International Journal of Scientific Research and Engineering Development-– Volume 4 Issue 4, July- Aug 2021

Available at www.ijsred.com

RESEARCH ARTICLE OPEN ACCESS

Computing Set Difference of Streaming Sets

Vinanth S Bharadwaj*, Dr. Nagaraj G Cholli**

*(ISE, RV College of Engineering, Bengaluru

Email: vinanthsb.is17@rvce.edu.in)

** (ISE, RV College of Engineering, Bengaluru

Email: nagaraj.cholli@rvce.edu.in)

----------**********----------

Abstract:

In this paper we discuss how to calculate the set difference of two large streaming data sets. A lot of processes generate streams of data. Streams are a continuous set of data which come at speed which is its characteristic rate. There are many problems which arise during processing of streams. Often there will not be enough space in the main memory to completely accommodate a stream. Hence streams must be processed on the go, and any time delay in servicing an item in the stream, the buffer overflows leading to inconsistent output. We propose an algorithm in this paper which compares two such streams to compute the set difference with minimum loss of stream items and we propose modifications when there are additional constraints involved. This method can not only be used for data streams but also for use cases when there is a constraint on the memory used.

Keywords —Stream, Set delta difference, HashMap.

----------**********----------

I. INTRODUCTION

Various processes, sensors and API calls generate streams of data. The term stream is used to describe continuous, never-ending data streams with no beginning or end, that provide a constant feed of data that can be utilized/acted upon without needing to be stored in memory first. Similarly, data streams are generated by all types of sources, in various formats and volumes. From applications, networking devices, and server log files, to website activity, banking transactions, and location data, they can all be aggregated to seamlessly gather real-time information and analytic from a single source of truth. We had a use case where we need to constantly compare two such streams and find their difference. But the problem lies in the memory usage. If the whole stream would fit inside the memory, then we can use sorting or hash maps directly to compute the delta difference. But problem arises when there is not enough memory to hold the stream or even a part of the stream in the memory. For this to happen the stream need not be very big data, but when we want to compare the streams when there are other processes running on the CPU as well leaving very small amount of main memory, the proposed system works with all these constraints.

II. NEED FOR SUCH ALGORITHM

Some data comes in the form of an unending stream of occurrences. To accomplish batch processing, we would have to first store the data, then pause data collection for a period before processing it. Then there is a problem of aggregation of results across batches. Streaming, on the other hand, easily and naturally accommodates never-ending data streams. We can identify a sequence or patterns, evaluate outcomes, and compare several streaming data sets. Very often the data is so huge or due to any other hardware limitations, the data cannot be fully stored inside the memory to be batch processed. Stream processing does not require the whole stream to be completely inside the memory, but rather it processes the data on the go. But one limitation of such stream processing is when the stream rate crosses the processing rate. This would result in losing some of the items of the stream when the working memory queue overflows. So, there is need for a comparison algorithm which processes the

streams, so that minimal items are lost and there is very tight bound on the memory usage as well.

III. BASIC DESIGN OF THE PROPOSED SYSTEM

A straightforward approach would be to load the complete stream onto memory and then compare them using a HashMap by adding and removing elements. But this requires complete stream in the memory and also, we have to wait till the stream has ended to process it and they cannot be processed on the Consider two sources A and B, who are sending streams of data whose delta difference we need to find. Mathematically speaking we need to compute A-B and B-A,where'-' operator specifies set difference.

We spawn two separate threads, one to get the input from each of the source. We have a HashMap which is shared between the two threads so that they update the same memory to find the set difference. Each of the threads have their own queue which is the buffer space for the stream. We will be using shared mutex for the thread synchronization for shared memory reads and writes. The main parent process creates two threads to process data from respective streams. The thread keeps on getting the next item from the queue and then if the value is already present in the HashMap, then the value is removed as that item cannot be in the difference since it is present in both sets. If the value is not present in the HashMap then we add it to the hash map. When the parent process stops these threads later when the streams have closed, the remaining items in the HashMap will be the set difference or mathematically it will be A-B + B-A where'+' operator denotes the set union and'-' denotes set difference. A single bit can be added to the key of the HashMap to maintain which of the streams added that item, so we can later get each of the set difference i.e. A-B and B-A separately. This approach may pose a few problems, like starvation, or over memory usage when there is not much intersection in the two sets.

