CS 4440 A

Emerging Database Technologies

Lecture 20

04/08/26

Announcements

- Assignment 4 due today
- Assignment 5 released on Monday
  - Start early! Up to 3 submission per day
- Next Monday (Apr 13): Tech presentation
  - Required attendance

Agenda

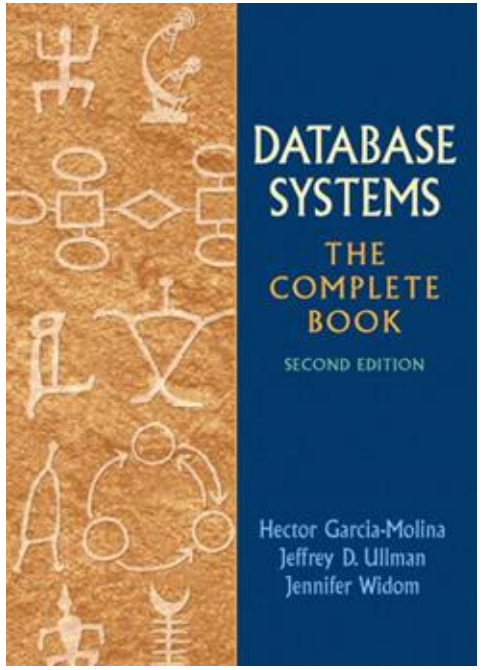
1. Spark

2. Parallel DBMS

Reading Materials

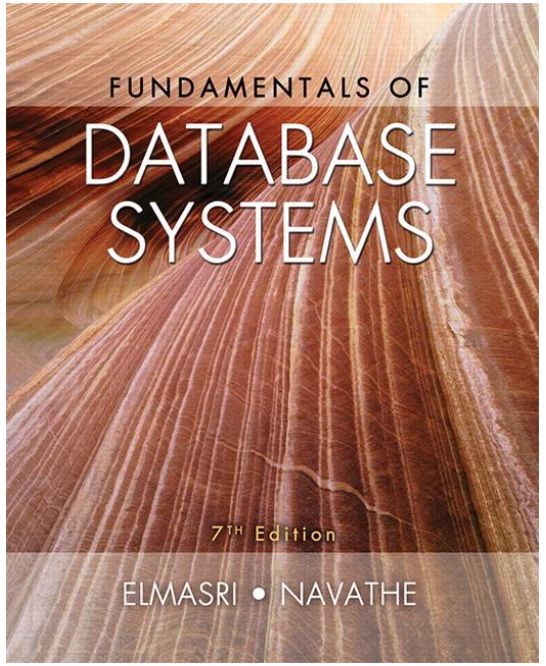
Database Systems: The Complete Book (2nd edition)

- Chapter 20: Parallel and Distributed Databases



Fundamental of Database Systems (7th Edition)

- Chapter 23: Distributed Database Concepts

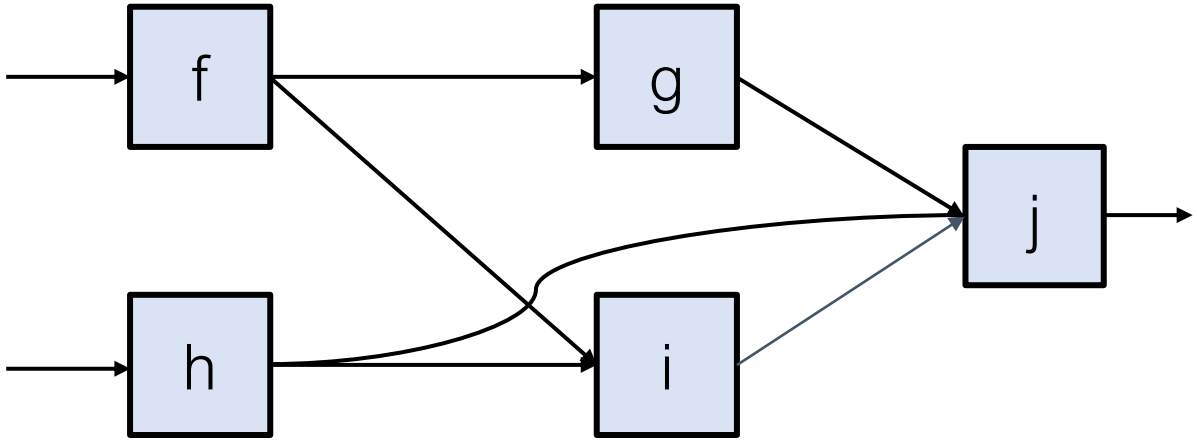


1. Spark

Workflow systems

- Extends MapReduce by supporting acyclic networks of functions
  - Simple two-step workflow → any acyclic (DAG) workflow of functions
  - Each function implemented by a collection of tasks
  - A master controller is responsible for dividing work among tasks

- Examples: Apache Spark and Google TensorFlow

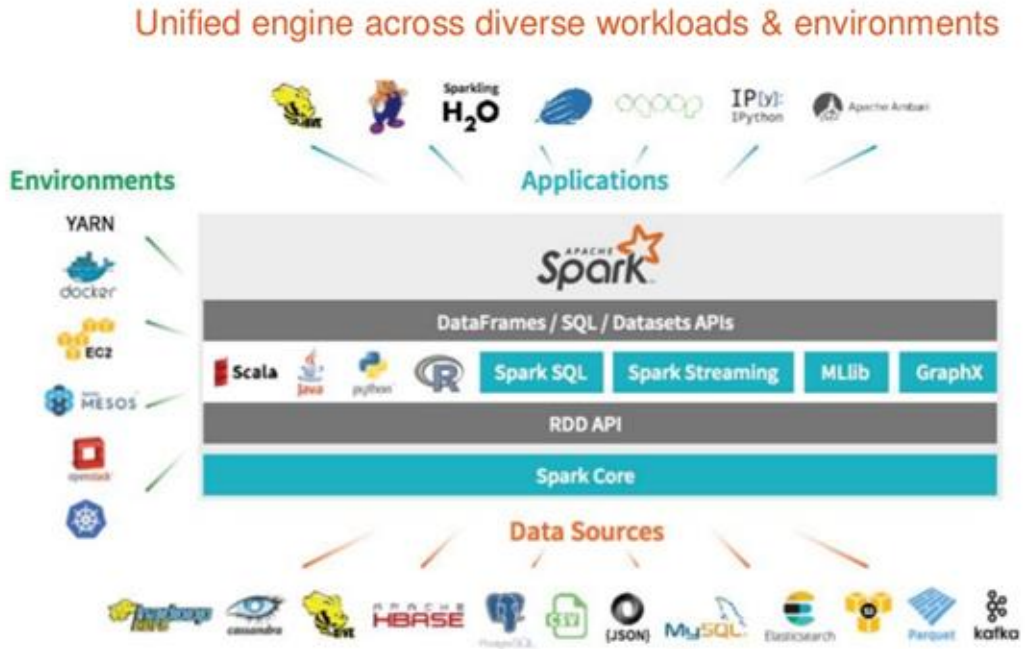


Blocking property

- Like MapReduce, workflow functions only deliver output after completion
- If task fails, no output is delivered to any successors in flow graph
- A master controller can therefore restart failed task at another compute node

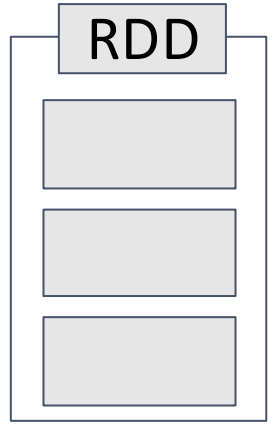
Spark: most popular workflow system

- Developed by UC Berkeley and Databricks, now maintained by Apache
- Advantages over early workflow systems
  - More efficient failure handling
  - More efficient grouping of tasks among compute nodes and scheduling function execution
  - Integration of programing language features such as looping and function libraries



Data Model: Resilient distributed dataset (RDD)

Central data abstraction of Spark



A file of objects of one type

- Statically typed: RDD[T] has objects of type T

Immutable collections of objects, together with its lineage

- Lineage = how a dataset is computed

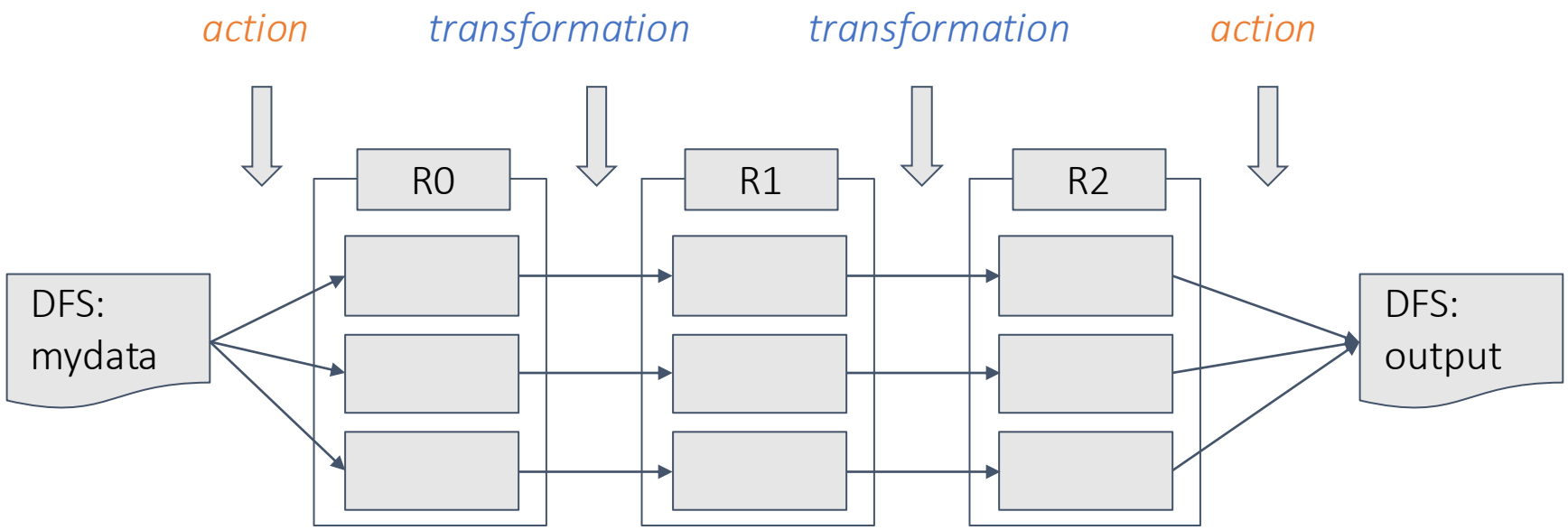
Spark is resilient against loss of any or all chunks of RDD

