MONASH University

Information Technology

FIT5202 (Version III - Join)

Week 3a – Parallel Join

Chapter 3
Parallel Search

Revision

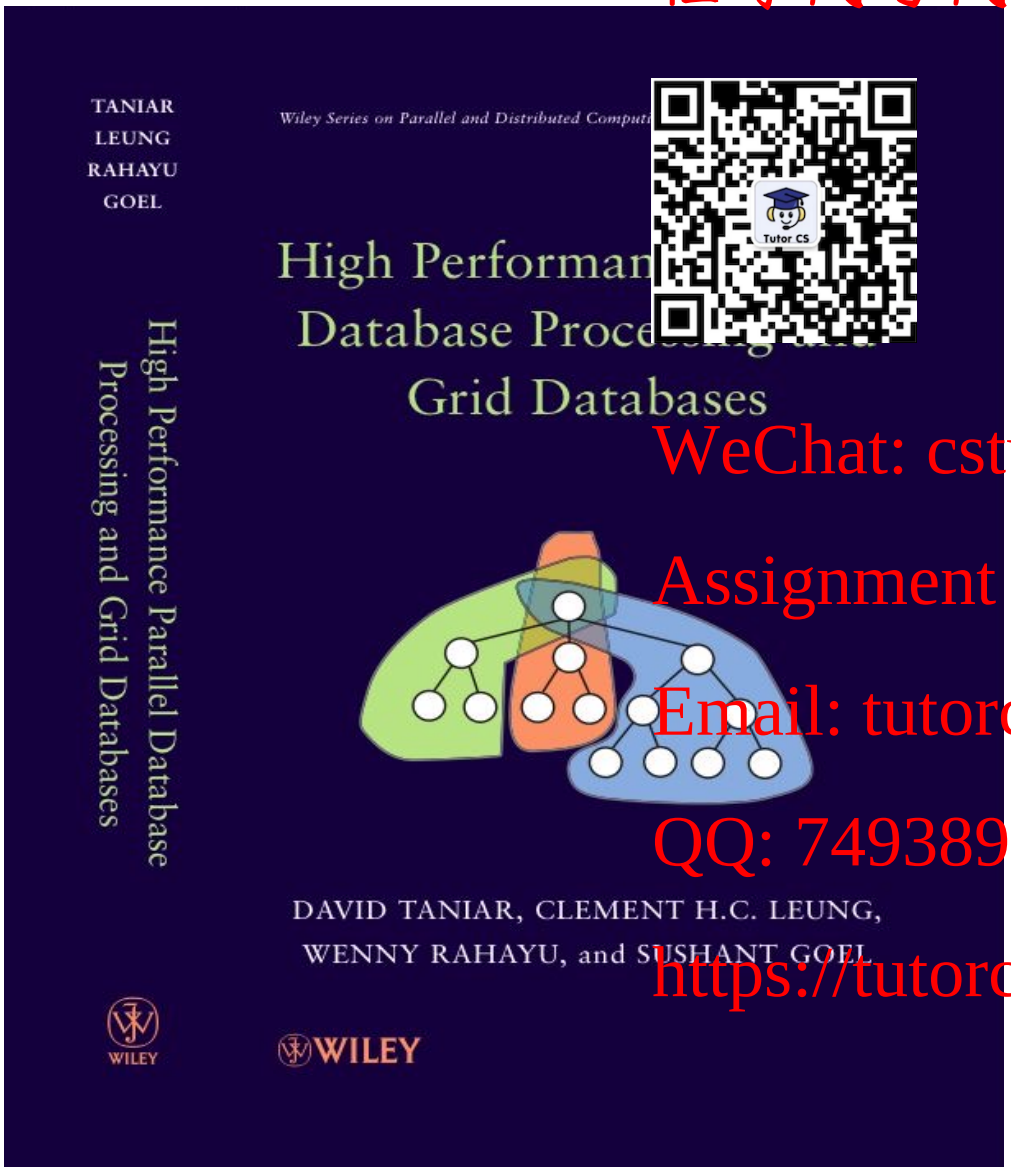
- **Exercise 1 (FLUX Quiz**

  - If a query runs on a multi e(e.g. Windows server with 16 cores), it is called parallel query processing f multiple different queries run at the same time in a multi-core machine?
  - A. Intra-query parallelism
  - B. Inter-query parallelism
  - C. Intra-operation parallelism
  - D. Inter-operation parallelism

Revision

- **Exercise 2 (FLUX Quiz**

  - If a **hash data partitioni** store the data, and the query is a **discrete range search**, how many proce be used to process such a query efficiently in order to get the query results?
  - A. 1 processor
  - B. Selected processors
  - C. All processors



Chapter 5
Parallel Join

5.1. Join Operations

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


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

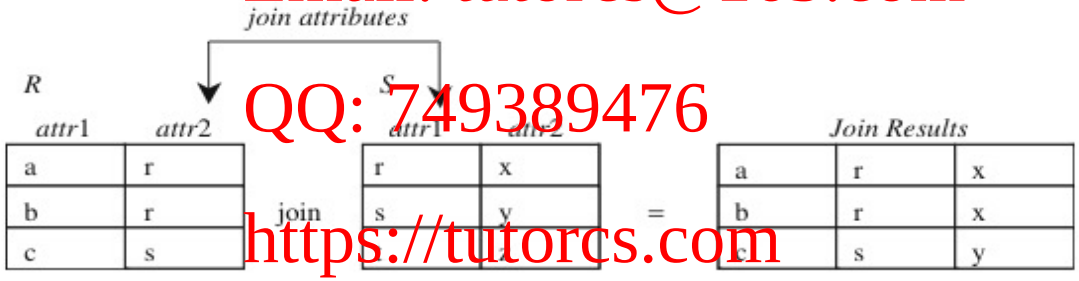


Figure 5.1 The join operation

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	23
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

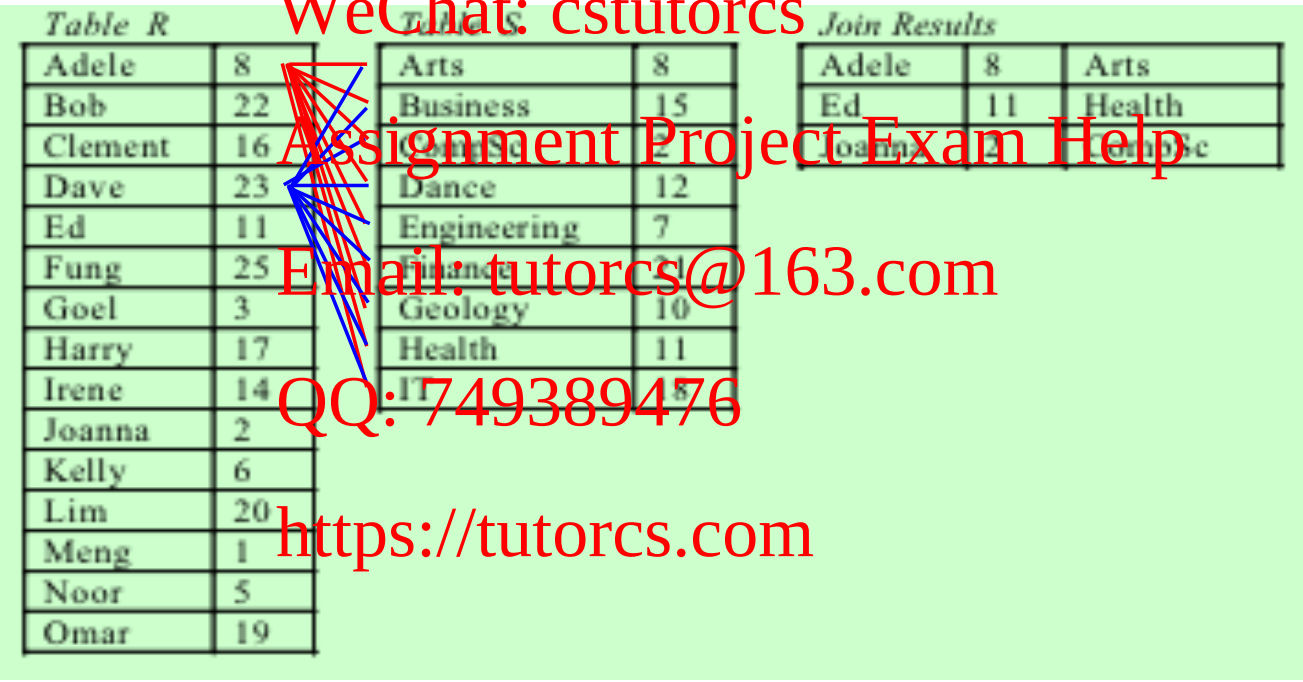
Join Results

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*



5.2. Serial Join Algorithms (cont'd)

- **Sort-Merge Join**
  - 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

	2
ng	7
Arts	8
Geology	10
Health	11
Dance	12
Business	15
[illegible]	[illegible]
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

	2
ng	7
Arts	8
Geology	10
Health	11
Dance	12
Business	15
[illegible]	[illegible]
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

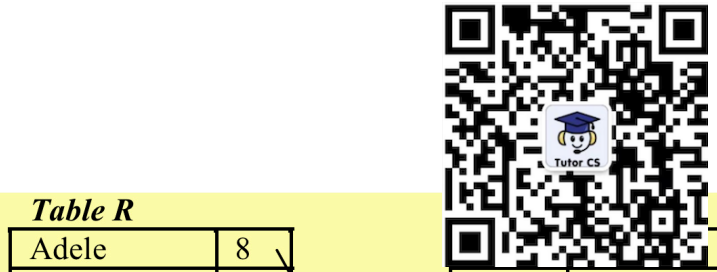
hashed into

Hash Table

x	*Entries*	
	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)



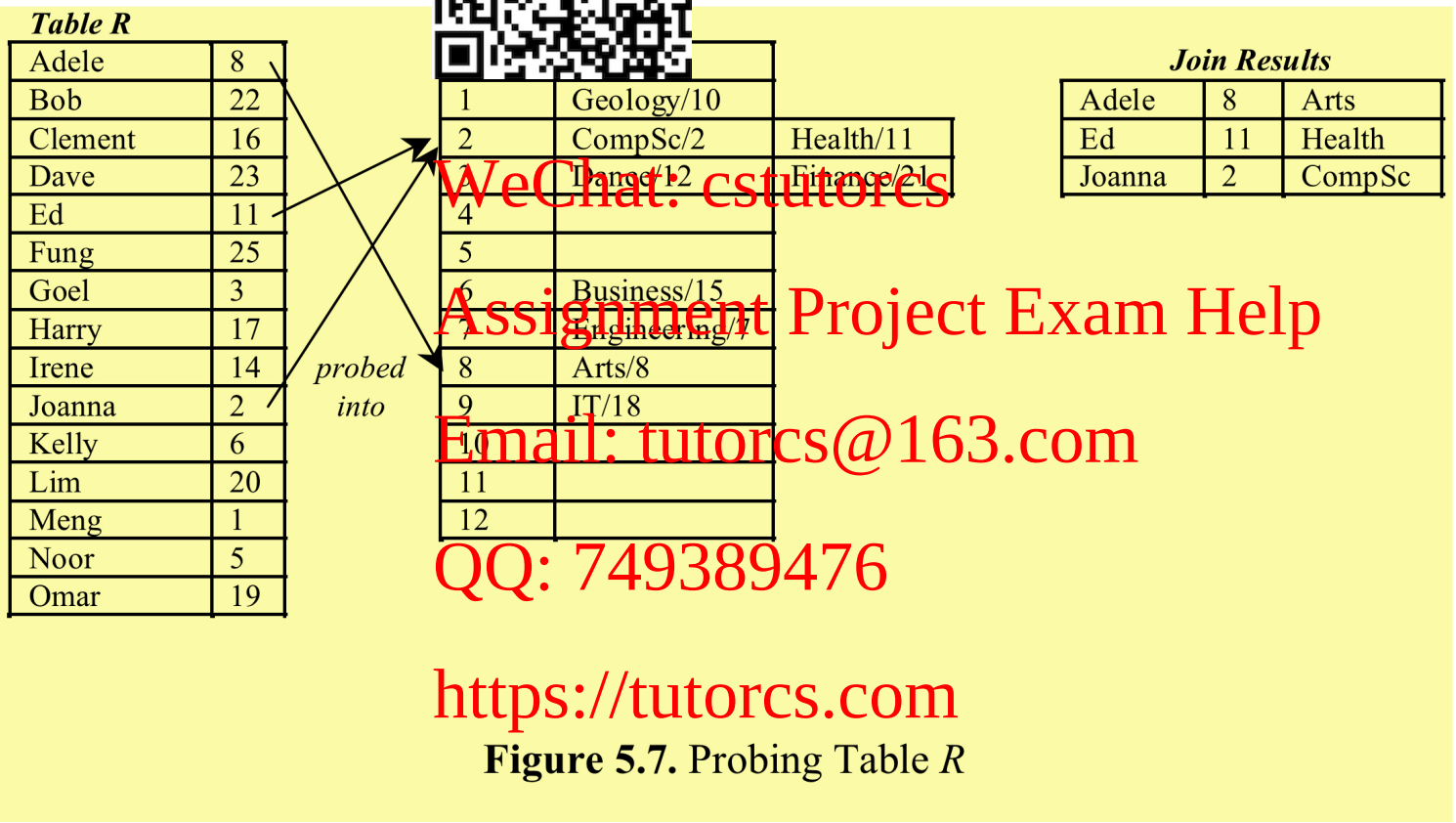
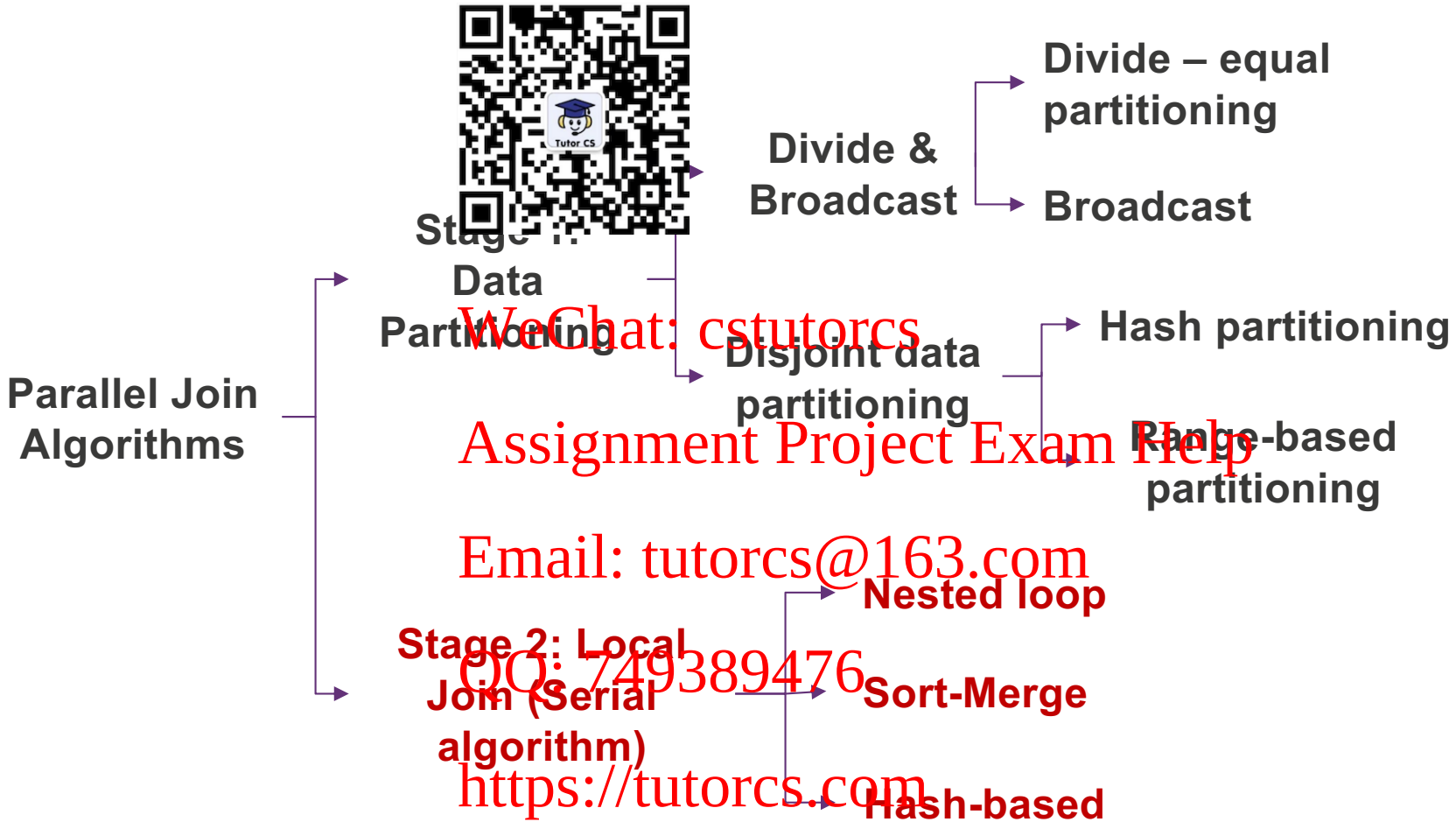


