Some Reminders for a Seamless Online Class…

- Please turn on your video
- Mute yourself (press and hold spacebar when you'd like to talk)
- Don't do anything you wouldn't do in an in-person class
- I will occasionally check the chat for messages if you'd like to share there instead
- Please say your name before you speak

Logistics

- Deadlines have been moved
  - HW2 — due on 5/4; released before W
  - Final project presentations — pushed to 5/7 along with report
- Format for project reports and presentations
  - Similar to mid-term project report, but more on what you've done than your plan
    - At most 8 page summary on goals, relevance to project, design and architecture (and justification thereof), evaluation of design or architecture, concepts from class that were used in the project, related work, contributions of individual team-members. You can go beyond 8 pages only for additional figures or references.
    - FAQ forthcoming — will be released after HW2
  - Project presentation:
    - 5 minute video on youtube. Please post the link on piazza to a specific thread we will create.

Recap

- Data-savviness is the future!
- "Classical" relational databases
    - Notion of a DBMS
    - The relational data model and algebra: bags and sets
    - SQL Queries, Modifications, DDL
    - Database Design
    - Views, constraints, triggers, and indexes
    - Query processing & optimization
    - Transactions
- Non-classical data systems
    - Data preparation:
        - Semi-structured data and document stores
        - Unstructured data and search engines
    - Data Exploration:
        - Cell-structured data and spreadsheets
        - Dataframes and dataframe systems
        - OLAP, summarization, and visual analytics
    - Batch Analytics:
        - Compression and column stores
        - **Parallel data processing and map-reduce**

Some material drawn from Mining Massive Data Sets, Leskovec, Rajaraman, Ullman

Parallel data processing and map-reduce

- We've studied OLAP — a specialization of relational databases targeted at business analytics and reporting at scale with data cube materialization and column stores

- Today, we're going to be studying the primitives for processing large volumes of relational, unstructured, or semi-structured data at scale

- Often, when we're trying to process really large volumes of data, we need to span across multiple nodes/machines
    - This hasn't been a focus of our class so far
- We'll start by covering what map-reduce offers, before switching over to cover parallel databases

Let's revisit search engines

- To create an inverted index, we need to read all the webpages
  - Size of the web: 20+B web pages x 20 KB = 400+ TB
  - Disk reading speed ~50MBPS
  - On a single disk drive: (400 x 1000 x 1000) / (35 x 60 x 60 x 24 x 30)
    - 4 months to read the web!
  - If each hard-drive can store 1TB, then 400 hard-drives to store the web.

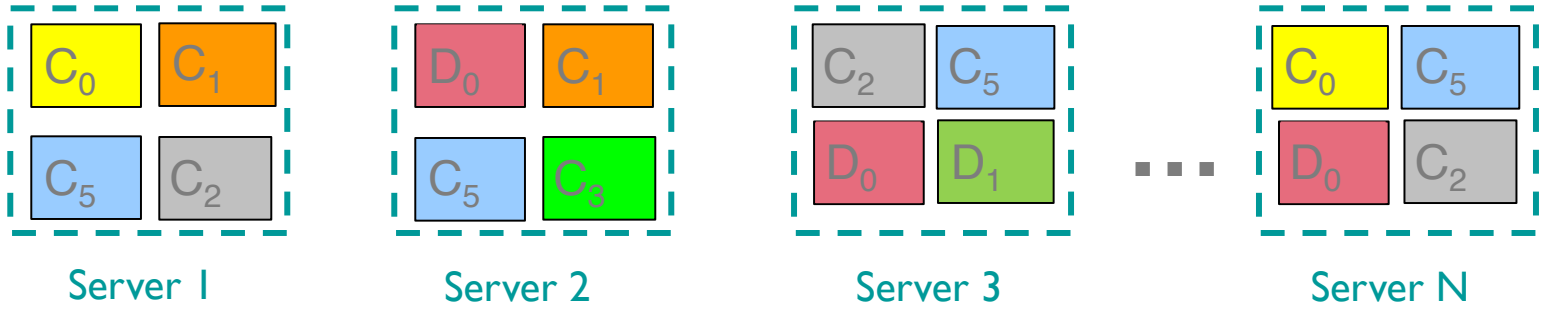
- Instead, parallelize!
  - Modern data system architectures use many cheap (commodity) machines connected by cheap network (ethernet)
- Q: If we could read from all 400 hard-drives at once, we could get the job done in how much time?
  - 120 x 24 hard-drive-hours / 400 hard-drives = 7.2 hours!

Large Scale Data Processing

- Challenges:
  - How do we specify the processing task?
  - How do we distribute this processing across these many machines?
    - Transferring data across network is expensive
  - How do we deal with failures?
    - If a server dies once in a 100 days, then if we have a 1000 servers, on average 10 will die per day

Google's Approach: Circa '00s

- First component: a distributed, replicated file system
  - GFS, modern open-source incarnation: HDFS
  - Just like a file system in your machine, but each file is replicated across machines for reliability
  - Files are very large; rarely updated, but reading and appending are more common
  - There is a "coordinator" node that keeps track of which file is where



- Second component: a distributed data processing programming paradigm
  - Map-Reduce, modern open-source incarnation
  - The processing usually happens "close" to the data if possible to avoid moving data across network (but in some cases unavoidable)

Map-Reduce

- Let's say we have the entire web crawl and we want to count the # of times each word appears on the web
  - The web is stored across 400 machines, each with its hard-drive

- Q: How would you go about doing this?

