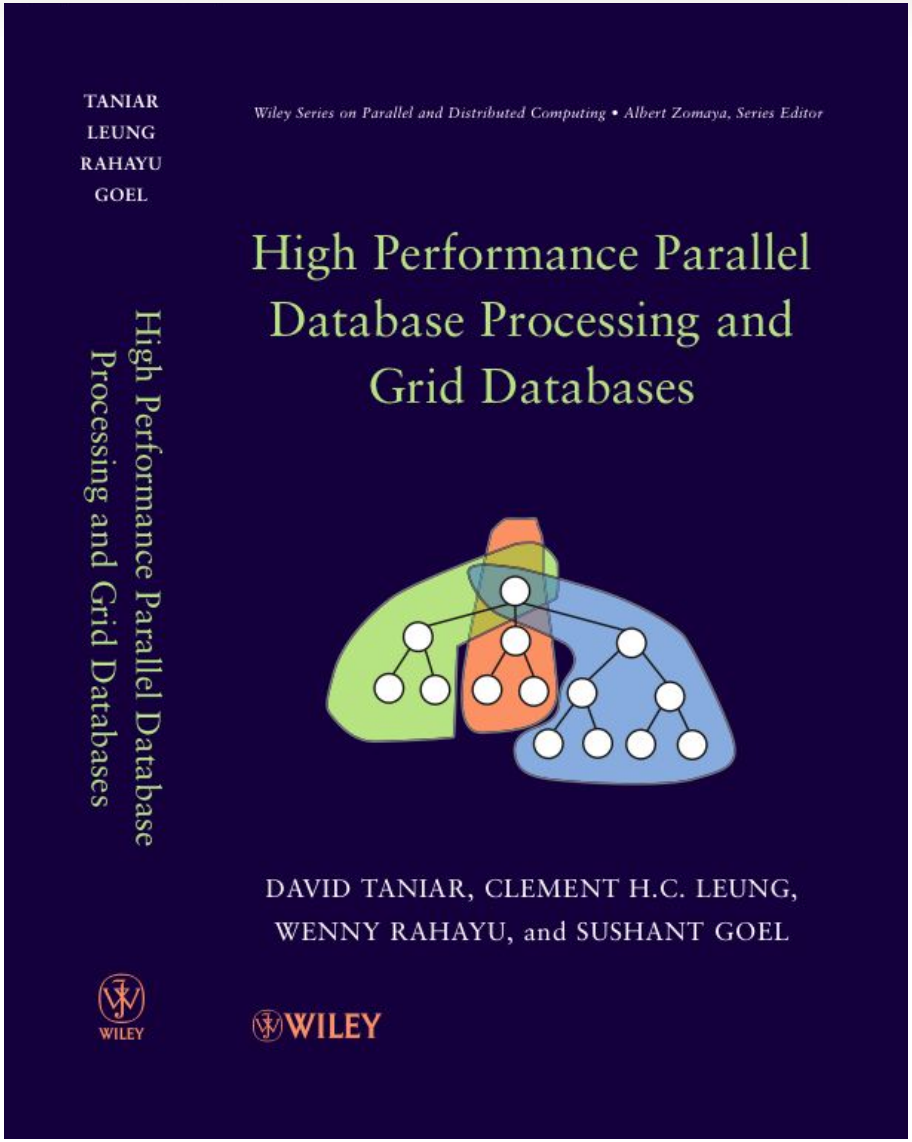

TANIAR
LEUNG
RAHAYU
GOEL
Wiley Series on Parallel and Distributed Computing • Albert Zomaya, Series Editor
High Performance Parallel Database Processing and Grid Databases
High Performance Parallel Database Processing and Grid Databases
DAVID TANIAR, CLEMENT H.C. LEUNG,
WENNY RAHAYU, and SUSHANT GOEL
WILEY
WILEY

Preface

- Exponential growth of data volume, steady drop in storage costs, and rapid increase in storage capacity
- Inadequacy of the sequential processing paradigm
  - Example: Assuming a data rate of 1 terabyte/sec, reading through a petabyte database will take over 10 days
- Parallel machines in the past, and parallel facilities in today's commercial DBMS
- **Importance of understanding high-performance and parallel database processing systems**
- Grid as global and distributed data centres
- New application domains (data-intensive applications): data warehousing and online analytic processing, and data mining

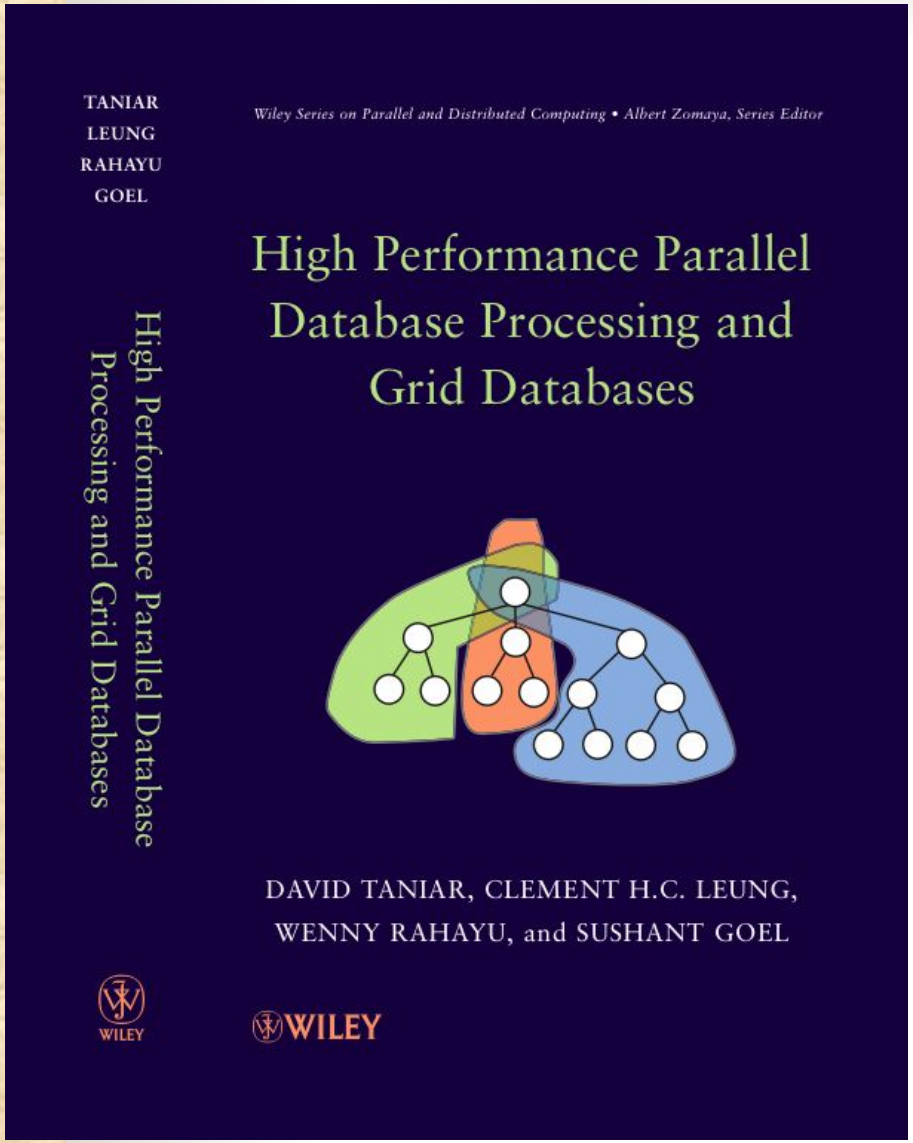
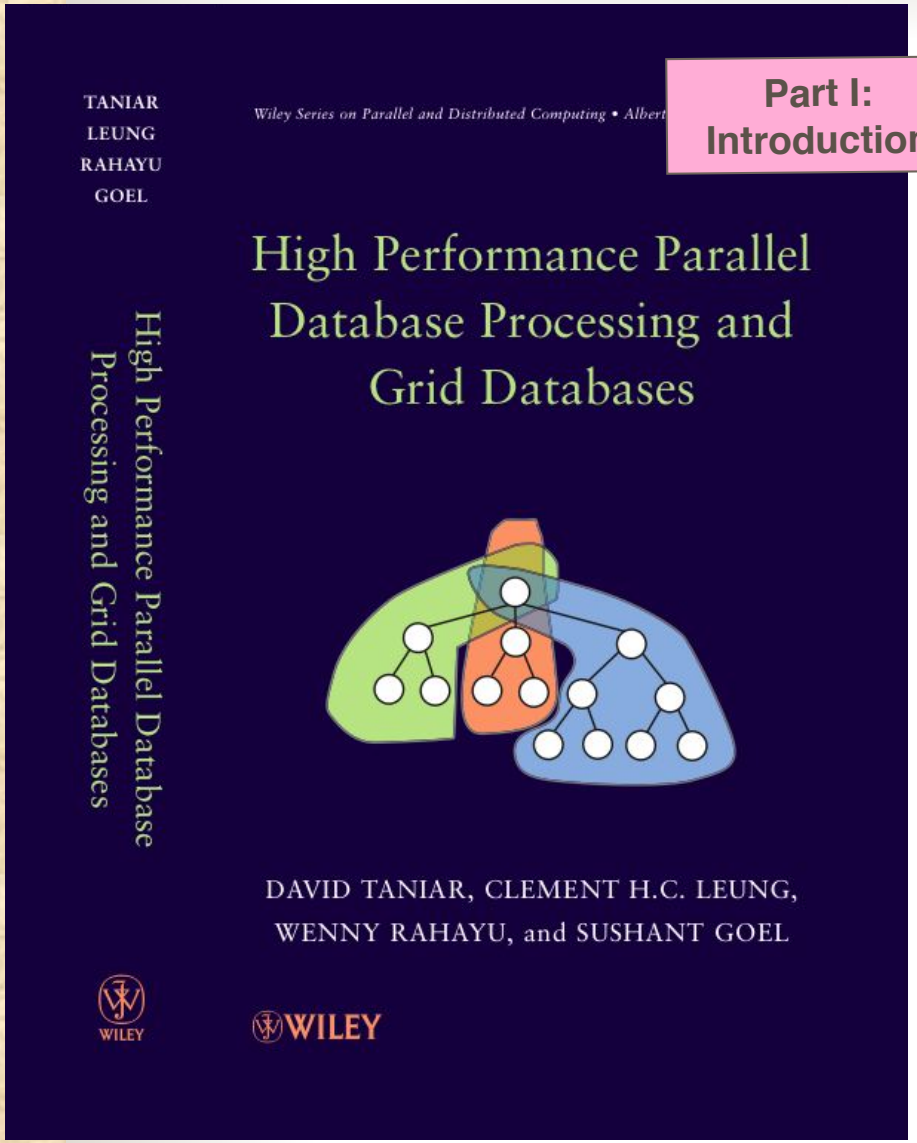
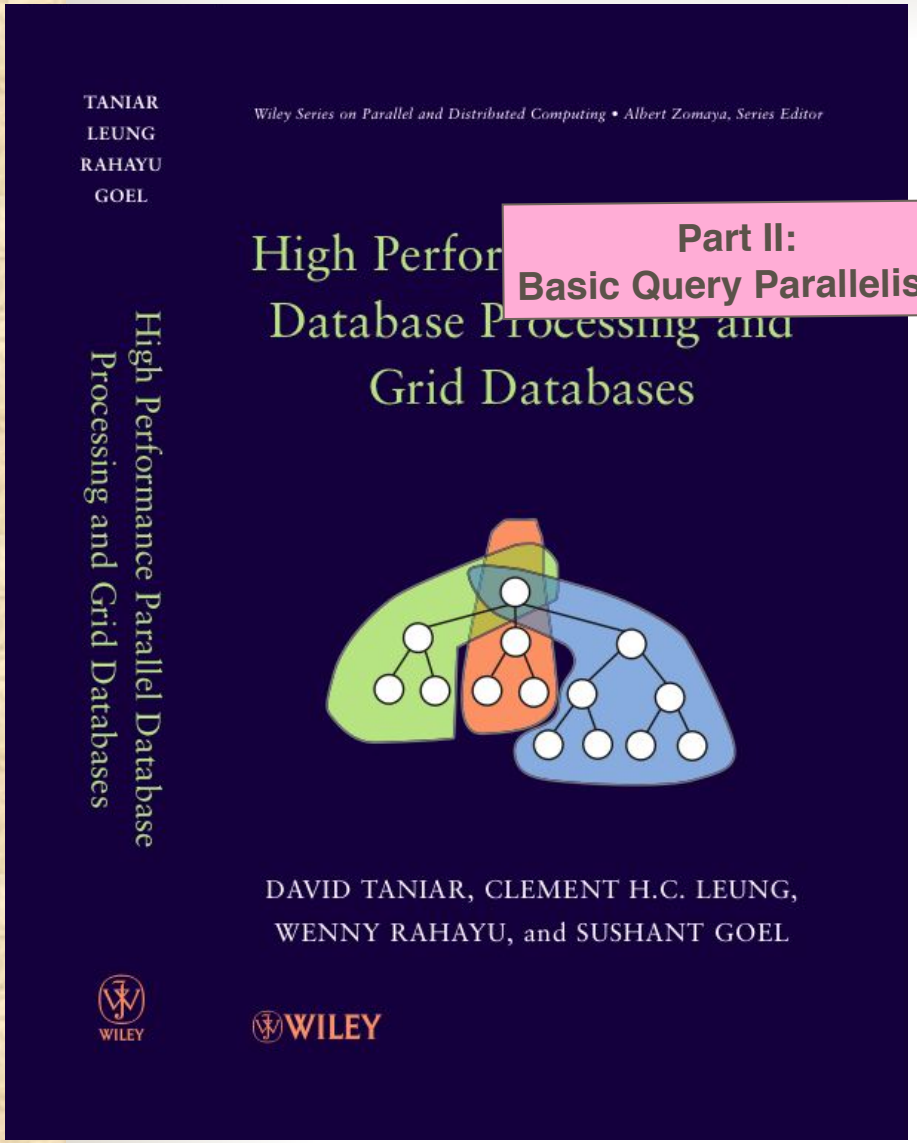


Table of Contents



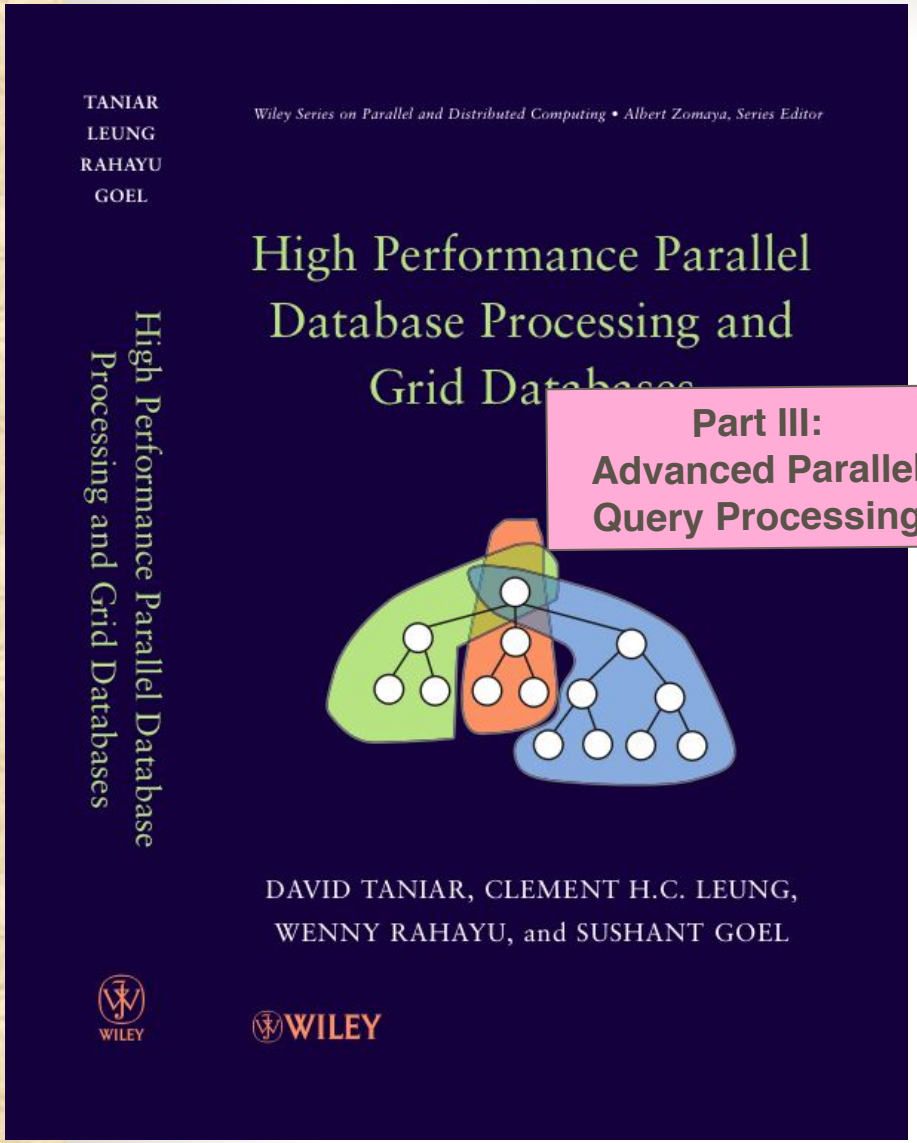
Part I: Introduction

Table of Contents



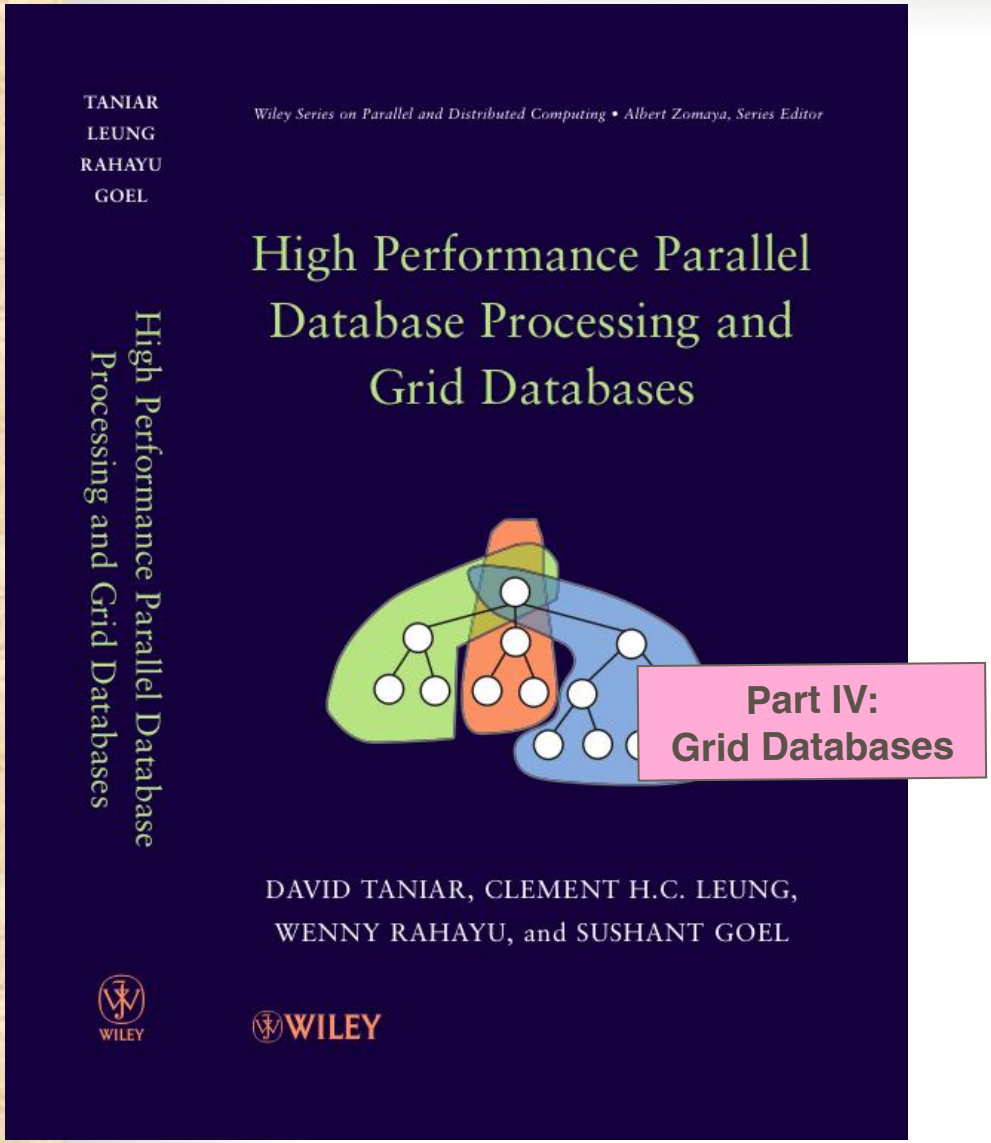
Part II: Basic Query Parallelism

Table of Contents



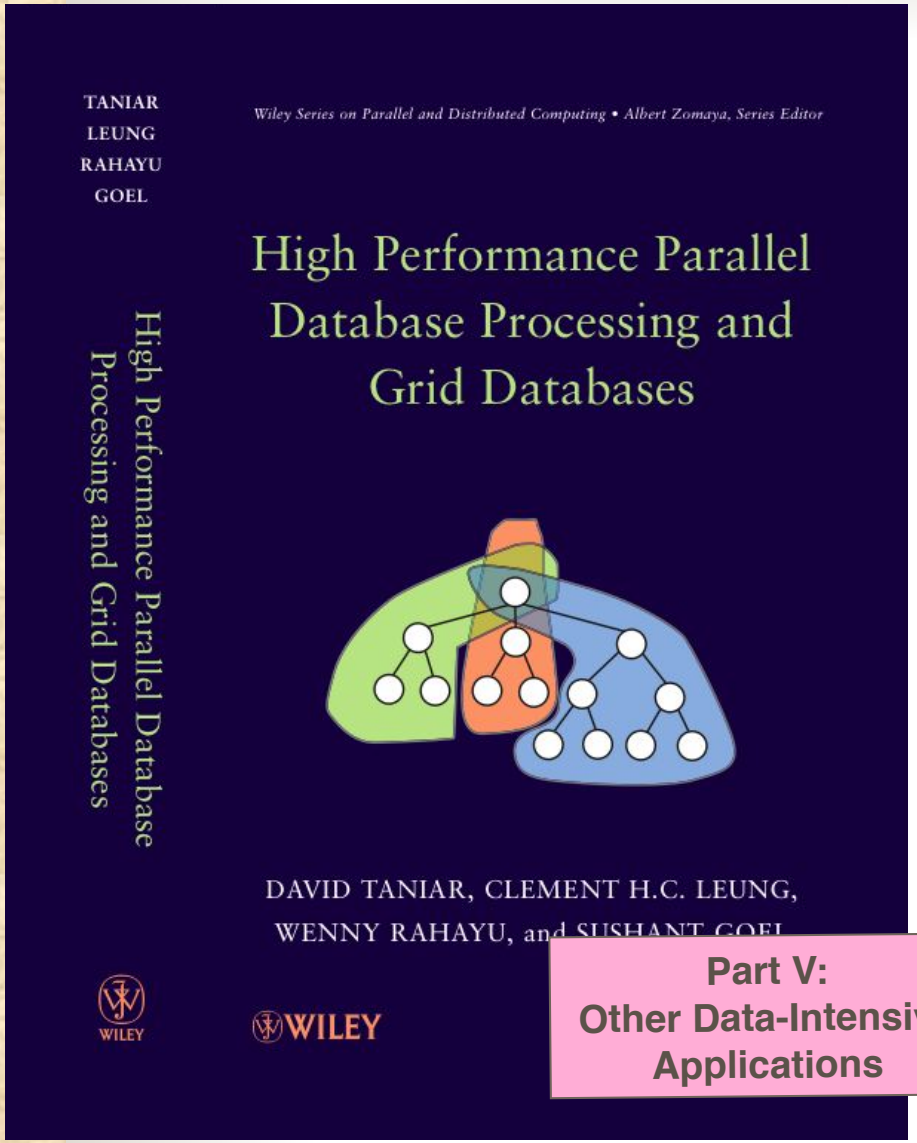
Part III: Advanced Parallel Query Processing

Table of Contents



Part IV: Grid Databases

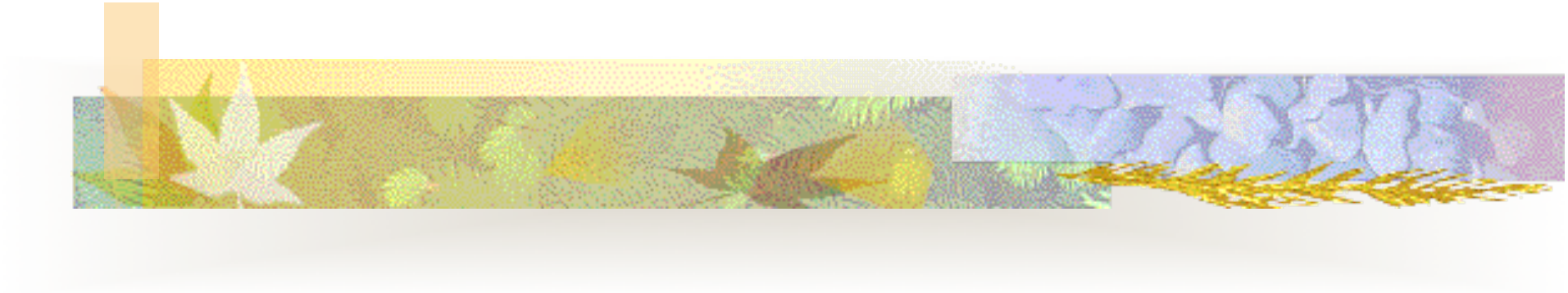
Table of Contents

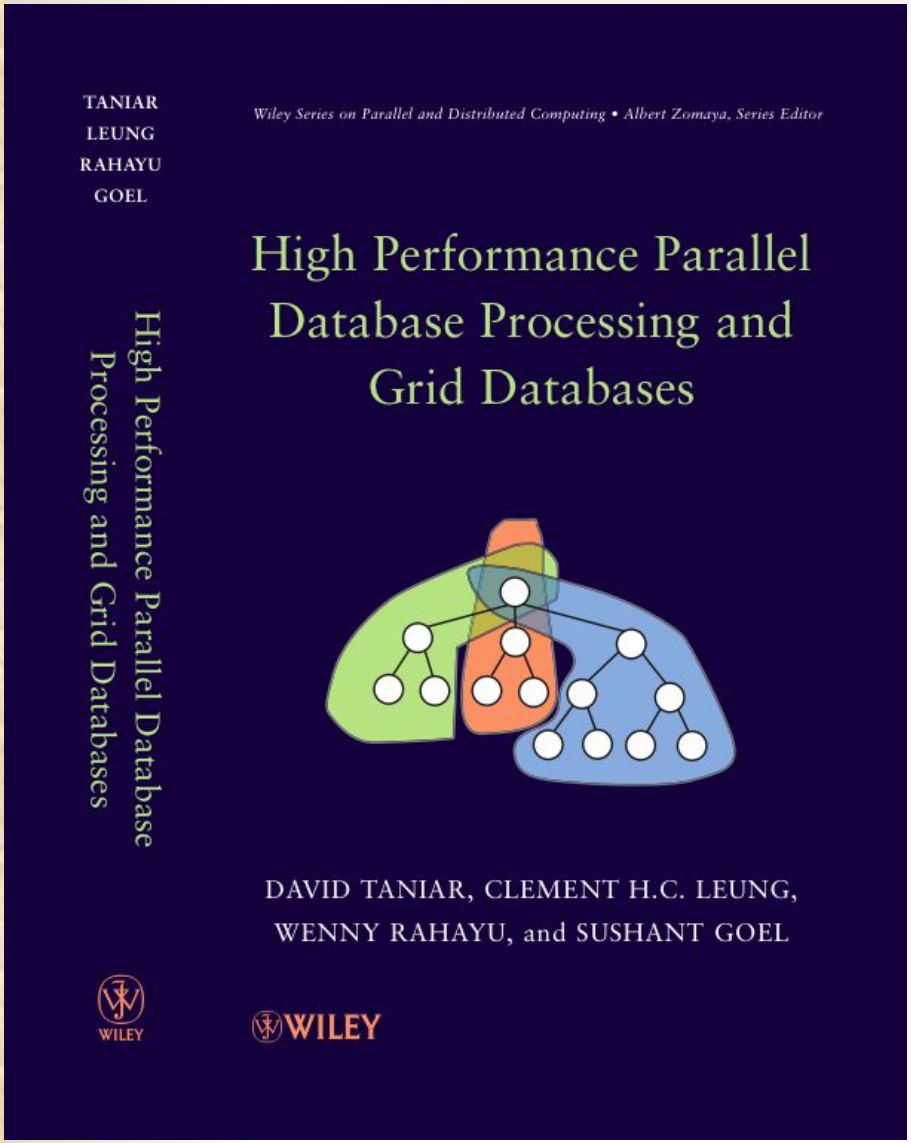


Part V: Other Data-Intensive Applications

Table of Contents

Let’s go to Chapter 1…





Chapter 1 Introduction

1.1. A Brief Overview

- Moore's Law: number of processors will double every 18-24 months
- CPU performance would increase by 50-60% per year
- Mechanical delays restrict the advancement of disk access time or disk throughput (8-10% only)
- Disk capacity also increases at a much higher rate
- I/O becomes a bottleneck
- Hence, motivates parallel database research

1.1. A Brief Overview (cont'd)

- Parallel Database Systems:
  - Single administrative domain
  - Homogeneous working environment
  - Close proximity of data storage
  - Multiple processors

- Grid Database Systems:
  - Heterogeneous collaboration of resources
  - Provide seamless access to geographically distributed data sources

1.2. Motivations

- An example:

$$
\begin{aligned}
10\ \text{TB} &= 10 \times 1024 \times 1024\ \text{MB} = 1{,}048{,}576\ \text{MB} \\
1{,}048{,}576\ \text{MB}/1\ \text{MB/sec} &\approx 1{,}048{,}576\ \text{seconds} \\
&\approx 174{,}760\ \text{minutes} \\
&\approx 2910\ \text{hours} \\
&\approx 120\ \text{days and nights}
\end{aligned}
$$

1.2. Motivations (cont'd)

- What is parallel processing, and why not just use a faster computer ?
  - Even fast computers have speed limitations
  - Limited by speed of light
  - Other hardware limitations
- Parallel processing divides a large task into smaller subtasks
- Database processing works well with parallelism (coarse-grained parallelism)
- Lesser complexity but need to work with a large volume of data

1.3. Objectives

- The primary objective of parallel database processing is to gain performance improvement
- Two main measures:
  - **Throughput**: the number of tasks that can be completed within a given time interval
  - **Response time**: the amount of time it takes to complete a single task from the time it is submitted
- Metrics:
  - Speed up
  - Scale up

1.3. Objectives

- The primary objective of parallel database processing is to gain performance improvement
- Two main measures:
  - Throughput: the number of tasks that can be completed within a given time interval
  - Response time: the amount of time it takes to complete a single task from the time it is submitted
- Metrics:
  - **Speed up**
  - **Scale up**

1.3. Objectives (cont'd)

- **Speed up**
  - Performance improvement gained because of extra processing elements added
  - Running a given task in less time by increasing the degree of parallelism

- Linear speed up: performance improvement growing linearly with additional resources
- Superlinear speed up
- Sublinear speed up

$$\text{Speed up} = \frac{\text{elapsed time on uniprocessor}}{\text{elapsed time on multiprocessors}}$$

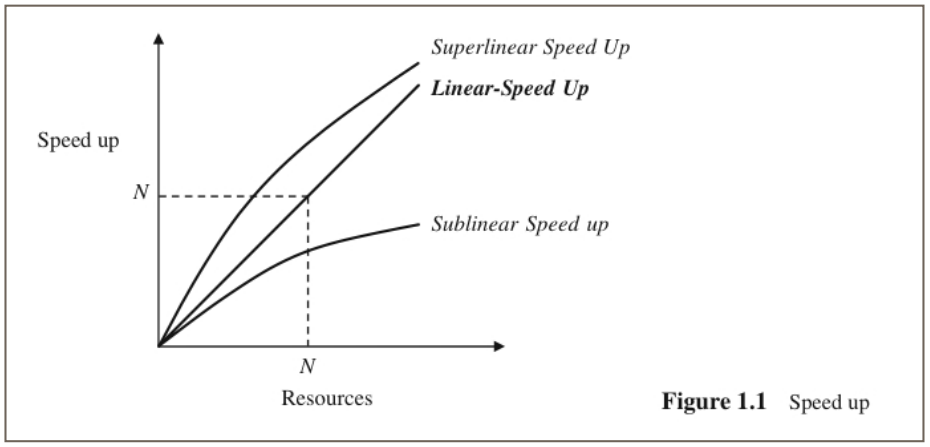


Figure 1.1 Speed up

1.3. Objectives (cont'd)

- **Scale up**
  - Handling of larger tasks by increasing the degree of parallelism
  - The ability to process larger tasks in the same amount of time by providing more resources.

- Linear scale up: the ability to maintain the same level of performance when both the workload and the resources are proportionally added
- Transactional scale up
- Data scale up

$$\text{Scale up} = \frac{\text{uniprocessor elapsed time on small system}}{\text{multiprocessor elapsed time on larger system}}$$

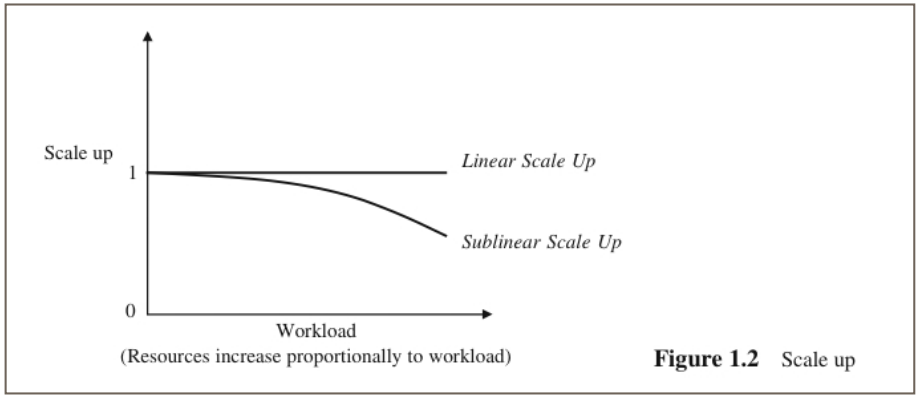


Figure 1.2 Scale up

1.3. Objectives (cont'd)

- **Transaction scale up**
  - The increase in the rate at which the transactions are processed
  - The size of the database may also increase proportionally to the transactions' arrival rate
  - *N*-times as many users are submitting *N*-times as many requests or transactions against an *N*-times larger database
  - Relevant to transaction processing systems where the transactions are small updates

- **Data scale up**
  - The increase in size of the database, and the task is a large job who runtime depends on the size of the database (e.g. sorting)
  - Typically found in online analytical processing (OLAP)

1.3. Objectives (cont'd)

- **Parallel Obstacles**
  - Start-up and Consolidation costs,
  - Interference and Communication, and
  - Skew

1.3. Objectives (cont'd)

- **Start-up and Consolidation**
  - Start up: initiation of multiple processes
  - Consolidation: the cost for collecting results obtained from each processor by a host processor

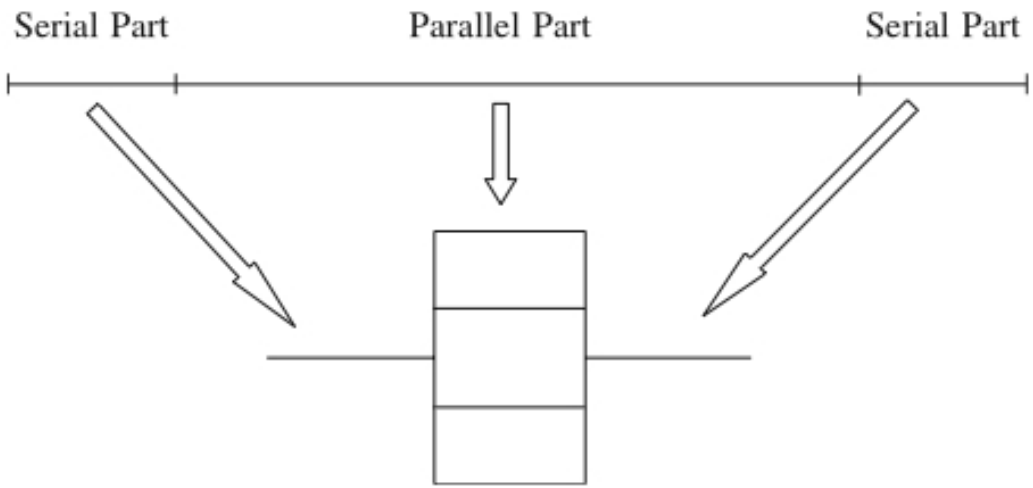


Figure 1.3 Serial part vs. parallel part

1.3. Objectives (cont'd)

- **Interference and Communication**
  - Interference: competing to access shared resources
  - Communication: one process communicating with other processes, and often one has to wait for others to be ready for communication (i.e. waiting time).

