Very Big Data Structures

CSE 30331/34331
2015/12/01

Overview

Today's lecture will discuss how to **store** and **process** large amounts of data.

Question: What are some examples of **big data** that would require large amounts of storage and processing?

1. Social/Information Networks (Facebook, Reddit, etc.)
2. Internet of Things (sensors, thermostats, etc.)
3. Transactions (credit cards, payments, ad clicks, etc.)
4. Ratings (Amazon, Netflix, App Store, etc.)
5. Biological Data (gene sequencing, Fitbit, etc.)
6. Multimedia (Instagram, YouTube, etc.)
7. Textual Data (Project Gutenberg, Google Books, etc.)

Because of the large amount of data and the need to process and analyze the data quickly and effectively, we must harness the computational resources of multiple machines (i.e., **distributed systems**).

In particular, we will briefly discuss how the **Google File System** stores large amounts of data, and how **Map-Reduce** is used to process and analyze the vast collection of information.

Google File System

Google File System (GFS) is a **scalable**[1], **distributed**[2] **file system**[3] on **inexpensive commodity hardware**[4] that provides:

1. **Fault Tolerance**
   File system runs on hundreds or thousands of storage machines with inexpensive commodity parts. Example is 1000 storage nodes with over 300 TB.

2. **High Aggregate Performance**
   Fully utilize bandwidth to transfer data to many clients, achieving **high system throughput**.

Terms:

1. **scalable:** system grows with amount of data without significant performance loss (remains efficient)

2. **distributed:** system is spread across multiple machines, possibly multiple clusters or data centers that communicate via a network

3. **file system:** system for organizing and managing persistent data (usually but not necessarily hierarchical)

4. **inexpensive commodity hardware:** non-exotic, run-of-the-mill, standard hardware that any consumer can buy

[Google Data Center]

Design

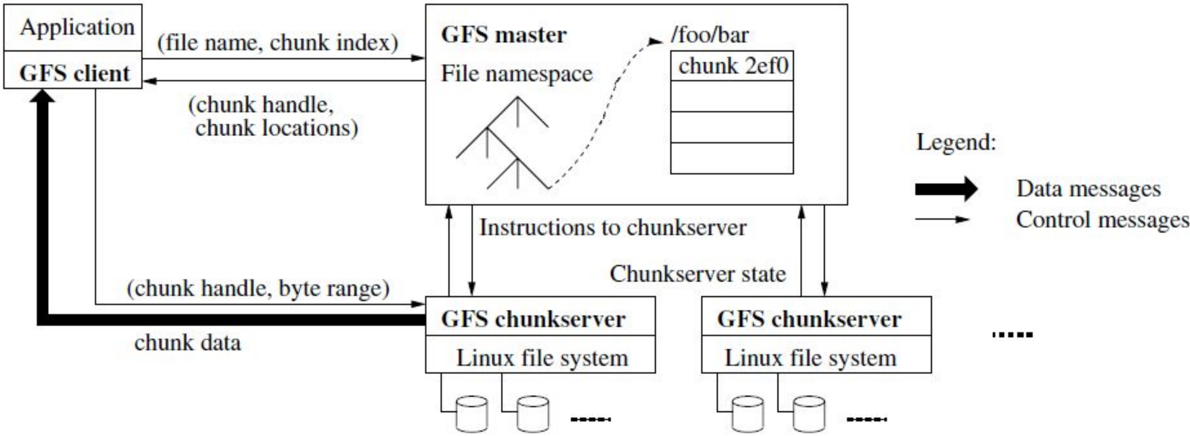
The design of GFS revolves around the following observations and assumptions:

Observations	Conventional	Google
Reliability	Systems assume a working environment and handle failures as worst case scenarios.	Component failures are the norm rather than the exception, therefore constant monitoring, error detection, fault tolerance, and automatic recovery must be integral to system.

File Sizes	File systems are composed of many small files and a few large ones and thus block sizes are minimized to support a wide variety of file sizes.	File system must store a modest number of huge files (multi-GB) organized into data sets in the range of TBs with billions of objects. Due to the focus on processing large amounts of data in bulk, high sustained bandwidth is more important than low latency.
I/O Pattern	Normally, files are updated in place, synchronization requires locking, and caching is important for performance.	Data is appended rather than overwritten. Random writes are rare. Once written, files are only read (usually sequentially). Caching is not important because most applications stream through huge files or have extremely large working sets.

These observations and assumptions are uncharacteristic for conventional systems and environments and are particular to Google's specific distributed and large scale applications and workloads.

