Simple, Effective and Reliable Tag Storage System Design

by Chuan Zhang

Instructor: Dina Katabi
TA: Szymon Chachulski
6.033 Spring 2007

I. Introduction

I.A. Overview

Applications such as Flickr or library catalogs often desire to store relationships between objects. Tags, <object, relationship, subject>, are versatile abstractions which succinctly capture this relationship. The following paper proposes a simple, effective, and reliable tag storage system design which applications can utilize to store and retrieve tags through an application programming interface (API).

I.B. Strategy

Our overall strategy emphasized the three points of simplicity, effectiveness, and reliability. A simple system consists of an overall structure and API implementation that is easy to understand, debug, and iteratively change. An effective system is one that meets stated performance criteria. In our case, performance is measured by completion time of API requests compared to a naïve implementation. A reliable system performs correctly over a range of applications and tag distributions.

We accomplished our goal through the use of double tagging, hash tables, and linked buckets.

I.C. Strategy Implementation

I.C.1. Double Tagging

Double tagging's use ensures a simple API. Applications send FIND requests for tags matching either a certain object and relationship or a certain subject and relationship. If tags were indexed only by relationship and object or only by relationship and subject, a FIND request sent on the un-indexed variable would suffer poor performance. Additionally, the API implementation would differ depending on which variable was sent in a request. Double tagging stores each tag twice, indexed by both relationship/object and relationship/subject. This technique simplifies the FIND API by making requests on either variable have identical implementations.

I.C.2. Hash Table

Hash tables provide effective performance. Hard disk is divided into 1,200,000 primary buckets. Given a certain input, a hash function can quickly return a corresponding bucket where matching tags should be located. Hash tables are extremely efficient methods of siphoning tags and allow us to find most

tags within one disk seek. Most other alternatives would either fail to offer acceptable performance or induce significantly increased complexity.

I.C.3. Linked Buckets

Linked buckets provide reliability when the system is used with different applications and tag distributions. Hash tables may fail if the number of tags assigned to a certain bucket exceeds the space available in the bucket. In our implementation, 150,000 secondary buckets may be "linked" to primary buckets. Tags which cannot fit into the primary bucket are put into a linked secondary bucket. Arbitrary numbers of secondary buckets, limited only by disk space, may be linked together to accommodate varying loads. This plan increases reliability when dealing with tag distributions with many "peaks".

II. Design Description

II.A. Storage Structure

II.A.1 Hard Disk Organization

Figure 1 shows an overview of hard disk organization. A 1000 GB hard disk was provided for the tag storage system. The hard disk was divided into three partitions. A root partition of size 100 GB was allocated for metadata. A primary partition of 600 GB was allocated for tag storage. A secondary partition of 300 GB was also allocated for tag storage.

The primary partition was equally divided into two 300 GB sections. One is used for storage indexed by object, the other for storage indexed by subject. Each section is further divided into 600,000 buckets each .5 MB in size. The secondary partition is divided into 150,000 buckets each 2 MB in size.

II.A.2. Bucket and Block Organization

Each bucket in the primary partition is identified by a number between 0 and 1,199,999. Each bucket in the primary partition is divided into 125 4 KB blocks.

Each bucket in the secondary partition is identified by a number between 1,200,000 and 1,349,999. Each bucket in the secondary partition is divided into 500 4 KB blocks.

Blocks store tags which can have a size between 0 and 160 bytes. Up to 25 tags can be stored in each block. Reading of individual tags can easily be done by reading the nth to the n+159 byte in a block where n is a multiple of 160. Blocks are identified by a number between 25,000,000 and 249,999,999.

II.A.3. Overhead

Figure 2 shows an overview of overhead organization. Two bucket tables reside in memory and track the state of buckets in the primary and secondary partitions. On shutdown, the two tables would be pushed into the root partition. On startup, the two tables would be read back into memory.

The primary bucket table tracks the state of buckets in the primary partition. Each of its 1,200,000 entries contains bucket #, start block #, full? bit and extend pointer (expointer).

The bucket # is simply the ID of the bucket. The start block # gives the index of the first block in a bucket. The full? bit is 1 when the all 160 byte slots in the bucket contain information and 0 otherwise. The expointer can either be false or contain the ID of a linked secondary bucket.

The secondary bucket table tracks the state of buckets in the secondary partition. Each of its 150,000 entires contains bucket #, start block #, an linked?

bit, a full? bit, and an extend pointer (expointer).