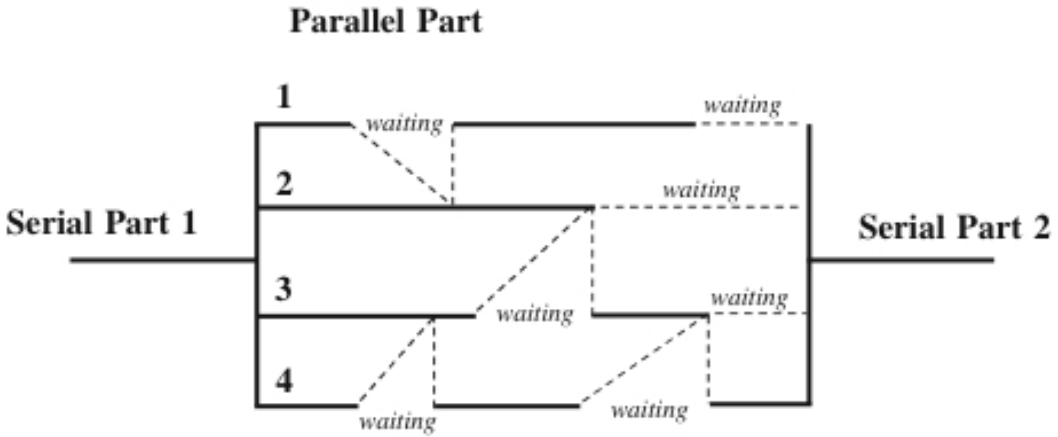


Figure 1.4 Waiting period

1.3. Objectives (cont'd)

- **Skew**
  - Unevenness of workload
  - Load balancing is one of the critical factors to achieve linear speed up

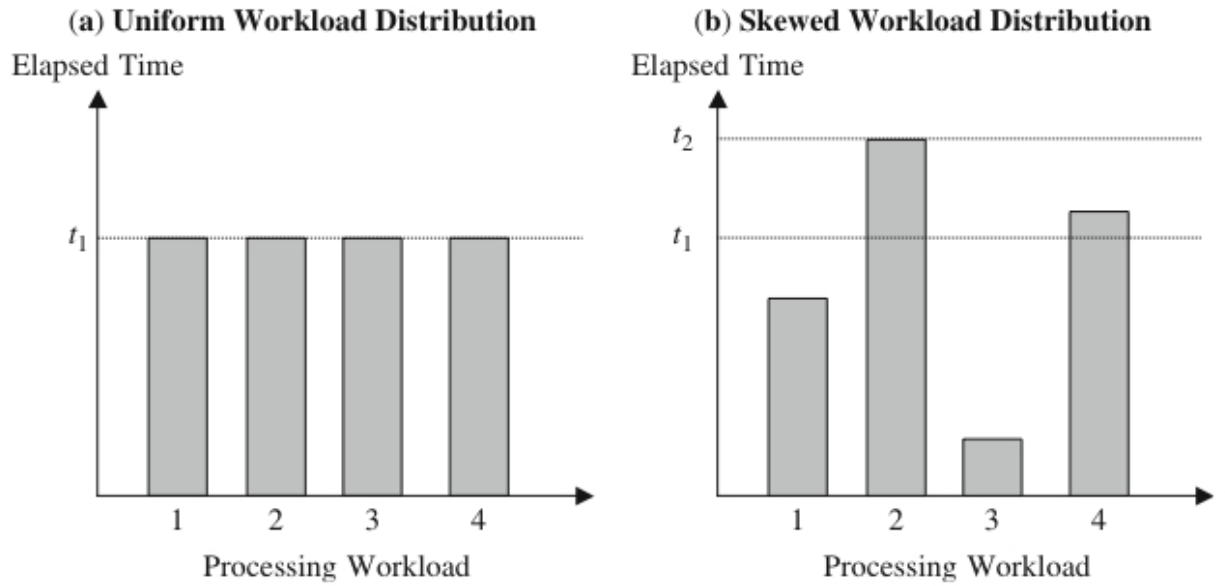


Figure 1.5 Balanced workload vs. unbalanced workload (skewed)

1.4. Forms of Parallelism

- Forms of parallelism for database processing:
  - Interquery parallelism
  - Intraquery parallelism
  - Interoperation parallelism
  - Intraoperation parallelism
  - Mixed parallelism

1.4. Forms of Parallelism (cont'd)

- **Interquery Parallelism**
  - "Parallelism among queries"
  - Different queries or transactions are executed in parallel with one another
  - Main aim: scaling up transaction processing systems

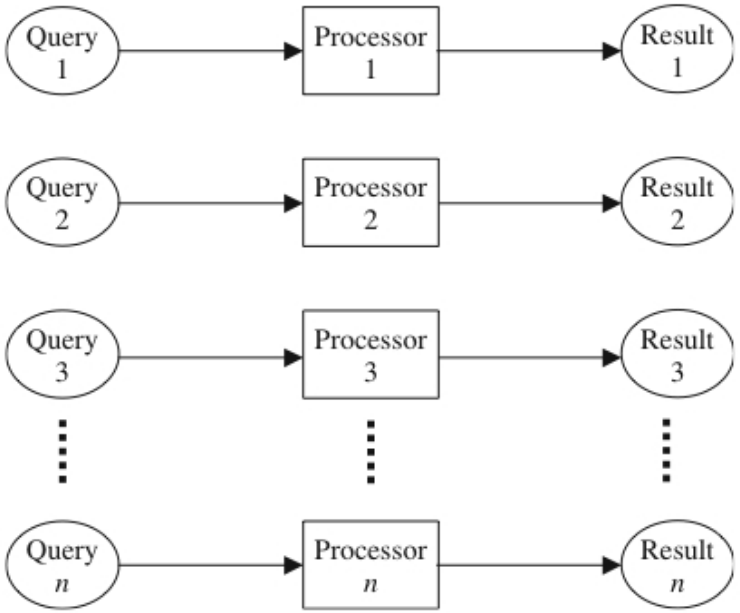


Figure 1.6 Interquery parallelism

1.4. Forms of Parallelism (cont'd)

- **Intraquery Parallelism**
  - "Parallelism within a query"
  - Execution of a single query in parallel on multiple processors and disks
  - Main aim: speeding up long-running queries

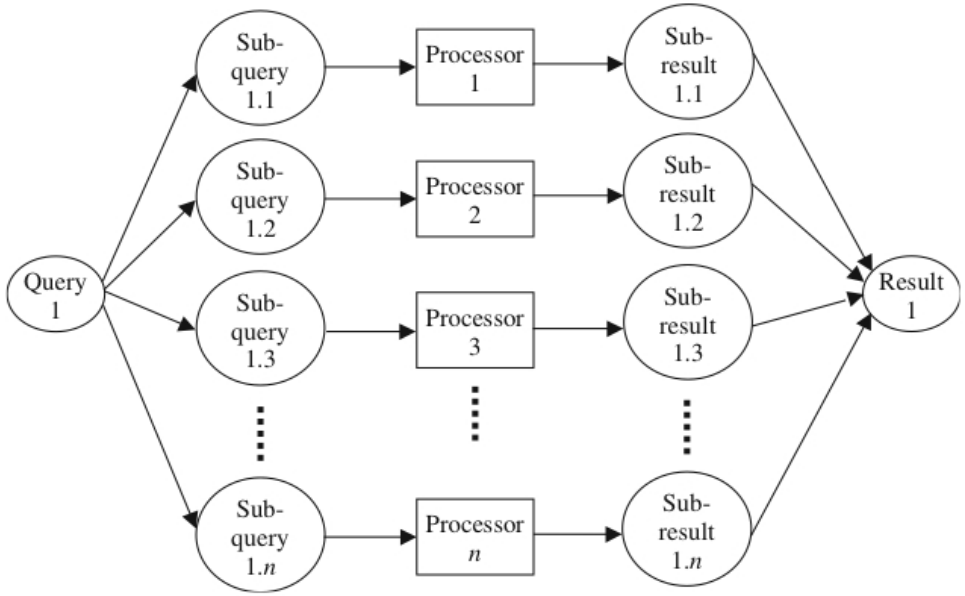


Figure 1.7 Intraquery parallelism

1.4. Forms of Parallelism (cont'd)

- Execution of a single query can be parallelized in two ways:
  - **Intraoperation parallelism**: Speeding up the processing of a query by parallelizing the execution of each individual operation (e.g. parallel sort, parallel search, etc)
  - **Interoperation parallelism**: Speeding up the processing of a query by executing in parallel different operations in a query expression (e.g. simultaneous sorting or searching)

1.4. Forms of Parallelism (cont'd)

- **Intraoperation Parallelism**
  - "Partitioned parallelism"
  - Parallelism due to the data being partitioned
  - Since the number of records in a table can be large, the degree of parallelism is potentially enourmous

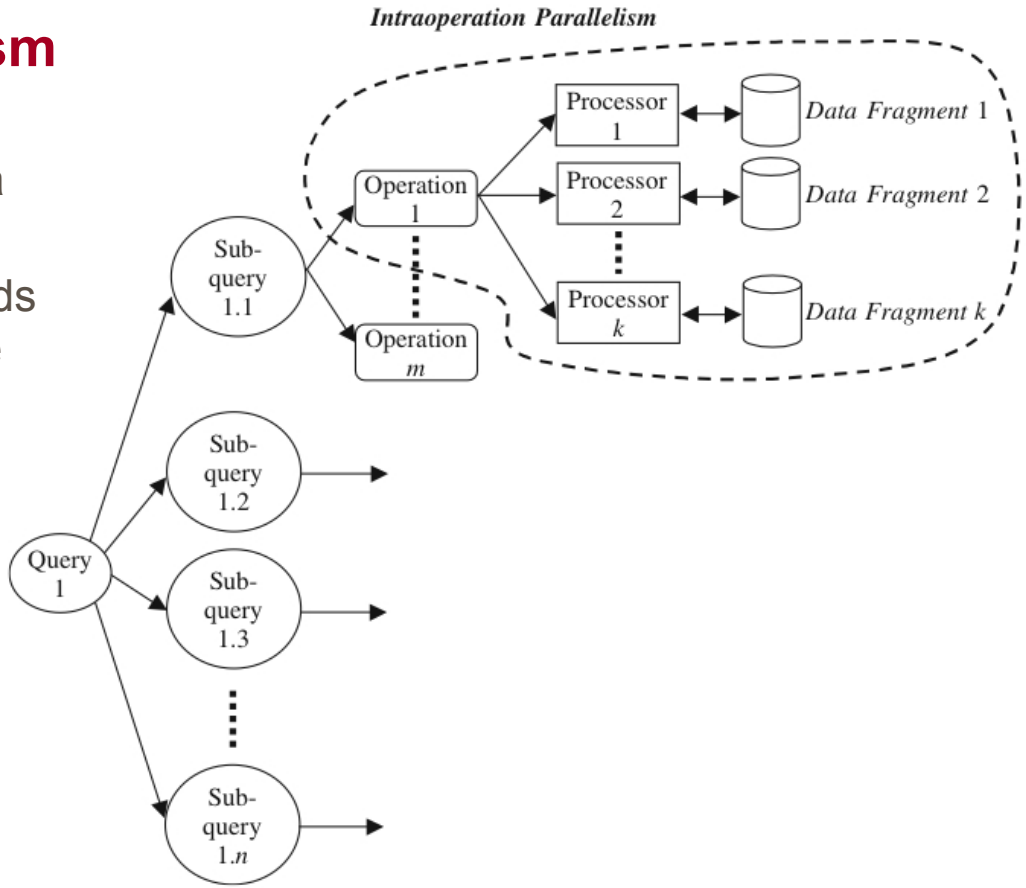


Figure 1.8 Intraoperation parallelism

1.4. Forms of Parallelism (cont'd)

- **Interoperation parallelism**: Parallelism created by concurrently executing different operations within the same query or transaction
  - Pipeline parallelism
  - Independent parallelism

1.4. Forms of Parallelism (cont'd)

- **Pipeline Parallelism**
  - Output record of one operation *A* are consumed by a second operation *B*, even before the first operation has produced the entire set of records in its output
  - Multiple operations form some sort of assembly line to manufacture the query results
  - Useful with a small number of processors, but does not scale up well

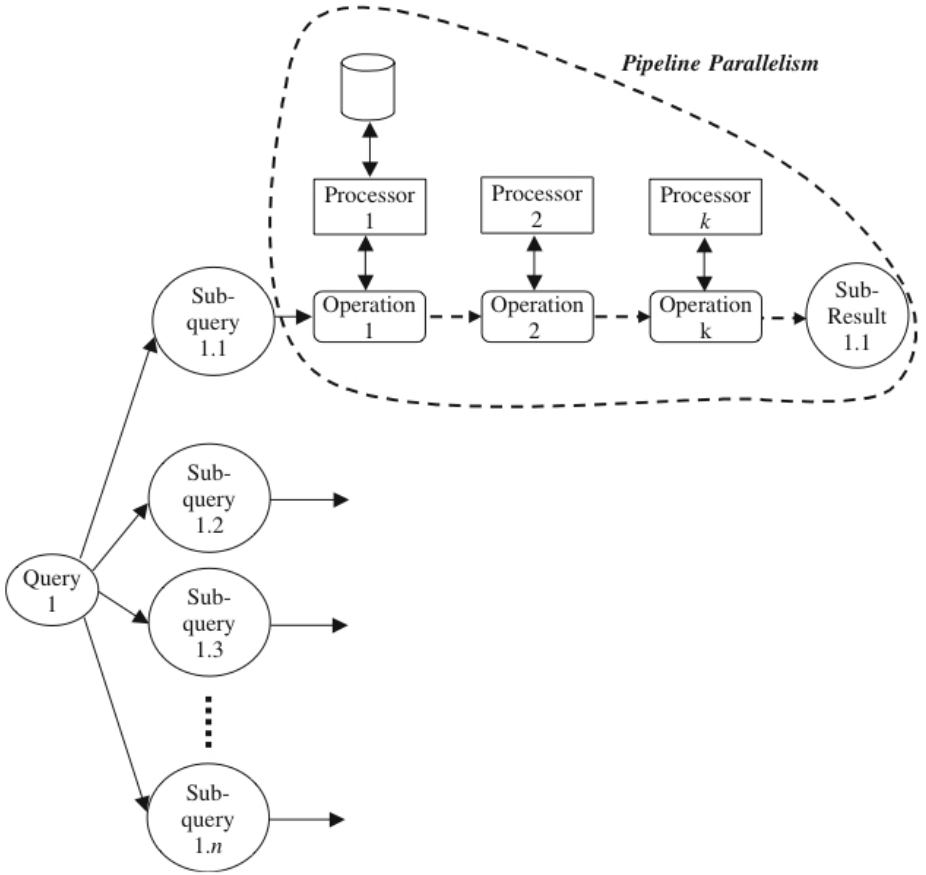


Figure 1.9 Pipeline parallelism

1.4. Forms of Parallelism (cont'd)

- **Independent Parallelism**
  - Operations in a query that do not depend on one another are executed in parallel
  - Does not provide a high degree of parallelism

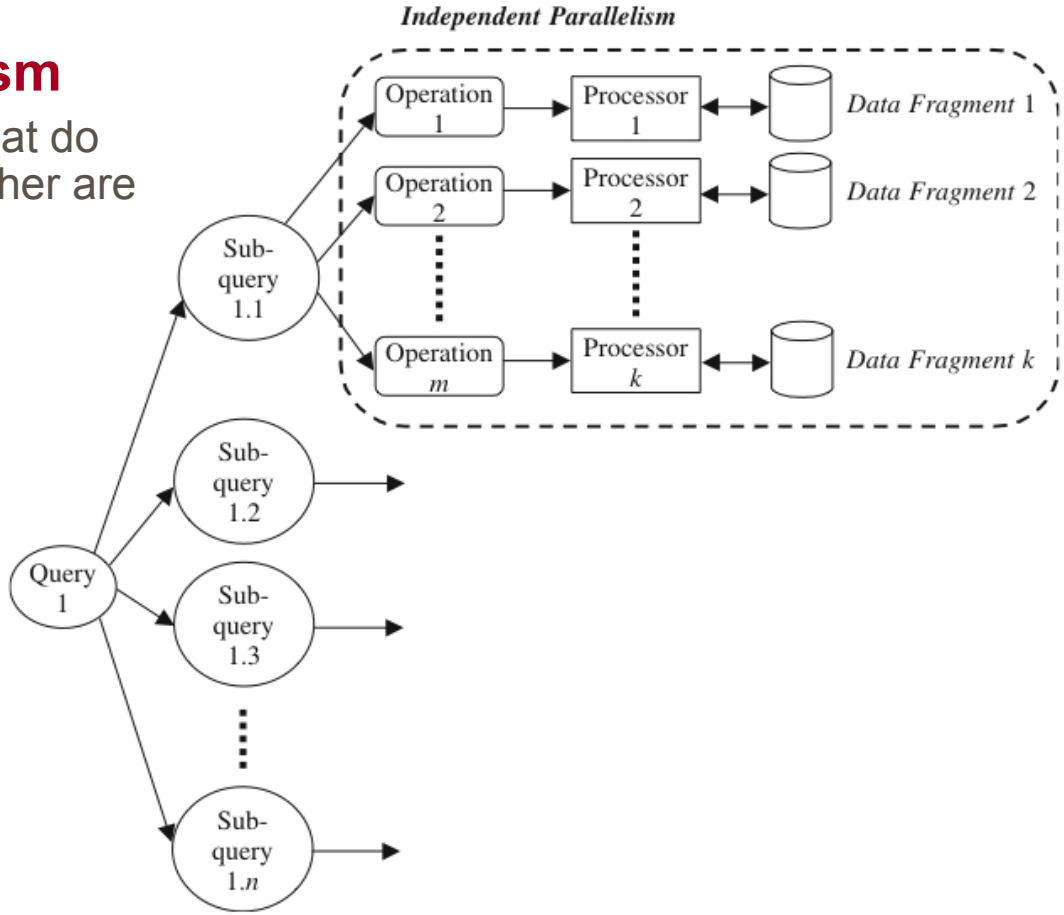


Figure 1.10 Independent parallelism

1.4. Forms of Parallelism (cont'd)

- **Mixed Parallelism**
  - In practice, a mixture of all available parallelism forms is used.

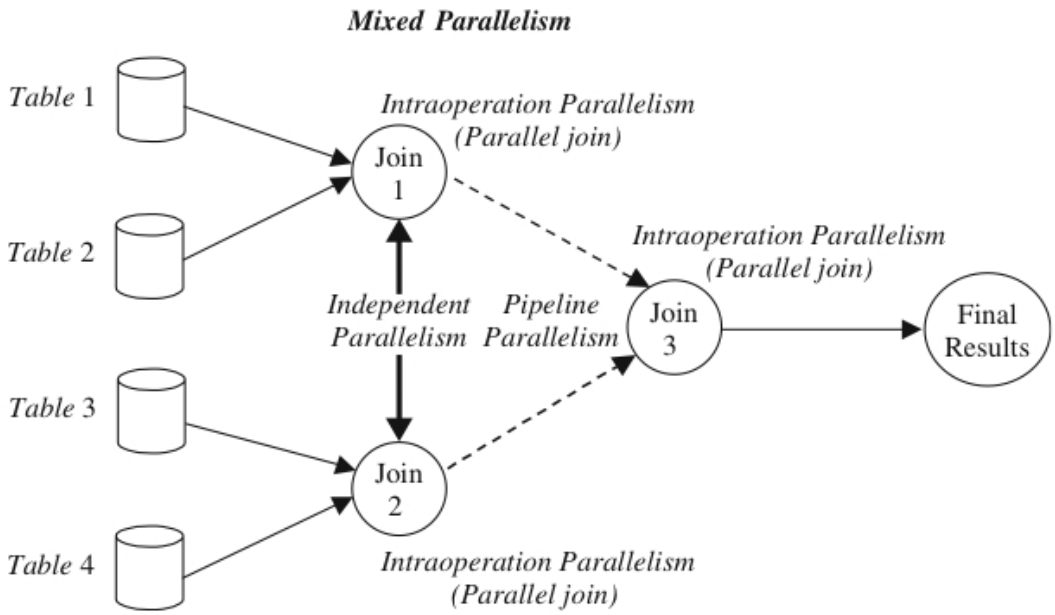


Figure 1.11 Mixed parallelism

1.5. Parallel Database Architectures

- Parallel computers are no longer a monopoly of supercomputers
- Parallel computers are available in many forms:
  - Shared-memory architecture
  - Shared-disk architecture
  - Shared-nothing architecture
  - Shared-something architecture

1.5. Parallel Database Architectures (cont'd)

- **Shared-Memory** and **Shared-Disk Architectures**
  - Shared-Memory: all processors share a common main memory and secondary memory
  - Load balancing is relatively easy to achieve, but suffer from memory and bus contention
  - Shared-Disk: all processors, each of which has its own local main memory, share the disks

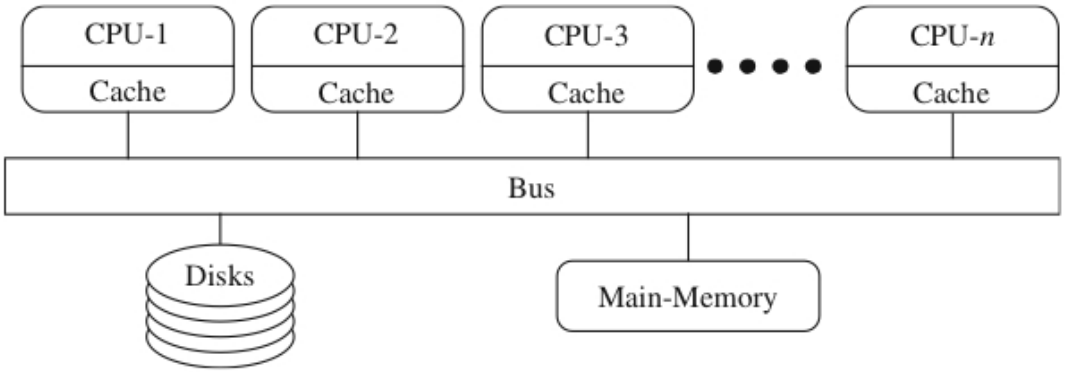


Figure 1.12 An SMP architecture

1.5. Parallel Database Architectures (cont'd)

- **Shared-Nothing Architecture**
  - Each processor has its own local main memory and disks
  - Load balancing becomes difficult

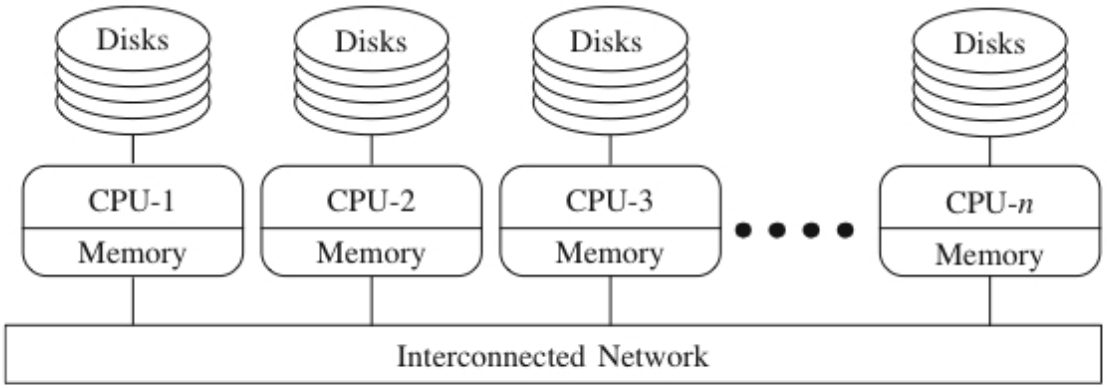


Figure 1.13 A shared-nothing architecture

1.5. Parallel Database Architectures (cont'd)

- **Shared-Something Architecture**
  - A mixture of shared-memory and shared-nothing architectures
  - Each node is a shared-memory architecture connected to an interconnection network ala shared-nothing architecture

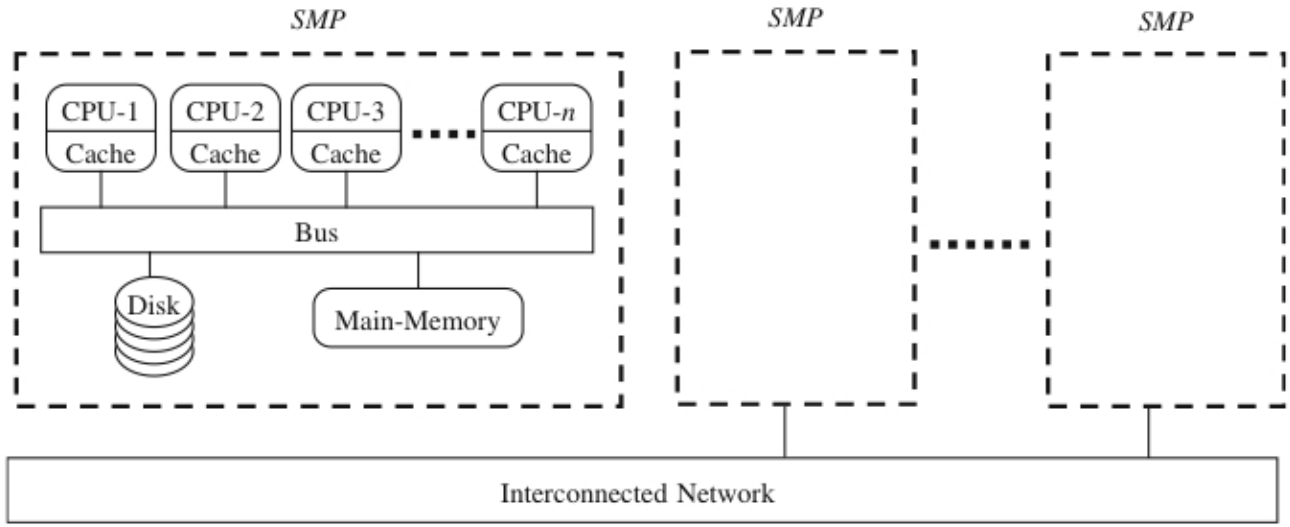


Figure 1.14 Cluster of SMP architectures

1.5. Parallel Database Architectures (cont'd)

- **Interconnection Networks**
  - Bus, Mesh, Hypercube

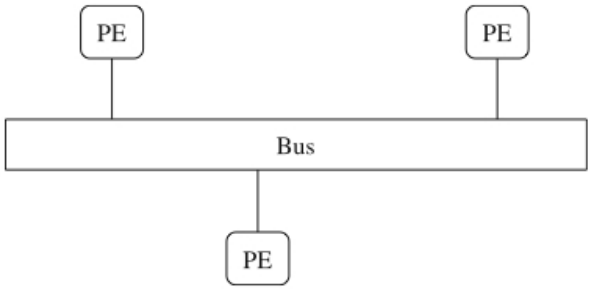


Figure 1.15 Bus interconnection network

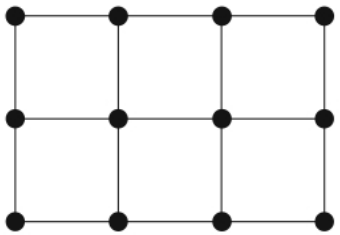


Figure 1.16 Mesh interconnection network

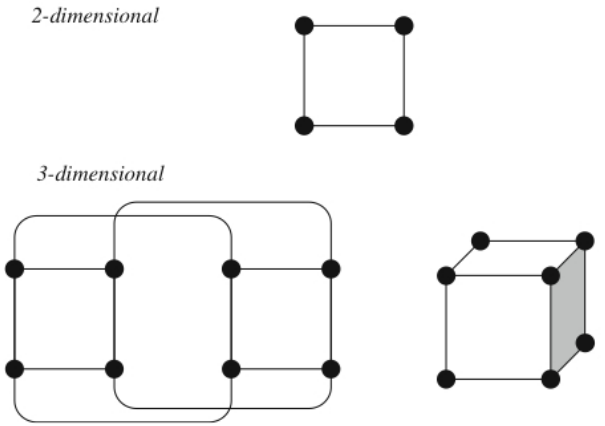


Figure 1.17 Hypercube interconnection network

1.6. Grid Database Architecture

- Wide geographical area, autonomous and heterogeneous environment
- Grid services (Meta-repository services, look-up services, replica management services, …)
- Grid middleware

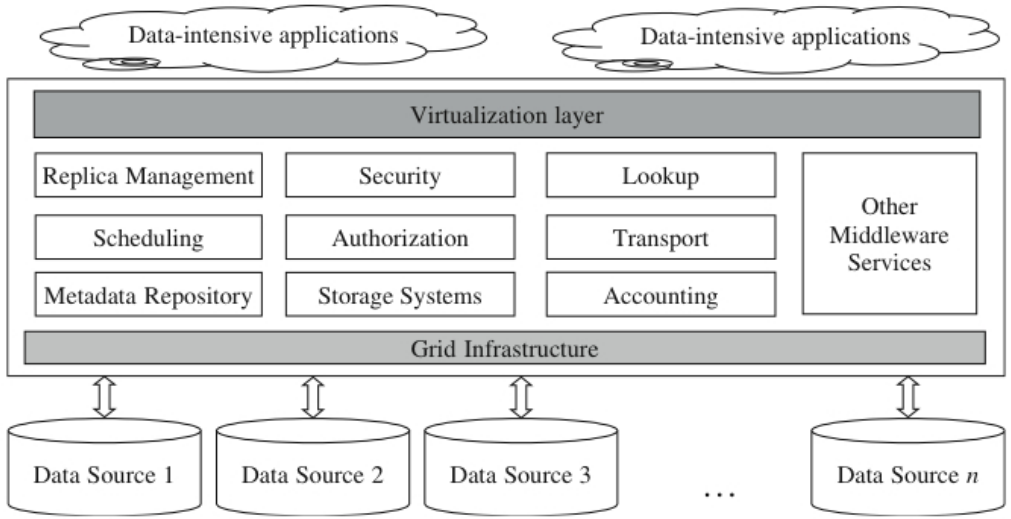


Figure 1.18 Data-intensive applications working in Grid database architecture

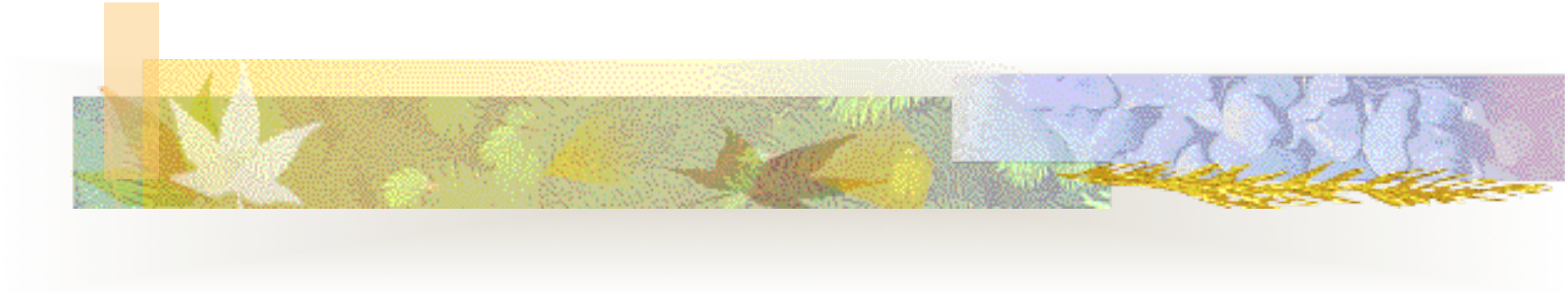
1.7. Structure of the book

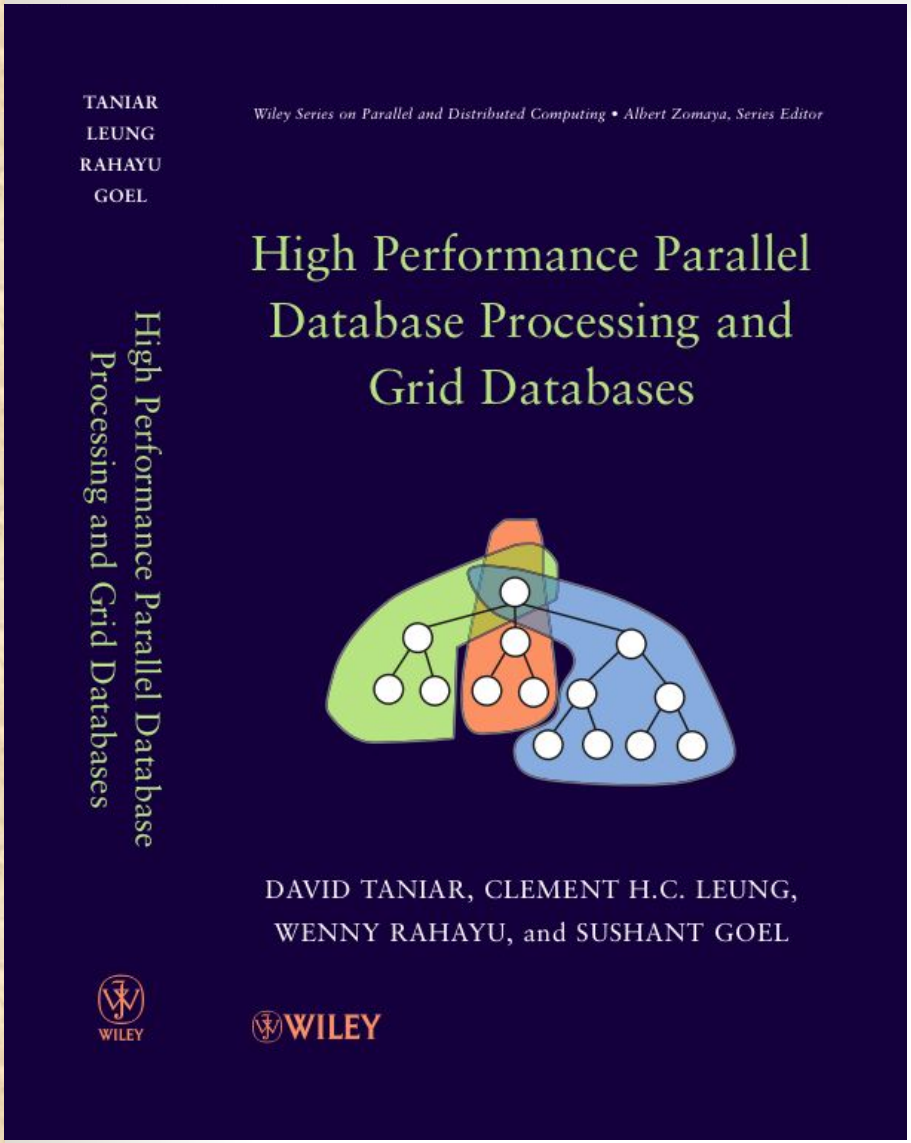
- **Part I**: Introduction and analytical models
- **Parts II** and **III**: Parallel query processing, including parallel algorithms and methods for all important database processing operations
- **Part IV**: Grid transaction management, covering the ACID properties of transaction as well as replication in Grid
- **Part V**: Parallelism of other data-intensive applications (OLAP and data mining)

1.8. Summary

- **Why**, **What**, and **How** of parallel query processing:
  - Why is parallelism necessary in database processing?
  - What can be achieved by parallelism in database processing?
  - How parallelism performed in database processing?
  - What facilities of parallel computing can be used?

Continue to Chapter 2...





Chapter 2
Analytical Models

2.1. Cost Models

- Cost equations/formulas to calculate the elapsed time of a query using a particular parallel algorithm for processing
- Composed of variables to be substituted with specific values at runtime of the query
- Although cost models may be used to estimate the performance of a query, the primary intention is to use them to describe the process involved and for comparison purposes
- Cost models serve as tools to examine every cost factor in more detail

2.2. Cost Notations

- Cost equations consists of a number of components:
  - Data parameters
  - Systems parameters
  - Query parameters
  - Time unit costs
  - Communications costs

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

2.2. Cost Notations (cont'd)

- **Data parameters**
  - Number of records in a table (|*R*|), and
  - Actual size (in bytes) of the table (*R*).

  - Data processing in each processor is based on number of records (***record level***)
  - I/O and data distribution in an interconnected network is done at a ***page level***

  - Use |*S*|) and *S* to indicate a second table

  - Use |*Ri*| and *Ri* to indicate largest fragment size located in a processor
  - Important factor: ***skewness***

2.2. Cost Notations (cont'd)

- **Systems parameters**
  - **Number of processors** (N)
  - For example: $|R|$ = 1,000,000; N = 10
  - Uniform distribution: $|Ri| = |R| / N$
    ($|R_i|$ = 1,000,000/10 = 100,000 records)
  - Skewed distribution: $|R|$ divided by 5 (for example),
    hence $|Ri|$ = 200,000 records
  - The actual number of the divisor must be modeled correctly

2.2. Cost Notations (cont'd)

- **Systems parameters**
  - **Page size** (*P*): the size of one data page in byte, which contains a batch of records
  - When records are loaded from disk to main memory, it is not loaded record by record, but page by page
  - *R* = 4 gigabytes, *P* = 4 kilobytes, hence $R / P = 1024^2$ number of pages

  - **Hash table size** (*H*): maximum size of the hash table that can fit into the main memory
  - *H* = 10,000 records

2.2. Cost Notations (cont'd)

- **Query parameters**
  - **Projectivity ratio** (π), and
  - **Selectivity ratio** (σ).

