Monitoring Stream Properties for Continuous Query Processing[*]

Utkarsh Srivastava, Shivnath Babu, and Jennifer Widom

Computer Science Department
Stanford University
{usriv,shivnath,widom}@cs.stanford.edu

We are developing a general-purpose *Data Stream Management System* for processing continuous queries over multiple continuous data streams [MW$^+$03]. When a new continuous query is registered, our query optimizer creates an initial *query plan* (possibly merged with existing plans for previously registered queries), and allocates initial *resources*, such as memory for join or aggregation *synopses* [GGR02] and for input and inter-operator queues. Dynamically, plans may be altered, resources may be reallocated, and scheduling decisions are made, all based on stream data and arrival characteristics, query plan execution behavior, and resource utilization.

Our query processor can be thought of as two separate components: a *Query Execution* component and a *Property Monitoring* component. Properties used by the Query Execution component, such as stream characteristics and plan operator behavior, are monitored by the Property Monitoring component. Of course, monitoring of operator behavior would be coupled with query execution. When significant changes are detected, the Property Monitoring component notifies the Query Execution component, which may choose to alter query plans or resource management accordingly. Note that our approach to dynamic query optimization is significantly more coarse-grained than *Eddies* [AH00], which makes decisions on a tuple-by-tuple level. Unless stream characteristics or query load fluctuates wildly, we expect our plans and resource allocation to migrate slowly over time, responding to clear and consistent changes in monitored properties.

Let us focus now on properties related to stream data and arrival characteristics. As a simple example of property-based query execution, if a stream S is known to be roughly sorted on an attribute X, and X is a join attribute or is the grouping attribute in an aggregate query, much less memory is required than for a plan that accommodates randomly arriving values of X. More subtly, the best plan for an n-way sliding-window stream join depends heavily on the relative arrival rates of the streams involved in the join [VNB02], and ”traditional” data statistics such as histograms, but over recent stream elements [DGIM02]. Scheduling decisions depend on stream burstiness and data characteristics [BBDM03].

In [BW02] we studied three specific stream properties and how they affect query execution: *sortedness*, *clustering*, and *stream-based referential integrity*. We identified an *adherence parameter* k for each type of constraint, which indicates how closely a stream (or pair of streams, for referential integrity) adheres to the constraint. We presented a query plan generation and resource allocation algorithm that minimizes memory

[*]This work was supported by the National Science Foundation under grants IIS-9817799 and IIS-0118173 and by a Stanford Graduate Fellowship from Sequoia Capital.

use based on k-constraints, where smaller k's require less memory. To make use of the algorithm, specific k-constraints must be declared in advance (often unrealistic) or monitored. If k values change over time, or constraints come and go entirely, it is straightforward for our algorithm to adjust query plans and memory allocation, making the approach a perfect fit for our coupled Query Execution and Property Monitoring model.

In a different aspect of query processing, our semantics of continuous queries requires that all input stream elements with timestamps $\leq \tau$ have arrived before we can determine the result of a continuous query at time τ [ABW02]. *Query heartbeat generation* is the process of maintaining the maximum timestamp τ' such that no further input stream element can have timestamp $\leq \tau'$. We have developed efficient algorithms for heartbeat generation based on stream properties such as k-sortedness of timestamp attributes and timestamp k-skew across multiple streams—properties very similar to those used by our constraint-based query processing algorithm described in the previous paragraph.

Stream characteristics may change over time, sometimes very rapidly, so we want to monitor relevant properties continuously. However it is generally too expensive to monitor every relevant property as each element arrives (or is processed by a query operator). Fortunately, in many cases query processing decisions are based on assumptions that need not be exact, so approximate property monitoring is usually acceptable. (Even in the case of constraint-based memory allocation and query heartbeats, approximate values of k result in approximate query results with imprecision proportional to the error in k [BW02].) We might even incorporate a "meta-resource-allocator" that allocates resources to the Property Monitor component dynamically, making use of surplus resources or devoting extra resources when monitoring is more crucial. Our general goal in Property Monitoring is to provide the best estimate of relevant properties with the resources available.

In the remainder of this short paper we delve into details for a specific property: sortedness. We provide an algorithm that fits our paradigm exactly—it determines the adherence parameter k for sortedness of a stream attribute X, and does so approximately in a very time and space-efficient manner. As an example scenario, consider network measurements transmitted via *UDP*. Since UDP does not guarantee in-order delivery, measurements can arrive out of order at a Data Stream Management System used for network monitoring, but the degree of unsortedness depends on the network latency and is likely to vary over time. Many network monitoring queries use time-based windows or can otherwise take advantage of timestamp-sorted streams [CGJ$^+$02], so monitoring sortedness will result in better query plans and resource management.

