Abstract

In relational database management systems, aggregation with group by fields requires keeping all intermediate results around in order to repeatedly update them when an input tuple with the same grouping key is ingested. Hence a common way to compute such aggregation is to use a hash table to store intermediate results, by having the group keys as the keys of the table and the aggregate results as the values, until all data is processed. This approach is used by the Myria system, a cloud-based big data management and analytics system that has been developed and actively improved by the University of Washington Database Group. In this project, we present the design, implementation, and evaluation of a streaming based aggregate with the assumption that input data is sorted on the grouping fields. A streaming aggregate has two important benefits: (1) use constant size data structure to manage intermediate result while processing, and (2) return the result as soon as generated, regardless of the remaining input data. The benefit would be to save memory use by removing the hash table for storing all intermediate results. It would also speed up the computation because results are returned incrementally, although the time to process the whole input is expected to be similar to the hash based aggregate. We then compare the computation measurements between the streaming aggregate and hash based aggregate that is currently used by Myria.

In addition to the streaming aggregate, we also change the limit operator algorithm for the system. Limit is the last operator of a database call chain before returning the result of a database query to user. It limits the number of tuples returned with the user-specified limit. The selection of the tuples is based on the previous operation, which is independent of the limit operator. As soon as the desired number of tuples have been returned, the operator closes its input stream and finishes the process.
Contents

1. Introduction .................................................................................................................3

2. Design and Implementation .........................................................................................4
   2.1. Streaming Aggregate
   2.2. Limit Operator

3. Evaluation ....................................................................................................................7
   3.1. Single group-by with distinct input
   3.2. Single group-by with repeating input
   3.3. Multi group-by with distinct input
   3.4. Multi group-by with Repeating input

4. Discussion ..................................................................................................................14
   4.1. Limitation of fixed size data structures
   4.2. Cost of sorted input for streaming functions

5. Conclusion .................................................................................................................15

6. Acknowledgement .....................................................................................................16

7. Reference ..................................................................................................................16
1 Introduction

Myria is a distributed database system as an online data management and analytics service. It aims for both big data availability and trouble-free usability. One of the features that outstands Myria from other big data management systems is its own programming language, called Myrial. Myrial is like a hybrid of SQL and Pig-Latin, which recognizes more English-like keywords, such as scan and store, in user written queries. When I first started to interact with Myria, I was able to easily write my own queries to generate my own data without the hassle of adapting a new interface or learning a new programming language. Another highlight of the system is that it does not require any configuration or installation of new software. Myria is accessible through a web interface, where user can query over his or her own uploaded data. User does not need to configure how masters assign work to its workers, how data should be partitioned, or any of those details that require knowledge in databases.

In Figure 1 (adapted from [1]) shows the overall architecture of the Myria system.

Myria currently supports the following aggregate functions, COUNT, SUM, MIN, MAX, AVERAGE, STANDARD DEVIATION, with hash based aggregators for single group by aggregate and multi group by aggregate. Single group by aggregate is the aggregation with a single column as the grouping key. Similarly, multi group by aggregate is the aggregation with multiple fields as its grouping key. In both cases, the aggregate operator uses a hash table to keep all intermediate records of aggregation, by having the grouping keys as the hash keys and the aggregate states as the hashed values. In this manner, aggregate can be
computed with a single path, but it requires memory to keep the hash table available throughout its process. In this project, I am adding streaming aggregate that accepts one or more fields specified as group by fields. Streaming aggregate eliminates the need of the hash table, which size depends of the input data. It only requires a constant size structure to manage intermediate result. The algorithm is explained in detail in Section 2. There is a key assumption under the algorithm of streaming aggregate, that input is sorted on grouping keys. This is an expensive assumption to make, because sorting the whole input data set is by itself expensive, especially for big data. In addition, the system will not know which fields will serve as grouping keys until user types a query, so the sorting must take place as part of the query plan. Nonetheless, streaming aggregate must follow the assumption the input is sorted on the grouping fields because once a tuple is processed, the algorithm does not keep the reference to previous values in any form. It can only access and modify aggregate result for the current grouping key, so all tuples with the same grouping key must arrive altogether. If this condition is met, by having streaming aggregate the space and time required to perform aggregation with the same size of input are expected to reduce.

2 Design and Implementation

In this section, we present the design and implementation of the new streaming aggregate and the limit operator.

