Low Bandwidth Google File System

Aadi Swadipto Mondal Mohil Patel Rahul Uday Chakwate
Project Group 3-15

1 LB-GFS DESIGN

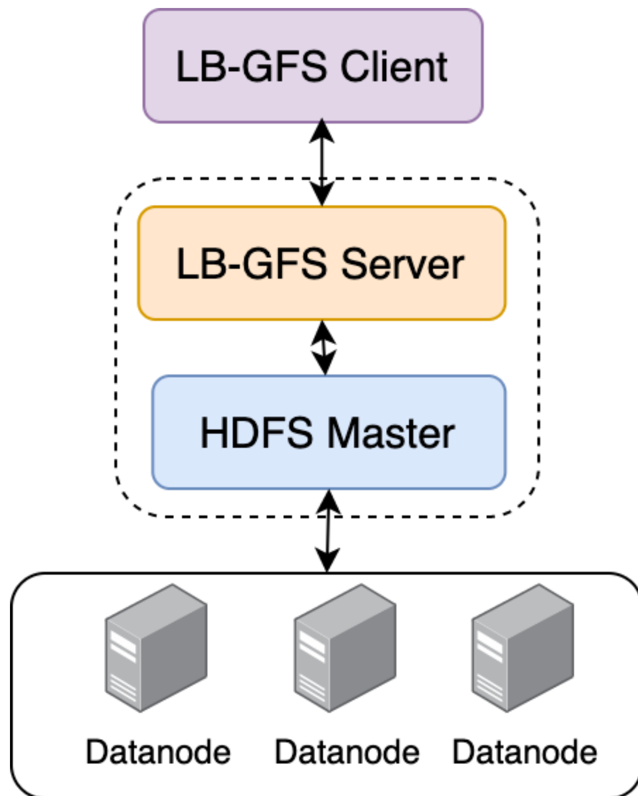


Figure 1: Overall Design

1.0.1 Key Design Principles. The overall design is summarized in figure 1

(1) We developed a wrapper over HDFS to optimize the system over low-bandwidth networks.
(2) We partition files into chunks (similar to HDFS or GFS). The hashes (specifically SHA-256) of these chunks are then used to identify similar chunks present both on the client and server and need not be transmitted over the network.
(3) Chunks are immutable in LB-GFS and are created as and when necessary. Un-referenced/unused chunks are garbage collected at regular intervals.
(4) The chunk size in LB-GFS and the block size in HDFS are exactly the same. Chunks are stored in HDFS as separate files. Each file in HDFS consists of only one HDFS block. This is to ensure that we can correlate HDFS blocks directly to LB-GFS chunks.

1.0.2 Performance issues because of wrapper methodology. Integrating ideas of LB-GFS into HDFS is always more efficient. Due to the restriction of time and resources, we found it difficult to change the HDFS code base and rather came up with a wrapper on HDFS to showcase our ideas. Although, this successfully portrays our ideas, using a wrapper introduces some crucial toll on performance.

(1) There is a duplication in meta-data because HDFS keeps meta-data for tracking files to HDFS blocks. Very similarly, the LB-GFS wrapper also keeps track of files to LB-GFS chunks. Since the HDFS master (i.e. namenode) is bottle-necked by the memory of the master machine, duplication of file meta-data can lead to faster depletion of precious memory resources, thus limiting the system.
(2) Data in HDFS can directly be moved to the chunk servers. Since the wrapper writes the data in HDFS, we have to first move the data to the wrapper (which sits on the same machine as that of the HDFS master, i.e. name node) and then to the chunk-servers. This can be partially averted by running one chunk server and HDFS master on the same machine so that the data can be replicated to at least one chunk server without data transmission over the network.
(3) Integrating ideas of LB-GFS within HDFS will save additional GRPC and other API calls, making the system more efficient.

1.0.3 Application Programming Interface (API). LB-GFS server communicates with the HDFS master and creates separate files for each chunk. It also tracks files to chunk mappings. The client keeps a separate

record of encountered chunks in its own cache and uses chunk hashes to identify common chunks present in both the server and the client.

(1) **CreateFile(filepath)**: Creates a new file in the system. Adds necessary meta-data to track LB-GFS files and corresponding chunks.
(2) **DeleteFile(filepath)**: Deletes file to chunk mappings if present.
(3) **ReadFile(filepath, offset, length)**: Reads specific portion of a file, returns data as a string.
(4) **TentativeAppend(filepath, data)**: Collect data locally for appends until the appends are committed.
(5) **CommitAppend(filepath)**: Push all tentative appends to the server.

