Toy DB Project Report
Disk Storage Simulator
Project No. 2

Rohit Kumar
120050028
rohit@cse.iitb.ac.in

Deependra Patel
120050032
pateldeependra06@gmail.com

November 25, 2014

1 Introduction

This report talks about a strategic simulation of a disk system. The simulation includes an efficient arm movement strategy for processing the requests on a disk system, providing reliability, high performance through implementation of the **RAID-5** technology and caching unit for faster access.

This disk-system is accessed during read and write requests randomly. The statistics related to various performance parameters are gathered and various conclusions and effect of the technologies used are displayed through graphs etc.

2 Structure of the disk-system

A main controller controls and executes all the functionalities provided.

The main controller provides a mapping of page number to disk addresses. It has an access to a cache controller which manages the disk cache and interfaces between the disk-system and the main controller. The cache controller checks if the data is present in the cache and returns it. If not present, it passes the request to a disk controller which simulates the disk-system. The disk controller passes this request to a request buffer which in turn processes these requests using the elevator algorithm.

While processing the requests various logs and data relating to measure the performance parameters etc. are also being stored.

The details of the various data structures and algorithms used in these classes are described in the below section.

3 Data Structures and Algorithm used for various classes

3.1 Main Controller (mainController.cpp & mainController.h)

The main controller stores a mapping of page numbers of a particular file to the disk address. For implementing this functionality, the map data structure provided by the STL of c++ has been used. Whenever a read request for a particular page in particular file is made by the paged file module to this, it checks in the map if an address corresponding to this page is present in the map. If present, a read request is made to the cache controller, otherwise if there is no such mapping, it denotes that the page wasn't written in the disk earlier and it is an invalid request. In case of the request is invalid, it is dropped.

Whenever a write request is made, it is checked that the mapping is already present in the map or not. In case it is present, it implies that the file is already present and is being overwritten. In case if it is not present (implying writing for the first time), a new mapping is assigned to the page randomly from the set of available free addresses in the disk and this mapping is put in the map.

There might be a case when the file is being deleted from the disk. In such a case the mapping is no longer necessary and it is deleted from the map.

This main controller passes the requests to the cache controller for performing the operations.

3.2 Cache Controller (cacheController.cpp & cacheController.h)

Cache for disks is implemented in this class. We have implemented fully associative cache as miss penalty is huge for reading and writing data to the disks. On receiving the diskAddress to read and write from mainController, it checks whether the page is already present in the cache. If already found in cache, Page data is returned from there. If requested page is not found in cache, then new page is allocated to it and that page is brought from disk using diskController.

For page allocation, if any page is free that is allocated. If no free page found then page which is not dirty is allocated, if no such found then Least Recently Used(LRU) page is allocated after writing its contents to the disk.

An unordered map is used to keep mapping of disk address to corresponding page number in cache. Two boolean arrays of free pages and dirty pages is also kept. Whenever a page is written in cache, dirty flag is marked true. If any page marked dirty is to be removed from cache then its data has to be written back to disk. A linked list of pages is kept to store the allocated pages ordered by time. LRU is implemented on these pages by popping out first element on page allocation and pushing back the newly allocated page.

3.3 Disk-System Controller (diskController.cpp & diskController.h)

Disk Controller handles all the requests(read/write) made to the disk. The input disk address is first translated into corresponding disk and CHS(Cylinder Head Sector) format. As we are using RAID5, parity disk for that sector is also calculated. For calculating disk number, we are using Right Asynchronous RAID5 type as given below:

3.3.1 Disk Number and CHS conversion

In case of a RAID-5 system, the CHS conversion is complicated to normal conversion. In this scenario we have to take care of the parity block. The data block may need to be shifted in the next block to accommodate the parity block.

