Tagged Data System

Design Project One
6.033 / Massachusetts Institute of Technology
Sam Madden TR 10 / Hooria Komal F1
March 22, 2007

Alexander Valys

1 Overview

This document describes the implementation of a database-like system for storing 'triplets', as required by the first 6.033 design project. These are short data elements of the form `subject,relationship,object`.

The system described is capable of storing approximately eight billion triplets (of approximately 100 bytes each) on a machine with 1 TB of available disk space, and 1 GB of RAM. Insert, delete, and wildcard-based search are supported. All operations in the example workloads run in constant time, with insert and delete taking between 72 ms and 96 ms, and searching as little as 18 ms (for a fixed number of results).

This performance is achieved by maintaining two redundant but differently-organized copies of the data set on disk, each consisting of one-or-two-level hash tables, and compressing the raw triplet data before storing it.

2 Design Description

2.1 Data Structure Overview

Each copy of the data set is referred to as a 'primary collection'. One is organized by each triplet's subject, and the other by each triplet's object.

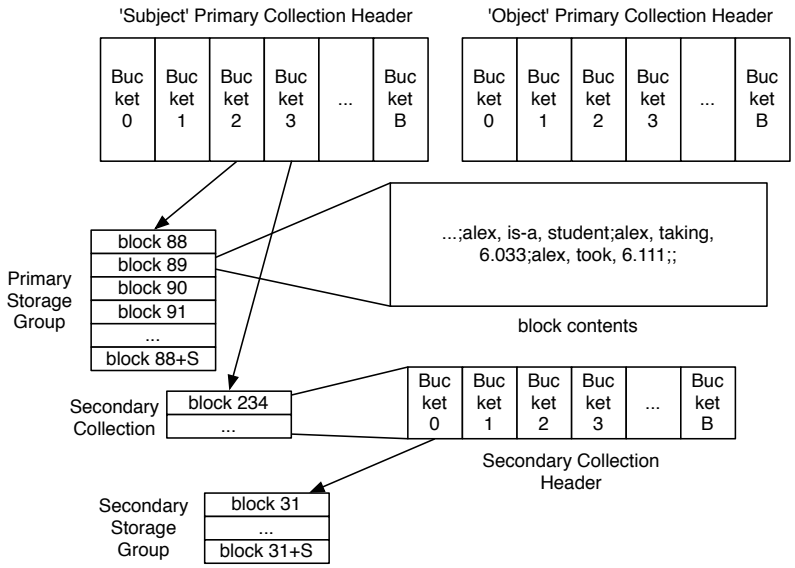

Figure 1: Primary and secondary collection structure overview.

Within each primary collection, location of specific triplets is accomplished using a hash-table-based lookup system. The collections are divided into some number B of buckets, with each bucket corresponding to a specific range of

values of the hash function. Either the subject or object triplet field value is used as a key, depending on the primary collection. Each bucket is associated with a storage group, which is a sequence of some number S of contiguous blocks on disk in which triplets are actually stored.

For extremely large buckets, the system uses another level of indirection. Rather than pointing directly to a storage group, the bucket points to a secondary collection. This is another hash table, with a variable number of buckets, keyed on the value opposite that of the primary collection it is within (i.e. a secondary collection within the subject primary collection would be keyed on object). Inside the secondary collection are a number of additional buckets, each pointing to a storage group of length S.

2.2 On-Disk Layout

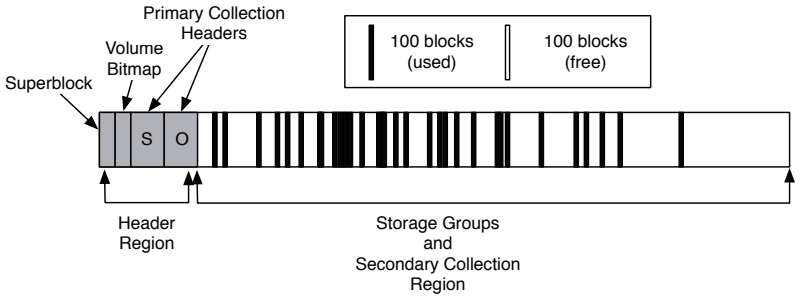

Figure 2: On-disk layout of headers and data structures.

Figure 2 indicates the on-disk layout of the data structures described above. The first block on disk, the superblock, contains the location (starting block and length) of the two primary collection headers. Each header is a large array, implementing a lookup table that associates each bucket in that primary collection with the starting block and length of either a storage group, or secondary collection header. The target group type is indicated with a bit flag. Secondary collection headers take the same form as primary collection headers, except they may only point to storage groups.

Between the superblock and the primary collection headers on disk is the volume bitmap, indicating which blocks are in use.

