11-442 / 11-642 / 11-742:
Search Engines

Large-Scale Indexes

Jamie Callan
Carnegie Mellon University
callan@cs.cmu.edu

1

Outline

Web-scale search
- Web corpus characteristics
- Computer clusters

Distributed indexes
- Partitioned indexes
- Tiered indexes

Caching
- Queries
- Inverted lists

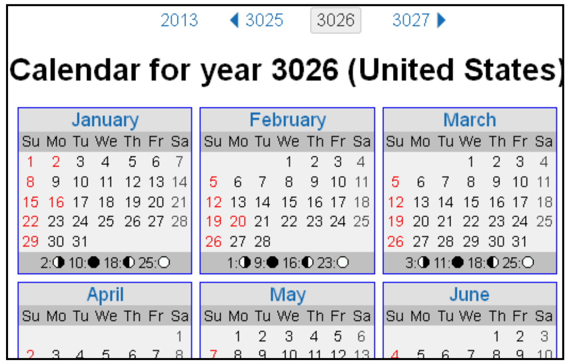
Index construction

2

How Big is a Web Search Engine?

Exact size of the web is difficult to determine

- Count duplicates?
- Count calendar pages
  - An infinite set
- Count obvious spam?
- …

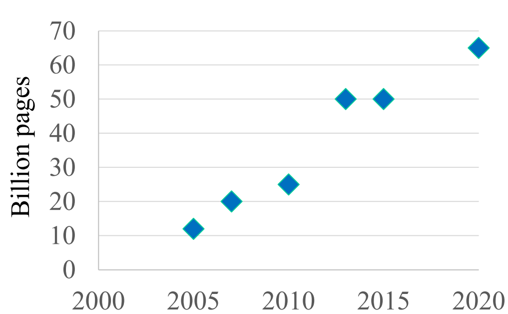


How Big is a Web Search Engine?

Google says that it knows about 130 trillion urls

- No commercial search engine indexes 130 trillion urls

Search engine size estimates



(Bharat and Broder, 2005)
(Baeza-Yates, et al., 2007)
(Yahoo, 2010)
(Greg Grefenstette, Exalead, 2013)
(van den Bosch, et al, 2015)
(*worldwidewebsize.com, 2020*)

Exact size changes and is not important

How Big is a Web Search Engine Index?

Assumptions

- Number of web pages/documents: 100 billion (in 2021)
  - Assume 50% are text-like (guess)
- Average html page size: 40K (37K in 2013 & industry estimates)
- Average inlink size for any page: 1K (guess)
- Index size is about 20% the size of the raw text (15% for HW3)

Text: 50 billion × (40K + 1K) + 50 billion × 1K = 2.1 PB

Index: 20% × 2.1 PB = 420 TB

These are approximations that provide a sense of the scale

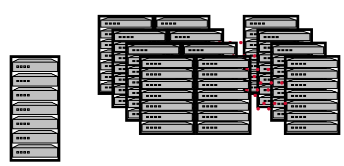
- E.g., the index fits on about 55 8TB disk drives
- 55 enterprise disks × $220/disk = $11,550

The Computing Environment

A Google data center consists of many computer clusters

Computer cluster ("rack")

- Each rack is 40-80 computers
- The rack has its own internal network



Computers

- A small number of ordinary disks on each computer
  - Typically 1-4
- Perhaps some SSD on each computer
- Large (but not huge) RAM on each computer

(Barroso, et al., 2003)

Outline

Web-scale search
- Web corpus characteristics
- Computer clusters

Distributed indexes
- Partitioned indexes
- Tiered indexes

Caching
- Queries
- Inverted lists

Index construction

7

7

A Document-Partitioned Index

The index is too large to fit on any single computer

The index is divided into partitions ("shards")
- Each partition contains a disjoint set of documents
- Each partition is assigned to a machine
- **Example:** 28 partitions × 2 disks/node × 8 TB = 450 TB of index

Many copies of the index are stored ("replication")
- Improved parallelism
- Fault tolerance

(Barroso, et al., 2003)

