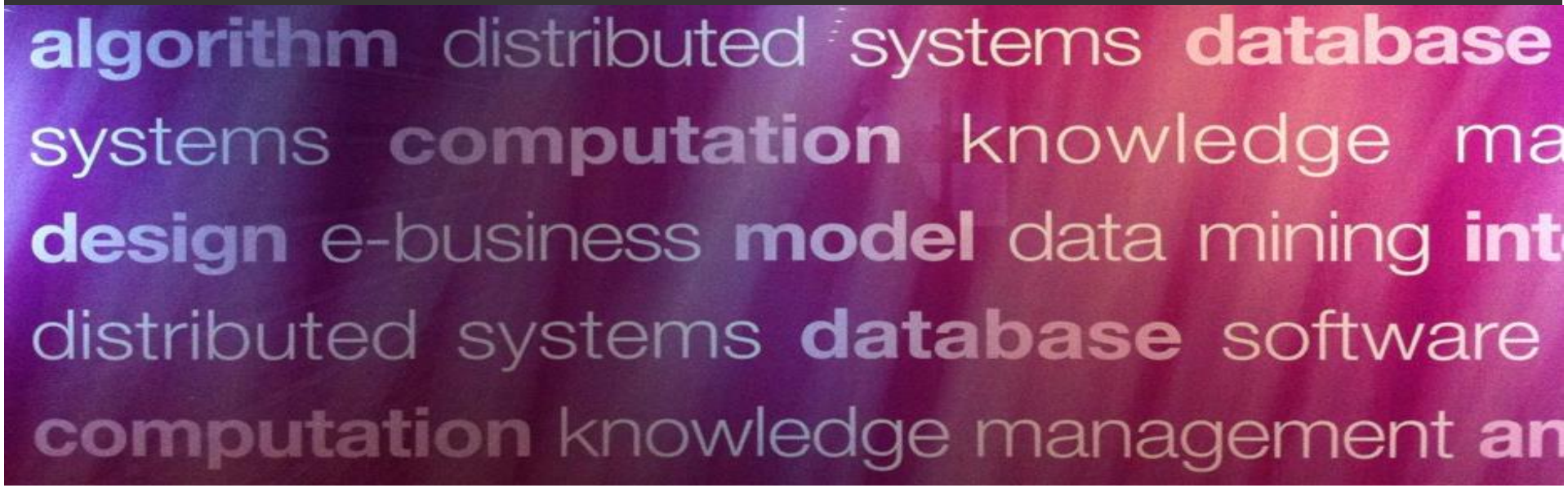
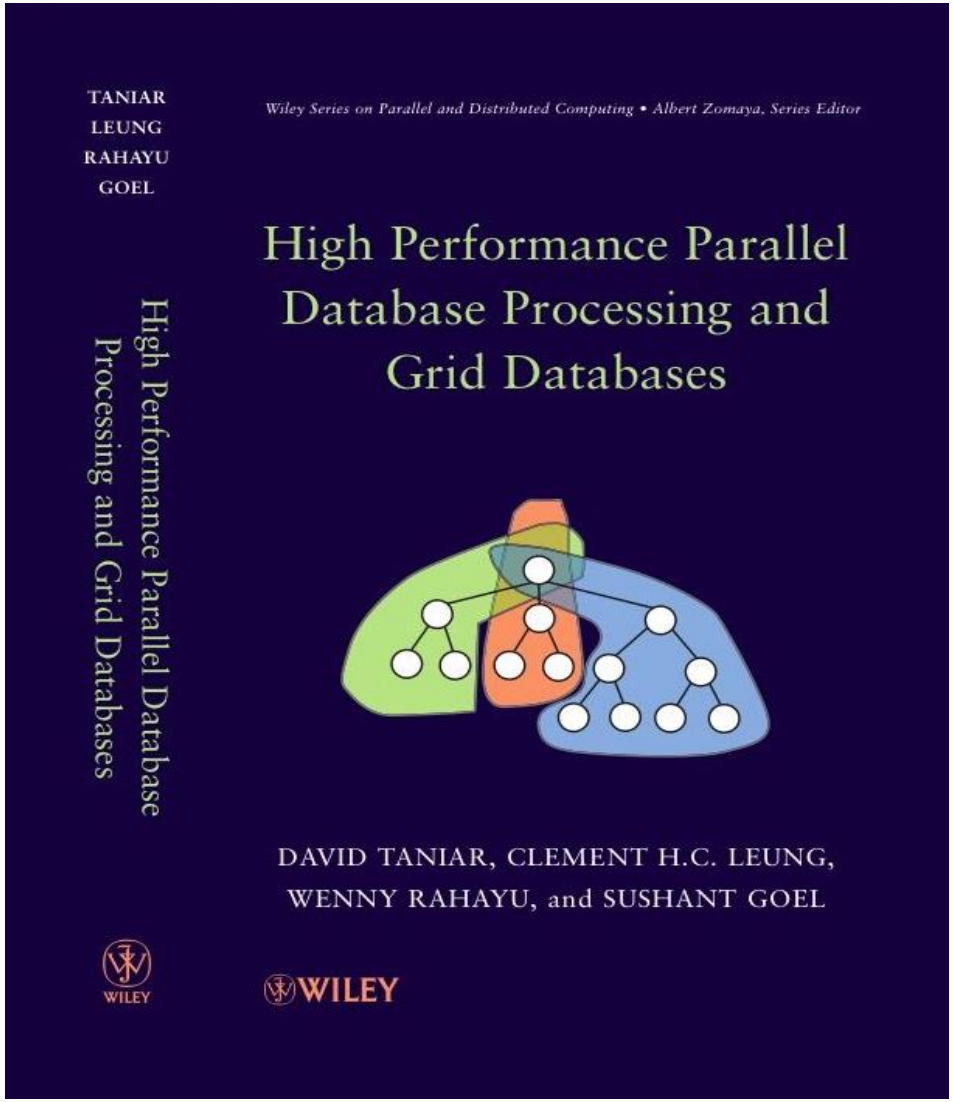
MONASH University

Information Technology

FIT5202 (Volume III - Join)

Week 3a – Parallel Join





Chapter 5
Parallel Join

Step 1: Data partitioning
Step 2: Perform parallel join in portioned data

5.1. Join Operations

- Join operations to link two tables based on the nominated attributes - one from each table

```
Query 5.1:
   Select *
   From STUDENT S, ENROLMENT E
   Where S.Sid = E.Sid;
```

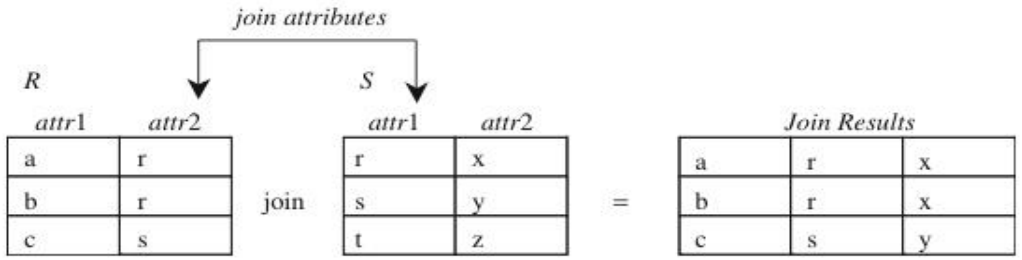


Figure 5.1 The join operation

5.2. Serial Join Algorithms

- Three serial join algorithms:
  - Nested loop join algorithm
  - Sort-merge join algorithm
  - Hash-based join algorithm

Table R

Adele	8
Bob	22
Clement	16
Dave	23
Ed	11
Fung	25
Goel	3
Harry	17
Irene	14
Joanna	2
Kelly	6
Lim	20
Meng	1
Noor	5
Omar	19

Table S

Arts	8
Business	15
CompSc	2
Dance	12
Engineering	7
Finance	21
Geology	10
Health	11
IT	18

Join Results

Adele	8	Arts
Ed	11	Health
Joanna	2	CompSc

Figure 5.2 Sample data

5.2. Serial Join Algorithms (cont'd)

- **Nested-Loop Join Algorithm**
  - For each record of table *R*, it goes through all records of table *S*

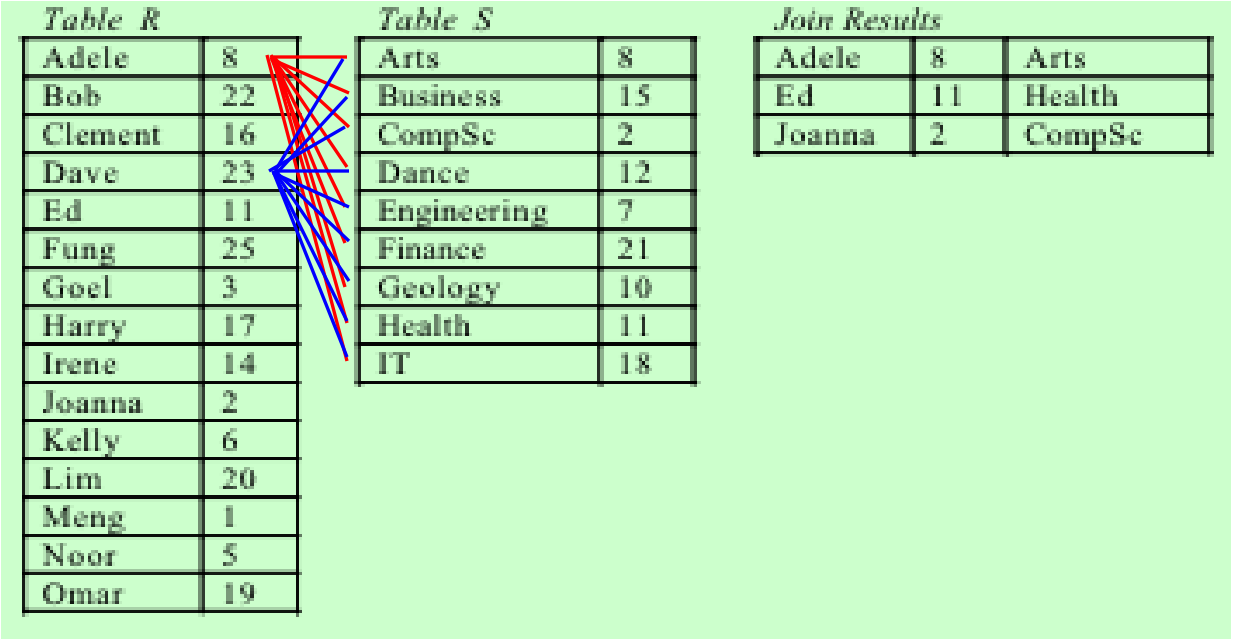


Table R

Adele	8
Bob	22
Clement	16
Dave	23
Ed	11
Fung	25
Goel	3
Harry	17
Irene	14
Joanna	2
Kelly	6
Lim	20
Meng	1
Noor	5
Omar	19

Table S

Arts	8
Business	15
CompSc	2
Dance	12
Engineering	7
Finance	21
Geology	10
Health	11
IT	18

Join Results

Adele	8	Arts
Ed	11	Health
Joanna	2	CompSc

5.2. Serial Join Algorithms (cont'd)

- **Sort-Merge Join Algorithm**
  - Both tables must be pre-sorted based on the join attribute(s). If not, then both tables must be sorted first
  - Then merge the two sorted tables

5.2. Serial Join Algorithms (cont'd)

Table R

Meng	1
Joanna	2
Goel	3
Noor	5
Kelly	6
Adele	8
Ed	11
Irene	14
Clement	16
Harry	17
Omar	19
Lim	20
Bob	22
Dave	23
Fung	25

