Chapter 18: Parallel Databases

Database System Concepts, 6th Ed.

Chapter 18: Parallel Databases

- Introduction
- I/O Parallelism
- Interquery Parallelism
- Intraquery Parallelism
  - Intraoperation Parallelism
    - Parallel Sort (2 types)
    - Parallel Join (4 types)
    - Other Relational Operations
    - Cost of Parallel Evaluations of Operations
  - Interoperation Parallelism
    - Pipelined Parallelism
    - Independent Parallelism
- Design of Parallel Systems

Introduction

- **Why Parallel Database System?**
- 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 audio, video, 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.
- Parallelism is used to provide speedup, where queries are executed faster because more resources, such as processors and disks, are provided.
- Parallelism is used to provide scaleup, where increasing workloads are handled without increased response time, via an increase in the degree of parallelism.

I/O Parallelism

- **I/O parallelism** refers to reduce the time required to retrieve relations from disk by partitioning the relations on multiple disks.
- Most common - *Horizontal partitioning* – records of a relation are divided (or declustered) among many disks, such that each record resides on one disk.
- **Partitioning Techniques:** Several partitioning strategies have been proposed. Assume there are *n disks,* D_0, D_1,....D_{n-1}, across which data are to be partitioned:

  1. **Round-robin**:
     - Scan the relation in any order and send the i^{th} record to the disk number $D_{i \bmod n}$.
     - This scheme ensures an even distribution of records across disks; that is, each disk has approximately the same number of records as the others.

  2. **Hash partitioning**:
     - This declustering strategy designates one or more attributes from the given relation's schema as the partitioning attributes.

I/O Parallelism (Cont.)

- A hash function is chosen with a range {0, 1,…, *n* -1}
- Each record of the original relation is hashed on the partitioning attributes. If the hash function returns *i* , then the record is places on disk D*i* .

● **Range partitioning**:

- This strategy distributes contiguous attribute-value range to each disks.
- Chooses a partitioning attribute, A, as a partitioning vector. Let [v_o, v_1, ..., v_{n-2}] denotes the partitioning vector, such that, if i< j, then $v_i < v_j$.
- Let x be the partitioning attribute (A) value of a record, t[A] = x. If x < $v_{0,}$, then t goes on disk D_0. If $x \geq v_{n-2}$, t goes on disk D_{n-1}. If $v_i \leq x < v_{i+1,}$ then t goes on disk D_{i+1}.

E.g., with a partitioning vector [5,11], a record with partitioning attribute value of 2 will go to disk 0, a record with value 8 will go to disk 1, while a record with value 20 will go to disk2.

Comparison of Partitioning Techniques

- Once a relation has been partitioned among several disks, we can retrieve it in parallel, using all the disks.
- Similarly, when a relation is being partitioned, it can be written to multiple disks in parallel.
- Thus, the transfer rates for reading or writing an entire relation are much faster with I/O parallelism than without it.
- Evaluate how well partitioning techniques support the following types of data access:

1. Scanning the entire relation.
2. Locating a record associatively – **point queries**.
    - E.g., $r.A = 25$.
3. Locating all records 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 records; retrieval work is thus well balanced between disks.
- Point queries and range queries are difficult to process, since each of the n disks must be used for the search.
  - No clustering - records are scattered across all disks

Comparison of Partitioning Techniques(Cont.)

Hash partitioning:

- Useful for sequential scans of the entire relation
  - Assuming hash function is good randomizing function, and partitioning attributes form a key, tuples will be equally distributed among disks
  - Retrieval work is then well balanced between disks. The time taken to scan the relation is approximately 1/n of the time required to scan in a single disk system.
- Best suited for point queries on partitioning attribute
  - Directing a query in a single disk saves the startup cost of initiating a query on multiple disks.
  - Leaves the other disks free for answering other queries.
  - Index on partitioning attribute can be local to disk, making lookup and update more efficient
- Not well suited for point queries on non-partitioning attribute.
- Not well suited for answering range queries, since, hash functions do not preserve proximity within a range. Therefore, all the disks need to be scanned for range queried to be answered.

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.
- For range queries on partitioning attribute, one to a few disks may need to be accessed
  - Remaining disks are available for other queries.
  - Good if result records are from one to a few blocks.
- If there are many tuples in the queried range (a larger fraction of the domain of the relation), many tuples have to be retrieved from a few disks, resulting in an I/O bottleneck (hot spot) at those disks. This is an example of **execution skew**, all processing occurs in one—or only a few—partitions**.** and potential parallelism in disk access is wasted.
- In contrast, hash partitioning and round-robin partitioning would engage all the disks for such queries, giving a faster response time for approximately the same throughput.

Comparison of Partitioning Techniques (Cont.)

- **Comment:** The type of partitioning also affects other relational operations, such as joins. Thus the choice of partitioning technique also depends on the operations that need to be executed.
- In general, hash partitioning or range partitioning are preferred to round-robin partitioning.

