Parallel Databases

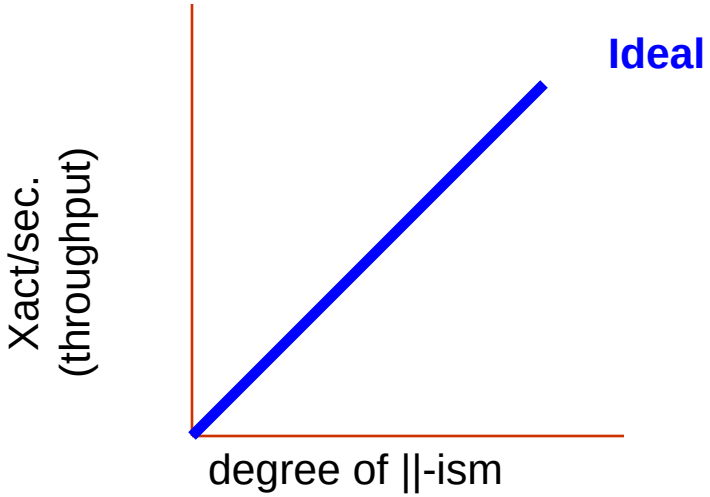
Database System Concepts, 6th Ed.

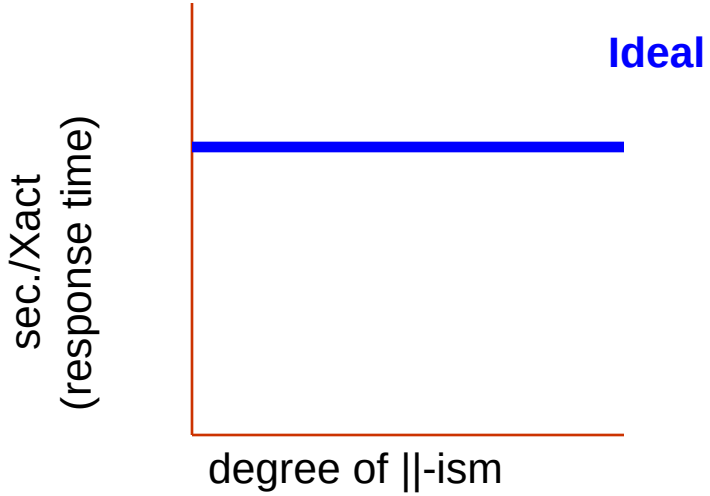
Parallel Databases

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

Some || Terminology

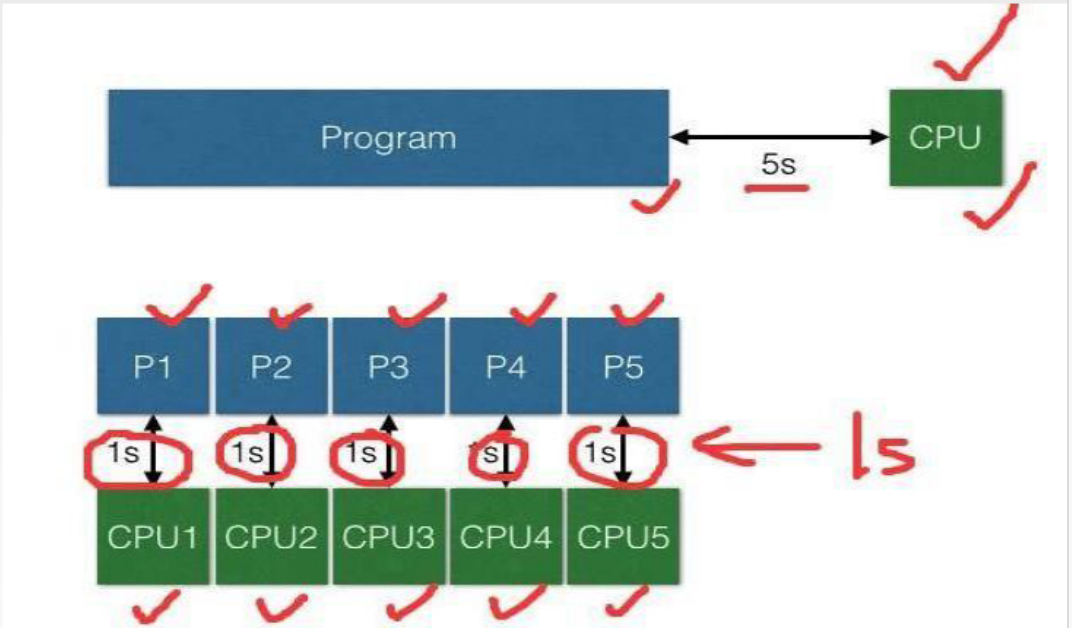
- Speed-Up
  - Adding more resources results in proportionally less running time for a fixed amount of data.
- Scale-Up
  - If resources are increased in proportion to an increase in data/problem size, the overall time should remain constant.





Parallel Systems

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



Parallel Systems

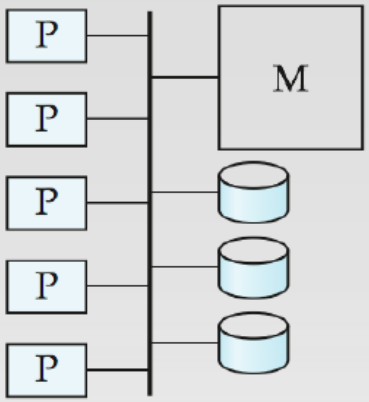
Two main performance measures:

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

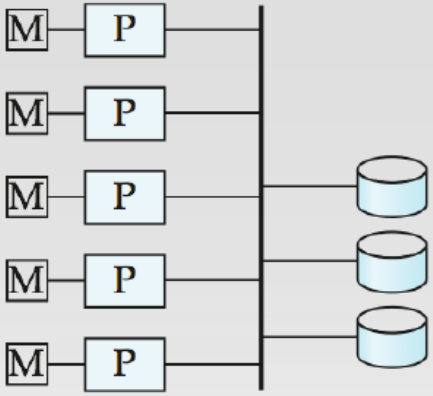
Parallel Database Architectures

- **Shared memory** --processors share a common memory
- **Shared disk** --processors share a common disk
- **Shared nothing** --processors share neither a common memory nor common disk
- **Hierarchical** --hybrid of the above architectures

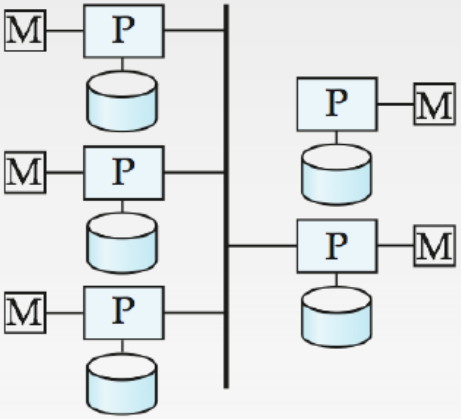
Parallel Database Architectures



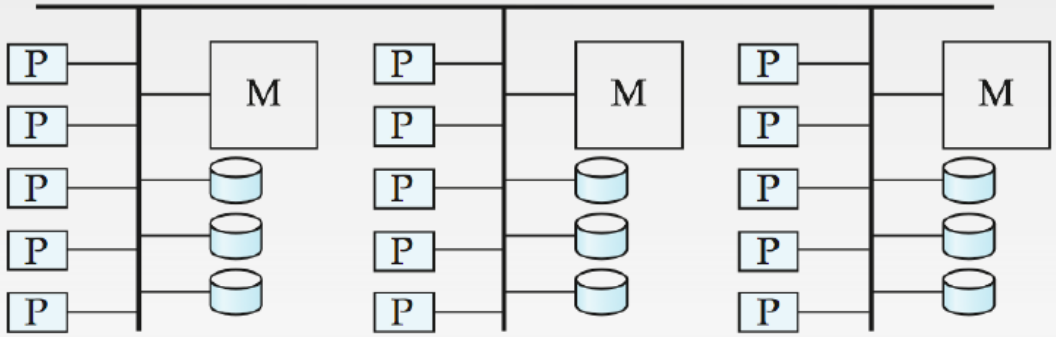
(a) shared memory



(b) shared disk



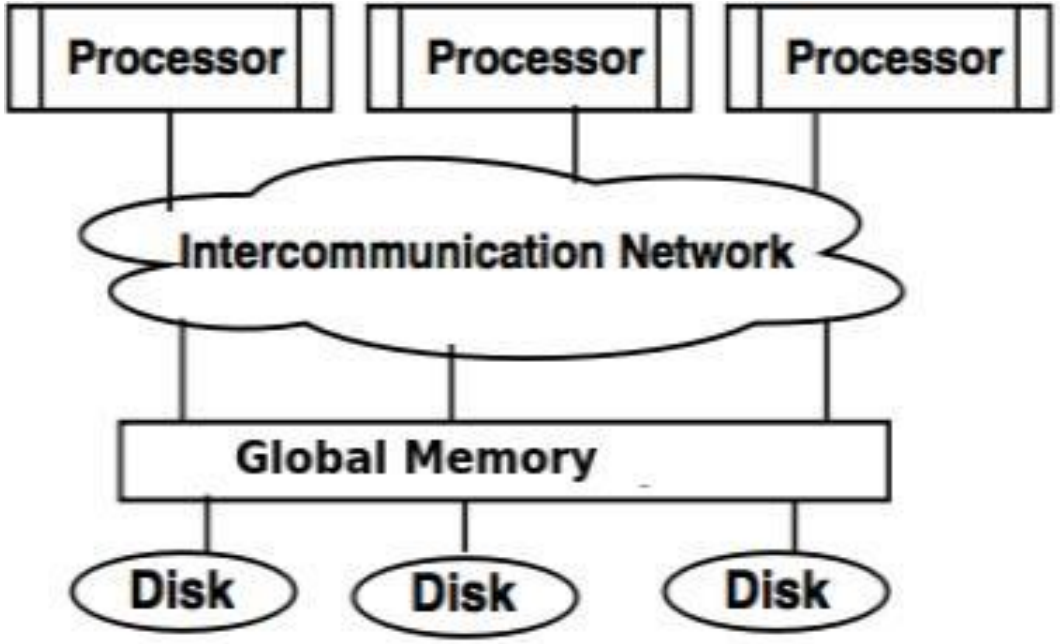
(c) shared nothing



(d) hierarchical

Architecture for parallel databases

Although data may be stored in a distributed fashion(in fragments), the distribution is governed solely by performance considerations. **Parallel databases** improve processing and input/output speeds by using multiple CPUs and disks in **parallel**. ... Shared memory **architecture**.



Shared Memory System in Parallel Databases

Why parallelism is needed in database?

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

Parallelism in Databases

- Data can be partitioned across multiple disks for parallel I/O.
- Individual relational operations (e.g., sort, join, aggregation) can be executed in parallel
  - data can be partitioned and each processor can work independently on its own partition.
- Queries are expressed in high level language (SQL, translated to relational algebra)
  - makes parallelization easier.
- Different queries can be run in parallel with each other. Concurrency control takes care of conflicts.
- Thus, databases naturally lend themselves to parallelism.

I/O Parallelism

- Reduce the time required to retrieve relations from disk by partitioning
- The relations on multiple disks.
- Horizontal partitioning – tuples of a relation are divided among many disks such that each tuple resides on one disk.
- Partitioning techniques (number of disks = *n*):

  Round-robin:

  Send the *i* th tuple inserted in the relation to disk *i* mod *n*.

