An Efficient Hash-Based Tag Storage System

6.033 Design Project 1
Kevin Wang
Alex Yip [TR11]
TA: Szymon Chachulski

1. Introduction

The goal of our project is to design a tag storage system that *efficiently* manages hundreds of gigabytes of tags to answer user queries. The API we must implement supports INSERT, REMOVE, FIND, and SHUTDOWN operations which allow users to add/delete tags, search through them, and turn off the system, respectively. Tags are triplets of strings, and how we store them on disk is the main focus of this project. To maximize efficiency and simplicity, our solution employs the following design decisions:

1. Data is stored on disk in contiguous pre-sized *chunks*. This allows us to find a specified chunk (given a chunk index number) with only one disk seek. When one chunk fills up, we link it to another chunk that also holds relevant information, thus creating linked lists of chunks ("chunk lists").
2. Tags are redundantly stored in three hash tables instead of one. This helps us quickly service FIND queries with one wildcard. Accessing the values of each table requires two layers of hashing.

The goal of this system is to provide reasonable runtimes for INSERT, REMOVE, and fast FIND queries while maintaining a simple design. Our rationale is that in most uses of a tag storage system, searches will occur much more often than modifications. Therefore, we optimize for finds. We accomplish this by a two-layer hashing scheme that quickly find relevant data chunks, as well as caching recently used chunks to potentially avoid disk seeks. Our analysis shows that for use cases of interest, the average performance for operations are acceptable, and average find operations are fast (< 100 ms).

2. Design Description

This section provides an overview of the design. Section 2.1 discusses how tags are stored on disk in chunks. Section 2.2 explains how chunks are addressed via hashing. Section 2.3 discusses performance optimizations. Section 2.4 describes the implementation of the API.

2.1 Tag Storage on Disk

Tags are triplets in the form `< SUBJECT, RELATIONSHIP, OBJECT >`. Tags are stored inside data chunks, the primary unit of storage in our system.

2.1.1 Chunk Size and Addressing

Each chunk is collection of 256 disk blocks (1 MB in size). Since our disk has 1 TB = 2^{40} bytes of storage, this implies we can store 2^{20} chunks, which is more than 1 million. We address chunks with 20 bits addresses. Given a chunk index *index*, the beginning location of that chunk on disk is *index* × 2^{20}. We call the 15 most significant bits of an address the "high end" and the 5 least significant bits of the address the "low end" for reasons related to hashing, which will soon be made clear.

2.1.2 Chunk Structure

Each chunk is partitioned into metadata and data. The metadata stores state information about its particular chunk. Since each chunk is < 2^{20} bytes, a 20 bit address can be used as an offset to address every byte in the chunk. Bits 0-19 of the metadata (the *free pointer*) point to the next free byte in the chunk data. This provides fast insertions and tells us how much space is left on the current chunk. Bits 20-39 (the *next chunk pointer*) contain a pointer to another chunk on disk. This allows us to create linked lists of chunks to store data, should we run out of space on this chunk. We calculate the value of *next chunk pointers* by quadratic probing. Figure 1 shows an overview of a chunk list. Bit 40 (the *end-of-list bit*) is set to 1 if and only if this chunk is the end of the list, and 0 otherwise.

