

# Non-Standard Database Systems

## Parallel Databases

Nikolaus Augsten

nikolaus.augsten@plus.ac.at  
Department of Computer Science  
University of Salzburg



<https://dbresearch.uni-salzburg.at>

Sommersemester 2024

Version 19. März 2024

## Introduction

- **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**
  - large volumes of transaction data are collected and stored for later analysis
  - large objects like multimedia data 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

Databases naturally lend themselves to parallelism:

- **Parallel I/O**: data can be partitioned across multiple disks.
- **Parallel execution**: execute individual relational operations in parallel
  - e.g., sort, join, aggregation can be executed in parallel
  - each processor can work independently on its own data partition
- Queries are expressed at the logical level and in a **high level language**:
  - SQL is declarative and is translated to relational algebra
  - separation of logical and physical level makes parallelization easier
- Different queries can run in parallel:
  - **concurrency control** takes care of conflicts

## Outline

- 1 I/O Parallelism
- 2 Interquery Parallelism
- 3 Intraquery Parallelism
  - Intraoperation Parallelism
  - Interoperation Parallelism
- 4 Query Optimization and System Design

## Outline

- 1 I/O Parallelism
- 2 Interquery Parallelism
- 3 Intraquery Parallelism
  - Intraoperation Parallelism
  - Interoperation Parallelism
- 4 Query Optimization and System Design

## 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.

## Horizontal Partitioning

Let  $n$  be the number of disks.

- **Round-robin:**
  - send the  $i$ -th tuple inserted in the relation to disk  $i \bmod n$ .
- **Hash partitioning:**
  - choose one or more **attributes  $A$**  as the partitioning attributes
  - choose **hash function  $h$**  with range  $0 \dots n - 1$
  - send tuple  $t$  with hash value  $i = h(t[A])$  to disk  $i$
- **Range partitioning:**
  - choose one or more **attributes  $A$**  as the partitioning attributes
  - choose a **partitioning vector**  $[v_0, v_1, \dots, v_{n-2}]$ 
    - tuples  $t$  with  $t[A] < v_0$  got to disk 0
    - tuples with  $v_i \leq t[A] < v_{i+1}$  to to disk  $i + 1$
    - tuples with  $v_{n-2} \leq t[A]$  go to disk  $n - 1$
  - **Example:** with partitioning vector  $[5, 11]$  on attribute  $A$ , a tuple  $t$  with partitioning attribute value of  $t[A] = 2$  will go to disk 0, a tuple with  $t[A] = 8$  will go to disk 1, while a tuple with  $t[A] = 20$  will go to disk 2.

## Comparison of Partitioning Techniques/1

- We distinguish three different types of data access:
  1. **sequential scan:** scan the entire relation
  2. **point query:** locate a specific tuple
    - predicate is equality, zero or one result tuple
    - e.g., tuple of relation  $r$  with  $r.A = 25$  ( $A$  is a key)
    - **multi point query:** zero or more result tuples ( $A$  is not a key)
  3. **range query:** locate all tuples within a specified value range
    - e.g., all tuples of relation  $r$  with  $10 \leq r.A < 25$ .

## Comparison of Partitioning Techniques/2

### Round robin:

- Good for **sequential scan**:
  - all disks have almost an equal number of tuples
  - retrieval work is thus well **balanced between disks**
- **Point queries and range queries** are difficult to process
  - **no clustering** — relevant tuples are scattered across all disks

## Comparison of Partitioning Techniques/3

### 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
  - lookup single disk, leaving others available for answering other queries
- **No clustering**, so difficult to answer range queries

## Comparison of Partitioning Techniques/4

### Range partitioning:

- Provides **data clustering** by partitioning attribute value.
- Good for **sequential access**.
- Good for **point queries**:
  - lookup single disk, leaving others available for answering other queries
- Good for **range queries** on partitioning attribute:
  - lookup single or few disks
  - good if result tuples are from one to a few blocks of a disk
- **Execution skew**: affects range queries and multi point queries
  - if many blocks are to be fetched, they may still be fetched from one to a few disks: potential parallelism in disk access is wasted
  - e.g., partition by order date, then tuples with recent order dates will be accessed more frequently

## Partitioning a Relation across Disks

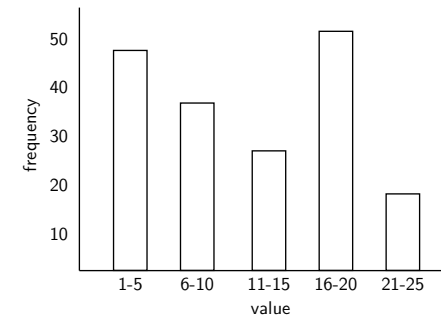
- If a relation contains only a **few tuples** which will **fit** into a **single disk block**, then assign the relation to a single disk.
- **Large relations** are preferably partitioned across all the available disks.
- If a relation consists of  $m$  disk blocks and there are  $n$  disks available,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

## Handling of Data Skew

- Distribution of tuples to disks may be **skewed**: some disks have many tuples, while others have fewer tuples.
- **Skew limits speedup**. Example:
  - relation with 1000 tuples is partitioned to 100 disks (10 tuples/disk)
  - expected speedup for scan:  $\times 100$
  - skew: one disk has 40 tuples  $\Rightarrow$  max. speedup is  $\times 25$
- Types of data skew:
  - **Attribute-value skew**:
    - Some 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 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  $10\times$  the number of processors)
  - **assign virtual processors** to partitions either in round-robin fashion or based on estimated cost of processing each virtual partition
- Basic idea:
  - If any normal partition would have been skewed, it is very likely the skew is spread over a number of virtual partitions.
  - Skewed virtual partitions get spread across a number of processors, so work gets distributed evenly.