Partitioning a Relation across Disks

- In a system with many disks, the number of disks across which to partition a relation can be chosen in this way:
  - If a relation contains only a few records 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

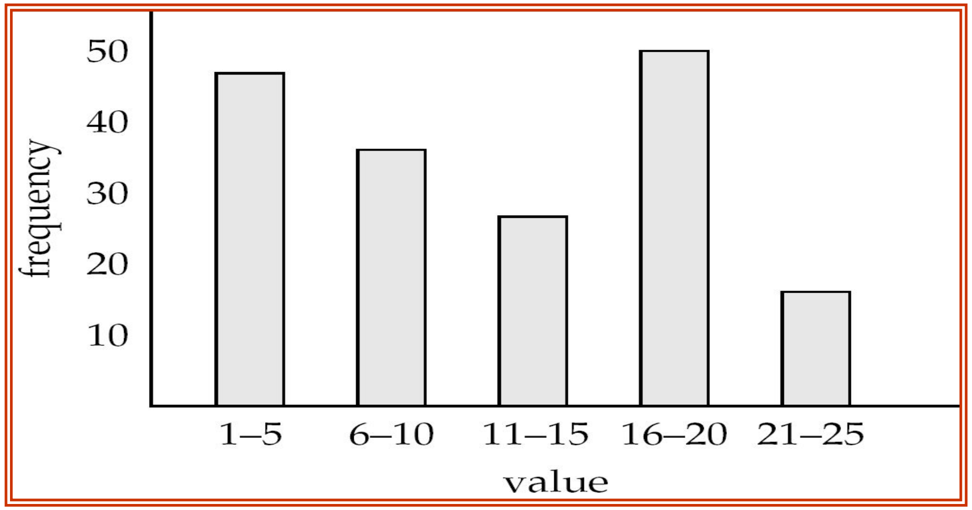
- When a relation is partitioned (other than round-robin), the distribution of records to disks may be **skewed** — that is, some disks have many records, while others may have fewer records.
- **Types of skew:**
  - **Attribute-value skew.**
    - 4 Some values appear in the partitioning attributes of many records; all the records with the same value for the partitioning attribute end up in the same partition.
    - 4 Can occur with range-partitioning and hash-partitioning.
  - **Partition skew**.
    - 4 Partition skew refers to the fact that there may be load imbalance in the partitioning, even when there is no attribute skew.
    - 4 With range-partitioning, badly chosen partition vector may assign too many records to some partitions and too few to others.
    - 4 Less likely with hash-partitioning if a good hash-function is chosen.

Handling Skew in Range-Partitioning

- Even a small skew can result in a significant decrease in performance. Skew becomes an increasing problem with higher degree of parallelism.
- To create a balanced partitioning vector :
  - 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 record is added to the partition vector.
  - *n* denotes the number of partitions to be constructed.
  - Imbalances or some skew can still result if duplicates are present in partitioning attributes.
  - The main disadvantage of this method is the extra I/O overhead incurred in doing the initial sort.
- Alternative technique based on **histograms** (frequency table) used in practice

Handling Skew using Histograms

- Balanced partitioning vector can be constructed by storing a frequency table or histogram of the attribute values for each attribute of each relation.
- A histogram takes up only a little space, so histograms on several different attributes can be stored in the system catalogue.
- It is straightforward to construct a balanced range-partitioning function given a histogram on the partitioning attributes.
- Histogram can be constructed by scanning relation, or sampling (random blocks) records of the relation, if histogram is not stored.



Handling Skew Using Virtual Processor Partitioning

- To minimize the effect of skew in range partitioning **virtual processor partitioning** technique can be used elegantly:
  - We pretend there are several times (10 to 20 times) as many *virtual processors* as the number of real processors
  - Any of the partitioning techniques and query-evaluation techniques can be used, but they map tuples and work to virtual processors instead of to real processors.
  - Virtual processors, in turn, are mapped to real processors, usually by round-robin partitioning.
- Basic idea:
  - Even if one range had many more tuples than the others because of skew, these tuples would get split across multiple virtual processor ranges.
  - Round-robin allocation of virtual processors to real processors would distribute the extra work among multiple real processors, so that one processor does not have to bear all the burden.

Interquery Parallelism

- In interquery parallelism queries or transactions execute in parallel with one another.
- Increases transaction **throughput**. However, the **response times** of individual transactions are no faster than they would be if the transactions were run in isolation. Thus, the primary use of this method is 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 system. Database systems designed for single-processor systems can be used with few or no changes on a shared memory parallel architecture, since even sequential database systems support concurrent processing.
- Transactions that would have operated in the time-shared concurrent manner on a sequential machine operate in parallel in the shared-memory parallel architecture.

Interquery Parallelism

- More complicated to implement on shared-disk or shared-nothing architectures
  - Locking and logging must be coordinated by passing messages to each other.
  - Must also ensure that two processors do not update the same data independently at the same time.
  - When a processor accesses or updates data, the database system must ensure that the processor has the latest version of the data in the buffer pool.
  - **This means, Cache-coherency** has to be maintained — (the problem of ensuring that the version is latest).
- Various protocols are available to guarantee cache coherency; often they are integrated with concurrency-control protocols to reduce the overhead. One such protocol for shared-disk system is:

Cache Coherency Protocol

  - 1. Before reading/writing to a page, the page must be locked in shared/exclusive mode, as appropriate. Immediately after the transaction obtains a lock on a page, it also reads the most recent copy of the page form the shared disk.
  - 2. Before a transaction releases an exclusive lock on a page, it flushes the page to the shared disk and then releases the lock.
- More complex protocols with fewer disk reads/writes exist. When a shared or exclusive lock is obtained, if the most recent version of a page is in the buffer pool of a processor, the page is obtained form there. The protocols have to be designed to handle concurrent requests
- The shared-disk protocols can be extended for shared-nothing systems. Each database page is assigned a **home processor**, Pi and is stored in disk D_i. When other processors want to read or write the page, they send requests to the home processor Pi of the page, since they cannot directly communicate with the disk. The other actions are same as in the shared-disks protocols.
- Example: The Oracle and Oracle Rdb systems use shared-disk interquery parallelism.