IV. ALGORITHM

```
STREAMSETDIFFERENCE(PROCESSSTREAMDATAFROMA
())
1ThreadA= THREAD(PROCESSSTREAMDATAFROMA())
2 ThreadB = THREAD(PROCESSSTREAMDATAFROMB())
3       for i = 0 to HashMap..SIZE()
4       PRINT(Hashmap[i])

PROCESSSTREAMDATAFROMA()
1 while true
2       data = GETNEXTITEMFROMQUEUEA()
3               if data! = NULL
4               sharedMutex.LOCK()
5               if HashMap.FIND(data)! =
                HashMap.END()
6               HashMap.ERASE(data)
7               else
8               HashMap.ADD(data)
9               sharedMutex.UNLOCK()
```

A similar algorithm runs on the other thread as well and the mutex lock does the synchronization between the two threads. The time complexity of the algorithm would come to O(n) where n is the total size of the streaming data if the HashMap is perfect. But since hashing won't be perfect and a sparse HashMap would only result in increased memory consumption, we use a binary search tree in place of the HashMap, then the time complexity would be O(nlogm),where m is the number of elements present in the HashMap at a given time, as insertion , deletion and getting the value of the key takes O(logn) time.

The sources need not be two at all the times. We might have multiple sources of streams and we might have to compute the difference between them. If we need to compute only the items which are not present in all the sources i.e. $A1 \cup A2 \cup A3 \cup ... \cup An-1 \cup An - A1 \cap A2 \cap A3 \cap ... \cap An-1 \cap An$, where n is the number of sources. This can be computed easily by tweaking the algorithm a bit, increment the value of key in HashMap for every time the, which keeps the count of how many sources the key has been present in.Once this count becomes equal to the number of available streams sources we can erase the data from the HashMap.


```
ProcessStreamDataFromA()
1   while true
2       data = getNextItemFromQueueA()
3                   if data! = NULL
4                   sharedMutex.lock()
5                   if HashMap.find(data)! =
                    HashMap.end()
6                   if HashMap.value(data) ==
                    Sources
7                   HashMap.erase(data)
8                   else
9                   HashMap[data]+ = 1
10                  else
11                  HashMap.add(data)
12                  sharedMutex.unlock()
```

But if we need to calculate, pairwise missing items i.e.

$A_i - A_j \cup A_j - A_i$ for all $1 \leq i,j \leq n$ and $i \neq j$ we would require ${}^nC_2$hash maps to maintain pairwise missing items. Each time the function has to update all the hash maps, that it is a part of and also remove any item from all the hash maps it is part of. This process has to be atomic, as in all the hash maps have to updated or none. The threads be synchronized using only one mutex not allowing multiple threads to execute concurrently inside the critical section. By following this method all the hash maps are updated or none of them are updated. Let's assume we have another function UpdateAllTheCorrespondingHashmaps(INT SourceId) which takes theradId as input and updates all the corresponding hash maps. This function would need a shared array of hash maps and it updates the each HashMap the sourceId is associated to.

The array would have hash maps in a order. For example if there are four sources namely A,B,C,D then the array would have hash maps in order like - AB,AC,AD,BC,BD,CD. This is easier to index while updating all the corresponding hash maps for a thread.GetTheIndices() returns all the indices that correspond to the hash maps of the *threadId*.Care should be taken to avoid deadlocks using techniquesdescribed in the paper to avoid deadlocks using resource allocation graphs [2].

```
UpdateAllTheCorrespondingHashmaps(SourceId,
data)
1   A[1..m] = GetTheIndices(SourceId)
2               for i= 0 to m
3               if ArrayHashMap[i].find(data)! =
4               ArrayHashMap[i].end()
5               if ArrayHashMap[i].value(data)
                == Sources
6               ArrayHashMap[i].erase(data)
7               else
8               ArrayHashMap[i][data]+ = 1
9               else
10              ArrayHashMap[i].add(data)
```

```
ProcessStreamDataFromA()
1   while true
2           data = GetNextItemFromQueueA()
3           if data! = NULL
4   sharedMutex.lock()
5   UpdateAllTheCorrespondingHAshmaps(
6           ThreadId
7           ,data)
8           sharedMutex.UNLOCK()
```

