

# Data Stream Management: A Brave New World

Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi

## 1 Introduction

Traditional data-management systems software is built on the concept of *persistent data sets* that are stored reliably in stable storage and queried/updated several times throughout their lifetime. For several emerging application domains, however, data arrives and needs to be processed on a continuous ( $24 \times 7$ ) basis, without the benefit of several passes over a static, persistent data image. Such *continuous data streams* arise naturally, for example, in the network installations of large Telecom and Internet service providers where detailed usage information (Call-Detail-Records (CDRs), SNMP/RMON packet-flow data, etc.) from different parts of the underlying network needs to be continuously collected and analyzed for interesting trends. Other applications that generate rapid, continuous and large volumes of stream data include transactions in retail chains, ATM and credit card operations in banks, financial tickers, Web server log records, etc. In most such applications, the data stream is actually accumulated and archived in a database-management system of a (perhaps, off-site) data warehouse, often making access to the archived data prohibitively expensive. Further, the ability to make decisions and infer interesting

---

M. Garofalakis (✉)

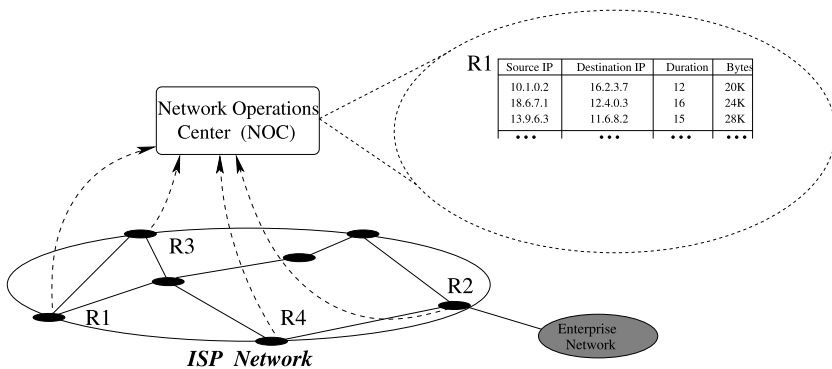
School of Electrical and Computer Engineering, Technical University of Crete,  
University Campus—Kounoupidiana, Chania 73100, Greece  
e-mail: [minos@softnet.tuc.gr](mailto:minos@softnet.tuc.gr)

J. Gehrke

Microsoft Corporation, One Microsoft Way, Redmond, WA 98052-6399, USA  
e-mail: [johannes@microsoft.com](mailto:johannes@microsoft.com)

R. Rastogi

Amazon India, Brigade Gateway, Malleshwaram (W), Bangalore 560055, India  
e-mail: [rastogi@amazon.com](mailto:rastogi@amazon.com)



**Fig. 1** ISP network monitoring data streams

patterns *on-line* (i.e., as the data stream arrives) is crucial for several mission-critical tasks that can have significant dollar value for a large corporation (e.g., telecom fraud detection). As a result, recent years have witnessed an increasing interest in designing data-processing algorithms that work over continuous data streams, i.e., algorithms that provide results to user queries while looking at the relevant data items *only once and in a fixed order* (determined by the stream-arrival pattern).

*Example 1* (Application: ISP Network Monitoring) To effectively manage the operation of their IP-network services, large Internet Service Providers (ISPs), like AT&T and Sprint, continuously monitor the operation of their networking infrastructure at dedicated Network Operations Centers (NOCs). This is truly a large-scale monitoring task that relies on continuously collecting streams of usage information from hundreds of routers, thousands of links and interfaces, and blisteringly-fast sets of events at different layers of the network infrastructure (ranging from fiber-cable utilizations to packet forwarding at routers, to VPNs and higher-level transport constructs). These data streams can be generated through a variety of network-monitoring tools (e.g., Cisco’s NetFlow [10] or AT&T’s GigaScope probe [5] for monitoring IP-packet flows). For instance, Fig. 1 depicts an example ISP monitoring setup, with an NOC tracking NetFlow measurement streams from four edge routers in the network  $R_1$ – $R_4$ . The figure also depicts a small fragment of the streaming data tables retrieved from routers  $R_1$  and  $R_2$  containing simple summary information for IP sessions. In real life, such streams are truly massive, comprising hundreds of attributes and billions of records—for instance, AT&T collects over one terabyte of NetFlow measurement data from its production network each day!