- If RDD in main memory is lost, can recompute lost partitions of RDD using lineage

Spark program

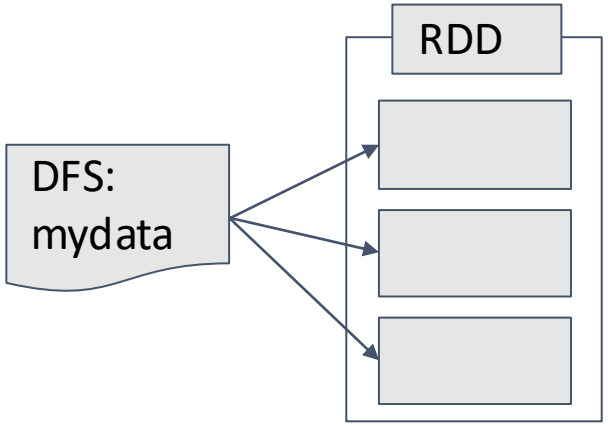
Sequence of steps of

- Transformations: apply some function to an RDD to produce another RDD
- Actions*:* Turn RDD into data in surrounding file system and vice versa



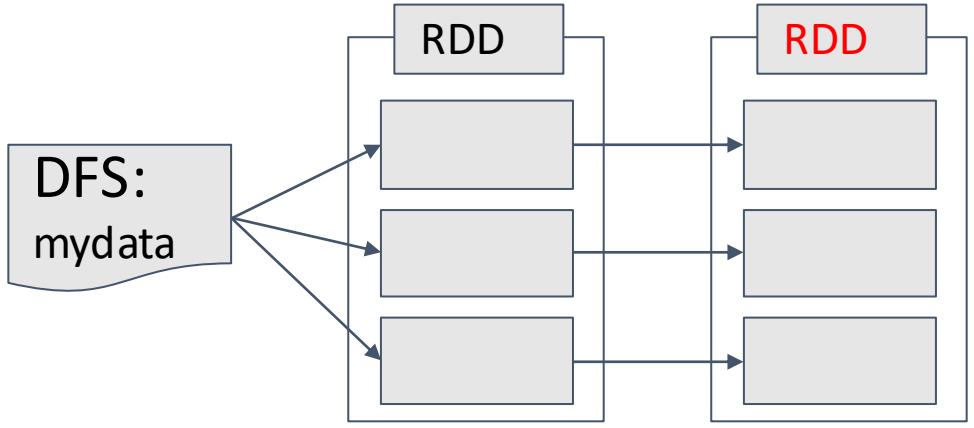
Example: average word length by letter

```
> avglens = sc.textFile(file)
```



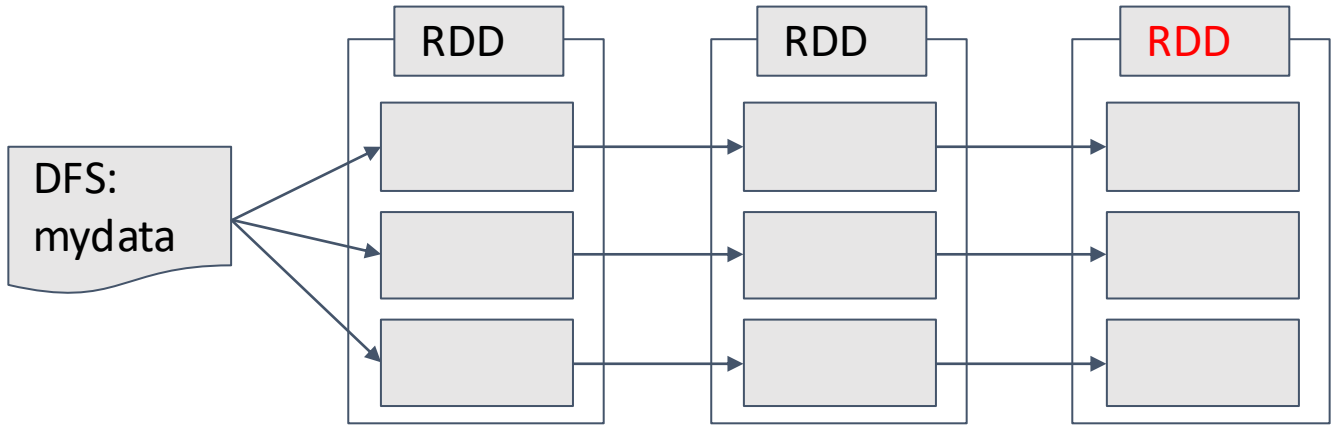
Example: average word length by letter

```
> avglens = sc.textFile(file) \
    .flatMap(lambda line: line.split())
```



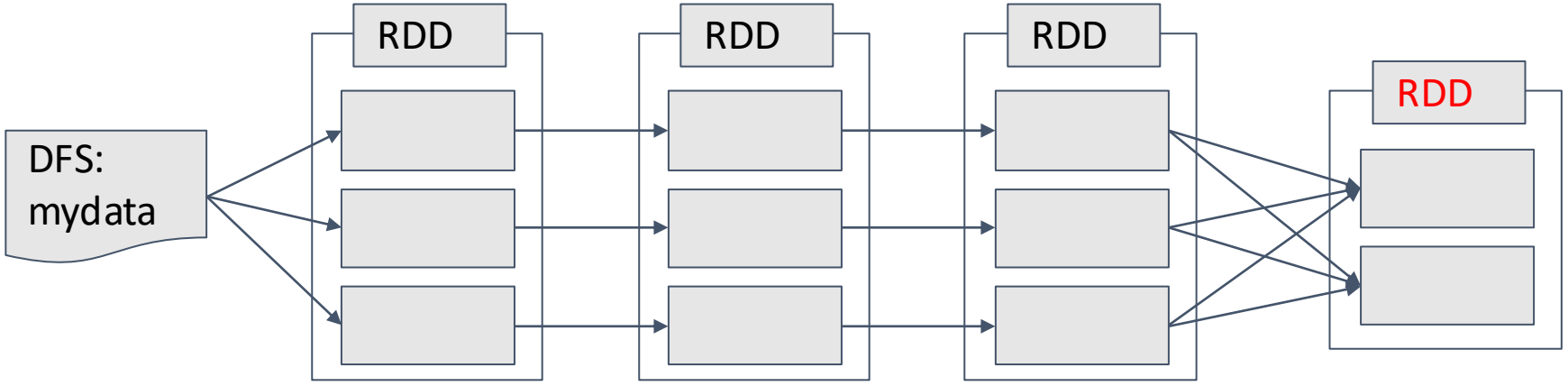
Example: average word length by letter

```
> avglens = sc.textFile(file) \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word[0], len(word)))
```



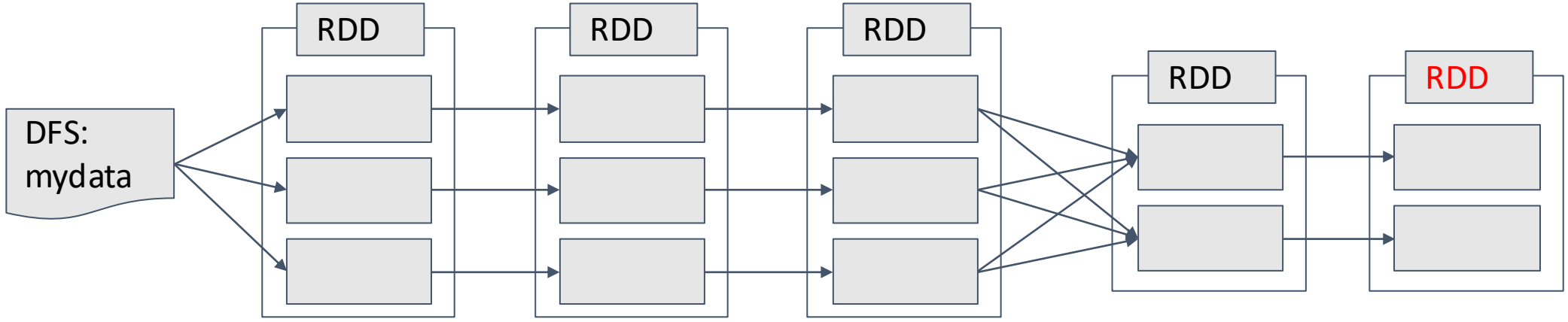
Example: average word length by letter

```
> avglens = sc.textFile(file) \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word[0], len(word))) \
    .groupByKey()
```



