SCC.311: Infrastructure at Google

Prof. Barry Porter

Overview

- Google has developed a massive-scale hardware and software infrastructure capable of serving millions of users per minute
  - The way in which this was built was pioneering at the time in its use of cheap, commodity hardware for their server population
  - The use of this kind of hardware has also dictated their software designs, which are built to expect frequent failures on a daily basis

Designing for scalability

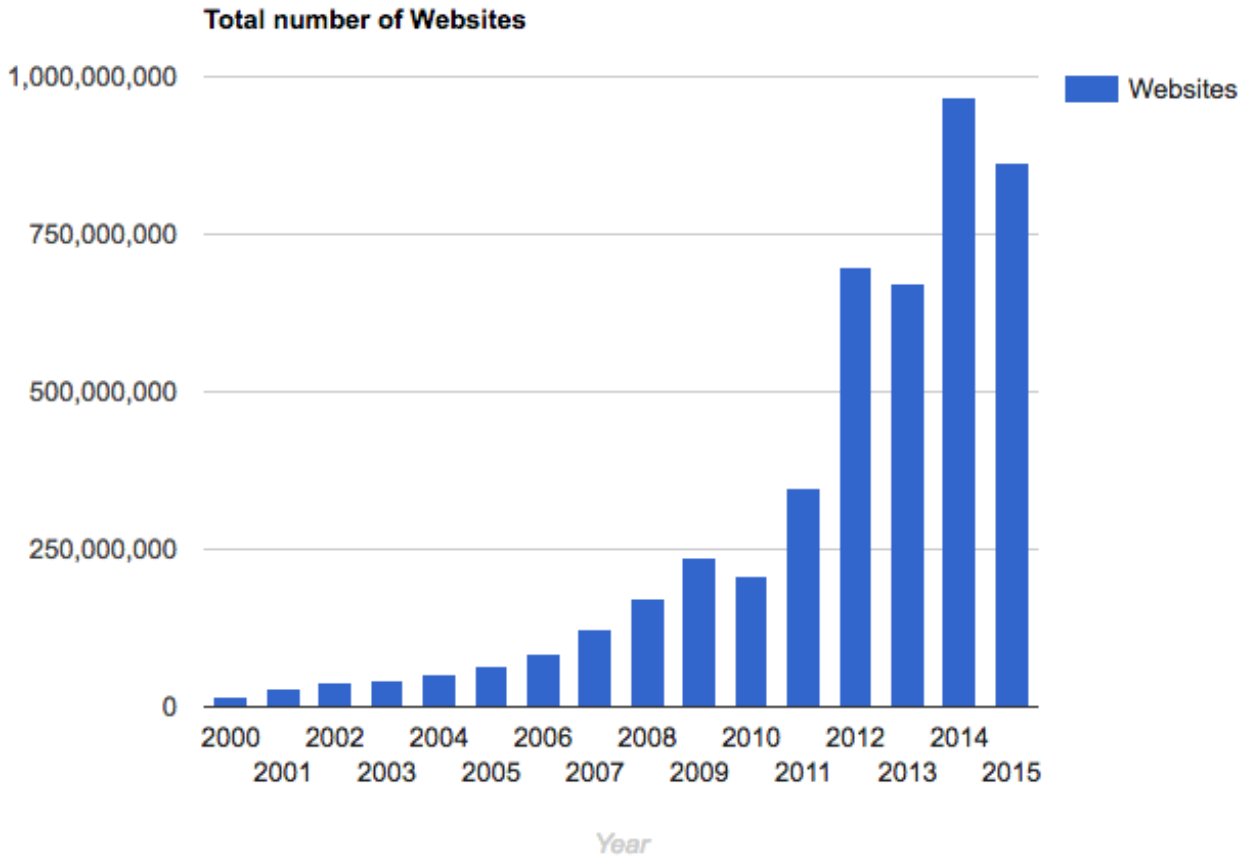
- Scalability with respect to size
  - e.g. support massive growth in the number of users of a service
- Scalability with respect to geography
  - e.g. supporting distributed systems that span continents (dealing with latencies, inconsistency, etc.)
- Scalability with respect to administration
  - e.g. supporting systems which span many different administrative organisations

Designing for scalability

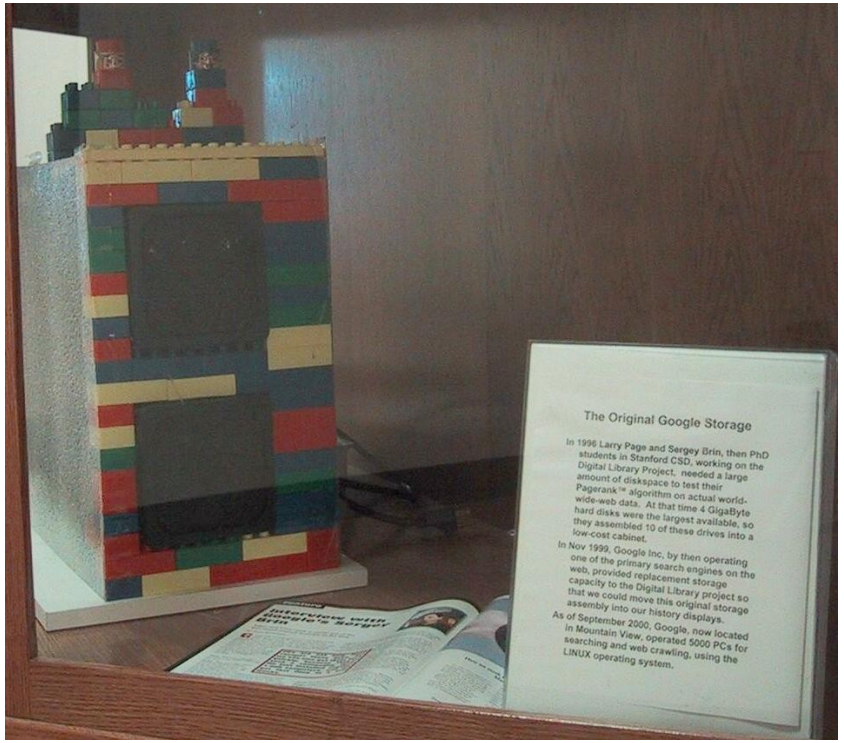
- About Google
  - World's leading search engine
    - market share ~75% on desktop, ~94% on mobile/tablet
  - Increasing focus on cloud computing

- Another view
  - World's largest and most ambitious distributed system with around:
    - 900,000 machines (a conservative estimate in 2011)
    - 2.5 million machines (Gartner report, 2016)
  - Extremely challenging requirements in terms of:
    - Scalability
    - Reliability
    - Performance
    - Openness

