Chapter 18: Parallel Databases

Database System Concepts, 6th Ed.

Chapter 18: Parallel Databases

Introduction

- Parallel machines are becoming quite common and affordable
  - Prices of microprocessors, memory and disks have dropped sharply
- 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
- Different queries can be run in parallel with each other (Inter-Query Parallelism)
  - Concurrency control takes care of conflicts
- Queries are expressed in high level language SQL, then translated to relational algebra
  - Individual relational operations (e.g., sort, join, aggregation) can be executed in parallel (Intra-Query Parallelism)
  - data can be partitioned and each processor can work independently on its own partition.
- Thus, databases naturally lend themselves to parallelism
  - Potential parallelism is everywhere in database processing

Chapter 18: Parallel Databases

- 18.1 Introduction
- 18.2 I/O Parallelism
- 18.3 Interquery Parallelism
- 18.4 Intraquery Parallelism
- 18.5 Intraoperation Parallelism
- 18.6 Interoperation Parallelism
- 18.7 Query Optimization
- 18.8 Design of Parallel Systems
- 18.9 Parallelism on Multicore Processors

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. (number of disks = *n*):

  * Round-robin partitioning: Send the i^{th} tuple inserted in the relation to disk i mod n

  * 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 to the partitioning attribute value of a tuple
      - Send the tuple to disk *i*

  * Range partitioning:

    - Choose an attribute v as the partitioning attribute
    - A partitioning vector $[v_0, v_1, ..., v_{n-2}]$ is chosen
      - Tuples such that $v_i \leq v \leq v_{i+1}$ go to disk $i + 1$
      - Tuples with $v < v_0$ go to disk 0
      - Tuples with $v \geq v_{n-2}$ go to disk n-1

  E.g., with a partitioning vector [5,11] and 3 disks, a tuple with value 2 goes to disk 0, a tuple with value 8 goes to disk 1, while a tuple with value 20 goes to disk2.

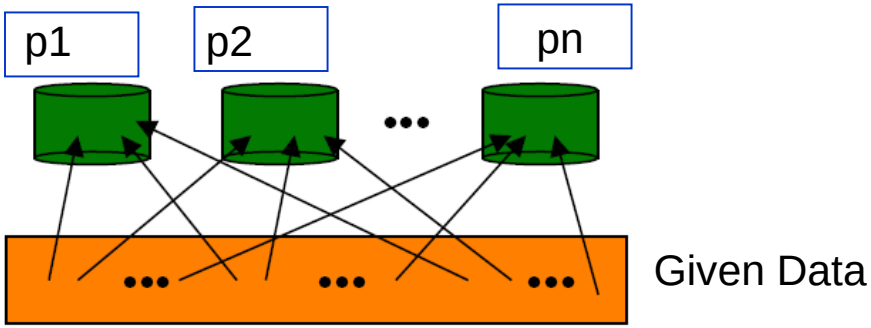
Comparison of Partitioning Techniques [1/3]

- Evaluate how well partitioning techniques support the following types of data access in a parallel fashion:

1. Scanning the entire relation – scan queries
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., $10 \leq r.A < 25$)

* Round robin partitioning:

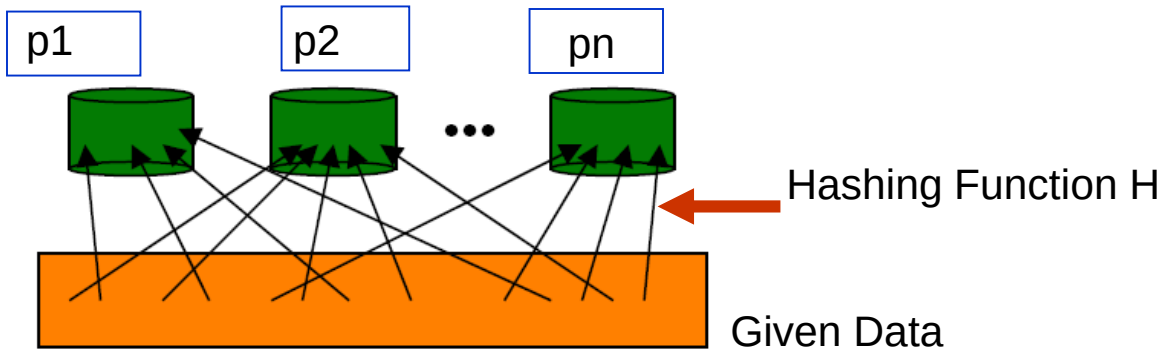
- Best suited for sequential scan of entire relation on each query
  - All disks have almost an equal number of tuples
  - Retrieval work for entire relation is thus well balanced between disks
- Point queries and Range queries are difficult to process
  - No clustering -- tuples are scattered across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



Comparison of Partitioning Techniques [2/3]

* Hash partitioning:

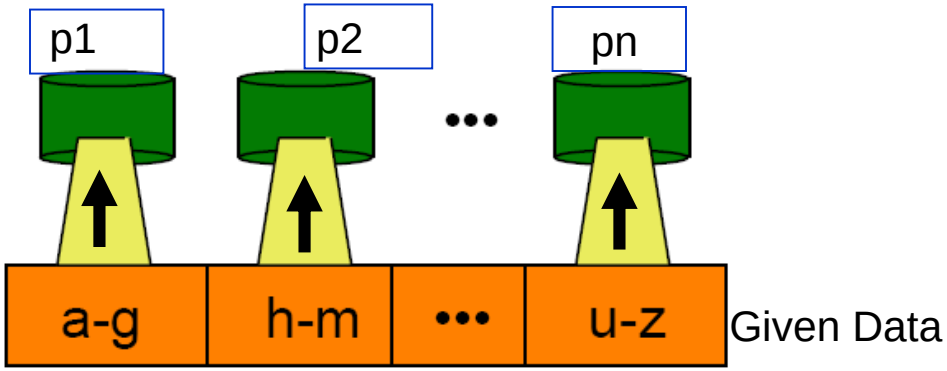
- Good for sequential access
  - Assuming hash function is good, and partitioning attributes form a key, tuples will be equally distributed between disks
  - Retrieval work for entire relation is then well balanced between disks
- Good for point queries on partitioning attribute
  - Can lookup single disk, leaving others available for answering other queries
  - Index on partitioning attribute can be local to disk, making lookup and update more efficient
- No clustering, so difficult to answer range queries



Comparison of Partitioning Techniques [3/3]

* Range partitioning: 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 tuples are from one to a few blocks
  - If many blocks are to be fetched and they are still fetched from one to a few disks, and potential parallelism in disk access is wasted
    - Example of **execution skew**
    - Round-robin or Hash partitioning might be better for this case



Handling of Skew Problem

- 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
- The distribution of tuples to disks may be **skewed**
  - Some disks have many tuples, while others may have fewer tuples
- **Types of skew:**
  - **Attribute-value skew**
    - 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

Partioning attribute

1
3
4
7
9
..
15

n = 3

balanced partitioning vector

1..7
9..11
12..15

- 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
  - Histograms can be stored in the system catalog

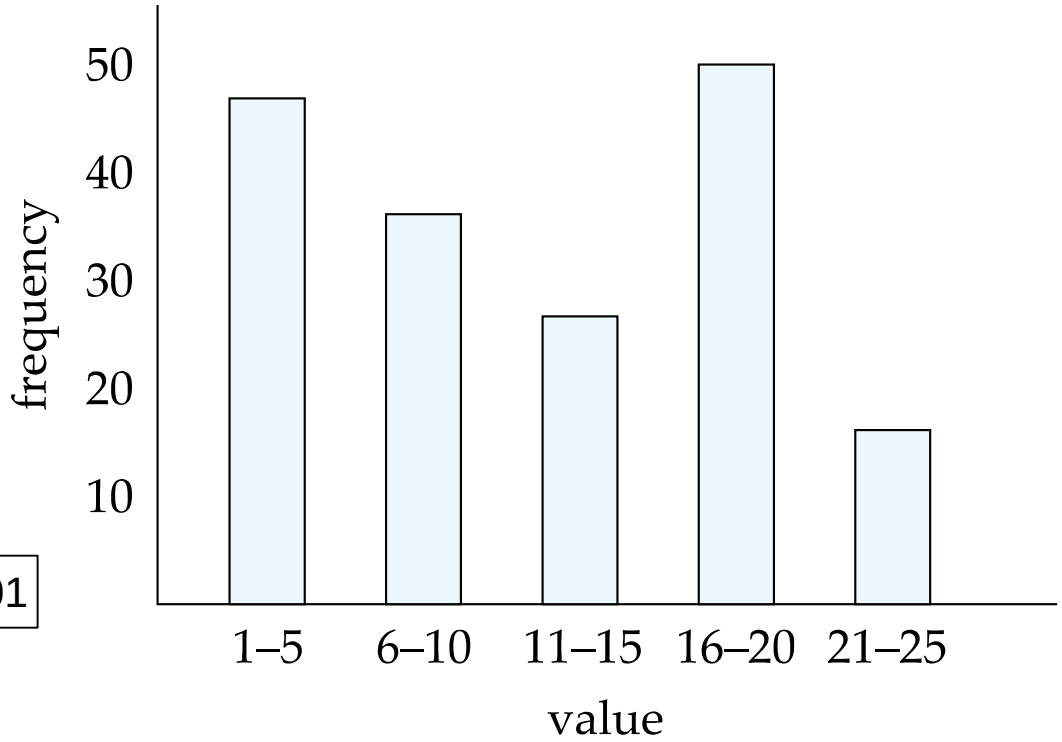


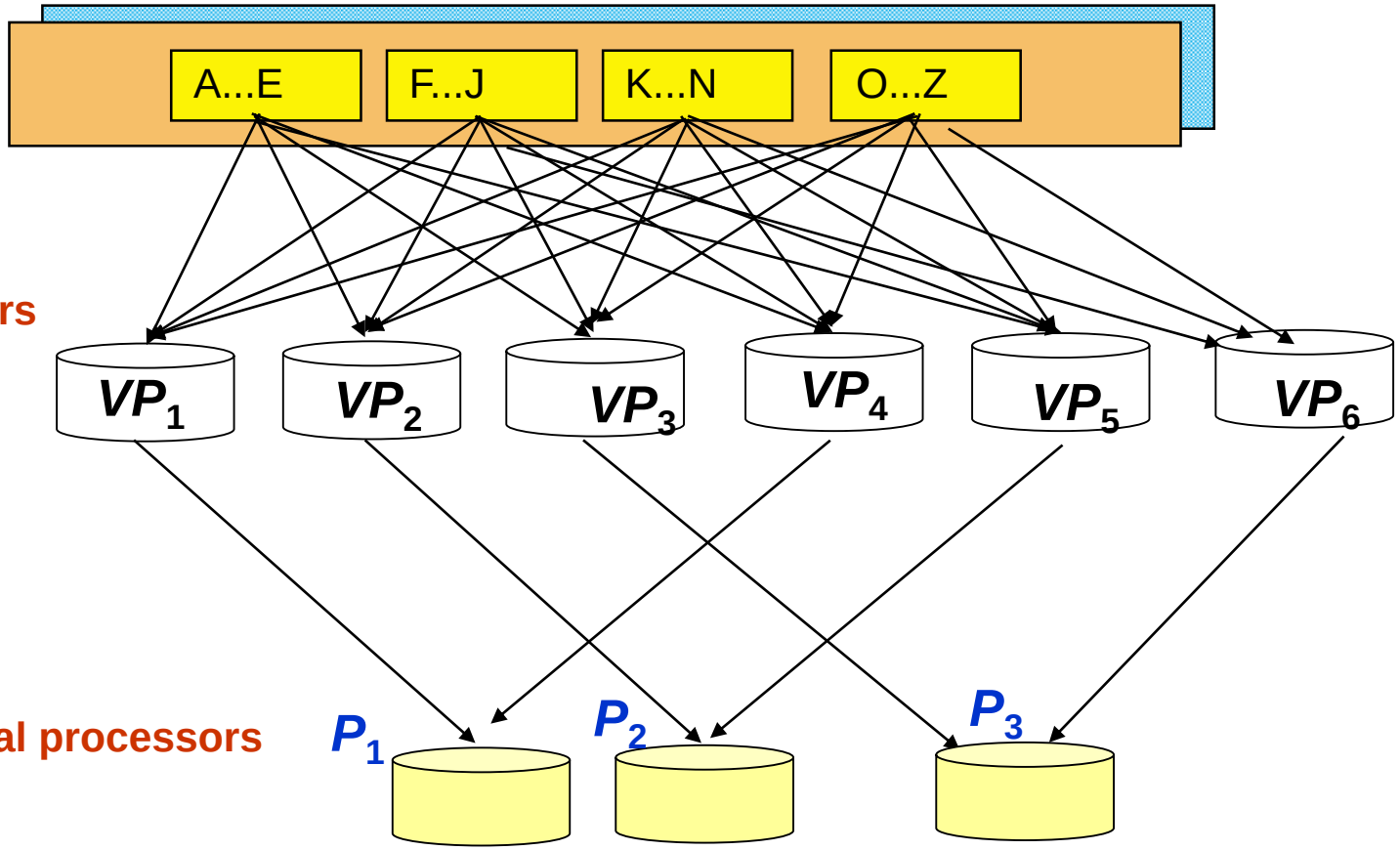
Fig 18.01

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)
  - 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!