  - The value for π and σ is between 0 and 1

  - R = 100 bytes, output record size = 45 bytes; hence π = 0.45

  - $|Ri|$ = 1000 records, query results = 4 records; hence σ = 4/1000 = 0.004

2.2. Cost Notations (cont'd)

- **Time unit costs**
  - Time to read from or write to a page on disk (*IO*);
    e.g. reading a whole table from disk to main memory is *R / P* x *IO*, or in a multiprocessor environment, it is *Ri / P* x *IO*
  - Time to read a record from main memory (*tr*)
  - Time to write a record to main memory (*tw*)
  - Time to perform a computation in the main memory
  - Time to find out the destination of a record (*td*)

2.2. Cost Notations (cont'd)

- **Communication costs**
  - **Message protocol cost per page** (*mp*): initiation for a message transfer
  - **Message latency cost per page** (*ml*): actual message transfer time
  - Both elements work at a page level, as with the disk

  - Two components: one for the sender, and the other for the receiver
  - To send a whole table, the sender cost is R / P x ($mp + ml$)
  - The receiver cost is R / P x mp

  - In a multiprocessor environment, the sender cost is determined by the heaviest processor, e.g. p_1 x ($mp + ml$), where p_1 is the number of records to be distributed from the heaviest processor
  - But the receiving cost is not p_1 x mp, because the heaviest processor receiving records might be a different processor with a different number of received records. Hence, receiving cost is p_2 x ml, where $p_1 \neq p_2$

2.3. Skew Model

- A major problem in parallel processing
- The non-uniformity of workload distribution among processing elements
- Two kinds of skew:
  - Data skew
  - Processing skew

2.3. Skew Model (cont'd)

- **Data skew**
  - Caused by unevenness of data placement in a disk in each local processor, or by the previous operator
  - Although initial data placement is even, other operators may have rearranged the data, and data skew may occur as a result

2.3. Skew Model (cont'd)

- **Processing skew**
  - Caused by unevenness of the processing itself, and may be propagated by the data skew initially
  - *Zipf* distribution model to model skew
  - Measured in terms of different sizes of fragments allocated to the processors

$$|R_i| = \frac{|R|}{i^{\theta} \times \sum_{j=1}^{N} \frac{1}{j^{\theta}}} \qquad \text{where } 0 \leq \theta \leq 1 \qquad (2.1)$$

  - The symbol θ denotes the degree of skewness, where $\theta = 0$ indicates no skew, and $\theta = 1$ indicates highly skewed

2.3. Skew Model (cont'd)

- **Processing skew**
  - When $\theta = 1$, the fragment sizes follow a pure *Zipf* distribution:

$$|R_i| = \frac{|R|}{i \times \sum_{j=1}^{N} \frac{1}{j}} = \frac{|R|}{i \times H_N} \approx \frac{|R|}{i \times (\gamma + \ln N)} \qquad (2.2)$$

  - where $\gamma = 0.57721$ (*Euler's constant*) and H_N is the *harmonic number* (approx $\gamma + \ln N$)
  - In case $\theta > 0$, $|R_1|$ is the largest fragment, and $|R_N|$ is the smallest
  - Hence load skew:

$$|R_{\max}| = \frac{|R|}{\sum_{j=1}^{N} \frac{1}{j^{\theta}}} \qquad (2.3)$$

2.3. Skew Model (cont'd)

- **Processing skew**
  - For simplicity, we use $|R_i|$ instead of $|R_{max}|$
  - No skew:

$$|R_i| = \frac{|R|}{N}$$

  - Highly skewed:

$$|R_i| = \frac{|R|}{\sum_{j=1}^{N} \frac{1}{j^{\theta}}}$$

2.3. Skew Model (cont'd)

- **Example**
  - $|R|$=100,000 records, N=8 processors

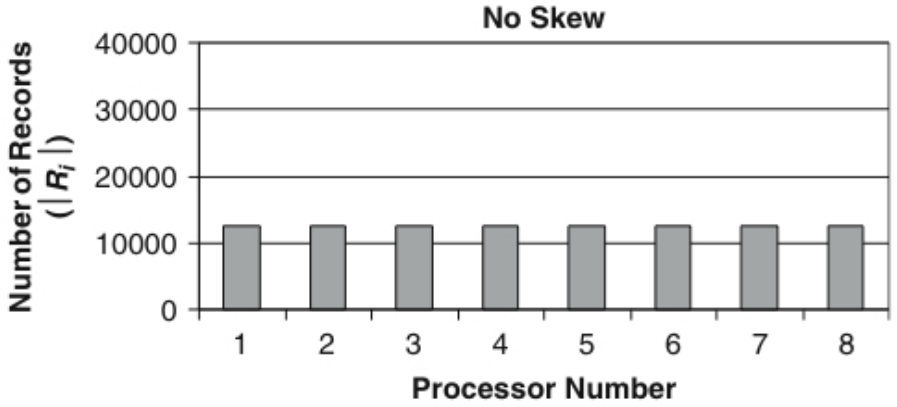


Figure 2.1 Uniform distribution (no skew)

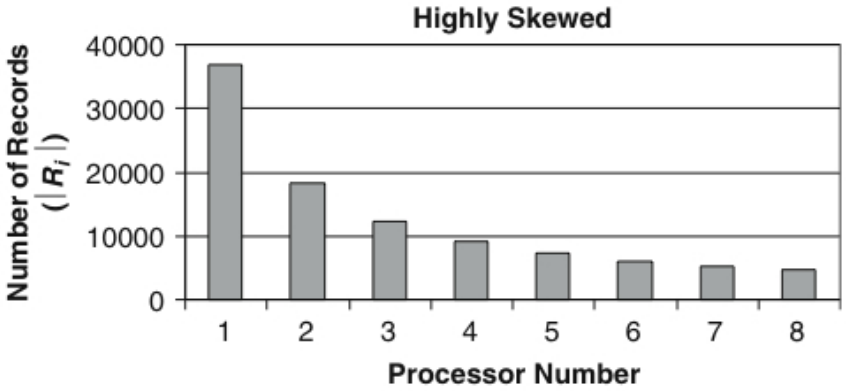


Figure 2.2 Highly skewed distribution

2.3. Skew Model (cont'd)

- No skew vs. highly skewed

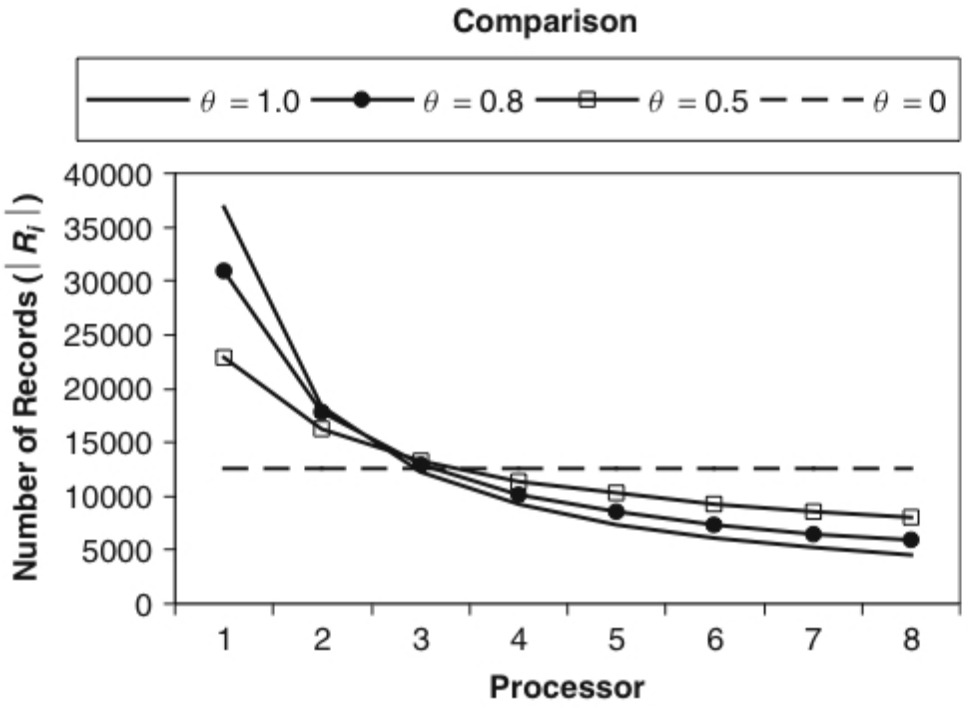
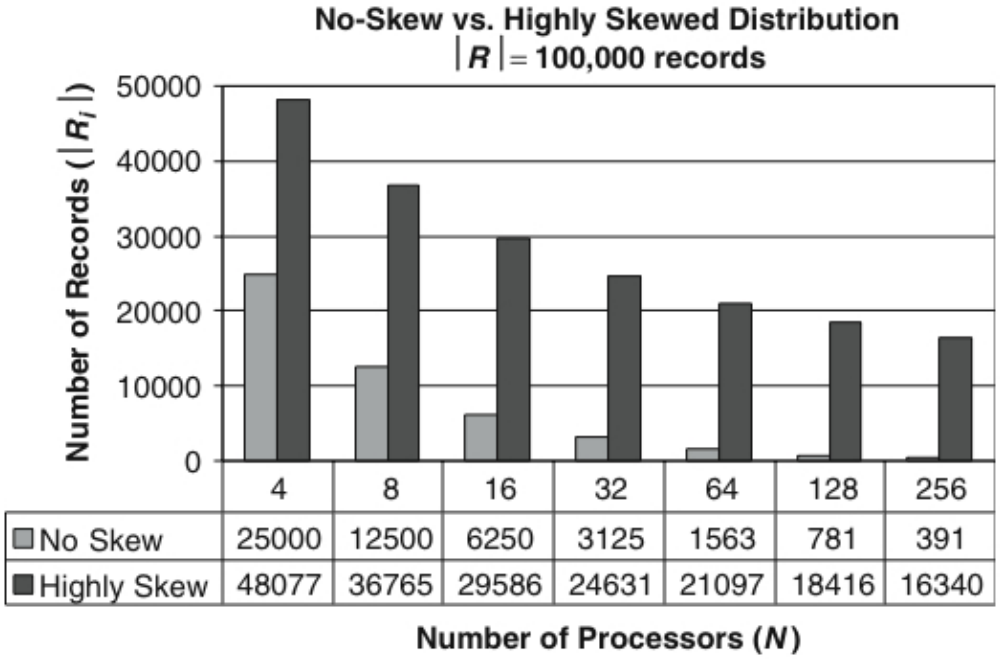


Figure 2.3 Comparison between highly skewed, less skewed, and no-skew distributions

2.3. Skew Model (cont'd)

- No skew vs. highly skewed



	4	8	16	32	64	128	256
No Skew	25000	12500	6250	3125	1563	781	391
Highly Skew	48077	36765	29586	24631	21097	18416	16340

Figure 2.4 Comparison between the heaviest loaded processors using no-skew and highly skewed distributions

2.3. Skew Model (cont'd)

- No skew vs. highly skewed

Table 2.2 Divisors (with vs. without skew)

N	4	8	16	32	64	128	256
Divisor without skew	4	8	16	32	64	128	256
Divisor with skew	2.08	2.72	3.38	4.06	4.74	5.43	6.12

2.4. Basic Operations

- Operations in parallel database systems normally follow these steps:
  - Data loading (scanning) from disk,
  - Getting records from data page to main memory,
  - Data computation and data distribution,
  - Writing records (query results) from main memory to data page, and
  - Data writing to disk.

2.4. Basic Operations (cont'd)

- **Disk operations**
  - Disk reading and writing is based on page (I/O page) (*P*)
  - Based on the heaviest processor (*Ri*)
  - Uniform distribution: $Ri = R / N$
  - Skewed distribution: $Ri = R / (\gamma + \ln N)$

  - ***Scanning cost*** = $Ri / P \times IO$

  - ***Writing cost*** = (*data computation variables* x *Ri*) / *P* x *IO*,
    where $0.0 \leq$ *data computation variable* ≤ 1.0
    data computation variable = 0.0 means that no records exist in the query results, whereas *data computation variable* = 0.0 indicates that all records are written back to disk.

2.4. Basic Operations (cont'd)

- **Main memory operations**
  - Once the data has been loaded from the disk, the record has to be removed from the data page and placed in main memory (*select cost*)
  - Main memory operations are based on records (|*Ri*|), not on pages (*Ri*)
  - The reading unit cost (*tr*) is the reading operation of records from the data page, the writing unit cost (*tw*) is to actually write the record to main memory
  - ***Select cost*** = |*Ri*| x (*tr* + *tw*)

  - Only writing unit cost is involved, and not reading unit cost, as the reading unit cost is already part of the computation
  - ***Query results generation cost*** = (*data computation variables* x |*Ri*|) x *tw*

2.4. Basic Operations (cont'd)

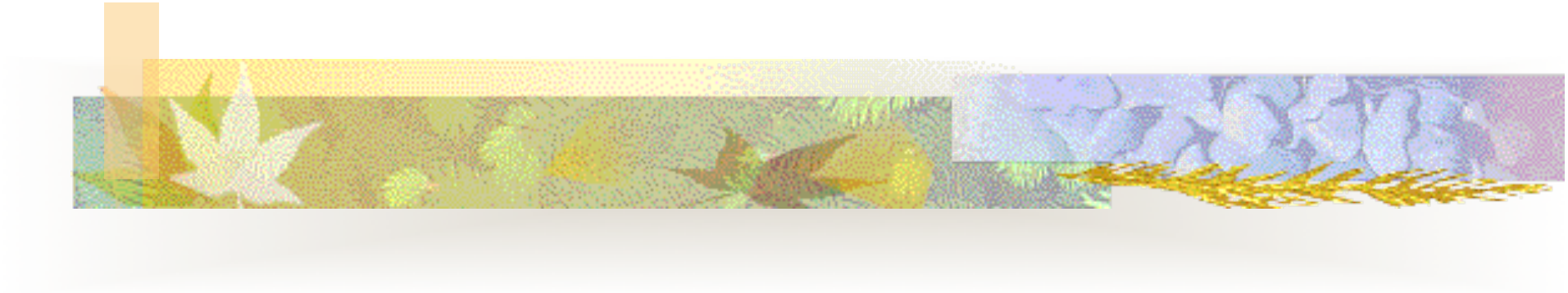
- **Data computation and data distribution**
  - Data computation cost is the cost of basic database operations
  - Works in main memory, and hence uses the number of records
  - Each data computation operation may involve several basic costs (unit costs for hashing, for adding the current record to the aggregate value, …)
  - Data computation unit cost is *tx*, and |*Ri*| may be skewed
  - ***Data computation cost*** = |*Ri*| x (*tx*)

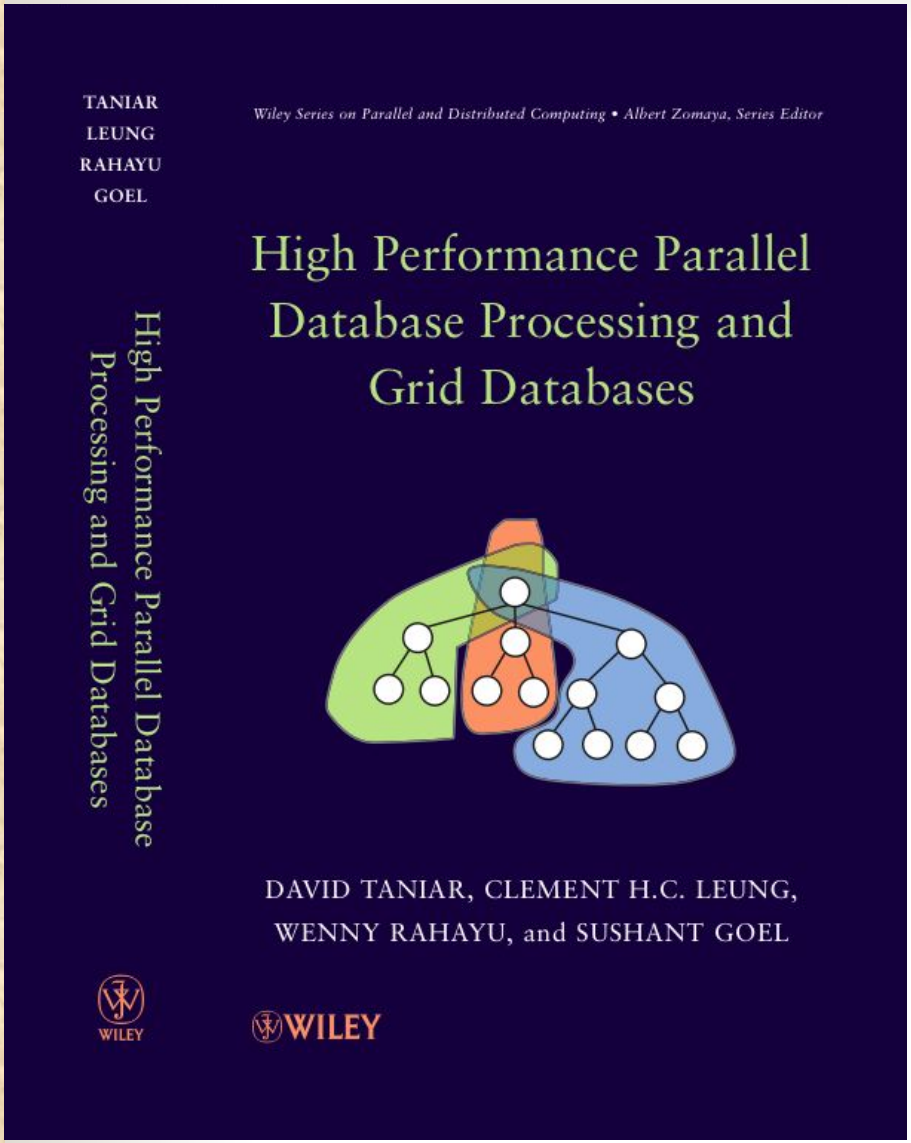
  - Data distribution is record transmission from one processor to another
  - Involves two costs: the cost associated with determining where each record goes, and the actual data transmission itself
  - The former works in main memory (number of records), the latter is based on number of pages
  - ***Determining the destination cost*** = |*Ri*| x (*td*)
  - Data transmission costs (communication costs) have been explained in section 2.2 previously

2.5. Summary

- Basic cost notations
  - Parameters, such as data parameters, systems parameters, query parameters, time unit costs, and communication costs
- Skew model
  - *Zipf* distribution model
- Basic parallel database processing costs
  - General steps of parallel database processing, such as disk costs, main memory costs, data computation costs, and data distribution costs

Continue to Chapter 3...





Chapter 3
Parallel Search

3.1. Search Queries

- Search is ***selection*** operation in database queries
- Selects specified records based on a given criteria
- The result is a horizontal subset (records) of the operand

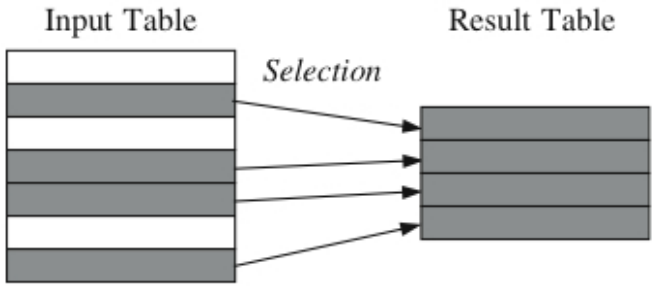


Figure 3.1 Selection operation

- Three kinds of search queries:
  - Exact-match search
  - Range search
  - Multi attribute search

3.1. Search Queries (cont'd)

- **Exact-Match Search**
  - Selection predicate on an attribute to check for an exact match between a search attribute and a given value
  - Expressed by the WHERE clause in SQL

  - Query 3.1 will produce a unique record (if the record is found), whereas Query 3.2 will likely produce multiple records

```
Query 3.1:
   Select *
   From STUDENT
   Where Sid = 23;
```

```
Query 3.2:
   Select *
   From STUDENT
   Where Slname = 'Robinson';
```

3.1. Search Queries (cont'd)

- **Range Search Query**
  - The search covers a certain range
  - **Continuous range search query**

```
Query 3.3:
   Select *
   From STUDENT
   Where Sgpa > 3.50;
```

  - **Discrete range search query**

```
Query 3.4:
   Select *
   From STUDENT
   Where Sdegree IN ('BCS', 'BInfSys');
```

3.1. Search Queries (cont'd)

- **Multiattribute Search Query**
  - More than attribute is involved in the search
  - Conjunctive (AND) or Disjunctive (OR)
  - If both are used, it must be in a form of *conjunctive prenex normal form* (CPNF)

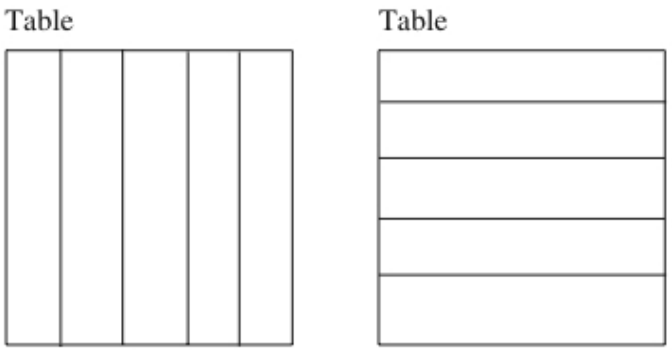
```
Query 3.6:
    Select *
    From STUDENT
    Where Slname = 'Robinson'
    And Sdegree IN ('BCS', 'BInfSys');
```

3.2. Data Partitioning

- Distributes data over a number of processing elements
- Each processing element is then executed simultaneously with other processing elements, thereby creating parallelism
- Can be physical or logical data partitioning
- In a shared-nothing architecture, data is placed permanently over several disks
- In a shared-everything (shared-memory and shared-disk) architecture, data is assigned logically to each processor
- Two kinds of data partitioning:
  - Basic data partitioning
  - Complex data partitioning

3.2. Data Partitioning (cont'd)

- **Basic Data Partitioning**
  - Vertical vs. Horizontal data partitioning
  - Vertical partitioning partitions the data vertically across all processors. Each processor has a full number of records of a particular table. This model is more common in distributed database systems
  - Horizontal partitioning is a model in which each processor holds a partial number of complete records of a particular table. It is more common in parallel relational database systems



(a) *Vertical Partitioning* (b) *Horizontal Partitioning*

Figure 3.2 Vertical and horizontal data partitioning

3.2. Data Partitioning (cont'd)

- **Basic Data Partitioning**
  - Round-robin data partitioning
  - Hash data partitioning
  - Range data partitioning
  - Random-unequal data partitioning

3.2. Data Partitioning (cont'd)

- **Round-robin data partitioning**
  - Each record in turn is allocated to a processing element in a clockwise manner
  - "Equal partitioning" or "Random-equal partitioning"
  - Data evenly distributed, hence supports load balance
  - But data is not grouped semantically

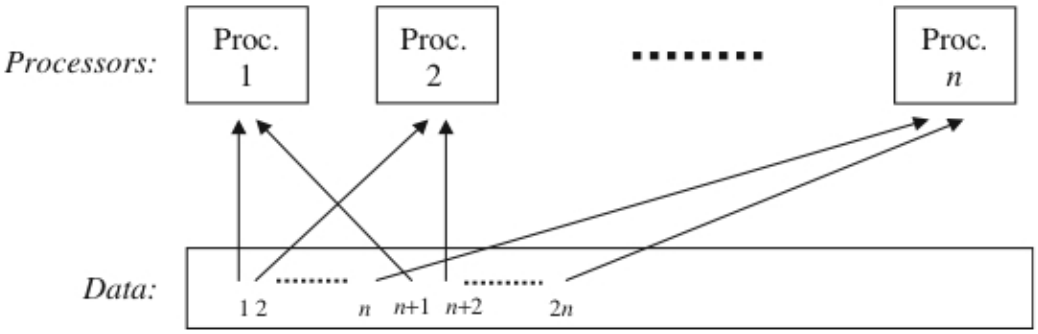


Figure 3.3 *Round-robin* data partitioning

3.2. Data Partitioning (cont'd)

- **Hash data partitioning**
  - A hash function is used to partition the data
  - Hence, data is grouped semantically, that is data on the same group shared the same hash value
  - Selected processors may be identified when processing a search operation (exact-match search), but for range search (especially continuous range), all processors must be used
  - Initial data allocation is not balanced either

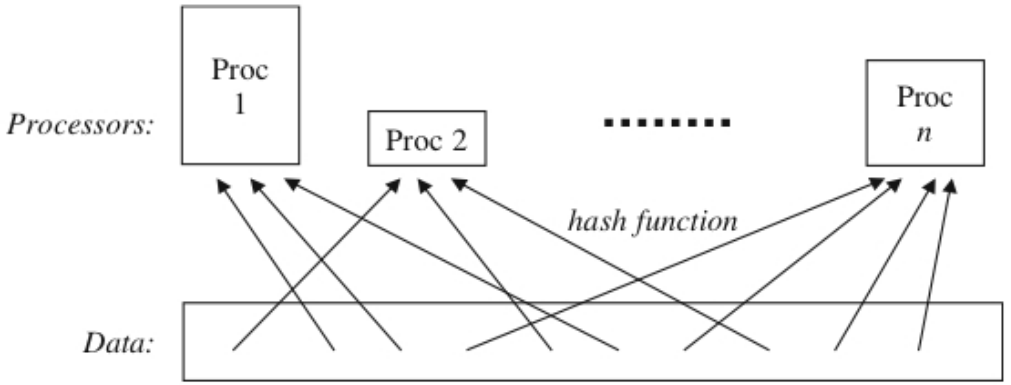


Figure 3.4 *Hash* data partitioning

3.2. Data Partitioning (cont'd)

- **Range data partitioning**
  - Spreads the records based on a given range of the partitioning attribute
  - Processing records on a specific range can be directed to certain processors only
  - Initial data allocation is skewed too

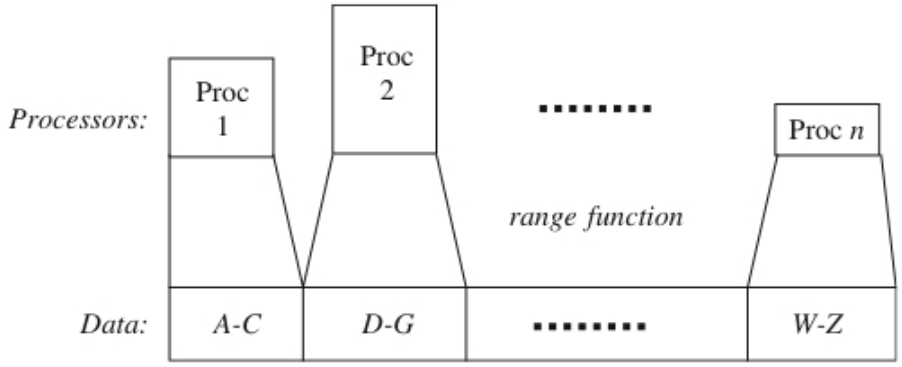


Figure 3.5 *Range* data partitioning

3.2. Data Partitioning (cont'd)

- **Random-unequal data partitioning**
  - Partitioning is not based on the same attribute as the retrieval processing is based on a nonretrieval processing attribute, or the partitioning method is unknown
  - The size of each partitioning is likely to be unequal
  - Records within each partition are not grouped semantically
  - This is common especially when the operation is actually an operation based on temporary results obtained from the previous operations

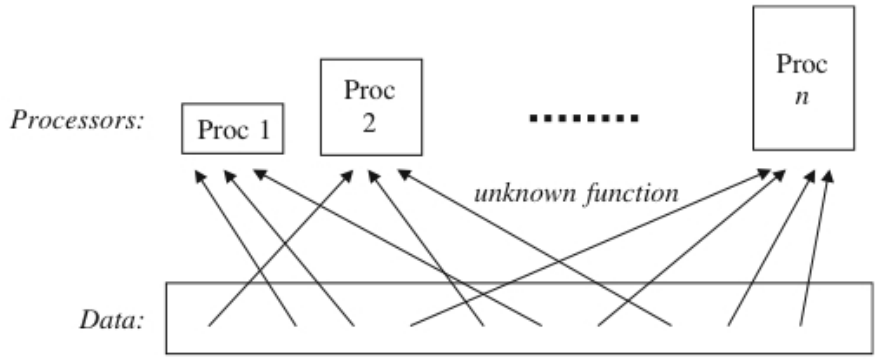


Figure 3.6 *Random-unequal* data partitioning

3.2. Data Partitioning (cont'd)

- **Basic Data Partitioning**
  - Attribute-based data partitioning
  - Non-attribute-based data partitioning

Table 3.1 *Attribute-based* versus *non-attribute-based* data partitioning

Attribute-Based Partitioning	Non-Attribute-Based Partitioning
Based on a particular attribute	Not based on any attribute
Has grouping semantics	No grouping semantics
Skew	Balanced

3.2. Data Partitioning (cont'd)

- **Complex Data Partitioning**
  - Basic data partitioning is based on a single attribute (or no attribute)
  - Complex data partitioning is based on multiple attributes or is based on a single attribute but with multiple partitioning methods

  - Hybrid-Range Partitioning Strategy (HRPS)
  - Multiattribute Grid Declustering (MAGIC)
  - Bubba's Extended Range Declustering (BERB)

3.2. Data Partitioning (cont'd)

- **Hybrid-Range Partitioning Strategy (HRPS)**
  - Partitions the table into many fragments using range, and the fragments are distributed to all processors using round-robin
  - Each fragment contains approx *FC* records

$$FC = \frac{RecordsPerQ_{Ave}}{M} \qquad (3.1)$$

Where $RecordsPerQ_{Ave}$ is the average number of records retrieved and processed by each query, and *M* is the number of processors that should participate in the execution of an average query

  - Each fragment contains a unique range of values of the partitioning attribute
  - The table must be sorted on the partitioning attribute, then it is partitioned that each fragment contains *FC* records, and the fragments are distributed in round-robin ensuring that *M* adjacent fragements assigned to different processors

3.2. Data Partitioning (cont'd)

- **Hybrid-Range Partitioning Strategy (HRPS)**
  - Example: 10000 student records, and the partitioning attribute is StudentID (PK) that ranges from 1 to 10000. Assume the average query retrieves a range of 500 records (*RecordsPerQ*=500). Queries access students per year enrolment wth average results of 500 records. Assume the optimal performance is achieved when 5 processors are used (*M*=5)

$$FC = \frac{RecordsPerQ_{Ave}}{M} = 100$$

  - The table will be partitioned into 100 fragments
  - Three cases: $M = N$, $M > N$, or $M < N$ (where N is the number of processors in the configuration, and M is the number of processors participating in the query execution

3.2. Data Partitioning (cont'd)

- **Hybrid-Range Partitioning Strategy (HRPS)**
  - **Case 1: *M* = *N***
  - Because the query will overlap with 5-6 fragments, all processors will be used (high degree of parallelism)
  - Compared with hash partitioning: Hash will also use *N* processors, since it cannot localize the execution of a range query
  - Compared with range partitioning: Range will only use 1-2 processors, and hence the degree of parallelism is small

HRPS	1-100 9501-9600	101-200 9601-9700	201-300 9701-9800	301-400 9801-9900	401-500 9901-10000
Range	1-2000	2001-4000	4001-6000	6001-8000	8001-10000

Figure 3.7 Case 1 ($M = N$) and a comparison with the range partitioning method

3.2. Data Partitioning (cont'd)

- **Hybrid-Range Partitioning Strategy (HRPS)**
  - **Case 2: *M* > *N*** (e.g. *M*=5, and *N*=2)
  - HRPS will still use all *N* processors, because it enforces the constraint that the *M* adjacent fragments be assigned to different processors whenever possible
  - Compared with range partitioning: an increased probability that a query will use only one processor (in this example)

HRPS	1-100 201-300 9801-9900	101-200 301-400 9901-10000
Range	1-5000	5001-10000

Figure 3.8 Case 2 ($M > N$) and a comparison with the range partitioning method

3.2. Data Partitioning (cont'd)

- **Hybrid-Range Partitioning Strategy (HRPS)**
  - **Case 3: *M* < *N*** (e.g. *M*=5, and *N*=10)
  - HRPS distributes 100 fragments to all *N* processors. Since the query will overlap with only 5-6 fragments, each individual query is localized to almost the optimal number of processors
  - Compared with hash partitioning: Hash will use all *N* processors, and hence less efficient due to start up, communication, and termination overheads
  - Compared with range partitioning: The query will use 1-2 processors only, and hence less optimal

HRPS	1-100	101-200	201-300	301-400	401-500	501-600	601-700	701-800	801-900	901-1000
	...	...	...	...	...	...	...	...	...	...
	...	...	...	...	...	...	...	...	...	...
	...	...	...	...	...	...	...	...	...	...
	9001-9100	9101-9200	9201-9300	9301-9400	9401-9500	9501-9600	9601-9700	9701-9800	9801-9900	9901-10000
Range	1-1000	1001-2000	2001-3000	3001-4000	4001-5000	5001-6000	6001-7000	7001-8000	8001-9000	9001-10000

Figure 3.9 Case 3 ($M < N$) and a comparison with the range partitioning method

3.2. Data Partitioning (cont'd)

- **Hybrid-Range Partitioning Strategy (HRPS)**
  - **Support for Small Tables**

    If the number of fragments of a table is less than the number of processors, then the table will automatically be partitioned across a subset of the processors

  - **Support for Tables with Nonuniform Distributions of the Partitioning Attribute Values**

    Because the cardinality of each fragment is not based on the value of the partitioning attribute value, once the HRPS determines the cardinality of each fragment, it will partition a table based on that value

3.2. Data Partitioning (cont'd)

- **Multiattribute Grid Declustering (MAGIC)**
  - Based on multiple attributes - to support search queries based on either of data partitioning attributes
  - Support range and exact match search on each of the partitioning attributes
  - Example: Query 1 (one-half of the accesses) Slname='Roberts', and Query 2 (the other half) SID between 98555 and 98600. Assume both queries produce only a few records
  - Create a two-dim grid with the two partitioning attributes (Slname and SID). The number of cells in the grid equal the number of processing elements
  - Determine the range value for each column and row, and allocate a processor in each cell in the grid

3.2. Data Partitioning (cont'd)

- **Multiattribute Grid Declustering (MAGIC)**
  - Query 1 (exact match on Slname): Hash partitioning can localize the query processing on one processor. MAGIC will use 6 processors
  - Query 2 (range on SID): if the hash partitioning uses Slname, whereas the query is on SID, the query must use all 36 processors. MAGIC on the other hand, will only use 6 processors.
  - Compared with range partitioning, suppose the partitioning is based on SID, then Q1 will use 36 processors whilst Q2 will use 1 processor

Table 3.2 MAGIC data partitioning

		Slname					
		A–D	**E–H**	**I–L**	**M–P**	**Q–T**	**U–Z**
Sid	**98000–98100**	1	2	3	4	5	6
	98101–98200	7	8	9	10	11	12
	98201–98300	13	14	15	16	17	18
	98301–98400	19	20	21	22	23	24
	98401–98500	25	26	27	28	29	30
	98501–98600	31	32	33	34	35	36

3.2. Data Partitioning (cont'd)

- **Bubba's Extended Range Declustering (BERB)**
  - Another multiattribute partitioning method - used in the Bubba Database Machine
  - Two levels of data partitioning: ***primary*** and ***secondary*** data partitioning
  - Step 1: Partition the table based on the primary partitioning attribute and uses a range partitioning method

Table 3.3 Primary partitioning in BERD

Sid	*Slname*
98001	Robertson
98050	Williamson

98001–98100

Sid	*Slname*
98105	Black
98113	White

98101–98200

Sid	*Slname*
98250	Chan
98270	Tan

98201–98300

3.2. Data Partitioning (cont'd)

- **Bubba's Extended Range Declustering (BERB)**
  - Step 2: Each fragment is scanned and an 'aux' table is created from the attribute value of the secondary partitioning attribute and a list of processors containing the original records
  - Table 3.4 shows the 'aux' table (called Table *IndexB*)

Table 3.4 Auxiliary table in the secondary partitioning

Slname	*Processor*
Robertson	1
Black	2
Chan	3
Williamson	1
White	2
Tan	3

3.2. Data Partitioning (cont'd)

- **Bubba's Extended Range Declustering (BERB)**
  - Step 3: The 'aux' table is range partitioned on the secondary partitioning attribute (e.g. Slname)
  - Step 4: Place the fragments from steps 1 and 3 into multiple processors

Table 3.5 BERD partitioning combining the primary partitions and the secondary partitions

