CS 6400 A

Database Systems Concepts and Design

Lecture 23

11/18/26

Agenda

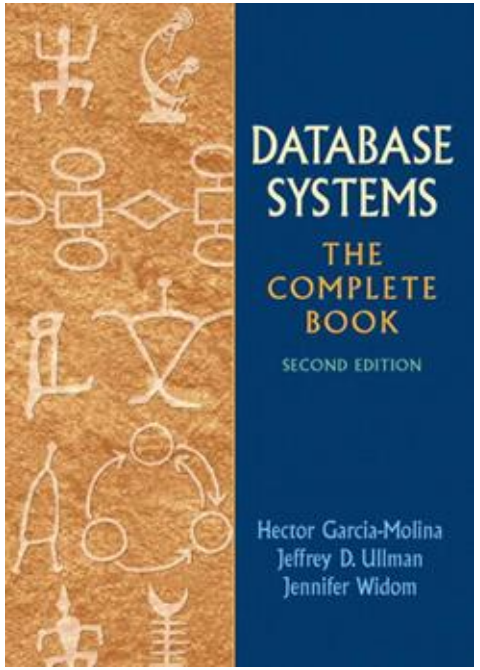
1. Spark

2. Parallel DBMS

Reading Materials

Database Systems: The Complete Book (2nd edition)

- Chapter 20: Parallel and Distributed Databases



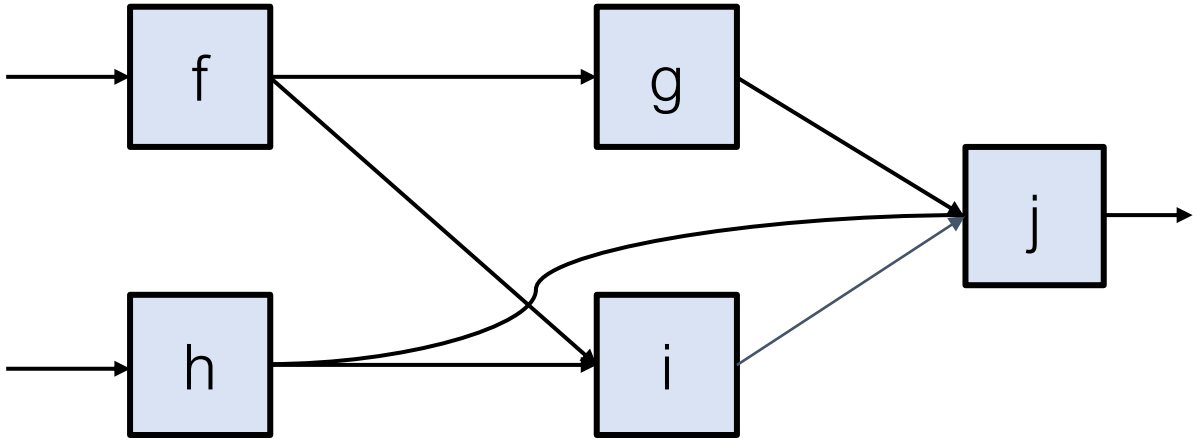
Research Papers:

- Spark

1. Spark

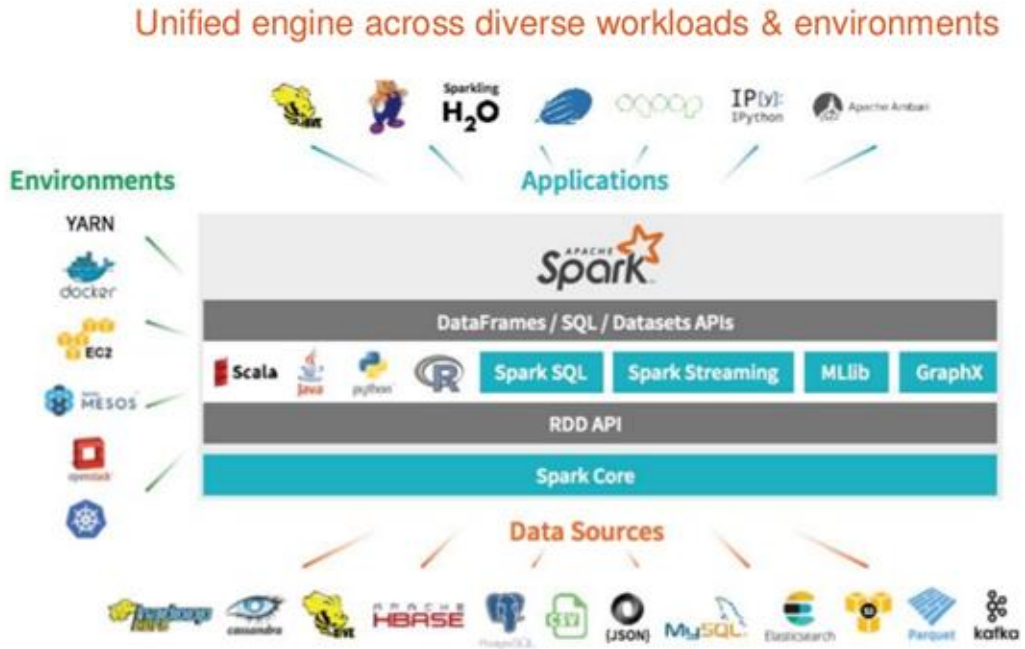
Workflow systems

- 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



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



Coordinating Distributed Execution

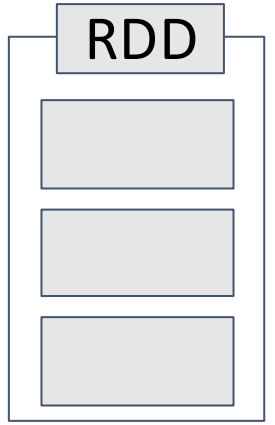
- **Master/Driver:** Divides a job into tasks, assigns them to workers, and tracks their progress.
- **Workers/Executors:** Execute tasks on partitions of the input data and report their status.
- **Fault tolerance:** When a worker fails, the coordinator reschedules lost work.

Comparisons:

- **MapReduce:** Fixed map/reduce execution structure.
- **Spark:** General DAG of transformations, divided into stages and tasks.

Data Model: Resilient distributed dataset (RDD)

Central data abstraction of Spark



A partitioned dataset spread across a cluster.

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

Immutable collections of objects, together with its lineage

- Lineage = how a dataset is computed

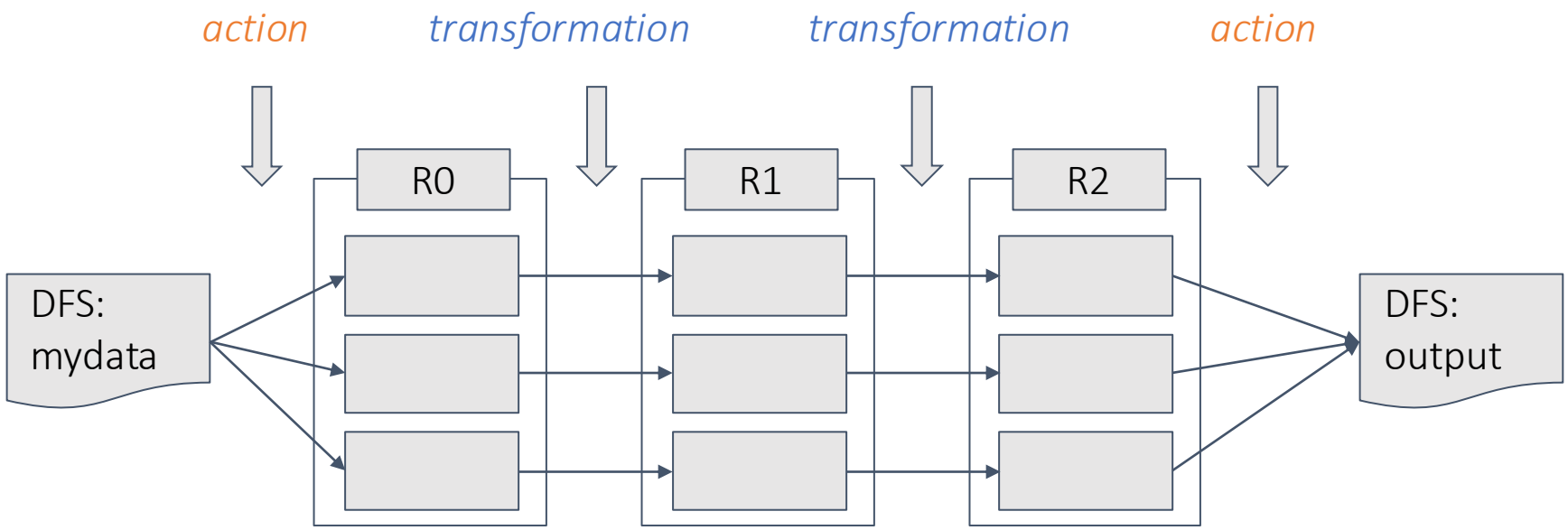
Spark is resilient against loss of any or all chunks of RDD

- 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



Common Transformations

`map`

One input → one output

*e.g., apply x * 2*



`flatMap`

One input → zero or more outputs

e.g., split each line, flatten



`filter`

Keep elements satisfying a condition

e.g., keep even numbers



Common Transformations (Relational)

`groupByKey`

Collect values with the same key

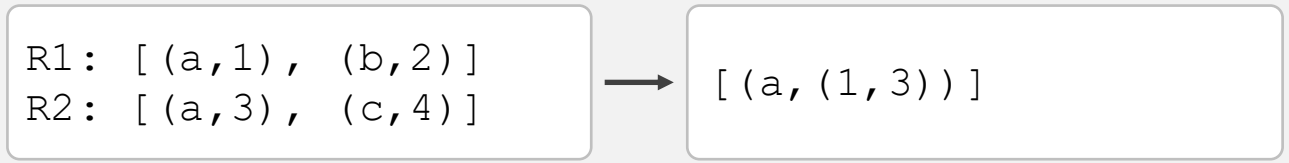
one key → list of values



`join`

Match records from two RDDs by key

only keys in both survive



Both operations commonly require data redistribution (**shuffle**).