Figure 5.7. Probing Table *R*

5.3. Parallel Join Algorithms

- Parallelism of join algorithms 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

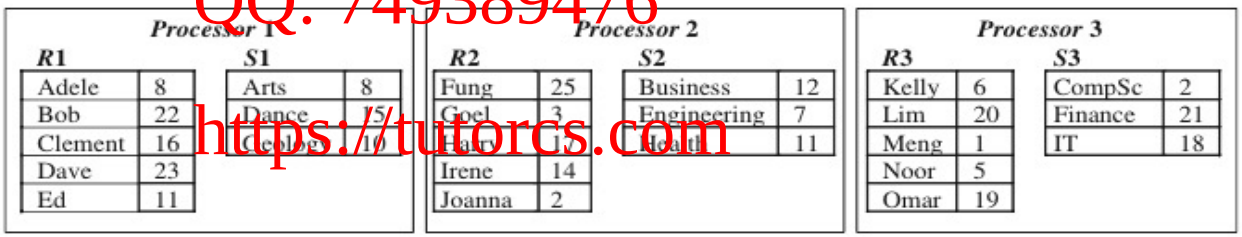
Overview



5.3. Parallel Join Algorithms (cont'd)

- **Divide and Broadcast-based Parallel Join Algorithms**
  - Two stages: data partitioning using the divide and broadcast method, and a local join
  - Divide and Broadcast method: Divide one table into multiple disjoint partitions, where each partition is allocated a processor, and broadcast the other table to all available processors
  - Dividing one table can simply use equal division
  - 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)



Processor 1			
R1		S1	
Adele	8	Arts	8
Bob	22	Dance	15
Clement	16	Geology	10
Dave	23		
Ed	11		

Processor 2			
R2		S2	
Fung	25	Business	12
Goel	3	Engineering	7
Faury	17	Health	11
Irene	14		
Joanna	2		

Processor 3			
R3		S3	
Kelly	6	CompSc	2
Lim	20	Finance	21
Meng	1	IT	18
Noor	5		
Omar	19		

Figure 5.10 Initial data placement

5.3. Parallel Join Algorithms (cont'd)

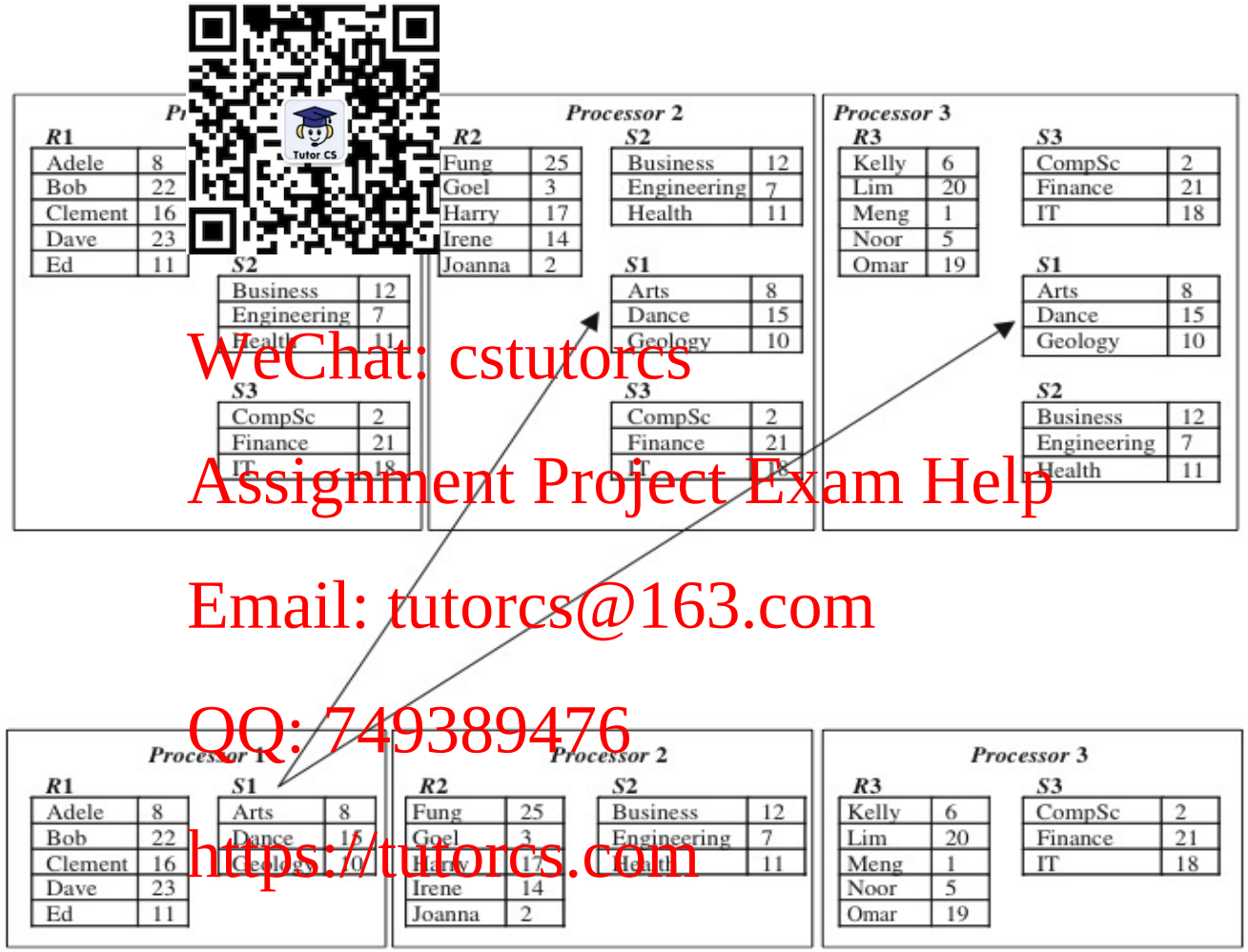


Figure 5.11 Divide and broadcast result

5.3. Parallel Join Algorithms (cont'd)

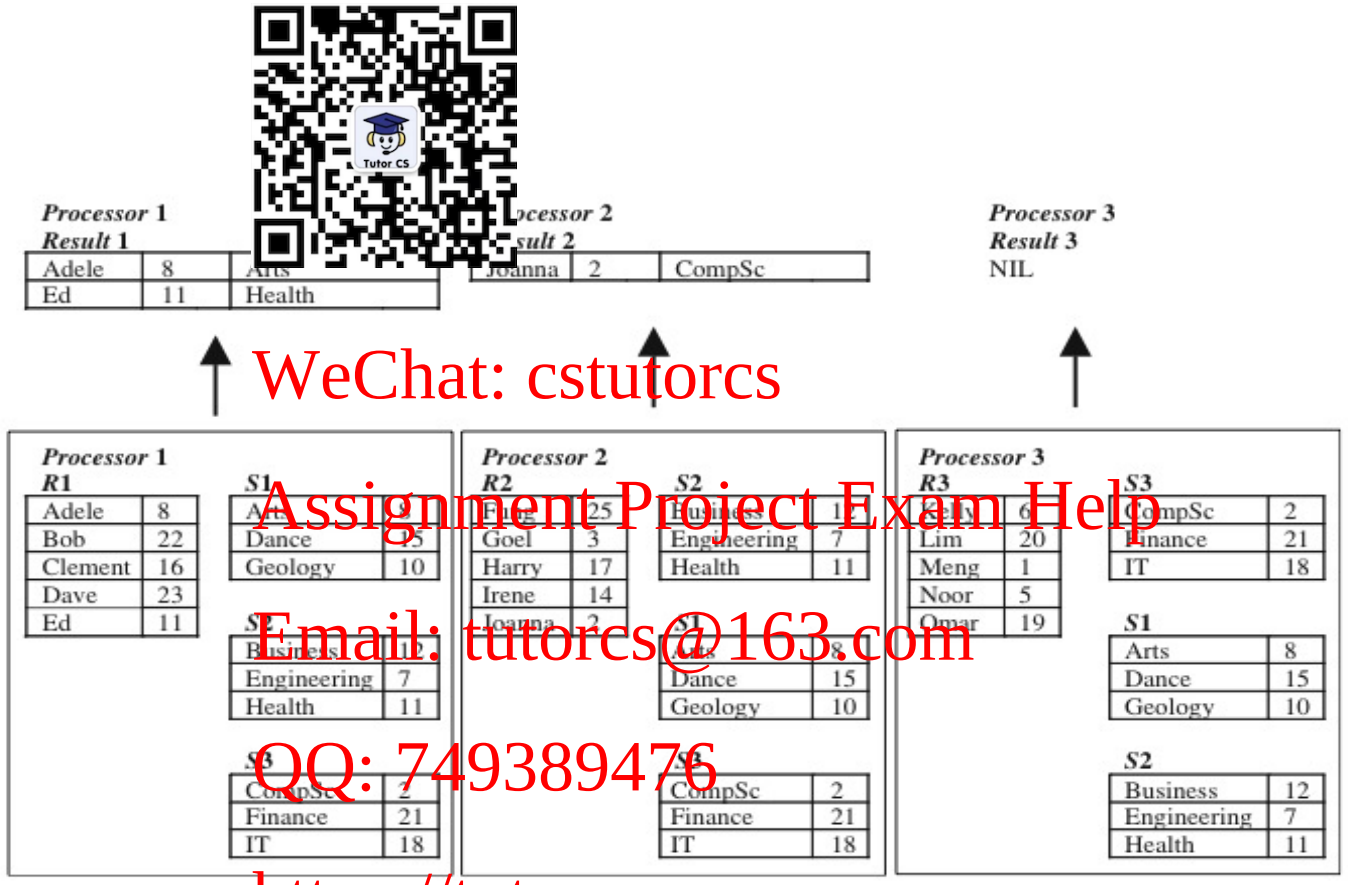


Figure 5.12 Join results based on divide and broadcast

5.3. Parallel Join Algorithms (cont'd)

- **Divide and Broadcast-based Parallel Join Algorithms**
  - 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
  - If shared-memory is used, then there is no replication of the broadcast table. Each processor will access the entire table *S* and a portion of table *R*. But if each processor does not have enough working space, then the local join might not be able to use a hash-based join

5.3. Parallel Join Algorithms (cont'd)

- **Disjoint Partitioning-based Parallel Join Algorithms**
  - Two stages: data partitioning using a disjoint partitioning, and local join
  - Disjoint partitioning: range or hash partitioning
  - Local join: any serial local join algorithm