IndexB

Black	2
Chan	3

Student

98001	Robertson
98050	Williamson

IndexB

Robertson	1
Tan	3

Student

98005	Black
98113	White

IndexB

Williamson	1
White	2

Student

98250	Chan
98270	Tan

3.3. Search Algorithms

- **Serial** search algorithms:
  - Linear search
  - Binary search

- **Parallel** search algorithms:
  - Processor activation or involvement
  - Local searching method
  - Key comparison

3.3. Search Algorithms (cont'd)

- **Linear Search**
  - Exhaustive search - search each record one by one until it is found or end of table is reached

  - **Scanning** cost: 1/2 x R / P x IO
  - **Select** cost: 1/2 x $|R|$ x (tr + tw)
  - **Comparison** cost: 1/2 x $|R|$ x tr
  - **Result generation** cost: σ x $|R|$ x tw, where σ is the search query selection ratio
  - **Disk writing** cost: σ x R / P x IO

3.3. Search Algorithms (cont'd)

- **Binary Search**
  - Must be pre-sorted
  - The complexity is $O(\log_2(n))$
  - The cost components for binary search are similar to those of linear search, except that the component of 1/2 in linear search is now replaced with $\log_2$:

$$\begin{aligned} Scanning\ cost &= \log_2(R)/P \times IO \\ Select\ cost &= \log_2(|R|) \times (t_r + t_w) \\ Comparison\ cost &= \log_2(|R|) \times t_c \\ Result\ generation\ cost &= \sigma \times |R| \times t_w \\ Disk\ writing\ cost &= \sigma \times R/P \times IO \end{aligned}$$

3.3. Search Algorithms (cont'd)

- **Parallel** search algorithms:
  - Processor activation or involvement
  - Local searching method
  - Key comparison

3.3. Search Algorithms (cont'd)

- **Processor activation or involvement**
  - The number of processors to be used by the algorithm
  - If we know where the data to be sought are stored, then there is no point in activating all other processors in the searching process
  - Depends on the data partitioning method used
  - Also depends on what type of selection query is performed

Table 3.6 Processor activation or involvement of parallel search algorithms

		Data Partitioning Methods			
		Random-Equal	**Hash**	**Range**	**Random-Unequal**
Exact Match		All	1	1	All
Range Selection	**Continuous**	All	All	Selected	All
	Discrete	All	Selected	Selected	All

3.3. Search Algorithms (cont'd)

- **Local searching method**
  - The searching method applied to the processor(s) involved in the searching process
  - Depends on the data ordering, regarding the type of the search (exact match of range)

Table 3.7 Local searching method of parallel search algorithms

		Records Ordering	
		Ordered	**Unordered**
Exact Match		Binary Search	Linear Search
Range Selection	**Continuous**	Binary Search	Linear Search
	Discrete	Binary Search	Linear Search

3.3. Search Algorithms (cont'd)

- **Key comparison**
  - Compares the data from the table with the condition specified by the query
  - When a match is found: continue to find other matches, or terminate
  - Depends on whether the data in the table is unique or not

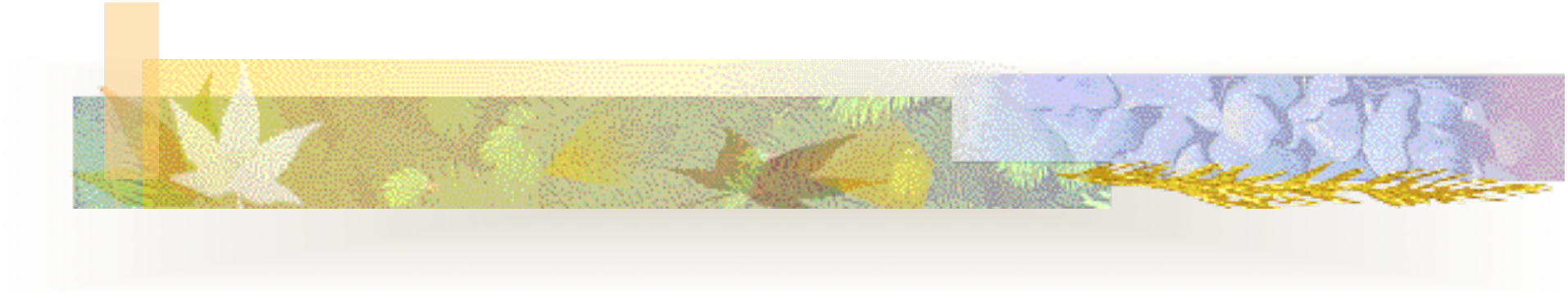
Table 3.8 Key comparison of parallel search algorithms

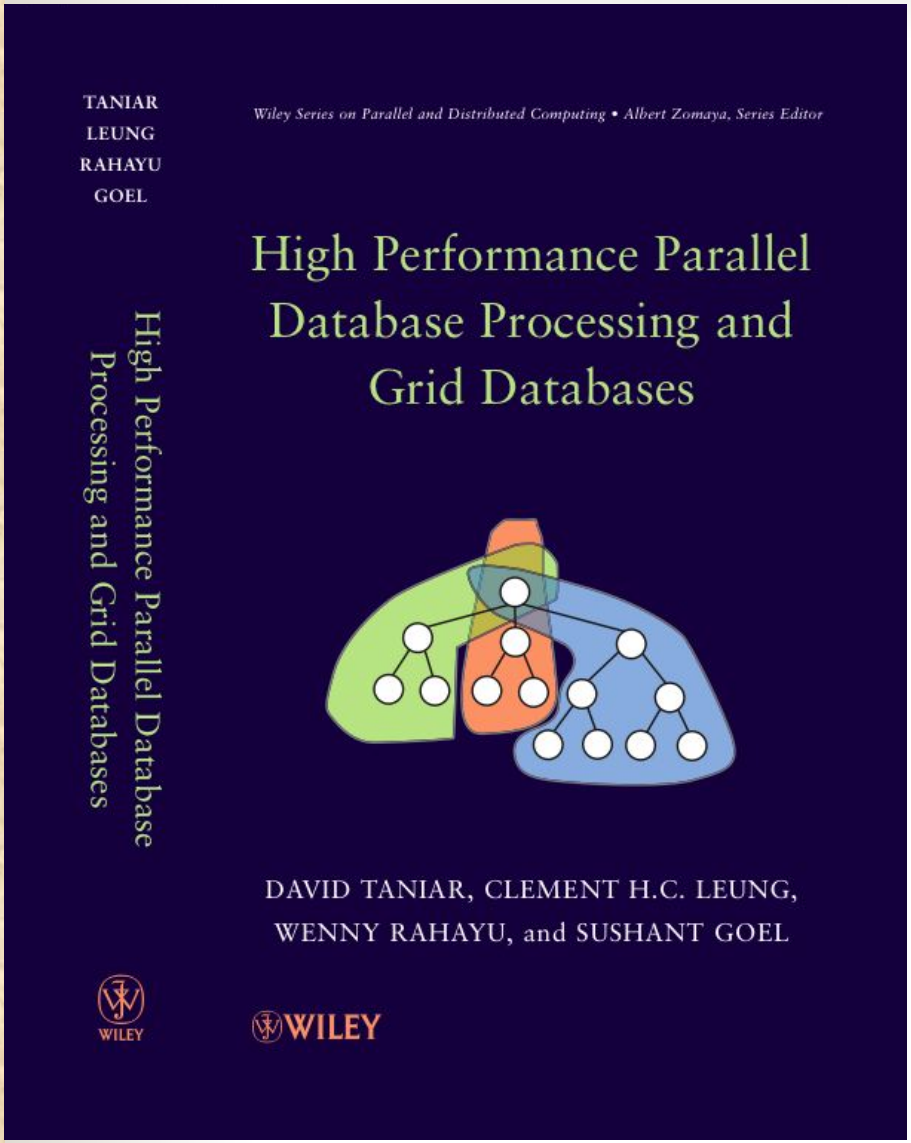
		Search Attribute Values	
		Unique	**Duplicate**
Exact Match		Stop	Continue
Range Selection	**Continuous**	Continue	Continue
	Discrete	Continue	Continue

3.4. Summary

- Search queries in SQL using the WHERE clause
- Search predicates indicates the type of search operation
  - Exact-match, range (continuous or discrete), or multiattribute search
- Data partitioning is a basic mechanism of parallel search
  - Single attribute-based, no attribute-based, or multiattribute-based partitioning
- Parallel search algorithms have three main components
  - Processor involvement, local searching method, and key comparison

Continue to Chapter 4...





Chapter 4
Parallel Sort and GroupBy

4.1. Sorting, Duplicate Removal and Aggregate

- 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;
```

- Basic aggregate queries:
  - Scalar aggregates - produce a single value for a given table
  - Aggregate functions - produce a set of values

Query 4.4:

```
Select MAX(Sgpa)
From STUDENT;
```

4.1. Sorting, Duplicate Removal and Aggregate (cont'd)

- **GroupBy**
  - Groups by specific attribute(s) and performs an aggregate function for each group

Query 4.6:

```
Select Sdegree, AVG(SAge)
From STUDENT
Group By Sdegree;
```

4.2. Serial External Sorting

- 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 sorted = 108 pages, number of buffer = 5 pages
  - Number of subfiles = 108/5 = 22 subfiles (the last subfile is only 3 pages long). Read, sort and write each subfile
  - Pass 0 (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

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

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

4.3. Parallel External Sort (cont'd)

- **Parallel Merge-All Sort**
  - 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

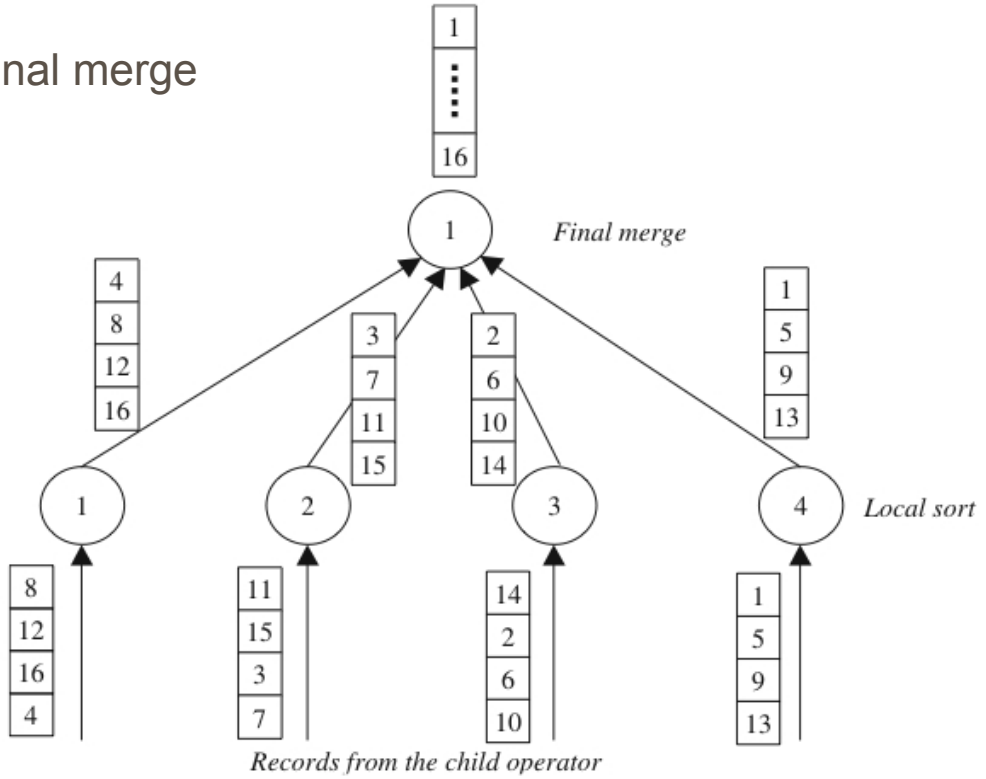


Figure 4.3 Parallel merge-all sort

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

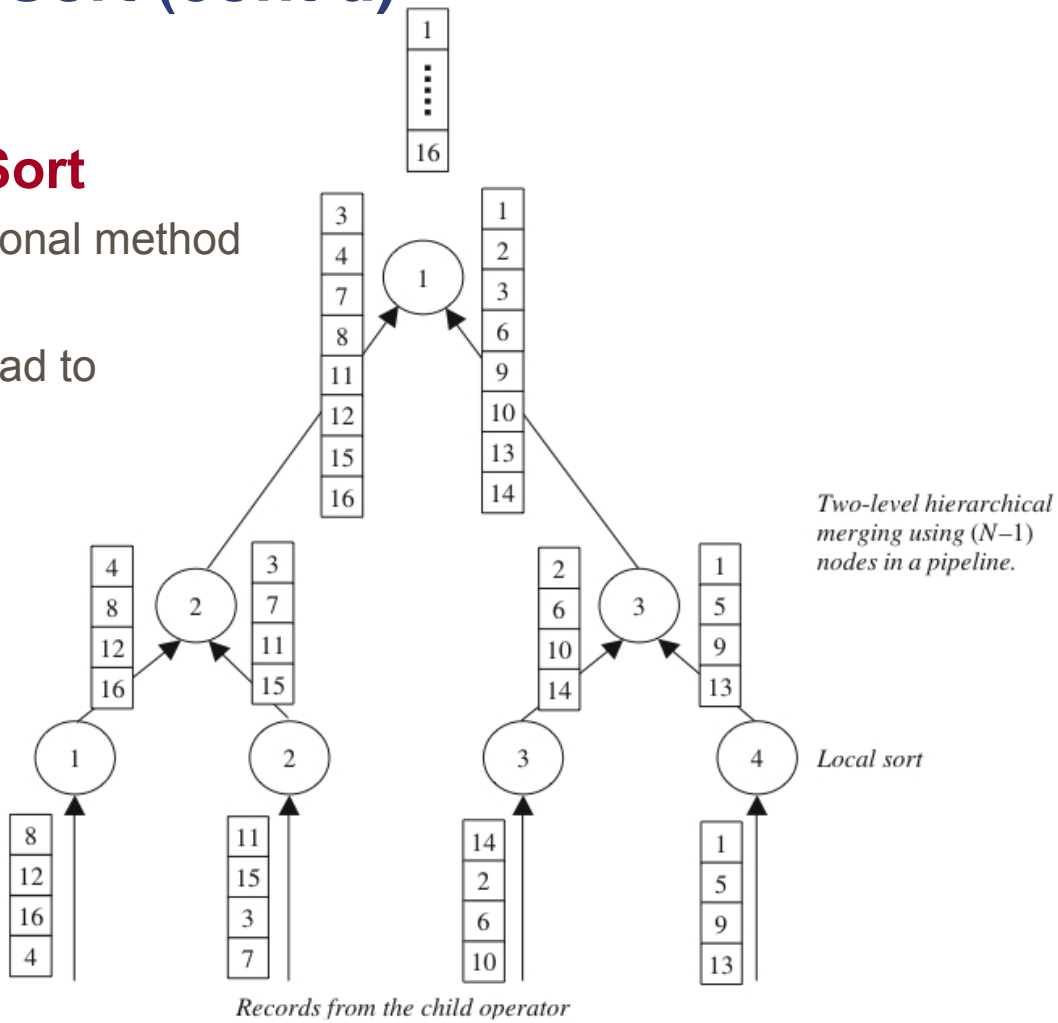


Figure 4.4 Parallel binary-merge sort

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

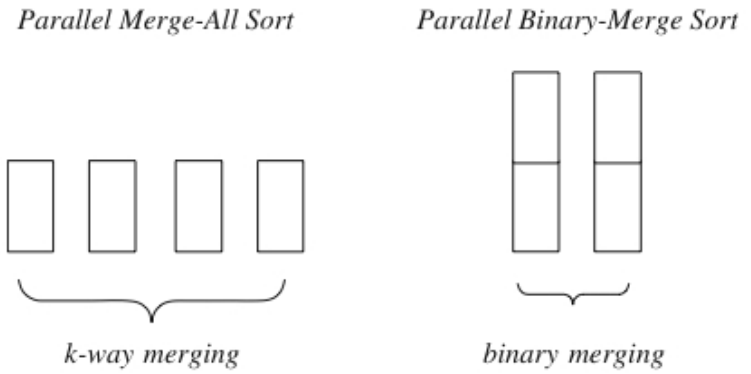


Figure 4.5 Binary-merge vs. *k*-way merge in the merging phase

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

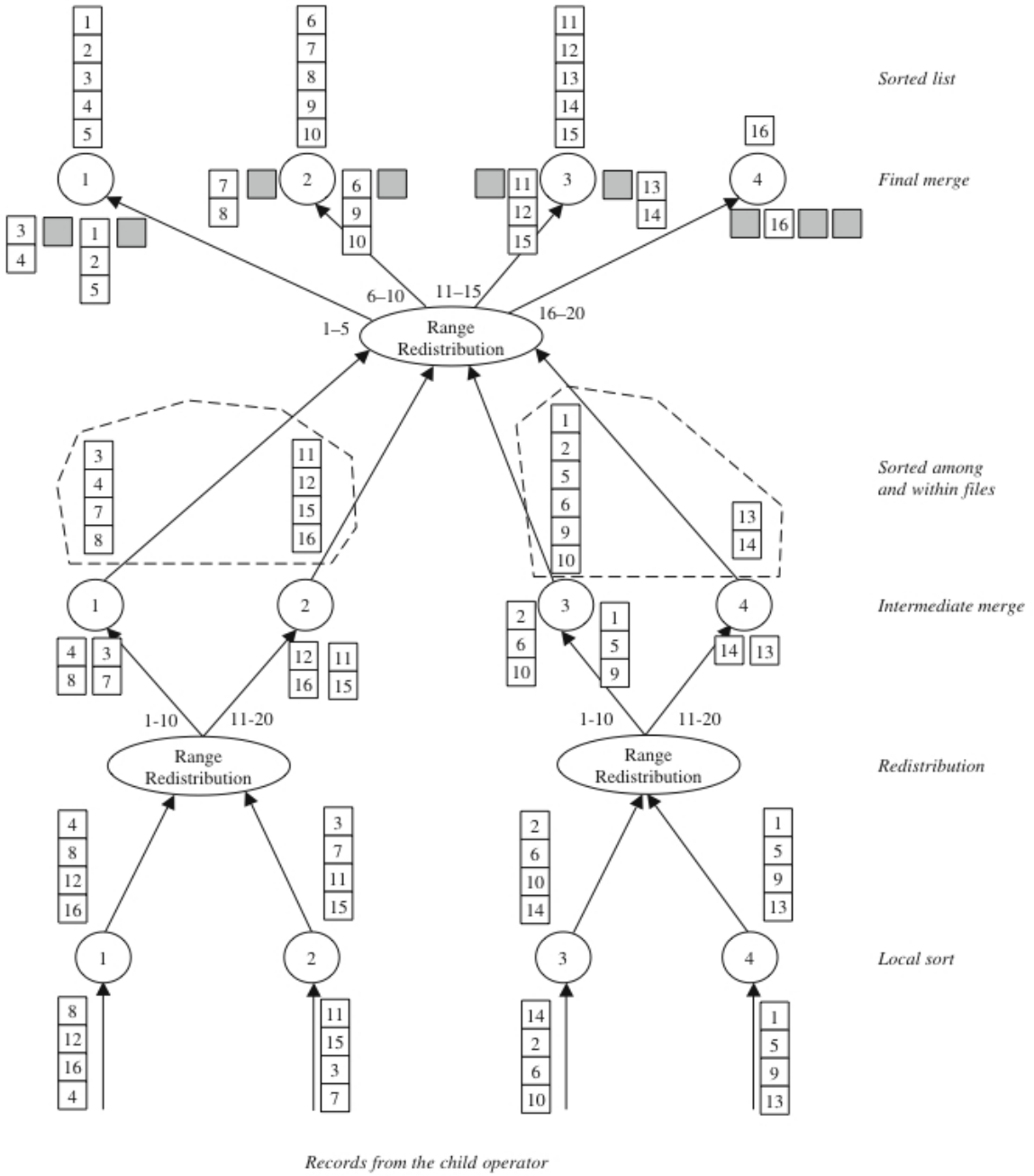


Figure 4.6 Parallel redistribution binary-merge sort

Parallel Redistribution Merge-All Sort

- Reduce the height of the tree, and still maintain parallelism
- Like parallel merge-all sort, but with redistribution
- The advantage is true parallelism in merging
- Skew problem in the merging

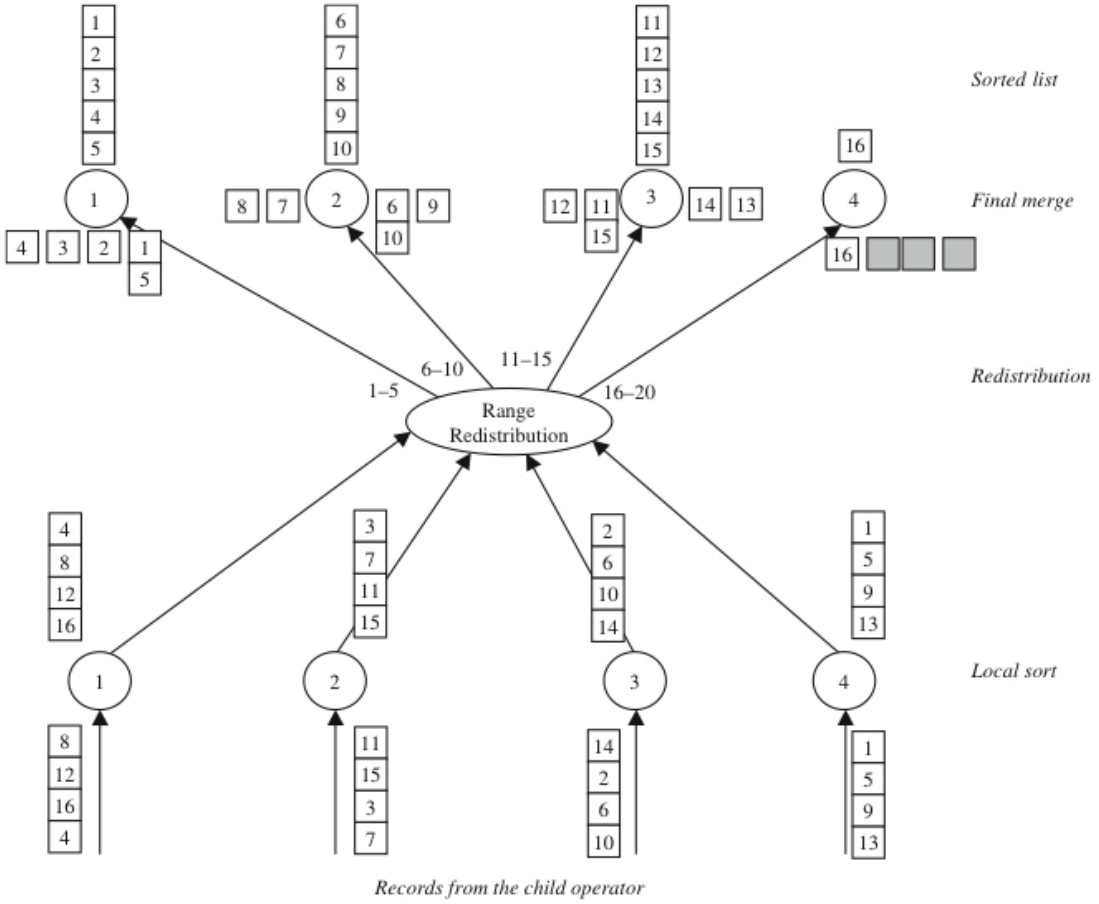


Figure 4.7 Parallel redistribution merge-all sort

Parallel Partitioned Sort

- Two stages: Partitioning stage and Independent local work
- Partitioning (or range redistribution) may raise load skew
- Local search is done after the partitioning, not before
- No merging is necessary
- Main problem: **Skew** produced by the partitioning

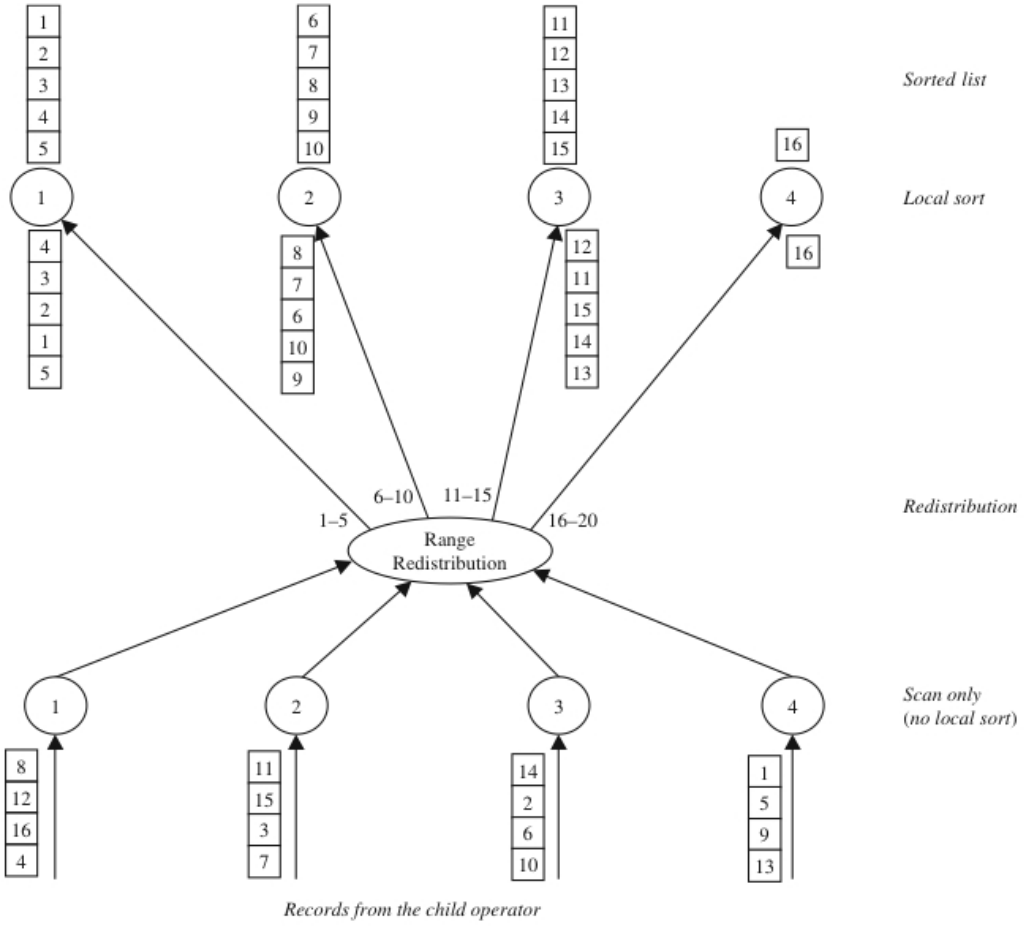


Figure 4.8 Parallel partitioned sort

Parallel Partitioned Sort

- Bucket tuning: produce more buckets than the available processors
- Bucket tuning does not work in parallel sort, because in parallel sort, the order of processor is important
- Bucket tuning for load balancing will later be used in parallel join

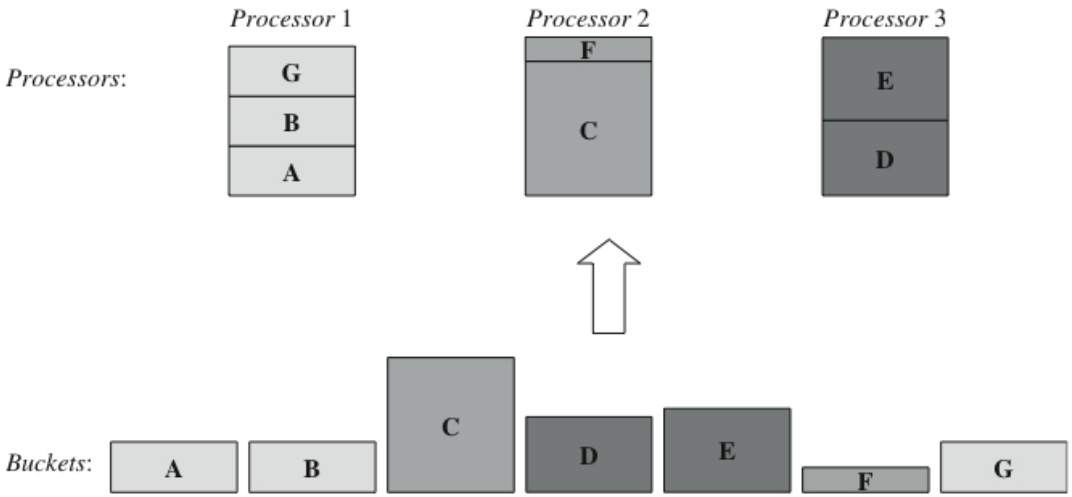


Figure 4.9 Bucket tuning load balancing

4.4. Parallel GroupBy

- Traditional methods (Merge-All and Hierarchical Merging)
- Two-phase method
- Redistribution method

4.4. Parallel GroupBy (cont'd)

- **Traditional Methods**
  - 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

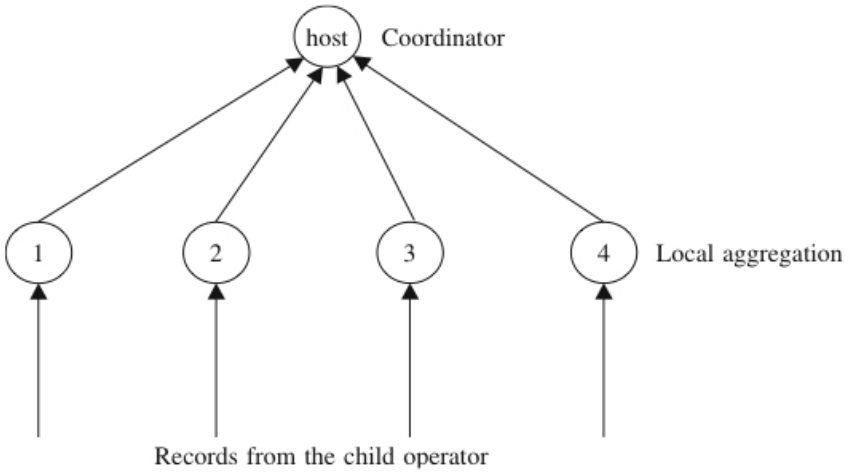


Figure 4.10 Traditional method

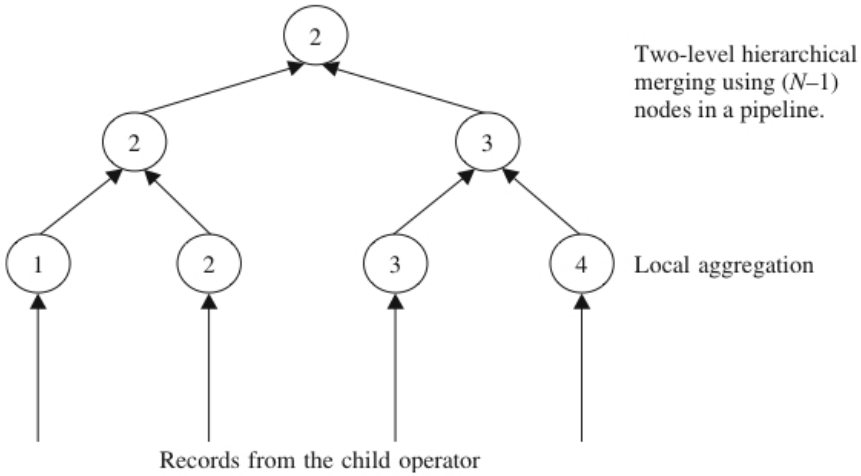


Figure 4.11 Hierarchical merging method

4.4. Parallel GroupBy (cont'd)

- **Two-Phase Method**
  - 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

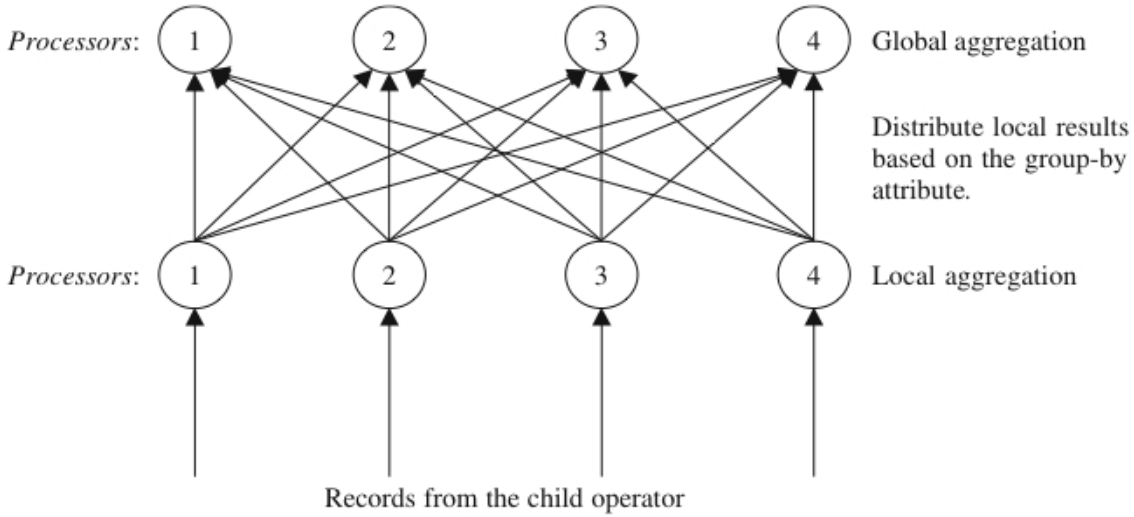


Figure 4.12 Two-phase method

4.4. Parallel GroupBy (cont'd)

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

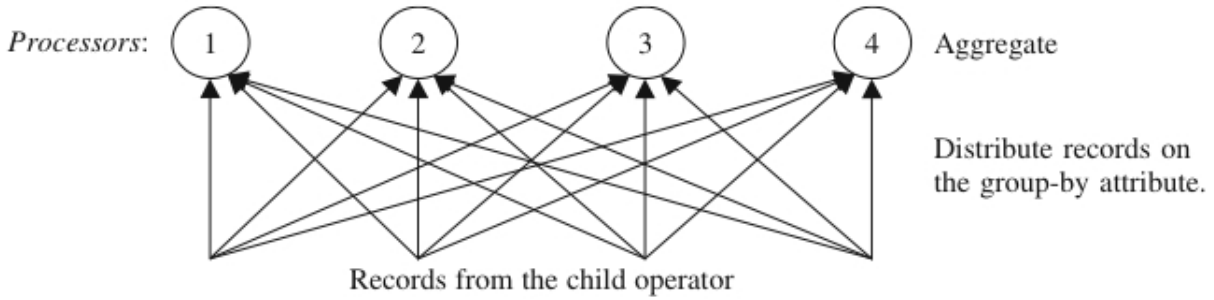


Figure 4.13 Redistribution method

4.5. Cost Models for Parallel Sort

- Additional cost notations

Table 4.2 Additional cost notations for parallel sort

Symbol	Description
System parameters	
B	Buffer size
Time unit costs	
t_m	Time to merge
t_s	Time to compare and swap two keys
t_v	Time to move a record

4.5. Cost Models for Parallel Sort (cont'd)

- **Serial External Merge-Sort**
  - **I/O cost components** are load and save costs
  - **Load cost** is the cost for loading data from disk to main memory

$$Load\ cost = Number\ of\ pages \times Number\ of\ passes \times Input/output\ unit\ cost$$

where $Number\ of\ pages = (R/P)$ and

$$Number\ of\ passes = (\lceil log_{B-1}(R/P/B) \rceil + 1) \qquad (4.1)$$

Hence, the above load cost becomes:

$$(R/P) \times (\lceil log_{B-1}(R/P/B) \rceil + 1) \times IO$$

  - **Save cost** is the cost of writing data from the main memory to the disk, which is identical to load cost equation

4.5. Cost Models for Parallel Sort (cont'd)

- **Serial External Merge-Sort**
  - **CPU cost components** are select, sorting, merging, and generation result costs
  - **Select cost** is the cost for obtaining a record from the data page

$$|R| \times Number\ of\ passes \times (t_r + t_w)$$

  - **Sorting cost** is the internal sorting cost which has $O(N \times \log_2 N)$

$$|R| \times \lceil log_2(|R|) \rceil \times t_s$$

  - **Merging cost** is applied to pass 1 onward

$$|R| \times (Number\ of\ passes - 1) \times t_m$$

  - **Generating result cost** is determined by the number of records being generated or produced in each pass before they are written to disk multiplied

$$|R| \times Number\ of\ passes \times t_w$$

4.5. Cost Models for Parallel Sort (cont'd)

- **Parallel Merge-All Sort**
  - **Local merge sort** costs are I/O costs, CPU costs, and Communication costs
  - **I/O costs** consist of load and save costs

$$Save\ cost = Load\ cost = (R_i / P) \times Number\ of\ passes \times IO \qquad (4.2)$$

  - **CPU costs** consist of select, sorting, merging and generating results cost

$$Select\ cost = |R_i| \times Number of passes \times (t_r + t_w)$$
$$Sorting\ cost = |R_i| \times \lceil log_2(|R_i|) \rceil \times t_s$$
$$Merging\ cost = |R_i| \times (Number of passes - 1) \times t_m$$
$$Generating\ result\ cost = |R_i| \times Number of passes \times t_w$$

where *Number of passes* is as shown in equation 4.2 above.

  - **Communication costs** for sending local sorted results to the host:

$$Communication\ cost = (R_i / P) \times (m_p + m_l)$$

4.5. Cost Models for Parallel Sort (cont'd)

- **Parallel Merge-All Sort**
  - **Final merging** costs are communication, I/O, and CPU costs
  - **Communication cost** is the receiving cost from local sorting operators

$$Communication\ cost = (R/P) \times m_p$$

  - **I/O cost** is the load and save costs

$$Save\ cost = (R/P) \times (Number\ of\ merging\ passes + 1) \times IO$$
$$Load\ cost = (R/P) \times Number\ of\ merging\ passes \times IO \qquad (4.3)$$

$$where\ Number\ of\ merging\ passes = \lceil \log_{B-1}(N) \rceil$$

  - **CPU cost** is the select, merging, and generating results costs

$$Select\ cost = |R| \times Number\ of\ merging\ passes \times (t_r + t_w)$$
$$Merging\ cost = |R| \times Number\ of\ merging\ passes \times t_m$$
$$Generating\ result\ cost = |R| \times Number\ of\ merging\ passes \times t_w$$

where *Number of merging passes* is as shown in equation 4.3 above.

4.5. Cost Models for Parallel Sort (cont'd)

- **Parallel Binary-Merge Sort**
  - The costs consist of local merge-sort costs, and pipeline merging costs
  - The local merge-sort costs are exactly the same as those of parallel merge-all sort, since the local sorting phase in both methods is the same
  - Hence, focus on pipeline merging costs
  - In pipeline merging, we need to determine the number of levels, which is $\log_2(N)$
  - In level 1, the number of processors used is up to half ($N'=N/2$)
  - The skew equation is then: $|R'_i| = \frac{|R|}{\sum_{j=1}^{N'} \frac{1}{j^{\theta}}}$

4.5. Cost Models for Parallel Sort (cont'd)

- **Parallel Binary-Merge Sort**
  - Costs for level 1:

$$
\begin{aligned}
Receiving\ cost &= (R_i'/P) \times m_p \\
Save\ cost &= (R_i'/P) \times IO \\
Load\ cost &= (R_i'/P) \times IO \\
Select\ cost &= |R_i'| \times (t_r + t_w) \\
Merging\ cost &= |R_i'| \times t_m \\
Generating\ result\ cost &= |R_i'| \times t_w \\
Data\ transfer\ cost &= (R_i'/P) \times (m_p + m_l)
\end{aligned}
$$

  - where *R'* indicates the number of records being processed at a node in a level of pipeline merging

4.5. Cost Models for Parallel Sort (cont'd)

- **Parallel Binary-Merge Sort**
  - In the subsequent levels, the number of processors is further reduced by half. The new *N'* value becomes *N' = N' / 2*. This also impact the skew equation
  - At the last level of pipeline merging, the host performs a final binary merging, where *N'* = 1
  - The total pipeline binary merging costs are:

$$\begin{aligned}
Receiving\ cost &= (R_i'/P) \times \lceil log_2(N) \rceil \times m_p \\
Save\ cost &= (R_i'/P) \times (\lceil log_2(N) \rceil + 1) \times IO \\
Load\ cost &= (R_i'/P) \times \lceil log_2(N) \rceil \times IO \\
Select\ cost &= |R_i'| \times \lceil log_2(N) \rceil \times (t_r + t_w) \\
Merging\ cost &= |R_i'| \times \lceil log_2(N) \rceil \times t_m \\
Generating\ result\ cost &= |R_i'| \times \lceil log_2(N) \rceil \times t_w \\
Data\ transfer\ cost &= (R_i'/P) \times (\lceil log_2(N) \rceil - 1) \times (m_p + m_l)
\end{aligned}$$

  - The values of R'_l and $|R'_l|$ are not constant throughout the pipeline, but increase from level to level as the number of processors *N'* is reduced by half when progressing from one level to another

4.5. Cost Models for Parallel Sort (cont'd)

- **Parallel Redistribution Binary-Merge Sort**
  - Local merge-sort costs, and pipeline merging costs
  - Local sort operation is similar to the previous two parallel sorts, but the temp results are being redistributed, which incurs additional overhead
  - The compute destination cost is:

$$Compute\ destination\ cost = |R_i| \times t_d$$

    where R_i may involve data skew
  - Pipeline merging costs are also similar to the those without redistribution
  - Differences: number of processors involved in each level, where all processors participate. Hence we use R_i and $|R_i|$, and not R'_i and $|R'_i|$; and the compute destination cost are applicable to all levels in the pipeline

4.5. Cost Models for Parallel Sort (cont'd)

- **Parallel Redistribution Binary-Merge Sort**
  - The pipeline merging costs are:

$$
\begin{aligned}
\textit{Receiving cost} &= (R_i/P) \times \lceil log_2(N) \rceil \times m_p \\
\textit{Save cost} &= (R_i/P) \times (\lceil log_2(N) \rceil + 1) \times IO \\
\textit{Load cost} &= (R_i/P) \times \lceil log_2(N) \rceil \times IO \\
\textit{Select cost} &= |R_i| \times \lceil log_2(N) \rceil \times (t_r + t_w) \\
\textit{Merging cost} &= |R_i| \times \lceil log_2(N) \rceil \times t_m \\
\textit{Generating result cost} &= |R_i| \times \lceil log_2(N) \rceil \times t_w \\
\textbf{\textit{Compute destination cost}} &= |\boldsymbol{R_i}| \times (\lceil \boldsymbol{log_2(N)} \rceil - 1) \times \boldsymbol{t_d} \\
\textit{Data transfer cost} &= (R_i/P) \times (\lceil log_2(N) \rceil - 1) \times (m_p + m_l)
\end{aligned}
$$

4.5. Cost Models for Parallel Sort (cont'd)

- **Parallel Redistribution Merge-All Sort**
  - Local merge-sort costs and merging costs
  - Local merge-sort costs are the same as those of parallel redistribution binary-merge sort with compute destination costs
  - Merging costs are similar to those of parallel merge-all sort, except one main difference. Here we use R_i and $|R_i|$, not R and $|R|$
  - The merging costs are then:

$$\begin{aligned}
Communication\ cost &= (R_i/P) \times m_p \\
Save\ cost &= (R_i/P) \times (Number\ of\ merging\ passes + 1) \times IO \\
Load\ cost &= (R_i/P) \times Number\ of\ merging\ passes \times IO \\
Select\ cost &= |R_i| \times Number\ of\ merging\ passes \times (t_r + t_w) \\
Merging\ cost &= |R_i| \times Number\ of\ merging\ passes \times t_m \\
Generating\ result\ cost &= |R_i| \times (Number\ of\ merging\ passes) \times t_w
\end{aligned}$$

where $Number\ of\ merging\ passes = \lceil log_{B-1}(N) \rceil$

4.5. Cost Models for Parallel Sort (cont'd)

- **Parallel Partitioned Sort**
  - Scanning/partitioning costs, and local merge-sort costs
  - Scanning and partitioning costs involve I/O, CPU, and communication costs
  - I/O costs consist of load cost: $(R_i / P) \times IO$
  - CPU costs consist of select costs: $|R_i| \times (t_r + t_w + t_d)$
  - Communication costs consist of data transfer costs: $(R_i / P) \times (m_p + m_l)$

4.5. Cost Models for Parallel Sort (cont'd)

- **Parallel Partitioned Sort**
  - The local merge-sort costs are similar to other local merge-sort costs, except communication costs are associated with data received from the first phase
  - Communication cost for receiving data:

$$Data\ receiving\ costs = (R_i/P) \times m_p$$

  - I/O costs which are load and save costs:

$$Save\ cost = (R_i/P) \times (Number\ of\ passes + 1) \times IO$$
$$Load\ cost = (R_i/P) \times Number\ of\ passes \times IO \qquad (4.4)$$

where $Number\ of\ passes = (\lceil log_{B-1}(R_i/P/B) \rceil + 1)$

  - CPU costs are:

$$Select\ cost = |R_i| \times Number\ of\ passes \times (t_r + t_w)$$
$$Sorting\ cost = |R_i| \times \lceil log_2(|R_i|) \rceil \times t_s$$
$$Merging\ cost = |R_i| \times (Number\ of\ passes - 1) \times t_m$$
$$Generating\ result\ cost = |R_i| \times Number of passes \times t_w$$

where *Number of passes* is as shown in equation 4.4

4.6. Cost Models for Parallel GroupBy

- Additional cost notations

Table 4.3 Cost notations

Symbol	Description
Query parameters	
σ_p	Selectivity ratio of local aggregate in a processor
σ_n	Selectivity ratio of local aggregate in a node
σ_g	Selectivity ratio of global aggregate
Time unit costs	
t_h	Time to compute hash value
t_a	Time to add a record to current aggregate value

4.6. Cost Models for Parallel GroupBy (cont'd)

- **Parallel Two-Phase Method**
  - **Phase 1: Local aggregation**
  - Scan cost: $(R_i / P) \times IO$
  - Select cost: $|R_i| \times (t_r + t_w)$
  - Local aggregation cost: $|R_i| \times (t_r + t_h + t_a)$
  - Reading/Writing of overflow buckets:

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

  - Generating result records cost: $|R_i| \times \sigma_p \times t_w$
  - Determining the destination cost: $|R_i| \times \sigma_p \times t_d$
  - Data transfer cost: $(\pi \times R_i \times \sigma_p / P) \times (m_p + m_l)$

4.6. Cost Models for Parallel GroupBy (cont'd)

- **Parallel Two-Phase Method**
  - **Phase 2: Consolidation (Merging)**
  - The number of records arriving at a processor:

$$|R_i| \times \sigma_p \quad \text{and} \quad \pi \times R_i \times \sigma_p$$

  - The first term is the number of selected records from the 1st phase
  - The second term is the table size of the selected records

  - Receiving records cost: $(\pi \times R_i \times \sigma_p / P) \times (m_p)$
  - Computing final aggregation value cost: $|R_i| \times \sigma_p \times (t_r + t_a)$
  - Generating final result cost: $|R_i| \times \sigma_g \times t_w$
  - Disk cost for storing the final result: $(\pi \times R_i \times \sigma_g / P) \times IO$

4.6. Cost Models for Parallel GroupBy (cont'd)

- **Parallel Redistribution Method**
  - **Phase 1: Distribution/Partitioning**
  - The scan and selection costs are the same as for those in the two-phase method
  - Scan cost: $(R_i / P) \times IO$
  - Select cost: $|R_i| \times (t_r + t_w)$

  - Apart from these two costs, the finding destination cost and the data transfer cost are added to this model
  - Finding destination cost: $|R_i| \times (t_d)$
  - Data transfer cost: $(\pi \times R_i / P) \times (m_p + m_l)$

  - If the number of groups is less than the number of processors, then $R_i = R$ / (*Number of groups*), instead of $R_i = R / N$

4.6. Cost Models for Parallel GroupBy (cont'd)

- **Parallel Redistribution Method**
  - **Phase 2: Aggregation**
  - Receiving records cost: $(\pi \times R_i / P) \times (m_p)$
    Only selected attributes are involved (π)
  - Computing aggregation cost: $|R_i| \times (t_r + t_h + t_a)$

    It does not include π, because we take into account the number of records, not the record size
  - Reading/Writing of overflow buckets cost:

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

    where s is the overall GroupBy selectivity ratio ($\sigma = \sigma_p$ x σ_g)
  - Generating final result cost: $|R_i| \times \sigma \times t_w$
  - Disk cost: $(\pi \times R_i \times \sigma / P) \times IO$

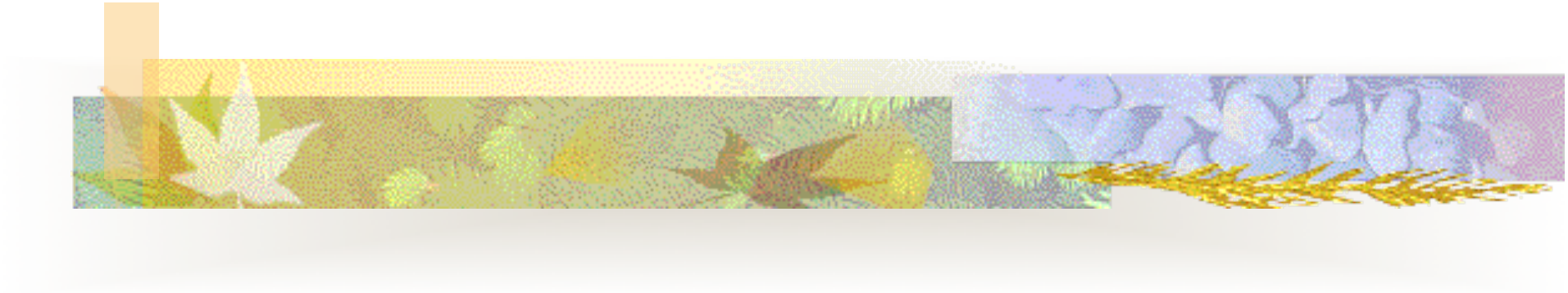
4.7. Summary

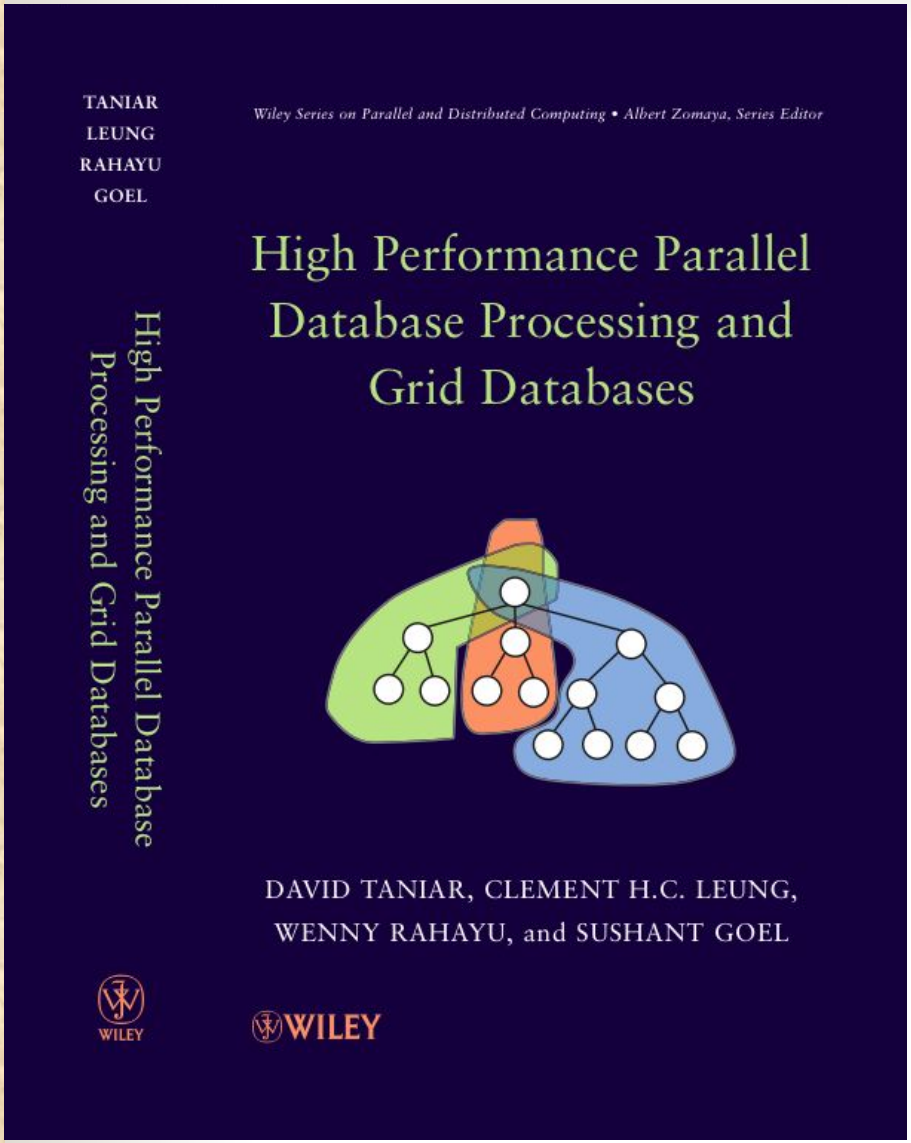
- Sorting and duplicate removal are expressed in ORDER BY and DISTINCT in SQL
- Parallel algorithms for database sorting
  - Parallel merge-all sort, parallel binary-merge sort, parallel redistribution binary-merge sort, parallel redistribution merge-all sort, and parallel partitioned sort
- Cost models for each parallel sort algorithm
  - Buffer size
- Parallel redistribution algorithm is prone to processing skew
  - If processing skew degree is high, then use parallel redistribution merge-all sort.
  - If both data skew and processing skew degrees are high or no skew, then use parallel partitioned sort

4.7. Summary (cont'd)

- Parallel groupby algorithms
  - Traditional methods (merge-all and hierarchical methods)
  - Two-phase method
  - Redistribution method
- 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

Continue to Chapter 5...





Chapter 5
Parallel Join

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;
```