Typically, this measurement data is periodically shipped off to a backend data warehouse for off-line analysis (e.g., at the end of the day). Unfortunately, such off-line analyses are painfully inadequate when it comes to critical network-management tasks, where reaction in (*near*) *real-time* is absolutely essential. Such tasks include, for instance, detecting malicious/fraudulent users, DDoS attacks, or Service-Level Agreement (SLA) violations, as well as real-time traffic engineering to avoid congestion and improve the utilization of critical network resources. Thus,

it is crucial to process and analyze these continuous network-measurement streams in real-time and a single pass over the data (as it is streaming into the NOC), while, of course, remaining within the resource (e.g., CPU and memory) constraints of the NOC. (Recall that these data streams are truly massive, and there may be hundreds or thousands of analysis queries to be executed over them.)

This volume focuses on the *theory and practice of data stream management*, and the difficult, novel challenges this emerging domain introduces for data-management systems. The collection of chapters (contributed by authorities in the field) offers a comprehensive introduction to both the algorithmic/theoretical foundations of data streams and the streaming systems/applications built in different domains. In the remainder of this introductory chapter, we provide a brief summary of some basic data streaming concepts and models, and discuss the key elements of a generic stream query processing architecture. We then give a short overview of the contents of this volume.

## 2 Basic Stream Processing Models

When dealing with structured, tuple-based data streams (as in Example 1), the streaming data can essentially be seen as rendering massive *relational table(s)* through a *continuous stream of updates* (that, in general, can comprise both insertions and deletions). Thus, the processing operations users would want to perform over continuous data streams naturally parallel those in conventional database, OLAP, and data-mining systems. Such operations include, for instance, relational selections, projections, and joins, GROUP-BY aggregates and multi-dimensional data analyses, and various pattern discovery and analysis techniques. For several of these data manipulations, the high-volume and continuous (potentially, unbounded) nature of real-life data streams introduces novel, difficult challenges which are not addressed in current data-management architectures. And, of course, such challenges are further exacerbated by the typical user/application requirements for continuous, near real-time results for stream operations. As a concrete example, consider some of example queries that a network administrator may want to support over the ISP monitoring architecture depicted in Fig. 1.

- To analyze frequent traffic patterns and detect potential Denial-of-Service (DoS) attacks, an example analysis query could be: Q1: “*What are the top-100 most frequent IP (source, destination) pairs observed at router R1 over the past week?*”. This is an instance of a *top-k* (or, “heavy-hitters”) query—viewing the *R1* as a (dynamic) relational table, it can be expressed using the standard SQL query language as follows:

```
Q1:      SELECT ip_source, ip_dest, COUNT(*) AS frequency
          FROM    R1
          GROUP BY ip_source, ip_dest
          ORDER BY COUNT(*) DESC
          LIMIT 100
```

- To correlate traffic patterns across different routers (e.g., for the purpose of dynamic packet routing or traffic load balancing), example queries might include: Q2: “How many distinct IP (source, destination) pairs have been seen by both R1 and R2, but not R3?”, and Q3: “Count the number of session pairs in R1 and R2 where the source-IP in R1 is the same as the destination-IP in R2.” Q2 and Q3 are examples of (multi-table) *set-expression* and *join-aggregate* queries, respectively; again, they can both be expressed in standard SQL terms over the R1–R3 tables:

```

Q2:  SELECT COUNT(*) FROM
      ((SELECT DISTINCT ip_source, ip_dest FROM R1
        INTERSECT
        SELECT DISTINCT ip_source, ip_dest FROM R2
       ) EXCEPT
      SELECT DISTINCT ip_source, ip_dest FROM R3)

Q3:  SELECT COUNT(*)
      FROM R1, R2
      WHERE R1.ip_source = R2.ip_dest

