MONASH University

Information Technology

FIT5202 (Volume IV – Sort and Group By)

Week 4a – Parallel Sort

Chapter 5
Parallel Join

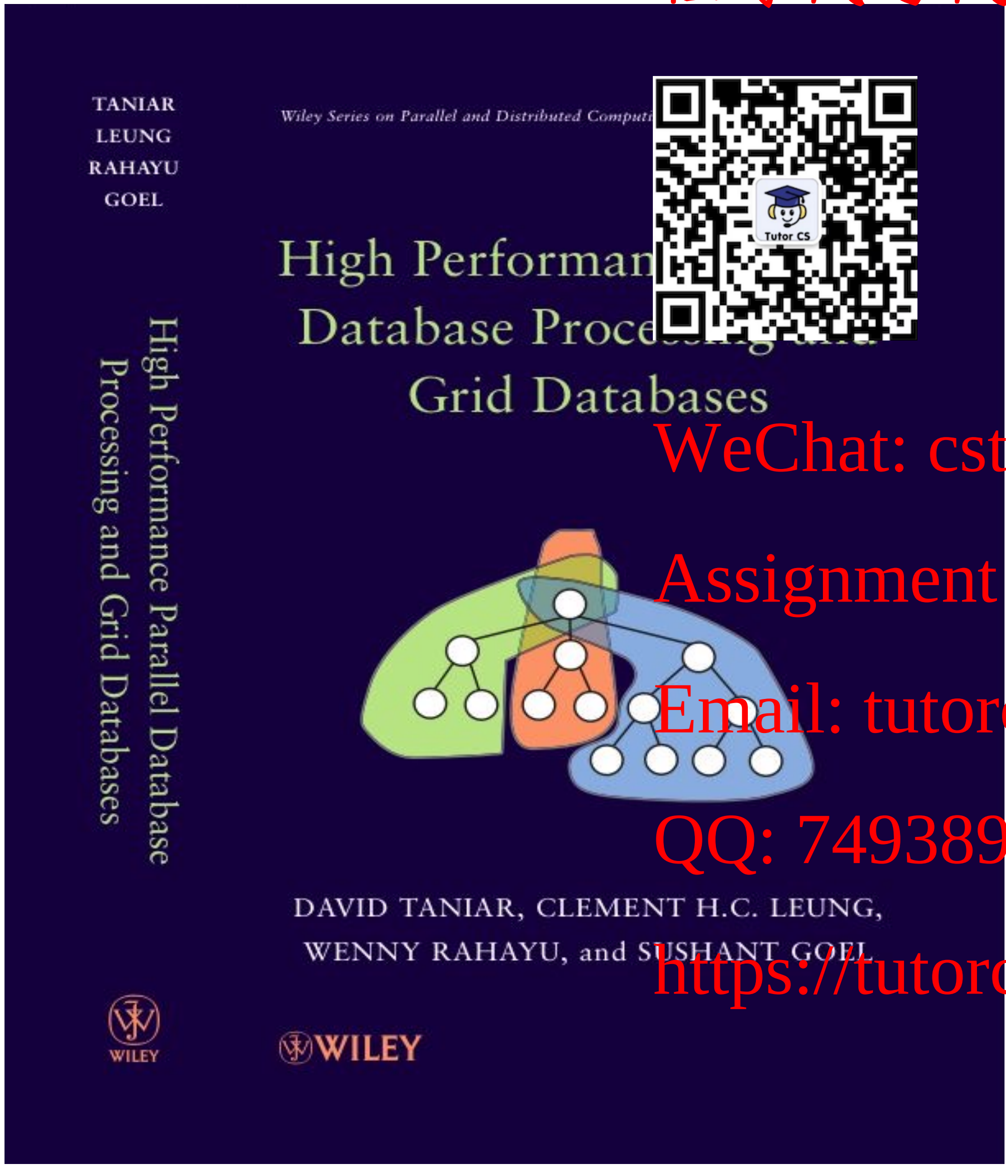
Last Week

Revision

- **Exercise 1 (FLUX Quiz**
  - Parallel Join algorithms f consists of two major phases: Data Partitioning, and Local Join
  - A. TRUE
  - B. FALSE

Revision

- **Exercise 2 (FLUX Quiz**
  - Parallel Join algorithms f queries are:
  - A. ROJA and DOJA
  - B. DER
  - C. OJSO
  - E. only A and B
  - F. A. B and C are correct



Chapter 4
Parallel Sort and GroupBy

4.1. Sorting, and Serial Sorting

- Serial Sorting – **IN**[illegible]
  - The data to be sorted fits entirely into the main memory

- Serial Sorting - **EXTERNAL**
  - The data to be sorted DOES NOT fit entirely into the main memory

4.1. Internal Serial Sorting (cont'd)

- **Bubble Sort**
  - Based on swapping
  - It compares the first two elements, and if the first is greater than the second, it swaps them.
  - It continues doing this for each pair of adjacent elements to the end of the data set.
  - It then starts again with the first two elements, repeating until no swaps have occurred on the last pass.
  - Example: 6 5 3 1 8 7 2 4

4.1. Internal Serial Sorting (cont'd)

6 5 3 1 8 7 2 4

Bubble Sort

6 5 3 1 8 7 2 4
5 6 3 1 8 7 2 4
5 3 6 1 8 7 2 4
5 3 1 6 8 7 2 4
5 3 1 6 8 7 2 4
5 3 1 6 7 8 2 4
5 3 1 6 7 2 8 4
5 3 1 6 7 2 4 8

5 3 1 6 7 2 4 8
3 5 1 6 7 2 4 8
3 1 5 6 7 2 4 8
3 1 5 6 7 2 4 8
3 1 5 6 7 2 4 8
3 1 5 6 2 7 4 8
3 1 5 6 2 4 7 8

3 1 5 6 2 4 7 8
1 3 5 6 2 4 7 8
1 3 5 6 2 4 7 8
1 3 5 6 2 4 7 8
1 3 5 2 6 4 7 8
1 3 5 2 4 6 7 8

1 3 5 2 4 6 7 8
1 3 5 2 4 6 7 8
1 3 5 2 4 6 7 8
1 3 2 5 4 6 7 8
1 3 2 4 5 6 7 8

1 3 2 4 5 6 7 8
1 3 2 4 5 6 7 8
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8

1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8

Finished

4.1. Internal Serial Sorting (cont'd)

- **Insertion Sort**
  - Based on inserting
  - It works by taking e the list one by one and inserting them in their correct position into a new sorted list. In arrays, the new list and the remaining elements can share the array's space, but insertion is expensive, requiring shifting all following elements over by one.
  - Example: 6 5 3 1 8 7 2 4

6 5 3 1 8 7 2 4	Take out 6, and insert it in the previous list	**6** 5 3 1 8 7 2 4
6 **5** 3 1 8 7 2 4	Take out 5, and insert it in the previous list	**5 6** 3 1 8 7 2 4
5 6 **3** 1 8 7 2 4	Take out 3, and insert it in the previous list	**3 5 6** 1 8 7 2 4
3 5 6 **1** 8 7 2 4	Take out 1, and insert it in the previous list	**1 3 5 6** 8 7 2 4
1 3 5 6 **8** 7 2 4	Take out 8, and insert it in the previous list	**1 3 5 6 8** 7 2 4
1 3 5 6 8 **7** 2 4	Take out 7, and insert it in the previous list	**1 3 5 6 7 8** 2 4
1 3 5 6 7 8 **2** 4	Take out 2, and insert it in the previous list	**1 2 3 5 6 7 8** 4
1 2 3 5 6 7 8 **4**	Take out 4, and insert it in the previous list	**1 2 3 4 5 6 7 8**

Finished

4.1. Internal Serial Sorting (cont'd)

- **Quick Sort**
  - Quick Sort is a divide and conquer algorithm which relies on a partition operation: to partition an array an element called a *pivot* is selected.
  - All elements smaller than the pivot are moved before it and all greater elements are moved after it.
  - The lesser and greater sublists are then recursively sorted.
  - The most complex issue in Quick Sort is choosing a good pivot element; consistently poor choices of pivots can result in drastically slower performance
  - Example: 6 5 3 1 8 7 2 4

6 5 3 1 8 7 2 4

4.1. Internal Serial Sorting (cont'd)

- **Quick Sort**
  - Quick Sort is a divide and conquer algorithm which relies on a partition operation: to partition an array an element called a *pivot* is selected.
  - All elements smaller than the pivot are moved before it and all greater elements are moved after it.
  - The lesser and greater sublists are then recursively sorted.
  - The most complex issue in Quick Sort is choosing a good pivot element; consistently poor choices of pivots can result in drastically slower performance
  - Example: 6 5 3 1 8 7 2 4

6 5 3 1 8 7 2 4

Homework: Work out step-by-step this Quick Sort example

4.2. Serial External Sorting

- Sorting is expressed by the ORDER BY clause in SQL
- Duplicate remove is identified by the keyword DISTINCT in SQL

```
Query 4.1:
    Select *
    From STUDENT
    Order By Sdegree;
```

```
Query 4.3:
    Select Distinct Sdegree
    From STUDENT;
```

4.2. Serial External Sorting (cont'd)

- External sorting assumes that the data does not fit into main memory
- Most common external sorting is sort-merge
- Break the file up into unsorted subfiles, sort the subfiles, and then merge the subfiles into larger and larger sorted subfiles until the entire file is sorted


```
Algorithm: Serial External Sorting

// Sort phase - Pass 0
1. Read B pages at a time into memory
2. Sort them, and Write out a sub-file
3. Repeat steps 1-2 until all pages have been processed

// Merge phase - Pass i = 1, 2, ...
4. While the number of sub-files at end of previous pass
   is > 1
5. While there are sub-files to be merged from
   previous pass
6.    Choose B-1 sorted sub-files from the previous pass
7.    Read each sub-file into an input buffer page
      at a time
8.    Merge these sub-files into one bigger sub-file
9.    Write to the output buffer one page at a time
```

Figure 4.1 External sorting algorithm based on sort-merge

4.2. Serial External Sorting (cont'd)

- **Example**
  - File size to be sc[illegible]ages, number of buffer (or memory size) = 5 pages
  - Number of subfil[illegible]22 subfiles (the last subfile is only 3 pages long).
  - **Pass 0** (sorting p[illegible]ach subfile, **read from disk**, **sort in main-memory**, and **write to disk** (Note: sorting the data in main-memory can use any fast in-memory sorting method, like Quick Sort)