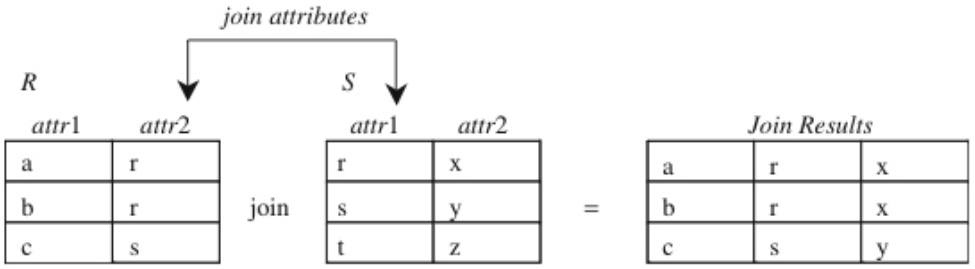


Figure 5.1 The join operation

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*
  - If there are *N* records in table *R* and *M* records in table *S*, the efficiency of a nested-loop join algorithm is *O*(*NM*)

Algorithm: Nested-loop join

```
Input: Tables R and S
Output: Query Result Qr
1. Let Qr = {}
2. For each record of table R
3.    Read record from table R
4.    For each record of table S
5.       Read record from table S
6.       Compare the join attributes
7.       If matched Then
8.          Store the records into Qr
```

Figure 5.3 Nested-loop 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

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

```
Algorithm: Sort-merge join
Input: Tables R and S
Output: Query Result Qr
1.  Let Qr = {}
2.  Sort records of table R based on the join attribute
3.  Sort records of table S based on the join attribute
4.  Let i = 1 and j = 1
5.  Repeat
6.    Read record R(i)
7.    Read record S(j)
8.    If join attribute R(i) < join attribute S(j) Then
9.      i++
10.   Else
11.     If join attribute R(i) > join attribute S(j) Then
12.       j++
13.     Else
14.       Put records R(i) and S(j) into the Qr
15.       i++; j++
16.   If either R(i) or S(j) is EOF Then
17.     Break
```

Figure 5.5. Sort-Merge join algorithm

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)

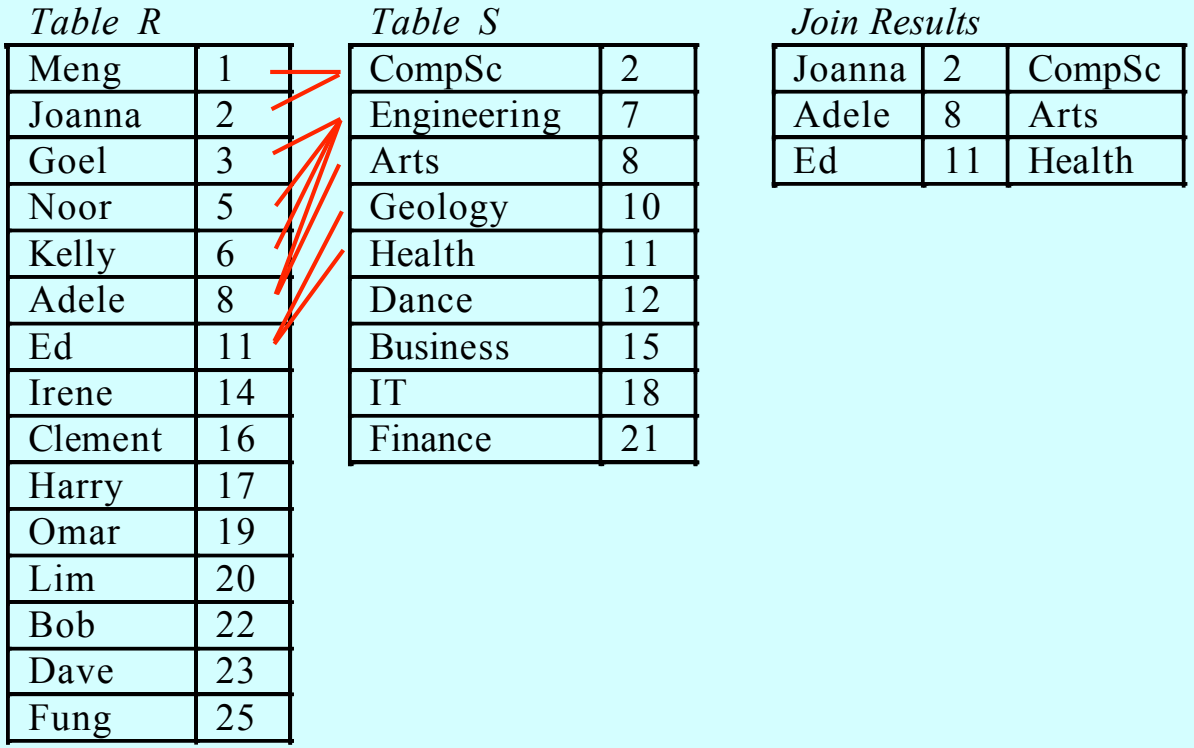


Table R

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)

```
Algorithm: Hash-based join
Input: Tables R and S
Output: Query Result Qr
1.  Let Qr = {}
2.  Let H be a hash function
3.  For each record in table S
4.    Read a record from table S
5.    Hash the record based on join attribute value using
      hash function H into hash table
6.  For each record in table R
7.    Read a record from table R
8.    Hash the record based on join attribute value using H
9.    Probe into the hash table
10.   If an index entry is found Then
11.     Compare each record on this index entry with
        the record of table S
12.     If matched Then
13.       Put the pair into Qr
```

Figure 5.8. Hash-based join algorithm

5.2. Serial Join Algorithms (cont'd)

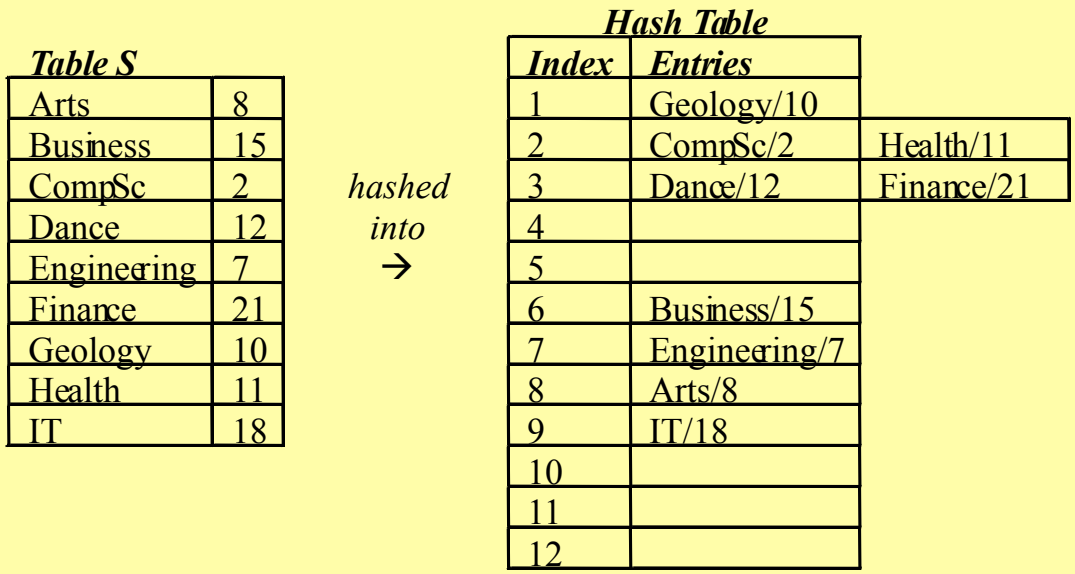


Table S

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

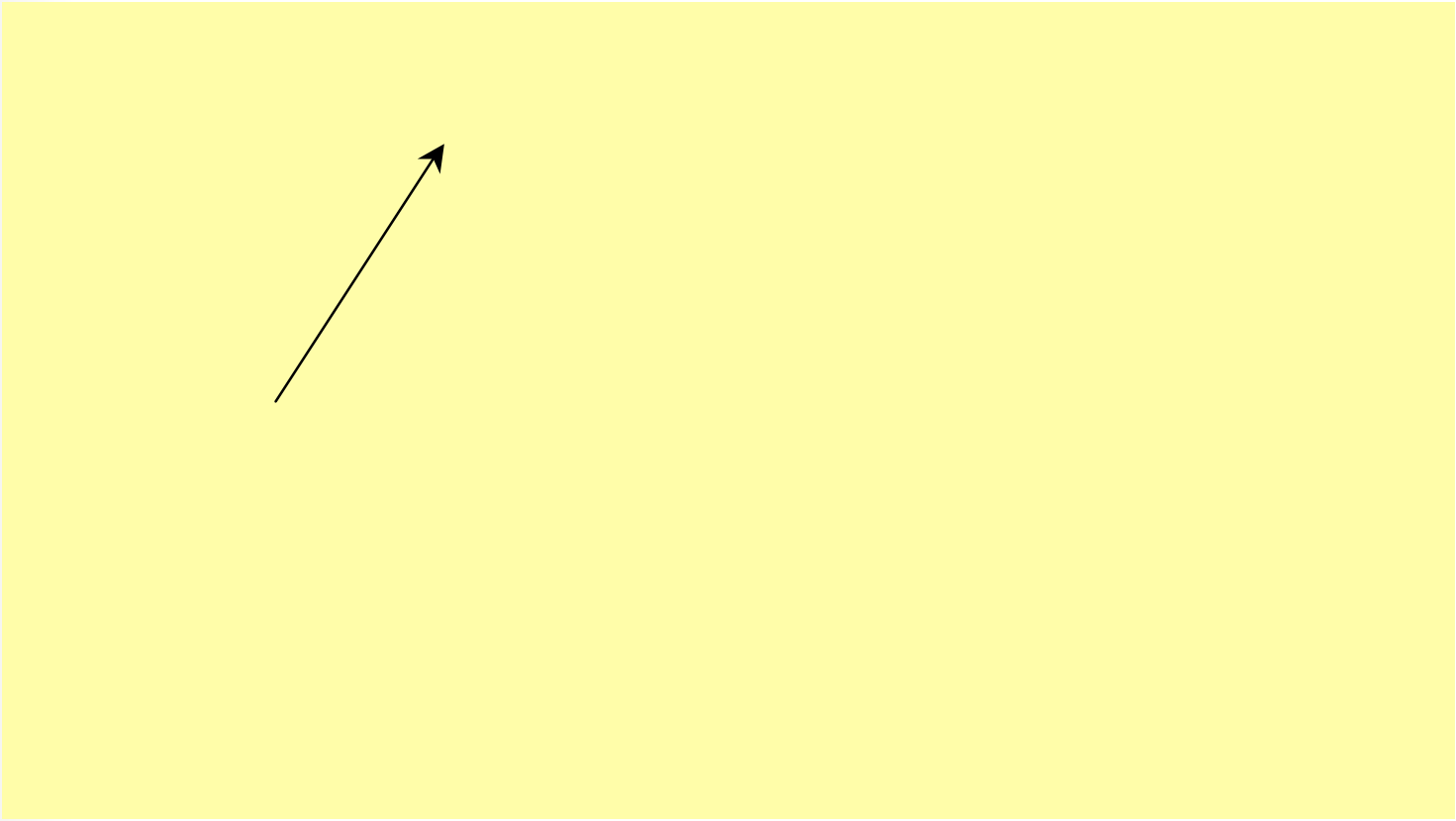
hashed into →

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)



5.2. Serial Join Algorithms (cont'd)

- **Comparison**
  - The complexity of join algorithms is normally dependent on the number of times that a disk scan needs to be performed
  - Nested-loop join algorithm = $O(NM)$
  - Sort-merge join algorithm = $O(N\log N + M\log M + N + M)$
  - Hash-based join algorithm = $O(N + M)$

N	M	*O(NM)*	*O(NlogN+MlogM+N+M)*	*O(N+M)*
10	10	100	40	20
100	100	10,000	600	200
1000	1000	1,000,000	8000	2000
10,000	10,000	100,000,000	100,000	20,000
100,000	100,000	10,000,000,000	1,200,000	200,000
1,000,000	1,000,000	1,000,000,000,000	14,000,000	2,000,000

Figure 5.9 Complexity comparison of the three serial join algorithms

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

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 disjoin 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	
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)

