

Efficient Algorithms for Large-Scale Temporal Aggregation

Nihal Dindar

Spring Semester, 2010

Contents

1	Introduction	3
1.1	Problem Statement	3
1.2	About Paper	3
2	Small-Scale Aggregation	4
2.1	Balanced Tree Algorithm	4
2.2	Merge-Sort Aggregation	5
2.3	Discussion	5
3	Large-Scale Aggregation	6
3.1	Bucket Algorithm	6
3.2	Discussion	7
4	Parallel Bucket Algorithm	7
4.1	Discussion	8
5	Evaluation	8
5.1	Small-Scale Aggregation	8
5.2	Bucket Algorithm	9
5.3	Parallel Bucket Algorithm	10
6	Conclusion	11

1 Introduction

1.1 Problem Statement

Databases need to capture and query time-varying nature of the data for many applications such as data warehousing and data mining. This need has been recognized by different areas of database research such as temporal databases, which has proposed temporal data models and language models. Temporal aggregations are an important feature of temporal languages, which enables to have a summary of time-varying information.

The key challenge of temporal aggregations is temporal grouping where the time line is partitioned over time, and tuples are grouped over these partitions. There are two types of temporal grouping: span grouping and instant grouping. In span grouping, the size of the time partitions are constant such as weeks, months. On the other hand, in instant grouping the size of the partitions is based on time attribute of the data and it varies. This small seemed difference makes the processing of instant grouping more complex, since simply considering each tuple in order in a sorted-by-time relation will not be sufficient. For example, in order to compute the maximum stock exchange price of technology companies based on time-intervals, the time intervals can not be considered as fixed and each tuple has to be examined, and overlapping of the validity intervals has to be investigated. Table 1 shows a sample Stock Exchange Table.

Several algorithms have been proposed to compute the temporal aggregates efficiently. The solutions are mostly using variations of an aggregation tree, where a tree is constructed while scanning a database based on time attribute of the tuples. Later, the tree is traversed to compute the result of the aggregation incrementally. The limitations of this method are the restriction on the size of aggregation by available memory and the requirement of a prior knowledge about the orderliness of an input database. In case of start time based ordering of the input tuples, for instance, aggregation tree would be like a linked list and will have a worst case performance ($O(N^2)$).

The paper[1], which is discussed in this report, proposes several sequential algorithms for efficient processing of small scale and large scale aggregations on instant groups, as well as a parallel algorithm in order to achieve scalability on large scale aggregations.

1.2 About Paper

The paper covered in this report was published in 2001 by Bongki Moon, Ines Fernando Vega Lopez and Vijaykumar Immanuel.

The authors describe several algorithms for the efficient and scalable processing of temporal aggregation queries. More specifically, authors propose a balanced tree algorithm for small scale aggregations which has improved the worst case and average case processing time of previous solutions. In addition to that, a bucket algorithm is proposed for large scale aggregation where available memory is less than the size of whole data, and lastly parallel version of the bucket algorithm is introduced which scales up linearly with the increasing number of processor.

Symbol	Price	Volume	Begin	End
IBM	37	1000	10	30
Microsoft	45	2000	7	16
Google	30	2000	7	15
HP	32	2000	20	21

Table 1: Stock Exchange Table

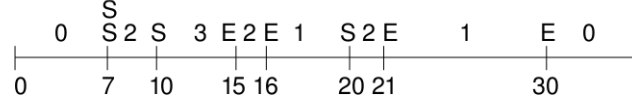


Figure 1: Example of count aggregation by sorting timestamps and tags

2 Small-Scale Aggregation

Important observation of the paper is the grouping the five most common aggregation operators into two groups. Group one includes count, sum, and avg whereas group two includes max and min. The criteria in the grouping is the demand to keep track of attribute values of tuples. Group two has a higher demand to keep track of attribute values of tuples. The proposed solution for the first group of operators is balanced tree algorithm and the solution for the second group of operators is merge-sort aggregation algorithm which is based on merge-sort[3].