  - Merging phase: **We use *B*-1 buffers (4 buffers) for input and 1 buffer for output**

  - **Pass 1**: Read 4 sorted subfiles and perform 4-way merging (apply a need *k*-way algorithm). Repeat the 4-way merging until all subfiles are processed. Result = 6 subfiles with 20 pages each (except the last one which has 8 pages)

  - **Pass 2**: Repeat 4-way merging of the 6 subfiles like pass 1 above. Result = 2 subfiles

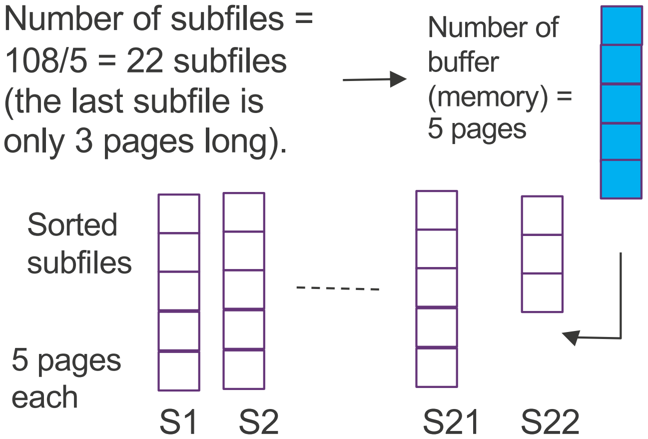
  - **Pass 3**: Merge the last 2 subfiles

  - Summary: 108 pages and 5 buffer pages require 4 passes

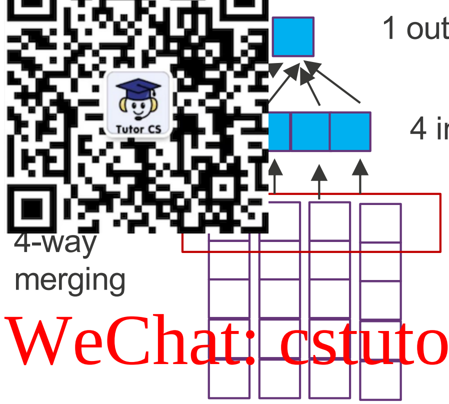
Pass 0 (sorting phase):

Number of subfiles = 108/5 = 22 subfiles (the last subfile is only 3 pages long).

Number of buffer (memory) = 5 pages



Pass 1 (Merging phase):



S1-S4, S5-S8, S9-S12, S13-S16, S17-S20 —4-way merging→ SS1, SS2, SS3, SS4, SS5 — 4 X 5 pages = 20 pages each

S21 & S22 —2-way merging→ SS6 — 5 + 3 pages = 8 pages

Pass 2 (Merging phase):

SS1-SS4 —4-way merging→ SSS1 — 4 X 20 pages = 80 pages

SS5 & SS6 —2-way merging→ SSS2 — 20 + 8 pages = 28 pages

Pass 3 (Merging phase):

SSS1 & SSS2 —2-way merging→ SSSS1 (sorted array)

4.2. Serial External Sorting (cont'd)

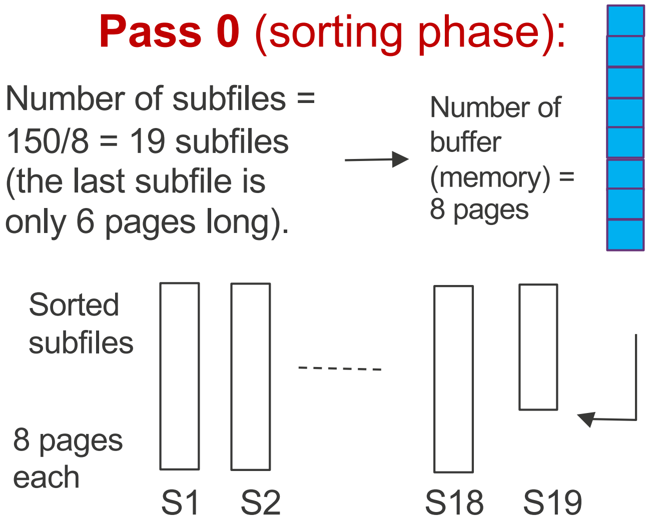
- **Exercise 3 (FLUX Quiz**
  - There are 150 data page[illegible]d. The machine that we have has a limited memory, and can only take 8 page[illegible]How many passes will it take to sort the 150 data pages?

  - A. 2
  - B. 3
  - C. 4
  - D. 5

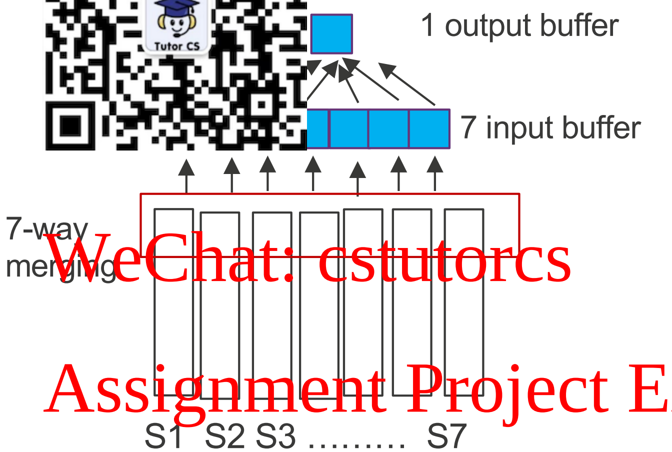
File size to be sorted = 150 pages, number of buffer (or memory size) = 8 pages

Pass 0 (sorting phase):

Number of subfiles = 150/8 = 19 subfiles (the last subfile is only 6 pages long).



Pass 1 (Merging phase):



S1-S7 S8-S14	7-way merging →	SS1 SS2	7 X 8 pages = 56 pages each
S15-S18 & S19	5-way merging →	SS3	4 X 8 + 6 pages = 38 pages

Pass 2 (Merging phase):

SS1 & SS2 & SS3 $\xrightarrow{\text{3-way merging}}$ SSS1 (sorted array)

4.2. Serial External Sorting (cont'd)

- **Example**
  - Buffer size plays an important role in external sort

Table 4.1 Number of passes in serial external sorting as number of buffer increases

R	B = 3	B = 5	B = 9	B = 17	B = 129	B = 257
100	7	4	3	2	1	1
1,000	10	5	4	3	2	2
10,000	13	7	5	4	2	2
100,000	17	9	6	5	3	3
1 million	20	10	7	5	3	3
10 million	23	12	8	6	4	3
100 million	26	14	9	7	4	4
1 billion	30	15	10	8	5	4

4.3. Parallel External Sort

5 different Algorithms

- Parallel Merge-All Sort
- Parallel Binary-Merge Sort
- Parallel Redistribution Binary-Merge Sort
- Parallel Redistribution Merge-All Sort
- Parallel Partitioned Sort

Without data redistribution

With data redistribution

4.3. Parallel External Sort (cont'd)

- **Parallel Merge-A**
  - A traditional approach
  - Two phases: local sort and final merge
  - Load balanced in local sort
  - Problems with merging:
    - Heavy load on one processor
    - Network contention

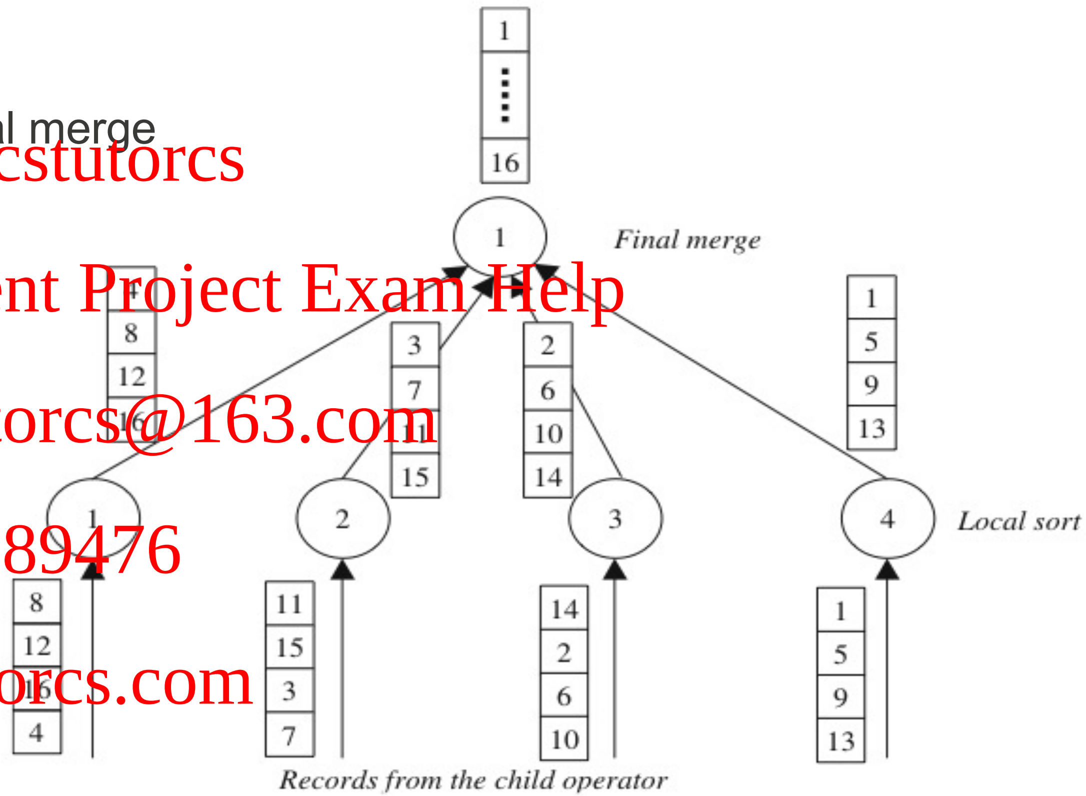


Figure 4.3 Parallel merge-all sort

4.3. Parallel External Sort (cont'd)

- **Parallel Binary-Merge Sort**
  - Local sort similar to traditional method
  - Merging in pairs only
  - Merging work is now spread to pipeline of processors, but merging is still heavy