Table S

CompSc	2
Engineering	7
Arts	8
Geology	10
Health	11
Dance	12
Business	15
IT	18
Finance	21

Join Results

Joanna	2	CompSc
Adele	8	Arts
Ed	11	Health

Figure 5.4. Sorted tables

5.2. Serial Join Algorithms (cont'd)

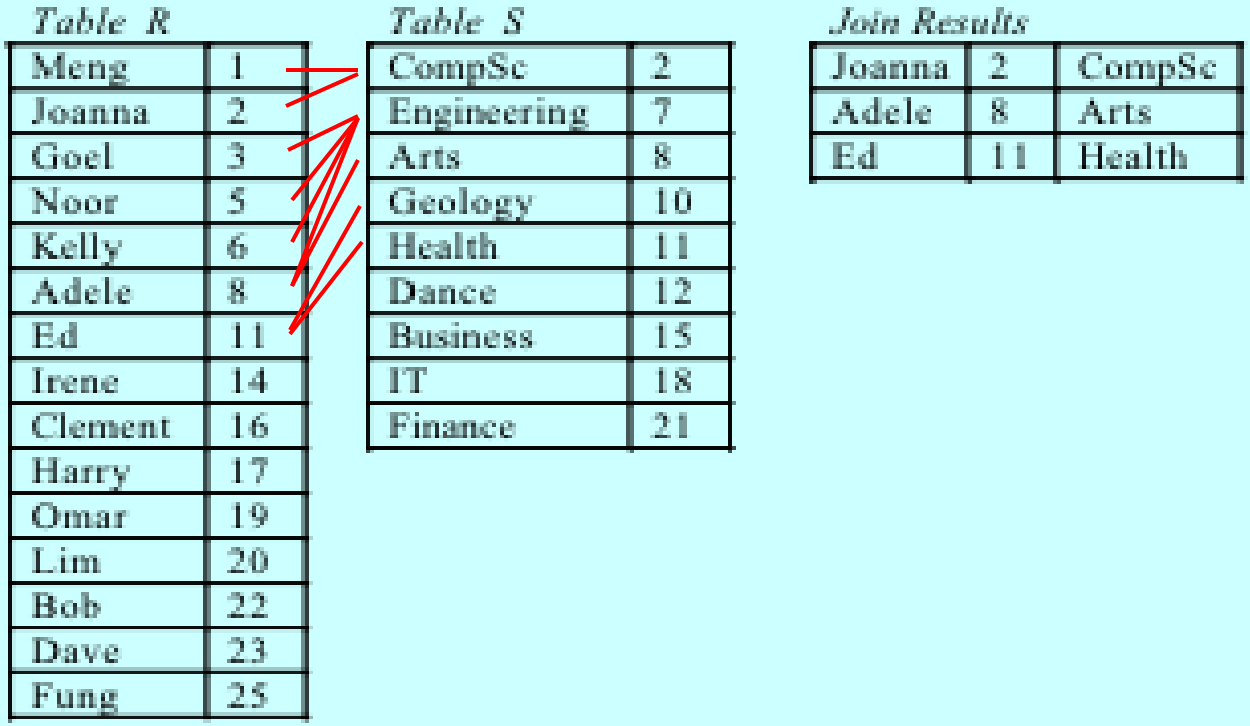


Table R

Meng	1
Joanna	2
Goel	3
Noor	5
Kelly	6
Adele	8
Ed	11
Irene	14
Clement	16
Harry	17
Omar	19
Lim	20
Bob	22
Dave	23
Fung	25

Table S

CompSc	2
Engineering	7
Arts	8
Geology	10
Health	11
Dance	12
Business	15
IT	18
Finance	21

Join Results

Joanna	2	CompSc
Adele	8	Arts
Ed	11	Health

Figure 5.4. Sorted tables

5.2. Serial Join Algorithms (cont'd)

- **Hash-based Join Algorithm**
  - The records of files *R* and *S* are both hashed to the *same hash file*, using the *same hashing function* on the join attributes *A* of *R* and *B* of *S* as hash keys
  - A single pass through the file with fewer records (say, *R*) hashes its records to the hash file buckets
  - A single pass through the other file (*S*) then hashes each of its records to the appropriate bucket, where the record is combined with all matching records from *R*

5.2. Serial Join Algorithms (cont'd)

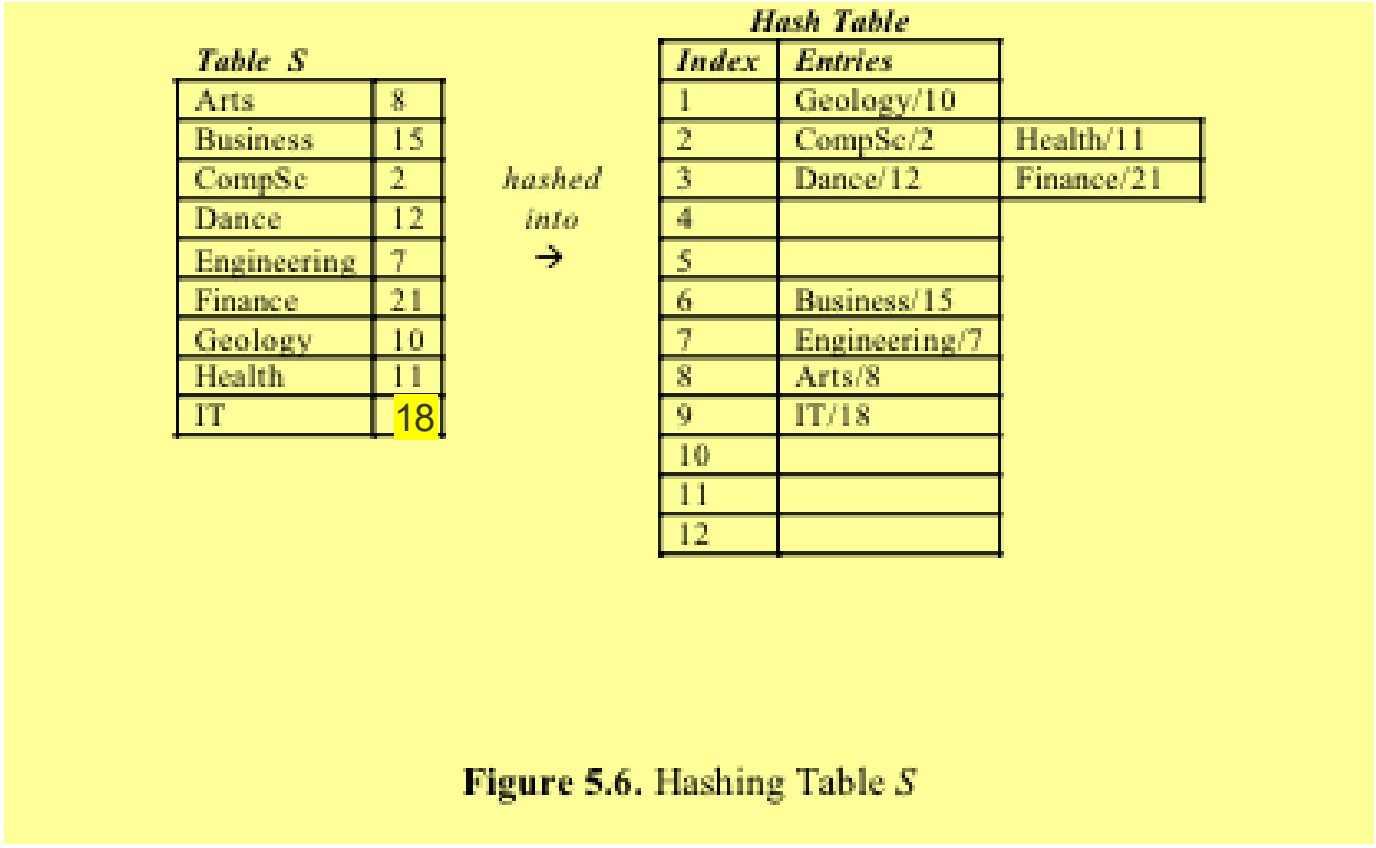


Table S

Arts	8
Business	15
CompSc	2
Dance	12
Engineering	7
Finance	21
Geology	10
Health	11
IT	18

hashed into →

Hash Table

Index	*Entries*	
1	Geology/10	
2	CompSc/2	Health/11
3	Dance/12	Finance/21
4		
5		
6	Business/15	
7	Engineering/7	
8	Arts/8	
9	IT/18	
10		
11		
12		

