An Approach to Adaptive Memory Management in Data Stream Systems

Michael Cammert, Jürgen Krämer, Bernhard Seeger, Sonny Vaupel
Department of Mathematics and Computer Science
University of Marburg, Germany
{cammert,kraemerj,seeger,sonny}@mathematik.uni-marburg.de

Abstract

Adaptivity is a challenging open issue in data stream management. In this paper, we tackle the problem of memory adaptivity inside a system executing temporal sliding window queries over continuous data streams. Two different techniques to control the memory usage at runtime are proposed which refer to changes in window sizes and time granularities. Both techniques differ from standard load shedding approaches based on sampling as they ensure precise query answers for user-defined Quality of Service (QoS) specifications, even under query re-optimization.

1. Introduction

Memory adaptivity is of particular importance for a data stream management system (DSMS) since long-running queries as well as changing stream characteristics may cause a DSMS to exceed its memory capacities. Thus, the memory manager has to reduce the amount of memory allocated for the running queries to prevent critical system states and failures. Moreover, the execution of newly registered queries may necessitate memory re-distribution in order to have sufficient memory available.

In accordance with [2], our work is based on the assumption that the major amount of available memory is allocated for the state maintenance of stateful operators like join and aggregation, whereas the memory occupied by inter-operator queues is almost negligible. This assumption is valid for unsaturated systems that are able to cope with the incoming data streams. Our goal is to effectively control the memory allocated by stateful operations in a query graph. But how can this goal be achieved? Reducing the amount of assigned memory may imply that the state of an operation cannot be maintained properly any longer. Hence, the question arises as to which guarantees hold for the query results. It is known that for random load shedding [6], which reduces system load by probabilistically dropping elements, no strong guarantees on the query results can be given. In the best case, the output is a sample of the exact output having specific qualities. However, already in this case, query optimization becomes quite limited, as, for example, the join over samples of its input does not return a sample of the join's output [3].

Continuous query semantics as well as execution commonly relies on the utilization of temporal information [1, 5]. For that reason, our techniques are also related to time, namely to the size of temporal windows and to the time granularity. From an analysis of typical continuous queries, we observed that users tend to be too conservative when defining window sizes, i. e., they choose relatively long time windows. Therefore, a natural approach for a QoS constraint is to allow a user to specify a passable upper and lower bound for the window size. By adjusting the window size within these bounds, the DSMS can increase or decrease its memory usage at runtime. Even for small windows and high system load, this technique keeps those results computed from elements that are close to each other in time. Note that this preserves the basic idea of sliding windows [4]. In contrast, load shedding [6] is likely to drop some of those results. Our second technique refers to the time granularity at which results, especially aggregates, are computed. For a lot of real-world queries, users neither need nor want results at finest time granularity. Hence, user-defined QoS specifications are useful to save memory as well as CPU resources.

2. Memory Adaptation Techniques

Let $\mathbb{T} = (T, \leq)$ be a discrete time domain with a total order $\leq$. We use $\mathbb{T}$ to model the notion of application time, not system time. For the sake of simplicity, let T be the non-negative integers. We use the generally accepted semantics for continuous queries proposed in [5, 1] as foundation. The implementation is based on our time-interval operator algebra [5], where each stream element is associated with a time interval specifying its validity. Each stream is ordered by the start timestamps of the time intervals.

2.1. Window Size Adjustment

Technique Let us assume a temporal sliding window query consisting of multiple windows with size $w_1, \ldots, w_n, i \in \{1, \ldots, n\}, w_i \in T, n \in \mathbb{N}$. Adjusting the size of a window w_i means to change it to a new value $w' \in T$ at a certain application time instant $t \in T$.

Implementation In a physical query plan, windows are expressed by special window operators assigning the validity of each stream element according to their current window size. Whenever the memory manager wants to change the window size of a sliding window, it informs the respective window operator about the new window size and the point in application time t the window size should be changed. As soon as t is reached during processing, i. e., the start timestamps of the window operator's input stream are larger than or equal to t, the window operator assigns the new window size as validity.