Under the assumption of large enough memory to store the entire data structure for each aggregation algorithm, next balanced tree algorithm and merge-sort aggregation algorithm will be discussed.

2.1 Balanced Tree Algorithm

In this subsection, count operator will be used as the representative of the first group operators.

Basic approach to compute the count aggregation would be 1) loading the entire tuples in memory, 2) sorting them based on start and end timestamps in an increasing order and 3) initializing the aggregate result as 0, and then scanning the ordered timestamps and increasing the count result by the number of start timestamps and decreasing it by the number of end timestamps till the end of the table. Figure 1 illustrates a sorted list of timestamps and intermediate results of count aggregate for the sample data in Table 1. For instance, at timestamp 7 the count is increased to 2 from 0, since two tuples have starting time of 7; at timestamp 10, the count is decreased by 1 since third tuple is expired. The worst case processing time of this approach is $O(N \log N)$, where N is the number of tuples in an input database.

The simple approach stores whole dataset in memory and then sorts them based on timestamps. An improvement would be to sort the data incrementally when it is scanned, which is the idea pursued in balanced tree algorithm. The second idea of balanced tree algorithm is to store the repeatable timestamps only once since in real world timestamps of the tuples are likely to overlap. Next, the details of the algorithm will be explained.

Each node in the balanced tree represents a unique timestamp value and stores two counters (for counter example): the number of tuples starting at the timestamp and the number of tuples ending at the timestamp. Additionally, a color tag is stored in each node, as red-black insertion algorithm [2] is used to keep the tree balanced dynamically.

Step 1: Construction of the balanced tree

For each scanned tuple,

1. the start and end timestamps of the tuple is extracted
2. and then for each timestamp value
 - (a) if there is no node for the start(end) timestamp, a new node is created and its start(end)-counter is set to 1
 - (b) if there is a node, its start(end)-counter is increased by one

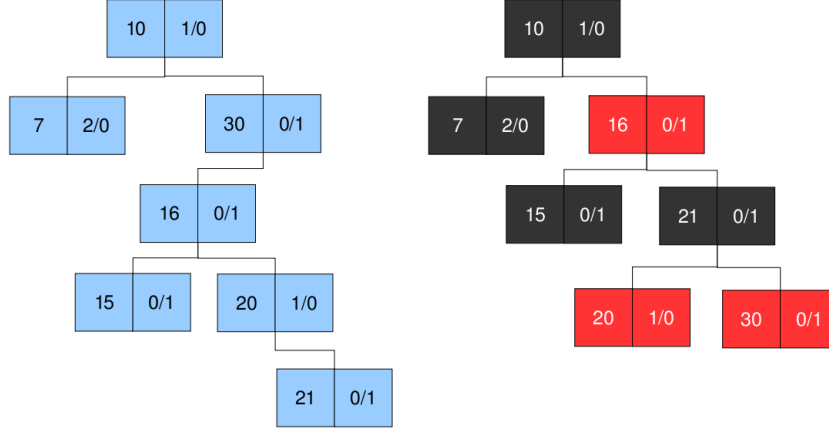


Figure 2: Balanced Tree Algorithm for count aggregation on Stock Exchange Example

Figure 2.1 depicts the construction of the balanced tree for the Stock Exchange example before (see 2(a)) and after it is balanced by red-black insertion (see 2(b)).

Step 2: Computation of the count aggregate

In order to compute the result of the count aggregation, the balanced tree is traversed in order and the count aggregate result is increased by the start-counter of each node and it is decreased by the end-counter of each node.

2.2 Merge-Sort Aggregation

In this subsection, max operator is used as the representative of second group aggregation operators.

Basically, Merge-Sort Aggregation algorithm computes an aggregation result of larger interval by merging aggregate result for two smaller intervals. Figure 2.2 illustrates the algorithm for Stock Exchange Example in Table 1. Figure 3(a) represents the initial step of the algorithm where price values for each interval in Table can be seen. Figure 3(b) depicts the intermediate max aggregate results after the first merging step. In this step intervals of 7 to 16 and 10 to 30, and 7 to 10 and 20 to 21 are merged. Nil is used for the intervals where price value is not known such as interval 15 to 20. Finally, Figure 3(c) represents the intervals after the second merge, intervals 7 to 30 and 7 to 21 are merged.