Intraquery Parallelism

- **Intraquery parallelism** refers to the execution of a single query in parallel on multiple processors and disks; important for speeding up of long-running queries. Interquery parallelism does not help in this task, since each query is run sequentially.
- Two complementary forms of intraquery parallelism :
  - **Intraoperation Parallelism** – can speed up processing of a query by parallelizing the execution of each individual operation such as sort, select, project and join.
  - **Interoperation Parallelism** – can speed up processing of a query by executing in parallel the different operations in a query expression.
- The two forms of parallelism are complementary, and can be used simultaneously on a query. Since the number of operations in a typical query is small, compared to the number of tuples processed by each operation, the first form of parallelism can scale better with increasing parallelism.
- However, with the relatively small number of processors in typical parallel systems today, both forms of parallelism are important.

Intraquery Parallelism

- Our discussion of parallel algorithms assumes:
  - *read-only* queries
  - 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.

Intraoperation Parallelism

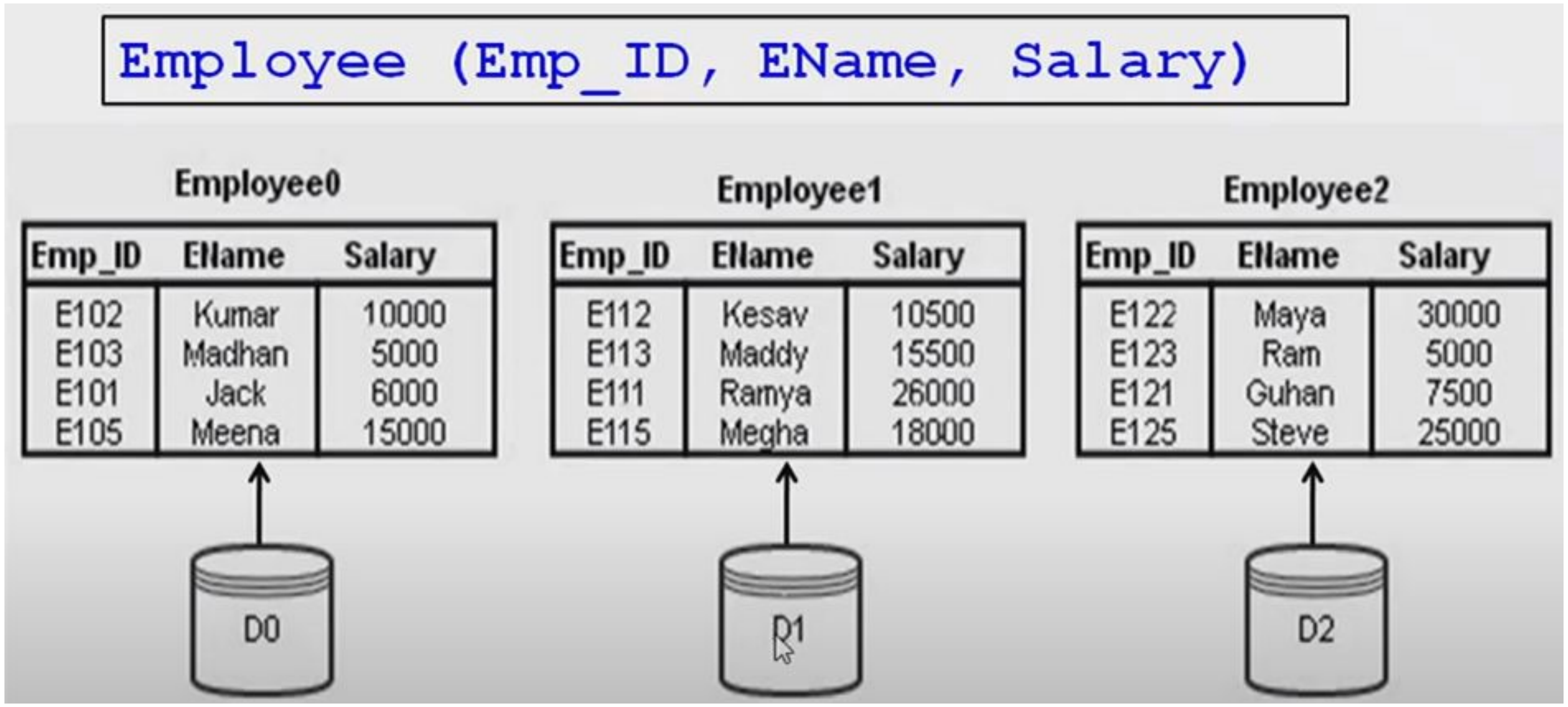
- Since relational operations work on relations containing large sets of records, we can parallelize the operations by executing them in parallel on different subsets of the relations.
- Since the number of records in a relation can be large, the degree of parallelism is potentially enormous. Thus, intraoperation parallelism is natural in a database system.
- The parallel versions of some common relational operations are:
  - Parallel Sort
    - 4 Range-Partitioning Sort
    - 4 Parallel External Sort - Merge
  - Parallel Join
    - 4 Partitioned Join
    - 4 Fragment-and-Replicate Join
    - 4 Partitioned Parallel Hash Join
    - 4 Parallel Nested-Loop Join

