3. PROPOSED SYSTEM

AppDedupe, an optimized distributed deduplication is proposed that uses application awareness, data similarity and locality through maintaining application data at file level and similar data in same storage. It is a technique that performs deduplication at block level. This is introduced because it performs Application aware independent deduplication, reduces data redundancy with load balancing and improves scalability. With all these advantages it moves to the higher level when compared to those discussed in Literature survey.

3.1 AppDedupe system:

In this section, the following three design principles are discussed to govern with AppDedupe system design.

Throughput: The deduplication throughput should scale with the number of nodes by parallel deduplication across the storage nodes.

Capacity: Similar data should be forwarded to the same deduplication node to achieve high duplicate elimination ratio.

Scalability: The distributed deduplication system should easily scale out to handle massive data volumes with balanced workload among nodes.

To achieve high deduplication throughput and good scalability with negligible capacity loss, a scalable inline distributed deduplication framework is discussed in this section.

3.2 System Overview:

The architecture of our distributed deduplication system is shown in Fig.3.1. It consists of three main components: clients dedupe storage nodes and director. In what follows, we first show the architecture of our AppDedupe system. Then we present our two-tiered data routing scheme to achieve scalable performance with high deduplication efficiency. This is followed by the description of the application-aware data structures for high deduplication throughput in deduplication nodes.

Clients:

There are three main functional modules in a client: data partitioning, chunk fingerprinting and data routing. The client component stores and retrieves data files, performs data chunking with fixed or variable chunk size and super-chunk grouping in the data partitioning module for each data stream, and calculates chunk fingerprints by a collision-resistant hash function, like MD5, SHA-1 or SHA-2, then routes of each super-chunk to a dedupe storage node with high similarity by the two-tiered data routing scheme. To improve distributed system scalability by saving the network transfer bandwidth during data store, the clients determine whether a chunk is duplicate or not by batching chunk fingerprint query in the deduplication node at the super-chunk level before data chunk transfer, and only the unique data chunks are transferred over the network.

Dedupe storage nodes:

The dedupe server component consists of three important functional modules: application-aware similarity index lookup, chunk index cache management and parallel container management. It implements the key deduplication and storage management logic, including returning the results of application-aware similarity index lookup for data routing, buffering the recent hot chunk fingerprints in chunk index cache to speed up the process of identifying duplicate chunks and storing the unique chunks in larger units, called containers, in parallel.

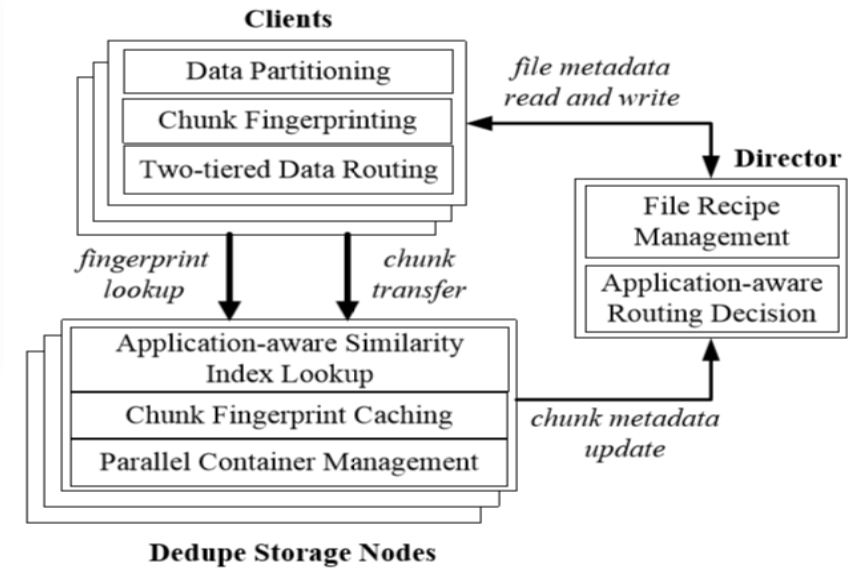

Fig.3.1 The architectural overview of AppDedupe

Director:

It is responsible for keeping track of files on the dedupe storage node, and managing file information to support data store and retrieve. It consists of file recipe management and application-aware routing decision. The file recipe management module keeps the mapping from files to chunk fingerprints and all other information required to reconstruct the file. All file-level metadata are maintained in the director.

All file-level metadata are maintained in the director. The application aware routing decision module selects a group of corresponding application storage nodes for each file, and gives the client a feedback to direct super-chunk routing. The director supports up to two servers in an active/passive failover to avoid the single node failure with high availability.