Actions: Trigger Execution

- 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

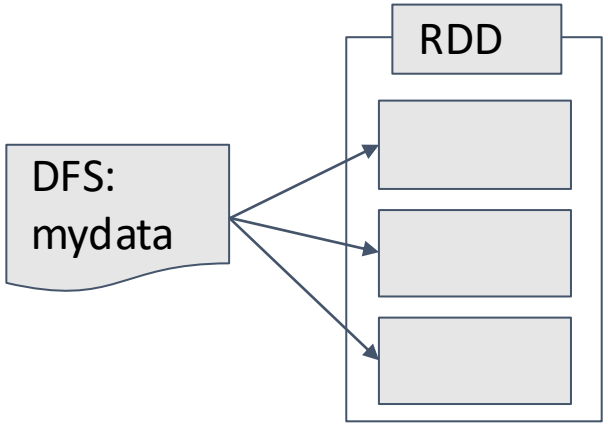
```
# Collect RDD elements to the drive program
collected_data = rdd.collect()


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


# Aggregate to one value
rdd.reduce(lambda a,b: a+b)


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

Transformations vs Actions

	Transformations	Actions
Purpose	Create a new RDD	Trigger execution
Examples	map, filter, flatMap, join	count, collect, reduce, saveAsTextFile
Execution	Lazy	Executes the required computation

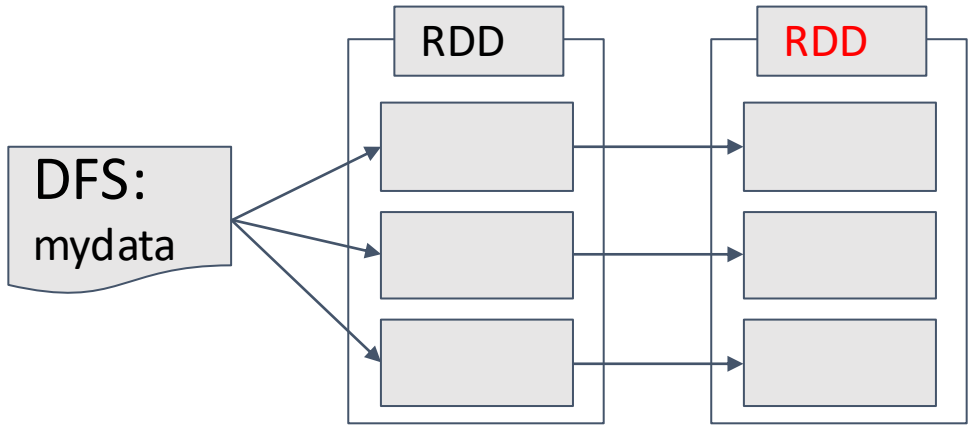
Example: average word length by letter

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



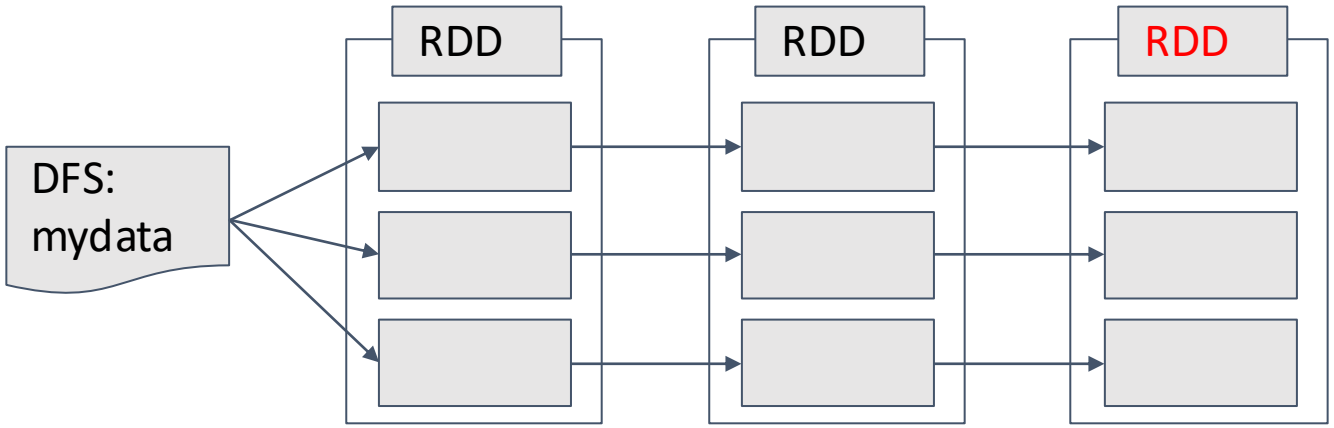
Example: average word length by letter

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



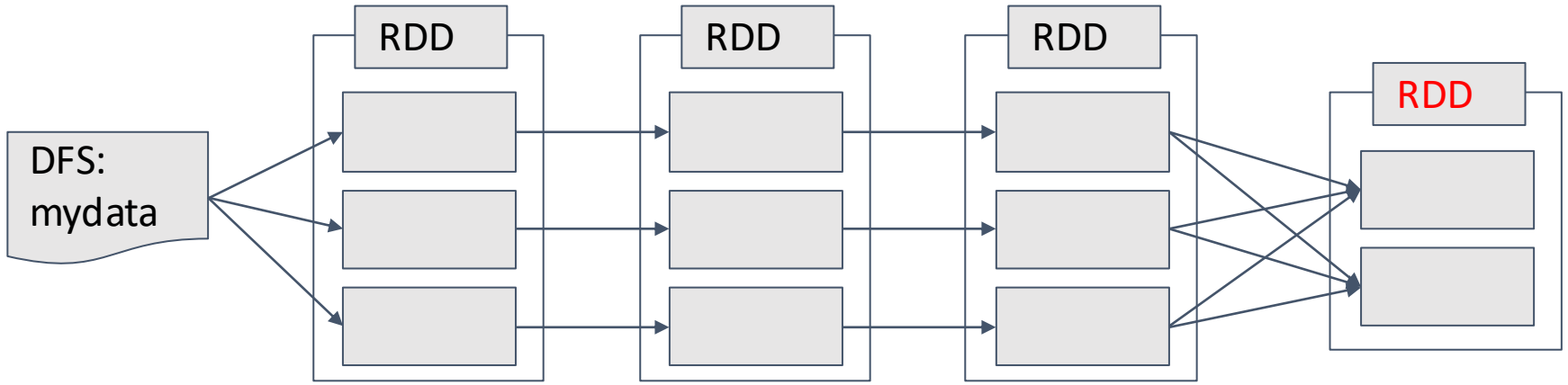
Example: average word length by letter

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



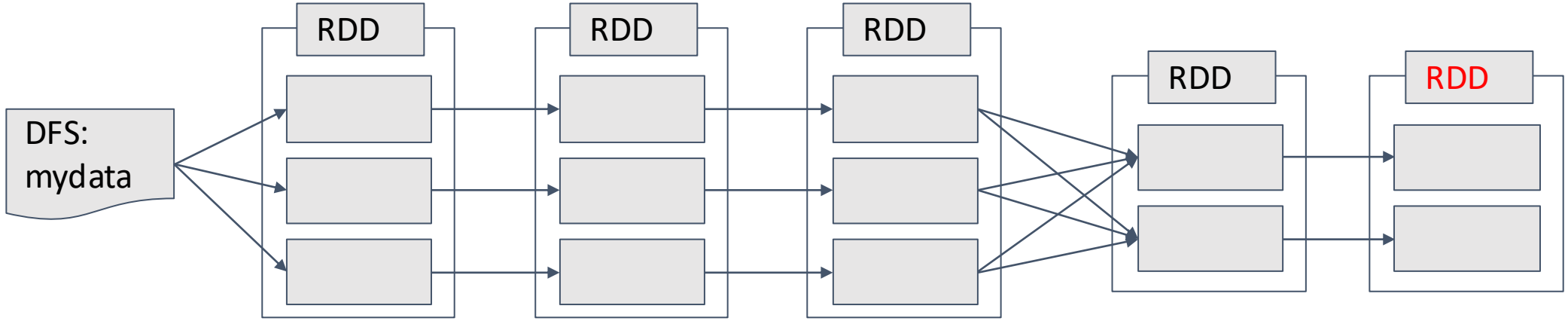
Example: average word length by letter

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



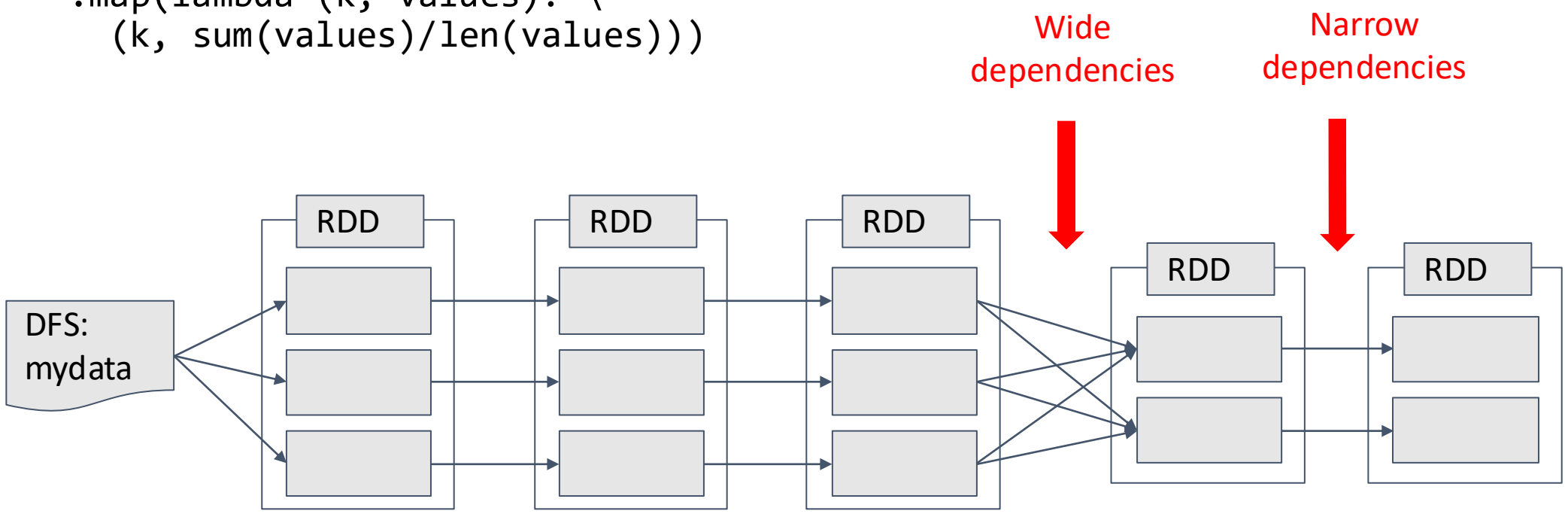
Example: average word length by letter