  Ensures approx same tuples across disks

  Hash partitioning:

  - Choose one or more attributes as the partitioning attributes.
  - Choose hash function *h* with range 0…*n* - 1
  - Let *i* denote result of hash function *h* applied tothe partitioning attribute value of a tuple. Send tuple to disk *i*.

I/O Parallelism (Cont.)

- Partitioning techniques (cont.):
- **Range partitioning:**
  - Choose an attribute as the partitioning attribute.
  - A partitioning vector $[v_o, v_1, ..., v_{n-2}]$ is chosen.
  - Let v be the partitioning attribute value of a tuple. Tuples such that $v_i \leq v_{i+1}$ go to disk $i + 1$.
  - E.g., range partitioning with 3 disks numbered as 0, 1 and 2 may assign tuples with values less than 5 to disk 0, values between 5 and 40 to disk1 and values greater than 40 to disk 2.

Example

Comparison of Partitioning Techniques

- Evaluate how well partitioning techniques support the following types of data access:
  1. Scanning the entire relation.
  2. Locating a tuple associatively – **point queries**.
     - E.g., $r.A = 25$.
  3. Locating all tuples such that the value of a given attribute lies within a specified range – **range queries**.
     - E.g., $10 \leq r.A < 25$.

Comparison of Partitioning Techniques (Cont.)

Round robin:

- Advantages
  - Best suited for sequential scan of entire relation on each query.
  - All disks have almost an equal number of tuples; retrieval work is thus well balanced between disks.
- Range queries are difficult to process
  - No clustering -- tuples are scattered across all disks

Comparison of Partitioning Techniques (Cont.)

Hash partitioning:

- Good for sequential access
  - Assuming hash function is good, and partitioning attributes form a key, tuples will be equally distributed between disks
  - Retrieval work is then well balanced between disks.
- Good for point queries on partitioning attribute
  - Can lookup single disk, leaving others available for answering other queries.
  - Index on partitioning attribute can be local to disk, making lookup and update more efficient
  - Not suitable for point queries on non partitioning attributes
- No clustering, so difficult to answer range queries as hash functions do not preserve proximity within range

Comparison of Partitioning Techniques (Cont.)

Range partitioning:

- Provides data clustering by partitioning attribute value.
- Good for sequential access
- Good for point queries on partitioning attribute: only one disk needs to be accessed referring partitioning vector
- For range queries on partitioning attribute, one to a few disks may need to be accessed based on partitioning vectors
  - Remaining disks are available for other queries.
  - Good if result tuples are from one to a few blocks.
  - If many blocks are to be fetched, they are still fetched from one to a few disks, and potential parallelism is wasted in disk access due to bottleneck
    - Example of execution skew.

    (if most of the required the tuples are residing on single disk, then that disk is accessed for more time compare to other disk storing tuples in requested range)

Partitioning a Relation across Disks

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

Handling of Skew

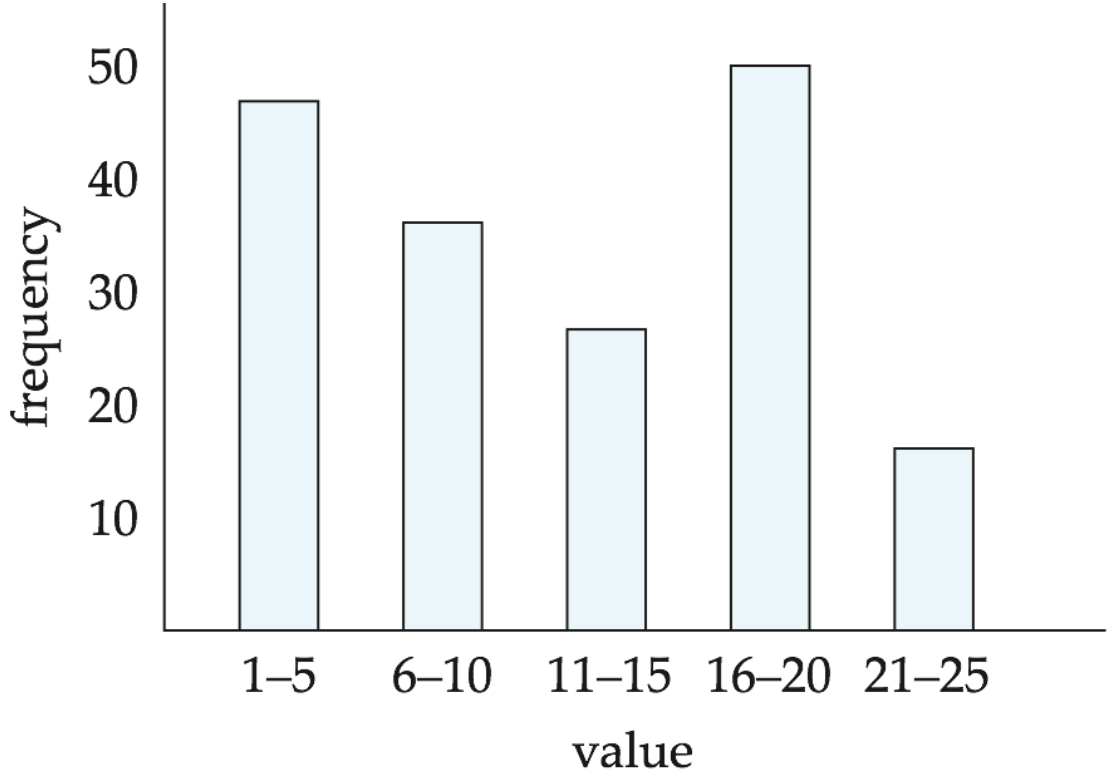
- The distribution of tuples to disks may be **skewed** — that is, some disks have many tuples, while others may have fewer tuples.
- **Types of skew:**
  - **Attribute-value skew**.
    - Same values appear in the partitioning attributes of many tuples; all the tuples with the same value for the partitioning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

Handling Skew in Range-Partitioning

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

Handling Skew using Histograms

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



Handling Skew Using Virtual Processor Partitioning

- Skew in range partitioning can be handled elegantly using **virtual processor partitioning**:
  - create a large number of partitions (say 10 to 20 times the number of processors)
  - Instead of real processors virtual processors are mapped to partitions
  - And virtual processors are mapped to real processors usually in round-robin fashion

  Basic idea:
  - if some range has more tuples than other because of skew then these tuples would get split across multiple virtual processors range.
  - And due to round-robin allocation of processors only one processor will not bear all the burden.