Monitoring Stream Sortedness

We present an efficient algorithm for estimating stream sortedness dynamically. Our definition of *sortedness* of a stream S on an attribute X belonging to an ordered domain is as follows: Let $X_0, X_1, \ldots, X_n$ be the sequence of X values in arrival order on stream S seen so far. S is k-sorted on X in a window of size W if $n - W < i, j \leq n$ and $j - i > k$ together imply $X_j \geq X_i$. We are interested in estimating the minimum value of k for which S is k-sorted on X. Intuitively, k is the maximum distance over which any inversion occurs in the last W tuples of S. Typically the window size for which we shall estimate k will be much larger than the distance over which any inversion is expected to occur. Note that 0-sortedness captures perfect sortedness. The k-sortedness property can be used to obtain significant space savings during query execution as studied in [BW02].

Estimating k-sortedness accurately requires $O(W)$ space and $O(\log W)$ per-tuple update time. Thus we give an algorithm that computes an approximate estimate of k using $O(1)$ space and $O(1)$ per-tuple update time such that the number of violations for our estimate is expected to be small. A pair X_i, X_j violates our estimated k if $n - W < i, j \leq n$ and $j - i > k$ and $X_j < X_i$.

The basic idea behind our algorithm is to assume a parametrized probabilistic model for the arrival of X values in the stream which is reasonably general. We then estimate the parameters of our model based on the stream prefix seen so far by a *maximum likelihood* [DGL96] approach. Finally we map the model parameters to an estimate of k such that the number of violations is expected to be small.

Before we describe our general model to capture unsortedness, let us examine the arrival model of a stream S that is perfectly sorted on attribute X and in which the X values are equispaced at intervals of r. S satisfies the following arrival model:

$$P(X_i = j) = \begin{cases} 1 & \text{if } j = x_0 + ri \\ 0 & \text{otherwise} \end{cases}$$

Deviation from perfect sortedness can often be captured via local reorderings on a sorted stream. (An example is when the source of the stream is itself emitting in sorted order but the tuples are reordered over a UDP network.) Thus our general model to capture k-sortedness adds some noise to this distribution. Specifically our model assumes that X_i's are distributed normally with mean $x_0 + ri$ and variance σ^2, i.e., $X_i \sim \mathcal{N}(x_0 + ri, \sigma)$. Note that we no longer require that X values be equispaced. Intuitively in the above model, x_0 is the value of X in the first tuple in S, r is the average value with which X is increasing with each tuple, and σ measures the potential for reordering that causes deviation from perfect sortedness.

The parameters of the model are chosen so as to maximize the *log-likelihood* [DGL96] of the observations $X_0, X_1, \ldots, X_n$ given by:

$$\ell(x_0, r, \sigma) = -\sum_{i=0}^{n} \alpha^{n-i} \left[\frac{1}{2} \left(\frac{X_i - x_0 - ri}{\sigma} \right)^2 + \ln(\sigma) \right]$$

where $\alpha < 1$ is a discounting factor in order to give a higher weight to recent observations. The value of α to be used depends on the window size W. Also, a lower value of α makes the algorithm more adaptive so that it responds quickly to changes in k.

The exact expressions for the parameters that maximize the log-likelihood function can be derived in a straightforward manner. Here we only note that those values can be expressed in terms of the following sums:

$$\sum_{i=0}^{n} \left(\alpha^{-i},\ i\alpha^{-i},\ i^2\alpha^{-i},\ X_i\alpha^{-i},\ iX_i\alpha^{-i},\ X_i^2\alpha^{-i} \right)$$

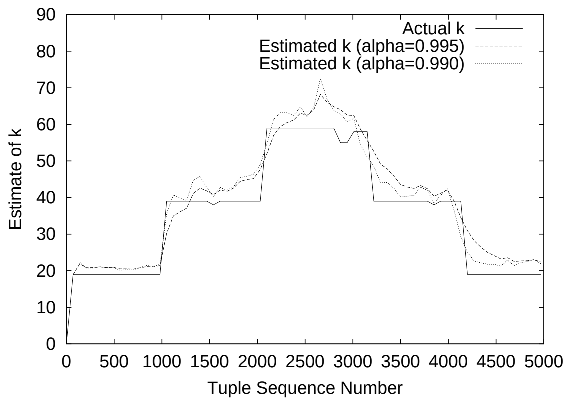

Figure 1: Experimental results

Since these sums may be computed incrementally, it follows that the parameters of our model can be computed using $O(1)$ state and with an update time of $O(1)$ per stream element.

