Approximating Streaming Window Joins Under CPU Limitations

Ahmed Ayad Jeffrey Naughton Stephen Wright
Computer Sciences Department
University of Wisconsin – Madison
{ahmed, naughton, swright}@cs.wisc.edu

Utkarsh Srivastava
Department of Computer Science
Stanford University
usriv@db.stanford.edu

Abstract

Data streaming systems face the possibility of having to shed load in the case of CPU or memory resource limitations. We study the CPU limited scenario in detail. First, we propose a new model for the CPU cost. Then we formally state the problem of shedding load for the goal of obtaining the maximum possible subset of the complete answer, and propose an online strategy for semantic load shedding. Moving on to random load shedding, we discuss random load shedding strategies that decouple the window maintenance and tuple production operations of the symmetric hash join, and prove that one of them – Probe-No-Insert – always dominates the previously proposed coin flipping strategy.

1. Introduction

In the context of data streaming systems, the system has no control over the rate of the incoming data. Hence, the adoption of a *push* model of computation is mandatory. In steady state, the system resources must be greater than what is required by the input, or the system is unstable and the query is infeasible [2]. The system CPU has to keep up with the arrival rate to avoid the input queues and response time growing indefinitely. Also, the amount of available memory has to be enough to hold the required state of the query plus tuples arriving in the input queues of the data streams that await processing.

If the system resources are less than the input requirements, some of the input must be shed to bring the load down to within the available capacity. Previous work has addressed the case of memory-limited execution [5][7]. Specifically, it looked at the case when the system can not hold the complete state of the operators (e.g., the tuples that satisfy the window predicate in a sliding window join). Such work, however, does not address the case when input queues are overflowing, which is a result of high CPU utilization. This means CPU limitation can be a cause of memory limitations, even if enough memory is available to store the state of the operators. Hence our focus on CPU-limited execution.

We investigate the problem of load shedding for the streaming window join operator under CPU limitations in detail. We start by revising the unit time model for the CPU cost of executing the join. We study semantic and random load shedding for the Max-Subset goal [5]. In particular, our contributions are:

- We present an accurate unit-time model for the CPU cost of executing a streaming window join.
- For the Max-Subset load shedding goal, we formulate the theoretical problem for shedding load under CPU limitations in an offline scenario. We prove that, unlike the case for memory limited execution [5], the problem is NP-Hard.
- We develop an online strategy for semantic load shedding for the same goal.
- We study a different set of strategies for random load shedding for the Max-Subset goal that decouples the decision of inserting and probing tuples.
- We prove that one such strategy, the Probe-No-Insert, outperforms previously suggested methods for random load shedding. In doing so, we show that having sufficient memory to store the whole state of the join does not help.

2. Cost Model

For the purpose of this work, we are concerned with the sliding window streaming join operator. Concrete definitions of data streams, sliding windows, and the sliding window streaming join are given in [1]. We will refer to the data structure that holds the tuples that satisfy the sliding window predicate on a stream as the *window synopsis* [3].

There are basically four operations performed for every incoming tuple. The four operations can be categorized into

update operations; which include insertion and expiration, and *production* related operations; which include probing and producing the results. The cost of update operations is proportional to the tuple arrival rate (in steady state, every tuple inserted in the window synopsis has to expire), and the cost of production operations is proportional to the output rate of the join. Let C_u be the cost of inserting and expiring a tuple from the window synopsis, C_p be the cost of probing and producing one result tuple, and $\lambda_{o/p}$ be the output rate. The cost of the join can then be expressed as

$$C_{L \bowtie R} = (\lambda_L + \lambda_R) \cdot C_u + \lambda_{o/p} \cdot C_p \quad (1)$$

By assigning the cost of probing and production to output tuples instead of input tuples, the model takes into account the variable cost of probing between different input tuples.

