CS-310 Scalable Software Architectures

Lecture 19: Distributed Computing

Steve Tarzia

Last time – Computing Platforms

- **Virtual Machines** let multiple tenants share a single physical server.
  - Apps can be distributed as a **VM disk image.**
- **Containers** create a consistent environment for your application.
  - Distribute as an **image** (like a VM disk image, but more lightweight),
  - Or as source code + a **Dockerfile** (a blueprint for the image).
- **Serverless functions** are code that is *staged* in the cloud, *ready to run:*
  - They are automatically deployed and run on *one or more* VMs **on demand**.

Pros:
- Gives more dynamic & fine-grained scalability than container/VM.
- Uses as many machines as needed, when needed. (zero to 1000s!)

Cons:
- "Cold start" delay of several seconds (to copy code & launch).
- Function runtime is often limited (eg., 15 minutes for Lambda).

Easy vs. Hard scalability problems

Easy:

- Handle **independent** requests from millions of customers.
- Assume the data is already available to service the requests.
- Trivially parallelizable – use horizontal scaling to *divide and conquer*.

Hard:

- Perform a calculation using **all the data** from millions of customers.
- Eg., build a recommendation system from Amazon order history.
- Requires lots of coordination, housing data on the compute nodes.
- Must use MapReduce, Spark, etc.
- Luckily, it's OK if the calculation takes several hours.

"Big Data" problems

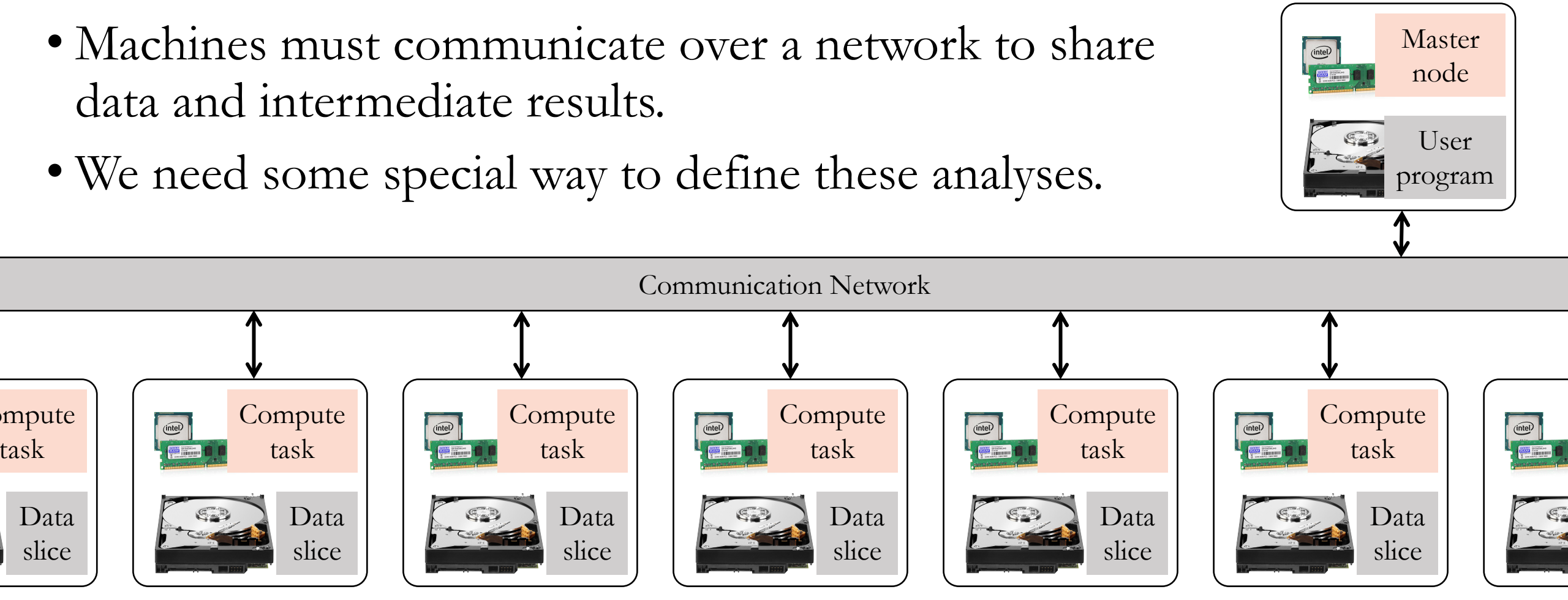
- Let's say you need to run analyses of 10 petabytes of data each day:
  - 1 Petabyte = 1024 Terabytes = 1024*1024 Gigabytes ≈ 10^{15} bytes
  - Theoretically, a single computer can do this in no less than:

    10^{16} bytes / ($80 * 10^9$ bytes/sec) = 125,00 seconds = **35 hours**.