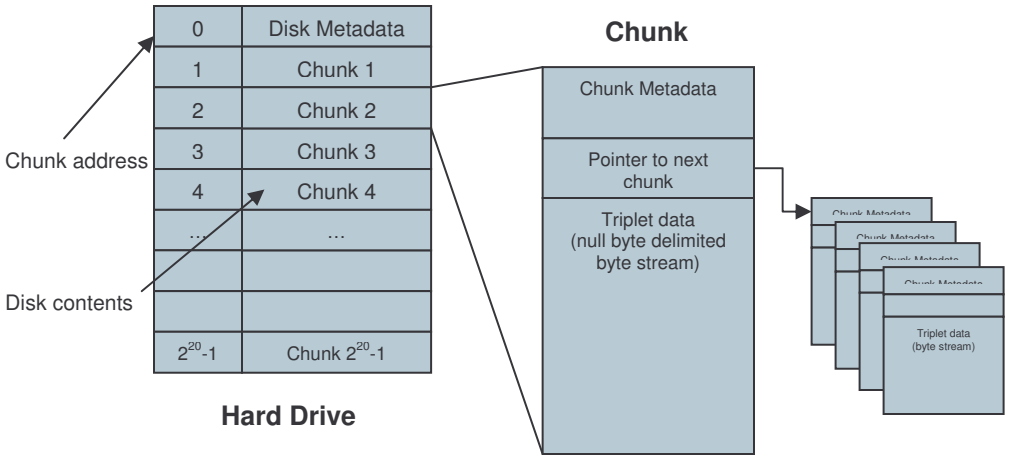

Figure 1: An overview of the triplet storage hierarchy on disk, with an emphasis on chunks. Notice how chunks form a linked list when we need more space to store triplet relevant to a string. We can store exactly 2^{20} chunks on disk. Chunk 0 is special in that it stores disk metadata, namely the hash functions we use. Its structure is different from other chunks in that it is only a 1 MB byte stream. If any hash function addresses Chunk 0, the read/write will get pushed automatically to Chunk 1.

The chunk data begins at bit 41 and continues until the end of the chunk. Each triplet `< A, B, C >` is stored in the chunk data by inserting a null byte (0x00, or ASCII value 0) between byte streams representing the triplet strings. In each of the three hash tables, this triplet is stored as three null byte delimited byte streams in the order "A" "B" "C". Reconstructing triplets requires reading the data byte stream and counting null bytes modulo 3. Refer to Figure 2 for to see how triplets are stored on disk.

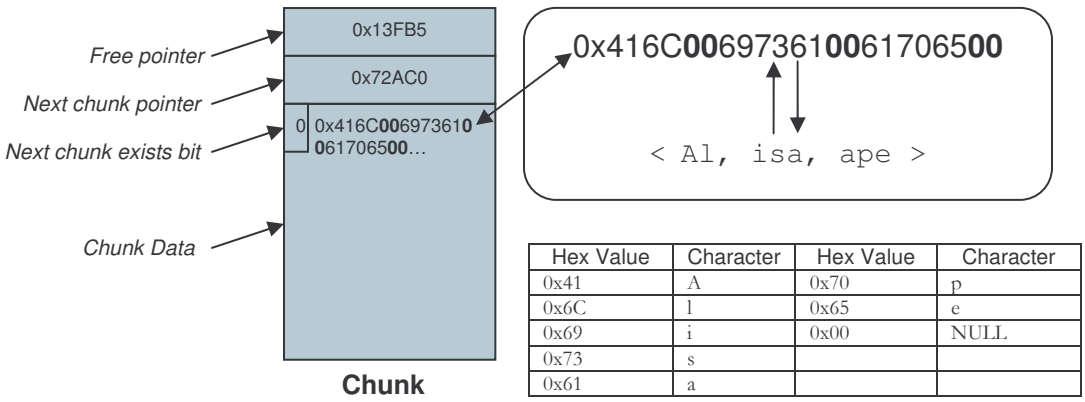

Figure 2: A more in-depth look at a particular chunk. Notice the free pointer, next chunk pointer, and end-of-list bit, as well as how the byte stream is null byte delimited (0x00, bold) and translated into a triplet.

2.2.1 Hash Functions

Given a triplet < SUBJECT, RELATIONSHIP, OBJECT >, we determine which chunk that triplet belongs by using uniform hash functions h_k that map strings to k-bit integers. We classified the most significant 15 bits of an address as the "high end" and the least significant 5 bits as the "low end". For example, to find a chunk address given a triplet, we build a 20 bit address such that the high end is h_{15}(SUBJECT) and the low end is h_5(RELATIONSHIP).

