**Study and Implementation of Data Deduplication**

COMP90055 – COMPUTING PROJECT

**PRAMUDITA SANTOSO (731188) –** [**psantoso@student.unimelb.edu.au**](mailto:psantoso@student.unimelb.edu.au)

**Abstract**

Many data deduplications have been implemented within the field of File System. A quick research has been done and none of them are designed to be in a multiplatform environment. My aim in this project is to provide a multiplatform approach of Data Deduplication in the context of interoperable Java Virtual Machine utilisation. Additionally, a small scale study of parameter exploration is also being conducted – leading to a comprehensive result and conclusion of this project – in a realistic setting and environment.

1. **Introduction**

The existence of extraordinary range of information storage facilities and massive internet-accessed cloud storage systems has created a common problem of storing duplicate data of identical and similar files. A technology that eliminates this redundancy leads to substantial storage savings. Such technology is called *data deduplication*. In this project, an open source re-implementation of significant components of the deduplication system is one of the goals, the other is to hold a small scale study on it and draw some findings in observation. Deduplication is not just a storage utilisation, it is also a bandwidth saving strategy. Deduplication has proven a highly effective technology in eliminating redundancy in backup data [2]. However, the greatest challenge in Data Deduplication is the efficiency. Memory usage, time, data structure and parameter setting are those factors that affect the efficiency of the system. In regards of Data Deduplication – technological classification, a practical benefit has been contemplated. Accordingly, some of the classifications have been decided and some is left with more than one option to be carried in the experiment.

**1.1 Technological Classification**

1.1.1 *Fingerprint (Hash code)*

As deduplication is elimination process of repeating data. To detect the duplicated data, a fingerprinting process using hash function is carried out. Later, those fingerprints are to be compared and act as a determinant whether it is duplicated or not. There are 3 hash functions that is being observed in the implementation: *MD5, SHA-1, SHA-256.* The shorter the fingerprint is; the lower memory usage will be. However, the drawback of shorter fingerprint is the system will be more exposed to collision prone. As an example, SHA-256 (the longest out of three) will have a collision probability about 2\^-256. “For reference, this is 50 orders of magnitude less likely than an undetected, uncorrected ECC memory error on the most reliable hardware you can buy” [4]. A comparative study will be provided in the result section in order to determine what fingerprint shall be chosen.

1.1.2 *Granularity*

What data to be deduplicated? Essentially, there are 3 levels of granularity. *File-level, block-level, and byte-level*. The nature of dataset will significant affect the decision making and there is always a tradeoff in every single level, for example that *file-leve*l has the lowest overhead and significant limitations such as it has to be entirely identical. In the contrary, *byte-level* can easily eliminate the repeating data, however it is highly costly for both the computation and overhead. *Block-level* is the finest granularity that makes sense, most suitable for a general purpose storage system, and most balanced one. In this project, the implementation will be based on *block-level* granularity.

1.1.3 *Block size*

As I have agreed upon block-level granularity as the choice. The size of the block has to be defined. It leads to second stage of granularity on how big the block is. The bigger the block we have, the less chance we have to find duplicated block, because it might just differ by a bit or a byte. However, the smaller the block is, there will be more computation and overhead as in when the dividing process executed, more often write operation and bigger memory usage and/or space because the more fingerprints is stored in deduplication table (stored in RAM).

1. **Implementation Details**
   1. **How does deduplication work?**

In order to make an ease of use to the outside world, this Deduplication system provides interfaces that support *READ* and *WRITE*. The deduplication concentrates on Block-level deduplication. The deduplication happens synchronously, meaning it happens instantly during write.

The important element in this deduplication system is the *deduplication table* – *ddt* which is a hash table that contains necessary information of all the blocks. It contains *fingerprint* and *offset* (on which part of the file the block is written)*.* To maximise the performance of deduplication at its best, the system is using RAM as the place to store the *ddt*. However due to volatility of RAM, once system is turned off, it will automatically flush the *ddt* into disk leaving it in a safe state.

When file is written through *WRITE* interface, it will be processed and divided into blocks. Those blocks are then paired with their hash code that generated from their own content (fingerprinting process). The *fingerprint* will be then looked up in *ddt* whether it has existed or not, by using hash table the lookup process only takes O(1) complexity in average. If it has not existed beforehand, the system writes the content of the block into the *storage* (a single big file sitting on the hard disk, the place where we keep all the raw data) and insert a new row of that block in the *ddt.* The row consists of *fingerprint* and *offset*. On the other hand, if the block has existed instead of writing it on storage and inserting a new row in ddt, it only gets the offset from the *ddt* then carry it to next process of ‘write into read data structure *(record)’*. This next process is executed; regardless data is duplicated or not. The read data structure will be explained in details at section 2.2. Essentially, this data structure is a crucial element for *READ* interface.

