Implementing A Tag Storage System Designed for Fast Retrieval

Rachel Shearer
6.033 Design Project 1

22 March 2007
Lewis Girod TR 1
David Schultz F 2

1. INTRODUCTION

In a storage system where the amount of data exceeds the size of the computer memory, it is necessary to store the data on disk rather than directly in memory. However, the time it takes to access and read from a disk is far greater than the time it takes to read from memory, and thus developing a way to represent the data on disk such that it can be efficiently accessed and updated is desirable.

This paper presents a design for the storage of tag triplets that optimizes for fast retrieval. Two key design choices permit the proposed tag storage system to achieve these goals:

1. A triplet is written to a particular block on the disk according to a hash function. Treating the blocks on disk as indices in a hash table provides constant-time triplet retrieval.
2. During tag retrieval, blocks are read into memory successively rather than randomly in order to take advantage of the speed of sequential disk access.

2. DESIGN DESCRIPTION

The proposed tag storage system stores all triplets on the disk, writing to and from main memory as necessary during insertion, retrieval, and deletion of tags from the system. This section describes the storage configuration and the implementation of the storage system interface.

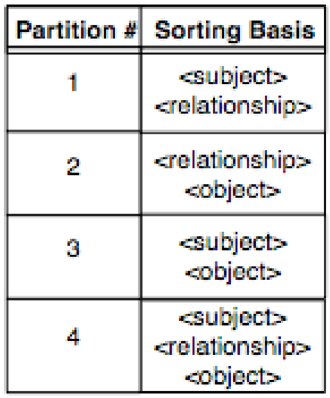
2.1 Storage Design

A triplet consists of three string tags in the form <*subject, relationship, object*>. The user must specify all three tags when inserting and removing triplets, but a user may use wildcard strings to retrieve all triplets that match just one or two given tags. Wildcard searches present a particular challenge for any storage scheme that uniformly organizes the triplets because any combination of tags can be specified for a wildcard search. Thus, a design that optimizes for retrieval must sort the triples in various ways.

The following sections explain how the triplets are organized on disk and the structure of main memory.

2.1.1 The Hard Disk

The proposed design takes advantage of the large amount of disk space by partitioning the disk into four parts where triplets are sorted by different combinations of their elements. As presented in Table 1, triplets are sorted according to their *subject* and *relationship* elements in the first partition, *relationship* and *object* elements in the second, *subject* and *object* in the third, and all three elements in the fourth partition.



Partition #	Sorting Basis
1	<subject> <relationship>
2	<relationship> <object>
3	<subject> <object>
4	<subject> <relationship> <object>

Table 1: This table presents the four different ways each triplet is sorted. In each partition, a triplet is hashed according to a different combination of its three elements.

Figure 1 presents the structure of the disk, including the four partitions and an extra 32 MB of disk space used to store data from memory during a shutdown. Each partition is 250 GB, divided into 62,500,000 4 KB blocks. Each block can store approximately 40 triplets of 100 bytes each.

Each partition on the disk is treated as an open-addressed hash table, with each block as an index. A triplet is placed in a block within a partition according to the hash value of the appropriate combination of elements of that triple. For example, in the first partition, a triplet is written to the block that corresponds to the hash value of the *subject* element combined with the *relationship* element. In the fourth partition, a triplet is placed according to the hash value of the combined *subject*, *relationship*, and *object* tags.

