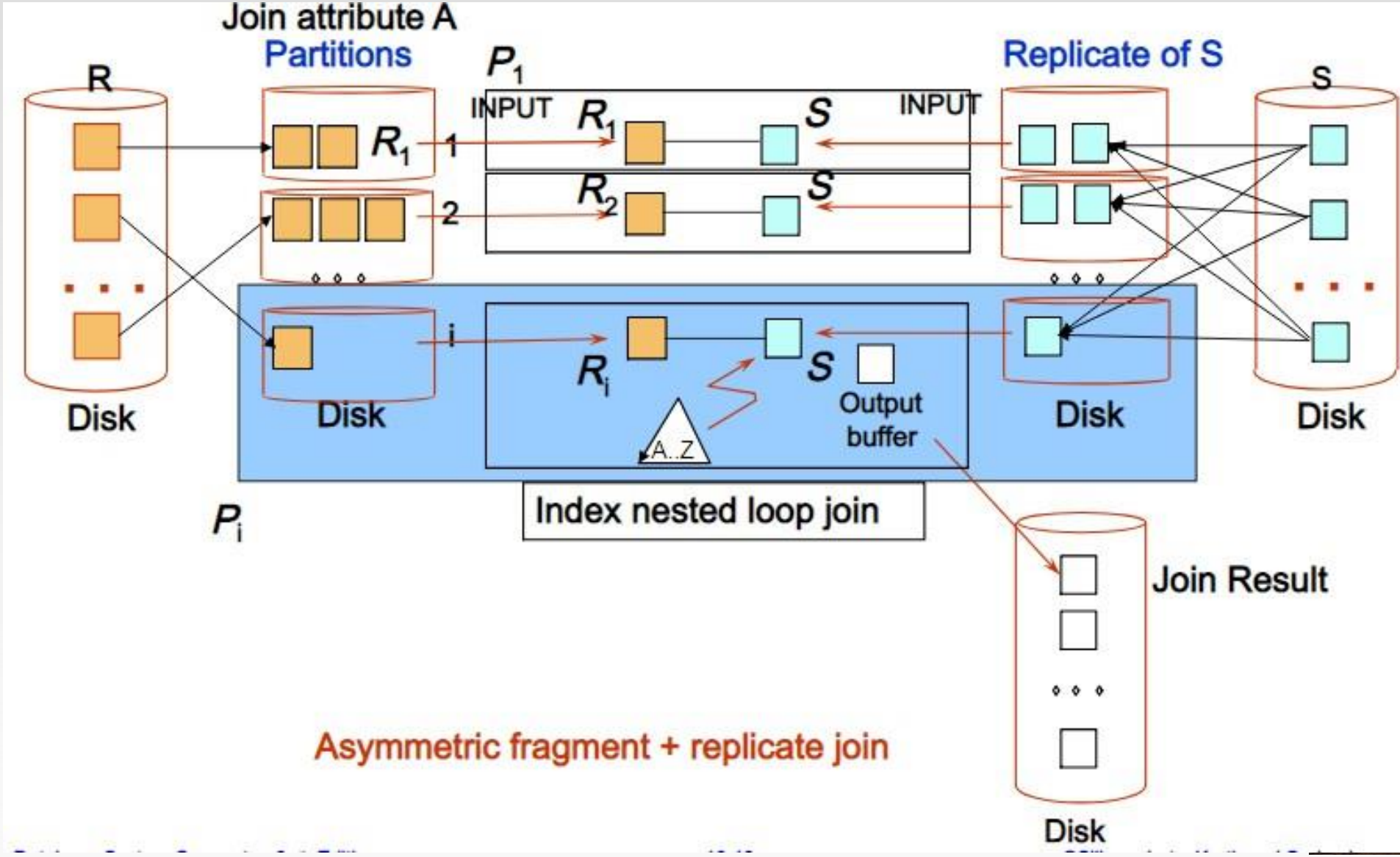
Parallel Nested-Loop Join

- Assume that
  - relation *s* is much smaller than relation *r* and that *r* is stored by partitioning.
  - there is an index on a join attribute of relation *r* at each of the partitions of relation *r*.
- Use asymmetric fragment-and-replicate, with relation *s* being replicated, and using the existing partitioning of relation *r*.
- 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*.
- Each processor P_i performs an indexed nested-loop join of relation *s* with the i^{th} partition of relation *r*.

