Effective storage of

Real time streaming data

Project Overview

A real time processing system is able to keep taking input continuously, do the operations as per requests, like store,process,read etc. and return the output. Good examples of real time data processing systems maybe in ATM machines, Traffic Signals, and modern computer Systems. This is different from batch processing data, in which first a batch (or packets of data) was collected and then processed. The processed data was then released. Real time data processing is also known as stream processing.
To process the data it is also important to store the data. So trying to implement a new algorithm which could effectively use all the resources and utilize the maximum of whats given. One such data structure can be LSM trees .

About LSM Trees

Using LSM Trees can be very useful in the effective storage of real-time data.
LOG Structured Merge Trees. This method will be effective if there is a write-intensive application rather than a read-intensive application.
Conventionally there are two levels in LSM trees that are SSD/HDD and mem table.
The data can be written in two ways. One is by directly hitting the disk which will be a very heavy option when it comes to traffic. The other is using a cache type memory which is memtable.

Memtable

The data is stored in a sorted manner, which will reduce the time complexity for reading from O(n) to O(log(n)). As soon as the memtable gets full, the data is flushed in the main memory and the memtable gets empty for further entries.

Sorted string tables (SST)

When the data is flushed from the memtable into the main memory it is stored as sorted string tables for a quick lookup.

Main memory (HDD/SSD)

The main memory consists of all the sorted string tables. If there is a cache miss then the query will be forwarded to the main memory. If there are k sst with each table having a size of n then the read time complexity will become k*log(n).

Let us say…

there is a request to write a new record, key=Walter,value=52. So first this will be inserted in the memtable. Them comes the request to write a record, key=Adam and value= 45. So Memtable will insert this record into it, in alphabetical order so that it is sorted. So now the memtable contains Adam,45 and Walter,52. If there is a third request to write key=Better,value=32 then the memtable will have Adam,45 ; Betty 32 and Walter 52. This is the alphabetical order. Now Lets say the maximum size of the memtable is 3. So this data will be flushed into the Disk(or SSD/HDD). If there is a read request then it will first search in the memtable, and if the key is present there it will return the value from the memtable saving the time to search in the disk. Because it is sorted, it can be searched in O(logN) time rather than O(N) time which is a huge improvement.
When the requested data is not in memtable then it goes into the disk and iteratively searches every SSTable and tries to find the data. This will take O(KlogN) time where K is the number of SSTables present in the Disk and N is size of the SSTable.

The problem with the above approach is duplication of data. The above approach doesn't update the value for the key, but instead appends the key and its value in the table. So if the old data is first found instead of the new data, then it will return the old data, which is a problem. To overcome this we will use a compactor.

Compactor

The above problem can be solved if the SSTables are periodically merged and updating the value for the keys as per the timestamp into a single SSTable.The merge is done in such a way that the keys are always sorted. Although merging will take its own time, but in the end updated values can be found quicker. This will solve the problem of duplication of data. Also this will remove the redundant data and clear that memory too. By compacting the data this way, we can improve the read time complexity from O(K*logN) to O(log(N*K)) which is a huge performance improvement.

Other problem which comes now is that when the number of sst increases a lot the time complexity will increase, specially in case of larger datas. To overcome this we will use bloom filter.

Bloom filter

Bloom Filter function is a probabilistic function which returns either 0 or 1. We pass in the function a key, and the bloom filter function returns either 0 or 1. 0 means that the key is not present in the SSTable and 1 means that the key is present in the SSTable. BloomFilter takes O(1) time to return the value which is constant. So it is very efficient. But the problem with the bloom filter function is that it returns 1 but the key is not present in the SSTable. If the function returns 0 then the key is definitely not present in the SSTable. The accuracy of bloom filter function is very high so it is a huge help in using with SSTables.

Our suggested improvements…

In real time applications, the requests will arrive on the application periodically. It will be the duty of the application to process them and answer them accordingly. So this application can basically have a small segment of memory in which it will store the data, and act like a disk. Another small portion of memory could act like a memtable. So this way we will be able to store the data effectively.
Basically, it is not very much likely that the data that is being entered will be queried right now. The memtable consists of the entries that are currently being fed. If we can come up with one other cache like memory that stores the entries that are likely to be searched for then we can gain a much higher cache hit rate.

1) Assigning Time stamp

Any data that will be entered will now be given a timestamp as a header attached to it. The cache will now consist of the entries that are being searched at the latest. After the timestamp expires the entry will be removed from the cache and when the cache is full the entry with the oldest time stamp will be removed from the cache to make space for the new entry.

Another thing that is to be taken care of is the task of the compactor. In the current algorithms the compactor delete and merges the entries in a duration of 30mins. This thing can be optimised by using other kind of update rate..

2) Compactor optimisation

Instead of clearing the duplicate entries every 30mins, it will be better to clear and merge the tables when the traffic on the database server is low and atleast under 30mins. By this the reads that are being done will most likely access the updated database.

Exploiting the bloom filter in a better way can help us save a lot of time.

3) Exploiting Bloom Filter Probability

Instead of taking 0-1 from the bloom filter, we can take a continuous value. By this we can search the tables which have the highest probability of having the entry. So we can save a lot of time searching unnecessary tables.

Problems faced by cache:

- Most processors used are multicore
- The current shared memory resources are a source of very high unpredictability of worst case execution time (WCET)
- The contention for allocation of memory block for the last line of cache is major source of unpredictability
- Sometimes a task can evict the cached block of another task

scheduled on same CPU
- Cache pollution
- Two tasks running on different cores evict each other in the shared cache
- These causes delay as the data are no longer in cache and they have to be fetched from memory or disk

Solving the problem:

- The previous problem is very common. There exists some solutions
- Some of them involve blocking the cache line from another core , so that they are not evicted by the other core
- This does not solve the problem completely as they add delays to the working
- We can solve the problem by proposing an integrated solution.
- This can be done by using memory profiling and colored lockdown.

Memory profiling

Collect all the virtual addresses accessed during the task execution.
Create a list of all the memory pages sorted by the number of times they were all accessed.
Record the list of memory regions assigned by the kernel during the execution.
Record the memory areas assigned by the kernel under normal conditions.
link together the list of memory regions assigned during the profiling (obtained in step 3) with those owned by the process under normal conditions(step 4)
determine in which region of the list obtained in step 4 each entry of the list obtained in step 2 falls, according to the mapping determined in step 5;
generate the execution-independent memory profile

Colored lockdown

This occurs in two phases:

1 . Startup colour/way assignment

The system assigns to every hot memory page of every considered task a color and a way number, depending on the cache parameters and the available blocks

2. Dynamic lockdown

The system prefetches and locks in the last level of cache all (or a portion of) the hot memory areas of a given task

Future work:

As directed by you in the presentation we will be looking to do the further work.

By:
Asif Ahmed (B17EE013)
Anshul kulhari (B17ME016)
Mihir sakaria (B18CS034)