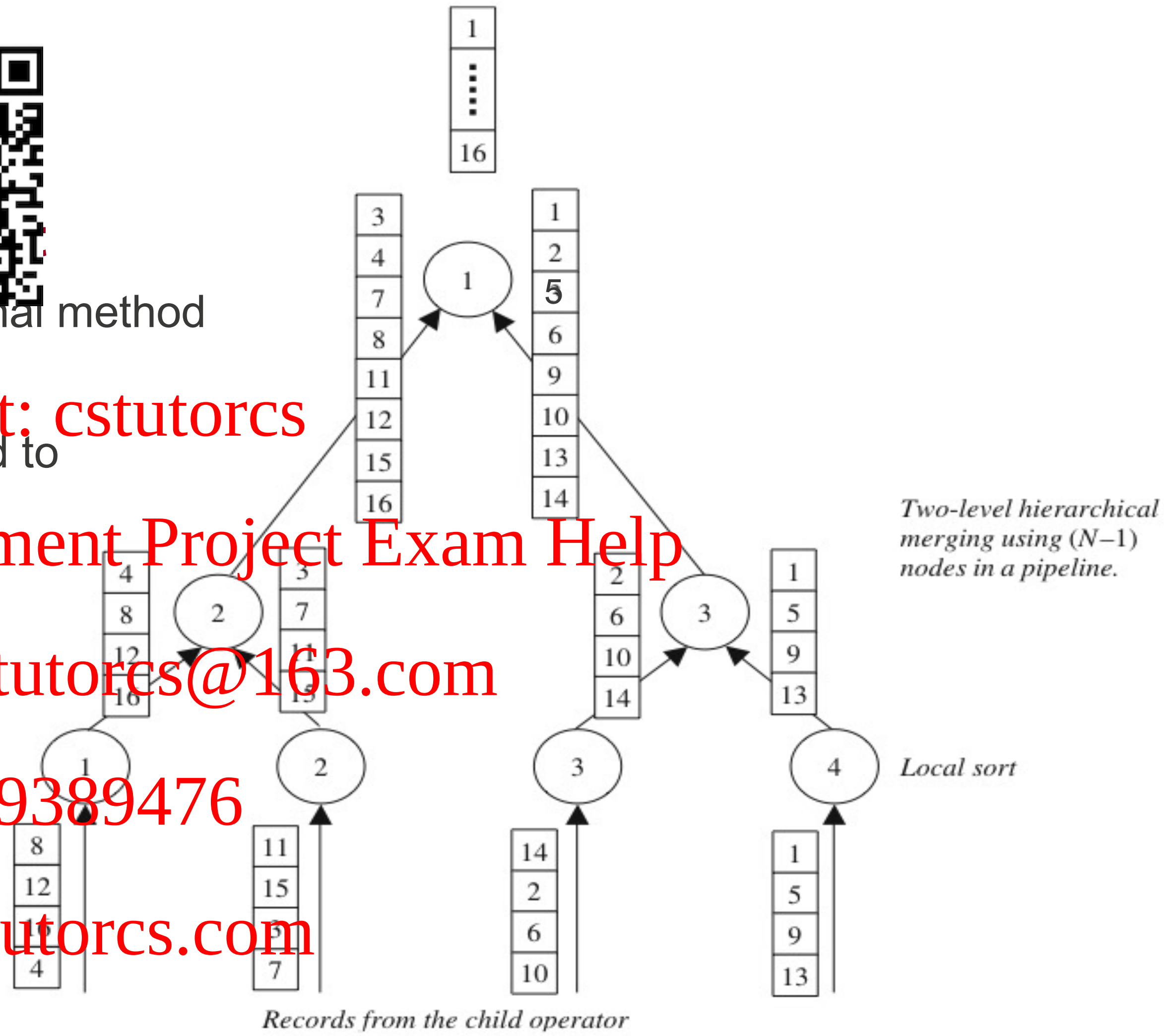
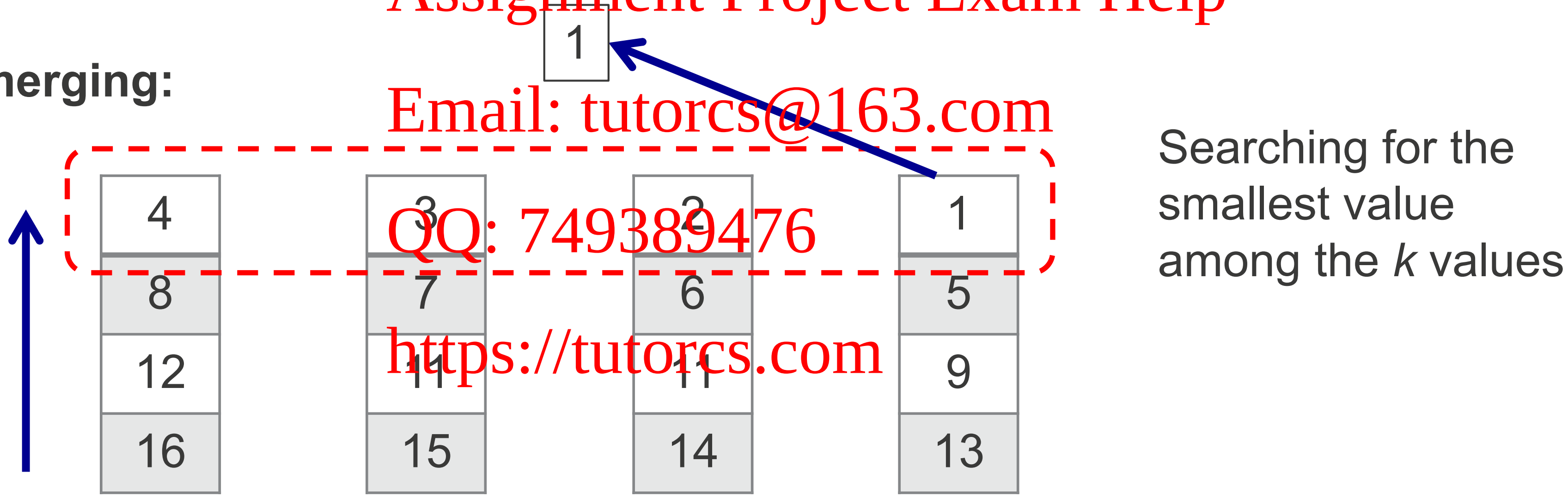


Figure 4.4 Parallel binary-merge sort

4.3. Parallel External Sort (cont'd)

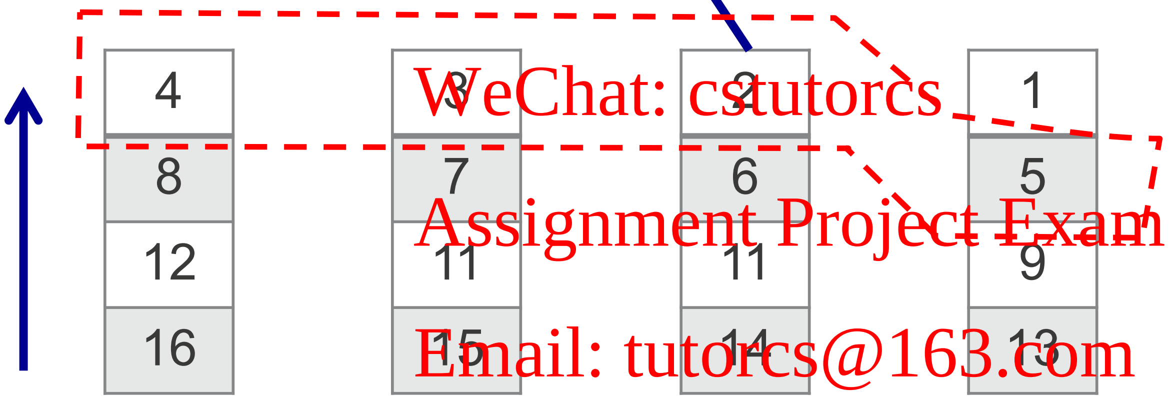
- **Parallel Binary-Merge Sort**
  - Binary merging vs. *k*-way merging
  - In *k*-way merging, the **searching for the smallest value among *k* partitions** is done at the same time
  - In binary merging, it is pairwise, but can be time consuming if the list is long
  - System requirements: ***k*-way merging requires *k* files open simultaneously**, but the pipeline process in binary merging requires extra overheads

***k*-way merging:**



4.3. Parallel External Sort (cont'd)

k-way merging:



Searching for the smallest value among the *k* values

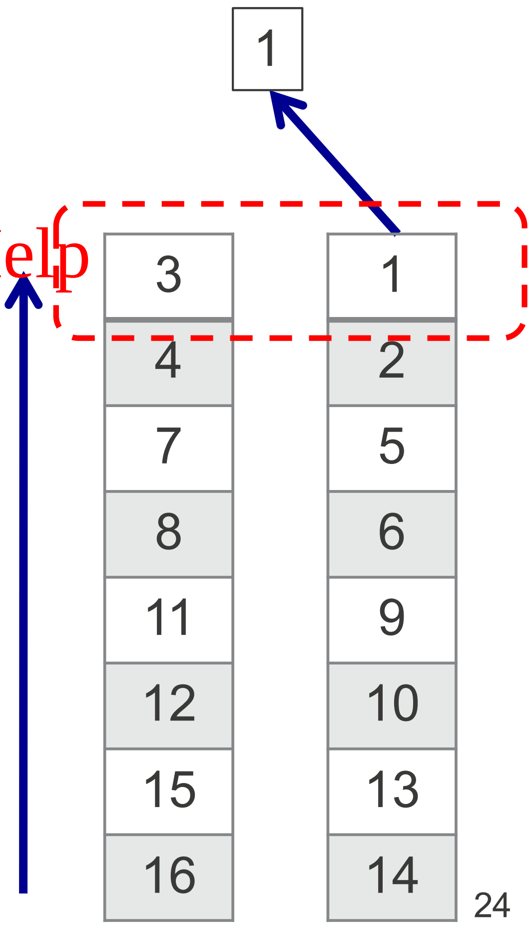
and so on…

4.3. Parallel External Sort (cont'd)

- **Parallel Binary-Merge Sort (Binary Merging step)**
  - Binary merging vs. *k*-w
  - In **binary merging**, it i ut can be time consuming if the list is long
  - System requirements: the pipeline process in binary merging requires extra overheads