2.3 Discussion

Balanced Tree Algorithm has the following advantages over simple approach: 1) more efficient response time ($O(\log N)$ vs $O(N)$), 2) less instant memory usage at the construction of the data structure (due to incremental sorting) 3) less memory usage for the data structure (due to storing the aggregate value per timestamp). First two advantages are basically from the usage of

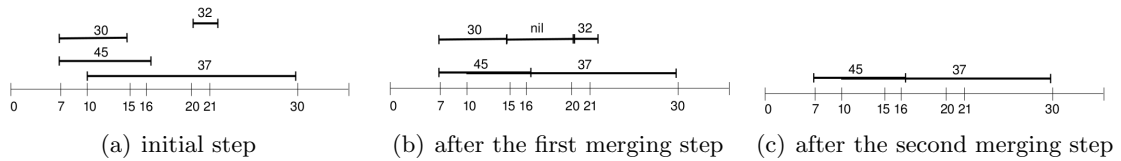


Figure 3: Merge Algorithm for max aggregation on Stock Exchange Example

a balanced tree as data structure. The key advantage of Balanced Tree Algorithm is less memory usage by storing a merged information for unique timestamp values. Definitely the gain will increase with the number of repeated timestamps. While the idea brings a gain in memory, the tuple information is lost. For instance, the Balanced Tree does not give an information for the individual tuple's life time, which limits its usage for first group aggregations. Additionally, the balanced tree algorithm does not eliminate the limitation of previous works which is the large memory requirement for storing whole data set. Similarly, balanced tree algorithm has to store the whole data structure in memory. Due to this limitation, the algorithm is suggested for small scale aggregations.

The worst case processing time of Merge-Sort Aggregation Algorithm is $O(N \log(N))$ which is better than the simple approach, but its memory usage is $O(N)$ similar to simple approach, where N is the number of input tuples. On the other hand, the memory needed for intermediary results might be less than the input size if the aggregation value is shared among neighbour intervals as intervals are merged. One critic to merge-sort aggregation can be regarding the changes in input data. In case of an update in input data, basically each merged interval should be visited and updated if necessary, which would have a processing cost of linear to the number of merged intervals. Similarly, the limitation of storing whole data structure is also valid for Merge-Sort Aggregation Algorithm.

3 Large-Scale Aggregation

So far, we have made an assumption that there is an available memory to store the whole data structure required to compute the aggregation. Actually, in case of databases which are substantially larger than the size of available memory, this assumption does not hold. In this section, the bucket algorithm which is based on data partitioning is discussed.

3.1 Bucket Algorithm

Bucket algorithm is inspired by relational hash join. In hash join two relational tables are partitioned based on join attribute by using the same hash function. In this way, the join is computed only among the partitions which share the same join attribute. The advantage here is only one database access per relation, if it is possible to store all the tuples of one relation in a partition(bucket) in memory. The idea of hash join can not be directly applied to the temporal data as the time intervals of tuples may be of any length. Therefore, bucket algorithm is introduced.

Bucket algorithm works as follows: Time interval is divided into N_B number of disjoint intervals, where N_B denotes number of buckets. Each bucket stores the tuples which have started and/or ended at its interval. In addition to the buckets, a meta array is stored for tuples whose life span more than one bucket. This is especially needed in order to prevent data replication. The size of a meta array is equal to the number of buckets. i^{th} value in meta array stores the aggregation value for long running tuples which have started before i^{th} bucket and will end after i^{th} bucket. Thus, the final aggregate result is computed by using both tuples inside a bucket and the metadata array.