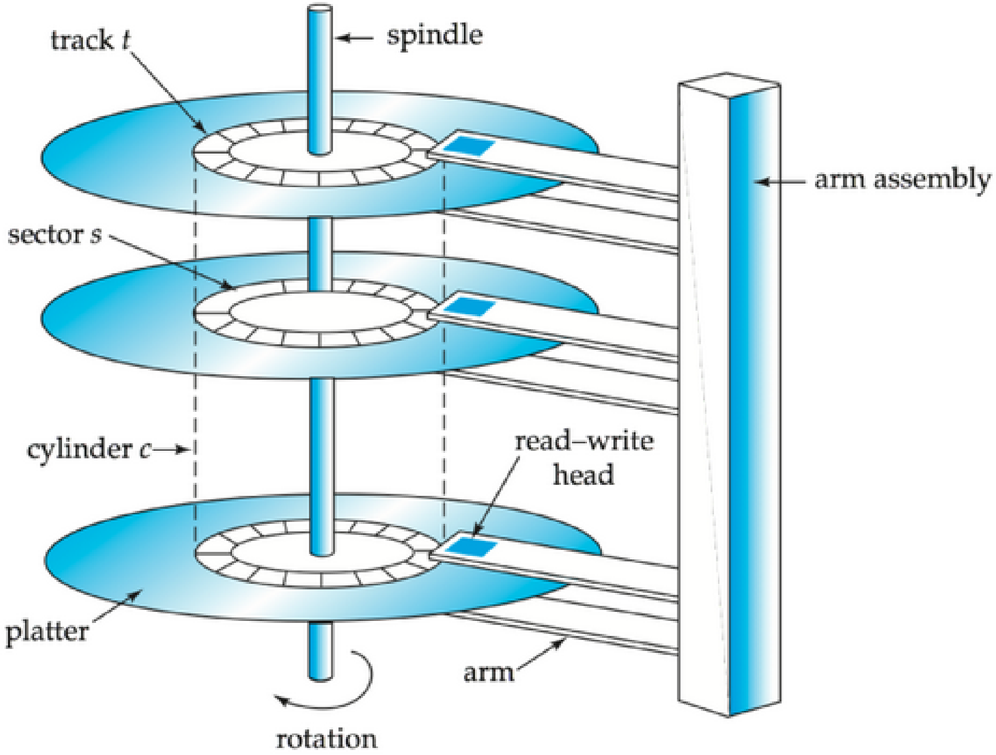


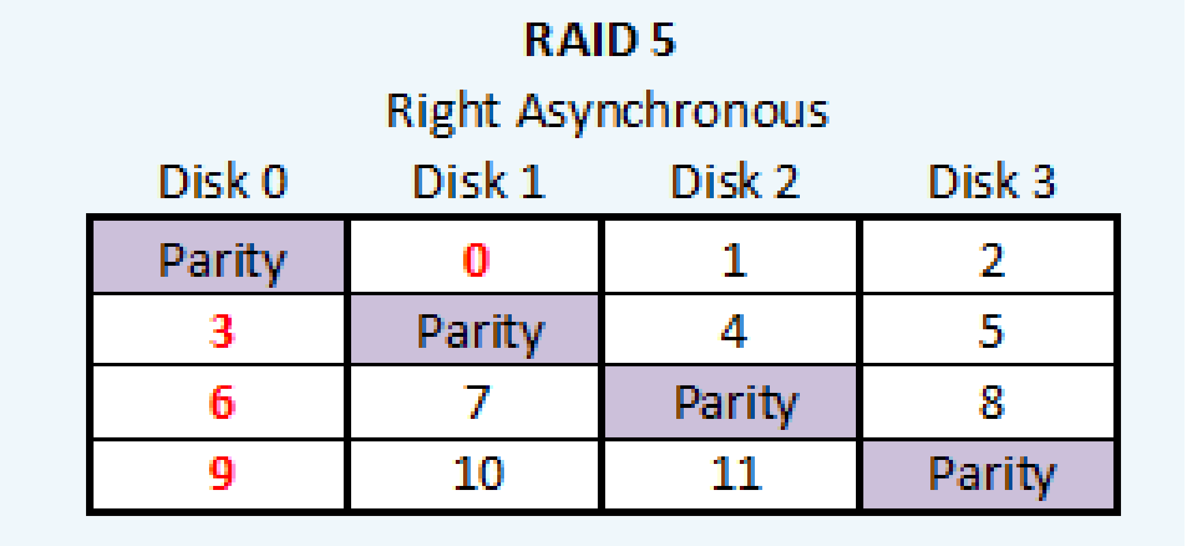
Figure 1: Design of the RAID-5 system and distribution of parity blocks

Let there be D disks are used in RAID5 (D+1 in total), each with each with N cylinders, M tracks/cylinder, and K sectors per track. Then the disk address of this sector can be calculated as :

- d is the disk number. d belongs to [0, 1, 2, D]
- c is the cylinder number. c belongs to [0, 1, 2, N - 1]
- t is the head/track number. h belongs to [0, 1, 2, M - 1]
- s is the sector number. s belongs to [0, 1, 2, K - 1]
- p is parity disk corresponding to a CHS
- a is address in that disk

So, given the disk address, c,h and s can be calculated as:

- a = DiskAddress/D;
- p = a%(D+1)
- If p is less than or equal to DiskAddress%D then d = DiskAddress%D
  else, d = DiskAddress%D + 1



RAID 5

Right Asynchronous

Disk 0	Disk 1	Disk 2	Disk 3
Parity	0	1	2
3	Parity	4	5
6	7	Parity	8
9	10	11	Parity

Figure 2: Design of the RAID-5 system and distribution of parity blocks

- c = a/(M*K)
- t = [a/K] modulo M
- s = a modulo K

An array of all disk objects are created and read/write operations are done on them.

