Distributed Duplicate Detection in Post-Process Data De-duplication

Atish Kathpal* *Atish.Kathpal@netapp.com* NetApp Inc.
Matthew John* *tmatthewjohn1988@gmail.com* Disney Playdom
Gaurav Makkar *gmakkar@netapp.com* NetApp Inc.
*Student Authors

Abstract

Data De-duplication is essentially a data compression technique for elimination of coarse-grained redundant data. A typical flavor of de-duplication detects duplicate data blocks within the storage device and de-duplicates them by placing pointers rather than storing multiple copies at various places within the disk. Since the advent of de-duplication the conventional approach has been to scale-up de-duplication at a storage controller by using more of the controller resources. This approach has led to several bottlenecks including the most evident one of hogging controller resources, in-turn leading to limiting the number of concurrent de-duplication threads running on the controller, finally ending up with poor de-duplication performance. Going by the rate at which we are experiencing data explosion, with data becoming the core entity separating one organization from other, high performing scalable de-duplication is one challenge organizations are already starting to face. Through the current effort, we propose a scalable design of a distributed de-duplication system which leverages clusters of commodity nodes to scale-out suitable tasks of a typical de-duplication system. We explain our distributed duplicate detection workflow, implemented in Hadoop's map-reduce programming abstraction. We also discuss the performance statistics we obtained with the scale-out de-duplication model.

1. Introduction

Data De-duplication is essentially a data compression technique for elimination of coarse-grained redundant data. Eliminating redundant data significantly improves storage and bandwidth efficiency. Most commercial and research storage systems deploy de-duplication to improve their storage utilization. This benefit, however, comes at a cost as de-duplication processes are both CPU and I/O intensive. Post-process de-duplication ensures that de-duplication does not come in way of data influx, as it is carried out after data has been written to the disks. However, post-process de-duplication has to suffer due to availability of limited resources as serving I/O is the major consideration for storage controllers. This really is a performance bottle neck, given the uncontrolled explosion of data in recent years, with storage systems having made a fast transition from giga to peta-scales. Once a system reaches its scalability limits, storage administrators can do little to increase the storage efficiency.

This work describes a mechanism for addressing the scalability issues with today's de-duplication engine implementations. We show that there are phases of de-duplication process, which have inherent concurrency and we leverage Hadoop (a distributed computing framework) to exploit the same. Our results show that with four concurrent execution units (Virtual Machines), we can match the performance of de-duplication at the controller. This opens up the opportunity for a much larger number of simultaneous de-duplication streams besides freeing up the controller resources for serving IOPs.

1.1 Need for Distributed De-duplication and Related work

As the data to be managed at organizations grows to possible exa-scales in coming years, one of the biggest challenges we are faced with is the aspect of managing de-duplication of this data. Efforts to address the scalability issues of de-duplication, so far, have been in the direction of using multi-threading [1], various forms of caching [7], file segmentation etc in pursuit of scaling up the de-duplication performance. Such implementations lead to dedication of more controller resources to the de-duplication process which potentially hits the I/O performance for the controller. This has the side-effect of artificially limiting the degree to which one would want to scale the process because the controller resources are best used for serving data and with every increment of resources that we take away from that core goal (for internal processes, like de-duplication), it has a direct impact to the number of IOPs we can serve. A common practice to escape such side-effects is to keep the number of de-duplication processes running on given storage controller to a small number to allow keeping the controller resource utilization under check. Efforts have also been made in the direction of optimizing the algorithms in pursuit of improved de-duplication efficiency and performance, but none of such research efforts [5, 6] seem to boast of a scalable design and suffer from same issues of high resource utilization at the storage controller. There have also been efforts to perform de-duplication at a higher level (file or object

or variable sized blocks) [8] rather than block level, for faster results, but they again suffer from scalability issues in presence of billions of objects. For post-process de-duplication, it is assumed that the data center would have "sleep" times when the application load would be significantly lesser, which is when the de-duplication tasks are best scheduled, but with data center's serving data worldwide, such assumptions are no longer valid. It's inevitable for the de-duplication process to compete with IOPs for resources at the controller.

We present a scalable distributed de-duplication design, scales out the de-duplication tasks.