    PCI express v4 max bandwidth
  - However, in practice you'll find the performance is even slower than this.
- To speed up an analysis you must use many machines to work on the problem in **parallel**.
  - The analysis is somehow split into pieces (subproblems)
  - Each machine works on subproblems that contribute to the final answer.
  - Subproblem results are redistributed to contribute to final solution.
  - Parallel data analysis is an active area of research in industry and academia.

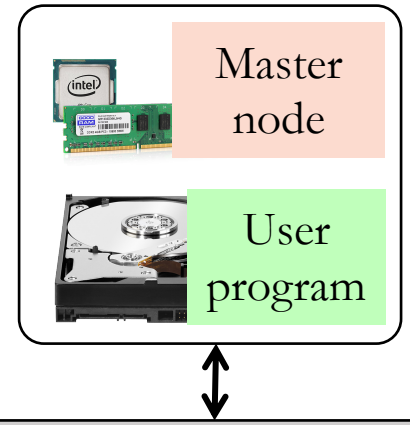
Distributed computing

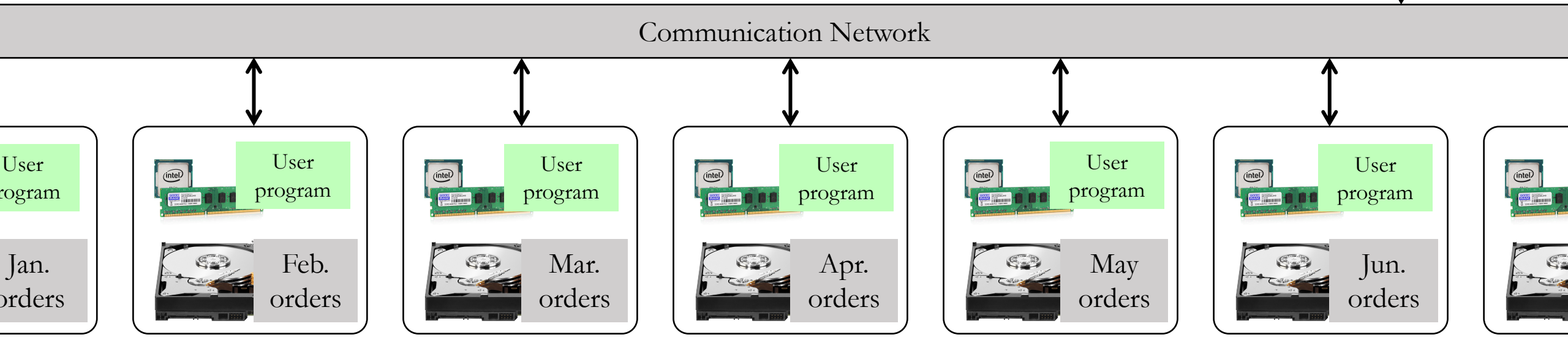
- Data & analysis are distributed across many machines.
- No one machine has *direct* access to all the data.
- Machines must communicate over a network to share data and intermediate results.
- We need some special way to define these analyses.