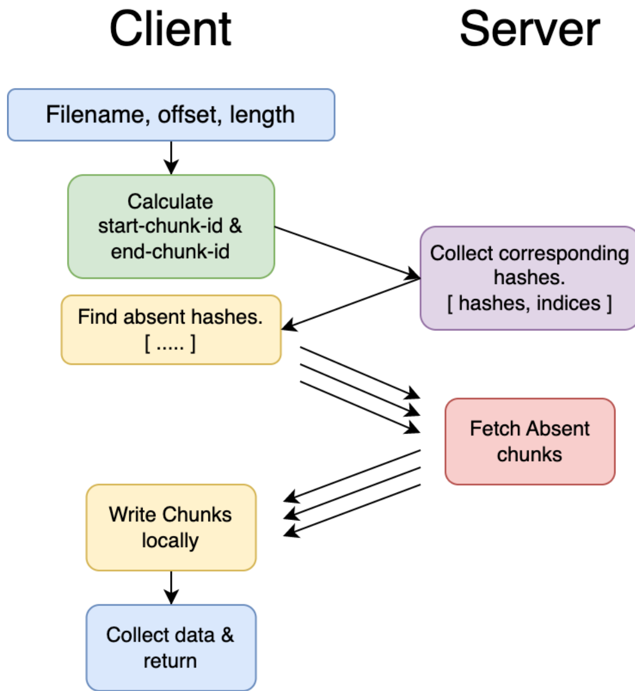


Figure 2: Read Protocol

1.0.4 Read Protocol. The read protocol is described in figure 2.

(1) The client receives a filename along with the offset to read from and the number of bytes to read.
(2) Using this information, we derive the start and the end chunks (chunk size is constant).
(3) Client requests the hashes from the server for the corresponding filename and chunks.
(4) For the hashes that are present in the client, the client reads from the chunks that it is tracking.
(5) For chunks that are absent, the client fetches those chunks from the server and records them in its own cache.
(6) Finally client coagulates all the data from the chunks to the exact offset and length and returns it to the client.

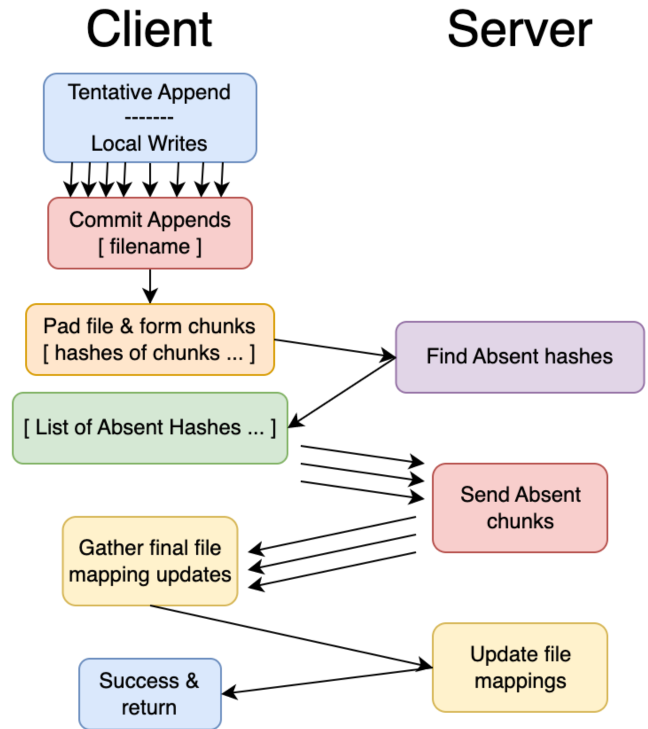


Figure 3: Append Protocol

1.0.5 Append Protocol. The append protocol is described in figure 3.

(1) Tentative appends received from the client are locally written in a temporary file (created when it is the first time).
(2) When the client calls to commit, the temporary file is padded to the multiple of the chunk size, and chunks are created.
(3) The client sends the chunk hashes to the servers and receives a list of absent chunks.
(4) Client then, sends the absent chunks to the server.

(5) Once the client ensures that the server has all the chunks, the client sends a request to the server to update file mappings.

1.0.6 Reconstructing file on server. After the commit is called from the client and the client requests to update the file mappings, a series of steps happen in the server to append the new data sent from the client.

(1) Server receives a list of chunk hashes that correspond to the newly appended data chunks.
(2) If the last chunk is incomplete, the server reads the last chunk and writes to a temporary file.
(3) Server reads the new data and appends it to the temporary file.
(4) The temporary file is again used to form new chunks and is updated in the chunk-reference map.
(5) Finally file mapping and the file sizes are updated with the new chunk hashes.

