COSC 460 Lecture 20: Map Reduce

Instructor: Michael Hay
Fall 2018

Credits: Slides adapted from Franklin, Miklau, and Kot

MapReduce

- MapReduce is two things
  - A programming language abstraction
  - A system for running programs
- Invented by Google, announced publicly in 2004
- Used widely at Google, Facebook, Yahoo!, Microsoft, ...

What is (was?) MapReduce used for?

- At Google:
  - Index building for Google Search
  - Extracting properties from web pages (e.g., geographic locations for local search)
  - Analysis of user query logs (e.g., Google Zeitgeist)
  - Large-scale graph computations
  - Statistical machine translation
- At Yahoo!
  - Spam detection for Yahoo! Mail
- At Facebook (using Hadoop)
  - Data mining
  - Ad optimization
  - Spam detection

slide courtesy of Mike Franklin

Motivating example

- Word frequencies
  - Count often each word occurs on WWW
- Assumption
  - WWW has already been crawled, each web page stored as a file on disk

`word_frequencies.py`

Computations on big data

- How long would it take to count word occurrences for each word in the entire World Wide Web?
  - 20+ Billion pages x 20 KB = 400+ Terabytes
  - 400,000,000,000,000 bytes
  - One disk can read ~128 MB/s
  - 3,125,000 seconds = 36.2 days

Traditional computational model

- Single machine

Big data computational model

- Large numbers of commodity Linux boxes, connected by commodity network

Parallel & distributed

Central challenges

- Parallelism
  - How do we rethink our algorithms to make them parallel?
- Distributed computation
  - How do we deal with the complexities of distributing computation across multiple machines?
- MapReduce is two things
  - A programming abstraction: helps programmers think parallel
  - A system: manages big ugly mess of distributed comp.

Complexities of distributed computation

- **Coordination:**
  - which machines do what work? how and where are results combined?
- **Data distribution:**
  - where is input data located? where does output do go?
- **Load balancing:**
  - how do we distribute work evenly across servers?
- **Fault tolerance:**
  - what happens when one or more machine crash in the middle of execution?

What is hard about distributed computation?

MapReduce: Simplified Data Processing on Large Clusters

by Jeffrey Dean and Sanjay Ghemawat

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large datasets that is amenable to a broad variety of real-world tasks

> "As a reaction to this complexity, we designed a ***new abstraction*** that allows us to express the simple computations we were trying to perform but ***hides the messy details*** of parallelization, fault tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the map and reduce primitives present in Lisp and many other functional languages."
> - Dean and Ghemawat, 2004

tions are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a

interface tailored towards our cluster-based computing environment. Section 4 describes several refinements of the programming model that we have found useful. Section 5 has performance measurements of our implementation for a variety of tasks. In Section 6, we explore the use of MapReduce within Google including our experiences in using it as the basis for a rewrite of our production indexing system. Section 7 discusses related and future work.

2 Programming Model

The computation takes a set of *input* key/value pairs, and produces a set of *output* key/value pairs. The user of the MapReduce library expresses the computation as two functions: map and reduce.

Typical Workflow

- Iterate over a large number of records
- Extract something of interest from each
- Bring together intermediate results
- Aggregate intermediate results
- Generate final output
- (Repeat as needed)

- Most of the real "computation" occurs in the two blue phases

Map Reduce

- A general framework for writing parallel programs that follow the workflow we saw
- Idea:
  - You write the code for the two blue phases
    - *Because that's what is unique to your computation*
  - System takes care of the rest

Typical Workflow

- Iterate over a large number of records
- Map: Extract something of interest from each
- Bring together intermediate results
  - In some standardized way
- Reduce: Aggregate intermediate results
- (Generate final output)
- Repeat as needed

Understanding Map Reduce

- Key to understanding Map Reduce is the third point in workflow
  - How are intermediate results brought together?
  - The framework does it for you, but you have to understand how
- Fundamental idea: key-value model
  - This is the data model for Map Reduce jobs
  - Helps provide a unified interface for bringing results together

Key-Value Data Model

- MapReduce framework can pull data from variety of sources (files, BigTable, RDBMS, etc.)
- Useful to think of any input data source as collection of (key, value) pairs
- Key can be any (hashable) object, value can be any object (e.g., JSON)

- Map reduce program:
  - Input: a collection of (input key, value) pairs
  - Output: a collection of (output key, value) pairs
  - In/out keys can be different!

