**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* [5]*.* 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, and most suitable for a general purpose storage system, as well as the most balanced one [4]. Thus 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. **Deduplication with 2 interfaces.**

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* [Table 1] 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.

|  |  |
| --- | --- |
| **Fingerprint** | **Offset** |
| 9e8ccacdb12dff3… | 0 |
| 7f60418aaf0cc5a… | 15,360 |

Table 1 – Deduplication Table (Hash Map)

Each row represents a single block

When file is written through *WRITE* interface, it will be processed and divided into blocks [Fig. 1 (1)]. Those blocks are then paired with their hash code that generated from their own content (fingerprinting process) [Fig. 1 (2)]. 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 [Fig. 1 (3)]. 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 I keep all the raw data) and insert a new row of that block in the *ddt* [Fig. 1 (4a)].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)’* [Fig. 1 (4b)]. 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.

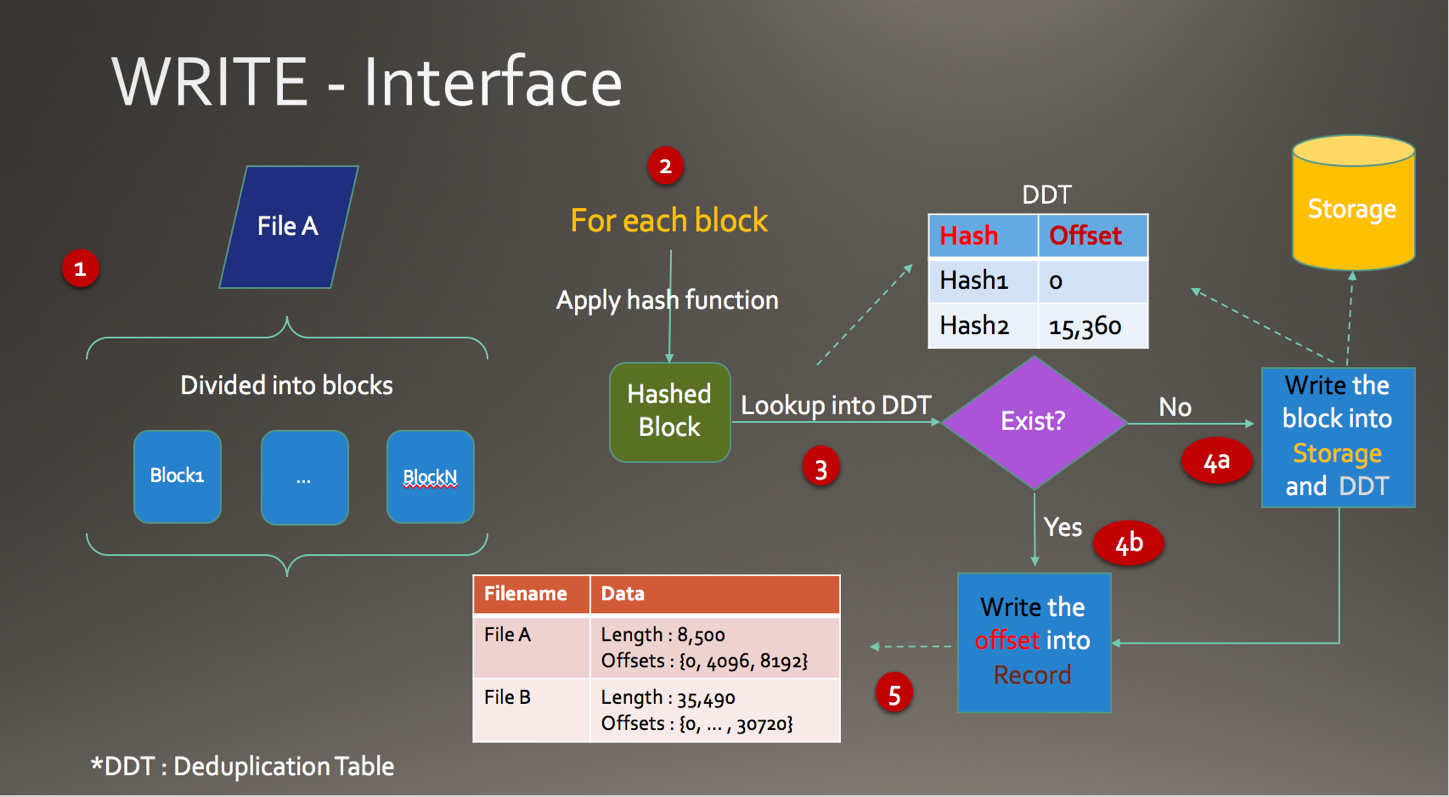


Fig. 1 – Flowchart Diagram on WRITE interface.

When file is *READ*,the system is using the file-name (assuming the file name is unique) to look up read data structure [Fig. 1 (5)] then collect all the blocks needed and glue it up altogether to be a prominent single file. For example, suppose a file named “FileA” with size of 8500bytes divided into 3 blocks {Block1,Block2,Block3} with offsets of {0,4096,8192} respectively. When read interface is executed to retrieve “FileA”, the system will refer to *record* and obtain the information of “FileA” then connect all the blocks [Fig. 2].

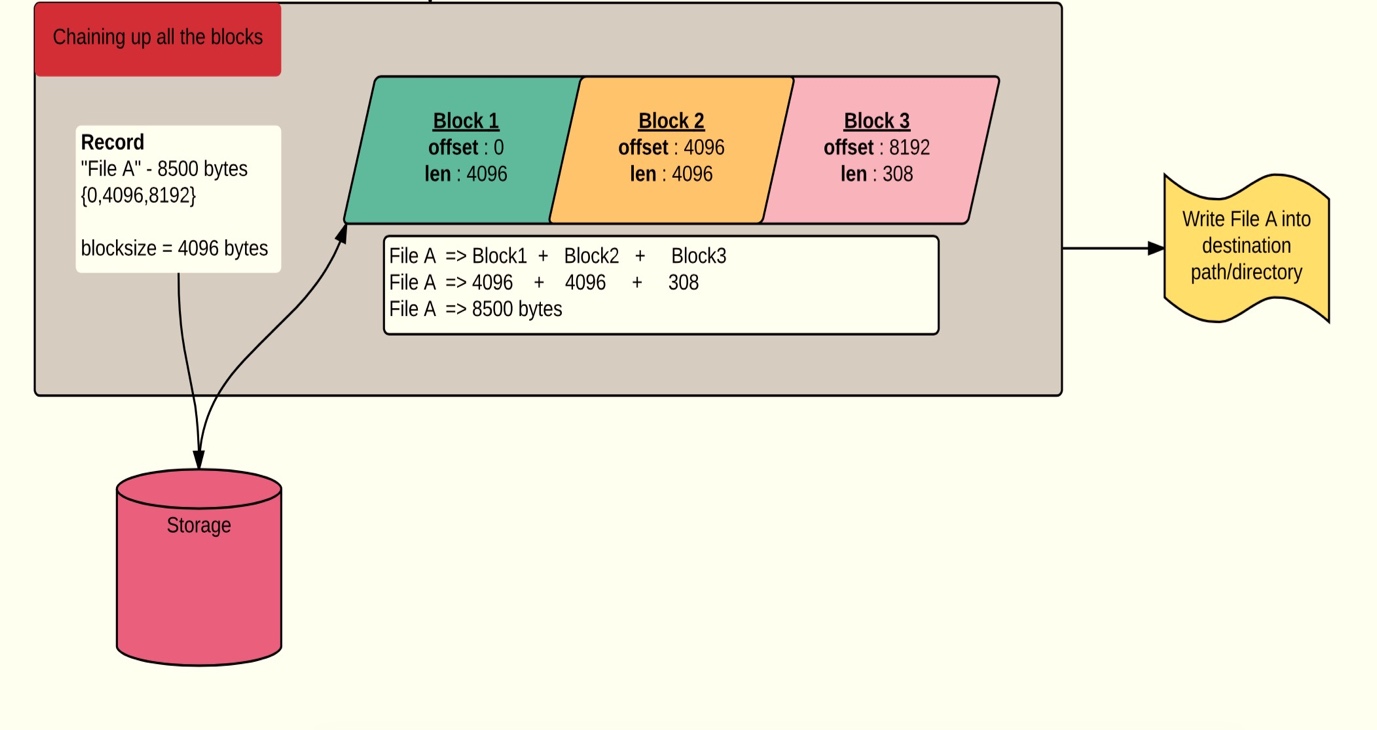


Fig 2. Chaining up all the blocks.