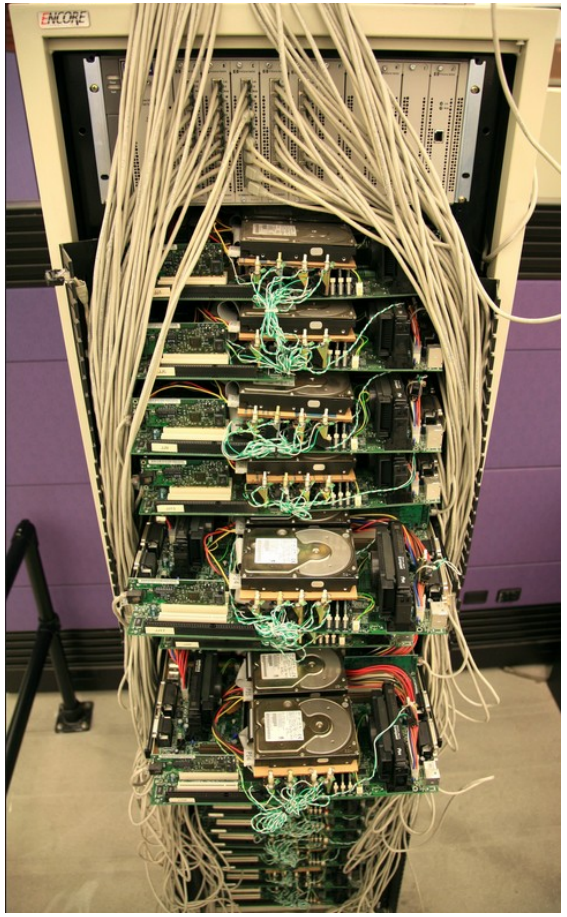
Designing for scalability

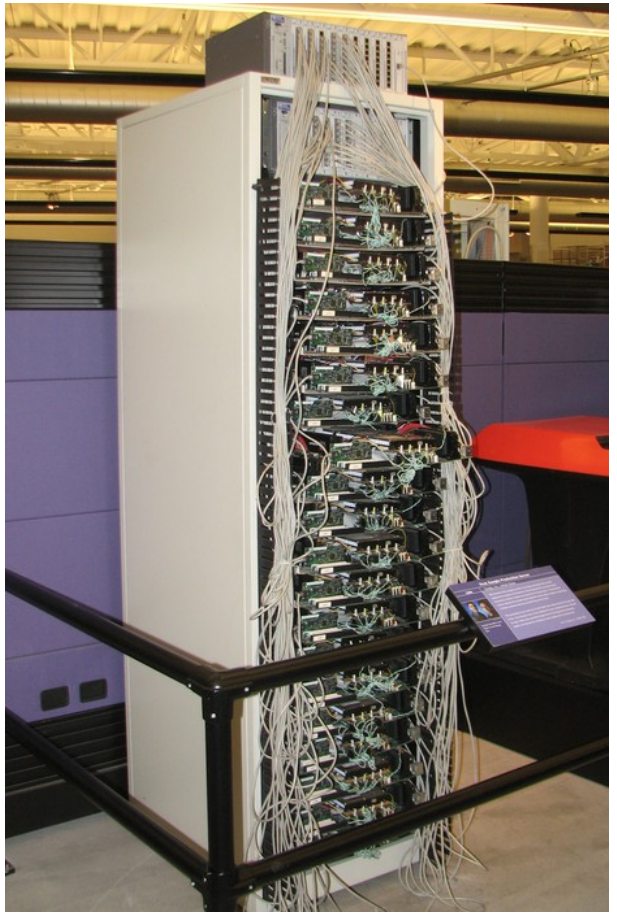


Designing for scalability

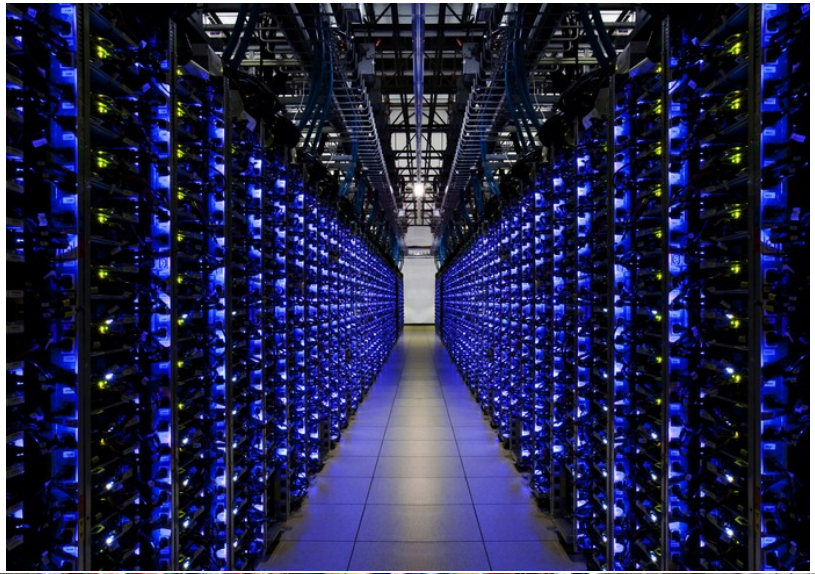


Designing for scalability

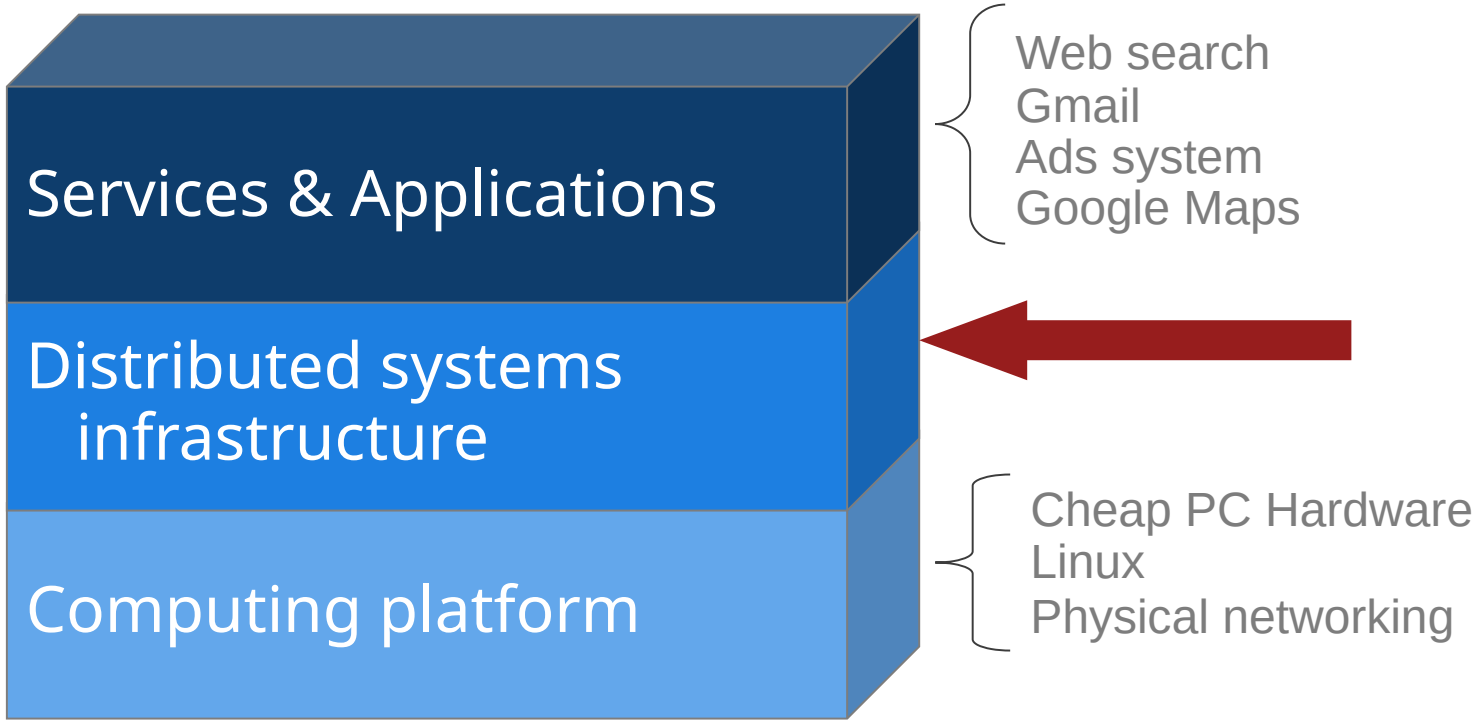




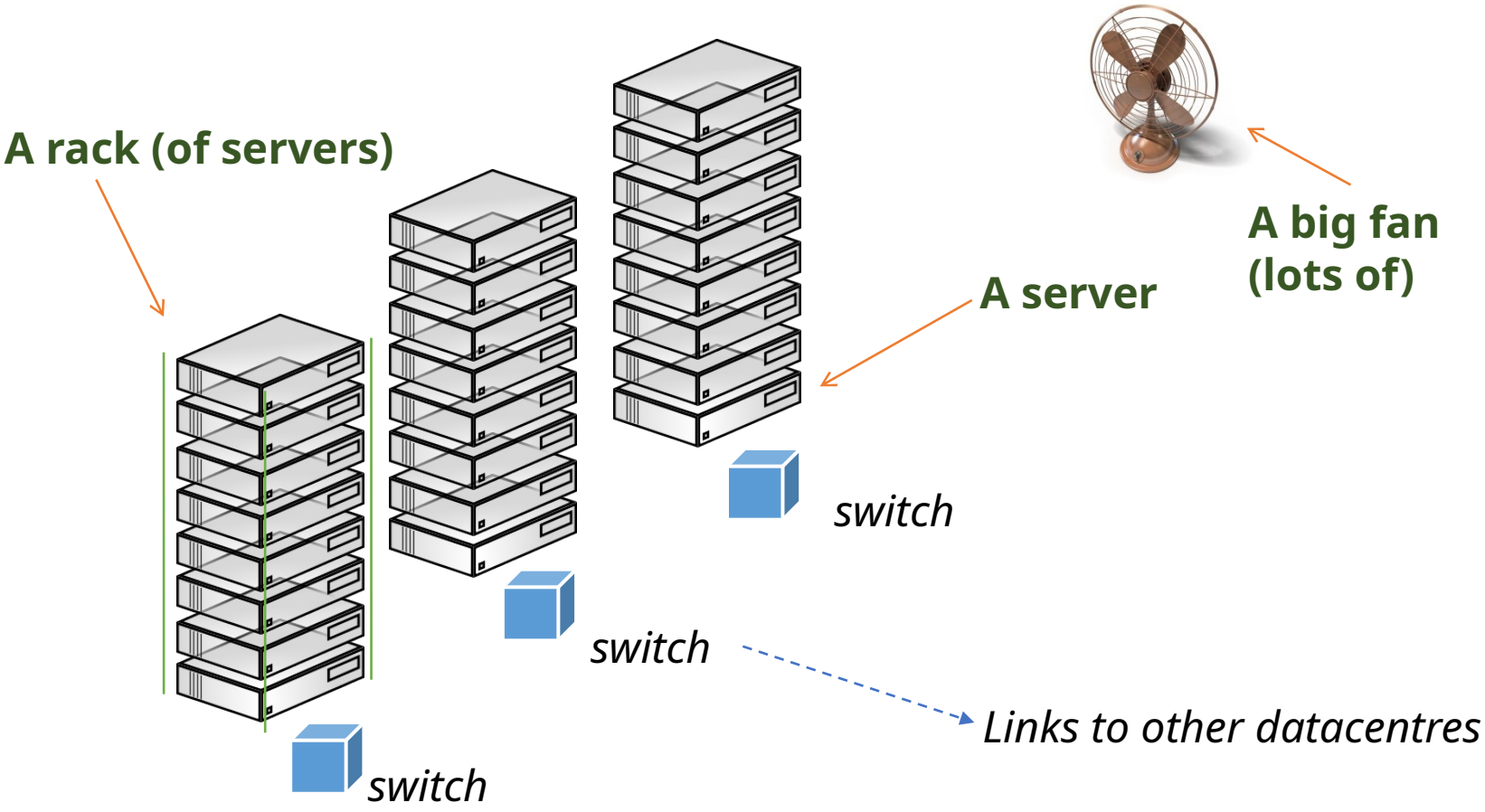
Designing for scalability



Distributed systems at Google



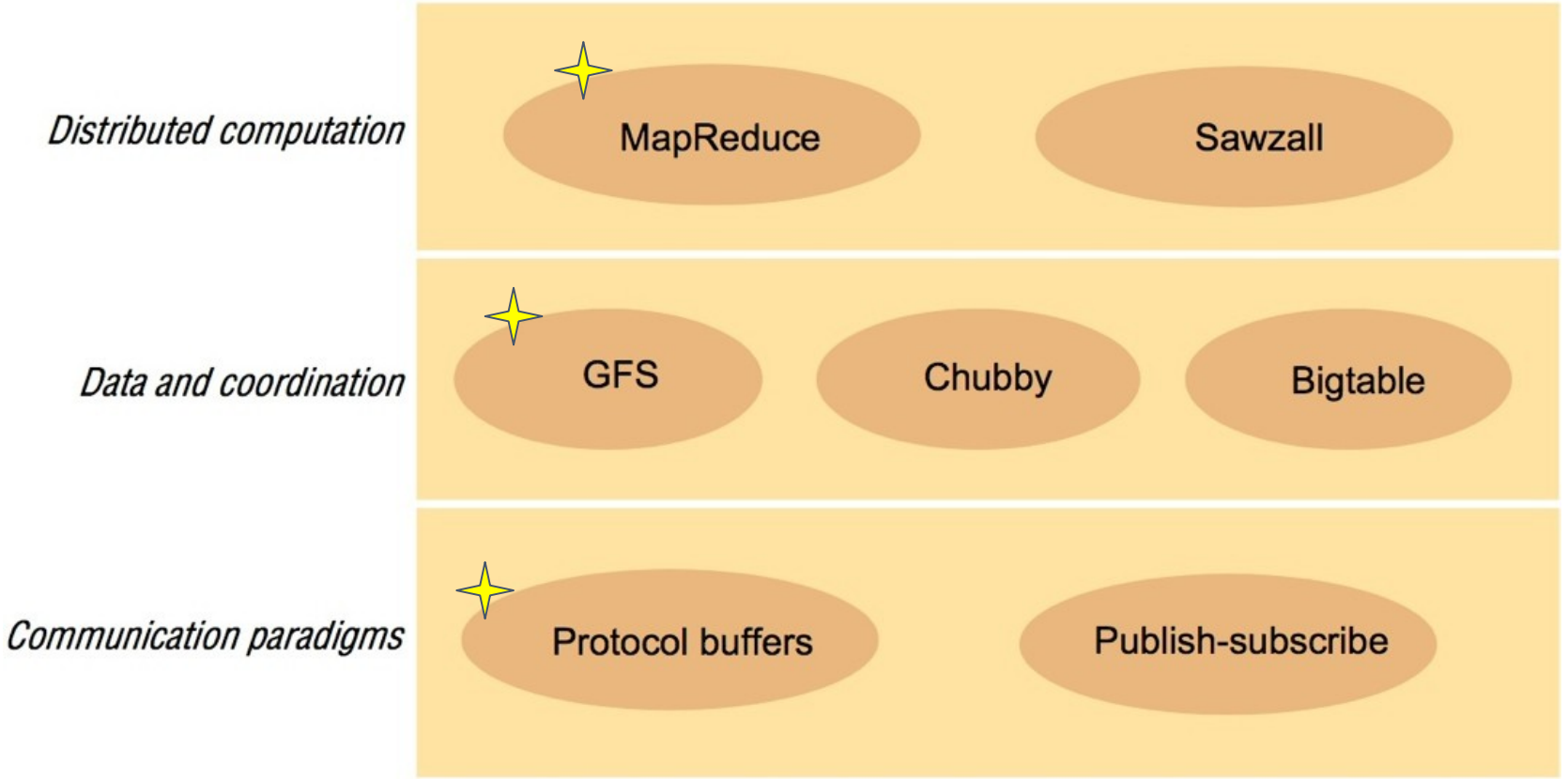
The Hardware Platform



The Hardware Platform

- Each PC is a standard commodity PC running a cut down Linux and featuring around 2 Terabytes of storage
- A given rack consists of 80 PCs, 40 on each side with redundant Ethernet switches (=> 160 Terabytes)
- A given cluster may consist of 30 racks (=> 4.8 Petabytes)
- It is unknown exactly how many clusters Google currently has but say this number is 200 (=> 960 Petabytes – or just short of 1 Exabyte = 10^{18} bytes)
- 2-3% of PCs replaced per annum due to failure
  - Note: hardware failures are dwarfed by software failures

Distributed systems platform



Protocol Buffers

- Google developed the "protocol buffer" as a simple middleware service, which is used for almost all inter-machine communication

- It has two parts:
  - A data serialization format which converts programming language data representations into flattened binary for network transport

  - A simple RPC implementation, where operations support one parameter and one result

