ADAPTIVE STREAM PROCESSING

Zachary Ives
Computer and Information Science Department
University of Pennsylvania
Philadelphia, PA 19104-6389
zives@cis.upenn.edu

SYNONYMS

Adaptive query processing

DEFINITION

When querying long-lived data streams, the characteristics of the data may change over time or data may arrive in bursts — hence, the traditional model of optimizing a query prior to executing it is insufficient. As a result, most data stream management systems employ feedback-driven *adaptive stream processing*, which continuously re-optimizes the query execution plan based on data and stream properties, in order to meet certain performance or resource consumption goals. Adaptive stream processing is a special case of the more general problem of *adaptive query processing*, with the special property that intermediate results are bounded in size (by stream windows), but where query processing may have quality-of-service constraints.

HISTORICAL BACKGROUND

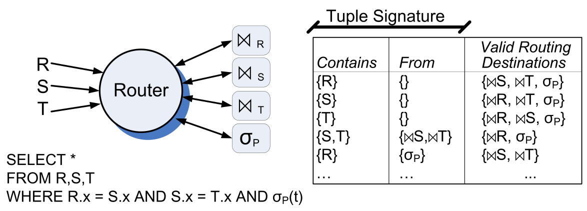
The field of adaptive stream processing emerged in the early 2000s, as two separate developments converged. *Adaptive* techniques for database query processing had become an area of increasing interest as Web and integration applications exceeded the capabilities of conventional static query processing [10]. Simultaneously, a number of data stream management systems [1, 6, 12, 8] were emerging, and each of these needed capabilities for query optimization. This led to a common approach of developing feedback-based re-optimization strategies for stream query computation. In contrast to Web-based adaptive query processing techniques, the focus in adaptive stream processing has especially been on maintaining quality of service under overload conditions.

SCIENTIFIC FUNDAMENTALS

Data stream management systems (DSMSs) typically face two challenges in query processing. First, the data to be processed comes from remote feeds that may be subject to significant variations in distribution or arrival rates over the lifetime of the query, meaning that no single query evaluation strategy may be appropriate over the entirety of execution. Second, DSMSs may be *underprovisioned* in terms of their ability to handle bursty input at its maximum rate, and yet may still need to meet certain *quality-of-service* or resource constraints (e.g., they may need to ensure data is processed within some latency bound). These two challenges have led to two classes of adaptive stream processing techniques: those that attempt to *minimize the cost* of computing query results from the input data (the problem traditionally faced by query optimization), and those that attempt to manage query processing, possibly at reduced accuracy, in the presence of *limited resources.* This article provides an overview of significant work in each area.

Minimizing Computation Cost

The problem of adaptive query processing to minimize computation cost has been well-studied in a variety of settings [10]. What makes the adaptive stream processing setting unique (and unusually tractable) is the fact that joins are performed over *sliding windows* with size bounds: As the data stream exceeds the window size, old data values are expired. This means intermediate state within a query plan operator has constant maximum size; as opposed to being bounded by the size of the input data. Thus a windowed join operator can be modeled as a pair of filter operators, each of which joins its input with the bounded intermediate state produced from the other



```
SELECT *
FROM R,S,T
WHERE R.x = S.x AND S.x = T.x AND σP(t)
```

Contains	*From*	*Valid Routing Destinations*
{R}	{}	{$\bowtie$S, $\bowtie$T, σ_P}
{S}	{}	{$\bowtie$R, $\bowtie$T, σ_P}
{T}	{}	{$\bowtie$R, $\bowtie$S, σ_P}
{S,T}	{$\bowtie$S,$\bowtie$T}	{$\bowtie$R, σ_P}
{R}	{σ_P}	{$\bowtie$S, $\bowtie$T}
…	…	…

Figure 1: Illustration of eddy with SteMs.

input. Optimization of joins in data stream management systems becomes a minor variation on the problem of optimizing selection or filtering operators; hence certain theoretical optimality guarantees can actually be made.

Eddies

Eddies [2, 14, 11] are composite dataflow operators that model select-project-join expressions. An eddy consists of a *tuple router*, plus a set of primitive *query operators* that run concurrently and each have input queues. Eddies come in several variations; the one proposed for distributed stream management uses *state modules* (SteMs) [14, 11]. Figure 1 shows an example of such an eddy for a simplified stream SQL query, which joins three streams and applies a selection predicate over them.