* 1. **Development and Methodology**

In this project I am using Iterative and Incremental methodology [Fig. 3], 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.

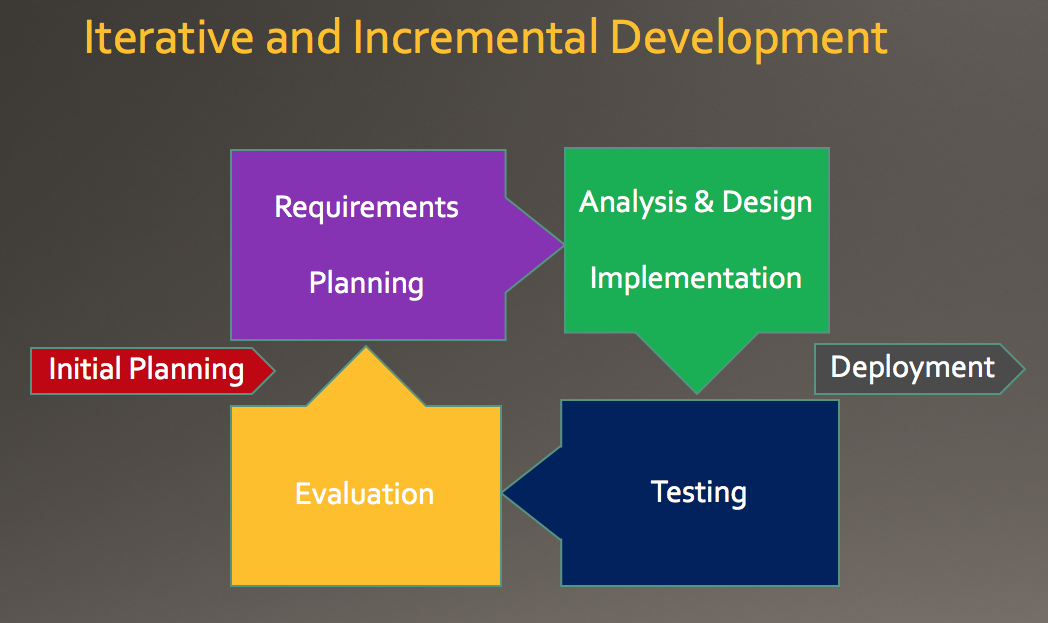


Fig. 3 – Methodology used[6].

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 be able only to do a single file process. 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* is that it is using *AVL tree* to store the *fingerprint.* It is rather odd 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 pretty different compare to section 2.1. By using 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 : 512 | length : 512  hashMap: [“FileA”, Null], [“FileB”, Hash3]  **}** |
| Hash3 | **Data{**  offset : 1024 | length : 512  hashMap: [“FileB”, Null]  **}** |

Table 2. DDT in linked list data structure.

As we can see a block will store all the links/pointers to other block, such a link path is defined uniquely by the filename. Basically, all it does is to follow the link path, glue it up together and when it hit null that will be the last piece of molding file process.

FileA

Hash 3

Null

Hash 1

Hash 2

FileB

Fig. 4 – Linked list data structure.

In terms of correctness this design is working just well and fine. However, there is a problem that failed to be foreseen due to simplicity of the first iteration goal – single file processing.

