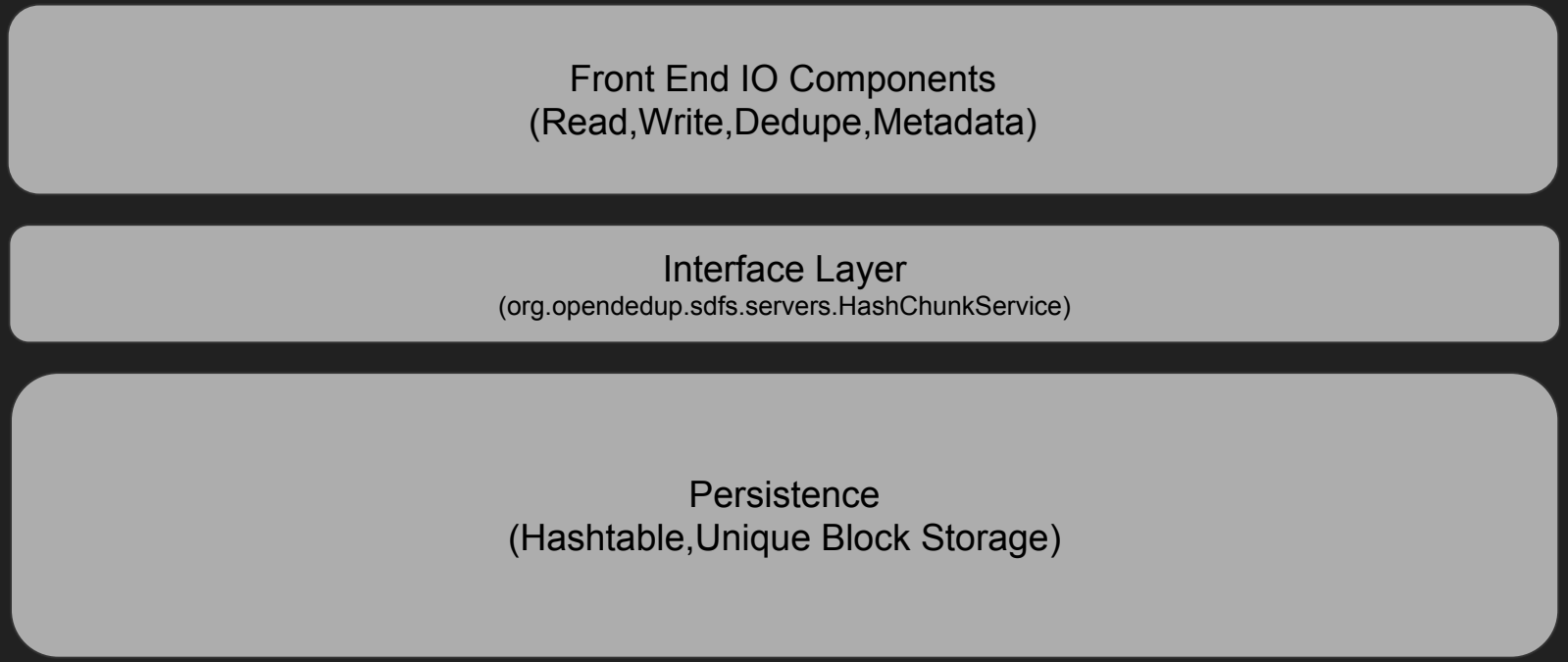
OpenDedupe

Architecture

Components


Front End IO Components
(Read,Write,Dedupe,Metadata)
Interface Layer
(org.opendedup.sdfs.servers.HashChunkService)
Persistence
(Hashtable,Unique Block Storage)

Front End Components

OST Layer

Fuse
(fuse.SDFS.SDFSFileSystem)

Http
(org.opendedup.sdfs.mgmt.Io)

Windows
(org.opendedup.sdfs.windows.fs.WinSDFS)

Data IO
(org.opendedup.sdfs.mgmt.io.DedupFileChannel)

Write Buffers
(org.opendedup.sdfs.io.WritableCacheBuffer)

Data Map
(org.opendedup.sdfs.mgmt.io.SparseDedupFile)

MetaData
(org.opendedup.sdfs.io.MetaDataDedupeFile)

Write Process

1. Write goes through filesystem interface
2. Writes to File Channel
3. Spooled into write buffer
4. Write Buffer flushed to Dedupe Hashing Process
5. Hashing Process breaks buffer into chunks org.opendedup.hashing.VariableHashEngine
6. Each unique chunk (org.opendedup.hashing.Finger) is sent to hash table for comparison. A unique chunk is determined by the number of unique hash fingerprints in WriteBuffer.