8

8

A Document-Partitioned Index

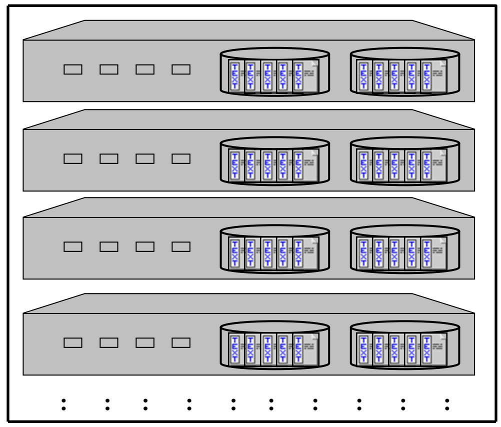
An index server

- 40 machines in a rack
- 2 × 8 TB disks / machine
- 640 TB of storage / rack

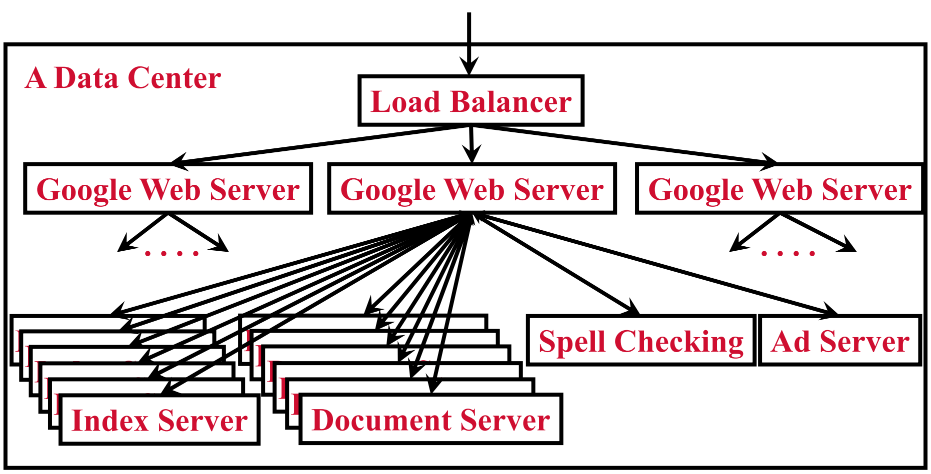
This is a standard architecture

… the details vary

- E.g., machines / rack
- E.g., number of disks
- E.g., size of disks



The Google Query-Serving Architecture



(Barroso, et al., 2003)

A Document-Partitioned Index

How should the corpus be partitioned?

- How should 100 billion documents be assigned to 28 partitions?

Random assignment

- Balances query traffic across partitions, simple update policy

Source-based assignment

- Better compression of inverted lists

Other policies

- …?

11

11

Evaluating a Query on a Document-Partitioned Index

Select an available machine from each pool

- I.e., select a machine for each index partition

Broadcast the query to each selected machine

- Each machine (partition) responds with a ranked list of matches

An aggregator assembles them into a final ranked list of doc ids

- Essentially a simple (and fast) merge-sort based on document score

Other machines looks up titles, URLs, etc., for each result

- A similar partitioning / pooling strategy is used for documents

(Barroso, et al., 2003)

12

12

Outline

Web-scale search
- Web corpus characteristics
- Computer clusters

Distributed indexes
- Partitioned indexes
- Tiered indexes

Caching
- Queries
- Inverted lists

Index construction

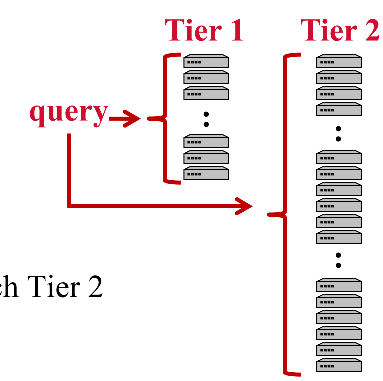
13

13

Tiered Indexes

Most web pages are useless, so why search them all?

Organize the web into two (or more) tiers
- Tier 1 (10%): High-value pages
- Tier 2 (90%): Low-value pages



Search process
- Search Tier 1 first
- If not enough good results are found, also search Tier 2

Can be generalized to additional tiers

14

14

Tiered Indexes: What Goes into the Top Tier(s)?