Triplets are stored within storage group blocks as compressed plaintext, delimited by semicolons and commas. For example:

```
;subject,relationship,object;another,relationship,another-object;;
```

Delimiters are permitted in field values, but they must be escaped with forward slashes. The last triplet stored within each block of a storage group is indicated by a double semicolon.

Each block within a storage group is compressed separately, using an algorithm such as BZ2.

2.3 Caching

The primary collection headers and volume bitmap are stored and modified in-memory during normal execution of the system, and written to disk only during shutdown.

The remaining in-memory space is used to cache secondary collection headers, using a least-recently-used replacement strategy.

3 API Implementations

3.1 Search(Subject, Relationship, Object, Start, Count, SessionInfo)

Because of the way the data is laid out on disk, different search behavior is required depending on the pattern of wildcards used in the request.

3.1.1 A: (*, reln, obj), (subj, reln, *), (*, *, obj), (subj, *, *)

To perform a search of this type, we must first select a primary collection to use. We use the object collection if the subject search field is a wildcard - otherwise, we use subject.

Having selected a collection, we then determine the bucket we will search in by computing the hash of the appropriate field, and then find the structure corresponding to that bucket from the corresponding primary collection header. If it is a storage group, we can read it directly - if it is a secondary collection, we must read the secondary collection header from disk or cache, and start reading from the storage group associated with the first secondary bucket.

We read the storage group with a single `read_blocks` command, and return the triplets that match the search criteria.

If we are reading a secondary collection and the first bucket does not contain *Count* results, we proceed to read the second bucket.

3.1.2 B: (subj, *, obj)

This search type uses the subject primary collection.

If the structure associated with the primary bucket is a storage group, we read that directly. If it is a secondary collection, we determine the secondary bucket to read using the hash of the object field, and then read the associated storage group, returning triplets that match the search criteria.

3.1.3 C: (*, reln, *), (*, *, *)

For this type of search, we have no choice but to read all storage groups linearly, looking for any triplets with the specified relation.

3.1.4 SessionInfo

I had to modify the API slightly to improve performance. The Search procedure now returns a SessionInfo object along with the result set. This object must be passed back to Search when the client wants to retrieve additional values from the same result set. It contains a pointer to the storage group that the last triplet in the set came from, allowing the server to resume the search starting from that location.

3.2 Insert(Subject, Relationship, Object)

To insert a triplet, we must add it to both primary collections. The procedure is identical for both of them: we compute the primary bucket we will insert into by computing the hash value of the appropriate triplet field, and find the group associated with that bucket. If it is a storage group, we insert it directly (after verifying that it is not a duplicate) - if it is a secondary collection, we repeat the lookup process for the secondary buckets.

If the storage group associated with a primary bucket is full, we must convert it into a secondary collection. The storage capacity of the new secondary collection is the capacity of the original storage group, multiplied by the expansion factor E. If a secondary bucket becomes full, we increase the number of buckets by E. Both operations require reading and rewriting the entire contents of the group or collection.

3.3 Delete(Subject, Relationship, Object)

To remove a triplet, we follow the same procedure as an insertion, except that we remove it from the storage groups, rather than inserting it. However, the delete operation does not attempt to reverse any expansions created by $Insert$.

3.4 Shutdown()

During shutdown, we write the in-memory volume bitmap and primary group headers to disk, as well as any cached secondary group headers.

3.5 Low-Volume Maintenance

During periods of low activity, the system scans all secondary collections, looking for those that have become sparse - which is to say, collections that are less than $\frac{1}{E}$ full. Upon finding one, it collapses it into either a smaller collection, or a single storage group.

4 Performance Analysis

4.1 Operational Parameters

The system uses several parameters to control the size and organization of its data structures. These are summarized in table 1.

Table 1: System Parameter Descriptions

Parameter	Default	Purpose
B	625,000	Number of buckets in each primary group
S	100	Length of a storage group, in blocks
E	2	Secondary group expansion factor

The default values listed in table 1 are the ones we will use for performance analysis. These were chosen somewhat arbitrarily: experimental testing would be required to fine-tune them for a given installation.

The default of S was chosen so that reading a single storage group takes 18 ms, and the default value of B was chosen so that 500 GB of our available 1TB is pre-allocated to primary storage groups (with 500 GB available for expansion with secondary collections).

With these parameters, the primary collection headers occupy approximately

$$625,000 \text{ buckets} \times 5 \text{ bytes/bucket} \times 2 \text{ buckets} = 6.25 \text{ MB} \quad (1)$$

of storage space - leaving plenty of room in RAM to cache the secondary collection headers, which will be much smaller (625 bytes for a million-triplet collection). The system should be able to keep several million secondary collection headers cached at any given time.

4.2 General Workload Analysis

Before analyzing specific workloads, it will be helpful to calculate some performance figures for the three operations in general.

