Efficient Aggregation of Time Series Data

Igor Manojlović*, Aleksandar Erdeljan**
* Schneider Electric DMS NS LLC, Novi Sad, Serbia
** Faculty of Technical Sciences, University of Novi Sad, Novi Sad, Serbia
igor.manojlovic@schneider-electric-dms.com, ftn_erdeljan@uns.ac.rs

***Abstract*—Modern supervisory control and data acquisition (SCADA) systems collect large number of measurements, inherently time series data, from various sensors in critical infrastructures such as transmission and distribution of electrical power, water and gas. Analysis of such number of measurements individually, even with modern utility management systems (UMS) on top of SCADA, is rarely feasible, especially in real-time smart and sensing systems that are continuously acquiring sensor readings. One approach to overcome this problem is to extract important characteristics of such data using business intelligence solution that is capable of performing temporal aggregation and providing access to a significantly smaller amount of aggregated data that can then be analyzed instead of raw data. Developing a technique for efficient temporal aggregation of such large data sets, thus, becomes an essential but non-trivial task. This paper proposes one such algorithm that can efficiently compute and retrieve temporal aggregates for large volumes of time series data, continuously, as such data arrives. The algorithm's efficiency was proven both theoretically and experimentally. Complexity analysis showed that the algorithm achieves logarithmic update processing time cost, logarithmic query time cost and linear storage space consumption. Moreover, performance results from experimental studies show strong indication that the algorithm could be applied in practice for aggregating time series data on a large scale in real-time.**

I. INTRODUCTION

Time series data plays an important role in analysis of critical infrastructures with modern UMS. With large amount of such data being collected with modern SCADA from various sensors such analysis becomes extremely difficult and temporal aggregation becomes increasingly important.

In query languages, aggregation is identified as the process of computing a single value, an aggregate, which summarizes a set of attribute values by means of an aggregation function. In a conventional database management system (DBMS), aggregation is done by first partitioning the argument relation into groups of tuples with equivalent values for one or more attributes and then applying an aggregation function to each group in turn. Classical aggregation functions include min, max, sum, count and average. However, for interval-valued databases, such as temporal databases, more partitioning predicates other than equality, such as temporal overlapping, emerge as being important along with a variety of aggregation functions, such as moving average. Therefore, aggregation possibilities are greatly expanded and the process of temporal aggregation is far more complex problem than aggregation in conventional DBMS and, hence, more prone to inefficiencies.

In general, temporal database is a database system that supports the time domain and is thus able to manage time-varying data [1]. On the other hand, temporal aggregation is a process based on temporal grouping where the timeline is partitioned over time, tuples are grouped over these partitions and aggregate values are computed over these groups [2]. There are two types of temporal grouping [3]: span grouping and instant grouping. Span grouping is based on pre-defined time periods, such as days or weeks, while instant grouping depends on tuples whose intervals contain a given time instant. Aggregations based on span and instant groupings are called span temporal aggregation (STA) and instant temporal aggregation (ITA), respectively. There are also two types of aggregation functions [4]: computational, obtained by accumulating certain computations on tuples, such as sum and count, and selective, obtained by selecting a representative value among values of tuples, such as min and max.

Since each time series acquired by SCADA originates from a different sensor called point, it is essential to compute temporal aggregates for each point individually so that each point can still be analyzed as a separate entity, but with a significantly smaller amount of data. Examples of such analysis are detection of correlation between variables and forecasting [5]. In such case, it is impractical to use ITA because when applied on individual points it results in aggregated data of the same amount as raw time series data. This makes STA incomparably better choice.

This paper introduces an algorithm that efficiently performs continuous STA of time series data (C-STATS) for each point individually as such data arrives. The C-STATS algorithm is based on a new I/O and main memory efficient data structure, the linked sequence tree (LS-tree), which is used for managing both raw and aggregated time series data. Also, while most of existing approaches consider only regular time spans expressed in terms of temporal granularities, such as years and months [6], this paper proposes a new, generic granularity model (G-model) as an approach to definition of temporal granularities or simply granularities, that is also crucial for the algorithm's efficiency. The proposed algorithm is also applicable to both computational and selective aggregation functions.