Binary merging:

Compare two values only, but lists are longer



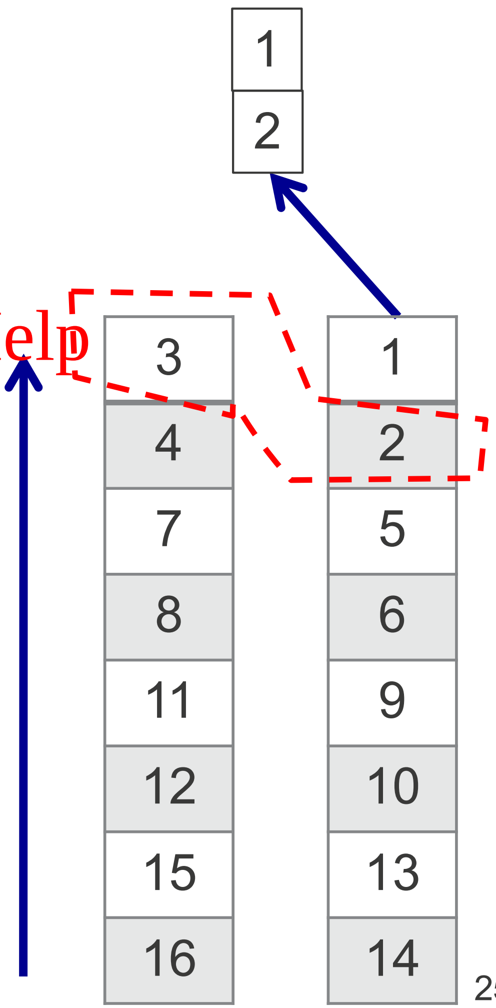
4.3. Parallel External Sort (cont'd)

- **Parallel Binary-Merge Sort (Binary Merging step)**
  - Binary merging vs. *k*-w
  - In **binary merging**, it i ut can be time consuming if the list is long
  - System requirements: the pipeline process in binary merging requires extra overheads

Binary merging:

Compare two values only, but lists are longer

And so on…



- **Parallel Redistribution Binary-Merge Sort**
  - Parallelism at all levels in the pipeline hierarchy
  - Step 1: local sort
  - Step 2: redistribute the results of local sort
  - Step 3: merge using the same pool of processors
  - Benefit: merging becomes lighter than without redistribution
  - Problem: height of the tree

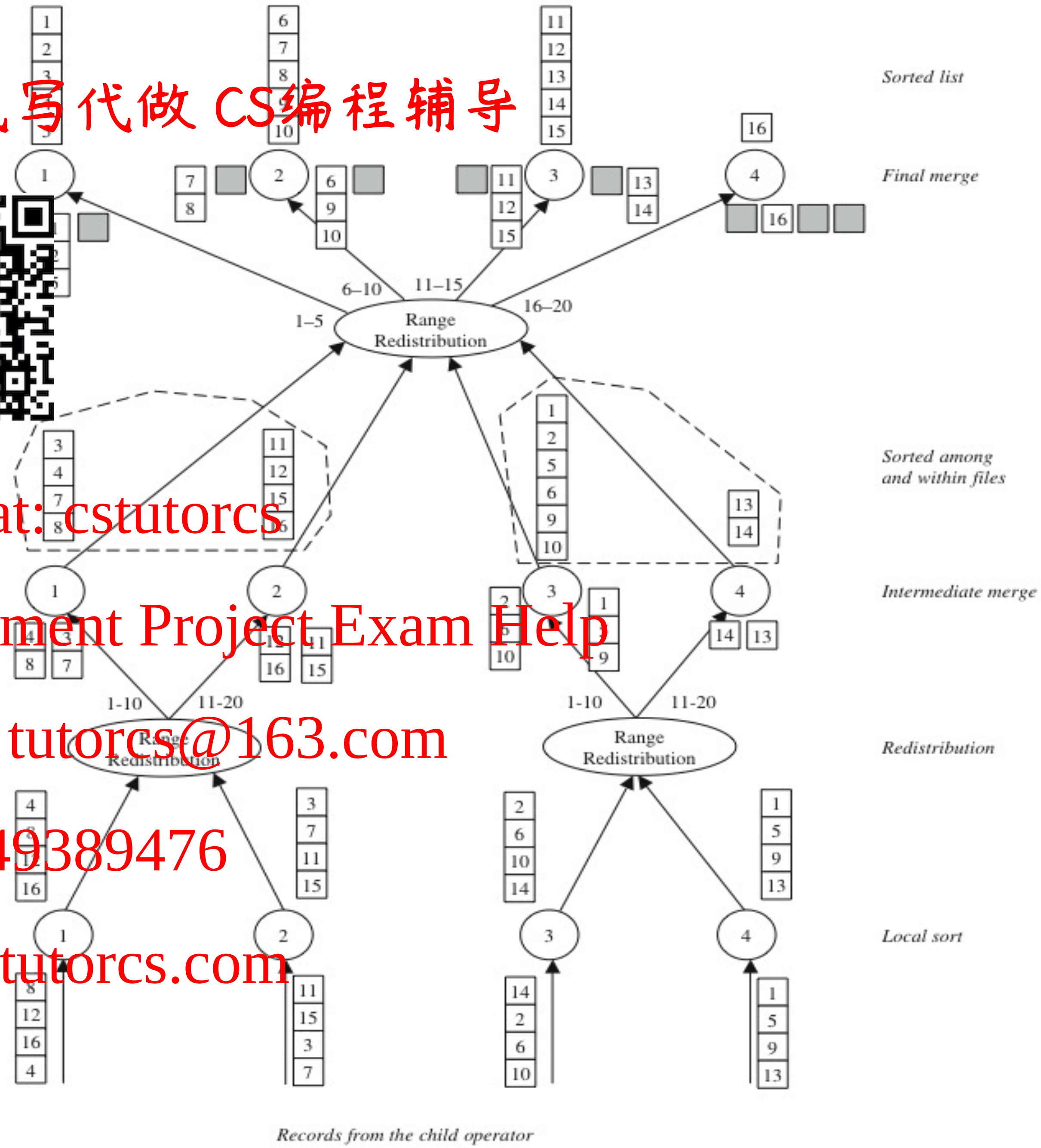


Figure 4.6 Parallel redistribution binary-merge sort

- **Parallel Redistribution Merge-All Sort**
  - Reduce the height of the tr
    and still maintain parallelism
  - Like parallel merge-all sort, but
    with redistribution
  - The advantage is true parallelism
    in merging
  - Skew problem in the merging

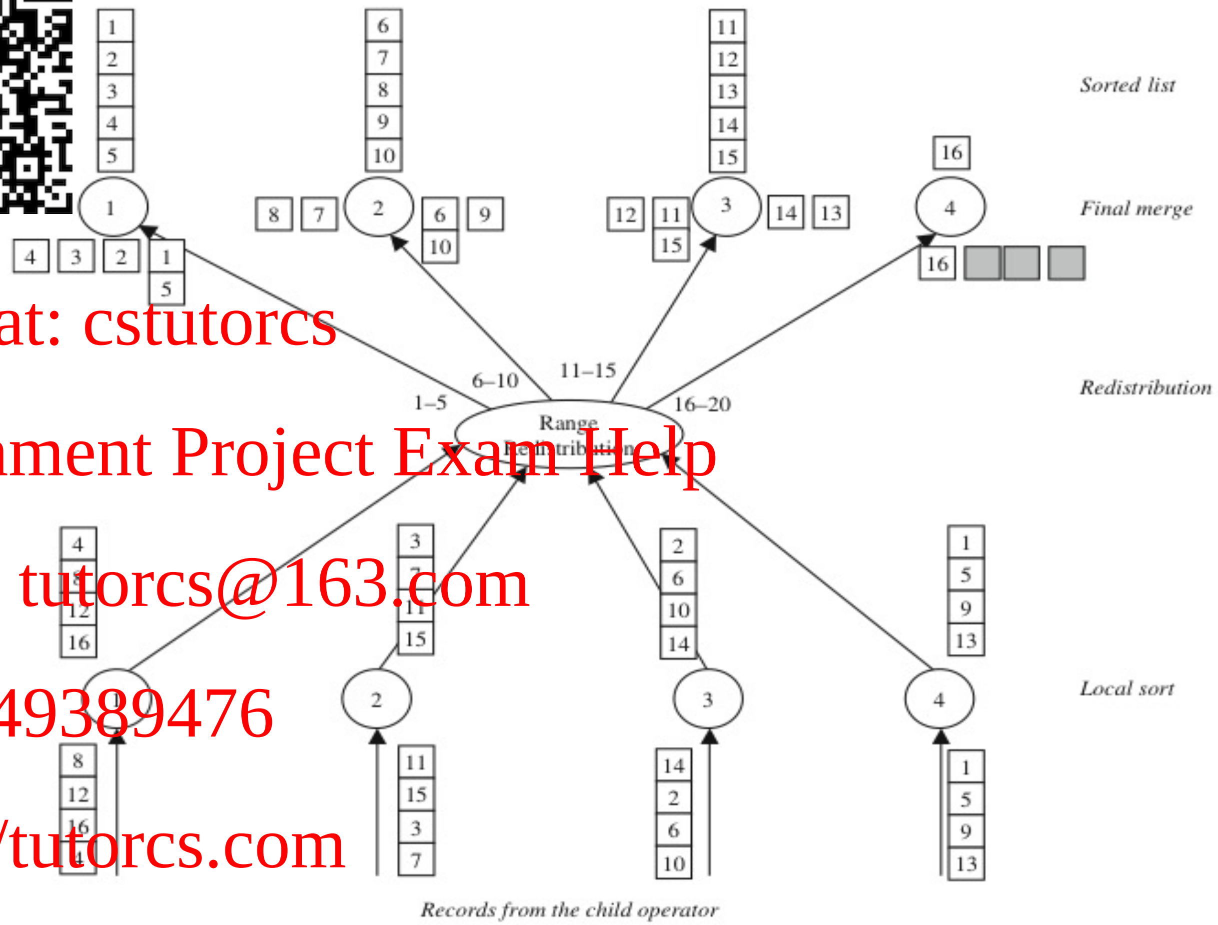


Figure 4.7 Parallel redistribution merge-all sort

- **Parallel Partitioned Sort**
  - Two stages: Partitioning st and Independent local wor
  - Partitioning (or range redistribution) may raise load skew
  - Local sort is done after the partitioning, not before
  - No merging is necessary
  - Main problem: **Skew** produced by the partitioning