2.2.2 Hash Tables

We keep three hash tables to cater to FIND queries with one wildcard: the subject-relationship table, the object-relationship table, and the subject-object table. The first is keyed on subject and relationship. It hashes to an address as described in Section 2.2.1. The second is keyed on object and relationship. The high end of the chunk address is h_{15}(OBJECT) and the low end is h_5(RELATIONSHIP). The third table is keyed on subject and object and is accessed slightly differently. Since subjects and objects are about equally important, we use hash functions that map from strings to 10 bit integers. The high *ten* bits of the chunk address is h_{10}(SUBJECT) and the low *ten* bits is h_{10}(OBJECT).

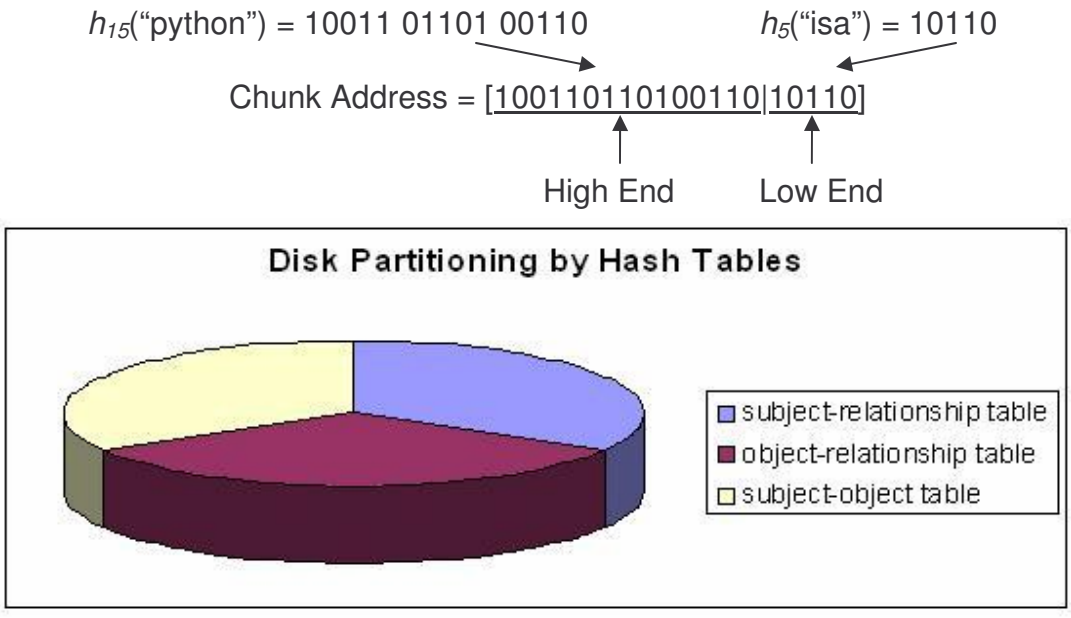

Figure 3: A sample chunk address showing how it is built from two hash functions (in this case, the address is for the subject-relationship table). This figure also shows a pie chart of how the disk is partitioned into three areas, one for each hash table.

It follows that every triplet is stored in three chunks on disk. To prevent these three tables from overlapping, we partition the disk into three contiguous sectors, one for each table (roughly $2^{40}/3$ bits per partition). Refer to Figure 3 for an overview of the hashing system.

Note from Figure 1 that Chunk 0 is special: it is allocated for disk metadata, namely the hash functions we use. Its structure is not the same as other chunks, but is instead just a 1 MB byte stream.