* + 1. *Handle multiple files*

As a deduplication system should be able to retrieve a whole repository or directory, in this iteration the interface is extended to be able to handle multiple files packaged in a folder. As I assume filename is unique and I use filename as identifier, the folder/repository/directory is regarded as root. Suppose “FolderA” has “FileA” ,“FileB” and “FolderB” with “FileBA” inside. The 3 files will be named uniquely using their path:

1. FolderA/FileA
2. FolderA/FileB
3. FolderA/FolderB/FileBA

Fundamentally, *WRITE* or *READ* interface execute the folder contents one by one. Thus, when we *WRITE* a folder/directory we need to keep the information of what are the contents of folder. In the implementation, it was named *folderDetails* a hash map that contains foldernameand list of its contents.

|  |  |
| --- | --- |
| Folder Name | Contents |
| FolderA/ | [“FileA”, ”FileB”, “FolderB/FileBA”] |
| FolderX/ | [“File1”, ”Folder1/Folder11/File111”, “Folder2/File21”] |

Table 3. Example of folderDetails.

The addition and development itself was straightforward, however an issue was found (related to linked list data structure). If we process more files and data, we will be storing many and many more pointers (represented by arrows). Back to the performance goal that *ddt* is to be storedin the RAM for the purpose of rapid access, this design will be not sustainable, not scalable and moreover not efficient at all. To put it in an example, suppose we have 1,000 blocks and 1,000 files. The max possibility of the pointers is up to 1,000,000 (1000x1000)and we store all of those in RAM, even though all we want is to retrieve a single file that consists of 4 blocks only, the RAM still maintain those 1,000,000 pointers. Thus, this linked list is not the right data structure.

* + 1. *Record data structure*

|  |  |
| --- | --- |
| **Filename** | **Data** |
| File A | Length : 8,500 Offsets : {0, 4096, 8192} |
| File B | Length : 35,490 Offsets : {0, … , 30720} |

Table 4 – Record data structure.

Record data structure is a revision from linked list data structure due to its inefficiency. The *ddt* structure will simply only consist of *fingerprint* and *offset* that represent one block for each row. The detailed discussion of record data structure can be found at section 2.1.

* + 1. *Compression*

In fourth iteration, compression feature was added. Compression is also an important element for deduplication. When it reduces the storage size, consequently it increases the deduplication ratio. The compression is executed whenever content of a block is written in the storage. In this project, compression algorithm used is the default one from java that is balanced on the speed and effectiveness. Some impacts from compression that worth noting are:

1. *Length* of block is no longer uniform in majority, because compression result size differs from one block to another.
2. *Fingerprints* remain the same as before, because compression only applies to block content before it is written to the storage.
3. The shorter the block size is, the higher possibility that compression makes it bigger in size instead.
   1. **Environment**

The implementation and test case is done by using:

* **IDE**: IntelliJ IDEA 2016.1.1 (Build #IC-145.597, built on March 29, 2016)
* **JRE**: 1.8.0\_40-release-b132 x86\_64
* **JVM**: OpenJDK 64-Bit Server VM by JetBrains s.r.o
* **Machine**: MacBook (Retina, 12-inch, Early 2015)
  + - **Processor**: 1.1 GHz Intel Core M
    - **Memory**:8 GB 1600 MHz DDR3
    - **Storage**: Flash Storage – SSD 256 GB
  1. **Challenges**
     1. Memory Efficiency

In deduplication, memory is one crucial point of the system. As a standard rule in the world, nothing is free and there is something to be given up in order to gain something. Memory is the one. Essentially, we sacrifice memory to achieve storage utilisation. Consequently, the challenge is how to sacrifice the memory as less as possible. To begin with, there is a need to list what is going to be stored in the memory. Those are:

1. *Deduplication Table (ddt)*

As our aim here is deduplication there is no compromise to not keep the ddt. Since ddt is the determinant key to decide whether data is duplicated or not and as a junction of what step should be taken. Obviously, ddt can be stored in the disk (as alternative when it exceeds the RAM size) but this feature is not in the scope due to time limitation, and after all I desire the quick computation as reading from RAM is much faster than from disk.

1. *Record and folderDetails*

As we have gone far in this discussion, it is all clear that *ddt* only play role in *WRITE* interface. *Record* and *folderDetails* get populated in *WRITE* and get used in *READ*. Same concept with the ddt, due to speed, these two instances will be stored in the RAM. One optimisation that can be done is instead storing all the *record* and *folderDetails*, I only store the relevant one at the time.