Interquery Parallelism

- Queries/transactions **execute in paralle**l with one another.
- **Increases transaction throughput**; used primarily to scale up a transaction processing system to **support a larger number of transactions per second.**
- Easiest form of parallelism to support, particularly in a **shared-memory parallel database, because even sequential database systems support concurrent processing.**
- More **complicated to implement on shared-disk or shared-nothing architectures**
  - Locking and logging must be coordinated by passing messages between processors.
  - Data in a local buffer may have been updated at another processor.
  - **Cache-coherency** has to be maintained — reads and writes of data in buffer must find latest version of data.

Cache Coherency Protocol

- Example of a cache coherency protocol for shared disk systems:
  - Before reading/writing to a page, the page must be locked in shared/exclusive mode.
  - On locking a page, the page must be read from disk
  - Before unlocking a page, the page must be written to disk if it was modified.
- More complex protocols with fewer disk reads/writes exist.
- Cache coherency protocols for shared-nothing systems are similar. Each database page is assigned a *home* processor.

  Requests to fetch the page or write it to disk are sent to the home processor.

Intraquery Parallelism

- Execution of a single query in parallel on multiple processors/disks; important for speeding up long-running queries.
- Two complementary forms of intraquery parallelism:
  - **Intraoperation Parallelism** – parallelize the execution of each individual operation in the query.
  - **Interoperation Parallelism** – execute the different operations in a query expression in parallel.

the first form scales better with increasing parallelism because the number of tuples processed by each operation is typically more than the number of operations in a query.

Parallel Processing of Relational Operations

- Our discussion of parallel algorithms assumes:
  - *read-only* queries(so no read write conflit)
  - shared-nothing architecture
  - *n* processors, P_0, ..., P_{n-1}, and *n* disks D_0, ..., D_{n-1}, where disk D_i is associated with processor P_i.
- If a processor has multiple disks they can simply simulate a single disk D_i.
- Shared-nothing architectures can be efficiently simulated on shared-memory and shared-disk systems.
  - Algorithms for shared-nothing systems can thus be run on shared-memory and shared-disk systems.
  - However, some optimizations may be possible.

Parallel Sort

Range-Partitioning Sort

- Choose processors P_0, ..., P_m, where $m \leq n$ -1 to do sorting.
- Create range-partition vector with m entries, on the sorting attributes
- Redistribute the relation using range partitioning
  - all tuples that lie in the i^{th} range are sent to processor P_i
  - P_i stores the tuples it received temporarily on disk D_i.
  - This step requires I/O and communication overhead.
- Each processor P_i sorts its partition of the relation locally.
- Each processors executes same operation (sort) in parallel with other processors, without any interaction with the others (**data parallelism**).
- Final merge operation is trivial: range-partitioning ensures that,for i, j m, the key values in processor P^i are all less than the key values in P_j.

Parallel Sort (Cont.)