Architecture



1. **Chunks:** files split into fixed-sized chunks which is given a globally unique chunk handle.
    a. Chunks replicated on multiple **chunkservers** (default is **3**) for reliability.
    b. Chunk size is **64MB** which is much larger than normal file system blocks.

i. **Advantages**:
    1. Reduces interaction w/ master.
    2. Reduces metadata stored on master.

ii. **Disadvantages**:
    1. Small files may become hotspots.

2. **Master:** Single node maintains all of the metadata such as namespace, ACLs, mapping from files to chunks, and current location of chunks. Also sets policies regarding chunk management (garbage collection, migration, etc).
    a. **Metadata** kept in memory:
        i. File and chunk namespaces.
        ii. Mapping from files to chunks.
        iii. Locations of chunk's replicas.

    b. **Operation log** is used to persistently store metadata operations and record order of concurrent operations.
        i. Recovers filesystem by replaying this log.
        ii. Checkpoints used to minimize startup time.
        iii. Replicated to local disk and remote machines.

    Many data storage systems (e.g databases, filesystems) perform some sort of transaction logging or journaling whereby intended actions are recorded, performed, and then committed.

    Example:

```
TIMESTAMP OPERATION CHUNK PROPERTY VALUE
100000000 CREATE    2
100000001 UPDATE     2     NAME     DATA.TXT
100000002 CREATE     3
100000003 UPDATE     3     NAME     DATA.CSV
100000004 REMOVE     2
```

    Question: How would you implement this? (**Hash Table**)

```
100000000 Chunks = {2: {}}
100000001 Chunks = {2: {NAME: DATA.TXT}}
100000002 Chunks = {2: {NAME: DATA.TXT}, 3: {}}
100000003 Chunks = {2: {NAME: DATA.TXT}, 3: {NAME: DATA.CSV}}
100000004 Chunks = {3: {NAME: DATA.CSV}}
```

    c. Periodic scans enable garbage collection, re-replication and chunk migration.

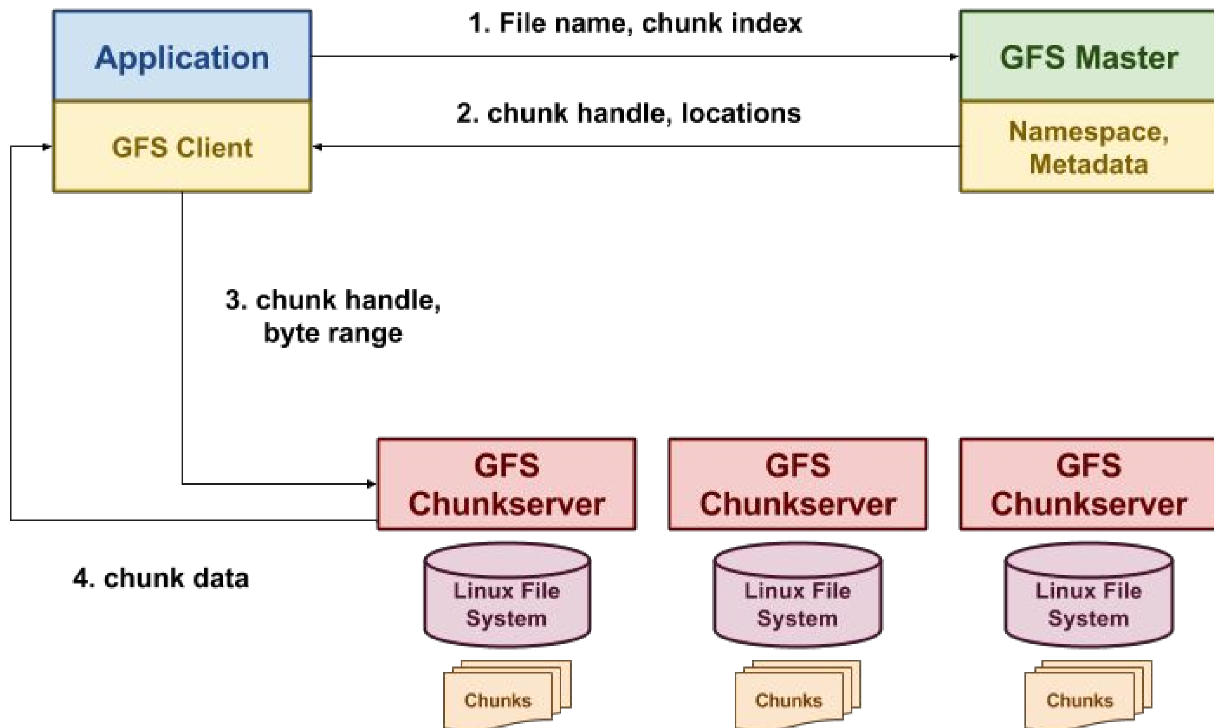
d. Single master ensures that file namespace mutations are atomic.

e. Shadow masters provide read-only access to file system when master is down.