We analyzed the proposed algorithm as an access method by its complexity and performance that are characterized by three costs [7]: (1) the update processing time cost (the time to update the method's data structures as a result of a change: insert, update or delete operation), (2) the query time cost for each of the basic queries and (3) the storage space cost for physically storing data structures. The complexity of the algorithm was derived from theoretical proof, while its performance was tested

experimentally on large data sets. Conducted experiments were designed to simulate the ratio between the number of points and the number of tuples for each point in laboratory conditions rather than to fully imitate the amount of data that would accumulate over time and reach much greater numbers in practice.

The rest of the paper is organized as follows. Section 2 discusses closely related work. Section 3 introduces core concepts used in the rest of the paper. Proposed G-model and LS-tree are introduced in sections 4 and 5, respectively. The C-STATS algorithm is introduced in section 6 and its complexity analyzed in section 7, while performance results are discussed in section 8. Finally, section 9 presents conclusions and open problems for further research.

II. Related Work

A considerable amount of work has been done by the scientific community to support temporal aggregation that has been formalized in [8]. The most related work is focused on solving the problems of answering basic temporal aggregation queries in terms of temporal databases and coping with user-defined granularities. A large number of techniques have been proposed to efficiently support ITA [4, 6, 9 – 12]. However, there are much less techniques proposed for STA, none of which considers efficient aggregation of such large time series data sets as the ones in modern UMS.

An approach for computing temporal aggregates with multiple levels of granularities was proposed in [13]. A model called the Hierarchical Temporal Aggregation (HTA) model was introduced to aggregate more recent data with finer detail and older data at coarser granularities. However, the proposed algorithm is focused only on computational aggregation functions. It uses a structure called the 2SB-tree that maintains two segment balanced trees [10] with tuples that beside values have timestamps instead of time intervals. However, each of these timestamps implies an interval from it to $+\infty$, unlike a timestamp recorded for a point in a UMS that implies a time interval from it to the next recorded timestamp for the same point. The HTA model divides the timeline into dynamically adjustable segments and maintains separate index structure for aggregates in each segment. Depending on what triggers the adjustment, the HTA model can be separated into two sub-models. The fixed-storage model aggregates older data at coarser granularity when the total storage of the aggregate index exceeds a fixed threshold. The fixed-time-window model assumes that the lengths of all segments except the first one are fixed, which causes dividing times to increase with advancement of time. The time complexity of the algorithm under the fixed storage model is $O(G \cdot log_B S/G)$ for tuple insertion and $O(log_B S/G)$ for computing aggregates. Here, G is the number of used time granularities, S is the storage limit in the number of blocks and B is the block capacity in the number of records. The time complexity of the algorithm under the fixed time window model cannot be guaranteed because advancing of dividing times doesn't depend on index size but on length of segments. In the worst case, all tuples fall on the last partition and thus are kept at finest granularity.

Another approach for using multiple levels of granularities for temporal aggregation was proposed in [14]. Here, aggregation for different granularities resembles the roll-up operation examined in data warehousing studies. In a data warehouse, information is stored at finest granularity and the roll-up takes place at query time, which allows a user to obtain aggregates at coarser granularity. This approach provides a set of conversion functions that can be used to convert aggregates between granularities on the basis of temporal relationships between them.

A generic algorithmic framework that can be applied to many different forms of aggregation was proposed in [15] and implemented in the context of Tripod [16][17], a prototype of spatiotemporal object-oriented DBMS. The proposed technique is based on construction of hash tables for both the attributes that determine how tuples are partitioned and the attributes that aggregates are computed for (aggregation attributes). Once the two hash tables are populated they can be used to determine which aggregation attribute value belongs to which partition. Then, an aggregation function can be applied on aggregation attribute values per partition in order to obtain desired aggregates. The results of performance studies indicate that the framework provides a scalable solution for many cases of aggregation. However, it does not provide any guarantee on performance because its primary target is generality rather than efficiency. One of open challenges it leaves behind is handling update-intensive applications based on sampled continuous functions, such as time series data discussed in this paper.

Several approaches to generalization of temporal aggregation and formulation of temporal query languages were proposed in [18]. These approaches are based on the temporal multi-dimensional aggregation operator (TMDA-operator) [19] that generalizes temporal aggregation by decoupling the partitioning of the timeline for resulting aggregates from the specification of groups of tuples associated with resulting aggregates. The TMDA-operator was evaluated with two algorithms: one for ITA and another for STA. The algorithm for STA processes tuples in chronological order and computes aggregates that are then stored in a table and updated as tuples are being scanned. The average time complexity of the algorithm is $O(N \cdot logM)$ for N tuples and M resulting aggregates. A general model that offers a uniform way of expressing the various forms of temporal aggregation was later proposed in [20]. This model is based on the formal relational database framework available in SQL that includes aggregation [21]. However, it implies an implementation that requires costly operations, such as joins and multiple scans of the argument relation, which would make it inefficient.