2.1. Goals of Load Shedding

Streaming applications differ in their requirements when faced with the inability to produce the full answer. Different applications require different characteristics in the approximate answer produced by the load shedding scheme. In this work, we focus on the Max-Subset goal, in which it is required to produce a subset of the join of the maximum size allowed by the system's computational resources. We also look at an orthogonal dimension which is the availability of statistical information on the data distributions of the input. If any such information is available, semantic load shedding, in which the strategy intelligently picks tuples to discard based on their values, is possible. Otherwise, random load shedding is the only option. In Section 3, we discuss the load shedding problem under CPU-limited constraints for the Max-subset problem.

3. The Max-subset Goal

We investigate the problem for the goal of obtaining the maximum possible subset of the answer given the CPU budget. The problem is formally defined in [5].

3.1. Semantic Load Shedding

Using the model developed in Section 2, we investigate the optimum load shedding strategy. Since the join query is continuous, the sizes of the input streams are infinite. Hence, modeling of the complete result of join is infeasible. Instead, we model only a prefix of the join result that extends until a specific time in the future. Consider the join $L[T_1] \bowtie R[T_2]$. Assuming we are looking T time units into the future, the total CPU cost budget available for the join operator K is $T * C$ where C is the unit time capacity of the CPU. According to equation 1, there is a cost C_u for maintaining each tuple in the input and a cost C_p for producing an output tuple. We can represent the join result as a bipartite graph in which the set of nodes on the right (left) hand side represents tuples of L (R) and an edge joining two input tuples represents the tuple resulting from their join, with C_u attached to the nodes and C_p to the edges. The optimum algorithm should select a subset of the output tuples such that the cost of their production is less than K while maximizing the number of selected edges. In the full version of this note [1], we concretely define the problem and analyze two variants of an off-line algorithm called the Offline Max-Subset and the Offline Induced Max-Subset algorithms for solving the problem.

The following theorem gives the complexity of these two variants [1]:

Theorem 1. *Both Offline Max-Subset and Offline Induced Max Subset are NP-Hard.*

We can try to approximate the procedure of finding the optimum answer for both variants by a deterministic algorithm that selects edges for inclusion in the answer according to a specific criterion. We propose in [1] two such criteria and methods to quantify them. We also propose a greedy algorithm to approximate the optimum for the offline case. We use the algorithm as a basis for an online semantic load shedding strategy for the CPU-limited execution of the streaming join.

3.2. Random Load Shedding

We now examine the case in which the details of the distribution of the input are unknown. We only assume the input follows the frequency model [7].

The approach previously taken for random load shedding was to find the best setting of random sampling operators applied to the input stream so that the maximum subset is obtained while keeping the load within CPU limits. We shall call this the coin flipping strategy (or CF) [4]. Our contribution is to investigate whether more can be gained by decoupling the update and the production procedures of an incoming tuple in the shedding process. Recall that to execute the join, every incoming tuple has to be inserted in the window of its stream for later matching and it has to probe the window of the opposite stream for matching with tuples that arrived earlier to produce results. The coin flipping approach couples these two procedures by insisting that either the whole contribution of the tuple to the join be taken completely or none is. This is not necessary, since the two procedures are independent. Realizing this, two other approaches arise; namely the Insert-No-Probe (or INP) and the Probe-No-Insert (or PNI) strategies.

Consider the join $L[T_1] \bowtie R[T_2]$ with λ_L and λ_R as the rates of the input streams, with join selectivity f. If the join

is feasible, the output rate is [2]:

$$\lambda_{o/p} = f \cdot \lambda_L \cdot \lambda_R \cdot (T_L + T_R) \quad (2)$$

In the following we describe each shedding strategy in some detail.