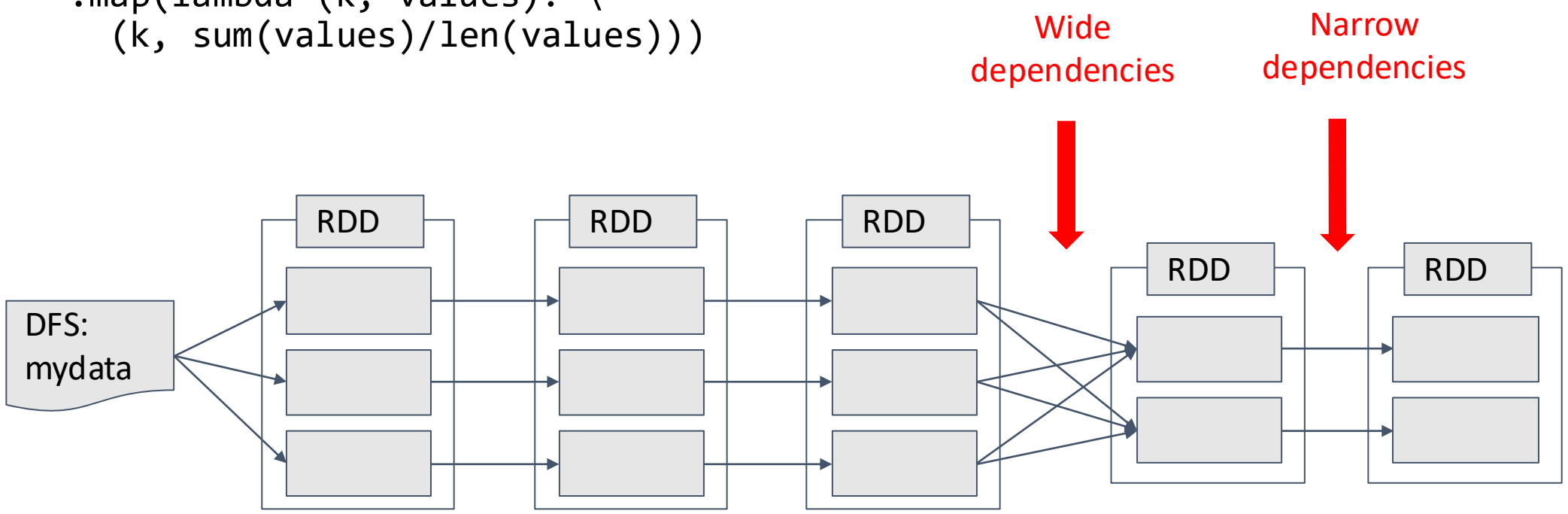
Example: average word length by letter

```
> avglens = sc.textFile(file) \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word[0], len(word))) \
    .groupByKey() \
    .map(lambda (k, values): \
      (k, sum(values)/len(values)))
```



Example: average word length by letter

```
> avglens = sc.textFile(file) \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word[0], len(word))) \
    .groupByKey() \
    .map(lambda (k, values): \
      (k, sum(values)/len(values)))
```



Map

- Transformation that takes function as parameter and applies it to every element of RDD
- Returns a new RDD where each input element is transformed into exactly one output element (one-to-one mapping).
- Not exactly the same as Map of MapReduce
  - In MapReduce, a Map function is applied to a key-value pair and produces a set of key-value pairs
  - In Spark, a Map function can apply to any object type, but produces exactly one object

```
> avglens = sc.textFile(file) \
   .flatMap(lambda line: line.split()) \
   .map(lambda word: (word[0], len(word))) \
   ...
```

Flatmap

- Transformation analogous to MapReduce Map, but no restriction on the type
- In comparison to a Spark Map, each object maps to a list of 0 or more objects
- All the lists are then “flattened” into a single RDD of objects

```
> avglens = sc.textFile(file) \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word[0], len(word))) \
    ...
```

Filter

- Transformation that takes a predicate that applies to the RDD object type and returns elements that satisfy predicate

```
> avglens = sc.textFile(file) \
    .flatMap(lambda line: line.split()) \
    .filter(lambda word: word not in stoplist)) \
    ...
```

Reduce

- An action (not transformation) that returns a value instead of an RDD
- Takes parameter that is a function of type (V, V) => V
  - When applied to RDD, the function is repeatedly applied on pairs of elements to produce a single one
  - Function can be associative and commutative (e.g., addition), but this is not required

```
> totlen = sc.textFile(file) \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: len(word)) \
    .reduce(lambda a, b: a + b)
```

Other examples of actions

- Actions are operations that trigger the execution of the Spark computation and return results to the driver program or write data to external storage systems

```
# Collect RDD elements to the drive program
collected_data = rdd.collect()


# Count the number of elements in the RDD
count = rdd.count()


# Get the first three elements of the RDD
element = rdd.take(3)


# Save RDD elements to a text file
rdd.saveAsTextFile(”output_folder”)
```

Relational database operations

- Some Spark operations behave like relational algebra operations on relations that are represented by RDD's

Join

- Takes two RDD's of type key-value pair where the key types are the same
- For each pair (*k*, *x*) and (*k*, *y*), produce (*k*, (*x*, *y*))
- Output RDD consists of all such objects

```
> x = sc.parallelize([("a", 1), ("b", 4)])
> y = sc.parallelize([("a", 2), ("a", 3)])
> x.join(y).collect()
[('a', (1, 2)), ('a', (1, 3))]
```

GroupByKey

- Takes RDD of key-value pairs, produces a set of key-value pairs
  - The value type for the output is a list of values of the input type
- Sorts input RDD by key
- For each key *k* produces the pair (*k*, [v_1, v_2, …, v_n]) for v_i's associated with *k*

```
> avglens = sc.textFile(file) \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word[0], len(word))) \
    .groupByKey() \
    ...
```

Spark implementation

Similar to MapReduce

- RDD is divided into chunks, which are given to different compute nodes
- Transformation on RDD can be performed in parallel on each of the chunks

Two key improvements

- Lazy evaluation of RDD's
- Lineage for RDD's

Lazy evaluation

Spark does not actually apply transformations to RDD's until it is required to do so (e.g., storing RDD to file system or returning a result to application)

```
val data = sc.textFile("input.txt")      // No execution yet
    .map(line => line.split(" "))        // Not executed
    .filter(words => words.length > 2)   // Still not executed
    .count()                             // Now it executes everything
```

Lazy evaluation

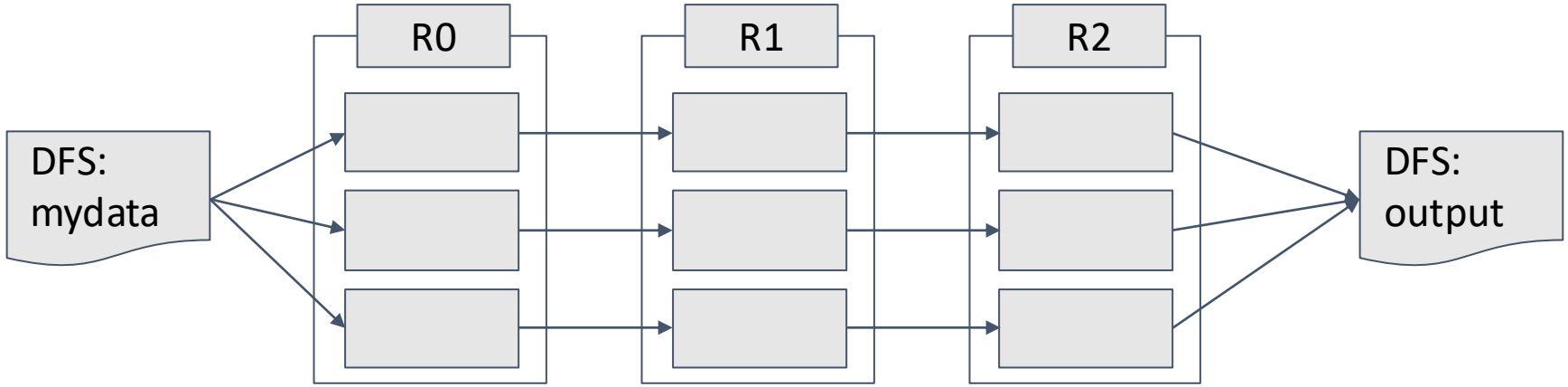
Spark does not actually apply transformations to RDD's until it is required to do so (e.g., storing RDD to file system or returning a result to application)

Potential Benefits:

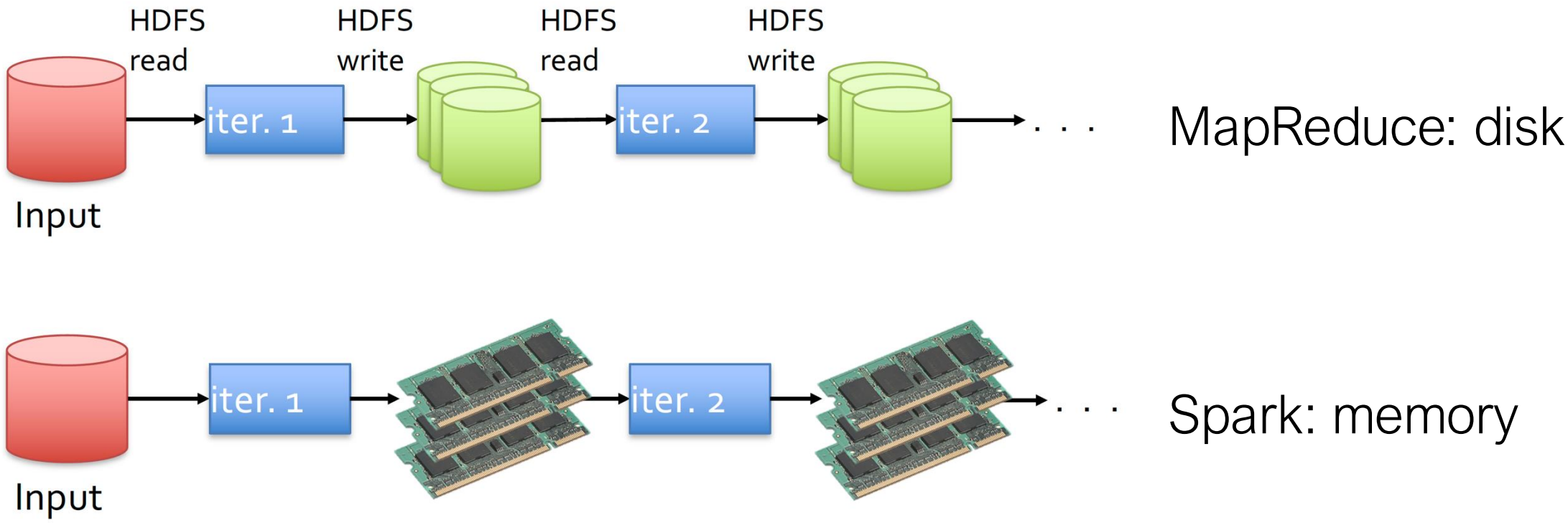
- Spark can analyze entire chain of operations and combining multiple operations to reduce unnecessary computations
- No immediate computation/memory usage; resources allocated only when needed
- Optimizes data shuffling and stages

Resilience of RDD's

- Spark records the *lineage* of every RDD, which can be used to re-create any RDD
  - If R_2 is lost, reconstruct from R_1
  - If R_1 is lost, reconstruct from R_0
  - If R_0 is lost, reconstruct from file system



Data Sharing in MapReduce vs Spark



This is why Spark is significantly faster for iterative algorithms

Spark programming guide and paper

- To learn more about writing Spark applications, please read the Spark programming guide: https://spark.apache.org/docs/latest/rdd-programming-guide.html

- Recommend reading: the Spark paper

2. Parallel DBMS

Parallel vs. Distributed DBMS

Parallel Databases:

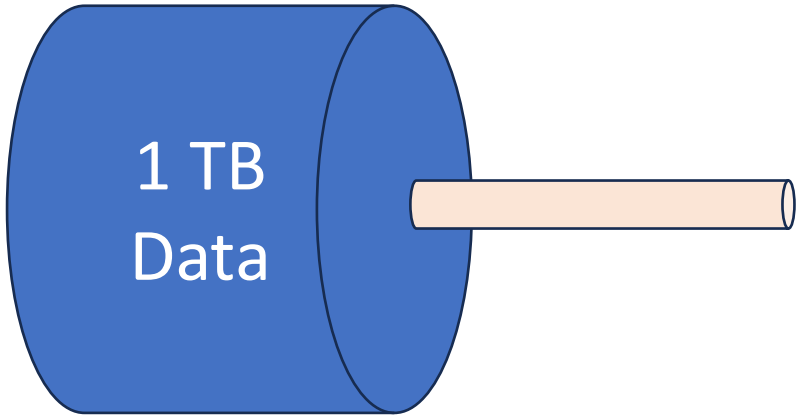
- Parallelization of various operations
  - e.g., loading data, building indexes, evaluating queries
- Data may or may not be distributed initially
- Distribution is governed by performance consideration

Distributed Databases:

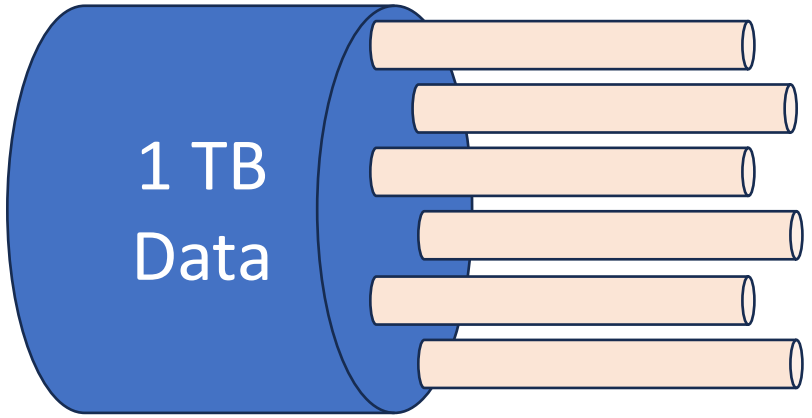
- Data is physically stored across different sites
  - Each site is typically managed by an independent DBMS
- Location of data and autonomy of sites have an impact on query optimization, concurrency control and recovery
- Distribution also governed by other factors
  - Increased availability for system crash
  - Local ownership and access

Benefits of Parallelism

Parallelism: divide a big problem into many smaller ones to be solved in parallel



At 10 MB/s
1.2 days to scan

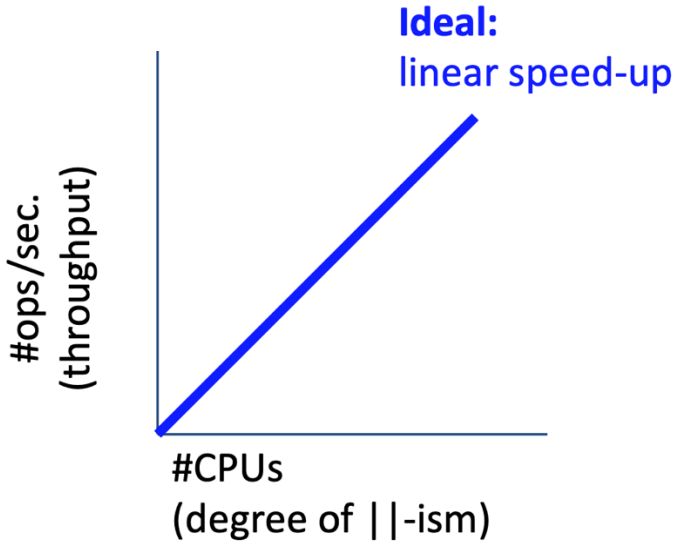


1,000 x parallel
1.5 minute to scan

Key Metrics

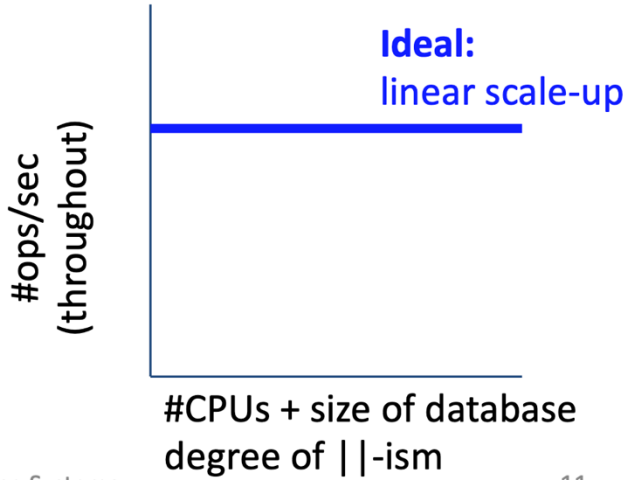
Speed-up

- Increase HW, keep workload
- More resources means proportionally less time for given amount of data



Scale-up

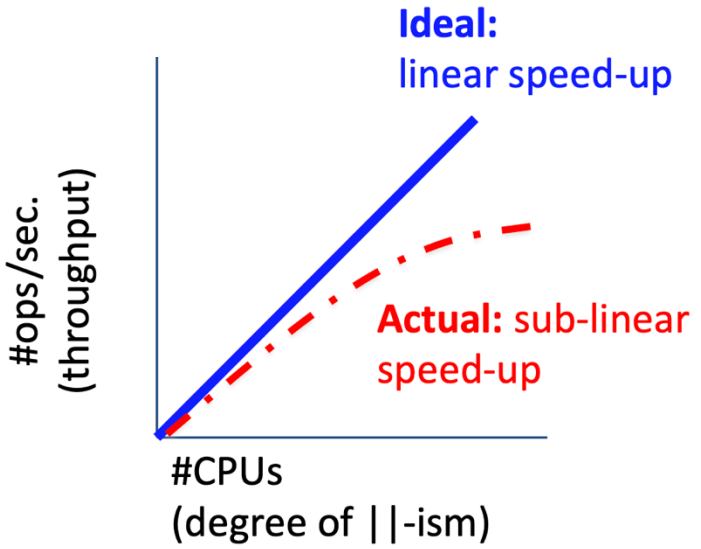
- Increase HW, increase workload
- If resources increased in proportion to increase in data size, time is constant.