2. Design

In our design, we propose a scalable enhancement to current de-duplication systems. The design works best in a clustered storage environment. De-duplication can be sub-divided into two basic tasks, that of detecting duplicates and sharing the duplicates (instead of storing multiple instances of the same object). Our design advocates to fan-out the compute and memory intensive, detection of duplicates phase of de-duplication process, to a cluster of compute nodes which carry out processing of block fingerprints (typically a hash of data in the block) and detection of candidate duplicate blocks in a parallel fashion. The cluster of compute nodes is constructed using commodity hardware and is part of the storage-tier. Leveraging a cluster of commodity nodes allows us to scale out the said task of de-duplication system. The duplicate detection phase of post process de-duplication is entirely based upon identification of duplicates among the list of collected fingerprints (which are nothing but computed hash indexes of data chunks in the storage system). The fingerprints are typically much smaller in size as compared to the data chunk they represent. Owing to the much smaller size of fingerprint database (in comparison to the size of original dataset), one does not have to move chunks of original dataset for the purpose of scaling out of duplicate detection phase. The sharing phase, however, is understandably best done where the actual data is present as it involves byte-wise comparison of data chunks, ensuring they are identical before de-duplicating them. We fan-out the fingerprint database for a given volume to our compute cluster for further processing i.e. duplicate detection, as shown in Fig 1.

2.1. Benefits

As was mentioned in the introduction, it is a common practice to limit the number of concurrent de-duplication threads on a controller in pursuit of not compromising on the core goal of serving I/O operations. With the aid of proposed design, we can prevent the critical storage controller resources from being spent in a workload which can be easily scaled-out. The other important aspect of this design is that it allows for leveraging cheaper resources which are not at the controller, but still are part of the storage tier (by using commodity hardware). This has an effect of lowering the overall cost of the solution, while allowing for higher IOPs to be served through saved resources at the controller. Another interesting aspect of this design is that it involves non-shared coarse-grained parallelism which enables better scalability. Thus with our design, the number of concurrent de-duplication processes could be increased by an order of magnitude without taking additional resources at the controller, by adding more commodity hardware to the storage cluster.

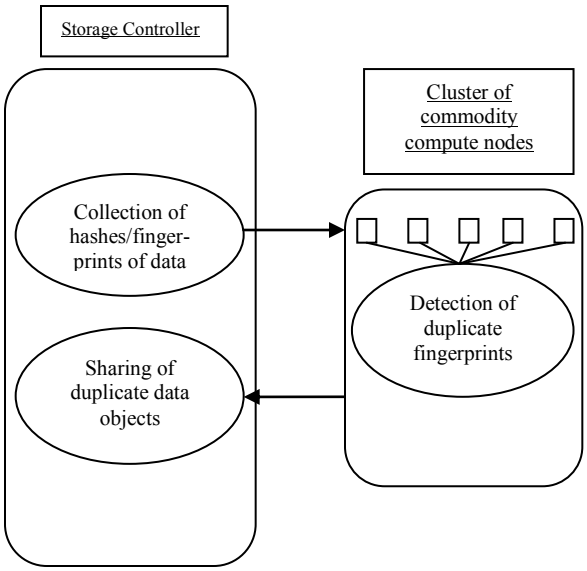
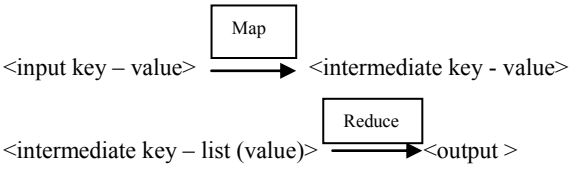


Fig.1: Distributed de-duplication

3. Implementation

As explained in the design, our intention is to take the duplicate-detection phase of offline de-duplication to an external cluster of compute nodes using commodity hardware. Hadoop, an Apache open source distributed computing framework serves as a potential candidate for the task since the duplicate-detection task is effectively sorting the fingerprints of the blocks and identifying the duplicates among them. This is an ideal use case of the Map-Reduce programming abstraction [2] which is used in Hadoop. Map-Reduce splits the compute problem into two stages: Map and Reduce. Map is a transformation function and Reduce acts as an aggregation function. Every record in the data is interpreted as a key-value pair in Map-Reduce programming paradigm.