5.3. Parallel Join Algorithms (cont'd)

- Example 1: Rang ning

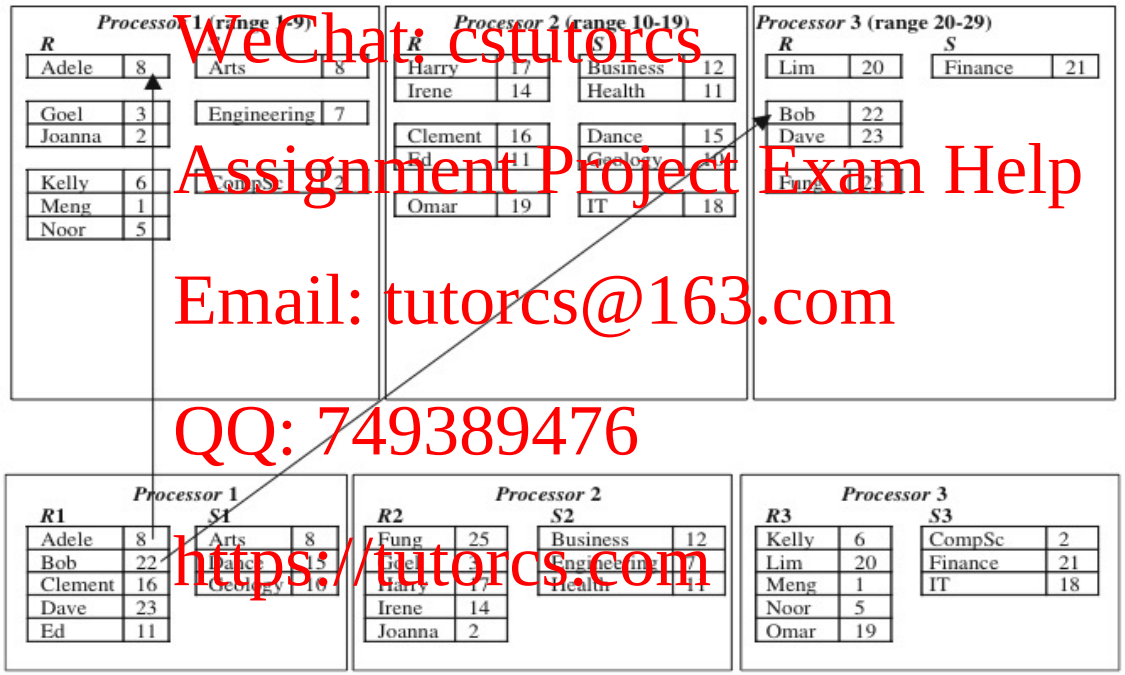


Figure 5.14 Range partitioning

5.3. Parallel Join Algorithms (cont'd)

- Example 1: Rang[illegible]ning

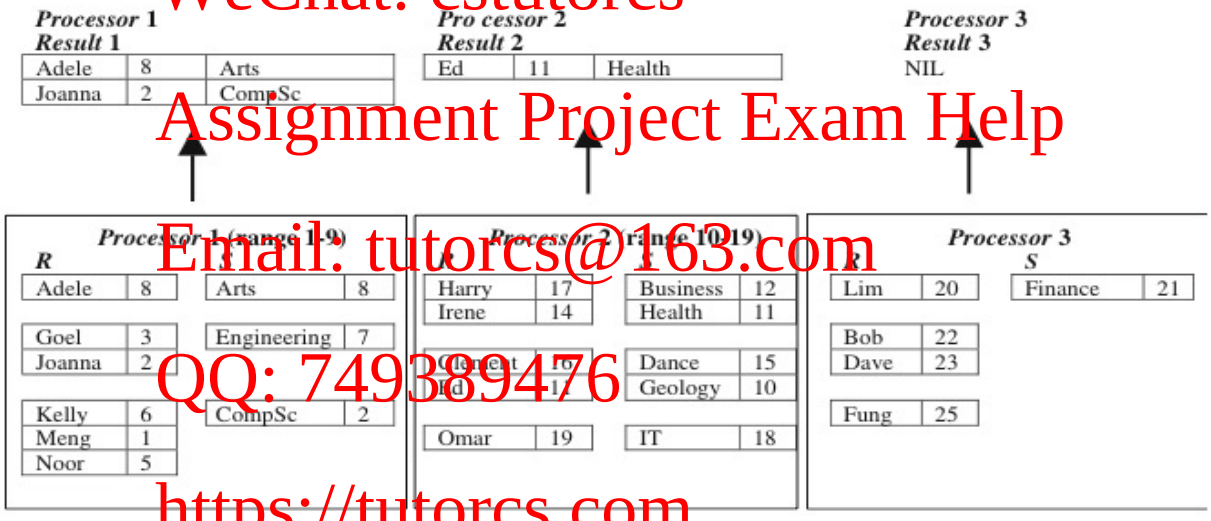


Figure 5.15 Join results based on range partitioning

5.3. Parallel Join Algorithms (cont'd)

- Example 2: Hash partitioning

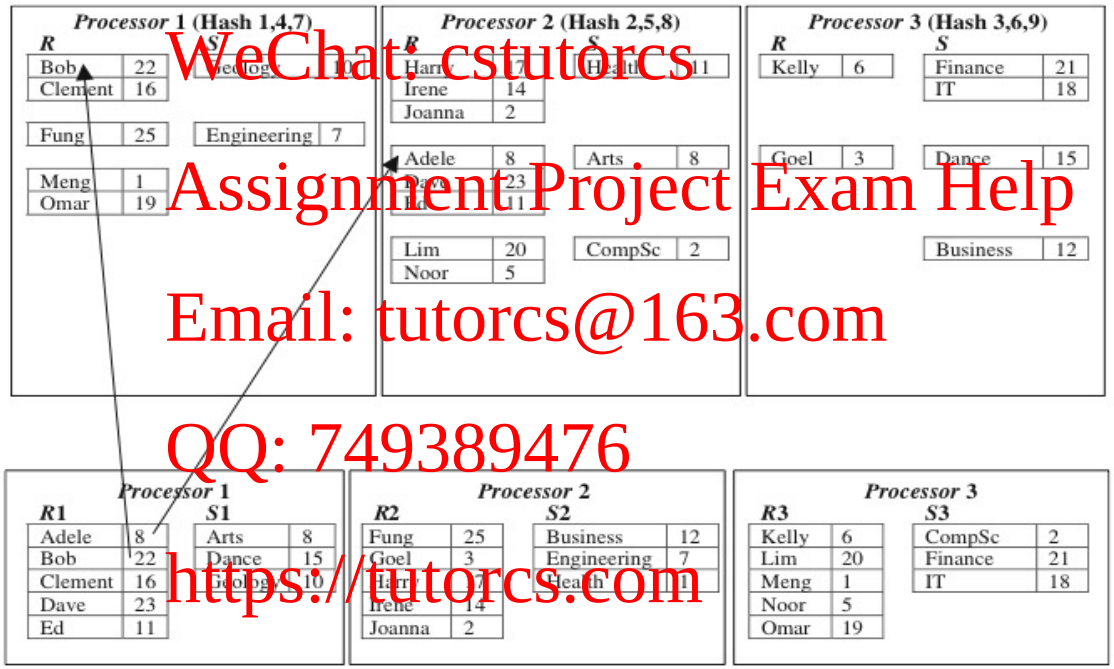


Figure 5.16 Hash partitioning

5.3. Parallel Join Algorithms (cont'd)

- Example 2: Hash Partitioning

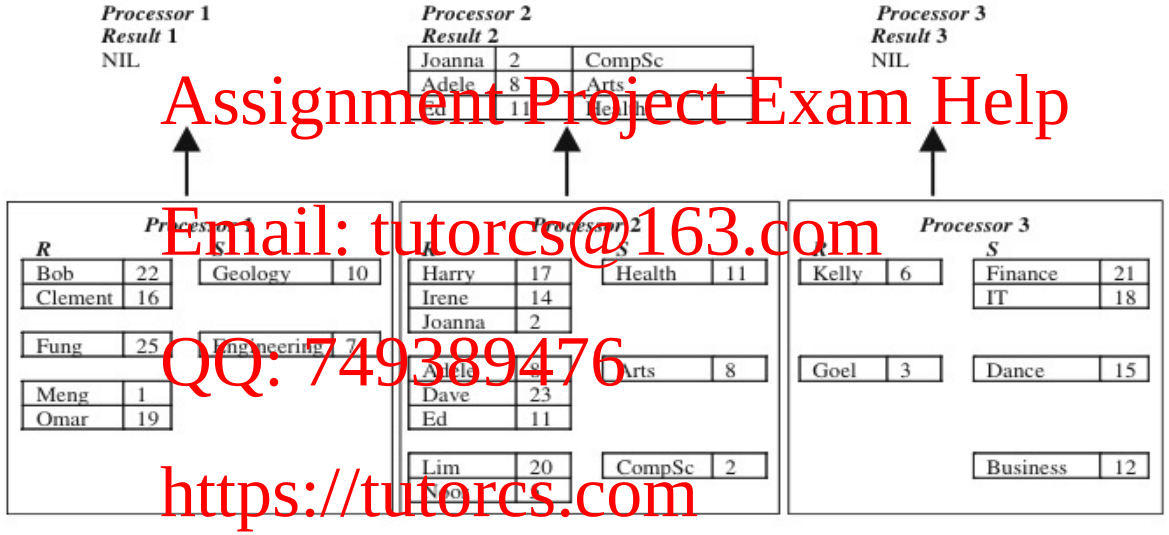


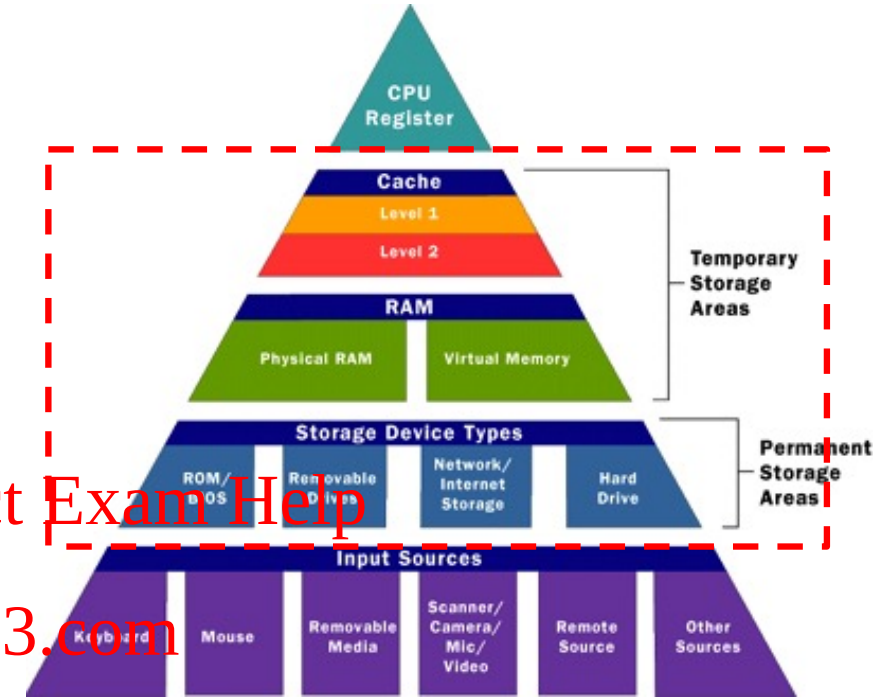
Figure 5.17 Join results based on hash partitioning

5.4. Cost Models for Parallel Join

- **Divide and Broadcas**
  - 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 **R**i or **S**i, where i is the processor number)

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

- **Divide and Broadcast**
  - **Phase 1: Data Loa**me the table to be broadca S.
  - Read all records fr n each processor
  - If there are 30,000 records in table S, how long does it take to complete the reading before the process can continue to the next phase?
  - Is it after reading 30,000 records, or after reading 10,000 records?



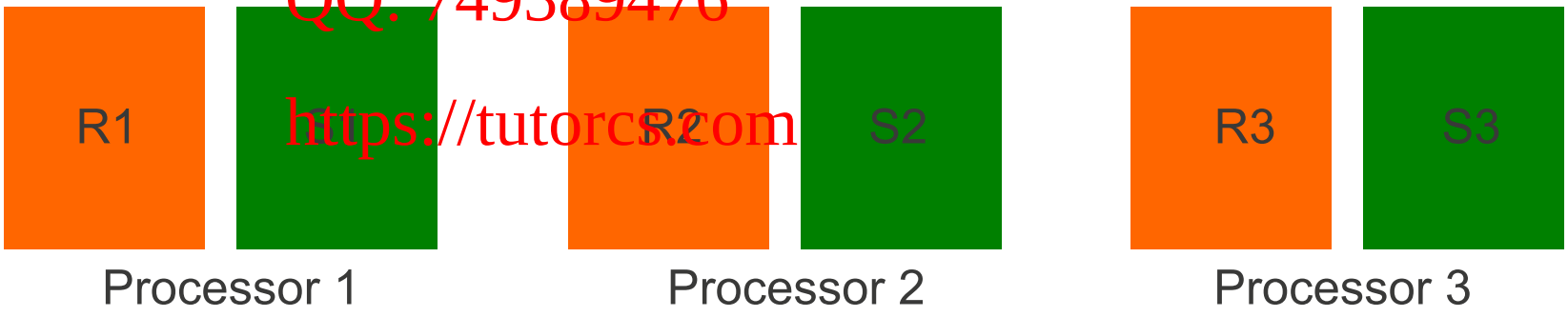
5.4. Cost Models for Parallel Join (cont'd)

- **Divide and Broadcas**
  - **|S|** = 30,000 rec
  - **N** = 3 processor
  - **|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)



- **Exercise 3 (FLUX Quiz**

  - |S| = 600 records, each r[illegible] length of 100 bytes, and N=3.

  - A. Si = 200 records
  - B. Si = 20,000 bytes
  - C. |Si| = 20,000 bytes
  - D. |Si| = 200 records
  - E. A and C
  - F. B and D
  - G. All of the above

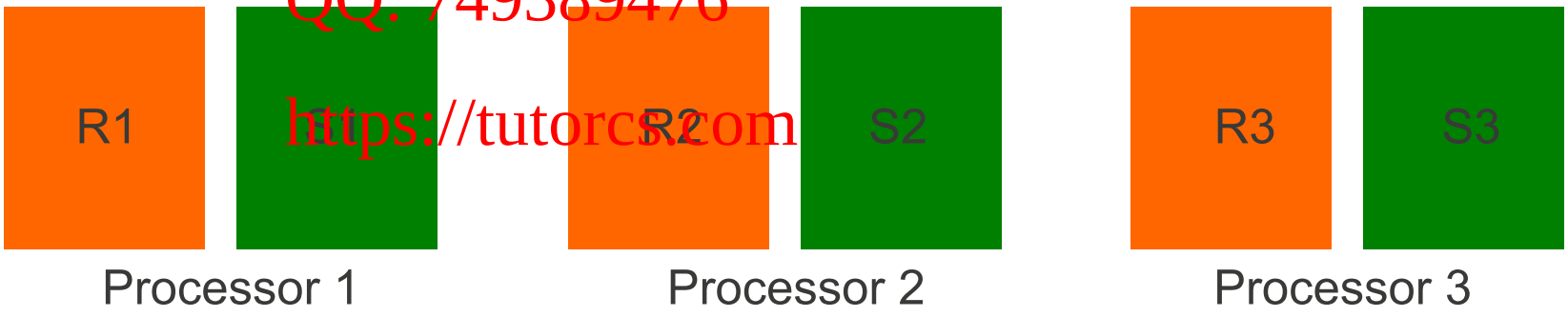
5.4. Cost Models for Parallel Join (cont'd)

- **Divide and Broadcas**
  - When a table fra ead from disk, it is based on the size of the table (in bytes), sed 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)

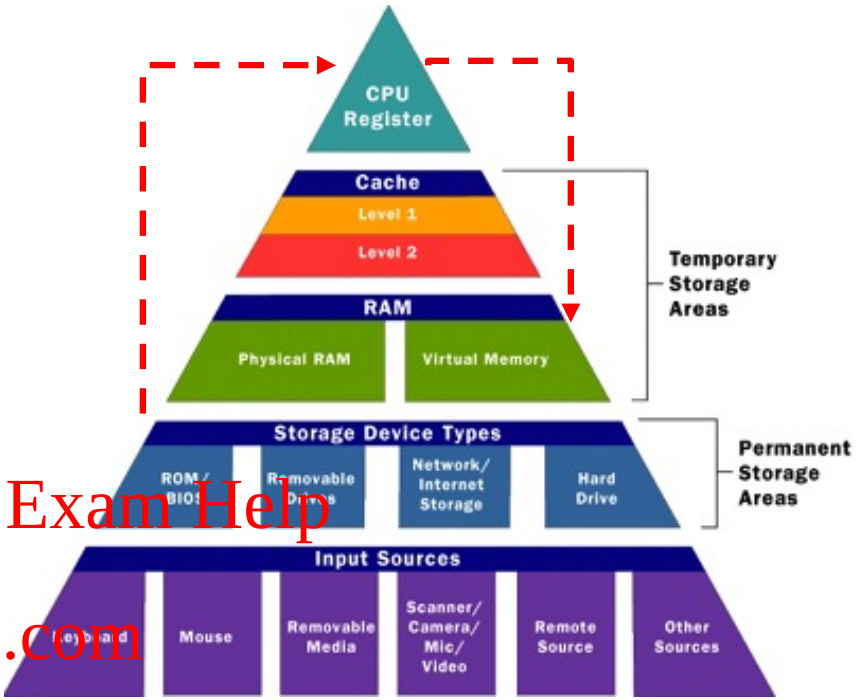
  - The loading time or **Scan cost = (Si/P) x IO**, where IO is the time taken to load 1 page from disk to main memory

Initial data placement (random-equal partitioning)