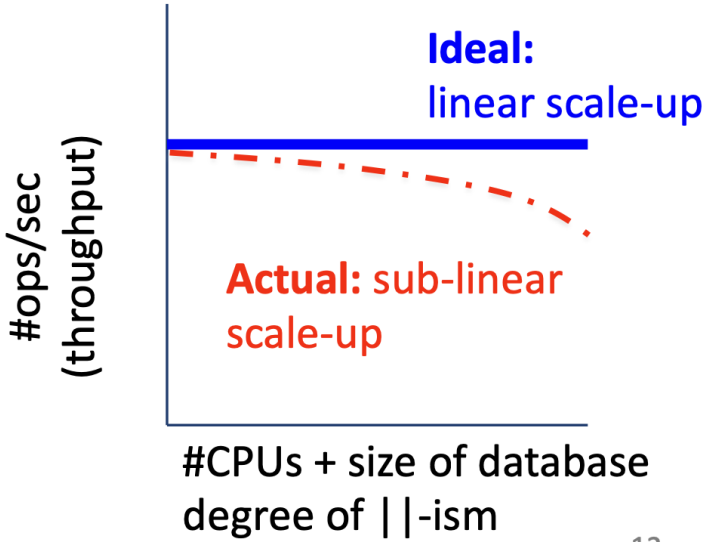


Key Metrics

In practice, due to overheads in parallel processing:

- **Start-up cost**: Starting the operation on many processor, might need to distribute data
- **Interference**: Different processors may compete for the same resources
- **Skew**: The slowest processor (e.g. with a huge fraction of data) may become the bottleneck





Models of Parallelism

Units: a collection of processors

- Think hundreds or thousands of processors
- assume always have local cache
- may or may not have local memory or disk (next)

A communication facility to pass information among processors

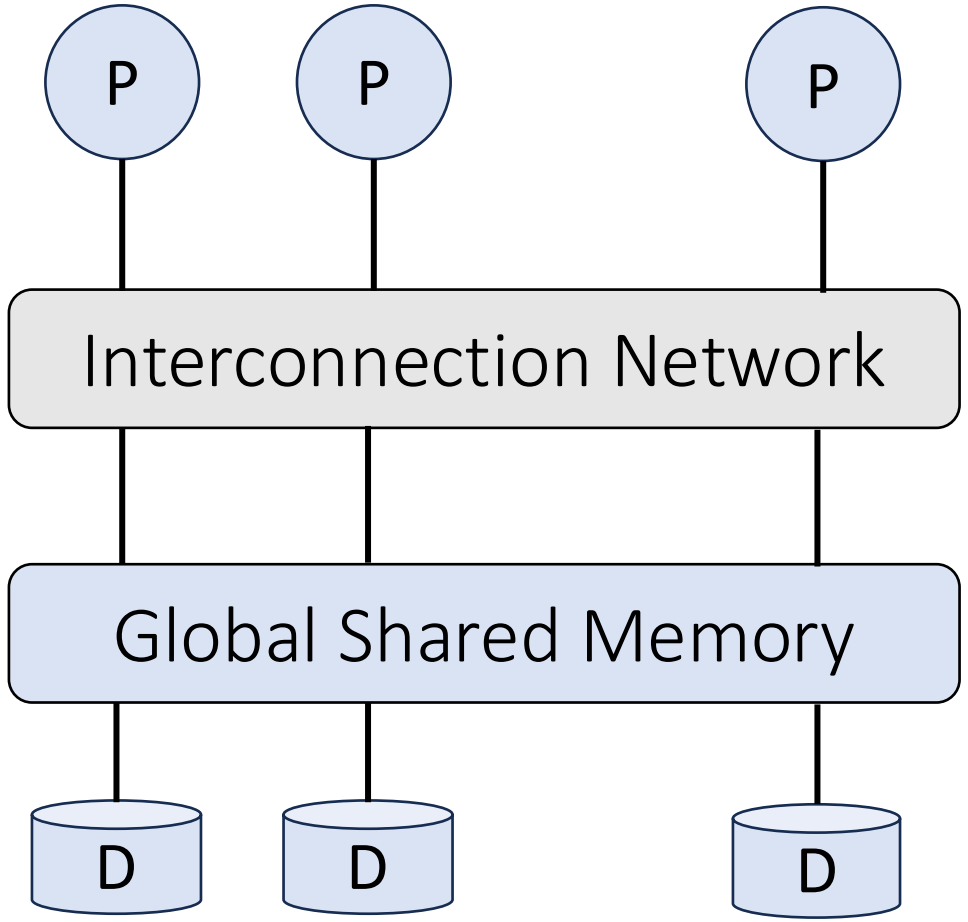
- a shared bus or a switch

Different architecture:

- Whether memory AND/OR disk are shared
- 3 main groups: shared-memory, shared-disk, shared-nothing

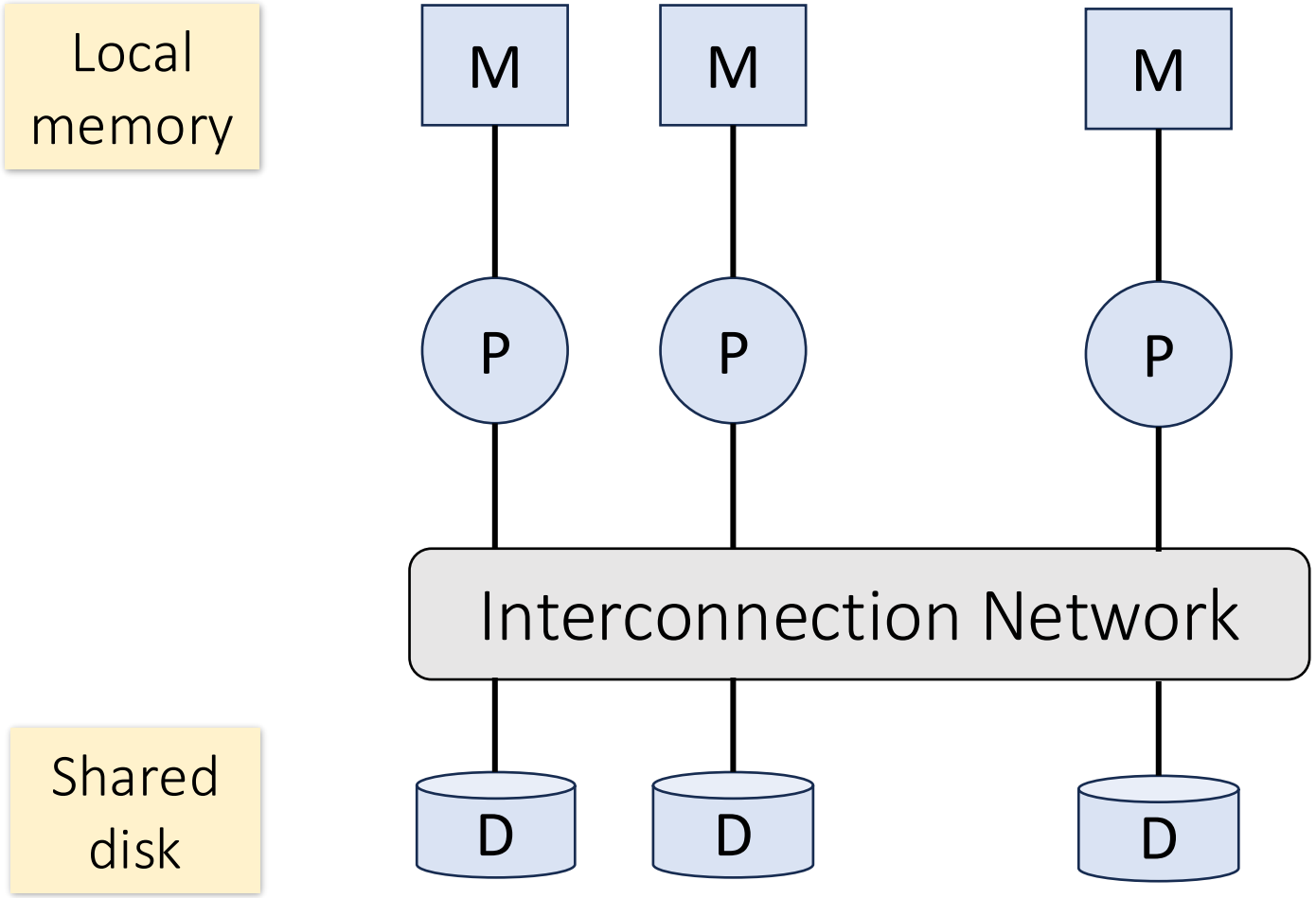
Shared-Memory Architecture

Shared memory



- e.g., NUMA (non uniform memory access)
- Easy to program
- Low communication overhead due to shared memory
- Difficult to scale up (memory contention) and expensive to build