Virtual Processor Partitioning

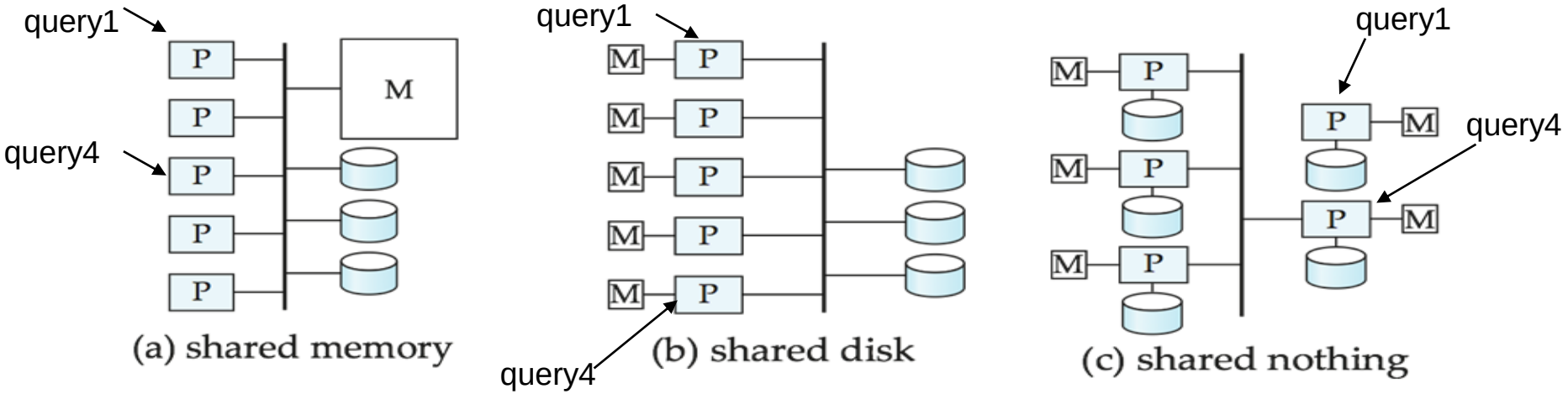
Given Data



Chapter 18: Parallel Databases

Interquery Parallelism [1/2]

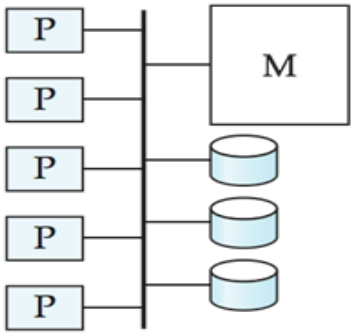
- Different queries / transactions execute in parallel with one another
- Increases transaction throughput
  - used primarily to scale up a transaction processing system to support a larger number of transactions per second
- Can use single-processor version of DBMS without drastic changes?
  - What about concurrency control
  - What about recovery
  - Many local memories may cause consistency problem



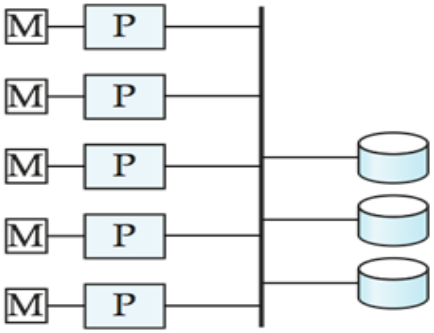
Interquery Parallelism [2/2]

- Shared-memory parallel database is easiest form of parallelism to support because even sequential database systems support concurrent processing
  - Single-processor version of DBMS can be used without drastic changes

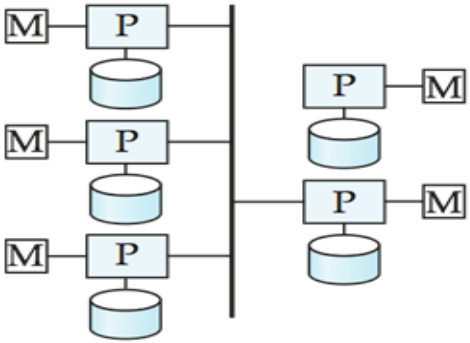
- More complicated to implement on shared-disk or shared-nothing architectures
  - Locking and logging must be coordinated by passing messages between processors
  - Data in a local buffer may have been updated at another processor
    - **Cache-coherency** has to be maintained — reads and writes of data in buffer must find latest version of data
  - Cache-coherency protocol may need to be combined with concurrency control



(a) shared memory



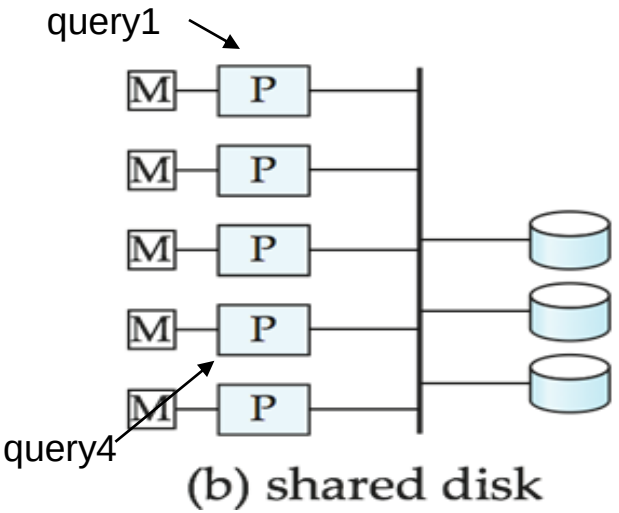
(b) shared disk

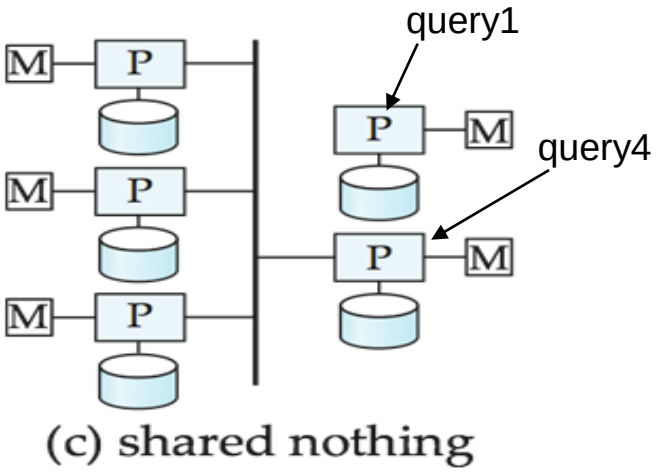


(c) shared nothing

Cache Coherency Protocol in Parallel Database

- 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





Chapter 18: Parallel Databases

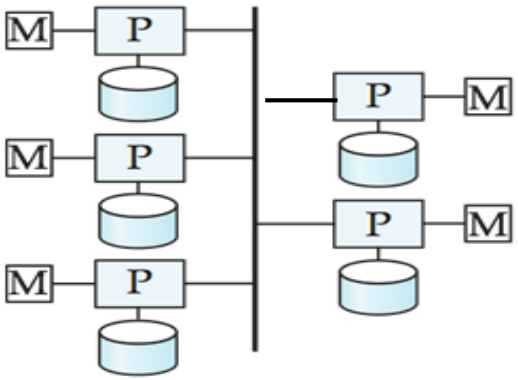
- 18.1 Introduction
- 18.2 I/O Parallelism
- 18.3 Interquery Parallelism
- 18.4 Intraquery Parallelism
- 18.5 Intraoperation Parallelism
- 18.6 Interoperation Parallelism
- 18.7 Query Optimization
- 18.8 Design of Parallel Systems
- 18.9 Parallelism on Multicore Processors

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 :
  - **18.5 Intra-operation Parallelism** – parallelize the execution of each individual operation in the query
    - This 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
  - **18.6 Inter-operation Parallelism** – execute the different operations in a query expression in parallel

Parallel Processing of Relational Operations

- 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 D_i ←→ 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



(c) shared nothing

Chapter 18: Parallel Databases

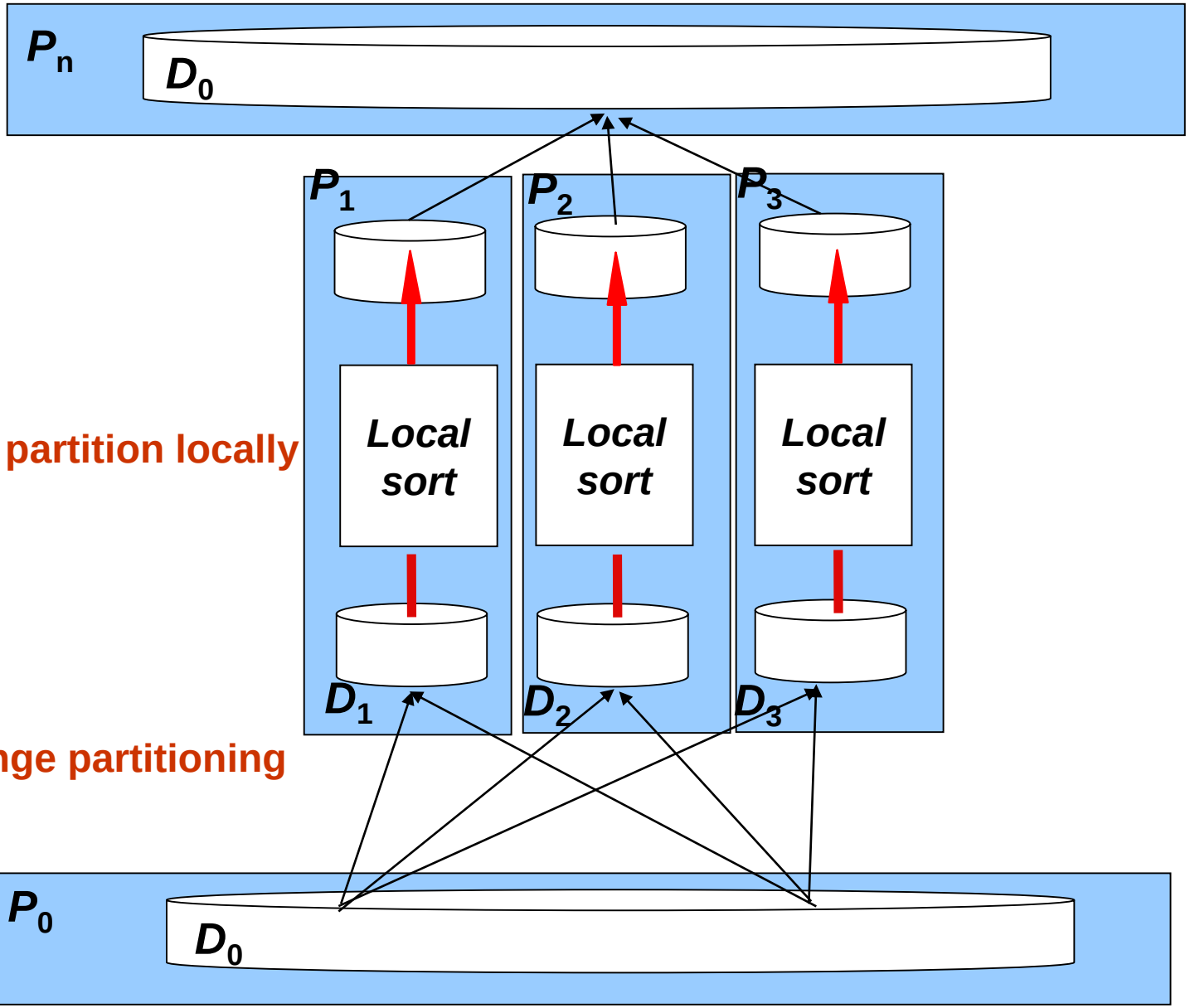
- 18.1 Introduction
- 18.2 I/O Parallelism
- 18.3 Interquery Parallelism
- 18.4 Intraquery Parallelism
- 18.5 Intraoperation Parallelism
- 18.6 Interoperation Parallelism
- 18.7 Query Optimization
- 18.8 Design of Parallel Systems
- 18.9 Parallelism on Multicore Processors

Parallel Sort [1/2]

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 $1 \leq i < j \leq m$, the key values in processor P_i. are all less than the key values in P_j