Parallel Sort

- Suppose that we wish to sort a relation that resides on *n* disks D_0, D_1 ..., D_{n-1} . If the relation has been range partitioned on the attributes on which it is to be sorted, then, we can sort each partition separately and can concatenate the results to get the full sorted relation.
- Since the records are partitioned in *n* disks, the time required for reading the entire relation is reduced by the parallel access.
- If the relation is partitioned in any other way, it can be sorted in any one of the two ways:
  - 1. Range partition it on the sort attribute and then sort each partition separately.
  - 2. Use a parallel version of the external sort-merge algorithm.

Parallel Sort - Range-Partitioning Sort

- Let us consider the following relation schema **Employee**
- ***Employee (Emp_ID, Ename, Salary)***
- Here the table is not partitioned by the attribute **Salary**
- Rather it is partitioned by **Emp_ID**



```
Employee (Emp_ID, EName, Salary)
```

Employee0

Emp_ID	EName	Salary
E102	Kumar	10000
E103	Madhan	5000
E101	Jack	6000
E105	Meena	15000

Employee1

Emp_ID	EName	Salary
E112	Kesav	10500
E113	Maddy	15500
E111	Ramya	26000
E115	Megha	18000

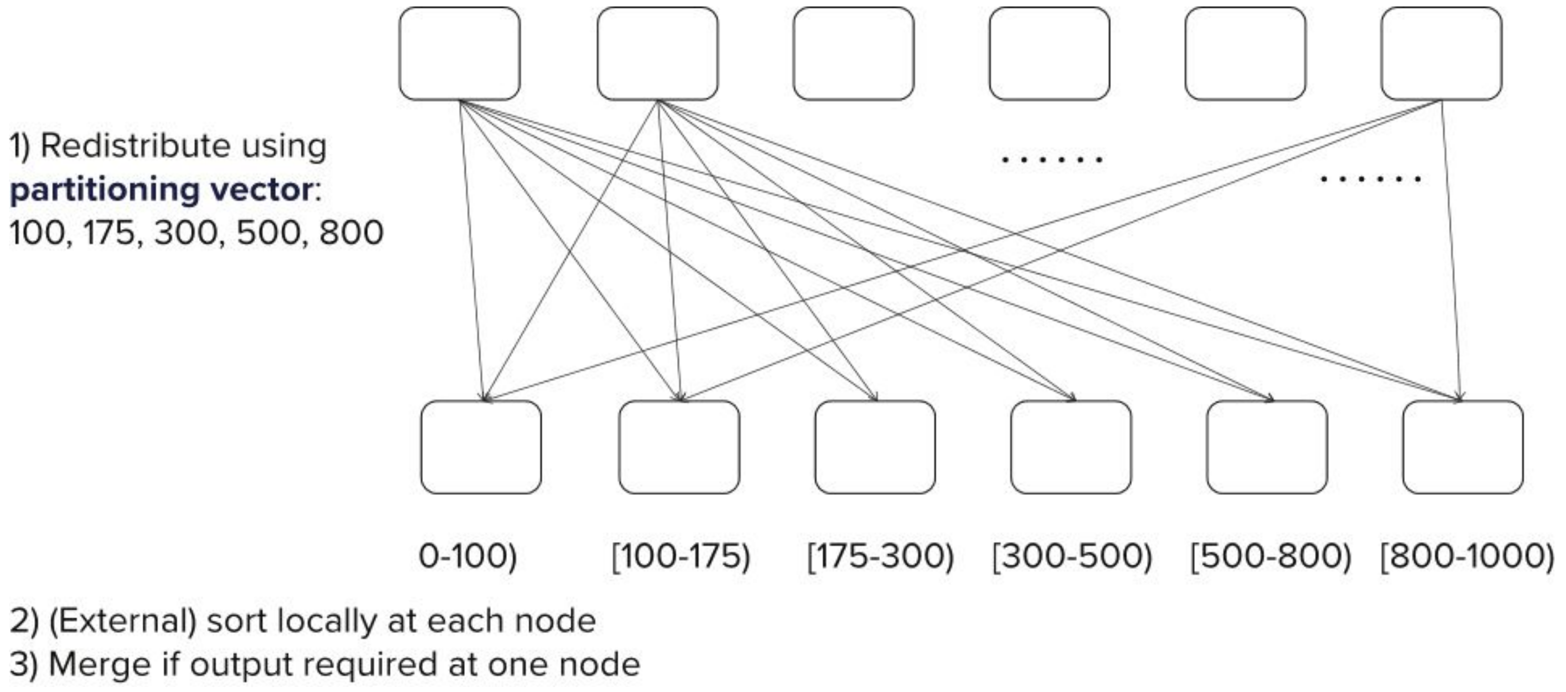
Employee2

Emp_ID	EName	Salary
E122	Maya	30000
E123	Ram	5000
E121	Guhan	7500
E125	Steve	25000