4. TOOLS AND TECHNIQUES

4.1 Two-tiered data routing scheme:

As a new contribution the two tiered data routing scheme is presented including: the file-level application aware routing decision in director and the super chunk level similarity aware data routing in clients.

The application aware routing decision is inspired by application difference redundancy analysis. It can distinguish from different types of application data by exploiting application awareness with filename extension, and selects a group of dedupe storage nodes as the corresponding application storage nodes, which have stored the same type of application data with the file in routing. This operation depends on an application route table structure that builds a mapping between application type and storage node ID. The application aware routing algorithm is shown in Algorithm 1, which performs in the application aware routing decision module of director.

The similarity aware data routing scheme is a stateful data routing scheme motivated by super-chunk resemblance analysis. It routes similar super-chunk to the same dedupe storage node by looking up storage status information in only one or a small number of nodes, and achieves near-global capacity load balance without high system overhead. In the data-partitioning module, a file is first divided it into c small chunks, which are grouped into a super-chunk S. Then, all the chunk fingerprints {fp1, fp2…, fpc} are calculated by a cryptographic hash function in the chunk fingerprinting module. The data routing algorithm, shown in Algorithm 2, performs in the data routing module of the clients.

Handprinting based data routing scheme can improve load balance for the m application dedupe storage nodes by adaptively choosing least loaded node in the k candidate nodes for each super-chunk. It can be proved that the global load balance can be approached by virtue of the universal distribution of randomly generated handprints by cryptographic hash functions. Its consistent hashing based data assignment is scalable since it can avoid re-shuffling all previously stored data when adding or deleting a node in the storage cluster.

Algorithm 1: Application Aware Routing Algorithm

Input: The full name of a file, *fullname*, and a list of all dedupe storage nodes {S1, S2, …, SN}

Output: A ID list of application storage node, ID_list ={A1, A2, … , Am}

1. Extract the filename extension as the application type from the file full name *fullname*, sent from client side;

2. Query the application route table in director, and find the dedupe storage node Ai that have stored the same type of application data; We get the corresponding application storage nodes ID_list={A1, A2, … , Am} ⊆ {S1, S2, …, SN};

3. Check the node list: if ID_list=□ or all nodes in ID_list are overloaded, then add the dedupe storage node SL with lightest workload into the list ID_list ={SL};

4. Return the result ID_list to the client.

Algorithm 2: Handprinting Based Stateful Data Routing

Input: A chunk fingerprint list of super-chunk S in a file, {fp1, fp2, … , fpc}, and the corresponding application storage node ID list of the file, ID_list={A1, A2, … , Am}.

Output: A target node ID, i

1. Select the k smallest chunk fingerprints {rfp1, rfp2, …, rfpk} as a handprint for the super-chunk S by sorting the chunk fingerprint list {fp1, fp2, …, fpc}, and sent the handprint to k candidate nodes with IDs mapped by consistent hashing in the m corresponding application storage nodes;

2. Obtain the count of the existing representative fingerprints of the super-chunk S in the k candidate nodes by comparing the representative fingerprints of the previously stored super-chunks in the application-aware similarity index, are denoted as {r1, r2, …, rk};

3. Calculate the relative storage usage, which is a node storage usage value divided by the average storage usage value, to balance the capacity load in the k candidate nodes, are denoted as {w1, w2… wk};

4. Choose the dedupe storage node with ID i that satisfies ri/wi = max{r1/w1, r2/w2, …, rk/wk} as the target node.

4.2 Interconnect communication:

The interconnect communication is critical for the design of AppDedupe. The detailed operations carried out when storing and retrieving a file are mentioned here. A file store request is processed as shown in the Fig.4.1 client sends a PutFileReq message to the director after file partitioning and chunk fingerprinting. The message includes file metadata like: file ID (the SHA1 value of file content), file size, file name, timestamp, the number of super-chunk in the file and their checksums. The director stores the file metadata as a file recipe, and makes sure that there has enough space in the distributed storage systems for the file. It also performs the application aware routing decision to select a group of corresponding application storage nodes for each file.