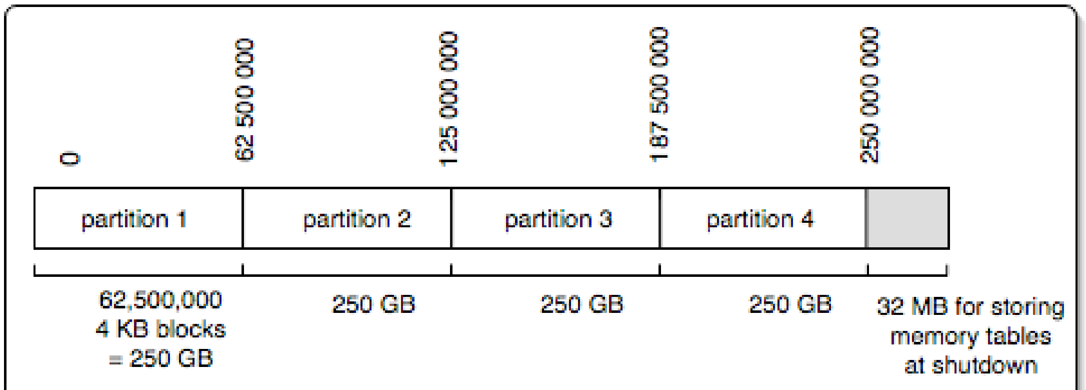


Figure 1: Each partition on the disk consists of 62,500,000 blocks, each 4 KB in size. The last 32 MB of disk is used when writing the state of the memory to disk at shutdown.

In an open-addressed hash table, collisions are resolved by probing the table; in other words, if two elements hash to the same index, an alternate location is found for one of the elements by searching the table. The proposed design utilizes a linear probing method to find a new location for triplets. As presented in Figure 2, once a block has been filled, each successive block in the partition is checked for free space: the triplet will be placed in the first free block found.

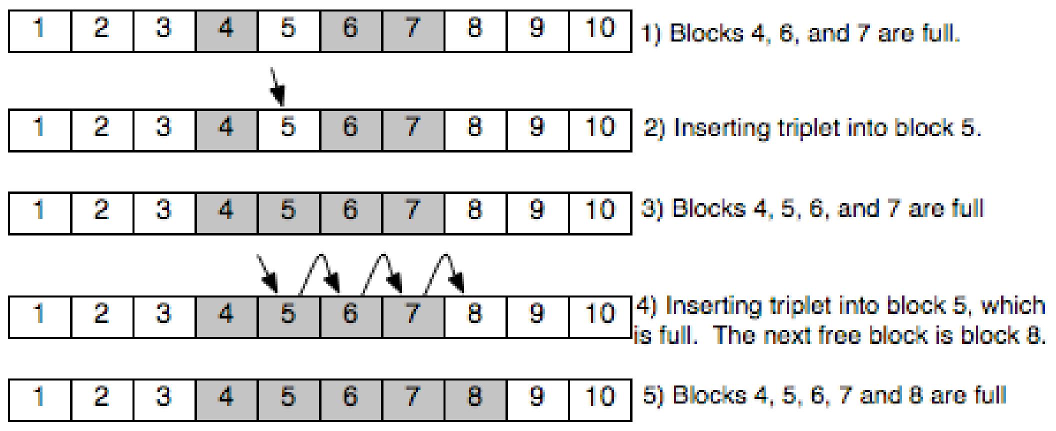

1 2 3 4 5 6 7 8 9 10
1) Blocks 4, 6, and 7 are full.
2) Inserting triplet into block 5.
3) Blocks 4, 5, 6, and 7 are full
4) Inserting triplet into block 5, which is full. The next free block is block 8.
5) Blocks 4, 5, 6, 7 and 8 are full

Figure 2: A simple linear probing example. The first insert does not involve any probes because block 5 has free space. But the second insert into block 5 involves looking at each block after block 5 in succession until one is found that has free space.

2.1.2 Main Memory

Due to its limited size, the main memory is not used to store data. Instead, the memory contains a Free Map (presented in Figure 3), which records which blocks on the disk have free space and which blocks are filled with triplets. Every time a block becomes full by the insertion of a triplet, the Free Map is changed to indicate that future triplets may not be placed in that particular block. Similarly, every time a triplet is removed from a full block, the Free Map is changed to indicate that triplets may once again be placed in that block. Main memory also contains pointers to the first block of every partition.

