of the Scheduler is to minimize queue time and optimize executive efficiency.

## Cross-References

▸ Activity

# Scheduling Strategies for Data Stream Processing

Mohamed Sharaf[1] and Alexandros Labrinidis[2]
[1]Electrical and Computer Engineering, University of Toronto, Toronto, ON, Canada
[2]Department of Computer Science, University of Pittsburgh, Pittsburgh, PA, USA

## Synonyms

Continuous query scheduling; Operator scheduling; Scheduling policies

## Definition

In a Data Stream Management System (DSMS), data arrives in the form of continuous streams from different data sources, where the arrival of new data triggers the execution of multiple continuous queries (CQs). The order in which CQs are executed in response to the arrival of new data is determined by the CQ scheduler. Thus, one of the main goals in the design of a DSMS is the development of scheduling policies that leverage CQ characteristics to optimize the DSMS performance.

## Historical Background

The growing need for *monitoring applications* [8] has forced an evolution on data processing paradigms, moving from Database Management Systems (DBMSs) to Data Stream Management Systems (DSMSs) [4, 11]. Traditional DBMSs employ a store-and-then-query data processing paradigm, where data are stored in the database and queries are submitted by the users to be answered in full, based on the current snapshot of the database. In contrast, in DSMSs, monitoring applications register continuous queries which continuously process unbounded data streams looking for data that represent events of interest to the end-user.

The data stream concept permeated the data management research community in the mid- to late 90's, with general-purpose research prototypes of data stream management systems materializing shortly afterwards, for example Aurora [8], TelegraphCQ [10] and STREAMS [5].

Scheduling is one of the fundamental research challenges for effective data stream management systems; as such, it has received a lot of attention, with early works on scheduling in 2003 [2, 9].

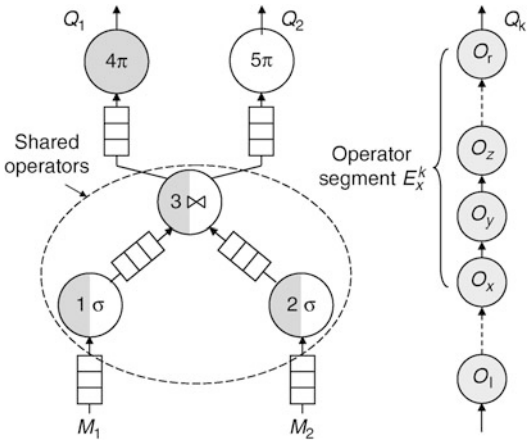
## Foundations

### System Model

A continuous query evaluation plan can be conceptualized as a data flow tree [2, 8], where the nodes are operators that process tuples and edges represent the flow of tuples from one operator to another (Fig. 1). An edge from operator $O_x$ to operator $O_y$ means that the output of $O_x$ is an input to $O_y$. Each operator is associated with a *queue* where input tuples are buffered until they are processed.

Multiple queries with common sub-expressions are usually merged together to eliminate the repetition of similar operations. For example, Fig. 1 shows the global plan for two queries $Q_1$ and $Q_2$. Both queries operate on data streams $M_1$ and $M_2$ and they share the common sub-expression represented by operators $O_1$, $O_2$ and $O_3$, as illustrated by the half-shaded pattern for these operators.

A *single-stream query* $Q_k$ has a single *leaf* operator $Q_l^k$ and a single *root* operator $Q_r^k$, whereas a *multi-stream* query has a single root operator and more than one leaf operators. In a query plan




**Scheduling Strategies for Data Stream Processing, Fig. 1** Continuous queries plans

$Q_k$, an *operator segment* $E^k_{x,y}$ is the sequence of operators that starts at $O^k_x$ and ends at $O^k_x$. If the last operator on $E^k_{x,y}$ is the root operator, then that operator segment is simply denoted as $E^k_x$. For example, in Fig. 1, $E^1_1 =< O_1, O_3, O_4 >$, whereas $E^2_1 =< O_1, O_3, O_5 >$.

In a query, each operator $O_x{}^k$ (or simply $O_x$) is associated with two parameters:

1. *Processing cost* or *Processing time* ($c_x$ ) is the amount of time needed to process an input tuple.
2. *Selectivity* or *Productivity* ($s_x$ ) is the number of tuples produced after processing one tuple for $c_x$ time units. $s_x$ is less than or equal to 1 for a filter operator and it could be greater than 1 for a join operator.