III. Core Concepts

This section provides formal definitions of time series, granularity and STA concepts used in the rest of the paper.

Time series are sequences of values ordered with respect to associated timestamps from the time domain. Formally:

Definition 1. ***Time domain.***
A time domain is a pair $(T, \leq)$ *where* $T \neq \emptyset$ *is a set of timestamps and* $\leq$ *is a total order on* T.

Definition 2. ***Time series.***
A time series τ *is an ordered sequence of values* $\tau = \{x_t : t \in T\}$ *where* $(t = t_1, ..., t_n) \wedge (\forall i \in \{1, ..., n\})(t_i < t_{i+1})$.

Time series tuples (TS tuples) are detonated as <*key*, *time*, *value*>, where *key* represents an identifier of an individual time series called point, *time* represents a timestamp associated with *value*, and *value* represents a time series value itself.

Granularity concepts were formalized in [22]. Granularities are mappings γ from an index set to the time domain. Each non-empty subset $\gamma(i)$ is called a granule of the granularity γ. Granules in a granularity do not overlap and their index order is the same as their time domain order. Formally:

DEFINITION 3. ***Granularity.***
A granularity is a function $\gamma : N_0 \rightarrow T$ *such that*

$$\gamma(i) \neq \emptyset \wedge \gamma(j) \neq \emptyset \wedge i < j \Rightarrow \gamma(i) < \gamma(j) \quad (1)$$

and

$$\gamma(i) \neq \emptyset \wedge \gamma(j) \neq \emptyset \wedge i < k < j \Rightarrow \gamma(k) \neq \emptyset \quad (2)$$

where $i \in N_0, j \in N_0$ *and* $k \in N_0$.

Granules can be composed of a set of contiguous time instants (time interval), a single time instant, or even a set of non-contiguous time instants. However, STA applies only to granules composed of time intervals.

Finally, span temporally aggregated time series (STATS) can be defined as a series of aggregates that summarize original time series values by means of aggregation functions on time intervals of pre-defined granularities. Formally:

DEFINITION 4. ***STATS***
A STATS is an ordered sequence elements $\alpha = \{a_i : i \in N_0\}$, *where* $a_i = \{f_1(x),\ldots, f_k(x)\}$ *is a set of aggregates that are result of aggregation functions* $\{f_1,\ldots, f_k\}$ *for a time series* τ *on a granule* $\gamma(i)$.

STATS tuples are denoted as <*key, start time, end time, aggregates*>, where *key* represents point identifier, *start* and *end time* represent aggregated time interval, and *aggregates* represents a set of computed aggregates.

IV. THE G-MODEL

The property of granularities that does not allow granules to overlap, consequently, ensures that granules in a granularity can be uniquely identified by their start and end times. This paper exploits these properties to model granularities with two functions: σ and ε. Given a timestamp as an input, these functions produce start and end time, respectively, of a granule that overlaps with given timestamp. Formally:

DEFINITION 4. ***Functions σ and ε.***
Let $s(i)$ *and* $e(i)$ *be start and end time, respectively, of a granule* $\gamma(i)$. *Then, functions* σ: $T \rightarrow T$ *and* ε: $T \rightarrow T$ *are such that*

$$\sigma(t) = s(i) \wedge \varepsilon(t) = e(i) \Leftrightarrow s(i) \leq t < e(i) \quad (3)$$

Fig. 1 shows an example of the G-model whose granules represent calendar days.

```
Time σ(Time t):
   return new Time(t.Year, t.Month, t.Day)

Time ε(Time t):
   return σ(t).AddDays(1)
```

Figure 1. An example of the G-model.

V. THE LS-TREE

The LS-tree is a new tree data structure for indexing an ordered sequence of elements. Its design goal is to provide an index structure that the C-STATS algorithm can utilize for indexing TS and STATS tuples to achieve desired efficiency. To meet these requirements, the LS-tree indexes tuples by their unique identifiers and keeps each tuple linked to previous and next tuple. The unique identifiers, in this case, represent *keys* and *times* of TS tuples or *keys* and *start times* of STATS tuples. The purpose of these identifiers is to enable efficient retrieval of tuples and application of changes (insert, update and delete operations). On the other hand, the purpose of linking is to enable efficient computation of aggregation functions, such as integral, that do not depend only on changed tuples, but also on tuples that they are linked to.

