Shared Processing of Multiple Aggregate Continuous Queries against Spanning and Out-of-Order Events (abstract)

Aurélie Suzanne, Guillaume Raschia, José Martinez

Shared Processing of Multiple Aggregate Continuous Queries against Spanning and Out-of-Order Events

Aurélie Suzanne
aurelie.suzanne@ls2n.fr
Université de Nantes
Nantes, France
Expandium
Saint-Herblain, France

Guillaume Raschia
guillaume.raschia@ls2n.fr
Université de Nantes
Nantes, France

José Martinez
jose.martinez@ls2n.fr
Université de Nantes
Nantes, France

CCS CONCEPTS

• **Information systems** → **Stream management**; **Query planning**; **Query optimization**.

KEYWORDS

data stream management systems, spanning events, out-of-order events, sliding windows, aggregate continuous queries, multiple query optimization, shared processing

1 INTRODUCTION

Data Stream Management Systems are ubiquitous. They process the huge amount of data generated every day, from our personal devices, as cell phones and IoT, to worldwide transactions, as network traffic for the Internet, stock exchanges or even transportation. Stream processing highly focuses on aggregate computation that provides Key Performance Indicators (KPI) to the end user. The KPIs are expressed as Aggregate Continuous Queries (ACQ) defined by temporal windows.

Nowadays, those streaming systems handle events with a lifespan, such as phone calls, as points in time. They also mainly assume streams with no delay. Both spanning events and out-of-order events undoubtedly yield to noisy aggregates.

Ultimately, multi-ACQs requires near real-time processing and is prone to duplicate computation of aggregates in every query due to overlapping and containment of windows. It then gives the opportunity to save execution cost by sharing sub-aggregates through slicing techniques.

In this communication, we develop an engine for Aggregate Continuous Query (ACQ), which is able to (i) incorporate lifespan to provide exact aggregate computation, (ii) properly manage out-of-order events, and (iii) follow a cost-based policy that elaborates at run-time the most efficient query execution plan of multiple ACQs. The query engine is supported by data structures dedicated to spanning and out-order events and a hybrid sharing schema that aggressively saves computation cost among multiple queries. A lot of experiments have been conducted to show the efficiency of the approach in a large variety of settings and stream profiles.

2 SLICING TECHNIQUE

Up to now, shared execution of ACQs followed two distinct paths: *pooling* or *feeding*. Both approaches are based on the *slicing* technique, which subdivides the windows into non-overlapping time ranges. Each slice then incorporates events in the form of a single partial aggregate. At window release those partial aggregates are combined to form a final aggregate. For a single ACQ, size of the slices is driven by the ACQ's parameters: the *range* defining window size and the *step* representing the window sliding frequency.

3 POOL SHARING

The pooling approach focuses on sharing a common slice set among several ACQs. It requires to create fine-grained slices to fit each and every ACQ. At any time, the system maintains a set of slices that cover all the ongoing windows, as shown in Figure 1, with an obvious slice width of 10 on the $\{a, b, c\}$ example. Such an approach allows reducing insertion cost since only one insertion per pool of ACQs is needed, instead of one insertion per ACQ, but it also increases release cost as slices are smaller.

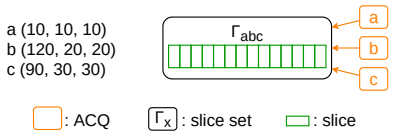

Figure 1: Creation of a slice set for a pool of queries.

The straightforward way of sharing slices is to consider all the ACQs at once, and then build one single slice set. However, if the ACQ steps do not match, it can drastically increase the number of slices required to compute the aggregates. To mitigate that drawback, it is possible to partition the ACQs set into pools of queries. Each pool of ACQs shares a common slice set, while there is no sharing among pools.

Using spanning and out-of-order events in such a pooling configuration is possible, it only requires to insert in several slices for spanning events (as one event may now span over multiple slices), and to scan past slices for out-of-order events. However, without any adaptation, characteristics of such events may also induce errors in results. Indeed, spanning events must be finished in order to release the window. Hence, we add a Time-to-Postpone (TTP) parameter which delays the window release by a fixed amount of time. This TTP is a property of an ACQ, which allows the user to fine tune the delay of each query. For example, one would have a

small TTP for small range ACQs, while a larger TTP is acceptable for longer ranges.