2.1 Streaming Aggregate

The design of the streaming aggregate focuses on two major characteristics. One is to use constant size data structure to manage intermediate aggregate state. The required data to keep are current grouping keys, current aggregate states associated with the given grouping keys, and the result buffer for finished aggregate groups. All of them can be managed with constant size data structures by repeatedly updating a single data structure. Another characteristic is to return result tuple batch as soon as a filled one is generated. This gives two benefits, one is that the result batches can be returned to the next operator in streaming manner before all input has been processed, hence fast return time. Another is that the result buffer will be at most one tuple batch large because result tuples will not be stacked before returning.
Figure 2.1.1 demonstrates the data structures required to compute aggregates based on different algorithms. In Figure 2.1.1 (a), the hash based aggregate allocates a hash table storing all aggregation states for its input tuples. The size of the hash table grows as the number of distinct grouping keys grow. Figure 2.1.1 (b) demonstrates the data structures used by streaming aggregate. The streaming aggregate only requires a tuple sized data structure to keep the current grouping keys, a single number to keep track of the current aggregate result, and a tuple buffer that is at most single tuple batch large to store the finished aggregate results. All data structure sizes can be bounded regardless the size or schema of the input.

Figure 2.1.2 is the pseudo-code for the fetchNextReady() method in the streaming aggregate class. It returns the batch worth of result tuples. As demonstrated in the design of this operator, the streaming aggregate returns the result batch as soon as there is a filled batch. There are three cases for this method to return. The first case is that when the operator has finished processing all input tuples. This case will be encountered only once, immediately after the input has been exhausted. In this case, the operator returns null to indicate the end of stream for its next operator, if any. If there are input tuples to process, the operator will first find the position of the tuple to process next. Then it checks if the grouping column values are the same as the grouping key stored in the global data structure. If it is, the operator needs to update the aggregate state with the current grouping key. If it is not, the operator first flushes the current aggregate keys and state as finalized result to the result buffer. Then it updates the current grouping key to the grouping column values of the current tuple, and reinitializes the aggregate state. When a new result tuple is flushed to the result buffer, the operator checks if the result buffer is full as the result of adding a new tuple. If it is full, the operator returns the filled tuple batch and pauses computation until next fetchNextReady()
fetchNextReady():
If child is end-of-stream:
    return null
If first time in this call:
    set current tuple batch to first tuple batch from child operator
    get tuple from first row in current tuple batch
    set current group key to be the first tuple’s group field value
While current tuple batch not null:
    for each tuple in current tuple batch:
        if current group key != current tuple’s group field value:
            add current aggregate state to result buffer
            set current group key to be the current tuple’s group field value
            reinitialize aggregate state
            if result buffer has filled tuple batch:
                return filled batch
            update current aggregate state to include current tuple’s agg fields
            get next tuple batch from child operator
    If child is end-of-stream:
        add current aggregate state to result buffer
        return filled or unfilled result batch

Figure 2.1.2 Pseudocode for fetchNextReady() of Streaming Aggregate

call is made. Notice that the actual aggregation with the new grouping key and aggregate state is computed with the following call of fetchNextReady(). The only case when an unfilled tuple batch can be returned as the result is when the operator finishes processing all input data. The operator returns the result batch when the input stream reaches the end of stream.
A streaming aggregate needs to keep the following information in memory: group key schema, the actual aggregators that do the work, the current tuple batch, current row in the current tuple batch, current grouping key, result buffer, and the current aggregate state. They are all stored as global field so that the previous state of the aggregate can be recorded when the result generating process is paused. This is because if a filled result batch is built up, even if in the middle of processing input tuple batches, fetchNextReady() will break and return the result batch to its caller. Aggregation will resume when fetchNextReady() is called again, hence the streaming aggregate operator must know what is the last point before the pause and be able to resume from that point. When all tuple batches from the child has been processed, which terminates the while loop, we need to return the last few tuples in the result buffer even if they do not fill up a tuple batch. This is the only case when it is allowed to return an unfilled tuple batch as result in order to maintain the least possible communication to other operators.
There can also be the case when child’s next ready tuple is null but child is not end-of-stream. In those cases, it will simply wait for further input stream to arrive to resume aggregation without clearing the streaming states.
Since the streaming aggregate will not stack filled result batches, if a call to fetchNextReady() sees its child as end-of-stream as the first thing, it knows that
this child has been all processed and the corresponding aggregate results are all
returned. At that point, it can simply return null, indicating end-of-stream of this
operator too.
For each incoming tuple, the streaming aggregate only keeps the group field
values of the tuple and the aggregate state (the repeatedly updated aggregation
values). It will not know if the same grouping key as the current current tuple is
encountered before or will come in the future, unless all of them arrive altogether.