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

- **Divide and Broadcast**
  - Once the loading is [illegible] the records are not yet ready in [illegible] memory. The records must be rea[illegible] 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 [illegible] 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:

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

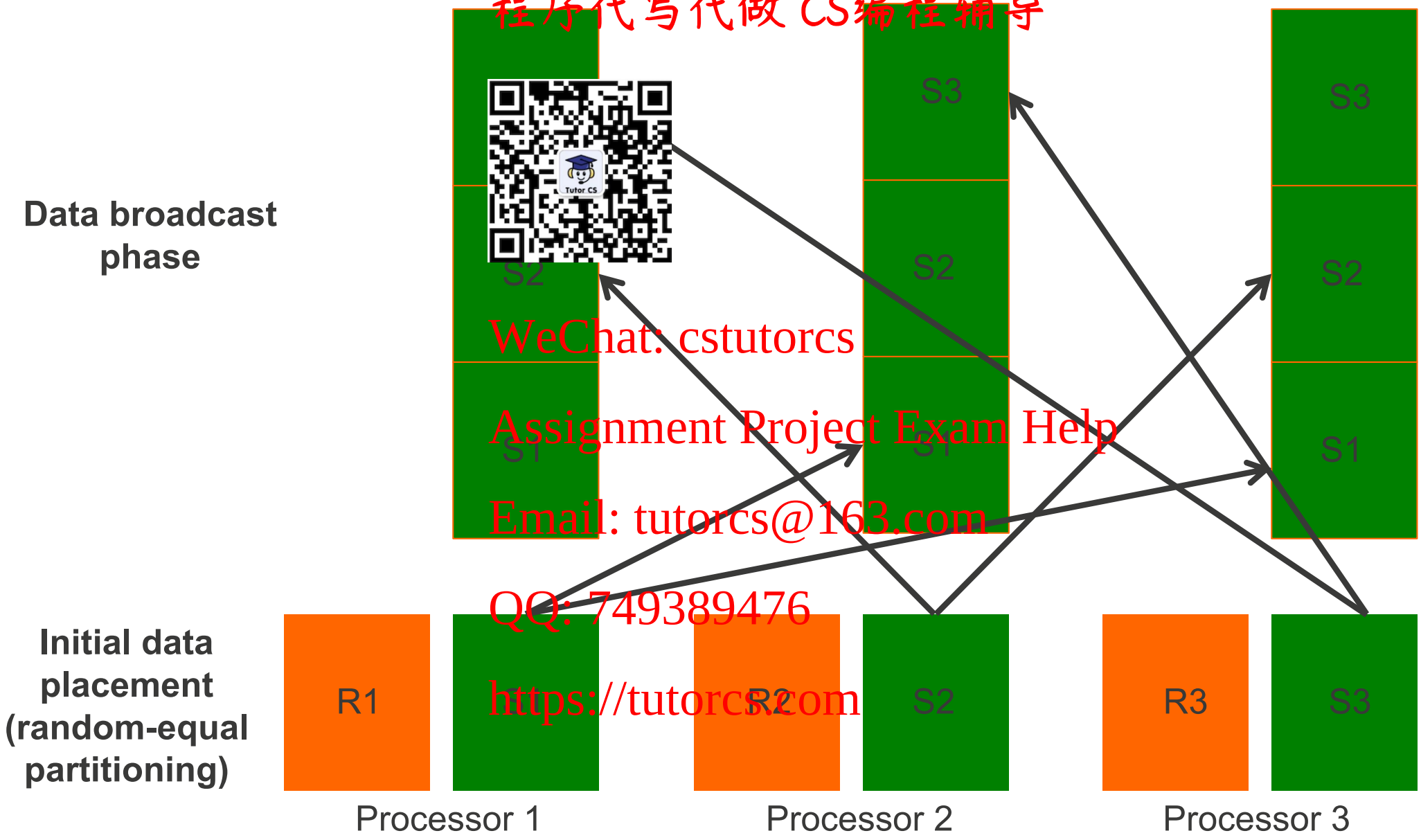
  - *Select cost* for getting record out of data page is:

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

- **Exercise 4 (FLUX Quiz**

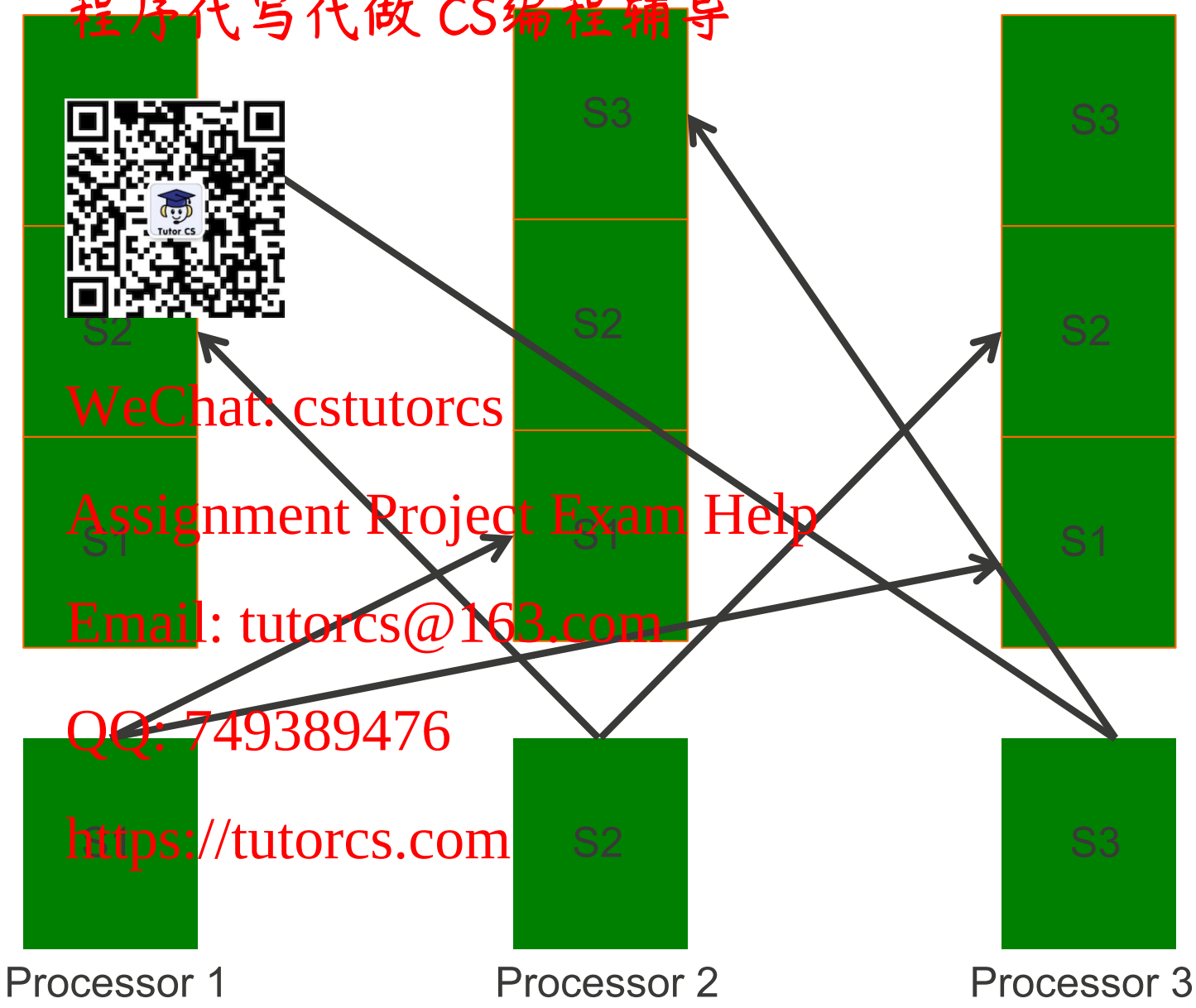
  - The cost (the time taken) from disk is called…
  - A. Scan Cost
  - B. Select Cost
  - C. Both A and B

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…



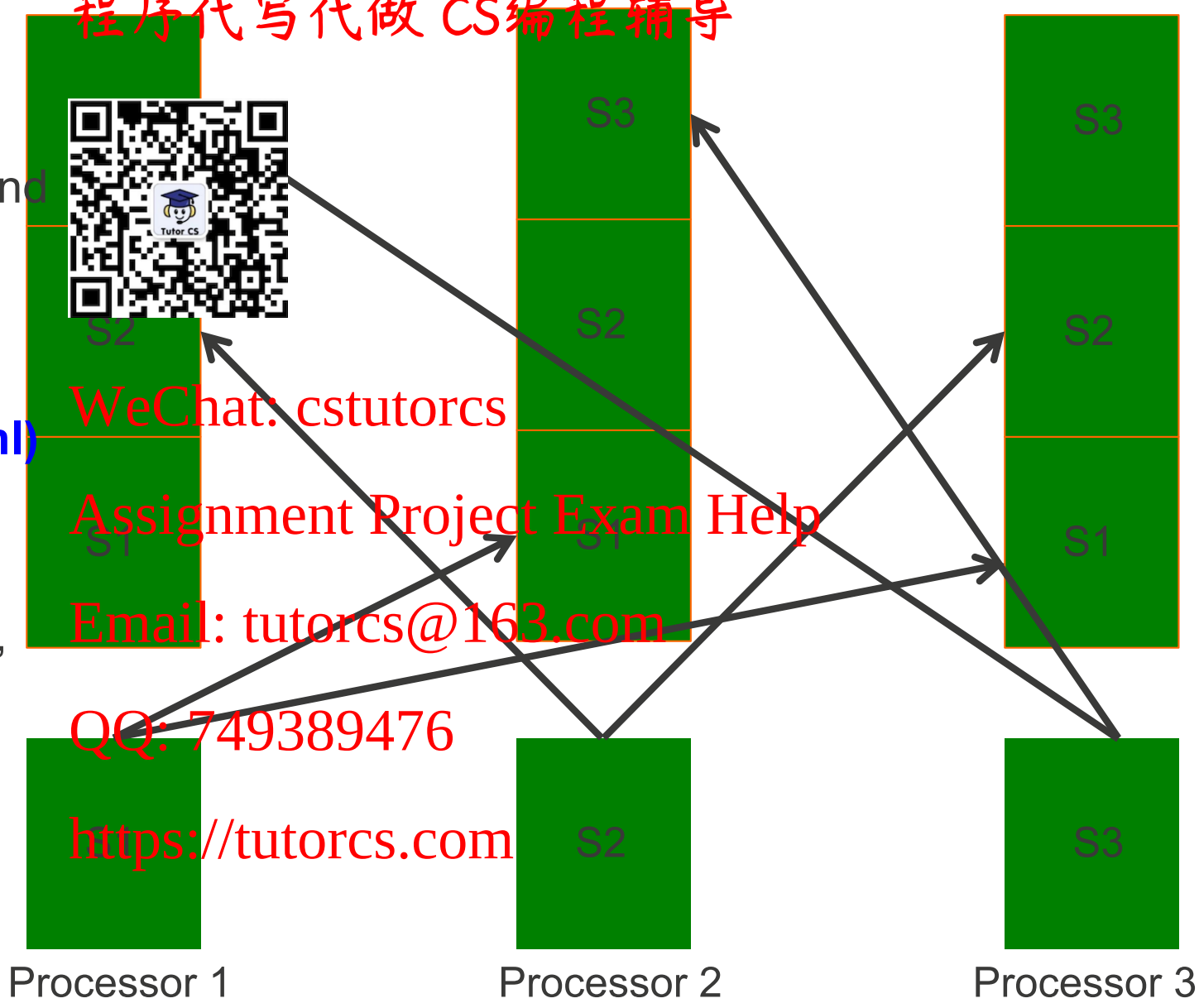
Phase 2: Data Broadcasting. Table fragment S1 must be broadcasted (copied) to processors 2 and 3. Table fragment S2 to processors 1 and 3, etc…

- 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**



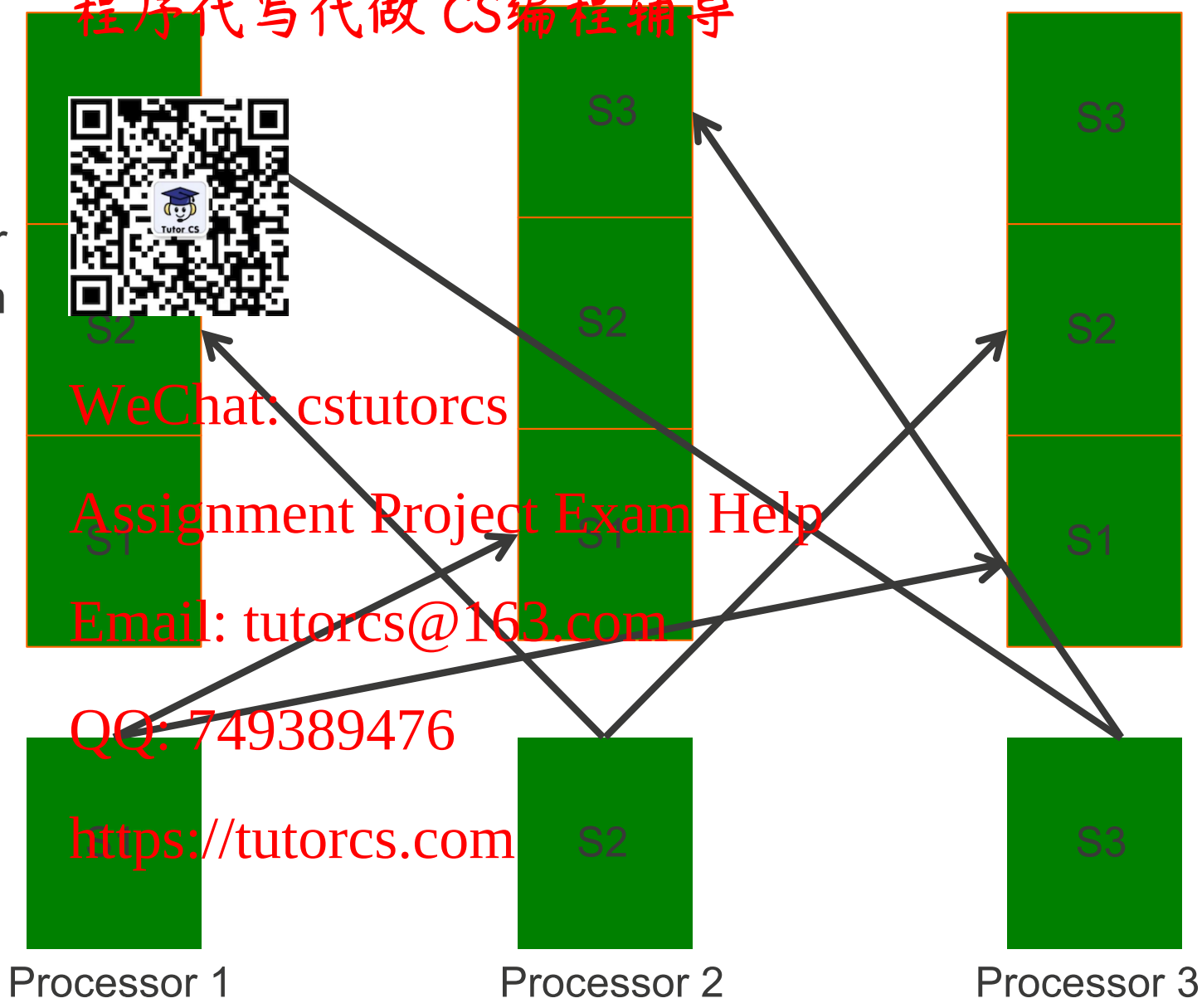
Phase 2: Data Broadcasting. Table fragment S1 must be broadcasted (copied) to processors 2 and 3. Table fragment S2 to processors 1 and 3, etc…

- S1 must be transferred twice (to processor 2 and to processor 3)
- **Transfer cost = (Si/P) x (N-1) x (mp+ml)**
- Where **mp** is message protocol per data page, and **ml** is message latency per data page.



