FLAT DATACENTER STORAGE

Paper-3
Presenter-Pratik Bhatt
fx6568
FDS Main discussion points

• A cluster storage system
• Stores giant "blobs" - 128-bit ID, multi-megabyte content
• Clients and servers connected by network with high bisection bandwidth
• For big-data processing (like MapReduce)
• Cluster of 1000s of computers processing data in parallel
• Compute and storage are logically separate
• Metadata management
• Networks
• Read and write performance
• Failure recovery
• Application performance
• FDS
FDS the storage system is pretty simple. It's a straight-up blob store, where blobs are arbitrarily large, but are read and written in 8MB chunks called tracts.

Blobs are identified with a 128-bit GUID, and tracts are 8MB offsets into the blob. Choosing 8MB means that the disks get essentially full sequential write bandwidth, since seek times are amortized over these long reads. It provides asynchronous APIs for read and write operations; pretty necessary for doing high-performance parallel I/O.

Data is placed on storage nodes (aka tractservers) through a simple hashing scheme. There's no Name node like in Hadoop. Instead, clients cache a tract locator table (TLT) which lists all of the tractservers in immutable order. Lookups for a tract in a blob are done by hashing the blob GUID, adding the tract number, and then moduling to index into the TLT.

Client access through the TLT is normally going to be sequential, since clients will mostly be sequentially scanning through the tracts of a blob.

To find the TLT index, the blob ID is hashed, but the tract number is just added. This is countered by taking 20 copies of the list of tractservers, shuffling each copy, and then concatenating them together. This super-list prevents auto-synchronizing lockstep, and also does a good job of balancing data distribution across tractservers.
High-level design

• Lots of clients
• Lots of storage servers ("tractservers")
• Partition the data
• Master ("metadata server") controls partitioning
• Replica groups for reliability
Why is this high-level design useful?

- 1000s of disks of space
- Store giant blobs or many big blobs
- 1000s of servers/disks/arms of parallel throughput
- Can expand over time - reconfiguration
- Large pool of storage servers for instant replacement after failure
- Recreate lost copies from replicas.
• **Data Placement**

• How do they place blobs on machines?
  • Break each blob into 8 MB tracts
  • "TLT" maintained by metadata server
  • for blob g tract i and TLT size n, slot = (hash(g) + i) mod n
  • TLT[slot] contains list of tractservers w/ copy of the tract
  • Clients and servers all have copies of the latest TLT table
Metadata Properties

- Metadata server is in critical path only when a client process starts. So tract size can be kept arbitrarily small.
- TLT can be cached since it changes only on cluster configuration (not each read & write), eliminate all traffic to metadata server in system under normal conditions.
- The metadata server stores metadata only about the hardware configuration, not about blobs. Because blob traffic is low, its implementation is simple and lightweight.
- Since the TLT contains random permutations of the list of tractservers, sequential reads and writes by independent clients are highly likely to utilize all tractservers uniformly and are unlikely to organize into synchronized convoys.
Pre-blob metadata

• Each blob has metadata such as its length and it is stored it in each blob’s special metadata tract (“tract -1”). Clients find a blob’s metadata on a tractservers using the same TLT used to find regular data.

• When a blob is created, the tractservers responsible for its metadata tract and creates that tract on disk and initializes the blob’s size to 0. When a blob is deleted, that tractservers deletes the metadata.

• Newly created blobs have a length of 0 tracts. Applications must extend a blob before writing past the end of it. The extend operation is atomic and is safe to execute concurrently with other clients, and returns the new size of the blob as a result of the client’s call.

• A separate API tells the client the blob’s current size. Extend operations for a blob are sent to the tractservers that owns that blob’s metadata tract. The tractservers serializes it, atomically updates the metadata, and returns the new size to each caller. If all writers follow this pattern, the extend operation provides a range of tracts and the caller may write without risk of conflict.
Dynamic work allocation

• In any system if a node falls behind, the only options for recovery is restarting its computation elsewhere. So here straggler period can represent a great loss in efficiency if most resources are idle while waiting for a slow task to complete.

• In FDS, since storage and compute are no longer co-located, the assignment of work to worker can be done dynamically, at fine granularity, during task execution. The best practice for FDS applications is to centrally give small units of work to each worker as it nears completion of its previous unit. This self-clocking system ensures that the maximum dispersion in completion times across the cluster is only the time required for the slowest worker to complete a single unit.
REPLICATION IN FDS

• To improve durability and availability, FDS supports higher levels of replication. When a disk fails, redundant copies of the lost data are used to restore the data to full replication.