Example: Find the largest sales order

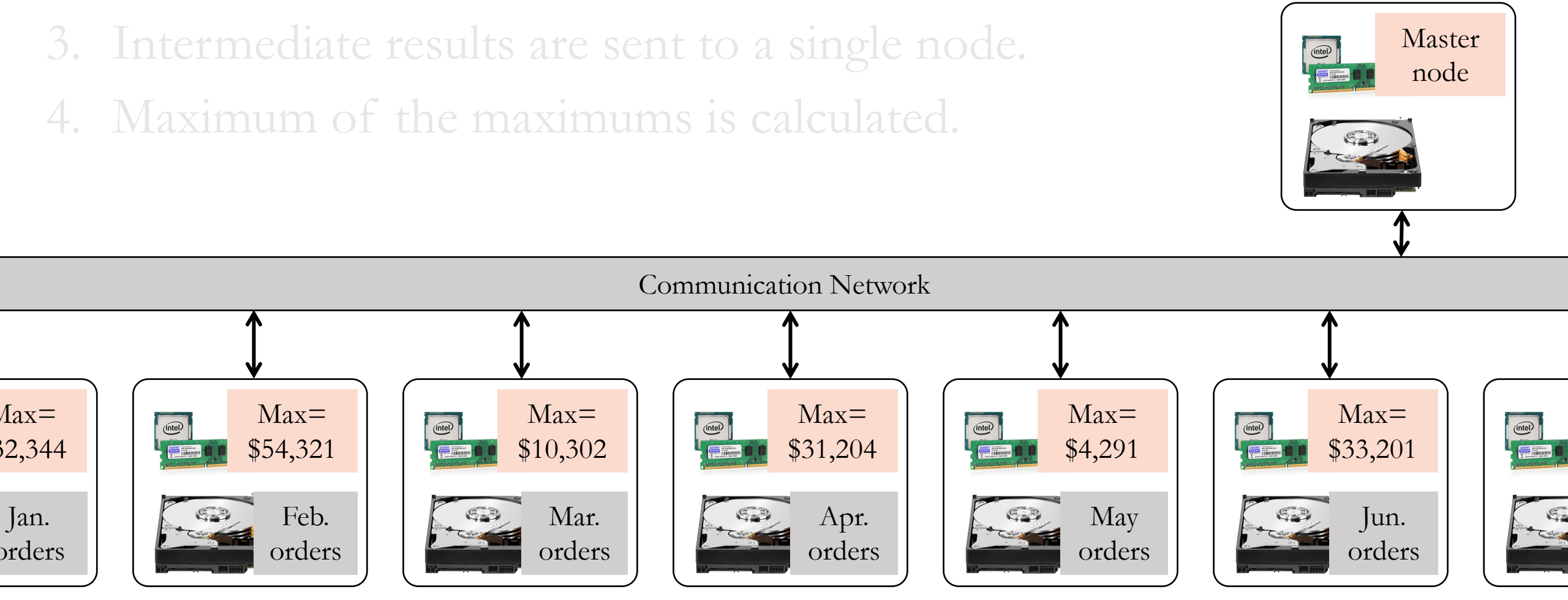
1. Distribute instructions to all the machines (nodes).
2. Each node computes its own maximum.
3. Intermediate results are sent to a single node.
4. Maximum of the maximums is calculated.





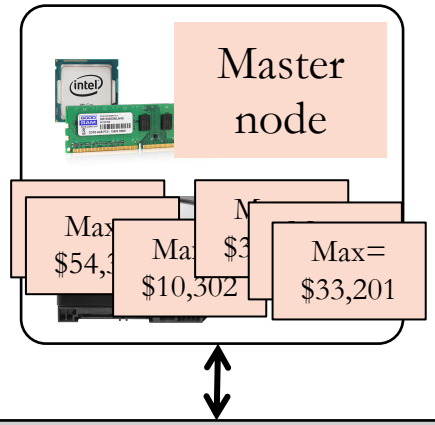
Example: Find the largest sales order

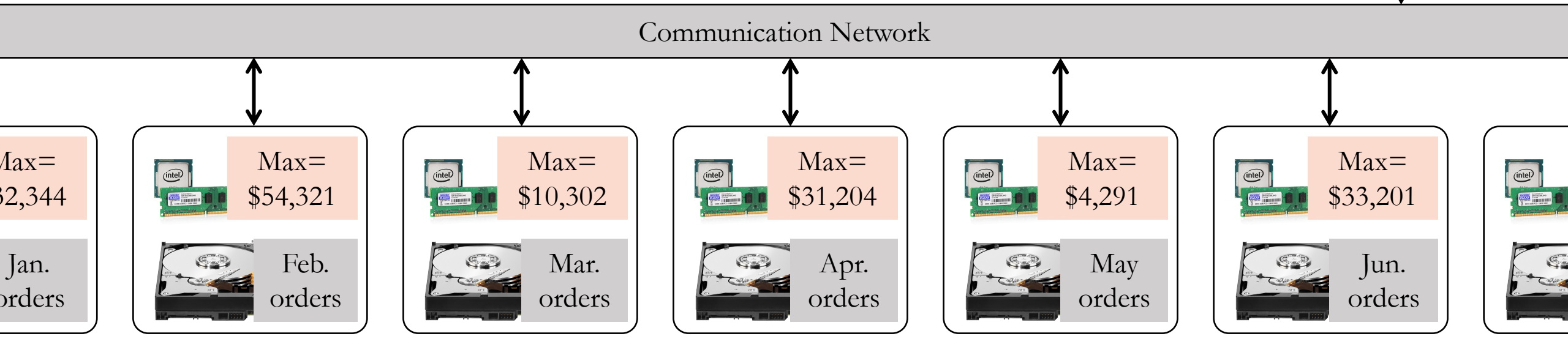
1. Distribute instructions to all the machines (nodes).
2. Each node computes its own maximum.
3. Intermediate results are sent to a single node.
4. Maximum of the maximums is calculated.