2 EXPERIMENTS

2.1 Experimental Setup

We measured the performance of our system on a 4 node cluster with 100 Mbps internode network speed.

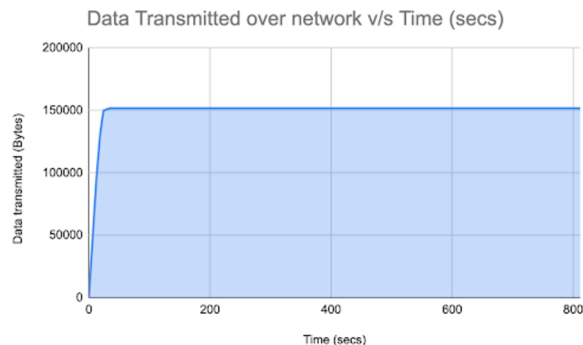
Our workload parameters are as follows.

- Block size - HDFS chunk size
- # distinct keys - Total number of distinct events strings that we append in the file
- key size - size of each individual event string

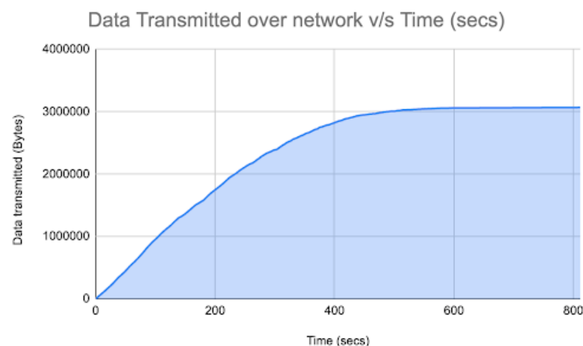
2.2 Data Transmitted over time experiment

In this experiment, we find the data transmitted over the network from client to the server over time. Figure 4a shows this plot for a block size of 1024 bytes, key size of 256 bytes and 4 distinct keys. We observe that as the experiment begins, data is sent from client to the server at a linear pace. Once the server has cached sufficient blocks, the data transmitted over the network reaches saturation. This means that the server has all the required data to recreate blocks and append them to the file. No new data is being transmitted to the server.

Figure 4b shows the same plot for a block size of 1536 bytes. We observe that as the block size increases, the total number of possible distinct blocks sent from client to server increases exponentially. Hence, it takes longer from the server to receive all possible distinct data and hence it reaches saturation at a later point in time, compared to the previous run.



(a) Block size of 1024 bytes



(b) Block size of 1536 bytes

Figure 4: Data transmitted over the network against time

2.3 Variation with block size

In the following set of experiments, we measure throughput variation and % of total data transmitted over the network against different tunable parameters of LB-GFS. We compare LB-GFS against **(a)** passthrough - a setup in which the appends from client to server are directly written to the HDFS file, and **(b)** passthrough with batching - appends are batched locally and written on server once the commit call is made.

Figure 5 shows throughput variation against block size. We increase the block size from 1024 bytes (1 KB) to 131072 bytes (128 KB) and observe that throughput is very high for the 1024 bytes block size. This is intuitive because only a limited distinct blocks can be formed

for a smaller block size and once server receives that data, throughput increases by a large margin. As block size increases, we see diminishing improvements in our system. In Figure 6 we observe the trend in percentage of total data transmitted over the network as the block size increases. Only around 1% of the data is transmitted for a smaller block size, and this number increases as block size increases.