Protocol Buffers

- The data serialization format is far more efficient than (for example) XML, and is *not* self-describing like XML/JSON
  - The data in a message is unreadable without a corresponding *schema* which explains the fields, and their types, that are contained in the data
  - Data representation in this format is between 3x and 10x smaller than an equivalent XML representation, and 20-100 faster to transmit

Protocol Buffers

- **Example of setting up a message (defined in a .proto file)**

```
message Person {
     required int32 id = 1;
     required string name = 2;
     optional string email = 3;
}
message Student {
     required Person person = 1;
     required string college = 2;
}
```

- **Example of setting up an RPC**

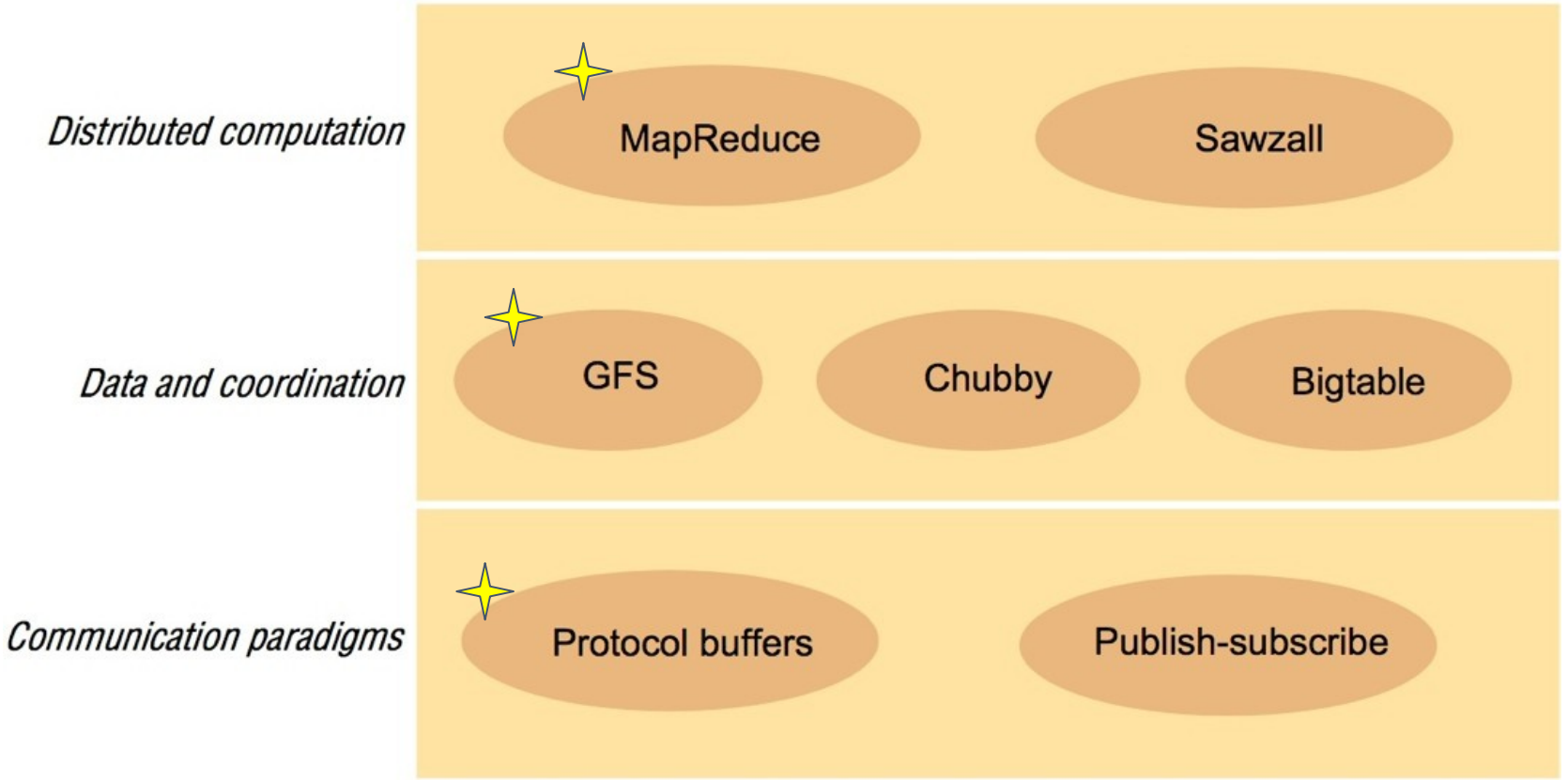
```
message SearchRequest {
     required string query = 1;
     optional int32 page_number = 2;
     optional int32 result_per_page = 3;
}
service SearchService {
     rpc Search (SearchRequest) returns (SearchResponse);
}
```


Protocol Buffers

- A wide range of languages support protocol buffers, making it a good choice for message interchange between different systems
  - Current support includes C++, Java, Python, Go, C, and Swift, among others...
  - In general, RPC is still very much a protocol of choice for communication within datacentres, and still an active research topic (e.g. [1])