Hadoop Map-Reduce fits very well in our case since it is distributed and has an inherent sort capability which sorts all the intermediate data on the basis of the intermediate key. This stage is called the Shuffle/Sort phase in Hadoop Map-Reduce. We try to exploit this feature and detect the duplicate fingerprint records. We also leverage the use of Hadoop Distributed File System [9] which acts as the source file system for the Map-Reduce jobs that are run.

We used NetApp offline de-duplication [3] to accommodate our Hadoop-based duplicate detection framework. We replaced the duplicate detection stage of NetApp de-duplication with our duplicate detection mechanism that uses Hadoop MapReduce.

The Hadoop-based duplicate detection workflow that we implemented, includes the following stages:

- Receive the fingerprints from the storage controller.
- Generate fingerprint database and store it persistently on the Hadoop Distributed File System (HDFS). The same can be further used for incremental offline de-duplication. It also aids in recovery and serves as a checkpoint, we can resume de-duplication from the previously saved fingerprint database, in case the current job of duplicate detection fails.
- Generate the duplicate records from the fingerprints and send it back to the storage controller. This triggers the rest of the de-duplication phase in the storage controller.

The following section would give finer implementation details.

3.1 Hadoop-based Duplicate detection

The structures involved in the data flow of duplicate detection:

Fingerprint:

Fingerprint	Block attributes

Duplicate:

Block1 attributes	Block2 attributes

The input to our Hadoop cluster is the fingerprints of all the blocks with no inherent ordering (Fig. 2). We sort these ingested fingerprints using Fingerprint as the key. The sorted collection is then used to detect the duplicates (fingerprints with same Fingerprint attribute). The duplicate record (shown above) contains the block attributes of the two blocks which are going to undergo sharing during de-duplication. Once we have detected the duplicates using our map-reduce jobs, send them back as a single metafile to the storage controller for further duplicate sharing phase of NetApp de-duplication. Meanwhile we also preserve a copy of the sorted fingerprints called the fingerprint database (FPDB) within the Hadoop cluster. Fig. 2 gives the outline of the MapReduce (MR) modules we implemented for duplicate detection.

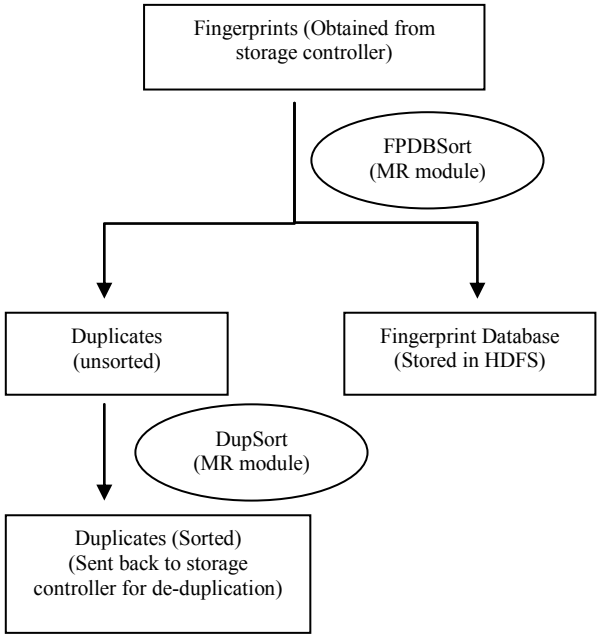


Fig.2: Hadoop based duplicate detection (MR: MapReduce)

We run 2 MapReduce jobs serially to perform duplicate detection in the Hadoop cluster.

