COMP 633 - Parallel Computing

Lecture 23
November 18, 2021

Datacenters and Large Scale Data Processing

Announcements

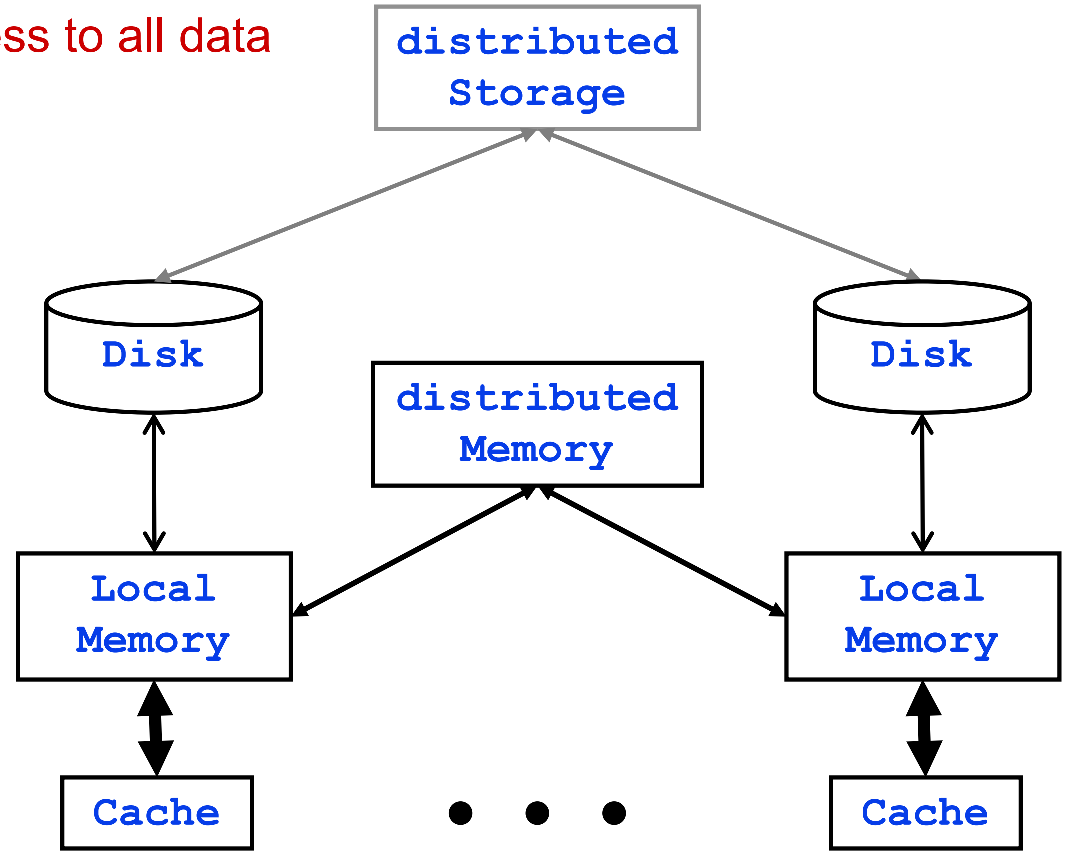
- **Written assignment 2**
  - due Tue Nov 23 at the start of class
- **Programming assignment 2**
  - due Tue Nov 30 (last day of class)
- **Final exam**
  - Saturday Dec 6, 4pm – 7pm in SN011

Topics

- **Parallel memory hierarchy**
  - extend to include disk storage
- **Google web search**
  - Large parallel application
  - Distributed over a large cluster
- **Programming models for large data collections**
  - MapReduce
  - Spark

Extending the parallel memory hierarchy

- **Incorporate disk storage**
  - Parallel transfers to disks
  - Global access to all data



Google data processing

- **Search and other services require parallel processing**
  - search processing and/or query rate are too large for a single machine
- **Data storage requires replication**
  - to tolerate and recover from storage errors
  - for parallel throughput
  - to reduce latency
- **multiple datacenters around the world**
  - to reduce latency and long-haul traffic
  - to tolerate network or power failures or bigger disasters