Range-Partitioning Sort

- Range-partitioning sort works in two steps: 1. range partitioning the relation, 2. sorting each partition separately. It is not necessary to range partition the relation on the same set of processors or disks as those on which that relation is stored.
- Choose processors P_0, ..., P_m, where $m < n$ to do sorting.
- Create range-partition vector with m entries, on the sorting attributes
- 1. Redistribute the relation using range partitioning
  - In parallel every processor reads the tuples from its disk and sends the tuples that lie in the ith range to processor P_i
  - P_i stores the tuples it received temporarily on disk D_i.
  - This step requires I/O and communication overhead.
- 2. Sort the partition
  - 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 $1 <= i < j <= m$, the key values in processor P_i are all less than the key values in P_j.
- Virtual processor partitioning can also be used to reduce skew

Parallel Sort - Range-Partitioning Sort



- We must do range partitioning with a balanced range-partition vector so that each partition will have approximately the same number of tuples
- Virtual processor partitioning can also be used to reduce skew

Parallel External Sort-Merge

- Parallel external sort-merge is an alternative to range partitioning.
- Assume the relation has already been partitioned among disks D_0, ..., D_{n-1} (in whatever manner). Then this method works this way:
  - 1. Each processor P_i locally sorts the data on disk D_i.
  - 2. The sorted runs on each processor are then merged to get the final sorted output.
- Parallelize the merging of sorted runs in step 2 by this seq. of actions:
  - 1. The system range partitions the sorted partitions at each processor P_i (all by the same partition vector) across the processors P_0, ..., P_{m-1}. It sends the tuples in sorted order, so that each processor receives the tuples in sorted streams.
  - 2. Each processor P_i performs a merge on the streams as they are received, to get a single sorted run.
  - 3. The sorted runs on processors P_0,..., P_{m-1} are concatenated to get the final result.
- An interesting form of **execution skew** can result (Next Slide).
- Example: Teradata Purpose-Built Platform Family machine

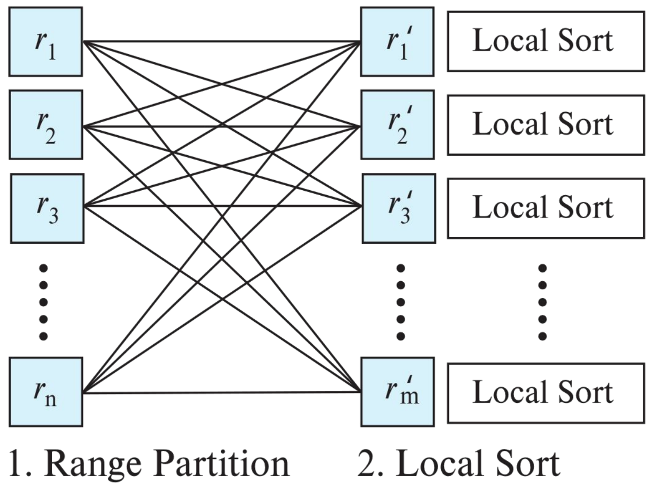
Execution Skew

- At first every processor sends all blocks of partition 0 to P_0, then every processor sends all blocks of partition 1 to P_1, and so on.
- Thus, while sending happens in parallel, receiving tuples becomes sequential: First only P_0 receives tuples, then only P_1 receives tuples, and so on.
- To avoid this problem, each processor repeatedly sends a block of data to each partition.
- In other words, each processor sends the first block of every partition, then sends the second block of every partition, and so on. As a result, all processors receive data in parallel.

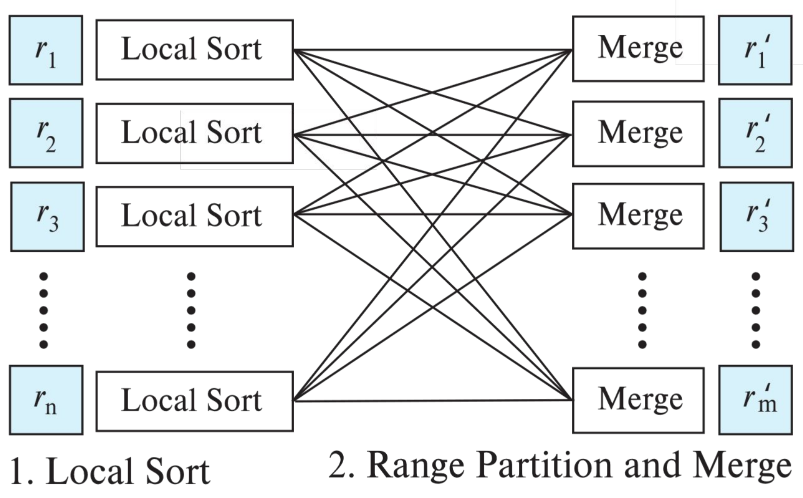
Execution Skew

- This sequence of actions results in an interesting form of **execution skew**, since
- At first every node sends all tuples of partition 1 to N_1, then every node sends all tuples of partition 2 to N_2, and so on.
- Thus, while sending happens in parallel, receiving tuples becomes sequential: First only N_1 receives tuples, then only N_2 receives tuples, and so on.
- To avoid this problem, the sorted sequence of tuples $S_{i,j}$ from any node i destined to any other node j is broken up into multiple blocks.
- Each node N_i sends the first block of tuples from $S_{i,j}$ node N_j, for each j; it then sends the second block of tuples to each node N_j, and so on, until all blocks have been sent.
- As a result, all nodes receive data in parallel. (Note that tuples are sent in blocks, rather than individually, to reduce network overheads.)