Parallel Nested-Loop Join – Index Nested loop join Figure



Parallel Nested-Loop Join - Figure



Other Relational Operations

Selection $\sigma_\theta(r)$

- If θ is of the form $a_i = v$, where a_i is an attribute and v a value.
  - If r is partitioned on a_i the selection is performed at a single processor.
- If θ is of the form $l <= a_i <= u$ (i.e., θ is a range selection) and the relation has been range-partitioned on a_i
  - Selection is performed at each processor whose partition overlaps with the specified range of values.
- In all other cases: the selection is performed in parallel at all the processors.

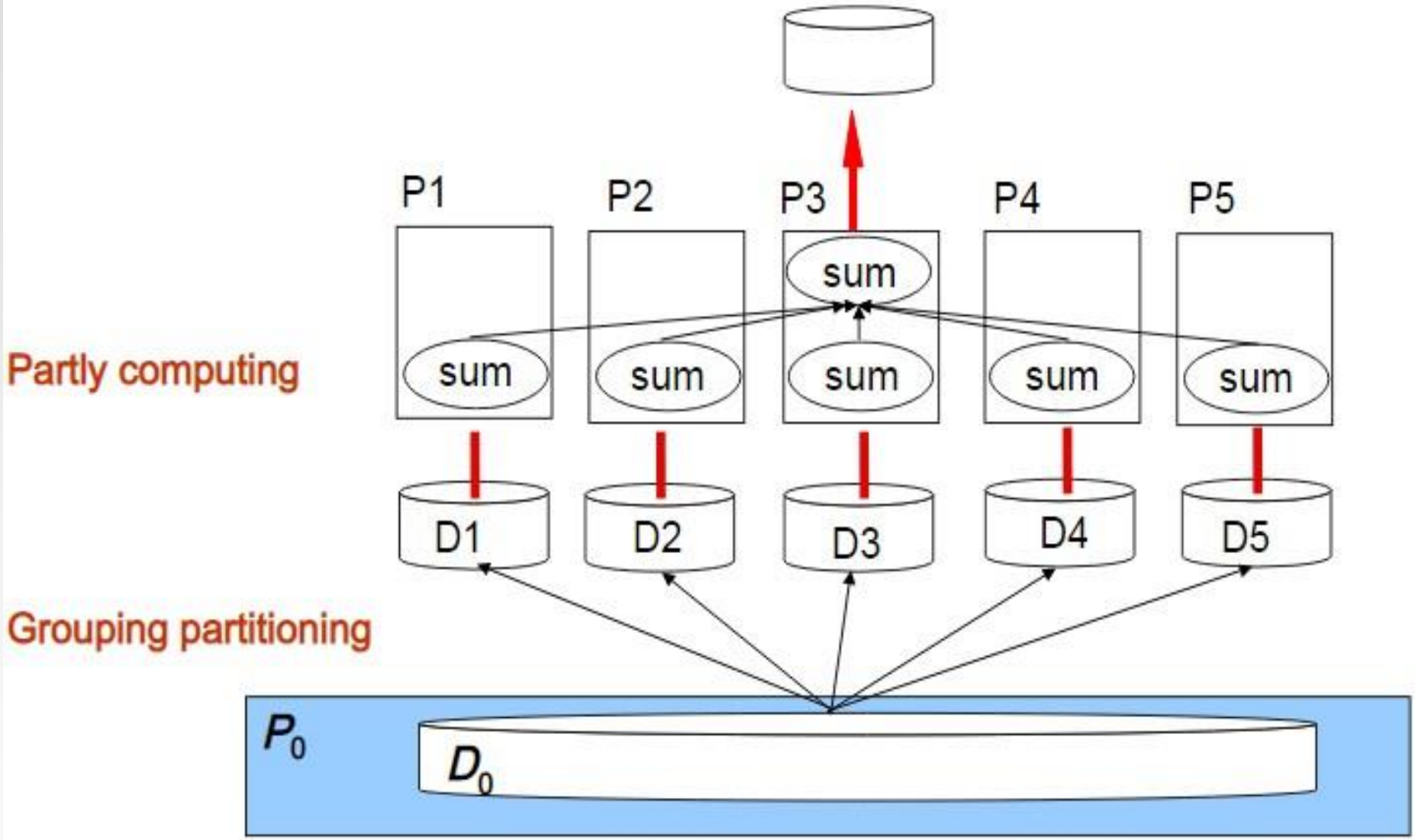
Other Relational Operations (Cont.)