2.2 Limit Operator

Limit operator is straightforward, it returns the limit number of tuples to its next
operator. Limit is typically the last operator in the database call chain, because it
does not control the order the tuples arrive. Instead, the order is determined by the
previous operators, such as ASC, DESC, or most of the case unordered. The
original limit operator does its job, but internally it is inefficient because it simply
drops the remainder of tuples and exhausts its child operator. I modify it such that
it closes the child operator when limit is reached, and returns end-of-stream to its
caller operator. It is safe to close the child because it is known that no more tuples
would be needed.

3 Evaluation

In this section, we compare the runtime and memory consumption of the original
hash based aggregate and the new streaming aggregate by running the same query
with the same input size. The experiments are performed by executing java source
code on a single machine. The java heap memory limit is set to 3.5 GB.
Measurements are taken with the YourKit Java Profiler. There are four
configurations for this evaluation, as the combinations of aggregate schema and
input data. For the input data, there are the case when all tuples in input have
distinct grouping key values, and the case when input tuples have some repeating
grouping key values. For the result, we separate the case when a single group by
field is present and when multiple group by fields are present.
The input schema for single group by aggregate experiments is in Figure 3.0.1. The first column is used for grouping, and aggregate over the second column. Similarly, for multi group by aggregate, the first two columns serve as grouping keys and the aggregate over the third column. We compute a simple sum query for the experiments. For all experiments, input is sorted on grouping fields.

3.1 Single group-by with distinct input

First pair of graphs as shown in Figure 3.1.1 is the result of running the above query with 20,000,000 distinct grouping key tuples. The result is the same size as the input tuple, in the same order as the input tuple. In the graph, the x-axis represents the uptime for running the query in Figure 3.0.2, and the y-axis represents the total memory allocated and used. The red line in the graph is the allocated memory space for the process, the green line is the used Eden Space in addition to the Survivor Space and the Old Generation. The purple line is the sum of the Survivor Space and the Old Generation. The blue line is the Old Generation alone. We focus on the behavior of the Eden Space, as it represents the measurement of memory use by the process.

The Eden Space measurements shows a clear trend in increasing in memory for hash based aggregate. This is because the aggregate builds a hash table for each grouping key until the end of file. The memory allocated (y-axis) increases as the number of tuples ingested increases. It also takes much longer time (x-axis) to finish computing the result. This is because allocating a new entry in the hash table for each of the incoming tuple is time consuming. After about 40 seconds, the memory does not grow anymore, which indicates that the aggregate has finished processing all input tuples. A hash table for all grouping keys is built, and from that point can generate result batches to return. All result batches are generated at the same time and stored, and each call to fetchNextReady() will return one batch at a time from the prepared result batch buffer. The heap memory
allocated is about 3.5 GB, which almost hits the limit for the configuration these tests are run.
On the other hand, streaming aggregate uses about 0.5 GB of memory throughout its process. The intermediate result is kept using constant size data structure, which does not affect the heap memory largely. The memory is allocated mostly for the result buffer, but it is known that the result buffer will not exceed 10,000 tuples either, hence the amount of memory used is almost a flat line. Also notice that the time to finish the same query is about 1/10 for streaming aggregate. Streaming aggregate does not need to wait for all input to be ingested neither stack up result buffer, hence the process is much smoother.

3.2 Single group-by with repeating input

However, the above input is an extreme case where all tuples are distinct. Now see a case when there are some tuples with repeated grouping key. The size of the hash table will be smaller, and therefore more input tuples can be processed. The memory bound is reached when constructing the hash table.