Figure 5.6. Hashing Table *S*

5.2. Serial Join Algorithms (cont'd)

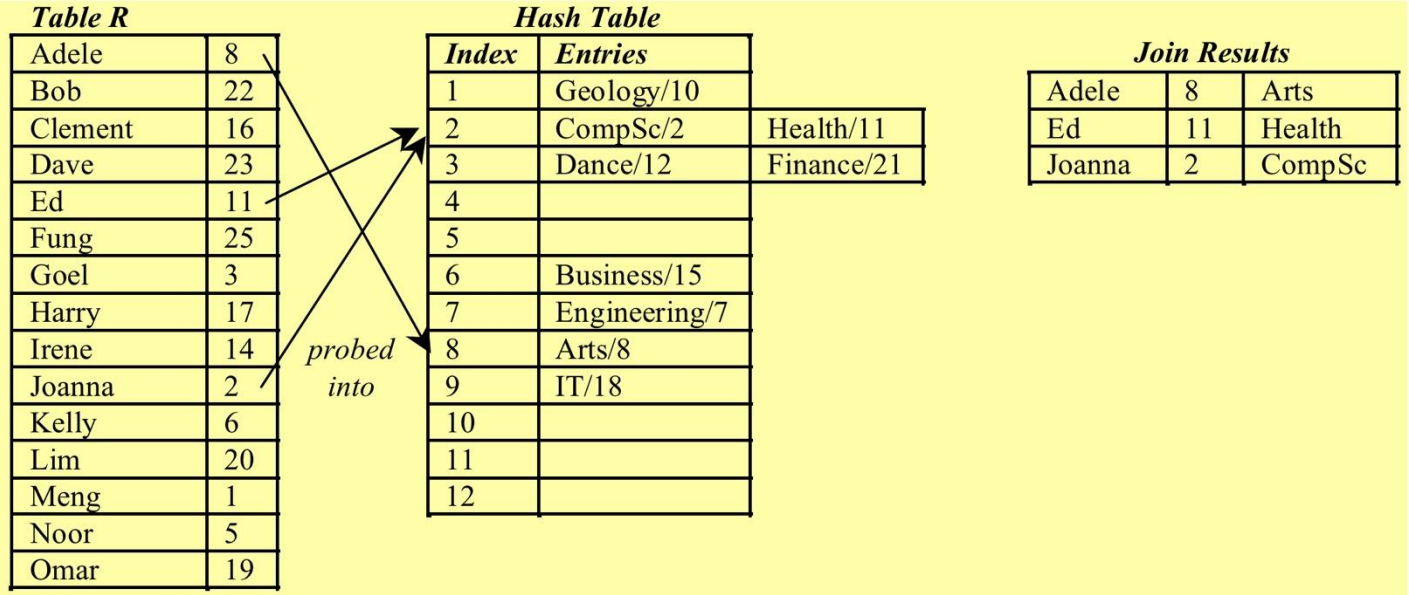
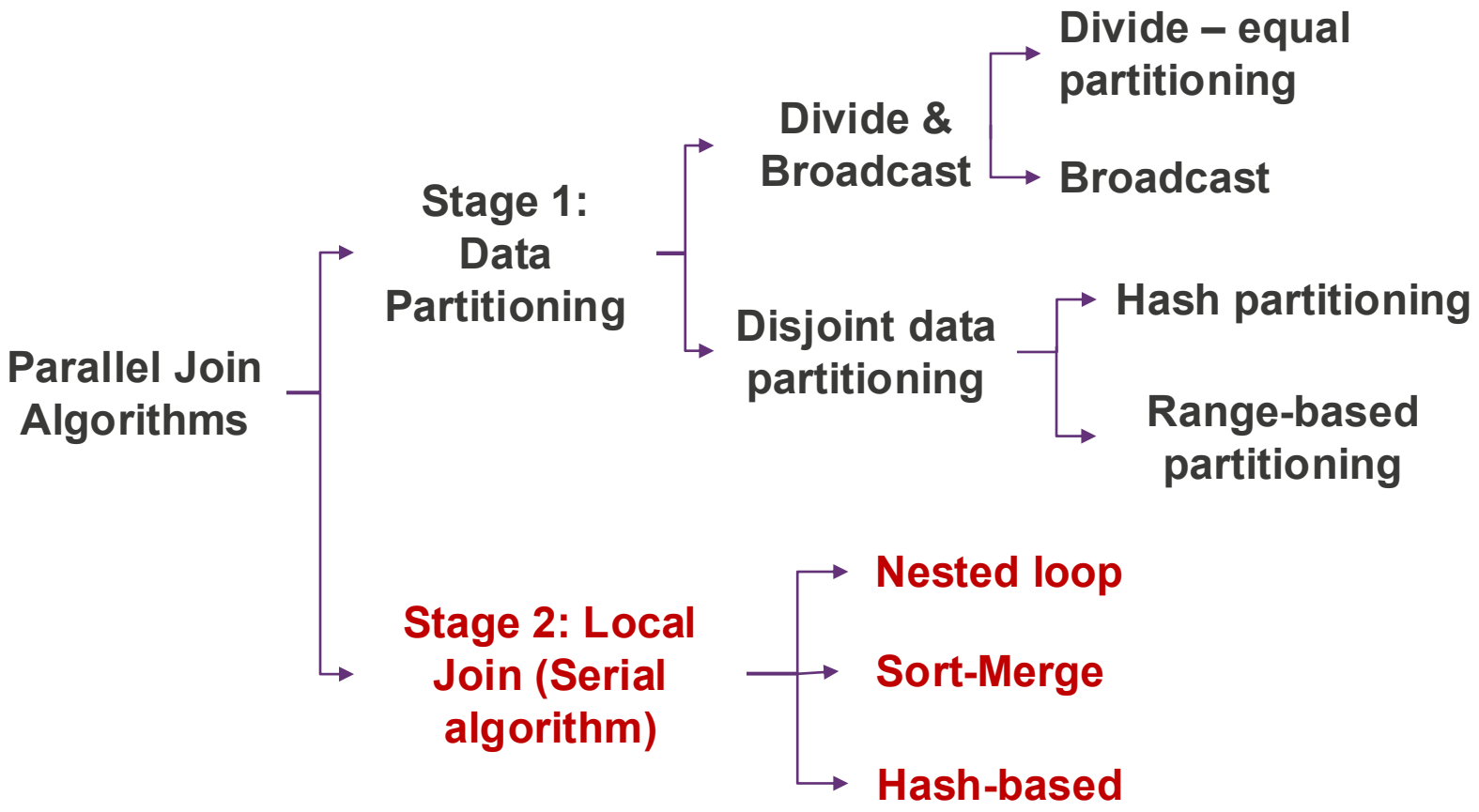


Figure 5.7. Probing Table *R*

5.3. Parallel Join Algorithms

- Parallelism of join queries is achieved through *data parallelism*, whereby the same task is applied to different parts of the data
- After data partitioning is completed, each processor will have its own data to work with using any serial join algorithm
- Data partitioning for parallel join algorithms:
  - Divide and broadcast
  - Disjoint data partitioning

Overview


Parallel Join Algorithms
Stage 1: Data Partitioning
Divide & Broadcast
Divide – equal partitioning
Broadcast
Disjoint data partitioning
Hash partitioning
Range-based partitioning
Stage 2: Local Join (Serial algorithm)
Nested loop
Sort-Merge
Hash-based

5.3. Parallel Join Algorithms (cont'd)

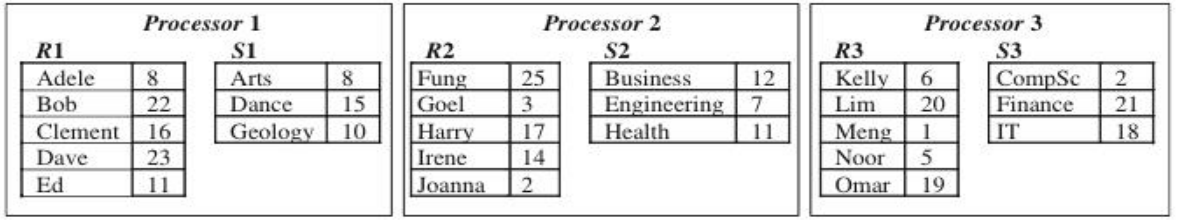
- **Divide and Broadcast-based Parallel Join Algorithms**
  - **Two stages**
    - **(1)** Data partitioning using the divide and broadcast method
    - **(2)** Local join
  - **(1) Divide and Broadcast method**:
    - **(1.1)** Divide one table into multiple disjoint partitions, where each partition is allocated a processor
      - Dividing one table can simply use equal division
    - **(1.2)** Broadcast the other table to all available processors
      - Broadcast means replicate the table to all processors
    - Hence, choose the smaller table to broadcast and the larger table to divide

5.3. Parallel Join Algorithms (cont'd)

Assume data is initially partitioned (based on non-join attributes), and stored in each processor

R: equally partitioned based on range
S: round-robin partitioning

Example: Shared-nothing architecture



Processor 1

R1	
Adele	8
Bob	22
Clement	16
Dave	23
Ed	11

S1	
Arts	8
Dance	15
Geology	10

Processor 2

R2	
Fung	25
Goel	3
Harry	17
Irene	14
Joanna	2

S2	
Business	12
Engineering	7
Health	11

Processor 3

R3	
Kelly	6
Lim	20
Meng	1
Noor	5
Omar	19

S3	
CompSc	2
Finance	21
IT	18

Figure 5.10 Initial data placement

5.3. Parallel Join Algorithms (cont'd)

Broadcast the smaller table

Each processor has one partition of R, and entire table of S

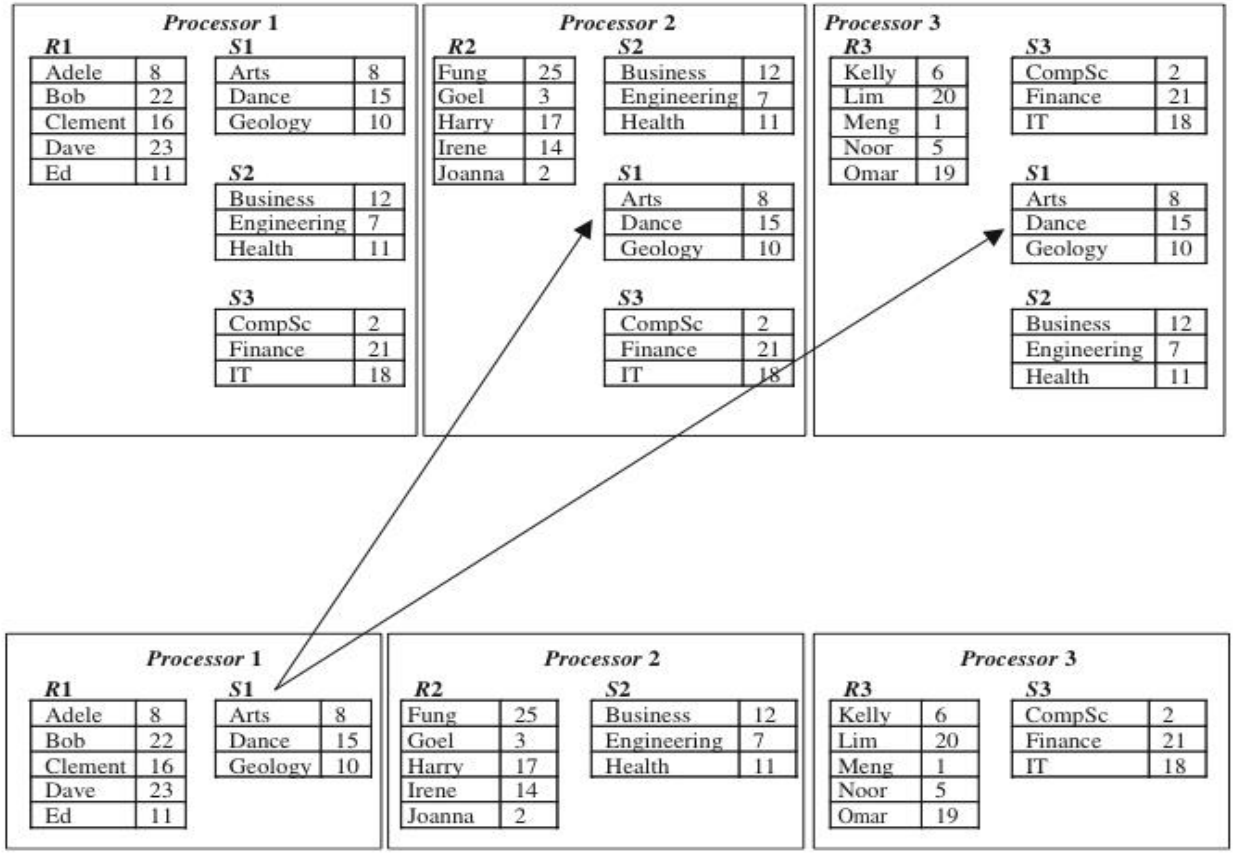


Figure 5.11 Divide and broadcast result

5.3. Parallel Join Algorithms (cont'd)

Local join for each partition

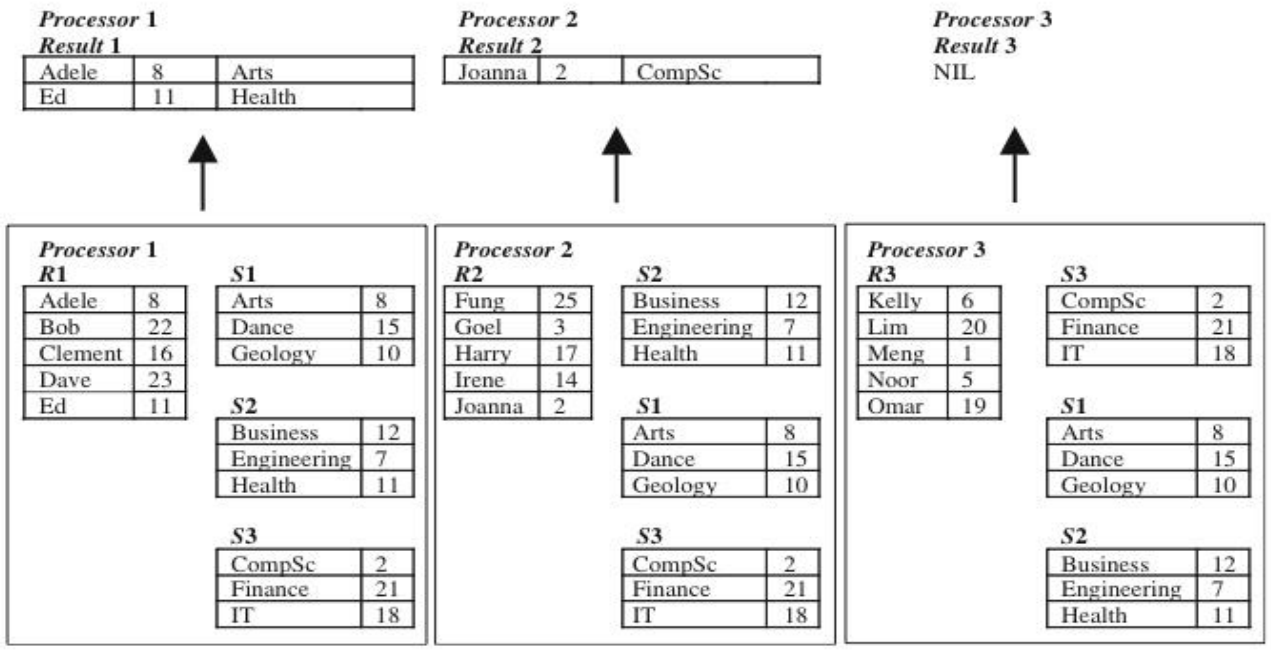


Figure 5.12 Join results based on divide and broadcast

5.3. Parallel Join Algorithms (cont'd)

- **Divide and Broadcast-based Parallel Join Algorithm**
  - There are broadcasting cost (network interference and communication cost), especially, if the table to be broadcasted is large.
  - No load imbalance problem, but the broadcasting method is inefficient
  - The problem of workload imbalance will occur if the table is already partitioned using random-unequal partitioning