1. **Read Sector** - Sector is read from the corresponding disk and returned back.
2. **Write Sector** - Both the requested and parity blocks are first read. XOR of bits of old data, the new data and the parity block is written back to parity block. New data is written to the corresponding sector. Total 2 reads and 2 writes are performed.

After any operation, one pending request from buffer of each disk is executed.

3.4 Request Buffer and Elevator algorithm (diskRequestHandler.cpp & diskRequestHandler.h)

This contains a class which places the requests into a request buffer and processes it using the elevator algorithm. A data-structure for every request is created which also stores the time-stamp of the request.

These requests are stored in the set data structure provided by the STL of C++. This is basically a balanced binary tree structure and serves operation like search, insert and delete in time logarithmic to the size of the set i.e. the number of requests buffered in the set.

Upon serving the requests for a particular cylinder, the request structure is deleted and the arm will now move to the next closest request in the direction of movement. This request will be obtained by iterating over the tree data structure. The first element of a tree is always the leftmost one. The next element after an element is the first element of its right subtree. If the element does not have a right child, the next element is the element's first right ancestor.

If the element has neither a right child nor a right ancestor, it is the rightmost element and it's last in iteration.

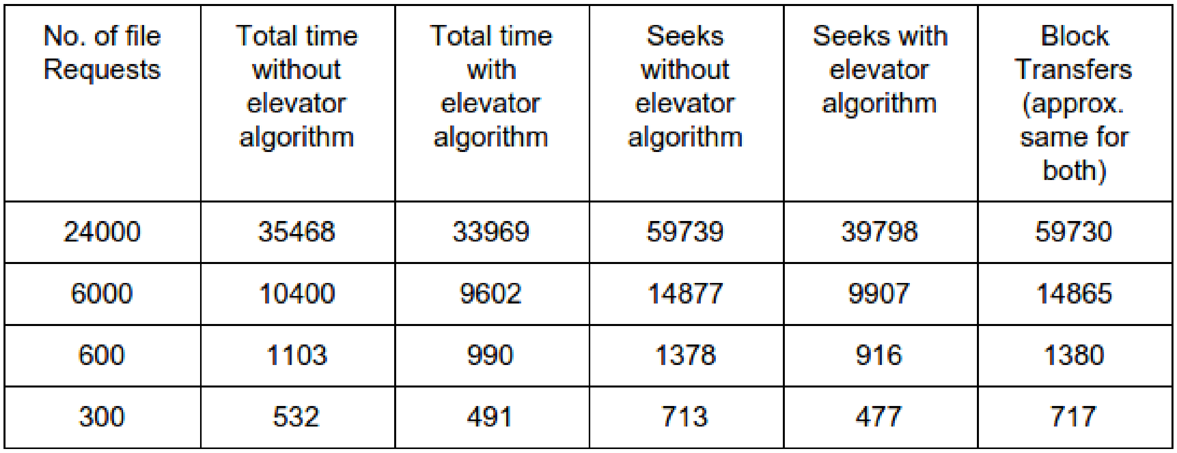
In this manner the requests will be served until the start/end of the tree has been reached (depending on the direction of the movement). When the start/end has been reached, the direction of iteration will be reversed.

While the traversal is being done, various statistics are being gathered like the number of times the cylinder switch is happening (i.e. the arm movement), the rotational latency incurred and the track switch. Printing the requests in the buffer is also supported which shows how the requests are being processed and queued up.

4 Analysis of data and statistics (main.cpp & testData.py)

For computing the statistics data, first all the file pages are written. Then requests are randomly generated which access disks to read or write data. Global flags cacheEffect and elevatorEffect are kept which can be changed true/false and all classes works correspondingly.