[1] Datacenter RPCs can be General and Fast, Kalia et al, USENIX NSDI 2019 (best paper)

Google File System



Google File System

- Prior to the development of GFS, distributed file systems were concerned with replicated meta-data for resilience, and tight consistency implementations to keep each node up to date

- GFS has two main features:
  - a single meta-data copy at a single node, which simplifies consistency management for the directory structure
  - a reduced consistency model for the data within files, which supports far higher scalability at the expense of some potential lost updates

Google File System

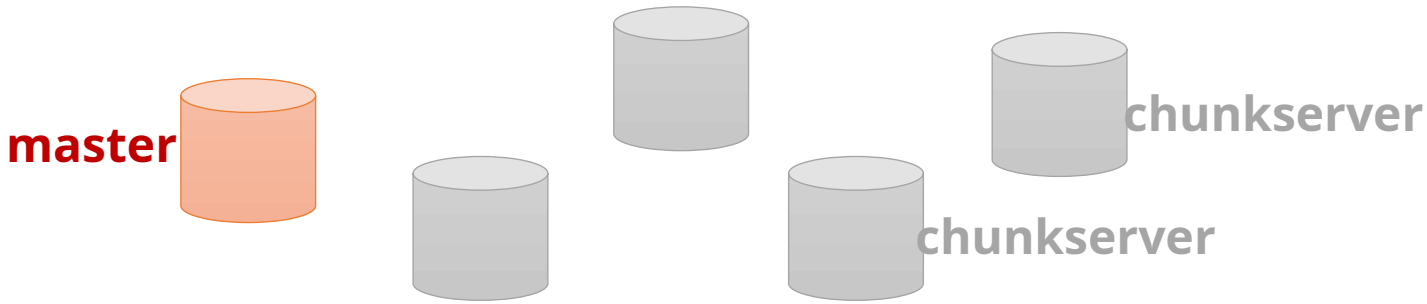
- GFS is one of the most scalable file system designs in the world
  - provides hundreds of terabytes of storage across thousands of disks on > 1,000 machines, concurrently accessible by hundreds of clients
  - works within Google's typical approach of using massive quantities of cheap, commodity hardware (the design quality of which means that frequent failures are expected across disks, memory, power supplies, etc.)
  - optimised for a particular access model: files are almost always modified by appending data; there are almost never writes to the middle of a file; data that has already been written is typically therefore only ever read

Google File System

- Overview
  - A GFS instance is called a “cluster”
  - One cluster comprises one “master” and many “chunkservers”
  - A cluster may be accessed concurrently by multiple clients; these clients may be writing and reading the exact same file at the same time

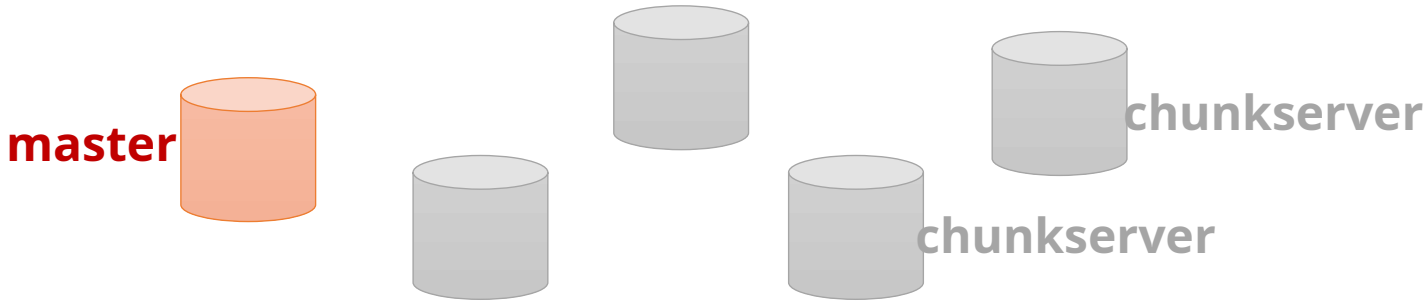
Google File System

- Overview
  - The master node stores all meta-data about the file system instance, including directories and file names; this data is backed up to secondary storage
  - Requests to open a file, create a new file, delete files, etc., all go via the master first – this supports a single atomic lock on core file system operations



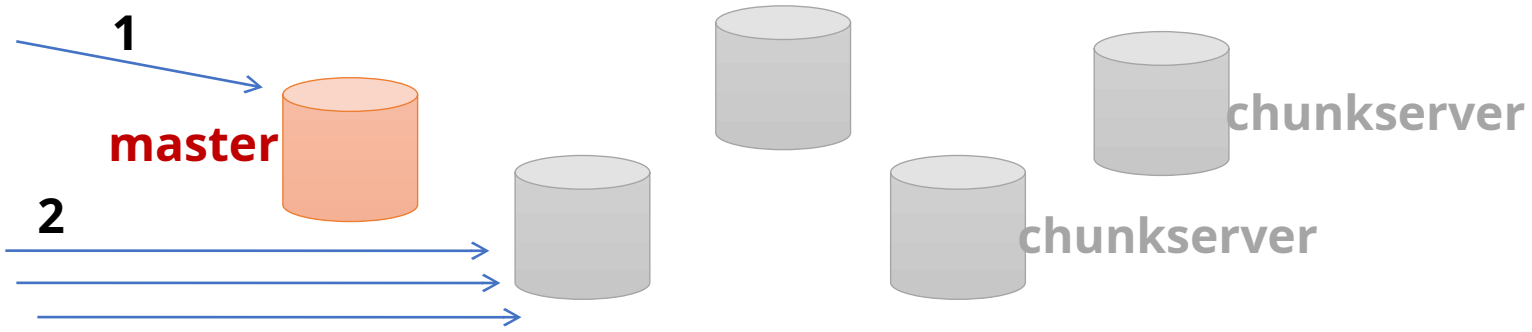
Google File System

- Overview
  - Chunkservers store the actual data of each file; this is stored in fixed-size "chunks" with a typical size by 64MB (cf. local file systems with e.g. 4KB chunks)
  - A chunk is just a regular Linux file on the hard disk; each chunk server periodically contacts the master to exchange information (including liveness and which chunks it stores)
  - Each chunk of each file is replicated *n* times to other chunk servers for resilience against frequent hardware failures



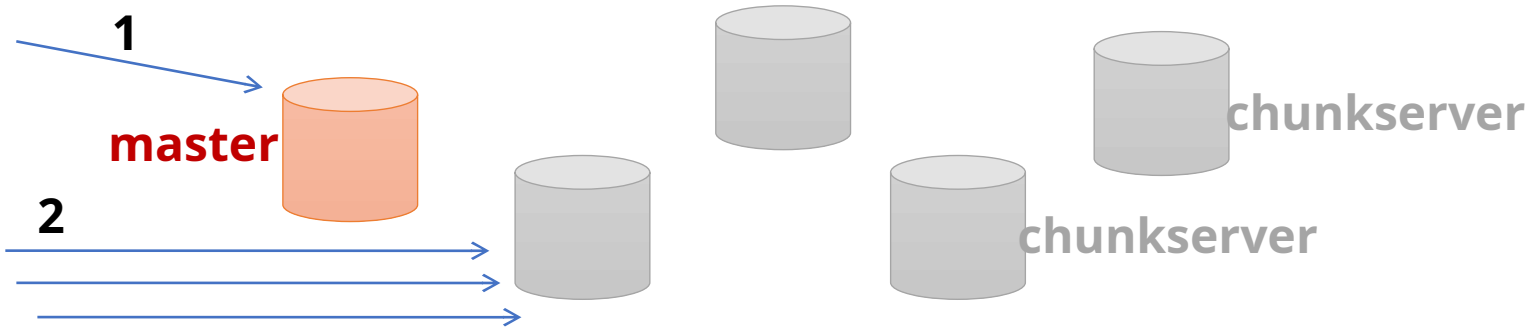
Google File System

- Procedure for a “read” operation from a client:
  - calculate chunk index required for read (chunk sizes are fixed, so clients can do this)
  - ask master to locate chunkserver for {filename, chunk_index}
  - master replies with *all* replicas holding that chunk
  - client selects a replica to talk to (usually the “closest” one)



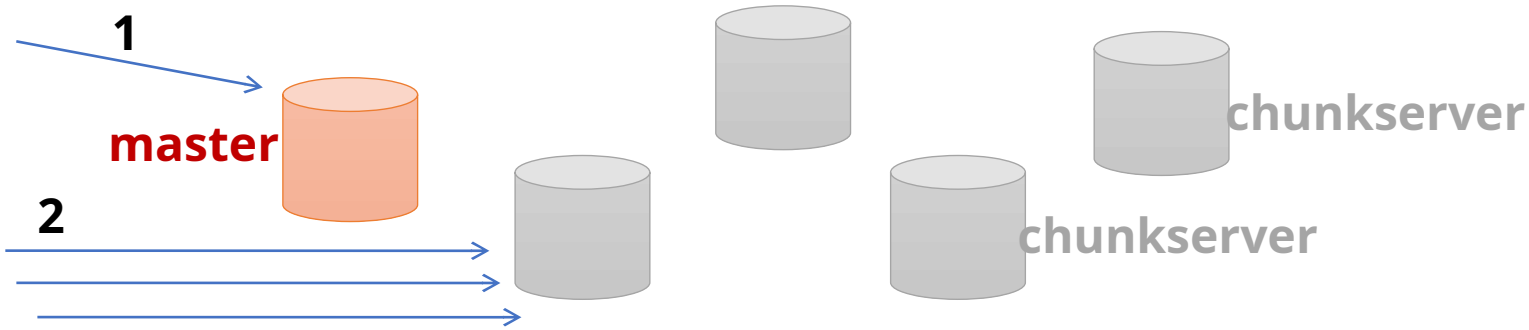
Google File System

- Procedure for a “read” operation from a client:
  - Information held by clients about chunkservers can be cached
  - Using this cached information, clients can therefore make multiple reads/writes to the same chunk without talking to the master again
  - Cached values may however expire, requiring clients to re-contact the master to refresh their information about chunk locations