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?



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

- **Cost Models for Hash-Based Broadcast**
  - Phase 2: The broadcast cost by each processor broadcasting its fragment to all other processors

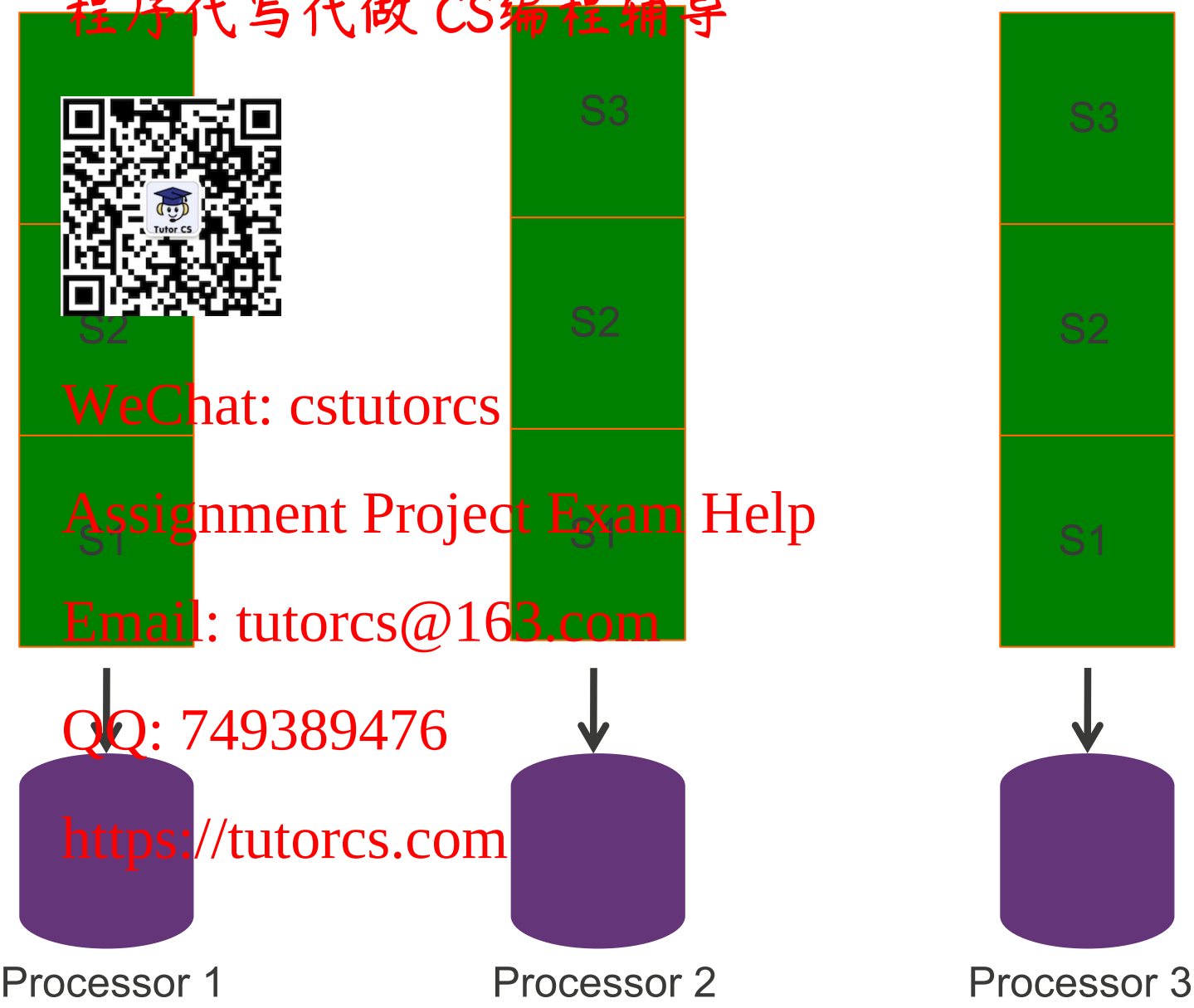
  - *Data transfer cost* is: (*Si / P*) x (*N* – 1) x (*mp* + *ml*)

  - 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)?

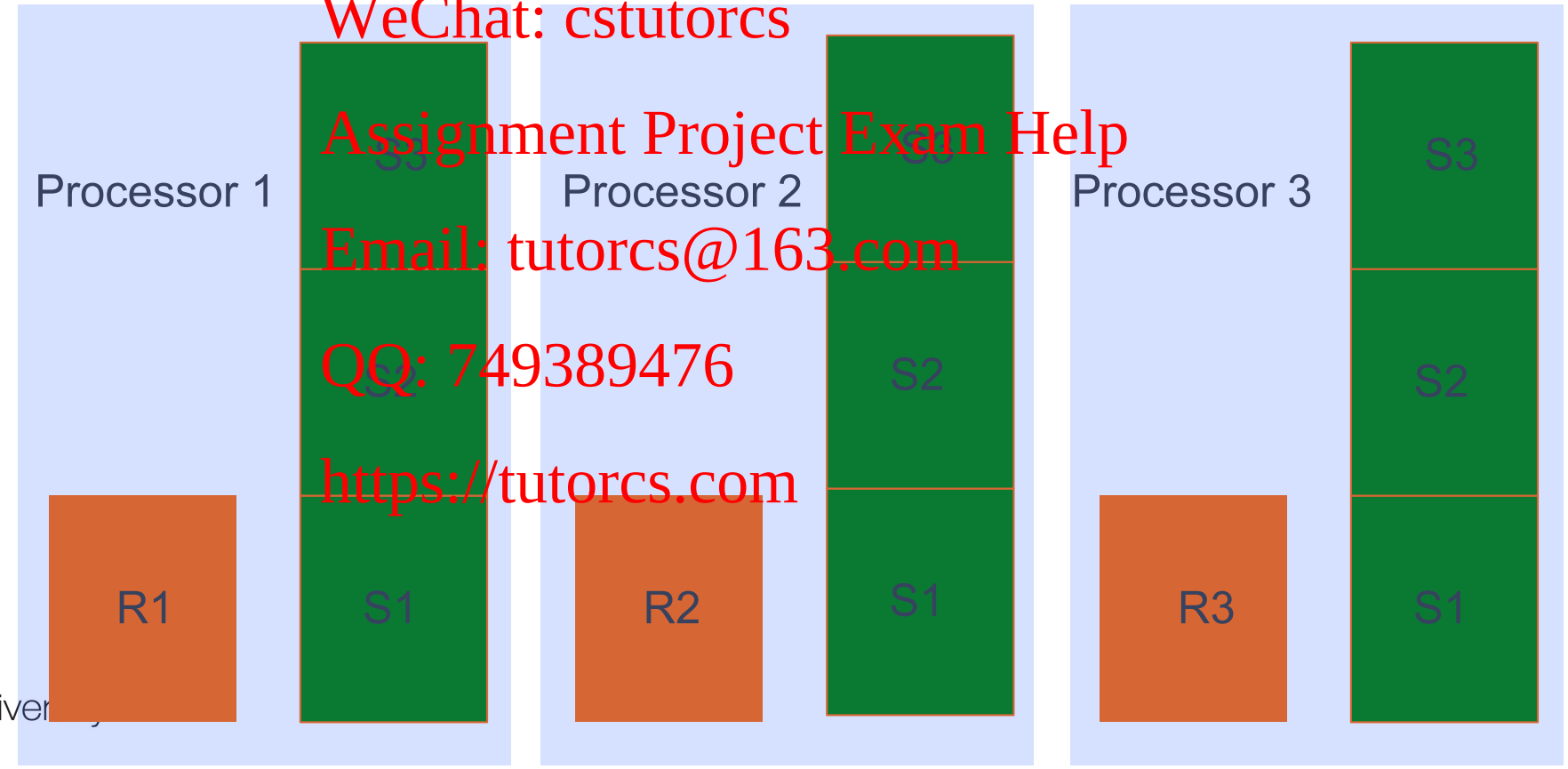


5.4. Cost Models for Parallel Join (cont'd)

- **Cost Models for [illegible]d 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
    - *Scan* cost = ((*R* / *P*) + (*S* / *P*)) x *IO*
    - *Select* cost = (|*Ri*| + |*S*|) x (*tr* + *tw*)



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

- **Cost Models for Join**
  - Assume to use hash-based join
  - Three main phases: data loading from each processor, the joining process (hashing and probing), and 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*)

Query Parameters

- **Projectivity ratio π :**
  - Ratio between pro[illegible]ute 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

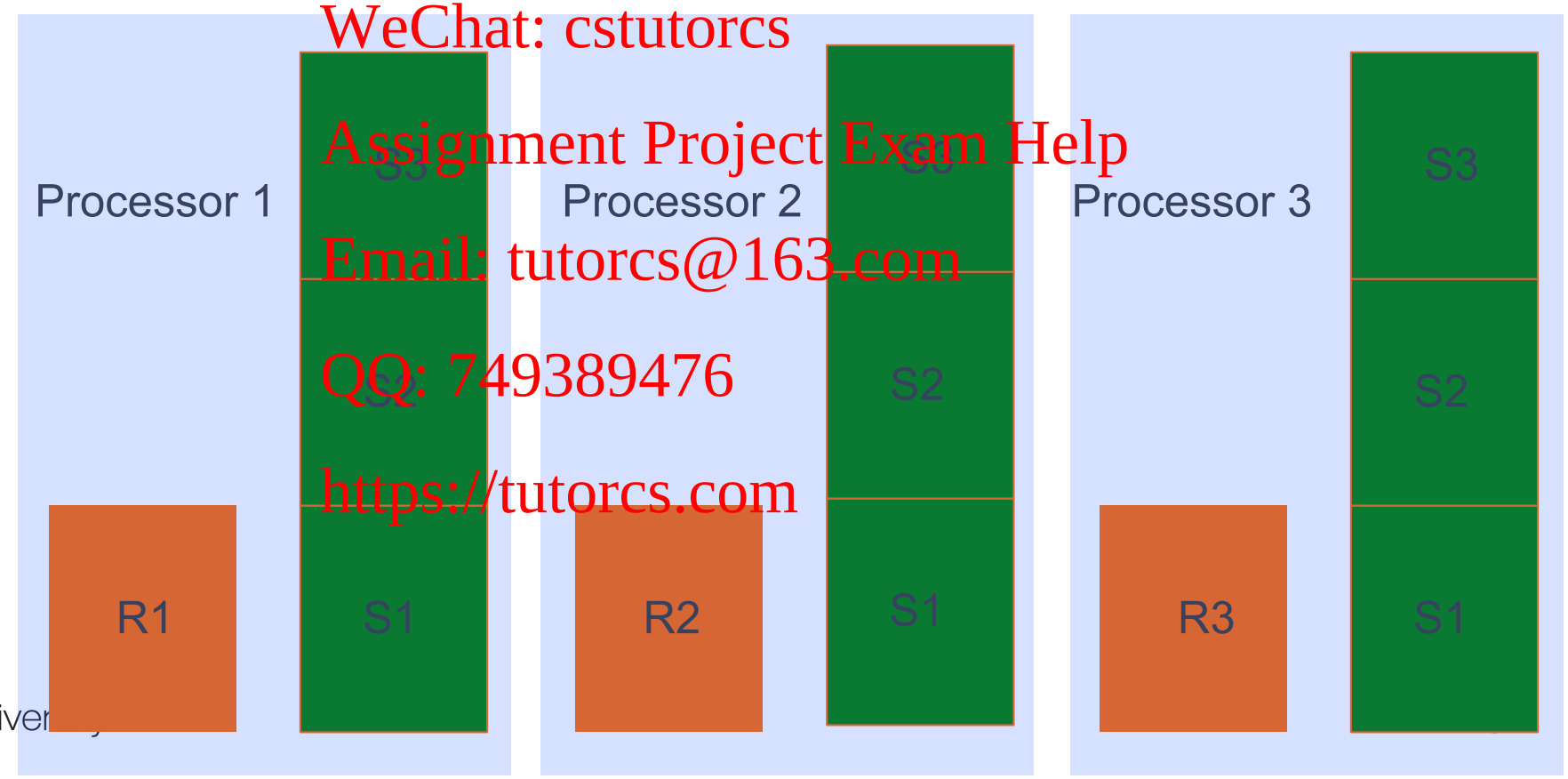
If the query operation involves two tables (like a join operation), a selectivity ratio can be written as σ_j, for example. The value of σ_j indicates the ratio between the number of records produced by a join operation and the number of records of the Cartesian product of the two tables to be joined.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

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

- **Local Join**
  - Phase 2: Join co… a Hash Join Algorithm)

($|Ri|$ x (tr + … (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 Hash 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 Hash-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 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: $(\pi_R \times Ri \times \sigma j \times \pi_S \times S / P) \times IO$

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**
  - 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**
  - 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:

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

  - 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 a problem in parallel query processing. It is normally caused by data skew, or processing skew
  - No load imbalance in hash- 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

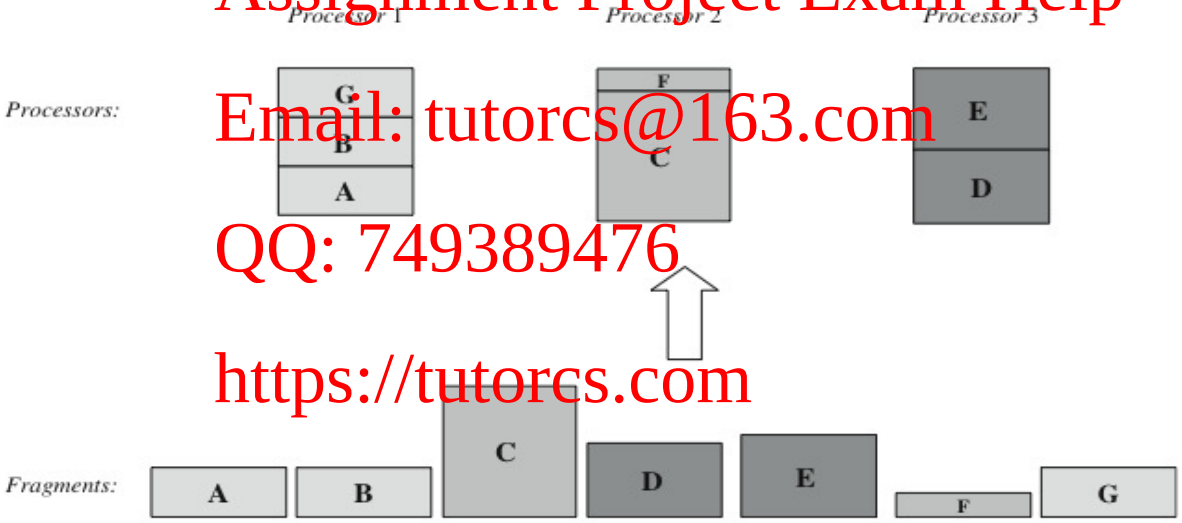


Figure 5.19 Load balancing

5.6. Summary

- Parallel join is one of the [illegible]ortant 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