Pages with high Page Rank, or from sites with high Page Rank

Pages that were important for frequent past queries

- E.g., pages that ranked highly
- E.g., pages that had high click through or high dwell time

Pages with short URLs: More likely to be home pages

Pages with low spam scores: i.e., probably not spam

… your guesses here

15

15

Advantages of Tiered Indexes

Lowers the cost of most searches

- A full search requires 10× more machines than a Tier 1 search

Improves search quality of most searches

- Search focused on "good" pages

16

16

Tiered Indexes

What does it mean for a query to go to Tier 2?

- **Few Tier 1 pages match the query**
  - – Perhaps few pages <u>anywhere</u> on the web match
  - – Perhaps a <u>simpler/faster ranking function</u> could be used
    - » E.g., exact match, with ranking by tf and Page Rank
- **The query is uncommon**
  - – More likely to be an error (and thus discarded)?

17

17

Outline

Web-scale search
- Web corpus characteristics
- Computer clusters

Distributed indexes
- Partitioned indexes
- Tiered indexes

Caching
- Queries
- Inverted lists

Index construction

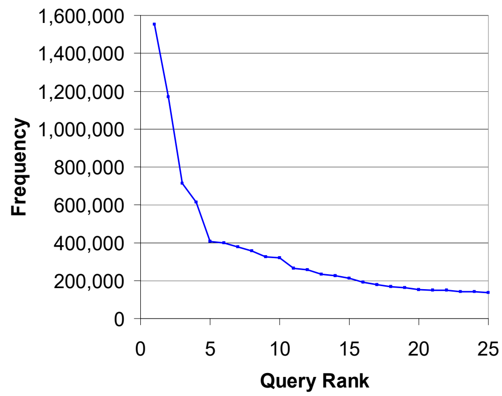
18

18

Evaluating Queries Quickly: Caching of Popular Results

Web traffic is highly skewed

- **A few <u>very</u> popular queries**
  - The top 25 queries are over 1% of the traffic
- **A long tail of rare queries**
- **Among distinct queries**
  - 64% occur once
  - 16% occur twice
  - 7% occur three times
  - 14% occur ≥ 3 times
- **Average query frequency: 4**



1 billion web log entries over 6 weeks

(Silverstein, et al., 2005)

Evaluating Queries Quickly: Caching of Popular Results

Caching recent queries is one way to speed up retrieval

- **In one study, 20-30% of queries had been seen "recently"**
  - Popular topics, re-retrieval, …

Evaluating Queries Quickly: Caching of Popular Results

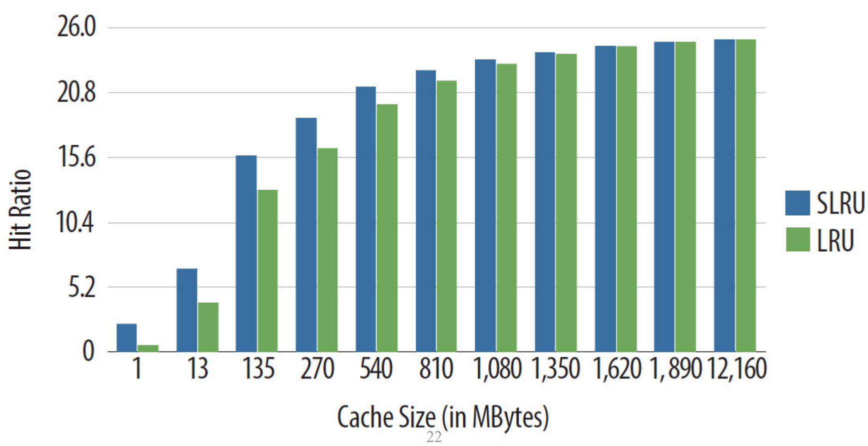
An example architecture

- **Allocate some of RAM to a query cache**
  - Store queries in canonical form
    - » E.g., terms sorted alphabetically
  - A 45 MB cache might store 1,500,000 queries
- **Allocate most of RAM to a result page cache**
  - One page is about 40 KB uncompressed, 13 KB compressed
  - A 20 GB cache might store 1,500,000 compressed result pages
- **Cache misses and hits only use RAM (very fast)**
- **The cache can be partitioned across multiple machines**
  - Cache more queries, but also a more complex design

Evaluating Queries Quickly: Caching of Popular Results

Markatos suggests that 30% of search queries match a cache

- Increasing cache size provides only a small increase in hit rate



(Silvestri, 2011)

Evaluating Queries Quickly: Caching of Popular Results

Why is the hit rate on result caching so low?