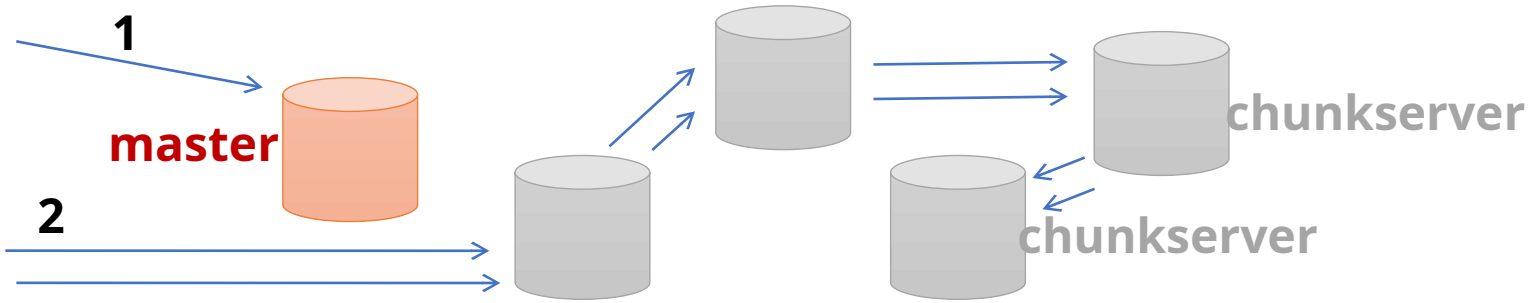
Google File System

- Procedure for a “write” operation from a client:
  - client asks the master for write access to a particular chunk of a file
  - the master checks if a “lease” has already been given out for that chunk, and if so, directs the new client to the chunkserver holding that lease
  - if no lease, the master picks a chunkserver from the replica set for that chunk and creates a lease (this chunkserver is known as the “primary replica” while that lease is valid)
  - the chunkserver holding the lease is responsible for controlling the order of writes and for propagating that order to replicas



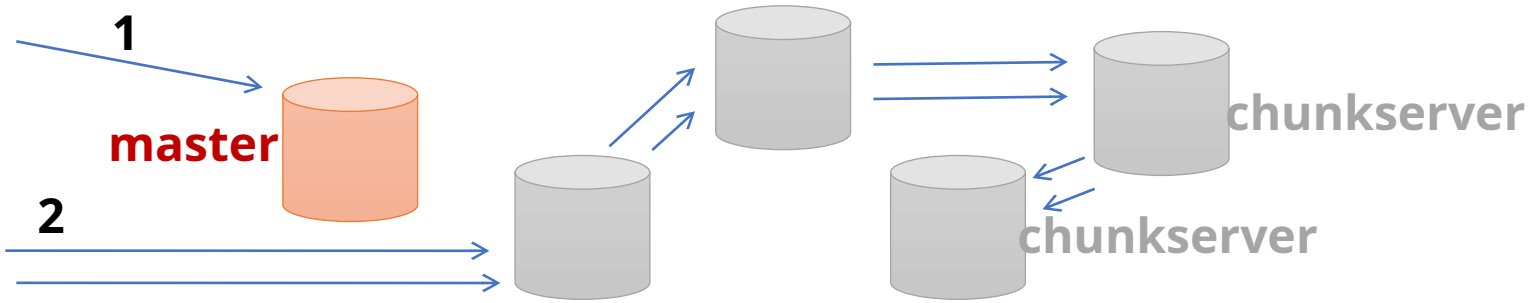
Google File System

- Procedure for a “write” operation from a client:
  - to actually write a region of data, the client sends its data to the primary (which holds the lease) which then sends it in a pipelined chain to all secondaries
  - once this is complete the client sends a write operation to the primary, referring to the data that it sent earlier, and identifying the order in which that data should be written
  - the primary chunkserver determines the actual order of writes (i.e. in conjunction with other clients writing at the same time) and pushes this order to the secondaries
    - once all secondaries reply OK, the write is “successful”



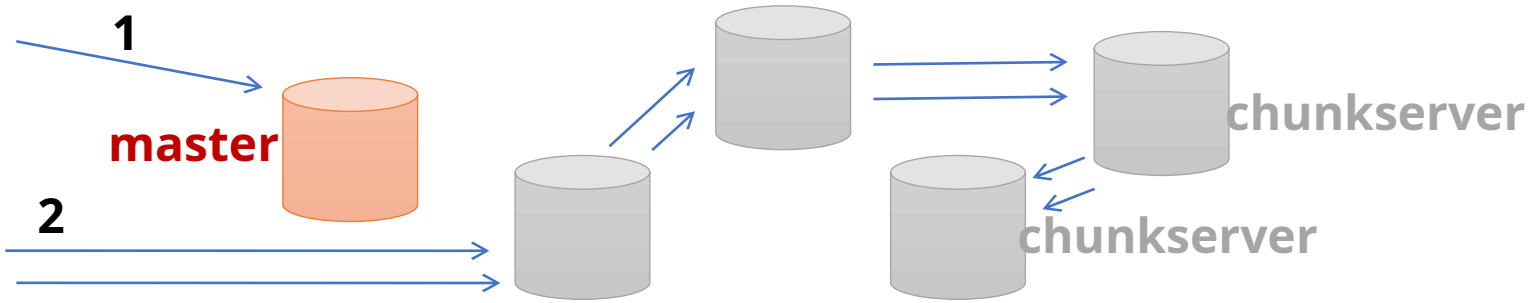
Google File System

- Procedure for a “write” operation from a client:
  - the master can revoke leases at any time (for example if a file is being renamed, copied, deleted)
  - this involves contacting the current lease-holding chunkserver to inform it that its lease has been invalidated
  - this causes a “lease expired” error in any client that tries to use this chunkserver, causing the client to ask the master for a new lease (where the master can make the client wait…)



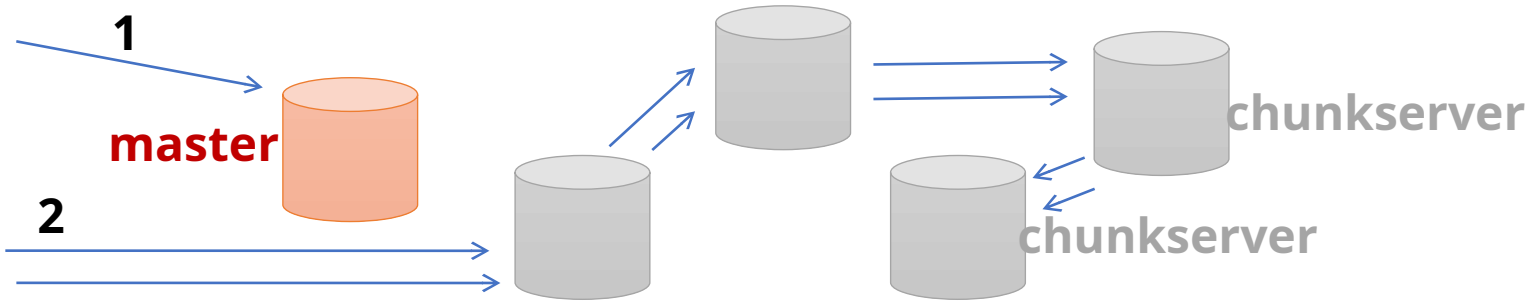
Google File System

- Relaxed consistency model
  - For data writes, we get total ordering orchestrated by the selected lease-holder / primary replica for a given chunk of a file
  - However, we can also be reading from the same chunk at the same time, and can read from any of the replicas (including switching replicas to read different parts of the same chunk)
  - Google's approach here is simply to accept that some clients may get stale / inconsistent data during reads, on the assumption that this will happen relatively rarely



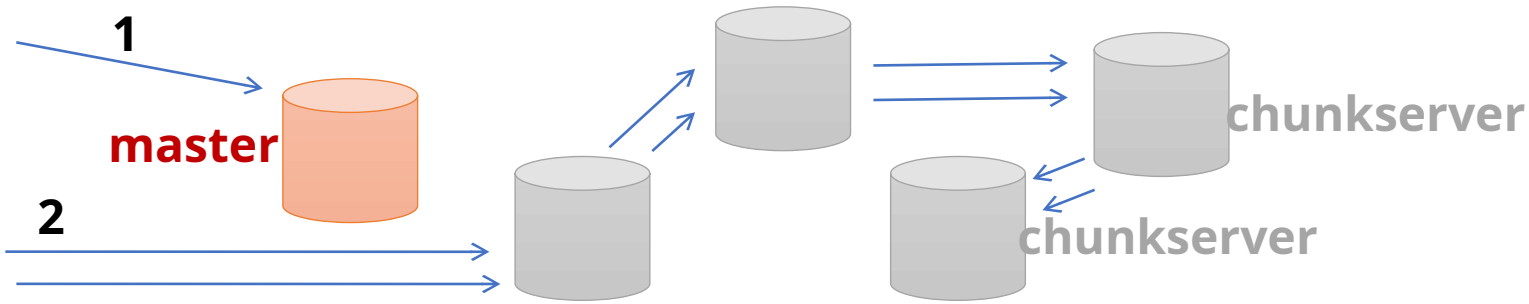
Google File System

- A little more on what's stored at the master...
  - The master's data is held in main memory for speed of lookup, because the master receives high volumes of requests for very small amounts of data
  - The master's entire state is periodically backed up by writing it to the disk; however, doing this frequently is expensive
  - In between periodic full-state backups, the master therefore uses an operation log to store things that have happened since the last full backup (file created, file deleted, ...)