## Multiple CQ Scheduling

At the arrival of new data, the MCQ scheduler decides the execution order of CQs, or more precisely, the execution order of operators within CQs. The execution order is decided with the objective of optimizing the DSMS performance under certain metrics. Towards this, the scheduler assigns a priority to each operator and operators are executed according to these priorities.

For a single-stream query $Q_k$ which consists of operators $O^k_l, \ldots, O^k_x, O^k_y, \ldots, O^k_r >$ (Fig. 1), the function for computing the priority of operator $O^k_x$ typically involves one or more of the following parameters:

- *Operator Global Selectivity* ($S^k_x$) *is the number of tuples produced at the root* $O^k_r$ *after processing one tuple along operator segment* $E^k_x$.

$$S^k_x = s^k_x \times s^k_y \times \cdots \times s^k_r$$

- *Operator Global Average Cost* ($\overline{C}^k_x$) is the expected time required to process a tuple along an operator segment $C^k_x$.

$$\overline{C}^k_x = \left(c^k_x\right) + \left(c^k_y \times s^k_x\right) + \ldots + \left(c^k_r \times s^k_{r-1} \times \cdots \times s^k_x\right)$$

If $O^k_x$ is a leaf operator ($x = l$), when a processed tuple actually satisfies all the filters in $E^k_l$, then $\overline{C}^k_l$ represents the ideal total processing cost or time incurred by any tuple *produced* or *emitted* by query $Q_k$. In this case, $\overline{C}^k_l$ is denoted as $T_k$ :

- *Tuple Processing Time* ($T_k$) is the ideal total processing cost required to produce a tuple by query $Q_k$.

$$T_k = c^k_l + \cdots + c^k_x + c^k_y + + c^k_r$$

The exact priority function depends on the performance metric to optimize, and in turn on the employed scheduling strategy.

## Metrics and Strategies

*Response Time*: Processing a tuple by a CQ might lead to discarding it (if it does not satisfy some filter predicate) or it might lead to producing one or more tuples at the output, which means that the input tuple represents an event of interest to the user who registered the CQ. Clearly, in DSMSs, it is more appropriate to define response time from a data/event perspective rather than from a query perspective as in traditional DBMSs. Hence, the

*tuple response time* or *tuple latency* is defined as follows:

**Definition 1**
*Tuple response time, $R_i$, for tuple $t_i$ is $R_i = D_i - A_i$, where $A_i$ is $t_i$'s arrival time and $D_i$ is $t_i$'s output time. Accordingly, the average response time for N tuples is: $\frac{1}{N}\sum_{i=1}^{N} R_i$.*

For a single CQ over multiple data streams, the *Rate-based* policy (*RB*) has been shown to improve the average response time of tuples processed by that CQ [17].

For multiple CQs, the Aurora DSMS [9], uses a two-level scheduling strategy where *Round Robin (RR)* is used to schedule queries and *RB* is used to schedule operators within the query. The work in [14] proposes the *Highest Rate* policy (*HR*) which extends the *RB* to schedule both queries and operators. Basically, *HR* views the network of multiple queries as a set of operators and at each scheduling point it selects for execution the operator with the highest priority (i.e., output rate).

Specifically, under *HR*, each operator $O_x^k$ is assigned a value called *global output rate* ($GR_x^k$). The output rate of an operator is basically the expected number of tuples produced per time unit due to processing one tuple by the operators along the operator segment starting at $O_x^k$ all the way to the root $O_r^k$. Formally, the output rate of operator $O_x^k$ is defined as follows:

$$GR_x^k = \frac{S_x^k}{\overline{C}_x^k} \tag{1}$$

where $S_x^k$ and $\overline{C}_x^k$ are the operator's global selectivity and global average cost as defined above. The intuition underlying *HR* is to give higher priority to operator paths that are both productive and inexpensive. In other words, the highest priority is given to the operator paths with the minimum latency for producing one tuple.

*Slowdown*: Under a heterogeneous workload, the processing requirements for different tuples may vary significantly and average response time is not an appropriate metric, since it cannot relate the time spent by a tuple in the system to its processing requirements. Given this realization, other on-line systems with heterogeneous workloads such as DBMSs, OSs, and Web servers have adopted *average slowdown* or *stretch* [13] as another metric. This motivated considering the stretch metric in [14].