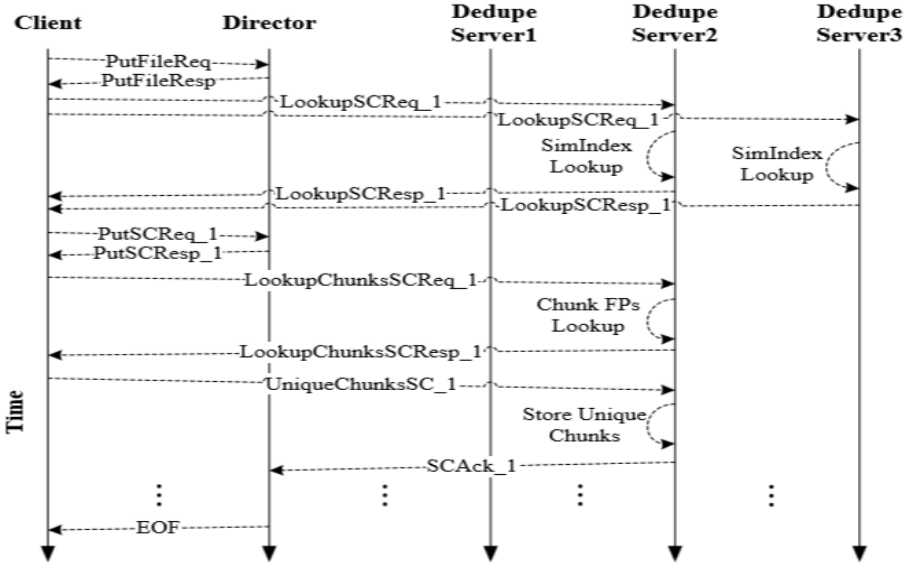

Fig.4.1 Message exchanges for store operation

The director replies to the client with the file ID and corresponding application storage no delist in *PutFileResp* message. After received the *PutFileResp,* the client sends k *LookupSCReq* requests to the k candidate dedupe storage nodes for each super-chunk in the file, respectively, to lookup the application-aware similarity index in dedupe storage nodes for the representative finger prints of the super-chunk. These candidate nodes reply to the client with a weighted resemblance value for the super-chunk. The client selects a candidate node as the target route node to store the super-chunk, and notifies the director its node ID by PutSC Req message. Then,

the client sends all chunk fingerprints of the super-chunk in batch to the target node to identify whether a chunk is duplicated or not. After the lookup of chunk fingerprints, the target dedupe storage node replies to the client with a list of unique chunks in the super-chunk. Moreover, the client only needs to send the unique chunks in the super-chunk to the target node in batch. We repeat the steps for each super-chunk, until the end of file is reached.

The process of retrieving a file is also initiated by a client request GetFileReq to the director, as depicted in Fig. 4.2. The director reacts to this request by querying the file recipe, and forwards the GetFileResp message to the client. The GetFileResp contains the super-chunk list in the file and the mapping from super-chunk to the dedupe storage node where it is stored. Then, the client requests each super-chunk in the file from the corresponding dedupe storage node with GetSuperChunk message. The dedupe server can retrieve super-chunk from data containers, and the performance of restore process can be accelerated. Finally, the client downloads each super-chunk and uses the checksums of super-chunks and file ID to verify the data integrity.

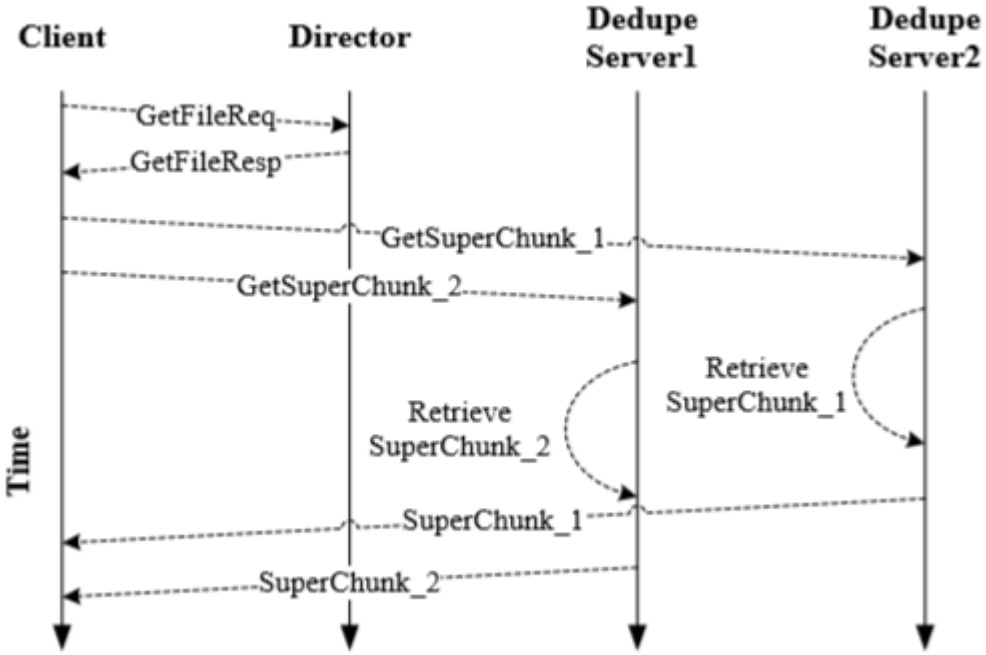

Fig.3.2 Message exchanges for retrieve operation

4.3 Some key data structures:

The salient features of the key data structures designed for the deduplication process in the director and dedupe storage nodes are outlined. As shown in Fig. 4.3, an application route table is located in the director to conduct application aware routing decision, while to support high deduplication throughput with low system overhead, a chunk fingerprint cache and two key data structures: application-aware similarity index and container, are introduced in our dedupe storage architecture design.

