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

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.

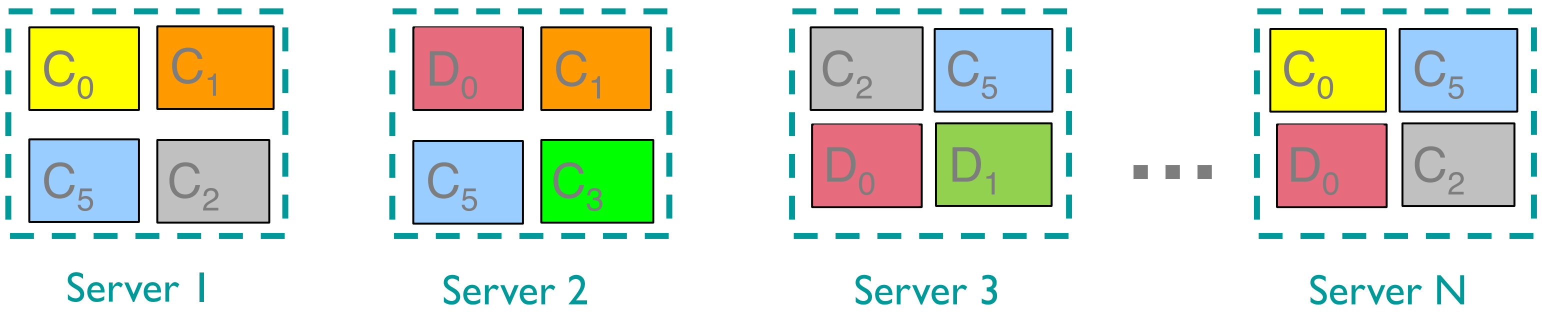
- Instead, parallelize!
    - Modern data system architectures use many cheap (commodity) machines connected by cheap network (ethernet)