Map-Reduce Overview

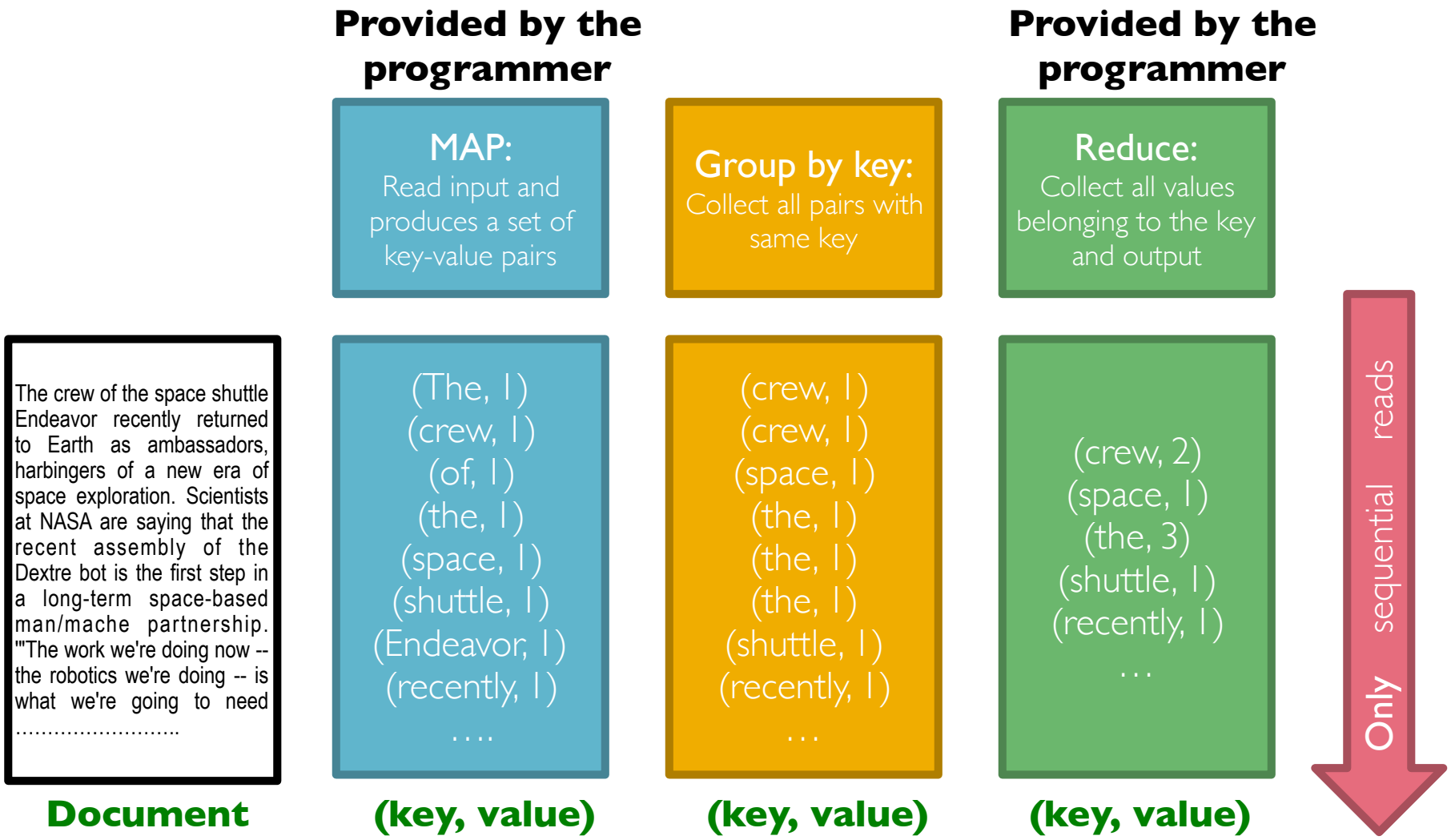
- Developer-provided Processing Step 1: **Map**
  - Read a lot of data and extract something of value
- Internal Step 2: Group by Keys (**Shuffle**)
- Developer-provided Processing Step 2: **Reduce**
  - Aggregate, summarize, filter, transform data organized by key

Map-Reduce Overview

- Developer-provided Processing Step 1: **Map**
  - Read a lot of data and extract something of value
  - *For each document, emit <word: 1> pairs — key: value pairs*

- Internal Step 2: Group by Keys (**Shuffle**)
  - *Group/sort pairs based on word, so: <word 1, 1>, <word 1, 1>, … <word 1, 1> <word2, 1> …*

- Developer-provided Processing Step 2: **Reduce**
  - Aggregate, summarize, filter, transform data organized by key
  - *For each word, sum up the total number of 1s in the value*


Provided by the programmer
MAP:
Read input and produces a set of key-value pairs
Group by key:
Collect all pairs with same key
Provided by the programmer
Reduce:
Collect all values belonging to the key and output
The crew of the space shuttle Endeavor recently returned to Earth as ambassadors, harbingers of a new era of space exploration. Scientists at NASA are saying that the recent assembly of the Dextre bot is the first step in a long-term space-based man/mache partnership. '"The work we're doing now -- the robotics we're doing -- is what we're going to need ……..
(The, 1)
(crew, 1)
(of, 1)
(the, 1)
(space, 1)
(shuttle, 1)
(Endeavor, 1)
(recently, 1)
….
(crew, 1)
(crew, 1)
(space, 1)
(the, 1)
(the, 1)
(the, 1)
(shuttle, 1)
(recently, 1)
…
(crew, 2)
(space, 1)
(the, 3)
(shuttle, 1)
(recently, 1)
…
Only sequential reads
Document
(key, value)
(key, value)
(key, value)

The Map-Reduce Paradigm is General!