Figure 4 depicts the buckets and Meta Array for count aggregation of the time intervals shown in the Figure. For example, t_1 is a long-lived interval and its life time spans first, second, third and fourth buckets. Since t_1 's life is started at the interval of Bucket1 and at the interval of Bucket4, it is stored in the Bucket1 and Bucket4. tn' denotes that long-lived n^{th} tuple's life is ended in that Bucket. On the other hand, t_4 is a short-lived tuple, whose life does not span more than one bucket. The information about the intermediate aggregate result of long-lived

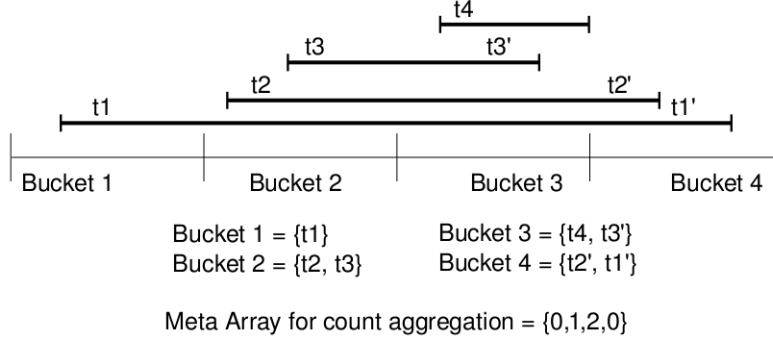


Figure 4: Buckets and Meta Array

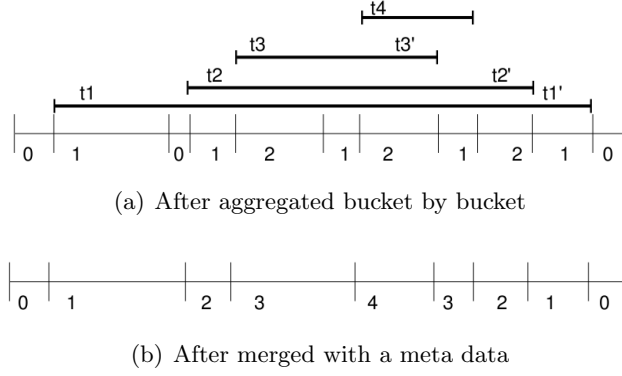


Figure 5: Steps of the count aggregation by using bucket algorithm.

tuples, which are not stored in Bucket is gathered from Meta Array. For instance, the content of Bucket2 is the second and third tuples, but actually t_1 's life also spans that bucket. Therefore, the second entry in Meta Array has the count value of 1.

Figure 5 illustrates the computation of count aggregation on the example. At first step, aggregation is performed for each bucket independently by considering the life time of tuples in that bucket. At second step, the intermediate count result is updated with the information from Meta Array and intervals are merged if they share the same aggregation value.

3.2 Discussion

The temporal aggregate operation can be computed by reading each bucket just once in bucket algorithm, if Meta Array can be stored in memory. The performance of the algorithm depends on the number of Buckets and the data distribution. In the worst case, each tuple can be replicated which makes the data size twice.

4 Parallel Bucket Algorithm

With the increasing availability of data, the parallel processing of the data becomes more important. In this section, the parallel bucket algorithm is introduced which is based on bucket algorithm. Parallel bucket algorithm aims to provide scalability which is defined as an increase in performance proportional to an increase in the number of participating processors. Next, the data structure of parallel bucket algorithm and then the algorithm will be explained.