MONASH University

Information Technology

FIT5202 (Vo[illegible]e III - Join)

Week 3b – Parallel Outer Join

Join Queries

- Two types of Join Qu
  - **Inner Join**

```
Select R.x, R.a, S.y, S.b
From R, S
Where R.a = S.b;
```

  - **Outer Join**

```
Select R.x, R.a, S.y, S.b
From R left outer join S on R.a = S.b;
```

Join Queries

R

	a
1	1
2	2
2	3

S

y	b
0	6
0	4
3	1
6	2
1	6
4	2
7	2
7	1
2	1
5	5
5	6
8	9

Results

x	a	y	b
0	1	3	1
0	1	7	1
0	1	2	1
1	2	6	2
1	2	4	2
1	2	7	2

- **Inner Join**

  Select R.x, R.a, S.y, S.b
  From R, S
  Where R.a = S.b;

Join Queries

R

	a
1	1
2	2
3	3

S

y	b
0	6
0	4
3	1
6	2
1	6
4	2
7	2
7	1
2	1
5	5
5	6
8	9

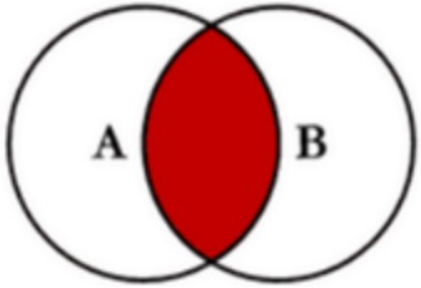
Results

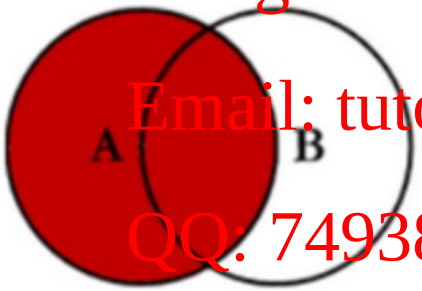
x	a	y	b
0	1	3	1
0	1	7	1
0	1	2	1
1	2	6	2
1	2	4	2
1	2	7	2
2	3	Null	Null

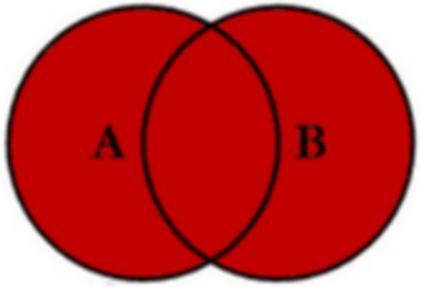
- **Outer Join**

  Select R.x, R.a, S.y, S.b
  From R left outer join S
  On R.a = S.b;

- **Exercise 1**
  - Identify the LEFT (IN?







https://www.codeproject.com/Articles/33052/Visual-Representation-of-SQL-Joins

Parallel Join Query Processing

- Parallel Inner Join co
  - **Data Partitioni**
    - Divide and Broadcast
    - Disjoint Partitioning
  - **Local Join**
    - Nested-Loop Join
    - Sort-Merge Join
    - Hash Join
- Example of a Parallel Inner Join Algorithm
  - **Divide and Broadcast, plus Hash Join**

Parallel Join Query Processing

- Parallel Outer Join pr[illegible]methods
  - **ROJA** (Redistri[illegible]er Join Algorithm)
  - **DOJA** (Duplication Outer Join Algorithm)
  - **DER** (Duplication & Efficient Redistribution)
- Load Balancing
  - **OJSO** (Outer Join Skew Optimization)

1. ROJA

Step 1: Distribute or reshuffle data based on join attribute.
Step 2: Each processor performs local outer Join.

SELECT R.x, R.a, S.y, S.b
FROM R left outer join S on R.a

Eg. Using hash func h(i) = i mod 3 + 1

h(i)= 1

R		S	
x	a	y	b
0	1	NULL	NULL

S	
0	4
3	1
6	2

Processor 1

h(i)= 2

R		S	
x	a	y	b
1	2	1	6

S	
4	2
7	2
7	1

Processor 2

h(i)= 3

R		S	
x	a	y	b
2	3	2	1

S	
5	5
5	6
8	9

Processor 3

2. DOJA

SELECT R.x, R.a, S.y, S.b
FROM R left outer join S on R.a

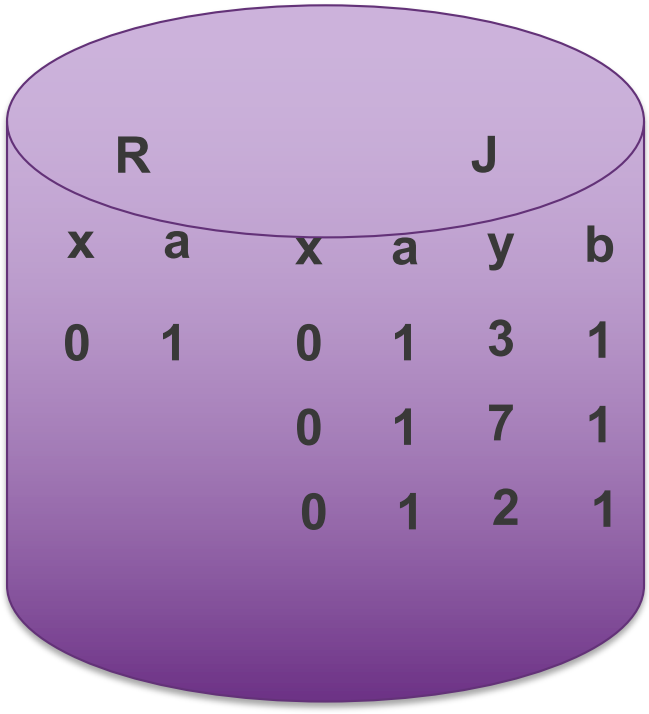
Step 1: Replicate small table.
Step 2: Local Inner Join
Step 3: Hash redistribute inner join result based on attribute x.

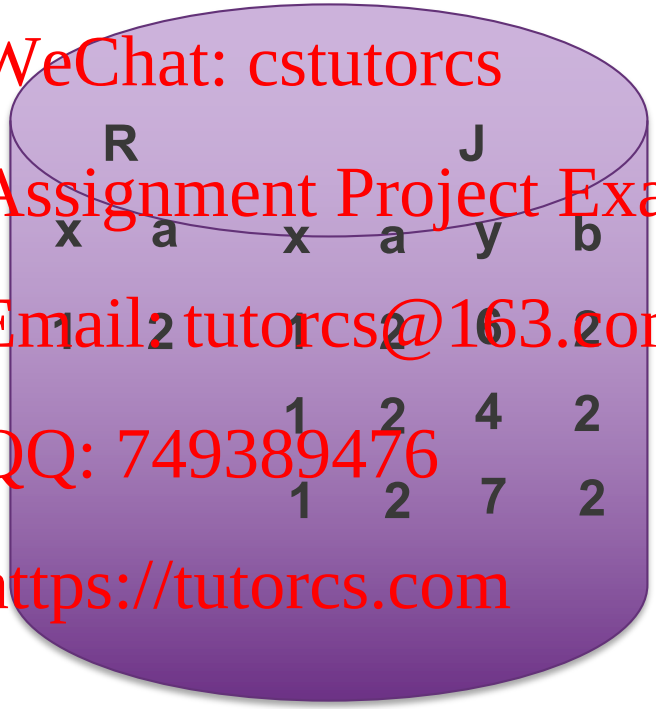
Processor 1 **Processor 2** **Processor 3**

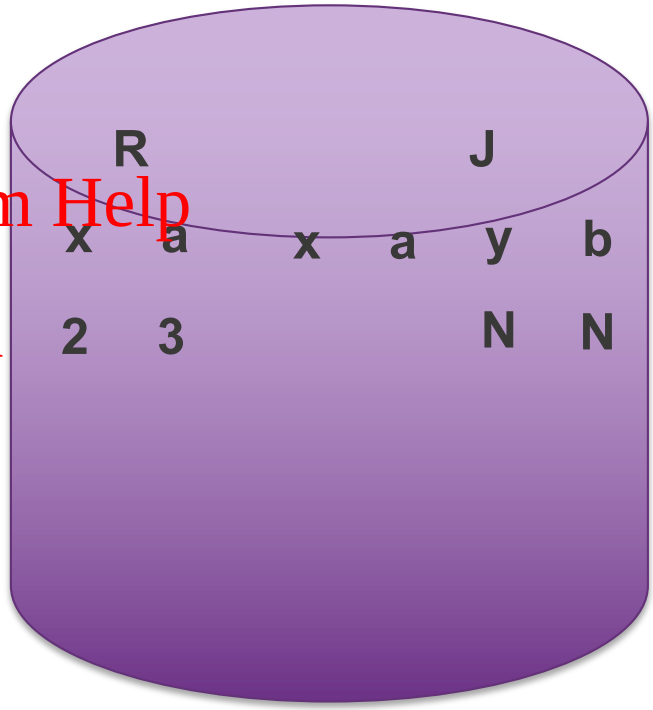
2. DOJA

Step 4: Local outer join

SELECT R.x, R.a, S.y, S.b
FROM R left outer join S on R.a





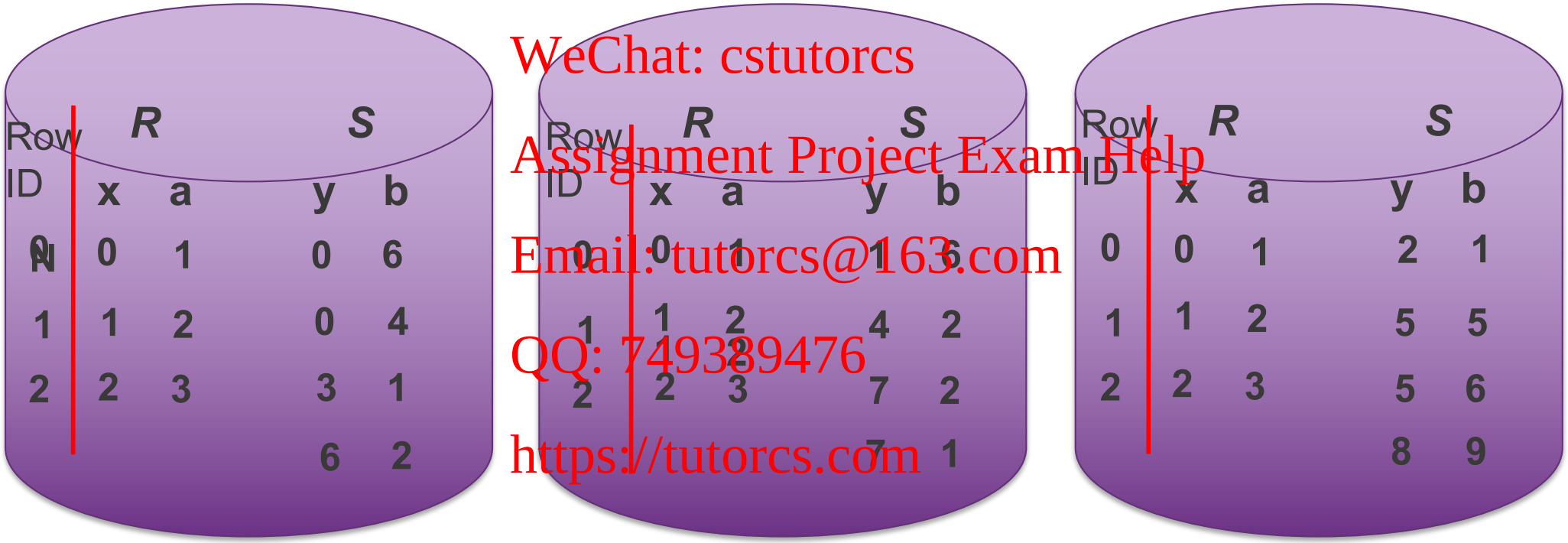


Processor 1 **Processor 2** **Processor 3**

3. DER

SELECT R.x, R.a, S.y, S.b
FROM R left outer join S on R.a =

Step 1: Replicate small table (left)
Step 2: Local Inner Join
Step 3: Select ROW ID of left table with no matches.
Step 4: Redistribute the ROW ID.
Step 5: Store the ROW ID that appears as many times as the number of processors

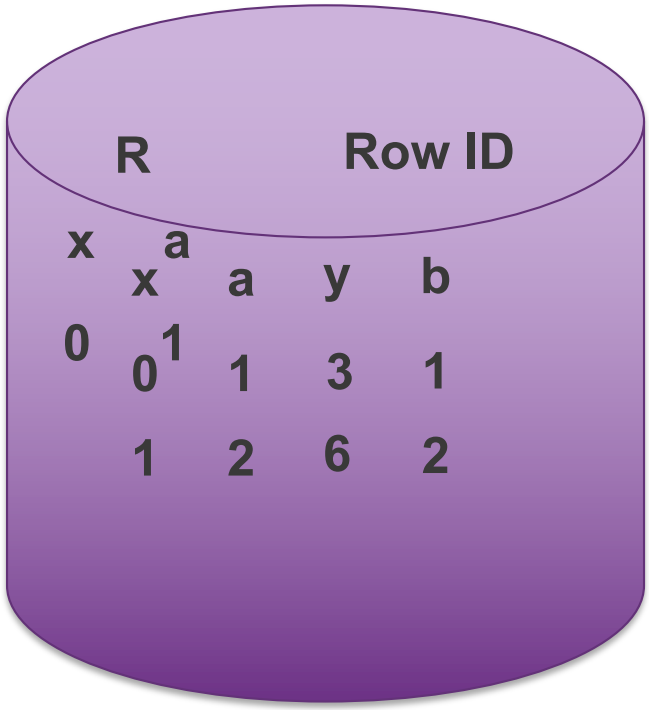


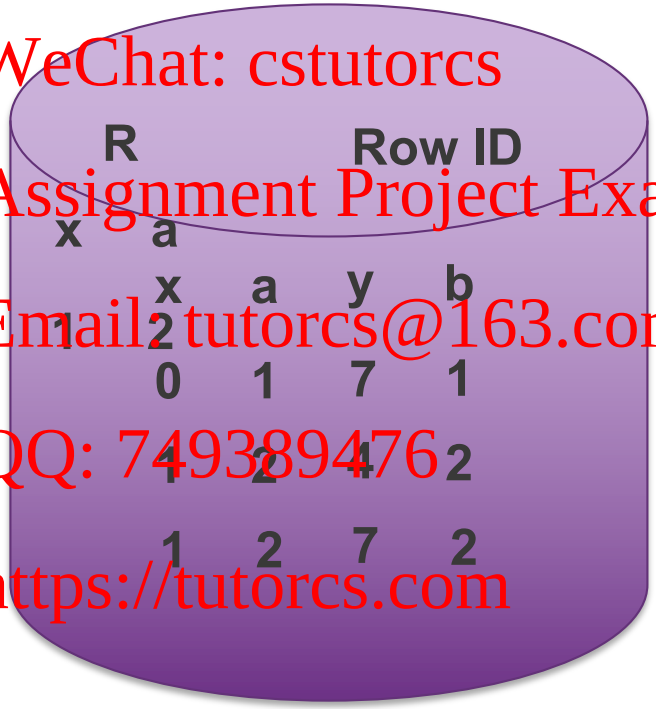
Processor 1 **Processor 2** **Processor 3**