The bucket # is simply the ID of the bucket. The start block # gives the index of the first block in a bucket. The linked? bit is 1 if the bucket# of the examined bucket is in the expointer field of any other bucket. The full? bit is 1 when the all 160 byte slots in the bucket contain information and 0 otherwise. The expointer can either be false or contain the ID of a linked secondary bucket.

Each entry of the primary bucket table can be grossly overestimated to be of size 50 bytes. Each entry of the secondary bucket table can be estimated to be of size 25 bytes. The total size is approximately 64 MB, a negligible amount in memory and hard disk.

II.A.4. Hash Functions

Two hash functions exist. One hash function takes as inputs a subject and a relationship and returns a bucket index between 0 and 599,999. A second hash function takes as input a object and a relationship and returns a bucket index between 600,000 and 1,199,999. It is assumed that two non-ideal hash functions can be created which attempt to minimize the probability that two different inputs produce the same output bucket. Collisions are allowed in our implementation. Linked secondary buckets accommodate overflows.

II.B API Implementation

II.B.1 INSERT Implementation:

INSERT Pseudocode	Comments

`insertprimary(input) {`
`  bucket# = hash(input.subject, input.relationship)`
`  bucket = primarytable(bucket#)`
`  if (bucket.full?) {`
`  if (bucket.expointer) {`
`  insertsecondary(input,bucket.expointer)`
`  }`
`  else {`
`  bucket.expointer = LINKBUCKET()`
`  linkedbucket = secondarytable(bucket.expointer)`
`  linkedbucket.linked? = 1;`
`  insertsecondary(input,bucket.expointer)`
`  }`
`  }`
`  else {`
`  READ_BLOCKS(temp,bucket.startblock,125)`
`  INSERTTAG(input)`
`  if (bucket is full) {`
`  bucket.full? = 1`
`  }`
`  WRITE_BLOCKS(temp,bucket.startblock,125)`
`  }`
`}`
	insertprimary tries to insert tags into the primary bucket first
	
	if full, it calls insertsecondary
	
	if not full, inserts and modifies bucket table

`insertsecondary(input, bucket#) {`
`  bucket = secondarytable(bucket#)`
`  if (bucket.full?) {`
`  if (bucket.expointer) {`
	Insertsecondary tries to insert tags into a linked

insertsecondary(input,bucket.expointer)
}
else {
bucket.expointer = ALLOCATE()
linkedbucket = secondarytable(bucket.expointer)
linkedbucket.linked? = 1;
insertsecondary(input,bucket.expointer)
}
}
else {
READ_BLOCKS(temp,bucket.startblock, 500)
INSERTTAG(input)
if (bucket is full) {
bucket.full? = 1
}
WRITE_BLOCKS(temp,bucket.startblock,500)
}
}
	secondary bucket
	
	If full and another bucket linked, calls insertsecondary on new bucket
	
	If full and no other buckets, links another bucket
	
	If not full, inserts and modifies bucket table

Figure 3: INSERT Pseudocode

INSERT checks bucket tables for a primary bucket or secondary bucket that has room. It prefers primary buckets over secondary buckets and secondary buckets linked earlier to secondary buckets linked later. INSERT then READ_BLOCK's the entire bucket into memory, insert the tag into the first empty slot, and WRITE_BLOCK's the bucket back into disk. INSERT can also link a new secondary bucket if there is no room. INSERT updates the bucket table as needed. Specifically, non full buckets can become full (full? bit). Unlinked secondary buckets can become linked (linked? bit and expointer).

Figure 3 shown above is pseudocode for ½ of the INSERT API Implementation. The code above is run twice, once to insert tags by subject/relationship and once to insert tags by object/relationship.

II.B.2. DELETE Implementation

DELETE Pseudocode	Comments

deleteprimary(input) {
bucket# = hash(input.subject, input.relationship)
bucket = primarytable(bucket#)
READ_BLOCKS(temp,bucket.startblock,125)
DELETETAG(input)
if (bucket was full) {
bucket.full? = 0
}
WRITE_BLOCKS(temp,bucket.startblock,125)
if (bucket.expointer) {
deletesecondary(input,bucket.expointer)
}
}
 	
deletesecondary(input, bucket#) {
bucket = secondarytable(bucket#)
READ_BLOCKS(temp,bucket.startblock,500)
DELETETAG(input)
if (bucket was full) {
bucket.full? = 0
}
WRITE_BLOCKS(temp,bucket.startblock,500)
	deleteprimary finds matches within primary block and removes them
	if block was full, changes bucket table
	
	if another bucket linked, calls deletesecondary on that bucket
	
	deletesecondary reads a linked secondary block and removes matches.