In one study (UK2007 query log) …

- 44% of the total query volume is queries that occur once
  - Caching results can't help 44% of the total query volume
- 56% of the total query volume is queries that occur more than once
  - Caching results might help these queries
  - But … it doesn't help the first occurrence of a query
- A cache with infinite memory has a hit ratio less than 50%

30% of Bing queries/day occur once (*Sue Dumais, TREC 2016)*

(Baeza-Yates, et al., 2007)

23

23

Efficient Query Evaluation: Caching Inverted Lists

In one study (UK2007 query log) …

- 4% of all query term occurrences is terms that occur once
  - Caching inverted lists can't help these terms
  - They are 73% of the query term vocabulary
- 96% of all query term occurrences is terms that occur more than once
  - Caching inverted lists might help these query terms

Why do queries and query terms behave differently?

- Query terms can be combined in many ways to produce queries
  - buy iPhone, hack iPhone, iPhone rebates, …
  - American Express, American president, American flag, …

(Baeza-Yates, et al., 2007)

24

24

Efficient Query Evaluation: Caching Inverted Lists

Using some RAM in each partition for an inverted list cache improves performance

Which terms should be cached?

- Prefer terms that are frequent in a query log (*qtf*, "training data")
  - This improves the hit rate
- Prefer terms that don't have massive inverted lists (*df* or *ctf*)
  - Those lists would consume the (limited) cache space

Rank terms by $Score(t) = \frac{qtf(t)}{df(t)}$

- Add terms to the list cache until it is full
- Typically the list cache size is a few GB

(Baeza-Yates, et al., 2007)

Efficient Query Evaluation: Caching of Popular Results and Inverted Lists

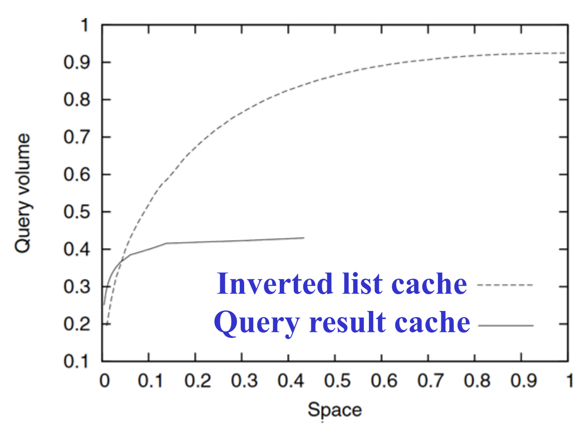
The two caching strategies can be used together

Query result cache

- Smaller
- Saturates faster
- Check first

Inverted list cache

- Larger
- Saturates more slowly
- Check second



(Baeza-Yates, et al., 2007)

Study Guide:
Query Frequency Summary

A large portion of query volume is unique queries

- 30-65%, depending on the study

A few queries are very common

- The top 25 queries are 1% of the query volume
- 20-30% of the volume is queries that have been seen recently
- 50% of the query volume is queries that have been seen before

Similar intuitions and trends as Zipf's Law

- The distribution differs somewhat, but the same intuitions apply

Outline

Web-scale search

- Web corpus characteristics
- Computer clusters

Distributed indexes

- Partitioned indexes
- Tiered indexes

Caching

- Queries
- Inverted lists

Index construction

How Inverted Files are Built: Distributed Processing

How is a partitioned web index built?

There are two main issues

1. The corpus is massive
2. The index needs to be divided into partitions ("shards")

Solution: The MapReduce computing framework

- Transparently map the problem onto multiple processors
- Combine the results into a single result

29

29

Map and Combine

MapReduce is a new use of an old idea in Computer Science

- **Map:** Apply a function to every object in a list
  - Each object is independent
    - » Order is unimportant
    - » Maps can be done in parallel
  - The function produces a results
- **Combine:** Combine the results to produce a final result