In this paper, the LS-tree is implemented as the B+ tree [23] with doubly linked leaves and doubly linked tuples for the same *key* in each leaf. B+ trees provide I/O efficiency in a block-oriented storage context by indexing tuples by their unique identifiers. Doubly linked lists provide I/O efficiency for retrieving previous and next LS-tree leaves and main memory efficiency for retrieving previous and next tuples. The LS-tree inherits the complexity of the B+ tree. Additional construction of a doubly linked list when a leaf is read into main memory does not affect the LS-tree complexity, because the list is constructed as tuples in the leaf are being read in search of a particular tuple.

Fig. 2 shows an example of the LS-tree of order 5 that indexes TS tuples. For the purpose of simplicity, *keys*, *times* and *values* are represented as natural numbers.

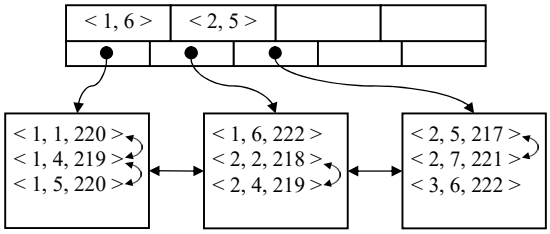

Figure 2. An example of the LS-tree.

VI. THE C-STATS ALGORITHM

The novel C-STATS algorithm efficiently performs STA of time series data for individual points, for multiple granularities and for both computational and selective aggregation functions. It ensures consistency between raw and aggregated data at all times by continuously changing STATS tuples according to changes upon TS tuples. Computing STATS tuples in advance therefore ensures efficiency at query time when such tuples are requested. To achieve this, the algorithm exploits the properties of the G-model and the LS-tree. TS tuples are indexed by their *keys* and *times* in one LS-tree and STATS tuples are indexed by their *keys* and *start times* in multiple LS-trees separated by associated granularities. Therefore, the algorithm maintains $G + 1$ LS-trees, where G is the total number of granularities. Each change of time series data causes all LS-trees to update their structures and the G-model helps to faster identify the affected tuples.

Given a *key*, *time* and *value* of a changed TS tuple, the C-STATS algorithm performs the following steps:

1. **Change TS tuple.** First, a position where a TS tuple with given *key* and *time* should be placed is located in the LS-tree that holds TS tuples. If a TS tuple exists at located position, its *value* is preserved for step 2 as *old value*. Then, a change is applied either by inserting new TS tuple at located position, updating *value* of existing tuple or deleting tuple at located position.
2. **Change STATS tuples.** To change STATS tuples that are affected by incoming changes their positions must first be located in LS-trees by given *key* and their *start times*. Their *start times* are calculated by applying functions σ and ε to *times* of TS tuples. In case of an insert operation, if there are no STATS tuples at located positions, then new ones are created. Finally, each located STATS tuple is changed according to an operation performed upon a TS tuple. If needed, aggregation functions can use both *old* and *new value* of changed TS tuple, as well as links to previous and next tuples, provided by doubly linked lists.

Fig. 3 shows pseudocode of the C-STATS algorithm that performs insert, update and delete operations upon TS tuples as well as selection of STATS tuples. In this pseudocode, integral is computed as an aggregation function that depends both on time series *values* and their *times*. Here, integral represents a surface covered by time series *values* between *start* and *end times* of STATS tuples in a coordinate system where *times* fall on x axis and associated *values* fall on y axis. For the purpose of simplicity, these coordinates are interpolated using the step interpolation.

VII. Complexity Analysis

The complexity of the algorithm was analyzed by three costs: (1) the update processing time cost, (2) the query time cost and (3) the storage space cost.

THEOREM 1. ***Update processing time cost.***
The update processing time cost of the C-STATS algorithm for processing a single change in time series data is O(logN), where N is the total number of TS tuples.