This code functions well,but we might miss out few data since updating single element takes a lot of time. A single mutex for all the hashmaps also reduces multiprocessing level as to update ${}^0n - {}^10$ hashmaps we lock all the ${}^nC_2$hashmaps. We can work around this by having an array of mutex one for each hashmap ,but it would mostly result in the thread waiting for some hashmaps,while other hashmaps are updated, and the hashmaps wouldn't be consistent.If data consistency is not a significant factor for the use case we can add a array of mutexs and a timer within which if the thread doesn't get access to the hashmap, it would not update that particular hashmap. Having such array of mutexes also causes starvation. Starvation is defined to be an infinite or unbounded delay, and classified in terms of its cause and the methods used for its control[3].This would take up most of the memory and all the other streams queue would lose data without the data being serviced. This can be

avoided by applying a constraint as to how many times can a particular thread lock the array of hashmaps in a particular time range. This value can be assigned to each of the streams based on the current stream rate , which can be calculated dynamically by *DynamicStreamRate=TotalItemCount/TotalTimeSinceStart*. The starvation can also be avoided by applying constraint to number of consecutive times the thread lock the resource. A wrapper for mutex can also be developed to customize the control even more. For example, if *k* threads among n threads are waiting for the mutex, the mutex can be assigned to the thread whose *bufferQueue* has a greater number of items. This could be a problem if the stream rates are different, as in even though a

thread's queue has small number of items, but has a substantial *dynamicStreamRate*, the thread's queue will be full even before the queues of other threads. So the priority can be assigned on the basis of

$$Priority = \frac{MaxSizeOfQueue - CurrentSizeOfQueue}{DynamicStreamRate}$$

The thread who has the lowest priority gets picked first and hence ensures that we do not leave out profuse number of items of the stream. This paper is a enhancement over [1] which discusses about the finding the delta difference between the sets based on finding the longest common sub sequence between the sets. For this to happen the sets must be in same order. Our approach works well even when the sets are not in the same order.

V. LIMITATIONS OF THE ALGORITHM

- Every stream has a rate at which it generates a new item, which is known as stream rate. If the rates of the sources have a vast difference, then one source keeps on adding new items to the hashmap, and the hashmap size grows since there isn't as many items as possible getting removed from the Hashmap, which would result in the Hashmap holding the complete set A or B, defeating the sole purpose of using stream processing.
- Using an array of mutexes can lead to deadlock and starvation if not handled properly.

VI. CONCLUSION AND FUTURE WORK

We have hence outlined the algorithm to compare data from multiple stream sources when there is memory constraint involved. The algorithm is simple but care should be taken about locking and unlocking mutexes so that all the streams are given equal access to the shared resource. We can also develop a hash function by analyzing stream over a period of time, so that the search and update time is reduced to $O(1)$ while not taking a lot of memory as well. The stream can also be analyzed, and the results can be used when we need to drop the items from the queue. An aggregate frequency of the item over a period can be found and probability of leaving the item can be computed by that. Adding such techniques would streamline the flow and also reduces the number of errors when we report the missing items from all the streams. The algorithm can be used for not only for stream data but also for use cases where the data doesn't fit completely inside the memory. The proposed algorithm hence provides a solution to compute the set difference of two data streams and works best when the main memory is limited.

REFERENCES

[1] Johan Arvidsson. "Finding delta difference in large data sets". In: 2019.
[2] V. Geetha and N. Sreenath. "Preventing deadlocks and starvation in distributed object oriented systems". In: Computers Electrical Engineering 39.2 (2013), pp. 582– 595. ISSN: 0045-7906. DOI: https : / / doi . org / 10 . 1016 / j . compeleceng . 2012 . 12 . 025. URL: https : / / www . sciencedirect . com / science / article / pii / S0045790613000050.
[3] GERTRUDE NEUMAN LEVINE. "THE CONTROL OF STARVATION". In: International Journal of General Systems 15.2 (1989), pp. 113–127. DOI: 10 . 1080 / 03081078908935036. eprint: https://doi.org/10.1080/ 03081078908935036. URL: https://doi.org/10.1080/ 03081078908935036.