5.3. Parallel Join Algorithms (cont'd)

Disjoint Partitioning-based Parallel Join Algorithms

- Two stages
  - (1) Data partitioning using a disjoint partitioning
    range or hash partitioning
  - (2) Local join
- any serial local join algorithm

5.3. Parallel Join Algorithms (cont'd)

- **Example 1: Range partitioning**
  - Redistribute/Reshuffle data based on join key using range logic

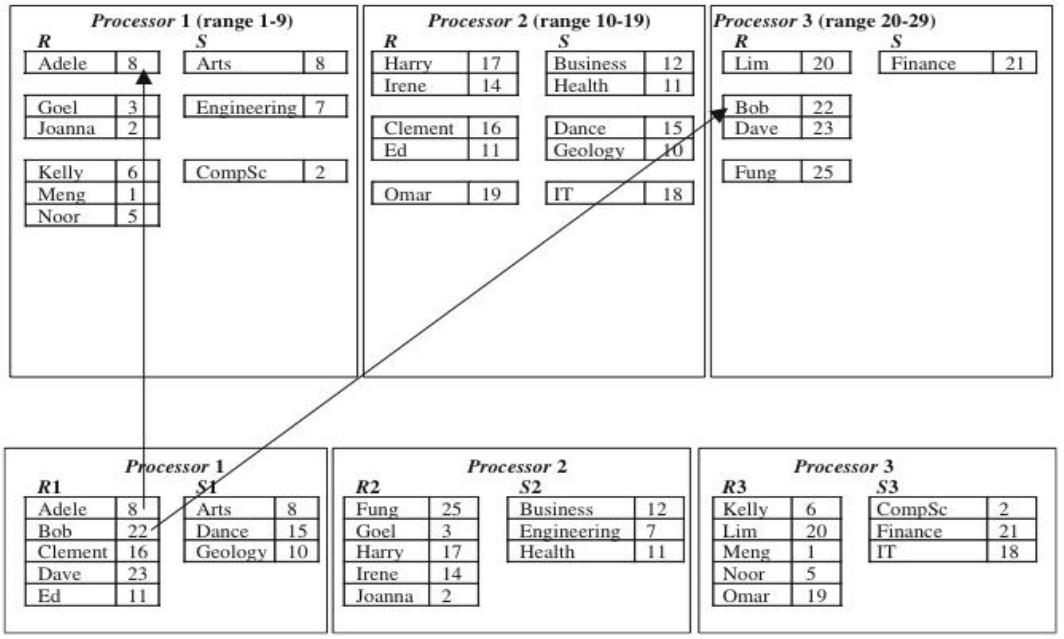


Figure 5.14 Range partitioning

5.3. Parallel Join Algorithms (cont'd)

- **Example 1: Range partitioning**
  - Perform local join

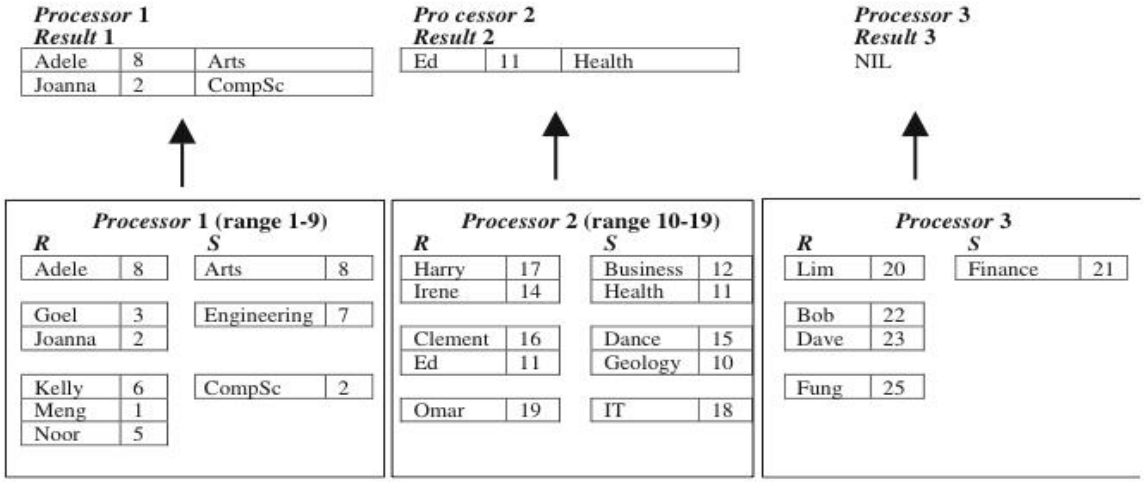


Figure 5.15 Join results based on range partitioning

5.3. Parallel Join Algorithms (cont'd)

- **Example 2: Hash partitioning**
  - Redistribute data based on join key using hash function (e.g. sum of digits and modulo)

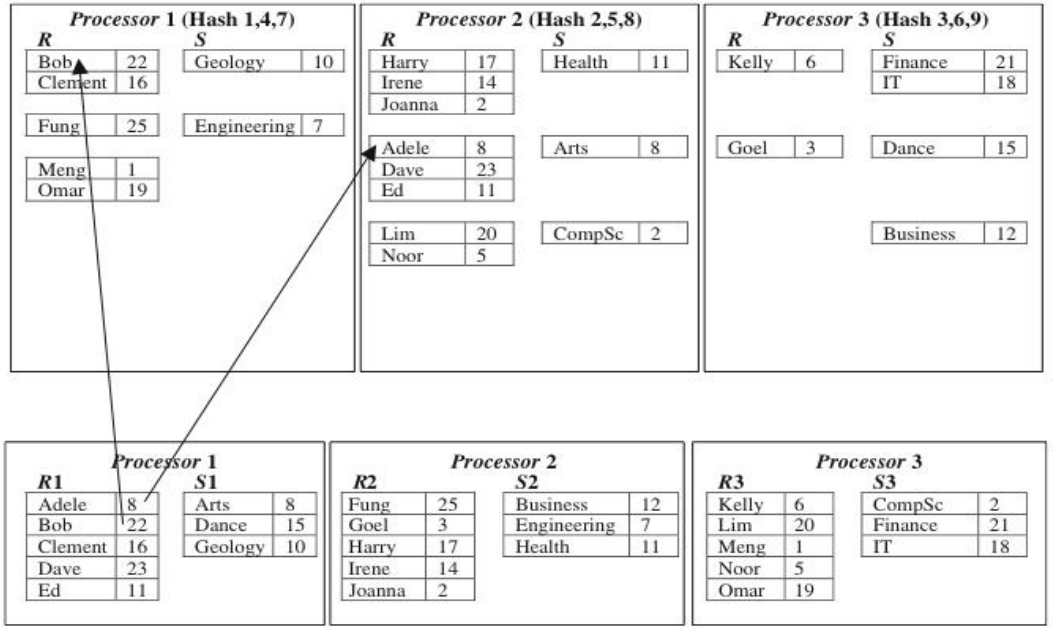


Figure 5.16 Hash partitioning

5.3. Parallel Join Algorithms (cont'd)

- **Example 2: Hash partitioning**
  - Perform local join

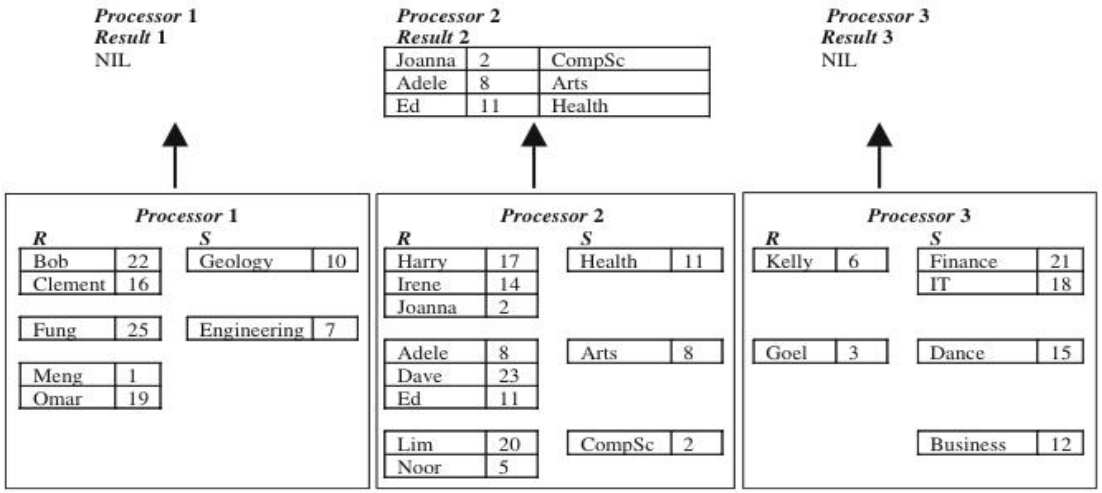


Figure 5.17 Join results based on hash partitioning

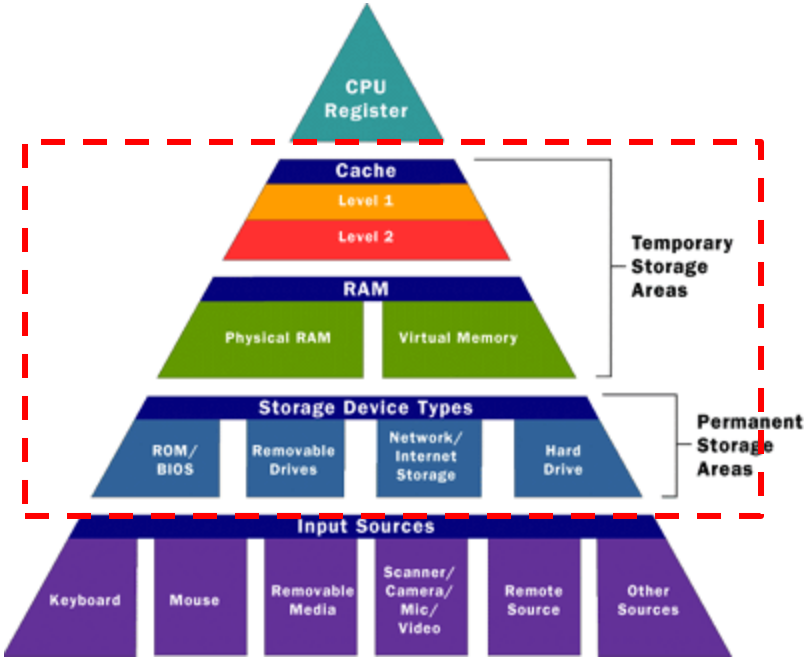
5.4. Cost Models for Parallel Join

- **Why Cost Model**
  - To measure effectiveness of parallelism (e.g., time) of database query processing
- **Divide and Broadcast**
  - Join two tables (**table R** and **table S**)
  - The two tables have been partitioned and stored in 3 processors
  - The tables have been partitioned using the random-equal data partitioning method
  - The table fragments are called R1, R2, R3, and S1, S2, S3 (in general, each fragment is called **Ri** or **Si**, where i is the processor number)