In all analyses, we ignore the effect of the secondary header cache. In addition, we assume that the maximum value of *Count* used in searches is 1000, and that a search of type A or B is being performed.

Table 2 contains the estimated execution times - see the sections that follow for their derivation.

Table 2: Operation Performance Estimates

Operation	Min	Max
Insert	72 ms	96 ms
Delete	72 ms	96 ms
Search	18 ms	48 ms

Compare these times to that of a naive implementation, in which the entire disk is read linearly during insertion (for duplicate checking), deletion and searching. Those operations would run in $O(n)$ time, with a maximum runtime on a billion-triplet database of approximately *25 minutes each*.

$$\frac{(12 + .06(\frac{100e9}{4000}))\,\text{millis}}{1000\,\text{millis/s} \times 60\,\text{s/min}} = 25\,\text{min} \tag{2}$$

4.2.1 Insert

On insert, we must write the new triplet to disk twice - once in the subject primary collection, and once in the object primary collection.

Inserting into a single primary collection, when no secondary collection is present, requires reading 100 blocks (the contents of the storage group), and writing 1 block (one of the 100, modified to store the new triplet). This takes:

$$12 + 0.06(100) + 12 = 36\,\text{ms} \tag{3}$$

Inserting when there is a secondary collection present requires reading 1 block (containing the secondary collection header), reading 100 blocks (the contents of the secondary storage group), and then writing 1 block. This takes

$$12 + (12 + 0.06(100)) + 12 = 48\,\text{ms} \tag{4}$$

Since this needs to be done for both primary collections, this means that a single insert will take anywhere from 72 to 96 ms, depending on whether there are secondary collections associated with the subject and object primary collection buckets.

However, the above analysis does not account for the times when the insert procedure finds that a collection is full, and is forced to either create a new secondary collection or expand the existing one.

We can determine the average insert time by amortized analysis - computing how much time the expansion takes, and distributing that cost over all the inserts leading up to it.

Whenever an expansion is required, we must read every n storage groups in the current collection, recompute the hash functions, and then write out the data they contain to $2n$ storage groups on disk. The amount of time this will take is given by this equation:

$$18n + 15 \times 2n = 48n\,\text{ms} \tag{5}$$

It takes 18 ms to read each of the n old, full storage groups, and 15 ms to write each of the $2n$ new storage groups, which will be only half-full.

After being expanded to n groups, a collection is only expanded again to $2n$ once an additional $4000n$ triplets are inserted into it. Thus, the average effect of an expansion on the average insert is given by:

$$\frac{48(2n)}{4000n} = \frac{3}{125} = 0.024\,\text{ms} \tag{6}$$

So, by amortized analysis, the need for occasional expansions adds 0.024 ms to each insert on average. This effect is negligible.

4.2.2 Delete

A delete operation operates identically to an insert - the only differences are that we remove a triplet from the final block we write (instead of adding one), and that we do not have to worry about expansions.

This means that a delete will also run in anywhere from 72 to 96 ms, again depending on whether secondary collections are present.

4.2.3 Search

To perform a find when a primary storage group is present, we must read only the 100 blocks in the group, taking just 18 ms.

When a secondary collection is present, we have to read the secondary collection header, as well as some number n of secondary groups. Reading the header takes 12 ms, and the secondary groups takes 18 ms each. The amount of time a secondary collection search takes is given by this equation:

$$12 + 18n \,\text{ms}, n \geq 1 \tag{7}$$

For the example workloads, the relationship tag is always quite common, and so n is small, ≤ 2.

4.3 Flickr++ Workload Analysis

Flickr++ is a photo-sharing application in which users can upload photos, organizing them both by albums and by tags.

4.3.1 Adding or Deleting an Image

To add an image to album, we must perform three inserts, taking anywhere from $72 \times 3 = 216$ ms to $96 \times 3 = 286$ ms.

4.3.2 Tagging an Image

To tag an image, we must perform a single insert, taking anywhere from 72 ms to 96 ms.

4.3.3 Finding Images With a Specific Tag

Finding images with a specific tag will run in 18 ms for tags that are in a primary collection, or 30 ms otherwise (due to the required secondary group header lookup).

4.3.4 Looking Up All a User's Images

This operation's characteristics are the same as finding images with a specific tag: 18 ms for users with less than approximately 8000 images, or 30 ms otherwise.

4.4 Library Workload Analysis

4.4.1 Load

Each book will require four inserts ("istype", "title", "publisher", and "author"), and each author will require three inserts ("istype", "name" and "affiliation").

Each insert will take between 72 and 96 ms.

4.4.2 Books of a Given Title or Publisher

These cases are identical to the case described above, of looking up all a user's images: 18 ms for titles and publishers in primary collections, and 30 ms otherwise.