- Word count example:
  - Input is collection of webpages (key:url, value:html)
  - Output is collection of word counts (key:word, value:frequency)

Programming Model

- Borrows from functional programming
- Users implement two functions ([…] denotes list)

  map (k, v) → [(k', v')]

  - map( ) takes *single* input key-value pair and produces one or more *intermediate* results: (output key, value) pairs
  - after map phase over, system combines all the intermediate values for a given output key together into a list.

  reduce (k', [v']) → [v'']

  - reduce( ) combines intermediate values into one or more final values for that output key

Example: Word Count

```
map(String key, String value):
   // key: document name
   // value: document contents
   for each word w in value:
      EmitIntermediate(w, "1");
```

special function: writes key, value pair out to storage

A mapper utility can apply this in parallel to a whole lot of documents

Now what?

- Have a whole lot of key-value pairs after the map
- Need to bring together: GROUP BY key, and aggregate values
- The *system* does the group by
  - In example: for each word, group consists of list of [“1”, “1”, …]
- The *programmer’s* reduce( ) does the aggregation
  - In example: for each word, compute single final value: the sum

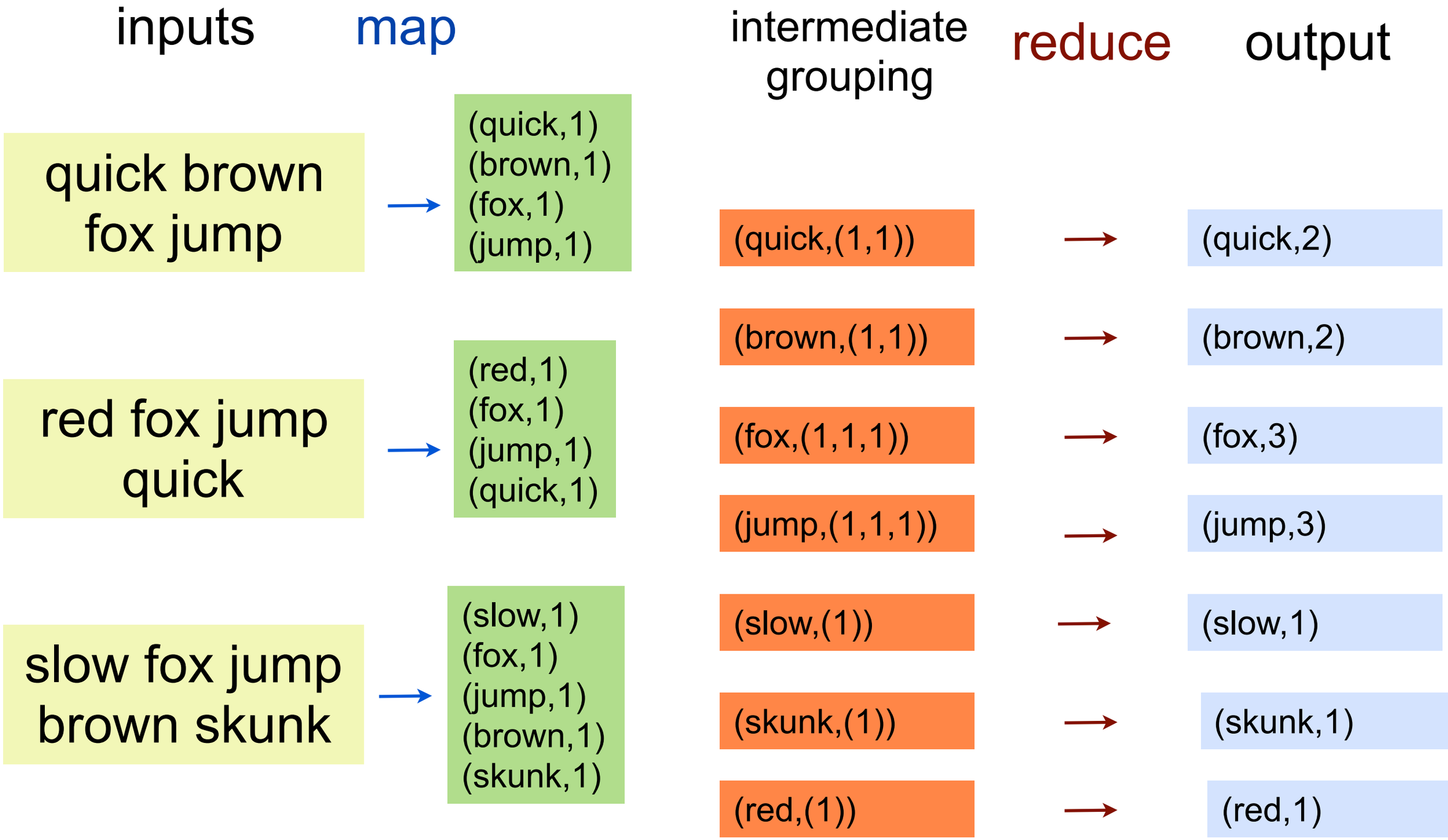
Example: Word Count

```
reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

special function: writes value(s) for given key out to storage

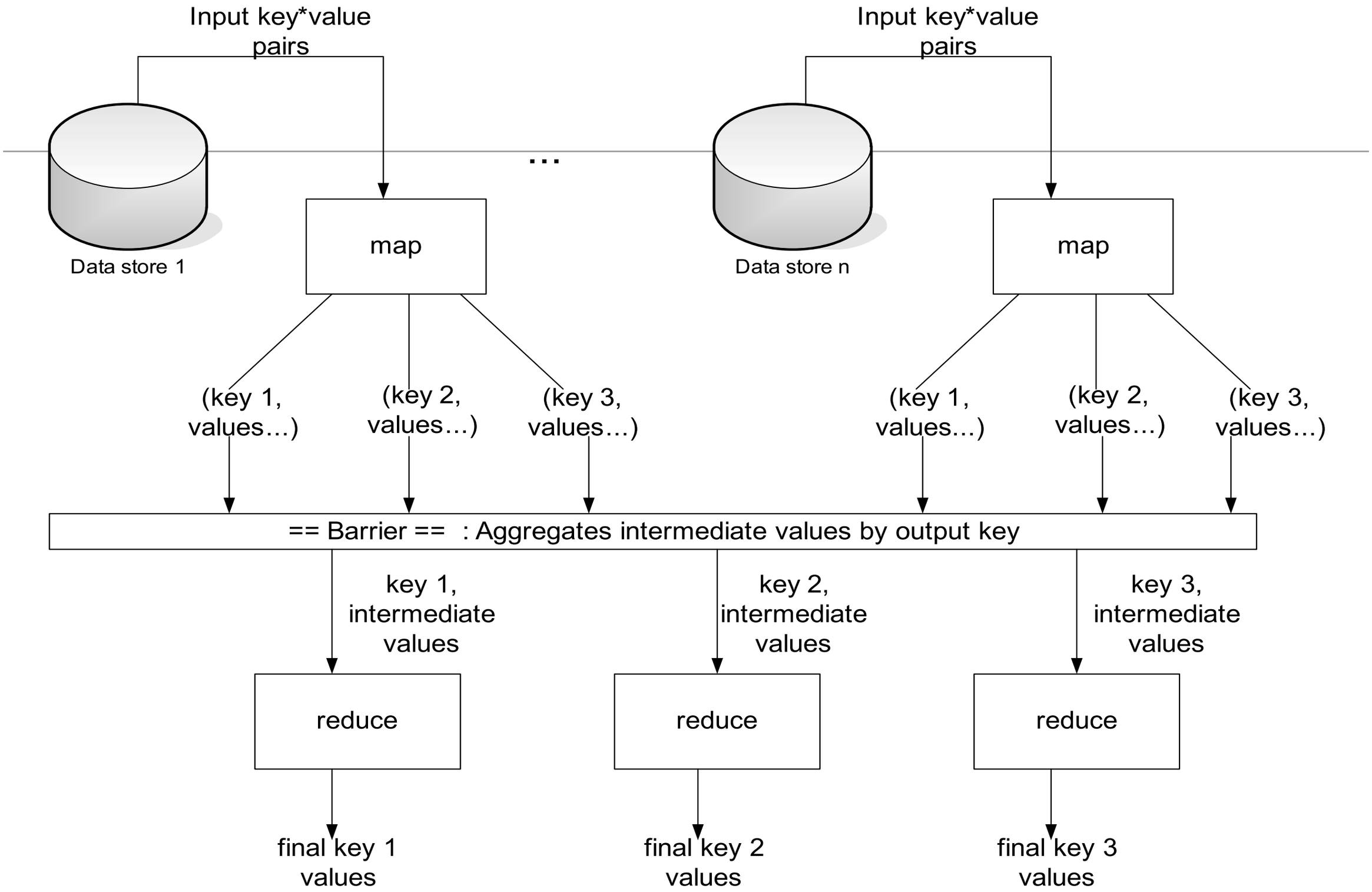