Parallel Sort



(a) Range Partitioning Sort



(b) Parallel External Sort-Merge

Parallel Join

- **Basic Idea**
- The join operation requires pairs of records 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 (hash join, merge join or nested-loop join) can be used.

Partitioned Join (Cont.)

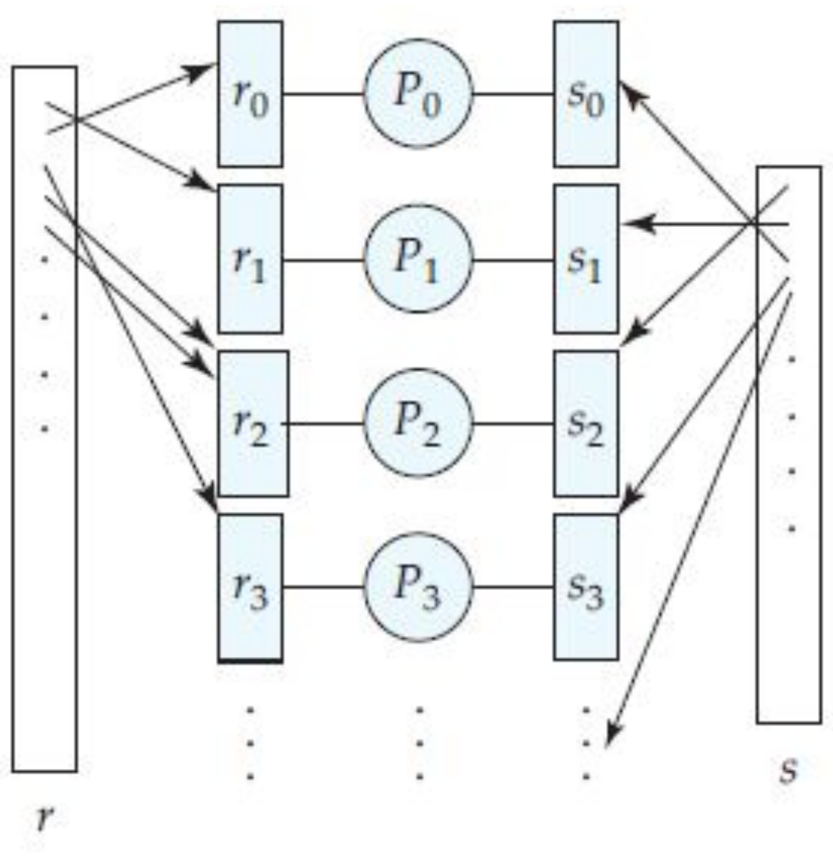


Figure 18.2 Partitioned parallel join.

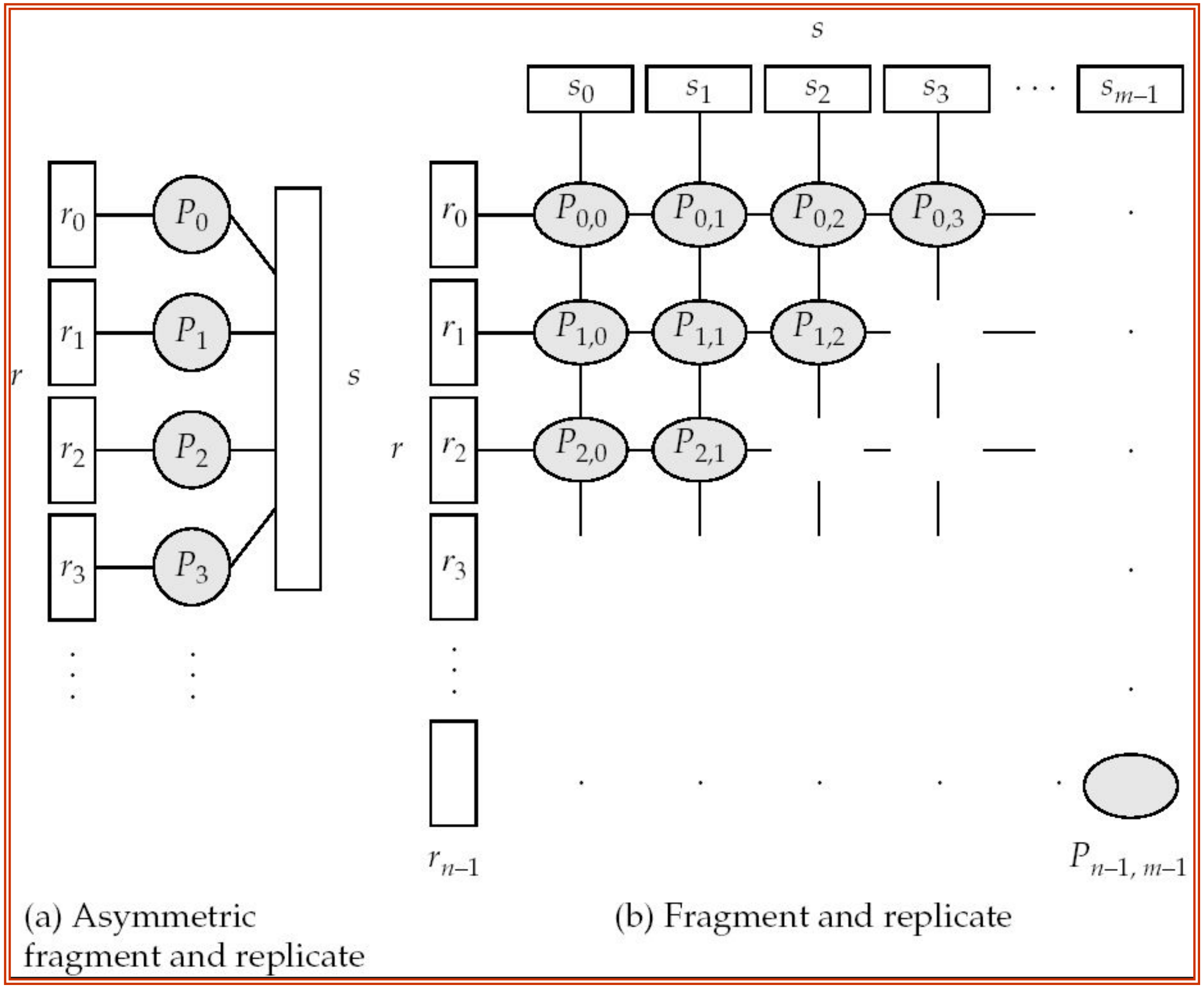
Partitioned Join (Cont.)