- Simply change the map and reduce functions to fit new applications
- Semantics:
  - Programmer specifies two methods:
    - *Map(k, v)* → *<k', v'>**
      - Takes a key-value pair and outputs a set of key-value pairs
      - E.g., key is the filename, value is a single line in the file
      - There is one Map call for every (k,v) pair
    - *Reduce(k', <v'>*)* → *<k', v''>**
      - All values v' with same key k' are reduced together and processed in v' order
      - There is one Reduce function call per unique key k'

For Our Word Count Example

- Map (key, value)
  - //key is documentname, value is contents
  - For each word in value
    - Output (word, 1)

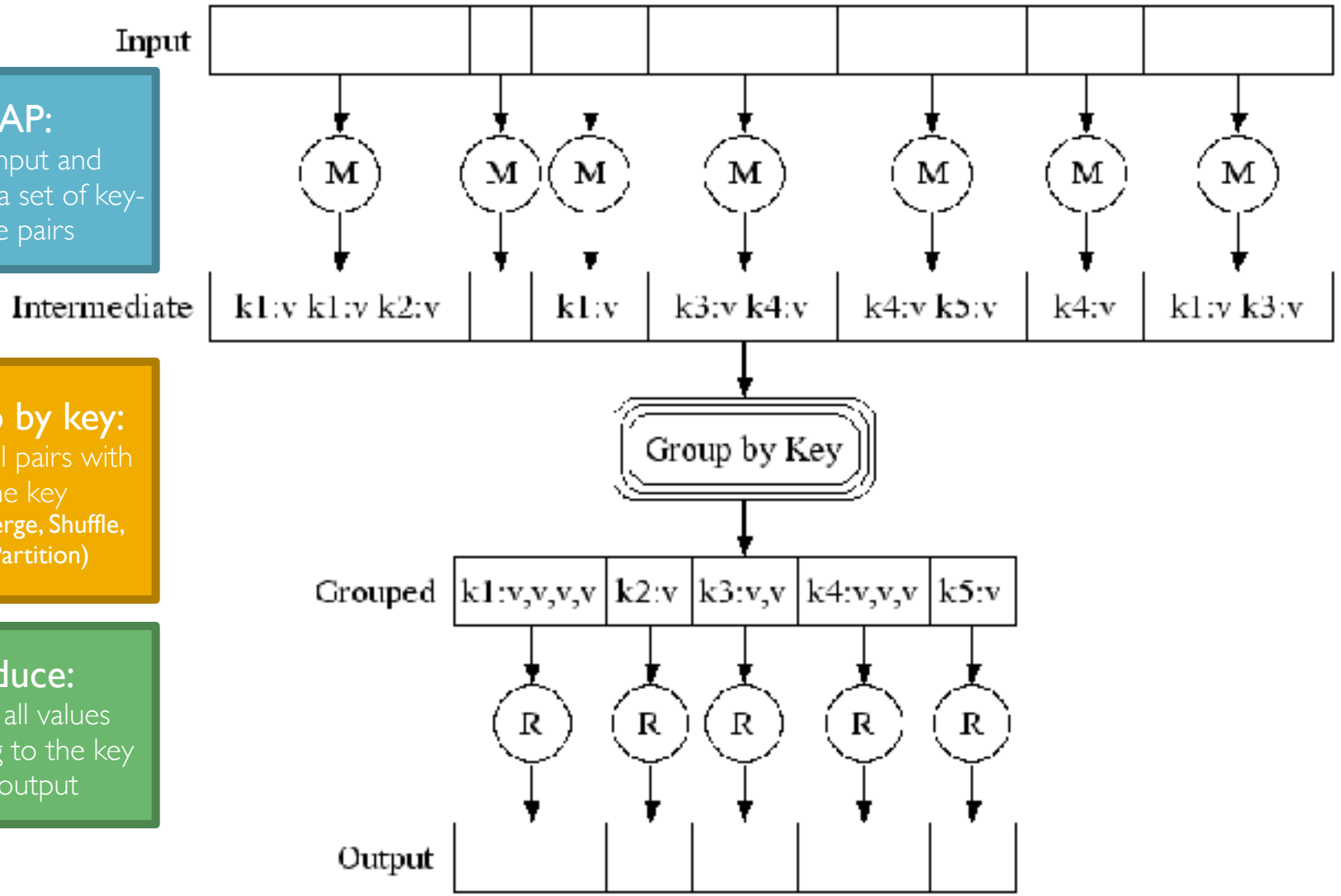
- Reduce (key, values)
  - // key is word, values is a list of 1s
  - Output (key, size (values))