Google web search – 2002

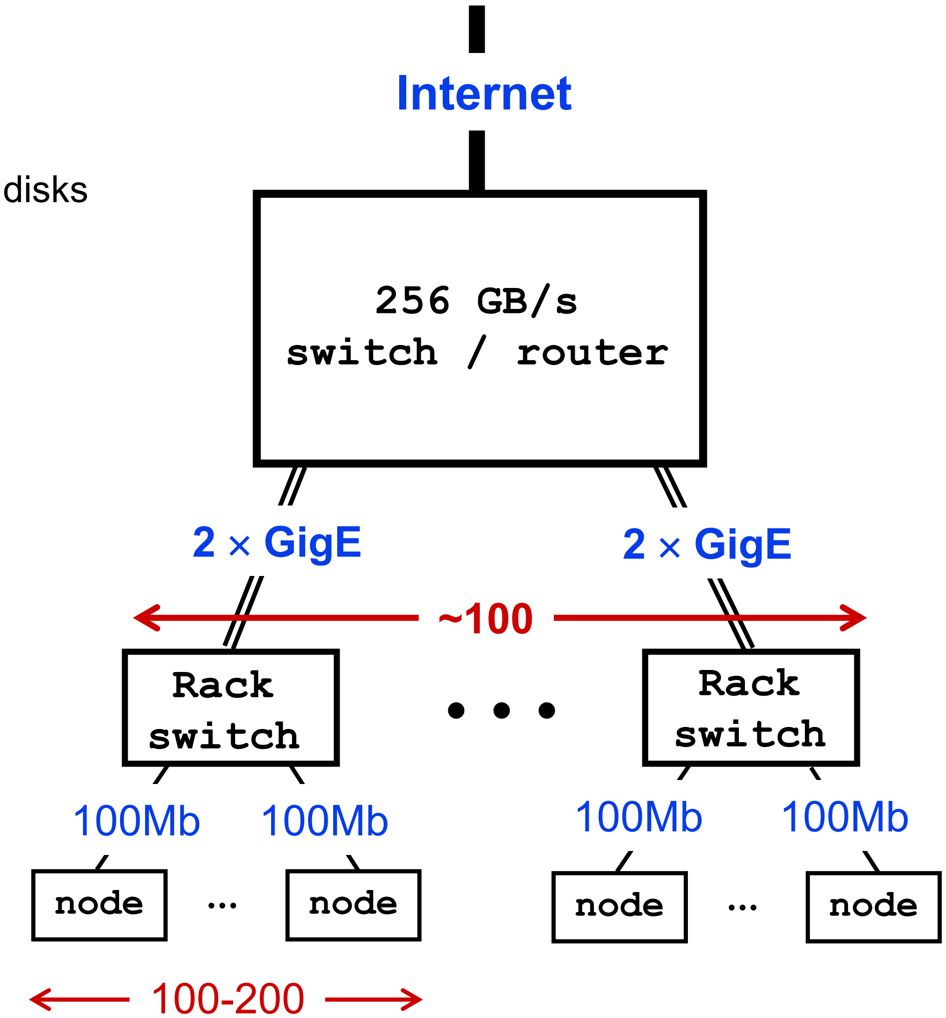
- **web statistics (2002)[1]**
  - 3+ Billion static web pages
    - » doubles every 8 months (2012: 1 Trillion pages)
  - 30% duplication

- **Google usage statistics (2002)[1,2]**
  - 260 million users
    - » 80% do searches
  - 150 million searches/day (2020: 5.8 billion searches/day, 70,000 searches/sec)
    - » ~2000 queries/sec average
    - » ~1000 queries/sec minimum
  - query response time
    - » less than 0.25 secs typical
    - » target 0.5 secs max
  - uptime
    - » target 100%

Sources: [1] Monika Henzinger, “Indexing the Web: A challenge for supercomputing”, invited talk, ISC 2002 Heidelberg, June 2002.
[2] Urs Hoelzle, “Google Linux Cluster”, Univ Washington Colloquium, November 2002.