Range-Partitioning Sort



Sorts its partition locally

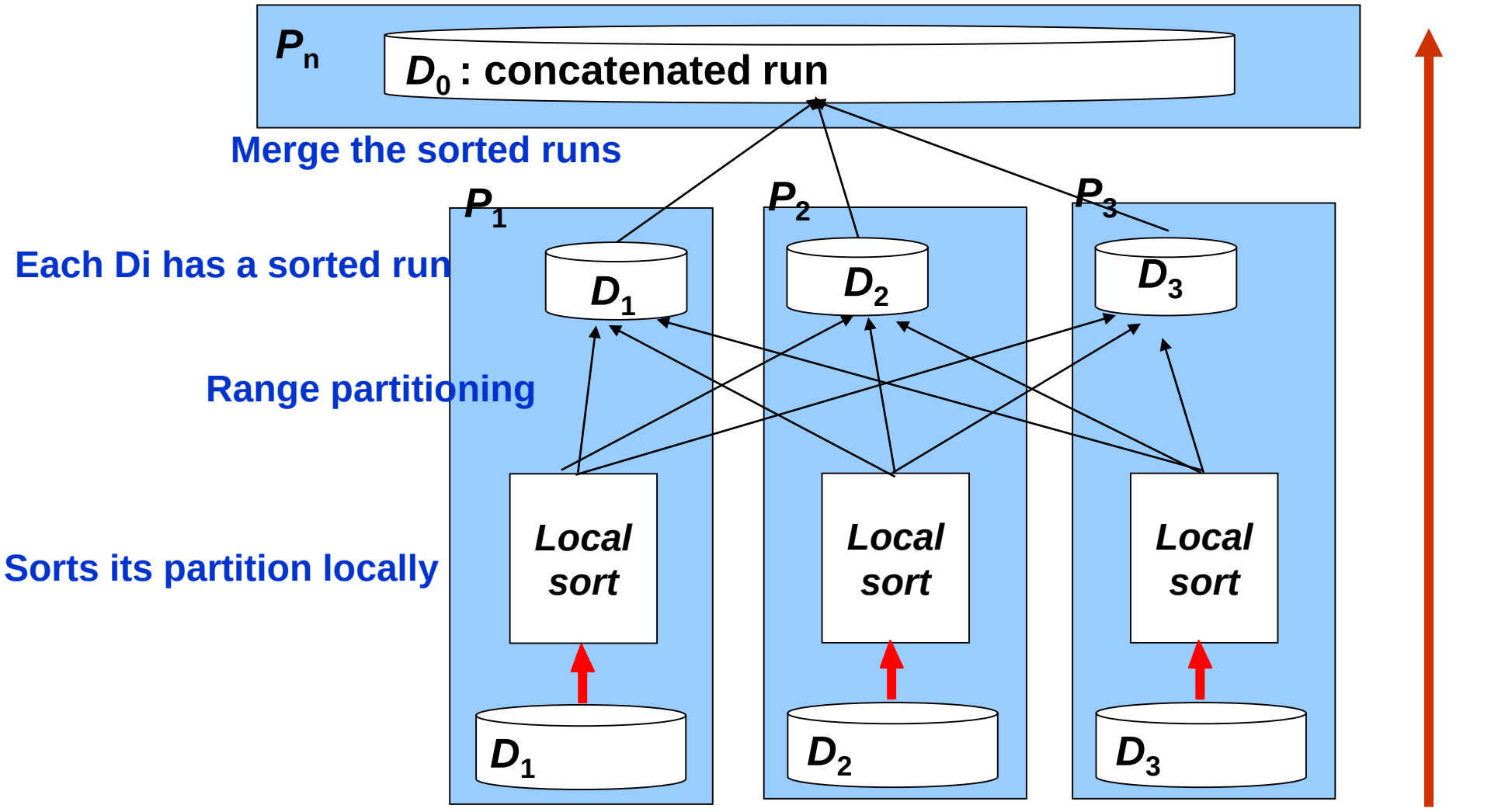
Range partitioning

Parallel Sort [2/2]

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 External Sort-Merge



※ Assume the relation has already been partitioned

Parallel Join Algorithms

- 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
  - 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
- Parallel Join algorithms
  - Partitioned Join
  - Fragment-and-Replicate Join
  - Partitioned Parallel Hash-Join
  - Parallel Nested-Loop Join

Partitioned Join [1/2]

- 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* are partitioned into *n* partitions, denoted $r_0, r_1, \ldots, r_{n-1}$ and $s_0, s_1, \ldots, s_{n-1}$
  - 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 [2/2]

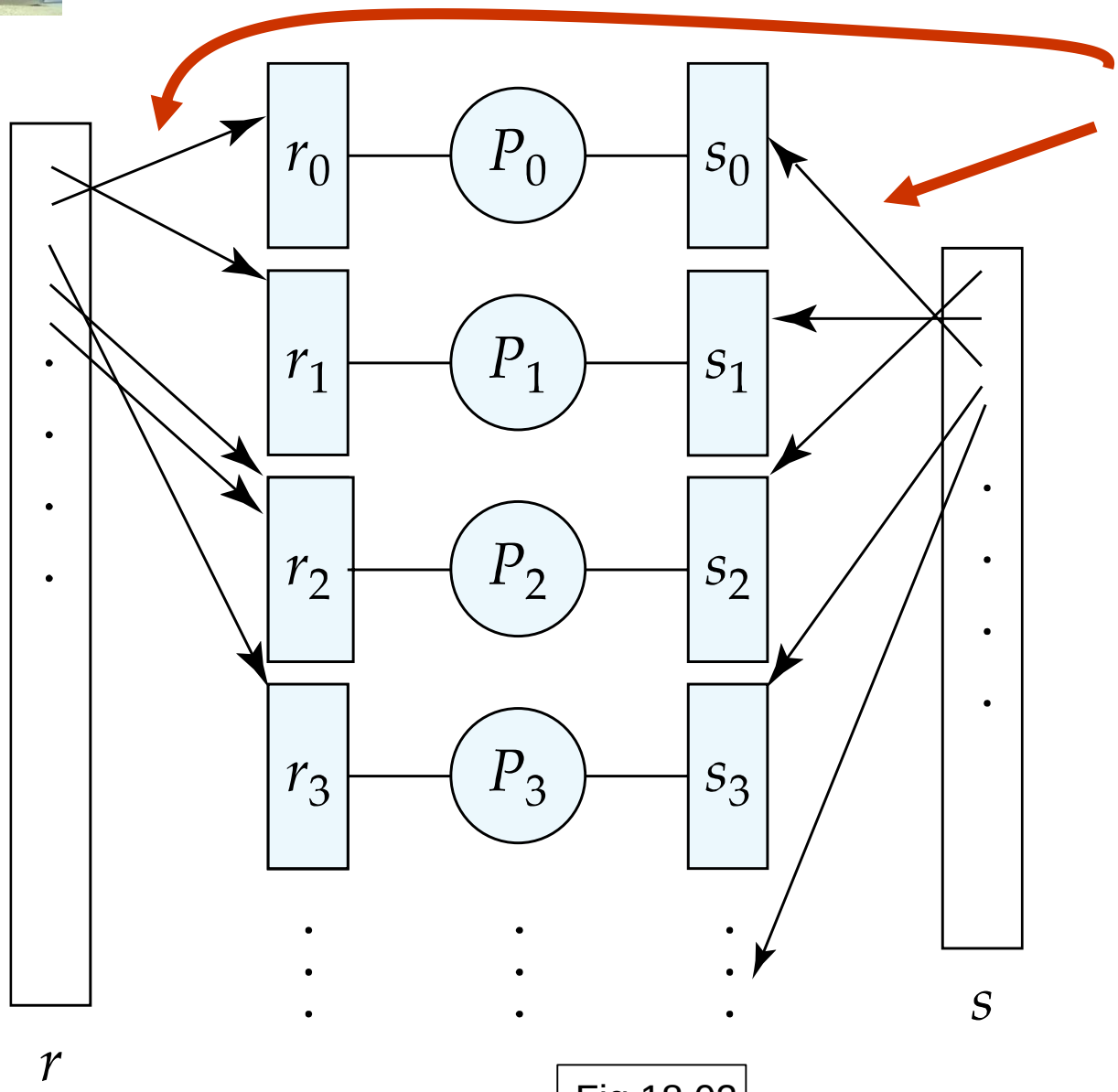


Fig 18.02

Range partitioning

or Hash partitioning

on join attributes

Fragment-and-Replicate Join [1/3]

- Partitioned join is 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
  - Depicted on next slide

- 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

Fragment-and-Replicate Join [2/3]

- 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 $P_{i,j}$

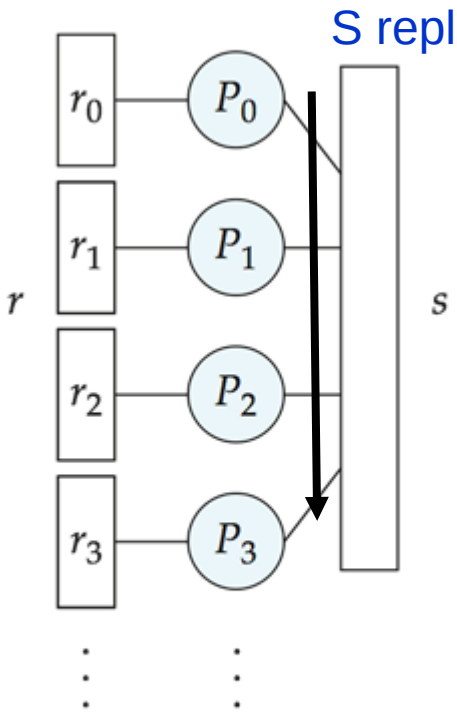
Fragment-and-Replicate Join [3/3]

- 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 partitioned join, 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., Suppose *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

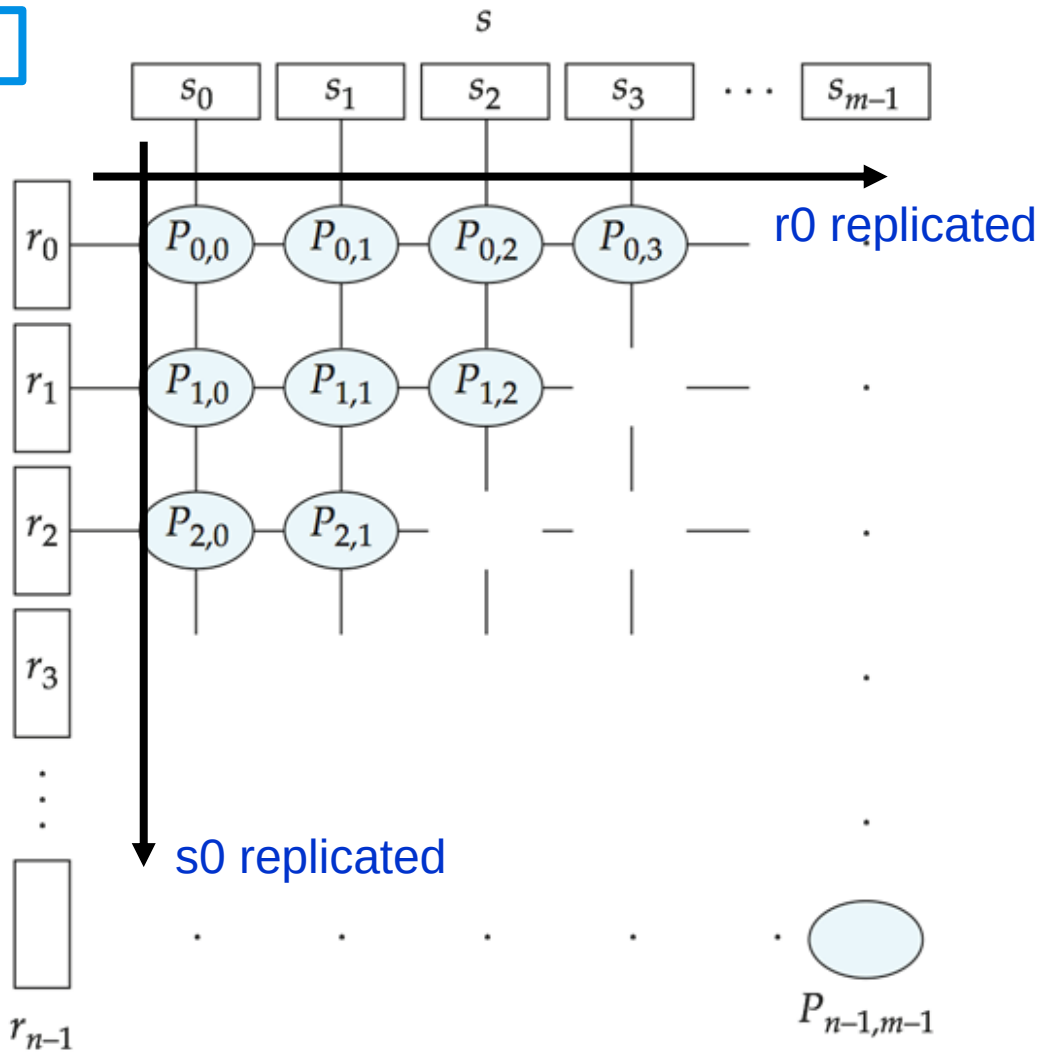
Depiction of Fragment-and-Replicate Joins

Fig 18.03

$r \bowtie_{r.A > s.B} s$



(a) Asymmetric fragment and replicate



(b) Fragment and replicate

When partitioned join is not possible!