What conceptually happens

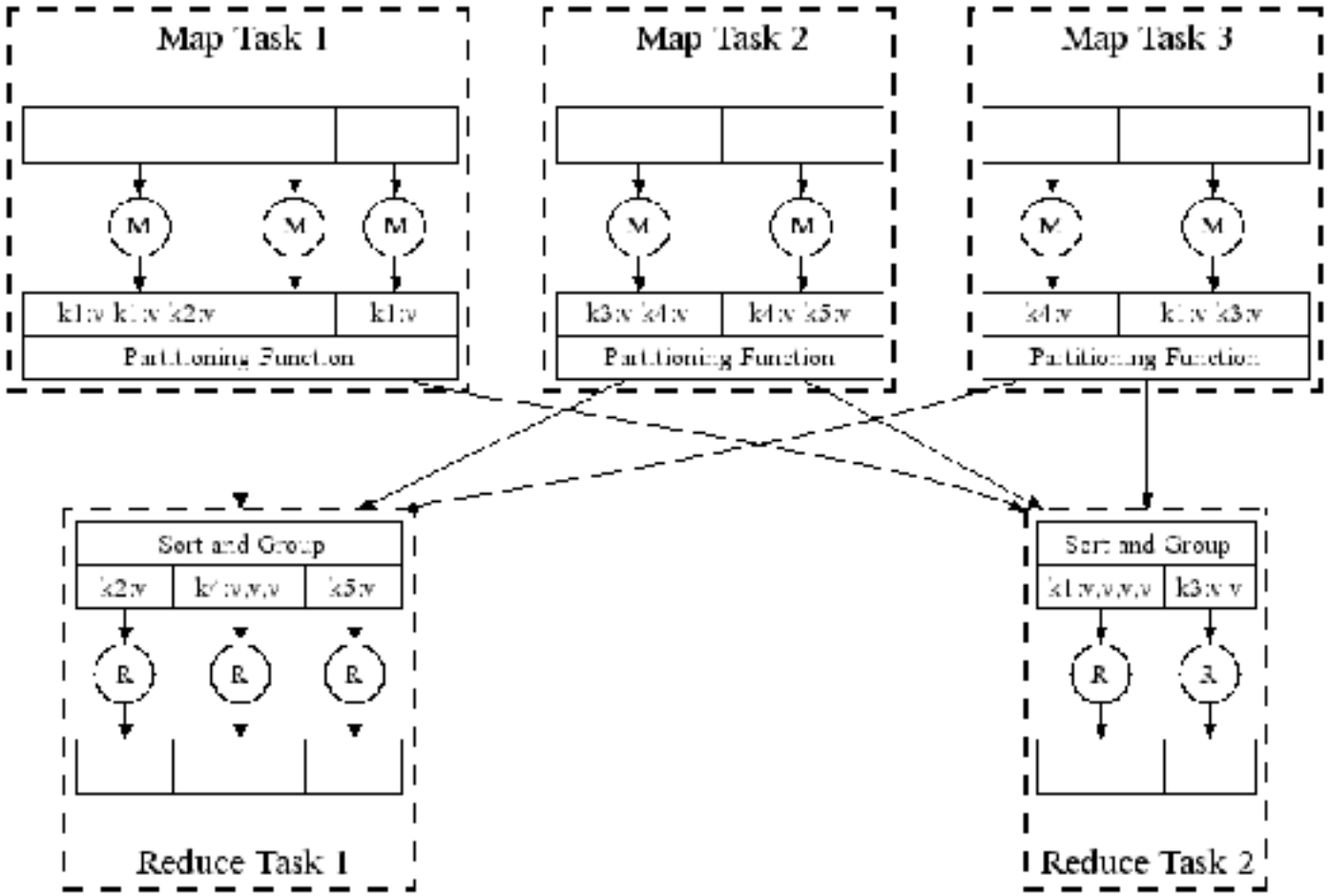
MAP:
Read input and produces a set of key-value pairs

Group by key:
Collect all pairs with same key
(Hash merge, Shuffle, Sort, Partition)

Reduce:
Collect all values belonging to the key and output



What actually happens



What does the Map-Reduce Paradigm Handle?

- Assigning tasks to machines
- Performing the "shuffle" step
- Handling failures and inter-machine communication
  - Does so via materialization after every step
- In some cases, can apply a "combiner" at the map task itself to reduce the amount of data shuffled, especially if the reduce function has some nice commutative properties
  - For example, we can sum up the 1s for each word for each document before shuffling across network

A slightly harder case: inverted index construction… remember this slide?

- Phase 1:
  - For each document, generate a "canonicalized term, documentID" pair for all non-stop-word terms in the document
  - Can encode more information with the pair, e.g., list of locations, frequency of occurrence, etc.
- Phase 2:
  - Sort these pairs based on the term, documentID pair
  - May need to use two or multi-pass sort (remember query processing!)
- Phase 3:
  - Read in the sorted pairs based on term, concatenate documentID together to form a posting list for that term
  - (We may be able to merge Phase 3 with Phase 2)

Q: How would we use M-R to construct the inverted index?

- Obama -> 10, 24, 125, 259, 1025, 2314, …
- Bush -> 10, 15, 17, 259, 2001, 2547, …
- Clinton -> 10, 15, 24, 17, 125, 1005, 2001, 2347, …

Answer

- Map
  - Input: document
  - Output: <word: documentID>
  - Optionally, position
- Shuffle based on word

- Reduce
  - Input: <word, documentID list>
  - Output: <word, sorted list of documentIDs, dropping duplicates>
- What would a combiner do here?

Other Map-Reduce Exercises

- Compute the total degree (indegree plus outdegree) for each node in a graph
- Input directed graph representation:
    - "Edge-pair" representation
    - G(V1,V2) split across many machines
- Q: how would we do this?

- Map:
    - Input <V1,V2>
    - Output <V1: 1>, <V2: 1>
- Reduce:
    - Input <V1: list of 1s>
    - Output: <V1: size of list>

Other Map-Reduce Exercises (II)

- Computing a natural join
- Input relations R(A, B), S(B, C) partitioned across many nodes
- Q: how would we do this?

- Map: for input R (S)
  - Output <B: (R, A)> (<B: (S, C)>)
- Reduce: for input <B: (R, A1)…(R, An), (S, C1)…(R, Cm)>
  - Output: cross product of Ai and Cj
    - (A1, B, C1), (A1, B, C2), …, (An, B, Cm)

Try at home!

- Q: How would you compute connected components in a graph?
- Q: How would you compute page rank?