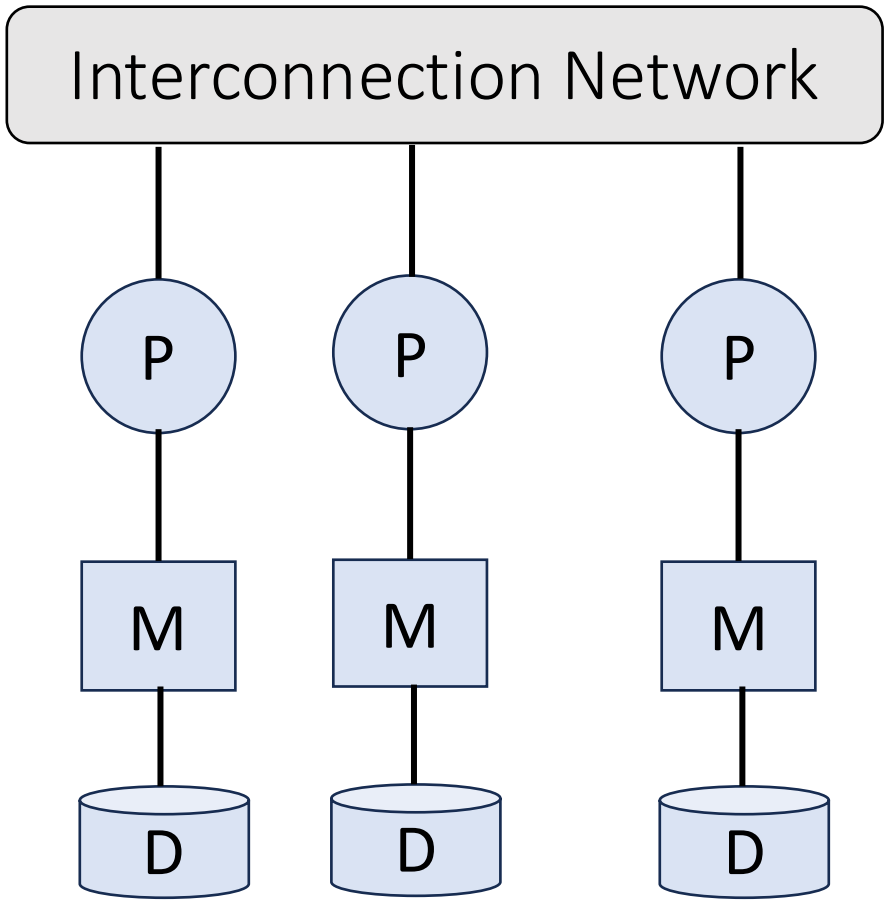
Shared-Disk Architecture



- Centralized storage system (e.g., SAN or NAS), but compute is distributed
- Better scalability than shared memory, but still subject to contention of disk/network bandwidth

Shared-Nothing Architecture

communicate over the network

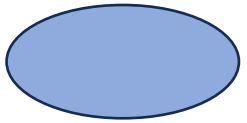


Local memory and disk

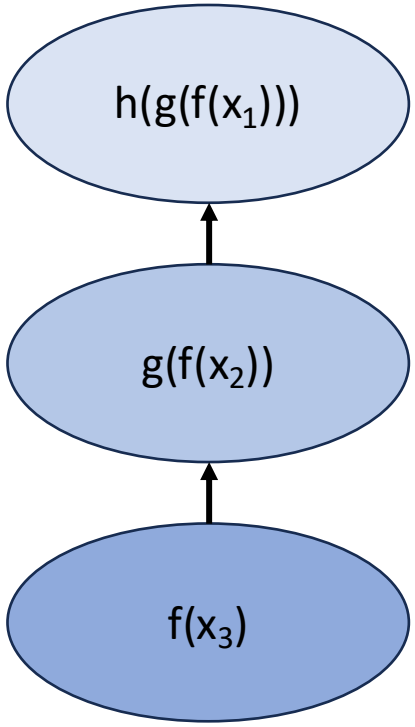
- Excellent horizontal scalability; relatively inexpensive to build
- Minimal resource contention but higher communication overhead
- Hard to program and design parallel algos

* We will assume this architecture by default

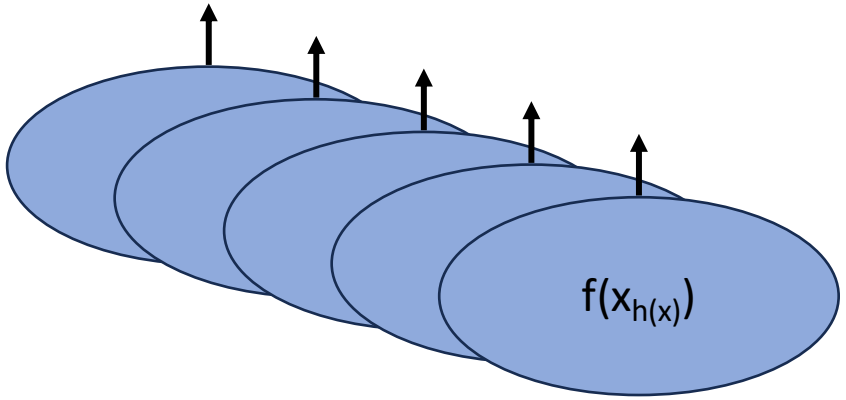
Types of Parallelism



Any sequential program,
e.g., a relational operator



Pipelining



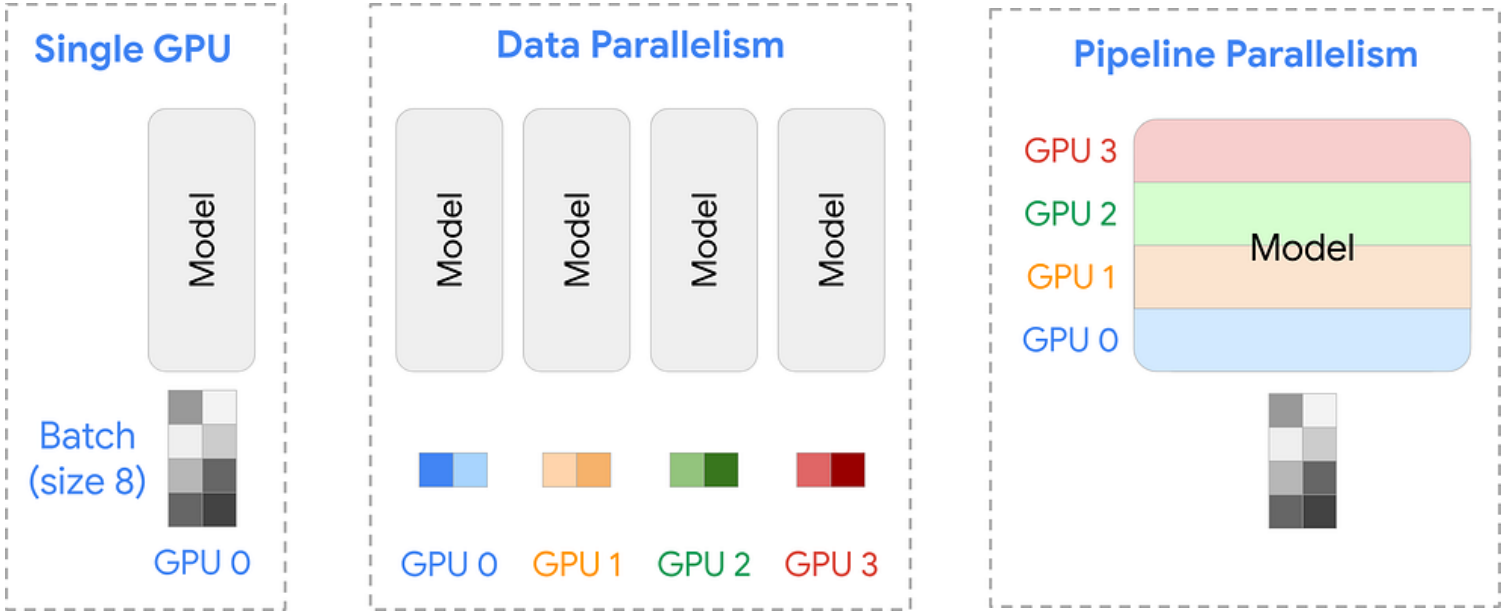
Partitioning

- **Pipelining:** each worker does one component of the calculation, then passes the result on to another worker
- **Partitioning:** each worker runs the same computations on a different set of data.

Ack: Berkeley CS186

Types of Parallelism

- **Pipelining:** each machine does one component of the calculation, then passes the result on to another machine
- **Partitioning:** each machine runs the same computations on a different set of data.



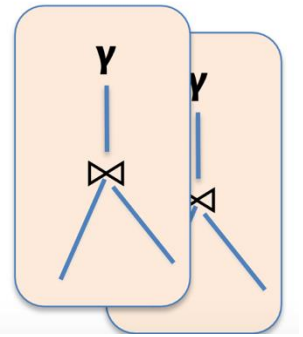
The same concepts are also used in distributed model training

Image source: https://selamjie.medium.com/the-frontier-of-pipeline-parallelism-an-overview-d4264cc9f877

Types of DBMS Query Parallelism

Inter-query parallelism

- "Inter": parallelism across queries
- Each query runs on a separate processor
  - Single thread (no parallelism per query)
- Requires concurrency control mechanism



Intra-query parallelism

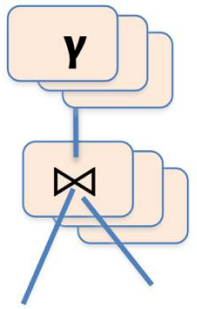
- "Intra": parallelism within a query
- a single query is broken dup and executed in parallel

Ack: Slide by Prof. Dan Suciu

Intra-query Parallelism

Intra-operator parallelism

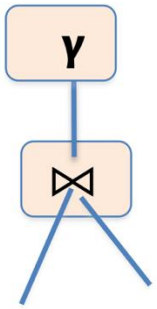
- Get all workers working to compute a given operation (scan, sort, join)
- Achieved via **partitioning**