Partitioned Parallel Hash-Join [1/4]

Parallelizing partitioned hash join:

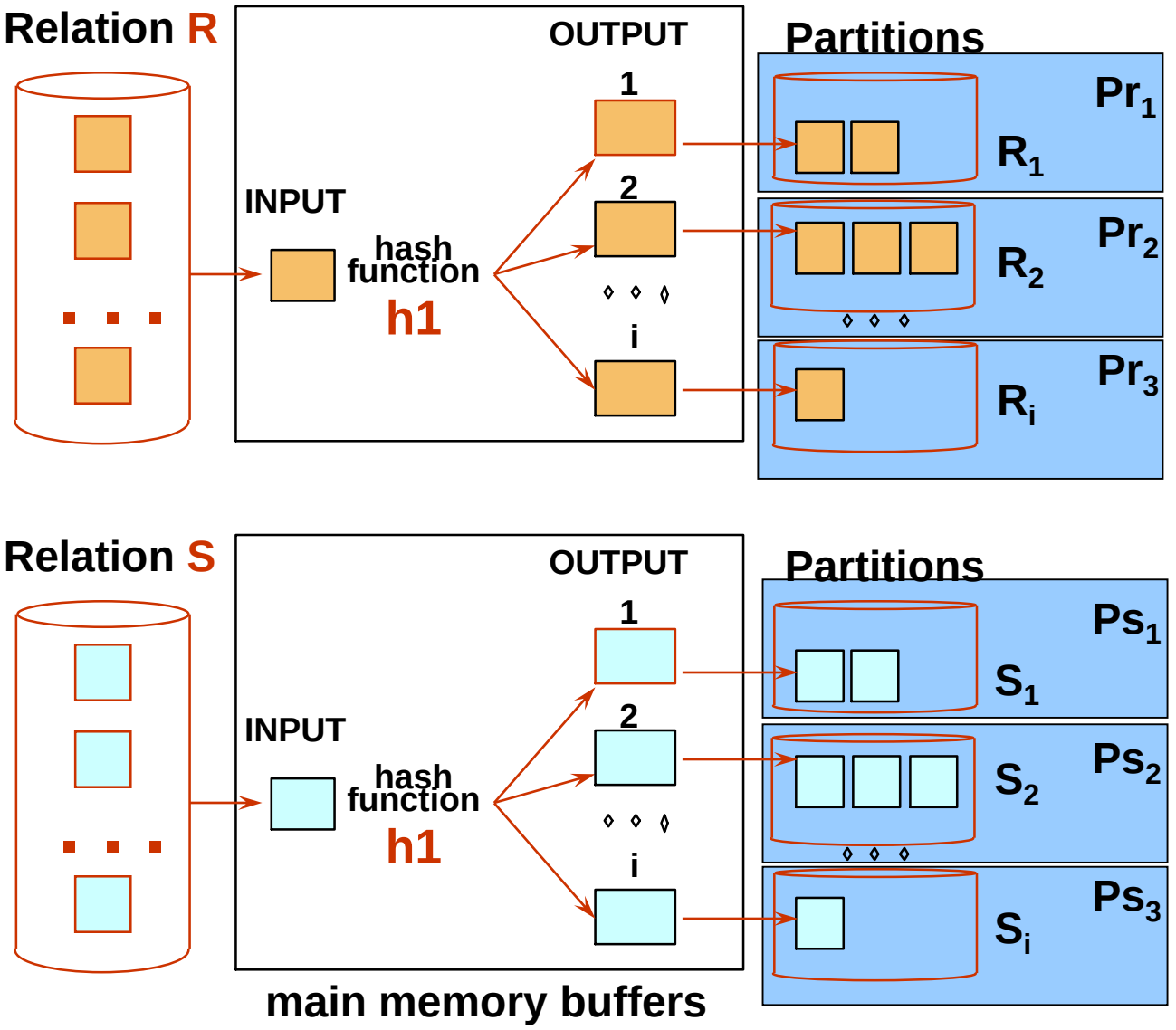
- Assume *s* is smaller than *r* and therefore *s* is chosen as the build relation.
- A hash function h_1 takes the join attribute value of each tuple in *s* and maps this tuple to one of the *n* processors
- Each processor P_i reads the tuples of *s* that are on its disk D_i, and sends each tuple to 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 [2/4]

- 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
  - (just as the probe relation is partitioned in the sequential hash-join algorithm).
- 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

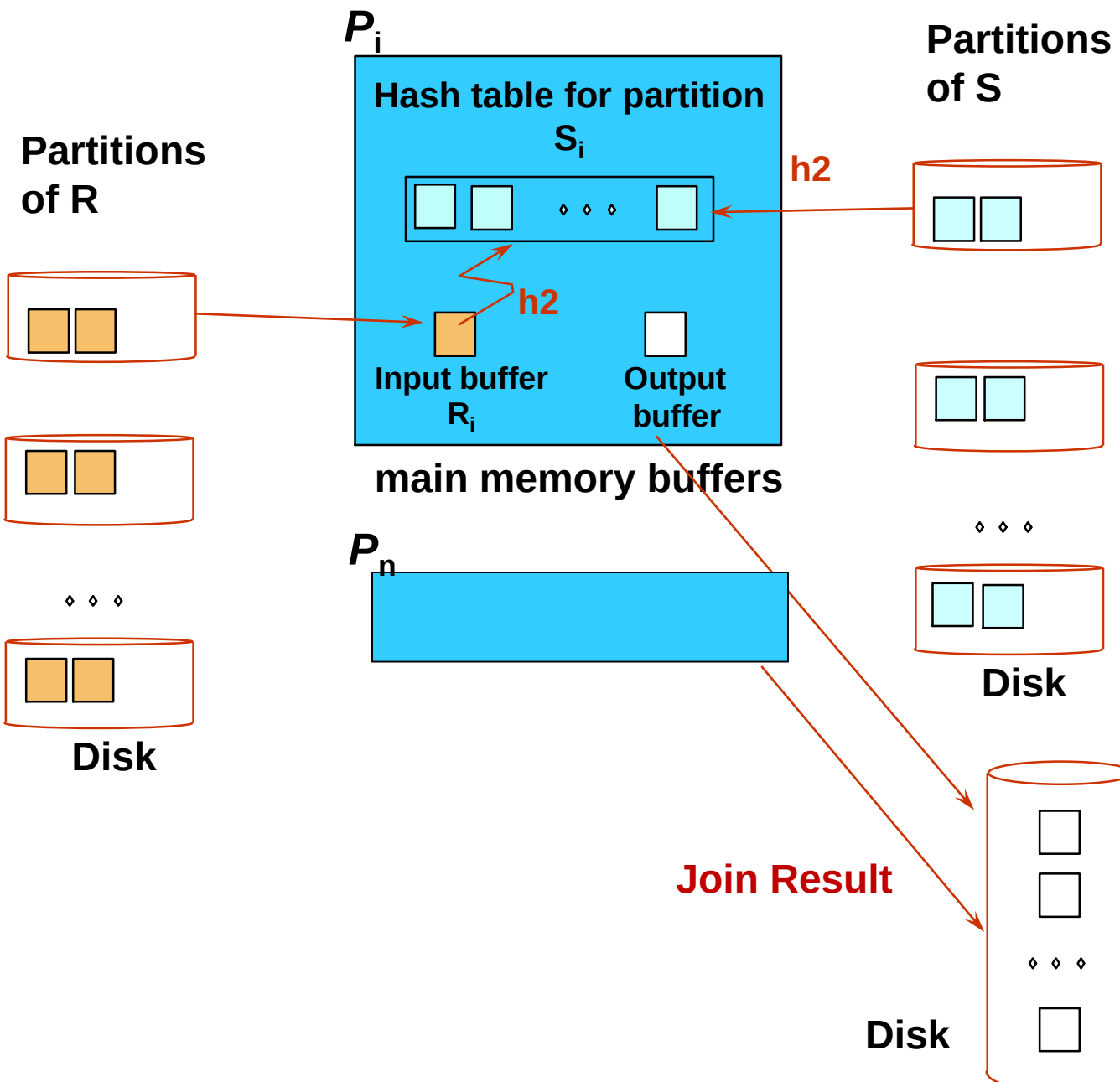
Partitioned Parallel Hash-Join [3/4]

Partition both relations using hash function h1



Partitioned Parallel Hash-Join [4/4]

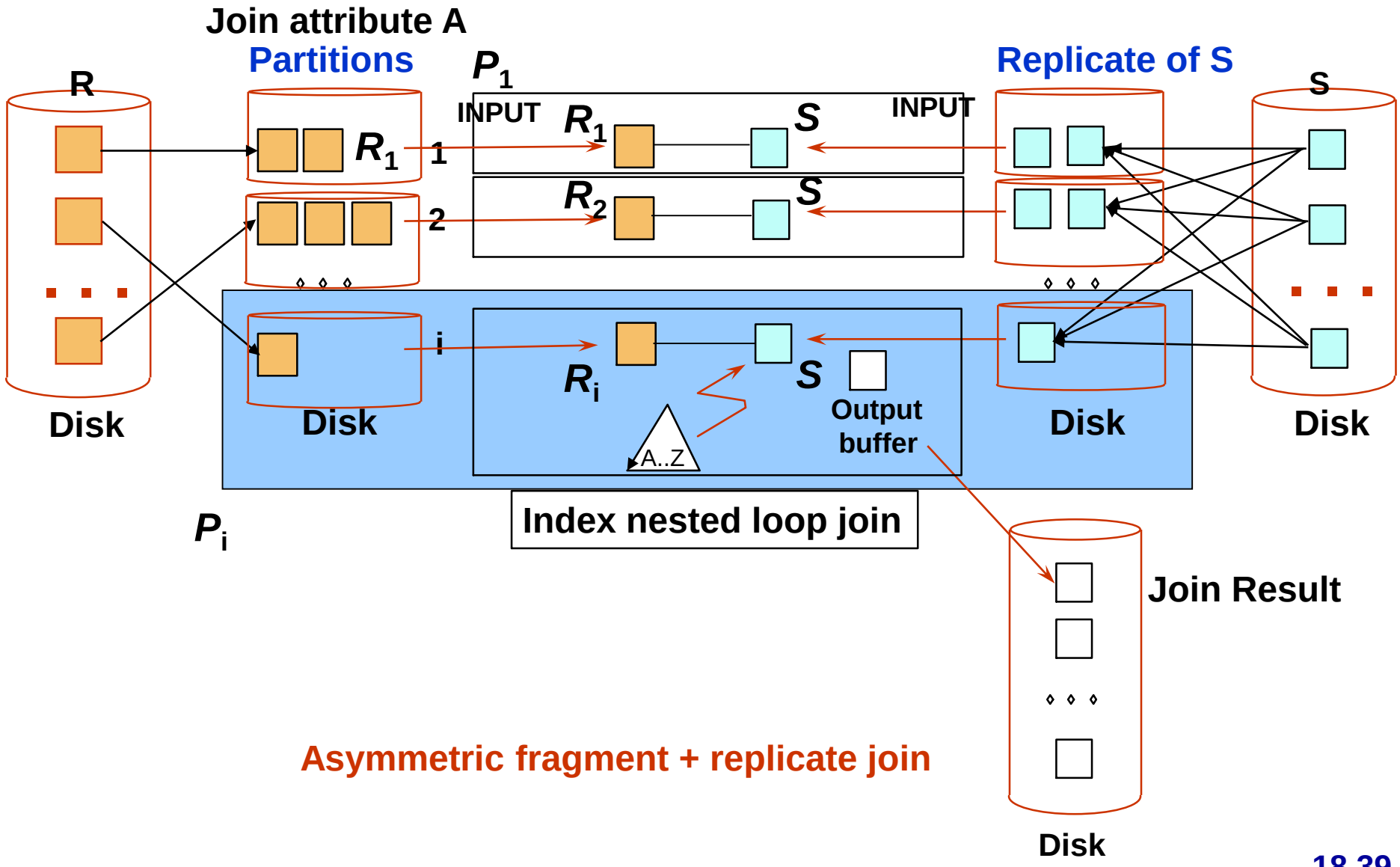
Read in a partition of R, hash it using h2



Parallel Nested-Loop Join [1/2]

- Assume that
  - Relation *s* is much smaller than relation *r* and that *r* is stored by partitioning
  - There is an index on a join attribute of relation *r* at each of the partitions of relation *r*
- Use asymmetric fragment-and-replicate join, 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 i^{th} partition of relation *r*

Parallel Nested-Loop Join [2/2]

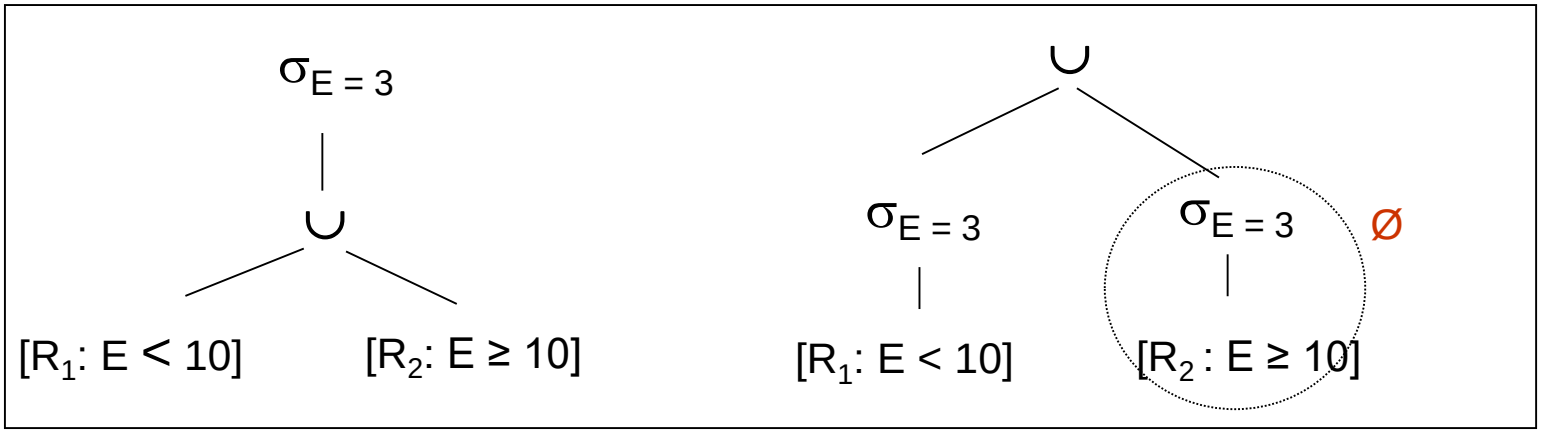


Asymmetric fragment + replicate join

Parallel Selection

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 $l \leq a_i \leq u$ (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