```
> 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)))
```

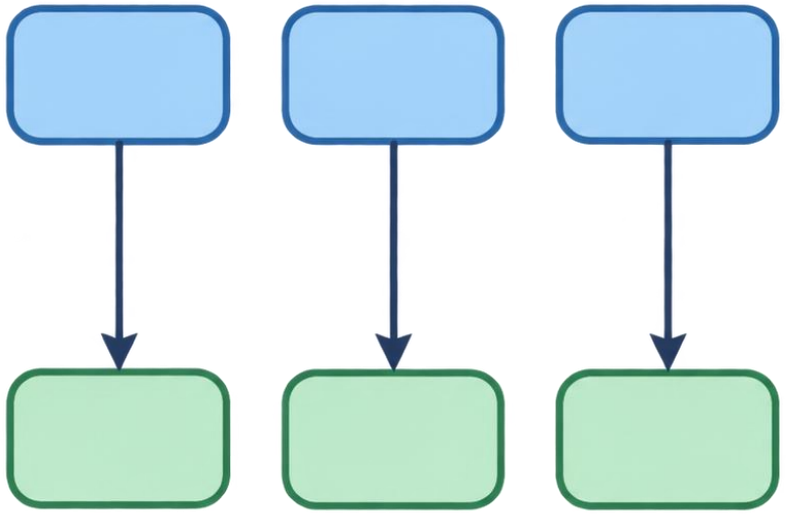


Example: average word length by letter

```
> 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)))
```

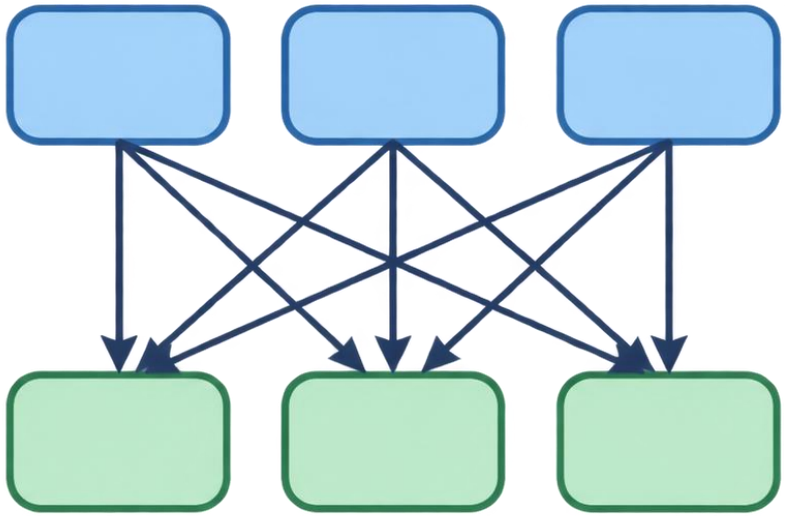


Narrow vs Wide Dependencies



Examples: map, filter

No shuffle; operations can be pipelined.

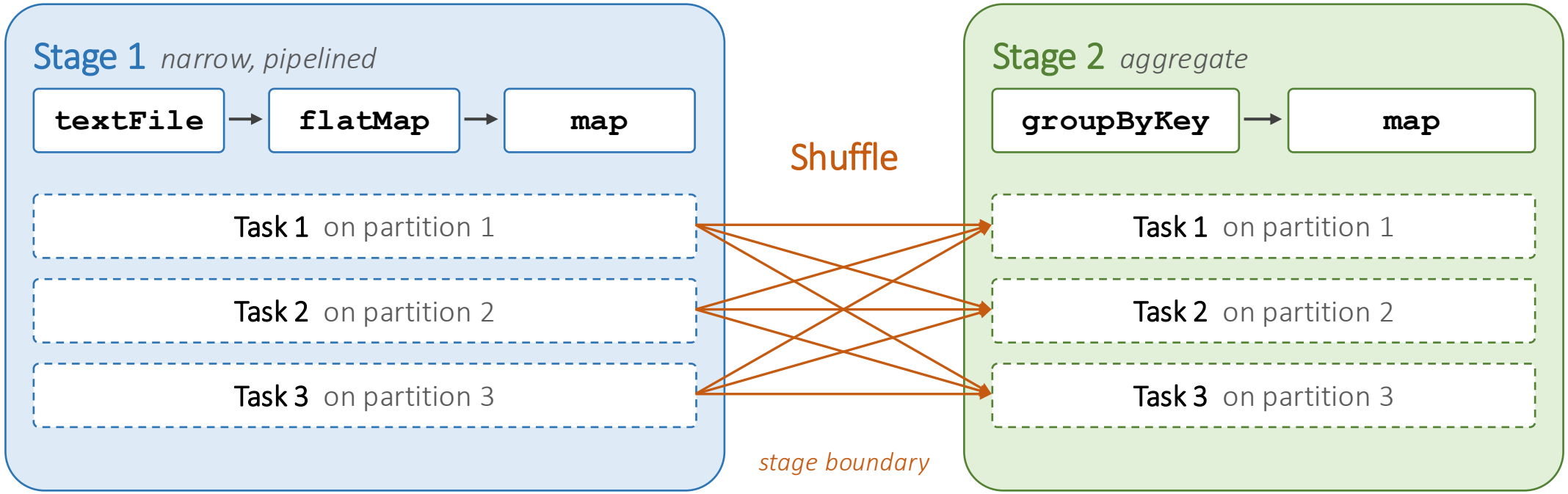


Examples: groupByKey, reduceByKey

Requires a shuffle; creates a stage boundary.

Spark uses these dependencies to determine task scheduling and fault recovery.

How Spark executes a DAG



- Spark groups pipelinable transformations into stages.
- A shuffle typically separates stages.
- Each stage contains parallel tasks, usually one per partition.

Blocking and Fault Tolerance

- **Blocking execution:** A downstream stage waits until the required upstream stage outputs are available.
- **Fault tolerance:** Intermediate outputs provide recovery boundaries. If a task fails, the system can rerun the failed work without restarting the entire computation.

Comparisons:

- **MapReduce:** Map and reduce phases are separated by a shuffle boundary.
- **Spark:** Wide dependencies introduce shuffle boundaries; narrow transformations can be pipelined.



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

When does Spark actually compute?