For example, in the system there are

**>** 100 directories with 10 files per directory.

**>** Meaning there are 100 folderDetails.

**>** 500 single files.

But all we want is to retrieve/*READ* one folder called “FolderA”. Thus, instead of having all

* + 100\*10+500 data in *records* and,
  + 100 data in *folderDetails* sitting in the RAM.

We only need to have “FolderA” information, that is

* 1\*10 data in records (all the files inside “FolderA”).
* 1 data in folderDetails (information of “FolderA” contents).

The question is how do we have a flexibility to remove/add those data from/into RAM. The answer: It is not that the unused/irrelevant data to *READ* process are flushed into disk, instead all the data are flushed. Only the relevant are fetched during *READ* and removed afterwards. There is also an instance that keep all of data address (offset in the disk) as an informant where to find them in a direct manner.

Together with Java implementation there are quite a few factors that affect the memory usage and its efficiency. There are 4 major factors:

1. Java Byte Wrapper

The *fingerprint* in *ddt* hash mapis merely an array of byte, as it is a hash code from one of the hash function **SHA-256 (32bytes)** or **SHA-1 (20bytes)** or **MD5 (16bytes)**. However, the implementation was not straightforward because in Java *byte[]* uses object identity for *equals* and *hashCode*[12] which means it does not compare its content and even if two object contain the same bytes it will be acknowledged as not equal by Java. Therefore, there has to be a Wrapper class to wrap the bytes and override the equality from “reference equality” to be “value equality”. Because of this wrapper, the memory usage is increasing. For example, in regards of fingerprint only (suppose we are using SHA-256) instead of having 32bytes cost per block, it now has additional 24bytes to store the Wrapper class as well.

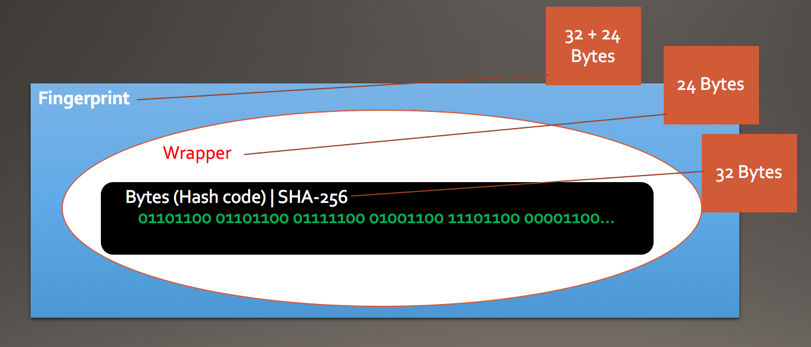


Fig. 5 – Wrapper is costing more memory.

1. Unexpected Int[]

As the observation begins there is a huge excess memory issue. Apparently there is int[] allocation that is unclear where it comes from. Some people also have the same problem and there seems no clear or certain answer to it (refer to [13] and [14]).

I am convinced to say that int[] is the major source of the excess as well as the one that keep increasing over time as ‘garbage’. Good thing is when Garbage Collector (GC) is called it will clean this garbage, however it is not recommended to call GC explicitly because it can affect the performance of the next GC - something to do with it's effectiveness as pointed out at the discussion in [15] and [16]. With that in mind, I cannot just free the memory allocation right away whenever its not being used anymore. It is suggested to let the GC clean it naturally. Consequently, monitoring the memory allocation is very difficult due to disturbance from int[] allocation.