Eddy creation. The eddy is created prior to execution by an optimizer: every selection operation (σ_P in the example) is converted to a corresponding operator; additionally, each base relation to be joined is given a *state module*, keyed on the join attribute, to hold the intermediate state for each base relation [14] ($\bowtie_R$, $\bowtie_S$, $\bowtie_T$)[1] In general, the state module can be thought of as one of the hash tables within a symmetric or pipelined hash join. The optimizer also determines whether the semantics of the query force certain operators to execute before others. Such constraints are expressed in an internal routing table, illustrated on the right side of the figure. As a tuple is processed, it is annotated with a *tuple signature* specifying what input streams' data it contains and what operator may have last modified it. The routing table is a map from the tuple signature to a set of *valid routing destinations*, those operators that can successfully process a tuple with that particular signature.

Query execution / tuple routing. Initially, a tuple from an input data stream (R, S, or T) flows into the eddy router. The eddy (1) adds the data to the associated SteM or SteMs, and (2) consults the routing table to determine the set of possible destination operators. It then chooses a destination (using a *policy* to be described later) and sends the tuple to the operator. The operator then either *filters* the tuple, or *produces* one or more output tuples, as a result of applying selection conditions or joining with the data within a SteM. Output tuples are marked as having been processed by the operator that produced them. If they have been processed by all operators, they will be sent to the query output, and if not, they will be sent back to the eddy's router and to one of the remaining operators.

Routing policies. The problem of choosing among alternate routing destinations has been addressed with a variety of strategies.

Tickets and lottery scheduling [2]. In this scheme, each operator receives a *ticket* for each tuple it receives from the router, and it returns the ticket each time it outputs a tuple to the router. Over time, each operator is expected to have a number of tickets proportional to $(1-p)$ where p is the operator's selectivity. The router holds a *lottery* among valid routing destinations, where each operator's chance of winning is proportional to its number of tickets. Additionally, as a flow control mechanism, each operator has an input queue, and if this queue fills, then the operator may not participate in the lottery.

Deterministic with Batching [9]. A later scheme was developed to reduce the per-tuple overhead of eddies by choosing destinations for batches of tuples. Here, each operator's selectivity is explicitly monitored and each predicate is assumed to be independent. Periodically, a *rank ordering* algorithm is used to choose a destination for a *batch* of tuples: the rank ordering algorithm sorts predicates in decreasing order of $c_i/(1-p_i)$, where c_i is the cost of the applying predicate σ_i and p_i is its selectivity.

Content-based Routing [7] (CBR) attempts to learn correlations between attribute values and selectivities. Using sampling, the system determines for each operator the attribute most strongly correlated with its selectivity — this

[1]If a base relation appears with multiple different join attributes, then it may require multiple SteMs.

is termed the *classifier attribute*. CBR then builds a table characterizing all operators' selectivities for different values of each classifier attribute. Under this policy, when the eddy needs to route a tuple, it first looks up the tuple's classifier attribute values in the table and determines the destination operators' selectivities. It routes the tuple probabilistically, choosing a next operator with probability inversely proportional to its selectivity.

Other Optimization Strategies
An alternative strategy that does not use the eddies framework is the *adaptive greedy* [5] (A-greedy) algorithm. A-greedy continuously monitors the selectivities of query predicates using a *sliding window profile*, a table with one Boolean attribute for each predicate in the query, and sampling. As a tuple is processed by the query, it may be chosen for sampling into the sliding window profile — if so, it is tested against every query predicate. The vector of Boolean results is added as a row to the sliding window profile. Then the sliding window profile is then used to create a *matrix view* $V[i, j]$ containing, for each predicate σ_i, the number of tuples in the profile that satisfy $\sigma_1 \ldots \sigma_{i-1}$ but not σ_j. From this matrix view, the reoptimizer seeks to maintain the constraint that the ith operation over an input tuple must have the lowest cost / selectivity ratio $c_i/(1 - p(S_i|S_1, \ldots, S_{i-1}))$. The overall strategy has one of the few performance guarantees in the adaptive query processing space: if data properties were to converge, then performance would be within a factor of 4 of optimal [5].