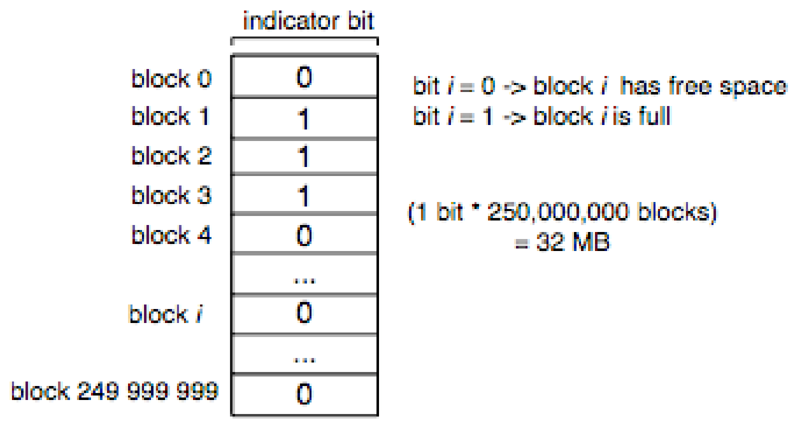

indicator bit
block 0
block 1
block 2
block 3
block 4
block *i*
block 249 999 999
0
1
1
1
0
...
0
...
0
bit *i* = 0 -> block *i* has free space
bit *i* = 1 -> block *i* is full
(1 bit * 250,000,000 blocks)
= 32 MB

Figure 3: The Free Map indicates which blocks are full. When bit *i* = 0, then block *i* has free space. When bit *i* = 1, then block *i* cannot hold any more triplets.

The Free Map uses 1 bit to represent each block on disk; therefore, a total of 32 MB is used to represent all 250,000,000 blocks. The pointers to the partitions take up negligible space in memory. The Free Map and the partition pointers are both written to disk during shutdown, and they are both read from disk during system startup.

2.2 Storage System Interface

Applications interact with the proposed tag storage system via *insert*, *remove*, *find*, and *shutdown* requests. The following sections describe the process for each request, as well as the process for when the system is idle.

2.2.1 insert(subject, relationship, object)

The *insert* request adds a new triple to the tag storage system.

1. Check the Free Map. If there are no unfilled blocks in the first partition (the first 62,500,000 blocks), return an error.

2. Calculate the hash value for each of the four different combinations of the three String elements. Each hash value corresponds to a block in a partition.
3. Check the Free Map for the first block (which corresponds to the first hash value).
    a. If the block has free space, continue to the next step.
    b. If the block does not have free space, check each successive block in the Free Map until a block with free space within the correct partition is found. The triplet will be placed in this block instead. Continue to the next step.
4. Read the block into memory and insert the triplet into the block. Write the block back into its proper place on disk.
5. Repeat Steps 2 and 3 for each of the other three blocks. The triplet will be copied onto the disk four times: once for each partition.
    a. If inserting a triplet into a block caused that block to become full, update the Free Map to reflect this.

2.2.2 remove(subject, relationship, object)

The *remove* request removes a triple to the tag storage system.

1. Calculate the hash value for each of the four different combinations of the three String elements. Each hash value corresponds to a block in a partition.
2. Check the Free Map for the first block (which corresponds to the first hash value).
    a. If the block has free space, read it into memory.
    b. If the original block does not have free space, check each successive block in the Free Map until a block with free space within the correct partition is found. Read the original block and each successive block (up until the block with free space) into memory.
3. Scan through the triplets contained in each of the blocks in memory for the triplet to be removed.
    a. If the matching triplet is found, remove it from the block. Write the block back into its proper spot on disk.
    b. If the matching triplet is not found, locate the next block with free space within the next partition, and read in all intervening blocks. If the end of the partition is

reached, start reading from the beginning. If the original block is reached again without finding a matching triplet, return an error.

4. Repeat Steps 2 and 3 for each of the other three blocks.
5. If any of the triplet removals was from a full block, update the Free Map to show that the block is no longer full.