```
rdd = sc.textFile("data.txt")

words = rdd.flatMap(lambda line: line.split())

long_words = words.filter(lambda w: len(w) > 5)

long_words.count()
```

Q: After which line does Spark start processing the data?

Lazy evaluation

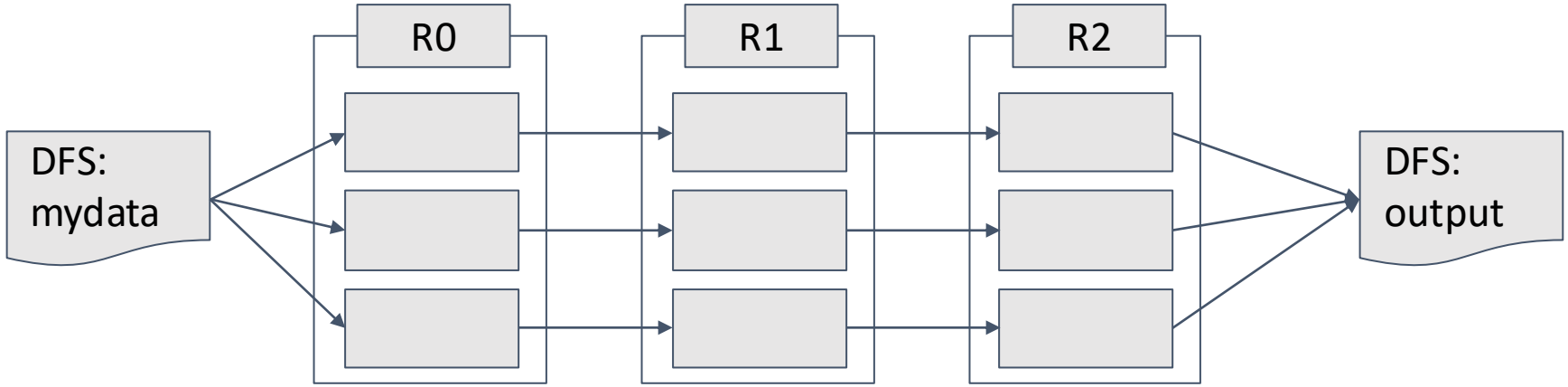
Spark does not actually apply transformations to RDD's until it is required to do so (e.g., storing RDD to file system or returning a result to application)

Potential Benefits:

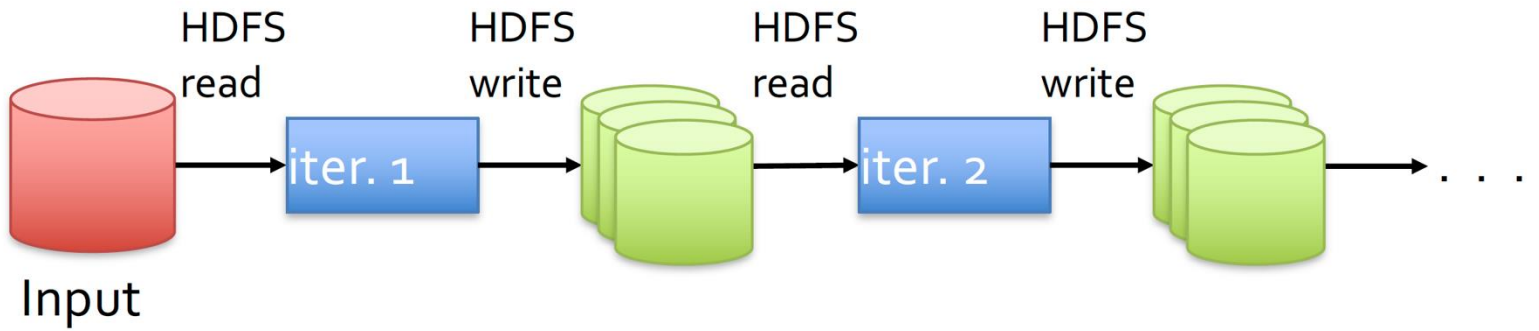
- Spark can analyze entire chain of operations and combining multiple operations to reduce unnecessary computations
- No immediate computation/memory usage; resources allocated only when needed
- Optimizes data shuffling and stages

Resilience of RDD's

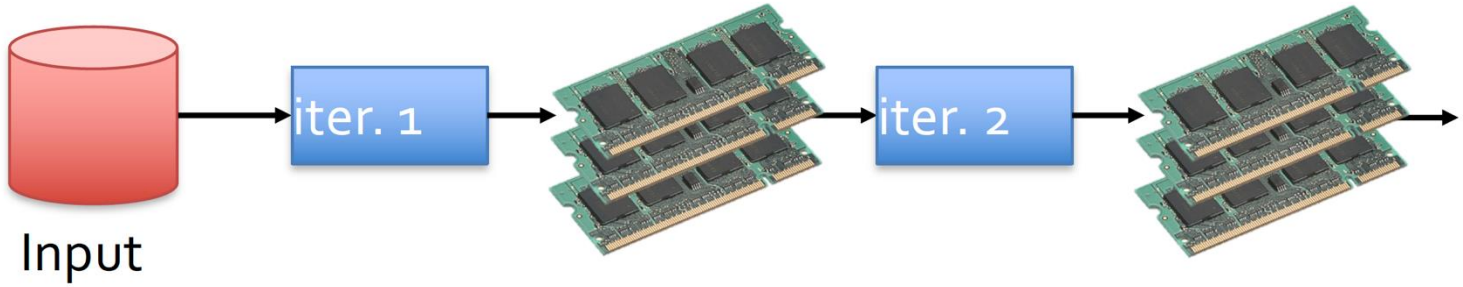
- Spark records the *lineage* of every RDD, which can be used to re-create any RDD
  - If R_2 is lost, reconstruct from R_1
  - If R_1 is lost, reconstruct from R_0
  - If R_0 is lost, reconstruct from file system



Data Sharing in MapReduce vs Spark



MapReduce: disk



Spark: memory

Spark avoids repeatedly writing intermediate datasets to distributed storage and can cache reused data in memory.