Google Linux Cluster (2002)

- **Overview**
  - 15,000+ PC cluster
    - » 5 PB disk storage
      - 5×10^{15} bytes = 50,000 × 100GB disks
  - node
    - » 100 Mb Ethernet
    - » 1-4 100 GB disks
    - » mid-range processor (P III)
    - » 256 MB - 2GB memory
    - » runs Linux
  - rack
    - » 100 to 200 nodes
    - » Ethernet switch
  - router/switch
    - » serves ~100 racks
    - » distributes search requests

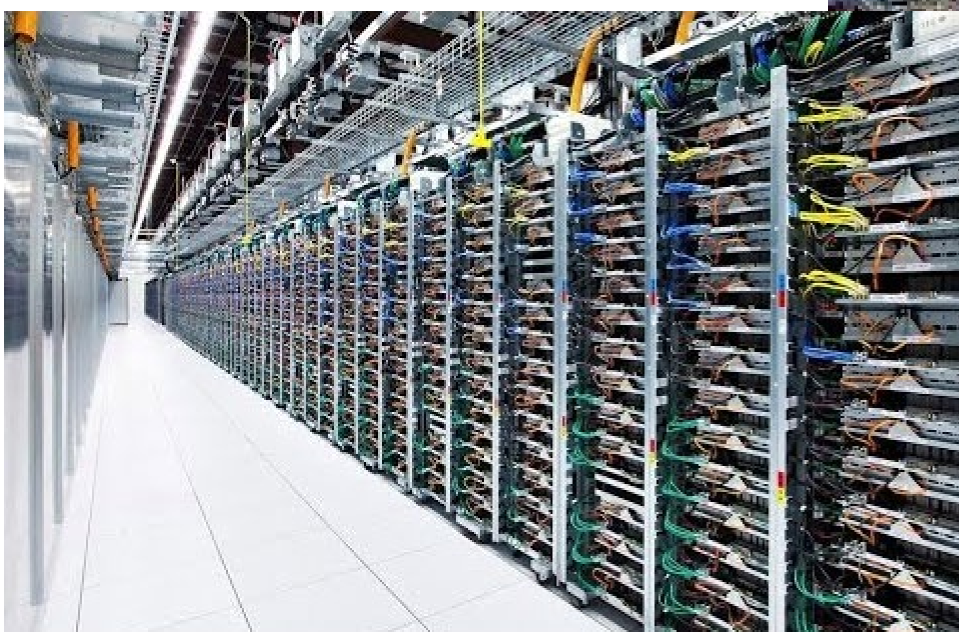


Google Web Search: 2010 vs. 1999*

▪ **# docs:** tens of millions to tens of billions	~1,000 x
▪ **queries processed/day:**	~1,000 x
▪ **per doc info in index:**	~3 x
▪ **update latency:** months to tens of secs	~50,000 x
▪ **avg. query latency:** <1s to <0.2s	~5 x
▪ **more machines * faster machines**	~1,000 x

* Jeff Dean - Building Software Systems at Google and Lessons Learned

Google data centers



The Joys of Real Hardware*

Typical first year for a new cluster:

- **~1 network rewiring** (rolling: ~5% of machines down over 2-day span)
- **~20 rack failures** (40-80 machines instantly disappear, 1-6 hours to get back)
- **~5 racks go wonky** (40-80 machines see 50% packet loss)
- **~8 network maintenances** (4 might cause ~30-minute random connectivity loss)
- **~12 router reloads** (takes out DNS and external vips for a couple minutes)
- **~3 router failures** (have to immediately pull traffic for an hour)
- **~dozens of minor 30-second blips for dns**
- **~1000 individual machine failures**
- **~thousands of hard drive failures** slow disks, bad memory, misconfigured machines, flaky machines, etc.
- **Long distance links:** wild dogs, sharks, dead horses, drunken hunters, etc.