Parallel External Sort-Merge

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

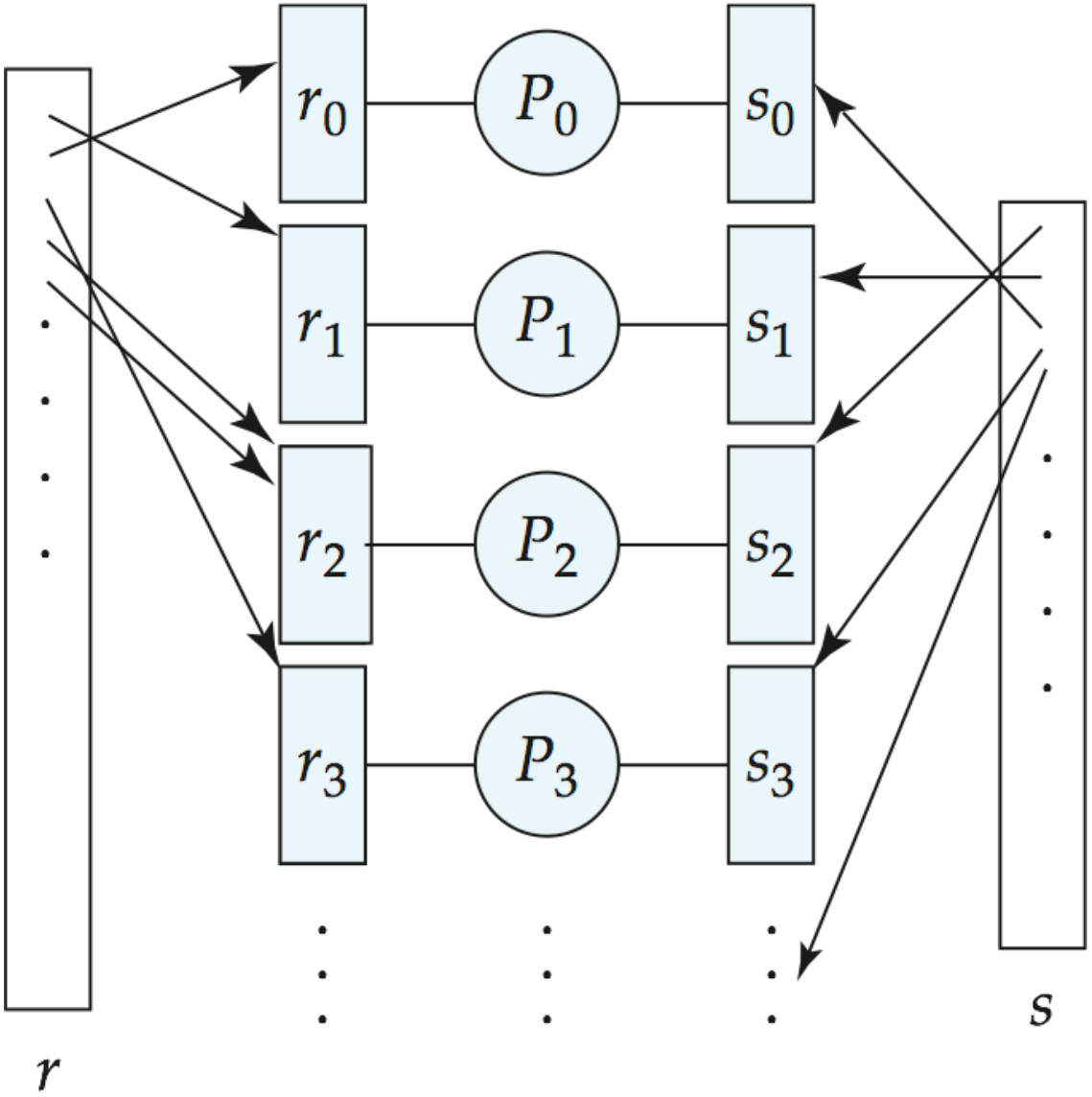
Parallel Join

- The join operation requires pairs of tuples to be tested to see if they satisfy the join condition, and if they do, the pair is added to the join output.
- Parallel join algorithms attempt to split the pairs to be tested over several processors. Each processor then computes part of the join locally.
- In a final step, the results from each processor can be collected together to produce the final result.

Partitioned Join

- For equi-joins and natural joins, it is possible to *partition* the two input relations across the processors, and compute the join locally at each processor.
- Let *r* and *s* be the input relations, and we want to compute $r \bowtie_{r.A=s.B} s$.
- *r* and *s* each are partitioned into *n* partitions, denoted $r_0, r_1, ..., r_{n-1}$ and $s_0, s_1, ..., s_{n-1}$.
- Can use either *range partitioning* or *hash partitioning*.
- *r* and *s* must be partitioned on their join attributes *r*.A and *s*.B), using the same range-partitioning vector or hash function.
- Partitions r_i and s_i are sent to processor P_i,
- Each processor P_i locally computes $r_i \bowtie_{ri.A=si.B} s_i$. Any of the standard join methods can be used.

Partitioned Join (Cont.)