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

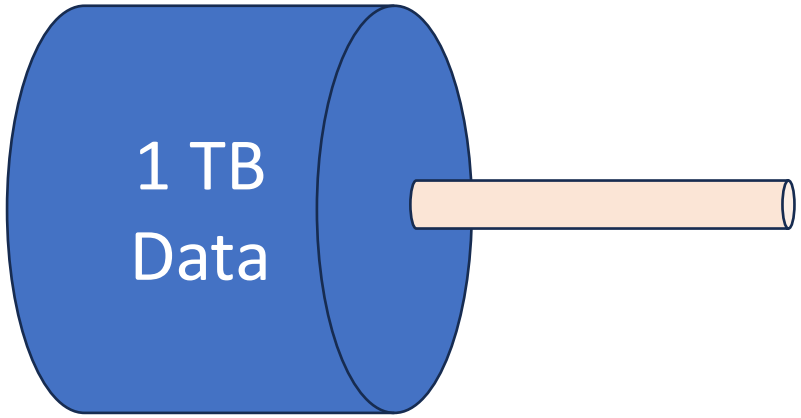
Parallel DBMS

What it is: A DBMS that runs database operations across many processors at once.

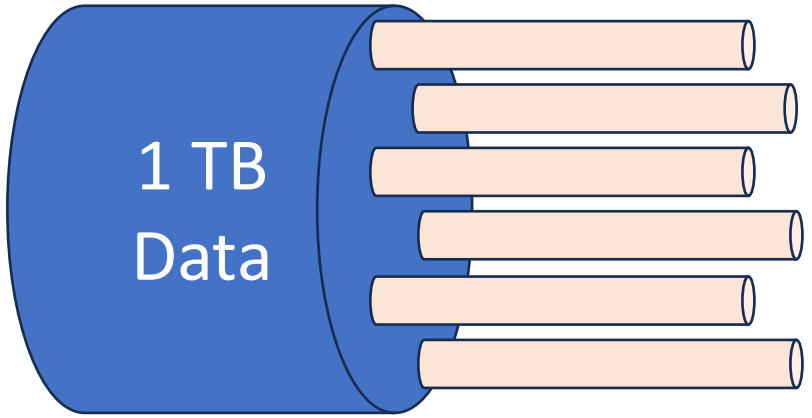
- **Goal:** Speed up query execution by using multiple processors.
- **Deployment:** Can run on a multicore machine or a cluster of machines.
- **Challenge:** Partition work and data, coordinate execution, and minimize communication overhead.

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

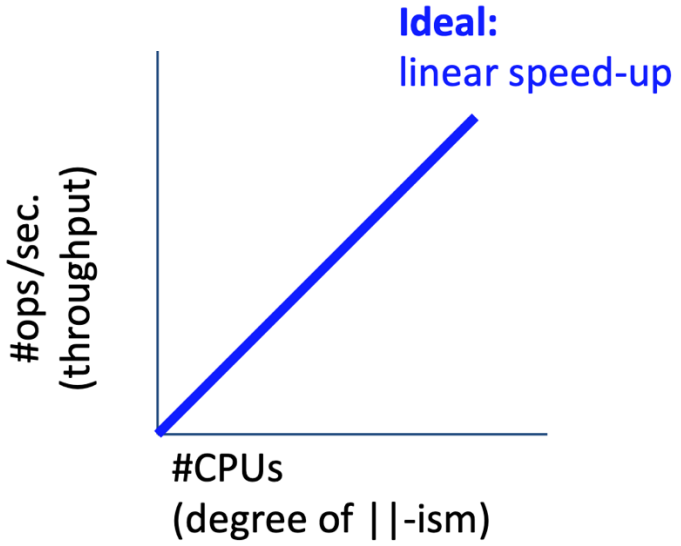


1,000 x parallel
1.5 minute to scan

Key Metrics

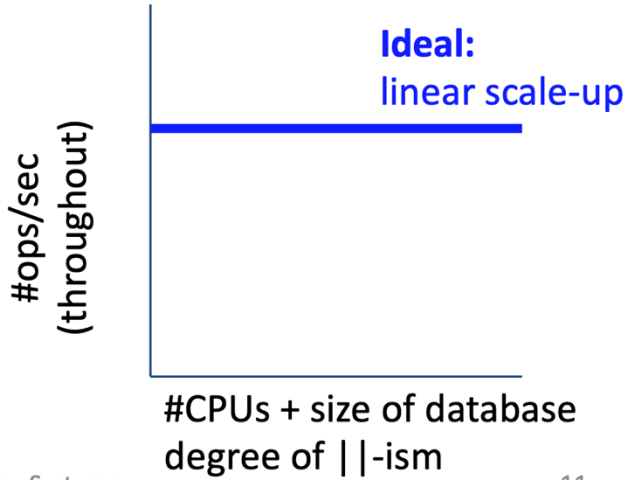
Speed-up

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



Scale-up

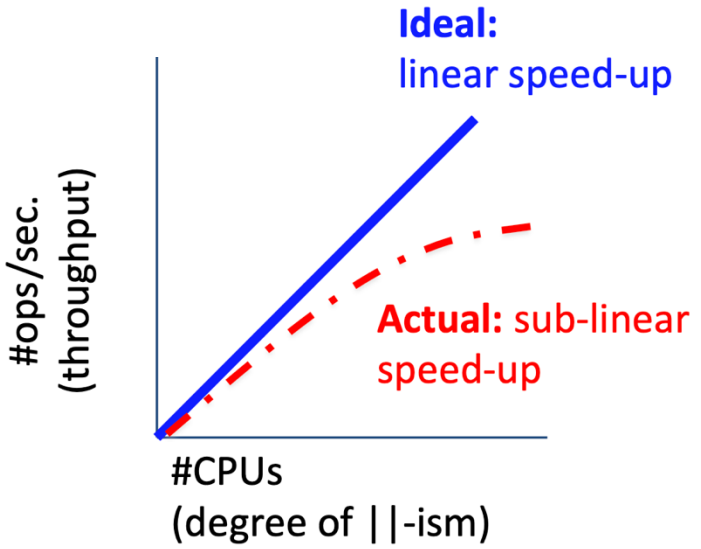
- Increase HW, increase workload
- If resources increased in proportion to increase in data size, time is constant.

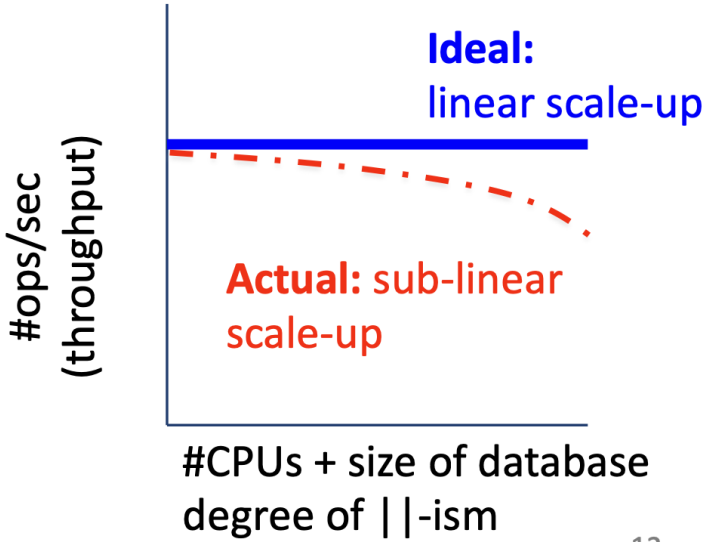


Key Metrics

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





Parallel Architectures

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