3. **Chunkservers:** Multiple storage nodes that store **chunks** on local disks as Linux files and read/write data specified by **chunk handle**.
    a. Stores chunk location information and sends to master on startup.

    b. Clients do not cache data, but do cache metadata. Chunkservers do not manually cache data because Linux's buffer cache will do it.

Reading:



MapReduce

MapReduce is a programming model for **processing** and generating large datasets:

- **Abstraction:** Hides complexity of parallelization, fault-tolerance, data distribution, load balancing behind a library and run-time layer.

- **Scalability:** Runs on large clusters of commodity machines.

Programming Model

Conceptually, the programming model behind MapReduce is rooted in the functional programming paradigm:

1. **Map:** transforms, filters, or selects input data

2. **Reduce:** aggregates, combines, collects intermediate results

Rather than performing explicit iteration, we apply functions to elements in patterns defined by higher-order functions

Example (C++):

```
vector<int> data {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

// Determine the sum of all the even squares of elements in data
int total = 0;
for (auto number: data) {
      int square = number*number;
      if (square % 2 == 0)
            total += square;
}
cout << total << endl;

// Now functionally
auto dbegin = data.begin();
auto dend   = data.end();

transform(dbegin, dend, dbegin, [](const int x){ return x*x; });
dend   = remove_if(dbegin, dend, [](const int x){ return x % 2 != 0; });
total  = accumulate(dbegin, dend, 0, plus<int>());
cout << total << endl;
```

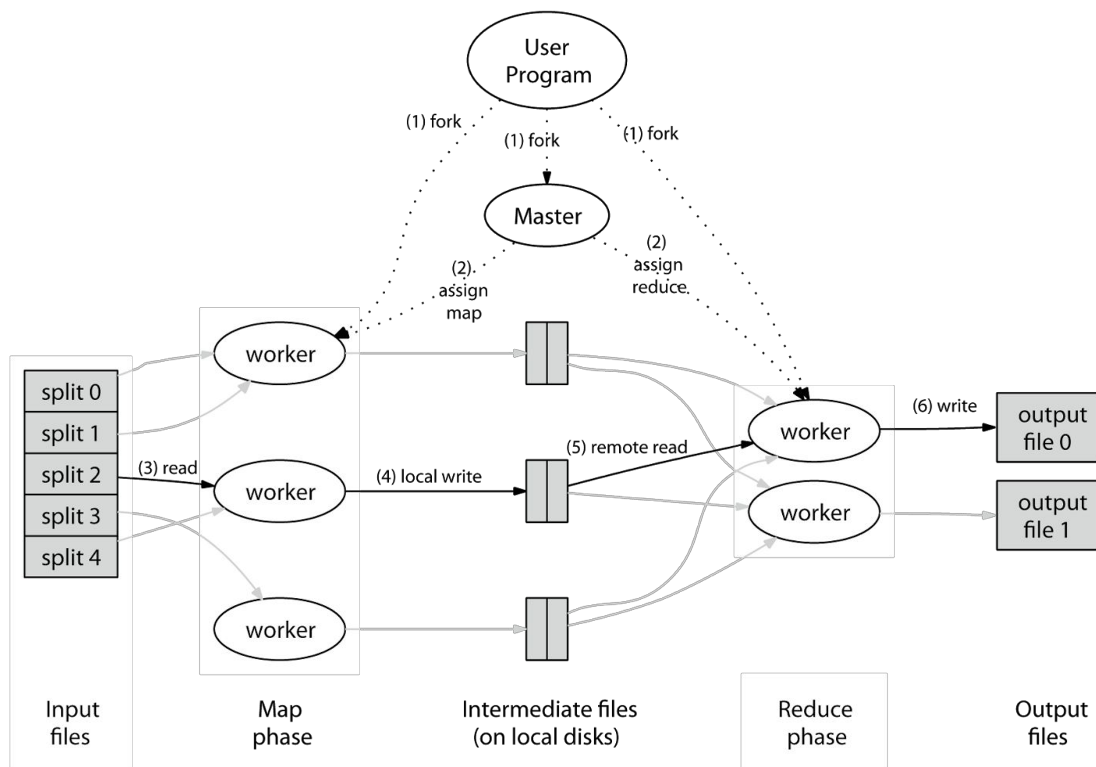
Example (Python):

```
# Multiple lines
data    = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
squares = map(lambda x: x*x, data)
evens   = filter(lambda x: not x % 2, squares)
total   = sum(evens)

# One line:
sum(filter(lambda x: not x % 2, map(lambda x: x*x, data))
```

By using higher-order functions such as **Map** and **Reduce**, we can take advantage of **data independence** and perform the computation in any order (**including in parallel**).