2.2.3 find(subject, relationship, object, start, count)

The *find* request retrieves *count* triplets that match the specified *subject*, *object*, and *relationship*, beginning with triplet number *start*. Each field may optionally be a wildcard string that matches any string.

1. Calculate the hash value for the combination of the specified String elements, disregarding wildcard Strings. Use as many non-wildcard Strings as possible in the combination.
2. Match the combination of String elements used in the hash value calculation to the appropriate partition.
   a. If only one String element was provided, pick any partition.
3. The calculated hash value corresponds to a block in the appropriate partition. Use the Free Map to check if the block has been filled.
   a. If the block has free space, read it into memory.
   b. If the original block does not have free space, check each successive block in the Free Map until a block with free space within the correct partition is found. Read the original block and each successive block (up until the block with free space) into memory.
4. Scan through the triplets contained in each of the blocks in memory, keeping track of the triplets that match the specified String elements.
   a. If the specified number of matching triplets is found, return them to the application.
   b. Otherwise, read in more blocks successively up until the next block with free space. If the end of the partition is reached, start reading from the beginning. If the original block is reached again, return any matching triplets found to the application.

   c. If the specified *start* integer was greater than 0, only keep track of the triplets once *start* matching triplets have been found.
5. Repeat Step 4 until the specified number of matching triplets has been found.

2.2.4 shutdown()

During a system shutdown, all memory tables are written to disk, as depicted in Figure 1. Upon startup, the tables are read from the disk back into main memory. In the event of a system crash, the tables read from the disk will not necessarily have the correct information regarding unfilled blocks.

2.2.5 Idle Time

When the system is idle and the number of requests is low, the data stored on disk is reorganized. Starting with block 0, as many blocks as possible are read into memory. Each triplet on each block is re-hashed, and the hash value is compared to the current location of the triplet. If the triplet is not in the correct block – and the Free Map indicates that the correct block has space – the triplet is removed from its current block and written to the correct block. Otherwise, the triplet will be written to any block with free space that lies between the triplet's current block and the correct block according to linear probing.

Reorganizing the data on disk during idle time makes sure that triplets are always placed in or as near as possible to their hashed blocks.

3. DESIGN ANALYSIS

The following sections explain the reasoning behind the disk and memory structure of the proposed tag storage system design when optimizing for retrieval. In addition, an analysis is provided of the system's performance on two different workloads.

3.1 Open-Addressed Hashing

There are two main methods for resolving collisions in hash tables: chaining and probing. A chained hash table places the elements in linked lists extending from each table index, whereas open-addressed hash tables use probing to inspect table indices until an unoccupied one is found

[1]. In the proposed design, the disk is structured as an open-addressed hash table that utilizes probing rather than a chained hash table because disk partitions have a fixed size and the number of blocks in a partition cannot be increased.

Storing a triplet according to a hash on its elements is an optimization for search: rather than sequentially reading through all the blocks on the disk while looking for a certain triplet, the system can focus on a specific block where the triplet should be. However, open-addressed hash tables degrade in performance as the table fills – the more similar triplets are stored on disk, the farther away each is stored from its original position, and the more time it will take to *insert*, *remove*, and *find* triplets. Because each triplet is repeated four times on disk in an optimization for wildcard searches, the disk fills much faster and degrade in performance much faster than if each triplet were only inserted once.