How would we use M-R?

- Since Google introduced its Map-Reduce in the early 2000s, open source alternatives have appeared:
  - Hadoop MapReduce (2006) — still exists to this day for open-ended large scale data processing
- Other document stores also support M-R on json-type documents
  - Key-value stores are a natural fit for M-R
  - CouchDB
  - MongoDB (demo!)

Downsides of Map-Reduce?

- Q: what are potential downsides of map-reduce?

- Hand-writing algorithms: no indexing, no query optimization
- Materialization after every step (no pipelining)
- No "declarative" query processing

From "A Comparison of Approaches to Large-Scale Data Analysis", SIGMOD'09

- Compares Vertica, an industry relational DBMS (DBMS-X), and Hadoop

- A few charts…
  - Since Hadoop writes out partial results locally, the "white" stacked bar shows the time to assemble the final result

Loading and Filtering

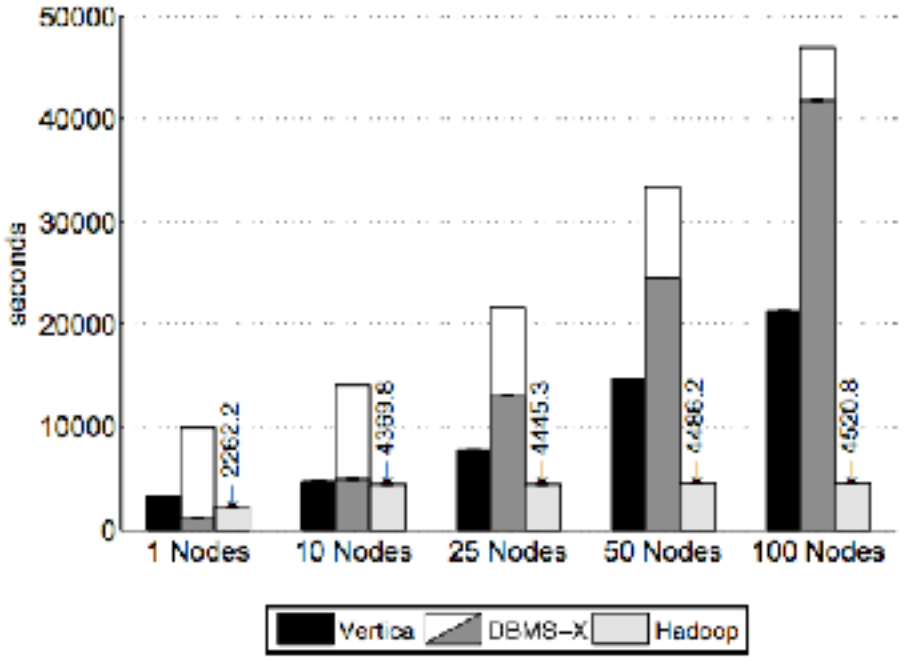


Figure 3: Load Times – UserVisits Data Set (20GB/node)