The definition of slowdown was initiated by the database community in [12] for measuring the performance of a DBMS executing multi-class workloads. Formally, the slowdown of a job is the ratio between the time a job spends in the system to its processing demands [13]. In a DSMS, the slowdown of a tuple is defined as follows [14]:

**Definition 2**
*The slowdown, $H_i$, for tuple $t_i$ produced by query $Q_k$ is $H_i = \frac{R_i}{T_k}$, where $R_i$ is $t_i$'s response time and $T_k$ is its ideal processing time. Accordingly, the average slowdown for N tuples is: $\frac{1}{N}\sum_{i=1}^{N} H_i$.*

Intuitively, in a general purpose DSMS where all events are of equal importance, a simple event (i.e., an event detected by a low-cost CQ) should be detected faster than a complex event (i.e., an event detected by a high-cost CQ) since the latter contributes more to the load on the DSMS.

The *HR* policy schedules jobs in descending order of output rate which might result in a high average slowdown because a low-cost query can be assigned a low priority since it is not productive enough. Those few tuples produced by this query will all experience a high slowdown, with a corresponding increase in the average slowdown of the DSMS.

The work in [14] proposes the *Highest Normalized Rate (HNR)* policy for minimizing the slowdown in a DSMS. Under HNR, each operator $O_x^k$ is assigned a priority $V_x^k$ which is the *weighted rate* or *normalized rate* of the operator segment $E_x^k$ that starts at operator $O_x^k$ and it is defined as:

$$V_x^k = \frac{1}{T_k} \times \frac{S_x^k}{\overline{C}_x^k} \tag{2}$$

The *HNR* policy, like *HR*, is based on output rate, however, it also emphasizes the ideal tuple

processing time in assigning priorities. As such, an inexpensive operator segment with low productivity will get a higher priority under *HNR* than under *HR*.

*Worst-Case Performance*: It is expected that a scheduling policy that strives to minimize the average-case performance might lead to a poor worst-case performance under a relatively high load. That is, some queries (or tuples) might starve under such a policy. The worst-case performance is typically measured using *maximum response time* or *maximum slowdown* [7].

Intuitively, a policy that optimizes for the worst-case performance should be pessimistic. That is, it assumes the worst-case scenario where each processed tuple will satisfy all the filters in the corresponding query.

The work in [14] shows that the traditional *First-Come-First-Serve (FCFS)* minimizes the maximum response time. Similarly, it shows that the traditional *Longest Stretch First (LSF)* [1] optimizes the maximum slowdown.

*Average- vs. Worst-Case Performance*: On one hand, the average value for a QoS metric provided by the system represents the expected QoS experienced by any tuple in the system (i.e., the average-case performance). On the other hand, the maximum value measures the worst QoS experienced by some tuple in the system (i.e., the worst-case performance). It is known that each of these metrics by itself is not enough to fully characterize system performance.

The most common way to capture the trade-off between the average-case and the worst-case performance is to measure the $\ell_2$ norm [6]. For instance, the $\ell_2$ norm of response times, $R_i$, is defined as:

**Definition 3**
*The $\ell_2$ norm of response times for N tuples is equal to $\sqrt{\sum_N^1 R_i^2}$.*

The definition shows that the $\ell_2$ norm considers the average in the sense that it takes into account all values, yet, by considering the second norm of each value instead of the first norm, it penalizes more severely outliers compared to the average metrics.

In order to balance the trade-off between the average- and worst-case performance, the *Balance Slowdown (BSD)* and the *Balance Response Time (BRT)* policies have been proposed in [14]. To avoid starvation, the two policies consider the amount of time an operator $O_x^k$ has been waiting for scheduling (i.e., $W_x^k$). Specifically, under *BSD*, each operator $O_x^k$ is assigned a priority value $V_x^k$ which is the product of the operator's normalized rate and the current highest slowdown of its pending tuples. That is:

$$V_x^k = \left(\frac{S_x^k}{\overline{C}_x^k T_k}\right)\left(\frac{W_x^k}{T_k}\right) \tag{3}$$

As such, under *BSD*, an operator is selected either because it has a high weighted rate or because its pending tuples have acquired a high slowdown.