- Q: If we could read from all 400 hard-drives at once, we could get the job done in how much time?
    - 120 x 24 hard-drive-hours / 400 hard-drives = 7.2 hours!

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

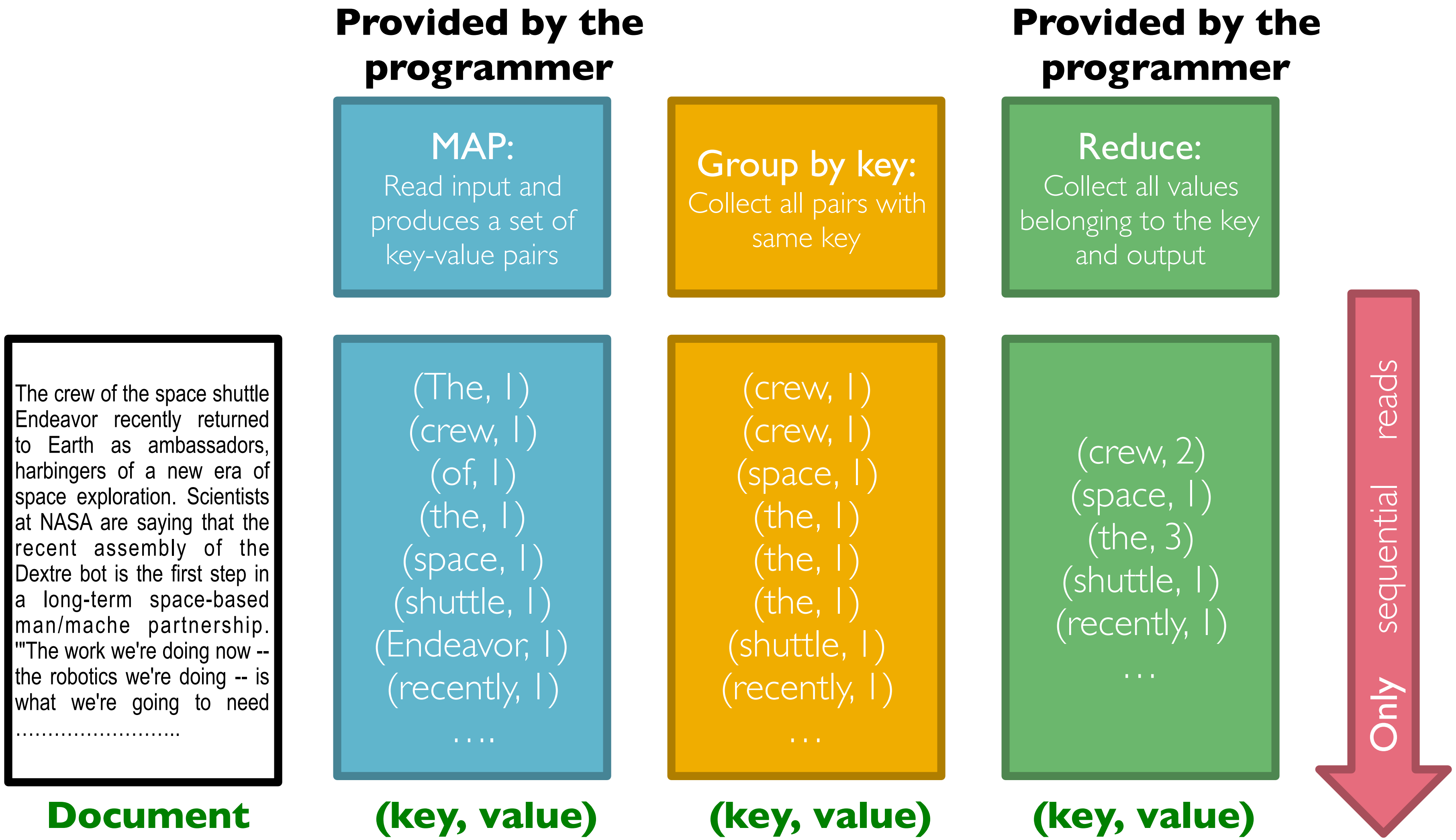
- Developer-provided Processing Step 1: **Map**
  - Read a lot of data and extract something of value