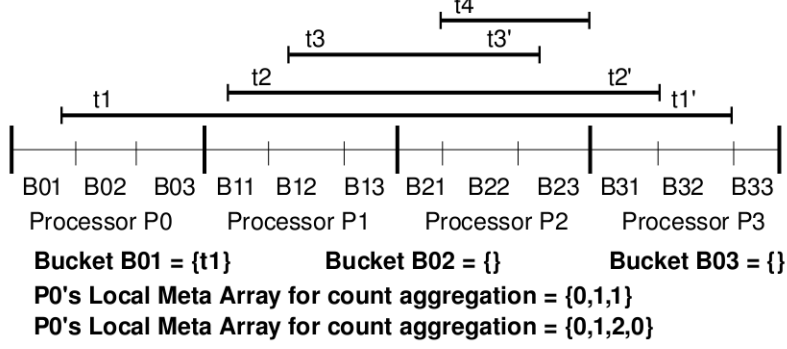


Figure 6: Buckets and Meta Array

Parallelizing bucket algorithm is relatively straightforward, since each bucket is treated separately. Basically, the idea behind the parallel bucket algorithm is two levels of bucketing. The time-line of a given temporal database is partitioned into P disjoint intervals, where P is the number of processors. Then, on each processor, the time-line of the processor is again partitioned into N_B disjoint intervals. Since the construction of meta arrays must also be processed in parallel in an efficient way, usage of a local meta array and a global meta array is proposed. Local and global meta array are used for tuples whose life spans overlap time-lines of multiple local buckets and multiple processors, respectively.

Figure 6 depicts a scenario where all the tuples are initially stored on Processor P0 and four processors P0, P1, P2, P3 participate in a count aggregation. Since the time interval of t_1 spans over four remote processors P0, P1, P2, and P3, t_1 is split into two segments, t_1 and t_1' , t_1 is assigned to P0's local bucket B01 and t_1' is sent to the processors P3. Then, the second and third element of the global meta array of P0 are incremented by one. In a similar way, t_2 is splitted into two and t_2, t_2' are sent to the processors P1 and P3, respectively. Afterwards, the third element of P0s global meta array is incremented by one. Figure 6 shows the resulting data distribution across P0s local buckets, P0s local and global meta arrays. The result aggregate is computed very similar to bucket algorithm, but in addition to local meta array, the intermediate aggregate results from global meta array should also be considered in computation.

4.1 Discussion

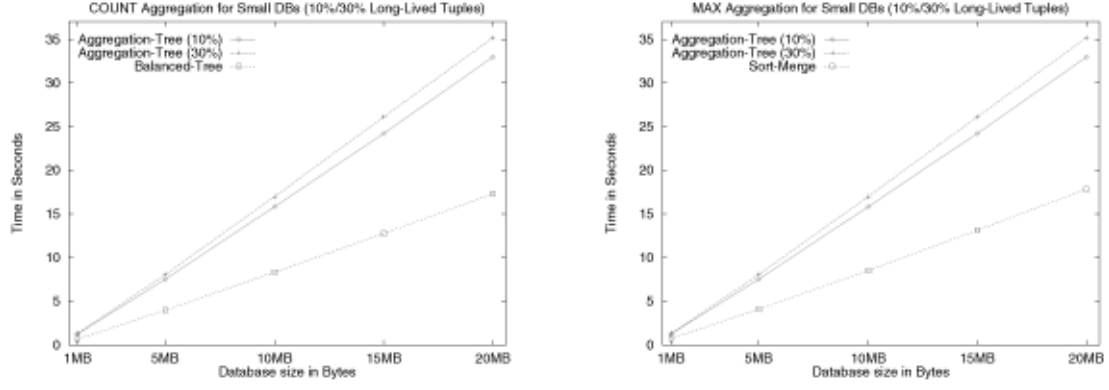
The parallel bucket algorithm partitions the data into $N_B * P$ disjoint intervals instead of N_B intervals in bucket algorithm. Important difference between two algorithms is the cost of communication among different processors to compute the final result of aggregation. Note that each processor computes its global meta array independently, but the combined global meta array is needed in computation of the final aggregation, which requires communication among processors.