PROOF. Let us denote G as the total number of granularities, M as the number of STATS tuples for a single granularity, k as the number of STATS tuples affected with a change in time series data, F as the total number of aggregation functions and r as the number of TS tuples required for computation of aggregation functions. To process a single change in time series data, the algorithm needs to perform two steps: (1) to update one LS-tree that holds N TS tuples and (2) to update G LS-trees, each holding M STATS tuples. In the first step, the algorithm can apply a change directly to a LS-tree that holds TS tuples using given key, time and value. Since LS-trees inherit complexity of B+ trees, the first step is performed in $O(logN)$ time. In the second step, the algorithm needs to find k STATS tuples in each of G LS-trees and for each of these tuples to perform F aggregation functions using r TS tuples. Since the algorithm can identify affected STATS tuples in LS-trees using given *key* and calculated *start times* and since all required TS tuples can be retrieved using doubly linked lists, the second step is performed in $O(k \cdot G \cdot (logM+F+r))$ time. Thus, the overall update processing time cost of the algorithm is $O(logN+k \cdot G \cdot (logM+F+r))$. This can be simplified to $O(logN)$ because N has higher growth rate then k, G, M, F and r.

THEOREM 2. ***Simple query time cost.***
The query time cost of the C-STATS algorithm for selecting a STATS tuple associated to given granularity, key and start time is O(logM), where M is the number of STATS tuples for a single granularity.

```
def STA:
LSTree TSTuples
Map<Granularity, LSTree> STATSTuples

Insert(Key k, Time t, Value v):
  TSTuple x = TSTuples.AddTuple(k, t, v)
  foreach (Granularity g in STATSTuples):
    Time s = g.σ(t), e = g.ε(t)
    STATSTuple y = STATSTuples[g].GetOrAddTuple(k, s, e, null)
    if (y.Integral = null):
      if (x.Prev ≠ null): y.Integral = (e - s) * x.Prev.Value
      else: y.Integral = 0
    Time nextStartTime = y.StartTime
    if (x.Next ≠ null): nextStartTime = g.σ(x.Next.Time)
    while (y.StartTime ≤ nextStartTime):
      Time from = y.StartTime, to = y.EndTime
      if (from < t): from = t
      if (x.Next ≠ null ∧ x.Next.Time < to): to = x.Next.Time
      if (x.Prev ≠ null): y.Integral – = (to – from) * x.Prev.Value
      y.Integral + = (to – from) * v
      y = y.Next

Update(Key k, Time t, Value v):
  TSTuple x = TSTuples.GetTuple(k, t)
  Value o = x.Value
  x.Value = v
  foreach (Granularity g in STATSTuples):
    STATSTuple y = STATSTuples[g].GetTuple(k, g.σ(t))
    Time nextStartTime = y.StartTime
    if (x.Next ≠ null): nextStartTime = g.σ(x.Next.Time)
    while (y.StartTime ≤ nextStartTime):
      Time from = y.StartTime, to = y.EndTime
      if (from < t): from = t
      if (x.Next ≠ null ∧ x.Next.Time < to): to = x.Next.Time
      y.Integral + = (to – from) * (v – o)
      y = y.Next

Delete(Key k, Time t):
  TSTuple x = TSTuples.GetTuple(k, t)
  Value o = x.Value
  TSTuples.Remove(x)
  foreach (Granularity g in STATSTuples):
    STATSTuple y = STATSTuples[g].GetTuple(k, g.σ(t))
    Time nextStartTime = y.StartTime
    if (x.Next ≠ null): nextStartTime = g.σ(x.Next.Time)
    while (y.StartTime ≤ nextStartTime):
      Time from = y.StartTime, to = y.EndTime
      if (from < t): from = t
      if (x.Next ≠ null ∧ x.Next.Time < to): to = x.Next.Time
      y.Integral – = (to – from) * o
      if (x.Prev ≠ null): y.Integral + = (to – from) * x.Prev.Value
      y = y.Next

Select(Granularity g, Key k, Time from, Time to):
  STATSTuple[] result
  STATSTuple a = STATSTuples[g].GetNexTuple(k, g.σ(from))
  while (a ≠ null ∧ a.StartTime < to ∧ a.EndTime > from):
    result.Add(a)
    a = a.Next
  return result
```

Figure 3. An example of the C-STATS algorithm.