4.1 Effect of elevator algorithm



No. of file Requests	Total time without elevator algorithm	Total time with elevator algorithm	Seeks without elevator algorithm	Seeks with elevator algorithm	Block Transfers (approx. same for both)
24000	35468	33969	59739	39798	59730
6000	10400	9602	14877	9907	14865
600	1103	990	1378	916	1380
300	532	491	713	477	717

Figure 3: Statistics of time taken, seeks and block transfers by varying no. request

The plot of number of seeks required vs no. of requests are plotted in the below graph. The blue plot describes seeks with the elevator algorithm and the red one without elevator.

From the above graph, we can see that for a given number of requests, Number of seeks without elevator algorithm is greater than that without it. In elevator algorithm, requests are sorted in order of cylinder number.Therefore requests on same cylinder can be completed without making any seek. Thus extra seek time is avoided using elevator algorithm.

From the graph given below, we can see for a given number of requests total time taken with elevator algorithm is always less than that without it.

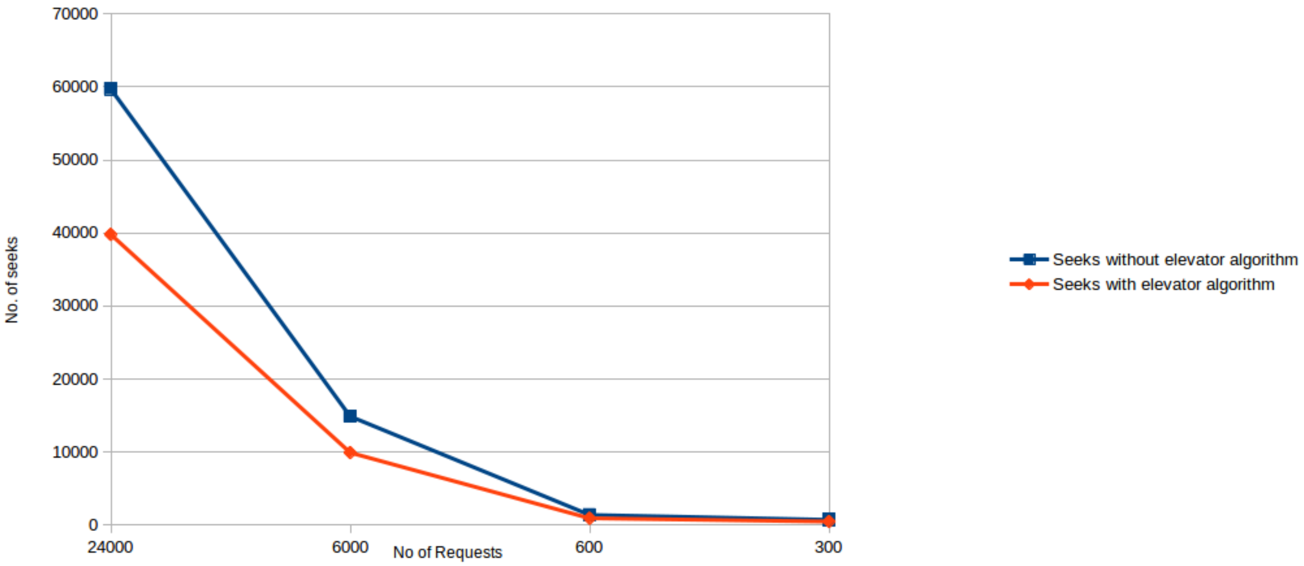


Figure 4: Seeks vs no. of requests

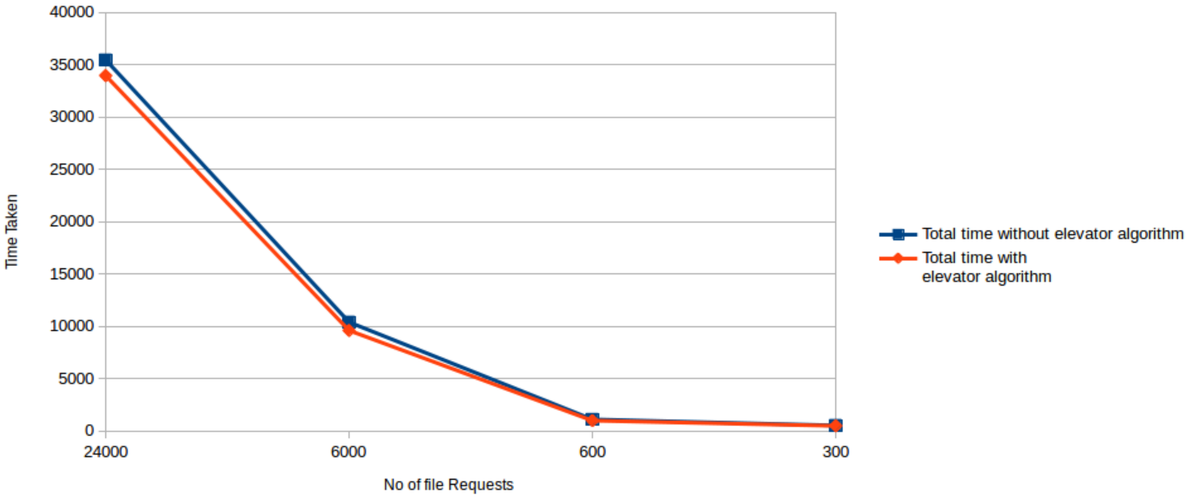


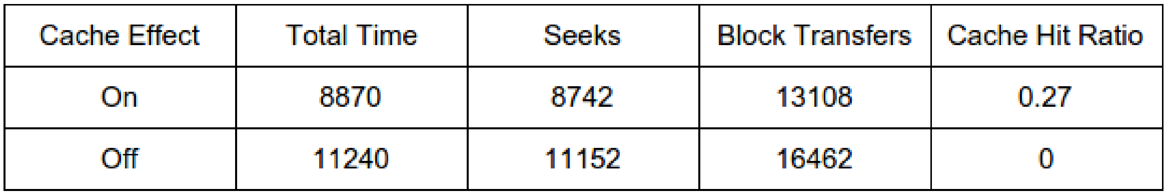
Figure 5: Time taken vs. no. of requests

4.2 Effect of caching on the disk