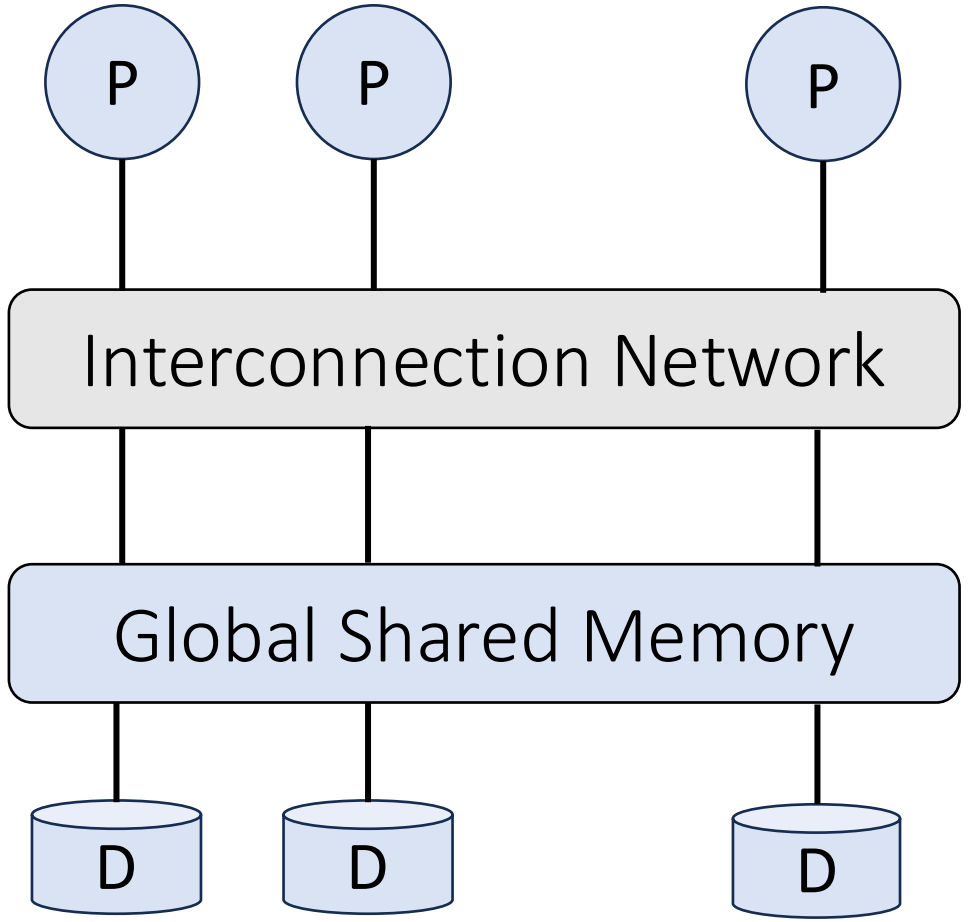
- a shared bus or a switch

Different architecture:

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

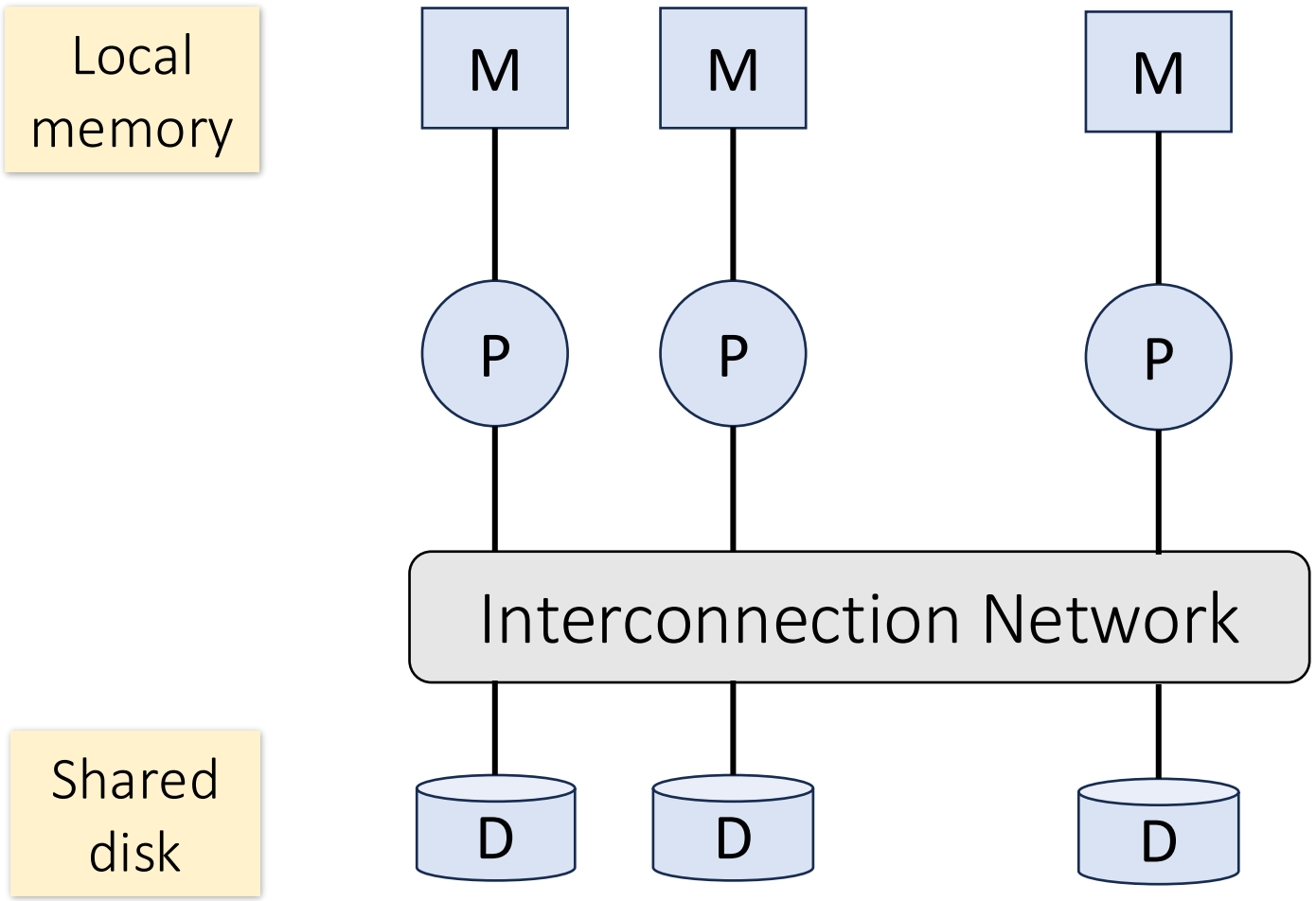
Shared-Memory Architecture

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

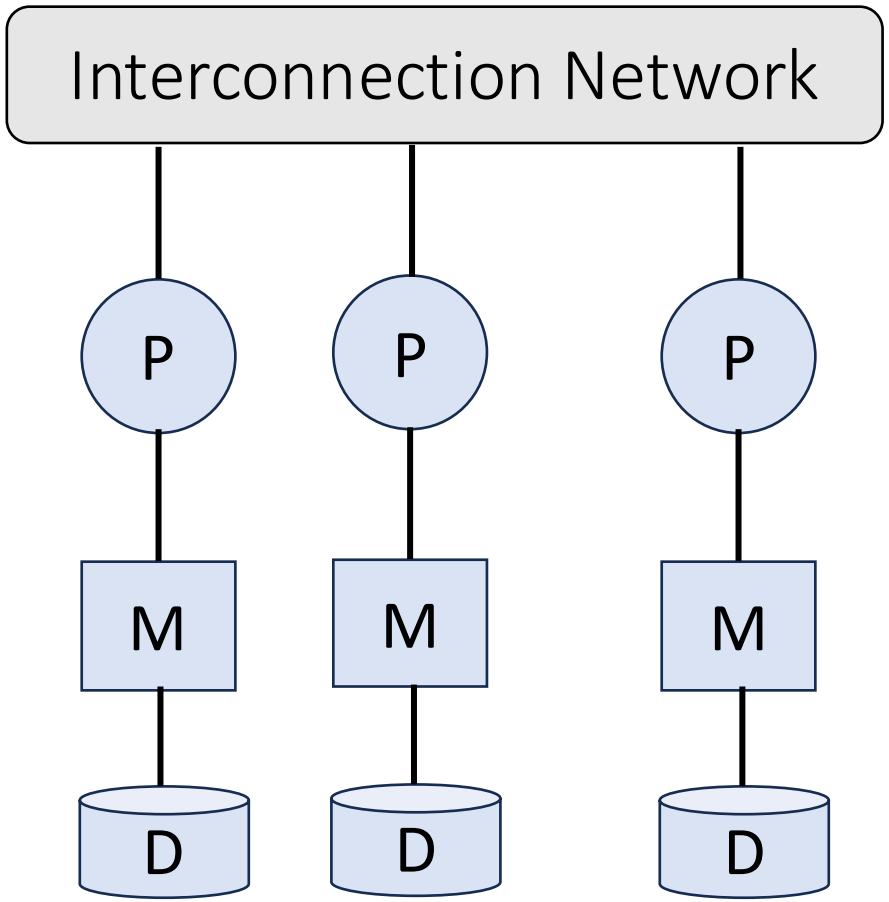
Shared-Disk Architecture



- 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

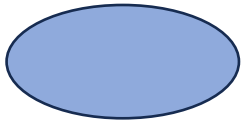


Local memory and disk

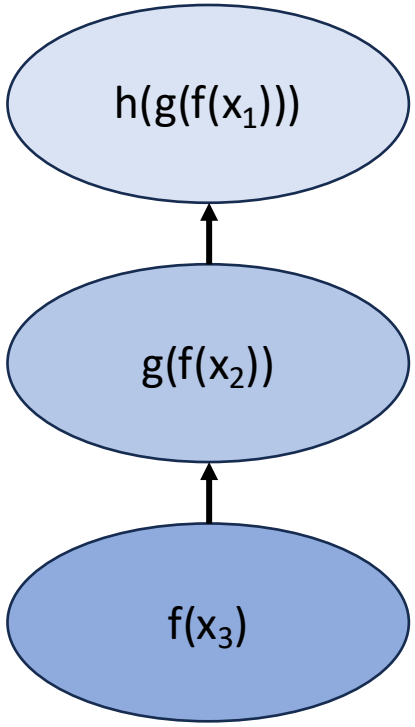
- Excellent horizontal scalability; relatively inexpensive to build
- Minimal resource contention but higher communication overhead
- Hard to program and design parallel algos

* We will assume this architecture by default

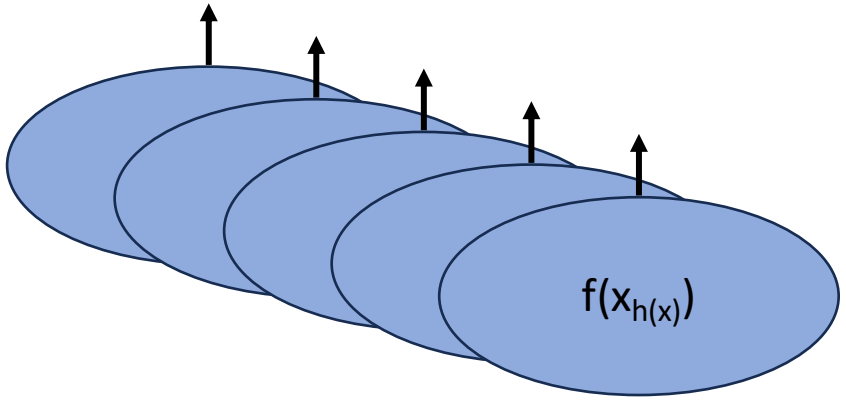
Types of Parallelism



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



Pipelining



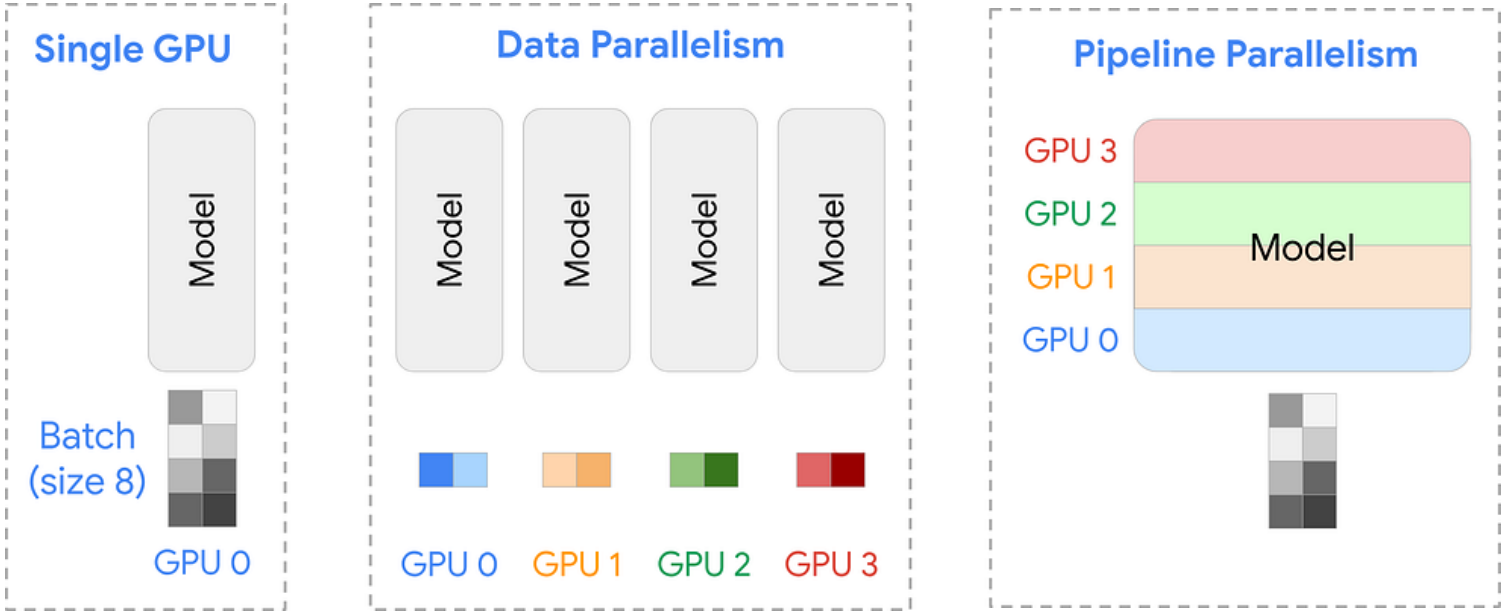
Partitioning

- **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.



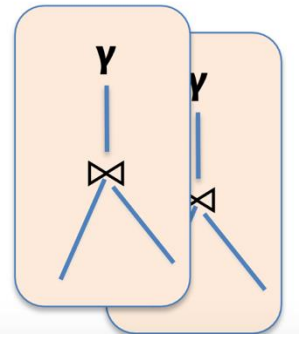
The same concepts are also used in distributed model training

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

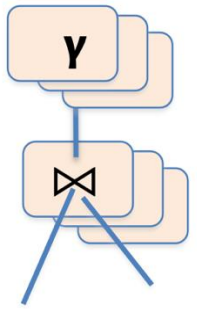