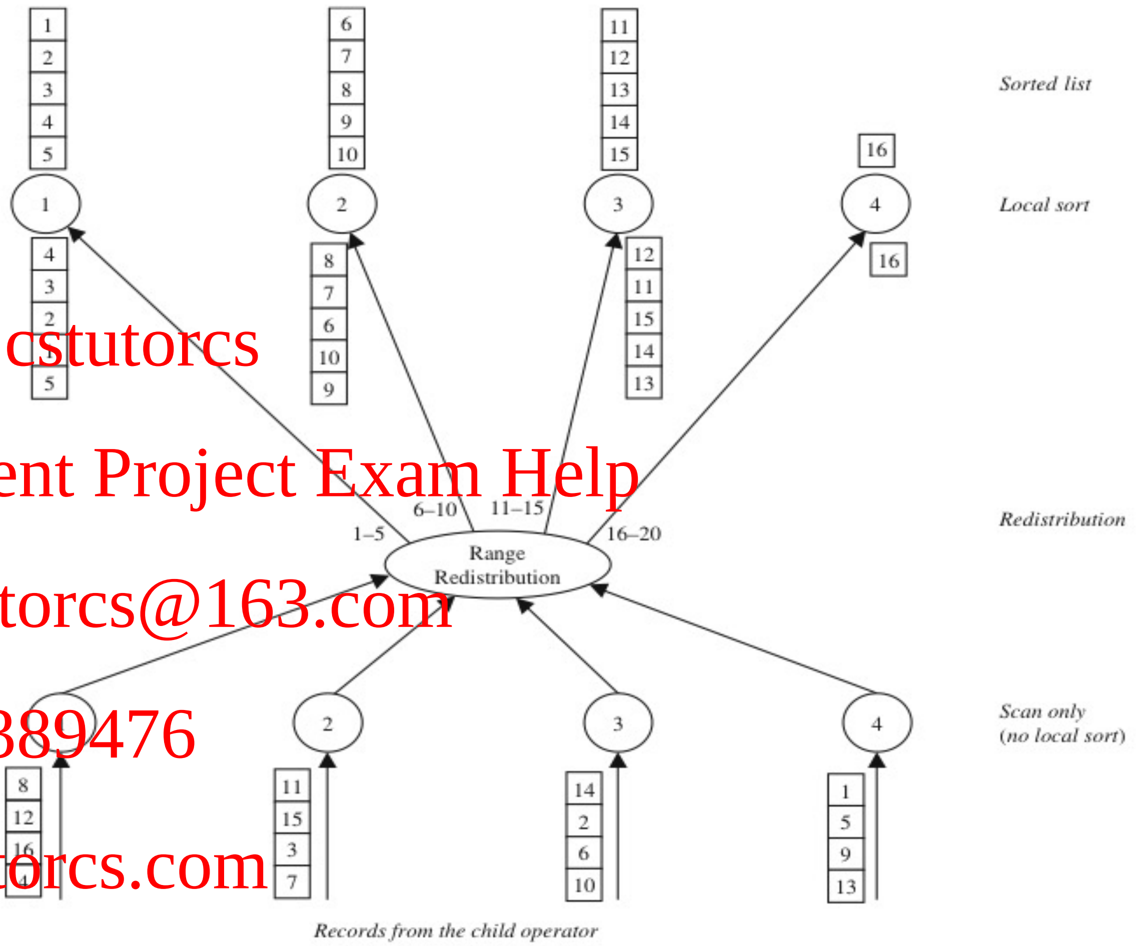


Figure 4.8 Parallel partitioned sort

- **Exercise 4 (Home Wor** [illegible]
  - Given a data set D = {55 [illegible]; 4; 49; 90; 34; 76; 82; 56; 31; 25; 78; 56; 38; 32; 88; 9; 44; 98; 11; 70; 66; 89; [illegible]6} and four processors, show step by step how the **Parallel Partitioned Sort** works.

4.2. Serial External Sorting (cont'd)

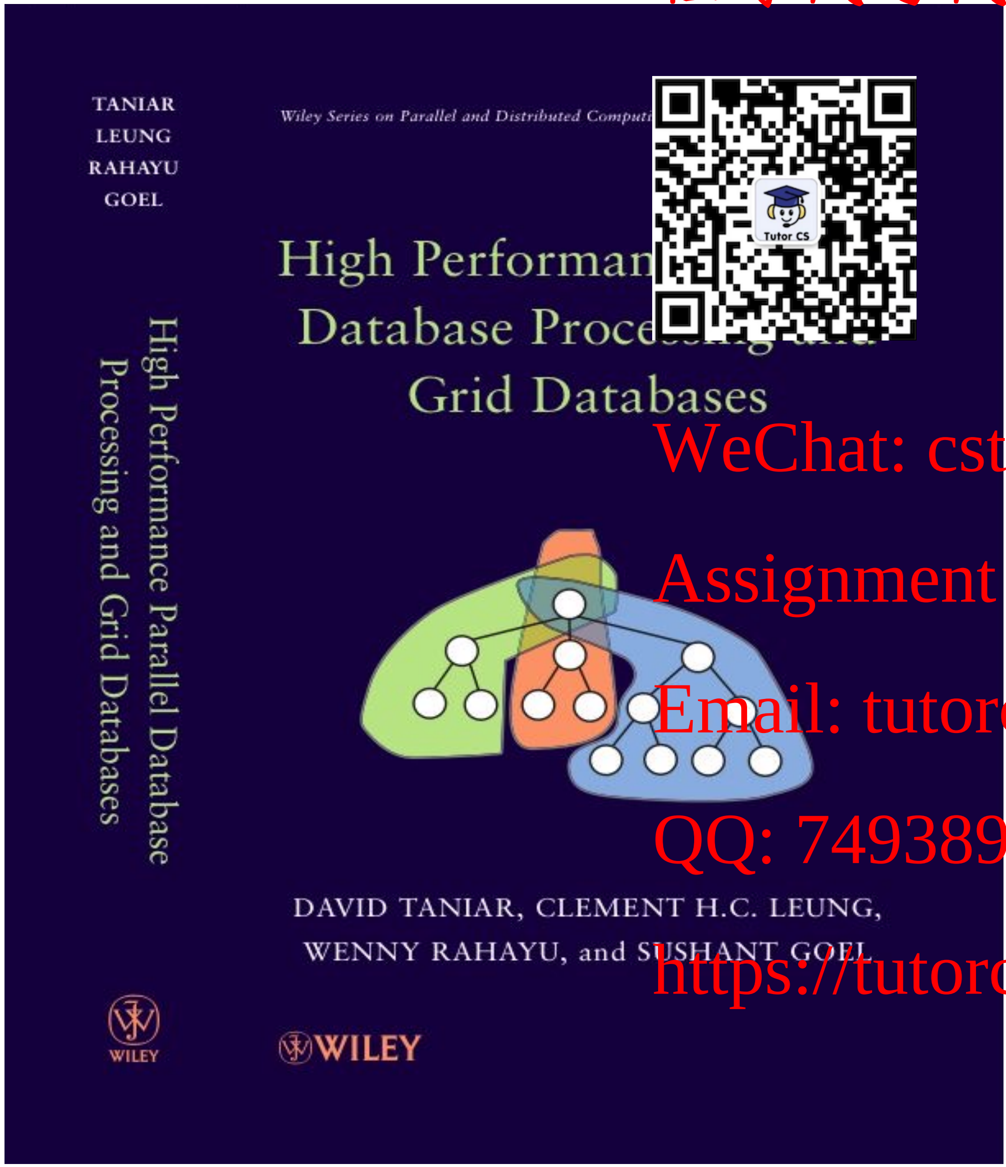
- **Exercise 5 (Difficult)**
  - Given the same dataset a… …ious question, and 4 processors, show how **Load Balancing** is achieved in… …**Partitioned Sort**.

MONASH University

Information Technology

FIT5202 (Volume IV – Sort and Group By)

Week 4b – Parallel Group By



Chapter 4 Parallel Sort and GroupBy

4.1. GroupBy and Serial GroupBy

Select Suburb, Count(*)
From Student
Group By Suburb;

Student	Suburb
Ad	Clayton
Be	Hawthorn
Chris	Doncaster
Daniel	Caulfield
Eric	Kew
Fred	Richmond
Garry	Hawthorn
Harold	Elwood
Irene	Clayton
Jessica	Caulfield
Katie	Malvern
Leonard	Balwyn
Mary	Hawthorn

4.1. Serial GroupBy Processing (cont'd)

Select Suburb, Count(*)
From Student
Group By Suburb;

Processing Steps:

1. Read the first student record, and hash the suburb to the hash table

Student	Suburb
Ad[illegible]	Clayton
Be[illegible]	Hawthorn
Chris	Doncaster
Daniel	Caulfield
Eric	Kew
Fred	Richmond
Garry	Hawthorn
Harold	Elwood
Irene	Clayton
Jessica	Caulfield
Katie	Malvern
Leonard	Balwyn
Mary	Hawthorn

Hash the record using a certain hash function

Hash Table

1		
2		
3		
4		
5		
6		
7		
8	Clayton	1
9		

4.1. Serial GroupBy Processing (cont'd)

Select Suburb, Count(*)
From Student
Group By Suburb;

Processing Steps:

1. Read the first student record, and hash the suburb to the hash table
2. Read the second record and hash it

Student	Suburb
Ad[illegible]	Clayton
Be[illegible]	Hawthorn
Chris	Doncaster
Daniel	Caulfield
Eric	Kew
Fred	Richmond
Garry	Hawthorn
Harold	Elwood
Irene	Clayton
Jessica	Caulfield
Katie	Malvern
Leonard	Balwyn
Mary	Hawthorn

Hash Table

1	Hawthorn	1
2		
3		
4		
5		
6		
7		
8	Clayton	1
9		

4.1. Serial GroupBy Processing (cont'd)

Select Suburb, Count(*)
From Student
Group By Suburb;