Execution Model



In MapReduce, instead of processing lists, we process collections of **key, value pairs**:

Input <k, v> Pairs => Mappers => Intermediate <k, v> Pairs => Reducers => Output <k, v> Pairs

1. **Map Phase:** Each ***Mapper*** reads in a *split*, which is an arbitrary subset of the input (key, value) pairs, and applies the Map function to each pair to produce intermediate (key, value) pairs.

2. **Partition and Shuffle Phase:** All intermediate (key, value) pairs are partitioned such that all pairs that have the same key are sent ("shuffled") to the same ***Reducer:***

   ```
   Partition(key) = Hash(key) mod R
   ```

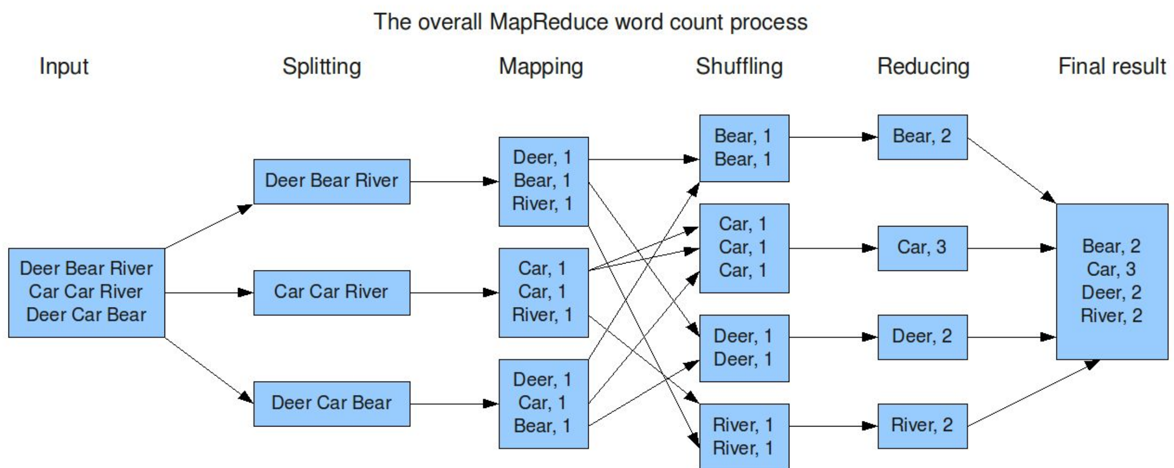
   where **R** is the number of reducers.

3. **Reduce Phase:** Each *Reducer* collects its corresponding intermediate (key, value) pairs, sorts them by key, and then applies the Reduce function on each subset that has the same key, i.e., {(key, value1), (key, value2), ...}, to yield output (key, value) pairs.

4. **Merge Phase:** Optionally, all the outputs of the ***Reducers*** can be merged into a single file.

[Hadoop Tutorial]

Example



The canonical MapReduce example is to produce a WordCount on a large corpus of data:

1. **Map:** For each word in each line, emit the word and the count 1

2. **Reduce:** For each word, tally up the counts and emit the word and the count

Another use of MapReduce is to produce an Inverted Index (ie. what you might find at the back of a book so you can find the page where a word appears):

1. **Map:** For each word in each line, emit the word and the name of the file

2. **Reduce:** For each word, collect all files associated with word and then emit them

C++ Example Code

Discussion

- What is the significance of using **(key, value) pairs** rather than just lists as the data format?

- Why do we sort at the beginning of the **Reduce** phase?

- How is work divided among the **Mappers**? How is work divided among the **Reducers**? Is the division of labor balanced? Why or why not?

- In a distributed system, network transfers are often a bottleneck. Looking at the word count example above, is there an optimization that can be performed after **Mapping** but before **Reducing** to minimize network traffic?

- Explain how you would merge the results of the **Reducing** phase into one file.