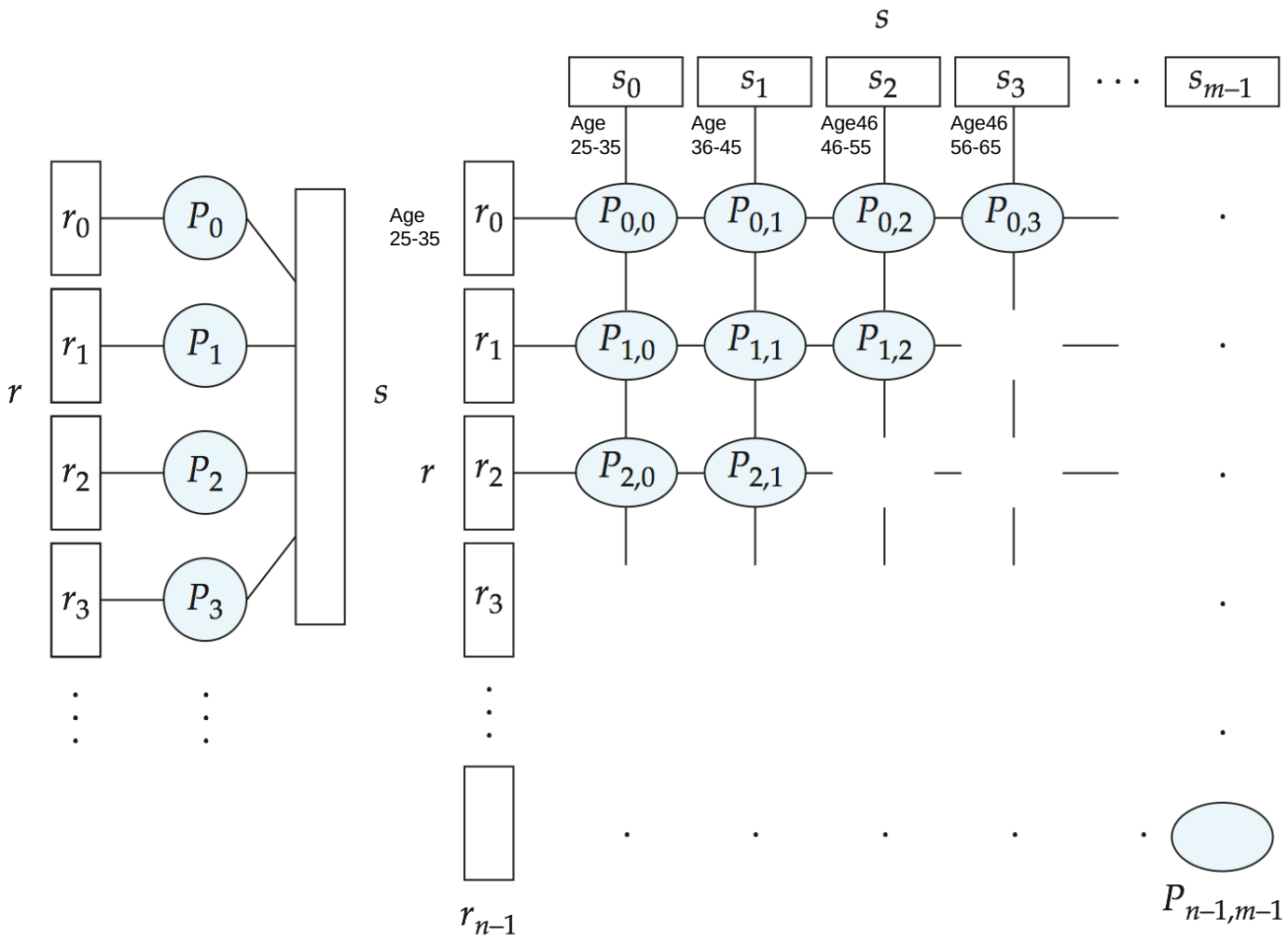


Fragment-and-Replicate Join

- Partitioning not possible for some join conditions
  - E.g., non-equijoin conditions, such as r.A > s.B.
- For **joins where partitioning is not applicable**, parallelization can be accomplished by **fragment and replicate** technique
  - It is possible that all tuples in relation r will join with some tuples in relation s
- Special case – **asymmetric fragment-and-replicate**:
  - One of the relations, say *r*, is partitioned; any partitioning technique can be used.
  - The other relation, *s(build relation)* is replicated across all the processors.
  - Processor P_i then locally computes the join of r_i with all of s using any join technique.

Depiction of Fragment-and-Replicate Joins

r.age<=s.age



(a) Asymmetric fragment and replicate

(b) Fragment and replicate

Fragment-and-Replicate Join (Cont.)

- General case: reduces the sizes of the relations at each processor.
  - r is partitioned into n partitions,r_0, r_1, ..., r_{n-1};s is partitioned into m partitions, s_0, s_1, ..., s_{m-1}.
  - Any partitioning technique may be used.
  - There must be at least m * n processors.
  - Label the processors as
  - $P_{0,0}$, $P_{0,1}$, ..., $P_{0,m-1}$, $P_{1,0}$, ..., P_{n-1m-1}.
  - $P_{i,j}$ computes the join of r_i with s_j. In order to do so, **r_i is replicated to $P_{i,0}$, $P_{i,1}$, ..., $P_{i,m-1}$, while s_i is replicated to $P_{0,i}$, $P_{1,i}$, ..., $P_{n-1,i}$**
  - Any join technique can be used at each processor $P_{i,j}$.

Fragment-and-Replicate Join (Cont.)

- **Both versions of fragment-and-replicate work with any join condition,** since every tuple in *r* can be tested with every tuple in *s*.
- Usually **has a higher cost than partitioning**, since one of the relations (for asymmetric fragment-and-replicate) or both relations (for general fragment-and-replicate) have to be replicated.
- **Sometimes asymmetric fragment-and-replicate is preferable even though partitioning could be used**.
  - E.g., **say *s* is small** and *r* is large, and already partitioned. It may be cheaper to replicate *s* across all processors, rather than repartition *r* and *s* on the join attributes.

Partitioned Parallel Hash-Join

Parallelizing partitioned hash join:

- Assume ***s* is smaller** than *r* and therefore ***s* is chosen as the build** relation.
- A hash function h_1 takes the **join attribute value of each tuple in *s*** and maps this tuple to one of the *n* processors.
- Each processor P_i reads the tuples of *s* that are on its disk D_i, and sends each tuple to the appropriate processor based on hash function h_1. Let s_i denote the tuples of relation *s* that are sent to processor P_i.
- As tuples of relation *s* are received at the destination processors, they are partitioned further using another hash function, h_2, which is used to compute the hash-join locally. *(Cont.)*

Partitioned Parallel Hash-Join (Cont.)

- Once the tuples of *s* have been distributed, the larger relation *r* is redistributed across the *m* processors using the hash function h_1
  - Let r_i denote the tuples of relation *r* that are sent to processor P_i.
- As the *r* tuples are received at the destination processors, they are repartitioned using the function h_2
- Each processor P_i executes the build and probe phases of the hash-join algorithm on the local partitions r_i and *si* of *r* and *s* to produce a partition of the final result of the hash-join.
- Note: Hash-join optimizations can be applied to the parallel case
  - e.g., the hybrid hash-join algorithm can be used to cache some of the incoming tuples in memory and avoid the cost of writing them and reading them back in.

Other Relational Operations

Selection $\sigma_\theta(r)$

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

Other Relational Operations (Cont.)

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

Other Relational Operations (Cont.)

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

Cost of Parallel Evaluation of Operations

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

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

  - T_{part} is the time for partitioning the relations
  - T_{asm} is the time for assembling the results
  - T_i is the time taken for the operation at processor P_i
    - this needs to be estimated taking into account the skew, and the time wasted in contentions.

Interoperator Parallelism

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

Factors Limiting Utility of Pipeline Parallelism

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

Independent Parallelism

- **Independent parallelism**
  - Consider a join of four relations

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

Query Optimization

- Query optimization in parallel databases is significantly more complex than query optimization in sequential databases.
- Cost models are more complicated, since we must take into account partitioning costs and issues such as skew and resource contention.
- When **scheduling** execution tree in parallel system, must decide:
  - How to parallelize each operation and how many processors to use for it.
  - What operations to pipeline, what operations to execute independently in parallel, and what operations to execute sequentially, one after the other.
- Determining the amount of resources to allocate for each operation is a problem.
  - E.g., allocating more processors than optimal can result in high communication overhead.
- Long pipelines should be avoided as the final operation may wait a lot for inputs, while holding precious resources