* Jeff Dean - Building Software Systems at Google and Lessons Learned

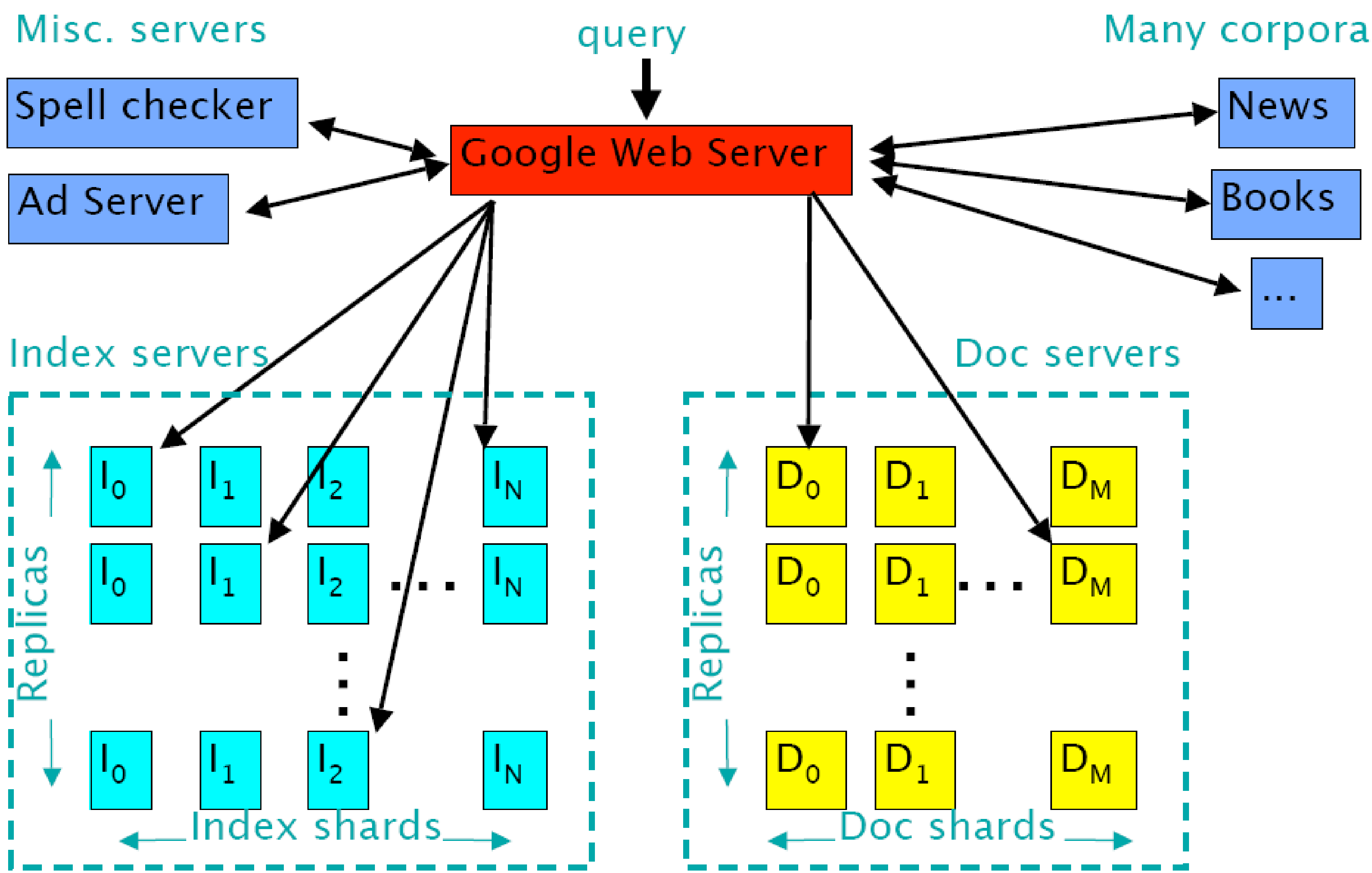
Google query processing steps (simplified)

1. **secret sauce to map query to search terms**
   – detect query language + fix spelling errors
2. **locate search terms in dictionary**
   – over 100 M words in dictionary per language
3. **for each search term in dictionary**
   – use inverted index to locate web pages containing term
   – ordered by page number
4. **compute and order pages satisfying the query**
   – explicit rules
      » conjunction, disjunction, etc. of terms
      » document language
   – implicit rules
      » search term proximity in documents
      » location of search terms in document structure
      » quality of page – PAGE RANK
5. **Construct synopsis reports from documents in order**
   – extract page from cache and highlight search terms in context
   – 10 results returned per query