*Our focus

Inter-operator parallelism

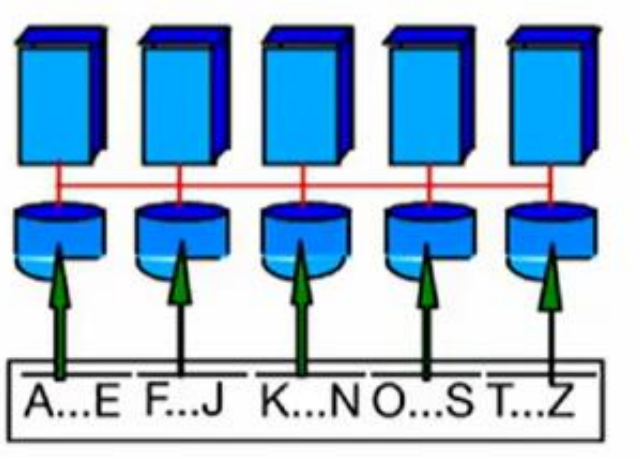
- each operator may run concurrently on a different site
- Achieved via **pipelining**



Ack: Slide by Prof. Dan Suciu

Common Data Partitioning Schemes

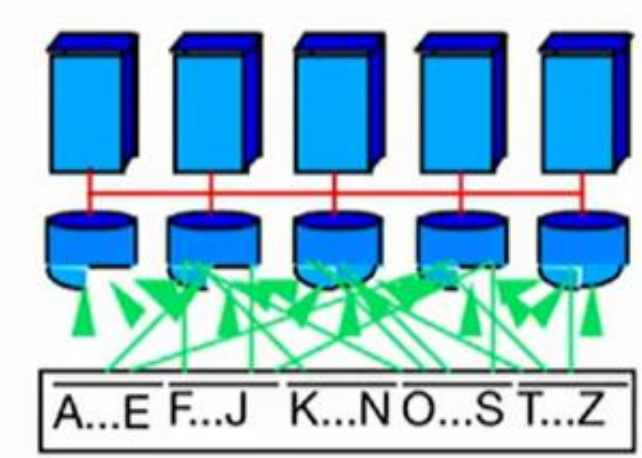
Range



Good for:

- Point look up
- Range queries
- Parallel SMJ

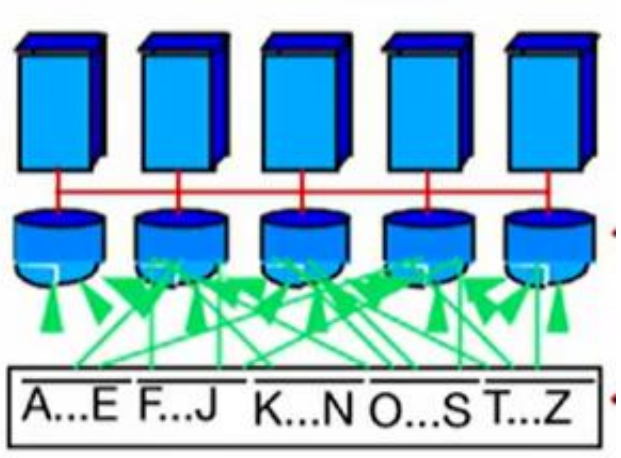
Hash



Good for:

- Point look up (but not for range queries)
- Parallel HJ

Round-Robin



Good for:

- Spreading the load
- When the entire relation is accessed

Shared disk and memory less sensitive to partitioning,
Shared nothing benefits from "good" partitioning

Ack: Berkeley CS186

In-class Exercise

```
SELECT *
FROM   students
WHERE  name = 'Jane Doe'
```

Assume that we have 5 machines and a 1000 page **students(sid, name, gpa)**table. Assume pages are 1KB.

- How many IOs will it take to execute the above query under round-robin partitioning?
- Suppose that we hash partition on the **name** column instead. How many IOs will the query take?
- Assume that an IO takes 1ms and the network cost is negligible. How long will the query take if the data is round-robin partitioned and if the data is hash partitioned on the **name** column.

Parallel Algorithms - Sorting

A simple idea:

- Have each CPU sort the part of the relation that is on its local disk
- Merge the sorted results Performance bottleneck

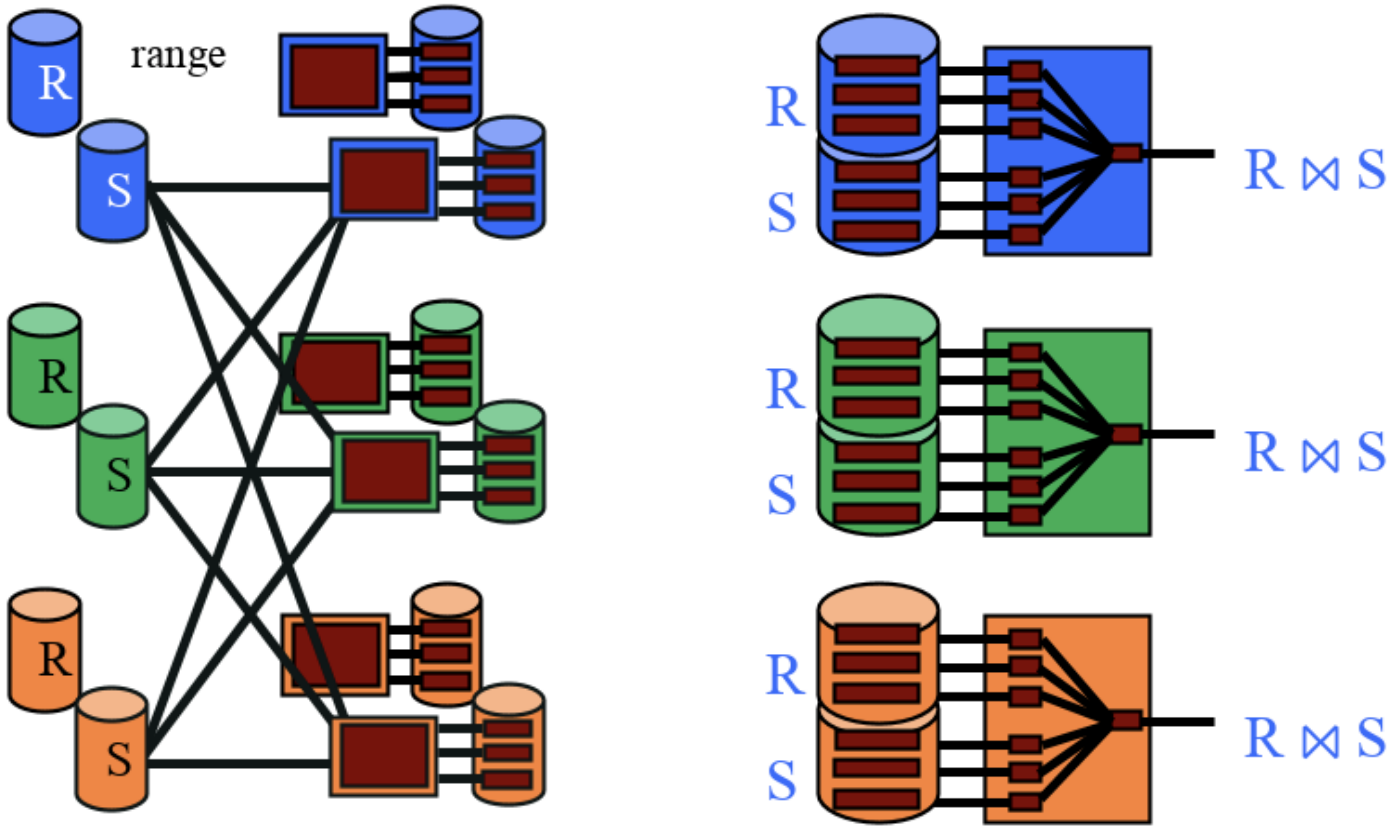
A better idea:

- Redistribute relation using range partitioning 1 pass
- Perform local sort on each machine

Parallel Algorithms – Sort Merge Join

Two Steps:

- Range partition each table **using the same ranges** on the join columr
- Perform local sort merge join on each machine