```

A data-stream processing engine turns the paradigm of conventional database systems on its head: Databases typically have to deal with a stream of queries over a static, bounded data set; instead, a stream processing engine has to effectively process a static set of queries over continuous streams of data. Such stream queries can be (i) *continuous*, implying the need for continuous, real-time monitoring of the query answer over the changing stream, or (ii) *ad-hoc* query processing requests interspersed with the updates to the stream. The high data rates of streaming data might outstrip processing resources (both CPU and memory) on a steady or intermittent (i.e., bursty) basis; in addition, coupled with the requirement for near real-time results, they typically render access to secondary (disk) storage completely infeasible.

In the remainder of this section, we briefly outline some key data-stream management concepts and discuss basic stream-processing models.

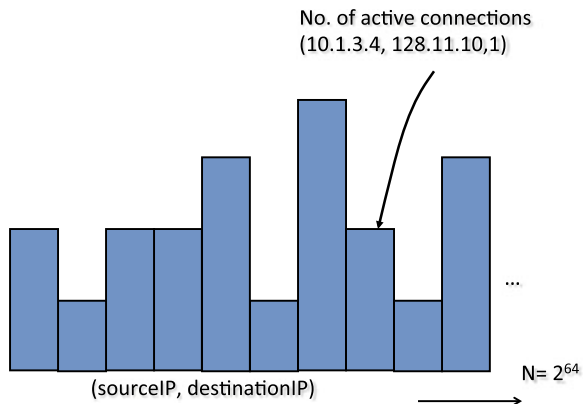
## 2.1 Data Streaming Models

An equivalent view of a relational data stream is that of a *massive, dynamic, one-dimensional vector*  $A[1 \dots N]$ —this is essentially using standard techniques (e.g., row- or column-major). As a concrete example, Fig. 2 depicts the stream vector  $A$  for the problem of monitoring active IP network connections between source/destination IP addresses. The specific dynamic vector has  $2^{64}$  entries capturing the up-to-date frequencies for specific (source, destination) pairs observed in IP connections that are currently active. The size  $N$  of the streaming  $A$  vector is defined as the product of the attribute domain size(s) which can easily grow very large, especially for multi-attribute relations.<sup>1</sup> The dynamic vector  $A$  is rendered through

---

<sup>1</sup>Note that streaming algorithms typically do not require a priori knowledge of  $N$ .

**Fig. 2** Example dynamic vector modeling streaming network data



a continuous stream of updates, where the  $j$ th update has the general form  $\langle k, c[j] \rangle$  and effectively modifies the  $k$ th entry of  $A$  with the operation  $A[k] \leftarrow A[k] + c[j]$ . We can define three generic data streaming models [9] based on the nature of these updates:

- **Time-Series Model.** In this model, the  $j$ th update is  $\langle j, A[j] \rangle$  and updates arrive in increasing order of  $j$ ; in other words, we observe the entries of the streaming vector  $A$  by increasing index. This naturally models *time-series* data streams, such as the series of measurements from a temperature sensor or the volume of NASDAQ stock trades over time. Note that this model poses a severe limitation on the update stream, essentially prohibiting updates from changing past (lower-index) entries in  $A$ .
- **Cash-Register Model.** Here, the only restriction we impose on the  $j$ th update  $\langle k, c[j] \rangle$  is that  $c[j] \geq 0$ ; in other words, we only allow increments to the entries of  $A$  but, unlike the Time-Series model, multiple updates can increment a given entry  $A[j]$  over the stream. This is a natural model for streams where data is just inserted/accumulated over time, such as streams monitoring the total packets exchanged between two IP addresses or the collection of IP addresses accessing a web server. In the relational case, a Cash-Register stream naturally captures the case of an *append-only* relational table which is quite common in practice (e.g., the fact table in a data warehouse [1]).
- **Turnstile Model.** In this, most general, streaming model, no restriction is imposed on the  $j$ th update  $\langle k, c[j] \rangle$ , so that  $c[j]$  can be either positive or negative; thus, we have a fully dynamic situation, where items can be continuously inserted and deleted from the stream. For instance, note that our example stream for monitoring active IP network connections (Fig. 2) is a Turnstile stream, as connections can be initiated or terminated between any pair of addresses at any point in the stream. (A technical constraint often imposed in this case is that  $A[j] \geq 0$  always holds—this is referred to as the *strict* Turnstile model [9].)