You may have seen this in a Lisp or functional programming course

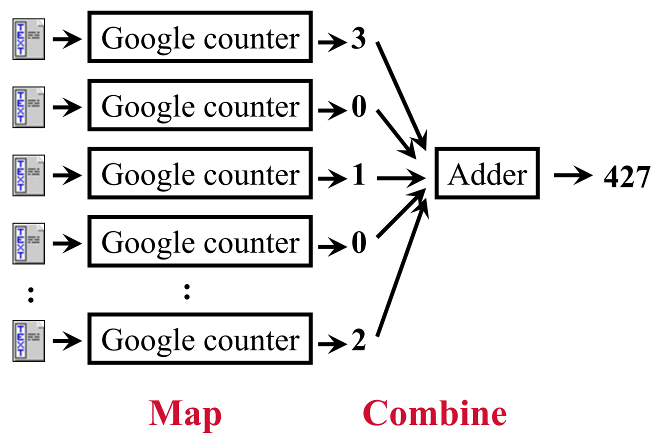
30

30

Map and Combine

Example: Count how often ‘Google’ appears in a set of documents

- Each Map can be on its own processor (if we want)
- Maps can occur in parallel (if we want)
- One Combine (simple case)



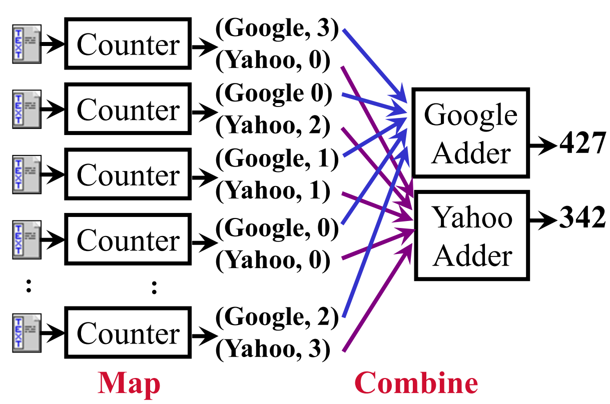
31

31

Map and Combine

Example: Count how often ‘Google’ and ‘Yahoo’ appear in a corpus

- Each Map process produces 2 counts
- Two combine processes
- A more sophisticated method of getting Map output to the right Combine



This is the basic idea in MapReduce

32

32

MapReduce and Hadoop

MapReduce

Developed first, by Google

- Proprietary
- Written in C++
- Reputed to be fast and efficient

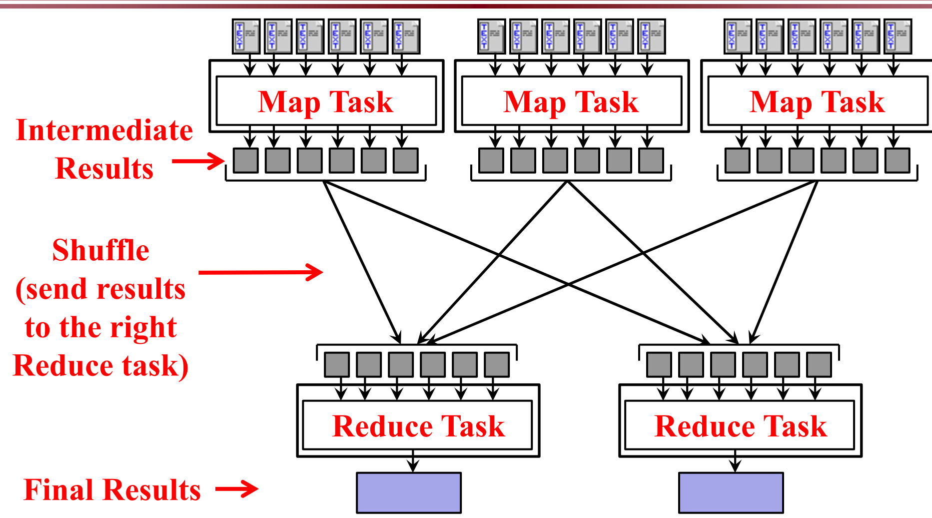
Hadoop

Development led by Yahoo

- Open-source
- Written in Java
- Reasonably fast and efficient
- Used widely

The expression 'MapReduce'
is often used to describe the architecture
not Google's specific implementation of the architecture

Overview of a Map and Reduce Architecture



MapReduce

MapReduce problems are essentially three parts

1. Map: $(k_1, v_1) \rightarrow [(k_2, v_{2,1})\ (k_3, v_{3,1})\ \ldots\ (k_2, v_{2,2})\ \ldots]$

2. Shuffle routes tuples to Reducers

$(k_4, v_{4,3}) \rightarrow \text{Reduce}_1\ \ (k_6, v_{6,4}) \rightarrow \text{Reduce}_2\ (k_4, v_{4,7}) \rightarrow \text{Reduce}_1\ \ldots$