PROOF. To select a STATS tuple, the algorithm needs to perform two steps: (1) to find the LS-tree that holds STATS tuples for given granularity and (2) to find a STATS tuple, among *S* STATS tuples in that particular LS-tree, by its *key* and *start time*. Since granularities are directly mapped to LS-trees that hold STATS tuples and since LS-trees inherit complexity of B+ trees, these steps are performed in $O(1)$ and $O(logM)$ time, respectively. Therefore, the overall query time cost of the algorithm is $O(1+logM) \Rightarrow O(logM)$.

THEOREM 3. ***Range query time cost.***
The query time cost of the C-STATS algorithm for selecting STATS tuples associated to given granularity and key and whose time intervals overlap with given time range is O(logM+k), where M is the number of STATS tuples for a single granularity and k is the number of STATS tuples that satisfy given conditions.

PROOF. The theorem 3 is a generalization of the theorem 2. Here, the algorithm needs to perform the following steps: (1) to calculate *start time* of a first granule that overlaps with given time range using the σ function, (2) to find a STATS tuple for given granularity using given *key* and calculated *start time* and (3) to use doubly linked lists to find next *k* STATS tuples that have the same *key* and whose time intervals also overlap with given time range. These three steps are performed in $O(1)$, $O(logM)$ and $O(k)$ time, respectively. Therefore, the overall query time cost of the algorithm is $O(1+logM+k) \Rightarrow O(logM+k)$.

THEOREM 4. ***Storage space cost.***
The storage space cost of the C-STATS algorithm for storing both raw and aggregated time series data is O(N), where N is the total number of TS tuples.

PROOF. The storage space cost of LS-trees is the same as for B+ trees, which is $O(X)$, where *X* is the number of tuples in the tree. The C-STATS algorithm maintains one LS-tree that holds *N* TS tuples and *G* LS-trees, each holding *M* STATS tuples, where *G* is the total number of granularities and *M* is the number of STATS tuples for a single granularity. Therefore, the overall storage space cost of the algorithm for both raw and aggregated time series data is $O(N+G \cdot M)$. This can be simplified to $O(N)$ because *N* has the highest growth.

VIII. PERFORMANCE RESULTS

The C-STATS algorithm was implemented in C#. All experiments were performed on a PC with eight-core 4GHz processor, 16GB main memory and 500MB/s solid state drive. Sizes of LS-tree blocks were adjusted to 8KB. The algorithm was set to serialize blocks and persist them to disk each time they change. However, in order to reduce disk reads, blocks were removed from main memory only if they have not been used for at least one second. The algorithm was performed on 1 billion TS tuples equally distributed among 10 to 30 thousand points within one-week time period. The purpose of this experimental amount of time series data was to simulate the ratio between the number of points and their TS tuples in laboratory conditions, rather than to fully imitate the amount of data that would accumulate over time and reach much greater numbers in practice. Experimental time series data was aggregated for three granularities (hour, day and week) with five aggregation functions (min, max, sum, count and integral).

Experimental results confirm the complexity analysis of the algorithm. Fig. 4 shows that the algorithm achieves logarithmic time cost for update processing, simple query and range query. None of the experiments showed major differences between insert, update and delete operations. This was also the case for using different number of aggregation functions. Update processing time was mostly influenced by amount of time series data that was much larger than amount of aggregated data, as shown in Fig. 6. Simple and range query showed only minor differences between them, influenced by the number of STATS tuples selected with the range query. However, there was major difference between update processing time and query time, in general, as expected by their complexities. Fig. 5 shows how the average speed of update processing drops as more points and granularities are used. Larger number of granularities requires equally large number of LS-trees and larger number of points results in larger number of STATS tuples. As there is more aggregated data for the algorithm to maintain the performance decreases. However, increasing the number of points showed much less decrease in performance than increasing the number of granularities. Therefore, one of the main challenges for further improvements of the C-STATS algorithm is to reduce these performance differences. Fig. 6 shows linearly increasing storage space consumption mostly dependent on TS tuples. Using different number of points, granularities or aggregation functions did not make any considerable difference in total amount of consumed storage space. In the worst case, when 30 thousand points was aggregated for all three granularities and all five aggregation functions, the storage space was still two orders of magnitude more consumed by TS tuples (~39GB) than by STATS tuples (~400MB).