Managing Resource Consumption

A common challenge in data stream management systems is limiting the use of resources — or accommodating limited resources while maintaining quality of service, in the case of bursty data. We discuss three different problems that have been studied: load shedding to ensure input data is processed by the CPU as fast as it arrives, minimizing buffering and memory consumption during data bursts, and minimizing network communication with remote streaming sites.

Load shedding allows the system to selectively drop data items to ensure it can process data as it arrives. Both the Aurora and STREAM DSMSs focused heavily on adaptive load shedding.

Aurora. In the Aurora DSMS [15], load shedding for a variety of query types are supported: the main requirement is that the user has a *utility function* describing the value of output data relative to how much of it has been dropped. The system seeks to place load shedding operators in the query plan in a way that maximizes the user's utility function while the system achieves sufficient throughput. Aurora precomputes conditional load shedding plans, in the form of a *load shedding road map* (LRSM) containing a sequence of plans that shed progressively more load; this enables the runtime system to rapidly move to strategies that shed more or less load.

LRSMs are created using the following heuristics: first, load shedding points are only inserted at data input points or at points in which data is split to two or more operators. Second, for each load shedding point, a *loss/gain ratio* is computed: this is the reduction in output utility divided by the gain in cycles, $R(p \cdot L - D)$, where R is the input rate into the drop point, p is the ratio of tuples to be dropped, L is the amount of system load flowing from the drop point, and D is the cost of the drop operator. Drop operators are injected at load shedding points in decreasing order of loss/gain ratio. Two different types of drops are considered using the same framework: *random drop*, in which an operator is placed in the query plan to randomly drop some fraction p of tuples; and *semantic drop*, which drops the p tuples of lowest utility. Aurora assumes for the latter case that there exists a utility function describing the relative worth of different attribute values.

Stanford STREAM. The Stanford STREAM system [4] focuses on aggregate (particularly SUM) queries. Again the goal is to process data at the rate it arrives, while minimizing the inaccuracy in query answers: specifically, the goal is to minimize the *maximum relative error across all queries*, where the relative error of a query is the difference between actual and approximate value, divided by the actual value.

A Statistics Manager monitors computation and provides estimates of each operator's selectivity and its running time, as well as the mean value and standard deviation of each query q_i's aggregate operator. For each q_i, STREAM computes an error threshold C_i, based on the mean, standard deviation, and number of values. (The results are highly technical so the reader is referred to [4] for more details.) A sampling rate P_i is chosen for query q_i that satisfies $P_i \geq C_i/\epsilon_i$, where ϵ_i is the allowable relative error for the query.

As in Aurora's load shedding scheme, STREAM only inserts load shedding operators at the inputs or at the start of shared segments. Moreover, if a node has a set of children who all need to shed load, then a portion of the load shedding can be "pulled up" to the parent node, and all other nodes can be set to shed some amount of additional load *relative* to this. Based on this observation, STREAM creates a query dataflow graph in which each path from source to sink initially traverses through a load shedding operator whose sampling rate is determined by the desired error rate, followed by additional load shedding operators whose sampling rate is expressed relative to that first operator. STREAM iterates over each path, determines a sampling rate for the initial load shedding operator to satisfy the load constraint, and then computes the maximum relative error for any query. From this, it can set the load shedding rates for individual operators.

Memory minimization. STREAM also addresses the problem of minimizing the amount of space required to buffer data in the presence of burstiness [3]. The Chain algorithm begins by defining a *progress chart* for each operator in the query plan: this chart plots the relative size of the operator output versus the time it takes to compute. A point is plotted at time 0 with the full size of the input, representing the start of the query; then each operator is given a point according to its cost and relative output size. Now a *lower envelope* is plotted on the progress chart: starting with the initial point at time 0, the *steepest* line is plotted to any operator to the right of this point; from the point at the end of the first line, the next steepest line is plotted to a successor operator; etc. Each line segment (and the operators whose points are plotted beside it) represents a chain, and operators within a chain are scheduled together. During query processing, at each time "tick," the scheduler considers all tuples that have been output by any chain. The tuple that lies on the segment with *steepest slope* is the one that is scheduled next; as a tie-breaker, the earliest such tuple is scheduled. This Chain algorithm is proven to be near-optimal (differing by at most one unit of memory per operator path for queries where selectivity is at most 1).