1) FPDBSort

This MapReduce module takes the fingerprints as the input from the HDFS and generates the Fingerprint Database (the sorted fingerprint file) and the duplicates file. The duplicates file generated contains the duplicate records but not in sorted order. The different parameters of the module are as stated below:

- Input: Fingerprints – directory from HDFS which contains the fingerprints.
- Output: FPDB – file which contains the fingerprints in sorted order.
- Dup_unsorted – file which contains the duplicate records but not sorted.

```
Map (byte [] fingerprint_record)
    For each fingerprint_record
    Emit (key, value);
    //key – Fingerprint of block
    //value – Block attributes

Reduce (byte [] key, Iterator <byte []> Values)
    1) For every value in values, emit (key, value)
       // this gives the FPDB file
    2) For every consecutive values, value_i and
       value_j
       Emit (value_i, value_j)
       //this populates the Dup_unsorted file
```

Fig. 3: Map-Reduce algorithm of FPDBSort

2) DupSort

This MapReduce module takes "Dup_unsorted" file from HDFS, sorts the duplicate record with a specific comparator and writes the output to an output stream that sends the data to the Storage controller via Socket communication. The different parameters of the module are as stated below.

- Input: "Dup_unsorted" file from HDFS
- Output: Sorted duplicates sent over the socket to the Storage controller.

```
Map (byte [] duplicate_record)
    For each duplicate_ record
    Emit (key, value);
    //key – duplicate_record
    //value – NULL

Reduce (byte [] key, Iterator <byte []> Values)
    For every value in Values, emit (key, value)
    // Output is sent back to controller via socket
```

Fig.4: Map-Reduce algorithm of DupSort

4. Performance

As mentioned previously, we implemented the Hadoop based scale out duplicate detection by modifying the offline de-duplication of NetApp. We present the performance comparisons made between the duplicate detection of NetApp de-duplication and duplicate detection using our Hadoop MapReduce framework. The size of fingerprint data-structure in our experiment is 32 bytes corresponding to a 4 KB data block.

Following are the datasets used for the de-duplication experiments:

- Dataset A – 498 GB (78% duplicates)
- Dataset B – 445 GB (67% duplicates)
- Dataset C – 217 GB (86% duplicates)
- Dataset D – 197 GB (53% duplicates)

We populated the datasets using Iozone Filesystem Benchmark tool [4]. We setup the Hadoop cluster on an ESX Hypervisor with four dedicated 32-bit virtual machines taking up the role of the Hadoop nodes. The configuration of each of the nodes is as given below:

Number of CPUs: 2
Memory: 4 GB
Operating system: Ubuntu 10.04

Duplicate detection phase of NetApp de-duplication was carried out in a NetApp Storage controller. The configuration of the controller we used, is as mentioned below:

Number of CPUs: 4
Memory: 16 GB
NVRam: 2 GB
Operating system: Data ONTAP

We recorded the time taken during the duplicate detection of NetApp de-duplication and distributed duplicate detection using Hadoop MapReduce (including the time taken for network transfer of metadata from controller to the Hadoop cluster and back.) with different Hadoop cluster sizes. Table 1 gives the performance statistics we observed during the experiments. The results described in Table 1 are the time taken by the de-duplication process, starting from the fingerprint collection stage, until the end of detection of duplicates. Note that our Hadoop implementation optimizes only the duplicate detection phase.

In-order to simulate a realistic workload environment in the controller, we ran de-duplication along with a simulated workload where six threads were assigned to perform read/writes of 1GB data each on the controller. We measured the controller performance with and without the simulated workload. The performance results (Fig 5) clearly indicate that our distributed duplicate detection framework (with four commodity/VM nodes) out-performs the standard duplicate detection implemented in controllers, under the effect of simulated workloads. As the magnitude of data increases, the Hadoop framework is expected to perform efficiently since it is tailor-made for big data processing with reliability. We also observed that scale-out duplicate detection freed up approximately 512MB of memory while running de-duplication on a single volume. Fig. 5 gives a bar-graph representation of the statistics obtained.

5. Conclusion and Future Work

As it is evident from the performance results obtained from our experiment, Hadoop-based distributed duplicate detection is a good enhancement to the single node duplicate detection run on storage controllers. It outperforms standard offline de-

duplication mechanism due to its scale-out capability. It also appears to be a good use case of leveraging commodity hardware in the present data storage scenario. Another positive that can be derived from this initiative would be to free the storage controller resources that could be utilized for other higher priority housekeeping functionalities and serve the main purpose of data storage more effectively. The same set of Hadoop nodes, can also be used to run other suitable applications like management and the like, which is beyond the scope of this paper.