On switching on the cache, seeks and block transfers decreases eventually decreasing the total time. Disk addresses are accessed randomly therefore cache hit ratio is not much higher.

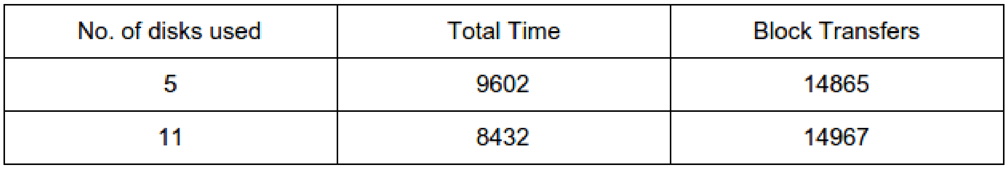
4.3 Effect of increasing the no. of disks, while keeping the total space constant

For same number of total disk size, we increased the no. of disks and found that as we increase raid level total time taken for same requests is decreased. This is due to dispersion



Cache Effect	Total Time	Seeks	Block Transfers	Cache Hit Ratio
On	8870	8742	13108	0.27
Off	11240	11152	16462	0

Figure 6: Effect of cache on no. of seeks, block transfers, total time and cache hit ratio

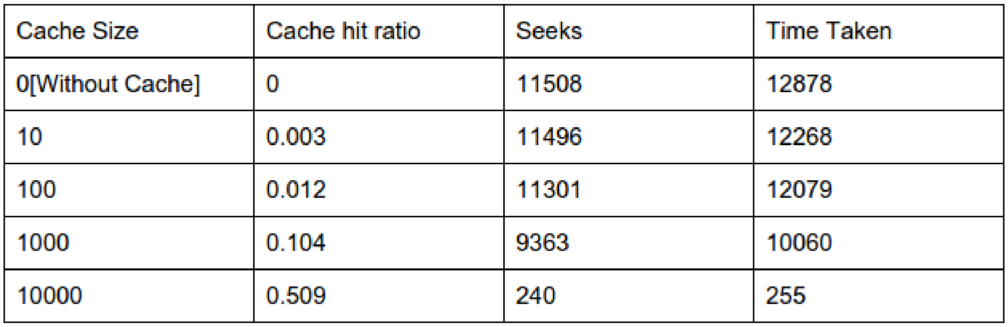


No. of disks used	Total Time	Block Transfers
5	9602	14865
11	8432	14967

Figure 7: Effect of increasing the no. of disks while keeping the total

of requests among many disks.

4.4 Effect of increasing the cache size on no. of seeks and time taken



Cache Size	Cache hit ratio	Seeks	Time Taken
0[Without Cache]	0	11508	12878
10	0.003	11496	12268
100	0.012	11301	12079
1000	0.104	9363	10060
10000	0.509	240	255

Figure 8: Effect of increasing the no. of disks while keeping the total

Graphs for the above table are

We can see that the cache hit ratio(#cache-hits/#total-accesses) decreases as cache size increases. More pages can be stored increasing number of cache hits.