2.3 Performance Optimizations

We use most of our 1GB of memory to cache recently accessed chunks. We allocate 900 MB of contiguous space in RAM for 900 chunks (the chunk cache). We keep track of the cache with a 900-node linked list. Every node stores 20 bit chunk address and an address that points to the beginning of that chunk in RAM. To determine if a chunk is cached, we walk down the list and compare addresses. If found, we follow the pointer and operate on the cached chunk in RAM, thus saving us a disk seek. If not found, the chunk read from disk placed in the cache at the first available 1 MB slot (with a corresponding node at the front of the linked list). This cache uses the Move-To-Front heuristic: every time a node is accessed, it is moved to the front of the list. This cache also uses the Least Recently Used (LRU) policy: when we run out of cache space, the node that was *least* recently accessed is removed. The displaced chunk is then written to disk. Refer to Figure 4 for an overview of the cache system.

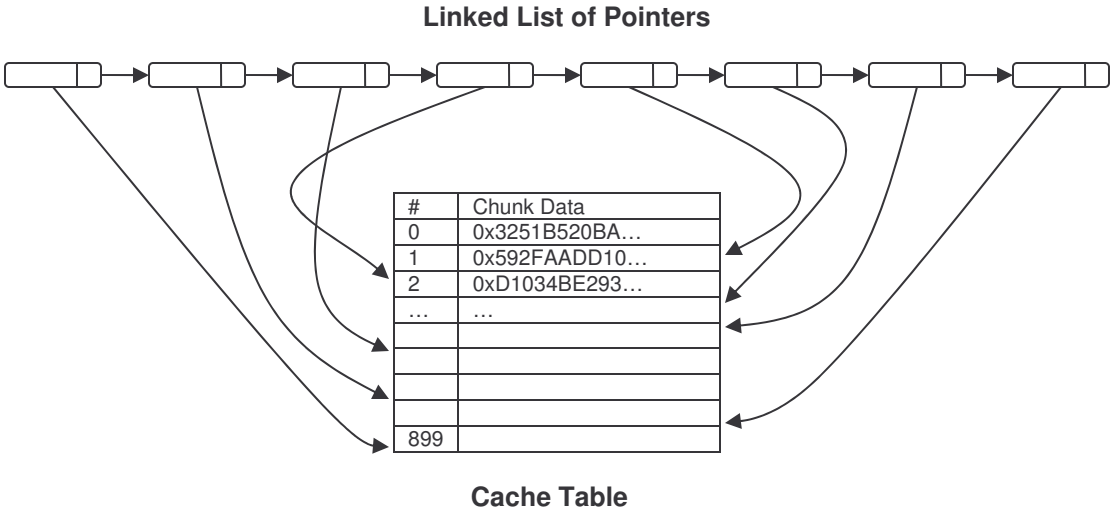

Figure 4: A sketch of the caching system. Each node in the linked list stores a chunk disk address as well as a pointer to the cache table that corresponds to the cached value of the chunk.

2.4 API Implementation

This section describes how INSERT, REMOVE, FIND, and SHUTDOWN calls are implemented. Note that `sizeof(chunk)` is the number of blocks per chunk (256 for us). We first present two convenient wrappers for READ_BLOCKS and WRITE_BLOCKS that use the chunk cache. `addr` is the disk address of the desired chunk.

READ_BLOCKS_CACHED(addr, blocks, start_block, num_blocks)

1. If the chunk corresponding to `addr` is cached in RAM, set `blocks` • the cached chunk from RAM
2. Otherwise call READ_BLOCKS(blocks, start_block, num_blocks) and insert the newly read chunk data into the chunk cache.

WRITE_BLOCKS_CACHED(addr, blocks, start_block, num_blocks)

1. If the chunk corresponding to `addr` is cached in RAM, overwrite the cached chunk with the new chunk data in `blocks`.
2. Otherwise call WRITE_BLOCKS(blocks, start_block, num_blocks)

2.4.1 Implementation of INSERT(SUBJECT, RELATIONSHIP, OBJECT)

We assume we have enough space on disk for insertion. We make inserts idempotent or else tag deletion would be troublesome.