*Application-Specific QoS*: Aurora also proposes a QoS-aware scheduler which attempts to satisfy application-specified QoS requirements [9]. Specifically, under that QoS-aware scheduler, each query is associated with a QoS graph which defines the utility of stale output.

Given, a QoS graph, the scheduler computes for each operator a *utility* value which is basically the slope of the QoS graph at the tuple's output time. The scheduler also computes for each operator its *urgency* value which is an estimation of how close is an operator to a critical point on the QoS graph where the QoS changes sharply. Then, at each scheduling point, the scheduler chooses for execution the operators with the highest utility value and among those that have the same utility, it chooses the one that has the highest urgency.

*Memory Usage*: Multi-query scheduling has also been exploited to optimize metrics beyond QoS. For example, *Chain* is a multi-query scheduling policy that optimizes memory usage in order to minimize space requirements for buffering tuples [2]. Towards this, for each query plan, *Chain* constructs what is called a *progress chart*. A progress chart is basically a set of segments where the slope of each segment represents the rate of change in the size of a tuple being processed by a set of consecutive operators

along the query plan. Given that progress chart, at each scheduling point, *Chain* schedules for execution the tuple that lies on the segment with the steepest slope. The intuition is to give higher priority to segments of operators with higher tuple consumption rate which will lead to quickly freeing more memory.

*Quality of Data (QoD)*: Another metric to optimize is Quality of Data (*QoD*). For instance, the work in [15] proposes the *freshness-aware* scheduling policy for improving the QoD of data streams, when QoD is defined in terms of freshness. The proposed scheduler exploits the variability in query costs, divergence in arrival patterns, and the probabilistic impact of selectivity in order to maximize the freshness of output data streams.

*Multiple-Objective Scheduling*: In DSMSs, and in computer systems in general, it is often desirable to optimize for multiple metrics at the same time. However, those metrics might be in conflict most of the time. This motivated the proposals of schedulers that are able to balance the trade-off between certain conflicting metrics.

For instance, the work in [3] attempts to balance the trade-off between memory usage and latency by formalizing latency requirements as a constraint to the *Chain* scheduler. This formulation lead to the *Mixed* policy which can be viewed as a heuristic strategy that is intermediate between Chain and FIFO. Specifically, *Mixed* is tuned via a parameter where a high value of that parameter causes *Mixed* to behave more like *FIFO*, whereas a lower value makes it behave more like *Chain*.

In another attempt towards multiple-objective scheduling, the work in [16] proposes *AMoS* which is an Adaptive Multi-objective Scheduling selection framework. Given several scheduling algorithms, AMoS employs a learning mechanism to learn the behavior of the scheduling algorithms over time. It then uses the learned knowledge to continuously select the algorithm that has statistically performed the best.

*Scheduler Implementation*: To ensure the applicability of scheduling policies in DSMSs, a low-overhead implementation is needed in order to reduce the amount of computation involved in computing priorities. For static policies (i.e., policies where an operator priority is constant over time), priorities are computed only once when a query is registered in the DSMS which naturally leads to a low-overhead implementation. Examples of such static policies include *HR*, *HNR*, and *Chain*. On the other hand, for dynamic policies where priority is a function of time, the priority of each operator should be re-computed at each instant of time. Such a naive implementation renders that class of policies very impractical. This motivated several approximation methods for efficient implementation of dynamic policies to balance the trade-off between scheduling overhead and accuracy. For instance the work in [9] proposes using bucketing as well as precomputation for an efficient implementation of the QoS-aware scheduling in Aurora. Similarly, [14] proposes using search space reduction and pruning methods in addition to clustered processing of continuous queries.

## Key Applications

There is a plethora of applications that require data stream management systems and, as such, proper scheduling strategies. The most well-known class of applications is that of *monitoring applications* [8], be it environmental monitoring (e.g., via sensor networks), network monitoring (e.g., by collecting router data), or even financial monitoring (e.g., by observing stock-market data). In all such cases, the sheer amount of input data precipitates the use of the data stream processing paradigm and proper scheduling strategies.