Parallel Duplicate Elimination and Parallel Projection

- 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-partitioning 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

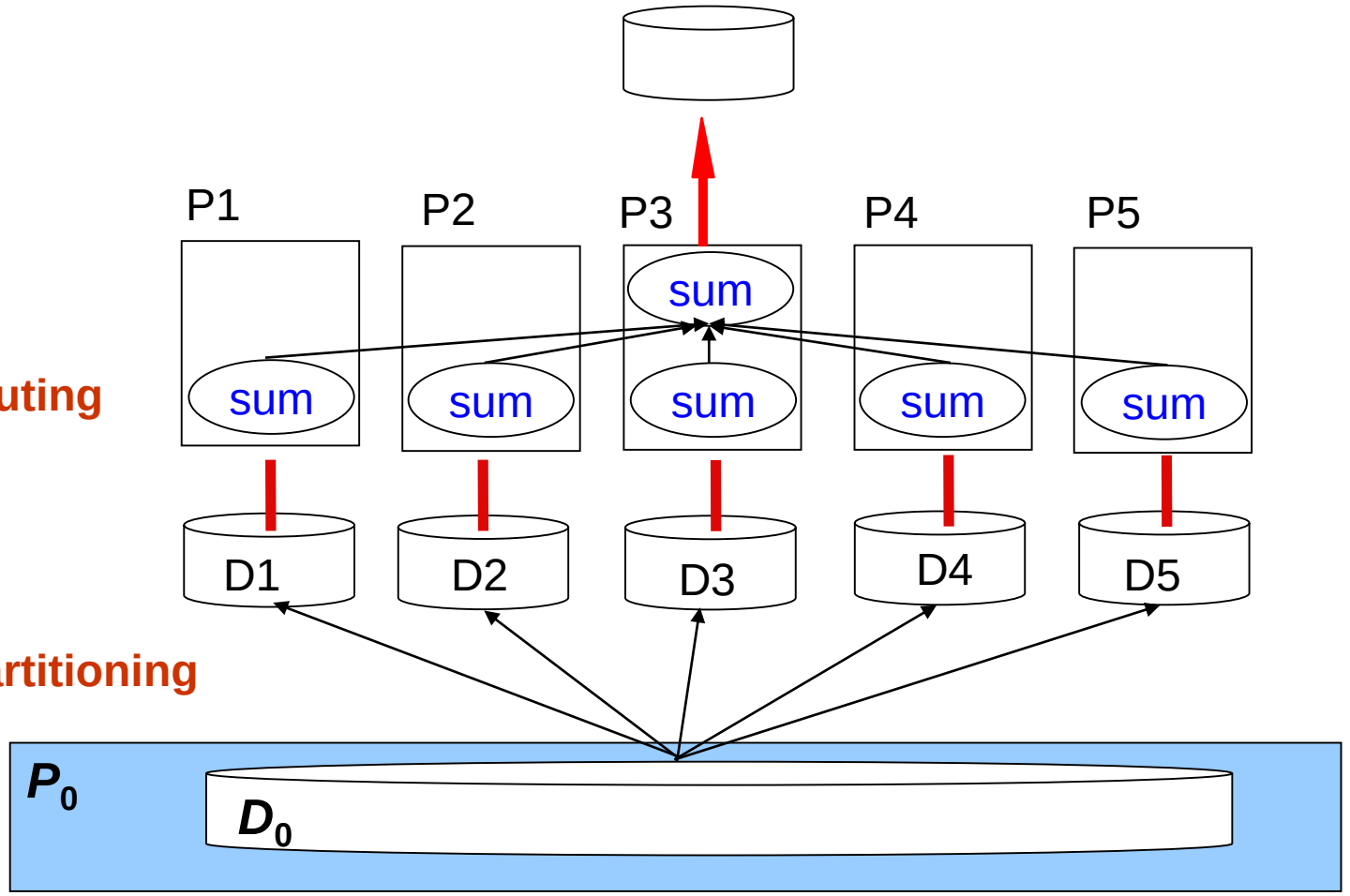
Parallel Grouping/Aggregation [1/2]

- 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

Parallel Grouping/Aggregation [2/2]

Partly computing

Grouping 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 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

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

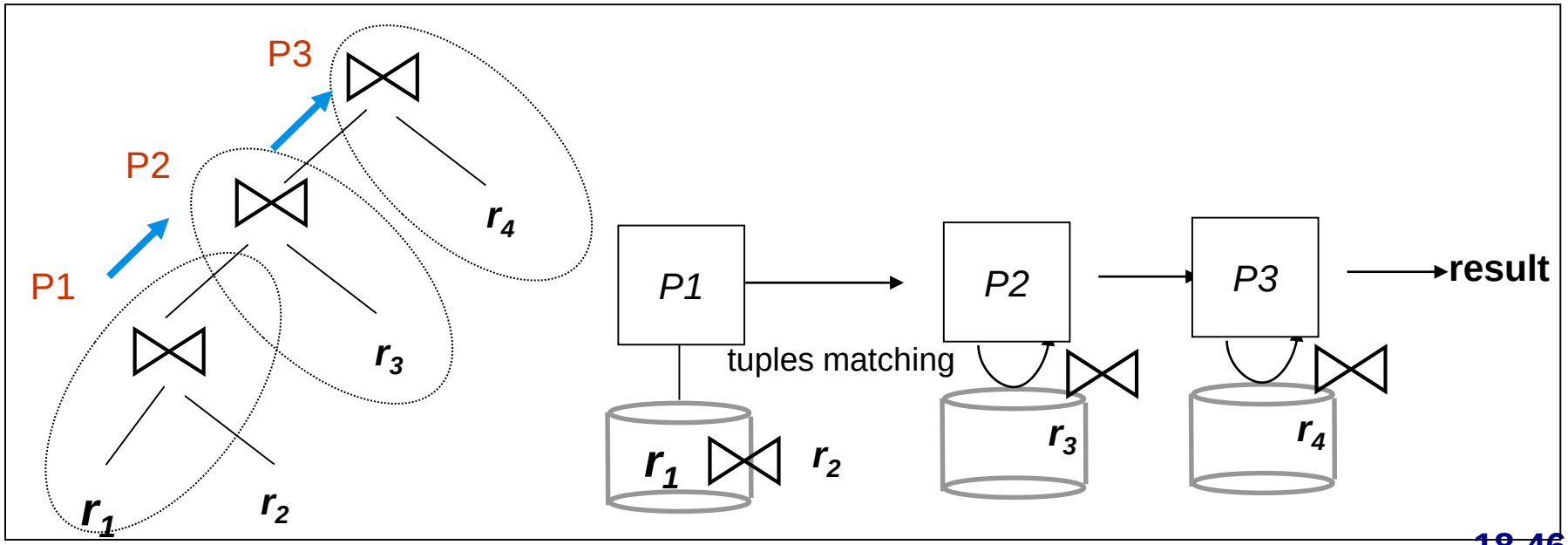
  - T_{part} is the time for partitioning the relations
  - T_{asm} is the time for assembling the results
  - T_i is the time taken for the operation at processor P_i
    - this needs to be estimated taking into account the skew, and the time wasted in contentions

Chapter 18: Parallel Databases

- 18.1 Introduction
- 18.2 I/O Parallelism
- 18.3 Interquery Parallelism
- 18.4 Intraquery Parallelism
- 18.5 Intraoperation Parallelism
- 18.6 Interoperation Parallelism
- 18.7 Query Optimization
- 18.8 Design of Parallel Systems
- 18.9 Parallelism on Multicore Processors

Inter-operator Parallelism: Pipelined Parallelism [1/2]

- 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 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

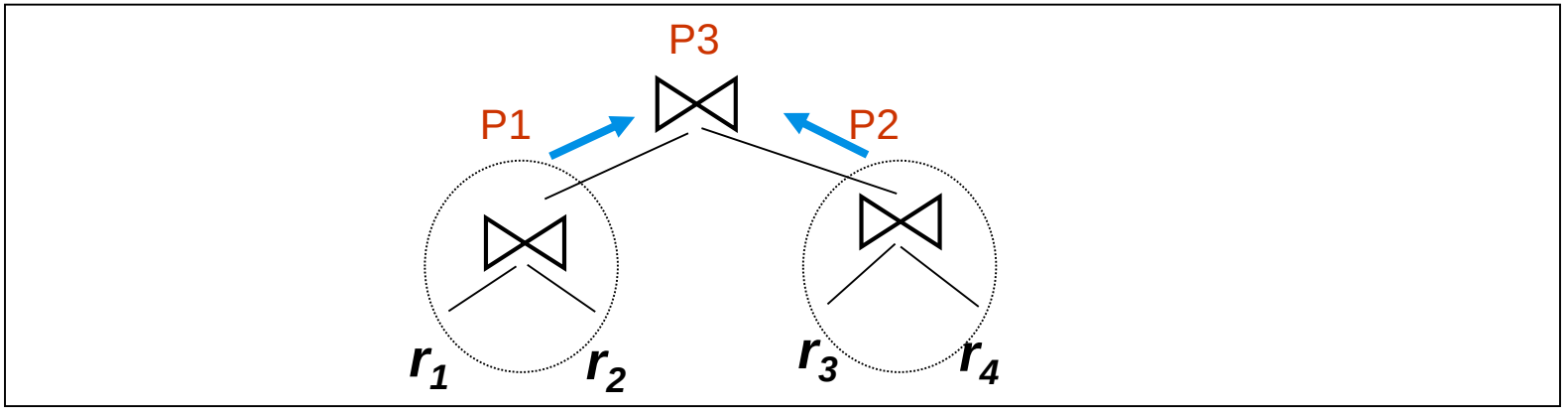


Inter-operator Parallelism: Pipelined Parallelism [2/2]

- Factors limiting Utility of Pipelined 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 operation like set minus, sort, etc)
  - Little speedup is obtained for the frequent cases of skew in which one operator's execution cost is much higher than the others

Inter-operator Parallelism: Independent Parallelism

- Consider a join of four relations : $r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$
  - Let P1 be assigned the computation of temp1 = $r_1 \bowtie r_2$
  - And P2 be assigned the computation of temp2 = $r_3 \bowtie r_4$
  - And P3 be assigned the computation of temp1 $\bowtie$ temp2
  - P1 and P2 can work **independently in parallel**
  - P3 has to wait for input from P1 and P2
    - Can pipeline output of P1 and P2 to P3, combining independent parallelism and pipelined parallelism
- Independent parallelism does not provide a high degree of parallelism
  - useful with a lower degree of parallelism
  - less useful in a highly parallel system



Chapter 18: Parallel Databases

- 18.1 Introduction
- 18.2 I/O Parallelism
- 18.3 Interquery Parallelism
- 18.4 Intraquery Parallelism
- 18.5 Intraoperation Parallelism
- 18.6 Interoperation Parallelism
- 18.7 Query Optimization
- 18.8 Design of Parallel Systems
- 18.9 Parallelism on Multicore Processors

Query Optimization in Parallel DB [1/2]

- 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 an execution tree in parallel system, DBMS 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 in Parallel DB [2/2]

- The number of parallel evaluation plans from which to choose is much larger than the number of sequential evaluation plans
  - Therefore heuristics are needed while optimization
- 2 alternative heuristics for choosing parallel plans:
  - No pipelining and inter-operation pipelining
    - just parallelize every operation across all processors
    - Finding the 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 the 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

Chapter 18: Parallel Databases

- 18.1 Introduction
- 18.2 I/O Parallelism
- 18.3 Interquery Parallelism
- 18.4 Intraquery Parallelism
- 18.5 Intraoperation Parallelism
- 18.6 Interoperation Parallelism
- 18.7 Query Optimization
- 18.8 Design of Parallel Systems
- 18.9 Parallelism on Multicore Processors

Design Issues of Parallel Systems [1/2]

- Large Scale Parallel DBMS
  - OLTP보다는 OLAP에 더 적합
  - Storing large volumes of data (Data Ware House and so on)
  - Processing decision-support queries (OLAP)

- Main Issues of Previous Sections
  - Parallelization of Data Storage
  - Parallelization of Query Processing

- Other Design Issues
  - Parallel loading of data
  - Availability
    - Resilience to failure of some processors or disks
    - On-line reorganization of data and schema changes

Design Issues of Parallel Systems [2/2]

- 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
- 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)