E.g., Use a hash on the key to find the reducer id

3. Reduce: $(k_8, v_{6,3})\ (k_2, v_{2,2})\ (k_2, v_{2,1})\ \ldots \rightarrow (k_2, [v_{2,1}, v_{2,2}, \ldots\ ])\ \ldots$

Aggregate tuples into larger objects

Typically keys (k) are integers or strings

- Although they can be any type of object

Values (v) are often objects

Using MapReduce to Construct Indexes: Preliminaries

The lecture covers construction of binary inverted lists

- The format is (term, [docids]) or (term, [docid, docid, …])
- E.g., (apple, [1, 23, 49, 127, …])
- Binary inverted lists fit on a slide more easily
- Everything also applies to frequency and positional inverted lists

A document id is an internal document id, e.g., a unique integer

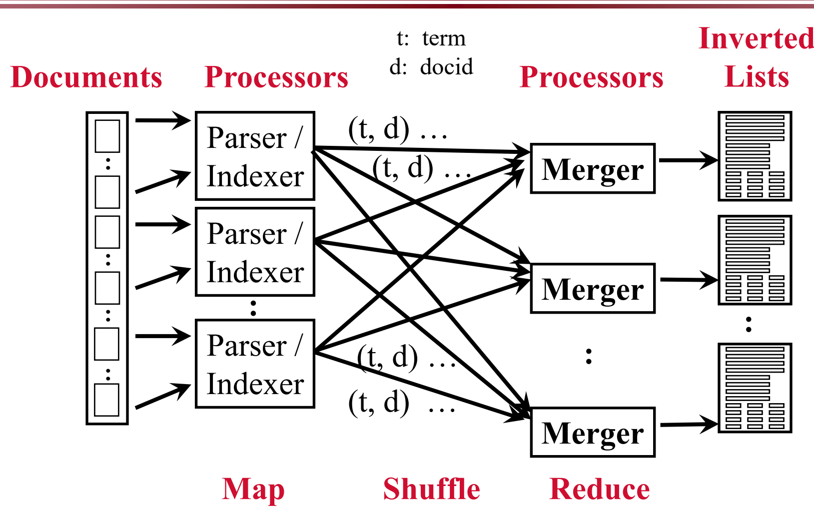
- Not an external document id such as a url

Using MapReduce to Construct Indexes: A Simple Approach (Unrealistic)

A simple approach to creating binary inverted lists

- **Each Map task is a document parser**
  - Input: A stream of documents
  - Output: A stream of (term, docid) tuples
    - » (men, 1) (and, 1) (women, 1) … (once, 2) (upon, 2) …
- **Shuffle routes tuples to Reducers**
- **Reducers convert streams of keys into streams of inverted lists**
  - Input: (men, 1) (men, 127) (men, 49) (men, 23) …
  - The reducer sorts the values for a key and builds an inverted list
  - Output: (men, [df:492, docids:1, 23, 49, 127, …])

Using MapReduce to Construct Indexes: A Simple Approach (Unrealistic)



Using MapReduce to Construct Indexes: A Simple Approach (Unrealistic)

A more succinct representation of the previous algorithm

- **Map:** $(\text{docid}_1, \text{content}_1) \rightarrow (t_1, \text{docid}_1)\ (t_2, \text{docid}_1) \ldots$
- **Shuffle** by t
- **Reduce:** $(t_5, \text{docid}_1)\ (t_1, \text{docid}_3) \ldots \rightarrow (t_3, \text{ilist}_3)\ (t_1, \text{ilist}_1) \ldots$