`if (bucket was full) {`
`  bucket.full? = 0`
`}`
`if (bucket.expointer)`
`  deletesecondary(input,bucket.expointer)`
`}`
`}`
	can modify bucket table
	
	if other linked blocks, calls deletesecondary

Figure 4: DELETE Pseudocode

DELETE must access the primary bucket and all linked buckets, and remove any matches. DELETE simply performs READ_BLOCK, replaces any matches with an empty 160 byte array, performs WRITE_BLOCK and moves on to the next linked bucket. DELETE does not replace the holes that it creates. These holes can be filled by INSERT's or corrected during maintenance. DELETE also modifies the bucket table as needed. Specifically, it can change bucket's status from full to not full. It cannot unlink buckets.

Figure 4 shown above is pseudocode for ½ of the DELETE API Implementation. The code above is run twice, once to remove tags by subject/relationship and once to remove tags by object/relationship.

II.B.3. FIND Implementation

FIND Pseudocode	Comments

`findprimary(input) {`
`  bucket# = hash(input.subject, input.relationship)`
`  bucket = primarytable(bucket#)`
`  READ_BLOCKS(temp,bucket.startblock,125)`
`  FINDTAGS(input.subject, input.relationship)`
`  if (bucket.expointer) {`
`  findsecondary(input,bucket.expointer)`
`  }`
`}`
 	
`findsecondary(input, bucket#) {`
`  bucket = secondarytable(bucket#)`
`  READ_BLOCKS(temp,bucket.startblock,500)`
`  FINDTAGS(input.subject, input.relationship)`
`  if (bucket.expointer) {`
`  findsecondary(input,bucket.expointer)`
`  }`
`}`
	findprimary reads in tags from primary bucket
	
	findsecondary reads in tags from linked secondary buckets

Figure 5: FIND Pseudocode

FIND must access the primary bucket and all linked secondary buckets, check for any matches, and retain them. It does this by using READ_BLOCK to read entire buckets into memory, retaining matches in memory, then repeating for all linked buckets. Unlike DELETE and INSERT, it must perform its operations only once. If there is not enough memory to store all results, FIND returns "ERROR".

Figure 5 shown above shows pseudocode for FIND calls on subject and relationship. A FIND call on object and relationship can be created easily by

modifying the inputs to the hash function. Tags found will be aggregated together in an array and written to some address space shared with the application.

II.B.4. SHUTDOWN Implementation

Primary and secondary bucket tables are written to the root partition of disk. All other memory is discarded.

II.B.5. Maintenance Operations

During evening hours with little use, the tag storage system can be taken offline and maintenance operations can be performed. Holes can be filled simply by writing a bucket to memory, defragmenting all tags, and rewriting the bucket to disk. This can be performed iteratively on all buckets. Secondary buckets can be unlinked accordingly. Hash functions can be rewritten to minimize overflow. Data will be read into memory and then written into new buckets corresponding to the new hash function. These functions are not fully defined. The performance of the system benefits, but is not dependent, on these maintenance operations.

III. Design Analysis and Discussion

III.A. Performance

Obviously, performance varies greatly based on the application and tag distribution. We attempt to quantitatively estimate performance in general and in two specific applications, Flickr++ and a library catalog.

III.A.1 FIND and DELETE Performance

Performance of FIND and DELETE is closely related to the number of tags assigned to a given primary bucket. If tag distribution is approximately uniform, then the stated workload of 100 GB divided among 600,000 buckets would yield 1,600 tags per bucket. However, if the tag distribution contains "peaks", then separate seeks would be required to access every linked secondary bucket, reducing performance. Table 1 below summarizes performance times for FIND and DELETE.

DELETE's times shown are approximately half of the actual time. DELETE performs two removals from disk due to the double tagging. The identical tags could be arranged in different linked structures, leading to different times for each removal.