From Figure 3.2.2, we see the increasing trend in hash based aggregate. But notice that the hash based aggregate improves its performance significantly. The query is finished in 7.5 seconds with about 1 GB of memory used. There is an increasing trend in memory use, but less rapidly than previous case. It makes sense because hash based aggregate keeps all previous intermediate results. The aggregator is able to process a much larger input set because the size of hash table is not large. There are 6 million distinct grouping keys, which is less than previous input data set.
The streaming aggregate also finishes this query faster than the previous one, despite the larger input size. With repeating grouping keys, less copying of group field values and reinitialization of aggregate state will be performed, hence saving computation time. The amount of memory used are approximately the same as previous case, which indicates that the data structures and the result buffer used by streaming aggregate is the same size regardless the input size and format. The fact that streaming aggregate does not require additional space for larger input size allows it to take in any input size to perform aggregation. Of course, the runtime increases with respect to the input size, but the memory allocated is
always similar. It can run the above query with a 500 MB input and finish smoothly on a personal computer, while hash based aggregate runs out of memory for 60 MB input.

3.3 Multi group-by with distinct input

Now we will see the comparison between hash based multi group by aggregate and streaming aggregate. As in the single group by cases, we use a simple sum query to measure the runtime and memory consumption. The schema has two fields as grouping key and a single field to aggregate over.

Since each tuple now has three fields, a batch worth of input tuples is larger in size than the previous input set. The hashing is performed based on both gkey0 and gkey1 so the hash table is expected to take more memory and the entry allocating step is expected to take more time.

As shown in Figure 3.3.1, the trend of increase in memory with number of distinct tuples is observed in multi group by aggregate as well. An interesting fact to notice is that the runtime and memory use for aggregation of the three field tuples here are actually less than the results obtained from using two field tuples as input. I think this is simply because there are less tuples to process, which means that the size of the hash table is smaller and less hash entry allocation needs to be performed. This emphasizes that the bottleneck of hash based aggregate is the memory consumed by an eventually discarded hash table and the work required to add new entry to the table when a new group key is encountered. For streaming aggregate, the memory used is again about 0.5 GB, regardless the change in format of tuple. It finishes fast because the number of tuples to process is less.
3.4 Multi group-by with repeating input

This experiment uses input tuples with some repeating grouping values. The input data is as in Figure 3.4.1.

The hash based aggregate uses more memory as more input data are ingested. Although there are tuples with same group field coming one after another, the number of distinct grouping keys is large enough to cause the hash table to expand. Also the amount of input tuples is simply large that the aggregator needs to take time to finish its process. This is does not follow the performance pattern observed in the single group by aggregate with duplicate tuples. Here it not only uses more memory, but also doubles in the runtime. But it is still reasonable since the input size is so large.

Streaming aggregate is able to complete the same query in seven seconds. From the graph it is clearly shown that the memory used for the most of the process is flat. The amount of memory used is about 1 GB, which is larger than the previous experiments. One reason for this would be that a batch worth of tuples take more memory with the three field schema, so that building one full batch actually requires more space. Another thought is that because tuples are grouped together now to fill a result batch requires more than one input batch to be processed. The operator needs to read in more than one batch for each call of fetch next. If the finished input tuples are not discarded immediately, they would take some space as well while the operator works on the next batches.
In summary, for input that has all grouping keys distinct, it is clear that streaming aggregate excels hash based aggregate. Memory used by streaming aggregate increases when the aggregation is instantiated, mainly for allocating memory for its constant size global data structures, working input tuple batch, and the result buffer, but soon after it enters the phase where the memory use is flat. Because it returns the result in streaming manner, previously processed tuples do not have their reference in the operator. In addition, result buffer will not be stacked up because each call to fetch the next ready batch of aggregate results only computes
a single batch worth of result, and pauses the computation there. Computation
resumes only if another fetch next call is made, so there will not be any buffer for
holding non-finalized results. All these features of streaming aggregate makes it
runs fast and consumes less memory when processing the same size of input
compared to the hash based aggregate.
When there are reasonable amount of tuples with repeating group field values,
hash based aggregate is quite efficient as well. The streaming aggregate improves
its performance, too. Notice that regardless the schema or number of tuples in
input data set, the total memory used by streaming aggregate is always around 0.5
GB at most. From both space and time perspective, streaming aggregate excels
hash based aggregate in performance.