## Outline

- 1 I/O Parallelism
- 2 Interquery Parallelism
- 3 Intraquery Parallelism
  - Intraoperation Parallelism
  - Interoperation Parallelism
- 4 Query Optimization and System Design

## Interquery Parallelism

- Queries/transactions **execute in parallel** 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 on **shared-disk** or **shared-nothing** architectures:
  - locking and logging: coordinate 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.

## Outline

- 1 I/O Parallelism
- 2 Interquery Parallelism
- 3 **Intraquery Parallelism**
  - Intraoperation Parallelism
  - Interoperation Parallelism
- 4 Query Optimization and System Design

## 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.
- **Intraoperation parallelism 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
  - shared-nothing architecture
  - $n$  processors,  $P_0, \dots, P_{n-1}$ , and  $n$  disks  $D_0, \dots, D_{n-1}$ , where disk  $D_i$  is associated with processor  $P_i$ .
- If processor has multiple disks: 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/1

### Range-Partitioning Sort

- Choose processors  $P_0, \dots, P_{m-1}$ , where  $m \leq n$  to do sorting.
- Create **range-partition vector** with  $m$  ranges, on the sorting attributes
- Redistribute the relation using range partitioning
  - all tuples that lie in the  $i^{\text{th}}$  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  $0 \leq i < j < m$ , the key values in processor  $P_i$  are all less than the key values in  $P_j$ .

## Parallel Sort/2

### Parallel External Sort-Merge

- Assume the relation has already been **partitioned** among disks  $D_0, \dots, D_{n-1}$  (in whatever manner).
- Each processor  $P_i$  **locally sorts** the data on disk  $D_i$ .
- Sorted runs of processors are **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  $P_0, \dots, P_{m-1}$ .
  - 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  $P_0, \dots, P_{m-1}$  are concatenated to get the final result.

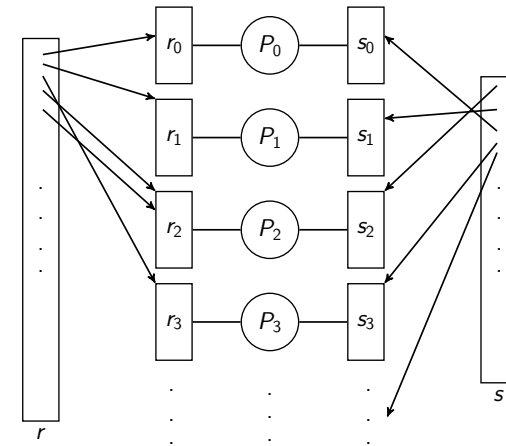
## Parallel Join

- 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/1

- 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  $r_0, r_1, \dots, r_{n-1}$  and  $s_0, s_1, \dots, s_{n-1}$ .
- 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  $r_i \bowtie_{r_i.A=s_i.B} s_i$ . Any of the standard join methods can be used.

## Partitioned Join/2



## Partitioned Parallel Hash-Join/1

## Parallelizing partitioned hash join:

- Assume  $s$  is smaller than  $r$ , then  $s$  is chosen as the **build relation**.
- A **hash function**  $h_1$  takes the join attribute value  $x$  of each tuple in  $s$  and maps this tuple to one of the  $n$  processors.
- All **tuples are sent** to the appropriate processors: a tuple with hash value  $h(x) = i$  is sent to processor  $P_i$ .
- 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  $P_i$ , they are partitioned further using another hash function,  $h_2$ , which is used to compute the hash-join locally.

## Partitioned Parallel Hash-Join/2

- Once the tuples of  $s$  have been distributed, **probe relation**  $r$  is redistributed across the  $n$  processors using hash function  $h_1$ .
- Let  $r_i$  denote the tuples of relation  $r$  that are sent to processor  $P_i$ .
- As tuples of relation  $r$  are received at the destination processors  $P_i$ , they are partitioned on  $P_i$  using hash 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  $s_i$  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 to disk and reading them back in.