1. Call FIND(subject, relationship, object, 0, 1). If the return value is 1 (triplet exists), return.
2. Based on the triplet to insert, compute the chunk addresses for each of the three hash tables.
3. For each of the three chunk addresses `addr`
    a. READ_BLOCKS_CACHED(addr, chunkData, addr*sizeof(chunk), sizeof(chunk))

b. Parse `chunkData` to see if we have space to insert the triplet (based on the *free pointer*)
c. While we *don't* have space
    i. `addr` ← value of *next chunk pointer*
    ii. If *end-of-list bit* `== 1` (no space in this chunk, and we're at the end of the list)
        1. Compute a new *next chunk pointer* by quadratic probing
        2. addr ← value of *next chunk pointer*
    iii. `READ_BLOCKS_CACHED(addr, chunkData, addr*sizeof(chunk), sizeof(chunk))`
d. Write the subject, relationship, and object, into `chunkData` at the address pointed to by *free pointer*, delimiting the three strings with the null byte (0x00) character.
e. `WRITE_BLOCKS_CACHED(addr, chunkData, addr*sizeof(chunk), sizeof(chunk))`

Notice we may not write the modified chunk to disk until our cached entry is displaced or on a call to `SHUTDOWN()`, which we'll see soon.

2.4.2 Implementation of `REMOVE(SUBJECT, RELATIONSHIP, OBJECT)`

We assume that we only call remove on a triplet that's been previously inserted.

1. Based on the triplet to remove , compute the chunk addresses for each of the three hash tables.
2. For each of the three chunk addresses `addr`
    a. `READ_BLOCKS_CACHED(addr, chunkData, addr*sizeof(chunk), sizeof(chunk))`
    b. Scan through `chunkData` to see if this triplet exists.
    c. While we *didn't* find the triplet in this chunk
        i. `addr` = value of *next chunk pointer*
        ii. `READ_BLOCKS_CACHED(addr, chunkData, addr*sizeof(chunk), sizeof(chunk))`
        iii. Scan through `chunkData` to see if this triplet exists.
    d. Overwrite the triplet by shifting up the remainder of `chunkData` (after the end of the `object` string) to the beginning point of the triplet (beginning of `subject` string). Update the *free pointer* in `chunkData` to reflect this change.
    e. `WRITE_BLOCKS_CACHED(addr, chunkData, addr*sizeof(chunk), sizeof(chunk))`

Once again, we may defer writing to disk until a chunk is displaced from the cache or `SHUTDOWN()` is called.

2.4.3 Implementation of `FIND(SUBJECT, RELATIONSHIP, OBJECT, START, COUNT)`

We show the implementation for zero or one wildcards in our query. This satisfies our specified use cases. We address the other cases informally.

1. If we have no wildcards, compute the chunk address for the subject-object table.
2. Else we have one wildcard
    a. subject is the wildcard: compute the chunk address `addr` for the object-relationship table.
    b. relationship is the wildcard: compute the chunk address `addr` for the subject-object table.
    c. object is the wildcard: compute the chunk address `addr` for the subject-relationship table.
3. Initialize `found` ← `0`
4. Initialize `results[count]` ← `empty`

5. Do while *next chunk exists bit* `== 1 && found - start < count`
    a. READ_BLOCKS_CACHED`(addr, chunkData, addr*sizeof(chunk), sizeof(chunk))`
    b. Scan through `chunkData` and for every triplet that matches the desired keys (taking into account any wildcards)
        i. If `found ≥ start && found - start < count`
            1. Build a triplet structure with the search result
            2. Set `results[found - start]` ← triplet structure
        ii. `found` ← `found + 1`
    c. `addr` = value of *next chunk pointer*
6. RETURN `results`

If all three arguments are wildcards, we need to scan through every single chunk on disk to find triplets. If we have two wildcards but the subject or object is specified, we can hash on the subject or object to get the high end of the address. The low end is a 5 bit number, so we need to scan through all possible chunks represented by the low end bits ($2^5 = 32$ chunks), in addition to any other chunks pointed to as part of chunk lists. We cannot service find queries where the relationship is specified but both the subject and objects are wildcards.