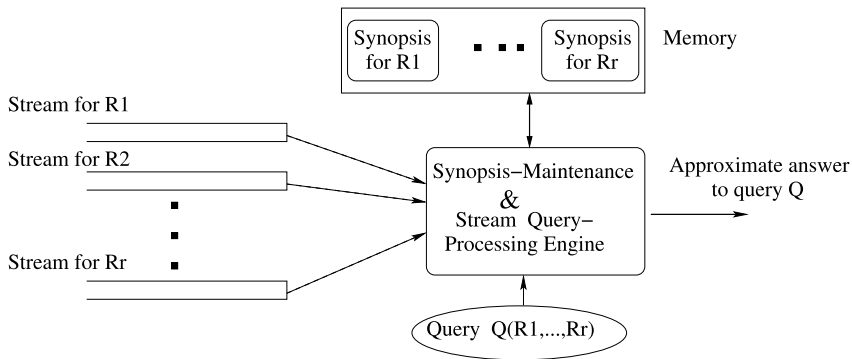
The above streaming models are obviously given in increasing order of generality: Ideally, we seek algorithms and techniques that work in the most general, Turn-

stile model (and, thus, are also applicable in the other two models). On the other hand, the weaker streaming models rely on assumptions that can be valid in certain application scenarios, and often allow for more efficient algorithmic solutions in cases where Turnstile solutions are inefficient and/or provably hard.

Our generic goal in designing data-stream processing algorithms is to *compute functions (or, queries) on the vector  $A$*  at different points during the lifetime of the stream (continuous or ad-hoc). For instance, it is not difficult to see that the example queries Q1–Q3 mentioned earlier in this section can be trivially computed over stream vectors similar to that depicted in Fig. 2, assuming that the complete vector(s) are available; similarly, other types of processing (e.g., data mining) can be easily carried out over the full frequency vector(s) using existing algorithms. This, however, is an unrealistic assumption in the data-streaming setting: The main challenge in the streaming model of query computation is that the size of the stream vector,  $N$ , is typically huge, making it impractical (or, even infeasible) to store or make multiple passes over the entire stream. The typical requirement for such stream processing algorithms is that they operate in *small space* and *small time*, where “space” refers to the working space (or, state) maintained by the algorithm and “time” refers to both the processing time per update (e.g., to appropriately modify the state of the algorithm) and the query-processing time (to compute the current query answer). Furthermore, “small” is understood to mean a quantity significantly smaller than  $\Theta(N)$  (typically, poly-logarithmic in  $N$ ).