Impact In a physical query plan, the window operators directly influence all stateful operators downstream because the window size correlates with their memory usage. Let us explain this dependency for our approach [5]. A stream element is considered to be valid for the state of an operation as long as its time interval can overlap with that of future incoming elements. Expired elements are purged from the state during a reorganization phase. Because stateful operators have to keep all valid elements in their state, the state is usually larger the longer the validity is and smaller the shorter the validity is. As a consequence, smaller windows imply that elements from the state can be discarded earlier and, thus, the state gets smaller.

Note that adjustments to the window size additionally affect the processing costs of operators as a smaller state generally implies faster insertion, probing, and reorganization. In the case of a nested-loops join, for example, half a state would speed up probing by a factor of two.

2.2. Time Granularity Adjustment

Technique A time granularity G is a non-empty subdomain of the time domain $\mathbb{T}$, $G \subseteq T, 0 \in G$, with the same total order $\leq_{|G}$. We assume the time distance g between each pair of successive time instants in G to be equal. This condition holds for the conventional time granularities such as milliseconds, seconds, minutes, hours. We denote a time granularity G' to be coarser than G, if $G' \subset G$. We say a substream has a time granularity G if the starting and ending time instants of all time intervals assigned to its elements belong to G. Adjusting the time granularity means to change it to a new time granularity G' at a certain application time instant $t \in T$. From this time instant onwards, a new substream starts having granularity G'.

Implementation A special operator, the granularity operator, is used in our physical algebra to adjust the granularity of a stream. If the memory manager decides to change the granularity at a future time instant $t \in T$, it informs this operator about t and the new granularity G'. Similar to the window operator, the granularity operator changes the time granularity as soon as the application time instant t is reached during processing. From this time instant on, it rounds the starting and ending instants of the time intervals according to G'. Let g' be the constant time distance of G'. For an incoming time interval $[t_S, t_E)$, the rounding looks as follows: $t'_S = \left\lceil \frac{t_S}{g'} \right\rceil g'$ and $t'_E = \left\lfloor \frac{t_E}{g'} \right\rfloor g'$.

Impact Compared to the window sizes used in a query, the time distance g of a granularity is typically small. Thus, a coarser granularity means that the time intervals in the respective substream usually become a little shorter. Because the length of time intervals, which corresponds to the validity, is modified only insignificantly by the granularity operator, an adjustment to the time granularity has generally little impact on the memory usage of most stateful operators. However, there is one important exception, namely the aggregation. The state of an aggregation is highly sensitive to the time granularity since a lower granularity implies a reduced number of time instants at which the aggregation values might change. As a consequence, the state gets smaller and less memory is allocated. Moreover, the output rate decreases which saves system resources downstream.

3. Runtime Environment

3.1. Quality-of-Service Specifications

Our idea is to allow the user to specify passable bounds for time windows and time granularities when posing a query. If the DSMS accepts the query, its results have to be returned according to the user-defined QoS specifications. This implies that the memory manager only changes the window sizes and time granularities within the pre-defined bounds in order to adapt memory usage. However, it should aim at maximizing overall QoS.

For temporal sliding windows, we propose to extend the *Range* expression in the FROM clause of CQL statements [1] with the keywords `Min` and `Max`. Instead of specifying windows by `[Range 5 hours]`, we now suggest `[Range Min 2 hours, Max 5 hours]`.

For an easier use, we suggest to introduce new keywords for the most relevant time granularities like `millis`, `seconds`, `minutes`, `hours`, and so on, although the system internally transforms these expressions into equivalent time distances in its internal time domain. With regard to aggregates, we propose the additional keyword `Granularity` plus lower

and upper bounds similar to the window case. For example, `AVG(speed)[Granularity Coarsest 2 hours, Finest 10 seconds]` means that the average speed has to be computed with 10 seconds as finest time granularity and 2 hours as coarsest.

3.2. Subquery Sharing

As a DSMS unifies the queries in a global operator graph, we have to clarify how the QoS constraints are handled under subquery sharing. The user-defined QoS constraints are stored as metadata information at the sinks, i. e., the nodes in the query graph where the results for a query are transferred to. Through a depth-first traversal starting at a sink, the QoS metadata information is propagated upstream to update the existing QoS bounds at the window and granularity operators.