Can also run this in parallel

Concrete example



So what do we have so far?

- See how to write map and reduce functions
  - Work on a key-value model
- Believe that both map and reduce functions can be executed in parallel
- But what about the middle step?
  - Bucketize output of mapper based on value of output key…
- Fortunately, this step is same for all map-reduce programs
  - So map reduce frameworks have it built-in!
  - You don't need to write this logic yourself



Extract (key, value) using map( ); Group By key Apply reduce( )

Map Reduce Programming Model

- You only need to specify two functions:

  map (k, v) → [(k', v')]

  reduce (k', [v']) → [v'']

  […] denotes a list

- Framework takes care of the actual execution:
  - Applies your map function to every initial (k,v)
  - Reshuffles the output of the map to group by k'
  - Applies your reduce function

Parallelism

- map( ) functions run in parallel, creating different intermediate results from different inputs
- reduce( ) functions also run in parallel, each working on a different output key
- All values are processed *independently*
- Bottleneck: reduce phase can’t start until map phase is completely finished.

Examples

- Reverse web graph
  - Input: collection of web pages (url, html)
  - Goal: for each url, the list of pages that link to it
- Map(url, html): for each url’ that occurs in html, emit (url’, url)
- Reduce(url, [urls of pages that link to url]): do nothing!

Examples

- Inverted index