## 2.2 Incorporating Recency: Time-Decayed and Windowed Streams

Streaming data naturally carries a temporal dimension and a notion of “time”. The conventional data streaming model discussed thus far (often referred to as *landmark streams*) assumes that the streaming computation begins at a well defined starting point  $t_0$  (at which the streaming vector is initialized to all zeros), and at any time  $t$  takes into account all streaming updates between  $t_0$  and  $t$ . In many applications, however, it is important to be able to downgrade the importance (or, weight) of older items in the streaming computation. For instance, in the statistical analysis of trends or patterns over financial data streams, data that is more than a few weeks old might naturally be considered “stale” and irrelevant. Various *time-decay models* have been proposed for streaming data, with the key differentiation lying in the relationship between an update’s weight and its age (e.g., exponential or polynomial decay [3]). The *sliding-window model* [6] is one of the most prominent and intuitive time-decay models that essentially considers only a window of the most recent updates seen in the stream thus far—updates outside the window are automatically “aged out” (e.g., given a weight of zero). The definition of the window itself can be either *time-based* (e.g., updates seen over the last  $W$  time units) or *count-based* (e.g., the last  $W$  updates). The key limiting factor in this streaming model is, naturally, the size of the window  $W$ : the goal is to design query processing techniques that have space/time requirements significantly sublinear (typically, poly-logarithmic) in  $W$  [6].



**Fig. 3** General stream query processing architecture

### 3 Querying Data Streams: Synopses and Approximation

A generic query processing architecture for streaming data is depicted in Fig. 3. In contrast to conventional database query processors, the assumption here is that a stream query-processing engine is allowed to see the data tuples in relations *only once and in the fixed order of their arrival* as they stream in from their respective source(s). Backtracking over a stream and explicit access to past tuples is impossible; furthermore, the order of tuples arrival for each streaming relation is arbitrary and duplicate tuples can occur anywhere over the duration of the stream. Furthermore, in the most general turnstile model, the stream rendering each relation can comprise tuple deletions as well as insertions.

Consider a (possibly, complex) aggregate query  $Q$  over the input streams and let  $N$  denote an upper bound on the total size of the streams (i.e., the size of the complete stream vector(s)). Our data-stream processing engine is allowed a certain amount of memory, typically orders of magnitude smaller than the total size of its inputs. This memory is used to continuously maintain concise *synopses/summaries* of the streaming data (Fig. 3). The two key constraints imposed on such stream synopses are:

- (1) **Single Pass**—the synopses are easily maintained, during a single pass over the streaming tuples in the (arbitrary) order of their arrival; and,
- (2) **Small Space/Time**—the memory footprint as well as the time required to update and query the synopses is “small” (e.g., poly-logarithmic in  $N$ ).

In addition, two highly desirable properties for stream synopses are:

- (3) **Delete-proof**—the synopses can handle both insertions and deletions in the update stream (i.e., general turnstile streams); and,
- (4) **Composable**—the synopses can be built independently on different parts of the stream and composed/merged in a simple (and, ideally, lossless) fashion to obtain a synopsis of the entire stream (an important feature in distributed system settings).

At any point in time, the engine can process the maintained synopses in order to obtain an estimate of the query result (in a continuous or ad-hoc fashion). Given that the synopsis construction is an inherently lossy compression process, excluding very simple queries, these estimates are necessarily *approximate*—ideally, with some guarantees on the approximation error. These guarantees can be either *deterministic* (e.g., the estimate is always guaranteed to be within  $\epsilon$  relative/absolute error of the accurate answer) or *probabilistic* (e.g., estimate is within  $\epsilon$  error of the accurate answer except for some small failure probability  $\delta$ ). The properties of such  $\epsilon$ - or  $(\epsilon, \delta)$ -estimates are typically demonstrated through rigorous analyses using known algorithmic and mathematical tools (including, sampling theory [2, 11], tail inequalities [7, 8], and so on). Such analyses typically establish a formal tradeoff between the space and time requirements of the underlying synopses and estimation algorithms, and their corresponding approximation guarantees.