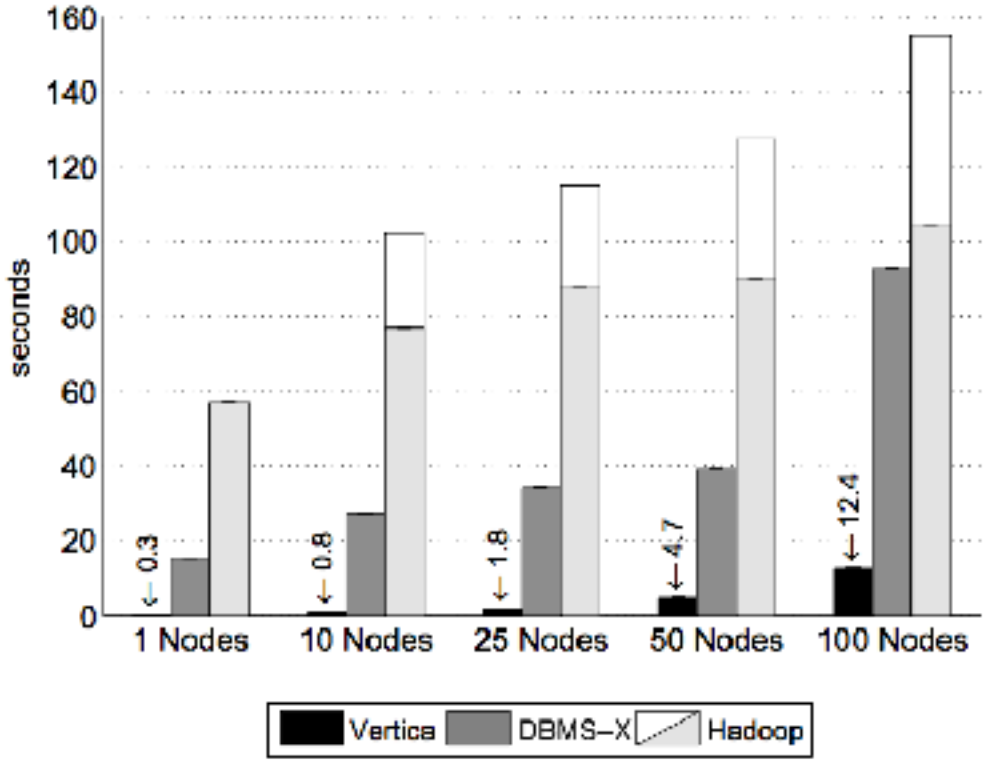


Figure 6: Selection Task Results

Aggregation and Joins

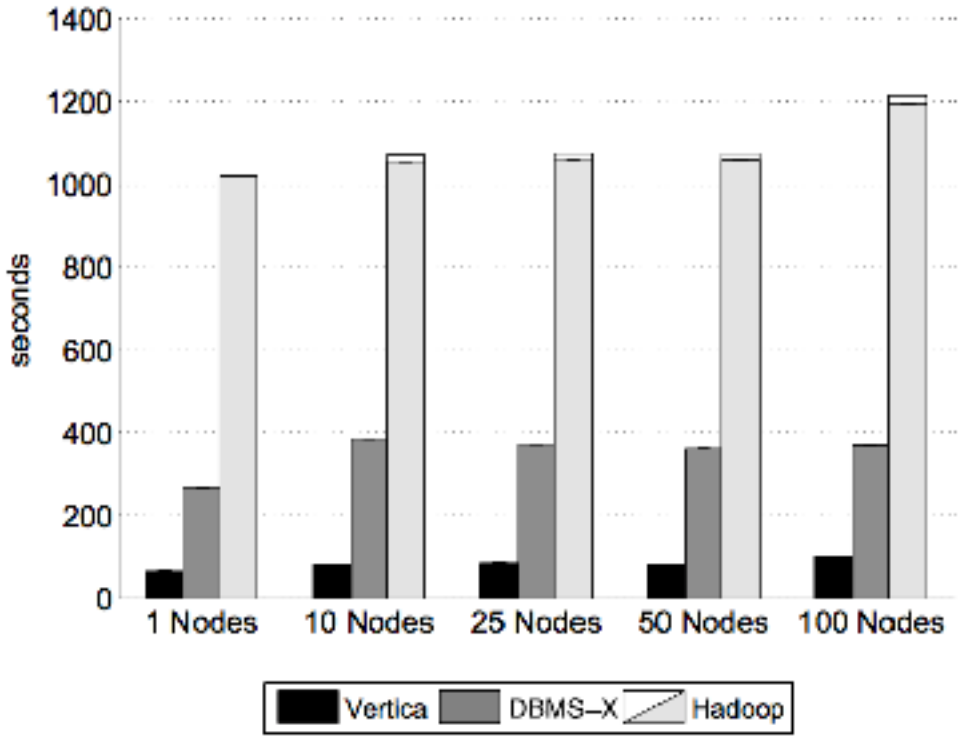


Figure 8: Aggregation Task Results (2,000 Groups)

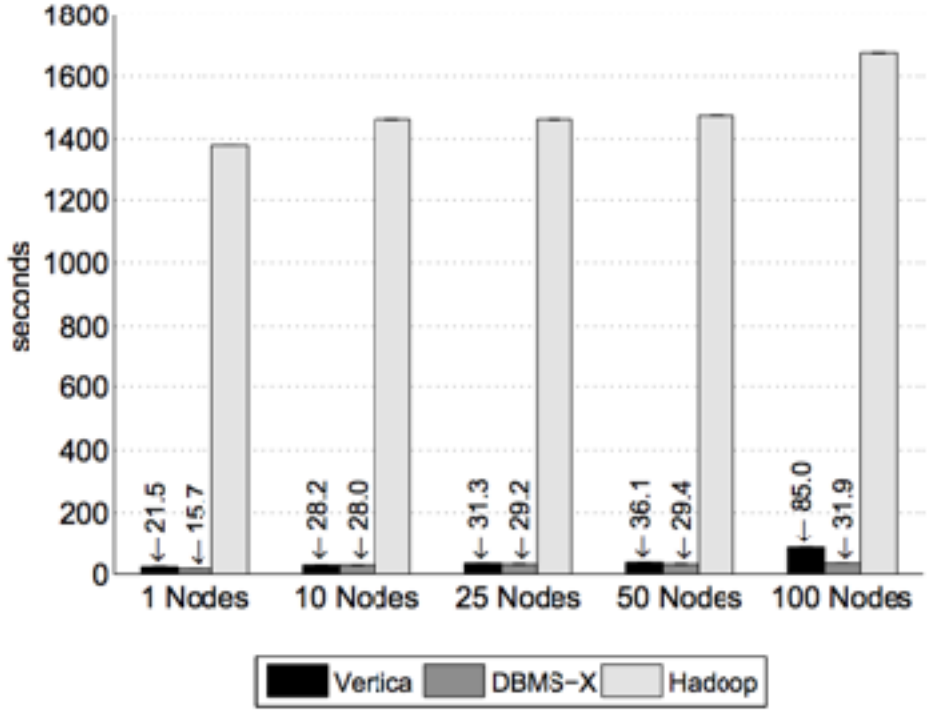


Figure 9: Join Task Results

Moral of the Story

- While convenient to do arbitrary large scale parallel data processing on arbitrary data, Map-Reduce isn't actually all that efficient compared to parallel implementations of relational databases (column or row stores)
  - So if your data is relational, relational databases is definitely a better choice.

- Parallel databases have been around since the late 90s!

- So how do column and row stores work in parallel anyway?

Parallelism

- We've already seen pipelined parallelism — where different operators work "in parallel" with each other, pipelining results

- Also can do "partitioned parallelism"
  - Where each operator has many parallel threads or processes reading data in parallel across partitions
  - These partitions could be in the same machine or across multiple machines

How does one partition the data?