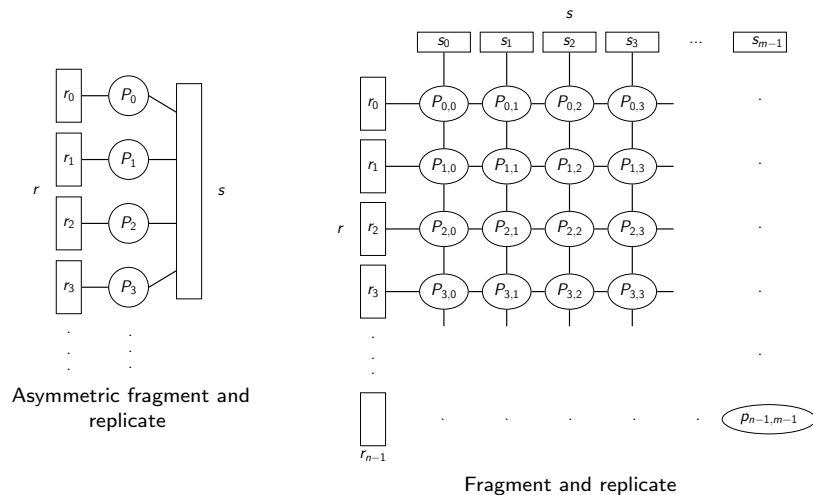
## Fragment-and-Replicate Join/1

- Partitioning not possible for some join conditions
  - E.g., non-equi-join conditions, such as  $r.A > s.B$ .
- For joins where partitioning is not applicable, parallelization can be accomplished by **fragment and replicate** technique
- Special case – **asymmetric fragment-and-replicate**:
  - One of the relations, say  $r$ , is partitioned; any partitioning technique can be used.
  - The other relation,  $s$ , 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.

## Parallel Nested-Loop Join

- Assume that
  - relation  $s$  is much smaller than relation  $r$
  - $r$  is stored by partitioning (partitioning technique irrelevant)
  - 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  $r$  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  $i^{th}$  partition of relation  $r$ .

## Fragment-and-Replicate Join/2



## Fragment-and-Replicate Join/3

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



## Fragment-and-Replicate Join/4

- 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.

## Other Relational Operations/1

### Selection $\sigma_{\theta}(r)$

- If  $\theta$  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  $\theta$  is of the form  $l \leq a_i \leq u$  (i.e.,  $\theta$  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/2

- **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.

## Grouping/Aggregation

- **Partition** the relation **on the grouping attributes** and then compute the aggregate values locally at each processor.
- Can reduce cost of transferring tuples during partitioning by partly computing aggregate values before partitioning.
- Consider the **sum aggregation operation**:
  - Perform aggregation operation at each processor  $P_i$  on those tuples stored on disk  $D_i$ 
    - results in tuples 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 tuples need to be sent to other processors during partitioning.

## 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 **speedup** will be  $n$
- If **skew and overheads** are also to be taken into account, the time taken by a parallel operation can be estimated as

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

- $T_{part}$  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

- **Two types** of interoperation parallelism:
  - pipelined parallelism
  - independent parallelism

## Pipelined Parallelism

- **Example:** 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  $P_1$  be assigned the computation of  $temp_1 = r_1 \bowtie r_2$
  - And  $P_2$  be assigned the computation of  $temp_2 = temp_1 \bowtie r_3$
  - And  $P_3$  be assigned the computation of  $temp_2 \bowtie r_4$
- Each operation can **execute in parallel** sending result tuples to the next operation even while it is computing further results
- Requires pipelineable (**non-blocking**) join evaluation algorithm (e.g., indexed nested loops join)

## 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 **execution skew** in which one operator's execution cost is much higher than the others.
- **Advantage:** avoids writing intermediate results to disk

## Independent Parallelism

- **Example:** Consider a join of four relations  
 $r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$
- **Independent parallelism:**
  - Let  $P_1$  be assigned the computation of  $temp_1 = r_1 \bowtie r_2$
  - And  $P_2$  be assigned the computation of  $temp_2 = r_3 \bowtie r_4$
  - And  $P_3$  be assigned the computation of  $temp_1 \bowtie temp_2$
  - $P_1$  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.

## Outline

- 1 I/O Parallelism
- 2 Interquery Parallelism
- 3 Intraquery Parallelism
  - Intraoperation Parallelism
  - Interoperation Parallelism
- 4 Query Optimization and System Design

## Query Optimization/1

- 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/2

- **Use heuristics:** Number of parallel evaluation plans much larger than number of sequential evaluation plans.
- **Heuristic 1: No pipelining**, only intra-operation parallelism:
  - Parallelize every operation on all processors
  - Use standard optimization technique, but with new cost model
- **Heuristic 2:** First choose most efficient **sequential plan** and then choose how best to parallelize the operations in that plan.
  - Volcano parallel database popularized 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
- Choosing a **good physical storage organization** (partitioning technique) is important to speed up queries.

## Design of Parallel Systems/1

Some issues in the design of parallel systems:

- **Parallel loading** of data from external sources is needed in order to handle large volumes of incoming data.
- **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.

## Design of Parallel Systems/2

- **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).

## Examples of Parallel Database Systems

- Teradata (1979), appliance, still large market share
- IBM Netezza (1999), appliance
- Microsoft DATAlegro / Parallel Data Warehouse (2003), appliance
- Greenplum (2005), Pivotal, open source
- Vertica Analytic Database (2005) commodity hardware
- Oracle Exadata (2008), appliance
- SAP Hana (2010), main memory, appliance