5 Evaluation

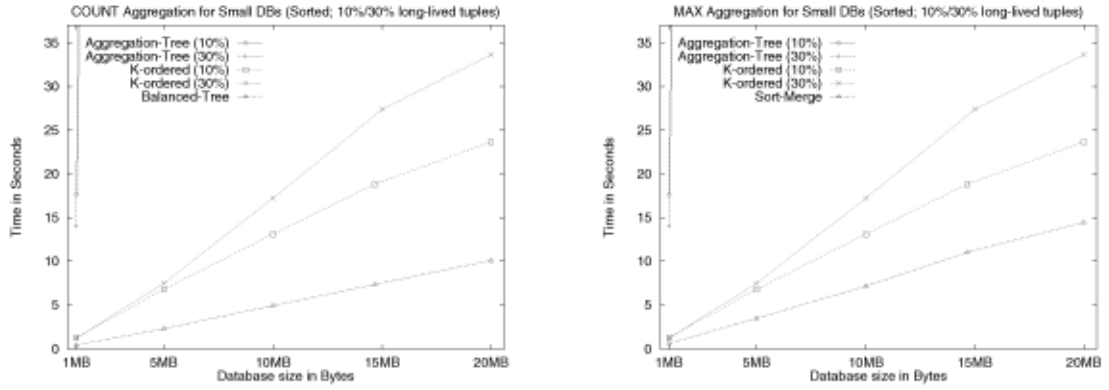
5.1 Small-Scale Aggregation

The assumption made about small-scale aggregation algorithms was the availability of enough memory to store the required data structure. Therefore, in the evaluation of small-scale algorithms, small databases having size of 1MBytes to 20MBytes are used.

Balanced Tree algorithm and merge-sort aggregation algorithms are compared with the state



(a) aggregation for unsorted data



(b) aggregation for sorted data

Figure 7: Aggregation time for small-scale databases [1]

of the art solution aggregation tree in terms of computation time. The results published in the paper is shown in Figure 7. Two scenarios are discussed. In first scenario, aggregation tree and the proposed solutions are compared (see 7(a)) when the the percentage of long-lived tuples varies and it is shown that the proposed solutions outperform the aggregation tree and they are not sensitive to the life span of the tuples unlike aggregation tree. In second scenario, the order property of the data is varied and the proposed solutions are compared to the aggregation tree (see 7(b)). The performance advantage of the proposed solutions are increased here, since the aggregation tree became highly unbalanced. An important critics about the paper could be, although it has been mentioned about balanced tree algorithm in related work section of the paper, comparison between balanced tree algorithms are not performed. It would have been also nice to see the total and instant memory comparison with previous methods especially in case of repeatable timestamps for balanced aggregation tree algorithm.

5.2 Bucket Algorithm

Bucket algorithm was designed for large-scale aggregations based on partitioning the data into equal intervals. As it has been discussed before the performance of the algorithm depends on the number of buckets and the distribution of the data.

Figure 8 shows the result of experiments published in the paper. First experiment8(a) shows the effect of bucket size clearly. In Figure 8(a), 3, 8, and 15 are the optimal bucket numbers for a database of 100 MBytes, 200 MBytes, and 400 MBytes with 10 percent of long-lived tuples,

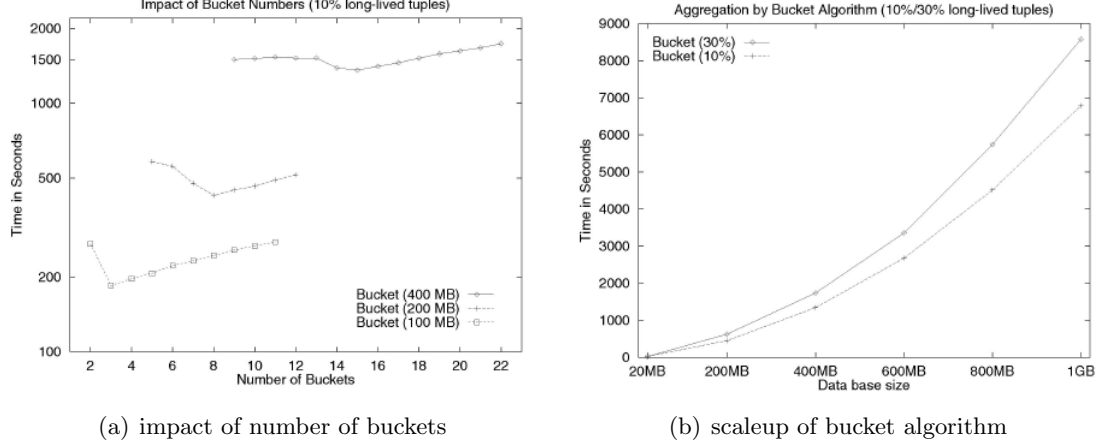


Figure 8: Aggregation time for large-scale databases [1]