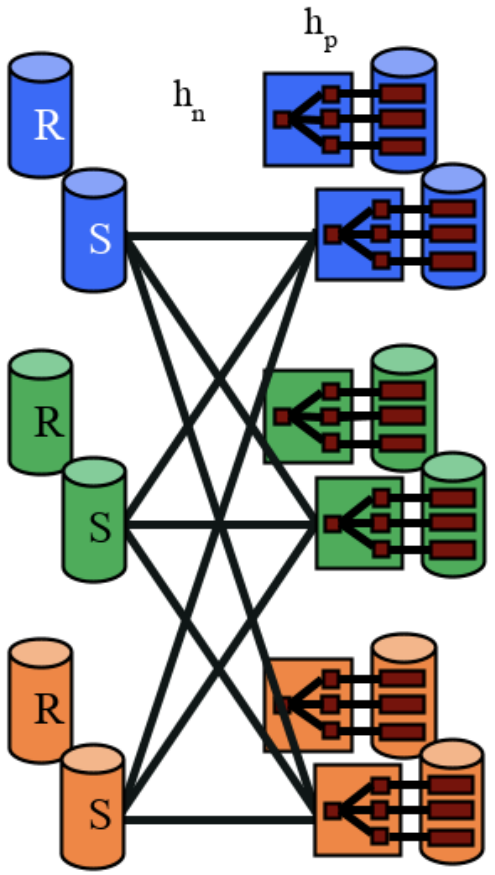


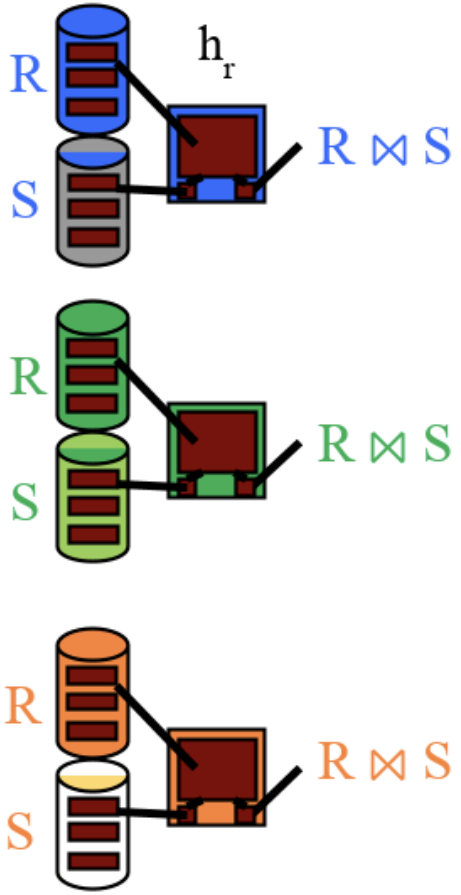
Ack: Figures from Berkeley CS186

Parallel Algorithms – Hash Join

Two steps

- Hash partition each table **using the same hash function** on the join column
- Perform local grace hash join on each machine

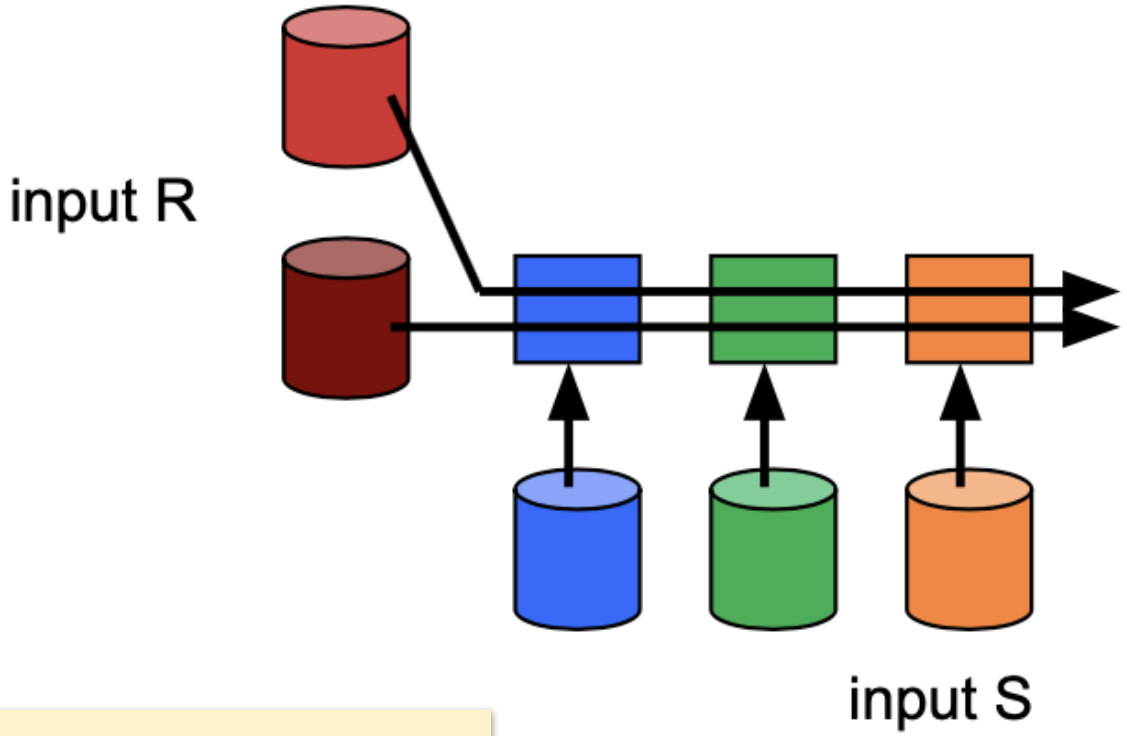




Ack: Figures from Berkeley CS186

Parallel Algorithms – Broadcast Join

- Sometimes, one join table is tiny and another table is huge.
- Too expensive to hash/range partition the large table
- "Broadcast" the small table to every machine; each machine will then perform a local join



More compute, less network

Ack: Figures from Berkeley CS186

Map Reduce vs Parallel DBMS

	MapReduce	Parallel DBMS
Programming	Imperative	Declarative
Indexing	No native support	B+ tree, hashing
Schema	Not required	Required
Flexibility	Highly flexible	Some flexibility via user defined functions
Fault Tolerance	Save intermediate results to disk – can restart fine-grained tasks during failure	Avoid saving intermediate results to disk – might need to restart a larger chunk of work (transaction) during failure

MR vs Parallel DBMS

Many commonalities:

- Designed for large-scale data processing
- Use data partitioning

Reading: A Comparison of Approaches to Large-Scale Data Analysis

A Comparison of Approaches to Large-Scale Data Analysis

Andrew Pavlo
Brown University
pavlo@cs.brown.edu

Erik Paulson
University of Wisconsin
epaulson@cs.wisc.edu

Alexander Rasin
Brown University
alexr@cs.brown.edu

Daniel J. Abadi
Yale University
dna@cs.yale.edu

David J. DeWitt
Microsoft Inc.
dewitt@microsoft.com

Samuel Madden
M.I.T. CSAIL
madden@csail.mit.edu

Michael Stonebraker
M.I.T. CSAIL
stonebraker@csail.mit.edu

ABSTRACT

There is currently considerable enthusiasm around the MapReduce (MR) paradigm for large-scale data analysis [17]. Although the basic control flow of this framework has existed in parallel SQL database management systems (DBMS) for over 20 years, some have called MR a dramatically new computing model [8, 17]. In this paper, we describe and compare both paradigms. Furthermore, we evaluate both kinds of systems in terms of performance and development complexity. To this end, we define a benchmark consisting of a collection of tasks that we have run on an open source version of MR as well as on two parallel DBMSs. For each task, we measure each system's performance for various degrees of parallelism on a cluster of 100 nodes. Our results reveal some interesting trade-offs. Although the process to load data into and tune the execution of parallel DBMSs took much longer than the MR system, the observed performance of these DBMSs was strikingly better. We speculate about the causes of the dramatic performance difference and consider implementation concepts that future systems should take from both kinds of architectures.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Parallel databases*

General Terms

Database Applications, Use Cases, Database Programming

1. INTRODUCTION

Recently the trade press has been filled with news of the revolution of "cluster computing". This paradigm entails harnessing large numbers of (low-end) processors working in parallel to solve a computing problem. In effect, this suggests constructing a data center by lining up a large number of low-end servers instead of deploying a smaller set of high-end servers. With this rise of interest in clusters has come a proliferation of tools for programming them. One of the earliest and best known such tools in MapReduce (MR) [8]. MapReduce is attractive because it provides a simple model through which users can express relatively sophisticated distributed programs, leading to significant interest in the educational community. For example, IBM and Google have announced plans to make a 1000 processor MapReduce cluster available to teach students distributed programming.

Given this interest in MapReduce, it is natural to ask "Why not use a parallel DBMS instead?" Parallel database systems (which all share a common architectural design) have been commercially available for nearly two decades, and there are now about a dozen in the marketplace, including Teradata, Aster Data, Netezza, DATAllegro (and therefore soon Microsoft SQL Server via Project Madison), Dataupia, Vertica, ParAccel, Neoview, Greenplum, DB2 (via the Database Partitioning Feature), and Oracle (via Exadata). They are robust, high performance computing platforms. Like MapReduce, they provide a high-level programming environment and parallelize readily. Though it may seem that MR and parallel databases target different audiences, it is in fact possible to write almost any parallel processing task as either a set of database queries (possibly using user defined functions and aggregates to filter and combine data) or a set of MapReduce jobs. Inspired by this question, our goal is to understand the differences between the MapReduce approach to performing large-scale data analysis and the approach taken by parallel database systems. The two classes of systems make different choices in several key areas. For example, all DBMSs require that data conform to a well-defined schema, whereas MR permits data to be in any arbitrary format. Other differences also include how each system provides indexing and compression optimizations, programming models, the way in which data is distributed, and query execution strategies.

The purpose of this paper is to consider these choices, and the trade-offs that they entail. We begin in Section 2 with a brief review of the two alternative classes of systems, followed by a discussion in Section 3 of the architectural trade-offs. Then, in Section 4 we present our benchmark consisting of a variety of tasks, one taken from the MR paper [8], and the rest a collection of more demanding tasks. In addition, we present the results of running the benchmark on a 100-node cluster to execute each task. We tested the publicly available open-source version of MapReduce, Hadoop [1], against two parallel SQL DBMSs, Vertica [3] and a second system from a major relational vendor. We also present results on the time each system took to load the test data and report informally on the procedures needed to set up and tune the software for each task.

In general, the SQL DBMSs were significantly faster and required less code to implement each task, but took longer to tune and load the data. Hence, we conclude with a discussion on the reasons for the differences between the approaches and provide suggestions on the best practices for any large-scale data analysis engine.

Some readers may feel that experiments conducted using 100

SIGMOD'09, June 29–July 2, 2009, Providence, Rhode Island, USA.