Processing Steps:

1. Read the first student record, and hash the suburb to the hash table
2. Read the second record and hash it
3. Read the subsequent records one-by-one and hash them

Student	Suburb
Ad	Clayton
Be	Hawthorn
Chris	Doncaster
Daniel	Caulfield
Eric	Kew
Fred	Richmond
Garry	Hawthorn
Harold	Elwood
Irene	Clayton
Jessica	Caulfield
Katie	Malvern
Leonard	Balwyn
Mary	Hawthorn

Hash Table

1	Hawthorn	3
2	Caulfield	2
3	Malvern	1
4	Balwyn	1
5	Kew	1
6	Richmond	1
7	Elwood	1
8	Clayton	2
9	Doncaster	1

4.1. Serial GroupBy Processing (cont'd)

Select Suburb, Count(*)
From Student
Group By Suburb;

Processing Steps:

1. Read the first student record, and hash the suburb to the hash table
2. Read the second record and hash it
3. Read the subsequent records one-by-one and hash them
4. Read the Hash Table, and store this in disk as the query results

Results in Disk

Hawthorn	3
Caulfield	2
Malvern	1
Balwyn	1
Kew	1
Richmond	1
Elwood	1
Clayton	2
Doncaster	1

Hash Table in Main-Memory

1	Hawthorn	3
2	Caulfield	2
3	Malvern	1
4	Balwyn	1
5	Kew	1
6	Richmond	1
7	Elwood	1
8	Clayton	2
9	Doncaster	1

Select Suburb, Count(*)
From Student
Group By Suburb;

St	uburb
Ad	layton
Be	awthorn
Leonard	Balwyn
Mary	Hawthorn

This will work, if we assume that the main-memory can hold the entire Hash Table.

How about if the Hash Table is so big that it cannot fit into the main-memory.

For example, how about if the main-memory can only hold 4 hash records at a time? How does the Group By processing work?

Hash Table

1	Hawthorn	3
2	Caulfield	2
3	Malvern	1
4	Balwyn	1
5	Kew	1
6	Richmond	1
7	Elwood	1
8	Clayton	2
9	Doncaster	1

4.1. Serial GroupBy Processing (cont'd)

Student	Suburb
Adam	Clayton
Ben	Hawthorn
Chris	Doncaster
Daniel	Caulfield
Eric	Kew
Fred	Richmond
Garry	Hawthorn
Harold	Elwood
Irene	Clayton
Jessica	Caulfield
Katie	Malvern
Leonard	Balwyn
Mary	Hawthorn

Hash Table

Assume that the main-memory can hold 4 records in the hash table.

It needs a bigger hash table, but it doesn't have.

Student	Suburb
Adam	Clayton
Ben	Hawthorn
Chris	Doncaster
Daniel	Caulfield
Eric	Kew
Fred	Richmond
Garry	Hawthorn
Harold	Elwood
Irene	Clayton
Jessica	Caulfield
Katie	Malvern
Leonard	Balwyn
Mary	Hawthorn

based on the **Suburb**

In Main-Memory

Ben	Hawthorn
Daniel	Caulfield
Garry	Hawthorn
Jessica	Caulfield
Mary	Hawthorn

In Disk

Eric	Kew
Fred	Richmond
Katie	Malvern
Leonard	Balwyn

Adam	Clayton
Chris	Doncaster
Harold	Elwood
Irene	Clayton

Hash Processing

In Main-Memory

Ben	Hawthorn
Daniel	Caulfield
Garry	Hawthorn
Jessica	Caulfield
Mary	Hawthorn

Still in Disk

Eric	Kew
Fred	Richmond
Katie	Malvern
Leonard	Balwyn

Adam	Clayton
Chris	Doncaster
Harold	Elwood
Irene	Clayton

Hash Table in Main-Memory

Hawthorn	3
Caulfield	2

Query Results in Disk

Hawthorn	3
Caulfield	2

Hash Processing

Ben	Hawthorn
Daniel	Caulfield
Garry	Hawthorn
Jessica	Caulfield
Mary	Hawthorn

Flush to Disk

Eric	Kew
Fred	Richmond
Katie	Malvern
Leonard	Balwyn

Load to Main-Memory

Adam	Clayton
Chris	Doncaster
Harold	Elwood
Irene	Clayton

Still in Disk

Hash Table in Main-Memory

Kew	1
Malvern	1
Richmond	1
Balwyn	1

Query Results in Disk

Hawthorn	3
Caulfield	2
Kew	1
Caulfield	1
Richmond	1
Balwyn	1

Hash Processing

Ben	Hawthorn
Daniel	Caulfield
Garry	Hawthorn
Jessica	Caulfield
Mary	Hawthorn

Flush to Disk

Eric	Kew
Fred	Richmond
Katie	Malvern
Leonard	Balwyn

Flush to Disk

Adam	Clayton
Chris	Doncaster
Harold	Elwood
Irene	Clayton

Load to Main-Memory

Hash Table in Main-Memory

Clayton	2
Doncaster	1
Elwood	1

Query Results in Disk

Hawthorn	3
Caulfield	2
Kew	1
Caulfield	1
Richmond	1
Balwyn	1
Clayton	2
Doncaster	1
Elwood	1

4.4. Parallel GroupBy

- Traditional methods ([illegible] and Hierarchical Merging)
- Two-phase method
- Redistribution method

Without data redistribution

With data redistribution

4.4. Parallel GroupBy (cont'd)

- **Traditional Meth**
  - Step 1: local aggregate in each processor
  - Step 2: global aggregation
  - May use a Merge-All or Hierarchical method
  - Need to pay a special attention to some aggregate functions (AVG) when performing a local aggregate process

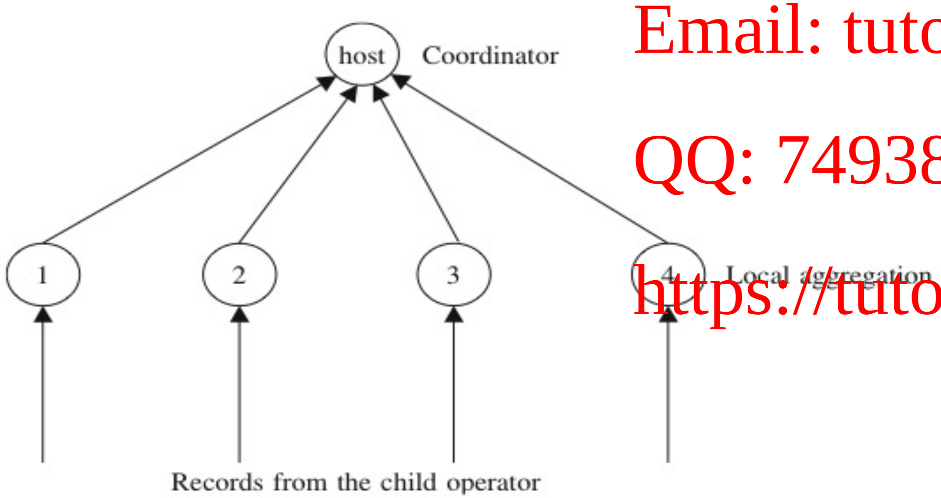


Figure 4.10 Traditional method

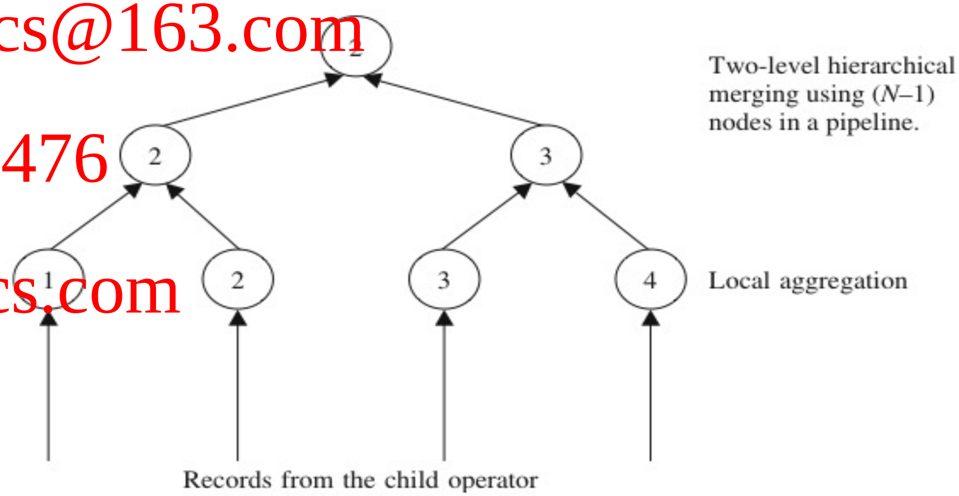
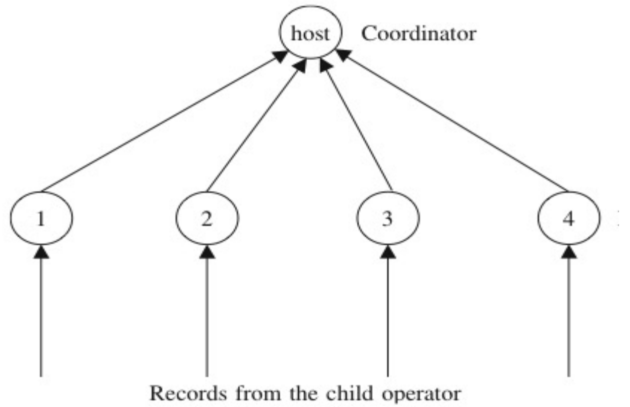


Figure 4.11 Hierarchical merging method

4.4. Parallel GroupBy (cont'd)

- **Traditional Method: Merge All**



Initial Data Placement

Processor 1

Adam	Clayton
Ben	Clayton
Chris	Caulfield
Dennis	Malvern
Eric	Vermont

Processor 2

Fred	Hawthorn
George	Richmond
Harold	Elwood
Irene	Malvern
Jessica	Kew

Processor 3

Kelly	Balwyn
Lesley	Hawthorn
Megan	Kew
Naomi	Richmond
Oscar	Vermont

Processor 4

Peter	Elwood
Quin	Kew
Roger	Balwyn
Sarah	Malvern
Tracy	Clayton

4.4. Parallel GroupBy (cont'd)

- **Traditional Method: Merge All**

Local Aggregation Phase

Processor 1 (local aggregation):