- If one or both the relations r and s are already partitioned on the join attributes, the work needed for partitioning is greatly reduced.
- If the relations are not partitioned or are partitioned on attributes other than the join attributes, then the tuples need to be repartitioned.
- We can optimized the join algorithms used locally at each processor to reduce I/O by buffering some of the tuples to memory, instead of writing them to disk.
- Skew presents a special problem when range partitioning is used, since a partition vector that splits one relation of the join into equal-sized partitions may split the other relations into partitions of widely varying size.
- The partition vector should be such that the sum of the sizes of r_i and s_i is roughly equal over all i = 0, 1, …., n-1
- With a good hash function, hash partitioning is likely to have a smaller skew, except when there are many tuples with the same values for the join attributes.

Fragment-and-Replicate Join

- Partitioning not possible for some join conditions
  - e.g., non-equijoin (inequality)conditions, such as r.A > s.B.
- For joins where partitioning is not applicable, parallelization can be accomplished by **fragment and replicate** technique
  - Depicted on next slide
- **Special case** – **asymmetric fragment-and-replicate**:
  - 1. One of the relations, say *r*, is partitioned; any partitioning technique can be used, including round-robin partitioning.
  - 2. The other relation, *s*, is replicated across all the processors.
  - 3. Processor P_i then locally computes the join of r_i with all of s using any join technique.
- If r is already stored by partitioned, no need to partition it further, just replicate s across all the processors.

Depiction of Fragment-and-Replicate Joins



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.
  - The values of *m* and *n* do not need to be equal, 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 $P_{i,j}$.
- Asymmetric fragment and replicate is simply a special case of general fragment and replicate, where $m = 1$.

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:

- Suppose we have n processors and two relations *r* and *s* , such that the relations *r* and *s* are partitioned across multiple disks.
- Assume *s* is smaller than *r* and therefore *s* is chosen as the build relation and the parallel hash-join algorithm proceeds this way:
- 1. A hash function h_1 takes the join attribute value of each tuple in *r* and *s* and maps the tuple to one of the *n* processors.
- 2. a. 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.
- 2. b. 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. Each processor P_i executes this step independently from the other processors.

Partitioned Parallel Hash-Join (Cont.)

- 3. a. Once the tuples of *s* have been distributed, the larger relation *r* is redistributed across the *n* processors using the hash function $h_{1.}$ Let r_i denote the tuples of relation *r* that are sent to processor P_i.
- 3. b. As the *r* tuples are received at the destination processors, they are repartitioned using the function h_2 .
- 4. Each processor P_i executes the build and probe phases of the hash-join algorithm on the local partitions r_i and s_i 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.

Parallel Nested-Loop Join

- Assume that
    - relation *s* is much smaller than relation *r* and that *r* is stored by partitioning (partitioning attribute does not matter)
    - there is an index on a join attribute of relation *r* at each of the partitions of relation *r*.
- Use asymmetric fragment-and-replicate, with relation *s* being replicated, and using the existing partitioning of relation *r*.
- Each processor P_j where a partition of relation *s* is stored reads the tuples of relation *s* stored in D_j, and replicates the tuples to every other processor P_i.
    - At the end of this phase, relation *s* is replicated at all sites that store tuples of relation *r*.
- Each processor P_i performs an indexed nested-loop join of relation *s* with the ith partition of relation *r*.