Example: Find the largest sales order

1. Distribute instructions to all the machines (nodes).
2. Each node computes its own maximum.
3. Intermediate results are sent to a single node.
4. Maximum of the maximums is calculated.





Example: Find the largest sales order

1. Distribute instructions to all the machines (nodes).
2. Each node computes its own maximum.
3. Intermediate results are sent to a single node.
4. Maximum of the maximums is calculated.

Maximum aggregator is trivially parallelizable

- Most of the work can be done with no coordination.
- Very little communication is required.

- What other calculations are easy to parallelize?
  - Sum, min, max, mean, search/grep, etc.
- How would you parallelize *median* calculation if data is distributed?
  - Hint: first look up the basic (sequential) *quickselect* algorithm.
  - Can be parallelized by sharing *pivot* value and size of subset set ≤ pivot.
    - One node (leader) chooses a pivot, shares with all other nodes.
    - Each node rearranges its data into two sets: L (≤ pivot) and R (> pivot).
    - Nodes send (|L|) to the leader. Leader chooses a new pivot either greater or less than the previous pivot, depending on whether on sum(|L|) < n/2.
    - Repeat on subset L or R (as appropriate) until sum(|L|) == n/2

Basic features of distributed computing

- Input data is split among many nodes.
- Code must be distributed to many connected nodes.
- Many rounds of communication may be needed for an algorithm.

- A distributed algorithm can be written *from scratch* in any language by:
  - reading/writing memory and files, and
  - making network connections to other nodes to communicate.
- However, **distributed computing frameworks** have been developed to manage the common tasks required by all big data analyses.
  - Eg.: MapReduce/Hadoop, Dryad, Spark, MPI, OpenMP.
  - Computing frameworks require you to follow their own patterns, but by doing so you gain many powerful features (analogous to using a DB for storage).

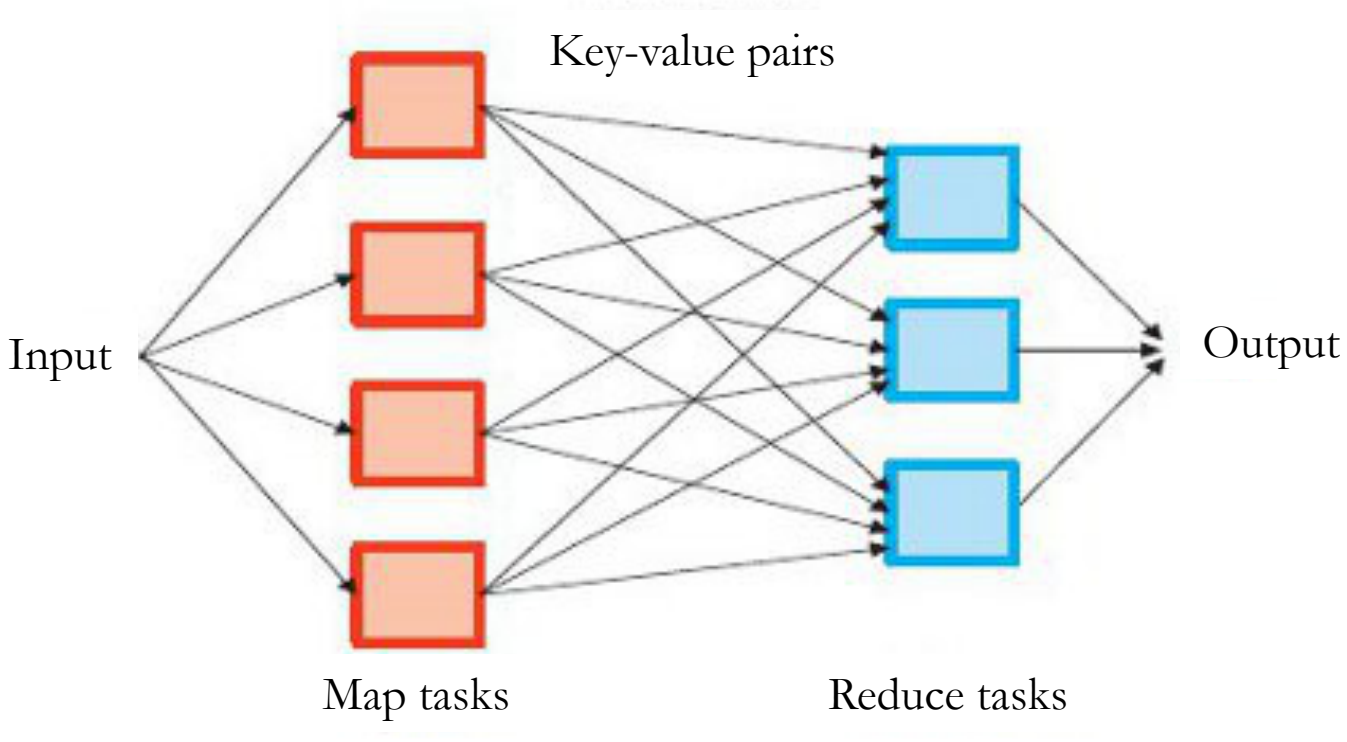
Hadoop is a distributed computing platform