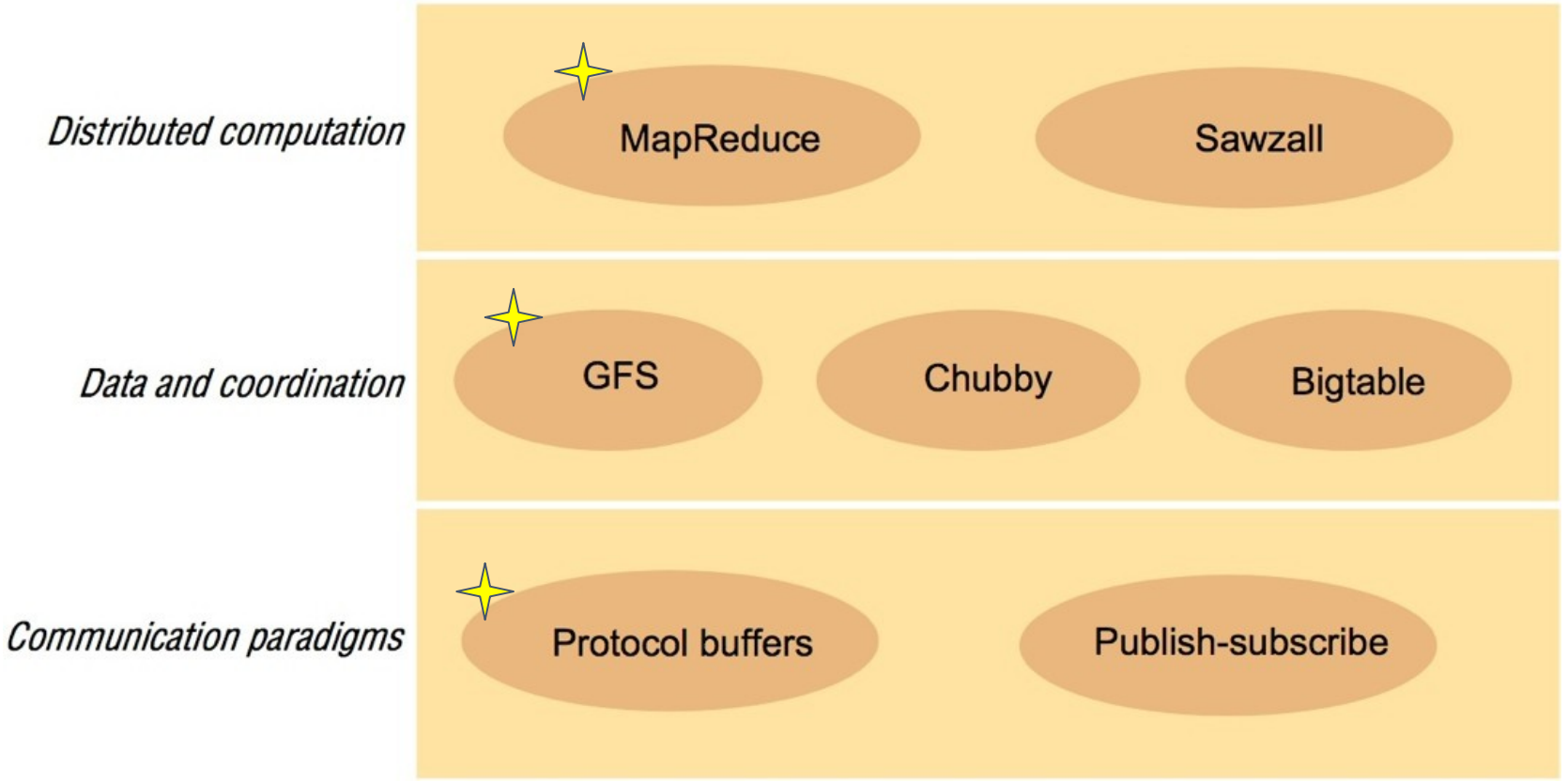


Google File System

- A little more on what's stored at the master...
  - If the master node fails, a new master will take over by loading the most recent full state backup, then replaying anything in the operation log
  - This full-state-backup + operation log design is a common approach for high-performance checkpointing



MapReduce


Distributed computation
MapReduce
Sawzall
Data and coordination
GFS
Chubby
Bigtable
Communication paradigms
Protocol buffers
Publish-subscribe

MapReduce

- A simple distributed computation paradigm to process large amounts of data in a highly parallel way, using thousands of machines
  - MapReduce is a natural counterpart of GFS to help analyse the kinds of data that Google is storing (your Internet journey, search history, ad views, etc.)

- Since its initial design, MapReduce has inspired a range of more complex distributed processing frameworks, including Apache Spark

MapReduce

- Imagine we want to count how many times the word "India" appears in all books in the library
  - We might use a group of people, and assign each person one bookshelf to read; all of the people would then report back with their total

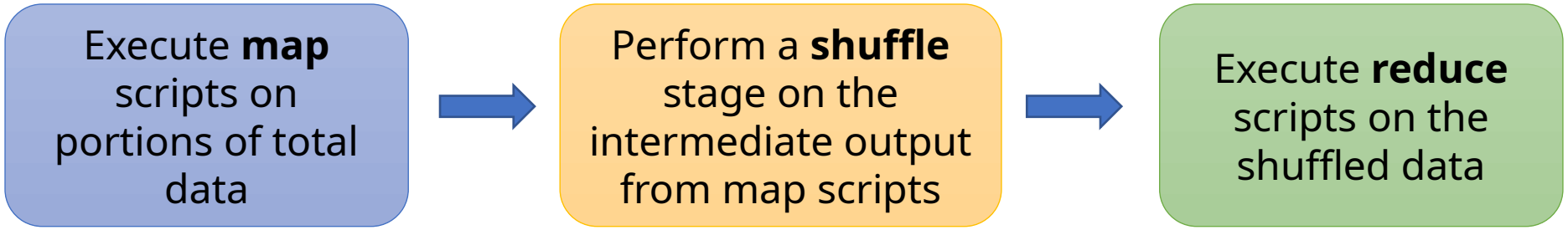
MapReduce

- MapReduce has a simple API with which the programmer can specify: (1) a data source (which will be automatically divided up); (2) a map program, which is just a simple script; (3) a reduce program

- The MapReduce engine then executes the program by moving data to nodes, executing map scripts, then executing reduce scripts

MapReduce

- Every MapReduce process has three main stages:



MapReduce

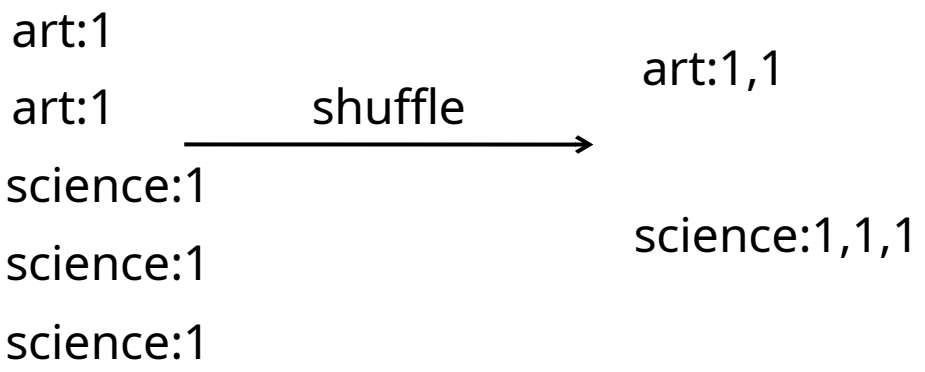
- Map

Takes input data (often a string) and generates a **key : value** pair

MapReduce

- Shuffle (other variations are available)

For each key generated by map processes, this stage **groups all values that have the same key**, placing them at the same worker node

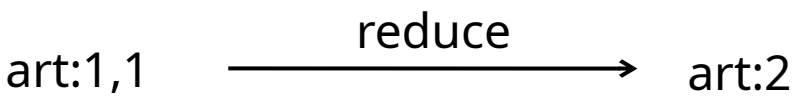


This stage may also **sort all keys** as part of its process

MapReduce

- Reduce

A reducer process looks at one **key : value-list** and performs a computation on the value list. It outputs a **key : value** pair.