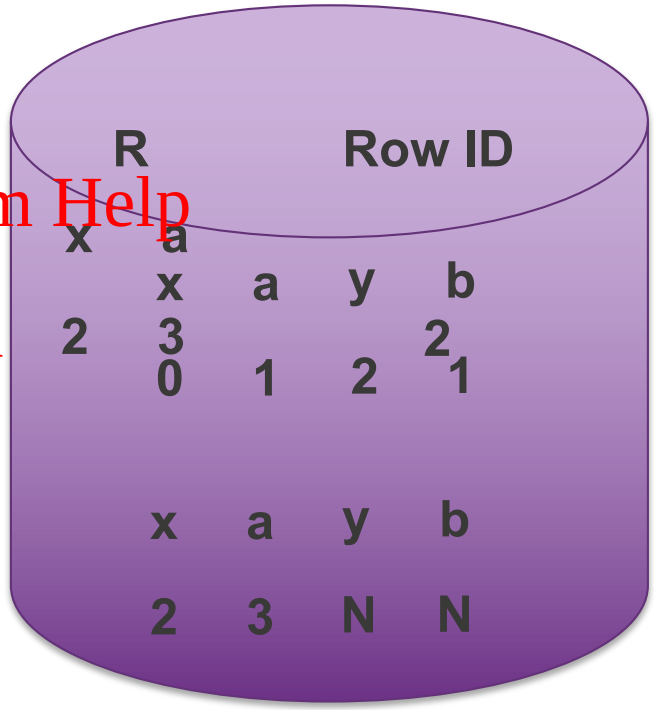
3. DER

Step 6: Inner join

SELECT R.x, R.a, S.y, S.b
FROM R left outer join S on R







Processor 1 **Processor 2** **Processor 3**

Parallel Join Query Processing

- Parallel Outer Join pr[illegible]methods
  - **ROJA** (Redistri[illegible]er Join Algorithm)
  - **DOJA** (Duplication Outer Join Algorithm)
  - **DER** (Duplication & Efficient Redistribution)
- Load Balancing
  - **OJSO** (Outer Join Skew Optimization)

	ROJA	DOJA	DER
Steps	**Step 1:** Distribute or reshuffle the data based on the join attribute. **Step 2:** Each processor performs the Local outer Join.	**Step 1:** Replication. We duplicate the small table. **Step 2:** Local Inner Join **Step 3:** Redistribute the inner join based on attribute X. **Step 4:** Local outer join	**Step 1:** Replication. We broadcast the left table. **Step 2:** Local Inner Join **Step 3:** Select the ROW ID of left table with no matches. **Step 4:** Redistribute the ROW ID. **Step 5:** Store the ROW ID that appears as many times as the number of processors. **Step 6:** Inner join
Pros	fast performance, only two steps	None. ROJA is faster than DOJA.	Redistributes dangling row IDs instead of actual records.
Cons	redistribution of data -> data skew, communication cost	In the replication step, if the table is large, the replication cost is expensive. In the distribution step, data skew and communication cost similar to ROJA	In the replication step, if the table is large, the replication cost is expensive.

Another example…

```
Select x, y, z, a,
From R left outer          on R.a=S.b
       left outer          on S.c=T.d;
```


- Initial Data Placement

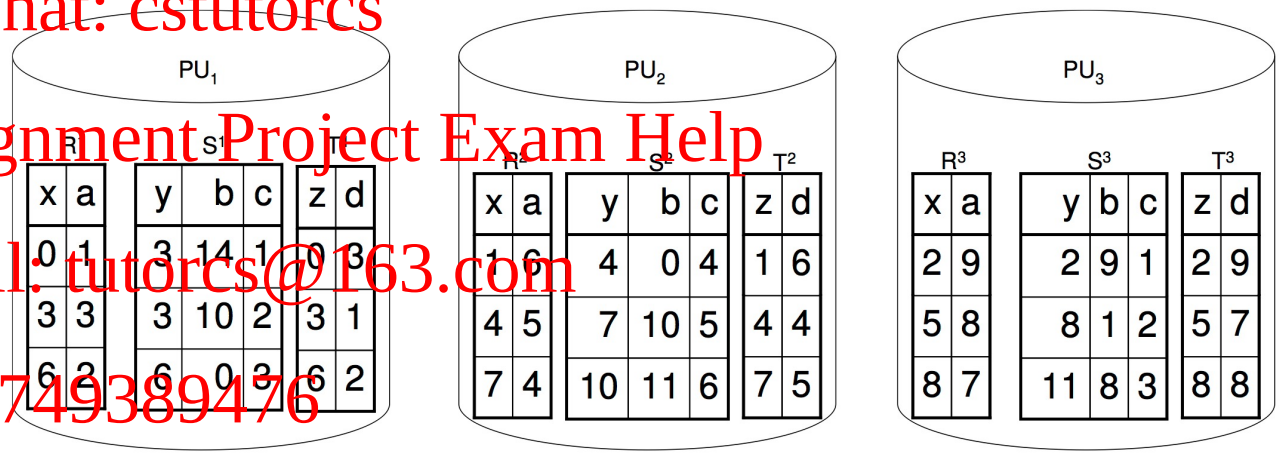


Figure 1: Three relations R, S and T are hash partitioned on a three parallel-unit system. The partitioning columns are $R.x$, $S.y$ and $T.z$ respectively. The hash function, $h(i) = i \ mod \ 3 + 1$, places a tuple with value i in the partitioning column on the $h(i)$-th PU.

Another example…

- Step 1: Redistribution of R and S (**why do we need to redistribute?**)

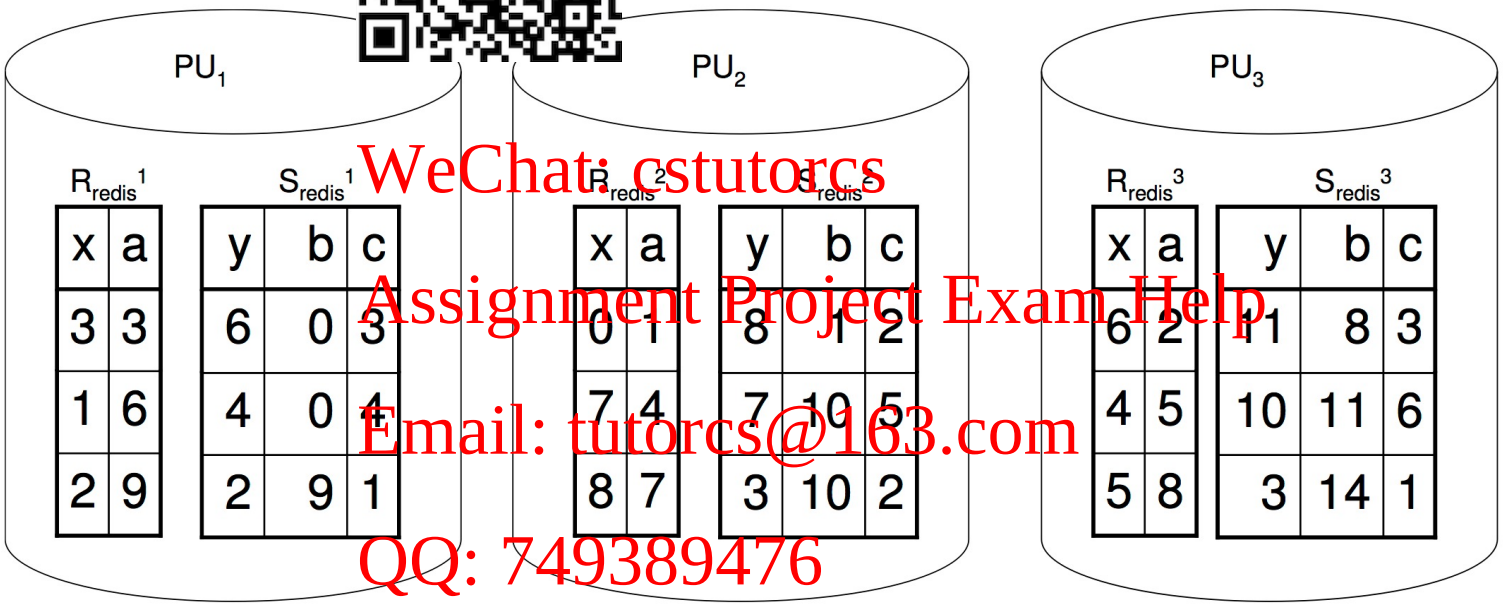


Figure 2: The result of hash redistributing R and S on their join attributes ($R.a$ and $S.b$) to two temporary tables R_{redis} and S_{redis}.

Another example…

- Step 2: (a) Outer Join R_{redis} and S_{redis} and store in J

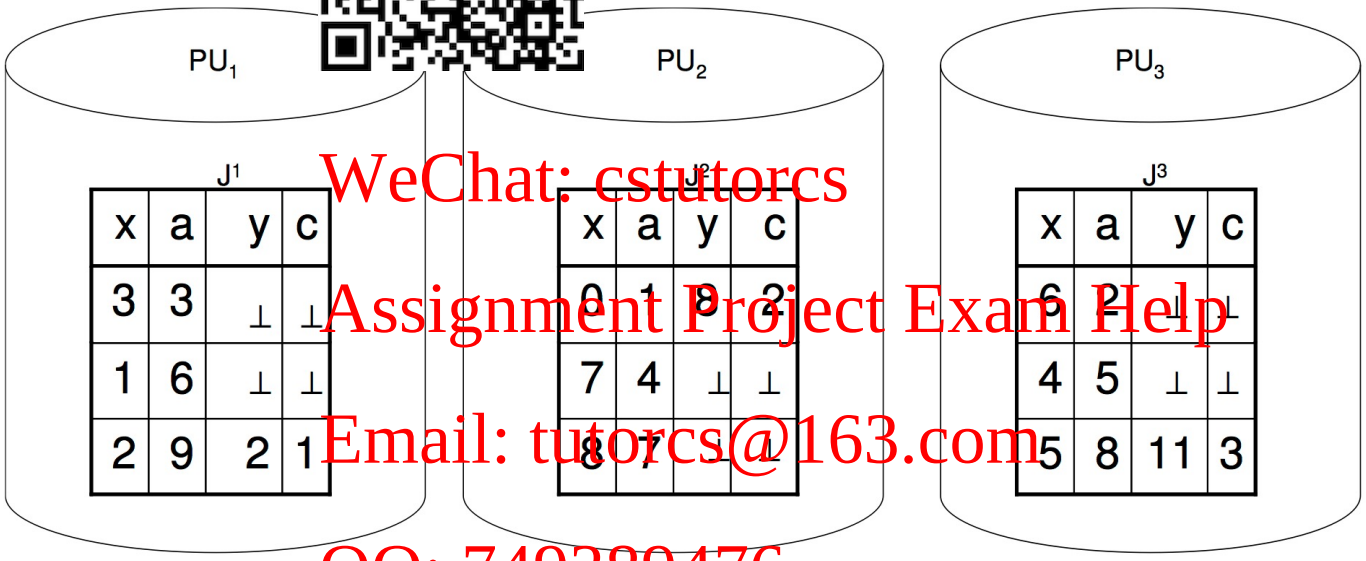


Figure 3: The results of left outer joining R_{redis} and S_{redis} (R_{redis} and S_{redis} are shown in Figure 2) are stored in a temporary table J.

Another example…

- Step 2: (b) Redistribu

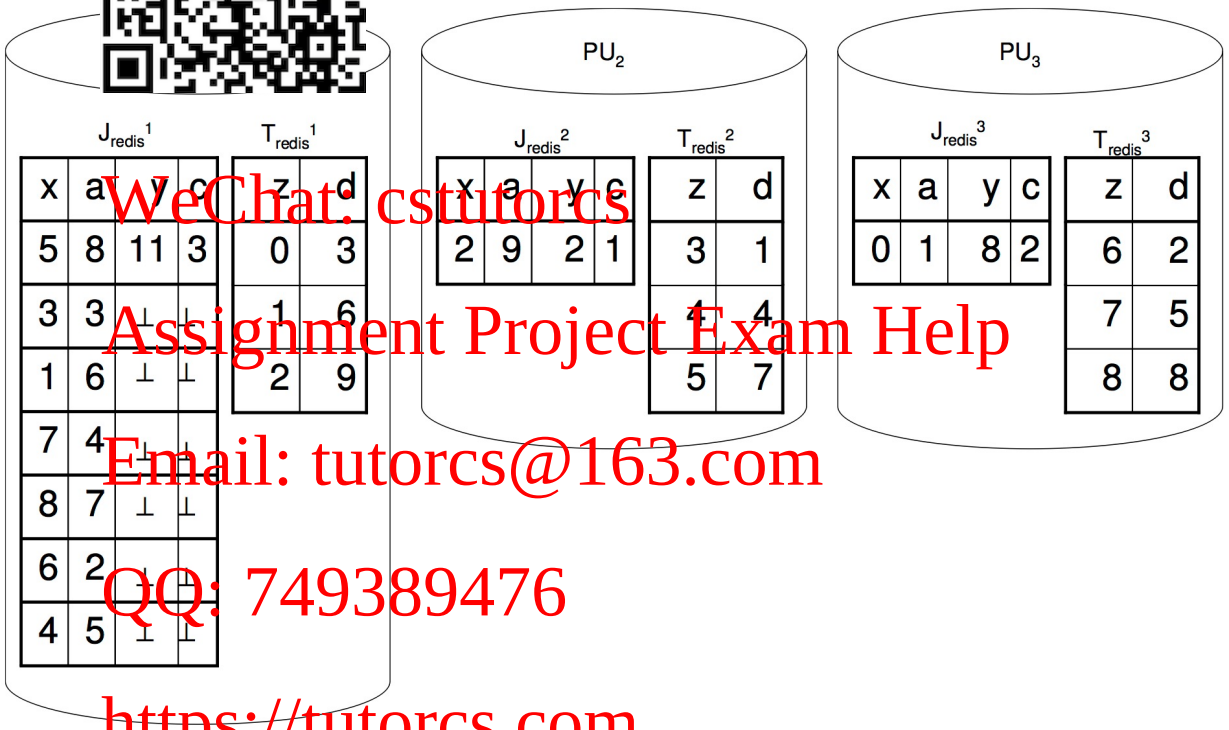


Figure 4: The result of hash redistributing J (shown in Figure 3) and T (shown in Figure 1) on their join attributes ($J.c$ and $T.d$) to two temporary tables J_{redis} and T_{redis}.