Minimizing communication. In some cases, the constrained resource is the network rather than CPU or memory. Olston et al. [13] develop a scheme for reducing network I/O for AVERAGE queries, by using accuracy bounds. Each remote object O is given a *bound width* w_O: the remote site will only notify the central query processor if O's value V falls outside this bound. Meanwhile, the central site maintains a *bound cache* with the last value and the bound width for every object.

If given a *precision constraint* δ_j for each query Q_j, then if the query processor is to provide query answers within δ_j, the sum of the bound widths for the data objects of Q_j must not exceed δ_j times the number of objects. The challenge lies in the selection of widths for the objects.

Periodically, the system tries to tighten all bounds, in case values have become more stable; objects whose values fall outside the new bounds get reported back to the central site. Now some of those objects' bounds must be loosened in a way that maintains the precision constraints over all queries. Each object O is given a *burden score* equal to $c_O/(p_O w_O)$, where c_O is the cost of sending the object, w_O is its bound width, and p_O is the frequency of updates since the previous width adjustment. Using an approximation method based on an iterative linear equation solver, Olston et al. compute a *burden target* for each query, i.e., the lowest overall burden score required to always meet the query's precision constraint. Next, each object is assigned a *deviation*, which is the maximum difference between the object's burden score and any query's burden target. Finally, a queried objects' bounds are adjusted in decreasing order of deviation, and each object's bound is increased by the largest amount that still conforms to the precision constraint for every query.

KEY APPLICATIONS

Data stream management systems have seen significant adoption in areas such as sensor monitoring and processing of financial information. When there are associated quality-of-service constraints that might require load shedding, or when the properties of the data are subject to significant change, adaptive stream processing becomes vitally important.

FUTURE DIRECTIONS

One of the most promising directions of future study is how to best use a combination of offline modeling, selective probing (in parallel with normal query execution), and feedback from query execution to find optimal strategies quickly. Algorithms with certain optimality guarantees are being explored in the online learning and theory communities (e.g., the k-armed bandit problem), and such work may lead to new improvements in adaptive stream processing.

CROSS REFERENCE

Stream processing.
Distributed stream.
Query processor or query execution engine.

RECOMMENDED READING

[1] Daniel J. Abadi, Don Carney, Ugur Cetintemel, Mitch Cherniack, Christain Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *VLDB J.*, 12(2), August 2003.

[2] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD*, 2000.

[3] Brian Babcock, Shivnath Babu, Mayur Datar, and Rajeev Motwani. Chain: Operator scheduling for memory minimization in data stream systems. In *SIGMOD*, 2003.

[4] Brian Babcock, Mayur Datar, and Rajeev Motwani. Load shedding for aggregation queries over data streams. In *ICDE*, 2004.

[5] Shivnath Babu, Rajeev Motwani, Kamesh Munagala, Itaru Nishizawa, and Jennifer Widom. Adaptive ordering of pipelined stream filters. In *SIGMOD*, 2004.

[6] Magdalena Balazinska, Hari Balakrishnan, and Michael Stonebraker. Demonstration: Load management and high availability in the medusa distributed stream processing system. In *SIGMOD*, 2004.

[7] Pedro Bizarro, Shivnath Babu, David J. DeWitt, and Jennifer Widom. Content-based routing: Different plans for different data. In *VLDB*, 2005.

[8] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Fred Reiss, and Mehul A. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[9] Amol Deshpande. An initial study of overheads of eddies. *SIGMOD Rec.*, 33(1), 2004.

[10] Amol Deshpande, Zachary Ives, and Vijayshankar Raman. Adaptive query processing. *Foundations and Trends in Databases*, 2007.

[11] Sam Madden, Mehul A. Shah, Joseph M. Hellerstein, and Vijayshankar Raman. Continuously adaptive continuous queries over streams. In *SIGMOD*, 2002.

[12] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, resource management, and approximation in a data stream management system. In *CIDR*, 2003.

[13] Chris Olston, Jing Jiang, and Jennifer Widom. Adaptive filters for continuous queries over distributed data streams. In *SIGMOD*, 2003.

[14] Vijayshankar Raman, Amol Deshpande, and Joseph M. Hellerstein. Using state modules for adaptive query processing. In *ICDE*, 2003.

[15] Nesime Tatbul, Ugur Cetintemel, Stanley B. Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. In *VLDB*, 2003.