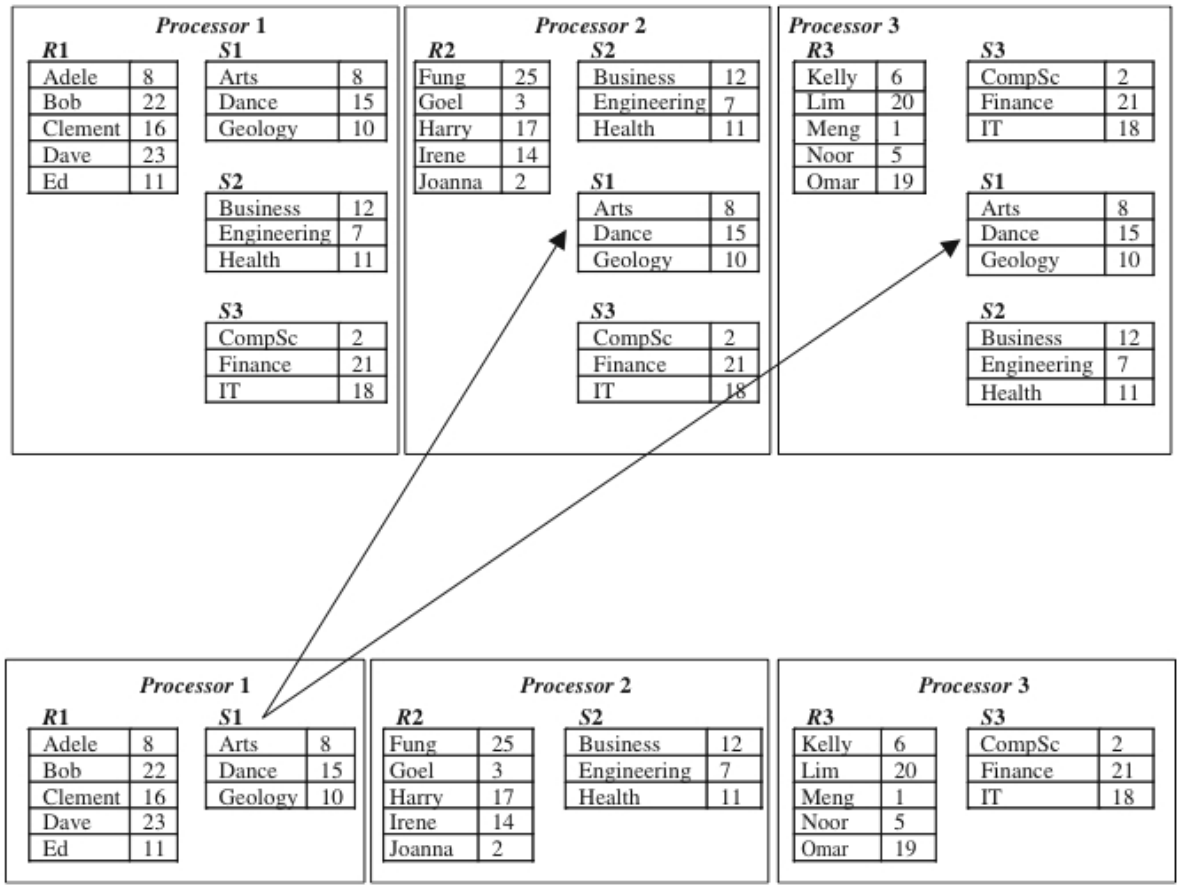


Figure 5.11 Divide and broadcast result

5.3. Parallel Join Algorithms (cont'd)

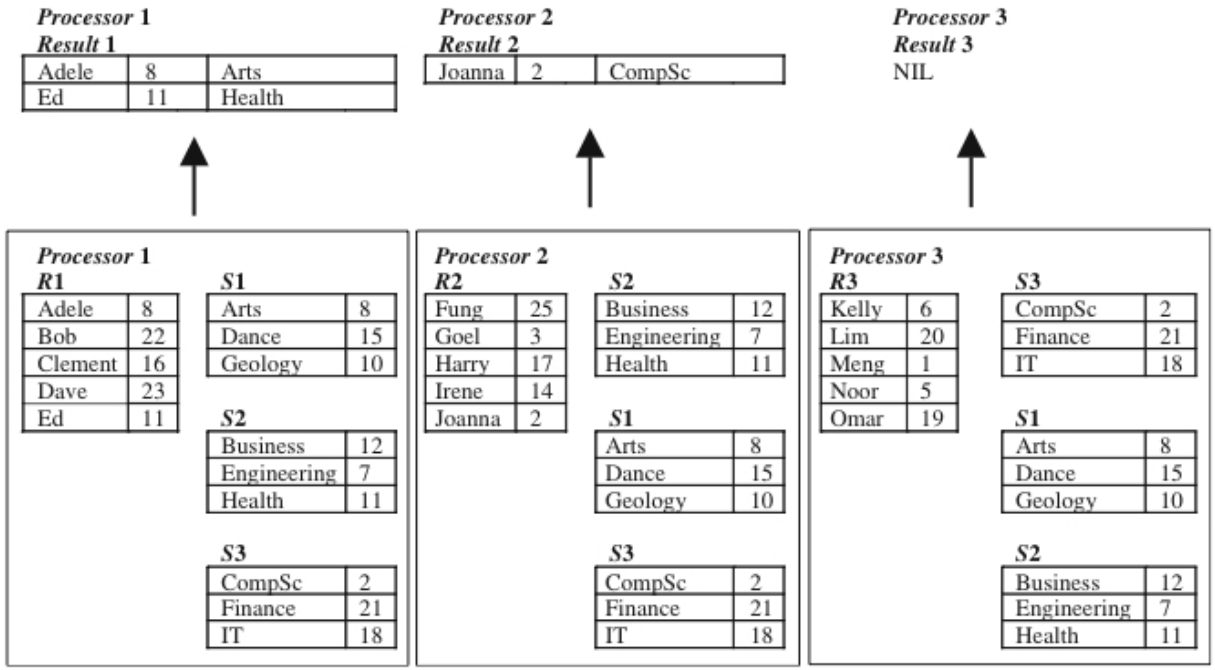


Figure 5.12 Join results based on divide and broadcast

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: Range partitioning**

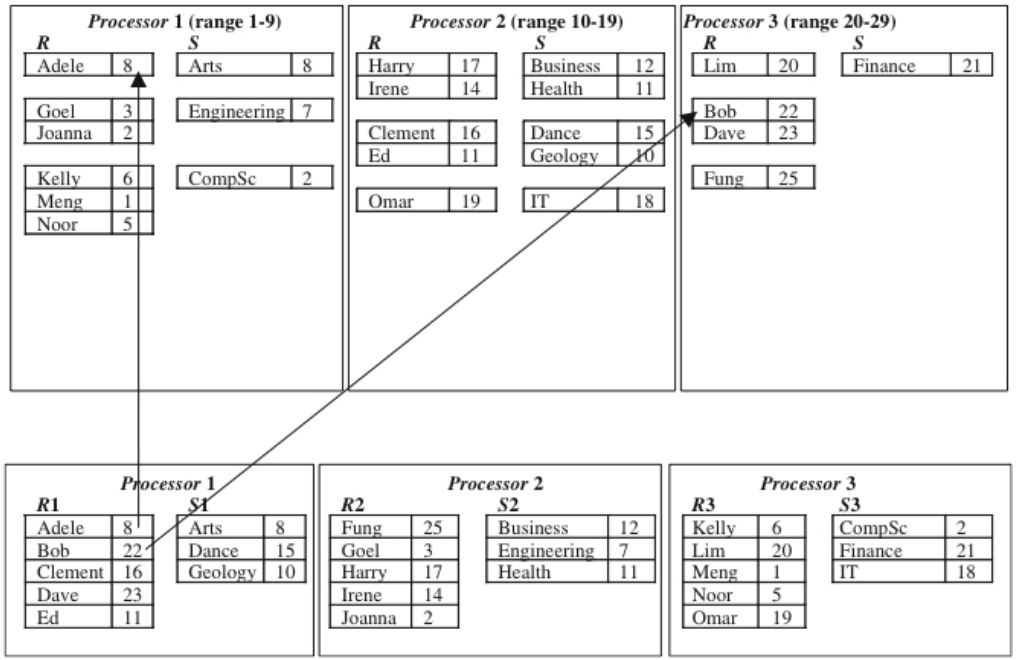


Figure 5.14 Range partitioning

5.3. Parallel Join Algorithms (cont'd)

- **Example 1: Range partitioning**

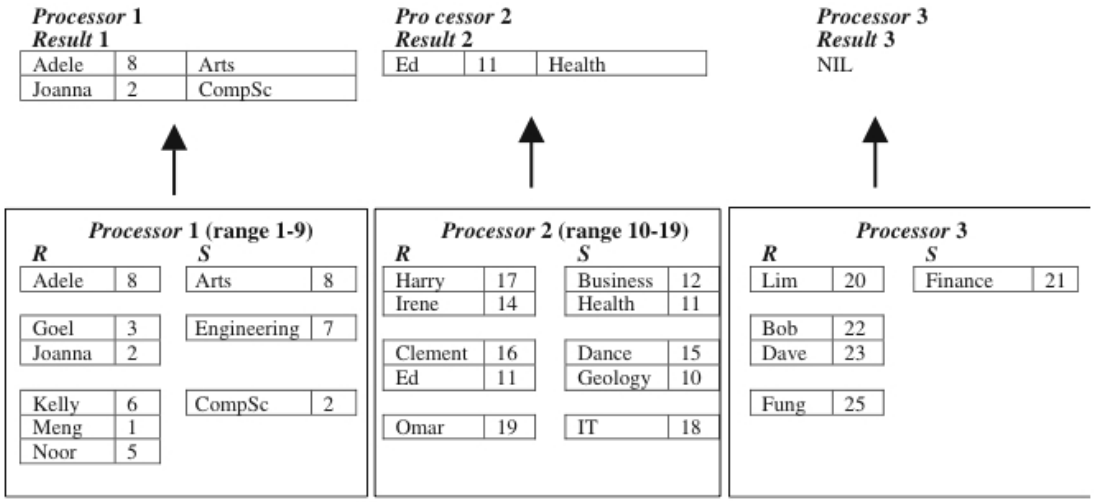


Figure 5.15 Join results based on range partitioning

5.3. Parallel Join Algorithms (cont'd)

- **Example 2: Hash partitioning**

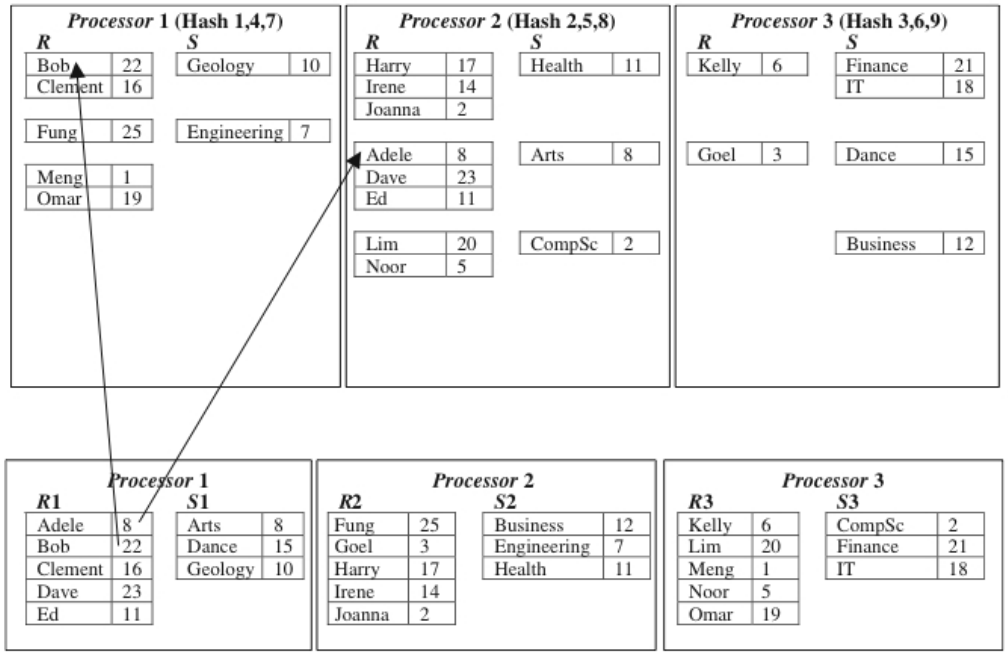


Figure 5.16 Hash partitioning

5.3. Parallel Join Algorithms (cont'd)

- **Example 2: Hash partitioning**

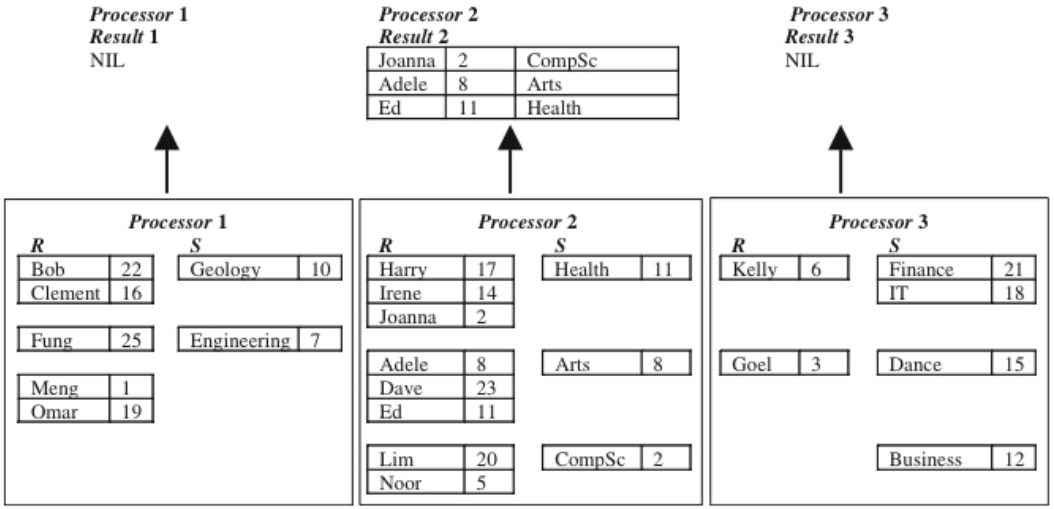


Figure 5.17 Join results based on hash partitioning

5.4. Cost Models for Parallel Join

- **Cost Models for Divide and Broadcast**
  - Assume the tables have already been partitioned and placed in each processor
  - The cost components for the broadcasting process has three phases
  - Phase 1: data loading
  - Phase 2: data broadcasting
  - Phase 3: data storing

5.4. Cost Models for Parallel Join

- **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:

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

5.4. Cost Models for Parallel Join

- **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 - Si) x (mp + ml), where (S - S_i) is the size of other fragments.
  - *Receiving records cost* is: (S - Si) x (mp)

5.4. Cost Models for Parallel Join

- **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* - *Si*) x *IO*

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

- **Cost Models for Disjoint Partitioning**
  - Three main cost components: loading costs, distribution costs, and storing costs
  - The loading costs include scan costs and select costs
  - *Scan cost* for loading tables R and S from local disk in each processor is: $((Ri / P) + (Si / P)) \times IO$
  - *Select cost* for getting record out of data page is: $(|Ri| + |Si|) \times (tr + tw)$

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

- **Cost Models for Disjoint Partitioning**
  - The distribution costs contains: the cost of determining the destination of each record, the actual sending and receiving costs
  - *Finding destination* cost is: (|*Ri*| + |*Si*|) x (*td*)
  - *Data transfer* cost is: ((*Ri* / *P*) + (*Si* / *P*)) x (*mp* + *ml*)
  - *Receiving records* cost is: ((*Ri* / *P*) + (*Si* / *P*)) x (*mp*)

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

- **Cost Models for Disjoint Partitioning**
  - Finally, the last phase is the data storing which involves storing all records received by each processor
  - *Disk cost* for storing the result of data distribution is: ((*Ri* / *P*) + (*Si* / *P*)) x *IO*

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

- **Cost Models for Local 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*) + (*Si* / *P*)) x *IO*
  - *Select* cost = (|*Ri*| + |*Si*|) x (*tr* + *tw*)

  - ($|R_i| + |S_i|$) and ((R_i / P) + (S_i / P)) correspond to the values in the receiving and disk costs of the disjoint partitioning

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*) + (|*Si*| 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.

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

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: $|Ri| \times \sigma j \times |Si| \times tw$
  - *Disk* cost for storing the final result is: $(\pi_R \times Ri \times \sigma j \times \pi_S \times Si / P) \times IO$

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:

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

  - And the local join scan cost is then reduced by *M* as well:

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

  - 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

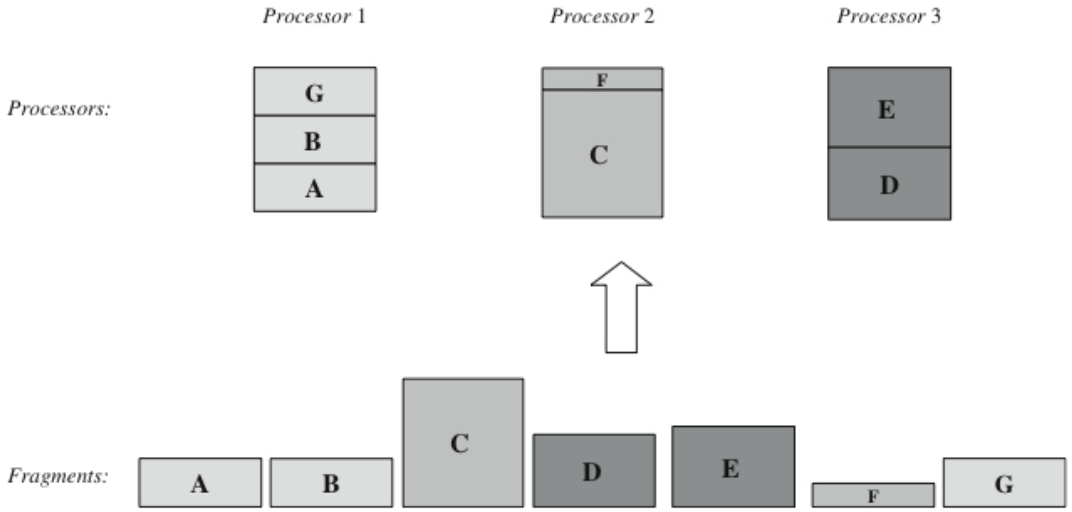
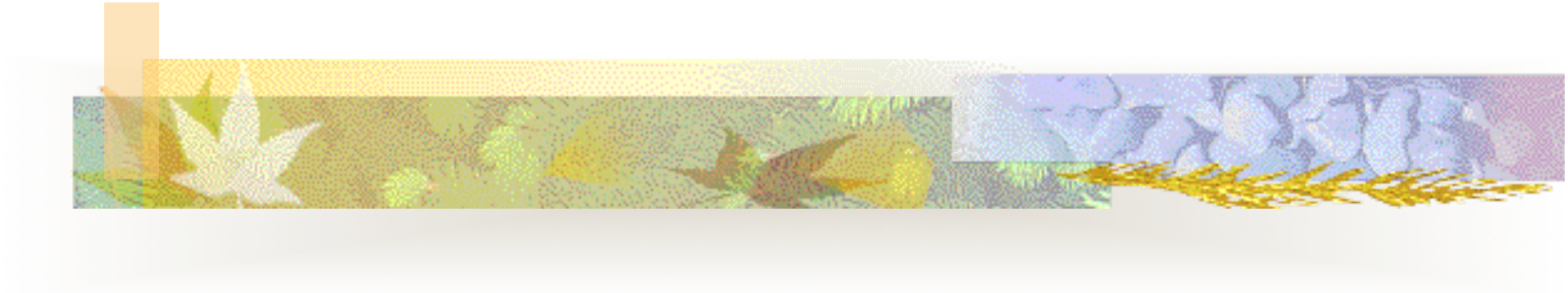


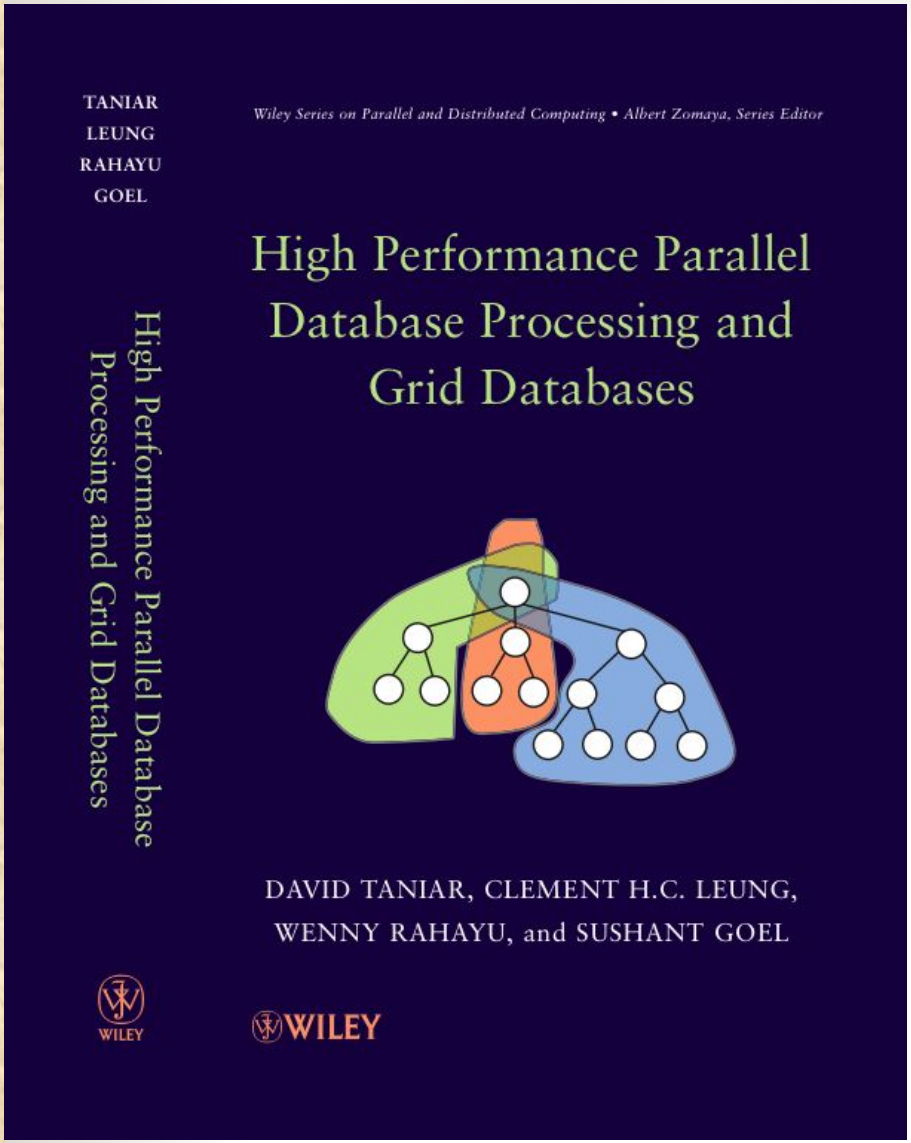
Figure 5.19 Load balancing

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

Continue to Chapter 6...





Chapter 7
Parallel Indexing

7.1. Parallel Indexing

- Index is an important element in databases
- Parallel index structure is essentially data partitioning. However, index partitioning is not as straightforward as table partitioning, because index is not flat like table
- B^+ tree is the most common indexing structure
  - Each non-leaf node may consist up to *k* keys and *k*+1 pointers to the nodes on the next level
  - The data is pointed by the leaf nodes
  - All child nodes which are on the left-hand side of the parent node, have key values less than or equal to the key on their parent node.
  - The keys of child nodes on the right-hand side of the parent node are greater than the key of their parent nodes

7.1. Parallel Indexing (cont'd)

Table (ID, Name):

23	Adams
65	Bernard
37	Chris
60	David
46	Eric
92	Fred
48	Greg
71	Harold
56	Ian
59	Johanna

18	Kathy
21	Larry
10	Mary
74	Norman
78	Oprah
15	Peter
16	Queenie
20	Ross
24	Susan
28	Tracey

39	Uma
43	Vera
47	Wenny
50	Xena
69	Yuliana
75	Zorro
8	Agnes
49	Bonnie
33	Caroline
38	Dennis

Index (B+ Tree):

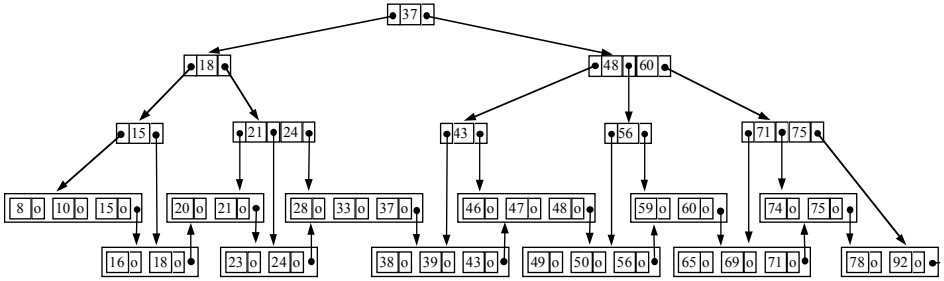


Figure 7.1. A Sample Table and Index

7.1. Parallel Indexing (cont'd)

- Three parallel indexing structures
  - Nonreplicated index (NRI)
  - Partially replicated index (PRI)
  - Fully replicated index (FRI)
- There are different variations to each parallel index, depending on two factors
  - Index partitioning attributes
  - Table partitioning attributes

	Indexed Attribute = Table Partitioning Attribute	*No Index Partitioning Attribute*	*Indexed Attribute ≠ Table Partitioning Attribute*
Non-Replicated Index NRI	**NRI-1**	**NRI-2**	**NRI-3**
Partially-Replicated Index PRI	**PRI-1**	**PRI-2**	**PRI-3**
Fully-Replicated Index FRI	**FRI-1**		**FRI-3**

Figure 7.2. Parallel Indexing Structures

7.2. Parallel Indexing Structures

- **Nonreplicated Indexing (NRI) Structures**
  - The global index is partitioned into several disjoint and smaller indices
  - Each of these small indices is placed in a separate processing element
  - **NRI-1**: the index partitioning attribute is the same as the table partitioning attribute

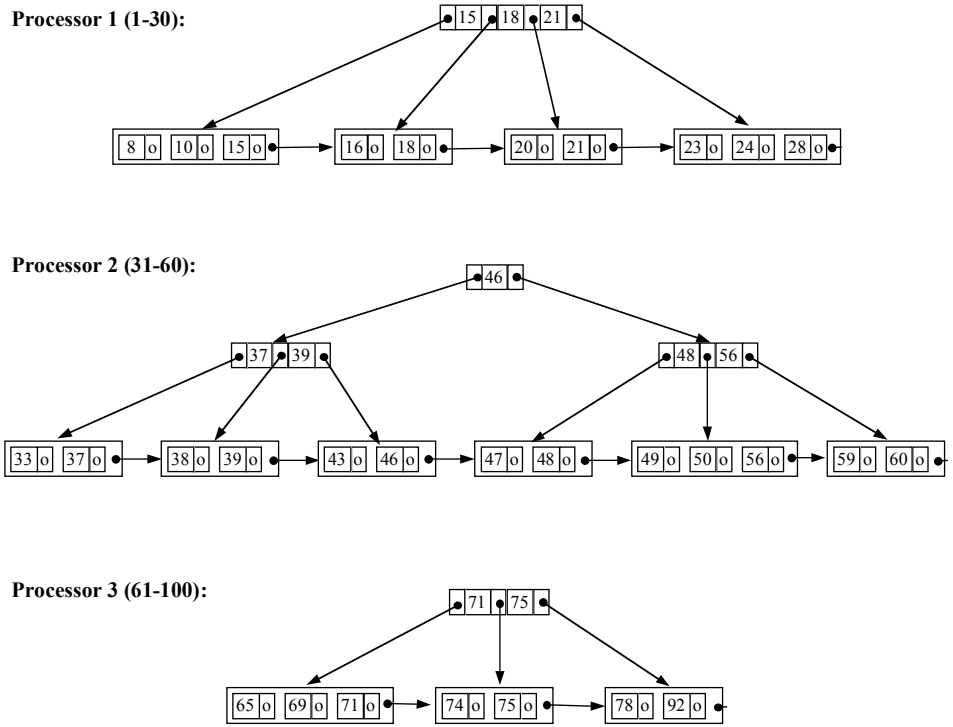
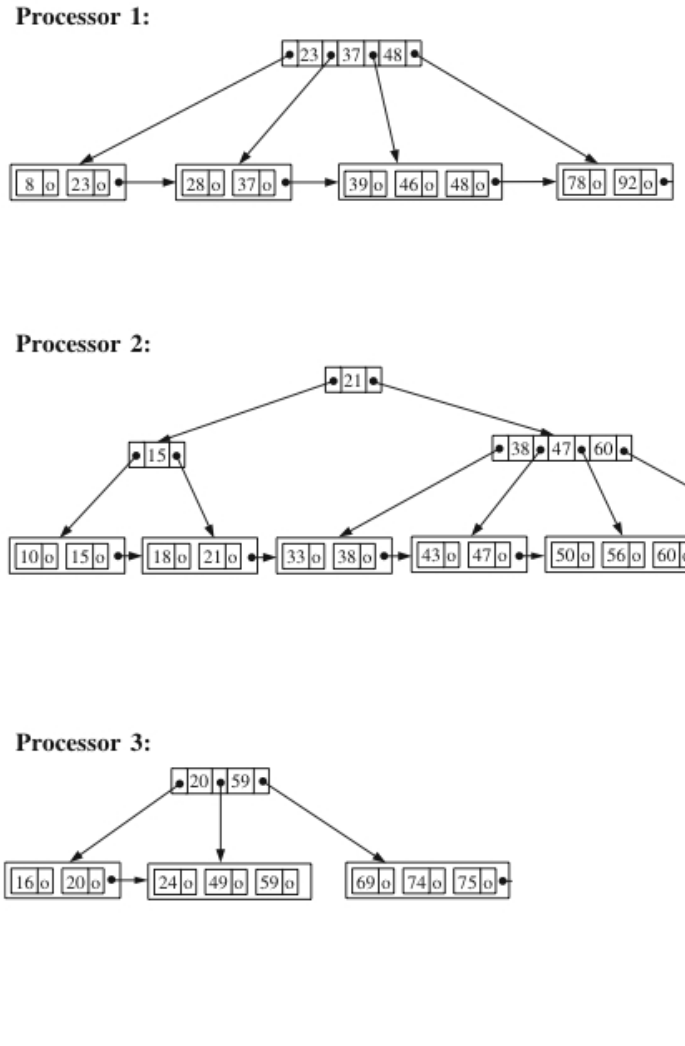


Figure 7.3. NRI-1 structure (index partitioning attribute = table partitioning attribute)

- **Nonreplicated Indexing (NRI) Structures**
  - **NRI-2**: the local indices are built on whatever data already exists in each processing element



Processor 1	
23	Adams
37	Chris
46	Eric
92	Fred
48	Greg
78	Oprah
28	Tracey
39	Uma
8	Agnes

Name: –x-------
***x** = consonant*

Processor 2	
65	Bernard
60	David
71	Harold
56	Ian
18	Kathy
21	Larry
10	Mary
15	Peter
43	Vera
47	Wenny
50	Xena
33	Caroline
38	Dennis

Name: -x-------
***x**= vowel (a,e)*

Processor 3	
59	Johanna
74	Norman
16	Queenie
20	Ross
24	Susan
69	Yuliana
75	Zorro
49	Bonnie

Name: –x-------
***x** = vowel(i,o,u)*

Figure 7.4 NRI-2 (no index partitioning attribute is used)

- **Nonreplicated Indexing (NRI) Structures**
  - **NRI-3**: the attribute used in index partitioning is different from that in data partitioning
  - Hence, the pointers from the leaf nodes to the actual record may cross to different processor, because the actual record is located at a different processor

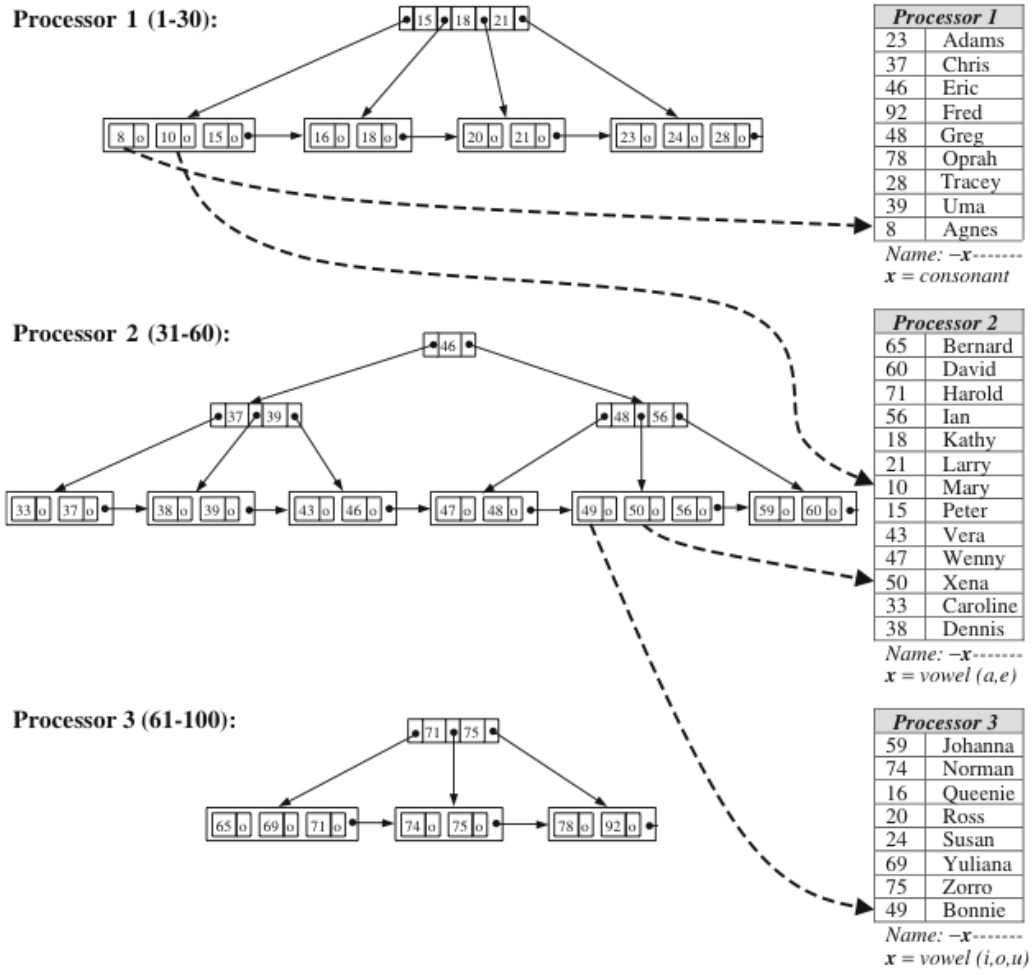


Figure 7.5 NRI-3 (index partitioning attribute ≠ table partitioning attribute)

7.2. Parallel Indexing Structures (cont'd)

- **Partially Replicated Index (PRI)**
  - Like NRI, there are three variants (PRI-1, PRI-2, and PRI-3), depending on index partitioning attributes and table partitioning attributes
  - In PRI, the global index is maintained and is not partitioned. Each processing element has a different part of the global index, and the overall structure of the index is preserved
  - Ownership rule: Processor owning a leaf node also owns all nodes from the root to that leaf. Hence, the root node is replicated to all processors, and non-leaf nodes may be replicated to some processors
  - If a leaf node has several keys belonging to different processors, this leaf node is also replicated to the processors owning the keys

7.2. Parallel Indexing Structures (cont'd)

- **PRI-1**
  - Index partitioning attribute = table partitioning attribute

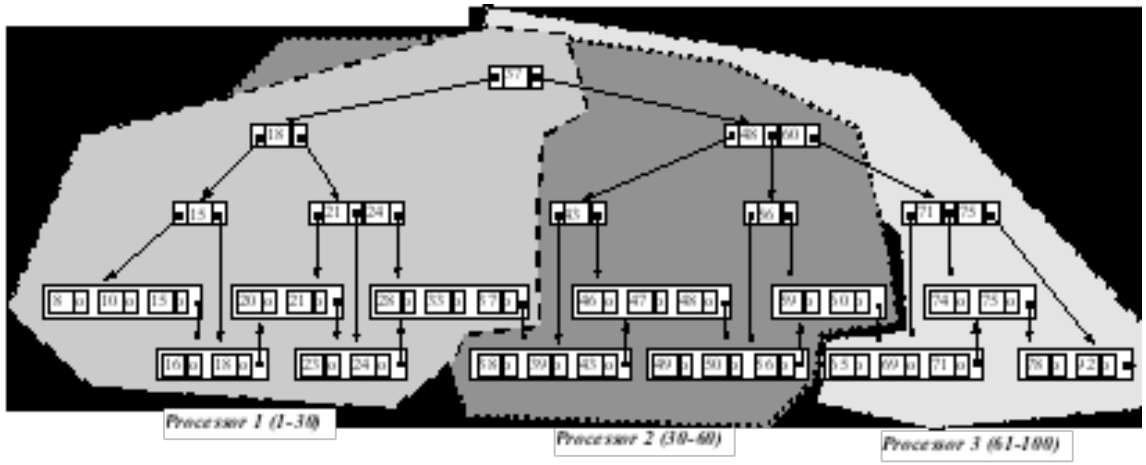


Figure 7.6. PRI-1 (index partitioning attribute = table partitioning attribute)

7.2. Parallel Indexing Structures (cont'd)

- **PRI-1 implementation**
  - Multiple node pointers model - impractical

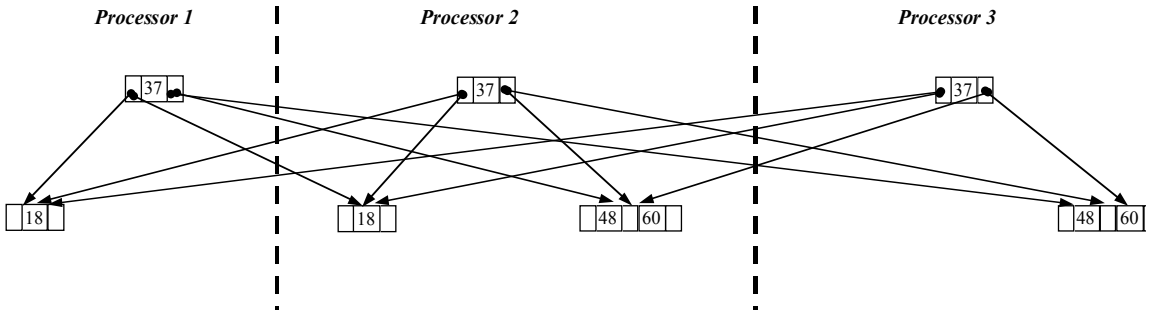


Figure 7.7. Multiple Node Pointers Model for PRI

7.2. Parallel Indexing Structures (cont'd)

- **PRI-1 implementation**
  - Single node pointer model

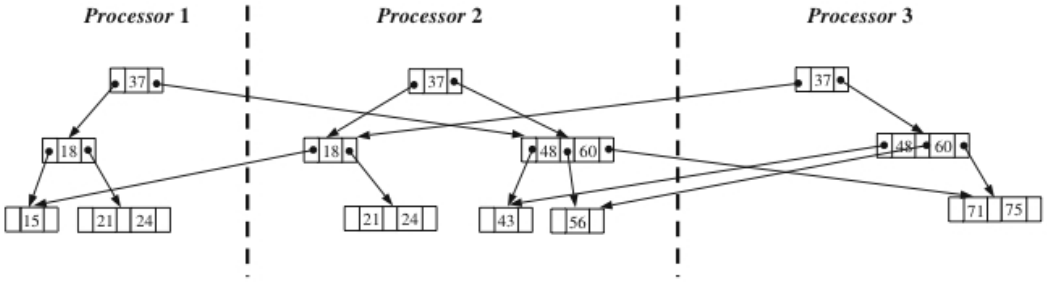


Figure 7.8 Single node pointer model for PRI

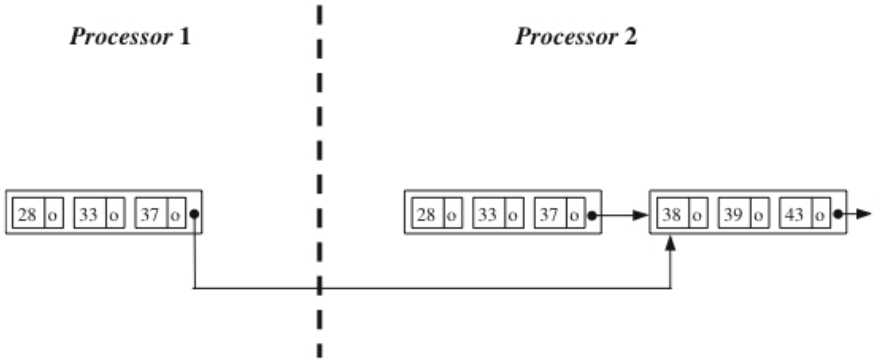


Figure 7.9 Node pointers at a leaf node level crossing from one processor to another

7.2. Parallel Indexing Structures (cont'd)

- **PRI-2**
  - No index partitioning is used

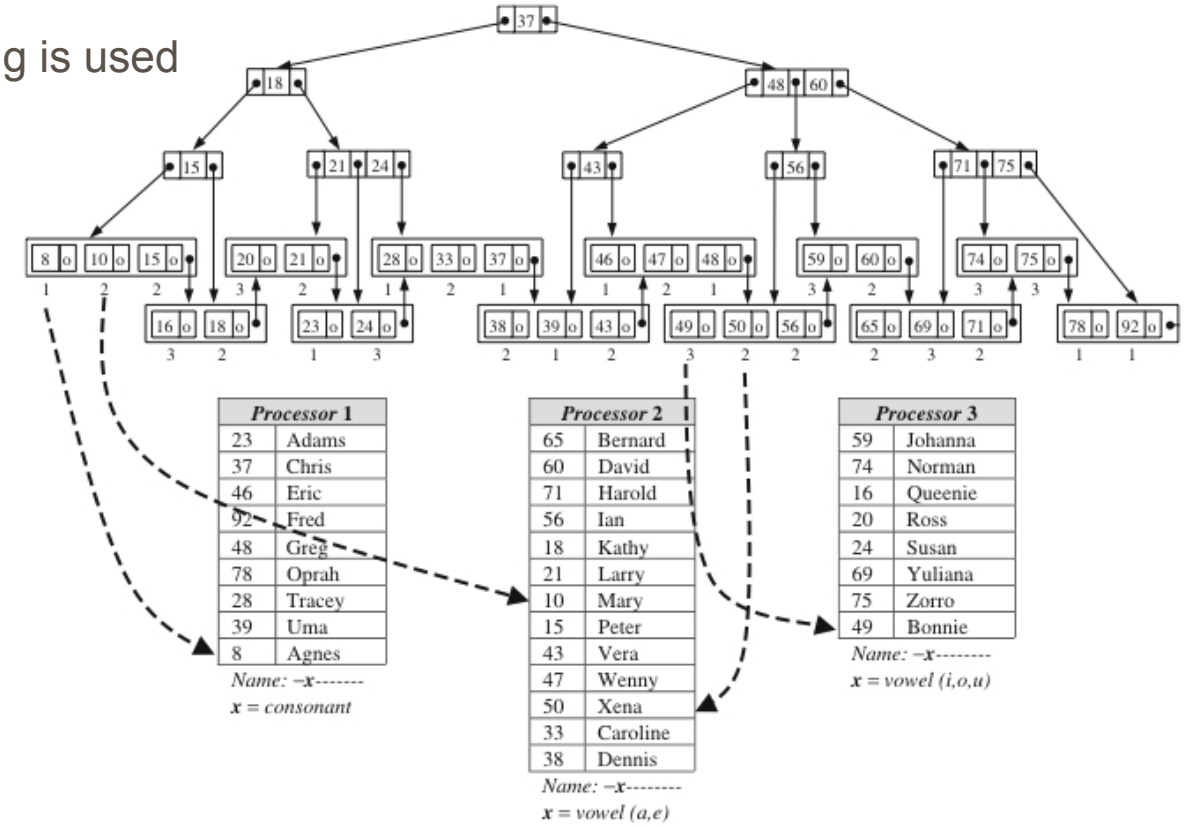


Figure 7.10(a) PRI-2 (no index partitioning attribute is used)