docid: a unique integer
t: a term, e.g., "apple"
ilist: a complete inverted list

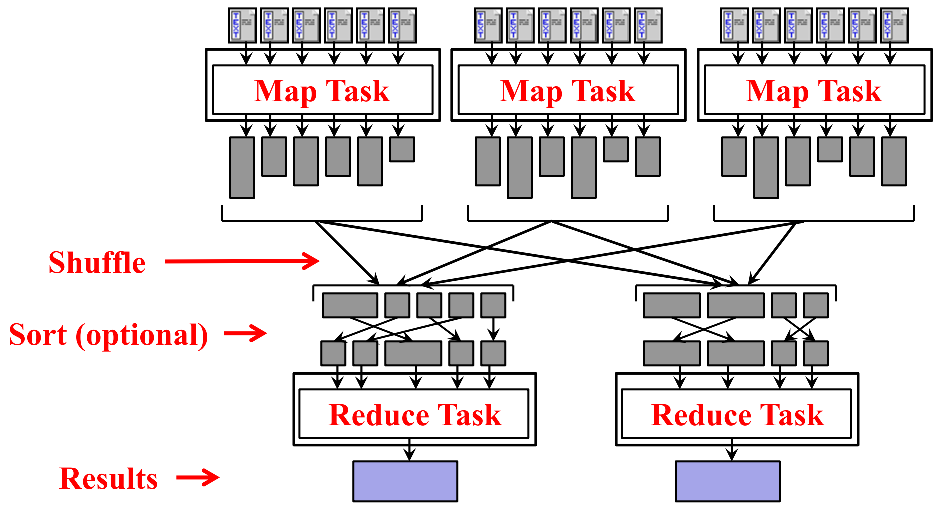
This works, but it is inefficient

Using MapReduce to Construct Indexes: A More Advanced Approach

Efficiency can be improved in several ways

- Map tasks produce partial inverted lists instead of docids
  - (t ilist) instead of (t d)
  - Similar to the single-processor architecture
  - Fewer shuffle operations (fewer data movement operations)
- Reduce input stream is sorted before the Reduce task receives it
  - Each Reduce receives an ordered stream of keys

Overview of a Map and Reduce Architecture



41 © 2021, Jamie Callan

41

Using MapReduce to Construct Indexes: A More Advanced Approach

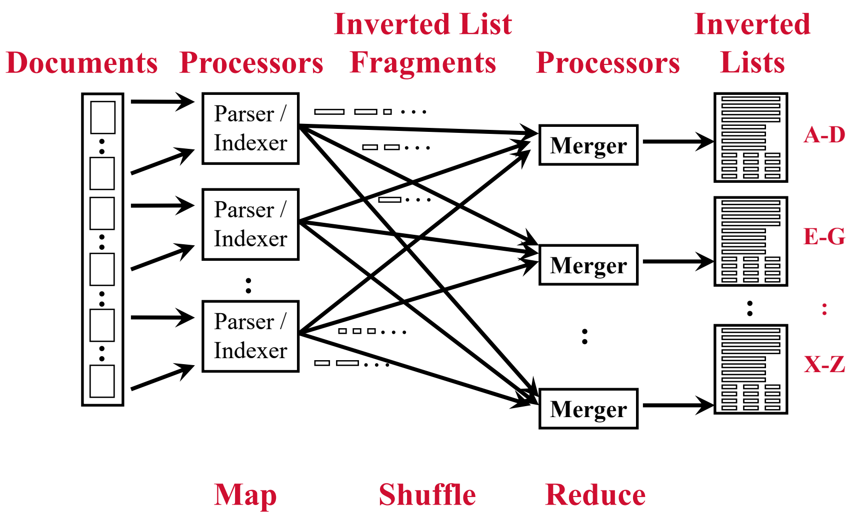
- **Map:** (docid$_1$, content$_1$) → (t$_1$, ilist$_{1,1}$) (t$_2$, ilist$_{2,1}$) (t$_3$, ilist$_{3,1}$) …
  - Each output inverted list covers a sequence of documents
- **Shuffle** by t
- **Sort** Reducer input by t
  (t$_4$, ilist$_{4,1}$) (t$_1$, ilist$_{1,3}$) … → (t$_1$, ilist$_{1,2}$) (t$_1$, ilist$_{1,4}$) (t$_4$, ilist$_{4,1}$) …
- **Reduce:** (t$_1$, [ilist$_{1,2}$, ilist$_{1,1}$, ilist$_{1,4}$, …]) → (t$_1$, ilist$_{final}$)

ilist$_{i,j}$: the j'th inverted list fragment for term i

42 © 2021, Jamie Callan

42

Using MapReduce to Construct Indexes: A More Advanced Approach



Using MapReduce to Construct Partitioned Indexes

Remember that large systems use partitioned indexes

- Each partition ("index shard') covers a different set of documents
- Thus, each partition has its own inverted list for 'apple'
- Where do they come from?