- It's a type of computing cluster for storing data and running analyses.
- Main components are:
  - **HDFS** – *Hadoop Distributed File System* for storing the data to be analyzed and also for storing intermediate and final results.
    - Data is distributed across all the nodes (machines) in the cluster.
    - HDFS is closely coupled to the analysis engine.
    - Moving/querying the data would be too slow, so we run analysis on the storage cluster.
  - **MapReduce** – a programming model for defining the parallel analysis steps.
    - Analysis is usually written in Java, but other languages are possible (Python, R, etc.)
    - Analysis is defined in terms of two main operations – *Map* and *Reduce*.
    - Based on a 2004 Google article: https://research.google/pubs/pub62/

MapReduce

MapReduce is a framework for parallel processing imposing a certain pattern:

- First, **Map** does something, producing many intermediate results.
- Later, **Reduce** aggregates the intermediate results.
- The framework handles the distribution of intermediate results, job scheduling, etc. Programmer just implements Map and Reduce.

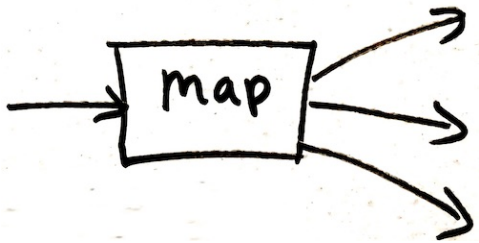


MapReduce programming model

- Input & Output: each a set of key/value pairs
- Programmer's task is to define just two functions that will complete the work in parallel:

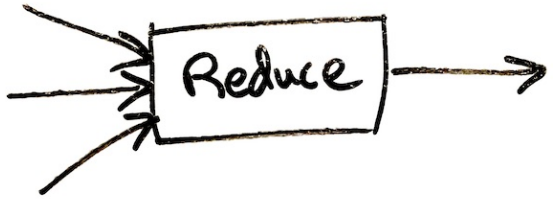
```
map(in_key, in_value) -> List(⟨out_key, intermediate_value⟩)
```

- Processes input key/value pair
  - Usually key=filename, value=line of text
- Produces a **set** of intermediate pairs




```
reduce(out_key, Iterator(intermediate_value)) -> list(out_value)
```

- Combines all intermediate values for a particular key
- Produces a set of merged output values (usually just one)