End of Chapter

Figure 18.01

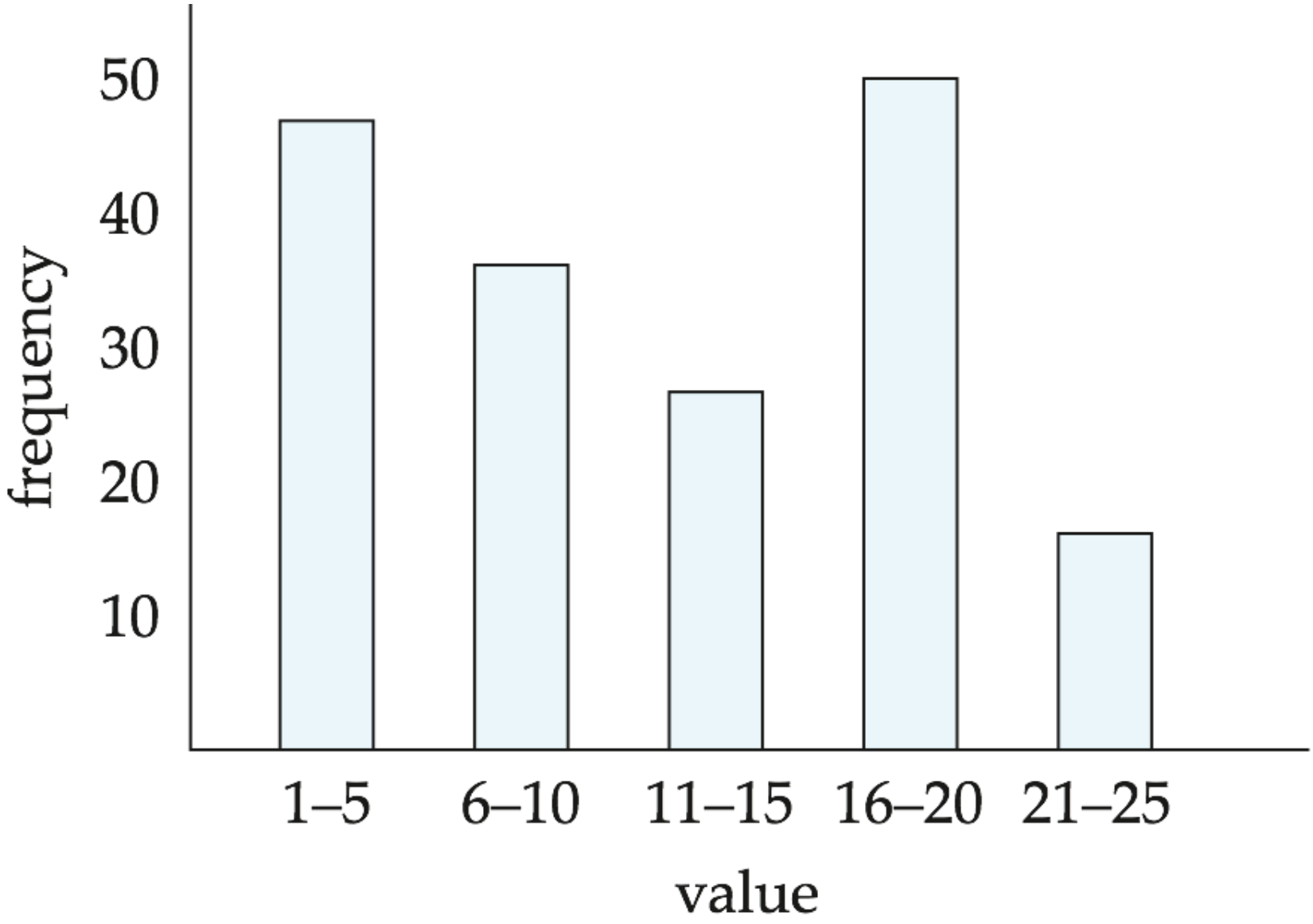


Figure 18.02

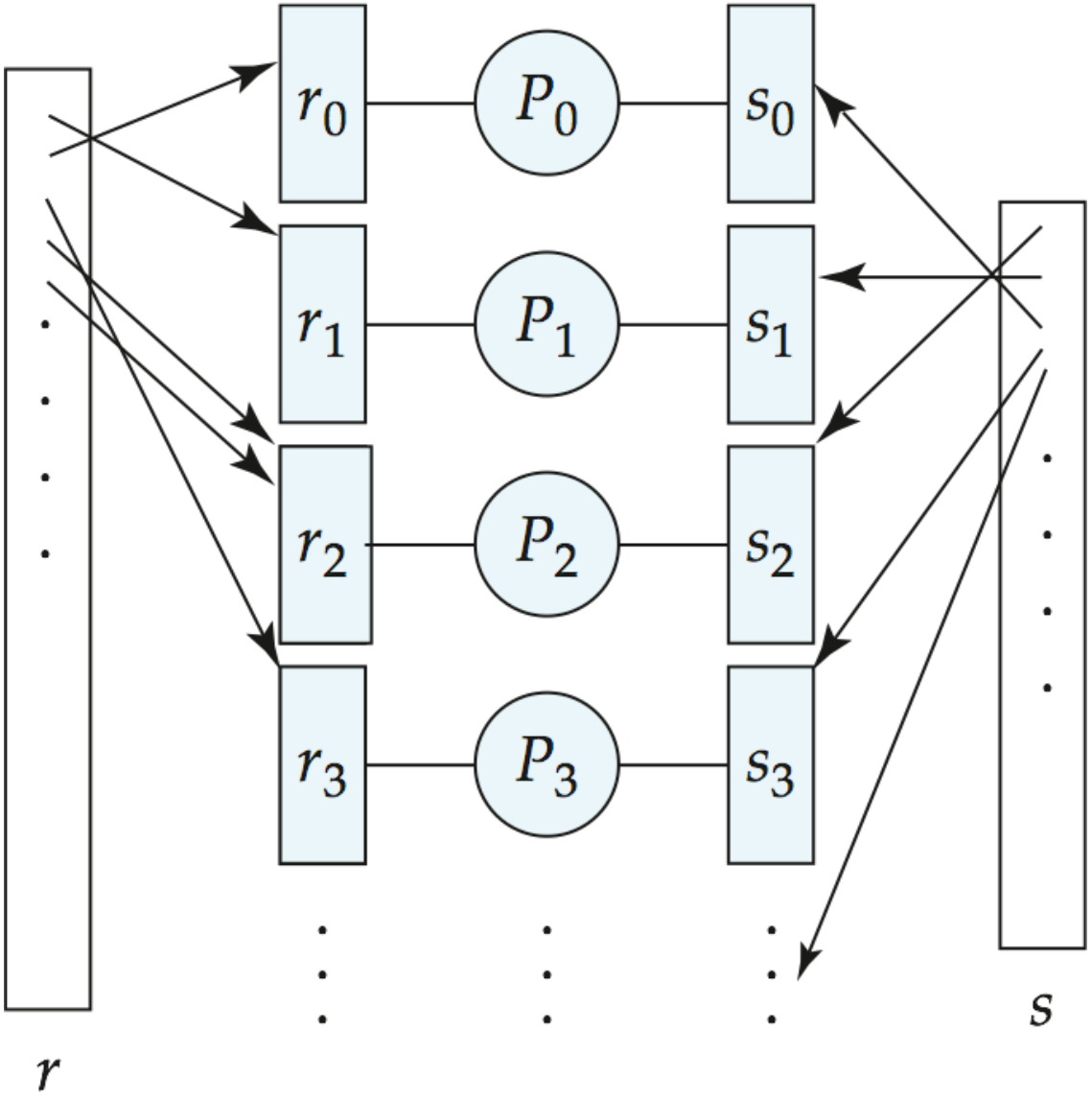