- 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
Provided by the programmer
MAP:
Read input and produces a set of key-value pairs
Group by key:
Collect all pairs with same key
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)

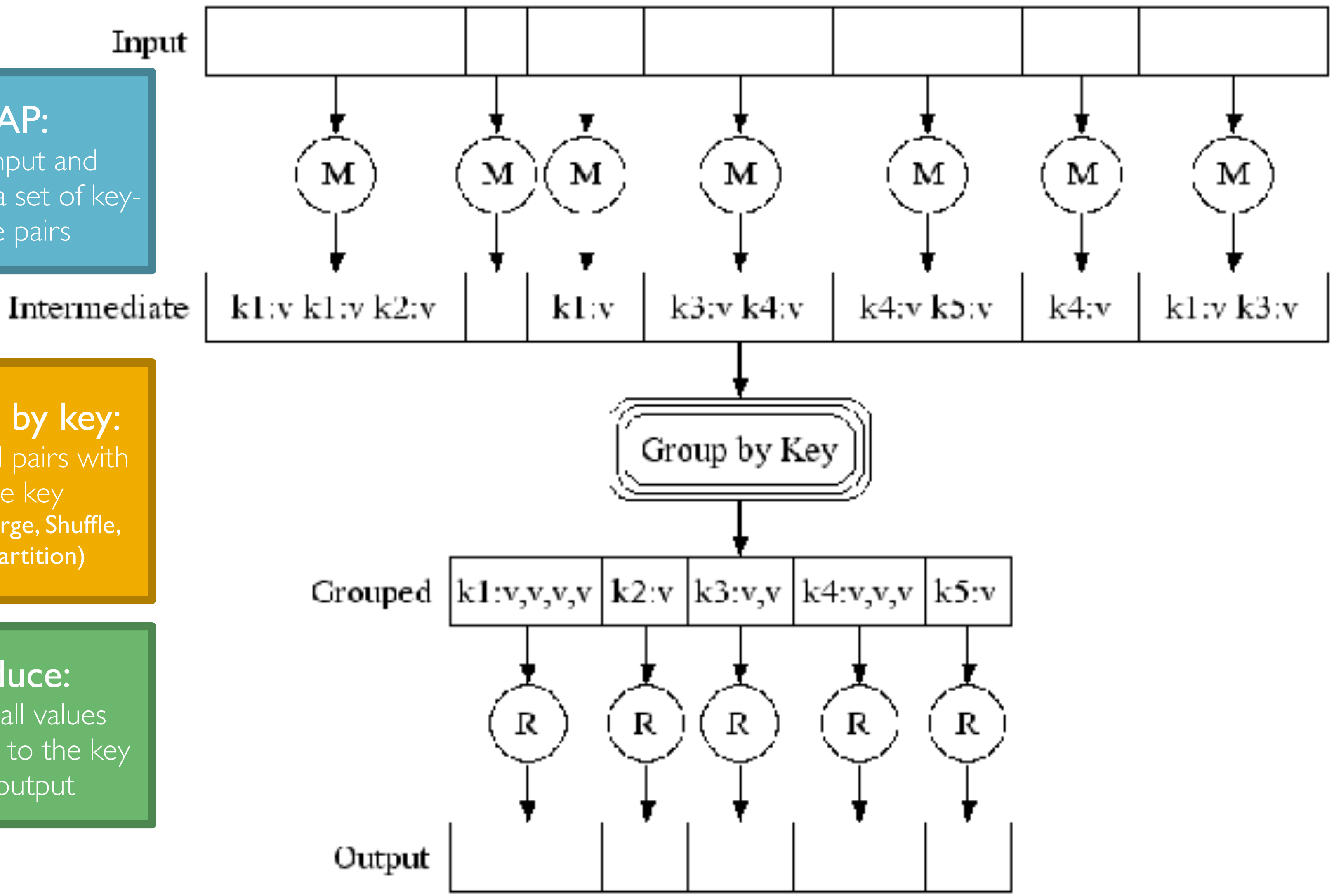
- Reduce (key, values)
  - // key is word, values is a list of 1s
  - Output (key, size (values))