Option 1: Build a full 'apple' inverted list', then break it into pieces

- Effective, but inefficient

Option 2: Build a different 'apple' inverted list for each partition

- Easy to accomplish with Map/Reduce
- Keys become [p, t] instead of t

  p: partition id   t: term

Using MapReduce to Construct Partitioned Indexes

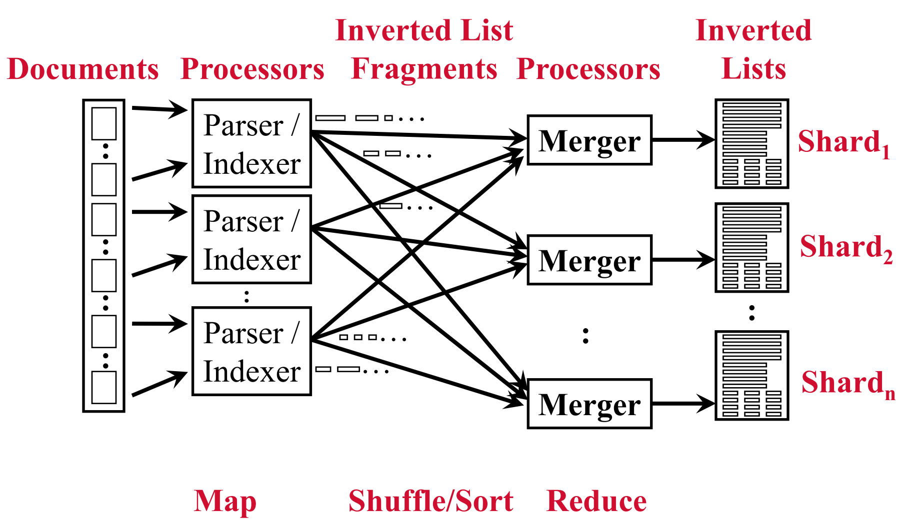
- **Map:** ($docid_1$, $content_1$) → ([p, t_1], $ilist_{1,1}$)
- **Shuffle** by p
- **Sort** by [p, t]
- **Reduce:** ([p, t_1], [$ilist_{1,2}$, $ilist_{1,1}$, $ilist_{1,4}$, …]) → ([p, t_1], $ilist_{final}$)

p: partition (shard) id

The Map process assigns each document to a partition p

- Note: it uses a more complex key, e.g., [$partition_{21}$, "apple"]

Using MapReduce to Construct Partitioned Indexes



MapReduce as a Programming Paradigm

MapReduce makes it easier for you work on massive datasets

- Most of your software uses only small pieces of the dataset
- More computers can be applied to the task

MapReduce simplifies distributed data processing

- Task scheduling, data movement, sorting, etc., are provided for you
- Transient failures and hardware failures are handled for you

Outline

Web-scale search

- Web corpus characteristics
- Computer clusters

Distributed indexes

- Partitioned indexes
- Tiered indexes

Caching

- Queries
- Inverted lists

Index construction

For More Information

- L. A. Barroso, J. Dean, and U. Hölzle. "Web search for a planet: The Google cluster architecture." *IEEE Micro*, 2003.
- R. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, F. Silvestri. The Impact of caching on search engines. SIGIR 2007, pp. 183-190. 2007.
- A. van den Bosch, T. Bogers, and M. de Kunder. "A longitudinal analysis of search engine index size." *The 15th International Conference on Scientometrics & Informetrics*. 2015.
- J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters." *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI 2004)*, pages 137-150. 2004.
- S. Ghemawat, H. Gobioff, and S.-T. Leung. "The Google File System." *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP-03)*, pages 29-43. 2003.
- A. Ntoulas and J. Cho. Pruning policies for two-tiered index with correctness guarantee. SIGIR 2007. 2007.
- F. Silvestri. "Mining query logs: Turning search usage data into knowledge." Foundations and Trends in Information Retrieval, 4(1-2). 2010.
- I.H. Witten, A. Moffat, and T.C. Bell. *Managing Gigabytes*. Morgan Kaufmann. 1999.
- J. Zobel and A. Moffat. "Inverted files for text search engines." *ACM Computing Surveys*, 38 (2). 2006.
- http://hadoop.apache.org/common/docs/current/mapred_tutorial.html. "Map/Reduce Tutorial". Fetched January 21, 2010.