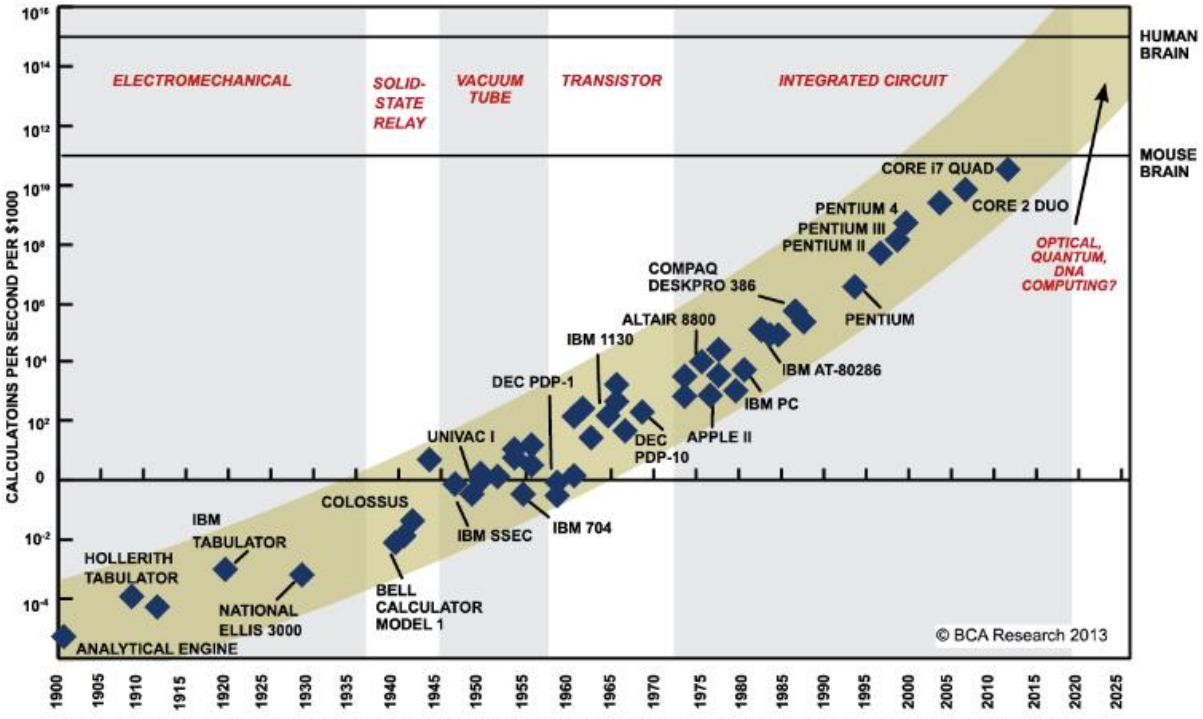
Chapter 18: Parallel Databases

- 18.1 Introduction
- 18.2 I/O Parallelism
- 18.3 Interquery Parallelism
- 18.4 Intraquery Parallelism
- 18.5 Intraoperation Parallelism
- 18.6 Interoperation Parallelism
- 18.7 Query Optimization
- 18.8 Design of Parallel Systems
- 18.9 Parallelism on Multicore Processors

Parallelism vs Raw Speed

- Moore's Law
  - Intel Co-founder Gordon Moore
  - Observation on doubling processor speed in every 18 to 24 months
- The term core is used for an on-chip processor
- Most modern-day computers have a multicore processor

Moore's Law



SOURCE: RAY KURZWEIL, "THE SINGULARITY IS NEAR: WHEN HUMANS TRANSCEND BIOLOGY", P.67, *THE VIKING PRESS*, 2006. DATAPOINTS BETWEEN 2000 AND 2012 REPRESENT BCA ESTIMATES.

Cache Memory and MultiThreading

- Cache misses are critical in DBMS performance
  - DBMS controls disk IO between disk and main memory
  - Computer HW controls data transfer between main memory and caches
- Thread is an executing stream that shares memory with other threads running on a same core
  - If a thread runs into a cache miss, the core proceeds to another thread
- Multicore and mutithreads are source of parallelism
- Sun UltraSPARC T2 one chip processor
  - 8 cores
  - Each core supports 8 threads

Adapting DBMS on MultiCore & MultiThread

- How about one tread for each transaction?
  - Concurrency control schemes may cause cache misses
  - The buffer manager, the log manager, the recovery manager are all potential bottlenecks for multithread environment

- Instead, let multiple cores work on a single transaction
  - The query processor may parallelize queries without excessive demands on cache
  - Pipelining of database operations

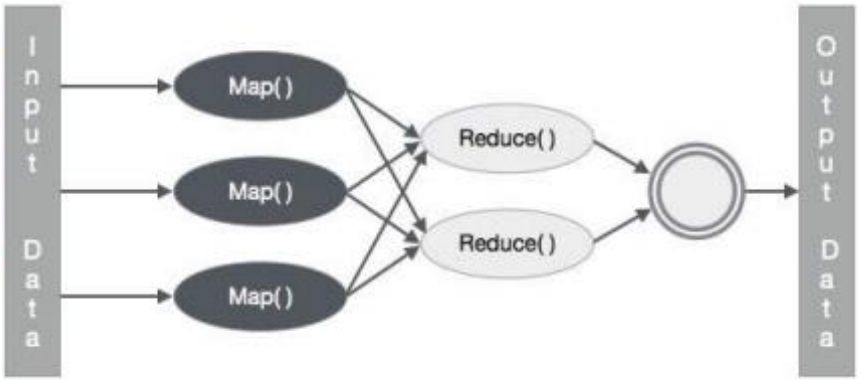
MapReduce [1/2]

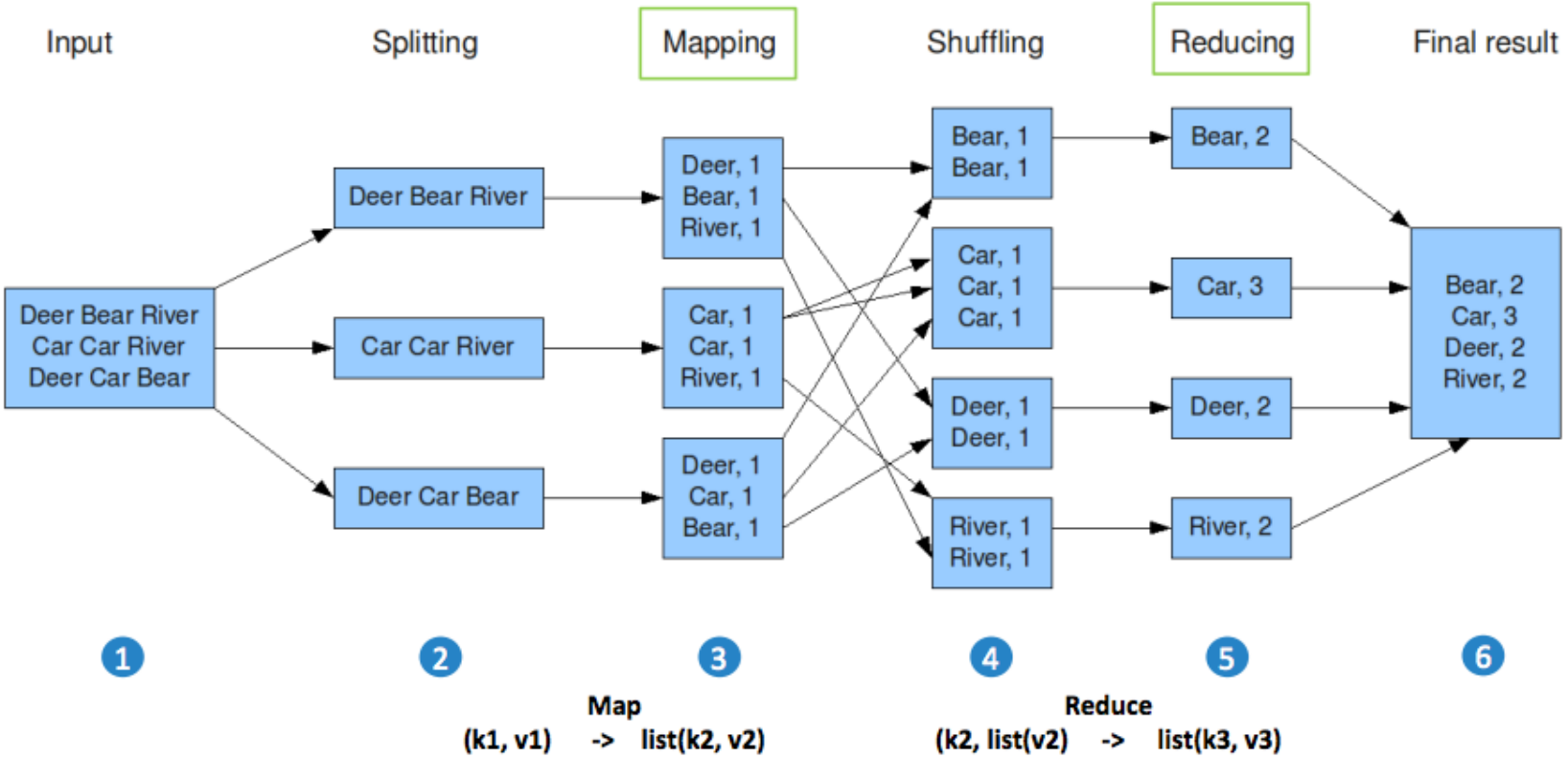
- **MapReduce** is a programming model and an associated implementation for processing and generating large data sets with a parallel, distributed algorithm on a cluster
  - "MapReduce: Simplified Data Processing on Large Clusters", OSDI 2004
- MapReduce
  - User writes Map and Reduce functions
  - Clean abstraction for programmers
  - Automatic parallelization & distribution
  - Fault-tolerant
- Limitations
  - Use disk for data processing
    - Large data for big-data analytics can be loaded only using main memory
    - MapReduce or Hadoop lack supporting in-memory data processing
  - Particularly inefficient for iterative jobs such as machine learning
    - Need to write the intermediate result to the disk
    - Need to load it again for every iteration

MapReduce [2/2]

Map: (k1, v1) -> list(k2, v2)

Reduce: (k2, list(v2) -> list(k3, v3)





Apache Hadoop

- Open-source Framework based on MapReduce for the reliable, scalable, distributed computing of large data sets across clusters of computers using simple programming models
- Designed to scale up from single servers to thousands of machines, each offering local computation and storage
- Provides status and monitoring tools
- Components of Hadoop framework (Architecture)
  - Data Storage : HDFS(Hadoop distributed file system)
    - A distributed file-system that stores data on commodity machines
    - Inspired by GFS(Google File System)
  - Data Processing : Hadoop MapReduce
    - An implementation of the MapReduce programming model for large scale data processing.
    - Inspired by Google MapReduce

Hadoop WordCount Example

- Mapper

```
public static class TokenizerMapper
     extends Mapper<Object, Text, Text, IntWritable>{

  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();

  public void map(Object key, Text value, Context context
                  ) throws IOException, InterruptedException {
    StringTokenizer itr = new StringTokenizer(value.toString());
    while (itr.hasMoreTokens()) {
      word.set(itr.nextToken());
      context.write(word, one);
    }
  }
}
```

- **Mapper implementation, via map method**

```
Hello World Bye World  ⇒  < Hello, 1>
                          < World, 1>
                          < Bye, 1>
                          < World, 1>
```

Hadoop WordCount Example

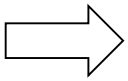
- Reducer

```
public static class IntSumReducer
     extends Reducer<Text,IntWritable,Text,IntWritable> {
  private IntWritable result = new IntWritable();

  public void reduce(Text key, Iterable<IntWritable> values,
                     Context context
                     ) throws IOException, InterruptedException {
    int sum = 0;
    for (IntWritable val : values) {
      sum += val.get();
    }
    result.set(sum);
    context.write(key, result);
  }
}
```

- Reducer implementation, via reduce method

```
< Hello, 1>
< World, 1>
< Bye, 1>
< World, 1>
```



```
< Bye, 1>
< Hello, 1>
< World, 2>`
```

Hadoop WordCount Example

- Main

```
public static void main(String[] args) throws Exception {
  Configuration conf = new Configuration();
  Job job = Job.getInstance(conf, "word count");
  job.setJarByClass(WordCount.class);
  job.setMapperClass(TokenizerMapper.class);
  job.setCombinerClass(IntSumReducer.class);
  job.setReducerClass(IntSumReducer.class);
  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);
  FileInputFormat.addInputPath(job, new Path(args[0]));
  FileOutputFormat.setOutputPath(job, new Path(args[1]));
  System.exit(job.waitForCompletion(true) ? 0 : 1);
 }
}
```

- main for running wordcount example
- Create a job, and set mapper class, reducer class etc.