- Vertical partitioning
  - Partitioning by columns
    - Very natural in column stores, certain column groups can be stored
- Horizontal partitioning
  - Partitioning by rows
  - e.g., rows A-M by name are in node 1, N-Z are in node 2
  - Various ways of doing this with trade-offs
    - Hash-based partitioning, e.g., based on h(A)
    - Range-based partitioning, e.g., A from [1, 100] in node 1, [101, 200] in node 2
    - Round-robin partitioning, add a tuple to each partition in sequence
- Let's talk about point access, range access, scans, updates, deletes, and skew

Detour: Skew

- When dealing with parallel processing across many machines, a common problem is *skew*
  - i.e., uneven distributions
- This could be in the data distribution, e.g., R has many tuples with value of A = a
- Or, could be in the access patterns, e.g., tuples in R corresponding to A = a are more frequently accessed than other values of A
  - This could be, for example, because more recent data is more frequently accessed

Range-based partitioning

- Point queries
  - On the partitioning attribute: great — just one partition
  - Else need to look at all partitions
- Range queries, e.g., A in [3,100]
  - If on partitioning attribute, then great — just one partition
  - Else need to look at all partitions
- Updates/deletes
  - Easy if on partitioning attribute, otherwise need to look at all partitions
- Skew:
  - Fairly susceptible to skew if there are some partitioning attribute ranges that are very popular

Hash-based partitioning

- Point queries
  - On the partitioning attribute: great — just one partition
  - Else need to look at all partitions
- Q: Range queries, e.g., A in [3,100]
  - Irrespective, need to look at all possible partitions
- Updates/deletes
  - Easy if on partitioning attribute, otherwise need to look at all partitions
- Skew:
  - somewhat susceptible to skew if there are some partitioning attribute values that are very popular (both from a data or an access standpoint)

Round-Robin partitioning

- Q: Point or range queries
  - need to look at all partitions
- Updates/deletes
  - need to look at all partitions
- Q: Skew:
  - not at all susceptible to skew since work is equally divided
    - but there is more work involved since all partitions need to be consulted

Replication

- In some cases, there may be benefits to not just partition the data but replicate it across nodes
  - Allowing for failures to some nodes to not affect the progress of queries
  - If one node is "busy" other node with same data can help with that query
- However: same issue as with materialization — need to keep replicas in sync to avoid staleness issues
- More on transactional issues possibly later (not a focus of this class)

Other Query Optimization Concerns

- Indexes:
  - Can be constructed globally (which partition contains the relevant data) and locally (which block within a partition contains the relevant data)
- Parallel versions of scans, sorts, joins, …
  - Scans are easy — scan each partition in parallel
  - Q: how do we sort in parallel?
    - If already partitioned on sorting attribute, then easy

Parallel Two-Pass Sort-Merge

- Recall regular sort:
  - Pass 1:
    - Sort B(R)/M subsets of M blocks of R each
    - Each is output to disk as a *run*
  - Pass 2:
    - "Merge" these runs by bringing in one block for each of them

- Pass 1: Locally sort data at each node
- Pass 2: Shuffle runs across the network
- Pass 3: Merge these partitioned runs across nodes
  - Node 1 gets all runs corresponding to [1-100] and then then merges them
  - Node 2 gets all runs corresponding to [101-200] and then then merges them
  - …
- Pass 4: The sorted runs at Node 1, Node 2, … are kept as is in a partitioned manner, or are concatenated to give an overall sort

Parallel Sort Approach 2: Partitioned Sort

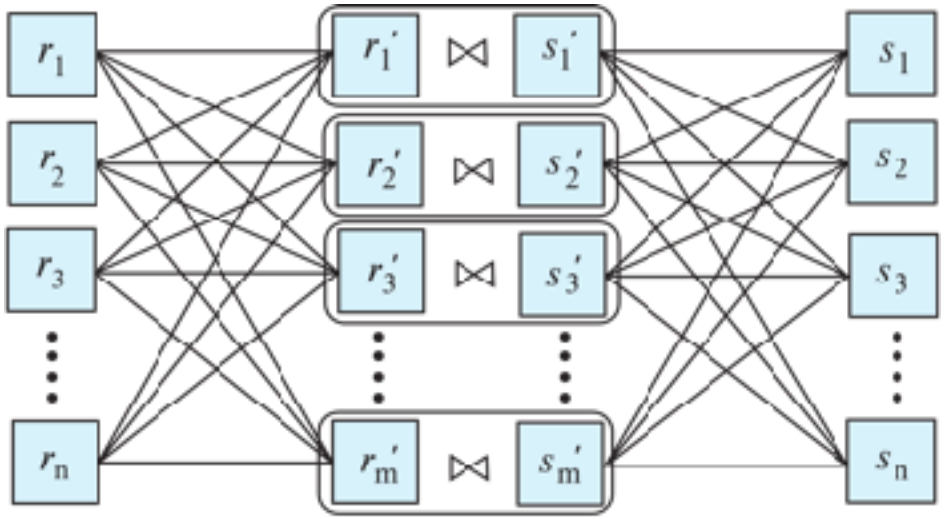
- Pass 1: Each node does partitioning, sending tuples across the network:
  - Node 1 gets all tuples corresponding to [1-100]
  - Node 2 gets all tuples corresponding to [101-200]
  - …
- Pass 2: Each receiving node then sorts all the tuples locally
- Pass 3: The sorted runs at Node 1, Node 2, … are kept as is in a partitioned manner, or are concatenated to give an overall sort

- Both approaches are roughly equivalent…

Parallel Joins

- Say we are trying to natural join R and S on A
- Let's assume that neither R nor S is partitioned on A
- Q: How would we do this join?

- Partition R on A across nodes
- Partition S on A across nodes
- Then each node individually does a hash join, a merge join, a nested loops, etc.