Going ahead, we plan to work on the lines of increasing the number of concurrent de-duplication streams that could be initiated by the storage controller and try to address the bottlenecks we encounter in this context. We also intend to investigate on how to achieve end-to-end scale-up in the rate of overall de-duplication in this scenario. We are also interested in evaluating the results using a practically larger Hadoop cluster and huge datasets that could be an indicator of the storage scenario in the years to come. Another future work prospect would be on the lines of assessing how to scale-out other stages of the de-duplication using commodity hardware – fingerprint (hash value of the data blocks) generation and even data processing modules involved in duplicate sharing phase of de-duplication. We would also like to explore if the memory at the commodity Hadoop nodes, could be utilized as a layer of secondary cache for the controller when some of the nodes are idle.

Dataset	Duplicate detection on Storage controller (without workload) (min)	Duplicate detection on Storage controller (with workload) (min)	Duplicate detection on 2-node/VM Hadoop Cluster (min)	Duplicate detection on 4-node/VM Hadoop Cluster (min)
A	29	35	38	30
B	28	32	36	23
C	14	21	20	14
D	13	19	15	10

Table 1: Total time taken during fingerprint collection and duplicate detection stages, with and without the Hadoop implementation

6. Acknowledgements

We would like to acknowledge the key inputs from VR Bharadwaj on modifying NetApp A-SIS for us to carry out our experiments. We thank members of NetApp Advanced Technology Group for their guidance. We would also like to thank the Apache Hadoop users community for their timely tips and help.

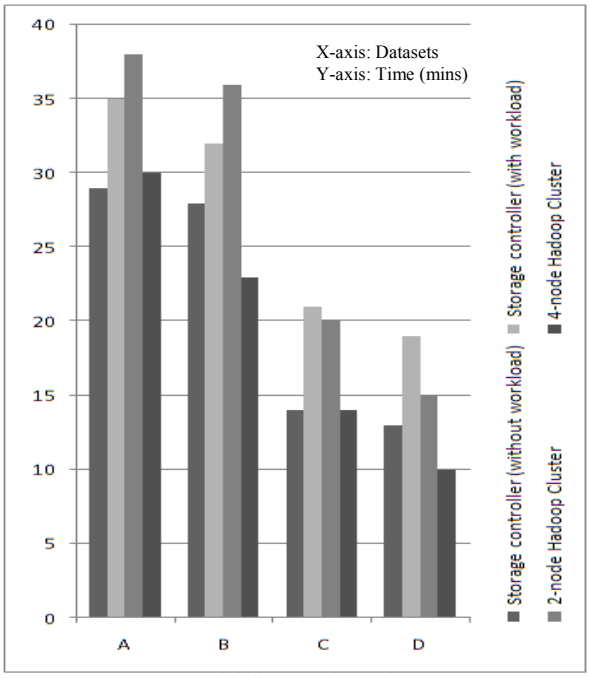


Fig.5. Bar-graph indicating the performance results

6. References

[1] Petros Efstathopoulos and Fanglu Guo. Rethinking Deduplication Scalability. *HotStorage 2010.*

[2] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *OSDI 2004*.

[3] Carloz Alvarez. NetApp Technical Report: NetApp Deduplication for FAS and V-Series Deployment and Implementation Guide.

[4] William D. Norcott and Don Capps. Iozone Filesystem Benchmark. http://www.iozone.org.

[5] Sean Quinlan and Sean Dorward, "Venti: A new approach to archival storage," in FAST '02: Proceedings of the Conference on *File and Storage Technologies*, Berkeley, CA, USA, 2002, pp. 89–101, USENIX Association.

[6] OpenDedup, "A userspace deduplication file system (SDFS)," March 2010, http://code.google.com/p/opendedup/.

[7] ZHU, B., LI, K., AND PATTERSON, H. Avoiding the disk bottleneck in the Data Domain deduplication file system. In *File and Storage Technology Conference (2008)*.

[8] Shai Halevi, Danny Harnik, Benny Pinkas, and Alexandra Shulman-Peleg. Proofs of Ownership in Remote Storage Systems. *CCS 2011*

[9] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler. The Hadoop Distributed File System. *MSST2010*