As the cache size increases, the time taken to execute reduces drastically after some time. This is because some pages are already cached which need not be written back or read from the disk. As size of cache increases more such pages can be cached.

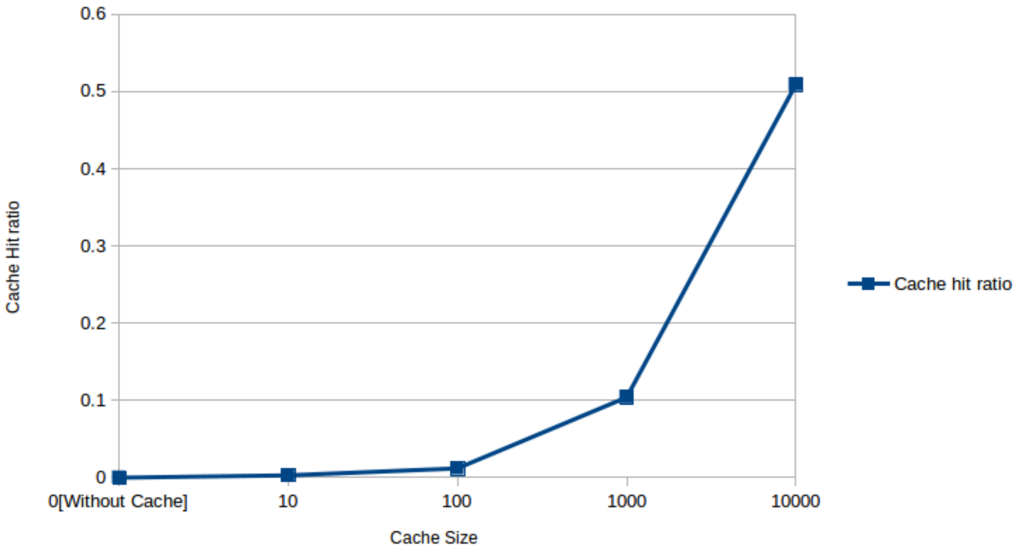


Figure 9: Effect on cache hit rate by increasing the cache size

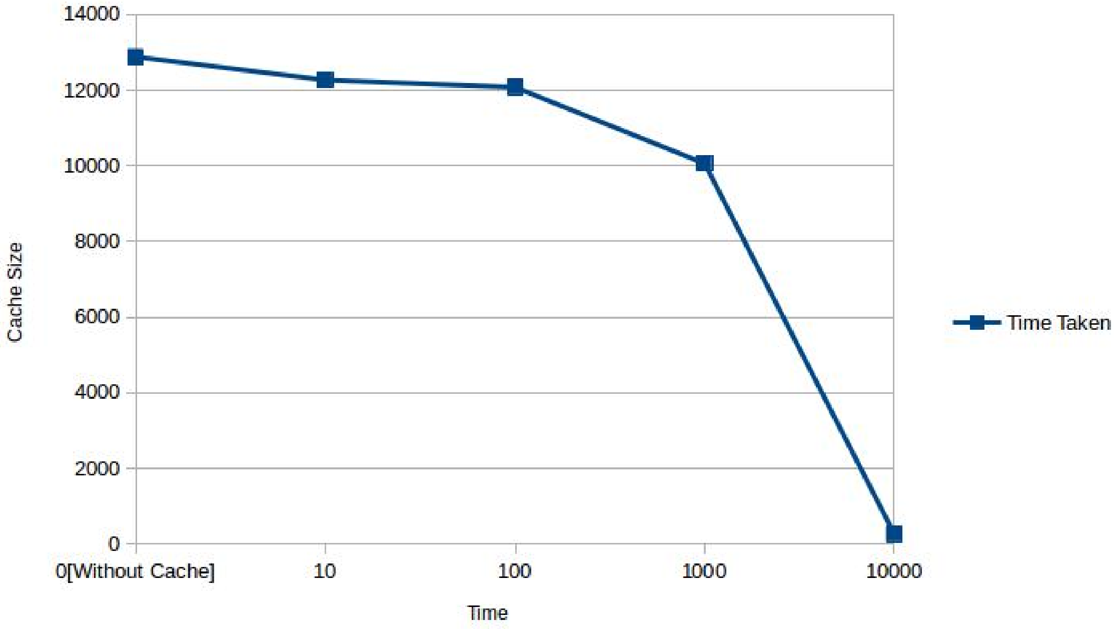


Figure 10: Effect on total time by increasing the cache size