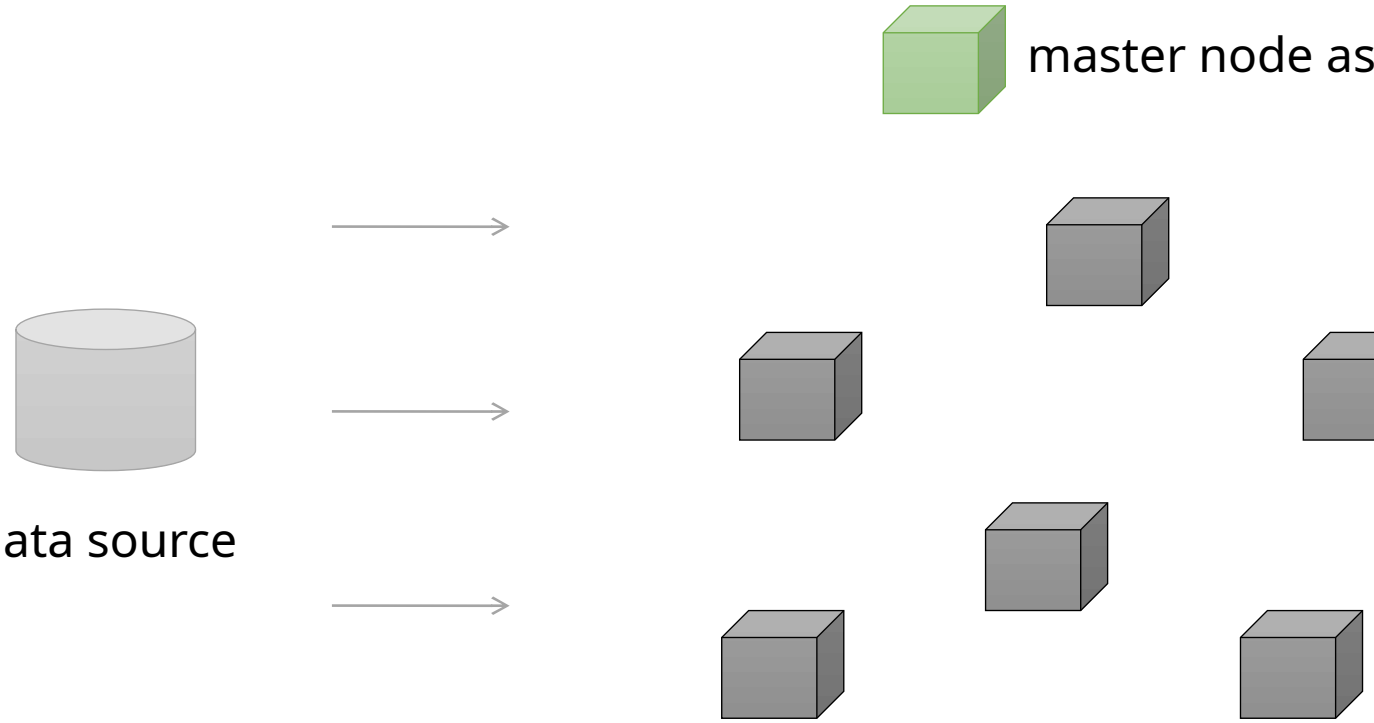
The key : value pairs output by reducers are the result of this processing iteration. In this case we've counted how many words and apples there are (which could take a very long time from a large data set, ideal for parallel processing).

MapReduce

- All MapReduce programs must fit into this basic paradigm

- Although simple, there are a surprising number of different useful programs which happen to work within this framework, including searching, sorting, and basic data aggregation / statistical analysis

MapReduce



master node assigns jobs to workers

data source

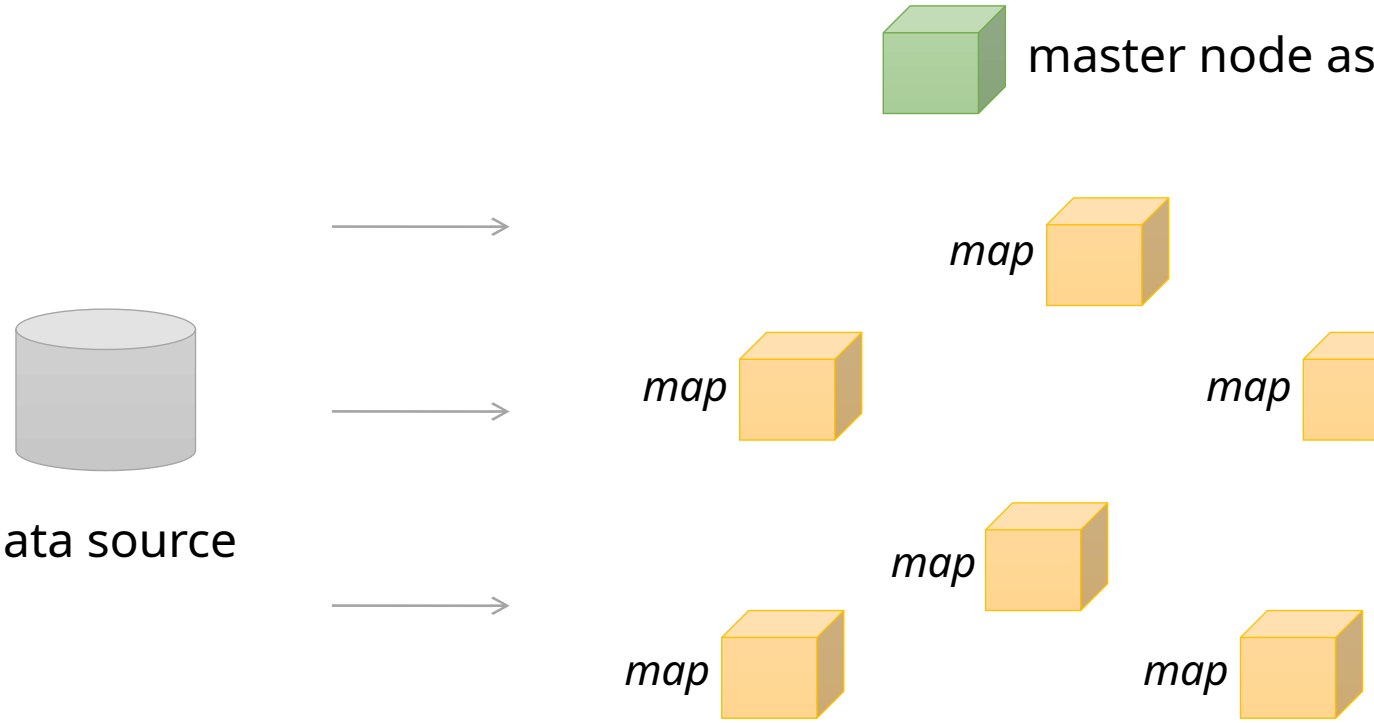
split input data into chunks, one chunk for each map process

worker nodes are assigned to be map or reduce jobs as the program executes

> worker nodes are told whether to be a map or reduce job, where to get their input data from for that job, and where to put their output data

> the size of an input chunk to a map task is typically between 16 and 64MB

MapReduce



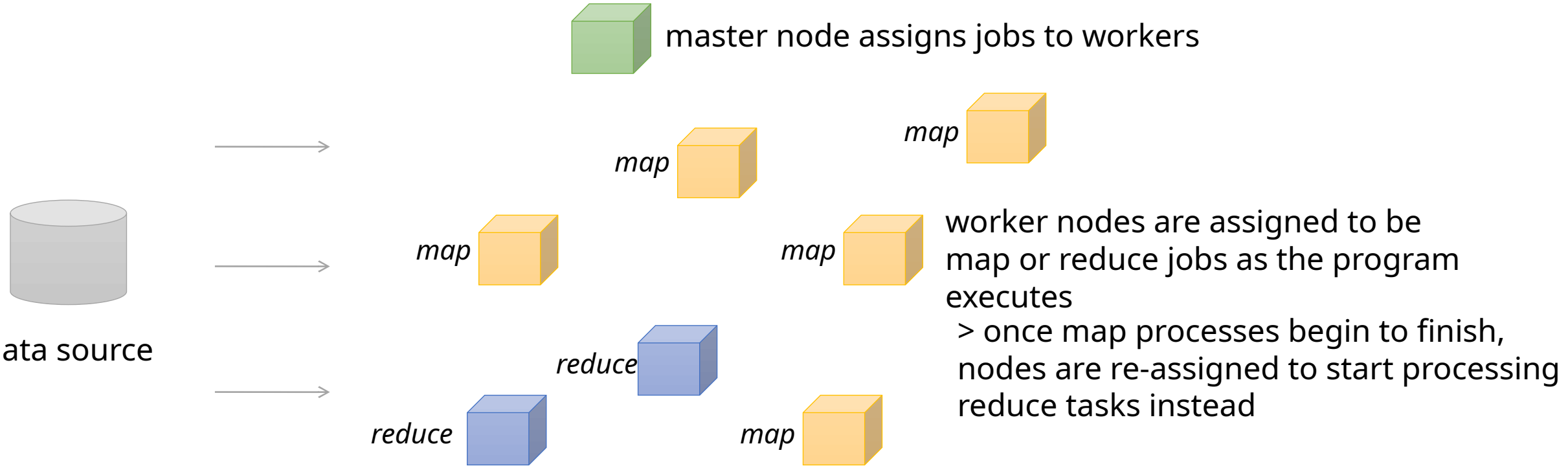
worker nodes are assigned to be map or reduce jobs as the program executes

> worker nodes are told whether to be a map or reduce job, where to get their input data from for that job, and where to put their output data