4 Discussion

4.1 Limitation of fixed size data structures

The streaming aggregate does not keep information about the passed input. As an
algorithm to perform aggregations, which are based on the whole input data set,
using the fixed size data structures is difficult and inconsistent. If the input tuples
that should be grouped together do not arrive in together, the streaming aggregate
could not know there is already an aggregate result with the same grouping key,
and it will treat the new tuple as a different group and generate incorrect
aggregate result. It is especially difficult for a distributed system like Myria to
maintain the groups of tuples to be assigned to arrive together. It not only requires
rehashing on the grouping fields of aggregate to get them to the same worker, but
also needs them to arrive in the sorted order such that all tuples with the same
group by value arrive together. It is impractical to control the stream that input
arrives after hashing, so it must be achieved by sorting the input on the group
fields. Sorting is an expensive operation, with more than linear runtime. And the
sorting can be pushed down to each worker to perform locally. Hence a trade off
has to be made between the hash based aggregate to perform everything in a
single pass and using sorting plus streaming aggregate to save memory space.

4.2 Cost of sorted input for streaming functions

Streaming aggregate requires input tuples are sorted on the grouping fields.
However, as stated in the previous section, it is impossible to sort the data with
respect to group by fields beforehand. The system will not know which fields will
be grouped by until user issues a query. Hence the grouping must be performed as
a part of the query plan, which is a real time computation for each query issued. It
requires one extra step to scan all input data and partition them with respect to
group field. The most efficient way to do this would be to use a hash partition.
But then, if a hash table based operation is required at this step, with the same hash key that would be used by hash based aggregate, there would be no benefit in both execution time and memory space using streaming aggregate. In contrast, hash based aggregate would be simpler by not requiring the extra step to group input tuples.

There are other techniques to perform in-memory sorting. Streaming aggregate can be combined with any of the sorting algorithms to satisfy its precondition of sorted input. Those algorithms might be slower than the hash based grouping, but if the system provides fast computation and little memory, the streaming aggregate would suit more for such environment.

Although it is shown that if the input is grouped, streaming aggregate has better performance than hash based aggregate, the high cost of grouped input makes the use case for streaming aggregate in real world very limited. To apply streaming aggregate to the actual Myria system requires few more tasks to be addressed.

5 Conclusion

In this project, we implemented streaming aggregate for potential substitute of hash based aggregate. It is thoroughly tested on the correctness and consistency of the algorithm using data with various schema, size, and contents to show that the algorithm can generate the same result as the hash based aggregate does.

Furthermore, detailed computation measurements are taken to be compared to the performance of hash based aggregate. From the experiments, we showed that streaming aggregate excels in both time and space usage for any input and query. The streaming aggregate finishes in about one tenth of the runtime compared to the original hash based aggregate when doing aggregation on all distinct data set. For the case where there are tuples with repeated group field values, where hash based aggregate can take advantage of, streaming aggregate finishes even faster.

Streaming aggregate keeps flat memory use because it uses constant size data structures to manage intermediate aggregate results. It also does not stack up result batches by returning filled result batch as soon as one is constructed. However streaming aggregate requires its data to be grouped or sorted on grouping keys of the aggregation before processing. This condition is difficult to meet in real world setting. No preprocessing can be done since the system cannot know the group fields until a query is issued. If the system include real time grouping as the call chain of the query, it could instead do the hash based aggregate to reduce the extra work of grouping. Some other possibilities to enforce grouping for streaming aggregate are discussed too, including adding in-memory sorting before doing the streaming aggregate. Streaming aggregate has significant advantage in computation efficiency over hash based aggregate. Further work regarding its precondition on data should be conducted for its general application to the Myria system.
6    Acknowledgement

I would like to thank my advisor, Magda Balazinska, to give me the opportunity to work on this project. It is my first time to work on a real world system that is published and used by users. Besides the work on the project itself, like writing code and setting up experiments, I learned the way to collaborate with others in a professional setting. I really appreciate this experience as an undergraduate student. I think the experience in this independent research sets me one step ahead of my fellow classmates as a software engineer. I also received a lot of help from the UW Database group members. All of them were helpful, motivating, committed, and smart. This project would not go as smoothly if I did not receive the help and guidance by them. I would have difficult time finding the start point of the project if I did not learn insightful knowledge about the Myria system and call chain. My algorithm would simply not work without the code review. Again, I really appreciate this opportunity to contribute to Myria and to work with the Database group.

The experiment measurements are obtained by executing Java source code with the YourKit Profiler enabled.

7    Reference