Tags assigned to primary bucket	# linked secondary buckets	FIND time	½ DELETE time *(see below)
1-3,125	0	19.67 ms	39.34 ms
3,126-15,625	1	61.84 ms	123.68 ms
15,626-28,125	2	104.01 ms	208.02 ms
28,126-40,625	3	146.18 ms	292.36 ms

Table 1: Find and Delete Performance

III.A.2 INSERT Performance

INSERT's performance depends little on the number of tags assigned to a certain primary bucket. The bucket tables in memory track full and not full buckets. INSERT simply has to find a not full primary bucket or linked secondary bucket and perform a READ_BLOCKS and WRITE_BLOCKS. If all buckets are full, INSERT can link a new secondary bucket and insert the tag there. Table 2 below summarizes performance times for INSERT.

INSERT's time shown is approximately half of the actual time. INSERT performs two insertions one indexed by subject and the other indexed by object. These two insertions could require different times.

Type of bucket tag inserted in	½ INSERT time * (see below)
primary bucket	39.34 ms
linked secondary bucket	84.34 ms

Table 2: Insert Performance

III.A.3. Flickr++ Performance

75% of requests in Flickr++ request are of the form FIND <*, “isa”, object>. If there are much fewer objects than buckets or some objects are tagged more than others, “peaks” will form and performance will suffer. However, it is highly unlikely this tag storage system will perform worse on average than the naïve implementation. 15% of requests are of the form FIND <subject, “owns”, *> and will take around 19.67 ms to complete. 5% of the requests are DELETE’s which may take a long time. However, this will have negligible impact on overall performance due to its rarity. Similarly, INSERT’s will have negligible impact on performance.

III.A.4. Library Catalog Performance

INSERT’s are called when the system initializes and DELETE’s are not allowed. Thus, the performance is determined solely by FIND requests. We do not expect many “peaks” in the library catalog tags. It is hard to imagine one author writing thousands of books, or one title being in more than one book. It is possible for there to be many thousand subjects of type “book”. However, no searches are conducted on this. Finally, 10% of FIND’s look for books from a given publisher. This could be a “peak” which would lead to decreased performance. However, overall, we expect the performance to be close to 19.67 ms.

III.B. Tradeoffs

III.B.1. Double Tagging: FIND vs. INSERT/DELETE

Double tagging increased the completion time of INSERT and DELETE and increased their complexity. Double tagging reduced the complexity of FIND and made its completion time consistent regardless of the input. Double tagging also reduced the maximum number of tags that could be stored. This tradeoff was justified because the overwhelming majority of API calls were to FIND.

III.B.2. Table Overhead: Complexity vs. INSERT Performance

Use of the extend pointer and full? bit added complexity to FIND, INSERT,

and DELETE. API implementations were required to alter the bucket tables whenever any changes were made to a linked bucket structure. A traditional linked list would have reserved the last spot of a bucket for a pointer to a linked bucket and determined full? by reading a bucket. The traditional implementation would not have allowed INSERT to make only one seek and would require it to read through a chain of linked buckets to find an empty slot. The increased performance justified the added complexity.

III.B.3. DELETE leaves holes: Complexity vs. Performance

DELETE leaves empty 160 byte slots in buckets after removing a tag. Holes waste space and are undesirable. If 3126 tags were written to a primary bucket and linked secondary bucket, a DELETE of a tag in the primary bucket could potentially allow the unlinking of the secondary bucket through rearrangement. The downside is enormous added complexity. Because INSERT first looks for holes in existing buckets, hole formation and deletion will balance in the steady state. Thus the performance does not justify the cost.

III.C. Limitations

One limitation of the system currently is the small number (150,000) of secondary buckets compared to 1,200,000 primary buckets. This confines the number of “peaks” to ~10% of the number of buckets. It is possible that “peaks” are much more common and as the size grows, not enough secondary buckets will be able to be allocated. A possible solution is reducing the number of primary buckets and increasing the number of secondary buckets.

IV. Conclusion

The tag storage system explored provides a simple, effective, and reliable design for a wide range of applications. Double tagging ensures simplicity for the FIND API. Hash buckets ensure effectiveness by quickly finding the corresponding bucket for any input. Linked secondary buckets ensure reliability by allowing flexible data structures to accommodate “peaks”.

Overall, the tag storage system provides good performance for the Flickr++ and library catalog applications. This system could be improved by tweaking primary bucket sizes and secondary bucket sizes. The optimal sizes and number of primary/secondary buckets depend on the hash function and the particular application. Other performance enhancements could also be added. These include the use of caches or last block # for bucket.

Word Count: 2460