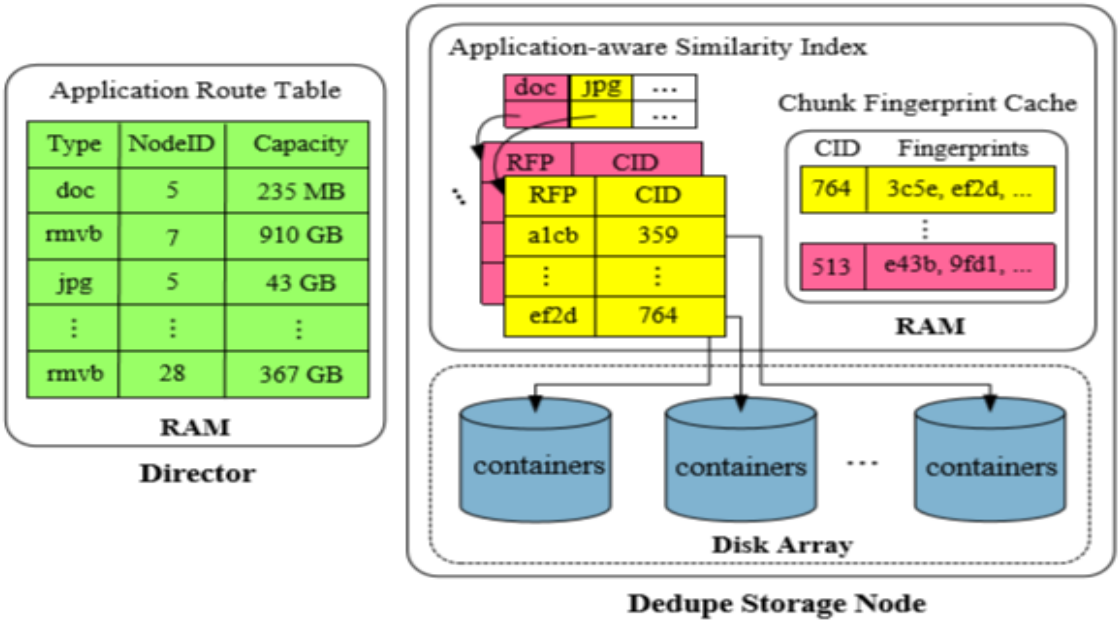

Fig. 4.3 Key data structures in dedupe process.

Application route table is built in director to conduct application aware routing decision. Each entry of the table stores a mapping from application type to node ID and the corresponding capacity for that kind of application data in the storage node. The director can find out the application storage node list for a given application type, and calculate the workload level with storage utilization in nodes. In consideration of the application level and node level are both coarse grained, the whole application route table can easily be fit into the director memory to speed up the query operations. .

Application-aware similarity index is an in-memory data structure. It consists of an application index and small hash-table based indices classified by application type. According to

the accompanied file type information, the incoming super-chunk is directed to a small index with the same file type. Each entry contains a mapping between a representative fingerprint (RFP) of super-chunk handprint and the container ID (CID) where it is stored. Since handprinting has very low sampling rate, it is much smaller than the traditional chunk fingerprint disk index that builds a mapping from all chunk fingerprints to the corresponding containers that they're stored in. To support concurrent lookup operations in application-aware similarity index by multiple data streams on multicore deduplication nodes, a parallel application-aware similarity index lookup design is adopted and controls the synchronization scheme by allocating a lock per hash bucket or for a constant number of consecutive hash buckets. Container is a self-describing data structure stored in disk to preserve locality, which includes a data section to store data chunks and a metadata section to store their metadata information, such as chunk fingerprint, offset and length. The dedupe server design supports parallel container management to allocate, deallocate, read, write and reliably store containers in parallel. For parallel data store, a dedicated open container is maintained for each coming data stream, and a new one is opened up when the container fills up. .

Besides the fore mentioned data structures, the chunk fingerprint cache also plays an important role in deduplication performance improvement. It keeps the chunk fingerprints of recently accessed containers in RAM. Once a representative fingerprint is matched by a lookup request in the application-aware similarity index, all the metadata section belonging to the mapped container are perfetched into the cache to speedup chunk fingerprint lookup. When the cache is full, a reasonable cache replacement policy, like Least-Recently-Used (LRU), is applied to make room for future prefetching and caching.