What happen to replicas when application writes and reads?

When an application writes a tract, the client library finds the appropriate row of the TLT and sends the write to every tractserver it contains. Applications are notified that their writes have completed only after the client library receives write acknowledgments from all replicas. Reads select a single tractserver at random.

Applications are notified that their writes have completed only after the client library receives write acknowledgments from all replicas.

• Replication also requires changes to CreateBlob, ExtendBlobSize, and DeleteBlob.

• When a tractserver receives one of these operations, it executes a two-phase commit with the other replicas. The primary replica does not commit the change until all other replicas have completed successfully.

• FDS also supports per-blob variable replication.
Comparison with hadoop

• In FDS, data is only read and written to disk once. Hadoop's intermediate data is sorted and spilled to disk by the mapper, while FDS keeps this entirely in memory.

• FDS also streams and sorts data during the read phase, while Hadoop has a barrier between its map and reduce phases. Hadoop can do some pre-aggregation in the mapper with a combiner, but not as flexibly as in FDS, not on the reduce side, and not without hogging an entire task slot.

• FDS has better task scheduling, with a dynamic work queue and dynamic work item sizes.

• FDS uses a single process for both the read and write phases, so the JVM startup cost is gone, and there's no unnecessary movement of data between address spaces.
Questions and Answers

1. “Consider a centralized file server in a small computer science department. Data stored by any computer can be retrieved by any other. This conceptual simplicity makes it easy to use: computation can happen on any computer, even in parallel, without regard to first putting data in the right place.” Is GFS a centralized file system? To achieve good performance for an I/O-intensive program running on GFS, how does the data placement affect its performance? [Hint: consider the case where processes of the program run on the chunk servers.]

Ans. The GFS is a centralized file system. In order to simplify the design, increase its reliability and flexibility.

- The GFS got a simple centralized management system. As the system has centralized data storage system. But general GFS is a distributed file system. In GFS the data is distributed as chunks. Master does not store all the useful data on a single chunk rather it distributes it to balance load and disk space on chunk server. So in the scenario of the question the GFS can be viewed as centralized file system but generally the GFS is a distributed file system.

- In particular, a centralized master makes it much easier to implement sophisticated chunk placement and replication policies since the master already has most of the relevant information and controls how it changes. The fault tolerance by keeping the master state small and fully replicated on other machines.

- Scalability and high availability (for reads) are currently provided by our shadow master mechanism.
2. The root of this cascade of consequences was the locality constraint, itself rooted in the datacenter bandwidth shortage.” Show example consequences of relying on locality in program’s execution. Why does a sufficient I/O bandwidth help remove the constraint?

Ans. Stragglers is one example.

- Locality constraints can sometimes even hinder efficient resource utilization. One example is stragglers: if data is singly replicated, a single unexpectedly slow machine can preclude an entire job’s timely completion even while most of the resources are idle. The preference for local disks also serves as a barrier to quickly re-tasking nodes: since a CPU is only useful for processing the data resident there, re-tasking requires expensive data movement. If we consider the network is not oversubscribed then we could provide full bisectional bandwidth between the disks. So that there would be no distinction between local and remote disks. So constraints like re-tasking doesn’t require expensive data movement.
3. “In FDS, data is logically stored in blobs. ... Reads from and writes to a blob are done in units called tracts.” What are blob and tracts? Are they of constant sizes?

- Ans. Data is stored in logical blobs
- Byte sequences with a 128-bit Global Unique Identifiers (GUID)
- Blob can be any length up to system’s storage capacity.
- Divided into constant sized units called tracts
- Tracts are sized so random and sequential accesses have same throughput
- Both tracts and blobs are mutable
- Disk is managed by a tractserver process
- Read/write to disk directly without filesystem; tract data cached in memory
4. “In our cluster, tracts are 8MB”. Why is a tract in FDS sized this large?

Ans. Tracts are sized such that random and sequential access achieves nearly the same throughput. In our cluster, tracts are 8MB

• The tract size is set when the cluster is created based upon cluster hardware. For example, if flash were used instead of disks, the tract size could be made far smaller (e.g., 64kB). Every disk is managed by a process called a tract server.