- Duplicate elimination
  - Perform by using either of the parallel sort techniques
    - eliminate duplicates as soon as they are found during sorting.
  - Can also partition the tuples (using either range- or hash-partitioning) and perform duplicate elimination locally at each processor.
- Projection
  - Projection without duplicate elimination can be performed as tuples are read in from disk in parallel.
  - If duplicate elimination is required, any of the above duplicate elimination techniques can be used.

Grouping/Aggregation

- Partition the relation on the grouping attributes and then compute the aggregate values locally at each processor.
- Can reduce cost of transferring tuples during partitioning by partly computing aggregate values before partitioning.
- Consider the **sum** aggregation operation:
  - Perform aggregation operation at each processor P_i on those tuples stored on disk D_i
    - results in tuples with partial sums at each processor.
  - Result of the local aggregation is partitioned on the grouping attributes, and the aggregation performed again at each processor P_i to get the final result.
- Fewer tuples need to be sent to other processors during partitioning.

Grouping/Aggregation



Cost of Parallel Evaluation of Operations

- If there is no skew in the partitioning, and there is no overhead due to the parallel evaluation, expected speed-up will be 1/n
- If skew and overheads are also to be taken into account, the time taken by a parallel operation can be estimated as

$$T_{part} + T_{asm} + \max(T_0, T_1, \ldots, T_{n-1})$$

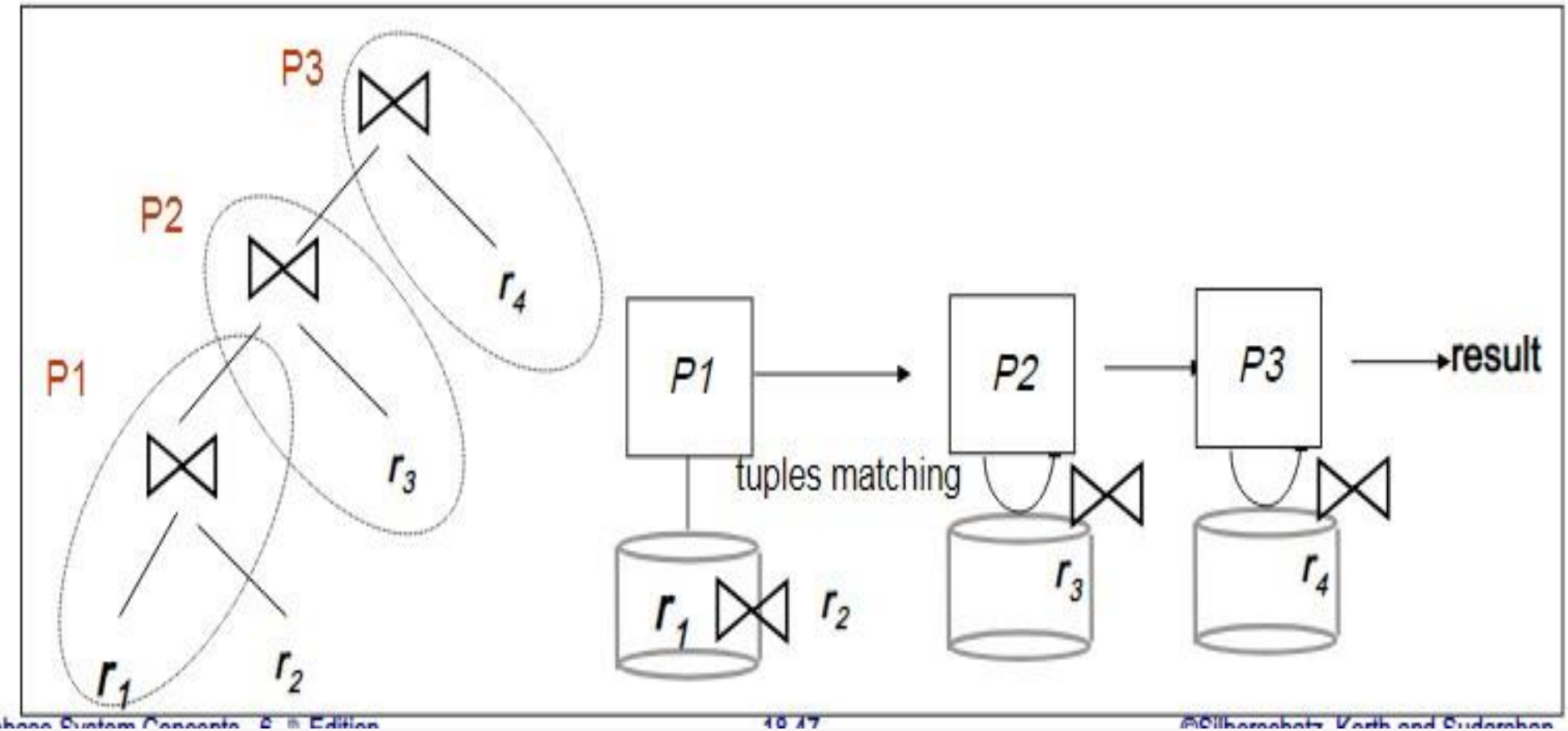
  Where,
  - T_{part} is the time for partitioning the relations
  - T_{asm} is the time for assembling the results
  - T_i is the time taken for the operation at processor P_i
- this needs to be estimated taking into account the skew, and the time wasted in contentions for resources(such as memory, disk and communication n/w)