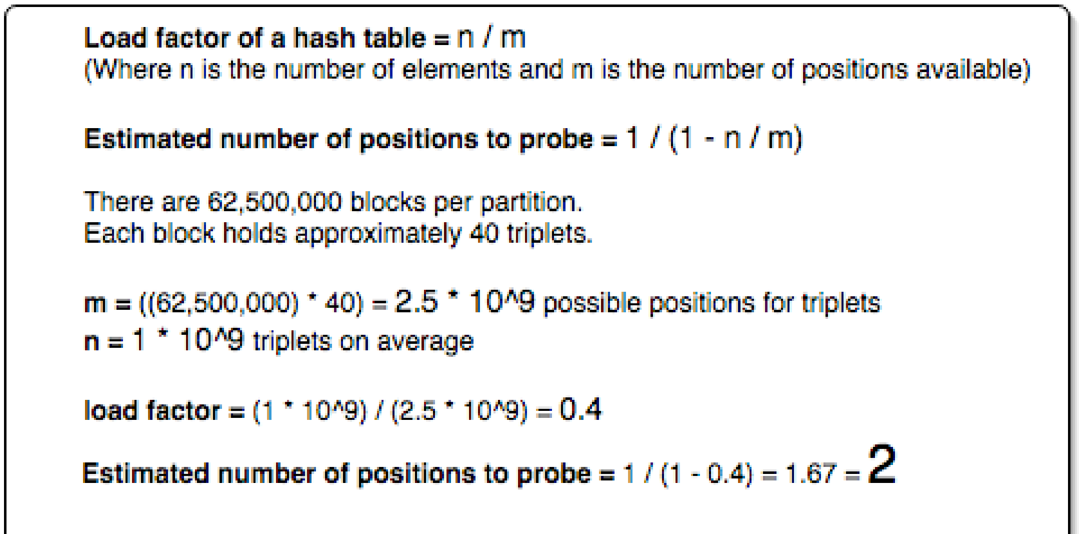
3.1.1 Linear Probing

Although there are more likely to be collisions as the database grows in size and the number of triplets with similar elements increases, the linear probing method of collision resolution guarantees that a triplet is located in a block successive to its original hash, easily accessible by a fast sequential disk read. Section 3.1.3 illustrates in more detail how sequential disk reads are used in conjunction with fast memory access time to optimize for fast triplet retrieval.

Ideally, triplets would be hashed according to every combination of elements rather than just four. A partition with triplets hashed on just their *subject* element, for example, would be helpful when a search is conducted for all triplets with a particular *subject*. As the proposed design stands, that kind of search requires a sequential read of the entire contents of a partition and receives none of the benefits that hashing provides for fast retrieval. However, given that the performance of an open-addressed hash table degrades as the table fills, repeating each triplet seven times on the disk would result in slow searches and a limited number of inserts before the disk fills completely. Therefore, the design sacrifices better performance on one search case for better average performance on all search cases.

3.1.2 Expected Number of Probes

Given the number of triplets on disk, one can estimate the number of probes it will take to reach any given triplet. As shown in Figure 4, the proposed design will require approximately 2 probes to reach a triplet if the database is about 100 GB.



Load factor of a hash table = n / m
(Where n is the number of elements and m is the number of positions available)

Estimated number of positions to probe = 1 / (1 - n / m)

There are 62,500,000 blocks per partition.
Each block holds approximately 40 triplets.

m = ((62,500,000) * 40) = 2.5 * 10^9 possible positions for triplets
n = 1 * 10^9 triplets on average

load factor = (1 * 10^9) / (2.5 * 10^9) = 0.4

Estimated number of positions to probe = 1 / (1 - 0.4) = 1.67 = 2

Figure 4: A calculation for the expected number of positions necessary to probe before inserting or retrieving a particular element.

It is important to note that the preceding formula assumes that the triplets are distributed uniformly throughout a partition. As long as the hash function can produce unique hash values for each triplet, uniform hashing is guaranteed – but as the number of triplets grows, it is more difficult for a hash function to produce unique hash values and for each triplet to be assigned to a unique block. Furthermore, there are certain to be collisions in the partitions where triplets are hashed on two elements rather than three because triplets are likely to have similar elements. Therefore, the estimate of 2 probes before any given triplet is reached may not hold for the majority of the partitions.