Initial data placement (random-equal partitioning)

5.4. Cost Models for Parallel Join (cont'd)

- **Divide and Broadcast**
  - **Phase 1: Data Loading (from disk to memory)**.
  - Assume the table to be broadcast is table S.
  - Read all records from table S in each processor



Initial data placement (random-equal partitioning)

5.4. Cost Models for Parallel Join (cont'd)

- **Divide and Broadcast**
  - **|S|** = 30,000 records
  - **N** = 3 processors
  - **|Si|** = 10,000 records (**|Si| = |S|/N**)

  - Each record has a fixed length, in bytes
  - The size of table S is denoted as S (in bytes)

Table 2.1 Cost notations

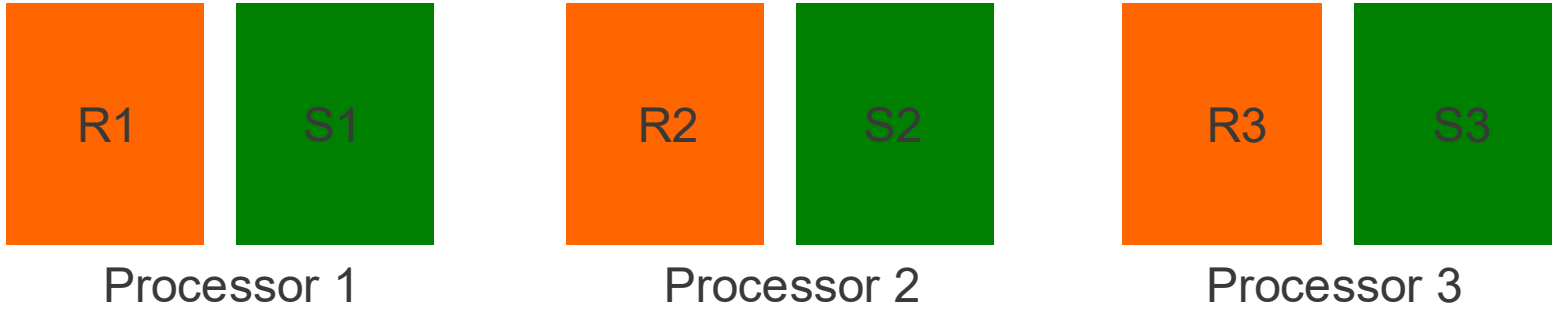
Symbol	Description
Data parameters	
R	Size of table in bytes
R_i	Size of table fragment in bytes on processor i
$\|R\|$	Number of records in table R
$\|R_i\|$	Number of records in table R on processor i
Systems parameters	
N	Number of processors
P	Page size
H	Hash table size
Query parameters	
π	Projectivity ratio
σ	Selectivity ratio
Time unit cost	
IO	Effective time to read a page from disk
t_r	Time to read a record in the main memory
t_w	Time to write a record to the main memory
t_d	Time to compute destination
Communication cost	
m_p	Message protocol cost per page
m_l	Message latency for one page

Initial data placement (random-equal partitioning)

5.4. Cost Models for Parallel Join (cont'd)

- **Divide and Broadcast**
  - Phase 1: Data Loading
    - When a table fragment is read from disk, it is based on the size of the table (in bytes), and not based how many records.
    - Hence, it uses Si, and not |Si|
    - When the disk reads a table fragment (**Si**), it reads a disk block at a time. The size of a disk block is **P** (or page size)
    - The loading time or **Scan cost = (Si/P) x IO**, where IO is the time taken to load 1 page from disk to main memory

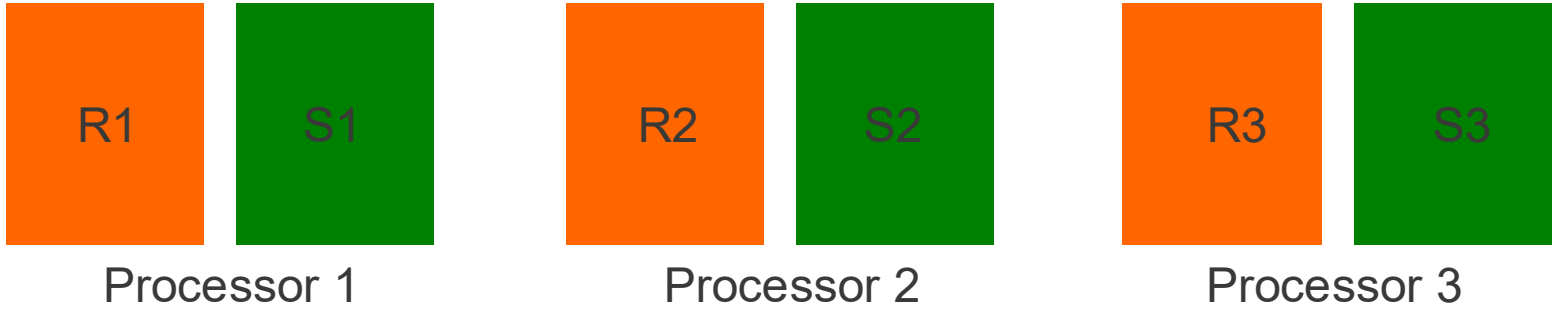
Initial data placement (random-equal partitioning)



5.4. Cost Models for Parallel Join (cont'd)

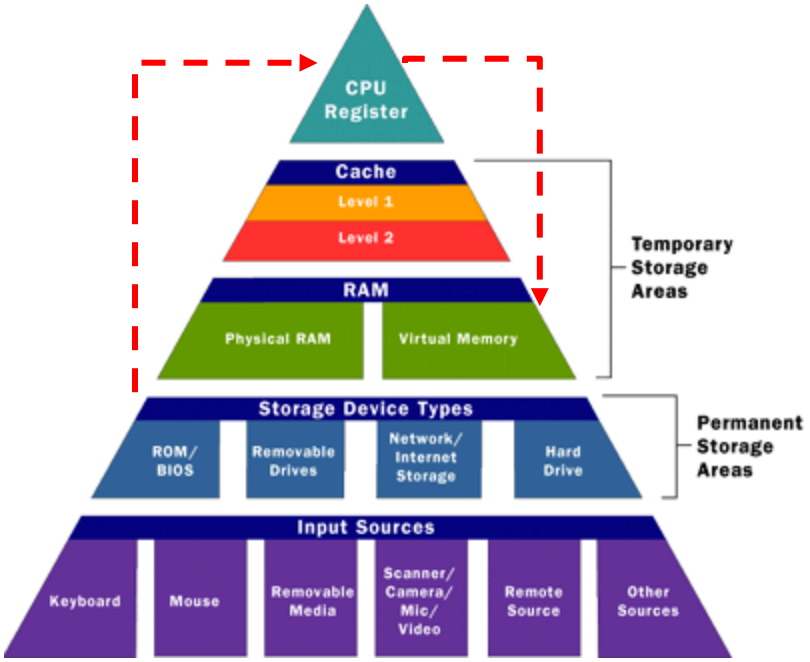
- **Divide and Broadcast**
  - Phase 1: Data Loading
    - (Si/P) = number of disk blocks/pages in fragment Si
    - Ex: (Si/P) = 20,000 bytes/ 4000 bytes = 5 pages
    - Assuming Reading 1 page takes 2ms
    - Scan cost = (Si/P) x IO = 5 × (2 ms) = 10ms

Initial data placement (random-equal partitioning)



5.4. Cost Models for Parallel Join (cont'd)

- **Divide and Broadcast**
  - Once the loading is complete, the records are not yet ready in the main memory. The records must be read by the CPU and be written to the main memory data page.
  - All processing in the processor is based on number of records, and not the byte size
  - **Select cost = |Si| x (tr + tw)**
  - tr is the reading time by the CPU, and tw is the writing time by the CPU



Initial data placement (random-equal partitioning)

5.4. Cost Models for Parallel Join (cont'd)

- **Cost Models for Divide and Broadcast**
  - Phase 1: data loading consists of the *scan costs* and the *select costs*

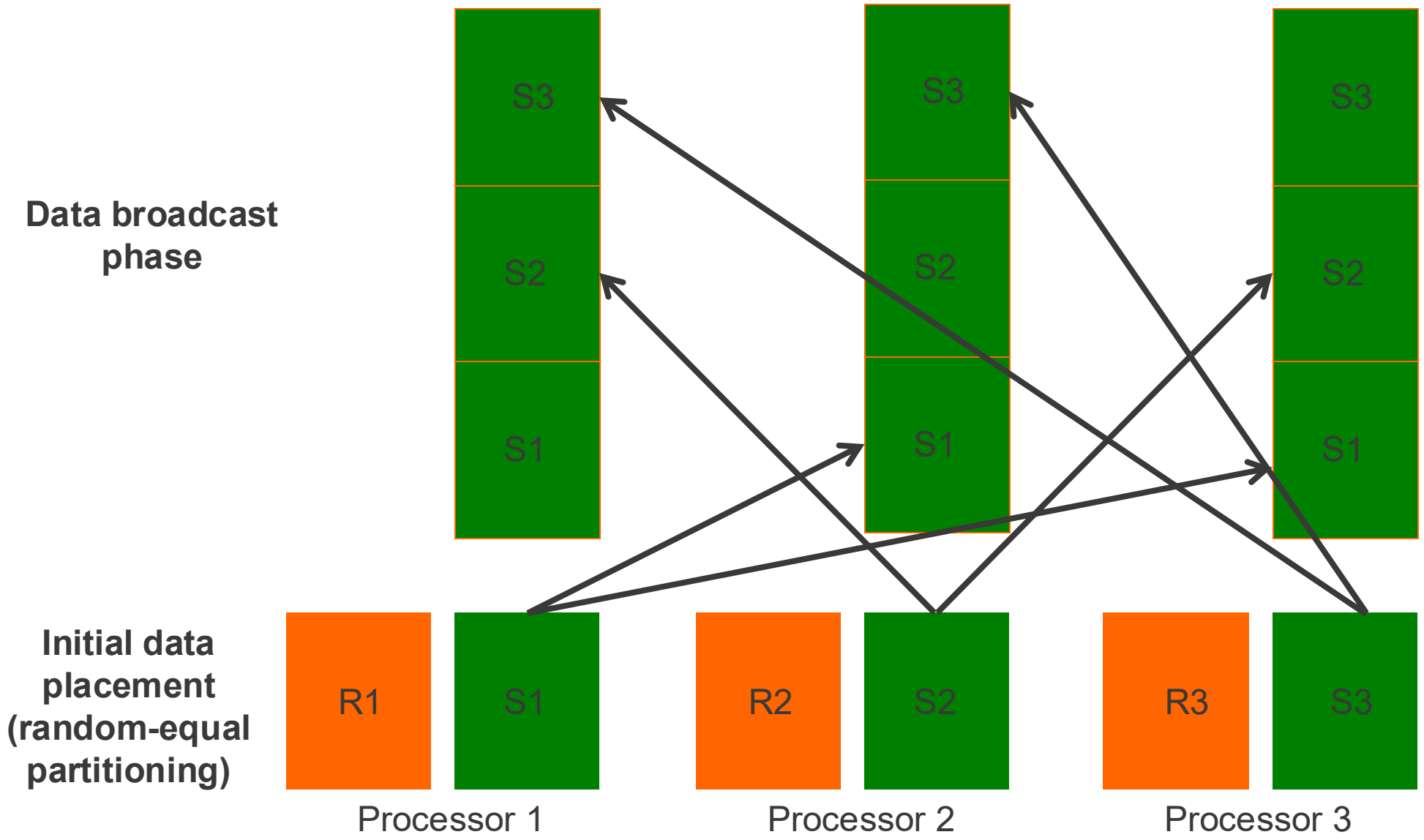
  - *Scan cost* for loading data from local disk in each processor is:

    $$(Si / P) \times IO$$

  - *Select cost* for getting record out of data page is:

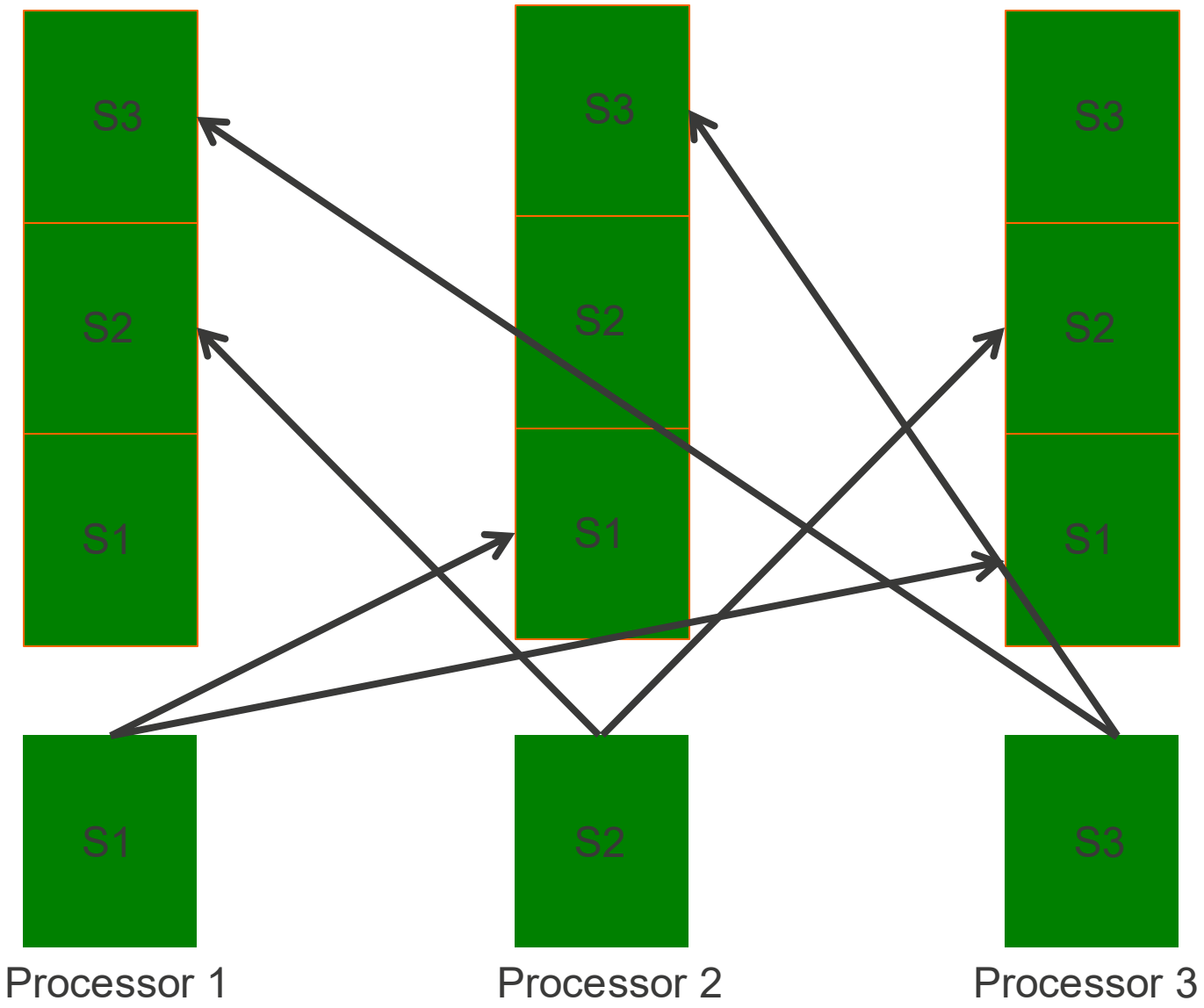
    $$|Si| \times (tr + tw)$$

Phase 2: Data Broadcasting. Table fragment S1 must be broadcasted (copied) to processors 2 and 3. Table fragment S2 to processors 1 and 3, etc…



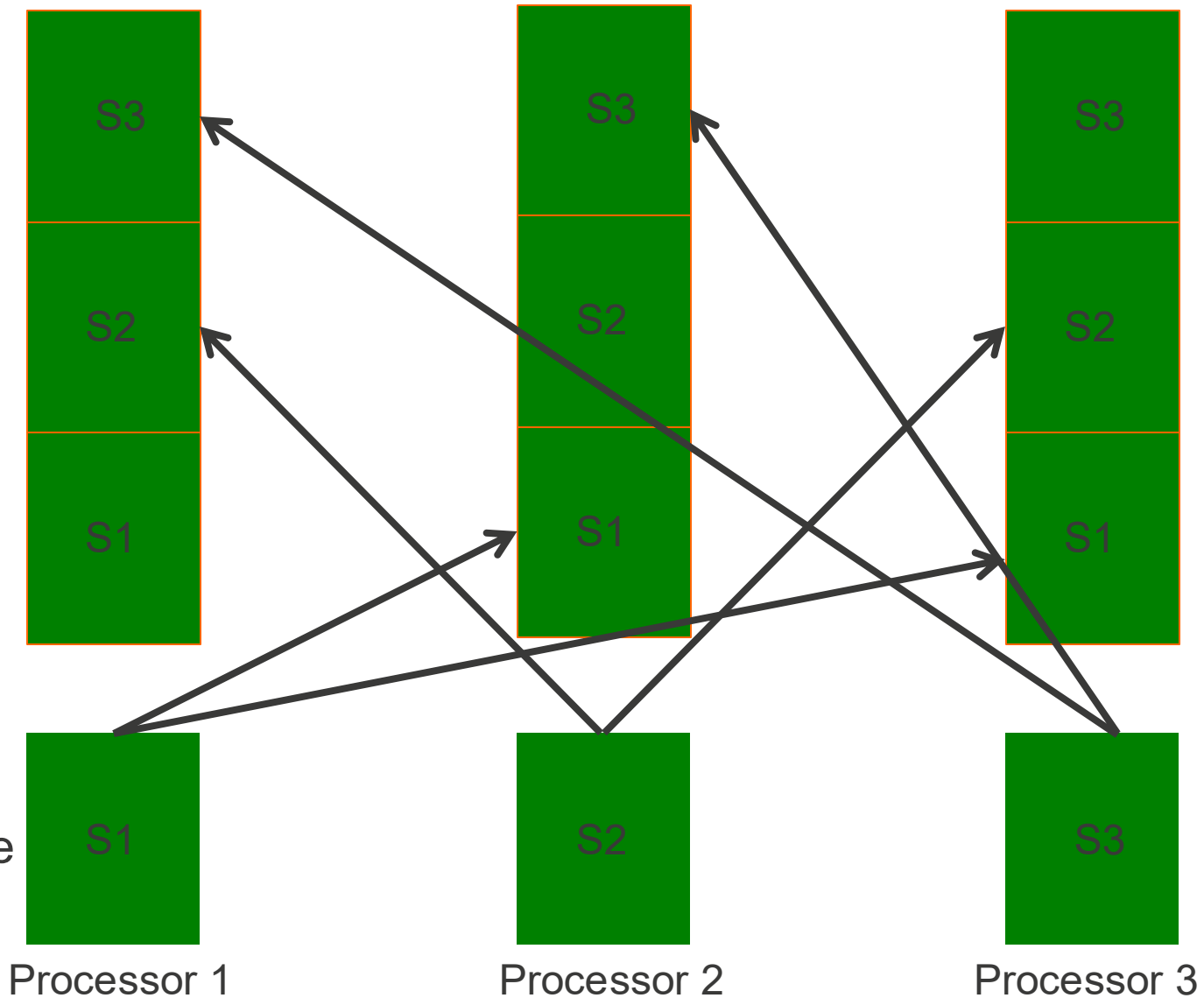
Phase 2: Data Broadcasting. Table fragment S1 must be broadcasted (copied) to processors 2 and 3. Table fragment S2 to processors 1 and 3, etc…

- Data broadcast is done through network (so it is a network data transfer)
- The transfer cost is based on **how many bytes** of data being transferred
- Hence, we use **Si**, instead of |Si|
- Data transfer is also done page-per-page (**P**)
- Hence, **Si/P**



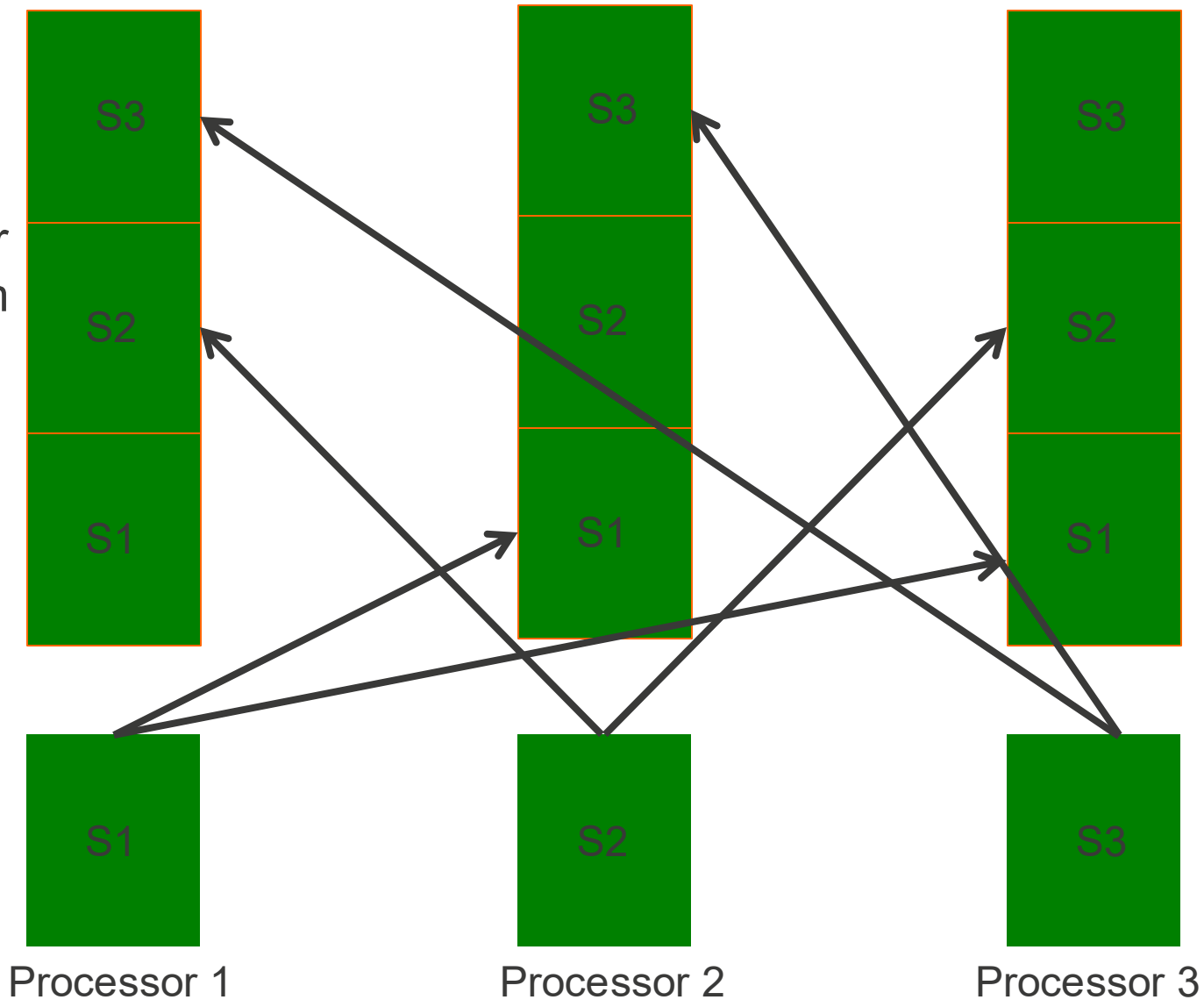
Phase 2: Data Broadcasting. Table fragment S1 must be broadcasted (copied) to processors 2 and 3. Table fragment S2 to processors 1 and 3, etc…