7.2. Parallel Indexing Structures (cont'd)

- **PRI-2**
  - No index partitioning is used

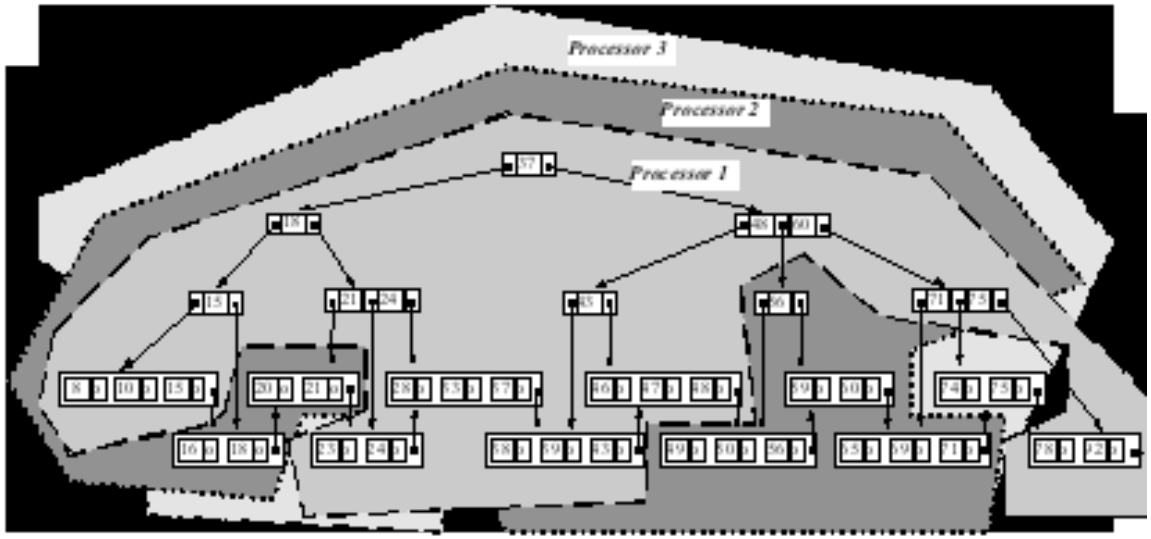


Figure 7.10(b). Replication in PRI-2

7.2. Parallel Indexing Structures (cont'd)

- **PRI-3**
  - Index partitioning attribute ≠ table partitioning attribute

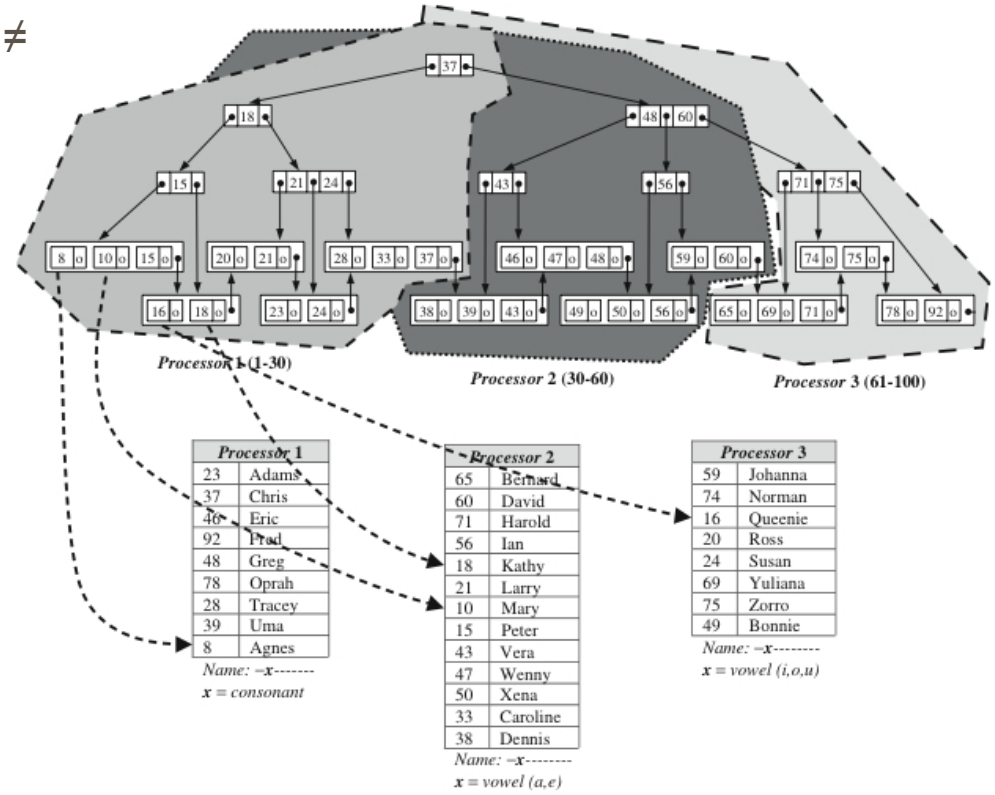


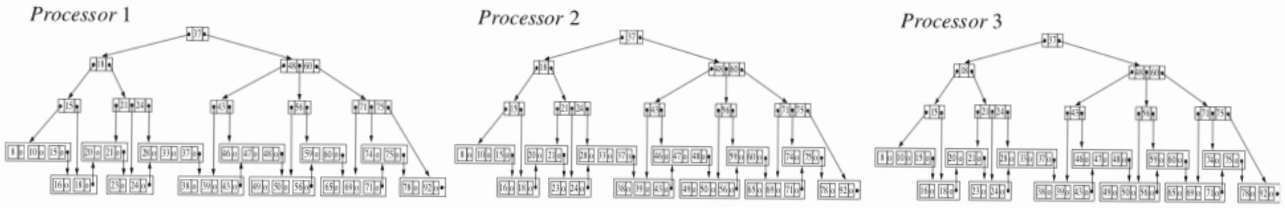
Figure 7.11 PRI-3 (index partitioning attribute ≠ table partitioning attribute)

7.2. Parallel Indexing Structures (cont'd)

- **Fully Replicated Index (FRI)**
  - The entire global index is replicated to all processors
  - There are only two variants: index partitioning attribute is the same as or is different from table partitioning attribute (FRI-1 and FRI-3)

7.2. Parallel Indexing Structures (cont'd)

- **FRI-1**



Processor 1	
23	Adams
18	Kathy
21	Larry
10	Mary
15	Peter
16	Queenie
20	Ross
24	Susan
28	Tracey
8	Agnes

ID = **1** *to* **30**

Processor 2	
37	Chris
60	David
46	Eric
48	Greg
56	Ian
59	Johanna
39	Uma
43	Vera
47	Wenny
50	Xena
49	Bonnie
33	Caroline
38	Dennis

ID = **31** *to* **60**

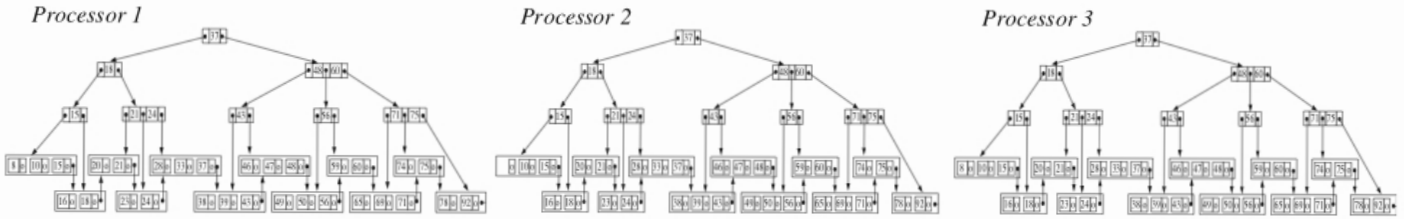
Processor 3	
65	Bernard
92	Fred
71	Harold
74	Norman
78	Oprah
69	Yuliana
75	Zorro

ID = **61** *to* **100**

Figure 7.12 FRI-1 (index partitioning attribute = table partitioning attribute)

7.2. Parallel Indexing Structures (cont'd)

- **FRI-3**



Processor 1	
23	Adams
37	Chris
46	Eric
92	Fred
48	Greg
78	Oprah
28	Tracey
39	Uma
8	Agnes

Name: –x-------
***x** = consonant*

Processor 2	
65	Bernard
60	David
71	Harold
56	Ian
18	Kathy
21	Larry
10	Mary
15	Peter
43	Vera
47	Wenny
50	Xena
33	Caroline
38	Dennis

Name: –x--------
***x** = vowel (a,e)*

Processor 3	
59	Johanna
74	Norman
16	Queenie
20	Ross
24	Susan
69	Yuliana
75	Zorro
49	Bonnie

Name: –x--------
***x** = vowel (i,o,u)*

Figure 7.13 FRI-3 (index partitioning attribute ≠ table partitioning attribute)

7.3. Index Maintenance

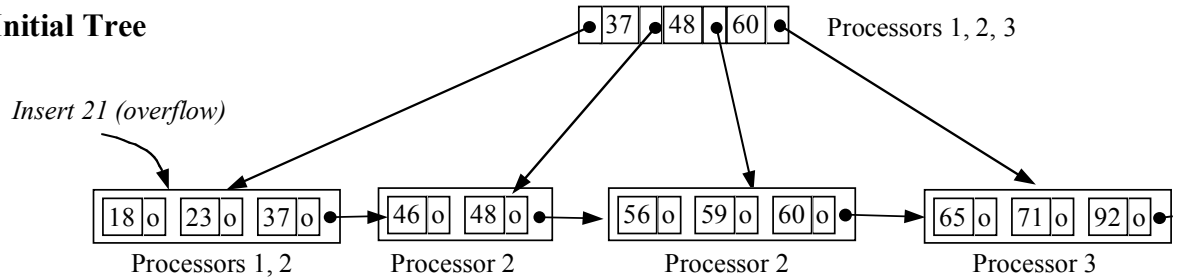
- Index maintenance covers insertion and deletion of index nodes
- General steps:
  - Insert/delete a record to the table (carried out in processor p_1)
  - Insert/delete an index node to/from the index tree (carried out in processor p_2)
  - Update the data pointer
- Two issues:
  - Whether $p_1 = p_2$. This is data pointer complexity
  - Whether maintaining an index (insert/delete) involves multiple processors. This is index tree restructuring issue

7.3. Index Maintenance (cont'd)

- Maintaining a Parallel Non-Replicated Index (**NRI**)
  - Involves a single processor, and hence it is really whether p_1 is equal to p_2
  - For NRI-1 and NRI-2 structures, $p_1 = p_2$, therefore it is done as per normal index maintenance on sequential processors
  - For NRI-3, because $p_1 \neq p_2$, location of the record to be inserted/deleted may be different from the index node insertion/deletion. So, after both the record and the index entry (key) have been inserted, the data pointer from the new index entry in p_1 has to be established to the record in p_2. Deletion is also similar.

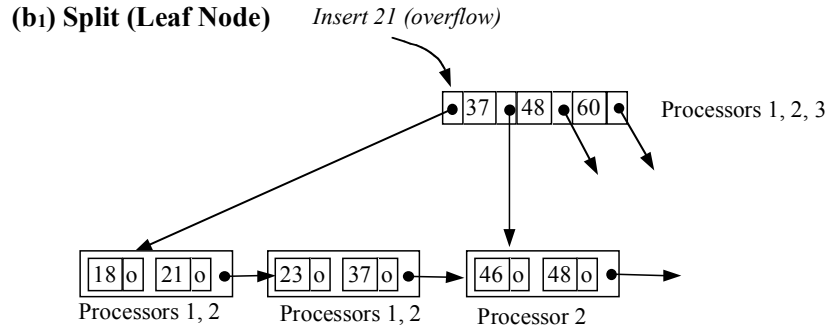
7.3. Index Maintenance (cont'd)

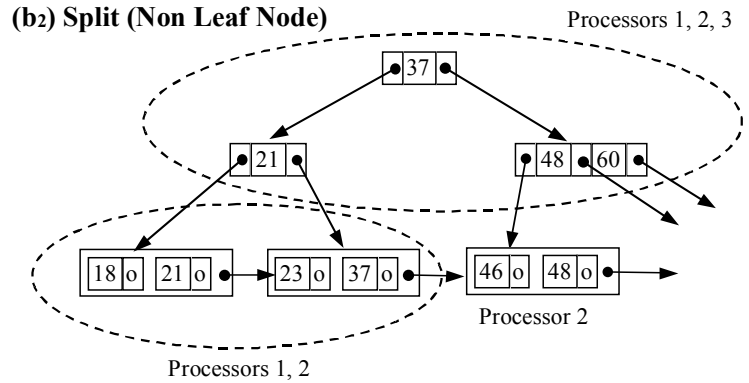
- Maintaining a Parallel Partially-Replicated Index (**PRI**)
  - Maintenance of PRI-1 and PRI-2 is similar to that of NRI-1 and NRI-2 where p1= p2. PRI-3 is also similar to NRI-3; that is, p1≠ p2.
  - Main issue is: index restructuring
  - Example: insert node 21



7.3. Index Maintenance (cont'd)

- Maintaining a Parallel Partially-Replicated Index (**PRI**)
  - Example: insert node 21

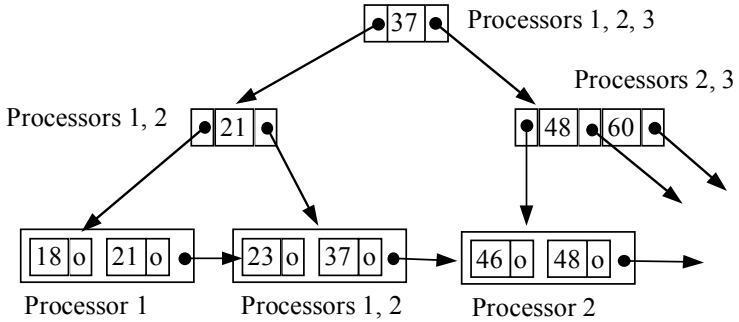




7.3. Index Maintenance (cont'd)

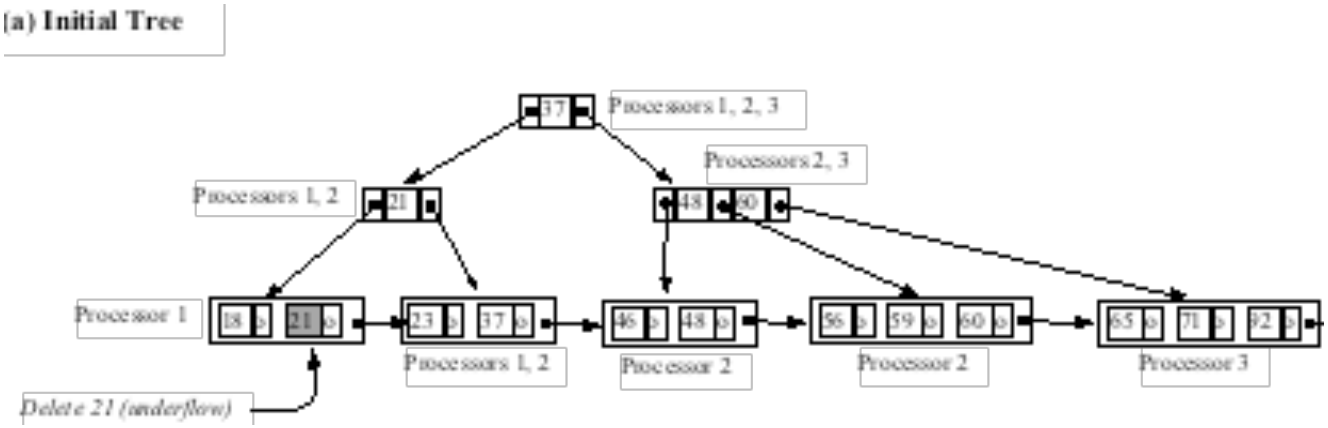
- Maintaining a Parallel Partially-Replicated Index (**PRI**)
  - Example: insert node 21

(c) Restructure (Processor Re-Allocation)



7.3. Index Maintenance (cont'd)

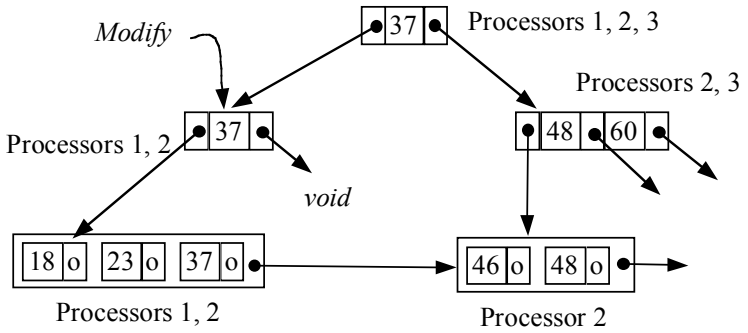
- Maintaining a Parallel Partially-Replicated Index (**PRI**)
  - Example: **delete** node 21



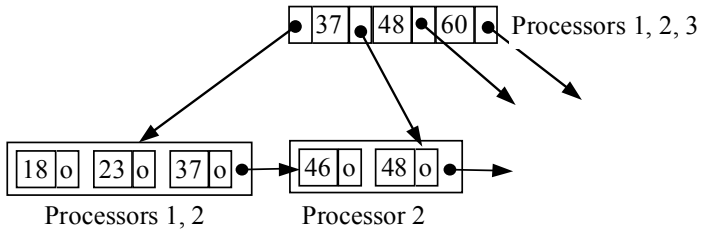
7.3. Index Maintenance (cont'd)

- Maintaining a Parallel Partially-Replicated Index (**PRI**)
  - Example: **delete** node 21

(b) Merge



(c) Collapse



7.3. Index Maintenance (cont'd)

- Maintaining a Parallel Fully-Replicated Index (**FRI**)
  - Index maintenance of the FRI structures is similar to that of the NRI structures, as all indexes are local to each processor.

7.3. Index Maintenance (cont'd)

- Comparisons
  - The simplest forms are NRI-1 and NRI-2 structures, as $p_1 = p_2$ and only single processors are involved in index maintenance (insert/delete).
  - The next level complexity is on data pointer maintenance, especially when index node location is different from based data location. The simpler one is the NRI-3 structure, where data pointer from an index entry to the record is 1-1. The more complex one is the FRI structures, where the data pointers are *N*-1 (from *N* index nodes to 1 record).
  - The highest complexity level is on index restructuring. This applicable to all the three PRI structures.

7.4. Index Storage Analysis

- **Storage cost models for Uniprocessors**
  - **Record storage**: the length of each record, and the blocking factor

Record length = *sum of all fields* + 1 *byte deletion marker*

Blocking factor = floor (*Block size* / *Record length*)

Total blocks for all records = ceiling (*Number of records* / *Blocking factor*)

7.4. Index Storage Analysis (cont'd)

- **Storage cost models for Uniprocessors**
  - **Index storage**: contains *leaf* nodes and *non-leaf* nodes

    The relationship between number of keys in a leaf node and the size of each leaf node:

    $$(pleaf \times (Key\ size + Data\ pointer)) + Node\ pointer \leq Block\ size$$

    where p_{leaf} is the number of keys in a leaf node, *Key size* is the size of the indexed attribute (or key), *Data pointer* is the size of the data pointer, *Node pointer* is the size of the node pointer, and *Block size* is the size of the leaf node.

    $$Number\ of\ \textbf{leaf nodes}\ b1 = \text{ceiling}\ (Number\ of\ records\ /\ (Percentage \times pleaf))$$

    where *Percentage* is the percentage that indicates by how much percentage a node is full

7.4. Index Storage Analysis (cont'd)

- **Storage cost models for Uniprocessors**
  - **Index storage**:

    Number of entries in each non-leaf node (indicated by p; as opposed to p_{leaf}) is:

    $$(p \times \textit{Node pointer}) + ((p - 1 \times \textit{Key size}) \leq \textit{Block size}$$

    The *fanout* (*fo*) of non-leaf node is influenced by the Percentage of an index tree to be full:

    $$fo = \text{ceiling } (\textit{Percentage} \times p)$$

    Number of levels in an index tree is:

    $$x = \text{ceiling } (\log_{fo} (b_1)) + 1)$$

7.4. Index Storage Analysis (cont'd)

- **Storage cost models for Uniprocessors**
  - **Index storage**:

$$\textit{Total non-leaf nodes} = \sum_{i=2}^{x} b_i$$

where b_i = ceiling (b_{i-1} / fo)

Total index blocks = b_1 + *Total non-leaf nodes*

7.4. Index Storage Analysis (cont'd)

- **Storage cost models for Parallel Processors**
  - **NRI storage**: The same calculation applied to uniprocessor indexing can be used by NRI. But the number of records is smaller than that of the uniprocessors
  - **PRI storage**: … next slides …
  - **FRI storage**: Record storage is the same for all indexing structures, as the records are uniformly partitioned to all processors. Index storage is very similar to NRI, except:

    The number of records used in the calculation of the number of entries in leaf nodes is not divided by the number of processors.

    The sizes of data pointers and node pointers must incorporate information on processors. This is necessary since both data and node pointers may go across to another processor.

7.4. Index Storage Analysis (cont'd)

- **Storage cost models for Parallel Processors**
  - **PRI storage**:

    Record storage cost models for all NRI, PRI, and FRI are all the same; that is, divide the number of records evenly among all processors, and calculate the total record blocks in each processor

    Number of leaf nodes in each processor (we call this c_1, instead of b_1):

    $$c_1 = \text{ceiling } (b_1 / \textit{Number of processors}) + 2$$

    $$\textit{Total non-leaf nodes} = c_1 + \sum_{i=2}^{x-1} c_i + c_x$$

7.5. Parallel Search using Index

- Parallel one-index search
  - Queries on the search operation of one indexed attribute. This includes exact match or range queries
- Parallel multi-index search
  - Queries having search predicates on multiple indexed attributes

7.5. Parallel Search using Index (cont'd)

- **Parallel one-index search**
  - Depending on the query type and parallel index
  - **Parallel exact-match search**: processor involvement, index tree traversal, and record loading
  - **Parallel range search**: continuous-range search, and discrete-range search

7.5. Parallel Search using Index (cont'd)

- **Parallel exact-match search (using one index)**
  - **Processor involvement**: Ideally parallel processing may isolate into the processor(s) where the candidate records are located. Involving more processors in the process will certainly not do any good, especially if they do not produce any result.
  - Case 1 (selected processors): Applicable to all indexing structures, except for the NRI-2 structure.
  - Case 2 (all processors): Applicable to the NRI-2 indexing structure only, because using the NRI-2 indexing structure, there is no way to identify where the candidate records are located without searching in all processors

7.5. Parallel Search using Index (cont'd)

- **Parallel exact-match search (using one index)**
  - **Index tree traversal**: Searching is done through index tree traversal starting from the root node and finishing either at a matched leaf node or no match is found.
  - Case 1 (isolated to local processors): Applicable to all indexing structures, but PRI-2.
  - Case 2 (crossing from one processor to another): Applicable to PRI-2 only, where searching that starts from a root node at any processor may end up on a leaf node at a different processor

7.5. Parallel Search using Index (cont'd)

- **Parallel exact-match search (using one index)**
  - **Record loading**: Once a leaf node containing the desired data has been found, the record pointed by the leaf node is loaded from disk.
  - Case 1 (local record loading): Applicable to NRI/PRI/FRI-1 and NRI/PRI-2 indexing structures, since the leaf nodes and the associated records in these indexing schemes are located at the same processors.
  - Case 2 (remote record loading): Applicable to NRI/PRI/FRI-3 indexing structures where the leaf nodes are not necessarily placed at the same processor where the records reside.

7.5. Parallel Search using Index (cont'd)

- **Parallel range search (using one index)**
  - **Continuous range**: May need to involve multiple processors, need to identify the lower and upper bound of the range, and once lower/upper bound is identified, it becomes easy to trace all values within a given range, by traversing leaf nodes of the index tree
  - **Discrete range**: each discrete value in the search predicate is converted into multiple exact match predicates. Further processing follows the processing method for exact match queries.

7.5. Parallel Search using Index (cont'd)

- **Parallel multi-index search**
  - There are two methods:
  - **Intersection method**: all indexed attributes in the search predicate are first searched independently. Each search predicate will form a list of index entry results found after traversing each index. After all indexes have been processed, the results from one index tree will be intersected with the results of other index trees to produce a final list
  - **One-index method**: Just use one of the indexes

7.5. Parallel Search using Index (cont'd)

- **Parallel multi-index search (using the Intersection method)**
  - Since multiple indexes are used, there is a possibility that different indexing structures are used by each indexed attribute:

  - **Case 1 (one index is based on NRI-1, PRI-1, or FRI-1)**:

    Processor involvement: If the second indexing structure is NRI-2, PRI-2, or FRI-3, only those processors used for processing the first search attribute (which uses either NRI/PRI/FRI-1) will need to be activated. This is "*early intersection*"

    Intersection operation: for NRI-3 and PRI-3, the leaf nodes found in the index traversal must be sent to the processors where the actual records reside, so that the intersection operation can be carried out there. Leaf node transfer is not required for NRI-2, PRI-2, or even FRI-3.

7.5. Parallel Search using Index (cont'd)

- **Parallel multi-index search (using the Intersection method)**
  - **Case 2 (one index is based on NRI-3, PRI-3, or FRI-3)**:

    Applicable to the first index based on NRI/PRI/FRI-3 and the other indexes based on any other indexing structures, including NRI/PRI/FRI-3, but excluding NRI/PRI/FRI-1. The combination between NRI/PRI/FRI-3 and NRI/PRI/FRI-1 has already been covered by case 1

    Processor involvement: No "*early intersection*"

    Intersection operation: particularly for NRI/PRI-3, it will be carried out as for case 1; that is, leaf nodes found in the searching process will need to be sent to where the actual records are stored and the intersection will be locally performed there.

7.5. Parallel Search using Index (cont'd)

- **Parallel multi-index search (using the Intersection method)**
  - **Case 3 (one index is based on NRI-2 or PRI-2)**:

    Processor involvement: No "*early intersection*" since none of NRI/PRI/FRI-1 is used

7.5. Parallel Search using Index (cont'd)

- **Parallel multi-index search (using the One-Index method)**
  - Two main factors:
  - **The selectivity factor of each search predicate**: it will be ideal to choose a search predicate which has the lowest selectivity ratio, with a consequence that most records have already been filtered out by this search predicate and hence less work will be done by the rest of the search predicates
  - **The indexing structure** which is used by each search predicate: It will be ideal to use an indexing structure which uses selected processors, local index traversals, and local record loading

7.6. Parallel Join using Index

- Parallel one-index join
  - Involves one **non-indexed** table (say table ***R***) and one **indexed** table (say table ***S***)
- Parallel two-index join
  - Both tables are indexed by the join attribute

7.6. Parallel Join using Index (cont'd)

- **Parallel One-Index Join**
  - Data partitioning and local join steps
  - In the ***data partitioning*** step, depending on which parallel indexing scheme is used by table *S*, data partitioning to table *R* may or may not be conducted.

  - **Case 1 (NRI-1 and NRI-3)**:

    Records of table *R* are re-partitioned according to the same range partitioning function used by table *S*. Both the records and index tree of table *S* are not at all mutated. At the end of the data partitioning step, each processor will have records *R* and index tree *S* having the same range of values of the join attribute.

  - **Case 2 (NRI-2)**:

    Broadcast the non-indexed table *R* has to be broadcast to all processors

7.6. Parallel Join using Index (cont'd)

- **Parallel One-Index Join**
  - **Case 3 (PRI)**:

    If table *S* is indexed using any of the PRI structures, the non-indexed table *R* do not need to be re-distributed, since by using a PRI structure, the global index is maintained and more importantly the root index node is replicated to all processors so that tracing to any leaf node can be done from any root node at any processor.

  - **Case 4 (FRI)**:

    If table *S* is indexed using any of the FRI structures (i.e. FRI-1 or FRI-3), like Case 3, the non-indexed table *R* is not redistributed either.

7.6. Parallel Join using Index (cont'd)

- **Parallel One-Index Join**
  - In the ***local join*** step, each processor performs its joining operation independently of the others. Using a nested block index join method as described earlier, for each record *R,* search for a matching index entry of table *S*. If a match is found, depending on the location of the record (i.e. whether it is located at the same place as the leaf node of the index), record loading is performed.

7.6. Parallel Join using Index (cont'd)

- **Parallel Two-Index Join**
  - Each processor performs an independent merging of the leaf nodes, and the final query result is the union of all temporary results gathered by each processor.

  - **Case 1 (all index structures, except NRI-2 and PRI-2)**:
    Whichever parallel indexing structure is used, they must adopt the same index partitioning function.The main processing is a merging operation of the leaf nodes of the two index trees in each processor.

  - **Case 2 (NRI-2 or PRI-2)**:
    Unfortunately, parallel two-index join query processing cannot make use of these indexes. Therefore, NRI-2 and PRI-2 are useless for parallelizing two-index join query processing.

7.7. Comparative Analysis

- Parallel search index
  - Parallel one-index search
  - Parallel multi-index search (Intersection method, One-index method)
- Parallel join index

7.7. Comparative Analysis (cont'd)

- **Parallel One-Index Search**
  - Processor involvement, index traversal, and record loading
  - Shaded cells show more expensive operations in comparison with others within the same operation

	NRI Schemes			PRI Schemes			FRI Schemes	
	NRI-1	**NRI-2**	**NRI-3**	**PRI-1**	**PRI-2**	**PRI-3**	**FRI-1**	**FRI-3**
Processor Involvement	Selected processors	All processors	Selected processors	Selected processors	Selected processors	Selected processors	Selected processors	Selected processors
Index Traversal	Local search	Local search	Local search	Local search	Remote search	Local search	Local search	Local search
Record Loading	Local record load	Local record load	Remote record load	Local record load	Local record load	Remote record load	Local record load	Remote record load

Figure 7.24. A Comparative Table for Parallel One-Index Selection Query Processing

7.7. Comparative Analysis (cont'd)

- **Parallel Multi-Index Search (with Intersection method)**
  - Individual index searching
  - Intersection operation

  - Case 1: one index based on NRI-1, PRI-1, or FRI-1
  - Case 2: one index based on NRI-3, PRI-3, or FRI-3
  - Case 3: one index based on NRI-2 or PRI-2

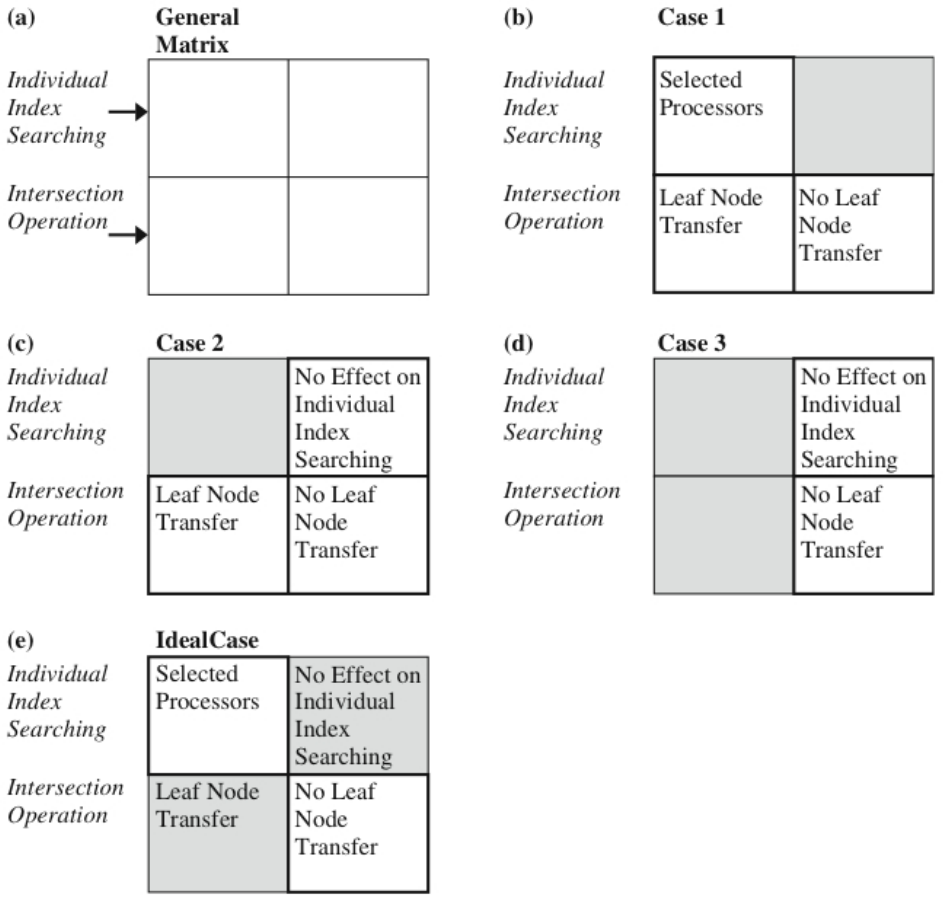


Figure 7.25 A comparative table for parallel multi-index selection query processing using an *intersection method*

7.7. Comparative Analysis (cont'd)

- **Parallel Multi-Index Search (with One-Index method)**
  - The main aim is to minimize I/O, The first search predicate, which uses an index, should have the smallest selectivity ratio.
  - The smallest selectivity ratio is given by an exact match search with unique records, and the most efficient indexing structure is NRI/PRI/FRI-1. This is the most preferable indexing structure.
  - The next preferable option is exact match search of non-unique records or continuous range search predicates depending on the selectivity ratio using NRI-2/3 or PRI-2/3 or FRI-3.