3.1.3 Disk Seeks

Non-uniform distribution of triplets is more likely when using linear probing [1]. Long sequences of full blocks will form when there are triplets with similar elements, causing probes to extend farther and farther away from the original block. A different method of collision resolution would produce a more uniform distribution. However, linear probing has the unique advantage that each of the blocks in a probe can be read into memory successively during one disk seek, which is far more efficient than reading in widely separated blocks with multiple disk seeks. In this instance, the time saved by reading from the disk sequentially allows the proposed system to optimize for fast retrieval even when triplets have similar elements. When the blocks are in memory, the correct triplets can be found quickly: memory operations are much faster in comparison with the time to access disk.

3.3 Workload Performance Analysis

This section presents an analysis of two workloads: Flickr++ and Library. Given that the vast majority of time in both workloads is spent conducting searches, the proposed triplet storage system performs similarly with both of the workloads. In either case, the proposed system takes the most time to process *inserts* and *removes*. In the following analysis, it is assumed that each system has a 100 GB database, which contains about one billion triplets. Thus, the expected number of positions to probe when retrieving a given triplet is at least 2.

3.3.1 Flickr++ Application

The analysis of the Flickr++ Application is presented in Figure 5.

	Request Specifics	Estimated Processing Time for Request
5%	Adding or deleting an image - at least three *inserts/removes* required for each command.	**Estimated time for adding =** 284.44 ms **Estimated time for deleting =** at least 292.2 ms (depending on distribution of triplets)
5%	Tagging an image using *insert.*	**Estimated time =** 96.48 ms
75%	Looking up images that have certain *relationship* and *object* tags.	**Estimated time =** 15.75 ms (for 0 to 1000 images meeting that criteria)
15%	Looking up images that have certain *subject* and *relationship* tags.	**Estimated time =** 15.75 ms (for 0 to 1000 images meeting that criteria)

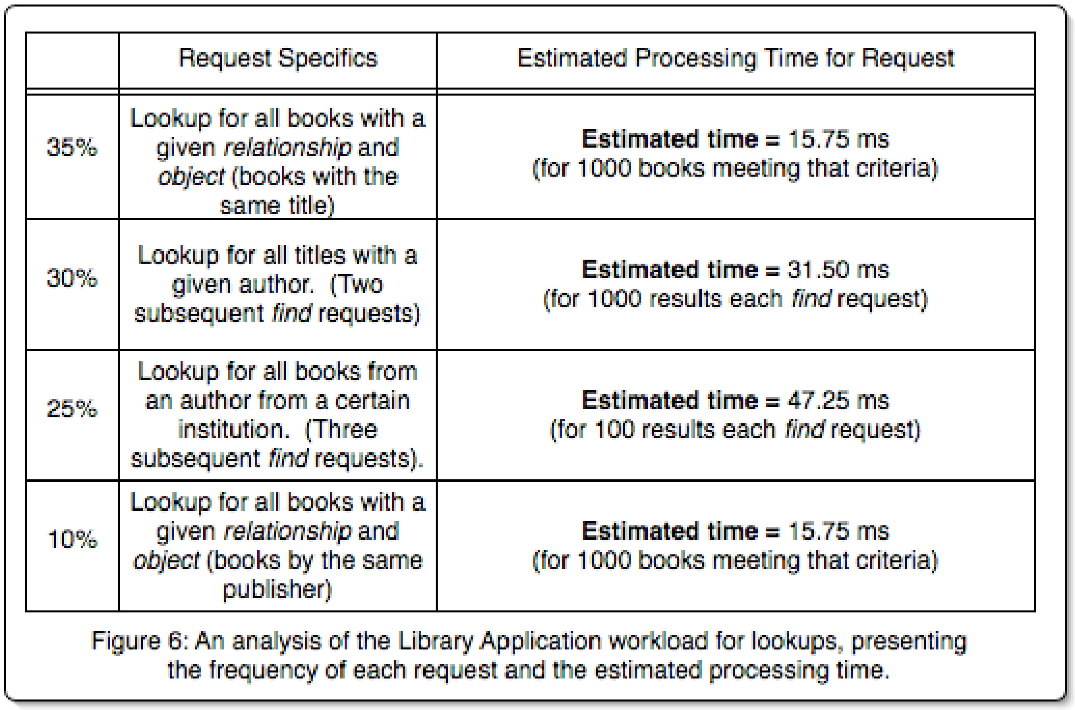
Figure 5: An analysis of the Flicker++ Application workload, presenting the frequency of each request and the estimated processing time.