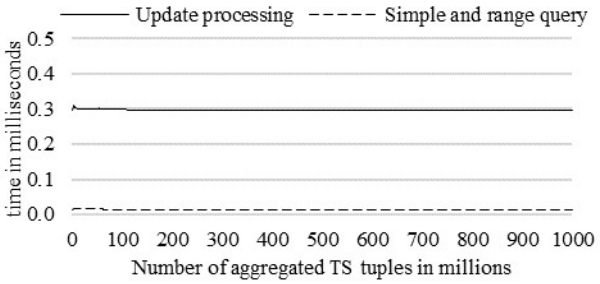

Figure 4. Performance of the C-STATS algorithm for aggregating 10 thousand points, for hour granularity, and for performing the simple query and the range query that selects 100 STATS tuples.

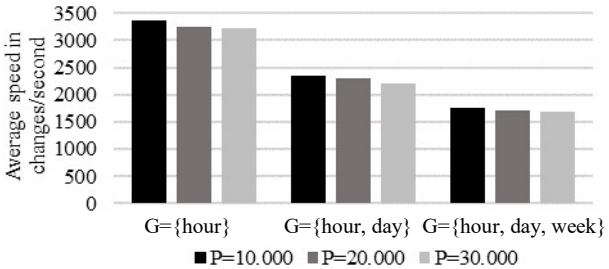

Figure 5. Performance of the C-STATS algorithm for aggregating 1 billion TS tuples for different number of points (P) and different granularities (G).

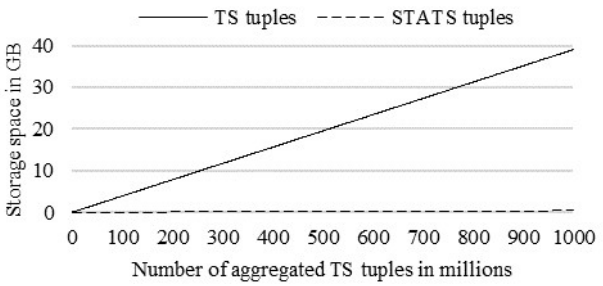

Figure 6. Storage space consumed by the C-STATS algorithm.

IX. CONCLUSION

We developed the C-STATS algorithm that uses the proposed G-model to perform STA of time series data for multiple granularities and that also uses new data structure, the LS-tree, to efficiently compute both classical aggregation functions and the ones specific for temporal databases and time series data. Both theoretical and experimental analysis show that the algorithm can efficiently compute and retrieve span temporal aggregates for large volumes of time series data continuously and individually for each point as time series data arrives. Complexity analysis shows that the algorithm achieves logarithmic update processing time cost, logarithmic query time cost and linear storage space consumption. Performance results from experimental studies also show strong indication that the algorithm could be applied in practice for aggregating time series data on a large scale in real-time. However, despite the achieved performance, there are still opportunities for further improvements.

Our future work will include several improvements of the C-STATS algorithm. Firstly, we plan to find an optimal solution between improving performance at update processing time and reducing performance at query time. Secondly, we plan to develop a parallel version of the C-STATS algorithm that could compute aggregates for different points and granularities in parallel. We also plan to develop a technique for specification of aggregation functions independently from the C-STATS algorithm. Such technique should handle both the aggregation functions that are performed directly on time series data, such as integral, and the ones whose aggregates are computed from other aggregates, such as moving average that can be obtained by dividing integral with the size of aggregated time interval.

REFERENCES

[1] G. Ozsoyoglu and R. T. Snodgrass, "Temporal and Real-Time Databases: A Survey," IEEE Trans. Knowl. Data Eng., vol. 7, no. 4, pp. 513–532, 1995.

[2] R. N. Kline and R. T. Snodgrass, "Computing Temporal Aggregates," in Proceedings of the 11th International Conference on Data Engineering, 1995, pp. 222–231.

[3] R. T. Snodgrass, S. Gomez, and J. McKenzie, L.E., "Aggregates in the Temporal Query Language TQuel," IEEE Trans. Knowl. Data Eng., vol. 5, no. 5, pp. 826–842, 1993.

[4] J. S. Kim, S. T. Kang, and M. H. Kim, "Effective Temporal Aggregation Using Point-Based Trees," in Database and Expert Systems Applications, vol. 1677, T. J. M. Bench-Capon, G. Soda, and A. M. Tjoa, Eds. Berlin, Heidelberg: Springer, 1999, pp. 1018–1030.

[5] A. Dedinec and A. Dedinec, "Correlation of variables with electricity," in Proceedings of the 6th International Conference on Information Society and Technology - ICIST 2016, 2016, pp. 118–123.