• 1) If the tract size is small, the client need to write number of times for a large blob. The process would be slow. If the tract size is large, the number of writes will reduce.

• 2) The researches say that they have done several experiments on 10000 rpm disks. The experiments results that the throughput of the disk is good when the tract size is 8MB.
5. “Tractservers do not use a file system.” Explain this design choice.

- The FDS is storage system. It is not a file system. The tractservers are just raw disks, there will be no file system on them. They will just take all the blobs on the disks, divide them into sequential tracts and use those sequential tracks number them sequentially. The client just communicate with the FDS through an API that abstracts some of the complexity around the messaging layer.

- Tractservers and their network protocol are not exposed directly to FDS applications. Instead, these details are hidden in a client library with a narrow and straightforward interface.
Did you read the API?

• Blob is not similar to a file. Blobs are typically images, audio or other multimedia objects, though sometimes binary executable code is stored as a blob similar as a file. FDS support several file operations as create blob, delete blob, open blob and close blob. We can also do several operations like writing and reading the tract etc.

```c
// create a blob with the specified GUID
CreateBlob(GUID, &blobHandle, doneCallbackFunction);
```
7. "FDS uses a metadata server, but its role during normal operations is simple and limited:..." What are drawbacks of using a centralized metadata server? How does FDS address the issue?

- Metadata server’s role during normal operations is simple and limited. When the system is initialized, tractservers locally store their position in the TLT.
- This means the metadata server does not need to store durable state. So it will simplify its implementation.
- So, in case of a metadata server failure, the TLT is reconstructed by collecting the table assignments from each tractservers. So we will not lose all our data in case of metadata server failure.
- It collects a list of system’s active tracts servers and distribute it to TLT with k-replicaton, each entry has k tractservers.

To read or write tract number i from a blob with GUID g,

Eq: - \(\text{Tract}_\text{Locator} = (\text{Hash}(g)+i) \mod \text{TLT}\_\text{Length}\) (uses eq to replace big size table)

<table>
<thead>
<tr>
<th>Locator</th>
<th>Disk 1</th>
<th>Disk 2</th>
<th>Disk 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>C</td>
<td>G</td>
</tr>
<tr>
<td>3</td>
<td>D</td>
<td>E</td>
<td>G</td>
</tr>
<tr>
<td>4</td>
<td>B</td>
<td>C</td>
<td>F</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1,526</td>
<td>LM</td>
<td>TH</td>
<td>JE</td>
</tr>
</tbody>
</table>
8. How does FDS locate the track server that stores a particular tract of a given blob? Why does FDS first identify a tract locator (an index to an entry of tract locator table) and then in the entry to find the track server, rather than directly identifying a trackserver using a hash function without having such a table?

- Tract_Locator = (Hash(g) + i) mod TLT_Length

If no hash then,

- If we do not have such table then we do not have any space to put the replicated data.
- So, in case where we do not have such table, we have just one (original) copy of the data and if something wrong happened to that data or disk then we cannot able to recover that lost data.
To be clear, the TLT does not contain complete information about the location of individual tracts in the system.” and in the GFS paper “The master maintains less than 64 bytes of metadata for each 64 MB chunk.” Compare the TLT table with GFS’s use of a full chunk-chunkserver mapping table in the context of efficiency, scalability, and flexibility. [Hint: “It is not modified by tract reads and writes.” “Its size in a single-replicated system is proportional to the number of tractservers in the system...”]

- **In GFS**, chunk is large. So, once you ask some data location from master second time you do not need to ask location for other data in this system. This makes data excess very flexible. But single master in a system can still become potential bottleneck.

- **In FDS**, it uses a metadata server, but its role during normal operations is simple and limited i.e. it collects a list of the system’s active tractservers and distribute it to clients and this list is called tract locator table, or TLT. And when the system is initialized, tractservers locally store their position in the TLT. In case of a metadata server failure, the TLT is reconstructed by collecting the table assignments from each tractservers. So here scalability and efficiency is very good.
10. “In our 1,000 disk cluster, FDS recovers 92GB lost from a failed disk in 6.2 seconds.” What is normal throughput of a hard disk? What’s the throughput of this recovery? How can this be possible? [Hint: Describe the procedure of recovering from a dead tract server to answer this question. See Figure 2 and read Section 3.3]

- Normal throughput of HD 80-100 MB/S.
- Approximately 15 GB/S, very high.
- Possible because of large number of replication