To map the parameters of the model to an estimate for k, given an error bound ϵ choose c such that $P[X_i > x_0+ri+c\sigma] \leq \epsilon/2$. Let E_1 be the event that $X_i > x_0+ri+c\sigma$. Let $k = 2c\sigma/r$. Let E_2 be the event that $X_{k+i} < x_0 + ri + c\sigma$. Since $X_{k+i} \sim \mathcal{N}(x_0 + ri + 2c\sigma, \sigma)$ we have $P[E_2] \leq \epsilon/2$. For this estimate of k to be violated, i.e., for $X_{k+i} < X_i$, either E_1 or E_2 must occur. Thus, $P[\text{violation}] \leq P[E_1] + P[E_2] \leq \epsilon$. Thus $k = 2c\sigma/r$, where c can be chosen so as to make the probability of a violation ϵ as small as required. A value of $c = 2$ gives $\epsilon \approx 0.05$, an accuracy of about 95%.

The above high confidence estimate for k has been obtained assuming that the stream arrival adheres perfectly to our model. However we have performed some preliminary experiments with synthetic streams showing that our technique works well, even for streams that are not generated according to the model we assume. A sample run of our algorithm with $c = 1.5$ on a synthetically-generated stream is shown in Figure 1. The actual k was calculated over a window of size 200. To measure the adaptivity of our algorithm, k was changed at regular intervals. It can be seen that the k estimated by our algorithm closely tracks the actual k. Note that the lower value of α makes the algorithm respond quickly to changes in k, but at the cost of more fluctuations in the estimate.

Our algorithm handles increasing and decreasing attributes automatically. Thus we need not hard code whether we are looking for increasing or decreasing attributes. For a decreasing attribute we get $r < 0$ which leads to $k < 0$. This can be interpreted as S being $|k|$-sorted on X in decreasing order.

Unsorted streams are also handled identically. If the X values are randomly distributed and follow no particular order, we get $r \approx 0$. Thus for an unsorted stream, $|k|$ tends to ∞. Thus this algorithm can be used to determine if at all an optimization based on k-sortedness of the stream could be useful. A very large value of k indicates that such an optimization cannot be employed.

References

[ABW02] A. Arasu, S. Babu, and J. Widom. An abstract semantics and concrete language for continuous queries over streams and relations. Technical report, Stanford University Database Group, November 2002. Available at http://dbpubs.stanford.edu/pub/2002-57.

[AH00] R. Avnur and J.M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data*, pages 261–272, May 2000.

[BBDM03] B. Babcock, S. Babu, M. Datar, and R. Motwani. Chain: Operator scheduling for memory minimization in data stream systems. In *Proc. of the 2003 ACM SIGMOD Intl. Conf. on Management of Data*, June 2003. (To appear).

[BW02] S. Babu and J. Widom. Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. Technical report, Stanford University Database Group, November 2002. Available at http://dbpubs.stanford.edu/pub/2002-52.

[CGJ$^+$02] C. Cranor, Y. Gao, T. Johnson, V. Shkapenyuk, and O. Spatscheck. Gigascope: high performance network monitoring with an sql interface. In *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, page 623, May 2002.

[DGIM02] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. In *Proc. of the 2002 Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 635–644, 2002.

[DGL96] Luc Devroye, Laszlo Gyorfi, and Gabor Lugosi. *A Probabilistic Theory of Pattern Recognition*. Springer-Verlag, New York, 1996.

[GGR02] M.N. Garofalakis, J. Gehrke, and R. Rastogi. Querying and mining data streams: You only get one look (*tutorial*). In *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, page 635, May 2002.

[MW$^+$03] R. Motwani, J. Widom, et al. Query processing, approximation, and resource management in a data stream management system. In *Proc. First Biennial Conf. on Innovative Data Systems Research (CIDR)*, January 2003.

[VNB02] S. Viglas, J. F. Naughton, and J. Burger. Maximizing the output rate of multi-join queries over streaming information sources. *manuscript*, 2002. Available at http://www.cs.wisc.edu/niagara/Publications.html.