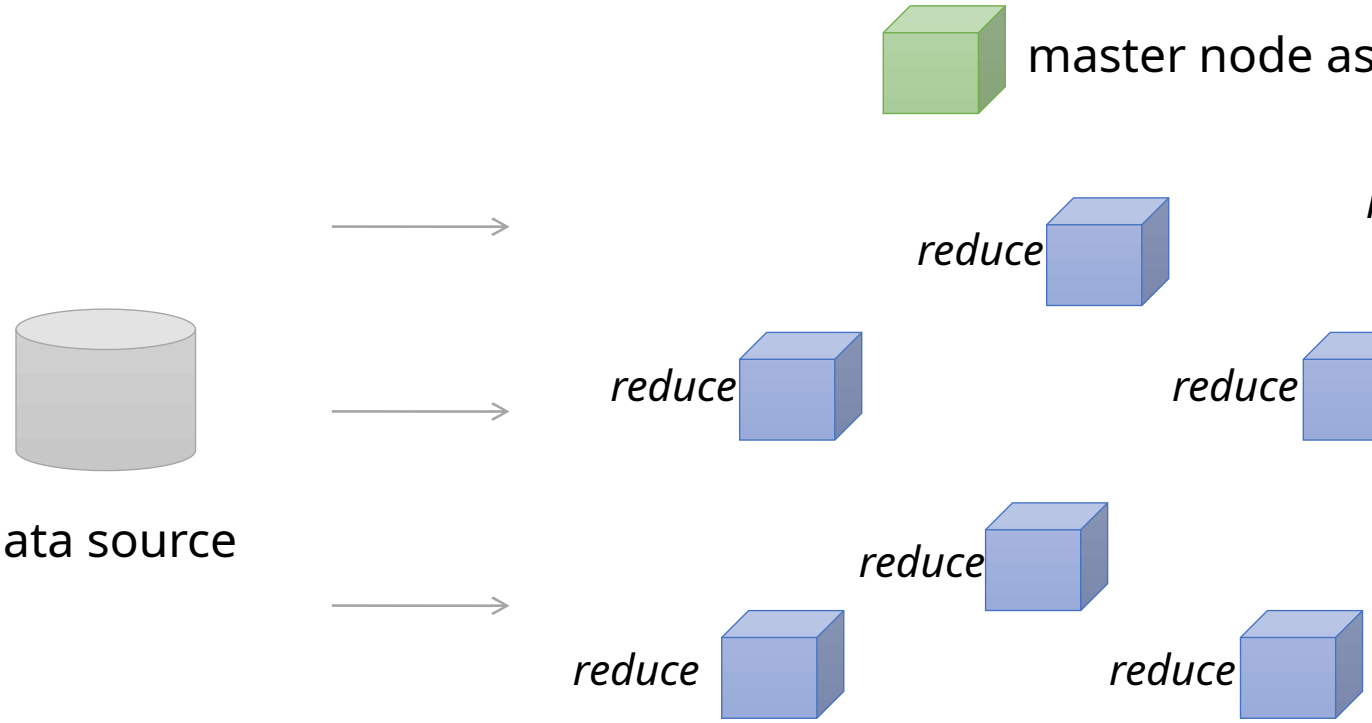
> the size of an input chunk to a map task is typically between 16 and 64MB

split input data into chunks, one chunk for each map process

MapReduce


master node assigns jobs to workers
map
map
map
map
reduce
reduce
map
data source
worker nodes are assigned to be map or reduce jobs as the program executes
> once map processes begin to finish, nodes are re-assigned to start processing reduce tasks instead
split input data into chunks, one chunk for each map process

MapReduce

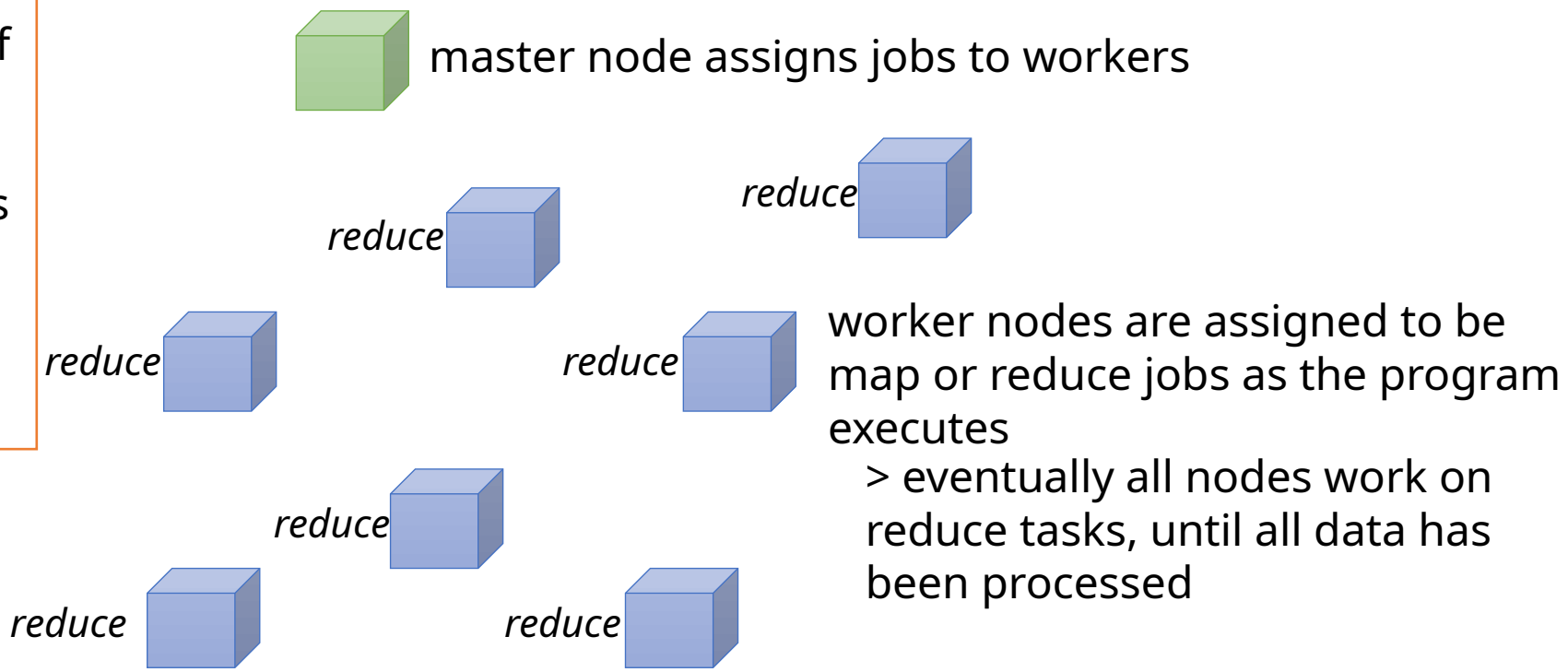


worker nodes are assigned to be map or reduce jobs as the program executes

> eventually all nodes work on reduce tasks, until all data has been processed

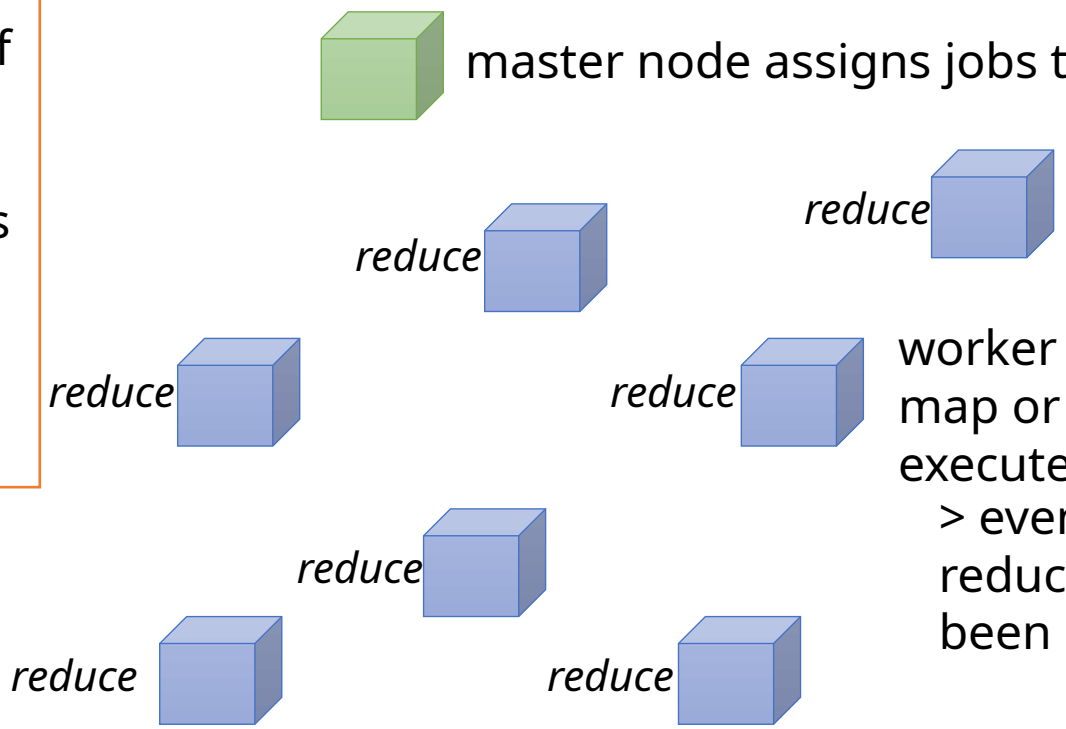
MapReduce

During a map-reduce, the master stores the status of each individual map and reduce process
- this includes if the task is pending, in-progress, or complete, and where any corresponding data is stored



MapReduce

During a map-reduce, the master stores the status of each individual map and reduce process
- this includes if the task is pending, in-progress, or complete, and where any corresponding data is stored



worker nodes are assigned to be map or reduce jobs as the program executes
> eventually all nodes work on reduce tasks, until all data has been processed

The master periodically checks on each worker node to see if it is still alive; if it is unreachable then its map / reduce tasks are re-started at another worker

Summary

- We've covered part of Google's core infrastructure, considering message transport, storage, and basic data processing

- Each of these technologies has since inspired a large number of similar open-source implementations, including Hadoop and Spark

Further reading

- CDKB: Chapter 21
- Lots of material at Google Labs:
  - The Anatomy of a Large-Scale Hypertextual Web Search Engine
  - The Google File System
  - MapReduce: Simplified Data Processing on Large Clusters