Whenever a window operator ω with bounds $[w_{min}, w_{max}]$ is reached for which QoS constraints w'_{min}, w'_{max} have been specified, its lower bound w_{min} is set to $max(w_{min}, w'_{min})$ and its upper bound w_{max} to $min(w_{max}, w'_{max})$. Similarly, the granularity bounds $[g_c, g_f]$ with g_c as coarsest and g_f as finest granularity are updated whenever a granularity operator is traversed for which QoS constraints g'_c, g'_f have been defined. The coarser bound g_c is set to $min(g_c, g'_c)$ and the finer bound g_f to $max(g_f, g'_f)$.

Subquery sharing is only permitted if the computed QoS constraints for the shared segments are compatible, i. e., the lower bound has to be smaller than or equal to the upper bound. Otherwise, the new query has to be run separately.

3.3. Additional Metadata

The results of a query may depend on several window and time granularity operators. In order to ensure the precise semantics of continuous queries, it is of particular importance to provide the user with information on the time instants the window sizes and time granularities were adjusted. For that reason, each sink of the query graph holds a list with references to those operators.

At each window operator in the query graph, a history of the window sizes is maintained as a list of tuples (t, w) where t denotes the time instant from which on the window size w has been applied. Analogously, each granularity operator maintains a history of applied granularities. Whenever the memory manager changes a time granularity or a window size, it updates the corresponding history.

4. Semantic Guarantees

Since the memory manager is only allowed to change the QoS settings within the pre-computed bounds, all query results have at least the minimum QoS demanded by the user. Due to the histories maintained at the window and granularity operators, it is possible to determine the exact QoS provided by the system for each result at each single time instant. Consequently, all query results are precisely defined and exact with regard to the chosen time windows and granularities. This still holds under query re-optimizations because applying conventional transformation rules guarantees a *snapshot-equivalent* output [5]. Note that such powerful re-optimizations at runtime are not applicable in general for load shedding as shown for the join in [3].

5. Conclusions

This work addresses adaptive memory management for continuous temporal sliding window queries in DSMS. We presented two novel techniques which rely on adjustments to window sizes and time granularities at runtime. Practicable QoS specifications proposed for both techniques enable the memory manager to dynamically control the memory usage within user-defined QoS bounds. The techniques are compatible with the generally accepted stream semantics and do not collide with query optimizations at runtime, which is an advantage compared to load shedding. Our experimental studies already demonstrated their basic feasibility, while a detailed analysis of the impact on system resources is beyond the scope of this paper.

Acknowledgements

This work has been supported by the German Research Society (DFG) under grant no. SE 553/4-3.

References

[1] A. Arasu, S. Babu, and J. Widom. An Abstract Semantics and Concrete Language for Continuous Queries over Streams and Relations. In *Proc. of the Intl. Conf. on Data Base Programming Languages (DBPL)*, 2003.

[2] S. Babu, U. Srivastava, and J. Widom. Exploiting k-Constraints to Reduce Memory Overhead in Continuous Queries over Data Streams. *ACM Transactions on Database Systems (TODS)*, 29(3):545–580, 2004.

[3] S. Chaudhuri, R. Motwani, and V. Narasayya. On Random Sampling over Joins. In *Proc. of the ACM SIGMOD*, pages 263–274, 1999.

[4] L. Golab and M. T. Özsu. Issues in Data Stream Management. *SIGMOD Record*, 32(2):5–14, 2003.

[5] J. Krämer and B. Seeger. A Temporal Foundation for Continuous Queries over Data Streams. In *Proc. of the Int. Conf. on Management of Data (COMAD)*, pages 70–82, 2005.

[6] N. Tatbul, U. Cetintemel, S. B. Zdonik, M. Cherniack, and M. Stonebraker. Load Shedding in a Data Stream Manager. In *Proc. of the Conf. on Very Large Databases (VLDB)*, pages 309–320, 2003.