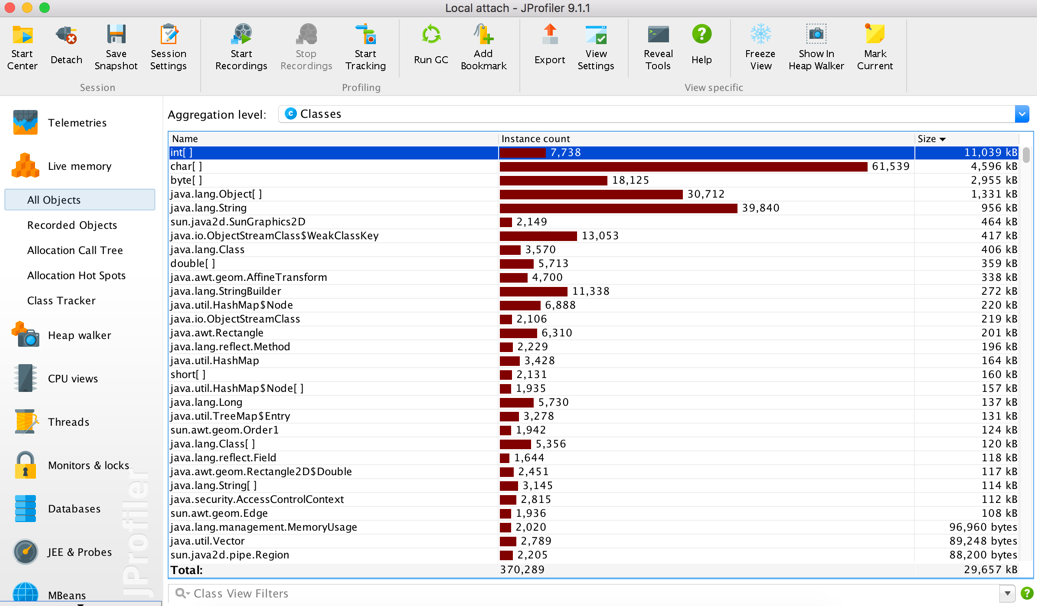


Fig. 6 – Int[] allocation is the highest right after the program started.

1. Custom Serialization
2. GNU Trove
   * 1. Verification
     2. Data Structure decision making
   1. **Test & 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>.

[5] 'MessageDigest (Java Platform SE 7 )' 2016, accessed May 31, 2016, from <http://docs.oracle.com/javase/7/docs/api/java/security/MessageDigest.html>.

[6] Larman, C. and Basili, V.R., 2003. Iterative and incremental development: A brief history. *Computer*, (6), pp.47-56.

[7] Shilane, P., Wallace, G., Huang, M. and Hsu, W., 2012, June. Delta Compressed and Deduplicated Storage Using Stream-Informed Locality. In*HotStorage*.

[8] Ma, J., Stones, R.J., Ma, Y., Wang, J., Ren, J., Wang, G. and Liu, X., 2016. Lazy Exact Deduplication.

[9] Meister, D., 2013. *Advanced data deduplication techniques and their application* (Doctoral dissertation, Universitätsbibliothek Mainz).

[10] Ahrens, M., 2014. OpenZFS: a Community of Open Source ZFS Developers.*AsiaBSDCon 2014*, p.27.

[11] Trove, G.N.U., High performance collections for Java.

[12]key, U 2016, 'Using a byte array as Map key', *Stackoverflow.com*, accessed June 1, 2016, from <http://stackoverflow.com/questions/1058149/using-a-byte-array-as-map-key>.

[13] int[], t 2016, 'track down allocations of int[]', *Stackoverflow.com*, accessed June 1, 2016, from <http://stackoverflow.com/questions/17145228/track-down-allocations-of-int>.

[14] hprof, O 2016, 'Objects with "no references" in hprof', *Stackoverflow.com*, accessed June 1, 2016, from <http://stackoverflow.com/questions/7304665/objects-with-no-references-in-hprof>.

[15] System.gc()?, W 2016, 'Why is it bad practice to call System.gc()?', *Stackoverflow.com*, accessed June 1, 2016, from <http://stackoverflow.com/questions/2414105/why-is-it-bad-practice-to-call-system-gc>.

[16] 'Bug ID: JDK-6668279 (spec) System.gc() should indicate that we don't recommend use and don't guarantee behaviour' 2016, accessed June 1, 2016, from <http://bugs.java.com/view\_bug.do?bug\_id=6668279>.