	NRI Schemes			PRI Schemes			FRI Schemes	
	NRI-1	**NRI-2**	**NRI-3**	**PRI-1**	**PRI-2**	**PRI-3**	**FRI-1**	**FRI-3**
Exact Match Search Queries	Isolated record loading	Record loading possibly spread (if non-unique)	Record loading possibly spread (if non-unique)	Isolated record loading	Record loading possibly spread (if non-unique)	Record loading possibly spread (if non-unique)	Isolated record loading	Record loading possibly spread (if non-unique)
Continuous Range Search Queries	Record loading possibly spread, but not random	Record loading possibly spread randomly	Record loading possibly spread randomly	Record loading possibly spread, but not random	Record loading possibly spread randomly	Record loading possibly spread randomly	Record loading possibly spread, but not random	Record loading possibly spread randomly

Figure 7.26. A Comparative Table for Parallel Multi-Index Selection Query Processing using a *One-Index Access Method*

7.7. Comparative Analysis (cont'd)

- **Parallel Index Join**
  - Parallel one-index join: Data partitioning, local join and indexed table searching
  - Paralel two-index join: Merging, searching start/end values, and data loading

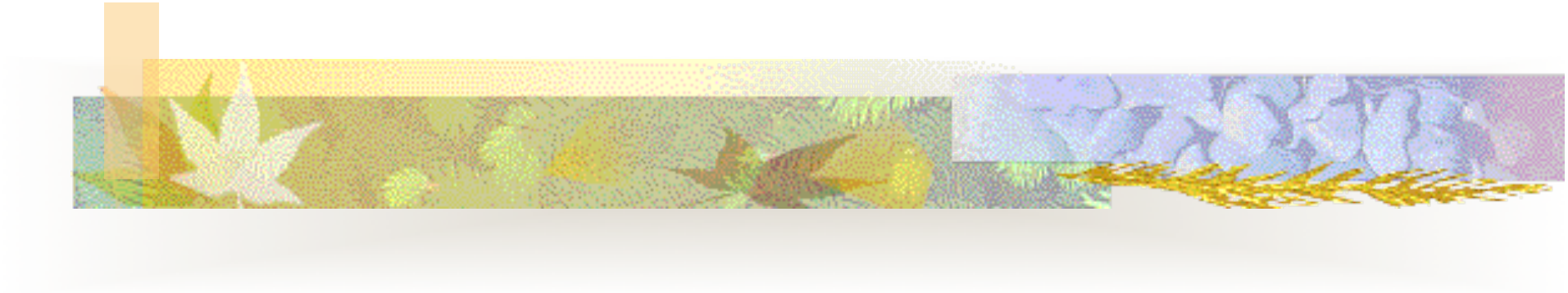
			NRI Schemes			PRI Schemes			FRI Schemes	
			NRI-1	**NRI-2**	**NRI-3**	**PRI-1**	**PRI-2**	**PRI-3**	**FRI-1**	**FRI-3**
Parallel One-Index Join	Data partitioning		Partition	Broadcast	Partition	No Partition	No Partition	No Partition	No Partition	No Partition
	Local join	Indexed table searching	Local search	Local search	Local search	Remote search	Remote search	Remote search	Local search	Local search
		Indexed table record loading	Local data load	Local data load	Remote data load	Remote data load	Remote data load	Remote data load	Remote data load	Remote data load
Parallel Two-Index Join	Merging	Searching start and end values	Not necessary	N/A	Not necessary	Not necessary	N/A	Not necessary	Searching needed	Searching needed
		Data loading	Local data load		Remote data load	Local data load		Remote data load	Local data load	Remote data load

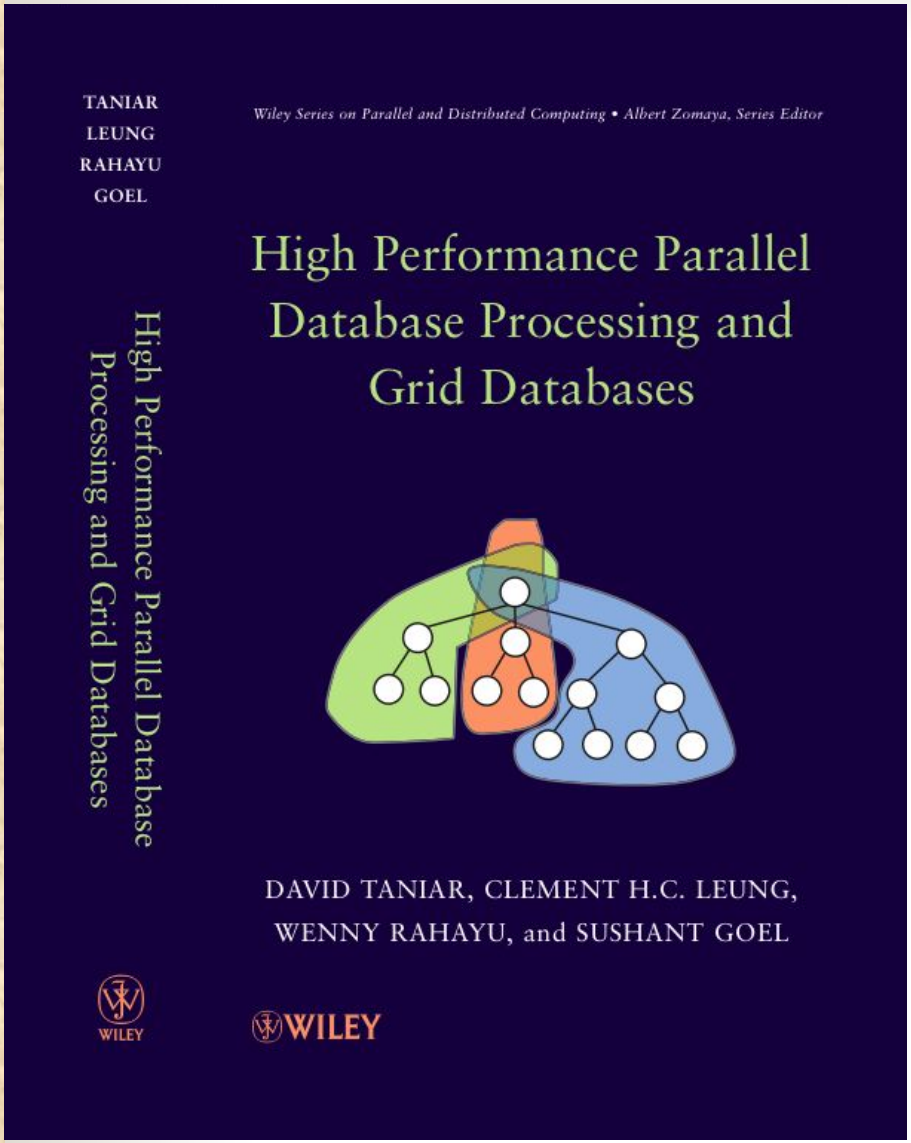
Figure 7.27. A Comparative Table for Parallel Index-Join Query Processing

7.8. Summary

- Parallel indexing structures
  - NRI, PRI, and FRI
- Parallel indexing maintenance
  - Insertion and deletion operations
- Parallel indexing storage
  - Storage for tables and for indices
- Parallel index-search query processing
  - One-index search and multiple-index search
- Parallel index-join query processing
  - Parallel one-index join and two-index join

Continue to Chapter 8...





Chapter 16 Parallel Data Mining

16.1. Data-Intensive Applications

- All of the three: ***databases***, ***data warehouses***, and ***data mining***, deal with data.

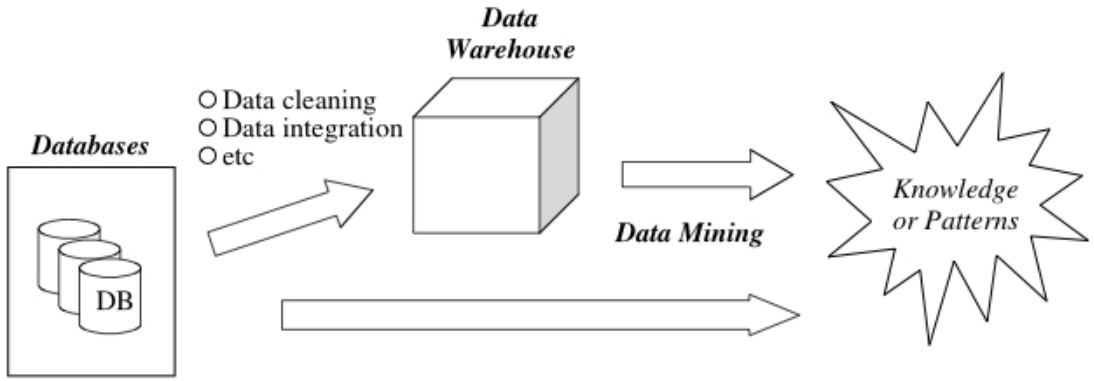


Figure 16.1 Evolution of data-intensive applications

16.1. Data-Intensive Applications (cont'd)

- **Databases** are commonly deployed in almost every organization. In a simple form, databases are referred to as data repositories. Database processing are queries, and transactions. The data contained in a database is normally operational data.

16.1. Data-Intensive Applications (cont'd)

- **Data warehouse** provides information from a historical perspective, whereas an operational database keeps data of current value. The process involves: data extraction, filtering, transforming, integrating from various sources, classifying the data, aggregating and summarizing the data. The result is a data warehouse where the data is integrated, time-variant, non-volatile, and commonly subject-oriented.

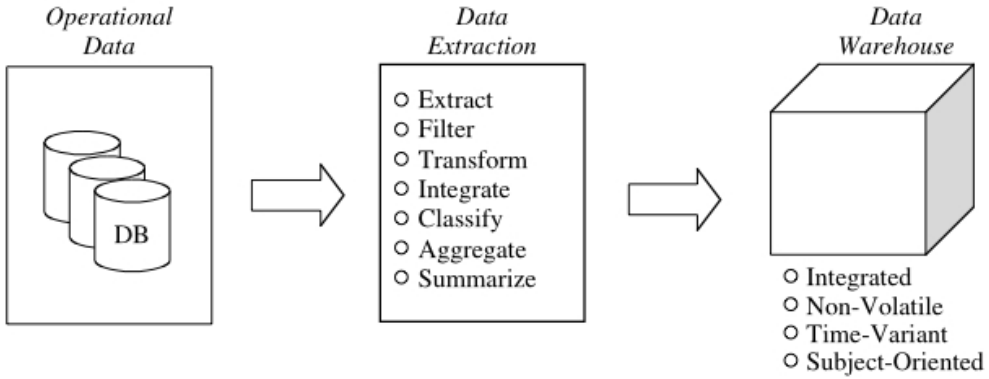


Figure 16.2 Building a data warehouse

16.1. Data-Intensive Applications (cont'd)

- **Data mining** analyzes a large amount of data stored in databases to discover interesting knowledge in the form of patterns, association, changes, anomalies, significant structures, etc. Data mining is also known as *knowledge discovery*, or more precisely, knowledge discovery of data.

16.2. Data Mining: A Brief Overview

- Data mining is a process for discovering useful, interesting, and sometimes surprising knowledge from a large collection of data.

- Data Mining Tasks
  - ***Descriptive data mining***: describes the data set in a concise manner and presents interesting general properties of the data; summarizes the data in terms of its properties and correlation with others.

  - ***Predictive data mining***: Predictive data mining builds a prediction model whereby it makes inferences from the available set of data, and attempts to predict the behaviour of new data sets.

16.2. Data Mining: A Brief Overview (cont'd)

- Data Mining Techniques
  - ***Class description*** or ***characterization*** summarizes a set of data in a concise way that distinguishes this class from others.
  - ***Association rules*** discover association relationships or correlation among a set of items.
  - ***Classification*** analyzes a set of training data and constructs a model for each class based on the features in the data.
  - ***Prediction*** predicts the possible values of some missing data or the value distribution of certain attributes in a set of objects.
  - ***Clustering*** is a process to divide the data into clusters, whereby a cluster contains a collection of data that is similar to one another.
  - ***Time-series analysis*** analyzes a large set of time series data to find certain regularities and interesting characteristics.

16.2. Data Mining: A Brief Overview (cont'd)

- **Querying vs. Mining**
  - Although it has been stated that the purpose of mining (or data mining) is to discover knowledge, it should be differentiated from querying (or database querying), which simply retrieves data.
  - In some cases, this is easier said than done. Consequently, highlighting the differences is critical in studying both database querying and data mining. The differences can generally be categorized into: ***unsupervised*** and ***supervised*** learning.

16.2. Data Mining: A Brief Overview (cont'd)

- **Unsupervised Learning**
  - Unsupervised learning is whereby the learning process is not guided, or even dictated, by the expected results. To put it in another way, unsupervised learning does not require a hypothesis. Exploring the entire possible space in the jungle of data might be overstating, but can be analogous that way.

  - **Association rule mining vs. Database querying**: Given a database D, association rule mining produces an association rule $Ar(D) = X \rightarrow Y$, where $X, Y \in D$. A query $Q(D, X) = Y$ produces records Y matching the predicate specified by X.

    The pattern $X \rightarrow Y$ may be based on certain criteria, such as: majority, minority, absence, exception

16.2. Data Mining: A Brief Overview (cont'd)

- **Unsupervised Learning**
  - **Sequential patterns vs. Database querying**: Given a database D, a sequential pattern $Sp(D) = O{:}X \rightarrow Y$, where O indicates the owner of a transaction and $X, Y \in D$. A query $Q(D, X, Y) = O$, or $Q(D, aggr) = O$, where *aggr* indicates some aggregate functions.
  - **Clustering vs. Database querying**: Given database D, a clustering $Cl(D) = \sum_{i=1}^{n} \{X_{i1}, X_{i2}, \ldots\}$, where it produces n clusters each of which consists of a number of items X. A query $Q(D, X_1) = \{X_2, X_3, X_4, \ldots\}$, where it produces a list of items $\{X_2, X_3, X_4, \ldots\}$ having the same cluster as the given item X_1.

16.2. Data Mining: A Brief Overview (cont'd)

- **Supervised Learning**
  - Supervised learning is naturally the opposite of unsupervised learning, since supervised learning starts with a direction pointing to the target.
  - **Decision tree classification vs. Database querying**: Given database *D*, a decision tree *Dt*(*D, C*) = *P*, where *C* is the given category and *P* is the result properties. A query *Q*(*D*, *P*) = *R*, is where the property is known in order to retrieve records *R*.

16.2. Data Mining: A Brief Overview (cont'd)

- **Parallelism in Data Mining**
  - Large volume of data
  - High dimension (large number of attributes)
  - High degree of complexity (not previously found or applicable to databases or even data warehousing)
  - Even a simple data mining technique requires a number of iterations of the process, and each of the iterations refines the results until the ultimate results are generated

  - Parallelism in data mining: **data parallelism** and **result parallelism**

Data Parallelism

- Data parallelism is created because the data is partitioned into a number of processors and each processor focuses on its partition of the data set.
- After each processor completes its local processing and produces the local results, the final results are formed basically by combining all local results.
- Since data mining processes normally exist in several iterations, data parallelism raises some complexities, not commonly found in database query processing.

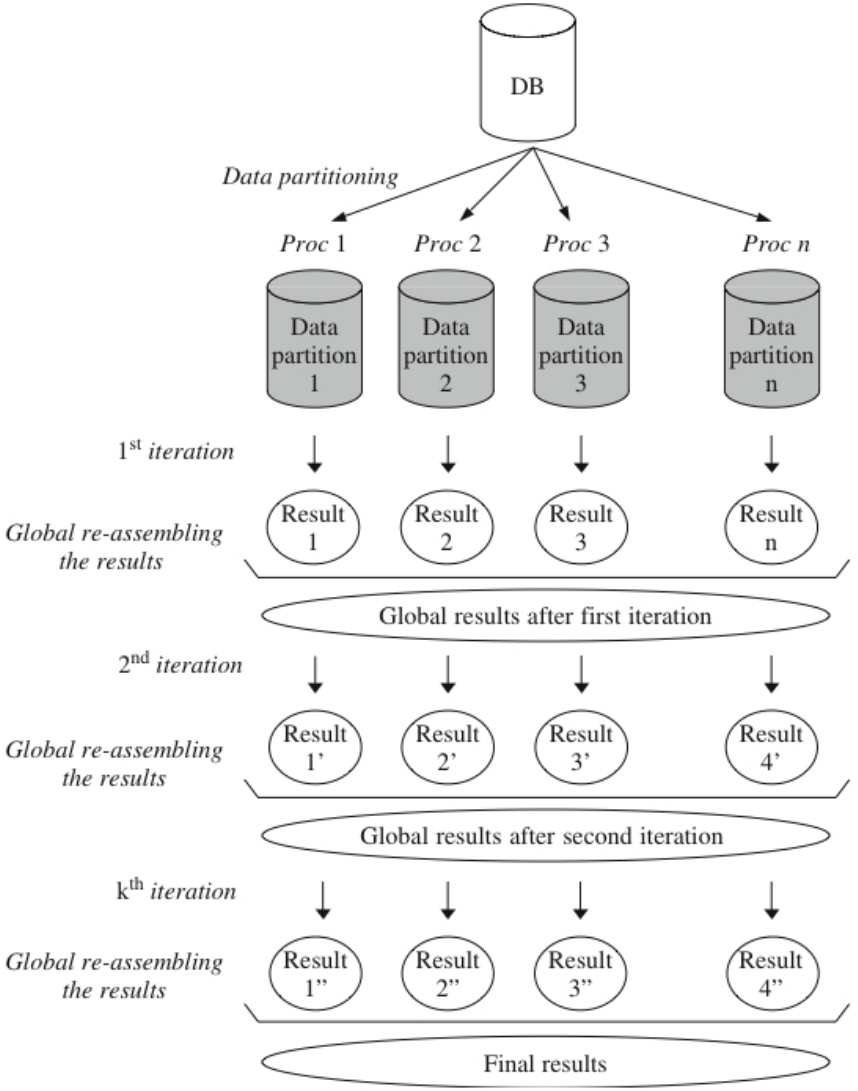


Figure 16.3 Data parallelism for data mining

Result Parallelism

- Result parallelism focuses on how the target results can be parallelized during the processing stage without having produced any results or temporary results.
- Result parallelism works by partitioning the target results, and each processor focuses on its target result partition.
- Each processor will do whatever it takes to produce the result within the given range, and will take any input data necessary to produce the desired result space.

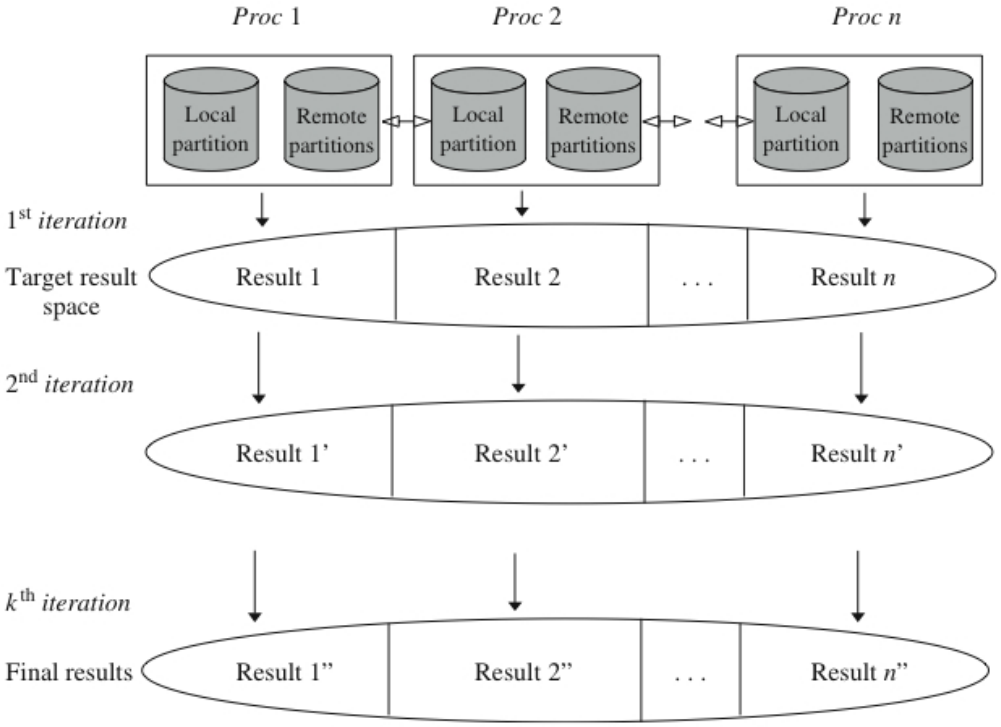


Figure 16.4 Result parallelism for data mining

16.3. Parallel Association Rules

- To discover rules based on the correlation between different attributes/items found in the dataset
- Two phases: (*i*) phase one: discover **frequent itemsets** from a given dataset, and (*ii*) phase two: **generate rules** from these frequent itemsets.
- The first phase is widely recognized as being the most critical, computationally intensive task. Since the frequent itemset generation phase is computationally expensive, most work on association rules, including parallel association rules, have been focusing on this phase only. Improving the performance of this phase is critical to the overall performance.

16.3. Parallel Association Rules (cont'd)

- **Some measurements**
  - **Support** and minimum support
  - **Confidence** and minimum confidence
  - **Frequent Itemset**: An itemset in a dataset is considered as frequent if its support is equal to, or greater than, the minimum support threshold specified by the user.
  - **Candidate Itemset**: Given a database *D* and a minimum support threshold *minsup* and an algorithm that computes *F*(*D*, *minsup*), an itemset *I* is called *candidate* for the algorithm to evaluate whether or not itemset *I* is frequent.
  - **Association rules**: At a given user-specified minimum confidence threshold *minconf*, find all association rules *R* from a set of frequent itemset *F* such that each of the rules has confidence equal to, or greater than *minconf*.

16.3. Parallel Association Rules (cont'd)

- **Example**

Transaction ID	Items Purchased
100	**bread, cereal, milk**
200	**bread, cheese, coffee, milk**
300	**cereal, cheese, coffee, milk**
400	**cheese, coffee, milk**
500	**bread, sugar, tea**

Figure 16.5. Example Dataset

Frequent Itemset	Support
bread	**60%**
cereal	**40%**
cheese	**60%**
coffee	**60%**
milk	**80%**
bread, milk	**40%**
cereal, milk	**40%**
cheese, coffee	**60%**
cheese, milk	**60%**
coffee, milk	**60%**
cheese, coffee, milk	**60%**

Figure 16.6. Frequent Itemset

Association Rules	**Confidence**
bread→milk	67%
cereal→milk	100%
cheese→coffee	100%
cheese→milk	100%
coffee→milk	100%
coffee→cheese	100%
milk→cheese	75%
milk→coffee	75%
cheese, coffee→milk	100%
cheese, milk→coffee	100%
coffee, milk→cheese	100%
cheese→coffee, milk	100%
coffee→cheese, milk	100%
milk→cheese, coffee	75%

Figure 16.7. Association Rules

- **Frequent itemset process**
  - Iteration 1: scan the dataset and finds all frequent 1-itemset
  - Iteration 2: join each frequent 1-itemset and generates candidate 2-itemset. Then it scans the dataset again, enumerates the exact support of each of these candidate itemsets and prunes all infrequent candidate 2-itemsets.
  - Iteration 3: joins each of the frequent 2-itemset and generates the following potential candidate 3-itemset. Prunes those candidate 3-itemset that do not have a subset itemset in F_2. Scans the dataset and finds the exact support of that candidate itemset. It finds that this candidate 3-itemset is frequent. In the joining phase, Cannot produce any candidate itemset for the next iteration.

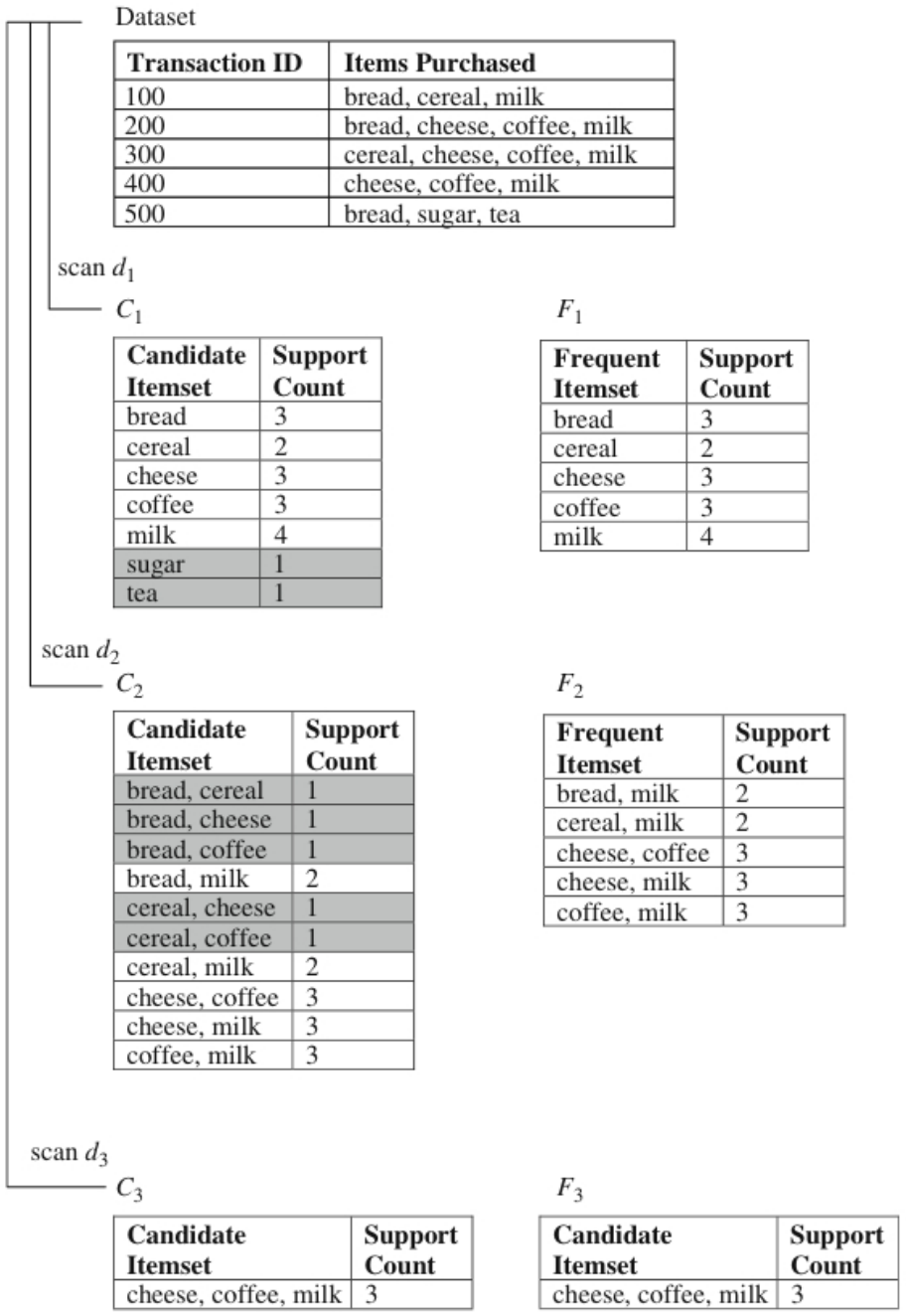


Figure 16.9 Example of the Apriori algorithm

16.3. Parallel Association Rules (cont'd)

- **Rules generation**
  - Using the frequent itemset {*cheese coffee milk*}, the following three rules hold, since the confidence is 100%

    cheese, coffee → milk

    cheese, milk → coffee

    coffee, milk → cheese
  - Then we use the *apriori_gen*() function to generate all candidate 2-itemsets, resulting {*cheese milk*} and {*coffee milk*}. After confidence calculation, the following two rules hold:

    coffee → cheese, milk (confidence=100%)

    cheese → coffee, milk (confidence=75%)
  - Therefore, from one frequent itemset {*cheese coffee milk*} alone, five association rules shown above have been generated (see Figure 16.7)

16.3. Parallel Association Rules (cont'd)

- **Parallel Association Rules**
  - ***Data parallelism*** for association rule mining is often referred to as *count distribution*
  - ***Result parallelism*** is widely known as *data distribution*.

- **Data Parallelism (or Count Distribution)**
  - Each processor will have a disjoint data partition to work with. Each processor, however, will have a complete candidate itemset, although with partial support or support count.
  - At the end of each iteration, since the support or support count of each candidate itemset in each processor is incomplete, each processor will need to '*redistribute*' the count to all processors. Hence, the term '*count distribution*' is used.
  - This global result reassembling stage is basically to redistribute the support count which often means global reduction to get global counts. The process in each processor is then repeated until the complete frequent itemset is ultimately generated.

Original dataset

Transaction ID	Items Purchased
100	bread,cereal,milk
200	bread,cheese,coffee,milk
300	cereal, cheese, coffee, milk
400	cheese,coffee,milk
500	bread, sugar, tea

Processor 1

TID	Items Purchased
100	bread, cereal, milk
200	bread, cheese, coffee, milk

Processor 2

TID	Items Purchased
300	cereal, cheese, coffee, milk
400	cheese,coffee,milk
500	bread, sugar,tea

Candidate Itemset	Support Count
bread	2
cereal	1
cheese	1
coffee	1
milk	2
sugar	0
tea	0

Candidate Itemset	Support Count
bread	1
cereal	1
cheese	2
coffee	2
milk	2
sugar	1
tea	1

Global reduction of counts

Processor 1

Candidate Itemset	Support Count
bread	3
cereal	2
cheese	3
coffee	3
milk	4
sugar	1
tea	1

Processor 2

Candidate Itemset	Support Count
bread	3
cereal	2
cheese	3
coffee	3
milk	4
sugar	1
tea	1

The process continues to generate 2-frequent itemset...

Figure 16.11 Count distribution (data parallelism for association rule mining)

- **Result Parallelism (or Data Distribution)**
  - Data distribution parallelism is based on result parallelism whereby parallelism is created due to the partition of the result, instead of the data. However, the term 'data distribution' might be confused with data parallelism (count distribution).
  - Initially, the dataset has been partitioned. However, each processor needs to have not only its local partition, but all other partitions from other processors.
  - At the end of each iteration, where each processor will produce its own local frequent itemset, each processor will also need to send to all other processors its frequent itemset, so that all other processors can use this to generate its own candidate itemset for the next iteration.

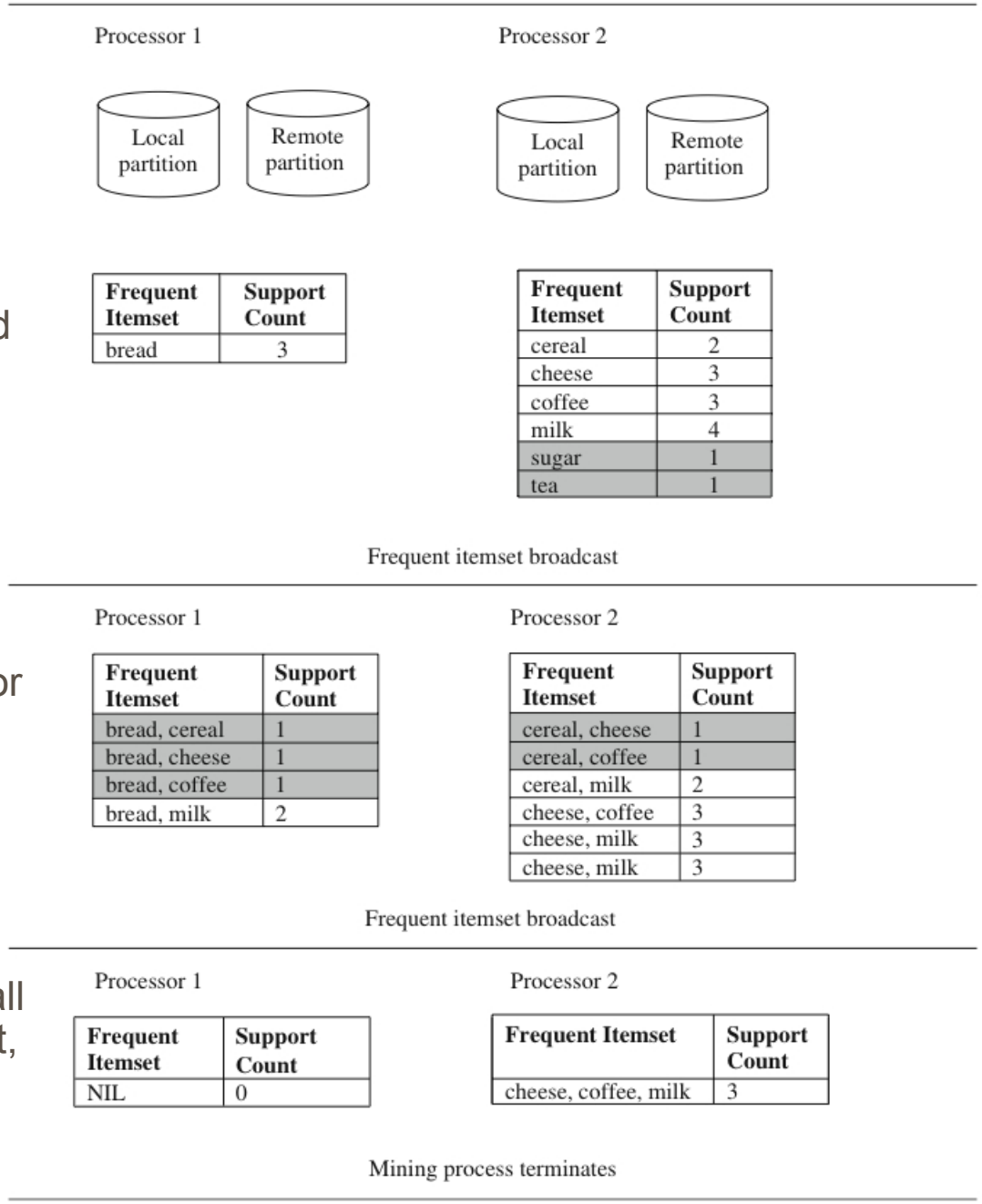


Figure 16.12 Data distribution (result parallelism for association rule mining)

16.4. Parallel Sequential Patterns

- **Sequential patterns**, also known as sequential rules, are very similar to association rules. They form a causal relationship between two itemsets, in a form of $X \rightarrow Y$, where because X occurs, it causes Y to occur with a high probability.
- Association rules are *intra-transaction* patterns or sequences, where the rule $X \rightarrow Y$ indicates that both items X and Y must exist in the same transaction.
- Sequential pattern are *inter-transaction* patterns or sequences. The same rule above indicates that since item X exists, this will lead to the existence of item Y in the near future transaction.

16.4. Parallel Sequential Patterns (cont'd)

- **Example**

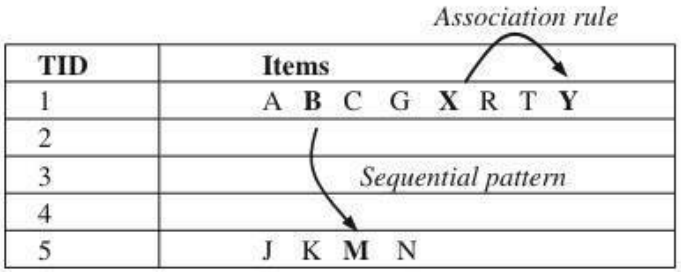


Figure 16.13 Sequential patterns vs. association rules

16.4. Parallel Sequential Patterns (cont'd)

- **Concepts**
  - Given a set of transactions *D* each of which consists of the following fields: customer ID, transaction time, and the items purchased in the transaction, mining sequential patterns is used to find the inter-transaction patterns/sequences that satisfy minimum support *minsup*, minimum gap *mingap*, maximum gap *maxgap*, and window size *wsize* specified by the user.

Cust ID	Timestamp	Items
10	20-Apr	Oreo, Aqua, Bread
10	28-Apr	Canola oil, Chicken, Fish
10	5-May	Chicken wing, Bread crumb

Figure 16.14 Sequences for customer ID 10

16.4. Parallel Sequential Patterns (cont'd)

- **Concepts**
  - A *sequence s* is an ordered list of itemsets *i*.
  - *Containment*: <(5 6) (7)> is contained in <(4 5) (4 5 6 7) (7 9 10)>, because (5 6) ⊆ (4 5 6 7) and (7) ⊆ (7 9 10). Whereas <(3 5)> is not contained in <(3) (5)>.

  - Four important parameters in mining sequential patterns: support, window size, minimum gap, and maximum gap.

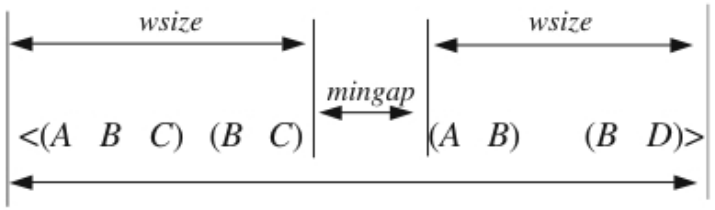


Figure 16.15 Time and sliding windows

16.4. Parallel Sequential Patterns (cont'd)

- **Example**

Customer transactions

Customer ID	Transaction time (days)	Items bought
100	1 3 7	(A C) (B C D) (C D)
200	2 4	(A D) (B D)
300	4 8	(A B) (B C)

The above table can be written as follows:

100 <(A C) (B C D) (C D)>
200 <(A D) (B D)>
300 <(A B) (B C)>

minsup = 2

Example 1: *wsize* = 0

Frequent 2-sequence	Support count (Support)
<(A) (B)>	3 (100%)
<(A) (C)>	2 (67%)
<(A) (D)>	2 (67%)
<(B) (C)>	2 (67%)

Example 2: *wsize* = 3

Frequent 2-sequence	Support count (Support)
<(A) (C)>	2 (67%)
<(B) (C)>	2 (67%)

Figure 16.16 *minsup* and *wsize* examples

16.4. Parallel Sequential Patterns (cont'd)

- **Sequential Patterns Process**
  - Phase 1: *k*=1
  - Phase 2: *k*>1

```
Algorithm: Mining sequential patterns

1. pass k=1
   a. Find all frequent 1-sequences

2. pass k>1
   a. Generate candidate sequences C_k from freq (k-1)
      sequences
   b. Count supports for candidate sequences C_k
   c. Get frequent k-sequences
   d. Prune frequent k-sequences where some (k-1)
      contiguous subsequences is not in frequent (k-1)
      sequences
```

Figure 16.17 Mining sequential pattern algorithm

16.4. Parallel Sequential Patterns (cont'd)

- **Parallel Processing**
  - **Data parallelism** (or count distribution)
  - Result parallelism (or data distribution)

Algorithm: Count Distribution Sequential Pattern Mining

1. Partition dataset into p partitions
   where p is the total number of processors
2. Allocate each partition to a processor

3. pass k=1
   a. Each processor P_i asynchronously finds 1-sequence support count from its local partition
   b. Synchronous to exchange local sequence support count
   c. Each processor P_i finds the same set frequent 1-sequences

4. pass k>1
   a. Generate candidate sequences C_k from freq (k-1) sequences
   b. Add all candidate k-sequences C_k into hash tree
   c. Processor P_i scans its local partition to update support count for C_k
   d. Synchronous all processors and exchange with all other processors their C_k local support count
   e. Each processor gets frequent k-sequences from C_k.

Figure 16.19 Count distribution algorithm for parallel mining sequential patterns

16.4. Parallel Sequential Patterns (cont'd)

- **Parallel Processing**
  - Data parallelism (or count distribution)
  - **Result parallelism** (or data distribution)

Algorithm: Data Distribution Sequential Pattern Mining

```
1. pass k=1
   a. Similar to Count Distribution

2. pass k>1
   a. Generate candidate sequences using previous
      iteration frequent sequences. Divide 1/P of them
      to each processor
   b. Use local and received data sequences to update
      candidate sequences support counts
   c. Get frequent sequences
   d. Gather all frequent sequences from all processors
```

Figure 16.20 Data distribution algorithm for parallel mining sequential patterns

16.5. Summary

- Parallelism in data mining
  - **Data parallelism**: Data parallelism in association rules and sequential patterns is often known as *count distribution* where the counts of candidate itemsets in each iteration are shared and distributed to all processors. Hence, there is a synchronization phase.
  - **Result parallelism**: Result parallelism, on the other hand, is parallelization of the results (i.e. frequent itemset and sequence itemset). This parallelism model is often known as *data distribution*, where the dataset and frequent itemsets are distributed and moved from one processor to another at the end of each iteration.
- Parallel association rules and parallel sequential patterns using data and result parallelism

Continue to Chapter 17…