2.4.4 Implementation of SHUTDOWN()

SHUTDOWN writes the cached chunks to disk.

1. For each chunk's `chunkData` in the cache and its corresponding `addr` stored in the linked list
    a. WRITE_BLOCKS`(chunkData, addr*sizeof(chunk), sizeof(chunk))`

3. Design Analysis and Discussion

Here we present a performance analysis of our system. Section 3.1 examines system performance for the specified use cases. Section 3.2 discusses design tradeoffs, their rationale, and consequences.

3.1 Performance Analysis

We do not consider gains from caching. As a useful preliminary calculation, we find reading one chunk from disk requires 12.7ms + 0.06ms/block × 256 blocks = 28.06 ms.

3.1.1 Assumptions

We make the following assumptions:

1. Applications will want < 1000 results at a time.
2. Calculating the chunk addresses takes negligible time compared to disk seeks.

3. For each insert, we have one seek on average.
4. To find 1000 results, we have two seeks seek on average.

Assumption 1 comes from a usability standpoint: displaying 1000 tags is overwhelming for any human to consume at once. Assumption 2 is safe since calculating hash values take nearly constant time. Assumptions 3 and 4 are reasonable because every chunk stores almost 10,000 entries, so the chance many chunks filling up is low. This is especially the case since our disk can effectively store about 300 GB of data (1 TB / 3 hash tables), and we only expect 100 GB of triplets.

3.1.2 Flickr++ Workloads

For each insert, we read on average one chunk per hash table, so reading three chunks takes 84.18ms. Writing those three chunks back to disk requires about the same time, so inserting one tag requires about 170 ms.

Table 1: The workload results for the Flickr++ application under the assumptions laid out before. We achieve a reasonable average time per operation that is < 100 ms.

Workload	% Freq.	Estimated Time (ms)	Explanation
Adding/Deleting an image	5	**500 ms** per image	We have to process 3 tags per insert/delete
Tagging an image with a "isa" relationship	5	**170 ms**	The cost of one insertion
Searching for images with a particular "isa" triplet	75	**66 ms**	For 1000 results: average two disk seeks ~60 ms, and scanning through RAM takes ~6 ms ($1/10^{th}$ of disk reading cost)
Searching for objects associated with a given user	15	**66 ms**	Cost of one `FIND`
Weighted Expected Time	100	**92.9 ms** average time per operation	Average time weighted by frequency

We note that find queries are faster than inserts since the latter requires accessing three hash tables whereas searching only reads from one hash table. The last row shows the average estimated time per operation weighted by % frequency. The high likelihood of find queries brings down the average cost to below 100 ms per operation.

3.1.2 Library Workloads

Table 2: The workload results for the Library application. The average time per operation is much higher than Flickr++, but is still less than one second.

Workload	% Freq.	Estimated Time (ms)	Explanation
Inserting Books	-	**500 ms** per book	Process 3 tags per insert/delete
Searching for a book with a given title	35	**66 ms**	Cost of one FIND
Searching for titles by a given author	30	**132 ms**	Cost of two FIND calls
Searching for books from an author from a particular institution	25	**330 ms**	First make two FIND calls, then a find on each intersection. We assume about 3 authors per institution with a given name (5 finds total)
Searching for books from a given publisher	10	**6.6 seconds**	Assuming each publisher prints 100,000 unique books on average, we need 100 FIND calls, each returning 1000 results
Weighted Expected Time	100	**805.5s ms**	Average time weighted by frequency

The weighted expected time is much higher than for the Flickr++ workloads. This can be traced to the 6.6 seconds required to find all books from a given publisher. However, an average time of 1 second per operation is a reasonable waiting time for a library user.

3.2 Design Tradeoffs

The primary objective in designing this system is supporting fast FIND queries while maintaining reasonable runtimes for insertion and deletion and keeping a simple design. Our design makes many tradeoffs to accomplish this, which has certain performance consequences.