- S1 must be transferred twice (to processor 2 and to processor 3)
- **Transfer cost = (Si/P) x (N-1) x (mp+ml)**
- **mp (message protocol)** - cost to add TCP headers & other info for data transfer
- **ml (message latency)** - latency in transfer due to network



Phase 2: Data Broadcasting. Table fragment S1 must be broadcasted (copied) to processors 2 and 3. Table fragment S2 to processors 1 and 3, etc…

- Processor 2 must receive S1 from processor 1; Processor 3 must receive S1 from processor 1
- **Receiving cost = (S/P – Si/P) x (mp)**
- Why (S/P – Si/P)?
- Why (mp) only?
- mp - cost to unpack received data

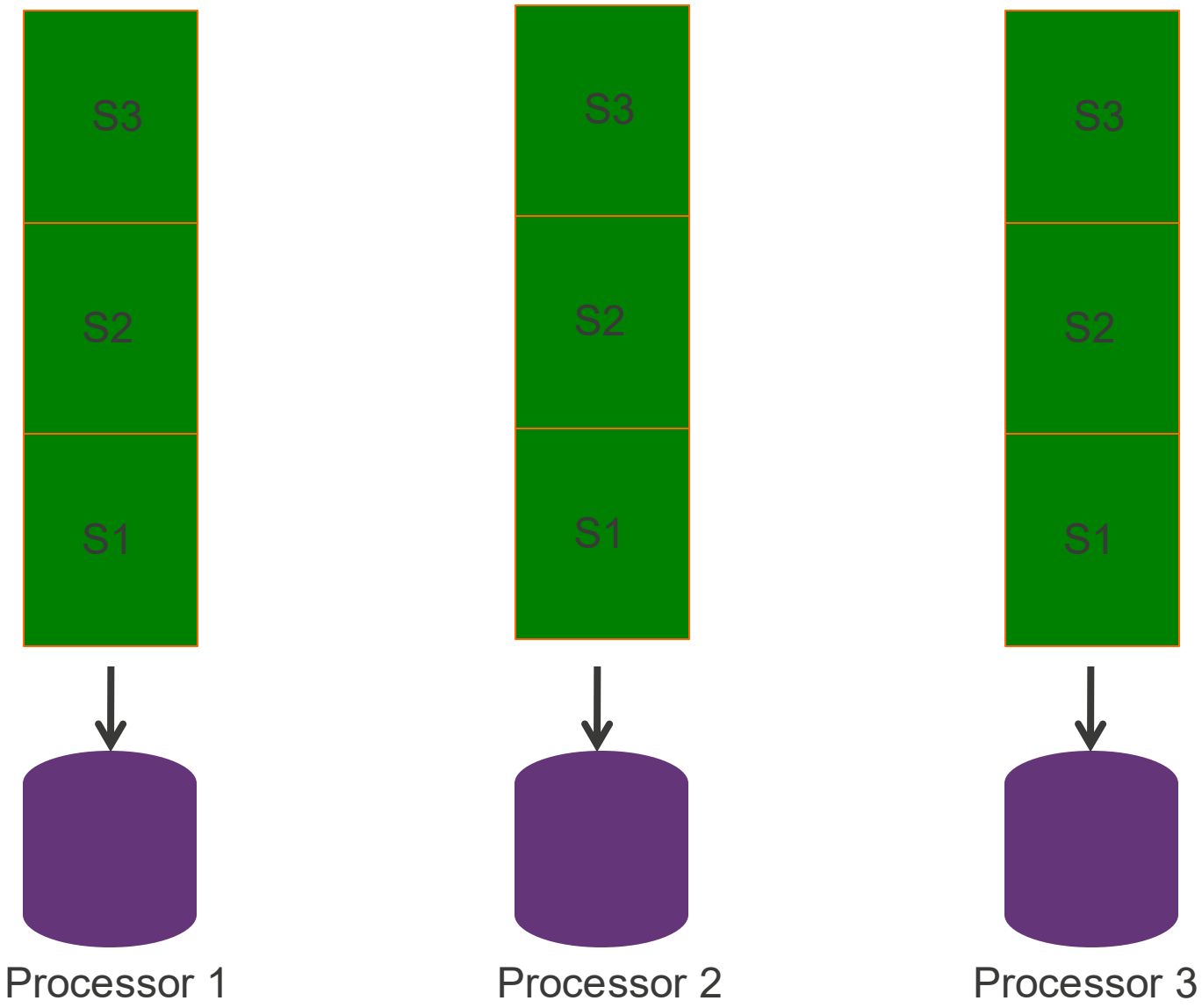


5.4. Cost Models for Parallel Join (cont'd)

- **Cost Models for Divide and Broadcast**
  - Phase 2: The broadcast cost by each processor broadcasting its fragment to all other processors
  - *Data transfer cost* is: (*Si / P*) x (*N* – 1) x (*mp* + *ml*)
  - The (*N*-1) indicates that each processor must broadcast to all other processors. Note that broadcasting from one processor to the others has to be done one processor at a time, although all processors send the broadcast in parallel. The above cost equation would be the same as (*S/P* – *Si/P*) x (*mp* + *ml*), where (*S/P* – *Si/P*) is the size of other fragments.
  - *Receiving records cost* is: (*S/P* – *Si/P*) x (*mp*)

Phase 3: Data Storing. Each fragment in each processor must be stored/written in the local disks

- **Storing cost = (S/P – Si/P) x (IO)**
- Why (S/P – Si/P)?



5.4. Cost Models for Parallel Join (cont'd)

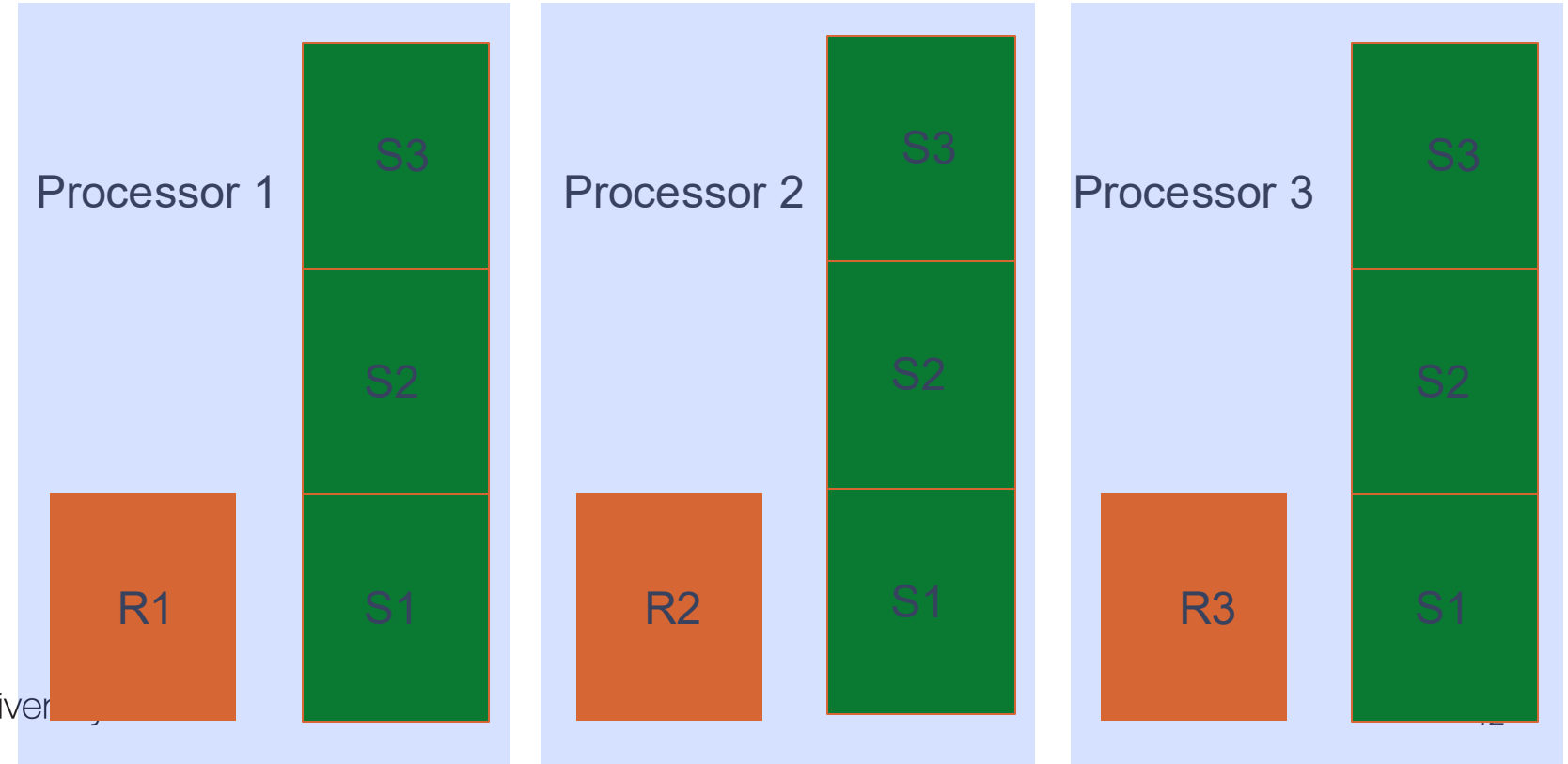
- **Cost Models for Divide and Broadcast**
  - Phase 3: Each processor after receiving all other fragments of table *S*, needs to be stored on local disk.
  - *Disk cost for storing the table* is: (*S/P – Si/P*) x *IO*

5.4. Cost Models for Parallel Join (cont'd)

- **Local Join**
  - Each processor performs a local join (using a Hash Join Algorithm)
  - Phase 1: Loading cost

    Scan cost = ((*Ri* / *P*) + (*S* / *P*)) x *IO*

    Select cost = (|*Ri*| + |*S*|) x (*tr* + *tw*)