4 FEED SHARING

The feeding approach is based on the idea that in many real-life examples, sliding window queries are not completely unrelated to each other and one ACQ answer might benefit to another ACQ. For example, the aggregates of the sliding window (15 min, 5 min) may be reused by the sliding window (1 hour, 15 min). Thus feed sharing allows feeding a slice set dedicated to an ACQ from the partial results of another slice set, as shown in Figure 2. The main idea is to reduce the number of insertions as the *subscriber* slice sets Γ_b and Γ_c receive a few slices from the *publisher* Γ_a rather than the whole event stream.

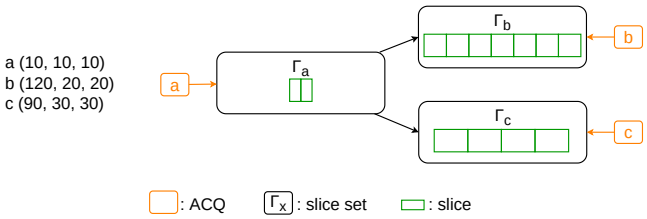

Figure 2: Three slice sets feeding each other. Each slice set is built from its own query step, and it is filled either by the data stream (Γ_a) or by another slice set (Γ_b and Γ_c).

While this idea is quite simple, its application with spanning and out-of-order events needs to be handled with care. Firstly, insertion in a feeding setting with non-delayed point events is always done in the first slice set of the feeding chain. With spanning and out-of-order events, insertion might, however, continue after the first slice set to active subscribing slice sets. Secondly, at window release, slices read will depend on the subscriptions made by a slice set, as an ACQ might read slices only in its slice set, or also in the feeding slice sets.

5 HYBRID SHARING

Pooling and feeding strategies can cohabit such that slice sets are shared among multiple ACQs and subscribe to other slice sets, as a straightforward extension from feeding. An example of such sharing is shown in Figure 3.

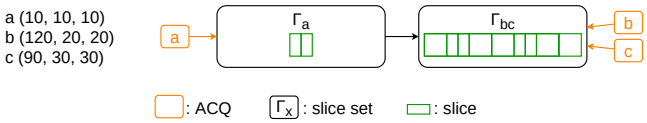

Figure 3: A pooled slice set Γ_{bc} built from queries $\{b, c\}$ and subscribing to another slice set Γ_a.

However, not all combinations are consistent. For two slice sets to be joined they need not to break the subscription chain already in place, and for one slice set to subscribe to another one all its slice bounds must be marked in the feeding slice set.

6 COST ESTIMATION

In order to decide for the best partition, i.e., pooling schema, of the ACQs combined with a relevant feeding schema, one needs to elaborate a cost function that estimates the number of aggregation operations of a specific distribution.

This cost is composed of three parts: insertion, release and shift cost. Insertion cost depends on event size and input rate of the stream. It uses those parameters and the query plan to identify how many slices are used for event insertion per time unit. Release cost uses the range and TTP of the ACQs to count the number of slices read at window release. Finally, the shift cost solely depends on the query plan generated to estimate the number of slices which will be transferred to subscribers at each time unit.

Figure 4 shows a toy example of execution plans given for three queries with different optimization strategies: no-sharing, pool-sharing, feed sharing and the hybrid pool-feed policy. Cost values are given by our cost estimation measure. It also shows promises for the hybrid technique as it exhibits the lowest cost.

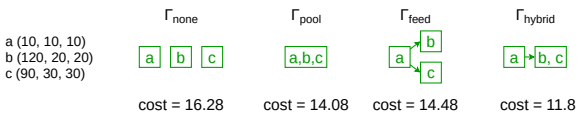

Figure 4: Cost of the query plan following, from left to right, no sharing, pool, feed and hybrid schemes.

As exhaustive search for the best query plan is not possible for medium to large workloads, our system uses a greedy algorithm which starts with all ACQs separated in their own slice set. Then the algorithm tests all the pooling and feeding possibilities between two slice sets and applies the best one. This procedure is done recursively until no option improves the overall cost.

7 EXPERIMENTS

We validate the efficiency of our approach with a set of experiments. In those experiments, we demonstrate the validity of our cost estimation with increasing event size. We also show that under varying settings, our hybrid approach is the most relevant one and provides results better than pooling or feeding approaches.

8 CONCLUSION

In this paper we study multi-ACQ optimization and propose slicing schemes to build efficient query plans in order to leverage shared processing of ACQs. Novelty of that line of work has two dimensions: (i) the assumptions about the data stream: it deals with spanning events and it allows for delays, and (ii) the fully fledged cost measure under those assumptions. Our technique proposes a hybrid schema that supports both pool and feed schemes and adapts to almost any kind of workload.