Write Process Continued

8. Hashtable sends back the if the data was already stored (dupe) and the location of the chunk (org.opendedup.sdfs.filestore.HashBlobArchive). The location is a long integer that represents the archive where the chunk is located. This data is passed as org.opendedup.collections.InsertRecord
9. The chunks associated with a WriteBuffer are then stored at an offset in a Map file (org.opendedup.collections.LongByteArrayMap). The offset is determined by the location where the data is written within the file.
    a. The WriteBuffer is stored as array with entries for each block. The collection is held in a org.opendedup.sdfs.io.SparseDataChunk entry is a org.opendedup.sdfs.io.HashLocPair it contains:
        i. Hash (16 bytes) - hash
        ii. HashBlobArchive where data is located (8 bytes) - hashloc
        iii. The length of the chunk (4 bytes)- len
        iv. Position is the buffer (4 bytes)- pos
        v. Offset in the chunk that is the start (4 bytes)- offset
        vi. The current lenth of the chunk (4 bytes)- nlen

Read Process

1. Read goes through filesystem interface
2. Reads File Channel at offset and length
3. FileChannel mashals writebuffer from SparseDedupe file at offset requested.
4. The writebuffer request all data it contains
    a. Reads the LongByteArrayMap at requested offset
    b. Reads HashLocPair for each entry
    c. Requests each chunk by hash and HashArchive id from HashTable
    d. Builds Buffer
5. The buffer is returned to FileChannel for Reading.
6. FileChannel reads buffer at offset
7. Returns Data

Write Process Continued

8. The chunks associated with a WriteBuffer are then stored at an offset in a Map file (org.opendedup.collections.LongByteArrayMap). The offset is determined by the location where the data is written within the file.
    a. The WriteBuffer is stored as array with entries for each block. The collection is held in a org.opendedup.sdfs.io.SparseDataChunk entry is a org.opendedup.sdfs.io.HashLocPair it contains:
        i. Hash (16 bytes) - hash
        ii. HashBlobArchive where data is located (8 bytes) - hashloc
        iii. The length of the chunk (4 bytes)- len
        iv. Position is the buffer (4 bytes)- pos
        v. Offset in the chunk that is the start (4 bytes)- offset
        vi. The current lenth of the chunk (4 bytes)- nlen

Back End Components

HashChunkServiceInterface (Put,Get,Delete)

AbstractHashesMap (HashMap Interface)

AbstractChunkStore (Block Pesistence Interface)

HashBlobArchive (Data Spooler)

AbstractBatchStore (Spooled Data Writer)

Write Process - Pickup from Step 6 in Front End

1. Each chunk (org.opendedup.hashing.Finger) is sent to hash table for comparison (Step 6 from Front End Process)
2. Call to org.opendedup.sdfs.servers.HashChunkService.writeChunk.
3. writeChunk Calls HashStore.addHashChunk
4. addHashChunk call AbstractChunkStore.put
    a. Hash is check to see if it exists
        i. If Exists reference count is incremented
        ii. New ref count is persisted
        iii. HashArchive location is returned in InsertRecord
        iv. Dupe is set to true in InsertRecord
    b. If not exist Data is persisted to local spool (HashBlobArchive)
    c. HashBlobArchive returns archive id
    d. HashTable put of hash as key [archive id + refcount] as value
    e. Dupe is set to false in InsertRecord

HashBlobArchive Write Process

1. HashBlockArchive spools to local directoy spool contains 1+ archives associated with current active or not closed Archives
    a. Archives are immutible (Except when data is local)
    b. Archives contain 1+ chucks
    c. Archives Contain Unique Data
    d. Each Archive has an associated SimpleByteArrayLongMap that contains hash to location in achive info
2. Data written to active archive
    a. Compressed
    b. Encrypted
    c. Written
3. Archive ID Returned to Upper level process

Archive Persist

- Archive Becomes immutable when
  - Becomes larger that selected size
  - Times out
- Archive is written to AbstractBatchStore
- AbstractBatchStore writes Archive and SimpleByteArrayLongMap to desired storage for later recovery
- Data is cached to local cache as well

Read Process - Pickup from Step 4 in Front End

1. Each chunk is requested from org.opendedup.sdfs.servers.HashChunkService.fetch
2. Fetch includes hash and archive id (HashBlobArchive) Location
3. HashBlobArchive requests archive id from cache
4. If ID not in cache Archive is requested from cloud or fails if storage is local
5. Once Archive is in cache SimpleByteArrayLongMap is read for achive
6. SimpleByteArrayLongMap include hash + location is archive where data can be read
7. Data is read at location and returned

Advantages/Disadvantages

- Advantages
  - Quicker Deployment. Can use off the shelf components such as Hazelcast or Apache Ignite to support Distributed HashTable
  - Scale out hashtable by adding more nodes
  - High Performance - Reads and Writes directly to CFS
  - Data protection provided by storage layer
- Disadvantage
  - Relies on S3 Bucket to be unlimited in scale
  - S3 Bucket Single point of failure
  - Storage scales independently of hashtable