Challenges

- **Query processing**
  - how to distribute data structures?
    - » dictionary
    - » inverted index
    - » web pages
  - how to implement query processing algorithms?
- **Fault tolerance**
  - component count is very large
    - » 10,000 servers with 3 year MTBF, expect to lose ten a day
    - » 50,000 disks with 10% failing per year is a disk failure every couple of hours
    - » 10^{-15} undetected bit error rate on I/O is ~50 incorrect bits in 5PB copy
- **Scaling**
  - how can the system be designed to scale with
    - » increasing number of queries
    - » increasing size of web (number of pages and total text size)
    - » increasing component failures (as a consequence of scaling up)

Google server architecture



When designing large distributed applications

- **"Numbers Everyone Should Know" - Jeff Dean**

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	100 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	10,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from network	10,000,000 ns
Read 1 MB sequentially from disk	30,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

Processing large data sets

- **Process data distributed across thousands of disks**
  - Large datasets pose an I/O bottleneck
    - » Attach disks to all nodes
    - » Stripe data across disks
    - » How to manage this?

- **MapReduce provides**
  - Parallel disk bandwidth
  - Automatic parallelization & distribution
  - Fault tolerance
  - I/O scheduling
  - Monitoring & status updates

MapReduce

- **MapReduce**
  - parallel programming schema
  - name inspired by functional language view of the schema
- **Many problems can be approached this way**
- **Easy to distribute across nodes**
- **Simple failure/retry semantics**

Map in Lisp (Scheme)

- **(map** *f list [list*$_2$ *list*$_3$ *…]***)**

- **(map square ‘(1 2 3 4))**

  (1 4 9 16)

- **(reduce + ‘(1 4 9 16))**

  (+ 16 (+ 9 (+ 4 1)))

  = 30

Map/Reduce a la Google

- **An input file contains a large list of items**
  - Each item is a (key,val) pair
  - The file is distributed across disks on p nodes

- **map(key, val) is run on each item in the list**
  - emits new-key / new-val pairs
  - map: $(k_1,v_1) \rightarrow list(k_2,v_2)$

- **reduce(key, vals) is run for each unique key emitted by map()**
  - reduce: $(k_2, list(v_2)) \rightarrow list(v_2)$
  - the result is written to a file distributed across disks attached to the nodes

Example 1: count words in docs

- Input consists of (url, contents) pairs

- map(key=url, val=contents):
  - » For each word w in contents, emit (w, “1”)

- reduce(key=word, values=uniq_counts):
  - » Sum all “1”s in values list
  - » Emit result “(word, sum)”

Count words - example

map(key=url, val=contents):

For each word *w* in contents, emit (w, “1”)

reduce(key=word, values=uniq_counts):

Sum all “1”s in values list

Emit result “(word, sum)”