Hadoop WordCount Example

- Compile & Run
  - Compile WordCount.java and create a jar

```
$ bin/hadoop com.sun.tools.javac.Main WordCount.java
$ jar cf wc.jar WordCount*.class
```

  - Sample text-files as input

```
$ bin/hadoop fs -cat /user/joe/wordcount/input/file01
Hello World Bye World

$ bin/hadoop fs -cat /user/joe/wordcount/input/file02
Hello Hadoop Goodbye Hadoop
```

  - Run the application

```
$ bin/hadoop jar wc.jar WordCount /user/joe/wordcount/input /user/joe/wordcount/output
```

  - Output

```
$ bin/hadoop fs -cat /user/joe/wordcount/output/part-r-00000`
Bye 1
Goodbye 1
Hadoop 2
Hello 2
World 2`
```

Hadoop Simple Example: Addition

- 입력 데이터: 1부터 9까지 정수
- 문제: 짝수끼리 더한 값, 홀수끼리 더한 값을 출력하시오
- 분산환경은 MAP node 3개, Reduce node 2개 인 것으로 가정

input.txt

1
2
3
4
5
6
7
8
9

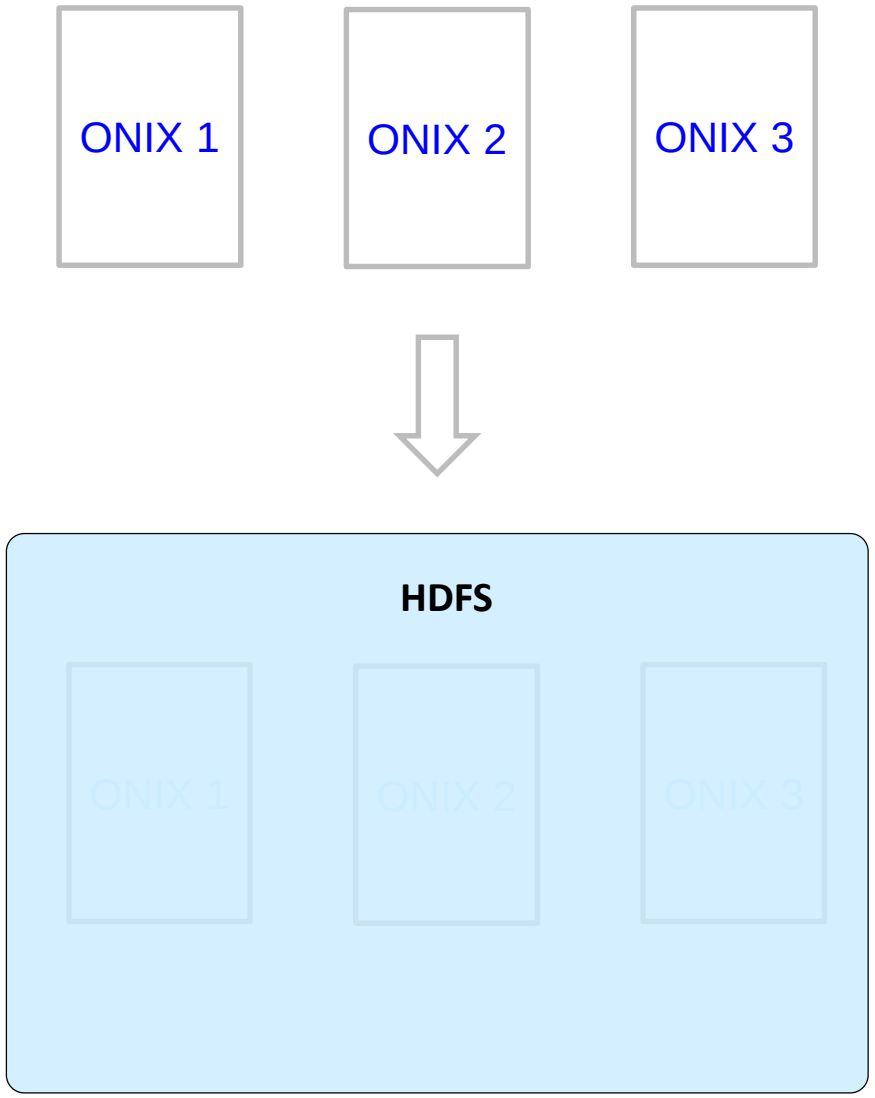
(보통 한 줄에 하나의 정보를 표현합니다)
이 경우, 한 줄에 각 정수 하나씩 표현

1. Data Input [1/2]

쉘에서 HDFS를 실행하면,

start-dfs.sh
…

하나의 가상 파일 시스템이 시작됩니다



1. Data Input [2/2]

실제 디렉토리 처럼 사용가능합니다.

> hadoop –mkdir HDFS/test

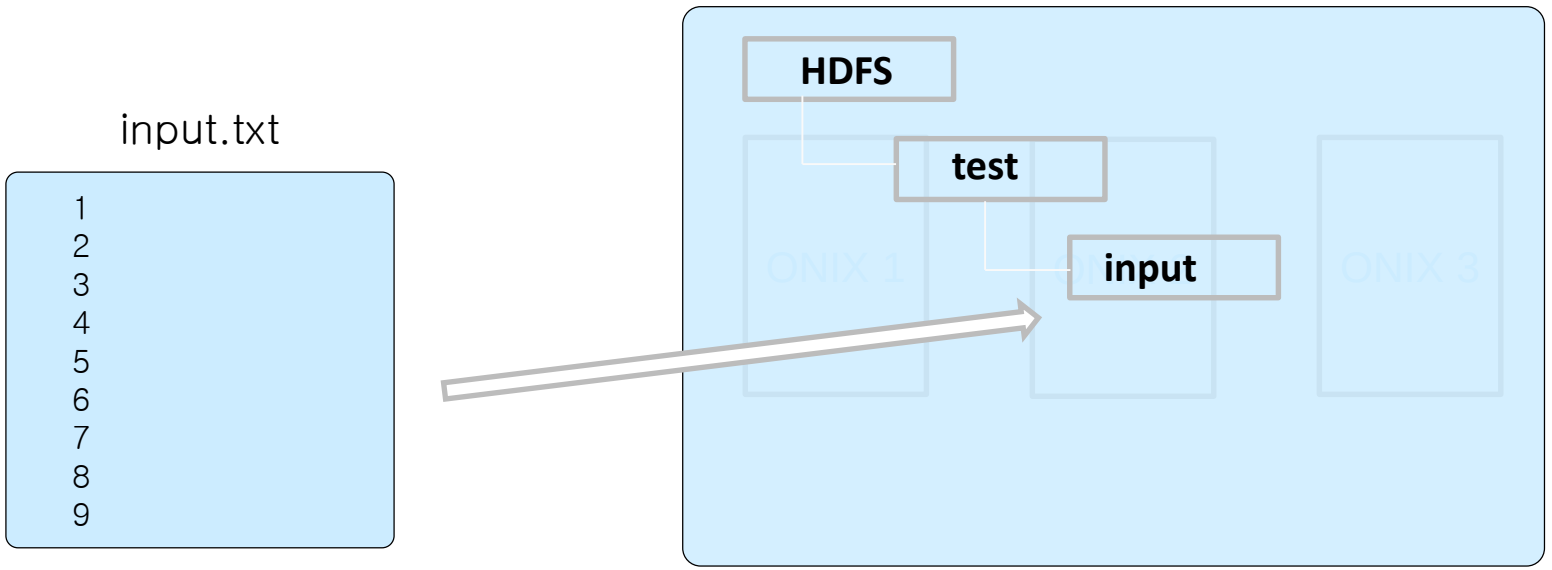
> hadoop –mkdir HDFS/test/input

> hadoop –put input.txt HDFS/test/input

hdfs 루트디렉토리 밑에 test 폴더 생성

hdfs/text 밑에 input 폴더 생성

hdfs/text/input 폴더에 input.txt 복사



2. MapReduce 실행

짝수 홀수 덧셈을 구현한 자바 패키지 파일이 simpleMR.jar 이고

그 안의 메인 클래스가 sumTest 이면,

> hadoop jar simpleMR.jar sumTest HDFS/test/input HDFS/test/result

HDFS/test/input 폴더에 있는 파일들을 입력으로 해서

(폴더에 input1.txt, input2.txt 처럼 여러 파일 나뉘어 있어도 다 처리해줌)

처리한 결과파일은 HDFS/test/result에 출력하라는 명령어

3. MAP Algorithm

- 입력파일을 한 줄 씩 읽어서,
- 그 숫자가 홀수면 Key="even", 으로 해서 Reduce로 보낼 준비
- 그 숫자가 짝수면 Key="odd", 로 해서 Reduce로 보낼 준비

Pseudo Code

```
MAP ( T ) {

        For each t in T
                if t is even
                        EMIT( "even", t )
                else
                        EMIT( "odd", t )
}
```

3. MAP Process

Hadoop이 입력데이터를
나누어 분산해줍니다

input.txt

1
2
3
4
5
6
7
8
9

1
2
3

4
5
6

7
8
9

Mapper 1

```
MAP ( T ) {
    For each t in T
        if t is even
            EMIT( "even", t )
        else
            EMIT( "odd", t )
}
```

(“odd”, 1)
(“even”, 2)
(“odd”, 3)

Mapper 2

```
MAP ( T ) {
    For each t in T
        if t is even
            EMIT( "even", t )
        else
            EMIT( "odd", t )
}
```

(“even”, 4)
(“odd”, 5)
(“even”, 6)

Mapper 3

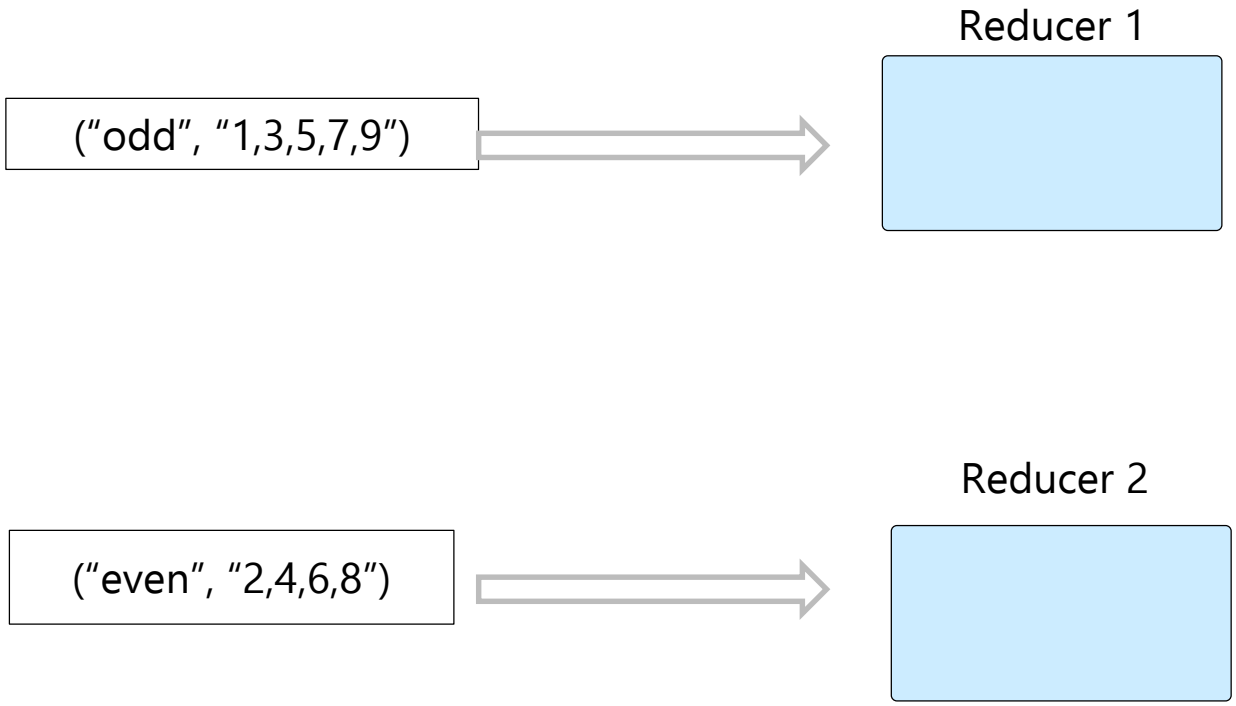
```
MAP ( T ) {
    For each t in T
        if t is even
            EMIT( "even", t )
        else
            EMIT( "odd", t )
}
```

(“odd”, 7)
(“even”, 8)
(“odd”, 9)

4. Intermediate process (Reduce 보내기 전)

4. Intermediate process (Reduce에 전송)

key 별로 묶인 데이터가 Reducer로 보내집니다



5. Reduce Algorithm

- 입력 받은 (key, value list) 데이터에서
- MAP에 의해 홀수, 짝수는 나뉘었으므로
- value list를 다 더해줘서 출력하면 완료

Pseudo Code

```
REDUCE ( key k, value V ) {
        int sum = 0

        For each v in V
                sum = sum + v

        EMIT(k, v)
}
```

5. Reduce Process

Reducer 1에 들어온 데이터는 key가 odd인 value만 있음, 다 합치면 1+3+5+7+9 =25

Reducer 1

("odd", "1,3,5,7,9")

```
REDUCE ( key k, value V ) {
    int sum = 0
    For each v in V
        sum = sum + v
    EMIT(k, v)
}
```

("odd", "25")

Reducer 2

("even", "2,4,6,8")

```
REDUCE ( key k, value V ) {
    int sum = 0
    For each v in V
        sum = sum + v
    EMIT(k, v)
}
```