MapReduce example: frequency of words

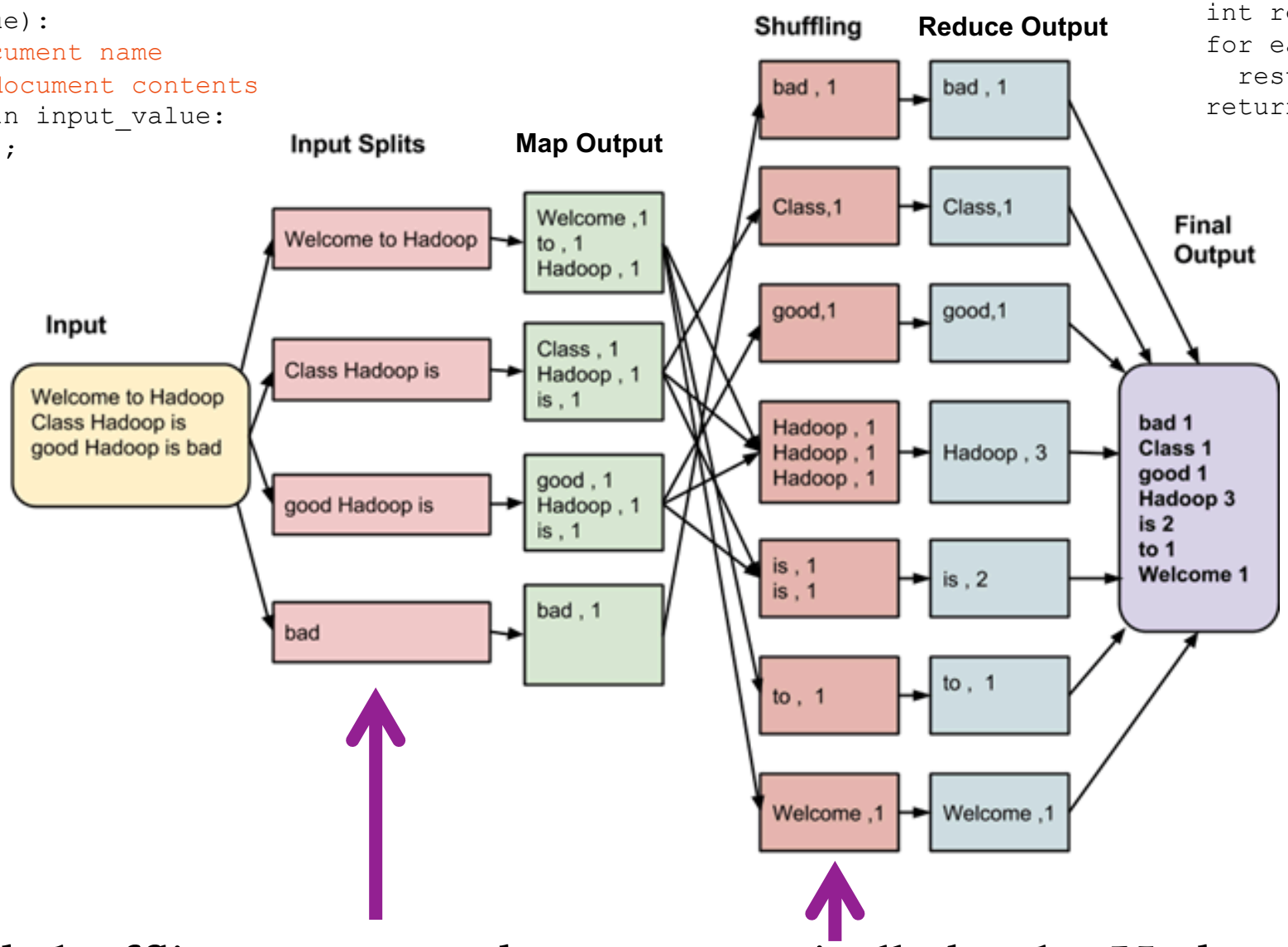
```
map(String input_key, String input_value):
    // input_key: document name
    // input_value: document contents
    for each word w in input_value:
      return List(⟨w, 1⟩);

reduce(String output_key, Iterator intermediate_values):
    // output_key: a word
    // output_values: a list of counts
    int result = 0;
    for each v in intermediate_values:
      result += v;
    return List(result);    // implicitly, output_key is also part of the output.
```