  - Input: collection of web pages (url, html)
  - Goal: for each word, the list of pages containing that word
- Map: for each word, emit (word, url)
- Reduce: accepts a word and list of urls; sort urls, eliminate duplicates.
- Could augment to keep track of word positions

Aside: Google rewrote their web indexing code as a sequence of 24 map-reduce jobs

Exercises

- Input: a collection of documents
  - Key: doc_id, Value: text of document
- Tasks
  1. Across *all documents* total number of words, characters, lines. Example:

doc_id	text
0	quick brown fox jump
1	red fox jump quick
2	slow fox jump brown skunk

key	value
chars	63
lines	3
words	13

Exercises

- Input: a relation of web logs
  - Key: tuple_id, Value: (ipaddr, url, category, timestamp)
- Tasks
  1. Urls that have at least V visits (entries in log)
  2. Categories that have at least S *distinct* urls
  3. Categories that have at least S urls with at least V visits each (hint: may require multiple rounds of map-reduce)

Exercises

- Input: a friends relation *Friend(user, friend)*
  - Key: tuple_id, Value: tuple *(u,f)*
- Tasks
  1. For each user, number of friends
  2. Set of pairs *(u, fof)* where u is a user and fof is a friend of a friend
  3. For each (u,f) pair, the number of mutual friends (hint: may require multiple rounds of map-reduce)

Exercises

- Input: a relation of numbers R*(x)*
  - Key: tuple_id, Value: x
- Tasks
  1. Largest number
  2. select AVG(x) from R
  3. select x, COUNT(x) from R group by x
  4. select count(distinct x) from R