[6] S. T. Kang, Y. D. Chung, and M. H. Kim, "An Efficient Method for Temporal Aggregation with Range-Condition Attributes," Inf. Sci. (Ny)., vol. 168, no. 1–4, pp. 243–265, 2004.

[7] B. Salzberg and V. J. Tsotras, "Comparison of Access Methods for Time-Evolving Data," ACM Comput. Surv., vol. 31, no. 2, pp. 158–221, 1999.

[8] I. F. V. Lopez and R. T. Snodgrass, "Spatiotemporal Aggregate Computation: A Survey," IEEE Trans. Knowl. Data Eng., vol. 17, no. 2, pp. 271–286, 2005.

[9] D. Zhang, A. Markowetz, V. Tsotras, D. Gunopulos, and B. Seeger, "Efficient Computation of Temporal Aggregates with Range Predicates," in Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems - PODS '01, 2001, pp. 237–245.

[10] J. Yang and J. Widom, "Incremental Computation and Maintenance of Temporal Aggregates," VLDB J., vol. 12, no. 3, pp. 262–283, 2003.

[11] B. Moon, I. F. V. Lopez, and I. Vijaykumar, "Efficient Algorithms for Large-Scale Temporal Aggregation," IEEE Trans. Knowl. Data Eng., vol. 15, no. 3, pp. 744–759, 2003.

[12] D. Gao et al., "Main Memory-Based Algorithms for Efficient Parallel Aggregation for Temporal Databases," Distrib. Parallel Databases, vol. 16, no. 2, pp. 123–163, 2004.

[13] D. Zhang, D. Gunopulos, V. J. Tsotras, and B. Seeger, "Temporal and Spatio-Temporal Aggregations Over Data Streams Using Multiple Time Granularities," Inf. Syst., vol. 28, no. 1–2, pp. 61–84, 2003.

[14] P. Terenziani, "Temporal Aggregation on User-Defined Granularities," J. Intell. Inf. Syst., vol. 38, no. 3, pp. 785–813, 2012.

[15] S. H. Jeong, A. A. A. Fernandes, N. W. Paton, and T. Griffiths, "A Generic Algorithmic Framework for Aggregation of Spatio-Temporal Data," in Proceedings of the 16th International Conference on Scientific and Statistical Database Management, 2004, pp. 245–254.

[16] T. Griffiths et al., "Tripod: A Comprehensive System for the Management of Spatial and Aspatial Historical Objects," in Proceedings of the 9th ACM international symposium on Advances in geographic information systems, 2001, pp. 118–123.

[17] T. Griffiths, A. Fernandes, N. W. Paton, and R. Barr, "The Tripod Spatio-Historical Data Model," Data Knowl. Eng., vol. 49, no. 1, pp. 23–65, 2004.

[18] M. H. Böhlen, J. Gamper, and C. S. Jensen, "How Would You Like to Aggregate Your Temporal Data?," in Proceedings of the International Workshop on Temporal Representation and Reasoning, 2006, vol. 2006, pp. 121–136.

[19] M. Böhlen, J. Gamper, and C. S. Jensen, "Multi-Dimensional Aggregation for Temporal Data," in Advances in Database Technology - EDBT, Y. Ioannidis, M. H. Scholl, J. W. Schmidt, F. Matthes, M. Hatzopoulos, K. Boehm, A. Kemper, T. Grust, and C. Boehm, Eds. Munich, German: Springer, 2006, pp. 257–275.

[20] M. H. Böhlen, J. Gamper, and C. S. Jensen, "Towards General Temporal Aggregation," in Sharing Data, Information and Knowledge, vol. 5071, A. Gray, K. Jeffery, and J. Shao, Eds. Berlin, Heidelberg: Springer, 2008, pp. 257–269.

[21] A. Klug, "Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions," J. ACM, vol. 29, no. 3, pp. 699–717, 1982.

[22] C. Bettini, C. E. Dyreson, W. S. Evans, R. T. Snodgrass, and X. S. Wang, "A Glossary of Time Granularity Concepts," in Temporal Databases: Research and Practice, vol. 1399, O. Etzion, S. Jajodia, and S. Sripada, Eds. Berlin, Heidelberg: Springer, 1998, pp. 406–413.

[23] D. Comer, "Ubiquitous B-Tree," ACM Comput. Surv., vol. 11, no. 2, pp. 121–137, 1979.