see bob throw

see spot run

see	1
bob	1
run	1
see	1
spot	1
throw	1

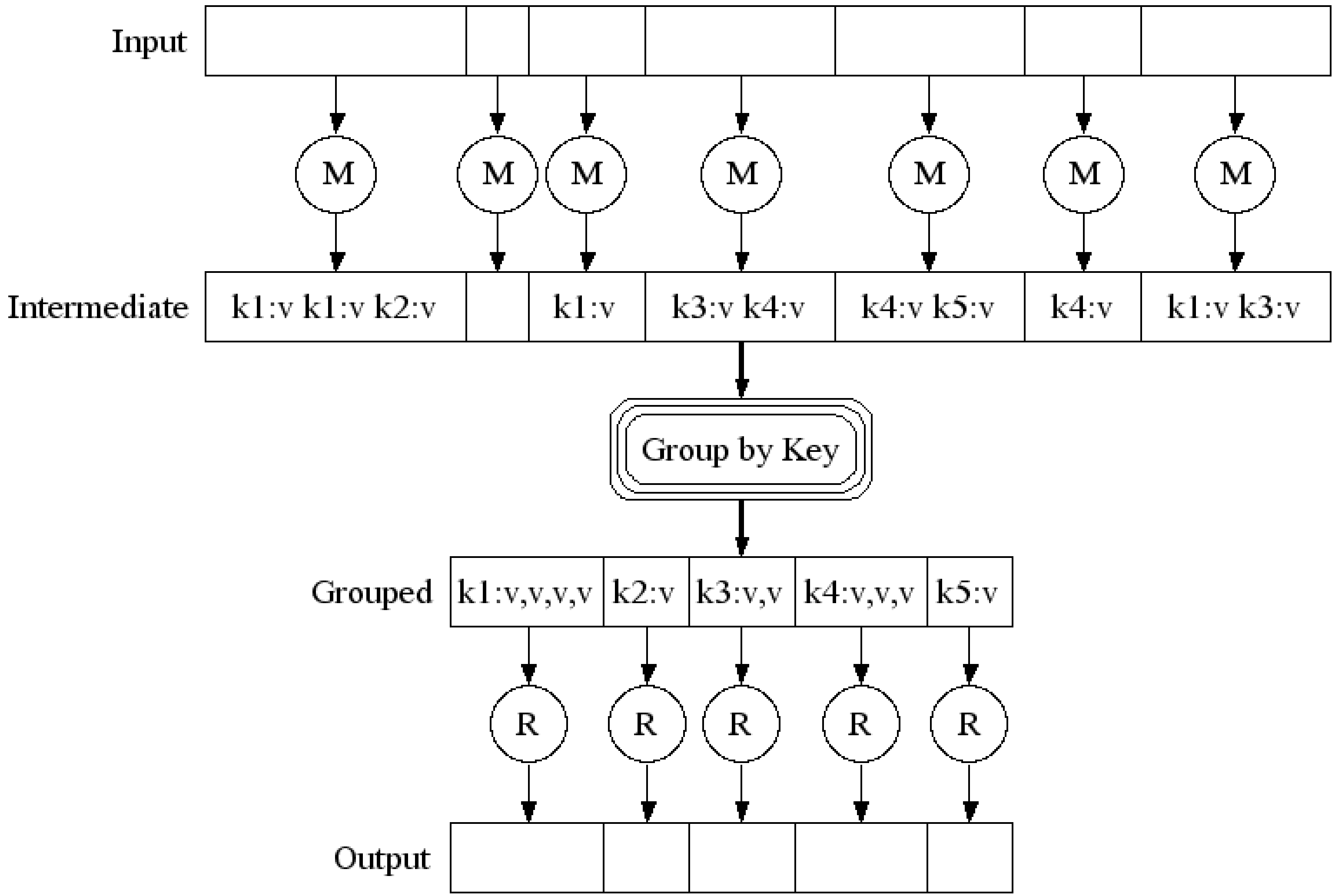
bob	1
run	1
see	2
spot	1
throw	1

Execution

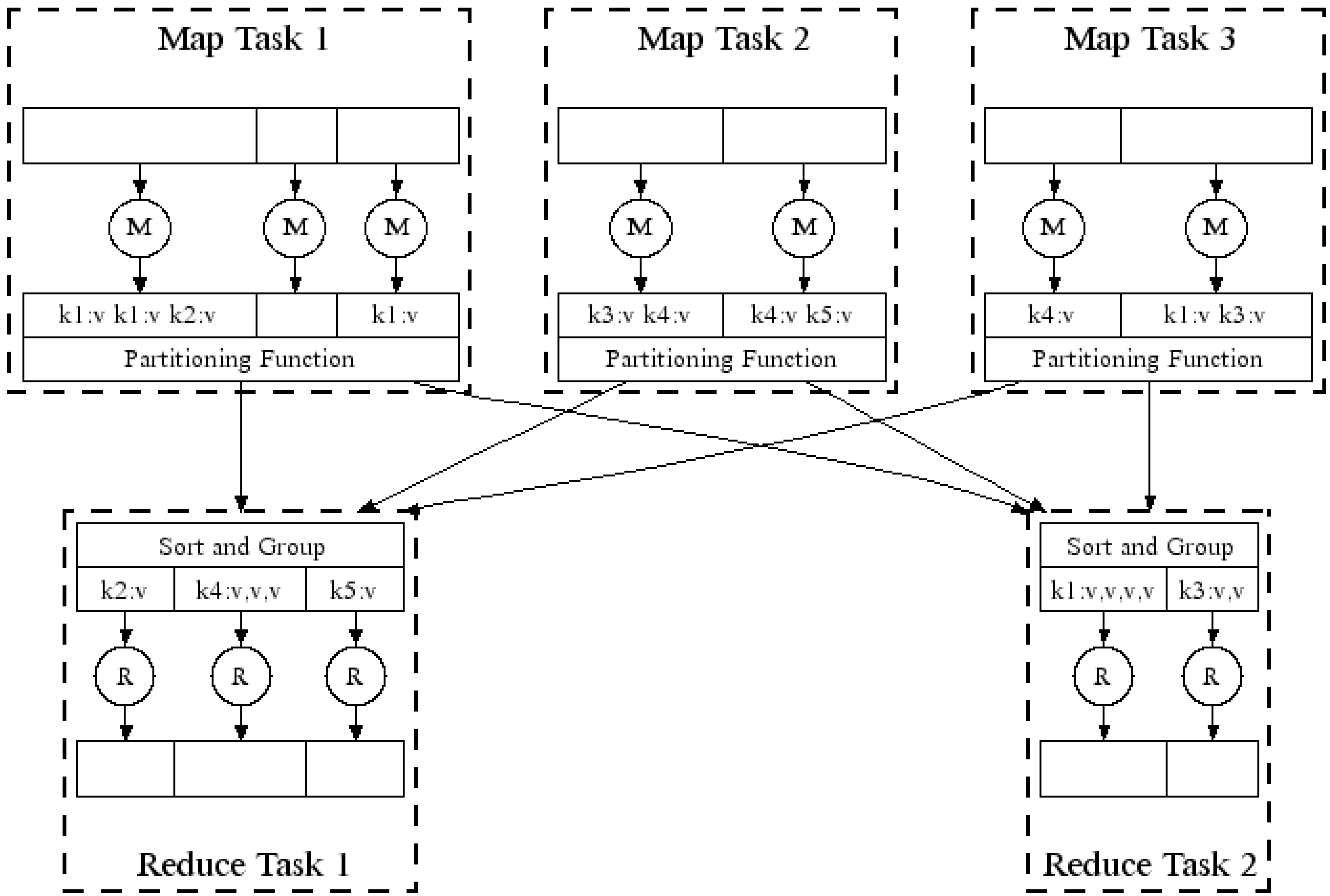
- **How is this distributed?**
  1. Partition input key/value pairs into chunks, run map() tasks in parallel
  2. After all map()s are complete, consolidate all emitted values for each unique emitted key
  3. Now partition space of output map keys, and run reduce() in parallel

- **If map() or reduce() fails, re-execute!**

Execution


Input
M
M
M
M
M
M
M
Intermediate
k1:v k1:v k2:v
k1:v
k3:v k4:v
k4:v k5:v
k4:v
k1:v k3:v
Group by Key
Grouped
k1:v,v,v,v
k2:v
k3:v,v
k4:v,v,v
k5:v
R
R
R
R
R
Output

Parallel Execution



Experience (10-15 years ago)

- **Rewrote Google's production indexing system using MapReduce**
  - Set of 10, 14, 17, 21, 24 MapReduce operations
  - New code is simpler, easier to understand
    - » 3800 lines C++ → 700
  - MapReduce handles failures, slow machines
  - Easy to make indexing faster
    - » add more machines

- **Redux**
  - MapReduce proved to be useful abstraction
  - MapReduce has an open-source implementation
    - » Hadoop
  - Extensively used with large datasets
    - » E.g. bioinformatics
    - » focus on problem
    - » let library deal w/ messy details

Improving MapReduce

- **Problem: All computation is disk to disk**
  - No notion of locality
- **Alternate approach: Spark**
  - System for expressing computations on objects distributed on disks
  - Computations are moved to data instead of vice-versa
    - » disk data streamed into node memory
    - » data flow model applies multiple processing steps in memory
      - Decreases number of map/reduce steps
      - Best performance when data fita in memory
  - MapReduce has better fault tolerance
  - Spark has superior performance
    - » also more flexible in processing languages and tools
- **Applications**
  - MapReduce
    - » linear processing of ultra-large datasets
  - Spark
    - » real-time analytics, graph processing, SparkSQL, machine learning

Apache Hadoop

- **Hadoop**
  - open source distributed file system (HDFS)
  - processing layer
    - » MapReduce, in Java (other languages supported)
    - » Spark