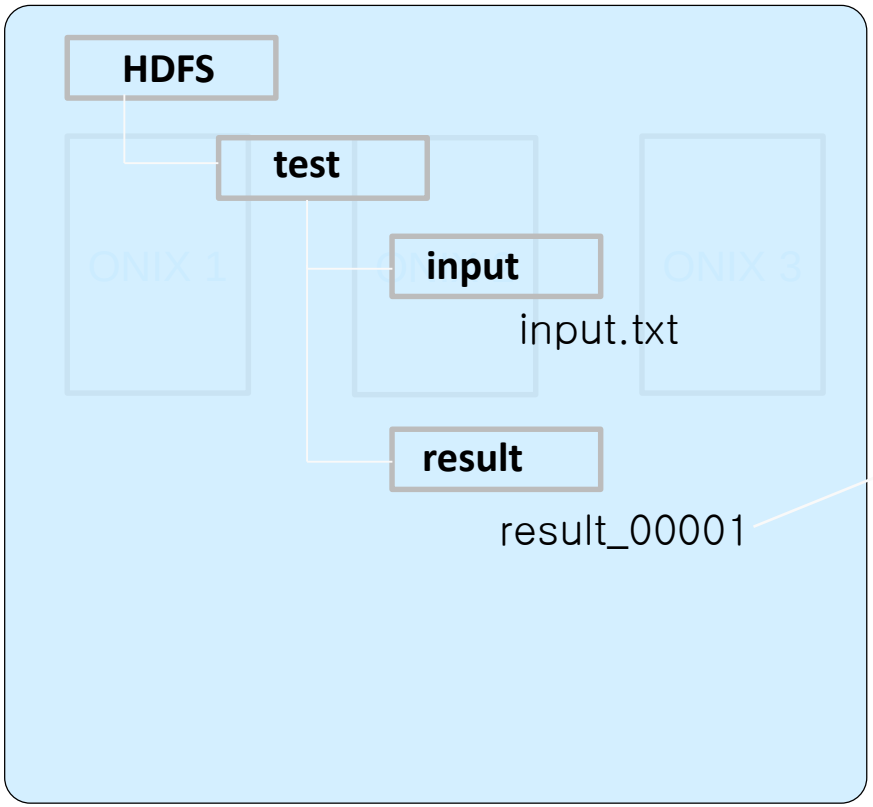
("odd", "20")

V = {2,4,6,8} 이므로 sum = 20이고 EMIT(k,v)에 의해 ("odd", "20") 이 출력됨

6. MapReduce 결과

아까 아래와 같은 명령에 의해 출력된 결과는 HDFS/test/result에 생깁니다

> hadoop jar simpleMR.jar sumTest HDFS/test/input HDFS/test/result



result_00001 파일의 내용

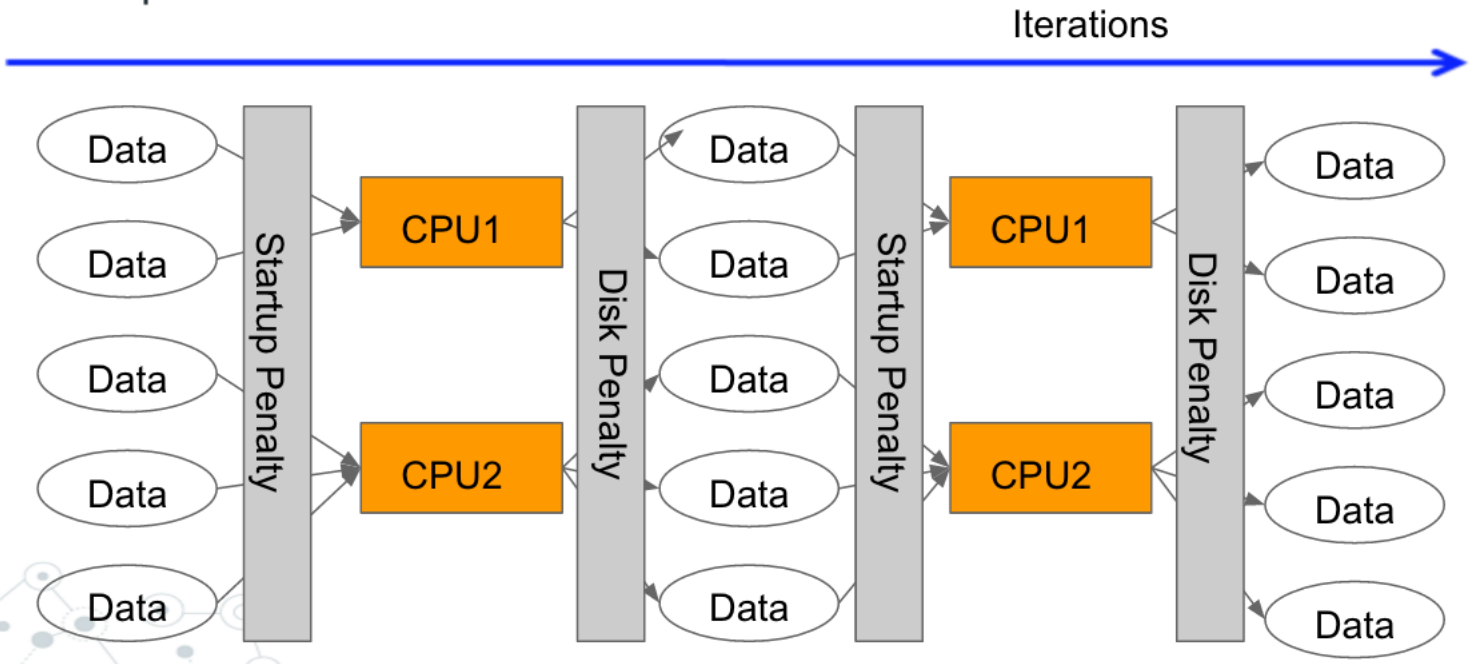
(결과 파일 이름은 하둡이 임의로 생성)

even	25
odd	20

Limitations of MapReduce & Hadoop

- Use disk for data processing
  - Large data for big-data analytics can be loaded only using main memory
  - MapReduce or Hadoop lack supporting in-memory data processing
- Particularly inefficient for iterative jobs such as machine learning
  - Need to write the intermediate result to the disk
  - Need to load it again for every iteration

MapReduce Overhead



Apache Spark [1/3]

- **Apache Spark** is an open source cluster computing framework
  - Originally developed at the University of California, Berkeley's AMPLab, 2014
  - provides an interface for programming entire clusters with implicit data parallelism and fault-tolerance

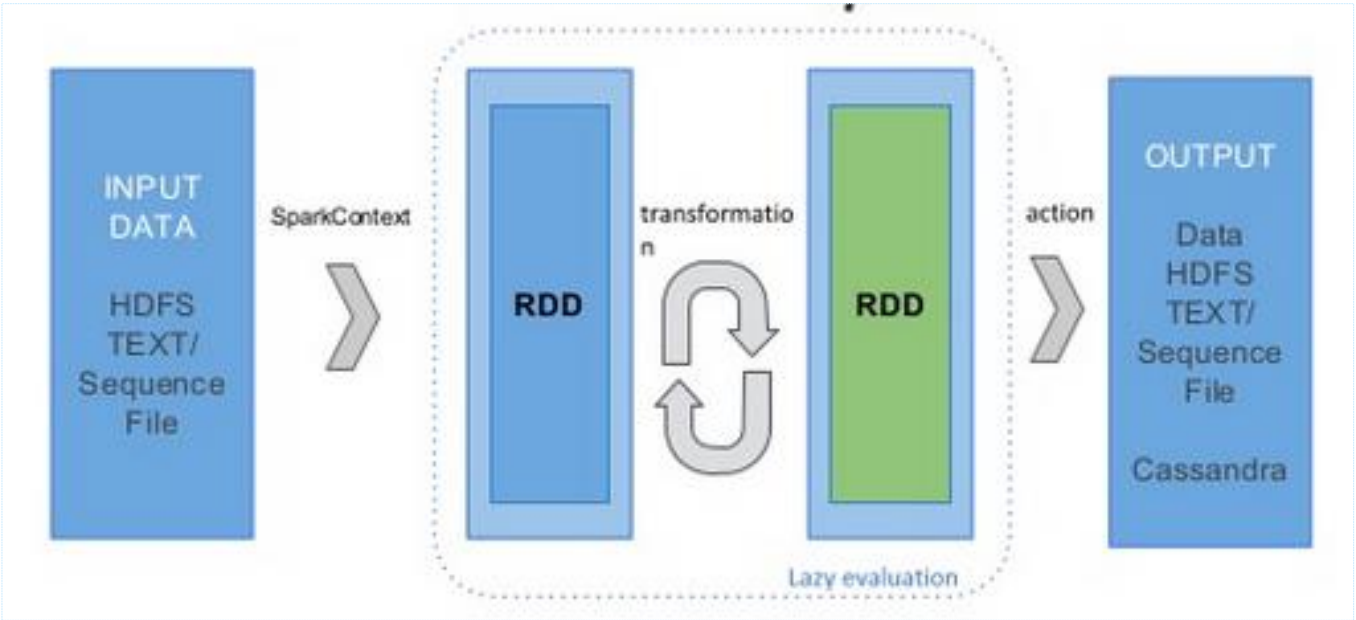
- In-memory cluster computing frameworks supporting general big-data analytics
  - Supports Python, Scala, Java
  - Supports a wide range of applications

- Provide RDD(Resilient Distributed Datasets) as an abstraction for data residing in memory

- Characteristics
  - Efficient fault recovery
  - Good generality and programmability

Apache Spark [2/3]

- RDD
  - Resilient Distributed Dataset
  - Read-only & Partitioned collection of records
  - RDD supports two types of operations
    - Transformation – making another RDD
      - Lazily executed
      - Optimized schedule would be executed later on
    - Action - making a result output



Spark WordCount Example

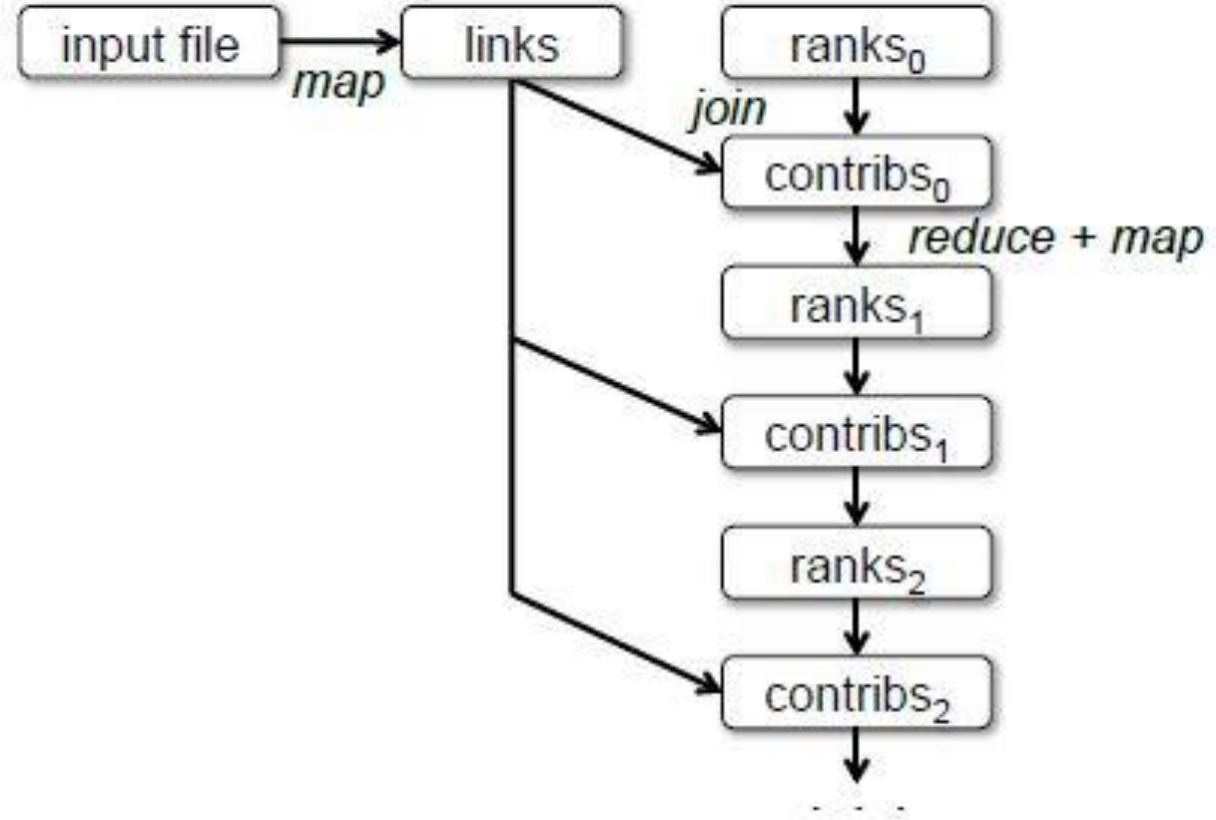
```
val textFile = sc.textFile("hdfs://...")
val counts = textFile
    .flatMap(line => line.split(" "))
    .map(word => (word, 1))
    .reduceByKey((x, y) => x + y)
counts.saveAsTextFile("hdfs://...")
```

Creating RDD

Transformations

Action -> Actually executed here!

Apache Spark [3/3]



- RDD Lineage
  - A graph for ensuring fault-tolerance used in Spark
  - Track how each and from which each RDD is transformed
  - Lineage data size is small enough for In-memory computing