Clayton	2
Caulfield	1
Malvern	1
Vermont	1

Processor 2 (local aggregation):

Hawthorn	1
Richmond	[illegible]
Elwood	[illegible]
Malvern	[illegible]
Kew	1

Processor 3 (local aggregation):

Balwyn	1
Hawthorn	1
Kew	1
Richmond	1
Vermont	1

Processor 4 (local aggregation):

Elwood	1
Kew	1
Balwyn	1
Malvern	1
Clayton	1

Processor 1

Adam	Clayton
Ben	Clayton
Chris	Caulfield
Dennis	Malvern
Eric	Vermont

Processor 2

Fred	Hawthorn
George	Richmond
Harold	Elwood
Irene	Malvern
Jessica	Kew

Processor 3

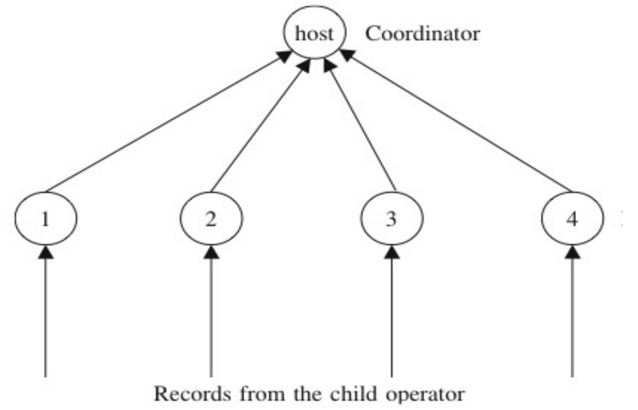
Kelly	Balwyn
Lesley	Hawthorn
Megan	Kew
Naomi	Richmond
Oscar	Vermont

Processor 4

Peter	Elwood
Quin	Kew
Roger	Balwyn
Sarah	Malvern
Tracy	Clayton

4.4. Parallel GroupBy (cont'd)

- **Traditional Method: Merge All**



Clayton	2
	1
	1
	1
…	…
…	…
Vermont	1
Clayton	1

Global Aggregation Phase

Processor 1

Clayton	2
Caulfield	1
Malvern	1
Vermont	1

Processor 2

Hawthorn	1
Richmond	1
Elwood	1
Malvern	1
Kew	1

Processor 3

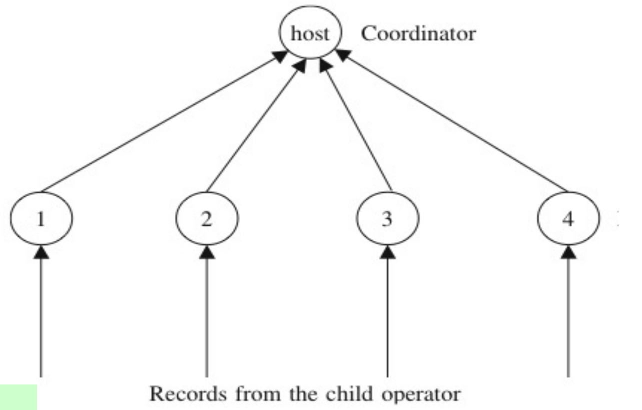
Balwyn	1
Hawthorn	1
Kew	1
Richmond	1
Vermont	1

Processor 4

Elwood	1
Kew	1
Balwyn	1
Malvern	1
Clayton	1

4.4. Parallel GroupBy (cont'd)

- **Traditional Method: Merge All**



Clayton	3
	2
	2
	…
…	…
Vermont	2

Final results

Clayton	2
Hawthorn	1
Balwyn	1
Elwood	1
…	…
…	…
Vermont	1
Clayton	1

Global Aggregation Phase

- **Exercise 6 (FLUX Quiz**
  - The limitations of the Tra oach (Merge All) to process a Group By query are:
  - A. Global aggregation is carried out by one processor
  - B. Network bottleneck when sending the local aggregation results to the coordinator
  - C. No parallelism in the global aggregation phase
  - D. All of the above
  - E. Some of the above

4.4. Parallel GroupBy (cont'd)

- **Two-Phase Meth**
  - Step 1: local aggregate in each processor. Each processor groups local records according to the groupby attribute
  - Step 2: global aggregation where all temp results from each processor are redistributed and then final aggregate is performed in each processor

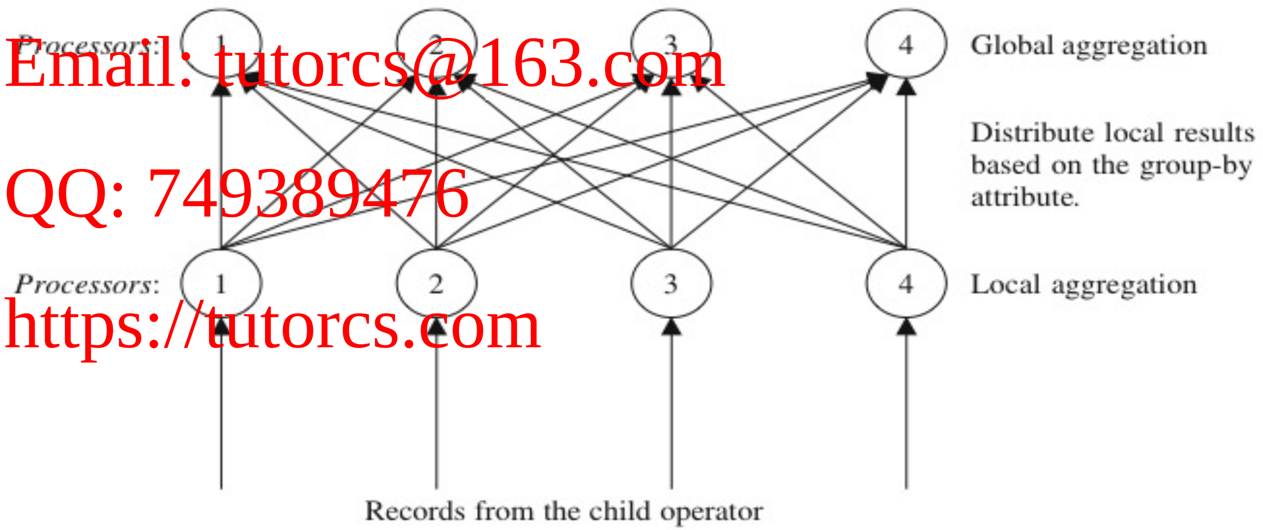
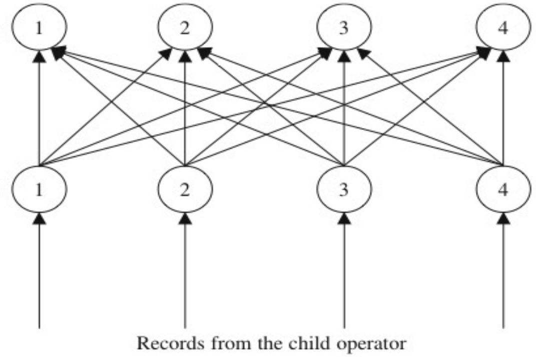


Figure 4.12 Two-phase method

4.4. Parallel GroupBy (cont'd)

- **Two-Phase Method**



Initial Data Placement

Processor 1

Adam	Clayton
Ben	Clayton
Chris	Caulfield
Dennis	Malvern
Eric	Vermont

Processor 2

Fred	Hawthorn
George	Richmond
Harold	Elwood
Irene	Malvern
Jessica	Kew

Processor 3

Kelly	Balwyn
Lesley	Hawthorn
Megan	Kew
Naomi	Richmond
Oscar	Vermont

Processor 4

Peter	Elwood
Quin	Kew
Roger	Balwyn
Sarah	Malvern
Tracy	Clayton

4.4. Parallel GroupBy (cont'd)

- **Two-Phase Method**

Clayton	2
Caulfield	1
Malvern	1
Vermont	1

Hawthorn	1
Richmo[illegible]	[illegible]
Elwood	[illegible]
Malver[illegible]	[illegible]
Kew	1

Balwyn	1
Hawthorn	1
Kew	1
Richmond	1
Vermont	1

Elwood	1
Kew	1
Balwyn	1
Malvern	1
Clayton	1

Local Aggregation Phase

Processor 1

Adam	Clayton
Ben	Clayton
Chris	Caulfield
Dennis	Malvern
Eric	Vermont

Processor 2

Fred	Hawthorn
George	Richmond
Harold	Elwood
Irene	Malvern
Jessica	Kew

Processor 3

Kelly	Balwyn
Lesley	Hawthorn
Megan	Kew
Naomi	Richmond
Oscar	Vermont

Processor 4

Peter	Elwood
Quin	Kew
Roger	Balwyn
Sarah	Malvern
Tracy	Clayton

4.4. Parallel GroupBy (cont'd)

- **Two-Phase Method**

Clayton	2
Balwyn	1
Elwood	1
Caulfield	1
Elwood	1
Balwyn	1
Clayton	1

Hawthorn	1
Hawthorn	[illegible]
Kew	[illegible]
Kew	[illegible]
Kew	1

Malvern	1
Malvern	1
Malvern	1

Richmond	1
Vermont	1
Richmond	1
Vermont	1

Distribute Local Aggregation Results Phase

Processor 1

Clayton	2
Caulfield	1
Malvern	1
Vermont	1

Processor 2

Hawthorn	1
Richmond	1
Elwood	1
Malvern	1
Kew	1

Processor 3

Balwyn	1
Hawthorn	1
Kew	1
Richmond	1
Vermont	1

Processor 4

Elwood	1
Kew	1
Balwyn	1
Malvern	1
Clayton	1

4.4. Parallel GroupBy (cont'd)

- **Two-Phase Method**

Final results

Clayton	3
Balwyn	2
Elwood	2
Caulfield	1

Hawtho[illegible]	[illegible]
Kew	[illegible]