Counting words example

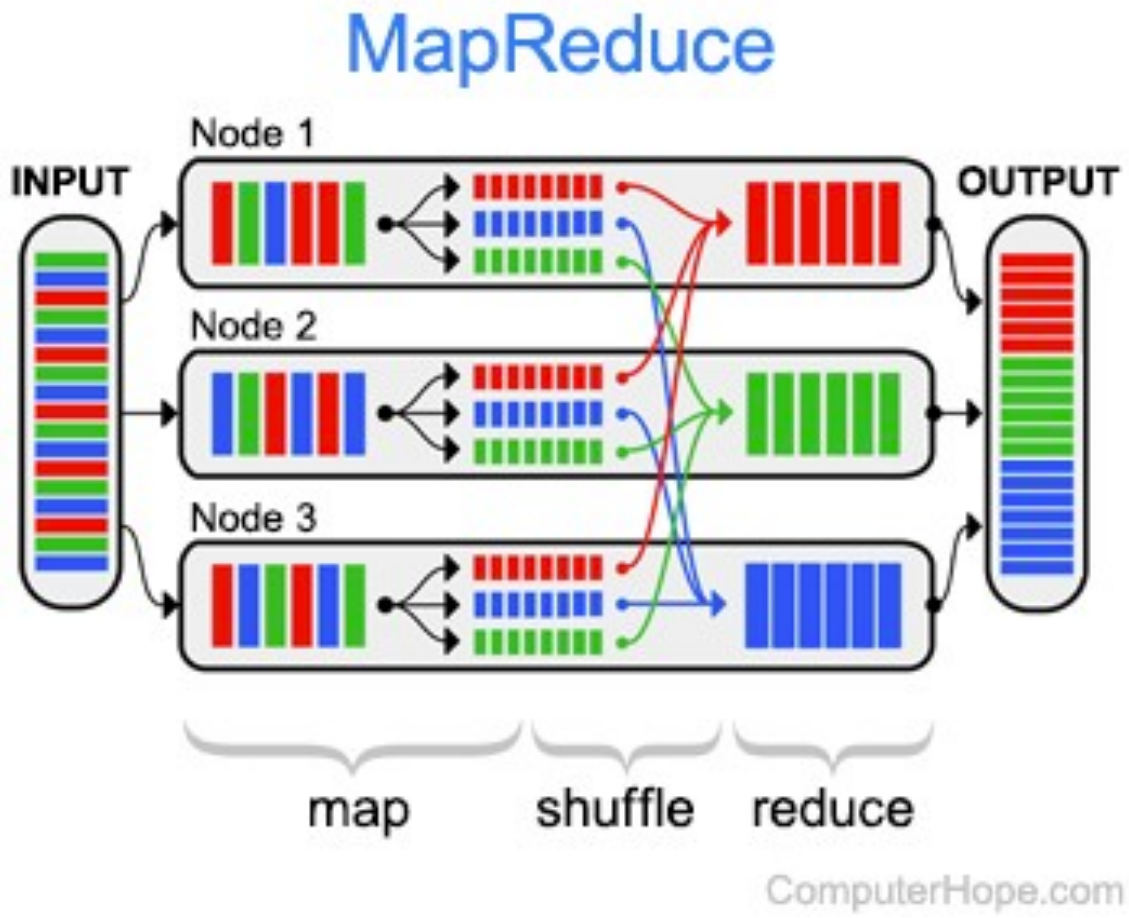
```
map(String input_key,
    String input_value):
    // input_key: document name
    // input_value: document contents
    for each word w in input_value:
      return (w, "1");
```

```
reduce(String output_key,
       Iterator intermediate_values):
    // output_key: a word
    // output_values: a list of counts
    int result = 0;
    for each v in intermediate_values:
      result += v;
    return result;
```



Splitting and shuffling steps are done automatically by the Hadoop runtime system

Communication between many nodes



The inter-node communication is handled entirely by:

- The **shuffle** step, and
- The **distributed file system** splitting the data across the compute nodes.
  - The input files stored on each node will be assigned to that node's mapper. This drastically reduces communication overhead.

The user code does not do any explicit communication.

The Hadoop framework provides:

- The Hadoop Distributed File System (**HDFS**) for distributing input and output data among many nodes.
  - Input to a program is usually a *folder* with files distributed on many nodes.
  - Again, the output of the program is a series of files written to a folder.
- A runtime system to manage job **scheduling** and coordination:
  - Many copies of your program are made (forked) and run on many nodes.
  - One node is set to be the "master." It assigns work (input data) to workers.
  - Workers (mappers and reducers) write results to HDFS and notify the master.
- A scalable, distributed **sorting** algorithm for the "shuffling" step.
  - All the intermediate data (produced by the mappers) must be sorted by the *key* so all data for a single key can be sent to one reducer.

To see lots of examples of MapReduce programs:

- https://research.google/pubs/pub62/

MapReduce tips:

- You can chain multiple MapReduces:
  (map1, reduce1, map2, reduce2)
- For complex problems, think about specifying a series of intermediate results.

Spark improves on MapReduce/Hadoop ideas

- Spark is very similar to Hadoop/MapReduce, but it solves some performance issues and its programs are easier to write and read.
- Hadoop is slowed by using the filesystem too much.
  - Recall that disk latency is 10M× greater than memory.
- Every Hadoop step uses the disk; it stores all of the following on disk:
  - Inputs to Map.
  - Intermediate values output by Map.
  - Reduce results.
- *Why?* Writing to disk simplifies data sharing and fault tolerance.
- Many **iterations** of MapReduce may be needed to solve a task (eg., PageRank). This is messy to write and involves a lot of disk activity.

Spark innovations

- User writes a **driver** program which can have an arbitrary number of steps. Analysis can also be run **interactively** for data exploration.
- More parallel operations are available (not just Map and Reduce):
  - **flatMap** (like Hadoop map), **map** (one to one), **filter** (return a subset).
  - **reduceByKey** (like Hadoop reduce), **reduce** (to one node).
  - **forEach** (process Scala collection items in parallel), **collect** (send all elements back to the driver node)

  } Allows parallel processing without HDFS.
- Operations to improve performance:
  - A **cache** operation to keep a data set in memory between parallel tasks.
  - **accumulators** with results only visible to the driver.