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 records (using either range- or hash- partitioning) and perform duplicate elimination locally at each processor.

- **Projection**
  - Projection without duplicate elimination can be performed as records are read in from disk in parallel.
  - If duplicate elimination is required, any of the above duplicate elimination techniques can be used.

Grouping/Aggregation

- Partition the relation (hash or range) on the grouping attributes and then compute the aggregate values locally at each processor.
- Can reduce cost of transferring records during partitioning by partly computing aggregate values before partitioning.
- Consider the **sum** aggregation operation:
  - Perform aggregation operation at each processor P_i on those records stored on disk D_i
    - results in records with partial sums at each processor.
  - Result of the local aggregation is partitioned on the grouping attributes, and the aggregation performed again at each processor P_i to get the final result.
- Fewer records need to be sent to other processors during partitioning.
- This idea can be extended easily to the **min**, **max, count and average** aggregate functions.

Cost of Parallel Evaluation of Operations

- Parallelism has been achieved by partitioning the I/O among multiple disks, and partitioning the CPU work among multiple processors.
- If there is no split (partitioning) overhead, no skew in the partitioning, and there is no overhead due to the parallel evaluation, the time cost of the parallel processing would then be 1/n of the time cost of sequential processing of the operation.
- We must account for the following costs:
  - **Partitioning cost** for partitioning the relation using any technique
  - **Start-up** costs for initiating the operation at multiple processors
  - **Skew** in the distribution of work among the processors, with some processors getting a larger no. of records than others.
  - **Contention for resources** – such as memory, disk and the communication network – resulting in delays.
  - **Cost of assembling** – the final result by transmitting partial results form each processor

Cost of Parallel Evaluation of Operations

- 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.
- The above is an optimistic estimation. In reality, skew is a great problem. Overflow resolution and avoidance techniques developed for hash joins to handle skew when hash partitioning is used. We can use balanced range partitioning and virtual processor partitioning to minimize skew due to range partitioning.

Interoperation Parallelism

- There are two forms of interoperation parallelism: pipelined parallelism and independent 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 records it computes 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.
  - First, pipeline chains do not attain sufficient length to provide a high degree of parallelism.
  - Second, it is not possible to pipeline relational operators that do not produce output until all inputs have been accessed (e.g. aggregate, sort, set-difference)
  - Third, only marginal speedup is obtained for the frequent cases of skew in which one operator's execution cost is much higher than the others.
- Considering all things, when the degree of parallelism is high, pipelining is less important source of parallelism than partitioning. Only one real reason of using it that it can avoid writing intermediate results to disks.

Independent Parallelism

- **Independent parallelism:** Operations in a query expression that do not depend on one another can be executed in parallel.
  - 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

Query Optimization (Cont.)

- The number of parallel evaluation plans from which to choose from is much larger than the number of sequential evaluation plans.
  - Therefore heuristics are needed while optimization
- Two alternative heuristics for choosing parallel plans:
  - No pipelining and inter-operation pipelining; just parallelize every operation across all processors.
    - Finding best plan is now much easier --- use standard optimization technique, but with new cost model
    - Volcano parallel database popularize the **exchange-operator** model
      - exchange operator is introduced into query plans to partition and distribute tuples
      - each operation works independently on local data on each processor, in parallel with other copies of the operation
  - First choose most efficient sequential plan and then choose how best to parallelize the operations in that plan.
    - Can explore pipelined parallelism as an option
- Choosing a good physical organization (partitioning technique) is important to speed up queries.

Design of Parallel Systems

- So far it has been concentrated on parallelization of data storage and query processing. Since large-scale parallel database systems are used primarily for storing large volume of data and processing decision support queries on those data, the following topics are most important in a parallel database system:
  - Parallel loading of data from external sources is needed in order to handle large volumes of incoming data.
  - A large parallel database system must also address these availability issues:
    - Resilience (flexibility) to failure of some processors or disks.
    - Online reorganization of data and schema changes.

Design of Parallel Systems (Cont.)

- With a large no. of processor and disks, the probability that at least one processor or disk will malfunction is significantly greater than in a single processor system with one disk.
- A poorly designed parallel system will stop functioning if any component (processor or disk) fails.
- Assuming that the probability of failure of a single processor or disk is small, the probability of failure of the system goes up linearly with number of processors and disks.
- If a single processor or disk would fail once every 5 years, a system with 100 processors would have a failure every 18 days (5*365/100).
- Therefore, large-scale parallel database systems, such as Compac Himalaya, Teradata and Informix XPS, are designed to operate even if a processor or disk fails.

Design of Parallel Systems (Cont.)

- Resilience to failure of some processors or disks.
  - Probability of some disk or processor failing is higher in a parallel system.
  - Operation (perhaps with degraded performance) should be possible in spite of failure.
  - Redundancy achieved by storing extra copy of every data item at another processor.
- On-line reorganization of data and schema changes must be supported.
  - For example, index construction on terabyte databases can take hours or days even on a parallel system.
    - Need to allow other processing (insertions/deletions/updates) to be performed on relation even as index is being constructed.
  - Basic idea: index construction tracks changes and ``catches up'' on changes at the end.
- Also need support for on-line repartitioning and schema changes (executed concurrently with other processing).

End of Chapter