What conceptually happens

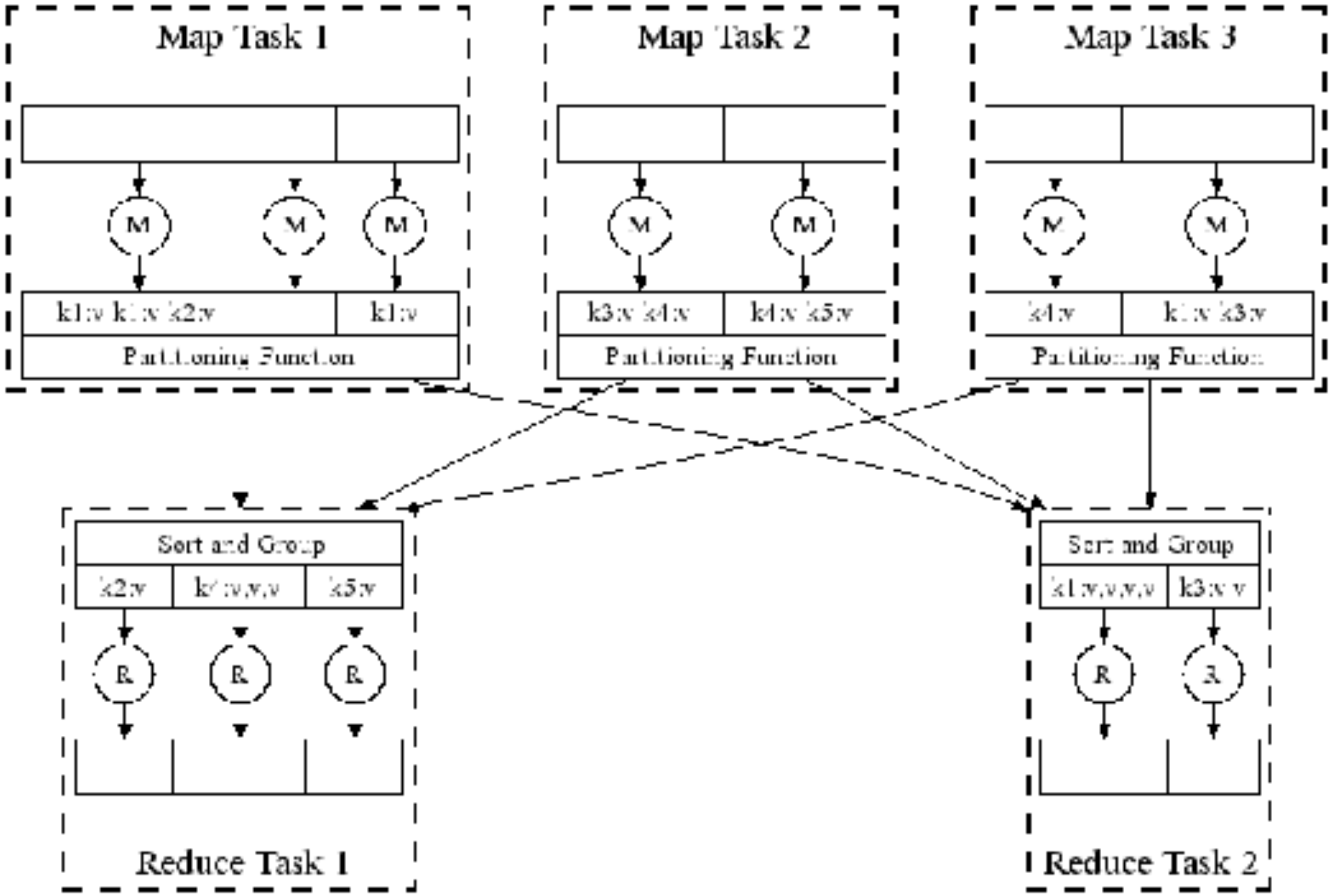
MAP:
Read input and produces a set of key-value pairs

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

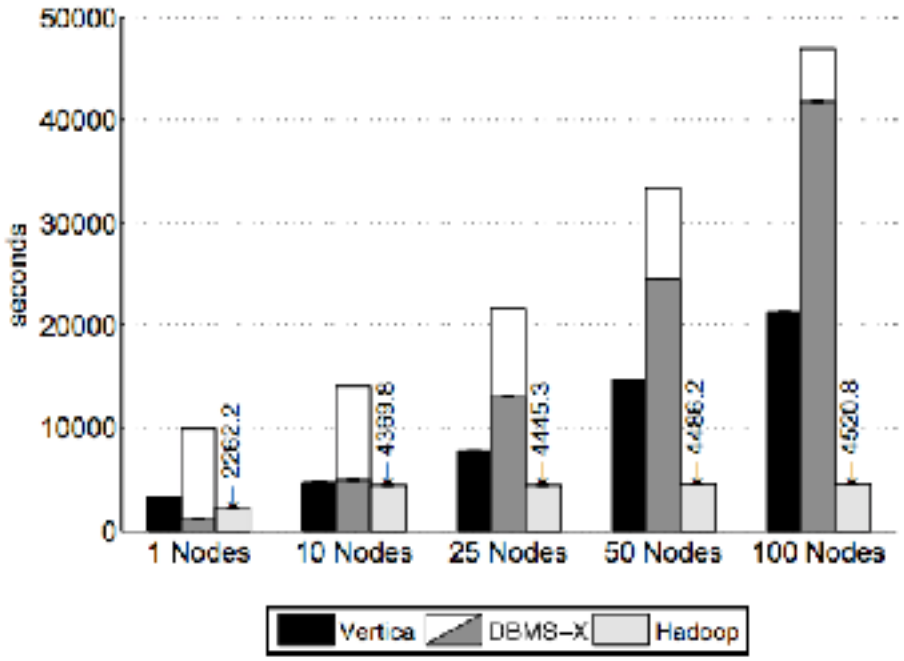


Figure 3: Load Times – UserVisits Data Set (20GB/node)