## Cross-References

- Adaptive Query Processing
- Adaptive Stream Processing
- Data Stream
- Event Stream
- Stream Processing
- Streaming Applications
- Stream-Oriented Query Languages and Operators

## Recommended Reading


1. Acharya S, Muthukrishnan S. Scheduling on-demand broadcasts: new metrics and algorithms. In: Proceedings of the 4th Annual International Conference on Mobile Computing and Networking; 1998.
2. Babcock B, Babu S, Datar M, Motwani R. Chain: operator scheduling for memory minimization in data stream systems. In: Proceedings of the ACM SIGMOD International Conference on Management of Data; 2003.
3. Babcock B, Babu S, Datar M, Motwani R, Thomas D. Operator scheduling in data stream systems. VLDB J. 2004;13(4):333–53.
4. Babcock B, Babu S, Datar M, Motwani R, Widom J. Models and issues in data stream systems. In: Proceedings of the ACM SIGMOD International Conference on Management of Data; 2002.
5. Babu S, Widom J. Continuous queries over data streams. ACM SIGMOD Rec. 2001;30(3):109–120.
6. Bansal N, Pruhs K. Server scheduling in the $L_p$ norm: a rising tide lifts all boats. In: Proceedings of the 35th Annual ACM Symposium on Theory of Computing; 2003.
7. Bender MA, Chakrabarti S, Muthukrishnan S. Flow and stretch metrics for scheduling continuous job streams. In: Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms; 1998.
8. Carney D, Cetintemel U, Cherniack M, Convey C, Lee S, Seidman G, Stonebraker M, Tatbul N, Zdonik S. Monitoring streams: a new class of data management applications. In: Proceedings of the 28th International Conference on Very Large Data Bases; 2002.
9. Carney D, Cetintemel U, Rasin A, Zdonik S, Cherniack M, Stonebraker M. Operator scheduling in a data stream manager. In: Proceedings of the 29th International Conference on Very Large Data Bases; 2003.
10. Chandrasekaran S, Cooper O, Deshpande A, Franklin MJ, Hellerstein JM, Hong W, Krishnamurthy S, Madden S, Raman V, Reiss F, Shah MA. TelegraphCQ: continuous dataflow processing for an uncertain world. In: Proceedings of the 1st Biennial Conference on Innovative Data Systems Research; 2003.
11. Golab L, Özsu MT. Issues in data stream management. ACM SIGMOD Rec. 2003;32(2):5–14.
12. Mehta M, DeWitt DJ. Dynamic memory allocation for multiple-query workloads. In: Proceedings of the 19th International Conference on Very Large Data Bases; 1993.
13. Muthukrishnan S, Rajaraman R, Shaheen A, Gehrke J.E. Online scheduling to minimize average stretch. In: Proceedings of the 40th Annual Symposium on Foundations of Computer Science; 1999.
14. Sharaf MA, Chrysanthis PK, Labrinidis A, Pruhs K. Efficient scheduling of heterogeneous continuous queries. In: Proceedings of the 32nd International Conference on Very Large Data Bases; 2006.
15. Sharaf MA, Labrinidis A, Chrysanthis PK, Pruhs K. Freshness-aware scheduling of continuous queries in the Dynamic Web. In: Proceedings of the 8th International Workshop on the World Wide Web and Database; 2005.
16. Sutherland T, Pielech B, Zhu Y, Ding L, Rundensteiner EA. An adaptive multi-objective scheduling selection framework for continuous query processing. In: Proceedings of the International Database Engineering and Applications Symposium; 2005.
17. Urhan T, Franklin M.J. Dynamic pipeline scheduling for improving interactive query performance. In: Proceedings of the 27th International Conference on Very Large Data Bases; 2001.


# Schema Evolution

John F. Roddick
Flinders University, Adelaide, SA, Australia

## Definition

Schema evolution deals with the need to retain current data when database schema changes are performed. Formally, *Schema Evolution* is accommodated when a database system facilitates database schema modification without the loss of existing data, (q.v. the stronger concept of *Schema Versioning*) (Schema evolution and schema versioning has been conflated in the literature with the two terms occasionally being used interchangeably. Readers are thus also encouraged to read also the entry for *Schema Versioning*.).

## Historical Background

Since schemata change and/or multiple schemata are often required, there is a need to ensure that extant data either stays consistent with the revised schema or is explicitly deleted as part of the change process. A database that supports schema evolution supports this transformation process.