5.4. Cost Models for Parallel Join (cont'd)

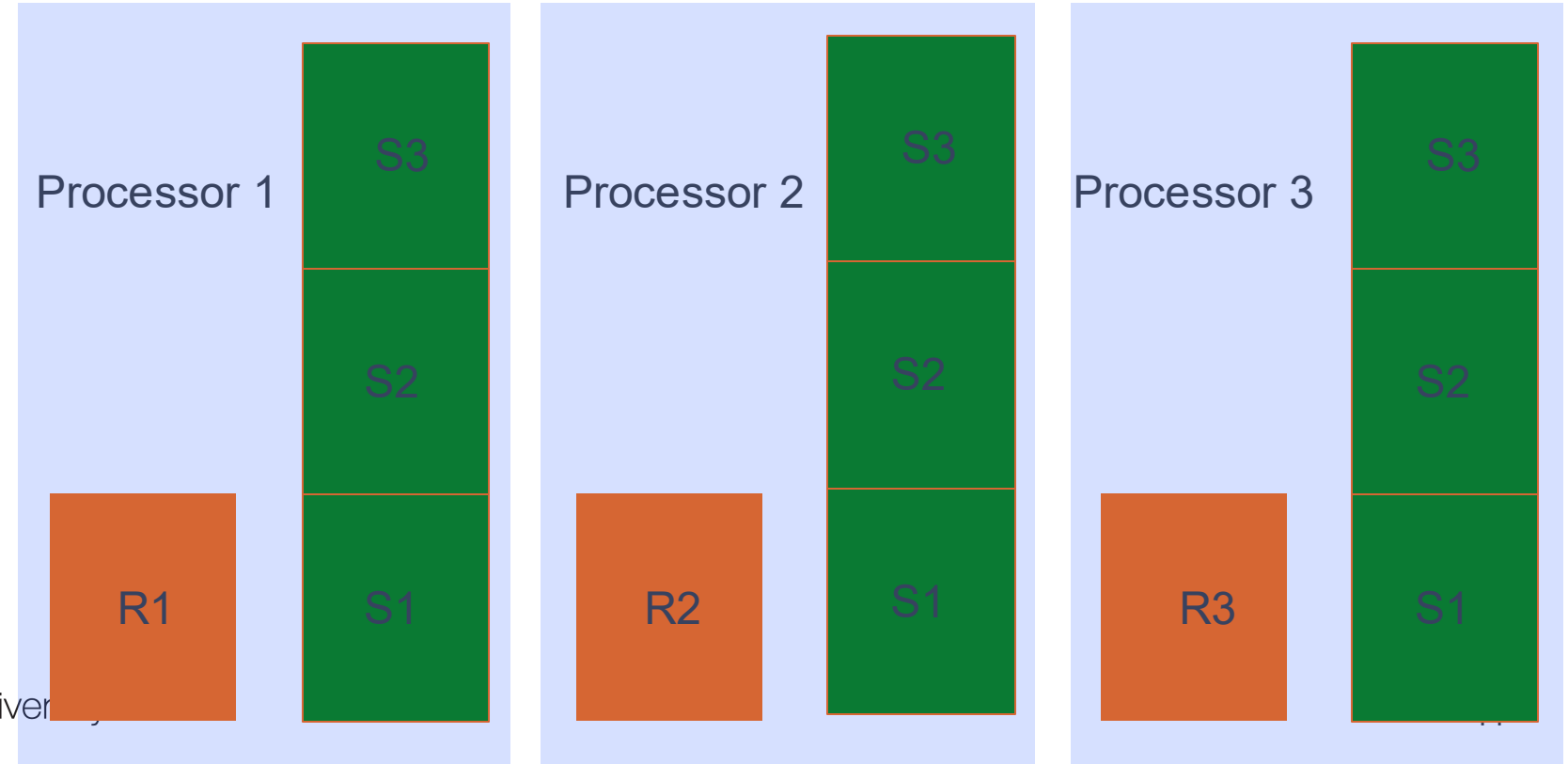
- **Cost Models for Local Join**
  - Assume to use hash-based join
  - Three main phases:
    - (1) data loading from each processor,
    - (2) the joining process (hashing and probing)
    - (3) result storing in each processor.
  - Phase 1: The data loading consists of scan costs and select costs
  - *Scan* cost = ((*Ri* / *P*) + (*S* / *P*)) x *IO*
  - *Select* cost = (|*Ri*| + |*S*|) x (*tr* + *tw*)

5.4. Cost Models for Parallel Join (cont'd)

- **Local Join**
  - Phase 2: Join cost (using a Hash Join Algorithm)

$$(|Ri| \times (tr + th) + (|S| \times (tr + th + tj))$$

Where *tr* is reading cost, *th* is hashing cost, and *tj* is joining cost



5.4. Cost Models for Parallel Join (cont'd)

- **Cost Models for Local Join**
  - Phase 2: The join process is the hashing and probing costs
  - *Join* costs involve reading, hashing, and probing: (|*Ri*| x (*tr* + *th*) + (|*S*| x (*tr* + *th* + *tj*))
  - If the memory size is smaller than the hash table size, we normally partition the hash table into multiple buckets whereby each bucket can perfectly fit into main memory. All but the first bucket is spooled to disk.
  - *Reading/Writing of overflow buckets* cost is the I/O cost associated with the limited ability of main memory to accommodate the entire hash table.

$$\left(1-\min\left(\frac{H}{|R_i|},1\right)\right)\times\left(\frac{R_i}{P}\times 2\times IO\right)$$

5.4. Cost Models for Parallel Join (cont'd)

- **Cost Models for Local Join**
  - *Reading/Writing of overflow buckets* cost is the I/O cost associated with the limited ability of main memory to accommodate the entire hash table.

$$\left(1-\min\left(\frac{H}{|R_i|},1\right)\right)\times\left(\frac{R_i}{P}\times 2\times IO\right)$$

  - For example, the Hash table can only occupy 10 records at a time from table Ri. Assume |Ri|=50 records. That means that there will be 5 buckets. Because the main memory can take one bucket only, it means the 4 buckets must be stored on disk.
  - (1-min(H/|Ri|,1)) = 1-min(0.2,1) = 1-0.2 = 0.8
  - If |Ri|=10 or less, then (1-min(H/|Ri|,1)) = 1-1 = 0; That means there is no overflow bucket cost.
  - (Ri/P x 2 x IO) ➔ the constant 2 means two input/output accesses: one for spooling, and the other for reading it back from the disk

5.4. Cost Models for Parallel Join (cont'd)

- **Cost Models for Local Join**
  - Phase 3: query results storing cost, consisting of generating result cost and disk cost.
  - *Generating result records* cost is: $|Ri| \times \sigma j \times |S| \times tw$
  - *Disk* cost for storing the final result is: $(\pi_R \times Ri \times \sigma j \times \pi_S \times S / P) \times IO$

Query Parameters (Homework)

- **Projectivity ratio π :**
  - Ratio between projected attribute size and original record length
- **Selectivity ratio σ :**
  - Ratio between number of records in the query result and original total number of records

Example: Join selectivity ratio

If the query operation involves two tables (like in a join operation), a selectivity ratio can be written as σ_j, for example. The value of σ_j indicates the ratio between the number of records produced by a join operation and the number of records of the Cartesian product of the two tables to be joined. For example, $|R_i| = 1000$ records and $|S_i| = 500$ records; if the join produces 5 records only, then the join selectivity ratio σ_i is $5/(1{,}000 \times 500) = 0.00001$.

5.4. Cost Models for Parallel Join (cont'd)

- **Home Work**

  - Disjoint data partitioning based parallel join algorithm
  - Cost Model for Disjoint data partitioning based parallel join algorithm

https://onlinelibrary-wiley-com.ezproxy.lib.monash.edu.au/doi/pdf/10.1002/9780470391365.ch5

5.5. Parallel Join Optimization

- The aim of query processing in general is to speed up the query processing time
- In terms of parallelism, the reduction in the query elapsed time is achieved by having each processor finish its execution as early as possible and as evenly as possible → **load balancing issue**
- In the disjoint partitioning, after the data is distributed to the designated processors, the data has to be stored on disk. Then in the local join, the data has to be loaded from the disk again → **managing main memory issue**

5.5. Parallel Join Optimization (cont'd)

- **Optimizing Main Memory**
  - Disk access is the most expensive operations, so need to reduce disk access as much as possible
  - If it is possible, only a single scan of data should be done. If not, then minimize the number of scan
  - If main memory size is unlimited, single disk scan is possible
  - However, main memory size is not unlimited, hence optimizing main memory is critical
  - Problem: In the distribution, when the data arrives at a processor, it is stored in disk. In the local join, the data needs to be reloaded from disk
  - This is inefficient. When the data arrives after being distributed from other processor, the data should be left in main memory, so that the data remain available in the local join process
  - The data left in the main memory can be as big as the allocated size for data in the main memory

5.5. Parallel Join Optimization (cont'd)

- **Optimizing Main Memory**
  - Assuming that the size of main memory for data is *M* (in bytes), the disk cost for storing data distribution with a disjoint partitioning is:

    $((Ri / P) + (Si / P) - (M/P)) \times IO$

    Data remains in memory, thus reducing data to be stored in disk
  - And the local join scan cost is then reduced by *M* as well:

    $((Ri / P) + (Si / P) - (M/P)) \times IO$

  - When the data from this main memory block is processed, it can be swapped with a new block. Therefore, the saving is really achieved by not having to load/scan the disk for one main memory block

5.5. Parallel Join Optimization (cont'd)

- **Load Balancing**
  - Load imbalance is the main problem in parallel query processing. It is normally caused by data skew and then processing skew
  - No load imbalance in divide and broadcast-based parallel join. But this kind of parallel join is unattractive, due to the heavy broadcasting
  - In disjoint-based parallel join algorithms, processing skew is common
  - To solve this skew problem, create more fragments than the available processors, and then rearrange the placement of the fragments

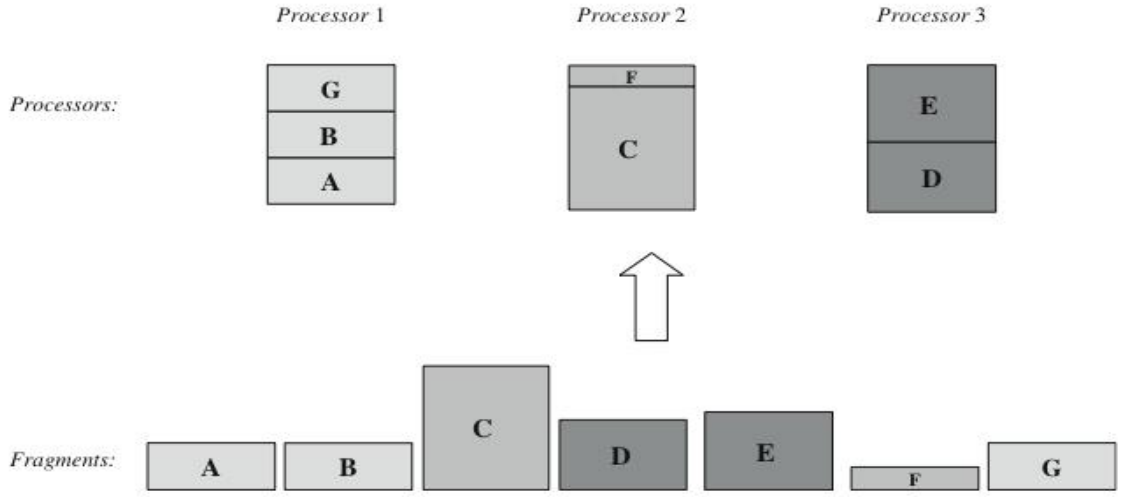


Figure 5.19 Load balancing

5.6. Summary

- Parallel join is one of the most important operations in parallel database systems
- Parallel join algorithms have two stages
  - Data partitioning
  - Local join
- Two types of data partitioning
  - Divide and broadcast
  - Disjoint partitioning
- Three types of local join
  - Nested-loop join
  - Sort-merge join
  - Hash-based join