Several classes of stream synopses are studied in the chapters that follow, along with a number of different practical application scenarios. An important point to note here is that there really is no “universal” synopsis solution for data stream processing: to ensure good performance, synopses are typically purpose-built for the specific query task at hand. For instance, we will see different classes of stream synopses with different characteristics (e.g., random samples and AMS sketches) for supporting queries that rely on *multiset/bag semantics* (i.e., the full frequency distribution), such as range/join aggregates, heavy-hitters, and frequency moments (e.g., example queries Q1 and Q3 above). On the other hand, stream queries that rely on *set semantics*, such as estimating the number of *distinct* values (i.e., set cardinality) in a stream or a set expression over a stream (e.g., query Q2 above), can be more effectively supported by other classes of synopses (e.g., FM sketches and distinct samples). A comprehensive overview of synopsis structures and algorithms for massive data sets can be found in the recent survey of Cormode et al. [4].

## 4 This Volume: An Overview

The collection of chapters in this volume (contributed by authorities in the field) offers a comprehensive introduction to both the algorithmic/theoretical foundations of data streams and the streaming systems/applications built in different domains. The authors have also taken special care to ensure that each chapter is, for the most part, self-contained, so that readers wishing to focus on specific streaming techniques and aspects of data-stream processing, or read about particular streaming systems/applications can move directly to the relevant chapter(s).

Part I focuses on basic algorithms and stream synopses (such as random samples and different sketching structures) for landmark and sliding-window streams, and some key stream processing tasks (including the estimation of quantiles, norms, join-aggregates, top- $k$  values, and the number of distinct values). The chapters in Part II survey existing techniques for basic stream mining tasks, such as clustering, decision-tree classification, and the discovery of frequent itemsets and temporal dynamics. Part III discusses a number of advanced stream processing topics, including



algorithms and synopses for more complex queries and analytics, and techniques for querying distributed streams. The chapters in Part IV focus on the system and language aspects of data stream processing through comprehensive surveys of existing system prototypes and language designs. Part V then presents some representative applications of streaming techniques in different domains, including network management, financial analytics, time-series analysis, and publish/subscribe systems. Finally, we conclude this volume with an overview of current data streaming products and novel application domains (e.g., cloud computing, big data analytics, and complex event processing), and discuss some future directions in the field.

## References

1. S. Chaudhuri, U. Dayal, An overview of data warehousing and OLAP technology. *ACM SIGMOD Record* **26**(1) (1997)
2. W.G. Cochran, *Sampling Techniques*, 3rd edn. (Wiley, New York, 1977)
3. E. Cohen, M.J. Strauss, Maintaining time-decaying stream aggregates. *J. Algorithms* **59**(1), 19–36 (2006)
4. G. Cormode, M. Garofalakis, P.J. Haas, C. Jermaine, Synopses for massive data: samples, histograms, wavelets, sketches. *Found. Trends® Databases* **4**(1–3) (2012)
5. C. Cranor, T. Johnson, O. Spatscheck, V. Shkapenyuk, GigaScope: a stream database for network applications, in *Proc. of the 2003 ACM SIGMOD Intl. Conference on Management of Data*, San Diego, California (2003)
6. M. Datar, A. Gionis, P. Indyk, R. Motwani, Maintaining stream statistics over sliding windows. *SIAM J. Comput.* **31**(6), 1794–1813 (2002)
7. M. Mitzenmacher, E. Upfal, *Probability and Computing: Randomized Algorithms and Probabilistic Analysis* (Cambridge University Press, Cambridge, 2005)
8. R. Motwani, P. Raghavan, *Randomized Algorithms* (Cambridge University Press, Cambridge, 1995)
9. S. Muthukrishnan, Data streams: algorithms and applications. *Found. Trends Theor. Comput. Sci.* **1**(2) (2005)
10. NetFlow services and applications. Cisco systems white paper (1999). <http://www.cisco.com/>
11. C.-E. Särndal, B. Swensson, J. Wretman, *Model Assisted Survey Sampling* (Springer, New York, 1992). Springer Series in Statistics

Data Stream Management

Processing High-Speed Data Streams

Garofalakis, M.; Gehrke, J.; Rastogi, R. (Eds.)

2016, VII, 537 p. 103 illus., 16 illus. in color., Hardcover

ISBN: 978-3-540-28607-3