Step 1: Partition r Step 2: Partition s

Step 3: Each node N_i computes $r_i' \bowtie s_i'$

Diagrams from "Parallel and Distributed Query Processing" — Database System Concepts

Parallel Joins with Complex Join Conditions

- When the join conditions become a little more complex
  - multiple predicates, <>, …
  - e.g., Join R(A, B) and S(C, D) where R.A + S.C > 10

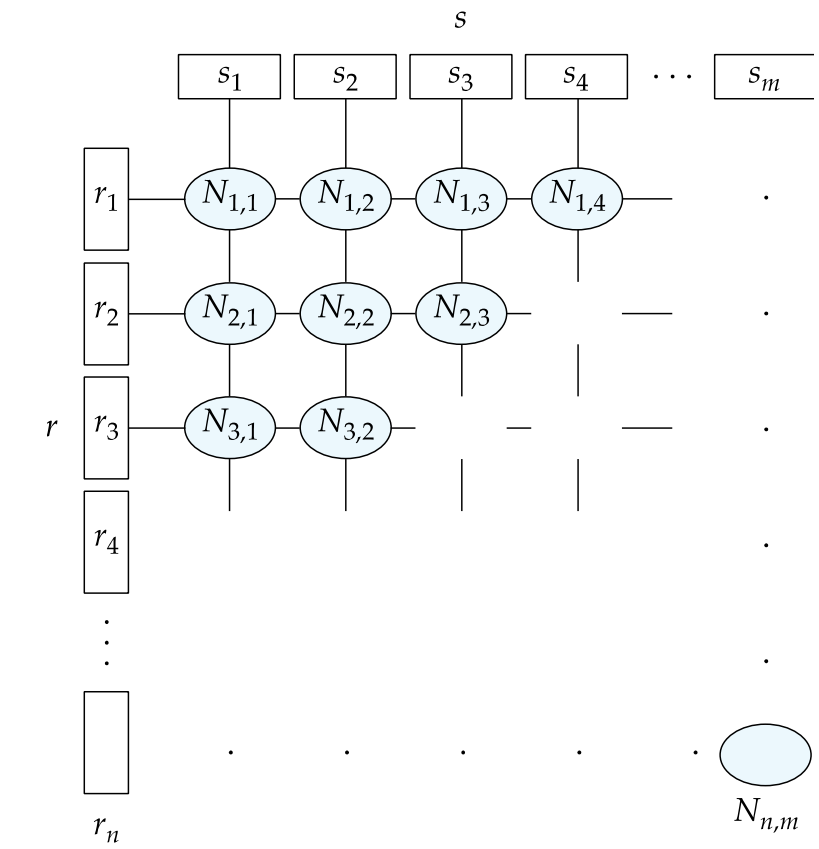
- Can't simply apply partitioning as is

- Q: What would we do for this?

Parallel Join: Fragment-Replicate Join

- Every fragment in R is joined with every fragment in S

- Extreme case: only one fragment of R (when it is small)
  - Called a broadcast or asymmetric fragment-replicate join



Other Parallel Operators

- Selection
  - Easy to do if selection on partitioning attribute; otherwise done in parallel across all partitions

- Aggregation
  - Easy to do if grouping is on partitioning attribute
  - Else two steps:
    - Partition into the groups
    - Each group performs aggregation
    - This is very similar to MR but:
      - The system will handle the right approach for partitioning and aggregation.
      - It will also do partial aggregation at the source nodes, negating the need for users to identify this opportunity and write a combiner function for it
      - Also, the aggregation can happen "pipelined" with the grouping: in MR the intermediate results need to be materialized.

Implementing this in a Relational Database: The Volcano Framework (from late 90's)

- Considering so many parallel variants for every operator can be quite a challenge
- A generalized framework that easily incorporates parallelism into the query processing process with the introduction of a new movement-oriented operator called *exchange*
    - All other operators are unaware of parallelism and perform purely local operations on local data
- Exchange does one of several things:
    - Hash/range-based partitioning of data
    - Replicating data to all nodes (broadcasting)
    - Sending all data to a single node
- Destination operators can read data from multiple "streams" via
    - Random merge (order doesn't matter)
    - Ordered merge

The Volcano Framework Examples

- Range partitioning sort:
  - Exchange operator using range partitioning, followed by
  - Local sort
- Parallel external sort merge
  - Local sort followed by
  - Exchange operator with ordered merge
- Partitioned join
  - Exchange operator with hash or range partitioning, followed by
  - Local join
- Asymmetric fragment and replicate
  - Exchange operator using broadcast replication, followed by
  - Local join

Takeaways

- Many relational operators are highly parallelizable
- Parallelism has been around in databases since the 90s

- Map-Reduce is still a good alternative for ad-hoc processing of unstructured or semi-structured data, but not for structured data
  - Many downsides:
    - cumbersome to write code, materialize after every step (slow), no optimization

Somewhere In Between: Spark!

- In 2010, Spark (from UC Berkeley!) was introduced as a way to bridge the gap between Map-Reduce and Parallel databases
- Spark introduced the notion of RDDs (resilient distributed datasets)
    - Fancy name for data cached in memory with a "query" attached to it
- Originally, Spark forced users to use a small set of operators to perform data manipulation — think more than relational databases, but less than dataframes
    - Many of these operators are targeted towards machine learning
- Spark is much faster than Map-Reduce while providing many of the same benefits: ease of distributed computation, ability to work on arbitrary data types, …
    - Spark's speed comes from some amount of query optimization (thanks to the restricted set of operators), pipelining, …
- More recently, Spark has made moves to support SQL as their primary language
    - So it has become more like a parallel database, but with some additional machine-learning-oriented bells and whistles