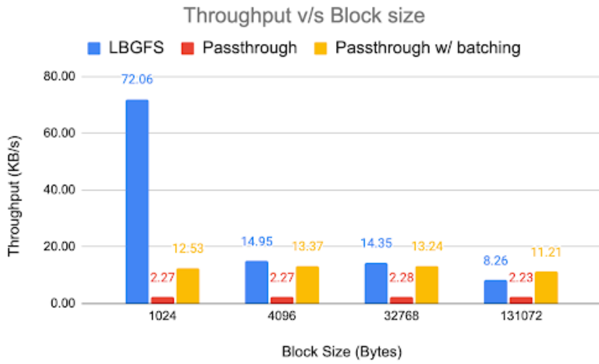


Figure 5: Throughput variation with respect to block size

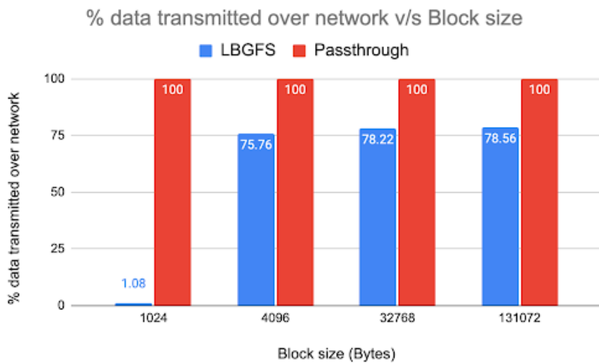


Figure 6: Percentage of total data transmitted over network against block size

2.4 Variation with number of distinct keys

Here we measure the performance variation with respect to the number of distinct events that get appended to the file in our workload. As seen in Figure 7, we observe high throughput for 4 distinct keys and diminishing throughput as we increase distinct events from 4 to 48. This suggests that our system works great for workloads with more repetitive appends to a file, i.e. when there exist only a few distinct events to be appended. Similarly in Figure 8 we see that as the number of distinct keys increases, more % of data is transmitted over the network, as now there are more possibilities generated for distinct blocks which may not match to the ones present on the server.

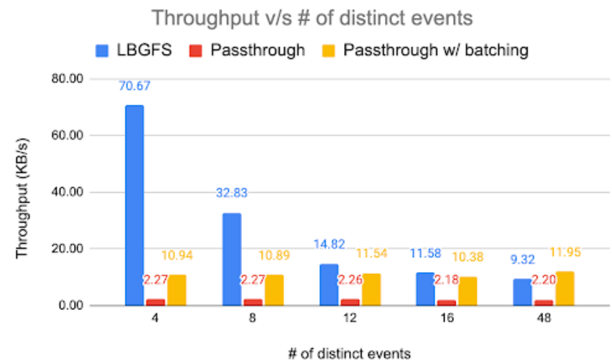


Figure 7: Throughput variation with respect to # distinct keys

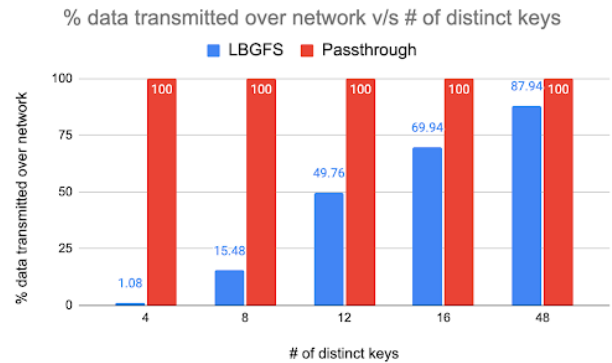


Figure 8: Percentage of total data transmitted over network against # distinct keys

2.5 Variation with key size

Here we measure our system's performance with respect to the size of the individual events that we append to our file. In Figure 9 we see an expected trend of throughput variation. As the key size increases, the number of

possibilities for distinct blocks reduces which leads to lesser percentage of total data being transmitted to the server (Figure 10), which also results in an increased throughput.

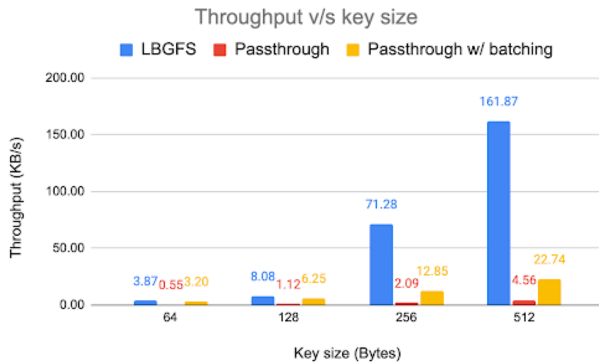


Figure 9: Throughput variation with respect to key size

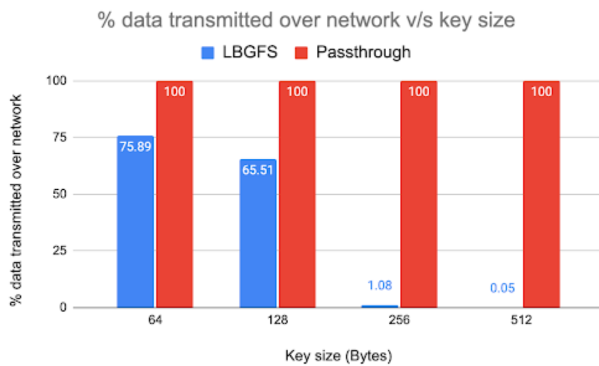


Figure 10: Percentage of total data transmitted over network against key size

3 TAKEAWAYS

- By implementing Low Bandwidth optimizations on HDFS, there is an improvement in saving both bandwidth and storage compared to base HDFS (or GFS).
- The extent of these benefits varies depending on the characteristics of the workload.
- The system warmup time is determined by the number of distinct blocks that are possible within a workload.
- This time can be shortened by creating new blocks through both client data transmission and server file reconstruction.