4.4.3 Books by a Given Author

To find books by a given author, we must perform two searches, each taking 18 to 30 ms.

This gives a total range from 36 to 60 ms.

4.4.4 Books from an Author from a Particular Institution

To find books by a given author from a particular institution, we must perform three searches, each taking 18 to 30 ms.

This gives a total range from 54 to 90 ms.

4.5 Problems and Limitations

4.5.1 Rare Relationships

There is a potential problem with the design that is worth nothing. If there exists a triplet in a secondary collection with a unique relationship relative to all others in that secondary collection, the *Search* procedure may have to iterate through every secondary storage group if a search of the following pattern is requested.

```
(subject, unique-relationship, *)
```

The problem occurs because the system does not provide any kind of indexing based on the relationship field value. It is illustrated in figure 3. The problem also exists for queries in which the object is known but the subject is not, and those in which both the subject and object fields are wildcards. It

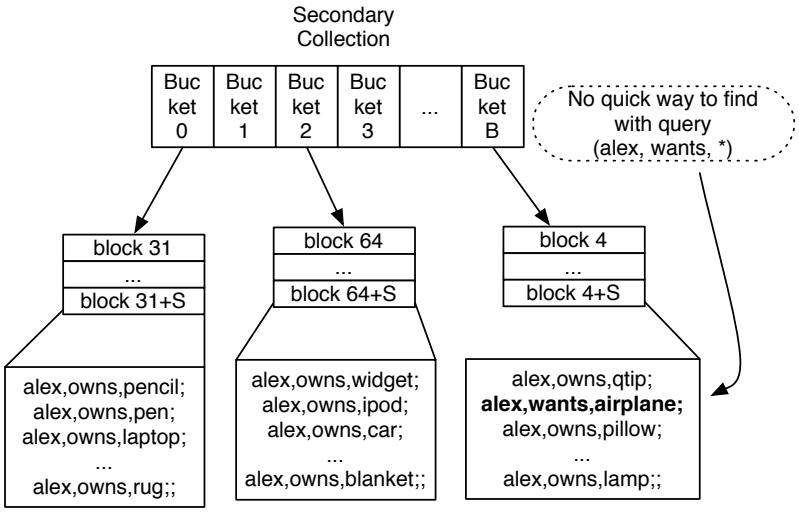

Figure 3: Illustration of problem with searching for rare tags in secondary collections.

is not an issue on insert or delete, because in those cases all triplet values are known, and the secondary hash table can be used to pinpoint the appropriate group to look in.

None of the example workloads in the design project encounter this problem, but one can imagine one that would.

5 Conclusion

The obvious tradeoff in this design was the decision to store two separate copies of the data set. This allows the search operation to run consistently quickly, regardless of what kind of search is being performed, at the expense of total storage capacity and insert/delete speed.

Additionally, the decision to use hash tables instead of B-trees or some other kind of variable-depth structure allows all operations in the workloads to run in $O(1)$ (amortized) time, with relatively low constant factors. Contrast this with other data structures, which may provide at best $O(log(n))$ performance. The flat nature of the data set makes this arrangement possible.

Using secondary hash tables allows us, at the expense of some complexity, to handle subject or object field values that occur any number of times without compromising our constant-time performance. Without secondary tables, the storage group associated with the bucket that "flower" maps to in the Flickr++ application might have one million triplets in it.

To support situations like this without two-level hashing, we would have had to implement variable-length or chained storage groups. This would cause

problems on insertion and deletion, since we would have to brute-force search through hundreds of thousands of blocks in order to check for duplicates on insertion, deletion of a triplet, or while performing certain types of search.

Variable-length storage groups have another problem, in that they introduce the problem of fragmentation. By setting all storage groups to a fixed length (B blocks), they fit together neatly, making allocation and deallocation of contigous blocks trivial.

Finally, as noted in section 4.5, the choice to only organize the data by subject and object limits our execution speed on certain search operations that involve uncommon relationship values. Solving this problem within this design framework would likely require storing additional copies of the data set on disk. This would compromise our storage capacity, increase our insert and delete times, and greatly increase the complexity of the system design in general. Because these types of searches are not used in the example workloads, and are likely to be fairly uncommon in practice, solving this problem does not seem worth the cost.

With secondary hash tables in place, the system's scalability looks good - a back of the envelope shows that we can store about eight billion triplets, all while maintaining the same constant-time performance characteristics exhibited at one billion. And, while unmentioned in the performance analysis, the ability to cache several million secondary collection headers should keep all performance figures on the low end of the estimated ranges.

All in all, this design provides a system that fully satisfies the requirements of the first 6.033 design project, without too many unpleasant characteristics, and many pleasant ones. If need be, it could be implemented exactly as described here to serve the workloads described in the assignment - no further refinement is required.

[0] 2555 words - sorry!