3.2.1 Threefold Redundant Hash Tables

By maintaining three hash tables, we improve the performance of FIND queries. This cost is a threefold decrease in available space and threefold slower insertions and deletions. The rationale is that in our use cases, FIND queries account for 90-100% of the prescribed workload. The space tradeoff is acceptable because we have three times more space than our expected space usage of 100 GB. Another advantage of having these three hash tables is that doing FIND operations with one wildcard costs on average the same, regardless of where the wildcard is. This makes our system robust to all find queries with one wildcard. As a downside, our system performs poorly when we do a find with two or three wildcards (we cannot even perform finds where the two wildcards are the subject and object). Our

justification is twofold: 1. These searches are not in *any* use case and 2. It would require much more space to support these operations.

3.2.2 Simplicity in Hashing versus B-Trees

In maintaining simplicity we choose to use hashing instead of B-trees. Our rationale is that hashing is *simple*, yet maintains good average runtimes without the complexity associated with B-trees: scheduling background tasks to rearrange blocks, balancing nodes, maintaining structure, etc. As the escalating complexity principle tells us: adding a feature increases complexity out of proportion. B-trees also present the problem of storing variable length strings in fixed sized nodes.

One disadvantage of hashing is that there is no guarantee of locality. We argue that a linear scan of triplets to find results is cheap because our two layer hashing system dramatically cuts down the search space. Also, once the data has been loaded into RAM, a linear scan does not cost much compared to reading data from disk. The scan costs approximately 10% more work. Another advantage of hashing is that it doesn't require extra disk space to store structural information.

3.2.3 Comparison to Naïve Approach

A simple naïve approach would be to keep one hash table keyed on one of the three strings of the triplets. After hashing into a chunk, we must scan the chunk to find all matching queries. The main problem is that one-layer hashing doesn't narrow down the search space enough to make linear scanning cheap.

3.2.4 Scaling Issues

We scale poorly with space. If the expected number of triplets were to triple from 1 billion to 3 billion, we would barely have space to store everything. Worse, the number of hash collisions would rise exponentially, increasing the cost of searching as we traverse more chunks and incur more disks seeks. One solution might be to remove the subject-object relationship table, which would cut our space usage by 1/3, as well as provide better performance with use cases, but worse performance for non use cases.

4. Conclusion and Future Work

The tag storage system we present is optimized for searching rather than modifying. The key aspects of this system is pre-sized disk blocks that allow for convenient indexing by hashing and two-layer hashing that increases efficiency by reducing disk seeks and linear scanning costs. Performance analyses show that the average times for operations on this system are within acceptable ranges (< 1 second), although some operations take much longer. The use cases show that these operations occur very infrequently.

A main theme behind this design is balancing simplicity with generality. We see simplicity in the hashing system, the fixed size chunks, and minimal metadata. The caching system is also simple, using only a linked list with an LRU policy. Despite these simplicities, our system still achieves reasonable runtimes for most use cases. It also provides modest generality to operations that are not specified in our use cases.

There are several areas which need improvement. We use quadratic probing as our hash collision mechanism. This reduces clustering, but also destroys locality of reference. Changing the system to take advantage of spatial locality of data is a good next step. Secondly, our method for servicing FIND operations with high start indices (say we wanted results 20,000-21,000) is inefficient, since we have to linear scan for the first 20,000 results before we begin collecting relevant search results. Caching helps this problem, but doing a sequence of consecutive reads that accesses chunks in the sequence: 1, 1-2, 1-2-3, … is an $O(n^2)$ algorithm (where n is the number of chunks in the chunk list). It is not difficult to conceive of an $O(n)$ time algorithm that stores extra chunk metadata. Lastly, the system could benefit from reducing space consumption. Perhaps there is some hashing algorithm that allows us to hash by any two of the three keys in the triplet, but store only one copy of the data on disk.

5. Acknowledgements

I would like to thank my TA Szymon Chachulski for a short but enlightening discussion.

Word count: 2489

(not including figures, tables, title page, pseudocode, and acknowledgements)