Interoperator Parallelism

Pipelined parallelism

- Consider a join of four relations
  - $r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$
- Set up a pipeline that computes the three joins in parallel
  - Let P1 be assigned the computation of temp1 = $r_1 \bowtie r_2$
  - And P2 be assigned the computation of temp2 = temp1 $\bowtie r_3$
  - And P3 be assigned the computation of temp2 $\bowtie r_4$
- Each of these operations can execute in parallel, sending result tuples it computes to the next operation even as it is computing further results
  - Provided a pipelineable join evaluation algorithm (e.g. indexed nested loops join) is used

Interoperator Parallelism- Pipelined Parallelism Figure



Factors Limiting Utility of Pipeline Parallelism

- Pipeline parallelism is useful since it avoids writing intermediate results to disk
- Useful with small number of processors, but does not scale up well with more processors. One reason is that pipeline chains do not attain sufficient length.
- Cannot pipeline operators which do not produce output until all inputs have been accessed (e.g. aggregate and sort)
- Little speedup is obtained for the frequent cases of skew in which one operator's execution cost is much higher than the others.

Independent Parallelism

- **Independent parallelism**
  - Consider a join of four relations $r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$
    - Let P_1 be assigned the computation of $temp1 = r_1 \bowtie r_2$
    - And P_3 be assigned the computation of $temp1 \bowtie temp_2$
    - And P_2 be assigned the computation of $temp2 = r_3 \bowtie r_4$
    - P_1 and P_2 can work **independently in parallel**
    - P_3 has to wait for input from P_1 and P_2
      - Can pipeline output of P_1 and P_2 to P_3, combining independent parallelism and pipelined parallelism
  - Does not provide a high degree of parallelism
    - useful with a lower degree of parallelism.
    - less useful in a highly parallel system,

Design of Parallel Systems

Some issues in the design of parallel systems:

- Parallel loading of data from external sources is needed in order to handle large volumes of incoming data.
- Resilience to failure of some processors or disks.
  - Probability of some disk or processor failing is higher in a parallel system.
  - Operation (perhaps with degraded performance) should be possible in spite of failure.
  - Redundancy achieved by storing extra copy of every data item at another processor.

Design of Parallel Systems (Cont.)

- On-line reorganization of data and schema changes must be supported.
  - For example, index construction on terabyte databases can take hours or days even on a parallel system.
    - Need to allow other processing (insertions/deletions/updates) to be performed on relation even as index is being constructed.
  - Basic idea: index construction tracks changes and ``catches up'' on changes at the end.
- Also need support for on-line repartitioning and schema changes (executed concurrently with other processing).

End of Chapter

References

- Korth