When file is *READ*,the system is using the file-name (assuming the file name is unique) to look up read data structure then collect all the blocks needed and glue it up altogether to be a prominent single file.

* 1. **Methodology**

In this project I am using Iterative and Incremental methodology, the reason being is because there is an advantage that I can build small feature then learn of its performance and revise it if necessary, otherwise keep developing. Hence, it gives a much cleaner development state. In summary, for each iteration either modification is made or new functionality is added, or even both. This methodology is working very well for my progress; in fact, this might be the best choice. Since it allows me to do kind of “trial-and-error” approach, if it does not work as I want I can just fix it or change it in the next iteration. In general, there are 4 major iterations:

1. *Initial planning & linked list data structure*

First iteration was where the initial planning conducted. As a simple start, *READ* and *WRITE* interfaces were intended to only process a single file. ZFS Deduplication system was chosen as the project’s role model, due to high similarity in technological classification. Interesting finding on ZFS data structure of ddt, it is using *AVL tree* to store the *fingerprint* which is rather shocking to choose *AVL tree* with O(log n) complexity over a *hash table* with O(1) complexity. Due to unclear intention of ZFS’s data structure choice, I decided to experiment and use my own data structure. It starts with *linked list* data structure, where it was rather different compare to section 2.1. With linked list instead of *ddt* stores only *fingerprint* and *offset*, it stores *fingerprint* and *data*. Where *data* is a custom class that contains:

1. *offset* – where the block stored.

2. *length* – how big is the block.

3. *hash map* of (*filename* and *fingerprint*) – as a pointer to the next block or NULL if this is the last block.

|  |  |
| --- | --- |
| **Fingerprint** | **Data** |
| Hash1 | **Data{**  offset : 0  length : 512  hashMap  [“fileA”, Hash2]  [“fileB”, Hash2]  **}** |
| Hash2 | **Data{**  offset : 0  length : 512  hashMap  [“fileA”, Hash3]  [“fileB”, Null]  **}** |
| Hash3 | **Data{**  offset : 0  length : 512  hashMap  [“fileB”, Null]  **}** |

Hash 3

Hash 1

Hash 2

FileA

FileB

1. *Handle multiple files*
2. *Record data structure*

|  |  |
| --- | --- |
| ***Filename*** | ***Offset*** |
| *File A* | *{0, … , 30720}* |
| *File B* | *{0, 15360}* |

1. *Compression*
   1. **Challenges**

**Questions**

**1) What are the interfaces to the outside world?**  
We suggest something like,  
  
WRITE(file-name,file-content)  
READ(file-name)  
  
We can assume that the file names are unique!

Interface written in the code

**2) What is a \*part\* of a file?**

* Metadata\*
* File name
* File size
* File content
* File permission (for operation)\*

\**I believe this is irrelevant for this project*

**3) How will you determine whether two parts of a file are the same?  
Consider the space and time needed.**  
  
\*\*THIS IS PERHAPS THE MOST COMPLEX PART ALGORITHMICALLY\*\*

*The concept that I hold so far (most likely will change after I do some research):* Comparing the checksum of two parts. – With awareness that collision might happen. Although using SHA256 checksum function, the possibility is 2^-256.

**4) Once you have determined which parts of file are identical to existing  
file parts, how will you actually write this information (to disk)?**

*The concept that I hold so far (most likely will change after I do some research):*

To begin with, there will be a hash table or some sort of table that maps data checksum to storage location.

If an identical part is identified in the existing part. Instead of allocating new space on disk, the system just increments the reference count on the existing data.

If reference count of block A is 2, it means there are 2 files point to block A. Thus, there should be a “detail table” as a list of that relationship. Perhaps this relationship can be stored in an AVL tree?

**5) What will you do with a file part that is NOT the same as any previous  
file part?**

If there is zero reference that refers to the file part, meaning it is not being used at all, I believe the system should delete that particular part. **6) When receiving a READ request, what is involved in satisfying it.**

* To give a valid uncorrupted file to the requester.
* To give appropriate error if file is corrupted or file name does not exist.
  1. **Result**
  2. **Conclusion**

**References**

[1] Meyer, D.T. and Bolosky, W.J., 2012. A study of practical deduplication. *ACM Transactions on Storage (TOS)*, *7*(4), p.14.

[2] El-Shimi, A., Kalach, R., Kumar, A., Ottean, A., Li, J. and Sengupta, S., 2012. Primary Data Deduplication—Large Scale Study and System Design. In*Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)* (pp. 285-296).

[3] 'Understanding Data Deduplication - Druva' 2009, accessed May 30, 2016, from <http://www.druva.com/blog/understanding-data-deduplication/>.

[4] 'ZFS Deduplication (Jeff Bonwick's Blog)' 2016, accessed May 30, 2016, from <https://blogs.oracle.com/bonwick/entry/zfs\_dedup>.