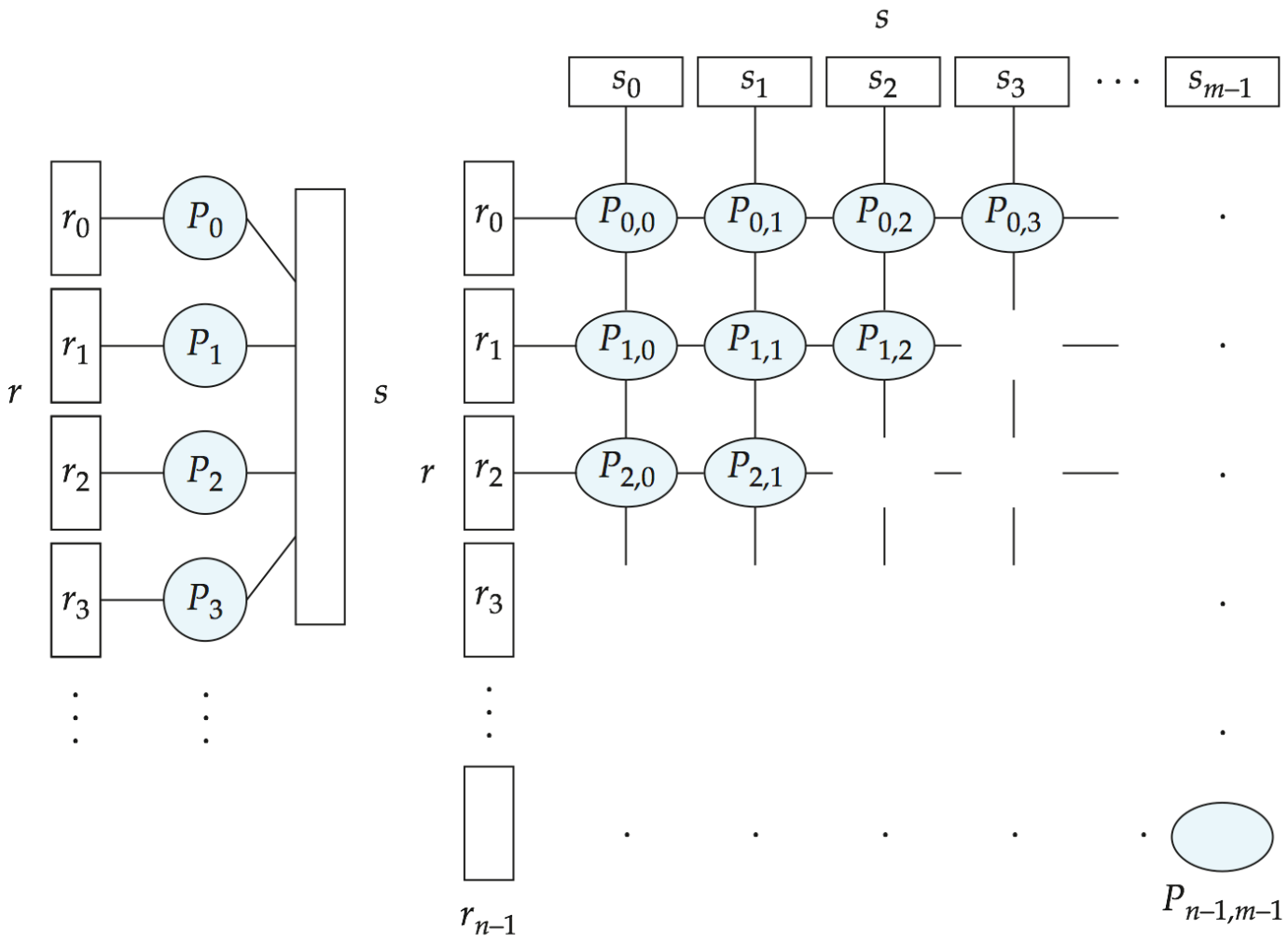


Figure 18.03



(a) Asymmetric fragment and replicate

(b) Fragment and replicate