Another example…

- Step 3: Outer Join J a[illegible]nal Results

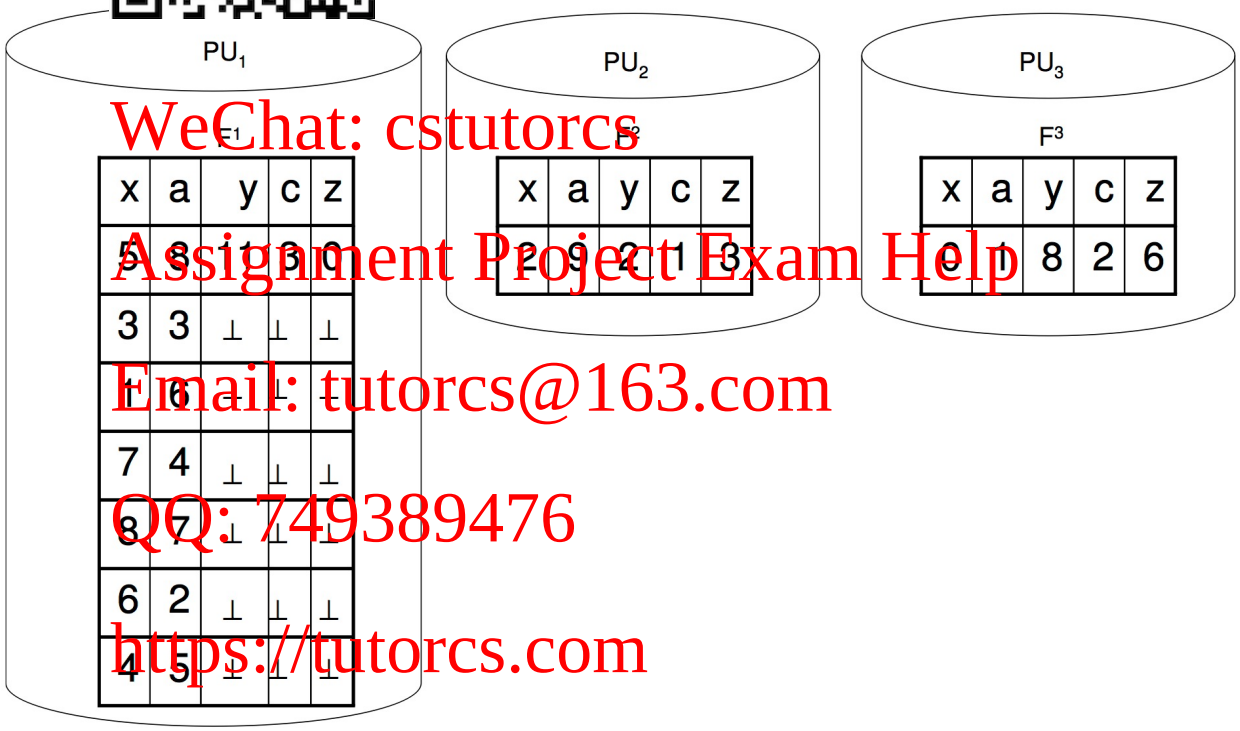


Figure 5: The final results of the two outer joins in Query 1 are stored in a temporary table F.

Conclusion…

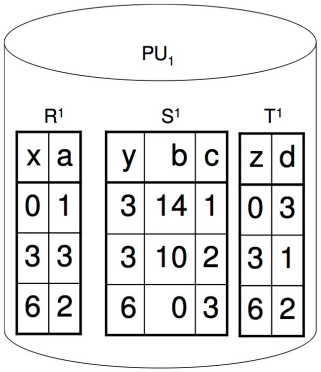
- Skew can easily happen in Outer Join queries

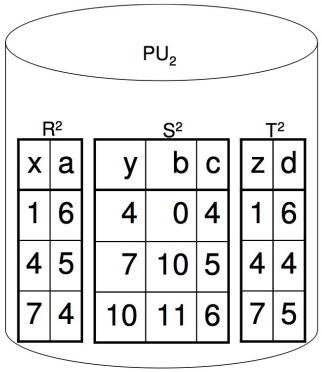
- **Exercise 1 (FLUX Quiz**

  - In the previous exa[illegible]uter join S outer join T), which parallel outer join method [illegible]
  - A. ROJA
  - B. DOJA
  - C. DER
  - D. None of the above

A better solution… OJSO

- Step 1: Redistribute R (same as the previous example)





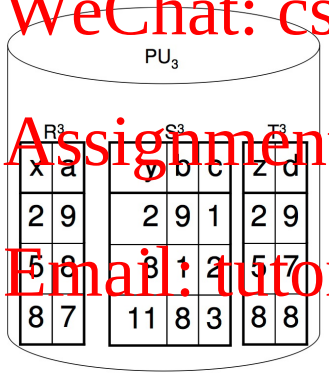
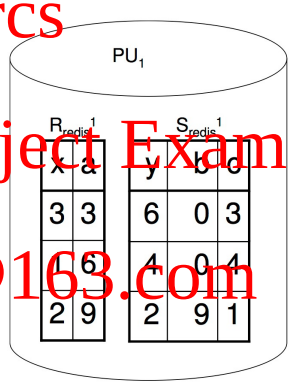
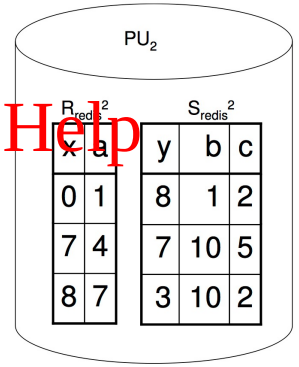


Figure 1: Three relations R, S and T are hash partitioned on a three parallel-unit system. The partitioning columns are $R.x$, $S.y$ and $T.z$ respectively. The hash function, $h(i) = i \ mod \ 3 + 1$, places a tuple with value i in the partitioning column on the $h(i)$-th PU.





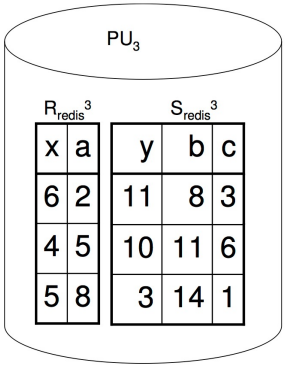


Figure 2: The result of hash redistributing R and S on their join attributes ($R.a$ and $S.b$) to two temporary tables R_{redis} and S_{redis}.

A better solution… OJSO

- Step 2: (a) Outer Join … but the results are divided into J_{2redis} and J_{local}

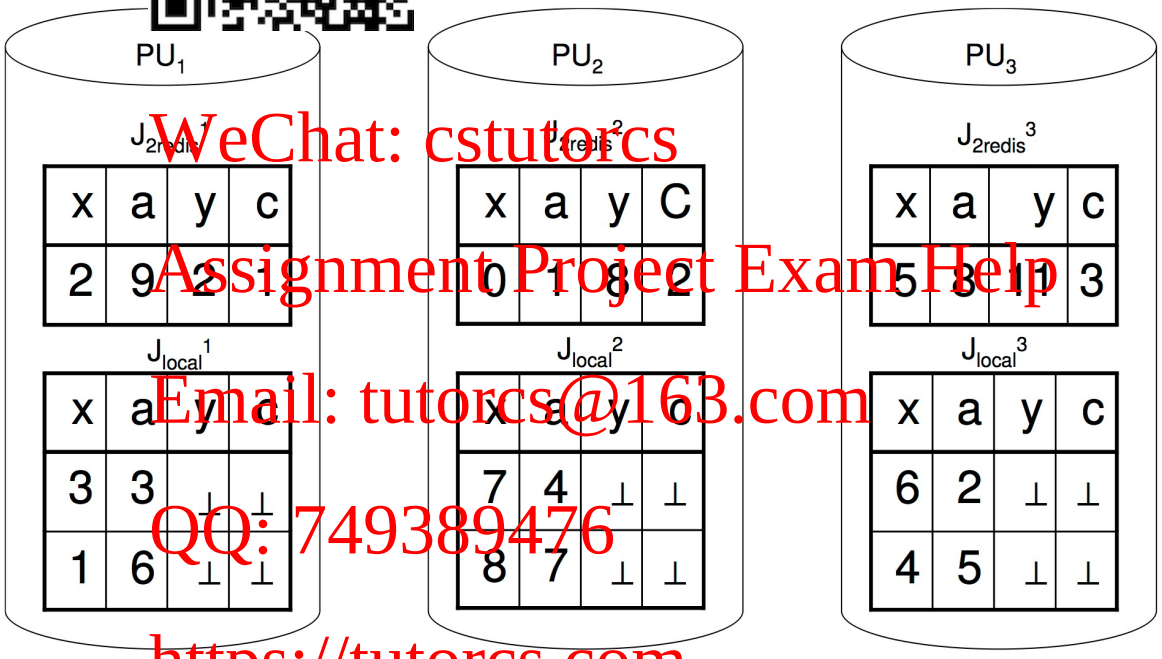


Figure 9: The results of left outer joining R_{redis} and S_{redis} are split into two temporary tables J_{2redis} and J_{local}.

A better solution… OJSO

- Step 2: (b) Redistribu … d T; and do an outer join

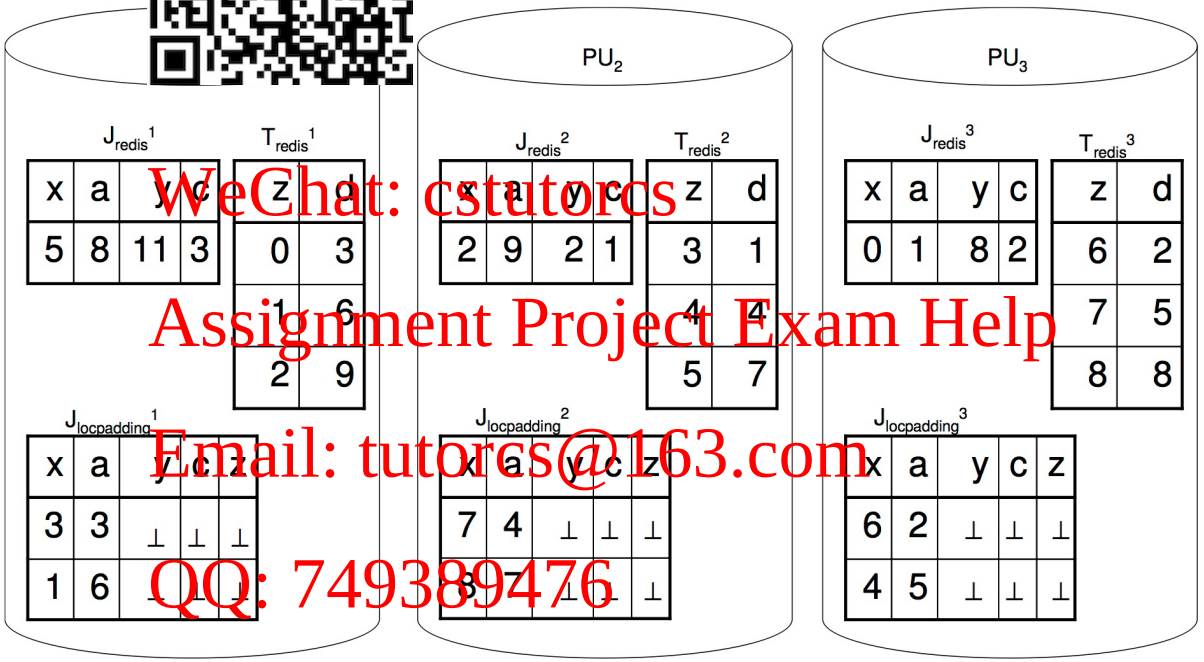


Figure 10: The result of hash redistributing J_{2redis} (shown in Figure 9) and T (shown in Figure 1) on their join attributes to two temporary tables J_{redis} and T_{redis}. $J_{locpadding}$ is created from J_{local} (shown in Figure 9) with padded nulls.

A better solution… OJSO

- Step 3: Union the fina each processor

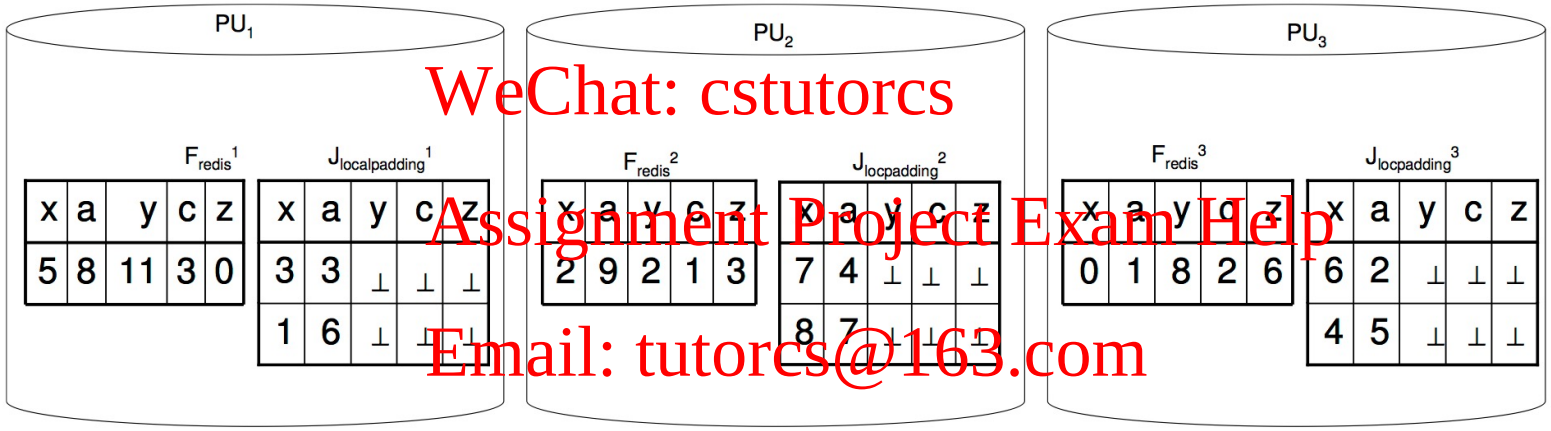


Figure 11: The results of the second outer join in Query 1 are stored in a temporary table F. The final result for Query 1 is the union of F and $J_{locpadding}$.

OJSO Conclusion…

- Do not redistribute th records from the previous outer join

Summary…

- Parallel Outer Join pr[illegible]ethods
  - **ROJA** (Redistri[illegible]er Join Algorithm)
  - **DOJA** (Duplication Outer Join Algorithm)
  - **DER** (Duplication & Efficient Redistribution)
- Load Balancing
  - **OJSO** (Outer Join Skew Optimization)

References

- Xu, Y. & Kostamaa, P. (2010). A new algorithm for small-large table outer joins in parallel DBMS. In *Proceedings of the 26th Intl Conference on Data Engineering (ICDE'2010)* (pp. 1018-1024), IEEE Comp Society Press.
- Xu,Y. & Kostamaa, P. (2009). Efficient Outer Join Data Skew Handling in Parallel DBMS. In *Proceedings of the 35th International Conference on Very Large Data Bases (VLDB'2009)* (pp. 1390-1396), VLDB Endowment.