Coin Flipping (CF)

In the CF strategy a sampling operator is placed in front of each the two streams, with x_L and x_R being the sampling probability of the one on stream L and R respectively. The coin flipping strategy can be formalized as the following optimization problem [2]:

Max:

$$\lambda_{o/p} = f \cdot \lambda_L \cdot \lambda_R \cdot (T_L + T_R) \cdot x_L \cdot x_R \quad (3)$$

Subject to:

$$(\lambda_L \cdot x_L + \lambda_R \cdot x_R) \cdot C_u + \lambda_{o/p} \cdot C_p \leq 1$$
$$0 \leq x_L, x_R \leq 1 \quad (4)$$

Insert-No-Probe (INP)

In the INP strategy, instead of dropping some of the tuples completely, all incoming tuples are admitted into the window. Then a coin is flipped with probability of probing x_L and x_R for L and R respectively. If the flip is a success, the tuple probes the opposite window, otherwise the tuple is dropped. The strategy calls for the best setting of x_L and x_R while maximizing the output rate. It can be formalized as the following optimization problem:

Max:

$$\lambda_{o/p} = f \cdot \lambda_L \cdot \lambda_R \cdot (T_L \cdot x_R + T_R \cdot x_L) \quad (5)$$

Subject to:

$$(\lambda_L + \lambda_R) \cdot C_u + \lambda_{o/p} \cdot C_p \leq 1$$
$$0 \leq x_L, x_R \leq 1 \quad (6)$$

The intuition behind this strategy is that since we are not constrained in terms of memory, why not keep all the state we can in the hope that the extra kept state produces more tuples. This strategy appears at first sight to be the candidate for delivering the maximum possible subset, since it capitalizes on the asset which is not constrained; in this case memory.

Probe-No-Insert (PNI)

The PNI strategy is the inverse of the previous one. All incoming tuples are allowed to probe the opposite window, and then a coin is flipped on inserting them in the window synopsis for later matching. It can be formalized as follows:

Max:

$$\lambda_{o/p} = f \cdot \lambda_L \cdot \lambda_R \cdot (T_L \cdot x_L + T_R \cdot x_R) \quad (7)$$

Subject to:

$$(\lambda_L \cdot x_L + \lambda_R \cdot x_R) \cdot C_u + \lambda_{o/p} \cdot C_p \leq 1$$
$$0 \leq x_L, x_R \leq 1 \quad (8)$$

Analyzing the three alternatives, we found the PNI strategy to be the superior alternative and the INP the inferior one. The extended version of this note [1] contains the details of the analysis. To understand the intuition behind this result, note that the cost of outputting a single tuple of the join result is divided into production cost, which cannot be reduced, and an amortized cost of maintaining the input tuples in the window. In the INP strategy, the tuples that do not probe the opposite window actually pays the price of maintenance while missing on producing tuples, hence the amortized cost of maintenance is high compared to the other strategies. The PNI strategy allows tuples already inserted in the window to be probed by all the possible tuples they later matche with, hence amortizing their update cost that is already incurred to the maximum possible. The CF strategy misses on some of that by dropping some of the incoming tuples.

Acknowledgements

This research was supported in part by NSF grant ITR 0086002.

Ahmed Ayad is thankful to Raghav Kaushik for fruitful discussions and suggestions.

References

[1] A. Ayad, J. Naughton, S. Wright, and U. Srivastava. Approximating streaming window joins under cpu limitations. Technical report, Department of Computer Sciences, University of Wisconsin - Madison, 2005.

[2] A. Ayad and J. F. Naughton. Static optimization of conjunctive queries with sliding windows over infinite streams. In *SIGMOD Conference*, pages 419–430, 2004.

[3] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*, pages 1–16, 2002.

[4] S. Chaudhuri, R. Motwani, and V. R. Narasayya. On random sampling over joins. In *SIGMOD Conference*, pages 263–274, 1999.

[5] A. Das, J. Gehrke, and M. Riedewald. Approximate join processing over data streams. In *SIGMOD Conference*, pages 40–51, 2003.

[6] J. Kang, J. F. Naughton, and S. Viglas. Evaluating window joins over unbounded streams. In *ICDE*, pages 341–352, 2003.

[7] U. Srivastava and J. Widom. Memory-limited execution of windowed stream joins. In *VLDB*, pages 324–335, 2004.