Malvern	3

Richmond	2
Vermont	2

Global Aggregation Phase

Processor 1

Clayton	2
Balwyn	1
Elwood	1
Caulfield	1
Elwood	1
Balwyn	1
Clayton	1

Processor 2

Hawthorn	1
Hawthorn	1
Kew	1
Kew	1
Kew	1

Processor 3

Malvern	1
Malvern	1
Malvern	1

Processor 4

Richmond	1
Vermont	1
Richmond	1
Vermont	1

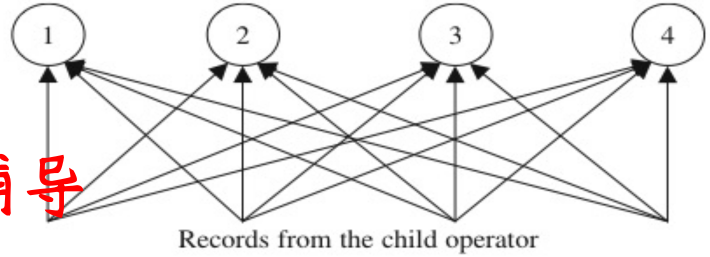
4.4. Parallel GroupBy (cont'd)

- **Redistribution Method**
  - Step 1 (Partitioning phase): redistribute raw records to all processors
  - Step 2 (Aggregation phase): each processor performs a local aggregation

Figure 4.13 Redistribution method

4.4. Parallel GroupBy (cont'd)

- **Redistribution Method**



Initial Data Placement

Processor 1

Adam	Clayton
Ben	Clayton
Chris	Caulfield
Dennis	Malvern
Eric	Vermont

Processor 2

Fred	Hawthorn
George	Richmond
Harold	Elwood
Irene	Malvern
Jessica	Kew

Processor 3

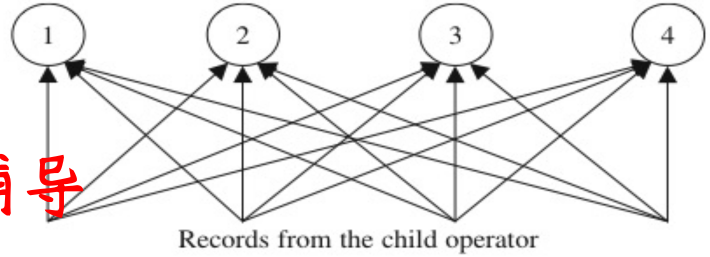
Kelly	Balwyn
Lesley	Hawthorn
Megan	Kew
Naomi	Richmond
Oscar	Vermont

Processor 4

Peter	Elwood
Quin	Kew
Roger	Balwyn
Sarah	Malvern
Tracy	Clayton

4.4. Parallel GroupBy (cont'd)

- **Redistribution Method**



Partitioning Phase

Processor 1

Adam	Clayton
Kelly	Balwyn
Peter	Elwood
Ben	Clayton
Chris	Caulfield
Harold	Elwood
Roger	Balwyn
Tracy	Clayton

Processor 2

Fred	[illegible]
Lesley	Hawthorn
Quin	Kew
Megan	Kew
Jessica	Kew

Processor 3

Dennis	Malvern
Irene	Malvern
Sarah	Malvern

Processor 4

George	Richmond
Naomi	Richmond
Eric	Vermont
Oscar	Vermont

Processor 1 (after redistribution)

Adam	Clayton
Ben	Clayton
Chris	Caulfield
Dennis	Malvern
Eric	Vermont

Processor 2 (after redistribution)

Fred	Hawthorn
George	Richmond
Harold	Elwood
Irene	Malvern
Jessica	Kew

Processor 3 (after redistribution)

Kelly	Balwyn
Lesley	Hawthorn
Megan	Kew
Naomi	Richmond
Oscar	Vermont

Processor 4 (after redistribution)

Peter	Elwood
Quin	Kew
Roger	Balwyn
Sarah	Malvern
Tracy	Clayton

4.4. Parallel GroupBy (cont'd)

- **Redistribution Method**

Final results

Clayton	3
Balwyn	2
Elwood	2
Caulfield	1

Hawthorn	[illegible]
Kew	[illegible]

Malvern	3

Richmond	2
Vermont	2

Aggregation Phase

Adam	Clayton
Kelly	Balwyn
Peter	Elwood
Ben	Clayton
Chris	Caulfield
Harold	Elwood
Roger	Balwyn
Tracy	Clayton

Processor 1

Fred	Hawthorn
Lesley	Hawthorn
Quin	Kew
Megan	Kew
Jessica	Kew

Processor 2

Dennis	Malvern
Irene	Malvern
Sarah	Malvern

Processor 3

George	Richmond
Naomi	Richmond
Eric	Vermont
Oscar	Vermont

Processor 4

4.4. Parallel GroupBy (cont'd)

- **Redistribution Method**

Clayton	3
Balwyn	2
Elwood	2
Caulfield	1

Hawthorn	[illegible]
Kew	[illegible]

Malvern	3

Richmond	2
Vermont	2

What is the problem here?

Processor 1

Adam	Clayton
Kelly	Balwyn
Peter	Elwood
Ben	Clayton
Chris	Caulfield
Harold	Elwood
Roger	Balwyn
Tracy	Clayton

Processor 2

Fred	Hawthorn
Lesley	Hawthorn
Quin	Kew
Megan	Kew
Jessica	Kew

Processor 3

Dennis	Malvern
Irene	Malvern
Sarah	Malvern

Processor 4

George	Richmond
Naomi	Richmond
Eric	Vermont
Oscar	Vermont

4.4. Parallel GroupBy (cont'd)

- **Redistribution Method (Task Stealing)**

Create 5 buckets, instead of 4

Adam	Clayton
Kelly	Balwyn
Ben	Clayton
Chris	Caulfield
Roger	Balwyn
Tracy	Clayton

Peter	Elwood
Harold	Elwood

Fred	[illegible]
Lesley	Hawthorn
Quin	Kew
Megan	Kew
Jessica	Kew

Dennis	Malvern
Irene	Malvern
Sarah	Malvern

George	Richmond
Naomi	Richmond
Eric	Vermont
Oscar	Vermont

Processor 1 **Processor 2** **Processor 3** **Processor 4**

Adam	Clayton
Ben	Clayton
Chris	Caulfield
Dennis	Malvern
Eric	Vermont

Fred	Hawthorn
George	Richmond
Harold	Elwood
Irene	Malvern
Jessica	Kew

Kelly	Balwyn
Lesley	Hawthorn
Megan	Kew
Naomi	Richmond
Oscar	Vermont

Peter	Elwood
Quin	Kew
Roger	Balwyn
Sarah	Malvern
Tracy	Clayton

4.4. Parallel GroupBy (cont'd)

Redistribution Method (Task Stealing)

Processor 1

Adam	Clayton
Kelly	Balwyn
Ben	Clayton
Chris	Caulfield
Roger	Balwyn
Tracy	Clayton
Peter	Elwood
Harold	Elwood

Processor 2

Fred	Hawthorn
Lesley	Hawthorn
Quin	Kew
Megan	Kew
Jessica	Kew

Processor 3

Dennis	Malvern
Irene	Malvern
Sarah	Malvern

Task stealing

Peter	Elwood
Harold	Elwood

Processor 4

George	Richmond
Naomi	Richmond
Eric	Vermont
Oscar	Vermont

4.4. Parallel GroupBy (cont'd)

. Redistribution Method (Task Stealing)

Processor 1

Clayton	3
Balwyn	2
Caulfield	1

Adam	Clayton
Kelly	Balwyn
Ben	Clayton
Chris	Caulfield
Roger	Balwyn
Tracy	Clayton

Processor 2

Hawtho[illegible]	[illegible]
Kew	[illegible]

Fred	Hawthorn
Lesley	Hawthorn
Quin	Kew
Megan	Kew
Jessica	Kew

Processor 3

Malvern	3
Elwood	2

Dennis	Malvern
Irene	Malvern
Sarah	Malvern

Peter	Elwood
Harold	Elwood

Processor 4

Richmond	2
Vermont	2

George	Richmond
Naomi	Richmond
Eric	Vermont
Oscar	Vermont

Exercise 7 (FLUX Quiz

- The Redistribution Metho balancing option, through the Task Stealing method. The Two-Phase Method e a load balancing problem.

- A. TRUE
- B. FALSE

4.7. Summary

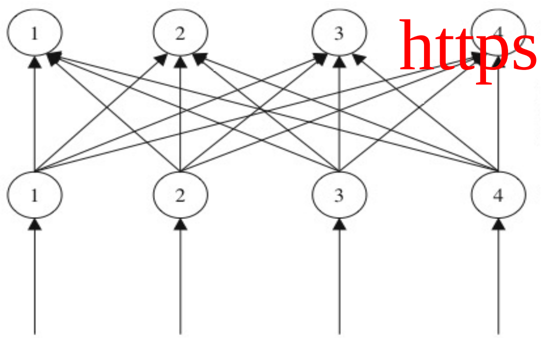
- Parallel groupby algorith
  - Traditional methods (me erarchical methods)
  - Two-phase method - – L ion before data redistribution
  - Redistribution method - – Local aggregation after data redistribution

- **Two-phase** and **Redistribution** methods perform better than the traditional and hierarchical merging methods

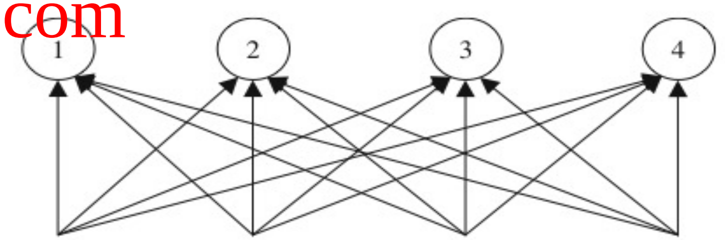
- **Two-phase method** works well when the number of groups is small, whereas the **Redistribution method** works well when the number of groups is large

Ambuj and Naughton. "Adaptive parallel aggregation algorithms." (1995):

Why??



Records from the child operator



Records from the child operator

Homework Exercises

- 1. Show how Load Balan Task Stealing be achieved in the Two Phase Method (using the same sample e) – EASY
- 2. Why is the Two-Phase Method good when the number of groups is small, whereas the Redistribution Method good when the number of groups is large? – MORE CHALLENGING
- 3. In what scenario may super linear speed up be achieved? – MORE CHALLENGING
- (Hints: See slides #8-#13 → the hash table cannot fit into main-memory)