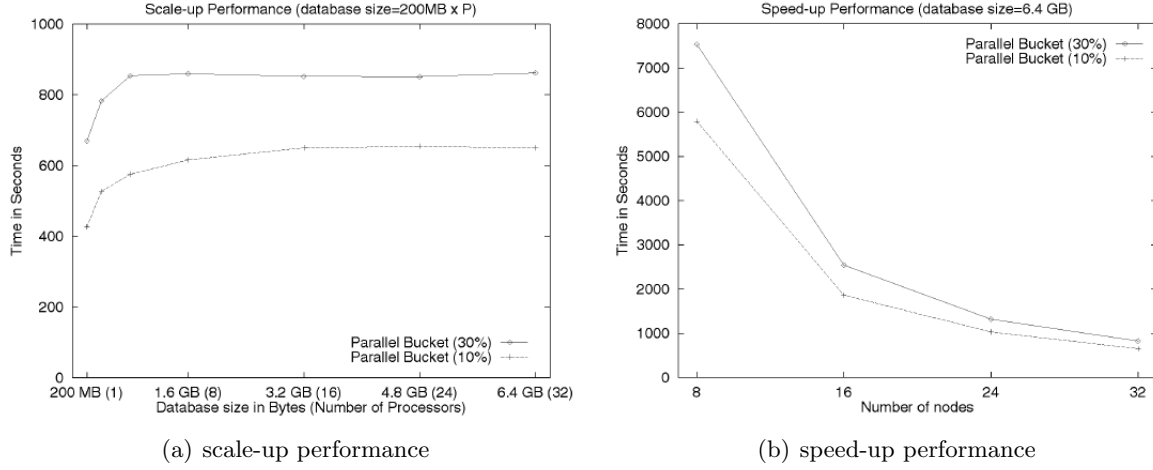


Figure 9: Performance of parallel bucket aggregation [1]

respectively. The second Figure 8(b) demonstrates that the bucket algorithm can compute temporal aggregates for databases larger than the size of available memory, but its performance reduced exponentially with the increasing data size.

5.3 Parallel Bucket Algorithm

Parallel bucket algorithm was designed to provide scalability with the increasing number of processor.

Figure 9 demonstrates the performance study of the algorithm. In Figure 9(a), it is shown that the algorithm has a nearly linear scale-up performance with respect to the increasing number of processors after the eighth processor. In this scenario, each processor has a database of 200MB. The immediate increase in computation time as the number of processors is increased from one to two is explained as the cost of message passing among processors. As the number of processors increases, the increase of overhead seems to reduce and becomes almost constant above the four processor case, and therefore the algorithm achieves nearly linear scale-up performance. Second experiment 9(b) demonstrates the speed-up performance when the size of the database is fixed to 320 million tuples and the number of processor is varied from 1 to 32. The

result of the experiment shows a super linear speed-up. The explanation of it is the overall aggregation cost is reduced by computing many smaller aggregations rather than computing a few larger aggregations, since the batch size gets smaller with the increasing number of processors.

6 Conclusion

In this report, the paper which is proposed several algorithms for computing temporal aggregates for small-scale, large-scale sequential algorithms and a parallel algorithm is discussed. It is shown in the paper that the proposed algorithms provide significant benefits over the current state-of-the-art. The balanced tree and merge-sort aggregation algorithms which are designed for small databases overperformed the aggregate tree in terms of worst-case and average-case processing time. Additionally, with the sequential and parallel bucket algorithms which are proposed for large databases whose data size is larger than the size of available memory, gain in performance and scalability is achieved.

References

- [1] Vijaykumar Immanuel Bongki Moon, Ines Fernando Vega Lopez. Efficient Algorithms for Large-Scale Temporal Aggregation. *IEEE Transactions on Knowledge and Data Engineering*, 2003.
- [2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press and McGraw-Hill Book Company, 2001.
- [3] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.