In the Flickr++ Application, many triplet tags share similar elements. For example, all objects are tagged with a type, which means that there will be many triplets with identical *relationship* and *object* tags. Therefore, the partitions with triplets hashed on two elements will have a non-uniform distribution of triplets and a large number of blocks will have to be read into memory during triplet retrieval (the *remove* or *find* operations).

The processing time for adding and deleting images with the Flickr++ Application is much larger than the processing time for search. This is due to the large number of *inserts* and *removes* (multiple reads to and from disk) required for each addition or deletion of an image – a particular image may have multiple triplets associated with it, distributed across the disk. Though execution of addition and deletion requests takes up the vast majority of total processing time for the Flickr++ Application, addition and deletion of images occur infrequently: together they make up just 10% of total requests. As long as there are not an excessive number of triplets in the storage system, search will continue to perform well.

3.3.2 Library Application

The analysis of the Library Application is presented in Figure 6.



	Request Specifics	Estimated Processing Time for Request
35%	Lookup for all books with a given *relationship* and *object* (books with the same title)	**Estimated time =** 15.75 ms (for 1000 books meeting that criteria)
30%	Lookup for all titles with a given author. (Two subsequent *find* requests)	**Estimated time =** 31.50 ms (for 1000 results each *find* request)
25%	Lookup for all books from an author from a certain institution. (Three subsequent *find* requests).	**Estimated time =** 47.25 ms (for 100 results each *find* request)
10%	Lookup for all books with a given *relationship* and *object* (books by the same publisher)	**Estimated time =** 15.75 ms (for 1000 books meeting that criteria)

Figure 6: An analysis of the Library Application workload for lookups, presenting the frequency of each request and the estimated processing time.

In the Library Application, similar to the Flickr++ Application, many triplet tags share similar elements. This causes a non-uniform distribution of triplets and a large number of blocks to sort through in memory. However, unlike the Flickr++ Application, all of the Library data is inserted at one time and is never deleted. Because insertions require multiple reads to and from disk in the proposed storage system, the insertion of multiple triplets for every book in a library will be exceedingly slow. Each load of a book will require at least four inserts for *type*, *title*, *author*, and *publisher*; furthermore, additional inserts may be required if the book has multiple authors. Each load with four inserts will take 96.48 ms, thus a load of all books might take as long as several minutes to finish because of all the reads to and from the disk.

After the initial load, the Library Application workload consists of a number of different searches. Most of the requests include multiple *find* calls, which makes each request take up to three times as long as a single *find* call would. But again, the processing time for adding books to the system is much larger than the processing time necessary for conducting searches. After the initially high cost involved in loading all of the information into the system, the Library Application can be used to search for books relatively quickly.

4. CONCLUSION

The proposed tag storage system is optimized for fast retrieval of triplets. By minimizing random disk access in favor of sequential disk reads, a search for a particular triplet can be accomplished by reading a sequential set of blocks into memory. The efficiency of sequential disk reads and fast memory access times enable the system to find the desired triplet quickly. In addition, hashing every triplet four times – each according to a different combination of its elements – allows searches with wildcard strings to take place relatively quickly.

The storage design performs well when the load factor of the disk is low. But as the number of triplets increases, the performance of the proposed design decays. Key changes could be made to the system design that would allow for increases in triplets: allowing partitions with non-uniform distribution on disk to expand and rehash triplets would restore the distribution to a more uniform level, making retrieval more efficient.

5. REFERENCES

[1] Loudon, Kyle. "Description of Open-Addressed Hash Tables," Mastering Algorithms with C, Section 8.5, 1999.

3,260 Words