- "Intra": parallelism within a query
- a single query is broken dup and executed in parallel

Ack: Slide by Prof. Dan Suciu

Intra-query Parallelism

Intra-operator parallelism

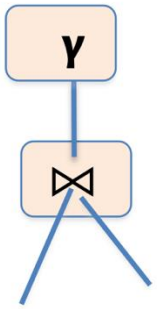
- Get all workers working to compute a given operation (scan, sort, join)
- Achieved via **partitioning**



*Our focus

Inter-operator parallelism

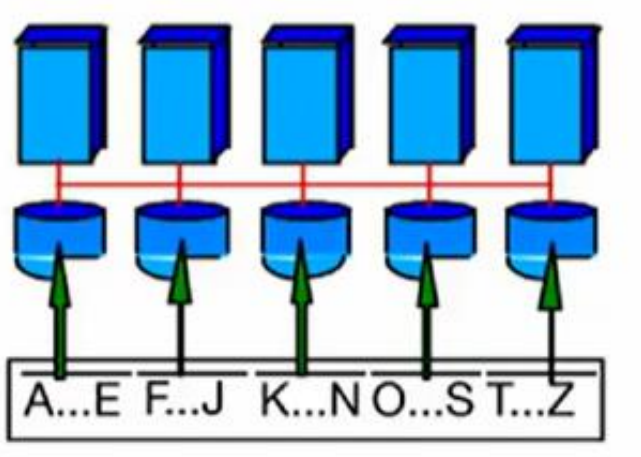
- each operator may run concurrently on a different site
- Achieved via **pipelining**



Ack: Slide by Prof. Dan Suciu

Common Data Partitioning Schemes

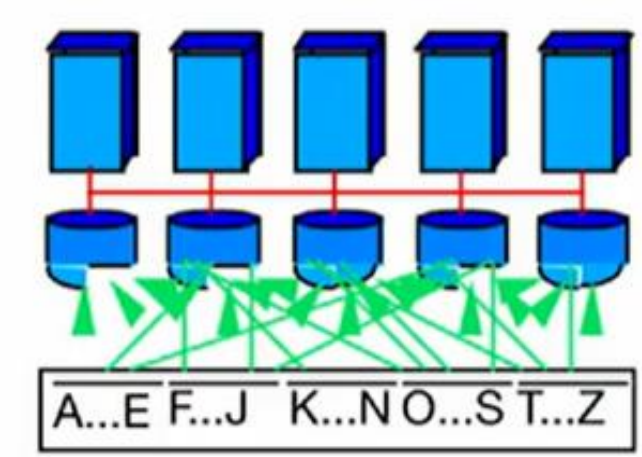
Range



Good for:

- Point look up
- Range queries
- Parallel SMJ

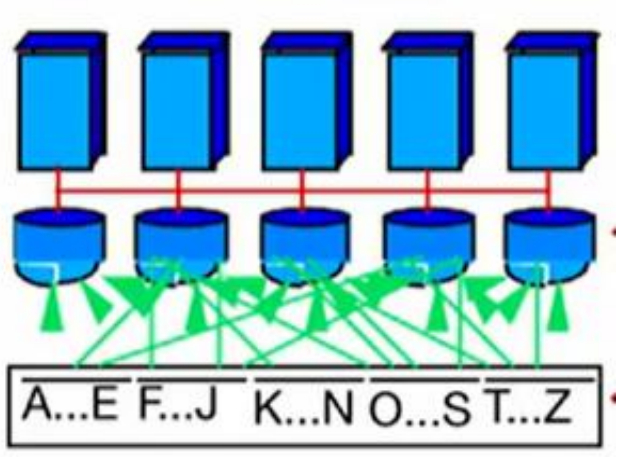
Hash



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

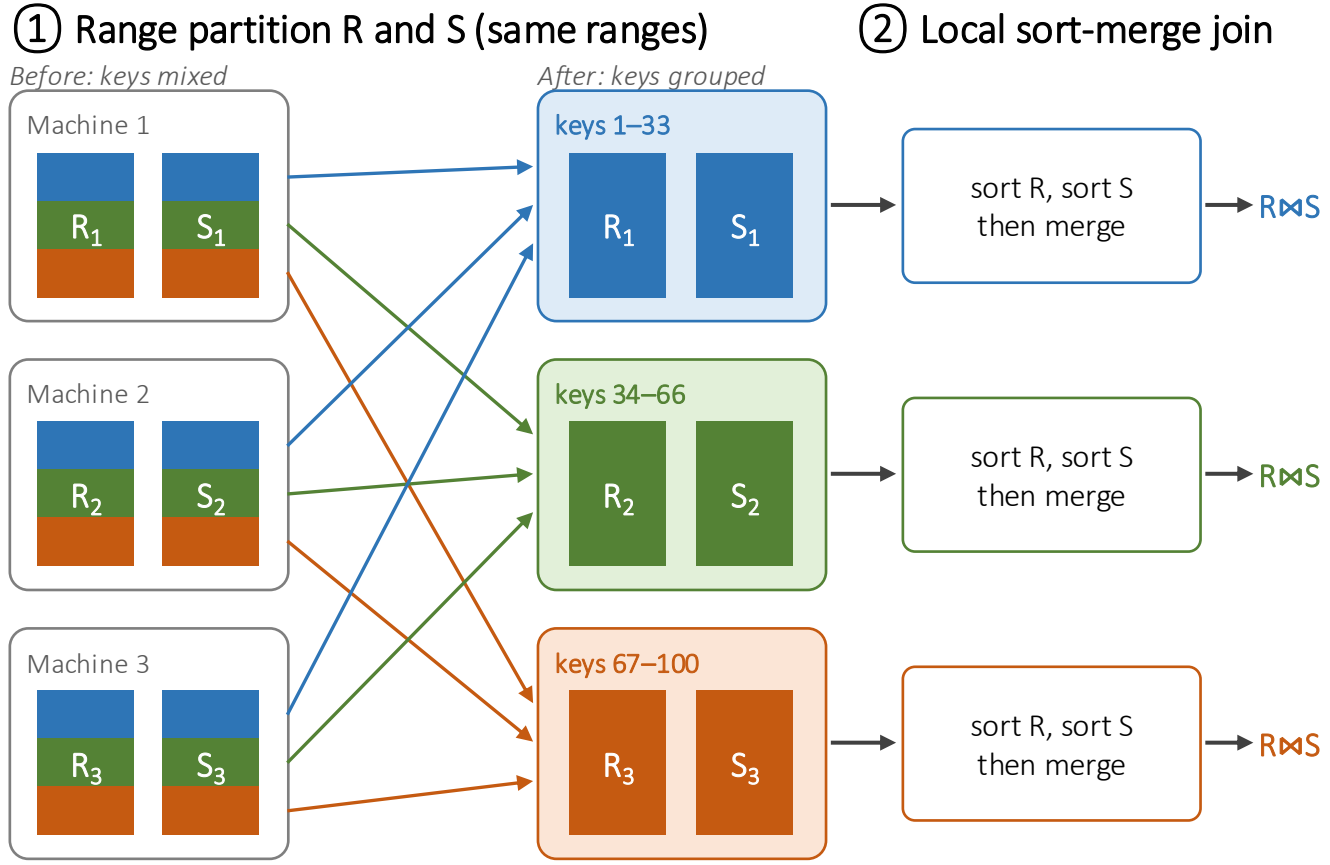
A better idea:

- 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 column
- Perform local sort merge join on each machine

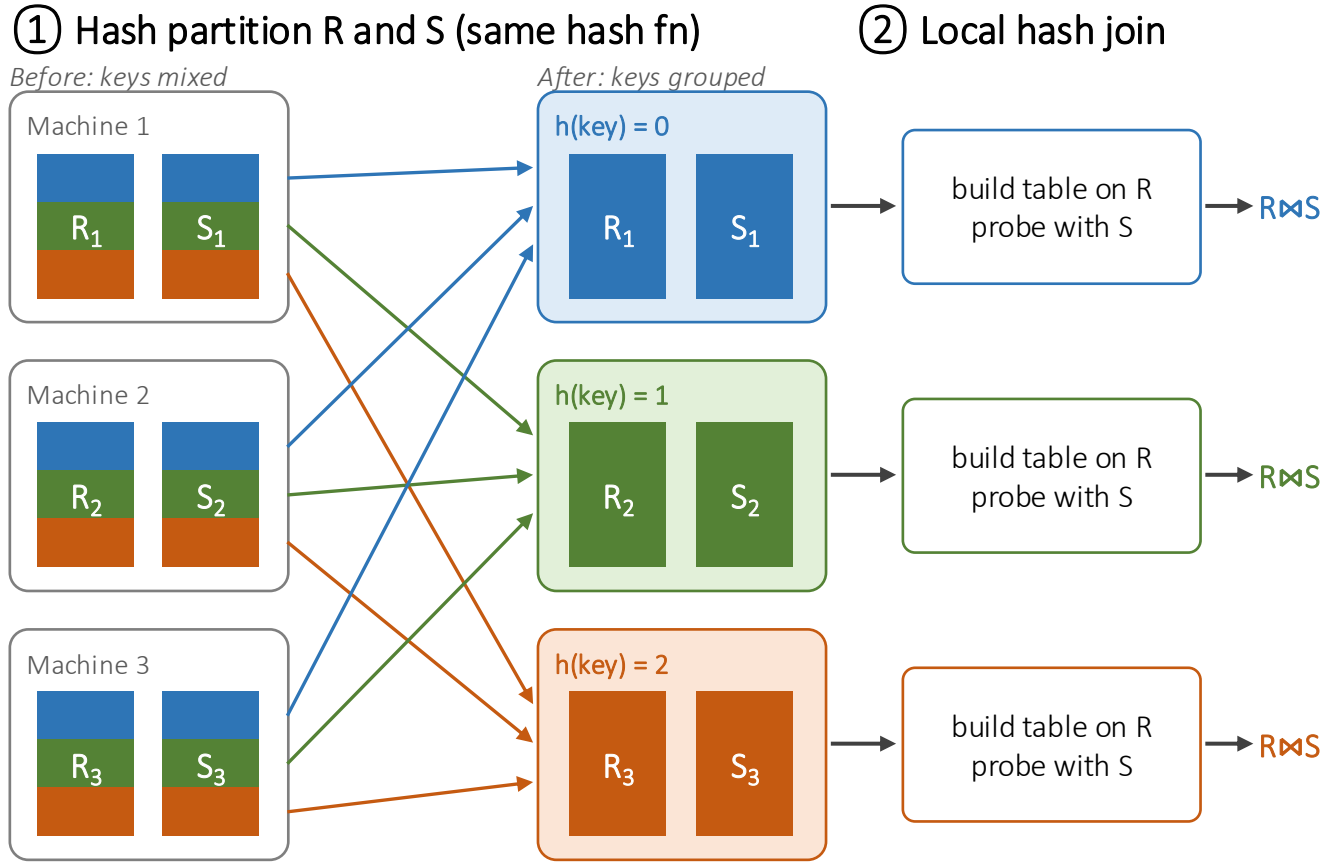


Ack: Figures adapted from Berkeley CS186

Parallel Algorithms – Hash Join

Two steps

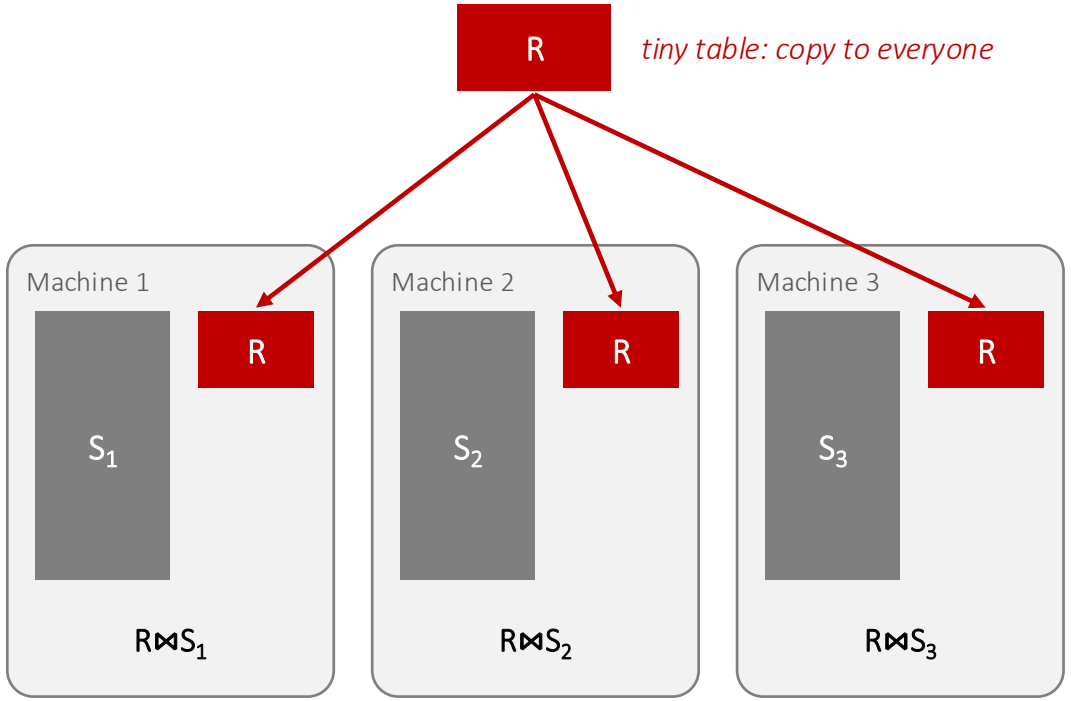
- Hash partition each table **using the same hash function** on the join column
- Perform local grace hash join on each machine



Ack: Figures adapted 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 adapted from Berkeley CS186

MapReduce vs. Parallel DBMS (Programming Model)

	MapReduce	Parallel DBMS
Interface	Map/reduce functions	Declarative SQL
Data	Files, often loosely structured	Structured tables
Optimization	Programmer specifies computation steps	Query optimizer selects execution plan

SQL describes what to compute; MapReduce programmers specify more of how to compute it.

MapReduce vs. Parallel DBMS (Execution)

	MapReduce	Parallel DBMS
Execution	Materializes outputs between jobs	Pipelines operators when possible
Fault tolerance	Re-executes failed tasks	Traditionally restarts failed queries
Strength	Flexible, fault-tolerant batch processing	Efficient relational query processing

MapReduce emphasizes robust execution across failures; parallel DBMSs emphasize efficient query execution.

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

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

Copyright 2009 ACM 978-1-60558-551-2/09/06 ...$5.00.