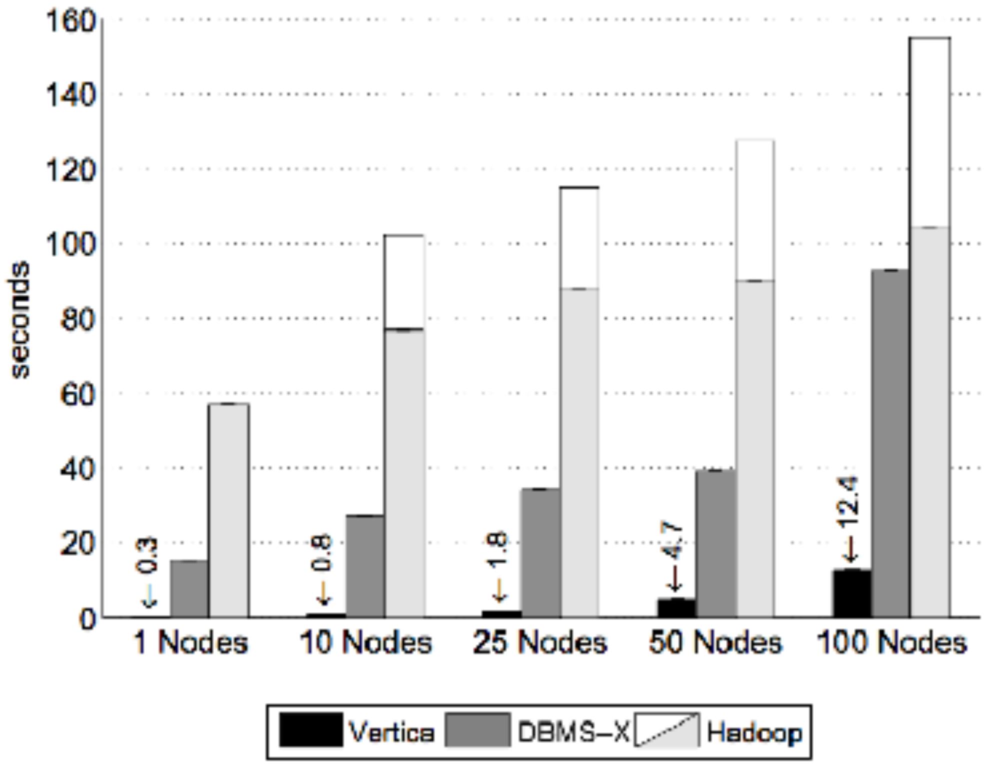


Figure 6: Selection Task Results

Aggregation and Joins

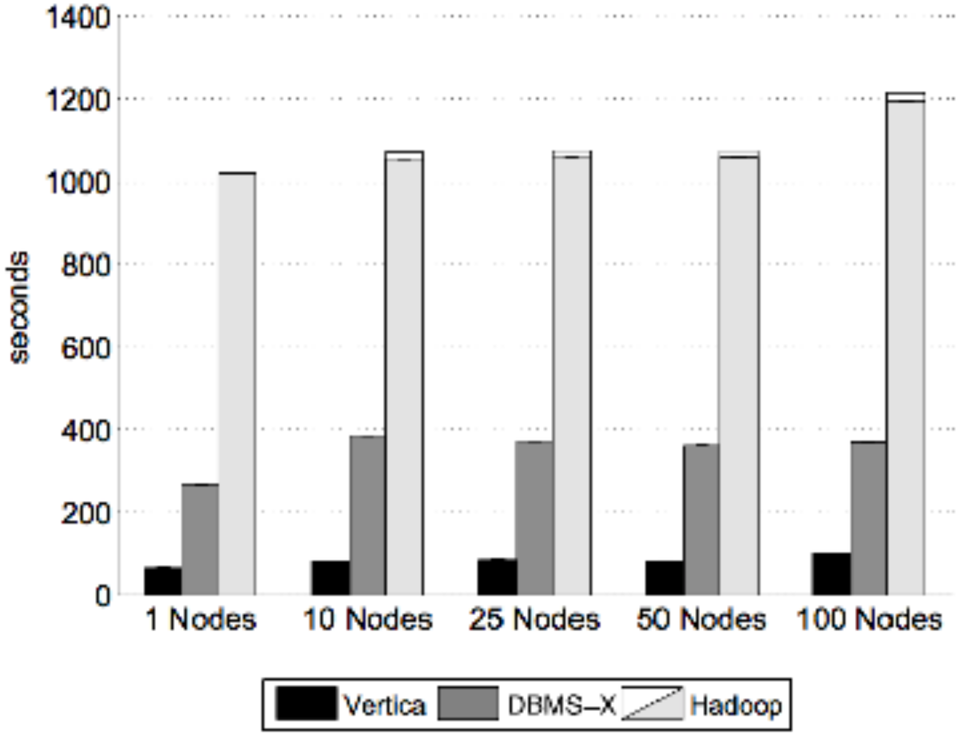


Figure 8: Aggregation Task Results (2,000 Groups)

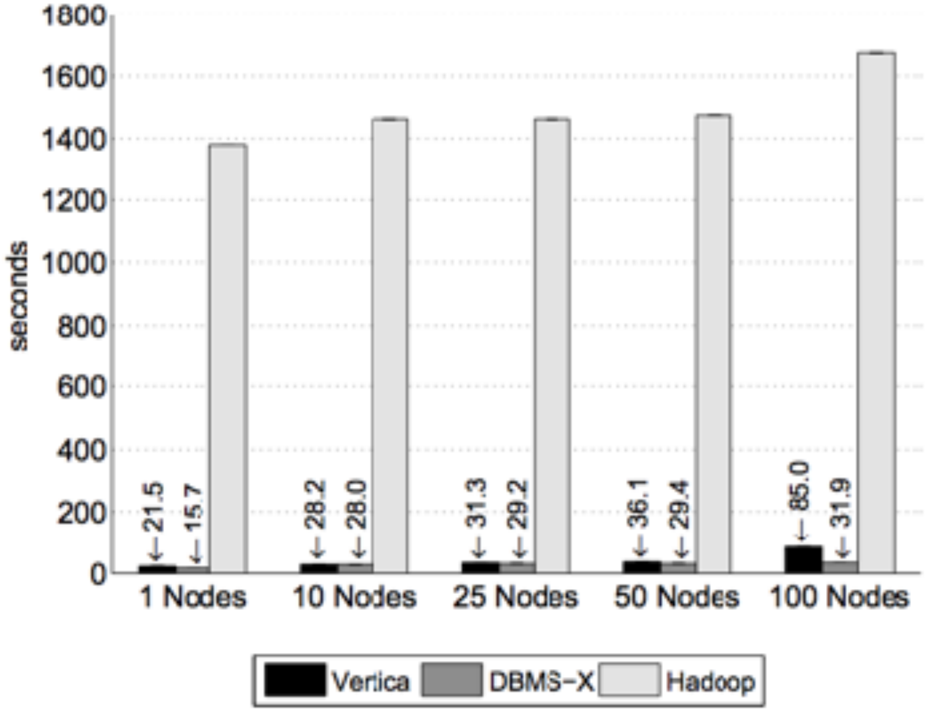


Figure 9: Join Task Results

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.

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

Parallelism

- We've 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
    - Q: pros/cons?
    - Hashing is good to allow for all tuples corresponding to a given value to be found at a node (can aggregate locally if needed), but somewhat susceptible to skew
    - Range-based partitioning is very good to allow for all tuples corresponding to a value or range to be found at a node (can aggregate locally if needed), but very susceptible to skew
    - Round-robin is not susceptible to skew but has no locality benefits. All nodes need to participate in every query.

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?

Parallel Sort

- 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

- Instead, for pass 1: construct *partitioned runs* at each node
  - e.g., node 1 creates runs from [1-100], [101-200]…
- Pass 2: 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 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