Resilient Distributed Datasets (RDDs)

- The most important innovation of Spark is to represent intermediate values as reproducible objects instead of saving to disk.
- RDDs are either:
  - A **filename** referring to something stored in HDFS.
  - A reference to an in-memory collection (eg., List) in the **driver program**.
  - The result of a parallel **transformation** of another RDD (eg., map, filter).
  - A **save** or **cache** action applied to an RDD. This does not change the data.

If the driver program crashes the analysis will fail anyway.

- An intermediate result might be represented as:
  - `myList[400:500].map(i -> sqrt(i))` //myList is in driver memory
  - `textFile("data0032.txt").map(line -> line.split(" ")).filter(w -> w=="Steve")`
  - If node processing either of the data above crashes, the in-memory copy is lost ☹
  - However, it can be re-generated from the lines above by re-running the fraction of the analysis that led to that data. ☺
- Data is ephemeral (in memory-only) but data **provenance** is safeguarded.

Spark example: Text Search

```
val file = spark.textFile("hdfs://...")
val errs = file.filter(_.contains("ERROR"))
val ones = errs.map(_ => 1)
val count = ones.reduce(_+_)
```

- The first three lines all define new RDDs.
- *errs* and *ones* are can be **lazy-evaluated:** for efficiency, don't run the filter and map until we know the final destination of the data.

Example: Logistic Regression (ML classification)

- Find a hyperplane *w* that best separates two sets of points.
- Solve with *gradient descent*.
- *cache* operation keeps points in memory through many iterations.
- Each training point contributes to gradient in parallel through *accumulator*.

```
// Read points from a text file and cache them
val points = spark.textFile(...)
                  .map(parsePoint).cache()
// Initialize w to random D-dimensional vector
var w = Vector.random(D)
// Run multiple iterations to update w
for (i <- 1 to ITERATIONS) {
  val grad = spark.accumulator(new Vector(D))
  points.foreach(p -> {  // Runs in parallel
    val s = (1/(1+exp(-p.y*(w dot p.x)))-1)*p.y
    grad += s * p.x
  })
  w -= grad.value
}
```

Traditional Parallel Computing

- Scientists have done large parallel computations starting decades before "big data" trends and the Internet.
- However, the tools they use are very different.
- Use clusters of machines with special low-latency networking.
  - Called "Supercomputing" or "High Performance Computing" (HPC).
- Data is stored on different nodes than where computing is done.
- Programming is done in C or Fortran.
- Parallelization is done with:
  - OpenMP – to automatically parallelize loops, etc.
  - MPI – programmer uses *gather*, *scatter*, etc., to distribute intermediates.
- Fault tolerance is a major problem in large supercomputing application because they can have ~100,000 machines (~10M cores).

Recap

- To efficiently analyze large data sets, it's important to merge computation and storage on the same machines. Minimize transfers.
- **Hadoop** is an open-source implementation of **MapReduce**.
  - A framework for parallel computing on Big Data problems.
  - Requires programmer to somehow express analysis as Maps and Reduces.
  - HDFS distributes input and output data files on the compute nodes.
- **Spark** improves performance by storing intermediate results in memory.
  - *Resilient Distributed Datasets* enable this.
  - Syntax is richer, leading to code that is easier to write and read. Multiple steps can be expressed easily.
  - It's easier for beginners to get started. Need not store inputs in HDFS.

Architectural concepts in this class:

Computing Platforms:

- Virtual Machines
- Containers.
- Serverless functions.

Programming Models:

- Stateless (micro)services.
- MapReduce/Spark.

Network-level optimization:

- HTTP caches.
- CDNs (Content Delivery Networks).

Coordination:

- Load balancers.
- REST APIs.
- Distributed Message Queues.
- Push Notifications, WebSockets.

Data storage:

- SQL Relational DBs.
- NoSQL DBs.
- Distributed file systems.
- Browser cookies, auth tokens.

Class Recap

- **The Goal**: to learn enough to build your own scalable startup product. Bypass the "on the job training" or self-study usually required.

To Explore this Further:

- CS-345 **Distributed Systems**
  - Shared-nothing distributed systems, consistency (NoSQL databases)
- CS-340 **Intro to Computer Networking**
  - Application layer protocols (HTTP, REST APIs)
- CS-343 **Operating Systems**
  - Processes and threads
  - Performance of storage vs RAM
- CS-339 **Intro to Databases**
  - Mostly covers relational database internals.